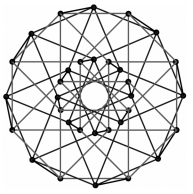
Adventures with *free* Mathematical Software

Edition 4

Adventures with *free* Mathematical Software

by

Justin R. Smith



Five Dimensions Press

Dedicated to the memory of my wonderful wife, Brigitte.

ISBN: 9798332487293 (Hardcover)

Also published by Five Dimensions Press

- *Introduction to Algebraic Geometry* (paperback and hardcover), Justin Smith.
- *Abstract Algebra* (paperback and hardcover), Justin Smith.
- *Eye of a Fly* (Kindle edition and paperback), Justin Smith.
- *The God Virus,* (Kindle edition and paperback) by Justin Smith.
- *Ohana,* (Kindle edition and paperback) by Justin Smith.
- *The Accidental Empress,* (Kindle edition and paperback) by Justin Smith.
- *Die zufällige Kaiserin,* German translation of *The Accidental Empress* (Kindle edition and paperback) by Justin Smith.

Five Dimensions Press page:
http://www.five-dimensions.org
Email:**jsmith@drexel.edu**

Foreword

> "The number system is like human life. First you have the natural numbers. The ones that are whole and positive. Like the numbers of a small child. But human consciousness expands. The child discovers longing. Do you know the mathematical expression for longing? The negative numbers. The formalization of the feeling that you're missing something. Then the child discovers the in-between spaces, between stones, between people, between numbers and that produces fractions, but it's like a kind of madness, because it does not even stop there, it never stops... Mathematics is a vast open landscape. You head towards the horizon and it's always receding..."
> — Smilla Qaavigaaq Jaspersen, in the novel *Smilla's Sense of Snow,* by Peter Høeg (see [32]).

This book arose out of one of the more enjoyable undergraduate courses I taught at Drexel University: Mathematical Software. I taught it for many years without using a textbook (and probably never taught the same exact course twice!). I toyed with writing a text for it but never found the time.

The courses I taught are varying subsets of the material in this manuscript.

At Drexel, we used commercial software that had a very slick user-interface (and that gives *incorrect* results in an important case!). In this book, I use the free and open-source Maxima system with its wxMaxima interface.


Sections marked in this manner are more advanced or specialized and may be skipped on a first reading.


Sections marked in this manner are even *more* advanced or specialized and may be skipped on a first reading (or skipped entirely).

I am grateful to the many editors of Wikipedia. The biographical sketches in this book owe a great deal to their vital work.

I am also grateful to Matthias Ettrich and the many other developers of the software, LyX — a free front end to LaTeX that has the ease of use of a word processor, with spell-checking, an excellent equation editor, and a thesaurus. I have used this software for years and the

current version is more polished and bug-free than most commercial software.

For the fourth edition, I have added chapters on wavelets and graph theory.

Contents

List of Figures

Adventures with Mathematical Software

CHAPTER 1

Introduction

> "In the broad light of day, mathematicians check their equations and their proofs, leaving no stone unturned in their search for rigour. But at night, under the full moon, they dream, they float among the stars and wonder at the miracle of the heavens. They are inspired.
>
> Without dreams there is no art, no mathematics, no life."
> — Sir Michael Atiyah, *Notices of the AMS,* January 2010, page 8.

1.1. Installation and first steps

Mathematical software development has made great strides in recent decades, and one of the most powerful systems is free and open-source. It is a modernized form of the Macsyma system developed from 1968 to 1982 at MIT's Project MAC. The original system remained available to academics and US government agencies, and was distributed by the US Department of Energy (DOE). That version, DOE Macsyma, was maintained by Bill Schelter, a professor of mathematics at the University of Texas at Austin.

Under the name of Maxima, it was released under the GPL in 1999, and remains under active maintenance.

Versions of it exist for Linux, Windows, and the MacOS, and FreeBSD (see the web site `https://maxima.sourceforge.io/`. For other systems, you can download the source code and try to compile it.

The original Maxima had a command-line interface. Professor Schelter developed a rudimentary GUI interface. This was further improved by the wxMaxima project and now includes menus for many of the maxima commands and the ability to save one's work in a kind of notebook.

Your first assignment is to download wxMaxima and install this on the system of your choice (Maxima is bundled with most distributions of wxMaxima).

(1) In windows, you can download a version of it from `http://wxmaxima-developers.github.io/wxmaxima/`.

(2) In Linux and the three BSD's[1], pre-packaged versions of wxMaxima are available that you can install if you have root access. After it is installed, you can run it from your applications menu or in any directory by typing

```
wxmaxima
```

(3) If you don't have root access or your distribution doesn't support it, you can download the "Applimage" version of wxMaxima from the web site listed above and install it in your user account by following the instructions. This image contains Maxima and all other dependencies.

After starting up wxMaxima, you will notice a number of menus:

File: this is self-explanatory. It allows you to save notebooks and open ones you have saved.

Edit: also self-explanatory. The *copy* menu-items are significant. Copy-as-text copies a formula in a format that can be input to Maxima. Copy-as-LaTex copies it in a format suitable for inclusion in a TeX typesetting document. Copy-as-mathml copies it in a format suitable for web pages. Copy-as-image is suitable for pasting into a word document or web pages that can't be viewed by a mathml-aware web browser.

View: this controls which palettes and menus you see. Play with it to see what it does!

Cell: this is important! Maxima statements are called *cells*, and this executes them (as well as manipulating them in other ways).

Maxima: this interacts with the Maxima program in various ways.

Equations: this contains Maxima commands to solve equations or differential equations.

Matrix: a menu containing commands for creating and manipulating matrices.

Calculus: a menu containing commands to differentiate and integrate functions, among other things.

Simplify: an important menu containing commands to simplify or expand expressions and manipulate complex numbers.

List: commands to manipulate lists.

Plot: commands to create plots.

Numeric: contains commands related to numeric computations.

Help: self-explanatory.

%: this is not a menu item, but is very important nevertheless. This symbol represents the value of the last computation Maxima performed. Most menu commands act on this by default (although you can override this easily).

[1] FreeBSD, NetBSD, and OpenBSD.

Operator	
^	Exponentiation
/,*,.	Division, multiplication, matrix-multiplication
+, -	Addition, subtraction
:	Assignment
=,#,<,>,<=,>=	Equal, not equal, greater than, less than, greater or equal
not	boolean not
and	boolean and
or	boolean or

TABLE 1.1.1. Hierarchy of operations

Go to the menu marked 'New' in the upper right portion of the screen and select a new Maxima session (there are several other options available). Type 1+1 and click **Cell**▷**Evaluate Cell(s)** to get '2'. Amazing! This software can add 1 and 1. We can also operate with numbers using '*' for multiplication, '/' for division, and '^' for raising to a power.

See table 1.1.1, so the expression 1/2*a^2−3=0 has *implied* parentheses ((1/2)*(a^2))−3=0. Operations at the same priority are evaluated from left to right, so 3/4/5=3/(4*5)=3/20.

At this point, it is a good idea to save your notebook and give it a name other than 'Untitled'. Go to the 'File' menu and select 'Save As'.

Maxima can factor numbers: type **factor**(121) and click **Cell**▷**Evaluate Cell(s)** to get 11^2. For something more challenging, try **factor**(123456789) to get $3^2\,3607\,3803$.

Maxima saves fractions in their lowest form. If you type `128/256`, Maxima will come back with `1/2`.

Maxima had the basic **abs**-function built in that computes absolute value:

$$\mathbf{abs}(2) = 2$$
$$\mathbf{abs}(-2) = 2$$

In general, Maxima has *examples* of its commands built into it. The general format of the **example**-command is

example(command)

For instance, try typing and click **example**(**factor**) and click **Cell**▷**Evaluate Cell(s)**.

This may be used in other commands to refer to that output. For instance, suppose you type 2^100 and click **Cell**▷**Evaluate Cell(s)** to get

1267650600228229401496703205376

Now you can type **factor**(%) and click **Cell▷Evaluate Cell(s)** to get 2^{100}.

To get an idea of the raw computing power of Maxima, consider the factorial function. Recall that factorials are defined by

$$n! = n \times (n-1) \times (n-2) \times \cdots \times 1$$

The Maxima command for computing this is n! or **factorial**(n). Try typing 100! or 1000! and clicking on **Cell▷Evaluate Cell(s)**.

Factorials like n! represent the number of ways of arranging n distinct objects: Given n slots, the first object can go into any one of them. After it has been placed, there are $n-1$ slots left for the second object, and $n-2$ for the third, and so on.

Maxima also has a **binomial**-command given by

$$\mathbf{binomial}(n,m) = \frac{n!}{m!(n-m)!}$$

It also has a mathematical significance: it represents the number of ways of selecting a set of m objects from a set of n distinct objects. The numerator is all possible arrangements of the original n objects. Since we don't care what order the m objects we've selected are in (because this is a *set* of m objects), we divide out by the ways of arranging these m objects. Since we *really* don't care what order the $n-m$ objects we *didn't* select are in, we also divide out by (n-m)!.

The **float**-command gives the *numeric* value of a quantity in scientific notation.

The word *float* is part of computing history. Early computers could only work with integers. When computers were built that could handle numbers in scientific notation, the numbers were called *floating-point* because the decimal point could "float" into any position. In Maxima, float numbers use the computer's intrinsic ability to do floating-point arithmetic. Maxima also has a **bfloat**-command with floating point arithmetic implemented in *software.* These numbers could potentially have thousands of significant digits. A **bfloat**-number followed by bnn means the number is to be multiplied by 10^{nn}.

If you type 123456789/987654321 and click **Cell▷Evaluate Cell(s)**, you get the fraction in its lowest terms: 13717421/109739369. If you type **bfloat**(123456789/987654321) and click **Cell▷Evaluate Cell(s)**, you get 0.124999998860938.

The Numeric menu has a Bigfloat-precision option that specifies the number of digits to use. If you set this to 100, typing bfloat(%pi) and selecting **Cell▷Evaluate Cell(s)** gives

```
3.14159265358979323846264338327950288419716939937510
```

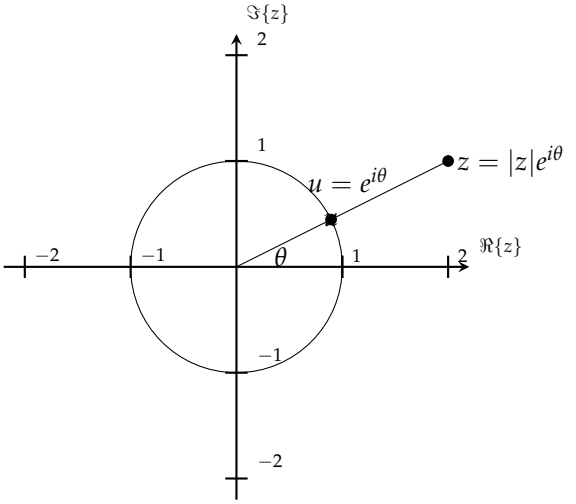


FIGURE 1.1.1. The complex plane

```
582097494459230781640628620899862803482534211706 8
```

Maxima has predefined mathematical constants such as e and π: typing bfloat(%e) and selecting **Cell▷Evaluate Cell(s)** produces 2.71828182845905: see Appendix B on page 291 for a list of them — including inf for *infinity*[2].

Identifiers in maxima are strings of: lower- and upper-case *letters*, *digits*, and '_'. They must not begin with a digit. Examples: 'set1', 'total_series', 'accum'. They are case-sensitive and must not equal any Maxima keyword:

integrate	**next**	**from**	**diff**
in	**at**	**limit**	**sum**
for	**and**	**elseif**	**then**
else	**do**	**or**	**if**
unless	**product**	**while**	**thru**
step	**block**	**return**	**derivative**

In Maxima, %i represents $\sqrt{-1}$ and we can compute with complex numbers. Recall that complex numbers can be either in a *rectangular* form like $a + bi$ or a *polar* form like $re^{i\theta}$— see figure 1.1.1. Maxima has commands to convert numbers between these forms: the **rectform**-command or menu-item **Simplify▷Complex Simplification▷Convert to Rectform** or the

[2]Try typing **bfloat**(inf) and **Cell▷Evaluate Cell(s)**!

polarform-command or menu-item **Simplify▷Complex Simplification▷Convert to Polarform**. Typing (2+3*%i)/(4+5*%i) and clicking **Cell▷Evaluate Cell(s)** causes Maxima to come back with $\frac{3\%i+2}{5\%i+4}$. Typing the **rectform**-command or menu-item **Simplify▷Complex Simplification▷Convert to Rectform** gives $\frac{2\%i}{41}+\frac{23}{41}$. Typing the **polarform**-command or menu-item **Simplify▷Complex Simplification▷Convert to Polarform** gives

$$\frac{\sqrt{13}\%e^{\%i\arctan\left(\frac{2}{23}\right)}}{\sqrt{41}}$$

The *argument* of the polar form (in this case arctan $\left(\frac{2}{23}\right)$) is returned by the **carg**-command.

Typing %e^(%pi*%i) and clicking **Cell▷Evaluate Cell(s)** results in –1, reproducing Euler's famous formula

$$e^{\pi i}=-1$$

and typing %e^(x*%i) and the **rectform**-command or menu-item **Simplify▷Complex Simplification▷Convert to Rectform** gives De Moivre's famous formula

$$\%i\sin\left(x\right)+\cos\left(x\right) \tag{1.1.1}$$

As you might expect, Maxima has basic functions like **realpart** and **imagpart** that extracts these aspects of complex numbers

$$\mathbf{realpart}(a+b*\%i)=a$$
$$\mathbf{imagpart}(a+b*\%i)=b$$

Unfortunately, the **abs**-function doesn't quite know how to handle complex numbers:

$$\mathbf{abs}(a+b*\%i)=|a+b*\%i|$$

For this purpose, we need the closely-related **cabs**-function ("complex" absolute value)

$$\mathbf{cabs}(a+b*\%i)=\sqrt{b^2+a^2}$$

Leonhard Euler (1707 – 1783) was, perhaps, the greatest mathematician of all time. Although he was born in Switzerland, he spent most of his life in St. Petersburg, Russia and Berlin, Germany. He originated the notation $f(x)$ for a function and made contributions to mechanics, fluid dynamics, optics, astronomy, and music theory. His final work, "Treatise on the Construction and Steering of Ships," is a classic whose ideas on shipbuilding are still used to this day.

To do justice to Euler's life would require a book considerably longer than the current one — see the article [24]. His collected works fill more than 70 volumes and, after his death, he left enough manuscripts behind to provide publications to the Journal of the Imperial Academy of Sciences (of Russia) for 47 years.

Typing %e^(%pi*%i/3) and clicking **Cell▷Evaluate Cell(s)** gives a *cube* root of -1, i.e., $\frac{\sqrt{3}i}{2} + \frac{1}{2}$. We can verify this claim by typing (1/2*%i*sqrt(3) + 1/2)^3. Unfortunately, Maxima just comes back with $\left(\frac{\sqrt{3}i}{2} + \frac{1}{2}\right)^3$.

What are we to do? Maxima has a command **expand**() that causes it to eliminate parentheses as much as possible and multiply factors out. Typing **expand**((1/2*%i*sqrt(3) + 1/2)^3) or clicking **Simplify▷Expand Expression** results in –1.

EXERCISES.

1. From a standard 52-card deck of playing cards, how many 5-card Poker hands are possible?

2. Write in the form $a + bi$

$$\frac{2}{3+i}$$

3. Write in the form $a + bi$

$$3i + \frac{1}{1-i}$$

4. Find equations for $\sin n\theta$ and $\cos n\theta$ in terms of $\sin\theta$ and $\cos\theta$. Hint: use de Moivre's Formula (1.1.1 on the facing page) and the binomial theorem.

CHAPTER 2

Number theory

"Mathematics is the queen of sciences and number theory is the queen of mathematics. She often condescends to render service to astronomy and other natural sciences, but in all relations she is entitled to the first rank."
— Carl Friedrich Gauss, see [68].

2.1. Introduction

People not interested in number theory can skip this chapter; none of the others depend on it.

Number theory is the study of *integers.* On the surface this makes it seem almost laughably simple, but some of the most difficult and complex problems in all of mathematics belong to number theory. For instance, Fermat's Last Theorem (stated in 1637):

The equation

$$a^n + b^b = c^n$$

has no solutions in integers with $a, b, c > 0$ *and* $n > 2$.

was only proved in 1995 by Andrew Wiles.

Pierre de Fermat (1607– 1665) was a French mathematician who is given credit for early developments that led to infinitesimal calculus, including his technique of adequality[a]. He is recognized for his discovery of an original method of finding the greatest and the smallest ordinates of curved lines, which is analogous to that of differential calculus, and his research into number theory. He made notable contributions to analytic geometry, probability, and optics. He is best known for his Fermat's principle for light propagation and his Fermat's Last Theorem in number theory, which he described in a note at the margin of a copy of Diophantus's *Arithmetica.*

[a] For finding maxima and minima of functions.

The famous Riemann Hypothesis (discussed in chapter 15 on page 269) is *still* unsolved.

Applied mathematicians regarded number theory as a subject only of theoretical interest[1]. This state of affairs changed in the 1960's when powerful new systems of cryptography were discovered that use number theory. Today, the National Security Agency (responsible for secure communications) employs more number theorists than any university.

We will begin by reviewing some very basic material.

Most people learned the following result in grade school — *long division* with a quotient and remainder:

PROPOSITION 2.1.1. *Let n and d be real numbers. Then it is possible to write*

$$n = q \cdot d + r$$

where q is an integer and $0 \leq r < d$. If $r = 0$, we say that $d \mid n$ — stated "d divides n". The negation of this is $d \nmid n$ (d doesn't divide n).

Maxima can compute this with the **mod**-command: type **mod**(987654321,123456789); and **Cell▷Evaluate Cell(s)** to get the remainder of dividing 987654321 by 123456789, namely 9.

Although this definition usually requires n and d to be integers, the Maxima command works for *real numbers* as well: **mod**(1.5,1); produces .5. Essentially,

$$\mathbf{mod}(a, b) = a - \left\lfloor \frac{a}{b} \right\rfloor \cdot b \tag{2.1.1}$$

The division algorithm gives rise to the concept of *greatest common divisor.*

DEFINITION 2.1.2. Let n and m be positive integers. The *greatest common divisor* of n and m, denoted $\gcd(n, m)$, is the largest integer d such that $d \mid n$ and $d \mid m$. The *least common multiple* of n and m, denoted $\operatorname{lcm}(n, m)$, is the smallest positive integer k such that $n \mid k$ and $m \mid k$.

Since 0 is divisible by any integer, $\gcd(n, 0) = \gcd(0, n) = n$.

There is a very fast algorithm for computing the greatest common divisor due to Euclid — see [17, 18].

REMARK. Euclid's original formulation was geometric, involving line-segments. Given two line-segments of lengths r_1 and r_2, it found a real number r such that

$$\frac{r_1}{r}, \frac{r_2}{r} \in \mathbb{Z}$$

An ancient proof of the irrationality of $\sqrt{2}$ showed that this process never terminates if one of the line-segments is of unit length and the other is the diagonal of a unit square.

[1] As a grad student at the Courant Institute, the author mentioned number theory and another student sneered "Does such a thing even *exist?*"

As trivial as proposition 2.1.1 on the facing page appears to be, it allows us to prove Bézout's Identity:

LEMMA 2.1.3. *Let n and m be positive integers. Then there exist integers u and v such that*

$$\gcd(n,m) = u \cdot n + v \cdot m \tag{2.1.2}$$

REMARK. Bézout proved this identity for polynomials — see [4]. However, this statement for integers can be found in the earlier work of Claude Gaspard Bachet de Méziriac (1581–1638) — see [33].

After loading via **load**("gcdex"), the Maxima function **igcdex**(n, k) computes the greatest common divisor and the values of u, v that appear in equation 2.1.2.

For example

```
load("gcdex");
igcdex(12345,98765432)
```

returns

$$[-39546175, 4943, 1]$$

where $\gcd(12345, 98765432) = 1$ and

$$-39546175 \cdot 12345 + 4943 \cdot 98765432 = 1$$

Étienne Bézout (1730–1783) was a French algebraist and geometer credited with the invention of the determinant (in [6]).

DEFINITION 2.1.4. A *prime number* is an integer that is not divisible by any integer other than 1 or ($\pm$)itself.

The Maxima commands regarding primes are:

- ▷ **primep**(n) returns **true** when n is a prime and **false** otherwise. The parameter **primep_number_of_tests** determines how many types of tests for primality will be performed. The default is 25.
- ▷ **primes**(n, m) — returns a list of all primes, p, such that $n \leq p \leq m$. For instance

  ```
  primes(2,20)
  ```

  returns

  $$[2, 3, 5, 7, 11, 13, 17, 19]$$

- ▷ **prev_prime** (n) — returns the largest prime $< n$.
- ▷ **next_prime** (n) — returns the smallest prime $> n$.

It is well-known that integers can be factored into powers of primes in a *unique* way (see [58, chapter 3]:

LEMMA 2.1.5. *Let n be a positive integer and let*

$$\begin{aligned} n &= p_1^{\alpha_1} \cdot \cdots \cdot p_k^{\alpha_k} \\ &= q_1^{\beta_1} \cdot \cdots \cdot q_\ell^{\beta_\ell} \end{aligned} \tag{2.1.3}$$

be factorizations into powers of distinct primes. Then $k = \ell$ and there is a reordering of indices $f\colon \{1,\dots,k\} \to \{1,\dots,k\}$ such that $q_i = p_{f(i)}$ and $\beta_i = \alpha_{f(i)}$ for all i from 1 to k.

The Maxima function **ifactors**(n) determines the unique factorization of n:

```
ifactors(123456789);
```

returns

$$[[3,2],[3607,1],[3803,1]]$$

showing that

$$123456789 = 3^2 \cdot 3607 \cdot 3803$$

In this case, the **factor**-command also works.

Unique factorization also leads to many other results:

PROPOSITION 2.1.6. *Let n and m be positive integers with factorizations*

$$\begin{aligned} n &= p_1^{\alpha_1} \cdots p_k^{\alpha_k} \\ m &= p_1^{\beta_1} \cdots p_k^{\beta_k} \end{aligned}$$

Then $n|m$ if and only if $\alpha_i \leq \beta_i$ for $i = 1,\dots,k$ and

$$\begin{aligned} \gcd(n,m) &= p_1^{\min(\alpha_1,\beta_1)} \cdots p_k^{\min(\alpha_k,\beta_k)} \\ \operatorname{lcm}(n,m) &= p_1^{\max(\alpha_1,\beta_1)} \cdots p_k^{\max(\alpha_k,\beta_k)} \end{aligned}$$

Consequently

$$\operatorname{lcm}(n,m) = \frac{nm}{\gcd(n,m)} \tag{2.1.4}$$

DEFINITION 2.1.7. If $n > 0$ is an integer, two integers r and s are *congruent modulo n*, written

$$r \equiv s \pmod{n}$$

if

$$n \mid (r-s)$$

REMARK. It is also common to say that r and s are *equal* modulo n. The first systematic study of these type of equations was made by Gauss in his *Disquistiones Arithmeticae* ([23]). Gauss wanted to find solutions to equations like

$$a_n x^n + \cdots + a_1 x + a_0 \equiv 0 \pmod{p}$$

In Maxima terms, $r \equiv s \pmod{n}$ if and only if **mod**(r,n)=**mod**(s,n).

PROPOSITION 2.1.8. *Equality modulo n respects addition and multiplication, i.e. if $r,s,u,v \in \mathbb{Z}$ and $n \in \mathbb{Z}$ with $n > 0$, and*

$$\begin{aligned} r &\equiv s \pmod{n} \\ u &\equiv v \pmod{n} \end{aligned} \tag{2.1.5}$$

then

$$\begin{aligned} r+u &\equiv s+v \pmod{n} \\ r \cdot u &\equiv s \cdot v \pmod{n} \end{aligned} \tag{2.1.6}$$

This elementary result has some immediate implications:

EXAMPLE. Show that $5|(7^k - 2^k)$ for all $k \geq 1$. First note, that $7 \equiv 2 \pmod{5}$. Equation 2.1.6, applied inductively, implies that $7^k \equiv 2^k \pmod{5}$ for all $k > 1$.

DEFINITION 2.1.9. If n is a positive integer, the set of equivalence classes of integers modulo n is denoted $\mathbb{Z}_n$.

REMARK. It is not hard to see that the size of $\mathbb{Z}_n$ is n and the equivalence classes are represented by integers

$$\{0, 1, 2, \ldots, n-1\}$$

Proposition 2.1.8 implies that addition and multiplication is well-defined in $\mathbb{Z}_n$. The Maxima command **zn_add_table**(n) returns a table of $\mathbb{Z}_n$ with the addition-operation. For instance:

```
zn_add_table(8);
```

returns table 2.1.1.

$$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 & 0 \\ 2 & 3 & 4 & 5 & 6 & 7 & 0 & 1 \\ 3 & 4 & 5 & 6 & 7 & 0 & 1 & 2 \\ 4 & 5 & 6 & 7 & 0 & 1 & 2 & 3 \\ 5 & 6 & 7 & 0 & 1 & 2 & 3 & 4 \\ 6 & 7 & 0 & 1 & 2 & 3 & 4 & 5 \\ 7 & 0 & 1 & 2 & 3 & 4 & 5 & 6 \end{pmatrix}$$

TABLE 2.1.1. Addition table for $\mathbb{Z}_8$

It is interesting to speculate on when a number has a multiplicative inverse modulo n. It turns out that:

PROPOSITION 2.1.10. *If $n > 1$ is an integer and $x \in \mathbb{Z}_n$, then there exists $y \in \mathbb{Z}_n$ with*

$$x \cdot y \equiv 1 \pmod{n}$$

if and only if $\gcd(x, n) = 1$. When this is true, we say that x is relatively prime to n.

Because of this, we are generally only interested in the elements $x \in \mathbb{Z}_n$ that are *relatively prime* to n. The set of such numbers is denoted $\mathbb{Z}_n^\times$, where the superscript $\times$ indicates that we're considering the elements of $\mathbb{Z}_n$ under *multiplication* rather than addition. Maxima has a command for computing the multiplication table for $\mathbb{Z}_n^\times$: **zn_mult_table**(n). For instance

```
zn_mult_table(8);
```

produces the table in 2.1.2.

$$\begin{pmatrix} 1 & 3 & 5 & 7 \\ 3 & 1 & 7 & 5 \\ 5 & 7 & 1 & 3 \\ 7 & 5 & 3 & 1 \end{pmatrix}$$

TABLE 2.1.2. Multiplication table for $\mathbb{Z}_8^\times$

We also have other commands for doing modular arithmetic:

- ▷ **power_mod**(a, n, m) — computes $a^n \pmod{m}$. Note: there are algorithms for computing powers modulo another number that are much faster than simply raising the number to that power.
- ▷ **inv_mod**(n, m) — computes $n^{-1} \pmod{m}$, if it exists (i.e., if $\gcd(n, m) = 1$), and **false** otherwise.

This section would not be complete without mention of the famous:

THEOREM 2.1.11 (Chinese Remainder Theorem). *If $n_1, \ldots, n_k$ are a set of positive integers with $\gcd(n_i, n_j) = 1$ for all $1 \leq i < j \leq k$, then the equations*

$$\begin{aligned} x &\equiv a_1 \pmod{n_1} \\ &\vdots \\ x &\equiv a_k \pmod{n_k} \end{aligned}$$

have a unique solution modulo $\prod_{i=1}^{k} n_i$.

REMARK. The Chinese Remainder Theorem was first published sometime between the 3rd and 5th centuries by the Chinese mathematician Sun Tzu (not to be confused with the author of "The Art of Warfare").

Naturally, Maxima has a command that implements this: In the notation of theorem 2.1.11 on the facing page, the command **chinese**($[a_1, \dots, a_k], [n_1, \dots, n_k]$) returns x. If any of the conditions of theorem 2.1.11 on the preceding page are not met, it returns **false**.

EXERCISES.

1. If n and m are two integers with $\gcd(n, m) = 1$, what can you say about the primes that appear in their factorizations?

2. If n and m are two integers with $\gcd(n, m) = 1$, show that $\mathbb{Z}_{n\cdot m}^{\times} = \mathbb{Z}_n^{\times} \times \mathbb{Z}_m^{\times}$(the right side of this consists of pairs (a, b), where $a \in \mathbb{Z}_n^{\times}$ and $b \in \mathbb{Z}_m^{\times}$). Hint: use the Chinese Remainder Theorem.

2.2. Euler's *totient* function

DEFINITION 2.2.1. If n is a positive integer then

$$\phi(n)$$

is the number of generators of $\mathbb{Z}_n$ — or the number of *elements* in $\mathbb{Z}_n^{\times}$, *or*

- ▷ If $n > 1$ it is the number of integers, d, with $1 \leq d < n$ with $\gcd(d, n) = 1$.
- ▷ If $n = 1$, it is equal to 1.

This is called the *Euler ϕ-function.* Euler also called it the *totient*. The Maxima command for computing this is called **totient**(n).

REMARK. If p is a prime number, then $\phi(p) = p - 1$ since the integers $1 \leq i \leq p - 1$ are all relatively prime to p.

Exercise 2 shows that, if n and m are integers with $\gcd(n, m) = 1$, then

$$\phi(mn) = \phi(n)\phi(m) \tag{2.2.1}$$

This ϕ-function has some interesting applications

PROPOSITION 2.2.2. *If n and m are integers > 1 with* $\gcd(n, m) = 1$, *then*

$$m^{\phi(n)} \equiv 1 \pmod{n} \tag{2.2.2}$$

It follows that, for any integers a and b

$$m^a \equiv m^b \pmod{n} \tag{2.2.3}$$

whenever

$$a \equiv b \pmod{\phi(n)}$$

REMARK. Fermat proved this for n a prime number — in that case, it is called Fermat's Little Theorem.

EXERCISES.

1. Why is $7^{999} \equiv 7^{-1} \pmod{100}$?

2.3. The Goldbach Conjecture

The Goldbach Conjecture is that:

Every even integer > 2 is the sum of two primes.

Christian Goldbach (1690 – 1764) was a Prussian mathematician connected with some important research mainly in number theory; he also studied law and took an interest in and a role in the Russian court. After traveling around Europe in his early life, he landed in Russia in 1725 as a professor at the newly founded Saint Petersburg Academy of Sciences. Goldbach jointly led the Academy in 1737. However, he relinquished duties in the Academy in 1742 and worked in the Russian Ministry of Foreign Affairs until his death in 1764. He is remembered today for Goldbach's conjecture and the Goldbach–Euler Theorem. He had a close friendship with famous mathematician Leonard Euler, serving as inspiration for Euler's mathematical pursuits.

The conjecture has been shown to hold for all integers less than 4×10^{18} but remains unproven despite considerable effort. We'll use this as an opportunity to introduce several new Maxima commands, namely commands involving *sets* — see section E.11 on page 311.

A set is a kind of list delimited by curly brackets in which every element is *unique:* If you type

$$\{1, 2, 3, 2\}$$

Maxima comes back with

$$\{1, 2, 3\}$$

having eliminated the extra 2. We need three set-operations to explore Goldbach's conjecture:

(1) **integer_partitions**(n, m) — Returns a set of lists of length m of positive integers that sum up to n.
(2) **every**(F, s) — returns **true** if $F(x)$ is **true** for *every* $x \in s$, where s is a set or list.
(3) **subset**(a, F) — the subset of s of elements $x \in s$ for which $F(x)$ is **true**.

EXERCISES.

1. Find all the ways 100 can be written as a sum of two primes. Hint: look at the code on page 315.

2.4. Public-key cryptography

> "Well, a regular code is like a strongbox with a key. You lock your message in it and nobody can read it without the key."
>
> "I understand. But Ed needs the key to read the messages, right? How do you get it to him without the bad guys also seeing it?"
>
> "That's the beauty of this system. It's like a magic box that comes with two *different* keys. When you lock it with one key, only the *other* key can open it."
>
> "You can't use the original one?" she said.
>
> "No," I replied. "So, I send Ed one key and keep the other for myself. Even if the bad guys get his key, they can't use it to decode my messages. Only the one I keep will do that."
>
> — Constance Fairchild, in the novel *Bloodline* (with the author's permission). See [59].

The idea of a public-private key cryptosystem is attributed to Whitfield Diffie and Martin Hellman, who published the concept in 1976.

Bailey Whitfield 'Whit' Diffie (1944–) An American cryptographer and mathematician and one of the pioneers of public-key cryptography along with Martin Hellman and Ralph Merkle. Diffie and Hellman's 1976 paper, [16], introduced a radically new method of distributing cryptographic keys, that helped solve key distribution — a fundamental problem in cryptography. They lacked a good implementation of their ideas.

Martin Edward Hellman (1945–) is an American cryptographer and mathematician, best known for his involvement with public key cryptography in cooperation with Whitfield Diffie and Ralph Merkle. Hellman is a longtime contributor to the computer privacy debate, and has applied risk analysis to a potential failure of nuclear deterrence.
Hellman was elected a member of the National Academy of Engineering in 2002 for contributions to the theory and practice of cryptography.

In 1977, Ron Rivest, Adi Shamir and Leonard Adleman, described an efficient algorithm for public key encryption based on proposition 2.2.2 on page 15. A description of the algorithm was published in

August 1977, in Scientific American magazine's Mathematical Games column[2].

Clifford Cocks, an English mathematician working for the British intelligence agency Government Communications Headquarters (GCHQ), described an equivalent system in an internal document in 1973. His description was classified until the RSA algorithm appeared.

Ronald Linn Rivest (1945–) is a cryptographer and computer scientist whose work has spanned the fields of algorithms and combinatorics, cryptography, machine learning, and election integrity. He is an Institute Professor at the Massachusetts Institute of Technology (MIT) and a member of MIT's Department of Electrical Engineering and Computer Science and its Computer Science and Artificial Intelligence Laboratory.
Along with Adi Shamir and Len Adleman, Rivest is one of the inventors of the RSA algorithm. He is also the inventor of the symmetric key encryption algorithms RC2, RC4, and RC5, and co-inventor of RC6. (RC stands for "Rivest Cipher". He also devised the MD2, MD4, MD5 and MD6 cryptographic hash functions.

Adi Shamir (1952–) is an Israeli cryptographer. He is a co-inventor of the Rivest–Shamir–Adleman (RSA) algorithm (along with Ron Rivest and Len Adleman), a co-inventor of the Feige–Fiat–Shamir identification scheme (along with Uriel Feige and Amos Fiat), one of the inventors of differential cryptanalysis and has made numerous contributions to the fields of cryptography and computer science

Leonard Adleman (1945–) is an American computer scientist. He is one of the creators of the RSA encryption algorithm, for which he received the 2002 Turing Award. He is also known for the creation of the field of DNA computing.

The basic idea:

> Let p and q be two *large* (30 digits or more) primes, and let $n = pq$ so $\phi(n) = (p-1)(q-1)$. Now let a, b be integers such that
>
> $$ab \equiv 1 \pmod{\phi(n)}$$
>
> If $1 \leq x \leq n$ is any number, then
>
> $$(x^a)^b = x^{ab} \equiv x^1 \pmod{n}$$
>
> So, to encode x, raise it to the a^{th} power mod n. The "encoded message" is (n, y), where $y \equiv x^a \pmod{n}$.

[2] To the consternation of the CIA!

To "decode" the message, compute $y^b \bmod n$, resulting in the original a.

Make the value of n and a widely available. If anyone wants to send you a message, they raise it to the a^{th} power mod$n$ and transmit it. When you receive it, you raise it to the b^{th} power mod$n$ and retrieve the original message.

How would a malicious person crack this code? They know a and n because these numbers were widely publicized. If they could compute $\phi(n)$ it would be fairly easy[3] to compute $b \equiv a^{-1} \pmod{\phi(n)}$. So the whole problem of cracking this code boils down to computing $\phi(n)$, given n. It turns out that there's no *known* way of doing this without *factoring* n to get p and q.

The conventional way to factor numbers involves trying primes like $2, 3, \ldots, 17$ and reducing the size of the number until it is manageable. Suppose the *smallest* prime that divides a number has 30 digits! Factoring that number will be quite difficult. Faster algorithms for factoring numbers have been discovered[4], but they are not substantially faster in all cases.

This is called the *RSA encryption* algorithm after its developers' surnames. Since converting a long message to numbers and raising them to a high power is computationally expensive, the "messages" sent via this algorithm are usually *keys* for other, more conventional ciphers — which is why it's called a *key-distribution* algorithm.

Its security depends on the difficulty of factoring certain large numbers.

Nowadays, the RSA algorithm uses the Carmichael λ-function rather than the ϕ-function:

DEFINITION 2.4.1. If $n > 1$ is an integer, the *Carmichael function*, $\lambda(n)$, is the smallest integer $1 \leq k \leq n$ such that

$$a^k \equiv 1 \pmod{n}$$

for all $a \in \mathbb{Z}$ such that $\gcd(n, a) = 1$.

If p and q are primes, it turns out that $\lambda(p \cdot q) = \operatorname{lcm}(p-1, q-1) \leq (p-1)(q-1)$, and computing this *still* requires factoring $p \cdot q$. It's used these days simply because it is usually smaller than $\phi(p \cdot q)$[5].

Naturally, Maxima has a command for computing $\lambda(n)$:

zn_carmichael_lambda(n)

For instance

[3] As it turns out!

[4] See appendix D on page 295.

[5] So the computations are slightly faster.

```
totient(100)
```

returns 40, while

```
zn_carmichael_lambda(100)
```

returns 20.

We have the related command

```
zn_order(x,n)
```

which computes the lowest exponent t such that

$$x^t \equiv 1 \pmod{n}$$

The computation uses a factorization of $\phi(n)$ (i.e. **totient**(n)). Since this might be time-consuming (or practically impossible), the user can "help the command" by supplying such a factorization[6] as the second parameter in the notation of **ifactors.**

So our cryptographic scheme involve the following steps:

(1) Choose two large prime numbers $q < p$. To make factoring harder, p and q should be chosen at random, be both large and have a large difference: If

$$\frac{1}{2q}\left(\frac{p-q}{2}\right)^2$$

is *small, Fermat factorization* can easily factor $p \cdot q$ — see appendix D on page 295. For choosing the primes, the standard method is to choose random integers and test for primality (using **prime_p** in Maxima, for instance) until two primes are found. The primes p and q are kept secret.

(2) Compute $n = pq$. This product, n, is used as the modulus for both the public and private keys. Its length, usually expressed in bits, is the key length.

(3) Compute $\lambda(n) = \text{lcm}(p-1, q-1)$.

(4) Choose an integer e such that $2 < e < \lambda(n)$ and $\gcd(e, \lambda(n)) = 1$. e having a short bit-length and small Hamming weight (number of 1's in its binary representation) results in more efficient encryption — the most commonly chosen value for e is $2^{16} + 1 = 65537$. The smallest (and fastest) possible value for e is 3, but such a small value for e has been shown to be less secure in some settings. The *public key* is the pair *(n,e)*. This is widely publicized.

(5) e is released as part of the public key. Determine $d \equiv e^{-1} \pmod{\lambda(n)}$. The number, d, is kept secret as the *private key* exponent. The *private key* is (n, d) — this is kept secret.

[6] Acquired by some magic, perhaps!

CLAIM. Everyone who has ever purchased something on the network has used a public key cryptosystem. The web server (for instance, the vendor selling things) generates a public and private key pair. Then it sends the public key to the web browser, which replies with an encrypted message containing a randomly generated key for a secure *conventional* cryptosystem (the message also includes a code for the desired *type* of conventional cryptosystem; most browsers and servers support many of them). The web server decrypts that and all further communication between the web server and the browser is encrypted via the conventional system using that key.

Another important application of public-key cryptosystems is in *digital signatures*. This passage from *Bloodline* says it all:

> ...Then use your *private key* to lock your message in the box.
>
> Although your message is locked away, anyone can read it — using your public key to unlock it. That's fine — this time, your aim wasn't to *hide* the message. The very fact that your public key works proves *you* locked the message in the box: Only the *mate* of the key that locked the magic box can *unlock* it...
>
> — from the novel *Bloodline* (with the author's permission). See [59].

In real life, a kind of *summary* of the message (a MD5-hash, for instance) is encrypted with the private key (not the whole message!) and sent along with the original message.

2.5. Diffie-Hellman-Merkle key exchange

Ralph C. Merkle (1942–) is a computer scientist and mathematician. He is one of the inventors of public-key cryptography, the inventor of cryptographic hashing, and more recently a researcher and speaker on cryonics.
Merkle is a renowned cryptographer, known for devising Merkle's Puzzles, co-inventing the Merkle–Hellman knapsack cryptosystem, and inventing cryptographic hashing (Merkle–Damgård construction) and Merkle trees. He received the IEEE Richard W. Hamming Medal in 2010 and has published works on molecular manipulation and self-replicating machines. He also serves on the board of directors for the cryonics organization Alcor Life Extension Foundation and appears in the science fiction novel *The Diamond Age.*

This is a variation on the public-key cryptography described in the last section, in that there is *no* private key. Let $n > 1$ be an integer and consider the multiplicative set $\mathbb{Z}_n^\times$. This has $\phi(n)$ elements and

DEFINITION 2.5.1. Given an integer, n, a *primitive root* modulo n, $x \in \mathbb{Z}_n^\times$, is an element with the property that for any $y \in \mathbb{Z}_n^\times$ there exists an integer m such that $y = x^m \pmod{n}$.

REMARK. Primitive roots exist if $n = 2, 4, p^k$ or $2p^k$ with p a prime > 2 — see [23] or [67].

Maxima has a command for computing primitive elements if they exist:

```
zn_primroot(n)
```

or **false** if they don't.

The parameter **zn_primroot_limit** determines how many attempts it will make (the default is 1000). The computation uses a factorization of $\phi(n)$ (i.e. **totient**(n)). Since this might be time-consuming (or practically impossible), the user can "help the command" by supplying such a factorization as the second parameter in the notation of **ifactors**:

```
zn_primroot(n,factorization)
```

Example

```
p:2^142 + 217;
ifs:ifactors(totient(p));
g:zn_primroot(p, ifs);
```

Our *public key* is the pair (n, x) where x is a primitive root modulo n; there is no private key. When A and B wish to communicate, they both select random numbers a and b modulo n.

A sends B the message $x^a \pmod n$, and B sends A the message $x^b \pmod n$. When A receives this, he raises it to the a^{th} power modulo n, and B raises A's message to the b^{th} power modulo n.

As the end of this exchange, both A and B have a *shared secret*

$$x^{a\cdot b} \pmod n$$

that no one else knows. This secret can be used as a key for a more conventional (agreed-upon) cryptosystem[7] that is used for further communication.

Its security depends on computing $a \pmod n$, given n, x, and $x^a \pmod n$ — the so-called *discrete logarithm problem*: In a manner of speaking $a = \log_x x^a$, the *logarithm* of x^a — or it would be if computations were done over $\mathbb{R}$ rather than $\mathbb{Z}_n^\times$.

There are no known efficient algorithms for solving this other than raising x to all possible powers and comparing the result with $x^a \pmod n$. Since a might be a large number, this could be computationally expensive.

Compare this with the treatment of *elliptic-curve cryptography* in [60, section 6.2.2].

[7] Which may *also* be publicly-known.

After all this, it's important to mention that Maxima has a discrete logarithm command

```
zn_log(a,g,n)
```

If g is a primitive root modulo n, this solves the congruence $g^x \equiv a \pmod{n}$, if a solution exists.

EXERCISES.

1. Modify the key-exchange algorithm to give a shared secret to m people, where $m > 2$.

2. Implement *electronic signatures* using Diffie-Hellman-Merkle key exchange.

2.6. Continued fractions

Here's an example of a continued fraction:

$$\sqrt{2} = 1 + \cfrac{1}{2 + \cfrac{1}{2 + \cfrac{1}{2 + \cfrac{1}{2 + \cfrac{1}{2 + \cfrac{1}{2 + \cfrac{1}{2 + \cfrac{1}{2 + \cfrac{1}{2 + \frac{1}{2 \cdots}}}}}}}}}}$$

Euler developed much of the theory of continued fractions, proving that

$$\arctan(x) = \cfrac{x}{1 + \cfrac{1^2 x^2}{3 - x^2 + \cfrac{3^2 x^2}{5 - 3x^2 + \cdots}}}$$

for $|x| \leq 1$, where the general term is

$$\frac{(2k-1)^2 x^2}{2k + 1 - (2k-1)x^2 + \cdots}$$

Setting $x = 1$ gives a nice continued fraction for $\pi/4$.

The *standard form* for continued fractions have numerators equal to 1, and it can be proved that every continued fraction is equal to one in the standard form (see [49]).

In Maxima, continued fractions are represented as *lists*

$$\sqrt{2} = [1, 2, 2, 2, 2, \ldots]$$

The results of truncating a continued fraction at a point is called a *convergent* of the fraction. In the case of algebraic numbers like $\sqrt{2}$,

the terms repeat indefinitely and Maxima usually simply lists the sequence that repeats, so we get

```
cf(sqrt(2))
[1,2]
```

The **cf**-command attempts to find a continued fraction form of its parameter. It can work with linear combinations of square roots of integers (which all produce *repeating* continued fractions) and floating point numbers:

```
cf(%pi);
cf: %pi is not a continued fraction./*error message!*/

f:cf(float(%pi));
[3,7,15,1,292,1,1,1,2,1,3,1,14]
```

The command **cfdisrep** displays a continued fraction in its normal mode

```
cfdisrep(f)
```

returns

$$3+\cfrac{1}{7+\cfrac{1}{15+\cfrac{1}{1+\cfrac{1}{292+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{2+\cfrac{1}{1+\cfrac{1}{3+\cfrac{1}{1+\frac{1}{14}}}}}}}}}}}}$$

The reader might wonder why we're interested in continued fractions (aside from the intriguing display they form on a printed page!). The answer is that their *convergents* (i.e. the results of truncating them after some finite point on) are rational numbers that converge to a real number faster than any other known representation — if the continued fraction is in standard form.

For instance, the command

```
cf(sqrt(2))
```

returns

```
[1,2]
```

which is not the answer[8]! The square root of 2 is the *infinite* periodic continued fraction

```
[1,2,2,2,2,2,2,,...]
```

[8] It is a rational number!

where the first term is the only one that isn't repeated. In mathematical notation, the nonperiodic portion is usually distinguished via a semicolon:

```
[1;2,2,2,2,2,2,,...]
```

Maxima doesn't do this, which might be confusing if one doesn't know that $\sqrt{2}$ is irrational from the outset.

The parameter **cflength** determines the number of periods of a periodic continued fraction that will be displayed. The default is 1. The author recommends setting this to something > 1 !

In theory, then, if we want to build a computer that works with real numbers, we should store them as continued fractions. Unfortunately, performing basic arithmetic with continued fractions is difficult.

There's a lengthy theory of continued fractions and how they can be used to prove numbers are irrational or transcendental. Again, see [49].

CHAPTER 3

Basic algebra and calculus

> "L'algèbre n'est qu'une géométrie écrite; la géométrie n'est qu'une algèbre figurée." (Algebra is merely geometry in words; geometry is merely algebra in pictures)
> — Sophie Germain, [26]

3.1. Introduction

We can type z:(a+b)^5. Try typing **expand**(z) to eliminate the parentheses and multiply out $a + b$ five times. The result is

$$b^5 + 5ab^4 + 10a^2b^3 + 10a^3b^2 + 5a^4b + a^5$$

Maxima can also factor rational algebraic expressions[1]: type z:a^10+b^10 and **Cell▷Evaluate Cell(s)** , then **factor**(z) and **Cell▷Evaluate Cell(s)** to get

$$\left(a^8 - a^6b^2 + a^4b^4 - a^2b^6 + b^8\right)\left(a^2 + b^2\right)$$

In the above, the letter z is an *expression*, and we can plug values in for its variables. For instance, type z(a=1) and **Cell▷Evaluate Cell(s)** to get

```
(b^8 - b^6 + b^4 - b^2 + 1)*(b^2 + 1)
```

Maxima can solve equations with the *solve* command. Typing

```
solve(a*x^2+b*x+c=0,x)
```

solves for x and reproduces the familiar quadratic formula

$$\left[x = -\frac{\sqrt{b^2 - 4ac} + b}{2a}, x = \frac{\sqrt{b^2 - 4ac} - b}{2a}\right]$$

If we type

```
solve(a*x^3+b*x^2+c*x+d=0,x)
```

we get Tartaglia's formula for the roots of a *cubic* equation in figure 3.1.1 on the following page.

[1] See section 3.1.1 on page 30.

$$\left[x = \left(\frac{-1}{2} - \frac{\sqrt{3}\,i}{2}\right)\left(\frac{\sqrt{27\,a^2\,d^2 + \left(4\,b^3 - 18\,a\,b\,c\right)\,d + 4\,a\,c^3 - b^2\,c^2}}{2\,3^{\frac{3}{2}}\,a^2} + \frac{\frac{b\,c}{a\,a} - \frac{3\,d}{a}}{6} + \frac{(-1)\,b^3}{27\,a^3}\right)^{\frac{1}{3}} - \frac{\left(\frac{\sqrt{3}\,i}{2} + \frac{-1}{2}\right)\left(\frac{(-1)\,b^2}{9\,a^2} + \frac{c}{3\,a}\right)}{\left(\frac{\sqrt{27\,a^2\,d^2 + \left(4\,b^3 - 18\,a\,b\,c\right)\,d + 4\,a\,c^3 - b^2\,c^2}}{2\,3^{\frac{3}{2}}\,a^2} + \frac{\frac{b\,c}{a\,a} - \frac{3\,d}{a}}{6} + \frac{(-1)\,b^3}{27\,a^3}\right)^{\frac{1}{3}}} + \frac{(-1)\,b}{3\,a},$$

$$x = \left(\frac{\sqrt{3}\,i}{2} + \frac{-1}{2}\right)\left(\frac{\sqrt{27\,a^2\,d^2 + \left(4\,b^3 - 18\,a\,b\,c\right)\,d + 4\,a\,c^3 - b^2\,c^2}}{2\,3^{\frac{3}{2}}\,a^2} + \frac{\frac{b\,c}{a\,a} - \frac{3\,d}{a}}{6} + \frac{(-1)\,b^3}{27\,a^3}\right)^{\frac{1}{3}} - \frac{\left(\frac{-1}{2} - \frac{\sqrt{3}\,i}{2}\right)\left(\frac{(-1)\,b^2}{9\,a^2} + \frac{c}{3\,a}\right)}{\left(\frac{\sqrt{27\,a^2\,d^2 + \left(4\,b^3 - 18\,a\,b\,c\right)\,d + 4\,a\,c^3 - b^2\,c^2}}{2\,3^{\frac{3}{2}}\,a^2} + \frac{\frac{b\,c}{a\,a} - \frac{3\,d}{a}}{6} + \frac{(-1)\,b^3}{27\,a^3}\right)^{\frac{1}{3}}} + \frac{(-1)\,b}{3\,a},$$

$$x = \left(\frac{\sqrt{27\,a^2\,d^2 + \left(4\,b^3 - 18\,a\,b\,c\right)\,d + 4\,a\,c^3 - b^2\,c^2}}{2\,3^{\frac{3}{2}}\,a^2} + \frac{\frac{b\,c}{a\,a} - \frac{3\,d}{a}}{6} + \frac{(-1)\,b^3}{27\,a^3}\right)^{\frac{1}{3}} - \frac{\frac{(-1)\,b^2}{9\,a^2} + \frac{c}{3\,a}}{\left(\frac{\sqrt{27\,a^2\,d^2 + \left(4\,b^3 - 18\,a\,b\,c\right)\,d + 4\,a\,c^3 - b^2\,c^2}}{2\,3^{\frac{3}{2}}\,a^2} + \frac{\frac{b\,c}{a\,a} - \frac{3\,d}{a}}{6} + \frac{(-1)\,b^3}{27\,a^3}\right)^{\frac{1}{3}}} + \frac{(-1)\,b}{3\,a}\right]$$

FIGURE 3.1.1. Roots of a cubic equation

Niccolò Fontana Tartaglia (1499/1500 – 1557) was a mathematician, architect, surveyor, and bookkeeper in the then-Republic of Venice (now part of Italy). Tartaglia was the first to apply mathematics to computing the trajectories of cannonballs, known as ballistics, in his *Nova Scientia*, "A New Science."

He outlined his formula for the roots of a cubic polynomial in a poem based on Dante's *Inferno.*

Tartaglia had a tragic life. As a child, he was one of the few survivors of the massacre of the population of Brescia by French troops in the War of the League of Cambrai. His wounds made speech difficult or impossible, prompting the nickname Tartaglia ("stammerer").

We can also do this for a fourth degree polynomial, resulting in a much more complex formula. Something interesting happens if we go to the fifth degree. Maxima gives back the same polynomial we input. This is because *no general formula exists* for the roots of a polynomial of degree 5 or higher. See [58, chapter 8] for a proof of this.

To some extent, we can find *approximate* roots of polynomials (and other functions) numerically. Numeric methods have difficulty computing *complex* roots, and can fail in many cases. The advantage of the formulas is that they give exact answers (and they always work).

Suppose we have a polynomial x^4+2x^3-3x+5. Since it's fourth-degree, a formula exists for computing its roots exactly. Note that the messy equation in figure 3.1.1 on the preceding page is enclosed in *square brackets*. This means it is a Maxima-*list*. We access members of a list via square brackets and an integer, starting from 1. If we type

```
roots:solve(x^4+2*x^3-3*x+5=0,x)
```

the 4 roots are roots[1], roots[2], roots[3], and roots[4].

Typing roots[1] and **Cell▷Evaluate Cell(s)** gives

$$x=-\frac{\sqrt{-\frac{4\sqrt{3}}{\sqrt[3]{3\left(\frac{\sqrt{47597}\%i}{23\sqrt{2}}+\frac{29}{2}\right)^{\frac{2}{3}}+3\left(\frac{\sqrt{47597}\%i}{23\sqrt{2}}+\frac{29}{2}\right)^{\frac{1}{3}}+26}}-\left(\frac{\sqrt{47597}\%i}{23\sqrt{2}}+\frac{29}{2}\right)^{\frac{1}{3}}-\frac{26}{3\left(\frac{\sqrt{47597}\%i}{23\sqrt{2}}+\frac{29}{2}\right)^{\frac{1}{3}}}+2}}{2}-\frac{\sqrt{3\left(\frac{\sqrt{47597}\%i}{23\sqrt{2}}+\frac{29}{2}\right)^{\frac{2}{3}}+3\left(\frac{\sqrt{47597}\%i}{23\sqrt{2}}+\frac{29}{2}\right)^{\frac{1}{3}}+26}}{2\sqrt{3}}-\frac{1}{2}$$

If we apply the **bfloat**-command or menu-item **Numeric▷To Bigfloat** to the expression, we get the same expression, with decimal numbers instead of exact integers! This is a complex number, so we can apply the **rectform**-command or menu-item **Simplify▷Complex Simplification▷Convert to Rectform** to try to put it into standard complex notation. This gives us

$$x=-\frac{1.603712691810368b0\%i}{\sqrt{2}}-\frac{1.814527633159785b0}{\sqrt{2}}-5.0b-1$$

Applying the **bfloat**-command or menu-item **Numeric▷To Bigfloat** to *this* gives

$$x=-1.133996119454043b0\%i-1.783064794057666b0 \tag{3.1.1}$$

How do we check this? We use the **subst**-command or select **Simplify▷Substitute**. The command's format is

```
subst(new_value,old_variable,expression)
```

and it gives

$$\begin{aligned}&(-1.133996119454043b0\%i-1.783064794057666b0)^4\\&\quad+2.0b0(-1.133996119454043b0\%i-1.783064794057666b0)^3\\&-3.0b0\,(-1.133996119454043b0\%i-1.783064794057666b0)+5.0b0\end{aligned}$$

which is not particularly enlightening. How do we get Maxima to multiply out the parenthesized expressions? We use the **expand**-command or menu-item **Simplify▷Expand Expression** to get

$$6.661338147750939b - 16\%i + 4.662936703425657b - 15$$

which is *very* close to 0. This shows that equation 3.1.1 on the preceding page defines an (approximate) root of $x^4 + 2x^3 - 3x + 5$.

The next root, roots[2], turns out to be its complex conjugate

$$x = 1.133996119454043b0\%i - 1.7830647940576 6b0$$

roots[3] and roots[4] also turn out to be a complex conjugate pair.

If we try this with a fifth-degree polynomial

```
solve(x^5+2*x-5=0,x);
```

we get

$$\left[0 = x^5 + 2x - 5\right]$$

which simply says that the roots of this polynomial *are the roots* — i.e., Maxima cannot find exact roots. In this case, we can ask Maxima to use a *numeric* algorithm via

```
allroots(x^5+2*x-5);
```

to get

(3.1.2)

$$x = 1.208917813386895, x = 0.9409544200647337\%i - 1.167042002184507,$$
$$x = -0.9409544200647337\%i - 1.167042002184507,$$
$$x = 1.234436184384532\%i + 0.5625830954910601,$$
$$x = 0.5625830954910601 - 1.234436184384532\%i$$

In some cases, numeric algorithms do not converge.

3.1.1. Subtleties of factorization. The basic **factor** command works with rational real numbers. If you type

```
factor(x^4+1);
```

you get

$$x^4 + 1$$

implying that there is no other way to factor it. If you type

```
factor(x^4+1,a^2-2=0);
```

implying that the square root of 2 exists, you get

$$\left(x^2 - ax + 1\right)\left(x^2 + ax + 1\right)$$

Typing

```
expand((x^2-sqrt(2)*x+1)*(x^2+sqrt(2)*x+1));
```

you get

$$x^4 + 1$$

The optional second parameter to factor must be a polynomial of a *single* variable that represents a new element adjoined to the rational field. If you want to factor the polynomial in a field with several new element added, you must refer to the Primitive Element Theorem (theorem 7.2.13 in [58]).

EXERCISES.

1. Suppose we only want a list of the roots of a polynomial rather than a list of equations like x=root. Hint: use the **rhs** and **map** commands (look them up in the index or appendix E on page 299).

3.2. Functions and programming

We have seen that identifiers like 'z', 'a', or 'b' can represent *variables* or *expressions.* They can also equal *functions:* type f(x):=x^2−3*x+3 and **Cell▷Evaluate Cell(s)** to define the identifier f to be a *function.* Note that := is used to define a function. We can also define "anonymous" functions using the **lambda-**command:

```
lambda([x,y],x*y)
```

Having defined a function, we can *plot* it with the command **plot2d**(f,[x,0,3]) to get figure 3.2.1 on the following page

Standard form of this command:

```
plot2d(function /*or list*/ [f1,f2,...,fn],
[x,low_x,high_x]
/*optional:*/,[y,low_y,high_y]
 )
```

The *list* of functions allows you to plot multiple functions in a single plot. Note that text between /* and */ is regarded as a *comment* and is treated as white space by Maxima.

We can plot **lambda**-functions in a command

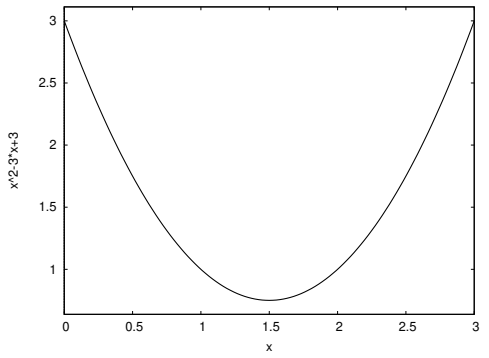


FIGURE 3.2.1. Simple plot

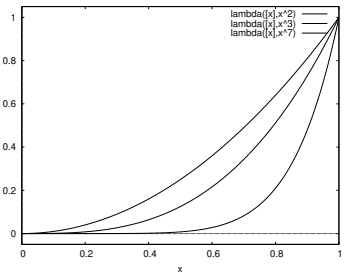


FIGURE 3.2.2. Lambda plots

```
plot2d([lambda([x],x^2),lambda([x],x^3),
lambda([x],x^7)],[x,0,1])
```

to get figure 3.2.2.

Now that we have functions, we can also do calculus. Maxima has a derivative-command that does what its name implies. Its format is

```
derivative(expression, variable);
```

An alternate way computing derivatives uses the **diff**-command

```
diff(expression, variable);
```

which also allows for *multiple* derivatives

```
diff(expression, variable,number);
```

So typing

```
diff(x^2, x,2);
```

gives 2.

Typing **derivative**(x^x, x); gives

$$x^x \left(\log\left(x\right)+1\right)$$

If the expression has several variables, this becomes the *partial* derivative with respect to the variable listed[2]. For instance, **derivative**(x^(x*y), x); and hitting **Cell▷Evaluate Cell(s)** gives

$$x^{xy} \left(\log\left(x\right) y + y\right)$$

and **derivative**(x^(x*y), y); and hitting **Cell▷Evaluate Cell(s)** gives

$$x^{xy+1} \log\left(x\right)$$

Since we can compute derivatives, we can compute Taylor series, using the **taylor**-command[3]

```
taylor(function, variable, center, highest_power);
```

So

```
taylor(sin(x),x,0,10);
```

gives

$$x - \frac{x^3}{6} + \frac{x^5}{120} - \frac{x^7}{5040} + \frac{x^9}{362880} + \ldots$$

Unfortunately, the Taylor series one gets is treated differently than a general polynomial. It doesn't get computed until a number is plugged into it. If you write

```
g(x):=taylor(sin(x),x,0,20);
```

and then type

```
g(2);
```

Maxima will complain that 2 is not a variable! A workaround is to type

```
at(g(x),x=2);
```

which will produce

$$\frac{241114102582}{265165275375}$$

The **at**-command evaluates a function *at* a value, and does this *after* the Taylor series has been computed. We could have also written

```
v(x):=at(taylor(sin(y),y,0,20),y=x);
```

[2]The listed variable is regarded as the *only* variable; all others are treated as *constants*.

[3]We'll leave out the "hitting **Cell▷Evaluate Cell(s)**" from now on; it's implied.

There is the closely related **powerseries**-command that attempts to compute a *formula* for the general coefficient. Its general form is

```
powerseries(function,variable,center);
```

For example

```
powerseries(sin(x),x,0);
/* Note that the number of terms is not specified*/
```

gives

$$\sum_{i1=0}^{\infty} \frac{(-1)^{i1} x^{2i1+1}}{(2i1+1)!}$$

In cases where the **powerseries**-command "doesn't know" a formula for the general term (for example, sin(sin(x))), it repeats the input. The **taylor**-command just computes derivatives and grinds out the taylor series to the required precision:

```
taylor(sin(sin(x)),x,0,10);
```

gives

$$x - \frac{x^3}{3} + \frac{x^5}{10} - \frac{8x^7}{315} + \frac{13x^9}{2520} + \cdots$$

Returning to f(x):=x^2−3*x+3, typing **integrate**(f(x),x) gives

$$\frac{x^3}{3} - \frac{3x^2}{2} + 3x$$

For definite integrals, we give the limits of integration: **integrate**(f(x),x,0,2); to get $\frac{8}{3}$. The general form of this command is

```
integrate(expression,variable)
```

with optional limits of integration. As with differentiation, the variable listed in the command is regarded as the *only* variable; the others are treated as *constants.* So

```
integrate(x*y*z,y)
```

results in

$$\frac{xy^2z}{2}$$

The **integrate**()-command "knows" all the rules of integration taught in a calculus course (or in a table of integrals at the back of a textbook). For instance, if you type **integrate**(1/(1+x^5),x), Maxima

comes back with

$$\frac{\sqrt{5}\left(\sqrt{5}+1\right)\arctan\left(\frac{4x+\sqrt{5}-1}{\sqrt{2\sqrt{5}+10}}\right)}{5\sqrt{2\sqrt{5}+10}}+\frac{\sqrt{5}\left(\sqrt{5}-1\right)\arctan\left(\frac{4x-\sqrt{5}-1}{\sqrt{-2\sqrt{5}+10}}\right)}{5\sqrt{-2\sqrt{5}+10}}$$
$$-\frac{\left(\sqrt{5}+3\right)\log\left(2x^2-x\left(\sqrt{5}+1\right)+2\right)}{10\left(\sqrt{5}+1\right)}-\frac{\left(\sqrt{5}-3\right)\log\left(2x^2+x\left(\sqrt{5}-1\right)+2\right)}{10\left(\sqrt{5}-1\right)}$$
$$+\frac{1}{5}\log\left(x+1\right)$$

This is clearly correct ☺!

In some cases, Maxima will simply return one's input. For instance, typing **integrate**(x^x,x) gives

$$\int x^x dx$$

which says "the integral *is* the integral". This is Maxima's way of saying it doesn't *"know"* how to get a more concrete formula for the integral of x^x.

If we type **integrate**(x*log(x), x, a, b);, Maxima asks us whether $1 < a$ and later, whether $a < b$. We can answer these questions beforehand via the **assume**-command:

```
assume(x>1)
assume(b>a)
```

type this prior to doing the integration, and Maxima won't ask questions.

Maxima has a complete programming language built into it for defining more complex functions. Suppose we want to define a function as follows:

$$f(x)=\begin{cases}0 & x<-1\\ 1 & -1\le x<0\\ x^2 & 0\le x\le 1\\ 0 & x>1\end{cases} \tag{3.2.1}$$

To implement this function, we need **if**-statements and relational operators — see table 3.2.1 on the following page. We could implement it as a complex nested **if**-statement

```
f(x):= if(x<-1) then 0
       else if (x<0) then 1
       else if (x<=1) then x^2
       else 0;
```

Maxima allows you to remove the space between **else** and **if** to form an **elseif-**command that does the same thing.

=	equality
>	left side greater than right
<	right side greater than left
<=	less than or equal
>=	greater than or equal
#	not equal

TABLE 3.2.1. Maxima relational operators

```
x:2;
block ([x:0,y,z], /*a local variable named x*/
        x:3,
        1);
/* x is still equal to 2 */
```

FIGURE 3.2.4. Local variables in a **block**-command

Maxima has a **block**-statement that can make it easier to define complex logical and other types of programs. Its general format given in figure 3.2.3.

```
block ([local_variables or empty list],
        statement1,
        statement2,
        ...,
        value);
```

FIGURE 3.2.3. the **block** command

The value at the end is the result of the **block**-command executing. The local variables are created inside the block and never conflict with variables of the same name outside of it. Figure 3.2.4 illustrates this. The list of local variables can also (optionally) assign initial values to them.

If there are no local variables, the **block**-command still requires an empty list. Another way to exit a block is with the **return**-command. It exits the block with whatever value (enclosed in parentheses) it has as its parameter.

To summarize:

A block is the word **block** followed by a comma-separated sequence in parentheses

(1) The first element is a list of local variables or an empty list.
(2) The remaining entries (before the last one) are expressions.
(3) The last entry is a (numeric or symbolic) value.

```
f(x) := block ([], /* no local variables */
        if(x<-1) then return (0),
        if(x<0) then return (1),
        if(x<=1) then return (x^2),
        0);   /*default final value */
```

FIGURE 3.2.5. f(x) written using a **block**-command

(4) **block** statements can be nested to any depth.

One exits the **block** by either

(1) dropping through the last entry, or
(2) a **return** statement. Note: this *only* jumps out of the **block** containing it, not necessarily out of the *function* in which it appears[4]. If there are several *nested* **blocks**, this must be taken into account.

So our discontinuous function in equation 3.2.1 could be coded as in figure 3.2.5.

Unfortunately, the **derivative** and **integrate** commands do not understand the logic of these little programs and we have to use a bit of ingenuity to compute them. For instance, to compute

$$\int_{-\infty}^{\infty} f(x)dx$$

we have to rewrite it as

$$\int_{-\infty}^{\infty} f(x)dx = \int_{-1}^{0} 1dx + \int_{0}^{1} x^2 dx = \frac{4}{3} \tag{3.2.2}$$

Plotting functions defined by programs also presents some special considerations. Maxima first tries to *evaluate* the function and *then* sends it to the plotting routines. For instance,

```
plot2d(f(x),[x,-5,5])
```

produces a very uninteresting result: figure 3.2.6 on the next page, where f(x) appears to be identically 0. This is because x starts out as < -1 and the first **if** statement is activated.

Oddly enough, we must suppress this initial evaluation of f(x) via the **quote**-command which sends the literal function-code to the plotting routines,

```
plot2d('f(x),[x,-5,5])
```

to produce figure 3.2.7 on the following page. Note that only a *single* quote is required.

[4]Which is somewhat atypical in programming languages.

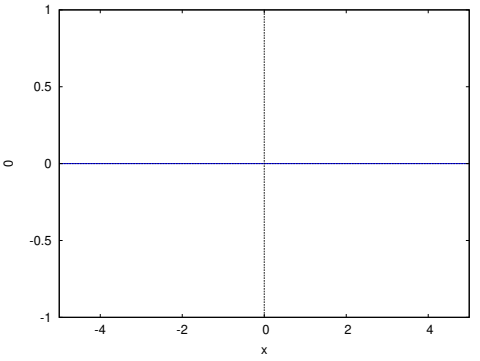


FIGURE 3.2.6. False plot of f(x)

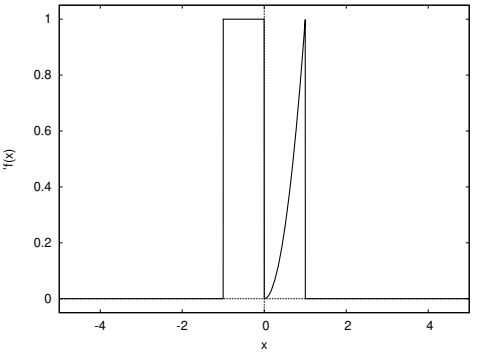


FIGURE 3.2.7. First plot of f(x)

It is somewhat difficult to see the behavior of this function from this particular plot. The plotting routines only use the minimum range of y-values necessary to represent the plot. It is better if we extend the range of y-values and restrict the x-values somewhat. Doing

```
plot2d('f(x),[x,-2,2],[y,-.5,1.5])
```

gives figure 3.2.8 on the next page

The programming language built into Maxima has many standard features that we will introduce as needed.

In many cases, it's difficult to analytically integrate a function and we have to resort to numerical methods. Calculus classes cover many methods for doing this, like Euler's Method, the Trapezoid Rule, and

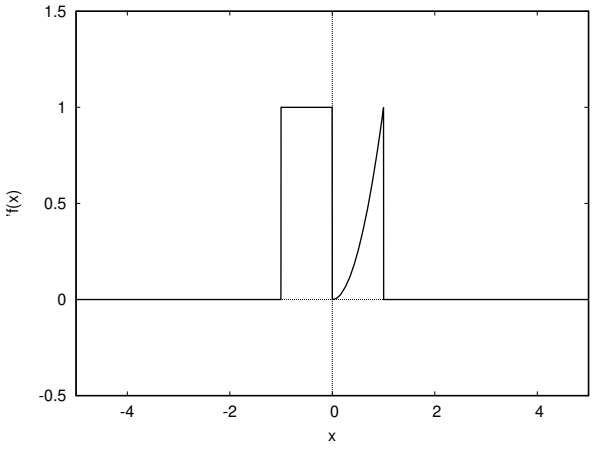


FIGURE 3.2.8. Better plot of f(x)

Simpson's Rule. One of the main Maxima commands for numeric integration is the **quad_qag**-command[5]. Its general format is

```
quad_qag(expression,variable,low,high,algorithm)
```

where algorithm is an integer from 1 to 6. The result is a list of four elements

```
[estimated integral,
estimated error,
number of iterations,
error code]
```

If we compute

$$\int_{.01}^{1} \sin\left(\frac{1}{x}\right) dx$$

via

```
integrate(sin(1/x),x,.01,1);
```

we get

$$\frac{2\sin(1) + \text{gamma\_incomplete}(0,i) + \text{gamma\_incomplete}(0,-i)}{2} - 0.5\,\text{gamma\_incomplete}(0,100i) - 0.5\,\text{gamma\_incomplete}(0,-100i) + 0.005063656411097588$$

[5]"quad" refers to *quadrature,* the act of estimating an area bounded by a curve (in ancient Greek, 'quad' literally refers to constructing a square of a given area). The author has no idea what 'qag' represents!

After typing **bfloat** and **expand**, we get

$$5.039818931754155b - 1$$

which we will regard as the (semi-)exact value. The numerical methods produce results close to this:

```
quad_qag(sin(1/x),x,.01,1,1);
```

produces

$$[0.5039818931754158, 2.307129694260507 * 10^{-9}, 615, 0]$$

Clearly, these numeric methods can produce very accurate estimates of definite integrals. Algorithm 6 gives a slightly more accurate estimate but the difference is not significant.

It is interesting that these numeric algorithms can handle functions defined like $f(x)$ defined in figure 3.2.5 on page 37. Typing

```
quad_qag(f(x),x,-5,5,1);
```

gives 0. If we change this to

```
quad_qag('f(x),x,-5,5,1);
```

we still get

$$[0.0, 0.0, 15, 0]$$

But if we use another algorithm,

```
quad_qag('f(x),x,-5,5,2);
```

we get

$$\left[1.333333332896747, 1.138680744111984 \cdot 10^{-8}, 2415, 0\right]$$

which is very close to the correct value of $4/3$. The other numeric algorithms give similar results.

EXERCISES.

1. There's a function $c(n)$ defined on positive integers by

$$c(n) = \begin{cases} 1 & \text{if } n = 1 \\ n/2 & \text{if } n \text{ is even} \\ 3n+1 & \text{if } n \text{ if odd} \end{cases}$$

The unsolved (as of 2024) *Collatz Conjecture* states that the process of replacing n by $c(n)$ over and over again eventually results in 1, regardless of what n started out as. Code this in Maxima and test this conjecture for various starting values of n. Hint: check the predicates in the appendix on the Maxima language.

3.3. Limits

Maxima can compute limits using L'Hôpital's rule and others. The **limit**-command has the format

```
limit(expression,variable,goal)
```

and can include an optional direction. For instance

```
limit((x^3-x)/(x+1),x,-1)
```

produces the result 2. The command

```
limit(x*log(x),x,zeroa)
```

takes the limit of $x\log(x)$ as $x \to 0^+$, and is completely equivalent to the command[6]

```
limit(x*log(x),x,0,plus)
```

3.4. Elimination theory

We consider the question

Given polynomials

$$f(x) = a_n x^n + \cdots + a_0 \tag{3.4.1}$$

$$g(x) = b_m x^m + \cdots + b_0 \tag{3.4.2}$$

when do they have a common root?

Sylvester studied this problem and solved it using a matrix from which he derived the *resultant* of the polynomials, $\mathrm{Res}(f, g, x)$.

PROPOSITION 3.4.1. *The polynomials $f(x)$ and $g(x)$ have a common root if and only if* $\mathrm{Res}(f, g, x) = 0$.

PROOF. See [58, section 6.2.4]. □

James Joseph Sylvester (1814–1897) was an English mathematician who made important contributions to matrix theory, invariant theory, number theory and other fields.

EXAMPLE. For instance, suppose we type

```
f:x^2-2*x+5;
g:x^3+x-3;
```

Then

```
resultant(f,g,x)
```

gives 169, so these two polynomials have no common roots.

[6] So the constants zeroa and zerob are unnecessary.

There are many interesting applications of the resultant. Suppose we are given parametric equations for a curve

$$\begin{aligned} x &= \frac{f_1(t)}{g_1(t)} \\ y &= \frac{f_2(t)}{g_2(t)} \end{aligned}$$

where f_i and g_i are polynomials, and want an implicit equation for that curve, i.e. one of the form

$$F(x, y) = 0$$

This is equivalent to finding x, y such that the polynomials

$$\begin{aligned} f_1(t) - xg_1(t) &= 0 \\ f_2(t) - yg_2(t) &= 0 \end{aligned}$$

have a common root (in t). So the condition is

$$\mathrm{Res}(f_1(t) - xg_1(t), f_2(t) - yg_2(t), t) = 0$$

This resultant will be a polynomial in x and y. We have *eliminated* the variable t — and the study of such algebraic techniques is the basis of Elimination Theory.

EXAMPLE 3.4.2. Let

$$\begin{aligned} x &= t^2 \\ y &= t^2(t+1) \end{aligned}$$

Then typing

```
resultant(t^2-x,t^2*(t+1)-y,t)
```

gives

$$y^2 - 2xy - x^3 + x^2$$

Issue the command

```
subst(t^2,x,y^2-2*x*y-x^3+x^2)
```

to get

```
y^2-2*t^2*y-t^6+t^4
```

and

```
subst(t^2*(t+1),y,y^2-2*t^2*y-t^6+t^4)
```

to get

```
t^4*(t+1)^2-t^6+t^4-2*t^4*(t+1)
```

Now, typing

```
expand(%)
```

gives 0. So

$$-x^3 + y^2 - 2yx + x^2 = 0$$

after plugging in the parametric equations for x and y.

What is the connection with elimination theory? If we had the equations

$$x - t^2 = 0$$
$$y - t^2(t+1) = 0$$

We could ask the question: "What conditions must x and y, *alone,* satisfy for these two equations to be satisfied?" or "How can we *eliminate* t from the original equations?"

Exercise 4 uses this to solve two simultaneous algebraic equations. This is the main application of the resultant. Solving more complex systems of algebraic equations requires a construction known as a Gröbner basis, which we will explore later.

EXERCISES.

1. Compute an implicit equation for the curve defined parametrically by
$$\begin{aligned} x &= t/(1+t^2) \\ y &= t^2/(1-t) \end{aligned}$$
2. Compute an implicit equation for the curve
$$\begin{aligned} x &= t/(1-t^2) \\ y &= t/(1+t^2) \end{aligned}$$
3. Compute an implicit equation for the curve
$$\begin{aligned} x &= (1-t)/(1+t) \\ y &= t^2/(1+t^2) \end{aligned}$$
4. Solve the equations
$$\begin{aligned} x^2+y^2 &= 1 \\ x+2y-y^2 &= 1 \end{aligned}$$
by computing a suitable resultant to eliminate y.
5. Find implicit equations for x, y, and z if
$$\begin{aligned} x &= s+t \\ y &= s^2-t^2 \\ z &= 2s-3t^2 \end{aligned}$$

Hint: Compute resultants to eliminate s from every pair of equations and then eliminate t from the resultants.

CHAPTER 4

Differential Equations

"Science is a differential equation. Religion is a boundary condition."
— Alan Turing.

4.1. Introduction

Suppose we have a first-order differential equation

$$\frac{dy}{dx} = f(x,y)$$

At each point, the function $f(x,y)$ defines a direction, i.e. a slope. A solution to the differential equation is a curve through the points whose slope matches the direction-field defined by $f(x,y)$. Intuition tells us that a solution passes through each point where $f(x,y)$ is well-defined. Simply draw a curve in the direction the arrows point. Intuition also tells us that this solution will be unique: if you steer a car the same way two times in a row, you end up at the same destination. This is the Cauchy-Lipschitz Theorem — see [65].

For instance, the equation

$$\frac{dy}{dx} = x + y \tag{4.1.1}$$

defines the direction-field in figure 4.1.1 on the following page, and a solution is the curve whose direction matches the arrows. We can see this direction-field by using the **plotdf**-command:

```
plotdf(x+y,[x,-2,2],[y,-2,2])
```

One nice feature of the resulting plot is that clicking on the plot produces a solution-curve (computed numerically) to equation 4.1.1 that passes through the point you clicked.

If the variables in the plot are not x and y, one must list them:

```
plotdf(u*v,[u,v],[u,-2,2],[v,-2,2])
```

This is a very complex command with many options that can be accessed from a menu on the plot itself or in the command-line. Each option in the command-line is enclosed in a list with the name of the option and its value:

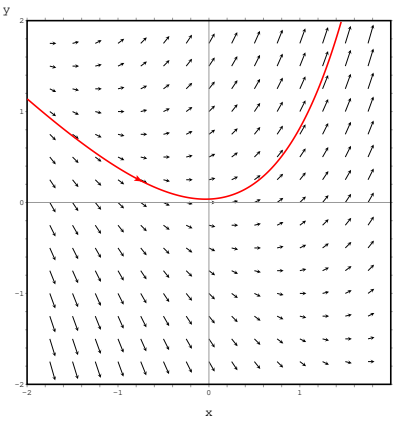


FIGURE 4.1.1. Direction-field defined by equation 4.1.1 on the preceding page

(1) [**tstep**,value] the size of the steps taken in approximating a solution to the differential equation. The default is .1.
(2) [**nsteps**,value] the number of steps taken to draw the solution-curve. Default is 100.
(3) [**direction**, option] defines the direction of the independent variable that will be followed to compute an integral curve. Possible values are **forward**, to make the independent variable increase nsteps times, with increments tstep, **backward**, to make the independent variable decrease, or **both** that will lead to an integral curve that extends nsteps forward, and nsteps backward. The keywords right and left can be used as synonyms for forward and backward. The default value is **both**.
(4) [**tinitial**,value] defines the initial value of variable t used to compute integral curves. Since the differential equations are autonomous, that setting will only appear in the plot of the curves as functions of t. The default value is 0. This refers to an *alternate* form of the **plotdf**-command in analyzing a *system* of *two* differential equations:

$$\begin{aligned} \frac{dx}{dt} &= f(x,y) \\ \frac{dy}{dt} &= g(x,y) \end{aligned} \tag{4.1.2}$$

and we plot the behavior of x *versus* y (the dependent variables could have other names, but the *independent* variable is always named t) in a command-line

```
plotdf([f,g],[x,y],[x,-2,2],[y,-2,2])
```

(5) [**versus_t**,number] is used to create a second plot window, with a plot of an integral curve, as two functions x, y, of the independent variable, t. If **versus_t** is given any value different from 0, the second plot window will be displayed. The second plot window includes another menu, similar to the menu of the main plot window. The default value is 0.
(6) [**trajectory_at**,coordinates] defines the coordinates xinitial and yinitial for the starting point of an integral curve. The option is empty by default. You can set this simply by clicking on the plot.
(7) ["parameter1=val1,parameter2=val2..."] defines a list of parameters, and their numerical values, used in the definition of the differential equations. The name and values of the parameters must be given in a string with a comma-separated sequence of pairs name=value.
(8) [**sliders**, "par1=min:max,par2=min:max..."] defines a list of parameters that will be changed interactively using slider buttons, and the range of variation of those parameters. The names and ranges of the parameters must be given in a string with a comma-separated sequence of elements name=min:max.
(9) [**xfun,** "function1,function2,..."] defines a string with semicolon-separated sequence of functions of x to be displayed, on top of the direction field.
(10) [x,min,max] sets up the minimum and maximum values shown on the horizontal axis. If the variable on the horizontal axis is not x, then this option should have the name of the variable on the horizontal axis. The default horizontal range is from -10 to 10.
(11) [y,min,max] sets up the minimum and maximum values shown on the vertical axis. If the variable on the vertical axis is not y, then this option should have the name of the variable on the vertical axis. The default vertical range is from -10 to 10.

Maxima "knows" the basic methods for symbolically solving first and second-order differential equations. One of the main commands for this is the **ode2**-command , which has the basic form

```
ode2(equation,dependent-var,independent-var)
```

For example

```
ode2('diff(y(x),x)=x+y(x),y(x),x)
```

results in

$$y(x) = \left((-x-1) * \%e^{-x} + \%c\right) * \%e^{x}$$

where %c is an arbitrary constant. We must quote the **diff**-command because we don't want it to *compute* the derivative; we just want to indicate that differentiation takes place. If we type

```
expand(%)
```

we get the simplified form

$$y(x) = \%c\%e^x - x - 1$$

The **ic1**-command selects (by adjusting the arbitrary constant) the solution that passes through a given point.

Given

```
sol:ode2('diff(y(x),x)=x+y(x),y(x),x)
```

and

```
ic1(sol,x=2,y(x)=3)
```

we get

$$y(x) = \%e^{-2}\left((y(2)+3)\,\%e^x - \%e^2 x - \%e^2\right)$$

and **expand**(%) gives

$$y(x) = y(2)\%e^{x-2} + 3\%e^{x-2} - x - 1$$

The **ode2**-command also handles *second*-order differential equations. For instance

```
sol:ode2(x*('diff(y,x,2))-'diff(y,x)+x=0,y,x);
```

produces the output

$$y = -\frac{2x^2\log(x) - x^2}{4} + \%k2x^2 - \frac{\%k1}{2}$$

where %k1 and %k2 are arbitrary constants[1].

As with first order differential equations, there's a command to set the arbitrary constants to appropriate values to satisfy initial conditions: the **ic2**-command.

```
ic2(sol, x=1, y=2, 'diff(y, x)=1);
```

produces

$$y = -\frac{2x^2\log(x) - x^2}{4} + \frac{x^2}{2} + \frac{5}{4}$$

[1] We can "absorb" the quotient by 2 into %k1!

This is a perfect opportunity to introduce the **ratsimp**-command which simplifies rational expressions[2]. Typing **ratsimp**(%) gives

$$y = -\frac{2x^2 \log(x) - 3x^2 - 5}{4}$$

In cases where Maxima doesn't "know" how to solve a differential equation, it returns with **False**.

Other interesting "simplification" commands are **radcan** and **fullratsimp**, which applies **ratsimp** over and over again until there's no change.

Consider the differential equation

$$\frac{dy}{dx} = \sqrt{\frac{y}{x}}$$

We code this via

```
z:ode2('diff(y(x),x)=sqrt(y(x)/x),y(x),x);
```

which produces a very unhelpful result

$$-\frac{2x\sqrt{\frac{y(x)}{x}} - 2y(x)}{\sqrt{y(x)}} = \%c$$

ratsimp, **expand**, and **solve** will not simplify this in any way. On the other hand **radcan** puts radicals in a standardized form and produces

$$2\sqrt{y(x)} - 2\sqrt{x} = \%c$$

at which point **solve** gives

$$y(x) = \frac{4x + 4\%c\sqrt{x} + \%c^2}{4}$$

EXAMPLE 4.1.1. The Logistics Equation. Imagine there is a population, P, and a disease is circulating through it. The function $y(t)$ is the number of people infected and, of course, $P - y(t)$ is the number uninfected. The probability of people getting infected is proportional to the product of these, so we get a differential equation

$$\frac{dy}{dt} = ky\left(1 - \frac{y}{P}\right) \tag{4.1.3}$$

We type

```
lsol:ode2('diff(y,t)=k*y*(1-y/P),y,t)
```

[2]Simplifying an expression is a complex process, and Maxima has several commands for doing this in different ways. It's not always clear what *constitutes* simplification.

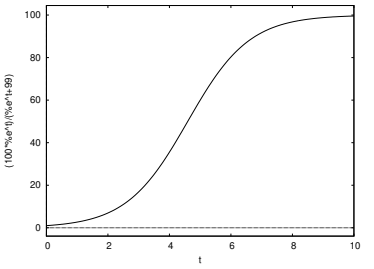


FIGURE 4.1.2. The Logistic Curve

and get

$$-\frac{\log(y-P)-\log(y)}{k}=t+\%c$$

Now, we decide that the first case of this disease happened at time 0 and issue the command

```
ic1(lsol,t=0,y=1);
```

to get

$$-\frac{\log(y-P)-\log(y)}{k}=\frac{kt-\log(1-P)}{k}$$

which doesn't quite solve the problem. If we issue the **logcontract**-command, we get

$$\frac{\log\left(\frac{y}{y-P}\right)}{k}=\frac{kt-\log(1-P)}{k}$$

which is slightly more useful. At this point, we can type

```
solve(%,y);
```

to get

$$\left[y=\frac{P\%e^{kt}}{\%e^{kt}+P-1}\right]$$

If we set $P=100$ and $k=1$, we can plot this via

```
plot2d((100*%e^t)/(%e^t+99),[t,0,10]);
```

to get the well-known *Logistic Curve* or the *Sigmoid Curve* in figure 4.1.2 Equation 4.1.3 on the preceding page was first proposed by Pierre-François Verhulst in modeling population growth with limited resources. In this case, P represents the carrying capacity of the environment. The logistic curve is also used to model bacterial growth.

Pierre-François Verhulst (1804 – 1849) was a Belgian mathematician from the University of Ghent. He is best known for the logistic growth model in his notable paper of 1845. His use of the term "logistic" was probably influenced by his association with the Belgian military (he briefly taught in their military academy). For the military, the word "logistics" represent supplies and shipping.

We also have the **desolve**-command for solving *systems* of linear ordinary differential equations. The general format is

```
desolve([list of equations],[list of functions])
```

EXAMPLE 4.1.2. Suppose we have equations

```
eqn_1: 'diff(f(x),x,2) = sin(x) + 'diff(g(x),x);
eqn_2: 'diff(f(x),x) + x^2 - f(x) = 2*'diff(g(x),x,2);
```

We solve them by typing

```
desolve([eqn_1,eqn_2],[f(x),g(x)])
```

and get a huge expression. We can simplify this somewhat by giving initial conditions

```
atvalue('diff(f(x),x),x=0,a);
atvalue(f(x),x=0,1);
```

specifying that $f(0) = 1$ and

$$\left.\frac{df}{dx}\right|_{x=0} = a$$

and re-issue the **desolve**-command to get

$$\mathrm{f}(x) = -\frac{3\sin(x)}{5} + \frac{\cos(x)}{5} + \frac{\left(\frac{\left(4(2a-2)-\frac{16}{5}\right)\sin\left(\frac{x}{2}\right)}{2} - \frac{8\cos\left(\frac{x}{2}\right)}{5}\right)\%e^{\frac{x}{2}}}{2} - \frac{2\%e^{-x}}{5} + x^2 + 2x + 2$$

and

$$\mathrm{g}(x) = -\frac{\sin(x)}{5} + \frac{2\cos(x)}{5} + \frac{\left(\frac{\left(\frac{2(10a-18)}{5}+\frac{16}{5}\right)\sin\left(\frac{x}{2}\right)}{2} + \frac{(10a-18)\cos\left(\frac{x}{2}\right)}{5}\right)\%e^{\frac{x}{2}}}{2} + \frac{2\%e^{-x}}{5} + 2x - a + \mathrm{g}(0) + 1$$

Important note: In using **desolve**, functions *must* be written as such: in other words, one must write $f(x)$ rather than just f!

Suppose we have a differential equation like

$$\frac{dy}{dx} = 3\sin(\sin(y)) \tag{4.1.4}$$

This is highly nonlinear, and **ode2** comes back with

$$\frac{\int \frac{1}{\sin(\sin(y))} dy}{3} = x + \%c$$

which isn't very helpful. Maxima's puny brain simply can't handle it. In this case, we are reduced to solving it *numerically*.

Euler proposed the first numeric algorithm for solving an equation like

$$\frac{dy}{dx} = f(x, y)$$

It involved replacing the derivative by finite differences:

$$\frac{\Delta y}{\Delta x} = f(x, y) \tag{4.1.5}$$

so

$$f(x_{i+1}) = f(x_i) + (x_{i+1} - x_i) \cdot f(x_i, y_i)$$

This crude approximation becomes more accurate the smaller $x_{i+1} - x_i$ becomes. Maxima uses a similar but more sophisticated algorithm called the fourth-order Runge-Kutta algorithm.

Carl David Tolmé Runge (1856 – 1927) was a German mathematician, physicist, and spectroscopist.
He was co-developer and co-eponym of the Runge–Kutta method, in the field of what is today known as numerical analysis. In addition to pure mathematics, he did experimental work studying spectral lines of various elements (together with Heinrich Kayser), and was very interested in the application of this work to astronomical spectroscopy.

Martin Wilhelm Kutta (1867 – 1944) was a German mathematician. In 1901, he co-developed the Runge–Kutta method, used to solve ordinary differential equations numerically. He is also remembered for the Zhukovsky–Kutta aerofoil (used in modern airplanes), the Kutta–Zhukovsky theorem and the Kutta condition in aerodynamics.

Maxima implements this with the **rk**-command, which takes one of the following two forms:

```
rk(diff-equation,dependent-variable,initial-value
[independent-var,start,finish,delta])
```

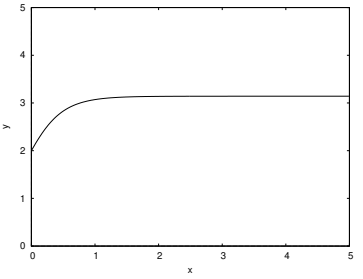


FIGURE 4.1.3. Output of the Runge-Kutta algorithm

Note: since the algorithm assumes that all equations are of the form

$$\frac{dy}{dx} = f(x,y)$$

you only list the '$f(x,y)$' in the algorithm.

The smaller delta is, the more accurate the approximation.

For equation 4.1.4 on the preceding page, we could program this as

```
rk(3*sin(sin(y)),y,2,[x,0,5,.01])
```

The command comes back with a long list of the form

```
[[x1,y1],[x2,y2],etc]
```

We can plot this using the **'discrete'** option to the **plot2d**-command. This has the general form

```
plot2d([discrete,[[x1,y1],[x2,y2], etc.],[y,lowy,highy])
```

So we run

```
point_list:rk(3*sin(sin(y)),y,2,[x,0,5,.01])
```

and

```
plot2d([discrete,point_list],[y,0,5])
```

to get the plot in figure 4.1.3. One nice feature of the **rk**-command is that it can handle *systems* of differential equations. In this form, it is coded

```
rk([ode1,ode2,etc,],[var1,var2,etc,],
[init1,init2,etc.],[independent-var,low,high,delta])
```

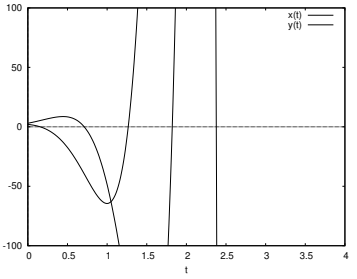


FIGURE 4.1.4. Plot of two solutions

We can solve the system

$$\frac{dx}{dt} = 3x - 4y \tag{4.1.6}$$
$$\frac{dy}{dt} = 2x + 3y$$

via

```
results:rk([3*x-4*y,2*x+3*y],[x,y],[2,3],[t,0,4,.01])
```

The output (results) is a list of the form

```
[[t1,x1,y1],[t2,x2,y2],etc.]
```

which must be reformatted (via the **makelist**-command) to suit the **plot2d**-command:

```
xgraph:makelist([p[1],p[2]],p,results)
```

and

```
ygraph:makelist([p[1],p[3]],p,results)
```

Now we can plot both solutions via

```
plot2d([[discrete,xgraph],[discrete,ygraph]],
[y,-100,100]);
```

to get figure 4.1.4.

EXERCISES.

1. Solve equations 4.1.6 on the facing page exactly via the **desolve**-command.

2. Convert the equation

$$\frac{dy^2}{dx^2} - 3\left(\frac{dy}{dx}\right)^3 + 2y = 0$$

into a system of first-degree equations suitable for solving via the **rk**-command.

3. Plot a solution to the differential equation

$$\frac{dy}{dx} = x - y^2$$

with $y(-1) = 3$ with a comparison plot of $\sqrt{x}$.

4. Plot solutions to the system of differential equations

$$\frac{dz}{dx} = v$$
$$\frac{dv}{dt} = -kz/m$$

through the point $(z, v) = (6, 0)$ with $k = 2$ and a slider for varying m from 1 to 5.

5. Try simplifying the output of **desolve** in example 4.1.2 on page 51 using **ratsimp**. The result will look more complicated, showing that simplification of an expression is a complex question with no obvious solution. Can these expresssions be further simplified?

4.2. Into the wild

Consider a forest with two populations: rabbits and foxes. The rabbits live in a leporine paradise — with unlimited resources allowing them to breed with reckless abandon. At least it *would* be a paradise were it not for the voracious foxes. Encounters between rabbits and foxes end badly for the rabbits and well for the foxes. Rabbits are the foxes' only sources of food.

The chances of rabbits meeting foxes is proportional to the product of their populations. If $r(t)$ and $f(t)$ represent the rabbit and fox

populations, respectively, we get the differential equations

$$\frac{dr}{dt} = \alpha r - \beta r \cdot f$$
$$\frac{df}{dt} = -\gamma f + \delta r \cdot f$$

where $\alpha, \beta, \gamma, \delta$ are nonnegative constants.

At first, we're tempted to use the **desolve** command to handle these equations. Unfortunately, the **desolve**-command only handles *linear* differential equations. The terms $r \cdot f$ make these equations very nonlinear.

Alfred James Lotka (1880 – 1949) was a US mathematician, physical chemist, and statistician, famous for his work in population dynamics and energetics. An American biophysicist, Lotka is best known for his proposal of the predator–prey model, developed simultaneously but independently by Vito Volterra. The Lotka–Volterra model is still the basis of many models used in the analysis of population dynamics in ecology.

Vito Volterra (1860 – 1940) was an Italian mathematician and physicist, known for his contributions to mathematical biology and integral equations, and being one of the founders of functional analysis.

We initially fall back on the trusty **plotdf**-command.

```
plotdf([a*r-b*r*f,-c*f+d*r*f], [r,f],
[parameters,"a=.2,b=.2,c=.1,d=.2"],
[sliders,"a=.1:5,b=.1:5,c=.1:5,d=.1:5"])
```

To produce figure 4.2.1 on the facing page. The fact that there is a closed curve shows that there is a periodic phenomena involved. It is interesting to move the sliders and see where the plot goes.

To see actual plots of foxes and rabbits, re-run the command with the **versus_t** option set to something nonzero:

```
plotdf([a*r-b*r*f,-c*f+d*r*f], [r,f],
[parameters,"a=.2,b=.2,c=.1,d=.2"],
[sliders,"a=.1:5,b=.1:5,c=.1:5,d=.1:5"],
[versus_t,1])
```

We get a second plot window with actual solutions for rabbits and foxes in figure 4.2.2 on the next page.

Again, moving the sliders around varies the behavior of the plots. They show that, when the fox-population is low, the rabbits freely multiply. Then the foxes have an abundant food-source and they multiply, causing the rabbit-population to plunge. This cycle repeats, with slight variations. This phenomena has been observed in the wild — see [53].

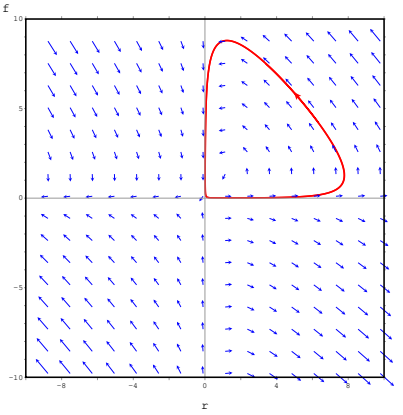


FIGURE 4.2.1. Plot of rabbits versus foxes

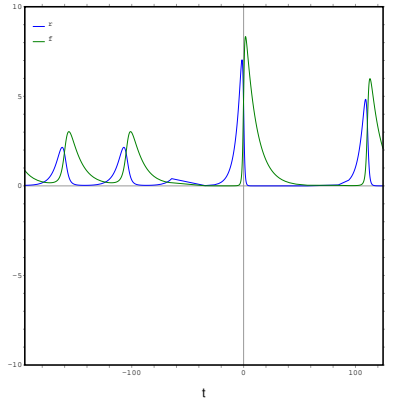


FIGURE 4.2.2. Rabbits and foxes

To get a more accurate (and quantitative solution) we can use the **rk**-command:

```
populations:rk([r-.01*r*f,-f+.01*r*f],[r,f],
[1000,10],[t,0,10,.01]);
```

If you examine the numbers coming from this simulation, you will notice *fractional* rabbits and foxes — the famous *atto-fox problem* — where an atto-fox is 10^{-18} of a fox[3]. See [42].

[3]In this simulation, you may see more atto-rabbits than foxes!

EXERCISES.

1. If a cannon is inclined 30°and fired with a muzzle-velocity of 1000 meters per second, what is the maximum altitude the shell will reach? Assume air-resistance is negligible and the acceleration of gravity is -9.8 m/second.

2. Same problem as the above with air-resistance given by

$$F = \frac{1}{2} C_D \rho A v^2 \tag{4.2.1}$$

where C_D is a dimensionless constant (assume it is .47), ρ is the density of the air (assume it is 1.225kg/m^3 at sea-level and doesn't change with altitude), and A is the cross-sectional area of the cannonball (assume it is .2 square meters). Assume the cannonball weighs 4kg.

4.3. The Heat Equation

Imagine a wire that is heated in some fashion. The flow and diffusion of heat through the wire is expressed by the one-dimensional *heat equation*

$$\frac{1}{a^2} \frac{\partial^2 \psi(x,t)}{\partial x^2} = \frac{\partial \psi(x,t)}{\partial t}$$

where $\psi(x,t)$ is temperature, x is distance, and t is time. We will discuss its "meaning" later. In this equation, a is a constant that represents how fast heat flows through the material in question; we will simplify matters by assuming it is 1.

In his research on this equation, Fourier discovered that *trigonometric polynomials* play an important part. What is a trigonometric polynomial? For our purposes, it is a linear combination

$$f(x) = b_0 + \sum_{j=1}^{k} a_j \sin(jx) + b_j \cos(jx) \tag{4.3.1}$$

where the coefficients, $\{a_i, b_i\}$ are real numbers.

Jean-Baptiste Joseph Fourier (1768 – 1830) was a French mathematician and physicist born in Auxerre and best known for initiating the investigation of Fourier series, which eventually developed into Fourier analysis and harmonic analysis, and their applications to problems of heat transfer and vibrations. The Fourier transform and Fourier's law of conduction are also named in his honor. Fourier is also generally credited with discovering the *greenhouse effect*.

Suppose someone provides us with an unknown function, $g(x)$, (a "black box" that gives us a function-value when we supply a value for x) and whispers "This is a trigonometric polynomial". How are we to check this claim or compute its coefficients?

If we type

```
integrate(sin(n*x),x,-%pi,%pi)
```

we get 0. If we type

```
integrate(cos(n*x),x,-%pi,%pi)
```

we get

```
2*sin(%pi*n)/n
```

Of course, this is for an *arbitrary* value of n (like 2.7, for instance). If we insist that n is an *integer*, via the **declare**-command

```
declare(n,integer)
```

then

```
integrate(cos(n*x),x,-%pi,%pi)
```

gives 0, so that

$$\int_{-\pi}^{\pi} \sin(nx)dx = \int_{-\pi}^{\pi} \cos(nx)dx = 0$$

for n an integer. For $f(x)$ in equation 4.3.1 on the facing page, it follows that

$$\int_{-\pi}^{\pi} f(x)dx = b_0 \int_{-\pi}^{\pi} dx = 2\pi b_0$$

so

$$b_0 = \frac{1}{2\pi} \int_{-\pi}^{\pi} f(x)dx \tag{4.3.2}$$

Next, Fourier noted that

$$\int_{-\pi}^{\pi} \sin(nx)\cos(mx)dx = 0 \tag{4.3.3}$$

because this is an *odd function*[4] integrated over a *symmetric* range. It follows that

$$\begin{aligned}\int_{\pi}^{\pi} f(x)\sin(nx)dx = a_1 \int_{-\pi}^{\pi} \sin(x)\sin(nx)dx + \\ \cdots + a_n \int_{-\pi}^{\pi} \sin^2(nx)dx + \cdots + a_k \int_{-\pi}^{\pi} \sin(nx)\sin(kx)dx \\ + 0 + \cdots + 0\end{aligned} \tag{4.3.4}$$

If we type

[4] A function, $f(x)$, is called *odd* if $f(-x) = -f(x)$.

```
declare(m, integer);
integrate(sin(n*x)*sin(m*x),x,-%pi,%pi);
integrate(cos(n*x)*cos(m*x),x,-%pi,%pi);
```

we learn that get

$$\int_{-\pi}^{\pi} \sin(nx)\sin(mx)dx = 0 \tag{4.3.5}$$

$$\int_{-\pi}^{\pi} \cos(nx)\cos(mx)dx = 0 \tag{4.3.6}$$

if $n \neq m$.

Incidentally, the reader might wonder what difference there is between the **assume**-command on page 35 and the **declare**-command used here. The **assume**-command describes *numeric* relations (usually inequalities) that exist between numeric identifiers, and the **declare**-command describes *properties* identifiers have (they might not be numeric ones).

If we type

```
integrate(sin(n*x)^2,x, -%pi,%pi)
```

we learn that

$$\int_{-\pi}^{\pi} \sin(nx)\sin(nx)dx = \pi \tag{4.3.7}$$

It follows that equation 4.3.4 on the previous page can be rewritten as

$$\int_{\pi}^{\pi} f(x)\sin(nx)dx = a_1 \cdot 0 + \cdots + a_n \cdot \pi + \cdots + a_k \cdot 0$$

from which we get

$$a_n = \frac{1}{\pi}\int_{-\pi}^{\pi} f(x)\sin(nx)dx \tag{4.3.8}$$

A similar line of reasoning shows that

$$b_n = \frac{1}{\pi}\int_{-\pi}^{\pi} f(x)\cos(nx)dx \tag{4.3.9}$$

So we have an answer to our question:

> $g(x)$ is a trigonometric polynomial if only a finite number of the a_n and b_n, computed using equations 4.3.8 and 4.3.9 are nonzero.

This would've ended matters if Fourier hadn't taken the next step: apply these equations to a function that is definitely *not* a trigonometric polynomial — for instance the bizarre function, $f(x)$, plotted in figure 3.2.8 on page 39! Recall that it is defined via

```
f(x) := block ([], /* no local variables */
        if(x<-1) then return (0),
        if(x<0) then return (1),
        if(x<=1) then return (x^2),
        0);
```

We will define functions to compute the coefficients in equations 4.3.8 on the preceding page and 4.3.9 on the facing page (following the example of equation 3.2.2 on page 37):

```
a(k):= (integrate(sin(k*x),x,-1,0)
+integrate(sin(k*x)*x^2,x,0,1))/%pi
```

If we type a(3), we get

$$\frac{\frac{2\sin(2)-\cos(2)}{4}+\frac{\cos(2)-1}{2}-\frac{1}{4}}{\pi}$$

which is a bit awkward. This expression should be simplified or consolidated. If we type **ratsimp**(%), we get

$$\frac{2\sin(2)+\cos(2)-3}{4\pi}$$

which is more compact. We incorporate this into our function for a(k):

```
a(k):= ratsimp((integrate(sin(k*x),x,-1,0)
+integrate(sin(k*x)*x^2,x,0,1))/%pi)
```

We also have a similar function to compute the cosine coefficients

```
b(k):= ratsimp((integrate(cos(k*x),x,-1,0)
+integrate(cos(k*x)*x^2,x,0,1))/%pi)
```

If we define b0 via equations 4.3.2 on page 59 and 3.2.2 on page 37,

```
b0:2/(3*%pi);
```

Now we write a function to add up terms of the trigonometric polynomials with these coefficients. There are several ways to do this. We'll start with the **while**-command with a general format

```
        while condition do
        (
        statement1,
        statement2,
        ...
        statementn
        )
```

To compute the sum of the first n terms of our trigonometric polynomial, we code

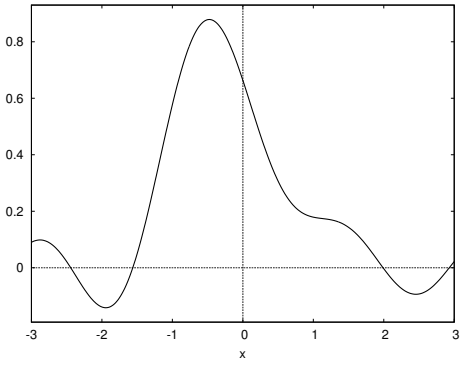


FIGURE 4.3.1. First three terms

```
first_n(n,x):= block (
        [sum:2/(3*%pi), k:1], /*local variables*/
        while k<=n do
        (
        sum:sum+a(k)*sin(k*x)+b(k)*cos(k*x),
        k:k+1              /*increment the counter*/
        ),
        sum                /* value returned */
        )
```

Now we can plot these trigonometric polynomials via

```
plot2d(first_n(3,x),[x,-3,3]);
```

to get figure 4.3.1. This doesn't tell us much but, like Fourier, we persist.

The sum of the first 10 terms gives figure 4.3.2 on the next page, which is evocative. Plotting this with f(x) (or Wxmaxima-menu item **Plot▷Plot 2d** via

```
plot2d([first_n(10,x),'f(x)],[x,-3,3]);
```

gives figure 4.3.3 on the facing page.

At this point, we go for broke and compare the first 100 terms via

```
plot2d([first_n(100,x),'f(x)],[x,-3,3]);
```

to get figure 4.3.4 on page 64.

This is *very* evocative! Although f(x) is not a trigonometric polynomial, an infinite series of trigonometric terms seems to converge to it almost everywhere. This is the famous Fourier Series, and was the beginning of a whole field of mathematics called harmonic analysis. Notice that Fourier series are more "powerful" than, say, Taylor series.

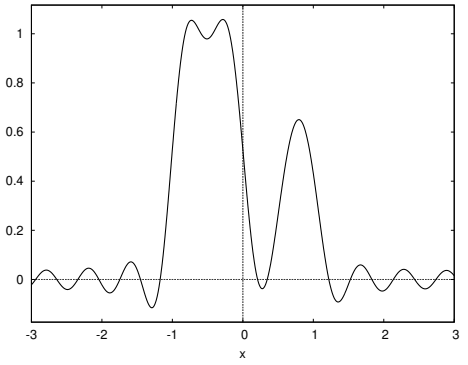


FIGURE 4.3.2. First 10 terms

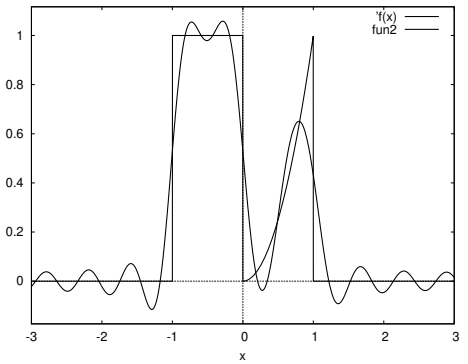


FIGURE 4.3.3. Comparison of first 10 with f(x)

They can represent functions that are not necessarily differentiable or even continuous.

Since all the functions that go into a Fourier series are periodic, so is the series itself — see figure 4.3.5 on the following page.

You may notice that the Fourier series "overshoots" and "undershoots" f(x) at the points where it is discontinuous. This is called *Gibbs Phenomena* and is illustrated by typing

```
plot2d([first_n(100,x)-'f(x))^2],[x,-3,3]);
```

to get figure 4.3.6 on page 65.

This does *not* go away as we add more terms; the peaks simply become narrower. This leads to the question:

In what sense does the Fourier series converge to $f(x)$?

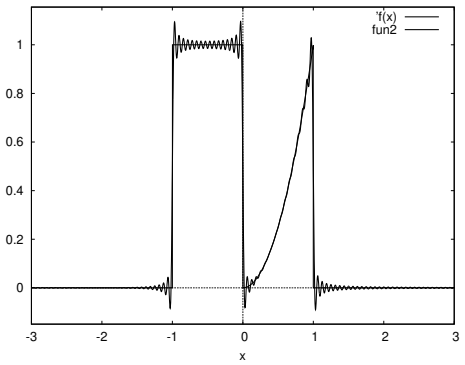


FIGURE 4.3.4. The first 100 terms

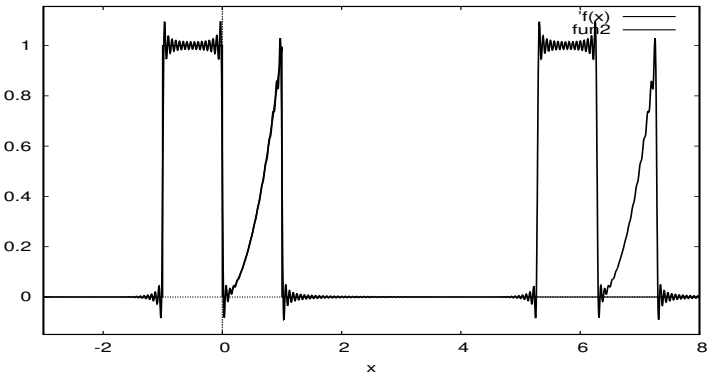


FIGURE 4.3.5. Periodicity of a Fourier series

It turns out[5] that if $f(x)$ is any function that can be integrated from $-\pi$ to π and if $S_n(x)$ is the sum of the first n terms of the Fourier series for $f(x)$, then

$$\lim_{n\to\infty} \int_{-\pi}^{\pi} (S_n(x) - f(x))^2 \, dx = 0 \tag{4.3.10}$$

In other words, the "space between the curves" of $f(x)$ and $S_n(x)$ goes to zero as n goes to infinity. This is called L_2-convergence.

If the original function, $f(x)$, is *continuous,* the "space between the curves" of $f(x)$ and $S_n(x)$ going to zero intuitively implies that $S_n(x)$ converges to $f(x)$ for every value of x. This is called *"pointwise convergence"*. See [54] for the details.

Claim: Virtually all readers of this book have used Fourier series.

[5] In other words, it is well-known but we won't prove it here. See [54].

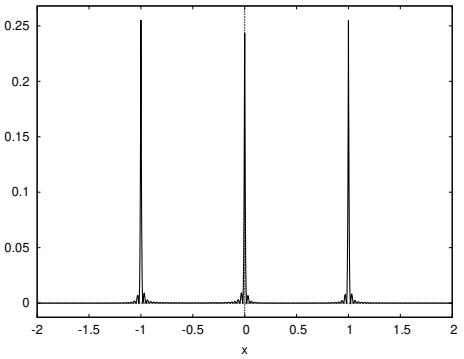


FIGURE 4.3.6. Gibbs Phenomena

How? The mp3 audio and the jpeg graphic formats are based on Fourier series. The jpeg format uses a two-dimensional version of it. The actual jpeg *file* is a string of Fourier coefficients. If the spikes in the Gibbs phenomena are narrower than a pixel, they have no effect on the final image. Something similar happens with mp3 files: the narrow spikes are high-frequency signals above the range of human hearing.

Maxima has a **sum**-command that would have eliminated the need for programming! Its general form is

```
sum(expression,index_variable,low,high);
```

and we could've written our function as

```
first_n(k,x):= 2/(3*%pi)
        +sum(a(j)*sin(j*x)+b(j)*cos(j*x),j,1,k);
```

4.4. Solution to the Heat Equation

At this point, the reader may wonder what all of this has to do with the Heat Equation. Recall that this is

$$\frac{1}{a^2}\frac{\partial^2\psi(x,t)}{\partial x^2} = \frac{\partial\psi(x,t)}{\partial t}$$

where we assume $a = 1$. Fourier attempted a particularly simple solution in the form

$$\psi(x,t) = u(x)\cdot v(t)$$

Plugging this into the heat equation gives

$$u''v = uv'$$

and we divide by uv to get

$$\frac{u''(x)}{u(x)} = \frac{v'(t)}{v(t)}$$

How can a function of one variable equal another of an *unrelated* variable? They both equal the same constant! So we have

$$\frac{u''(x)}{u(x)} = \frac{v'(t)}{v(t)} = c$$

We have

$$v'(t) = c \cdot v(t)$$

This is a simple differential equation, but we'll pretend we don't know the solution and use Maxima's **ode2**-command for solving ordinary differential equations of degree ≤ 2. We'll start with the equation for $v(t)$:

```
ode2('diff(v,t)=c*v,v,t);
```

Note that we must *quote* the **diff**-command because we don't want Maxima to try to *compute* a derivative; we just want to indicate that differentiation takes place.

We get

$$v = \%c\%e^{ct}$$

Here %c is an arbitrary constant that is completely unrelated to c. If $c > 0$, then, depending on the sign of %c, we realize that the temperature becomes exponentially hot over time or exponentially cold.

This is a reminder that not all solutions of the heat equation *physically occur!*

To avoid being burned alive or frozen, we'll assume that $c < 0$. This is traditionally written as

$$\frac{u''(x)}{u(x)} = \frac{v'(t)}{v(t)} = -\lambda$$

where $\lambda > 0$. The command

```
ode2('diff(v,t)=-lambda*v,v,t);
```

gives

$$v = \%c\%e^{-t\,\mathrm{lambda}}$$

and the command

```
ode2('diff(u,x,2)=-lambda*u,u,x);
```

prompts the question of whether lambda is positive, negative or zero[6]. We answer 'positive' and get

$$u = \%k1 \sin\left(x\sqrt{\mathrm{lambda}}\right) + \%k2 \cos\left(x\sqrt{\mathrm{lambda}}\right)$$

[6]Of course we could've preceded the **ode2** command with **assume**(lambda>0).

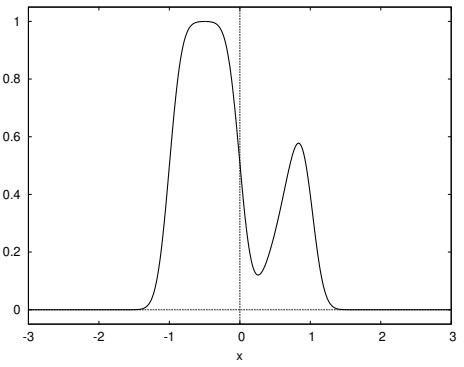


FIGURE 4.4.1. $\psi(x, .01)$

where %k1 and %k2 are arbitrary constants. This gives a basic solution to the Heat Equation

$$\psi(x,t) = \left(\%\text{k1}\sin\left(x\sqrt{\text{lambda}}\right) + \%\text{k2}\cos\left(x\sqrt{\text{lambda}}\right)\right)e^{-\text{lambda}t}$$

Since the Heat Equation is *linear,* any linear combination of these basic solutions is also a solution.

At this point, we can set $\sqrt{\text{lambda}} = n$, an integer, and get a basic solution[7]

$$\psi_n(x,t) = (\%\text{k1}_n\sin(nx) + \%\text{k2}_n\cos(nx))\,e^{-n^2t}$$

When $t = 0$, this looks like a term of a trigonometric polynomial. We hit upon Fourier's solution to the Heat Equation:

(1) expand $\psi(x, 0)$ — the *initial* heat distribution — in a Fourier series,
(2) Multiply the n^{th} term of this Fourier series by e^{-n^2t}. The resulting series defines $\psi(x, t)$ for $t \geq 0$.

We can test this with our Fourier series for the discontinuous function $f(x)$ defined in equation 3.2.1 on page 35. We replace our command for partial sums of this with

```
psi_n(k,x,t):= 2/(3*%pi)
        +sum((a(j)*sin(j*x)
        +b(j)*cos(j*x))*%e^(-j^2*t),j,1,k);
```

Figures 4.4.1 through 4.4.4 on the following page shows the time-evolution of ψ: Heat flows from the hotter parts of the wire to the cooler ones. The "sharp" edges of the function become smooth, and it's clear that the heat-distribution becomes constant in the limit as $t \to \infty$.

[7] We have absorbed the constant %c into %k1 and %k2.

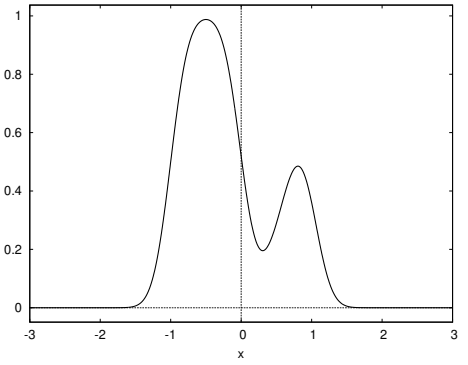


FIGURE 4.4.2. $\psi(x, .02)$

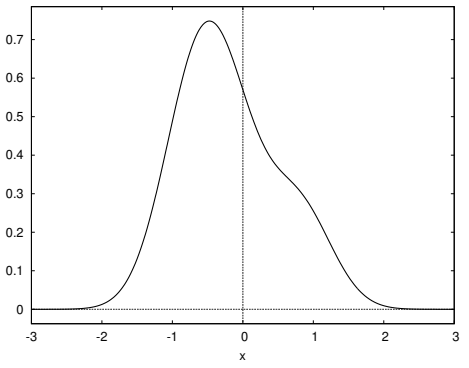


FIGURE 4.4.3. $\psi(x, .1)$

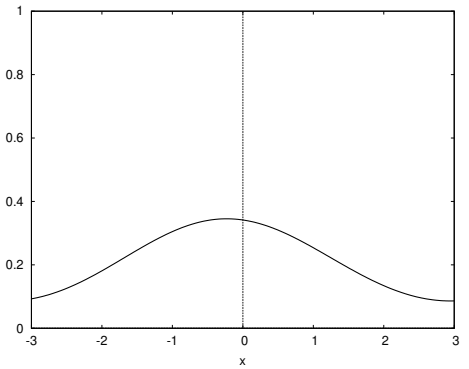


FIGURE 4.4.4. $\psi(x, 1)$

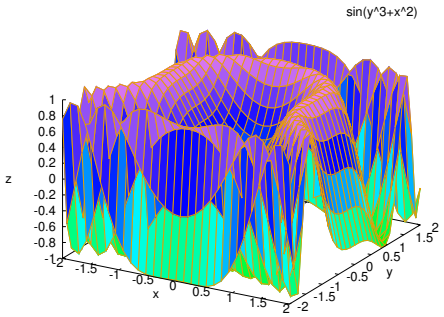


FIGURE 4.5.1. An example of plot3d

4.5. Finer points of plotting

Maxima has no built-in plotting capabilities. It uses a very powerful independent software package called Gnuplot (automatically installed with Maxima). It also uses powerful plotting commands built into wxMaxima.

The commands **plot2d** (and **plot3d**!) only use the most basic features of Gnuplot. Since we have mentioned **plot3d** we may as well discuss it. It's general form is

```
plot3d(two-variable-expression,[x,low,high],[y,low,high]);
```

For example

```
plot3d(sin(x^2+y^3),[x,-2,2],[y,-2,2]);
```

produces figure 4.5.1. The wxMaxima-menu **Plot▷Plot 3d** prompts you for all the necessary parameters. One nice thing about these plots is that you can rotate them with the mouse and see them from many different angles.

We can use the complete repertoire of Gnuplot commands to generate plots and diagrams. We would like to produce an animated image of the heat-flow in the wire. wxMaxima provides the **with_slider_draw**-command for for this. Its general format is given in figure 4.5.2. This command cycles through

```
with_slider_draw(
        variable, /*variable to attach to the slider */
        list of values,
        plot-command, /*plot */
        plot-options /*optional
             graphic command */
        ); /* end of with_slider_draw-command */
```

FIGURE 4.5.2. **with_slider_draw**

```
with_slider_draw(
        t, /*variable to attach to the slider */
        makelist(j,j,0,100), /*list of integers */
        explicit(psi_n(100,x,.01*t),x,-%pi,%pi),/*plot */
        yrange= [0,1.2] /*optional
             graphic command */
        ); /* end of with_slider_draw-command */
```

FIGURE 4.5.3. Evolution of the heat equation

the list of values, setting the variable to each of them and runs the corresponding plot command.

We would like to plot the flow of heat in our heated wire. We use the command in figure 4.5.3.

and wait a *long* time (Maxima is computing 101 plots of the ψ-function). Afterward, a window appears with a slider that animates the passage of time. The slider allows us to move time forward or backward and see how the process changed. One can right-click this plot to save it as an animated gif file.

In this context, **explicit** means plotting in the normal fashion that **plot2d** follows. The alternative is **implicit**, which plots points satisfying an *equation*

```
draw2d(implicit(x^4+y^4=1,x,-2,2,y,-2,2))
```

Before you click **Cell▷Evaluate Cell(s)**, several explanations are in order. The second parameter of **with_slider_draw** must be a *list,* i.e. data like

$$[1,2,3]$$

This particular list would make for a pathetic animation, though — with only 3 frames! What we really need is a list of integers from 0 to 100. Rather than typing out 101 numbers, we use the all-important **makelist**-command. It has several forms and can be used to create or modify lists.

(1) **makelist**(expression, variable, i0, i1) Makes a list of the expression with the variable set equal to integers from i0 to i1 (incremented by 1 each iteration). For instance

```
makelist(i, i, 1, 5)
```

produces the list

$$[1,2,3,4,5]$$

and

```
makelist(x=i^2, i, 1, 5)
```

produces the list

$$[x=1, x=4, x=9, x=16, x=25]$$

(2) **makelist**(expression, variable, list) Cycles through the elements of the list, setting the variable equal to each of them and creating a list of the expression evaluated at these values. For instance,

```
makelist(x=y, y, [a,b,[1,2]])
```

produces

$$[x = a, x = b, x = [1,2]]$$

This form of the command can be used to *reformat* lists. Suppose

$$a\text{: } [[1,2,3], [u,v,w], [i,j,k]]$$

and we issue the command **makelist**([p[2],p[1],p[2]], p,a). We get the result

$$[[2,1,2], [v,u,v], [j,i,j]]$$

Incidentally, these repeated computations (of a(k) and b(k)) could benefit from a process called *memoization*. Since $a(k)$ and $b(k)$ *only* depend on k it would be better if the functions stored their results and simply did a table-lookup whenever the same value of k is used a second time. This is called *memoizing* the computations. Maxima makes this very simple:

```
a[k]:= ratsimp((integrate(sin(k*x),x,-1,0)
        +integrate(sin(k*x)*x^2,x,0,1))/%pi)
b[k]:= ratsimp((integrate(cos(k*x),x,-1,0)
        +integrate(cos(k*x)*x^2,x,0,1))/%pi)
```

All we have done here is replace a(k) by a[k] and b(k) by b[k]. This is a signal to Maxima to store the computed values in an *array.* If an array-position already has a value in it, Maxima suppresses rerunning the function and simply returns the array-entry. This creates a problem if the program has bugs: it remembers the *incorrect* values. To erase these incorrect values, issue the **kill**-command: **kill**(a), **kill**(b).

We also have to rewrite the psi_n function slightly.

```
psi_n(k,x,t):= 2/(3*%pi)
        +sum((a[j]*sin(j*x) /*replaced a(j) by a[j] */
        +b[j]*cos(j*x))*%e^(-j^2*t),j,1,k);
```

Now you can click **Cell▷Evaluate Cell(s)**!

For more information on plotting, see Appendix F on page 323.

4.6. The Wave Equation

4.6.1. Introduction. The one-dimensional *wave equation* looks like the heat equation with a slight difference

$$\frac{\partial^2 \psi(x,t)}{\partial x^2} = \frac{1}{c^2}\frac{\partial^2 \psi(x,t)}{\partial t^2} \tag{4.6.1}$$

— the time derivative is *second*-order. One is to imagine a vibrating string, where the function $\psi(x,t)$ represents the displacement of the string at any given position and time.

The one-dimensional version was discovered by d'Alembert; the higher dimensional wave equation was discovered by Euler.

Jean-Baptiste le Rond d'Alembert (1717 – 1783) was a French mathematician, mechanician, physicist, philosopher, and music theorist. Until 1759 he was, together with Denis Diderot, a co-editor of the Encyclopédie. D'Alembert's formula for obtaining solutions to the wave equation is named after him. The wave equation is sometimes referred to as d'Alembert's equation, and the fundamental theorem of algebra is named after d'Alembert in French.

D'Alembert found a completely general solution to the one-dimensional wave equation:

$$\psi(x,t) = f(x+ct) + g(x-ct)$$

where f and g are arbitrary twice-differentiable *functions*[8]. As clever as this is, it is not clear how apply it to interesting situations. We will use a Fourier series approach.

As before, we assume $c = 1$ and write

$$\psi(x,t) = u(x) \cdot v(t)$$

Plugging this into equation 4.6.1 on the preceding page gives

$$u''(x) \cdot v(t) = u(x) \cdot v''(t)$$

or

$$\frac{u''(x)}{u(x)} = \frac{v''(t)}{v(t)} = -\lambda$$

and (via **ode2**, for instance) we get

$$u(x) = \alpha \cos(x\sqrt{\lambda} + \beta \sin(x\sqrt{\lambda})$$
$$v(t) = \gamma \cos(t\sqrt{\lambda}) + \delta \sin(t\sqrt{\lambda})$$

where $\alpha, \beta, \gamma, \delta$ are arbitrary constants.

Now imagine that our string is fixed between supports at $x = 0$ and $x = \pi$ so that $\psi(0,t) = 0 = \psi(\pi,t)$, for all values of t. The simplest way to ensure this is to set $\alpha = 0$ and $\sin(\pi\sqrt{\lambda}) = 0$, or $\sqrt{\lambda} = n$, an integer.

We will consider two important special cases.

4.6.2. The plucked string. This is the kind of string found in a guitar or harpsichord.

In this case, the string is initially not in motion, so that

$$\frac{\partial \psi}{\partial t} = 0 \tag{4.6.2}$$

when $t = 0$. Since a basic solution is

$$\psi_k(x,t) = a_k \sin(kx)\,(b_k \sin(kt) + c_k \cos(kt))$$

[8]The reader is invited to verify that this actually satisfies equation 4.6.1 on the previous page.

the easiest way to ensure equation 4.6.2 on the preceding page is to set $b_k = 0$ for all k. Our basic solutions reduce to

$$\psi_k(x,t) = a_k \sin(kx)\cos(kt)$$

and

$$\psi_k(x,0) = a_k \sin(kt)$$

If the initial configuration of the string (the "plucking" function) is $f(x)$ for $0 \leq x \leq \pi$, we can define an odd function from $-\pi$ to π:

$$f_1(x) = \begin{cases} f(x) & \text{if } x \geq 0 \\ -f(-x) & \text{otherwise} \end{cases}$$

and we can expand this in a Fourier series. Since $f_1(x)$ is *odd*, the cosine terms will all *vanish:*

$$\begin{aligned} b_k &= \frac{1}{\pi}\int_{-\pi}^{\pi} f_1(x)\cos(kx)dx \\ &= \frac{1}{\pi}\int_{-\pi}^{0} f_1(x)\cos(kx)dx + \frac{1}{\pi}\int_{0}^{\pi} f_1(x)\cos(kx)dx \\ &= -\frac{1}{\pi}\int_{0}^{\pi} f_1(x)\cos(kx)dx + \frac{1}{\pi}\int_{0}^{\pi} f_1(x)\cos(kx)dx \\ &= 0 \end{aligned}$$

The sine-terms tend to "double up"

$$\begin{aligned} a_k &= \frac{1}{\pi}\int_{-\pi}^{\pi} f_1(x)\sin(kx)dx \\ &= \frac{1}{\pi}\int_{-\pi}^{0} f_1(x)\sin(kx)dx + \frac{1}{\pi}\int_{0}^{\pi} f_1(x)\sin(kx)dx \\ &= \frac{1}{\pi}\int_{0}^{\pi} f_1(x)\sin(kx)dx + \frac{1}{\pi}\int_{0}^{\pi} f_1(x)\sin(kx)dx \\ &= \frac{2}{\pi}\int_{0}^{\pi} f_1(x)\sin(kx)dx \end{aligned}$$

So, we expand $f(x)$ in a Fourier series of sines, and the solution to the wave equation is

$$\psi(x,t) = \sum_{k=1}^{\infty} a_k \sin(kx)\cos(kt)$$

If we type

```
sin(n*x)*cos(n*t)
```

and issue the **trigreduce**-command (one of several commands for simplifying trigonometric expressions) we get

$$\frac{\sin(nx+nt)}{2} + \frac{\sin(nx-nt)}{2}$$

so

$$\psi(x,t) = \frac{1}{2}\left(\sum_{k=1}^{\infty} a_k \sin(kx+kt) + \sum_{k=1}^{\infty} a_k \sin(kx-kt)\right)$$

Now we ask ourselves "What *is* $\sum_{k=1}^{\infty} a_k \sin(kx)$?"

Well

$$\sum_{k=1}^{\infty} a_k \sin(kx) = \begin{cases} f(x) & \text{if } 0 \le x \le \pi \\ -f(-x) & \text{if } -\pi \le x \le 0 \\ \text{Periodic with period } 2\pi \end{cases}$$

So we get a *closed form* solution to the plucked wave equation:

Given a "plucking function[9]," $f(x)$, for $0 \le x \le \pi$ define

$$\bar{f}(x) = \begin{cases} f(x) & \text{if } 0 \le x \le \pi \\ -f(-x) & \text{if } -\pi \le x \le 0 \\ \text{Periodic with period } 2\pi \end{cases}$$

Then

$$\psi(x,t) = \frac{1}{2}\left(\bar{f}(x+t) + \bar{f}(x-t)\right)$$

Let's compute!

We start with a "realistic" plucking function

$$f(x) = \begin{cases} x/2 & \text{for } 0 \le x < 1 \\ -\frac{x-\pi}{2\pi-2} & \text{for } 1 \le x \le \pi \end{cases}$$

This programs as

```
f(x):=block([],
if (x<1) then return (x/2),
-(x-%pi)/(2*%pi-2)
)
```

and we can plot it via

```
plot2d('f(x),[x,0,%pi],[y,-1/2,1/2])
```

and we get figure 4.6.1 on the facing page.

Now we define $\bar{f}(x)$:

```
f_bar(x):=block([],
if (x>%pi) then return (f_bar(x-2*%pi)),
if (x<=-%pi) then return (f_bar(x+2*%pi)),
if(x>=0) then return (f(x)),
-f(-x)
);
```

and, to check this, we plot it

[9] I.e., shape of the string at time 0.

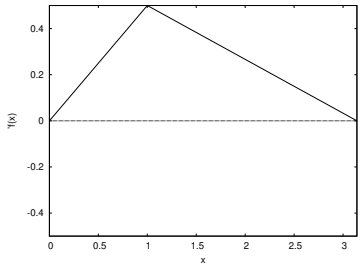


FIGURE 4.6.1. "Realistic" plucking function

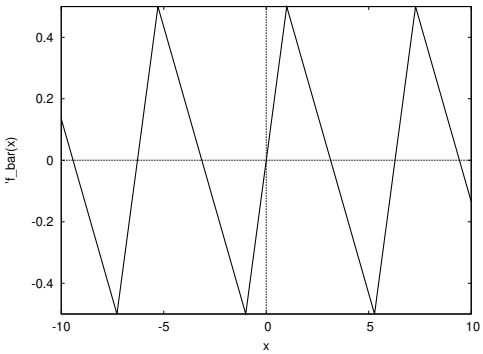


FIGURE 4.6.2. The extended plucking function

```
plot2d('f_bar(x),[x,-10,10],[y,-1/2,1/2])
```

and get figure 4.6.2.

```
psi_p(x,t):=(f_bar(x+t)+f_bar(x-t))/2
```

Which we can plot via

```
with_slider_draw(
        t, /*variable to attach to the slider */
        makelist(j,j,0,100), /*list of integers */
        explicit('psi_p(x,.1*t),x,0,%pi), /*plot */
        yrange= [-1/2,1/2] /*optional graphic command */
        ); /* end of with_slider_draw-command */
```

4.6.3. The "hammered" string. These occur in pianos or hammered dulcimers. We return to our basic solution

$$\psi_k(x,t) = a_k \sin(kx)\left(b_k \sin(kt) + c_k \cos(kt)\right)$$

Since $\psi(x,0) = 0$, we set the c_k to 0, so our basic solution looks like

$$\psi_k(x,t) = a_k \sin(kx)\sin(kt)$$

and

$$\frac{\partial \psi}{\partial t} = a_k \cdot k \sin(kx)\cos(kt)$$

If we set $t = 0$, this becomes

$$\left.\frac{\partial \psi}{\partial t}\right|_{t=0} = a_k \cdot k \sin(kx)$$

If $h(x)$ is our "hammering" function — the state of motion of the string at time 0 — to solve the wave equation we

(1) expand $h(x)$ in a Fourier sine-series as in the plucked case, with coefficients

$$a_k = \frac{2}{\pi}\int_0^\pi h(x)\sin(kx)dx$$

(2) the resulting series for $\psi(x,t)$ is

$$\psi(x,t) = \sum_{k=1}^{\infty} \frac{a_k}{k}\sin(kx)\sin(kt)$$

For instance, we can define our hammering function by

$$h(x) = \begin{cases} 0 & \text{if } 0 \le x < 1/2 \\ 1 & \text{if } 1/2 \le x \le 3/4 \\ 0 & \text{if } 3/4 < x \le \pi \end{cases}$$

```
h(x):=block([],
if (x<1/2) then return (0),
if (x<=3/4) then return (1),
0
);
```

and, to check this, we plot it. We define

```
a(k):=(2/%pi)*integrate(sin(k*x),x,1/2,3/4)
```

Now we define

```
psi_h(x,t):=sum(a(j)*sin(j*x)*sin(j*t)/j,j,1,100)
```

and plot it with

```
with_slider_draw(
        t, /*variable to attach to the slider */
        makelist(j,j,0,100), /*list of integers */
        explicit(psi_h(x,.1*t),x,0,%pi), /*plot */
        yrange= [-1/2,1/2] /*optional
                    graphic command */
        ); /* end of with_slider_draw-command */
```

You will benefit from *memoizing* these computations (as with the heat equation).

EXERCISES.

1. Find a closed-form solution of the wave-equation for a hammered string.

4.6.4. The two-dimensional case. In this case, the wave equation looks like

$$\frac{\partial^2\psi}{\partial x^2} + \frac{\partial^2\psi}{\partial y^2} = \frac{1}{c^2}\frac{\partial^2\psi}{\partial t^2}$$

where $\psi(x, y, t)$ is the displacement of a rectangular drum-head. We try the trick we used before:

$$\psi(x, y, t) = u(x)v(y)w(t)$$

and get the equation

$$\frac{d^2u}{dx^2}v(y)w(t) + u(x)\frac{d^2v}{dy^2}w(t) = \frac{1}{c^2}u(x)v(y)\frac{d^2w}{dt^2}$$

and divide by $u(x)v(y)w(t)$ to get

$$\frac{1}{u(x)}\frac{d^2u}{dx^2} + \frac{1}{v(y)}\frac{d^2v}{dy^2} = \frac{1}{c^2}\frac{1}{w(t)}\frac{d^2w}{dt^2}$$

As before, functions of x and y can only equal a function of t if they are equal to the same constant:

$$\frac{1}{c^2}\frac{1}{w(t)}\frac{d^2w}{dt^2} = -\lambda$$

$$\frac{1}{u(x)}\frac{d^2u}{dx^2} + \frac{1}{v(y)}\frac{d^2v}{dy^2} = -\lambda$$

The second of these equations implies that

$$\frac{1}{u(x)}\frac{d^2u}{dx^2} = -\lambda - \frac{1}{v(y)}\frac{d^2v}{dy^2}$$

so, again, we have a function of x equal to a function of y: they must both equal the same constant! We have equations

$$\frac{1}{c^2}\frac{1}{w(t)}\frac{d^2w}{dt^2} = -(a+b)$$

$$\frac{1}{u(x)}\frac{d^2u}{dx^2} = -a$$

$$\frac{1}{v(y)}\frac{d^2v}{dy^2} = -b$$

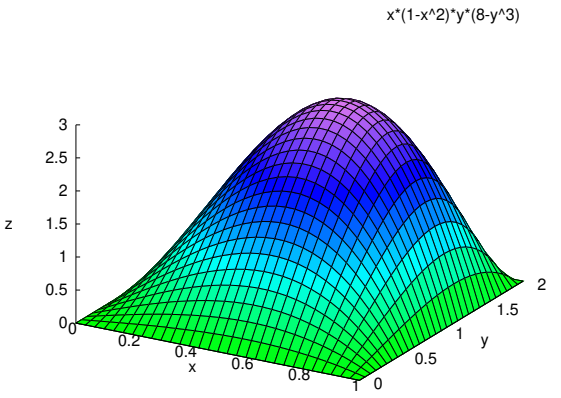


FIGURE 4.6.3. Initial position of a two dimensional membrane

Note that the solutions to the differential equations are

$$u(x) = r\sin(\sqrt{a}x) + s\cos(\sqrt{a}x) \tag{4.6.3}$$
$$v(y) = r'\sin(\sqrt{b}y) + s'\cos(\sqrt{b}y)$$
$$w(t) = \bar{r}\sin(c\sqrt{a+b}t) + \bar{s}\cos(c\sqrt{a+b}t)$$

Suppose our vibrating surface is L units long, W units wide, and is rigidly fixed on its boundaries. We'll also suppose the membrane is at rest initially and has a shape given by $f(x,y)$, so

$$f(0,y) = f(L,y) = 0$$
$$f(x,0) = f(x,W) = 0$$

We can expand $f(x,y)$ in a two-dimensional Fourier series

$$\sum_{n,m=1}^{\infty} c_{n,m}\sin\left(\frac{n\pi x}{L}\right)\sin\left(\frac{m\pi y}{W}\right)$$

where

$$c_{n,m} = \frac{4}{LW}\int_0^L\int_0^W f(x,y)\sin\left(\frac{n\pi x}{L}\right)\sin\left(\frac{m\pi y}{W}\right)dxdy$$

So equations 4.6.3 imply that an elementary solution looks like

$$\sin\left(\frac{n\pi x}{L}\right)\sin\left(\frac{m\pi y}{W}\right)\cos\left(c\pi t\sqrt{\frac{n^2}{L^2}+\frac{m^2}{W^2}}\right) \tag{4.6.4}$$

We will assume $W = 1$, $L = 2$, $c = 1$, and define $f(x,y) = xy(1-x^2)(8-y^3)$, which plots as figure 4.6.3

```
declare(n,integer);
declare(m,integer);
coef[n,m]:=2*integrate(integrate(x*y*(1-x^2)*
```

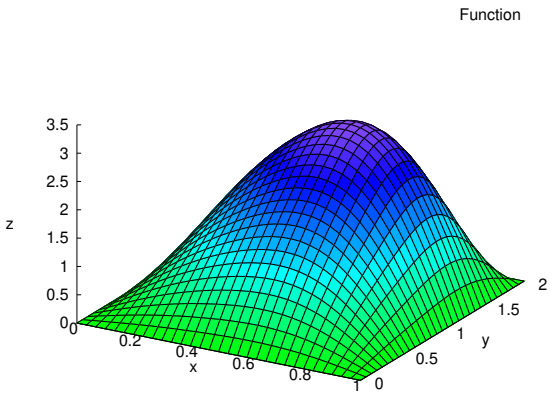


FIGURE 4.6.4. First three terms of a two-dimensional Fourier series

```
    (8-y^3)*sin(%pi*n*x)*sin(%pi*m*y/2),
     x,0,1),y,0,2);
```

which gives

$$\text{coef}(n,m) = -\frac{12\left(-\frac{\left(384\pi^2m^2-768\right)(-1)^m}{\pi^5m^5}-\frac{768}{\pi^5m^5}\right)(-1)^n}{\pi^3n^3}$$

Now we write a function to add up terms of the two-dimensional Fourier series

```
first_n(n,x,y):=block (
                       [],
                       sum(sum(
                         coef[i,j]*sin(%pi*i*x)*sin(%pi*j*y/2)
                         ,i,1,n),j,1,n))
```

If we plot the first three terms, via

```
plot3d(first_n(3,x,y),[x,0,1],[y,0,2]);
```

we get figure 4.6.4, which is not bad.

Now we apply equation 4.6.4 on the preceding page to get a solution of the wave equation

$$\sum_{n=1}^{\infty}\sum_{m=1}^{\infty}\text{coef}(n,m)\sin(n\pi x)\sin\left(\frac{m\pi y}{2}\right)\cos\left(\pi t\sqrt{n^2+\frac{m^2}{4}}\right)$$

which we code up via

```
vibrate_n(n,x,y,t):=block (
                       [],
                       sum(sum(
```

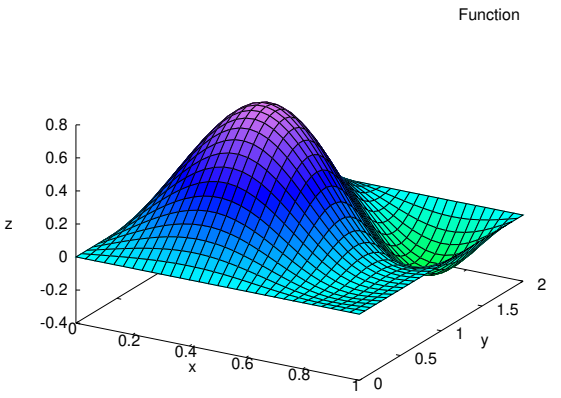


FIGURE 4.6.5. After .4 time units

```
                coef[i,j]*sin(%pi*i*x)
                  *sin(%pi*j*y/2)
                  *cos(%pi*t*sqrt(i^2+j^2/4))
                ,i,1,n),j,1,n))
```

At time $t = .4$, our plot

```
plot3d(vibrate_n(3,x,y,.4),[x,0,1],[y,0,2]);
```

looks like figure 4.6.5. Notice that the vibration is asymmetric (you might have to rotate it a bit to see this).

EXERCISES.

2. Show that

$$\int_{-\pi}^{\pi}\int_{\pi}^{\pi} \sin(nx)\sin(my)\cdot\sin(\bar{n}x)\sin(\bar{m}y)\,dxdy = 0$$

if $n \neq \bar{n}$ or $m \neq \bar{m}$.

CHAPTER 5

Integral transforms

> "We are rag dolls made out of many ages and skins, changelings who have slept in wood nests or hissed in the uncouth guise of waddling amphibians. We have played such roles for infinitely longer ages than we have been men. Our identity is a dream. We are process, not reality, for reality is an illusion of the daylight — the light of our particular day."
>
> — Loren Eiseley.

5.1. The Fourier Transform

In this chapter we will begin by approaching Fourier series from another direction, using the simple fact that

$$\int_{-\pi}^{\pi} e^{inx} \cdot e^{-imx} dx = \begin{cases} 2\pi & \text{if } n = m \\ 0 & \text{otherwise} \end{cases}$$

If $f(x)$ is a function, we can compute coefficients via

$$a_k = \frac{1}{2\pi} \int_{-\pi}^{\pi} f(x) e^{-ikx} dx$$

and get a series

$$f(x) = \sum_{k>-\infty}^{\infty} a_k e^{ikx}$$

If we expand our old friend, $f(x)$, defined in equation 3.2.1 on page 35, we get

$$a_k = \frac{1}{2\pi} \left(\int_{-1}^{0} e^{-ikx} dx + \int_{0}^{1} x^2 e^{-ikx} dx \right)$$

or

```
a[k]:= (integrate(%e^(-%i*k*x),x,-1,0)
+integrate(x^2*%e^(-%i*k*x),x,0,1))/(2*%pi)
```

```
first_n(k,x):= sum(a[j]*%e^(%i*j*x),j,-k,k);
```

Now, suppose we want to expand our horizons from $[-\pi, \pi]$ to $[-L, L]$. We rewrite the equations above to

$$a_k = \frac{1}{2L} \int_{-L}^{L} f(x) e^{-i2\pi kx/L} dx$$

and the Fourier series becomes

$$f(x) = \sum_{k>-\infty}^{\infty} a_k e^{2\pi i k x/L}$$

and we will rewrite this slightly

$$a_{k/L} = \int_{-L}^{L} f(x) e^{-2\pi i x k/L} dx$$

and

$$f(x) = \frac{1}{2L} \sum_{k>-\infty}^{\infty} a_{k/L} e^{2\pi i k x/L}$$

Now, we let $L \to \infty$ and set $s = k/L$ and get

$$a_s = a(s) = \int_{-\infty}^{\infty} f(x) e^{-2\pi i x s} dx$$

$$f(x) = \int_{-\infty}^{\infty} a(s) e^{2\pi i x s} ds$$

and $a(s)$ is defined to be the *Fourier Transform* of $f(x)$ — if these integrals are well-defined!

Let's compute the Fourier transform of our old friend, $f(x)$, defined in 3.2.5 on page 37 and plotted in figure 3.2.8 on page 39.

$$a(s) = \int_{-1}^{0} e^{-2\pi i x s} dx + \int_{0}^{1} x^2 e^{-2\pi i x s} dx$$

or

```
a(s):= integrate(%e^(-2*%pi*%i*s*x),x,-1,0)
+integrate(x^2*%e^(-2*%pi*%i*s*x),x,0,1)
```

Now we plot the real and imaginary parts of $a(s)$

```
plot2d([realpart(a(s)),imagpart(a(s))],[s,-4,4]);
```

to get figure 5.1.1 on the facing page. The plot-command complains about division by zero, although it manages to generate the plot.

To see why, do *indefinite* integrals:

```
integrate(%e^(-2*%pi*%i*s*x),x)
```

gives

$$\frac{i e^{-2\pi i x s}}{2\pi s}$$

```
integrate(x^2*%e^(-2*%pi*%i*s*x),x)
```

gives

$$\frac{\left(2\pi^2 i s^2 x^2 + 2\pi s x - i\right) e^{-2\pi i x s}}{4\pi^3 s^3}$$

with s in the *denominator* in both cases! On the other hand $a(0)$ is a perfectly well-defined $4/3$, as the plot shows.

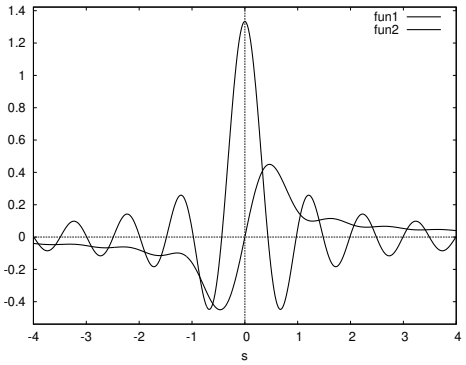


FIGURE 5.1.1. Fourier transform of $f(x)$

We have effectively decomposed $f(x)$ into a continuous infinity of periodic functions. The Fourier transform recognizes *periodic* behavior of a function and gives its intensity at different frequencies.

5.2. The discrete Fourier transform

Although Maxima doesn't have built-in commands to implement Fourier transforms analytically, it does implement fast numeric algorithms for *discrete* Fourier transforms.

If $\{x_j\}$ $i = 1 \ldots n$ is a sequence of numbers, its *discrete* Fourier transform is defined via

$$y(k) = \frac{1}{n} \sum_{j=0}^{n-1} x(j) e^{2\pi i \cdot jk/n} \tag{5.2.1}$$

and its inverse is defined by

$$x(j) = \sum_{j=0}^{n-1} y(k) e^{-2\pi i \cdot jk/n} \tag{5.2.2}$$

These are the definitions used by Maxima; there are many others. Many (most?) authors swap these definitions — they define the Fourier transform via equation 5.2.2 and the inverse via 5.2.1.

Although these are straightforward enough, they become cumbersome when the sequences are long (as they are in practice). An algorithm was discovered when $n = 2^m$ for some integer $m > 0$ — attributed to Cooley and Tukey (but really discovered centuries earlier by Gauss!), called the *Fast* Fourier Transform.

James William Cooley (1926 – 2016) was an American mathematician. He was a programmer on John von Neumann's computer at the Institute for Advanced Study, Princeton, NJ, from 1953 to 1956, where he notably programmed the Blackman–Tukey transformation.
He worked on quantum mechanical computations at the Courant Institute, New York University, from 1956 to 1962, when he joined the Research Staff at the IBM Watson Research Center, Yorktown Heights, NY. Upon retirement from IBM in 1991, he joined the Department of Electrical Engineering, University of Rhode Island, Kingston, where he served on the faculty of the computer engineering program.

John Wilder Tukey (1915 – 2000)) was an American mathematician and statistician, best known for the development of the fast Fourier Transform (FFT) algorithm and box plot. The Tukey range test, the Tukey lambda distribution, the Tukey test of additivity, and the Teichmüller–Tukey lemma all bear his name. He is also credited with coining the term 'bit' and the first published use of the word 'software'.

The Fast Fourier Transform commands occur in a library loaded via

```
load("fft")
```

The most basic commands in question are **fft** and **inverse_ff**t. The following code shows that they really are inverses.

```
load ("fft");
fpprintprec : 4;  /* number of digits to print */
L : [1, 1 + %i, 1 - %i, -1, -1, 1 - %i, 1 + %i, 1];
L1 : fft (L);
[0.5, 0.5, 0.25 %i - 0.25, (- 0.3536 %i) - 0.3536,
0.0, 0.5,(- 0.25 %i) - 0.25, 0.3536 %i + 0.3536]
L2 : inverse_fft (L1);
[1.0, 1.0 %i + 1.0, 1.0 - 1.0 %i, - 1.0, - 1.0,
     1.0 - 1.0 %i,1.0 %i + 1.0, 1.0]
lmax (abs (L2 - L));
                          0.0
```

The most straightforward application of the discrete Fourier transform is detecting periodic behavior in sequences of numbers.

To introduce a more interesting (and widely-used) application, we need:

DEFINITION 5.2.1. Let $A = \{a_i\}$, $i = 0, \ldots, n-1$ and $B = \{b_j\}$, $j = 0, \ldots m-1$ be sequences of numbers. The *convolution* $A \bigstar B = \{c_k\}$, $k = 0, \ldots, n+m-1$ of these sequences is defined by

$$c_t = \sum_{i=0}^{t} a_i b_{t-i}$$

where $a_i = 0$ if $i \notin 0, \dots n-1$ and $b_j = 0$ if $j \notin 0, \dots, m$.

REMARK. This also well-defined in the continuous case

$$f \bigstar g(s) = \int_{-\infty}^{\infty} f(x)g(s-x)dx$$

We have the well known

THEOREM 5.2.2 (Convolution Theorem). *If A is a sequence of numbers, let $\mathcal{F}(A)$ denote the discrete Fourier transform of A. If A and B are finite sequences of numbers of length n, then $\mathcal{F}(A \bigstar B)_i = n \cdot \mathcal{F}(A)_i \cdot \mathcal{F}(B)_i$ for all i. In particular*

$$A \bigstar B = n \cdot \mathcal{F}^{-1}\left(\mathcal{F}(A) \cdot \mathcal{F}(B)\right) \tag{5.2.3}$$

where '$\cdot$' represents element-by-element multiplication.

REMARK. See [50] for a proof. So Fourier transforms convert convolutions into simple multiplications. If we follow the widespread convention mentioned above, the factor of n is unnecessary. In other words, using *Maxima's* conventions

$$A \bigstar B = \mathcal{F}\left(\mathcal{F}^{-1}(A) \cdot \mathcal{F}^{-1}(B)\right) \tag{5.2.4}$$

A similar result is true in the *continuous* case (without the factor of $n!$).

The fast Fourier transformation and its inverse are *so fast*, equation 5.2.3 is faster than direct computation — at least if the sequences are sufficiently large.

The reader might ask

Why do we care about convolutions?

They have applications to

(1) Analyzing signals.
(2) Multiplication of polynomials (the coefficients of the product are a convolution of the coefficients of the factors), if the polynomials are large (hundreds of terms).
(3) Multiplication of numbers with hundreds of digits — we can regard them as polynomials evaluated at 10 with coefficients that are integers $0 \dots 9$.

EXAMPLE 5.2.3. Suppose we want to form the convolution of the sequences

$$\{1, 4, 2, 5\} \text{ and } \{3, 1, 3, 2\}$$

representing coefficients of cubic polynomials $1 + 4x + 2x^2 + 5x^3$ and $3 + x + 3x^2 + 2x^3$. Their convolution will be of length 7 so we extend these to length $8 = 2^3$ by zeroes on the right and execute the code:

```
load ("fft");
fpprintprec : 4;  /* number of digits to print */
A : [1,4,2,5,0,0,0,0];
B : [3,1,3,2,0,0,0,0];
fa : fft(A);
fb : fft(B);
fc:fa*fb;/* element by element multiplication */
C : realpart(8*inverse_fft(fc));
```

You will notice that the output of the **inverse_fft** command has imaginary parts that are very small ($\sim 10^{-17}$). All the intermediate computations used complex numbers that don't quite cancel in the end due to round-off errors. The simplest way to deal with these is to take the **realpart**.

There are other Maxima commands to take transforms of *real-valued* sequences (with a faster algorithm) or to use **bfloat**'s in the computations (so the round-off errors are much smaller).

EXERCISES.

1. Show that convolution is commutative and associative. In other words, if A, B, and C are sequences of numbers, show that

$$A\bigstar B = B\bigstar A$$
$$A\bigstar(B\bigstar C) = (A\bigstar B)\bigstar C$$

2. In example 5.2.3 on the preceding page, why did we extend the sequences until they had length 8?

3. Run example 5.2.3 on the previous page using equation 5.2.4 on the preceding page.

4. Compute the cube of the polynomial $2 - 4x + x^2 - x^3$ using convolution and equation 5.2.4 on the previous page.

5.3. The Laplace Transform

The Fourier transform inspired Laplace to develop a variation of it that is useful in solving linear differential equations. The *Laplace Transform* of a function, $f(x)$, is defined by

$$\mathcal{L}(f)(s) = \int_0^\infty e^{-sx} f(x)dx \tag{5.3.1}$$

(if the integral is well-defined!) with an inverse

$$\mathcal{L}^{-1}(F)(x) = \lim_{T\to\infty} \frac{1}{2\pi i} \int_{\gamma-iT}^{\gamma+iT} e^{xs} F(s) ds$$

where γ is a real number set to something that makes the integral converge (if possible!).

Pierre-Simon, marquis de Laplace (1749 – 1827) was a French scholar whose work was important to the development of engineering, mathematics, statistics, physics, astronomy, and philosophy. He summarized and extended the work of his predecessors in his five-volume *Mécanique céleste* (Celestial Mechanics) (1799–1825). This work translated the geometric study of classical mechanics to one based on calculus, opening up a broader range of problems. In statistics, the Bayesian interpretation of probability was developed mainly by Laplace.
Laplace formulated Laplace's equation, and pioneered the Laplace transform which appears in many branches of mathematical physics, a field that he took a leading role in forming. The Laplacian differential operator, widely used in mathematics, is also named after him. He restated and developed the nebular hypothesis of the origin of the Solar System and was one of the first scientists to suggest an idea similar to that of a black hole.
Laplace is regarded as one of the greatest scientists of all time. Sometimes referred to as the French Newton or Newton of France, he has been described as possessing a phenomenal natural mathematical faculty superior to that of almost all of his contemporaries. He was Napoleon's examiner when Napoleon attended the École Militaire in Paris in 1784. Laplace became a count of the Empire in 1806 and was named a marquis in 1817, after the Bourbon Restoration.

Both the Laplace transform and its inverse are built in to Maxima (it's not necessary to load any libraries):

```
laplace(x^3,x,s); /* The Laplace transform. */
```

gives

$$\frac{6}{s^4}$$

```
ilt(6/s^4,s,x); /* The Inverse Laplace transform. */
```

recovers x^3.

As we said earlier, the Laplace Transform is useful in solving linear differential equations. To see why, note that

$$\mathcal{L}(f'(x) = s \cdot \mathcal{L}(f) - f(0)$$

which you can see by integrating equation 5.3.1 on the facing page by parts or typing

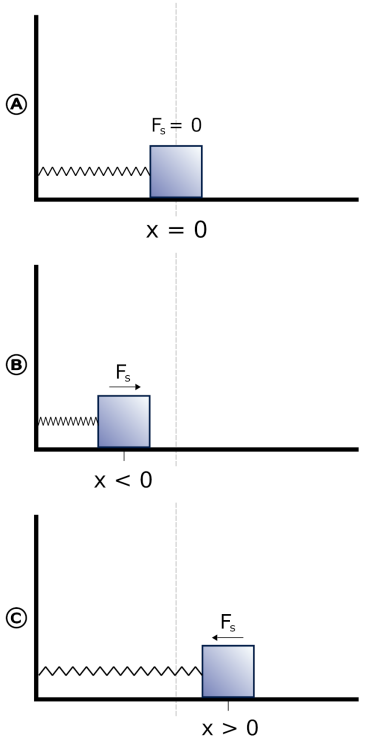


FIGURE 5.3.1. Harmonic oscillator

```
laplace('diff(f(x),x),x,s);
```

to get

```
s*laplace(f(x),x,s)-f(0)
```

Suppose we want to solve the differential equation of a harmonic oscillator as in figure 5.3.1.

We assume the mass bobs back and forth without friction and get a differential equation like

$$\frac{d^2f}{dx^2} + 3f = 0 \tag{5.3.2}$$

where f is the displacement of the mass and x is time. We apply the Laplace Transform to get

$$-f'(0) + s^2\mathcal{L}(f) + 3\mathcal{L}(f) - sf(0) = 0$$

and solve for $\mathcal{L}(f)$ to get

$$\mathcal{L}(f) = \frac{sf(0) + f'(0)}{s^2 + 3}$$

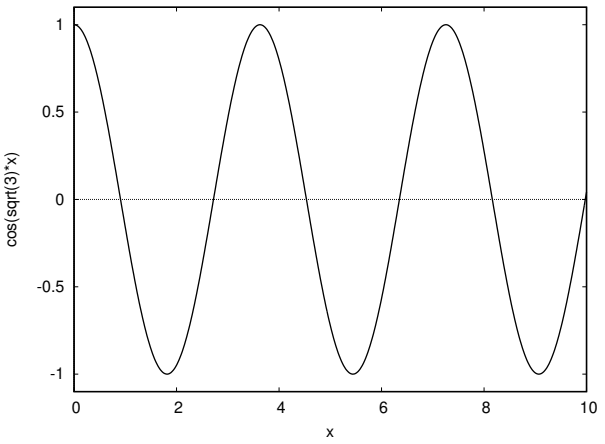


FIGURE 5.3.2. Simple harmonic motion

The command

```
ilt((s*f(0)+fp(0))/(s^2+3),s,x);
```

shows that

$$f(x) = \frac{f'(0)\sin\left(\sqrt{3}x\right)}{\sqrt{3}} + f(0)\cos\left(\sqrt{3}x\right)$$

Figure 5.3.2 shows a plot of the motion when $f'(0) = 0$ and $f(0) = 1$.

Now consider the case of a *forcing function:*

$$\frac{d^2f}{dx^2} + 3f = \sin(2x)$$

Here, the function $\sin(2x)$ is *driving* the oscillator. We apply the Laplace Transform to both sides of the equation to get

$$-f'(0) + s^2\mathcal{L}(f) + 3\mathcal{L}(f) - sf(0) = \frac{2}{s^2+4} \tag{5.3.3}$$

We can solve this for $\mathcal{L}(f)$ (using **solve**!) to get

$$\mathcal{L}(f) = \frac{f(0)s^3 + f'(0)s + 4sf(0) + 4f'(0) + 2}{s^4 + 7s^2 + 12}$$

Now we take the inverse Laplace Transform to get

$$f(x) = \frac{(f'(0)+2)\sin(\sqrt{3}x)}{\sqrt{3}} + f(0)\cos(\sqrt{3}x) - \sin(2x)$$

One nice aspect of this solution is that it explicitly shows the effect of initial conditions.

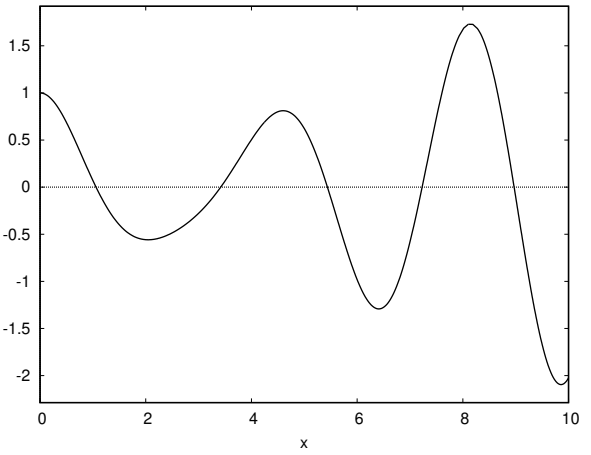


FIGURE 5.3.3. Forced harmonic motion

If $f'(0) = 0$ and $f(0) = 1$, we get the motion in figure 5.3.3.

One shortcoming of the Maxima's Laplace transform package is its failure to deal with *Heavyside functions*. These are functions of the form $H(x-\alpha)$ where $H(x)$ is defined by

$$H(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$$

These are interesting because:

- ▷ every piecewise-defined function can be expressed as a linear combination of Heavyside functions and ordinary functions
- ▷ Laplace transforms of such functions can be easily calculated

LEMMA 5.3.1. *Let $f(x)$ be a function that has a Laplace transform and let $\alpha \in \mathbb{R}$ be such that $\alpha \geq 0$. Then*

$$\mathcal{L}\left(H(x-\alpha)f(x)\right) = e^{-\alpha s}\mathcal{L}(f(x+\alpha)) \tag{5.3.4}$$

It follows that

$$\mathcal{L}^{-1}\left(e^{-\alpha s}g(s)\right) = H(x-\alpha)\cdot\mathcal{L}^{-1}(g)(x-\alpha) \tag{5.3.5}$$

PROOF. From the definition

$$\begin{aligned}\mathcal{L}(H(x-\alpha)f(x)) &= \int_0^\infty e^{-xs}H(x-\alpha)f(x)dx \\ &= \int_\alpha^\infty e^{-xs}f(x)dx\end{aligned}$$

Now let $u = x - \alpha$, so $x = u + \alpha$ and substitute

$$\begin{aligned}\int_\alpha^\infty e^{-xs} f(x)dx &= \int_0^\infty e^{-(u+\alpha)s} f(u+\alpha)dx \\ &= e^{-\alpha s}\int_0^\infty e^{-us} f(u+\alpha)dx \\ & e^{-\alpha s}\mathcal{L}(f(u+\alpha))\end{aligned}$$

□

EXAMPLE 5.3.2. Suppose the driving force of our harmonic oscillator is given by

$$d(x) = \begin{cases} 1 & \text{if } 1/2 \le x \le 1 \\ 0 & \text{otherwise} \end{cases}$$

Then $d(x) = H(x - 1/2) - H(x - 1)$ and our version of equation 5.3.3 on page 89 is

$$-f'(0) + s^2\mathcal{L}(f) + 3\mathcal{L}(f) - sf(0) = e^{-s/2} - e^{-s}$$

giving

$$\begin{aligned}\mathcal{L}(f) &= \frac{sf(0) + f'(0) + e^{-s/2} - e^{-s}}{s^2+3} \\ &= \frac{s}{s^2+3}f(0) + \frac{1}{s^2+3}f'(0) + \frac{e^{-s/2}}{s^2+3} - \frac{e^{-s}}{s^2+3} \qquad (5.3.6)\end{aligned}$$

Now we isolate each term of the form

$$\mathbf{e^{-as}}r(s)$$

where a is a real number and compute

$$H(x-a)\mathcal{L}^{-1}(r)(x-a)$$

In the case of equation 5.3.6, we take the inverse Laplace transform using the **ilt**-command for the first two terms, and the **ilt**-command coupled with equation 5.3.5 on the facing page to handle the remaining two terms:

$$\begin{aligned}f(x) = \cos(\sqrt{3}\cdot x)f(0) + \frac{\sin(\sqrt{3}\cdot x)}{\sqrt{3}}f'(0) \\ + H(x-1/2)\cdot\frac{\sin(\sqrt{3}\cdot(x-1/2))}{\sqrt{3}} \\ - H(x-1)\cdot\frac{\sin(\sqrt{3}\cdot(x-1))}{\sqrt{3}}\end{aligned}$$

If we assume that the string was initially at rest, we get

$$f(x) = H(x-1/2)\cdot\frac{\sin(\sqrt{3}\cdot(x-1/2))}{\sqrt{3}} - H(x-1)\cdot\frac{\sin(\sqrt{3}\cdot(x-1))}{\sqrt{3}}$$

which is plotted in figure 5.3.4 on the next page. Here, we have coded

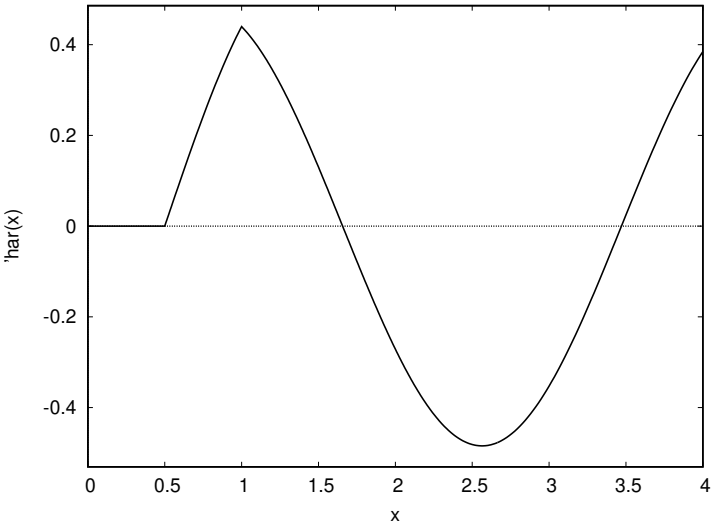


FIGURE 5.3.4. Discontinuous driving force

```
H(x):=block(
            [],
            if (x>=0) then return (1),
            0)
```

EXERCISES.

1. Is the function depicted in figure 5.3.3 on page 90 periodic?
2. Represent the piecewise function

$$f(x) = \begin{cases} 0 & \text{if } x < 2 \\ x^2 & \text{if } x < 3 \\ -x & \text{if } x < 5 \\ 0 & \text{if } x \geq 5 \end{cases}$$

as a linear combination of Heavyside functions (the coefficients may be arbitrary ordinary functions).

3. Use Laplace transforms to solve the differential equation

$$\frac{d^2y}{dx^2} + \frac{dy}{dx} + x = f(x)$$

where $f(x)$ is defined in exercise 2.

CHAPTER 6

Orthogonal polynomials

> "It is a matter for considerable regret that Fermat, who cultivated the theory of numbers with so much success, did not leave us with the proofs of the theorems he discovered. In truth, Messrs Euler and Lagrange, who have not disdained this kind of research, have proved most of these theorems, and have even substituted extensive theories for the isolated propositions of Fermat. But there are several proofs which have resisted their efforts."
> — Adrien-Marie Legendre.

6.1. Introduction

As we saw in section 4.3 on page 58, it is possible to expand functions (even discontinuous ones!) in a series of sines and cosines. How was this possible? After some thought, it becomes clear that the key was equations 4.3.3 on page 59, 4.3.5 on page 60, and 4.3.7 on page 60 — the so-called *orthogonality relations.*

Is it possible to find similar relations between other sets of functions?

We will construct a set of orthogonal *polynomials* $\{P_i(x)\}$ with

$$\int_{-1}^{1} P_i(x)P_j(x)dx = 0 \tag{6.1.1}$$

if $i \neq j$.

The simplest candidate for P_0 is 1, so we will pick it. The linear polynomial is of the form $a_0 + a_1 x$

$$\int_{-1}^{1} 1 \cdot (a_0 + a_1 x)dx = 2a_0$$

For the polynomials to be orthogonal, we set $a_0 = 0$ and set $a_1 = 1$, so $p_1(x) = x$. We also have

$$\int_{-1}^{1} P_1(x)^2 dx = \frac{2}{3}$$

The general form of $P_2(x)$ is $a_0 + a_1 x + a_2 x^2$. We get

```
integrate(1*(a_0+a_1*x+a_2*x^2),x,-1,1); ratsimp(%);
```

which gives

$$\frac{2a_2 + 6a_0}{3}$$

```
integrate(x*(a_0+a_1*x+a_2*x^2),x,-1,1); ratsimp(%);
```

gives

$$\frac{2a_1}{3}$$

We uniquely determine coefficients by requiring $P_i(1) = 1$:

```
solve([2*a_2+6*a_0=0,2*a_1=0,a_0+a_1+a_2=1],
[a_0,a_1,a_2])
```

and get

$$\left[\left[a_0 = -\frac{1}{2}, a_1 = 0, a_2 = \frac{3}{2}\right]\right]$$

so

$$P_2(x) = \frac{3}{2}x^2 - \frac{1}{2}$$

In this fashion, we can inductively construct a sequence of polynomials $\{P_i(x)\}$ satisfying equation 6.1.1 on the preceding page. Of course, we are not the first people to think of this. The $\{P_i(x)\}$ are called the *Legendre Polynomials* after the first person to study them.

Adrien-Marie Legendre (1752 – 1833) was a French mathematician who made numerous contributions to mathematics. Well-known and important concepts such as the Legendre polynomials and Legendre transformation are named after him.
In 1782, he first introduced his polynomials as coefficients in the expansion of the Newtonian Potential energy

(6.1.2)
$$\begin{aligned}\frac{1}{|\mathbf{x}-\mathbf{x}'|} &= \frac{1}{\sqrt{|\mathbf{x}|^2 + |\mathbf{x}'|^2 - 2|\mathbf{x}||\mathbf{x}'|\cos\theta}} \\ &= \sum_{\ell=0}^{\infty} \frac{|\mathbf{x}'|^\ell}{|\mathbf{x}|^{\ell+1}} P_\ell(\cos\theta)\end{aligned}$$

where θ is the angle between the vectors $\mathbf{x}$ and $\mathbf{x}'$, and $|\mathbf{x}'| < |\mathbf{x}|$ — see figure 6.1.1 on the next page.
Figure 6.1.2 on the facing page show plots of the first six Legendre polynomials.

Nowadays, they crop up when one converts the heat and wave equations to spherical coordinates (latitude, longitude, and radius). Since the Schrödinger Wave equation is related to the Heat equation, Legendre polynomials also are widely used in Quantum Mechanics.

Legendre and others have discovered some of their properties:

Besides orthogonality, we have

(6.1.3)
$$\int_{-1}^{1} P_n(x)^2 dx = \frac{2}{2n+1}$$

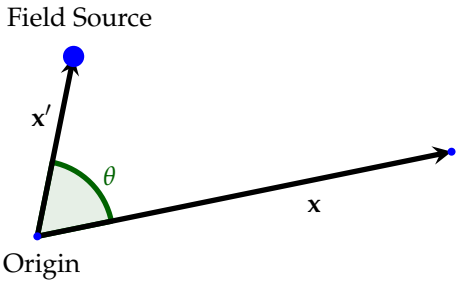


FIGURE 6.1.1. Model for Legendre polynomials

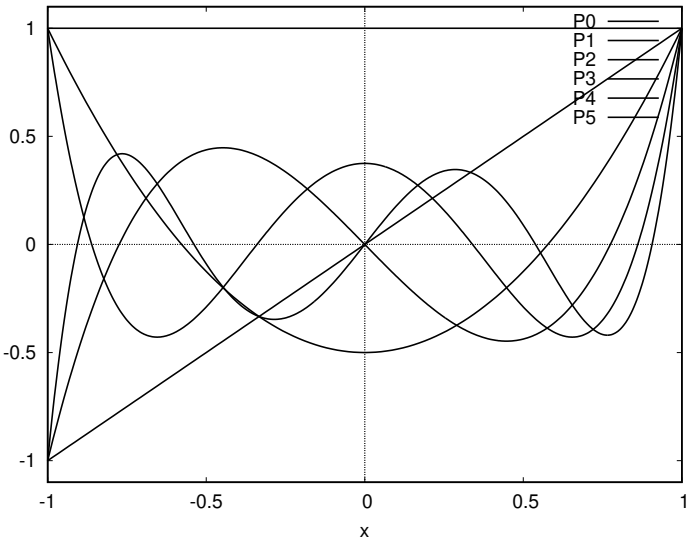


FIGURE 6.1.2. The first six Legendre polynomials

(this isn't obvious!).

Luckily, people have compiled a Maxima library of Legendre polynomials (and many other systems of orthogonal polynomials) accessed by

```
load("orthopoly")
```

The Legendre polynomials are given by legendre_p(n,x). If we type

```
legendre_p(5,x)
```

we get

$$-15(1-x)-\frac{63(1-x)^5}{8}+\frac{315(1-x)^4}{8}-70(1-x)^3+\frac{105(1-x)^2}{2}+1$$

and **expand**(%) gives a simplified form

$$\frac{63x^5}{8} - \frac{35x^3}{4} + \frac{15x}{8}$$

Recall our old friend, the discontinuous function $f(x)$, plotted in 3.2.8 on page 39:

```
f(x) := block ([], /* no local variables */
if(x<-1) then return (0),
if(x<0) then return (1),
if(x<=1) then return (x^2),
0);  /*default final value */
```

We'll expand this in a series of Legendre polynomials using the same methods as for Fourier series:

$$a_k = \frac{\int_{-1}^{1} \text{legendre\_p}(k,x) f(x) dx}{\int_{-1}^{1} \text{legendre\_p}(k,x)^2 dx}$$

Translated into Maxima, this is

```
a[n]:=((2*n+1)/2)*(integrate(legendre_p(n,x),x,-1,0)
+integrate(legendre_p(n,x)*x^2,x,0,1));
```

where

$$\frac{2n+1}{2} = \frac{1}{\int_{-1}^{1} \text{legendre\_p}(k,x)^2 dx}$$

— see equation 6.1.3 on page 94. We finally get our series:

```
partial_sum(k,x):=sum(a[n]*legendre_p(n,x),n,0,k)
```

If we plot the first five terms against $f(x)$:

```
plot2d(['f(x),partial_sum(5,x)],[x,-1,1]);
```

we get figure 6.1.3 on the next page.

If we try 20 terms, we get figure 6.1.4 on the facing page. The Legendre series is clearly trying to approximate $f(x)$ — just as a Fourier series did.

As with Fourier series, *it turns out*[1] that the Legendre series converges in the manner

$$\lim_{n\to\infty} \int_{-1}^{1} (\text{partial\_sum}(n,x) - f(x))^2 \, dx = 0$$

Many applications of Legendre polynomials come from equation 6.1.2 on page 94: if we are in spherical coordinates and have a charge situated at the end of vector $\mathbf{x}'$, the potential energy at the end of vector $\mathbf{x}$ is expressed in the series of Legendre polynomial given above.

[1] I.e., we're not going to prove this here!

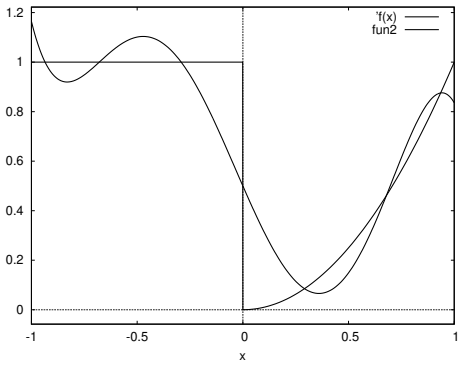


FIGURE 6.1.3. First 5 terms of a Legendre series

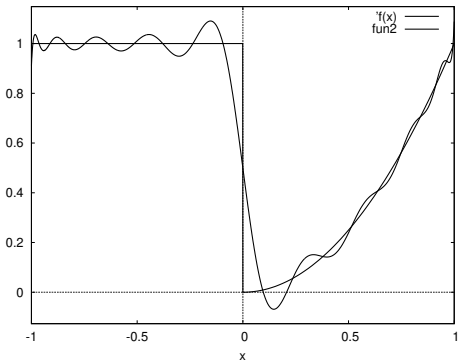


FIGURE 6.1.4. First 20 terms of a Legendre series

EXERCISES.

1. Legendre polynomials satisfy Bonnet's Recursion Formula

$$(n+1)P_{n+1}(x) = (2n+1)xP_n(x) - nP_{n-1}(x)$$

Write a Maxima function to compute $P_n(x)$, using this.

2. Expand $\sin x$ in a series of Legendre polynomials and compare with the Taylor series of the sine-function.

3. Do three-dimensional plots of $P_2(\cos\theta), P_3(\cos\theta), P_4(\cos\theta)$ in cylindrical coordinates, where θ is the angle from the $x-y$-plane and radius is 1.

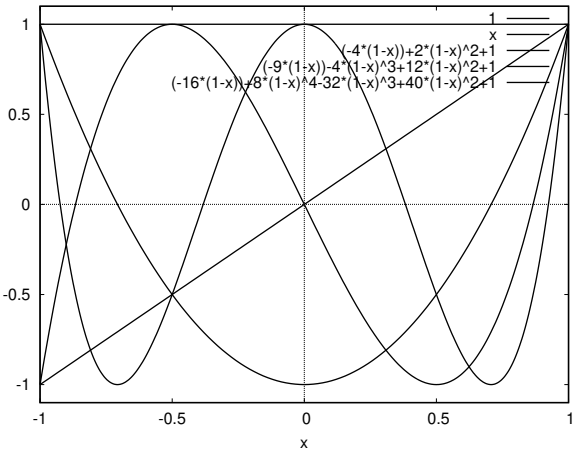


FIGURE 6.2.1. The first four Chebyshev polynomials

6.2. Weighted orthogonality

There are many other systems of orthogonal polynomials in common use. We will only touch on a few of them.

6.2.1. Chebyshev Polynomials. We begin with Chebyshev Polynomials, $\{T_n(x)\}$, defined by

$$T_n(\cos\theta) = \cos n\theta \tag{6.2.1}$$

They are orthogonal in the sense that[2]

$$\int_{-1}^{1} \frac{T_n(x)T_m(x)}{\sqrt{1-x^2}}dx = \begin{cases} 0 & \text{if } |m| \neq |n| \\ \pi & \text{if } n = m = 0 \\ \pi/2 & \text{if } n = m > 0 \end{cases} \tag{6.2.2}$$

so they are orthogonal with a *weight-function*

$$\frac{1}{\sqrt{1-x^2}}$$

Pafnuty Lvovich Chebyshev (Пафну́тий Льво́вич Чебышёв) (1821 – 1894) was a Russian mathematician and considered to be the founding father of Russian mathematics.

Figure 6.2.1 shows the first four Chebyshev polynomials.

[2]This is easily derived from definition 6.2.1, equations 4.3.3 on page 59, 4.3.6 on page 60, and 4.3.7 on page 60 and a suitable u-substitution.

```
a0:(1/%pi)*(integrate(1/sqrt(1-x^2),x,-1,0)
+integrate(x^2/sqrt(1-x^2),x,0,1));
a[n]:=(1/%pi)*
(integrate(chebyshev_t(n,x)/sqrt(1-x^2),x,-1,0)
+integrate(chebyshev_t(n,x)*x^2/sqrt(1-x^2),x,0,1));
```

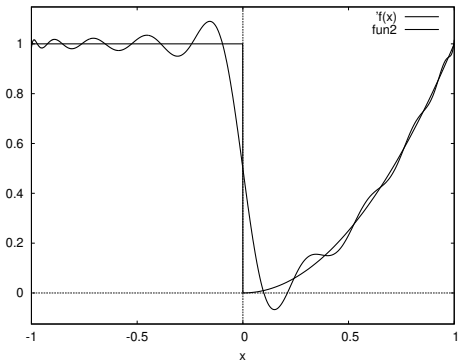


FIGURE 6.2.2. First 20 terms of a Chebyshev expansion

When we expand a functions (like $f(x)$) in a series of Chebyshev polynomials, we must take the weight function into account:

$$a_0 = \frac{1}{\pi}\int_{-1}^{1}\frac{f(x)}{\sqrt{1-x^2}}dx$$
$$a_{k>0} = \frac{2}{\pi}\int_{-1}^{1}\frac{f(x)T_k(x)}{\sqrt{1-x^2}}dx$$

In Maxima's orthopoly package, they are called chebyshev_t (n, x). Our Maxima commands are and we sum up terms of the series via

```
partial_sum(k,x):=a0+
    sum(a[n]*chebyshev_t(n,x),n,1,k)
```

If we plot the first twenty terms against our discontinuous function $f(x)$:

```
plot2d(['f(x),partial_sum(20,x)],[x,-1,1]);
```

we get figure 6.2.2. The weight-function plays a part in how the Chebyshev series converges to $f(x)$. We have

$$\lim_{n\to\infty}\int_{-1}^{1}\frac{(\text{partial\_sum}(n,x)-f(x))^2}{\sqrt{1-x^2}}dx = 0$$

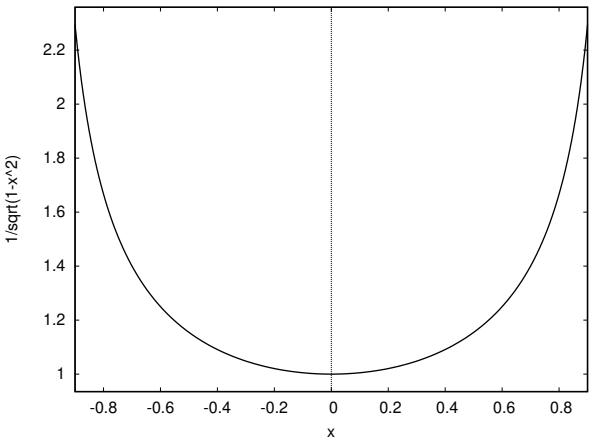


FIGURE 6.2.3. Weight-function for Chebyshev expansions

As figure 6.2.3 shows, it prioritizes the *endpoints* of the interval $[-1, 1]$.

It is known (which we won't prove!) that for *continuous* functions (something our $f(x)$ *isn't*) the Chebyshev series converges more rapidly than any other series of orthogonal polynomials — see [43]. This means they have important applications in numerical analysis. For instance many software-library functions for sines and cosines use Chebyshev expansions.

6.2.2. Laguerre Polynomials. Laguerre Polynomials are defined as certain solutions of the Laguerre Differential equation:

$$x\frac{d^2y}{dx^2} + (1-x)\frac{dy}{dx} + ny = 0$$

where $n \geq 0$ is an integer. The only nonsingular solution of this is the n^{th} Laguerre polynomial, $L_n(x)$ =**laguerre** (n, x). These polynomials are orthogonal with respect to the weight function e^{-x}:

$$\int_0^\infty e^{-x}\text{laguerre}(n,x)\cdot\text{laguerre}(m,x)dx = 0 \text{ if } n \neq m$$

and[3]

$$\int_0^\infty e^{-x}\text{laguerre}(n,x)^2 dx = 1$$

[3]These are not obvious!

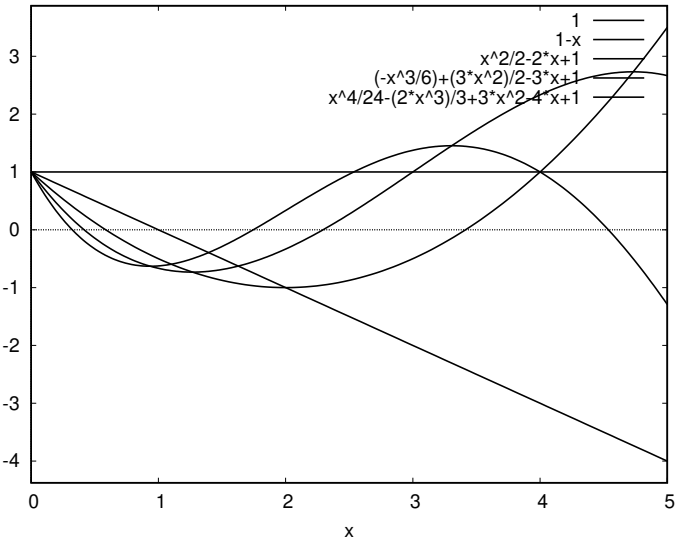


FIGURE 6.2.4. Laguerre polynomials

Edmond Nicolas Laguerre (1834 – 1886) was a French mathematician and a member of the Académie des sciences (1885). His main works were in the areas of geometry and complex analysis. He also investigated orthogonal polynomials. Laguerre's method is a root-finding algorithm tailored to polynomials.

The Laguerre polynomials arise in quantum mechanics, in the radial part of the solution of the Schrödinger equation for a one-electron atom. We will expand our discontinuous function in these polynomials with coefficients

```
a0:integrate(%e^(-x)*x^2,x,0,1);
a[n]:=integrate(laguerre(n,x)*x^2*%e^(-x),x,0,1);
```

and partial sum

```
partial_sum(k,x):=a0+sum(a[n]*laguerre(n,x),n,1,k)
```

If we plot the first 100 terms of the series, we get figure 6.2.5, which doesn't look very good until you realize that it only converges with the weight function e^{-x}.

In other words

$$\int_0^\infty e^{-x}|f(x) - p_n(x)|dx \to 0$$

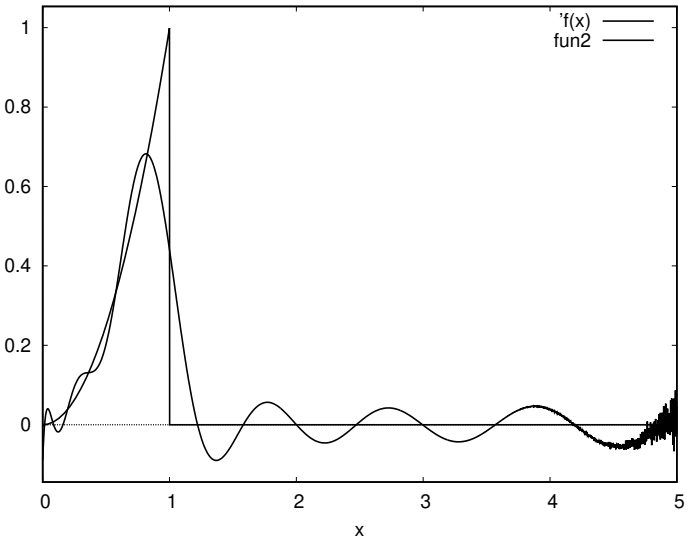


FIGURE 6.2.5. Expansion of f(x) in 100 Laguerre polynomials

as $n \to \infty$, where $p_n(x)$ is the n^{th} partial sum of a Laguerre series. Since e^{-x} tapers off rapidly as x increases, we don't really care what $p_n(x)$ does for large values of x.

Laguerre polynomials are often used to numerically estimate integrals of the form

$$\int_0^\infty e^{-x} f(x) dx$$

(for the degree-n form of the equation with $n \geq 1$) via the *Gauss-Laguerre quadrature formula*

$$\int_0^\infty e^{-x} f(x) dx \approx \sum_{i=1}^{n} w_i f(x_i) \tag{6.2.3}$$

where the $\{x_i\}$ are the roots of $L_n(x)$ and the weights $\{w_i\}$ are given by

$$w_i = \frac{x_i}{(n+1)^2 L_{n+1}(x_i)} \tag{6.2.4}$$

see [55].

EXERCISES.

1. Code a function that does Gauss-Laguerre quadrature using equations 6.2.3 and 6.2.4 on the preceding page . Hint: use the **allroots** command to find the roots of $L_n(x)$).

6.2.3. Hermite polynomials. These are orthogonal with respect to the weight function e^{-x^2}. They exist in two closely-related forms, the *physicist's* Hermite polynomials, defined by

$$H_n(x) = (-1)^n e^{x^2} \frac{d^n}{dx^n}\left(e^{-x^2}\right)$$

and the *probability-theorist's* form

$$He_n(x) = (-1)^n e^{x^2/2} \frac{d^n}{dx^n}\left(e^{-x^2/2}\right)$$

We will consider the physicist's form here, which appear naturally in the Schrödinger wave equation for a harmonic oscillator in quantum mechanics.

Their orthogonality relations are

$$\int_{-\infty}^{\infty} e^{-x^2} H_n(x) H_m(x) dx = \begin{cases} 0 & \text{if } n \neq m \\ \sqrt{\pi} 2^n n! & \text{otherwise} \end{cases}$$

In the **orthopoly** library, the n^{th} Hermite polynomial is denoted **hermite**(n,x). The first five Hermite polynomials are plotted in figure 6.2.6 on the following page.

Charles Hermite (1822 – 1901) was a French mathematician who did research concerning number theory, quadratic forms, invariant theory, orthogonal polynomials, elliptic functions, and algebra.
Hermite polynomials, Hermite interpolation, Hermite normal form, Hermitian operators, and cubic Hermite splines are named in his honor. One of his students was Henri Poincaré.
He did *not* discover Hermite polynomials[a]: Hermite polynomials were defined by Pierre-Simon Laplace in 1810 and studied in detail by Pafnuty Chebyshev in 1859. Chebyshev's work was overlooked, and they were named later after Charles Hermite, who wrote on the polynomials in 1864, describing them as new.

[a] See Stigler's Law of Eponymy in the index!

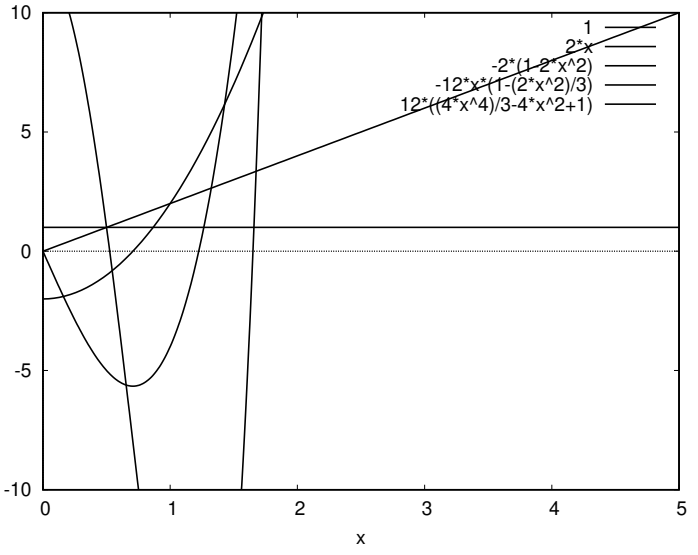


FIGURE 6.2.6. Hermite polynomials

EXERCISES.

2. Expand our discontinuous function, $f(x)$, (see equation 3.2.1 on page 35) in Hermite polynomials.

CHAPTER 7

Linear Algebra

"Life stands before me like an eternal spring with new and brilliant clothes."
— Carl Friedrich Gauss.

7.1. Introduction

We assume the reader is familiar with the basic concepts of linear algebra — see [58, chapter 6] as a general reference.

Initially, the focus of linear algebra was solving systems of linear equations in multiple variables. Some 4000 years ago, Babylonians were able to solve pairs of linear equations in two unknowns. In 200BC, the Chinese publication, "Nine Chapters of the Mathematical Art" (see [36]) showed how to solve systems of three equations in three unknowns.

The **solve**-command can handle simple systems of linear equations:

Given

$$\begin{aligned} 2x + 3y &= 5 \\ 6x - y &= 2 \end{aligned} \tag{7.1.1}$$

where we must solve for x and y. If we type

```
solve([2*x+3*y=5,6*x-y=2],[x,y])
```

and Maxima replies with

$$\left[\left[x = \frac{11}{20}, y = \frac{13}{10}\right]\right]$$

Here's another example:

$$\begin{aligned} x + 2y - z &= 0 \\ 3x - y + 2z &= 0 \end{aligned}$$

which we code as

```
solve([x+2*y-z=0,3*x-y+2*z=0],[x,y,z])
```

and Maxima replies with

$$\left[\left[x = -\frac{3\%r1}{7}, y = \frac{5\%r1}{7}, z = \%r1\right]\right]$$

Here, Maxima has introduced an auxiliary variable, %r1, that can take on *arbitrary* values, showing that there are an infinite number of solutions to this system.

In 1848, Sylvester realized that an array of *coefficients* was all that really mattered in these equations and coined the term "matrix" for them from the Latin word for "mother" and "womb".

Coding matrices in Maxima is done with the **matrix**-command:

```
a : matrix([1,2,3],[4,5,6],[7,8,9])
```

gives

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

The usual operations '+' and '-' work for matrices. The multiplication-operation, '$*$' multiplies them *element-by-element*, which is not what we want. To correctly multiply matrices, use the '**.**'-operator.

For instance, if we type

```
b : matrix([1,2,-3],[4,5,6],[7,8,10])
```

then

```
a . b
```

produces

$$\begin{bmatrix} 30 & 36 & 39 \\ 66 & 81 & 78 \\ 102 & 126 & 117 \end{bmatrix}$$

```
a+b
```

produces

$$\begin{bmatrix} 2 & 4 & 0 \\ 8 & 10 & 12 \\ 14 & 16 & 19 \end{bmatrix}$$

wxmaxima has a short-cut to entering matrices: select the menu-item **Matrix▷Enter Matrix**.

NOTE 7.1.1. Matrices are always assumed to be *two-dimensional*, even if they only have one row! So, if we define

```
a : matrix([1,2,-3])
```

we access the elements with *two* subscripts,

```
a[1,1], a[1,2], a[1,3]
```

rather than one.

Given a matrix, one can determine how many *rows* it has via the **length**-command:

```
length(b)
```

returns

3

7.1.1. Matrix-creation commands. Besides the **matrix**-command, we have several others to create matrices:

(1) the **ident**(n)-command creates an $n \times n$ identity matrix. If M is an $n \times n$ matrix, **identfor**(M) is an $n \times n$ *identity* matrix.
(2) the **zeromatrix**(m, n)-command creates an $m \times n$ matrix of zeroes.
(3) the **genmatrix**(*ident*,*m*,*n*)-command is the most powerful of the matrix-creation commands. If *ident* is an identifier with no other properties, **genmatrix** produces an $m \times n$ matrix with subscripted copies of *ident*. For example

```
genmatrix(a,3,4)
```

produces

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \end{bmatrix}$$

If *ident* is the name of a memoized function of two variables, **genmatrix** plugs row and column numbers into ident and posts the value of the function to the array

```
b[i,j]:=i+j;
genmatrix(b,3,4)
```

produces

$$\begin{bmatrix} 2 & 3 & 4 & 5 \\ 3 & 4 & 5 & 6 \\ 4 & 5 & 6 & 7 \end{bmatrix}$$

ident can also be an anonymous **lambda**-function, so

```
genmatrix(lambda([i,j],i-j),3,4)
```

produces

$$\begin{bmatrix} 0 & -1 & -2 & -3 \\ 1 & 0 & -1 & -2 \\ 2 & 1 & 0 & -1 \end{bmatrix}$$

7.1.2. Matrix operations. The **transpose**-command does what you'd expect it to do:

```
transpose(a)
```

gives

$$\begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$$

We will also be concerned with vectors, which we will regard as matrices with a single row or, more often, a single column. The '.' operation doubles as the dot product for vectors, so

```
r:transpose(matrix([1,2,-3]))
```

produces

$$\begin{bmatrix} 1 \\ 2 \\ -3 \end{bmatrix}$$

and

```
s:transpose(matrix([4,5,6]))
```

produces

$$\begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix}$$

Now

```
r.s
```

is undefined as a *matrix*-product, but gives the dot-product of r and s as vectors, namely -4. Technically, the valid matrix product is

```
transpose(r).s
```

which also produces the dot-product, -4.

We can write a simple function for the *norm* of a vector

```
norm(v):=sqrt(v.v);
```

or, the technically more correct

```
norm(v):=sqrt(transpose(v).v);
```

Matrix-assignment, like

```
x:b
```

merely causes x to become *an alias* for b. Changes to x like

```
x[2,1]:-100
```

will be immediately reflected in b:

$$\begin{bmatrix} 1 & 2 & -3 \\ -100 & 5 & 6 \\ 7 & 8 & 10 \end{bmatrix}$$

If we want an *independent* copy of a matrix, we must use the **copymatrix**-command

```
x : copymatrix(b)
```

We can easily compute integral powers of matrices too:[1]

```
a^^2
```

produces

$$\begin{bmatrix} 30 & 36 & 42 \\ 66 & 81 & 96 \\ 102 & 126 & 150 \end{bmatrix}$$

```
b^^-1
```

produces the inverse

$$b^{-1} = \begin{bmatrix} \frac{2}{15} & -\frac{44}{15} & \frac{9}{5} \\ \frac{2}{15} & \frac{31}{15} & -\frac{6}{5} \\ -\frac{1}{5} & \frac{2}{5} & -\frac{1}{5} \end{bmatrix}$$

DEFINITION 7.1.2. To analyze more complex linear systems, we simplify the matrix of coefficients by performing *elementary row-operations:*

Type 1: subtracting a multiple of one row from another**.** Maxima command: **rowop**(M, i, j, theta) replaces row i in the matrix M by row i - theta*row j.

Type 2: involves swapping two rows of a matrix. Maxima command: **rowswap**(M,i,j) which swaps rows i and j of the matrix M.

Type 3: involves multiplying a row of a matrix by a nonzero constant. There is no Maxima command to do this.

If the matrix consists of coefficients of a linear system, these operations produce a system that is mathematically equivalent to the original. Our goal is to make the matrix *triangular:*

DEFINITION 7.1.3. An $n \times n$ matrix, A, is called *upper-triangular* if$A_{i,j} = 0$ whenever $i > j$. The matrix, A, is *lower triangular* if $A_{i,j} = 0$ whenever $j > i$.

[1]The use of a *single* carat, ^, is also well defined but it raises the *entries* in the matrix to powers, which is not correct.

REMARK. The term "upper-triangular" comes from the fact that A looks like

$$\begin{bmatrix} A_{1,1} & A_{1,2} & \cdots & A_{1,n-1} & A_{1,n} \\ 0 & A_{2,2} & \ddots & A_{2,n-1} & A_{2,n} \\ 0 & 0 & \ddots & \vdots & \vdots \\ \vdots & \vdots & \ddots & A_{n-1,n-1} & A_{n-1,n} \\ 0 & 0 & \cdots & 0 & A_{n,n} \end{bmatrix}$$

The process of converting a matrix to upper triangular form is called *Gaussian Elimination*.

Carl Friedrich Gauss (1777 – 1855) was a German mathematician and physicist who made significant contributions to many fields in mathematics and science. Sometimes referred to as the *Princeps mathematicorum* (Latin for "the foremost of mathematicians") and "the greatest mathematician since antiquity", Gauss had an exceptional influence in many fields of mathematics and science, and is ranked among history's most influential mathematicians.
In surveying land around Hannover, he invented many modern surveying instruments and the field of differential geometry. This paved the way for Riemannian geometry and Einstein's theory of General Relativity. He also discovered least-squares approximations (see section 7.3.1 on page 119) for estimating orbits of planets and asteroids given many slightly differing observations. Least squares was used to predict the future location of the newly discovered asteroid, Ceres.
Ironically, Gauss *didn't* discover Gaussian Elimination, which was first mentioned (in Europe) by Isaac Newton. He *did* discover the fast Fourier transform 160 years before its *official* discoverers, Cooley and Tukey. This is a time-honored tradition in mathematics called *Stigler's law of eponymy*[a] of naming results after people who *didn't* discover them.

[a] True to itself, it was first proposed by the sociologist, Robert Merton, *not* Stigler ☺!

Maxima has a **triangularize**-command to do this

```
triangularize(a)
```

produces

$$\begin{bmatrix} 1 & 2 & 3 \\ 0 & -3 & -6 \\ 0 & 0 & 0 \end{bmatrix}$$

The related **echelon**-command produces a normalized form of this matrix with the first nonzero entry of each row set to 1. The 1 that begins each nonzero row is called its *pivot*. So

```
echelon(a)
```

```
/* We must write a function to compute
   reduced echelon form */
reduced_echelon(a):=block([rows,cols,k,temp],
          [rows,cols]:matrix_size(a),
          temp:echelon(a), /* this copies a */
          k:min(rows,cols),
          for i thru min(rows,cols)
             /* Find pivot */
             do (if temp[i,i]=0 then
                    (k:i-1,return())),
             /* Clear out column i */
          for i:k thru 2 step -1 do
             (for j from i-1 thru 1 step -1
               do temp:rowop(temp,j,i,temp[j,i])),
          temp) /* return the result */
```

FIGURE 7.1.1. Code for a reduced echelon matrix

produces

$$\begin{bmatrix} \mathbf{1} & 2 & 3 \\ 0 & \mathbf{1} & 2 \\ 0 & 0 & 0 \end{bmatrix}$$

where we have highlighted the pivots.

To solve a linear system, we really want a *reduced* echelon matrix where we perform additional row-operations to make the pivot in each row the *only* nonzero element in its column — see figure 7.1.1.

So

```
reduced_echelon(a)
```

subtracts $2\times$ row 2 from row 1 to produce

$$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & 2 \\ 0 & 0 & 0 \end{bmatrix}$$

DEFINITION 7.1.4. If $s = \{v_1, \ldots, v_n\}$ are elements of $\mathbb{R}^n$,their *span*, $\text{Span}(s)$ is the set of all possible linear combinations

$$\sum_{i=1}^{n} \alpha_i v_i$$

for $\alpha_i \in \mathbb{R}$. It forms a subspace of $\mathbb{R}^n$.

Recall that the *column space* of a matrix, A, is the vector space of all vectors of the form $A\mathbf{v}$ for all vectors $\mathbf{v}$. We have a **columnspace**-command to compute this

```
columnspace(a)
```

$$\text{span}\left(\begin{bmatrix}1\\4\\7\end{bmatrix}, \begin{bmatrix}2\\5\\8\end{bmatrix}\right)$$

Recall that the *null space*, $\text{Null}(A)$, of a matrix, A, is the set of vectors, $\mathbf{v}$, such that $A\mathbf{v} = 0$. This is the **nullspace**-command:

```
nullspace(a)
```

$$\text{span}\left(\begin{bmatrix}-3\\6\\-3\end{bmatrix}\right)$$

Recall that determinants are defined by

DEFINITION 7.1.5. If M is an $n \times n$ matrix, its *determinant*, $\det(M)$ is defined by

$$\det(M) = \sum_{\sigma \in S_n} \wp(\sigma) M_{1,\sigma(1)} \cdots M_{n,\sigma(n)} \tag{7.1.2}$$

where the sum is taken over all $n!$ permutations in S_n. Here $\wp(\sigma)$ is the *parity* of a permutation, defined in terms of the number of *inversions* it produces. An inversion exists for a permutation, σ, if there is a pair of elements x, y such that $x < y$ and $\sigma(x) > \sigma(y)$.

$$\wp(\sigma) = \begin{cases} +1 & \text{if the total number of inversions is even} \\ -1 & \text{otherwise} \end{cases}$$

REMARK. Equation 7.1.2 is due to Euler. It is not particularly suited to computation of the determinant since it is a sum of $n!$ terms.

DEFINITION 7.1.6. If $B = \prod_{i=1}^{n}[a_i, b_i]$ is a box in $\mathbb{R}^n$, its *volume*, $\text{vol}(B)$ is defined by

$$\text{vol}(B) = \prod_{i=1}^{n}(b_i - a_i)$$

This is used to define the Lebesgue measure:

DEFINITION 7.1.7. If $R \subset \mathbb{R}^n$ is a region, its *outer Lebesgue measure* is defined by

$$\lambda(R) = \inf\left\{\sum_{B \in \mathcal{C}} \text{vol}(B) : \mathcal{C} \text{ is a countable set of boxes whose union covers } R\right\}$$

Recall the geometric interpretation of the determinant:

THEOREM 7.1.8. *If A is an $n \times n$ matrix, $R \subset \mathbb{R}^n$, then*

$$\lambda(A(R)) = |\det A| \cdot \lambda(R)$$

REMARK. So the determinant gives the effect of a linear transformation on *volumes.* Analytic geometry and manifold theory considers volumes to have *signs,* in which case we do not take the *absolute value* of the determinant.

See [58] for a proof.

Maxima has a **determinant**-command that computes these efficiently

```
determinant(a)
```

$$0$$

```
determinant(b)
```

$$15$$

More recently, Fateman has implemented a **newdet**-command that is faster than **determinant** but uses more memory — see [25]. If

$$z = \begin{pmatrix} 2x-1 & 37 & -9 \\ 3x^2+x & 2x & 3x \\ 51 & 2x & 7 \end{pmatrix}$$

then

```
newdet(z)
```

produces

$$-66x^3 - 761x^2 + 6306x$$

EXERCISES.

1. What is the volume of the parallelepiped spanned by the three vectors

$$v_1 = \begin{bmatrix} 1 \\ 5 \\ 6 \end{bmatrix}, \quad v_2 = \begin{bmatrix} 2 \\ 1 \\ 3 \end{bmatrix} \quad v_3 = \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix}$$

?

2. What might the sign of the volume mean (when a determinant has a negative sign)?

7.2. Changes of basis

Recall that a basis of a vector-space is like a "coordinate system" for it: every vector can be *uniquely* written as a linear combination of the basis-elements.

Suppose we have a vector-space with basis $\{e_i\}$, $i = 1, \ldots, n$ and we are given a new basis $\{b_i\}$. If

$$\begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}$$

is a vector in this new basis, then

$$x_1 b_1 + \cdots + x_n b_n = \begin{bmatrix} b_1 & \cdots & b_n \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}$$

is the same vector in the old basis, where $P = \begin{bmatrix} b_1 & \cdots & b_n \end{bmatrix}$ is an $n \times n$ matrix whose columns are the basis-vectors. Since the basis-vectors are linearly independent, P is invertible. Since P converts from the new basis to the old one, P^{-1} performs the reverse transformation.

For instance, suppose $\mathbb{R}^3$ has the standard basis and we have a new basis

$$b_1 = \begin{bmatrix} 28 \\ -25 \\ 7 \end{bmatrix}, b_2 = \begin{bmatrix} 8 \\ -7 \\ 2 \end{bmatrix}, b_3 = \begin{bmatrix} 3 \\ -4 \\ 1 \end{bmatrix}$$

We form a matrix from these columns:

$$P = \begin{bmatrix} 28 & 8 & 3 \\ -25 & -7 & -4 \\ 7 & 2 & 1 \end{bmatrix}$$

whose determinant is verified to be 1. The vector

$$\begin{bmatrix} 1 \\ -1 \\ 1 \end{bmatrix}$$

in the new basis is

$$b_1 - b_2 + b_3 = P \begin{bmatrix} 1 \\ -1 \\ 1 \end{bmatrix} = \begin{bmatrix} 23 \\ -36 \\ 10 \end{bmatrix}$$

If we want to convert the vector

$$\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

in the standard basis into the new basis, we get

$$P^{-1}\begin{bmatrix}1\\2\\3\end{bmatrix} = \begin{bmatrix}1 & -2 & -11\\ -3 & 7 & 37\\ -1 & 0 & 4\end{bmatrix}\begin{bmatrix}1\\2\\3\end{bmatrix} = \begin{bmatrix}-36\\122\\11\end{bmatrix}$$

and a simple calculation shows that

$$-36b_1 + 122b_2 + 11b_3 = \begin{bmatrix}1\\2\\3\end{bmatrix}$$

For matrices, changes of basis are a bit more complicated. Suppose V is an n-dimensional vector-space and an $n \times n$ matrix, A, represents a linear transformation

$$f\colon V \to V$$

with respect to some basis. If $\{b_1, \ldots, b_n\}$ is a new basis for V, let

$$P = [b_1, \ldots, b_n]$$

be the matrix whose columns are the b_i. We can compute the matrix representation of f in this new basis, $\bar{A}$, via

$$\begin{array}{ccc} V_{\text{old}} & \xrightarrow{A} & V_{\text{old}} \\ {\scriptstyle P}\uparrow & & \downarrow{\scriptstyle P^{-1}} \\ V_{\text{new}} & \xrightarrow[\bar{A}]{} & V_{\text{new}} \end{array}$$

In other words, to compute a matrix representation for f in the new basis:

(1) convert to the *old* basis (multiplication by P)
(2) *act* via the matrix A, which represents f in the old basis
(3) convert the result to the *new* basis (multiplication by P^{-1}).

We summarize this with

THEOREM 7.2.1. *If A is an $n \times n$ matrix representing a linear transformation*

$$f\colon V \to V$$

with respect to some basis $\{e_1, \ldots, e_n\}$ and we have a new basis $\{b_1, \ldots, b_n\}$ with

$$P = \begin{bmatrix} b_1 & \cdots & b_n \end{bmatrix}$$

then, in the new basis, the transformation f is represented by

$$\bar{A} = P^{-1}AP$$

EXERCISES.

1. Solve the system of linear equations

$$\begin{aligned} 2x + 3y + z &= 8 \\ 4x + 7y + 5z &= 20 \\ -2y + 2z &= 0 \end{aligned}$$

2. Solve the system

$$\begin{aligned} 2x + 3y + 4z &= 0 \\ x - y - z &= 0 \\ y + 2z &= 0 \end{aligned}$$

3. If V is a 3-dimensional vector-space with a standard basis and

$$b_1 = \begin{bmatrix} 8 \\ 4 \\ 3 \end{bmatrix}, b_2 = \begin{bmatrix} -1 \\ 0 \\ -1 \end{bmatrix}, b_3 = \begin{bmatrix} 2 \\ 1 \\ 1 \end{bmatrix}$$

is a new basis, convert the matrix

$$A = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 1 & 3 \\ -1 & 2 & 1 \end{bmatrix}$$

to the new basis.

7.3. Dot-products and projections

Dot-products have a great deal of geometric significance. We start with:

DEFINITION 7.3.1. If $\mathbf{v} \in \mathbb{R}^n$, define $\|\mathbf{v}\| = \sqrt{\mathbf{v} \bullet \mathbf{v}}$ — the *norm* of $\mathbf{v}$. A *unit vector* $\mathbf{u} \in \mathbb{R}^n$ is one for which $\|\mathbf{u}\| = 1$.

THEOREM 7.3.2. *Let $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ be two vectors with an angle θ between them. Then*

$$\cos(\theta) = \frac{\mathbf{x} \bullet \mathbf{y}}{\|\mathbf{x}\| \cdot \|\mathbf{y}\|} \tag{7.3.1}$$

PROOF. See theorem 6.2.66 in [58]. □

One immediate consequence is:

REMARK 7.3.3. Vectors $\mathbf{u}$ and $\mathbf{v}$ are *perpendicular* if and only if

$$\mathbf{u} \bullet \mathbf{v} = 0$$

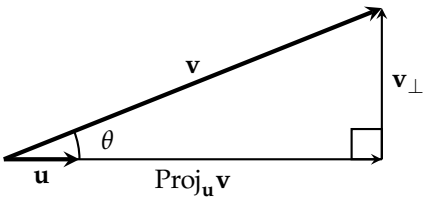


FIGURE 7.3.1. Projection of a vector onto another

DEFINITION 7.3.4. Let $\mathbf{u} \in \mathbb{R}^n$ be a unit vector and $\mathbf{v} \in \mathbb{R}^n$ be some other vector. Define *the projection of* $\mathbf{v}$ *onto* $\mathbf{u}$ via

$$\text{Proj}_{\mathbf{u}}\mathbf{v} = (\mathbf{u} \bullet \mathbf{v})\mathbf{u}$$

Also define

$$\mathbf{v}_\perp = \mathbf{v} - \text{Proj}_{\mathbf{u}}\mathbf{v}$$

REMARK. Note that $\text{Proj}_{\mathbf{u}}\mathbf{v}$ is parallel to $\mathbf{u}$ with a length of $\|\mathbf{v}\| \cdot \cos\theta$, where θ is the angle between $\mathbf{u}$ and $\mathbf{v}$. Also note that

$$\begin{aligned}\mathbf{u} \bullet \mathbf{v}_\perp &= \mathbf{u} \bullet (\mathbf{v} - (\mathbf{u} \bullet \mathbf{v})\mathbf{u}) \\ &= \mathbf{u} \bullet \mathbf{v} - (\mathbf{u} \bullet \mathbf{v})\mathbf{u} \bullet \mathbf{u} \\ &= \mathbf{u} \bullet \mathbf{v} - \mathbf{u} \bullet \mathbf{v} = 0\end{aligned}$$

so $\mathbf{v}_\perp$ is *perpendicular* to $\mathbf{u}$ as per remark 7.3.3 on the preceding page.

Since $\mathbf{v} = \text{Proj}_{\mathbf{u}}\mathbf{v} + \mathbf{v}_\perp$, we have represented $\mathbf{v}$ as a sum of a vector *parallel* to $\mathbf{u}$ and one *perpendicular* to it. See figure 7.3.1.

We can generalize projection to multiple dimensions:

DEFINITION 7.3.5. Let $\mathbf{u}_1, \ldots, \mathbf{u}_k \in \mathbb{R}^n$ be a set of vectors. This set is defined to be *orthonormal* if

$$\mathbf{u}_i \bullet \mathbf{u}_j = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise} \end{cases} \tag{7.3.2}$$

As with a single vector, we can define *projections* in this case.

DEFINITION 7.3.6. If V is an inner-product space, $\mathbf{v} \in V$, and $S = \{\mathbf{u}_1, \ldots, \mathbf{u}_k\}$ is an orthonormal set of vectors with $W = \text{Span}(S)$, then define

$$\text{Proj}_W\mathbf{v} = \sum_{i=1}^{k} (\mathbf{u}_i \bullet \mathbf{v})\, \mathbf{u}_i$$

If $\mathbf{v} \notin W$, what is the relation between $\mathbf{v}$ and $\text{Proj}_W\mathbf{v}$?

PROPOSITION 7.3.7. *If $S = \{\mathbf{u}_1, \ldots, \mathbf{u}_k\}$ is an orthonormal set of vectors that span $W \subset \mathbb{R}^n$, and $\mathbf{v}$ is any other vector, then*

$$\mathbf{v}_\perp = \mathbf{v} - \text{Proj}_W\mathbf{v}$$

has the property that $\mathbf{v}_\perp \bullet \mathbf{u}_j = 0$ *for all* $j = 1, \ldots, k$*, making it perpendicular to all of* W*. It follows that* $\text{Proj}_W\mathbf{v}$ *is the vector in* W *closest to* $\mathbf{v}$ *in the sense that*

$$\|\mathbf{v} - \mathbf{w}\| > \|\mathbf{v} - \text{Proj}_W\mathbf{v}\|$$

for any $\mathbf{w} \in W$ *with* $\mathbf{w} \neq \text{Proj}_W\mathbf{v}$.

PROOF. See proposition 6.2.88 in [58]. □

Maxima has a library called **eigen** that adds additional functions to the system. For instance:

```
load("eigen");
x:matrix([1,2,3]);
unitvector(x);
```

results in

$$\left(\frac{1}{\sqrt{14}} \quad \frac{2}{\sqrt{14}} \quad \frac{3}{\sqrt{14}}\right)$$

so it returns

$$\frac{x}{\|x\|}$$

EXERCISES.

1. Compute the angle between the vectors

$$\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} 1 \\ -2 \\ 0 \end{bmatrix}$$

2. Consider the unit vector

$$\mathbf{u} = \begin{bmatrix} 1/\sqrt{3} \\ 1/\sqrt{3} \\ 1/\sqrt{3} \end{bmatrix}$$

If

$$\mathbf{v} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

compute $\text{Proj}_\mathbf{u}\mathbf{v}$ and $\mathbf{v}_\perp$. Write Maxima functions to do this.

3. Given vectors

$$\mathbf{v}_1 = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}, \mathbf{v}_2 = \begin{bmatrix} 1 \\ 0 \\ -1 \\ 1 \end{bmatrix}, \mathbf{v}_3 = \begin{bmatrix} 2 \\ -1 \\ -2 \\ -1 \end{bmatrix}$$

in $\mathbb{R}^4$, find an orthonormal set $\{\mathbf{u}_1, \mathbf{u}_2, \mathbf{u}_3\}$ with

$$\text{Span}\{\mathbf{u}_1, \mathbf{u}_2, \mathbf{u}_3\} = \text{Span}\{\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3\}$$

Hint: $\mathbf{u}_1 = \mathbf{v}_1/\|\mathbf{v}_1\|$. Now project $\mathbf{v}_2$onto $\mathbf{u}_1$, compute $(\mathbf{v}_2)_\perp$, and make *that* into a unit vector, etc. This process is called the Gram-Schmidt Algorithm. The **eigen** library has a **gramschmidt**-command that *almost* carries this out.

7.3.1. Linear least squares. Suppose we are given a collection of data $\{(x_1, y_1), \ldots, (x_n, y_n)\}$ and would like to find a function $f(x)$ such that $f(x_i) = y_i$. Or, failing this, we would like to find a function that fits this data "as closely as possible". What do we mean by this?

Least squares tries to find a function that minimizes

$$\sum_{i=1}^{n} (f(x_i) - y_i)^2$$

as in figure 7.3.2 on the next page.

We will begin with the simplest case: $f(x) = c_1 x + c_0$. We get a vector

$$D = \begin{bmatrix} c_1x_1 + c_0 - y_1 \\ c_1x_2 + c_0 - y_2 \\ \vdots \\ c_1x_{t-1} + c_0 - y_{t-1} \\ c_1x_t + c_0 - y_t \end{bmatrix} = \begin{bmatrix} x_1 & 1 \\ x_2 & 1 \\ \vdots & \vdots \\ x_{t-1} & 1 \\ x_t & 1 \end{bmatrix} \begin{bmatrix} c_1 \\ c_0 \end{bmatrix} - \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_{t-1} \\ y_t \end{bmatrix}$$

or

$$D = XC - Y$$

where

$$X = \begin{bmatrix} x_1 & 1 \\ x_2 & 1 \\ \vdots & \vdots \\ x_{t-1} & 1 \\ x_t & 1 \end{bmatrix}, \; C = \begin{bmatrix} c_1 \\ c_0 \end{bmatrix}$$

and

$$Y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_{t-1} \\ y_t \end{bmatrix}$$

We want to minimize $\|D\|^2 = D^t D$. We get

$$\begin{aligned} (XC - Y)^t (XC - Y) &= (C^t X^t - Y^t)(XC - Y) \\ &= C^t X^t XC - C^t X^t Y - Y^t XC + Y^t Y \end{aligned}$$

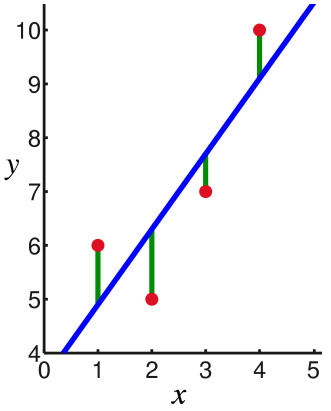


FIGURE 7.3.2. Least squares fit

Now we differentiate by the c_i (i.e.form the gradient) and set it to zero, to get

$$2X^tXC - 2X^tY = 0$$

since $C^tX^tY = Y^tXC$ (why?[2]).

Our least-squares problem becomes

$$X^tXC = X^tY \tag{7.3.3}$$

This simple (i.e., degree-1) case is often used in the business world, where it's called *linear regression*. It shows whether random-appearing data is (on the average) trending upward or downward.

Incidentally, equation 7.3.3 will be used in much more complex examples of least-squares fits; only the definition of X will change.

EXAMPLE 7.3.8. Suppose

$$Y = \begin{bmatrix} 1.827619199225791 \\ 0.3903692355955774 \\ 0.9647810497032392 \\ 0.7108801143185723 \\ 0.5044777533707618 \end{bmatrix}$$

and $x_i = i$

$$X = \begin{bmatrix} 1 & 1 \\ 2 & 1 \\ 3 & 1 \\ 4 & 1 \\ 5 & 1 \end{bmatrix}$$

[2]They're both scalars!

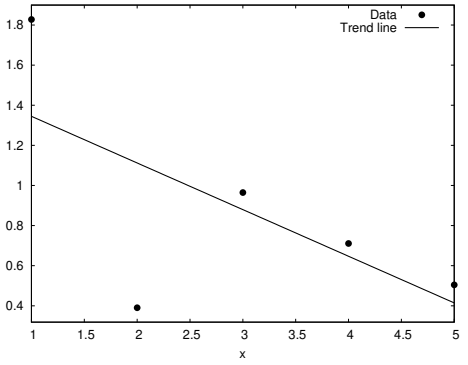


FIGURE 7.3.3. Linear regression

then equation 7.3.3 on the preceding page is

$$\begin{bmatrix} 55 & 15 \\ 15 & 5 \end{bmatrix} \begin{bmatrix} c_1 \\ c_0 \end{bmatrix} = \begin{bmatrix} 10.86861004365476 \\ 4.398127352213942 \end{bmatrix}$$

from which we get

$$\begin{bmatrix} c_1 \\ c_0 \end{bmatrix} = \begin{bmatrix} -0.2325772012987064 \\ 1.577357074338908 \end{bmatrix}$$

or

$$y \sim x \cdot (-0.2325772012987064) + 1.577357074338908$$

so the data is trending downwards. We can plot it with

```
plot2d([[discrete,[[1,y[1,1]],[2,y[2,1]],[3,y[3,1]],
[4,y[4,1]],[5,y[5,1]  ]  ]],
x*(-0.2325772012987064)+1.577357074338908],
     [x,1,5],[style,[points,4,7,1],[lines,2,1]],
[legend,"Data","Trend line"]);
```

to get figure 7.3.3.

Incidentally, the [**style**,[**points**,4,7,1],[**lines**,2,1]] trailing the rest of the plot-specifications gives the respective *styles* of the two function-plots. These always follow the other options. The specification [**points**,4,7,1] specifies that the first plot is *disconnected points* (the default is to connect the points with lines). The specification takes the form [**points**,diameter,type_of_point,color]. See table F.1.1 on page 327 in appendix F on page 323 for the codes.

Now we'll look at a more complex example: finding a fourth-degree polynomial that is a "formula" for the first 10 prime numbers.

EXAMPLE 7.3.9. In this case

$$Y = \begin{bmatrix} 2 \\ 3 \\ 5 \\ 7 \\ 11 \\ 13 \\ 17 \\ 19 \\ 23 \\ 29 \end{bmatrix}$$

and, since we're approximating these by $f(x) = c_4x^4 + c_3x^3 + c_2x^2 + c_1x + c_0$ with $x_i = i$ for $i = 1, \ldots, 10$, we get $X_{i,j} = i^{5-j}$ and use the command

```
X: genmatrix(lambda([i,j],i^(5-j)),10,5)
```

to get

$$X = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 16 & 8 & 4 & 2 & 1 \\ 81 & 27 & 9 & 3 & 1 \\ 256 & 64 & 16 & 4 & 1 \\ 625 & 125 & 25 & 5 & 1 \\ 1296 & 216 & 36 & 6 & 1 \\ 2401 & 343 & 49 & 7 & 1 \\ 4096 & 512 & 64 & 8 & 1 \\ 6561 & 729 & 81 & 9 & 1 \\ 10000 & 1000 & 100 & 10 & 1 \end{bmatrix}$$

so X^tX is

$$\begin{bmatrix} 167731333 & 18080425 & 1978405 & 220825 & 25333 \\ 18080425 & 1978405 & 220825 & 25333 & 3025 \\ 1978405 & 220825 & 25333 & 3025 & 385 \\ 220825 & 25333 & 3025 & 385 & 55 \\ 25333 & 3025 & 385 & 55 & 10 \end{bmatrix}$$

and X^tY is

$$\begin{bmatrix} 585514 \\ 66118 \\ 7726 \\ 952 \\ 129 \end{bmatrix}$$

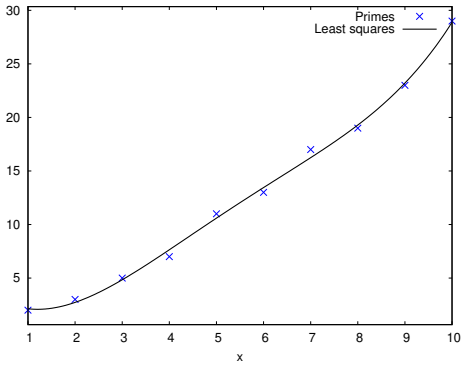


FIGURE 7.3.4. "Formula" for prime numbers

and C is given by

$$(X^tX)^{-1}(X^tY) = C = \begin{bmatrix} \frac{41}{3432} \\ -\frac{147}{572} \\ \frac{6869}{3432} \\ -\frac{542}{143} \\ \frac{25}{6} \end{bmatrix}$$

so that our "formula" for the prime numbers is

$$\frac{41 \cdot x^4}{3432} - \frac{147 \cdot x^3}{572} + \frac{6869 \cdot x^2}{3432} - \frac{542 \cdot x}{143} + \frac{25}{6}$$

and figure 7.3.4 shows a comparison plot.

In the most general form of linear least squares, we approximate data via a formula

$$f(x) = \sum_{i=1}^{n} c_i g_i(x)$$

where the g_j are some functions (not necessarily powers of x), and $X_{i,j} = g_j(x_i)$. In our examples above, the $g_j(x)$ were x^j.

EXERCISES.

4. Find a fifth degree polynomial that least-squares approximates the first 20 prime numbers, which are

$$2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71$$

7.4. Eigenvalues and the characteristic polynomial

Suppose V is a vector space and $A: V \to V$ is a linear transformation. Recall how eigenvalues and eigenvectors are defined in terms of each other

$$Av = \lambda v \tag{7.4.1}$$

where we require $v \neq 0$ and λ to be a *scalar*. A nonzero vector, v, satisfying this equation is called an *eigenvector* of A and the value of λ that makes this work is called the corresponding *eigenvalue*.

Eigenvectors and eigenvalues are defined in terms of each other, but eigenvalues are computed first.

We rewrite equation 7.4.1 as

$$Av = \lambda Iv$$

where I is the suitable identity matrix and get

$$(A - \lambda I)v = 0$$

This must have solutions for *nonzero* vectors, v. This can *only* happen if

$$\det(A - \lambda I) = 0$$

DEFINITION 7.4.1. If A is an $n \times n$ matrix

$$\det(\lambda I - A) = \chi_A(\lambda)$$

is a degree-n polynomial called the *characteristic polynomial* of A. Its roots are the *eigenvalues* of A. Maxima has a **charpoly**-command

EXAMPLE 7.4.2. If type

```
b:matrix([1,2,3,4],[5,6,7,8],[7,8,9,10],[11,12,13,14]);
```

to get

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 \end{bmatrix}$$

Its characteristic polynomial is computed by the **charpoly**-command

```
charpoly(b,x)
```

The first parameter is the matrix, and the second is the variable to appear in the polynomial. This generally gives a messy output expression that can be simplified by the **expand**-command to get

$$\chi_b(x) = x^4 - 30x^3 - 64x^2$$

with roots (*eigenvalues* of b) that can be computed directly (i.e., without first issuing the **charpoly**-command) by the **eigenvalues**-command which has a shorter abbreviation, **eivals**

```
eigenvalues(b)
```

to get

$$[[-2,32,0],[1,1,2]]$$

These two lists give, respectively, the eigenvalues themselves, and their corresponding multiplicities. So our eigenvalues are $0,-2,32$ with multiplicities $2,1,1$, respectively.

The eigen*vectors* are computed by the **eigenvectors**-command which finds the eigenvalues and then the corresponding eigenvectors[3]

$$[[[-2,32,0],[1,1,2]],\left[\left[\left[1,\frac{3}{11},-\frac{1}{11},-\frac{9}{11}\right]\right],\left[\left[1,\frac{7}{3},3,\frac{13}{3}\right]\right],[[1,0,-3,2],[0,1,-2,1]]\right]$$

This output consists of a list of

(1) Eigenvalues and multiplicities (identical to the output of the **eigenvalues**-command),
(2) For each eigenvalue, a list of the corresponding eigenvectors (there might be more than one). In this example, the eigenvalue 0 has *two* linearly independent eigenvectors. These vectors are listed as *row*-vectors rather than the *column*-vectors used in section 7.2 on page 114.

We will try to do what was done in section 7.2 on page 114 using the matrix b. We cut and paste the eigenvectors computed earlier into a matrix

First, we type

```
pt:matrix([1,3/11,-1/11,-9/11],[1,7/3,3,13/3],
[1,0,-3,2],[0,1,-2,1]);
```

to get

$$\begin{bmatrix}1 & \frac{3}{11} & -\frac{1}{11} & -\frac{9}{11}\\ 1 & \frac{7}{3} & 3 & \frac{13}{3}\\ 1 & 0 & -3 & 2\\ 0 & 1 & -2 & 1\end{bmatrix}$$

Since this has *row* vectors rather than the *column* vectors we want, we use the **transpose**-command

```
p:transpose(pt)
```

[3]It isn't necessary to issue the **charpoly** command or the **eigenvalues** command first.

to get

$$p = \begin{bmatrix} 1 & 1 & 1 & 0 \\ \frac{3}{11} & \frac{7}{3} & 0 & 1 \\ -\frac{1}{11} & 3 & -3 & -2 \\ -\frac{9}{11} & \frac{13}{3} & 2 & 1 \end{bmatrix}$$

Now we are ready to play!

```
d:(p^^-1).b.p
```

gives

$$d = \begin{bmatrix} -2 & 0 & 0 & 0 \\ 0 & 32 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

so the matrix b simply multiplies the first basis-vector by -2, the second by 32, and kills the remaining basis vectors.

The expression

```
p.d.(p^^-1)
```

recovers our original b-matrix.

Since diagonal matrices behave like scalars

$$d^n = \begin{bmatrix} (-2)^n & 0 & 0 & 0 \\ 0 & 32^n & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

for $n > 0$. We can get a *closed form* of the n^{th} power of b by writing

```
p.d^n.(p^^-1)
```

(here, we're using the fact that a *single* carat simply raises each *element* of the matrix to a power) to get a *closed form* expression for b^n:

$$\begin{bmatrix} \frac{11\cdot(-2)^n}{17}+\frac{39\cdot 32^{n-1}}{17} & \frac{45\cdot 32^{n-1}}{17}-\frac{11\cdot(-2)^{n-1}}{17} & 3\cdot 32^{n-1} & \frac{57\cdot 32^{n-1}}{17}+\frac{11\cdot(-2)^{n-1}}{17} \\ \frac{3\cdot(-2)^n}{17}+\frac{91\cdot 32^{n-1}}{17} & \frac{105\cdot 32^{n-1}}{17}-\frac{3\cdot(-2)^{n-1}}{17} & 7\cdot 32^{n-1} & \frac{133\cdot 32^{n-1}}{17}+\frac{3\cdot(-2)^{n-1}}{17} \\ \frac{117\cdot 32^{n-1}}{17}-\frac{(-2)^n}{17} & \frac{135\cdot 32^{n-1}}{17}+\frac{(-2)^{n-1}}{17} & 9\cdot 32^{n-1} & \frac{171\cdot 32^{n-1}}{17}-\frac{(-2)^{n-1}}{17} \\ \frac{169\cdot 32^{n-1}}{17}-\frac{9\cdot(-2)^n}{17} & \frac{195\cdot 32^{n-1}}{17}+\frac{9\cdot(-2)^{n-1}}{17} & 13\cdot 32^{n-1} & \frac{247\cdot 32^{n-1}}{17}-\frac{9\cdot(-2)^{n-1}}{17} \end{bmatrix}$$

which we can **ratsimp** to

$$\begin{bmatrix} \frac{39\cdot 32^n-11\cdot(-2)^{n+5}}{544} & \frac{11\cdot(-2)^{n+4}+45\cdot 32^n}{544} & 3\cdot 32^{n-1} & \frac{57\cdot 32^n-11\cdot(-2)^{n+4}}{544} \\ \frac{91\cdot 32^n-3\cdot(-2)^{n+5}}{544} & \frac{3\cdot(-2)^{n+4}+105\cdot 32^n}{544} & 7\cdot 32^{n-1} & \frac{133\cdot 32^n-3\cdot(-2)^{n+4}}{544} \\ \frac{(-2)^{n+5}+117\cdot 32^n}{544} & \frac{135\cdot 32^n-(-2)^{n+4}}{544} & 9\cdot 32^{n-1} & \frac{(-2)^{n+4}+171\cdot 32^n}{544} \\ \frac{9\cdot(-2)^{n+5}+169\cdot 32^n}{544} & \frac{195\cdot 32^n-9\cdot(-2)^{n+4}}{544} & 13\cdot 32^{n-1} & \frac{9\cdot(-2)^{n+4}+247\cdot 32^n}{544} \end{bmatrix}$$

The reader might wonder why we are interested in eigenvalues and eigenvectors. The answer is simple:

Equation 7.4.1 on page 124 shows that A behaves like a *scalar* when it acts on an eigenvector.

If we could find a basis for our vector space of *eigenvectors, A* would become a *diagonal matrix* in that basis — because it merely multiplies each basis-vector by a scalar.

EXAMPLE 7.4.3. It is also possible for eigenvectors to *not* span a vector space. Consider the matrix

$$B = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$$

This has a single eigenvalue, $\lambda = 1$, and its eigenspace is one-dimensional, spanned by

$$\begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

so there *doesn't exist* a basis of $\mathbb{R}^2$ of eigenvectors of B. All matrices (even those like B above) have a standardized form that is "almost" diagonal called Jordan Canonical Form.

We conclude this section with (see [58, chapter 6] for a proof):

THEOREM 7.4.4 (Cayley-Hamilton). *If A is an $n \times n$ matrix with characteristic polynomial*

$$\chi_A(\lambda)$$

then $\chi_A(A) = 0$.

REMARK. In other words, every matrix "satisfies" its characteristic polynomial.

The Cayley-Hamilton Theorem can be useful in computing *powers* of a matrix. For instance, if the characteristic polynomial of a matrix, A, is $\lambda^2 - 5\lambda + 3$, we know that

$$A^2 = 5A - 3I$$

so *all* integer powers of A will be linear combinations of A and I. Since A is invertible

$$A = 5I - 3A^{-1}$$

or

$$A^{-1} = \frac{1}{3}(5I - A)$$

This can also be used to calculate other functions of a matrix. If

$$f(X)$$

is a high-order polynomial or even an infinite series, write

$$f(X) = \chi_A(X) \cdot g(X) + r(X)$$

where $r(X)$ is the remainder with $\deg r(X) < \deg \chi_A(X)$ and

$$f(A) = r(A)$$

This also one of the reasons we have a **charpoly**-command in Maxima: sometimes the characteristic polynomial is useful in its own right and not only as a way to compute eigenvalues.

Sir William Rowan Hamilton, (1805 – 1865) was an Irish physicist, astronomer, and mathematician who made major contributions to mathematical physics (some had applications to quantum mechanics), optics, and algebra. He invented quaternions, a generalization of the complex numbers (see [58, chapter 9]).

EXERCISES.

1. If
$$A = \begin{bmatrix} -9 & 2 & -3 \\ 8 & 1 & 2 \\ 44 & -8 & 14 \end{bmatrix}$$
compute a closed form expression for A^n.

2. Compute a *square root* of the matrix, A.

3. Generate a 10×10 matrix whose entries are the row minus the column. Find its eigenvalues.

4. Consider two sequences recursively defined by
$$a_{n+1} = 2a_n + 2b_n$$
$$b_{n+1} = ka_n + 7b_n$$
with $a_0 = 1$ and $b_0 = 0$. How do you choose the real number, k, so that
$$\lim_{n\to\infty} \frac{a_{n+1}}{a_n} = 5$$
? In this case, find
$$\lim_{n\to\infty} \frac{b_{n+1}}{a_n}$$

7.4.1. Population dynamics. We will use linear algebra to study age-distributions of populations. Suppose we have a population of organisms that has a maximum lifespan of k years (or units of time that might not be years). Let $n_i \geq 0$ denote the number of creatures alive at year i, let $0 \leq s_i \leq 1$ be the fraction of creatures that survive from year i to year $i+1$, and let $0 \leq f_i$ be the average number of offspring creatures at age i have.

We have what is called the Leslie Matrix, describing the dynamics of this system (see [41])

$$\begin{bmatrix} n_0 \\ \vdots \\ n_{k-1} \end{bmatrix}_{t+1} = \begin{bmatrix} f_0 & \cdots & \cdots & \cdots & f_{k-1} \\ s_0 & 0 & \cdots & \ddots & 0 \\ 0 & s_1 & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & s_{k-2} & 0 \end{bmatrix} \begin{bmatrix} n_0 \\ \vdots \\ n_{k-1} \end{bmatrix}_t$$

or

$$\mathbf{N}_{t+1} = \mathbf{T}\mathbf{N}_t$$

We are interested in

(1) Ratios between the various age groups, when the population stabilizes,
(2) How fast it grows after this stabilization.

The population's age-ratios will have stabilized when

$$\mathbf{N}_{t+1} = \lambda \mathbf{N}_t$$

for some scalar, λ — i.e., when $\mathbf{N}_t$ is an *eigenvector* of $\mathbf{T}$. The corresponding eigenvalue, λ, shows how rapidly the overall population grows or shrinks.

Let's do an example.

EXAMPLE 7.4.5. A study of tribbles on planet ceti-alpha-6 shows that their Leslie Matrix is

$$T = \begin{bmatrix} 0 & 1 & 3 & 1 & 0 \\ 0.9 & 0 & 0 & 0 & 0 \\ 0 & 0.6 & 0 & 0 & 0 \\ 0 & 0 & 0.5 & 0 & 0 \\ 0 & 0 & 0 & 0.3 & 0 \end{bmatrix}$$

We dutifully type

```
eigenvalues(T)
```

and get a lengthy string of square roots and other radicals. Typing **bfloat**(%) gives

```
[[(-7.693353249757961b-1 %i) -6.381300992460435b-1,
7.693353249757961b-1 %i - 6.381300992460435b-1,
-1.84947536902627b-1, 1.461207735394714b0, 0.0b0],
[1.0b0, 1.0b0, 1.0b0, 1.0b0, 1.0b0]]
```

If we ask for eigen*vectors,* Maxima pauses for a long time and finally reports that there aren't any!

What has gone wrong?

Consider the eigenvalue 1.461207735394714.

The nullspace of

```
U:T-1.461207735394714*ident(5)
```

should be the eigenvector associated with this eigenvalue. If we type

```
nullspace(U)
```

Maxima comes back with

```
[?]
```

What has happened? *Is Maxima broken?* For U to have a nullspace, its determinant must vanish. The command

```
determinant(U)
```

gives

$$1.110223024625157 \cdot 10^{-16} \tag{7.4.2}$$

which is very small but *not* zero. This illustrates the strength and weakness of Maxima: it insists on *exact* arithmetic. The number 1.461207735394714 is very close to an eigenvalue but not *exactly equal* to one[4]. We need a linear algebra system that can work with non-exact arithmetic.

Luckily, there's a library of linear algebra routines that will treat numbers like that in equation 7.4.2 as *zero.* We load it with the command

```
load("lapack")
```

One command that is available is

```
dgeev(t,u,f);
```

where t is a square matrix of *real* numbers and u and f are truth-values (they default to '**false**' if omitted). The output is the set of eigenvalues and their multiplicities. If u is not **false**, it outputs an array of *right-eigenvectors*, i.e.

$$tv = \lambda v$$

If f is not **false**, it also outputs *left eigenvectors*

$$v^H t = \lambda v^H$$

where v^H is the conjugate-transpose of v. All we want are right eigenvectors, so we type

```
dgeev(t,true,false);
```

[4] It can't possibly be; the actual eigenvalue is irrational.

and get a list of eigenvalues

$$\begin{gathered} 0 \\ 1.461207735394714 \\ 0.7693353249757966 \cdot i - 0.6381300992460446 \\ -0.7693353249757966 \cdot i - 0.6381300992460446 \\ -0.1849475369026271 \end{gathered}$$

and a huge 5×5 array whose columns are corresponding eigenvectors.

The only positive real eigenvalue is 1.461207735394714, so we pick that. Its corresponding eigenvector is the second column of the eigenvector-matrix:

$$\begin{bmatrix} 0.8301290523484829 \\ 0.5113004325232492 \\ 0.2099497915887227 \\ 0.07184118537806998 \\ 0.01474968622965787 \end{bmatrix}$$

We normalize this to sum up to 1 (how is this done?) and get

$$\begin{bmatrix} 0.5068035295561001 \\ 0.3121549151101902 \\ 0.1281768119134142 \\ 0.04385988686228456 \\ 0.009004856558010907 \end{bmatrix} \tag{7.4.3}$$

so this is the population-distribution of tribbles when it stabilizes.

In each time unit, the population is multiplied by the eigenvalue 1.461207735394714, so tribbles multiply rapidly[5].

In general, there's nothing wrong with Maxima's exact computations (and a lot that is right). In dealing with approximate data and purely numeric computations, the lapack library may be advantageous.

That library contains a number of other routines:

dgeqrf(A) Computes the QR decomposition of the matrix A: $A = QR$, where Q is a square orthonormal[6] matrix with the same number of rows as A and R is an upper-triangular matrix.

dgesv(A,b) Solves the linear algebra problem $Ax = b$, where A and b are made up of real, floating point numbers.

▷ **dgemm**(A,B) **dgemm** (A, B, options) Compute the product of two matrices and optionally add the product to a third matrix.

---

[5] As anyone who has seen the old Star Trek episode, *The Trouble with Tribbles*, knows!

[6] If C and C' are any two *distinct* columns of the matrix, then $C \cdot C = 1$ and $C \cdot C' = 0$.

In the simplest form, **dgemm**(A, B) computes the product of the two real matrices, A and B.

In the second form, **dgemm** computes the alpha * A * B + beta * C where A, B, C are real matrices of the appropriate sizes and alpha and beta are real numbers. Optionally, A and/or B can be transposed before computing the product. The extra parameters are specified by optional keyword arguments: The keyword arguments are optional and may be specified in any order. They all take the form key=val. The keyword arguments are:

C The matrix C that should be added. The default is false, which means no matrix is added.

alpha The product of A and B is multiplied by this value. The default is 1.

beta If a matrix C is given, this value multiplies C before it is added. The default value is 0, which implies that C is not added, even if C is given. Hence, be sure to specify a non-zero value for beta.

transpose_a If true, the transpose of A is used instead of A for the product. The default is false.

transpose_b If true, the transpose of B is used instead of B for the product. The default is false.

zgeev(A) **zgeev** (A, right_p, left_p) Like **dgeev**, but the matrix A is *complex.*

zheev(A) **zheev**(A, eigvec_p) Like **zgeev**, but the matrix A is assumed to be a square complex Hermitian matrix. If **eigvec_p** is true, then the eigenvectors of the matrix are also computed.

No check is made that the matrix A is, in fact, Hermitian.

A list of two items is returned, as in **dgeev**: a list of eigenvalues, and false or the matrix of the eigenvectors.

EXERCISES.

5. Write a function that takes a vector (representing the initial number of tribbles available in each age-group) and an integer n that returns the population-distribution of tribbles in time-unit n. How fast does this converge to the stable distribution in equation 7.4.3 on page 131? Is there an initial distribution that *doesn't* converge to the stable distribution?

7.4.2. The 25-billion-dollar eigenvector[7]. Although several search engines predated Google (Yahoo, etc.), Google distinguished itself by the *quality* of its results. It seemed to find the most relevant web pages so one did not have to wade through countless links to find interesting information. This largely due to *Google's Page rank algorithm*, which manages to pick out these web pages.

This raises a question:

> How can one determine relevance in *any* search? Doesn't it depend on the subject matter?

Google solved this problem by counting the number of pages that *link* to a given web page, i.e. the number of *back-links* the page possesses: the more back-links, the more people are interested in the page — *regardless* of the subject-matter. See the groundbreaking paper [10].

Consider the web in figure 7.4.1 on the following page.

If x_i is the importance of node i, then back-link-counts gives us

- $\triangleright$ $x_1 = x_9 = x_{10} = 0$
- $\triangleright$ $x_2 = x_4 = x_5 = x_6 = x_7 = 1$
- $\triangleright$ $x_8 = 4$
- $\triangleright$ $x_3 = 6$

so the most important node is 3.

On the other hand, we should take into account the *importance* of a page that links to another so we get equations like

- $\triangleright$ $x_3 = x_2 + x_4 + x_9 + x_{10}$
- $\triangleright$ $x_2 = x_1$, $x_4 = x_1$
- $\triangleright$ etc.

Another consideration is the number of outgoing links a page has: the more of these, the greater the page's influence on the whole process. We remedy that by normalizing the effect of the links: each link of a page that has n outgoing links gets a weight of $1/n$. Every web page has the same effect on the final result.

[7]This is the approximate value of Google when the company went public in 2004. This section of the book uses material from the excellent paper, [11].

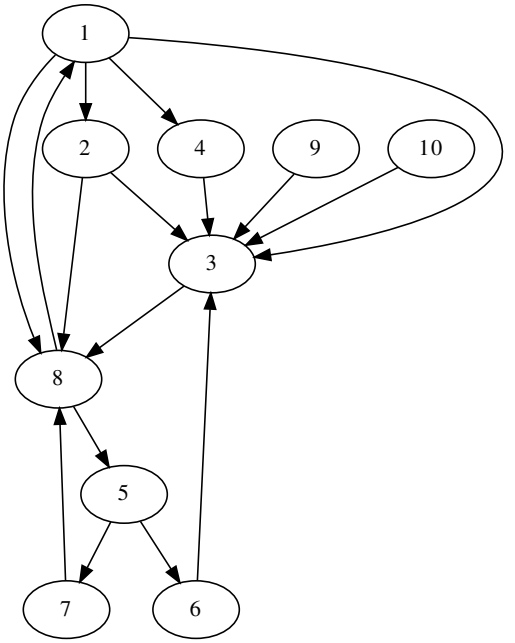


FIGURE 7.4.1. A sample web

If page k has n_k *outgoing* links, then the equations above are rewritten as

$$x_i = \sum_{k \in B_i} \frac{x_k}{n_k} \tag{7.4.4}$$

where B_i is the set of back-links of page i. We can enter this data into an $n \times n$ matrix, L, called the *link matrix*, where

$$L_{i,j} = \begin{cases} 0 & \text{if there's no link from page } j \text{ to page } i \\ 1/n_j & \text{if there's a link from page } j \text{ to page } i \end{cases}$$

For instance, the graph in figure 7.4.1 has a link matrix of

$$L = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & .5 & 0 & 0 \\ .25 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ .25 & .5 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 \\ .25 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & .5 & 0 & 0 \\ 0 & 0 & 0 & 0 & .5 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & .5 & 0 & 0 & 0 & 0 & 0 \\ .25 & .5 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{7.4.5}$$

and if $\mathbf{X}$ is a 10×1 matrix (i.e., vector) of the importance-values, then

$$L\mathbf{X} = \mathbf{X}$$

i.e., plugging the values into the *right* side of equations 7.4.4 on the facing page should give the importance-values back. In other words, X is an *eigenvector* of L with eigen*value* 1. The matrix, L has two properties

(1) All of its entries are nonnegative,
(2) Its columns all sum up to 1.

These properties make it what is called a *column-stochastic* matrix. This turns out to guarantee that it has 1 as an eigenvalue.

Typing

```
eigenvalues(L);
```

gives an incredibly messy answer, but shows that L does have an eigenvalue of 1. Typing

```
eigenvectors(L);
```

causes Maxima to complain that it cannot compute the first four eigenvectors[8], but the one corresponding to the eigenvalue 1 is

$$[0, 0, 0, 1, 0, 0, 0, 0, 0, 0]^t$$

showing that the most important page is number 4, a surprising result!

If we

```
load("lapack")
```

and type

```
dgeev(L,true,false);
```

we get a similar *numeric* result.

In the example above, the eigenspace corresponding to the eigenvalue 1 was one-dimensional, so we got a unique ranking. Suppose we had r disjoint *sub-webs* of our original web. In this case, our link-matrix would look like

$$L = \begin{bmatrix} \mathbf{D}_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \mathbf{D}_r \end{bmatrix}$$

where $\mathbf{D}_i$ is the link-matrix of the i^{th} sub-web. Each of these sub-webs will have a ranking independent of the others so that the eigenspace of L corresponding to the eigenvalue 1 will be r-dimensional. If we compute this eigenspace we get r vectors that span it, not necessarily the rank vectors of the sub-webs (which will be some linear combination of those r vectors).

[8] Because it's trying to find *exact* values, solving degree-10 polynomials.

Google solved this problem by *"slightly connecting"* the sub-webs together. Let the total number of web-pages be n and let S be an $n \times n$ matrices whose entries are all $1/n$. Then form

$$M = (1-c)L + cS \tag{7.4.6}$$

where c is some small number. Google initially used $c = .15$. The resulting M matrix will still be column-stochastic but will have a one-dimensional eigenspace for the eigenvalue 1.

Although Google has enormous computing power at its disposal, it is certainly unable to process billion-by-billion matrices in the usual fashion (in finding eigenvalues and eigenvectors, for instance). Google uses the *power method* to compute the eigenvector corresponding to the eigenvalue 1.

Very crudely (!), the idea is

$$MM^{\infty}\mathbf{X} = M^{\infty+1}\mathbf{X} = M^{\infty}\mathbf{X}$$

where $\mathbf{X}$ is a "typical" nonzero vector. We start with some nonzero vector $\mathbf{X}_0$ and define

$$\mathbf{X}_{k+1} = M\mathbf{X}_k$$

and *hope* this converges to the actual eigenvector, $\bar{\mathbf{X}}$ as $k \to \infty$. We also *normalize* $\mathbf{X}_{k+1}$ so it doesn't grow or shrink in this process: If $\mathbf{v}$ is an n-dimensional vector with components $\{v_i\}$, define

$$\|\mathbf{v}\|_1 = \sum_{i=1}^{n} |v_i|$$

and our iterative process is

$$\mathbf{X}_{k+1} = \frac{M\mathbf{X}_k}{\|M\mathbf{X}_k\|_1} \tag{7.4.7}$$

See [61] for an analysis of when this type of process converges.

Here's an *interpretation* of taking powers of the original link matrix, L:

$$\left(L^2\right)_{i,j} > 0$$

if and only if there a path of length ≤ 2 that connects node j to node i. Why?

$$\left(L^2\right)_{i,j} = \sum_{k=1}^{n} L_{i,k}L_{k,j}$$

and this is > 0 if both $L_{i,k}$ and $L_{k,j}$ are > 0 for at least one value of k. A simple induction shows that

$$\left(L^r\right)_{i,j} > 0$$

if and only if there's a path of length $\leq r$ connecting j to i. So:

Taking powers of the L-matrix is similar to randomly surfing the web and counting how many times we reach each page. The more often we reach a page, the more important that page is. The initial vector $\mathbf{X}_0$ represents the starting position of this random walk.

EXERCISES.

6. Write a Maxima function to implement the iteration used in the power method (equation 7.4.7 on the preceding page). Test it using the link matrix, L, in equation 7.4.5 on page 134 converted to M via equation 7.4.6 on the preceding page with $c = .15$. Does it converge?

Compare this with the actual eigenvector, computed via **dgeev**.

7.5. Functions of matrices

If M is a matrix, we can plug it into power series to compute functions like e^M or $\sin M$ and $\cos M$.

If a matrix can be diagonalized, it's easy to compute functions like these. Consider the matrix

$$E = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 \end{bmatrix}$$

from section 7.4 on page 124. We know that E can be diagonalized to

$$d = \begin{bmatrix} -2 & 0 & 0 & 0 \\ 0 & 32 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

where E=p.d.(p^^-1) and

$$p = \begin{bmatrix} 1 & 1 & 1 & 0 \\ \frac{3}{11} & \frac{7}{3} & 0 & 1 \\ -\frac{1}{11} & 3 & -3 & -2 \\ -\frac{9}{11} & \frac{13}{3} & 2 & 1 \end{bmatrix}$$

Then

$$e^d = \begin{bmatrix} e^{-2} & 0 & 0 & 0 \\ 0 & e^{32} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$e^E = p.e^d.p\text{^^-1}$$

giving

$$\begin{bmatrix} \frac{e^{-2}\left(39e^{34}+153e^2+352\right)}{544} & \frac{e^{-2}\left(45e^{34}-221e^2+176\right)}{544} & \frac{3e^{32}-3}{32} & \frac{e^{-2}\left(57e^{34}+119e^2-176\right)}{544} \\ \frac{e^{-2}\left(91e^{34}-187e^2+96\right)}{544} & \frac{e^{-2}\left(105e^{34}+391e^2+48\right)}{544} & \frac{7e^{32}-7}{32} & \frac{e^{-2}\left(133e^{34}-85e^2-48\right)}{544} \\ \frac{e^{-2}\left(117e^{34}-85e^2-32\right)}{544} & \frac{e^{-2}\left(135e^{34}-119e^2-16\right)}{544} & \frac{9e^{32}+23}{32} & \frac{e^{-2}\left(171e^{34}-187e^2+16\right)}{544} \\ \frac{e^{-2}\left(169e^{34}+119e^2-288\right)}{544} & \frac{e^{-2}\left(195e^{34}-51e^2-144\right)}{544} & \frac{13e^{32}-13}{32} & \frac{e^{-2}\left(247e^{34}+153e^2+144\right)}{544} \end{bmatrix}$$

The exponential is so important that Maxima has a command to produce it, namely the **matrixexp**-command.

Its format is

```
matrixexp(thematrix, optional scalar multiplier);
```

If the scalar multiplier is omitted, it is assumed to be 1. The reason for this multiplier will become clear momentarily.

Suppose we have a system of differential equations

$$\frac{dx}{dt} = 3x - 4y$$
$$\frac{dy}{dt} = 2x - y$$

written more compactly as

$$\frac{dv}{dt} = Mv$$

where

$$v = \begin{bmatrix} x \\ y \end{bmatrix}$$

and

$$M = \begin{bmatrix} 3 & -4 \\ 2 & -1 \end{bmatrix}$$

The well-known solution is

$$\begin{bmatrix} x \\ y \end{bmatrix} = e^{t\cdot M} \cdot \begin{bmatrix} x(0) \\ y(0) \end{bmatrix}$$

If you differentiate the power-series for $e^{t\cdot M}$ with respect to t, you get $Me^{t\cdot M}$. Typing

```
m: matrix([3,-4],[2,-1])
matrixexp(m, t);
```

results in

$$\begin{bmatrix} -\frac{\%e^{-2\cdot\%i\cdot t}\cdot\left((\%i-1)\cdot\%e^{4\cdot\%i\cdot t+t}+(-\%i-1)\cdot\%e^{t}\right)}{2} & \%e^{-2\cdot\%i\cdot t}\cdot\left(\%i\cdot\%e^{4\cdot\%i\cdot t+t}-\%i\cdot\%e^{t}\right) \\ -\frac{\%e^{-2\cdot\%i\cdot t}\cdot\left(\%i\cdot\%e^{4\cdot\%i\cdot t+t}-\%i\cdot\%e^{t}\right)}{2} & \frac{\%e^{-2\cdot\%i\cdot t}\cdot\left((\%i+1)\cdot\%e^{4\cdot\%i\cdot t+t}+(1-\%i)\cdot\%e^{t}\right)}{2} \end{bmatrix}$$

Since the original problem didn't include any imaginary numbers, it's safe to assume we should convert these imaginary exponentials to sines and cosines.

The **demoivre**-command replaces all occurrences of e^{ix} by $\cos(x) + i\sin(x)$:

```
demoivre(%)
```

but applying it as once might not be the best approach. A few experimental sequences of commands shows that

```
expand(%);
demoivre(%);
trigsimp(%);
```

provides the simplest result:

$$\begin{bmatrix} \%e^t \cdot \sin(2 \cdot t) + \%e^t \cdot \cos(2 \cdot t) & -2 \cdot \%e^t \cdot \sin(2 \cdot t) \\ \%e^t \cdot \sin(2 \cdot t) & \%e^t \cdot \cos(2 \cdot t) - \%e^t \cdot \sin(2 \cdot t) \end{bmatrix}$$

Here, the **trigsimp**-command can be used to simplify some trigonometric expressions. Alternatively, we could have applied the **realpart**-command to the matrix.

Other trigonometric commands are **trigreduce** and **trigrat**.

Note: the opposite of **demoivre** is the **exponentialize**-command which converts trigonometric functions into their exponential form.

EXERCISES.

1. Find the sine and cosine of the matrix

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

Hint: use De Moivre's formula and use the **realpart** and **imagpart** commands.

2. If x is a scalar variable, compute

$$x \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

7.5.1. Affine groups and motions in space. In this section we will analyze groups that originate in geometry — groups of symmetries and motions.

To understand the geometry of $\mathbb{R}^n$, it is not enough to simply be able to rotate space about a fixed point (namely, the origin — which matrix-operations do). We must also be able to *move* objects through space, to *displace* them. This leads to the *affine groups.*

Regard $\mathbb{R}^n$ as the plane $x_{n+1} = 1$ in $\mathbb{R}^{n+1}$. An $(n+1) \times (n+1)$ matrix of the form

$$D(a_1, \dots, a_n) = \begin{bmatrix} 1 & 0 & \cdots & 0 & a_1 \\ 0 & 1 & \ddots & \vdots & \vdots \\ \vdots & \ddots & \ddots & 0 & a_{n-1} \\ \vdots & \ddots & 0 & 1 & a_n \\ 0 & \cdots & \cdots & 0 & 1 \end{bmatrix} \tag{7.5.1}$$

preserves this imbedded copy of $\mathbb{R}^n$ and displaces it so that the origin is moved to the point $(a_1, \dots, a_n)$. A simple calculation shows that

$$D(a_1, \dots, a_n) \cdot D(b_1, \dots, b_n) = D(a_1 + b_1, \dots, a_n + b_n)$$

which implies that the matrices of the form $D(a_1, \dots, a_n) \in \mathrm{GL}(n, \mathbb{R})$ form a subgroup, $S \subset \mathrm{GL}(n+1, \mathbb{R})$ *isomorphic* to $\mathbb{R}^n$.

DEFINITION 7.5.1. If $n > 0$ is an integer and $G \subset \mathrm{GL}(n, \mathbb{F})$ is a subgroup, the subgroup of $\mathrm{GL}(n+1, \mathbb{F})$ generated by matrices

$$M = \begin{bmatrix} g & 0 \\ 0 & 1 \end{bmatrix}$$

for $g \in G$ and matrices of the form $D(a_1, \dots, a_n)$ with the $a_i \in \mathbb{F}$ is called the *affine group* associated to G and denoted $\mathrm{Aff}(G)$.

Recall that

DEFINITION 7.5.2. An $n \times n$ matrix, A will be called *orthogonal* if

$$AA^t = I$$

REMARK. Recall the properties of orthogonal matrices (as always, see [58])

(1) They form a group, $O(n)$,called the orthogonal group,
(2) if A is an orthogonal matrix, then $\det A = \pm 1$. If $\det A = 1$, A will be called *special orthogonal*.
(3) An orthogonal matrix, A, as a linear transformation $A\colon \mathbb{R}^n \to \mathbb{R}^n$ preserves *angles* between vectors and *lengths* of vectors. In other words, the matrices in $O(n)$ represent rotations and reflections. The group $\mathrm{SO}(n, \mathbb{R})$ eliminates the reflections.

Recall that if we want to represent rotation in $\mathbb{R}^2$ via an angle of θ in the counterclockwise direction, we can use a matrix

$$\begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} : \mathbb{R}^2 \to \mathbb{R}^2$$

Regard $\mathbb{R}^2$ as the subspace, $z = 1$, of $\mathbb{R}^3$. The linear transformation

$$f = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & a \\ \sin(\theta) & \cos(\theta) & b \\ 0 & 0 & 1 \end{bmatrix} : \mathbb{R}^3 \to \mathbb{R}^3 \tag{7.5.2}$$

in $\mathrm{Aff}(\mathrm{SO}(2, \mathbb{R}))$ sends

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \in \mathbb{R}^2 \subset \mathbb{R}^3$$

to

$$\begin{bmatrix} x\cos(\theta) - y\sin(\theta) + a \\ x\sin(\theta) + y\cos(\theta) + b \\ 1 \end{bmatrix} \in \mathbb{R}^2 \subset \mathbb{R}^3$$

and represents

(1) *rotation* by θ (in a counterclockwise direction), followed by
(2) *displacement* by (a, b).

Affine group-actions are used heavily in computer graphics: creating a scene in $\mathbb{R}^3$ is done by creating objects at the *origin* of $\mathbb{R}^3 \subset \mathbb{R}^4$ and moving them into position (and rotating them) via linear transformations in $\mathbb{R}^4$. A high-end (and not so high end) computer graphics card performs millions of affine group-operations per second.

7.6. Linear Programming

7.6.1. Introduction. We will discuss an application of linear algebra to a widely used industrial function.

Here's an example of a linear programming problem:

EXAMPLE 7.6.1. A calculator company produces a scientific calculator and a graphing calculator. Long-term projections indicate an expected demand of at least 100 scientific and 80 graphing calculators each day.

(1) Because of limitations on production capacity, no more than 200 scientific and 170 graphing calculators can be made daily.
(2) To satisfy a shipping contract, a total of at least 200 calculators much be shipped each day.
(3) If each scientific calculator sold results in a \$2 loss, but each graphing calculator produces a \$5 profit, how many of each type should be made daily to maximize net profits?

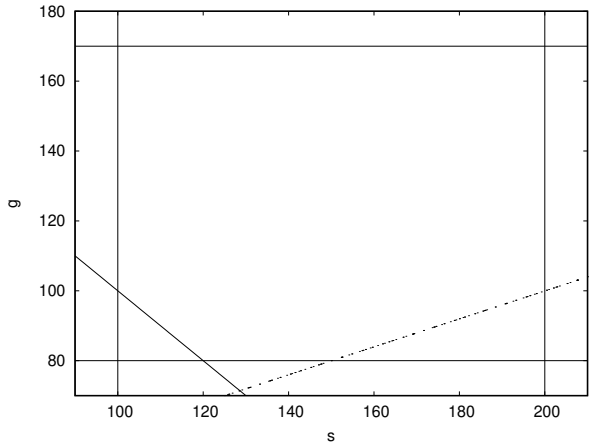


FIGURE 7.6.1. Feasible region

Let s denote the number of scientific calculators and g the number of graphing calculators produced each day.

The word problem translates into a series of inequalities

$$\begin{aligned} s + g &\geq 200 \\ s &\leq 200 \\ g &\leq 170 \\ s &\geq 100 \\ g &\geq 80 \end{aligned}$$

And we must maximize $P = -2s + 5g$, which is called the *objective function* of the problem. The inequalities define a polygonal region with 5 vertices as in figure on the current page. Points that satisfy the inequalities are called *feasible solutions*, and the set of them is called the *feasible region*. They don't necessarily solve the problem (maximize profits) but are *potential solutions.*

The line on the lower right of figure on this page is the profit-line, $P = -2s + 5g = 100$. It may be translated parallel to itself, and we arrive at the solution to the problem in figure on the next page, with a profit of 650.

Several things stand out:

The objective function is *linear,* so we can't find its extrema by setting its derivatives to 0. To see what happens, consider the function $f(x) = 2x - 1$ on the closed interval $[1, 5]$. Its derivative never vanishes, but its maximum occurs at $x = 5$. In greater generality, the extrema occur at critical points (where derivatives vanish) *or* on the *boundary* of the region.

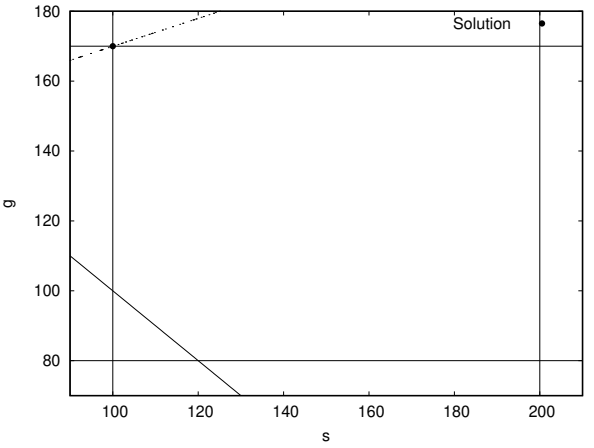


FIGURE 7.6.2. Linear programming solution

Induction shows that the extrema of an objective function occur at the *vertices* (i.e., boundary of the boundary of the...) of the feasible region.

So we could solve linear programming problems by:

(1) Computing all the vertices, using linear algebra.
(2) Plugging the objective function into each of these vertices.

Unfortunately, an n-dimensional cube has 2^n vertices and other constraints could easily increase this number. This exponential complexity can overwhelm the fastest computers; some industrial problems have $n > 500$.

In the mid 1940's, George Dantzig invented the *simplex algorithm* for solving linear programming problems. It attempts to minimize the number of vertices computed and move toward the one that solves the optimization problem.

> George Bernard Dantzig (1914 – 2005) was an American mathematical scientist who made contributions to industrial engineering, operations research, computer science, economics, and statistics.

We will not go into the simplex method's details; the interested reader is referred to [22]. Maxima has a library implementing it invoked by

```
load("simplex")
```

It's fairly easy to use. We have a command:

```
maximize_lp (objective, conditions, [pos])
```

The optional argument [pos] is a list of variables that are assumed to be positive (putting them in this list turns out to be more efficient that simply defining them to be > 0 in the list of conditions).

For instance, the sample problem at the beginning of this section could be solved by

```
maximize_lp (5*g-2*s, [s<=200,s>=100,
  g<=170,g>=80,s+g>=200])
```

to produce

$$[650, [g = 170, s = 100]]$$

We also have a

```
minimize_lp (objective, conditions, [pos])
```

command whose action is self-explanatory.

For much more complex problems (where it's hard to simply give a list of inequalities), we have the command

```
linear_program (A, b, c)
```

Here:

A is a matrix and b and c are vectors. The command computes a vector, x, that minimizes $c \cdot x$ among all the vectors with the property that $Ax = b$, and $x \geq 0$.

Here's an example:

```
A: matrix([1,1,-1,0], [2,-3,0,-1], [4,-5,0,0]);
b: [1,1,6];
c: [1,-2,0,0];
linear_program(A, b, c);
```

resulting in

$$\left[\left[\frac{13}{2}, 4, \frac{19}{2}, 0\right], -\frac{3}{2}\right]$$

where the first list gives the vector, x, and the second value is the objective function at that point.

There are two error-messages that these commands give:

- ▷ *Problem not feasible!* — in this case, the inequalities contradict each other so there are no feasible solutions. Example: $x >= 2, x <= 1$.
- ▷ *Problem not bounded!* — in this case the feasible region is unbounded and the solution is infinite. Example

  ```
  maximize_lp (x, [x>=0])
  ```

EXERCISES.

1. A manufacturer of ski clothing makes ski pants and ski jackets. The profit on a pair of ski pants is $3.00 and on a jacket is $2.00. Both pants and jackets require the work of sewing operators and cutters. There are 60 minutes of sewing operator time and 48 minutes of cutter time available. It takes 8 minutes to sew one pair of ski pants and 4 minutes to sew one jacket. Cutters take 4 minutes on pants and 8 minutes on a jacket.

Find the maximum profit and the amount of pants and jackets to maximize the profit.

2. A farmer has a field of 70 acres in which he plants potatoes and corn. The seed for potatoes costs $20/acre, the seed for corn costs $60/acre and the farmer has set aside $3000 to spend on seed. The profit per acre of potatoes is $150 and the profit for corn is $50 an acre.

Find the optimal solution for the farmer.

7.6.2. Integer programming and "industrial strength" problems. All of the problems we have considered must be regarded as "toy" problems: The number of variables and constraints are small enough to be listed in a command. In addition, we don't have constraints that require some variables to be *integers.* It turns out that last consideration makes linear programming infinitely harder. The mere simplex algorithm cannot handle it.

For these problems, we need specialized software, namely '**glpk**', developed (in the early 2000's) by the Department for Applied Informatics, Moscow Aviation Institute, Moscow, Russia. They used it for designing airplane and jet engines.

It has been released into the public domain (and is, therefore *free software)* and enhanced many times in the intervening years.

- ▷ Most Linux distributions have a packaged version of it.
- ▷ The original source code can be found at
  https://www.gnu.org/software/glpk/
- ▷ A Windows port can be found at
  https://winglpk.sourceforge.net/
- ▷ A Macintosh version can be found at
  https://ports.macports.org/port/glpk/

It implements its own (fairly simple) programming language that allows for

- ▷ *arrays* of variables, constraints, and data
- ▷ requiring some variables to be *integers* or even *binary* (0 or 1)
- ▷ reading these arrays from files

The manual that is packaged with the software is almost 200 pages, but most of it is devoted to accessing glpk from a C or C++ program (so, in particular, one can *do* that!). It can also be accessed from python programs.

We will want to run it in a standalone mode, which is relatively simpler.

Here's a sample program in the GNU MathProg modeling language:

```
# A TRANSPORTATION PROBLEM
#
# This problem finds a least cost shipping schedule that meets
# requirements at markets and supplies at factories.
#
# References:
# Dantzig G B, "Linear Programming and Extensions."
# Princeton University Press, Princeton, New Jersey, 1963,
# Chapter 3-3.


set I;
/* canning plants */

set J;
/* markets */

param a{i in I};
/* capacity of plant i in cases */

param b{j in J};
/* demand at market j in cases */

param d{i in I, j in J};
/* distance in thousands of miles */

param f;
/* freight in dollars per case per thousand miles */

param c{i in I, j in J} := f * d[i,j] / 1000;
/* transport cost in thousands of dollars per case */

var x{i in I, j in J} >= 0;
/* shipment quantities in cases */

minimize cost: sum{i in I, j in J} c[i,j] * x[i,j];
/* total transportation costs in thousands of dollars */

s.t. supply{i in I}: sum{j in J} x[i,j] <= a[i];
/* observe supply limit at plant i */

s.t. demand{j in J}: sum{i in I} x[i,j] >= b[j];
/* satisfy demand at market j */


data;
set I := Seattle San-Diego;
set J := New-York Chicago Topeka;

param a := Seattle    350
           San-Diego  600;

param b := New-York 325
           Chicago  300
           Topeka   275;

param d : New-York    Chicago     Topeka :=
Seattle       2.5         1.7         1.8
San-Diego     2.5         1.8         1.4 ;

param f := 90;

end;
```

You run this (in Linux) by putting it into a file 'transp.mod' and typing

```
glpsol -m transp.mod -o output.txt
```

Normally, this only prints the whether the program succeeded and some other information. The actual results of the program (i.e., the solution to the problem) go into the file 'output.txt', which is

```
Problem:    transp
Rows:       6
Columns:    6
Non-zeros:  18
Status:     OPTIMAL
Objective:  cost = 153.675 (MINimum)

   No.   Row name   St   Activity     Lower bound   Upper bound    Marginal
------ ------------ -- ------------- ------------- ------------- -------------
     1 cost         B        153.675
     2 supply[Seattle]
                    NU           350                         350         < eps
     3 supply[San-Diego]
                    B            550                         600
     4 demand[New-York]
                    NL           325           325                       0.225
     5 demand[Chicago]
                    NL           300           300                       0.153
     6 demand[Topeka]
                    NL           275           275                       0.126

   No. Column name  St   Activity     Lower bound   Upper bound    Marginal
------ ------------ -- ------------- ------------- ------------- -------------
     1 x[Seattle,New-York]
                    B             50             0
     2 x[Seattle,Chicago]
                    B            300             0
     3 x[Seattle,Topeka]
                    NL             0             0                     0.036
     4 x[San-Diego,New-York]
                    B            275             0
     5 x[San-Diego,Chicago]
                    NL             0             0                     0.009
     6 x[San-Diego,Topeka]
                    B            275             0

Karush-Kuhn-Tucker optimality conditions:

KKT.PE: max.abs.err = 0.00e+00 on row 0
        max.rel.err = 0.00e+00 on row 0
        High quality

KKT.PB: max.abs.err = 0.00e+00 on row 0
        max.rel.err = 0.00e+00 on row 0
        High quality

KKT.DE: max.abs.err = 0.00e+00 on column 0
        max.rel.err = 0.00e+00 on column 0
        High quality

KKT.DB: max.abs.err = 0.00e+00 on row 0
        max.rel.err = 0.00e+00 on row 0
        High quality

End of output
```

Several points become clear:

(1) The program has three types of identifiers: *sets*, *params*, and *vars*. Elements of sets can be names or numbers and are used to *index* identifiers.

Examples: A=1..10 (this is the set 1,2,3,4,5,6,7,8,9,10), B=0..1 **by** .1 (this is the set 0,.1,.2,.3,.4,.5.,6,.7,.8,.9,1; **by** is an

optional keyword), Weekdays=monday tuesday wednesday thursday friday. Sets do not need to have *names:* {1,2,7}, {a,b,1..50}.

(2) The constraints are all labeled and begin with '**s.t.**' (meaning '**subject to**', which can also be spelled out).
(3) The program is in three parts: *declarations, constraints* (including the objective function), and *data.*

The statement

```
param c{i in I, j in J} := f * d[i,j] / 1000;
```

shows how one implicitly iterates over all elements of a set without coding loop-commands (although these also exist in this language).

The statement

```
minimize cost: sum{i in I, j in J} c[i,j] * x[i,j];
/* total transportation costs in
            thousands of dollars */
```

defines the objective function and illustrates a command, '**sum**' that *combines* results of iterations over sets. Such *"combining commands"* include'**prod**', '**min**', and '**max**'.

The difference between '**var**'s and '**param**'s is that

(1) '**param**'s are assumed to be given, and
(2) '**var**'s are assumed to be initially undefined, and glpk attempts to assign values to the '**var**'s that satisfy the constraints.

Although this is a "toy" program, the data section could easily have had thousands of entries and be read from a file. If data is in a separate file, it could be named transp.dat and the program would be run via

```
glpsol -m transp.mod --data transp.dat -o output.txt
```

Data could also be read in a csv file (comma-separated values, a format spreadsheets use), or a mysql database.

The following example shows how linear programming with integer variables takes a problem out of the realm of linear algebra into that of arbitrary logic and general programming.

```
/* ZEBRA, Who Owns the Zebra? */

/* Written in GNU MathProg by Andrew Makhorin <mao@mai2.rcnet.ru> */

###########################################################################
#  The Zebra Puzzle is a well-known logic puzzle.
#
#  It is often called "Einstein's Puzzle" or "Einstein's Riddle"
#  because it is said to have been invented by Albert Einstein as a boy,
#  with the common claim that Einstein said "only 2 percent of the
#  world's population can solve". It is also sometimes attributed to
#  Lewis Carroll. However, there is no known evidence for Einstein's or
#  Carroll's authorship.
#
#  There are several versions of this puzzle. The version below is
#  quoted from the first known publication in Life International
```

```
#  magazine on December 17, 1962.
#
#   1. There are five houses.
#   2. The Englishman lives in the red house.
#   3. The Spaniard owns the dog.
#   4. Coffee is drunk in the green house.
#   5. The Ukrainian drinks tea.
#   6. The green house is immediately to the right of the ivory house.
#   7. The Old Gold smoker owns snails.
#   8. Kools are smoked in the yellow house.
#   9. Milk is drunk in the middle house.
#  10. The Norwegian lives in the first house.
#  11. The man who smokes Chesterfields lives in the house next to the
#      man with the fox.
#  12. Kools are smoked in the house next to the house where the horse
#      is kept.
#  13. The Lucky Strike smoker drinks orange juice.
#  14. The Japanese smokes Parliaments.
#  15. The Norwegian lives next to the blue house.
#
#  Now, who drinks water? Who owns the zebra?
#
#  In the interest of clarity, it must be added that each of the five
#  houses is painted a different color, and their inhabitants are of
#  different national extractions, own different pets, drink different
#  beverages and smoke different brands of American cigarettes. One
#  other thing: In statement 6, right means your right.
#
#  (From Wikipedia, the free encyclopedia.)
##########################################################################

set HOUSE := { 1..5 };

set COLOR := { "blue", "green", "ivory", "red", "yellow" };

set NATIONALITY := { "Englishman", "Japanese", "Norwegian", "Spaniard",
      "Ukranian" };

set DRINK := { "coffee", "milk", "orange_juice", "tea", "water" };

set SMOKE := { "Chesterfield", "Kools", "Lucky_Strike", "Old_Gold",
      "Parliament" };

set PET := { "dog", "fox", "horse", "snails", "zebra" };

var color{HOUSE, COLOR}, binary;
c1{h in HOUSE}: sum{c in COLOR} color[h,c] = 1;
c2{c in COLOR}: sum{h in HOUSE} color[h,c] = 1;

var nationality{HOUSE, NATIONALITY}, binary;
n1{h in HOUSE}: sum{n in NATIONALITY} nationality[h,n] = 1;
n2{n in NATIONALITY}: sum{h in HOUSE} nationality[h,n] = 1;

var drink{HOUSE, DRINK}, binary;
d1{h in HOUSE}: sum{d in DRINK} drink[h,d] = 1;
d2{d in DRINK}: sum{h in HOUSE} drink[h,d] = 1;

var smoke{HOUSE, SMOKE}, binary;
s1{h in HOUSE}: sum{s in SMOKE} smoke[h,s] = 1;
s2{s in SMOKE}: sum{h in HOUSE} smoke[h,s] = 1;

var pet{HOUSE, PET}, binary;
p1{h in HOUSE}: sum{p in PET} pet[h,p] = 1;
p2{p in PET}: sum{h in HOUSE} pet[h,p] = 1;

/* the Englishman lives in the red house */
f2{h in HOUSE}: nationality[h,"Englishman"] = color[h,"red"];

/* the Spaniard owns the dog */
f3{h in HOUSE}: nationality[h,"Spaniard"] = pet[h,"dog"];

/* coffee is drunk in the green house */
f4{h in HOUSE}: drink[h,"coffee"] = color[h,"green"];

/* the Ukrainian drinks tea */
f5{h in HOUSE}: nationality[h,"Ukranian"] = drink[h,"tea"];

/* the green house is immediately to the right of the ivory house */
```

```
f6{h in HOUSE}:
   color[h,"green"] = if h = 1 then 0 else color[h-1,"ivory"];

/* the Old Gold smoker owns snails */
f7{h in HOUSE}: smoke[h,"Old_Gold"] = pet[h,"snails"];

/* Kools are smoked in the yellow house */
f8{h in HOUSE}: smoke[h,"Kools"] = color[h,"yellow"];

/* milk is drunk in the middle house */
f9: drink[3,"milk"] = 1;

/* the Norwegian lives in the first house */
f10: nationality[1,"Norwegian"] = 1;

/* the man who smokes Chesterfields lives in the house next to the man
   with the fox */
f11{h in HOUSE}:
   (1 - smoke[h,"Chesterfield"]) +
   (if h = 1 then 0 else pet[h-1,"fox"]) +
   (if h = 5 then 0 else pet[h+1,"fox"]) >= 1;

/* Kools are smoked in the house next to the house where the horse is
   kept */
f12{h in HOUSE}:
   (1 - smoke[h,"Kools"]) +
   (if h = 1 then 0 else pet[h-1,"horse"]) +
   (if h = 5 then 0 else pet[h+1,"horse"]) >= 1;

/* the Lucky Strike smoker drinks orange juice */
f13{h in HOUSE}: smoke[h,"Lucky_Strike"] = drink[h,"orange_juice"];

/* the Japanese smokes Parliaments */
f14{h in HOUSE}: nationality[h,"Japanese"] = smoke[h,"Parliament"];

/* the Norwegian lives next to the blue house */
f15{h in HOUSE}:
   (1 - nationality[h,"Norwegian"]) +
   (if h = 1 then 0 else color[h-1,"blue"]) +
   (if h = 5 then 0 else color[h+1,"blue"]) >= 1;

solve;

printf "\n";
printf "HOUSE␣␣COLOR␣␣␣NATIONALITY␣␣DRINK␣␣␣␣␣␣␣␣␣SMOKE␣␣␣␣␣␣␣␣␣PET\n";
for {h in HOUSE}
{  printf "%5d", h;
   printf{c in COLOR: color[h,c]} "␣␣%-6s", c;
   printf{n in NATIONALITY: nationality[h,n]} "␣␣%-11s", n;
   printf{d in DRINK: drink[h,d]} "␣␣%-12s", d;
   printf{s in SMOKE: smoke[h,s]} "␣␣%-12s", s;
   printf{p in PET: pet[h,p]} "␣␣%-6s", p;
   printf "\n";
}
printf "\n";

end;
```

This program contains several new elements:

(1) the '**solve**' command that tells glpk to assign values to all of the '**var**'s that satisfy the constraints. It appears at the end of the constraints section, where a '**maximize**' or '**minimize**' command might appear.

(2) the '**binary**' declaration. This forces a variable to only take on the values 0 or 1.

(3) the '**printf**' statement that causes data to be printed out as the program executes, so one does not need to locate the information in the output file. The format-string is like that used in the C programming language.

CHAPTER 8

Wavelets

"Then dulcet music swelled
Concordant with the life-strings of the soul;
It throbbed in sweet and languid beatings there,
Catching new life from transitory death;
Like the vague sighings of a wind at even
That wakes the wavelets of the slumbering sea... "
— Percy Bysshe Shelley[1] (see [56]).

8.1. Introduction

In this section, we will discuss a variation on Fourier Expansions that has gained a great deal of attention in recent years. It has many applications to image analysis and data-compression. We will only give a very abbreviated overview of this subject — see [44] and [62] for a more complete description.

Recall the Fourier Expansions

$$f(x) = \sum_{k=0}^{\infty} a_k \sin(kx) + b_k \cos(kx)$$

The development of wavelets was originally motivated by efforts to find a kind of Fourier Series expansion of *transient* phenomena[2]. If the function $f(x)$ has large *spikes* in its graph, the Fourier series expansion converges very slowly. This makes some intuitive sense — it is not surprising that it is difficult to express a function with sharp transitions or discontinuities in terms of smooth functions like sines and cosines. Furthermore, if $f(x)$ is localized in space (i.e., vanishes outside an interval) it may be hard to expand $f(x)$ in terms of sines and cosines, since these functions take on nonzero values over the entire x-axis.

We solve this problem by trying to expand $f(x)$ in series involving functions that *themselves* may have such spikes and vanish outside of a small interval. These functions are called *wavelets*. The term "wavelet" comes from the fact that these functions are not *complete waves;* they

[1]According to the Oxford English Dictionary, this is the first recorded use of the word 'wavelet'.

[2]Wavelet expansions grew out of problems related to *seismic analysis* — see [28].

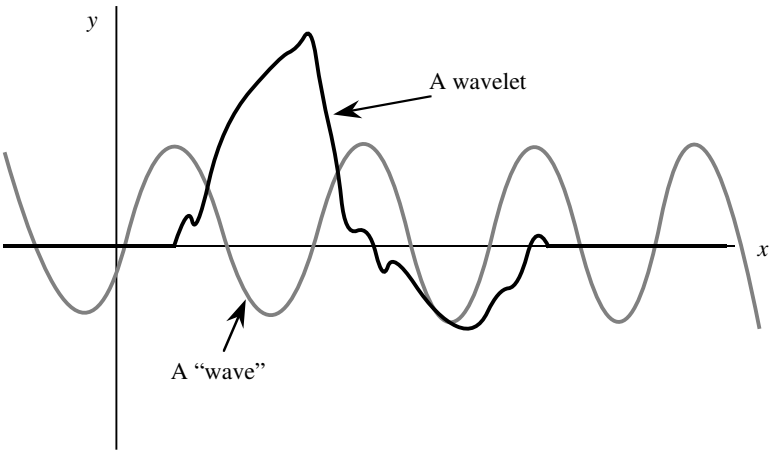


FIGURE 8.1.1. An example of a wavelet

vanish outside an interval. If a periodic function like $\sin(x)$ has a sequence of peaks and valleys over the entire x-axis, we think of this as a "wave", we think of a function with, say only small number of peaks or valleys, as a wave*let* — see figure 8.1.

Incidentally, the depiction of a wavelet in figure 8.1 is accurate in that the wavelet is "rough" — in many cases, wavelets are *fractal functions*, for reasons we will discuss a little later.

We will focus on Ingrid Daubechies's theory of wavelets, which lends itself to numerical computation.

Baroness Ingrid Daubechies (1954 –) is a Belgian-American physicist and mathematician. She is best known for her work with wavelets in image compression. Daubechies is recognized for her study of the mathematical methods that enhance image-compression technology. She is a member of the National Academy of Engineering, the National Academy of Sciences, and the American Academy of Arts and Sciences. She is a 1992 MacArthur Fellow. She also served on the Mathematical Sciences jury for the Infosys Prize from 2011 to 2013.
The name Daubechies is widely associated with the orthogonal Daubechies wavelet and the biorthogonal CDF wavelet. A wavelet from this family of wavelets is now used in the JPEG 2000 standard.

If we want to expand arbitrary functions like $f(x)$ in terms of wavelets, $w(x)$, like the one in figure 8.1, several problems are immediately apparent:

(1) How do we handle functions with spikes "sharper" than that of the main wavelet? This problem is solved in conventional Fourier series by *multiplying* the variable x by integers in the expansion. For instance, $\sin(nx)$ has peaks that are $1/n^{\text{th}}$ as

wide as the peaks of $\sin(x)$. For various reasons, the solution that is used in wavelet-expansions, is to multiply x by a *power of 2* — i.e., we expand in terms of $w(2^j x)$, for different values of j. This procedure of changing the scale of the x-axis is called *dilation*.

(2) Since $w(x)$ is only nonzero over a small finite interval, how do we handle functions that are nonzero over a much larger range of x-values? This is a problem that doesn't arise in conventional Fourier series because they involve expansions in functions that are nonzero over the whole x-axis. The solution use in wavelet-expansions is to *shift* the function $w(2^j x)$ by an integral distance, and to form linear combinations of these functions: $\sum_k w(2^j x - k)$. This is somewhat akin to taking the individual wavelets $w(2^j x - k)$ and assemble them together to form a *wave.* The reader may wonder what has been gained by all of this — we "chopped up" a wave to form a wavelet, and we are re-assembling these wavelets back into a wave. The difference is that we have direct control over how far this wave extends — we may, for instance, only use a *finite number* of displaced wavelets like $w(2^j x - k)$.

The upshot of this discussion is that a general wavelet expansion of of a function is *doubly indexed* series like:

$$f(x) = \sum_{\substack{-1 \le j < \infty \\ -\infty < k < \infty}} A_{jk} w_{jk}(x) \tag{8.1.1}$$

where

$$w_{jk}(x) = \begin{cases} w(2^j x - k) & \text{if } j \ge 0 \\ \phi(x - k) & \text{if } j = -1 \end{cases}$$

The function $w(x)$ is called the *basic wavelet* of the expansion and $\phi(x)$ is called the *scaling function* associated with $w(x)$.

We will begin by describing methods for computing suitable functions $w(x)$ and $\phi(x)$. We will usually want conditions like the following to be satisfied:

$$\int_{-\infty}^{\infty} w_{j_1 k_1}(x) w_{j_2 k_2}(x)\, dx = \begin{cases} 2^{-j_1} & \text{if } j_1 = j_2 \text{ and } k_1 = k_2 \\ 0 & \text{otherwise} \end{cases} \tag{8.1.2}$$

—these are called *orthogonality conditions*. Compare these to equations 4.3.5 on page 60, 4.3.6 on page 60, and 4.3.7 on page 60.

The reason for these conditions is that they make it very easy (at least in principle) to compute the coefficients in the basic wavelet expansion in equation (8.1.1): we simply multiply the entire series by $w(2^j x - k)$ or $\phi(x - i)$ and integrate. All but one of the terms of the result vanish due to the orthogonality conditions (equation (8.1.2)) and

we get:

$$A_{jk} = \frac{\int_{-\infty}^{\infty} f(x)w_{jk}(x)\,dx}{\int_{-\infty}^{\infty} w_{jk}^2(x)\,dx} = 2^j \int_{-\infty}^{\infty} f(x)w_{jk}(x)\,dx \tag{8.1.3}$$

In order to construct functions that are only nonzero over a finite interval of the x-axis, and satisfy the basic orthogonality condition, we carry out a sequence of steps. We begin by computing the *scaling function* associated with the wavelet $w(x)$.

A scaling function for a wavelet must satisfy the conditions[3]:

(1) Its support (i.e., the region of the x-axis over which it takes on nonzero values) is some finite interval. This is the same kind of condition that wavelets themselves must satisfy. This condition is simply a consequence of the basic concept of a wavelet-series.

(2) It satisfies the basic *dilation equation*:

$$\phi(x) = \sum_{i>-\infty}^{\infty} \xi_i \phi(2x - i) \tag{8.1.4}$$

Note that this sum is not as imposing as it appears at first glance — the previous condition implies that only a finite number of the $\{\xi_i\}$ can be nonzero. We write the sum in this form because we don't want to specify any *fixed* ranges of subscripts over which the $\{\xi_i\}$ may be nonzero.

This condition is due to Daubechies — see [15]. It is the heart of her theory of wavelet-series. It turns out to imply that the wavelet-expansions are orthogonal and easy to compute.

(3) Note that any multiple of a solution of equation (8.1.4) is also a solution. We select a preferred solution by imposing the condition

$$\int_{-\infty}^{\infty} \phi(x)\,dx = 1 \tag{8.1.5}$$

One points come to mind when we consider these conditions from a *computer science* point of view:

> Equation 8.1.4, the finite set of nonzero values of $\phi(x)$ at integral points, and the finite number of nonzero $\{\xi_i\}$ completely determine $\phi(x)$. They determine it at all *dyadic points* (i.e., values of x of the form p/q, where q is a power of 2). For virtually all modern computers, such points are the only ones that exist, so $\phi(x)$ is completely determined.

[3]Incidentally, the term scaling function, like the term wavelet refers to a whole *class* of functions that have certain properties.

Of course, from a *function theoretic* point of view $\phi(x)$ is far from being determined by its dyadic values. What is generally done is to perform a iterative procedure: we begin by setting $\phi_0(x)$ equal to some simple function like the box function equal to 1 for $0 \leq x < 1$ and 0 otherwise. We then define

$$\phi_{i+1}(x) = \sum_{k>-\infty}^{\infty} \xi_k \phi_i(2x-k) \tag{8.1.6}$$

It turns out that this procedure converges to a limit $\phi(x) = \phi_\infty(x)$, that satisfies equation (8.1.4) exactly. Given a suitable scaling-function $\phi(x)$, we define the associated wavelet $w(x)$ by the formula

$$w(x) = \sum_{i>-\infty}^{\infty} (-1)^i \xi_{1-i} \phi(2x-i) \tag{8.1.7}$$

We will want to impose some conditions upon the coefficients $\{\xi_i\}$.

DEFINITION 8.1.1. The defining coefficients of a system of wavelets will be assumed to satisfy the following two conditions:

(1) **Condition O:** This condition implies the orthogonality condition of the wavelet function (equation 8.1.2 on page 153):

$$\sum_{k>-\infty}^{\infty} \xi_k \xi_{k-2m} = \begin{cases} 2 & \text{if } m = 0 \\ 0 & \text{otherwise} \end{cases} \tag{8.1.8}$$

The orthogonality relations mean that if a function can be expressed in terms of the wavelets, we can easily *calculate* the coefficients involved, via equation 8.1.3 on the facing page.

(2) **Condition A:** There exists a number $p > 1$, called the *degree of smoothness* of $\phi(x)$ and the associated wavelet $w(x)$, such that

$$\sum_{k>-\infty}^{\infty} (-1)^k k^m \xi_k = 0, \text{for all } 0 \leq m \leq p-1$$

It turns out that wavelets are generally *fractal functions* — they are not differentiable unless their degree of smoothness is > 2.

This condition guarantees that the functions that interest us[4] *can* be expanded in terms of wavelets. If $\phi(x)$ is a scaling function with a degree of smoothness equal to p, it is possible to expand the functions $1, x, \ldots, x^{p-1}$ in terms of series like

$$\sum_{j>-\infty}^{\infty} A_n \phi(x-n)$$

[4]This term is deliberately vague.

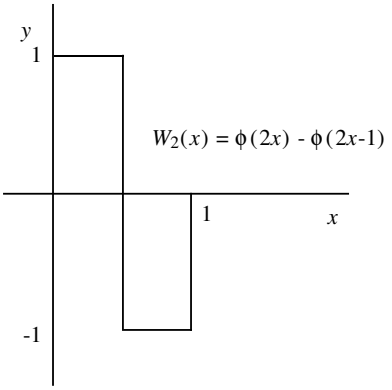


FIGURE 8.1.2. The Haar Wavelet

In order for wavelets to be significant to us, they (and their scaling functions) must be derived from a sequence of coefficients $\{\xi_i\}$ with a degree of smoothness > 0.

In [15], Daubechies discovered a family of wavelets W_2, W_4, W_6,$\cdots$ whose defining coefficients (the $\{\xi_i\}$) satisfy these conditions. All of these wavelets are based upon scaling functions that result from iterating the box function

$$\phi_0(x) = \begin{cases} 1 & \text{if } 0 \leq x < 1 \\ 0 & \text{otherwise} \end{cases}$$

in the dilation-equation 8.1.6 on the previous page. The different elements of this sequence of wavelets are only distinguished by the sets of coefficients used in the iterative procedure for computing $\phi(x)$ and the corresponding wavelets.

(Note: this function, *must vanish* at one of the endpoints of the interval $[0, 1]$.) This procedure for computing wavelets (i.e., plugging the box function into equation (8.1.4) and repeating this with the result, etc.) is not very practical. It is computationally expensive, and only computes approximations to the desired result[5].

Fortunately, there is a simple, fast, and *exact* algorithm for computing wavelets at all *dyadic points* using equation (8.1.4) and the values of $\phi(x)$ at integral points. Furthermore, from the perspective of computers, the dyadic points are the only ones that exist. We just perform a recursive computation of $\phi(x)$ at points of the form $i/2^{k+1}$ using the

[5]This slow procedure has theoretical applications — the proof that the wavelets are orthogonal (i.e., satisfy equation 8.1.2 on page 153) is based on this construction.

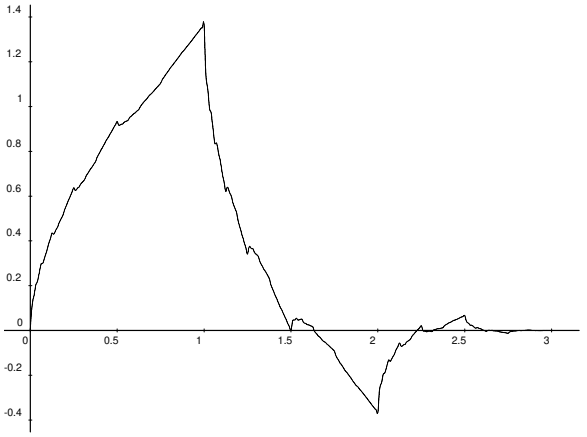


FIGURE 8.1.3. Daubechies degree-4 scaling function

values at points of the form $i/2^k$ and the formula

$$\phi(i/2^{k+1}) = \sum_{-\infty<m<\infty} \xi_m \phi(\frac{i}{2^k} - m) \tag{8.1.9}$$

It is often possible for the dilation-equation to imply relations between the values of a scaling function at distinct integral points. We must choose the value at these points in such a way as to satisfy the dilation-equation.

EXAMPLE 8.1.2. **Daubechies' W_2 Wavelet.** This is the simplest element of the Daubechies sequence of wavelets. This family is defined by the fact that the coefficients of the dilation-equation are $\xi_0 = \xi_1 = 1$, and all other $\xi_i = 0$.

In this case $\phi(x) = \phi_0(x)$, the box function. The corresponding *wavelet*, $W_2(x)$ has been described long before the development of wavelets — it is called the *Haar function*. It is depicted in figure 8.1 on the preceding page.

EXAMPLE 8.1.3. **Daubechies' W_4 Wavelet.** Here we use the coefficients $\xi_0 = (1+\sqrt{3})/4$, $\xi_1 = (3+\sqrt{3})/4$, $\xi_2 = (3-\sqrt{3})/4$, and $\xi_3 = (1-\sqrt{3})/4$ in the dilation-equation. This wavelet has smoothness equal to 2 (so it is continuous, but not differentiable), and its scaling function $\phi(x)$ is called $D_4(x)$. We can compute the scaling function, $D_4(x)$ at the dyadic points by the recursive procedure described above. We cannot pick the values of $\phi(1)$ and $\phi(2)$ arbitrarily because they are not independent of each other in the equation for $\phi(x) = D_4(x)$.

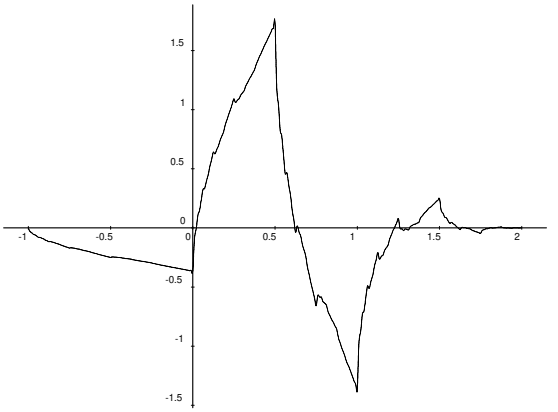


FIGURE 8.1.4. Daubechies degree-4 wavelet

Equation 8.1.9 on the preceding page implies they satisfy

$$\phi(1) = \frac{3+\sqrt{3}}{4}\phi(1) + \frac{1+\sqrt{3}}{4}\phi(2)$$
$$\phi(2) = \frac{1-\sqrt{3}}{4}\phi(1) + \frac{3-\sqrt{3}}{4}\phi(2)$$

This is an *eigenvalue* problem[6] like

$$\Xi x = \lambda x$$

where x is the vector composed of $\phi(1)$ and $\phi(2)$, and Ξ is the matrix

$$\begin{pmatrix} \frac{3+\sqrt{3}}{4} & \frac{1+\sqrt{3}}{4} \\ \frac{1-\sqrt{3}}{4} & \frac{3-\sqrt{3}}{4} \end{pmatrix}$$

We enter this into Maxima via

```
v:matrix([(3+sqrt(3))/4,(1+sqrt(3))/4],
         [(1-sqrt(3))/4,(3-sqrt(3))/4])
```

and find its eigenvalues via

```
eigenvalues(v)
```

these turn out to be 1 and 1/2. The problem only has a solution if $\lambda = 1$ is a valid eigenvalue of Ξ — in this case the correct values of $\phi(1)$ and $\phi(2)$ are given by some scalar multiple of the eigen*vector* associated with the eigenvalue 1. The corresponding eigenvectors are given by

```
eigenvectors(v)
```

[6] See 7.4.1 on page 124 for the definition of an eigenvalue.

which produces

$$\left[\left[\left[1,\frac{1}{2}\right],[1,1]\right],\left[\left[\left[1,\sqrt{3}-2\right]\right],[[1,-1]]\right]\right]$$

So the matrix (Ξ) does have an eigenvalue of 1, and its associated eigenvector is

$$\begin{pmatrix}1\\ \sqrt{3}-2\end{pmatrix}$$

We can normalize this so that it sums up to 1 by dividing by $\sqrt{3}-1$ to get

$$\begin{pmatrix}\frac{1+\sqrt{3}}{2}\\ \frac{1-\sqrt{3}}{2}\end{pmatrix}$$

and these become, from top to bottom, our values of $\phi(1)$ and $\phi(2)$, respectively. The scaling function, $\phi(x)$, is called $D_4(x)$ in this case[7] and is plotted in figure 8.1 on page 157.

Notice the *fractal* nature of the function. It is actually much more irregular than it appears in this graph. The associated wavelet is called W_4 and is depicted in figure 8.1.3 on the preceding page.

EXERCISES.

1. Write a program to compute $D_4(x)$ and $W_4(x)$ at dyadic points, using the recursive formula in equation 8.1.9 on page 157 described above. Generally we measure the extent to which $D_4(x)$ has been computed by measuring the "fineness of the mesh" upon which we know the values of this function — in other words, $1/2^n$.

2. Write Maxima code to plot $\phi(x)$ by only evaluating it at points of the form

$$\left\{\frac{n}{2^{10}}\right\}$$

where $0 \leq n \leq 3\cdot 2^{10}$. Hint: look at appendix E on page 299 (especially **makelist**) and *discrete* plots in appendix F on page 323.

[7] In honor of Daubechies.

8.2. Discrete Wavelet Transforms

Now we are in a position to discuss how one does a discrete version of the wavelet transform. We will give an algorithm for computing the wavelet-series in equation 8.1.1 on page 153 up to some finite value of j — we will compute the $A_{j,k}$ for $j \leq p$. We will call the parameter p the fineness of mesh of the expansion, and 2^{-p} the mesh-size.

DEFINITION 8.2.1. Define $B_{r,j}$ by

$$B_{r,j} = 2^r \int_{-\infty}^{\infty} \phi(2^r x - j) f(x)\, dx$$

where $0 \leq r \leq p$ and $-\infty < j < \infty$.

These quantities are important because they allow us to compute the coefficients of the wavelet-series.

In general, the $B_{r,j}$ are nonzero for at most a *finite* number of values of j:

PROPOSITION 8.2.2. *In the notation of 8.2.1, above, suppose that $f(x)$ is only nonzero on the interval $a \leq x \leq b$ and $\phi(x)$ is only nonzero on the interval $0 \leq x \leq R$. Then $B_{r,j} = 0$ unless $\mathcal{L}(r) \leq j \leq \mathcal{U}(r)$, where $\mathcal{L}(r) = \lfloor 2^r a - R \rfloor$ and $\mathcal{U}(r) = \lceil 2^r b \rceil$.*

We will follow the convention that $\mathcal{L}(-1) = \lfloor a - R \rfloor$, and $\mathcal{U}(-1) = \lceil b \rceil$

PROOF. In order for the integral 8.2.1 to be nonzero, it is at least necessary for the domains in which $f(x)$ and $\phi(2^r x - j)$ are nonzero to intersect. This means that

$$\begin{aligned} 0 \leq 2^r x - j &\leq R \\ a \leq x &\leq b \end{aligned}$$

If we add j to the first inequality, we get:

$$j \leq 2^r x \leq j + R$$

or

$$2^r x - R \leq j \leq 2^r x$$

The second inequality implies the result. □

The first thing to note is that the quantities $B_{p,j}$ determine the $B_{r,j}$ for all values of r such that $0 \leq r < p$:

PROPOSITION 8.2.3. *For all values of $r \leq p$*

$$B_{r,j} = \sum_{m > -\infty}^{\infty} \frac{\xi_{m-2j} B_{r+1,m}}{2}$$

PROOF. This is a direct consequence of the basic dilation equation 8.1.4 on page 154:

$$\begin{aligned}
B_{r,j} =& 2^r \int_{-\infty}^{\infty} \phi(2^r x - j) f(x)\, dx \\
=& 2^r \int_{-\infty}^{\infty} \sum_{s>-\infty}^{\infty} \xi_s \phi(2(2^r x - j) - s) f(x)\, dx \\
=& 2^r \int_{-\infty}^{\infty} \sum_{s>-\infty}^{\infty} \xi_s \phi(2^{r+1} x - 2j - s) f(x)\, dx \\
& \text{setting } m = 2j + s \\
=& 2^r \int_{-\infty}^{\infty} \sum_{m>-\infty}^{\infty} \xi_{m-2j} \phi(2^{r+1} x - m) f(x)\, dx \\
=& 2^r \sum_{m>-\infty}^{\infty} \xi_{m-2j} \int_{-\infty}^{\infty} \phi(2^{r+1} x - m) f(x)\, dx
\end{aligned}$$

□

The definition of $w(x)$ in terms of $\phi(x)$ implies that

PROPOSITION 8.2.4. *Let $A_{r,k}$ denote the coefficients of the wavelet-series, as defined in equation 8.1.3 on page 154. Then*

$$A_{r,k} = \begin{cases} \sum_{m>-\infty}^{\infty} (-1)^m \frac{\xi_{1-m+2k} B_{r+1,m}}{2} & \text{if } r \geq 0 \\ B_{-1,k} & \text{if } r = -1 \end{cases}$$

PROOF. This is a direct consequence of equation 8.1.7 on page 155. We take the definition of the $A_{r,k}$ and plug in equation 8.1.7 on page 155:

$$\begin{aligned}
A_{r,k} =& 2^r \int_{-\infty}^{\infty} f(x) w(2^r x - k)\, dx \\
=& 2^r \int_{-\infty}^{\infty} f(x) \sum_{s>-\infty}^{\infty} (-1)^s \xi_{1-s} \phi(2(2^r x - k) - s)\, dx \\
=& 2^r \int_{-\infty}^{\infty} f(x) \sum_{s>-\infty}^{\infty} (-1)^s \xi_{1-s} \phi(2^{r+1} x - 2k - s)\, dx \\
& \text{now we set } m = 2k + s \\
=& 2^r \int_{-\infty}^{\infty} f(x) \sum_{m>-\infty}^{\infty} (-1)^m \xi_{1-m+2k} \phi(2^{r+1} x - m)\, dx \\
=& 2^r \sum_{m>-\infty}^{\infty} (-1)^m \xi_{1-m+2k} \int_{-\infty}^{\infty} f(x) \phi(2^{r+1} x - m)\, dx
\end{aligned}$$

□

We can code Maxima functions to compute these. Our algorithm computes all of the $A_{r,k}$, given the values of $B_{p+1,k}$:

$$B_{k,i} = \frac{1}{2}\sum_{j=\mathcal{L}(k+1)}^{\mathcal{U}(k+1)} \xi_{j-2i}\cdot B_{k+1,j} \tag{8.2.1}$$

$$A_{k,i} = \frac{1}{2}\sum_{j=\mathcal{L}(k+1)}^{\mathcal{U}(k+1)} (-1)^j \xi_{1-j+2i}\cdot B_{k+1,j}$$

where i runs from $\mathcal{L}(k)$ to $\mathcal{U}(k)$, and k runs from $p+1$ *down* to 1. The $A_{k,*}$ are, of course, the coefficients of the wavelet-expansion.

The only elements of this algorithm that look a little mysterious are the quantities

$$B_{p+1,j} = 2^{p+1}\int_{-\infty}^{\infty} \phi(2^{p+1}x - j) f(x)\, dx$$

First we note that $\int_{-\infty}^{\infty}\phi(u)\,du = 1$ (by equation (8.1.5) on page 154), so $\int_{-\infty}^{\infty}\phi(2^{p+1}x-j)\,dx = 2^{-(p+1)}$ (set $u = 2^{p+1}-j$, and $dx = 2^{-(p+1)}du$) and

$$B_{p+1,j} = \frac{\int_{-\infty}^{\infty}\phi(2^{p+1}x-j)f(x)\,dx}{\int_{-\infty}^{\infty}\phi(2^{p+1}x-j)\,dx}$$

so that $B_{p+1,j}$ is nothing but a *weighted average* of $f(x)$ weighted by the function $\phi(2^{p+1}x-j)$. Now note that this weighted average is really being taken over a small interval $0 \le 2^{p+1}x - j \le R$, where $[0,R]$ is the range of values over which $\phi(x) \neq 0$. This is always some finite interval — for instance if $\phi(x) = D_4(x)$ (see figure 8.1 on page 157), this interval is $[0,3]$. This means that x runs from $j2^{-(p+1)}$ to $(j+R)2^{-(p+1)}$.

At this point we make the assumption:

ASSUMPTION 8.2.5. ***The width of the interval*** $[j2^{-(p+1)}, (j+R)2^{-(p+1)}]$***, is small enough that*** $f(x)$ ***doesn't vary in any appreciable way over this interval. Consequently, the weighted average is equal to*** $f(j2^{-(p+1)})$**.**

So we begin the inductive computation of the $A_{k,j}$ in 8.2.1 by setting

$$B_{p+1,j} = f(j/2^{p+1}) \tag{8.2.2}$$

We regard the set of values $\{f(j/2^{p+1})\}$ with $0 \le j < 2^{p+1}$ as the *inputs* to the discrete wavelet transform algorithm.

The *output* of the algorithm is the set of wavelet-coefficients $\{A_{k,j}\}$, with $-1 \le k \le p$, $-\infty < j < \infty$. Note that j actually only takes on a finite set of values — this set is usually small and depends upon the

type of wavelet under consideration. In the case of the Haar wavelet, for instance $0 \leq j \leq 2^k - 1$, if $k \leq 0$, and $j = 0$ if $k = -1$. In the case of the Daubechies W_4 wavelet this set is a little larger, due to the fact that there are more nonzero defining coefficients $\{\zeta_i\}$.

Now we will give a fairly detailed example of this algorithm. Let $f(x)$ be the function defined by:

$$f(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ x & \text{if } 0 < x \leq 1 \\ 0 & \text{if } x > 1 \end{cases}$$

We will expand this into a wavelet-series using the degree-4 Daubechies wavelet defined in 8.1.3 on page 157. We start with mesh-size equal to 2^{-5}, so $p = 4$, and we define $B_{5,*}$ by

$$B_{5,i} = \begin{cases} 0 & \text{if } i \leq 0 \\ i/32 & \text{if } 1 \leq i \leq 32 \\ 0 & \text{if } i > 32 \end{cases}$$

In the present case, the looping phase of equation 8.2.1 on the facing page involves the computation:

$$\begin{aligned} B_{k,i} =& \frac{1+\sqrt{3}}{8} B_{k+1,2i} + \frac{3+\sqrt{3}}{8} B_{k+1,2i+1} + \frac{3-\sqrt{3}}{8} B_{k+1,2i+2} + \frac{1-\sqrt{3}}{8} B_{k+1,2i+3} \\ A_{k,i} =& \frac{1-\sqrt{3}}{8} B_{k+1,2i-2} - \frac{3-\sqrt{3}}{8} B_{k+1,2i-1} + \frac{3+\sqrt{3}}{8} B_{k+1,2i} - \frac{1+\sqrt{3}}{8} B_{k+1,2i+1} \end{aligned}$$

▷ **Iteration 1:** The $B_{4,*}$ and the wavelet-coefficients, $A_{4,*}$ are all zero except for the following cases:

$$\left\{ \begin{aligned} B_{4,-1} =& \frac{1}{256} - \frac{\sqrt{3}}{256} \\ B_{4,j} =& \frac{4j+3-\sqrt{3}}{64} \text{ for } 0 \leq j \leq 14 \\ B_{4,15} =& \frac{219}{256} + \frac{29\sqrt{3}}{256} \\ B_{4,16} =& 1/8 + \frac{\sqrt{3}}{8} \end{aligned} \right\} \quad \left\{ \begin{aligned} A_{4,0} =& -\frac{1}{256} - \frac{\sqrt{3}}{256} \\ A_{4,16} =& \frac{33}{256} + \frac{33\sqrt{3}}{256} \\ A_{4,17} =& 1/8 - \frac{\sqrt{3}}{8} \end{aligned} \right\}$$

Notice that *most* of the $\{A_{4,i}\}$ are 0. This is how the original function has been *compressed:* we replaced its values by these coefficients.

Now we can calculate $B_{3,*}$ and $A_{3,*}$:

▷ **Iteration 2:**

$$\left\{\begin{array}{l} B_{3,-2}=\dfrac{1}{512}-\dfrac{\sqrt{3}}{1024} \\ B_{3,-1}=\dfrac{11}{256}-\dfrac{29\sqrt{3}}{1024} \\ B_{3,j}=\dfrac{8j+9-3\sqrt{3}}{64} \text{ for } 0\le j\le 5 \\ B_{3,6}=\dfrac{423}{512}-\dfrac{15\sqrt{3}}{1024} \\ B_{3,7}=\dfrac{121}{256}+\dfrac{301\sqrt{3}}{1024} \\ B_{3,8}=1/16+\dfrac{\sqrt{3}}{32} \end{array}\right\} \quad \left\{\begin{array}{l} A_{3,-1}=\dfrac{1}{1024} \\ A_{3,0}=\dfrac{1}{1024}-\dfrac{5\sqrt{3}}{512} \\ A_{3,7}=-\dfrac{33}{1024} \\ A_{3,8}=\dfrac{5\sqrt{3}}{512}-\dfrac{65}{1024} \\ A_{3,9}=-1/32 \end{array}\right\}$$

▷ **Iteration 3:**

$$\left\{\begin{array}{l} B_{2,-2}=\dfrac{35}{2048}-\dfrac{39\sqrt{3}}{4096} \\ B_{2,-1}=\dfrac{259}{2048}-\dfrac{325\sqrt{3}}{4096} \\ B_{2,0}=\dfrac{21}{64}-\dfrac{7\sqrt{3}}{64} \\ B_{2,1}=\dfrac{37}{64}-\dfrac{7\sqrt{3}}{64} \\ B_{2,2}=\dfrac{1221}{2048}+\dfrac{87\sqrt{3}}{4096} \\ B_{2,3}=\dfrac{813}{2048}+\dfrac{1125\sqrt{3}}{4096} \\ B_{2,4}=\dfrac{5}{256}+\dfrac{3\sqrt{3}}{256} \end{array}\right\} \quad \left\{\begin{array}{l} A_{2,-1}=\dfrac{23}{4096}-\dfrac{\sqrt{3}}{512} \\ A_{2,0}=-\dfrac{27}{4096}-\dfrac{3\sqrt{3}}{256} \\ A_{2,3}=\dfrac{15\sqrt{3}}{512}-\dfrac{295}{4096} \\ A_{2,4}=\dfrac{315}{4096}-\dfrac{35\sqrt{3}}{256} \\ A_{2,5}=-\dfrac{1}{256}-\dfrac{\sqrt{3}}{256} \end{array}\right\}$$

▷ **Iteration 4:**

$$\left\{\begin{array}{l} B_{1,-2}=\dfrac{455}{8192}-\dfrac{515\sqrt{3}}{16384} \\ B_{1,-1}=\dfrac{2405}{8192}-\dfrac{2965\sqrt{3}}{16384} \\ B_{1,0}=\dfrac{2769}{8192}-\dfrac{381\sqrt{3}}{16384} \\ B_{1,1}=\dfrac{2763}{8192}+\dfrac{3797\sqrt{3}}{16384} \\ B_{1,2}=\dfrac{7}{1024}+\dfrac{\sqrt{3}}{256} \end{array}\right\} \quad \left\{\begin{array}{l} A_{1,-1}=\dfrac{275}{16384}-\dfrac{15\sqrt{3}}{2048} \\ A_{1,0}=-\dfrac{339}{16384}-\dfrac{67\sqrt{3}}{4096} \\ A_{1,2}=\dfrac{531}{16384}-\dfrac{485\sqrt{3}}{4096} \\ A_{1,3}=-\dfrac{1}{512}-\dfrac{\sqrt{3}}{1024} \end{array}\right\}$$

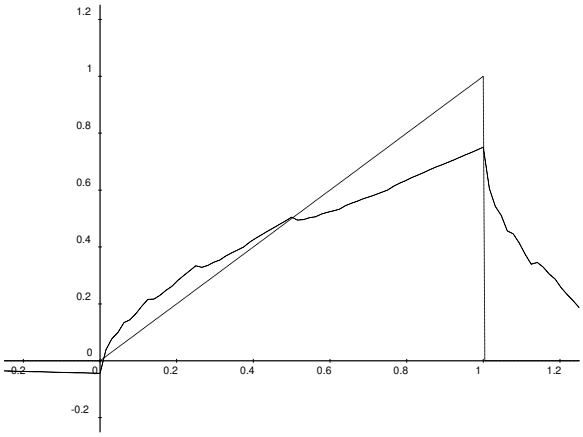


FIGURE 8.2.1. First term of the wavelet-series

▷ **Iteration 5:** In this phase we complete the computation of the wavelet-coefficients: these are the $A_{0,*}$ and the $B_{0,*} = A_{-1,*}$.

$$\left\{\begin{array}{rl} B_{0,-2} =& \frac{4495}{32768} - \frac{5115\sqrt{3}}{65536} \\ B_{0,-1} =& \frac{2099}{16384} - \frac{3025\sqrt{3}}{32768} \\ B_{0,1} =& \frac{19}{8192} + \frac{11\sqrt{3}}{8192} \end{array}\right\} \quad \left\{\begin{array}{rl} A_{0,-1} =& \frac{2635}{65536} - \frac{155\sqrt{3}}{8192} \\ A_{0,0} =& \frac{919\sqrt{3}}{16384} - \frac{5579}{32768} \\ A_{0,2} =& -\frac{5}{8192} - \frac{3\sqrt{3}}{8192} \end{array}\right\}$$

We will examine the convergence of this wavelet-series. The $A_{-1,*}$ terms are:

$$S_{-1} = \left(\frac{4495}{32768} - \frac{5115\sqrt{3}}{65536}\right) D_4(x+2) + \left(\frac{2099}{16384} - \frac{3025\sqrt{3}}{32768}\right) D_4(x+1) + \left(\frac{19}{8192} + \frac{11\sqrt{3}}{8192}\right) D_4(x)$$

This expression is analogous to the *constant term* in a Fourier series.

It is plotted against $f(x)$ in figure 8.2 — compare this (and the following plots with the partial-sums of the Fourier series in figures 4.3.2 on page 63 to 4.3.4 on page 64. If we add in the $A_{0,*}$-terms we get:

$$S_0(x) = S_{-1}(x) + \left(\frac{2635}{65536} - \frac{155\sqrt{3}}{8192}\right) W_4(x+1) + \left(\frac{919\sqrt{3}}{16384} - \frac{5579}{32768}\right) W_4(x) - \left(\frac{5}{8192} + \frac{3\sqrt{3}}{8192}\right) W_4(x-2)$$

It is plotted against the original function $f(x)$ in figure 8.2.

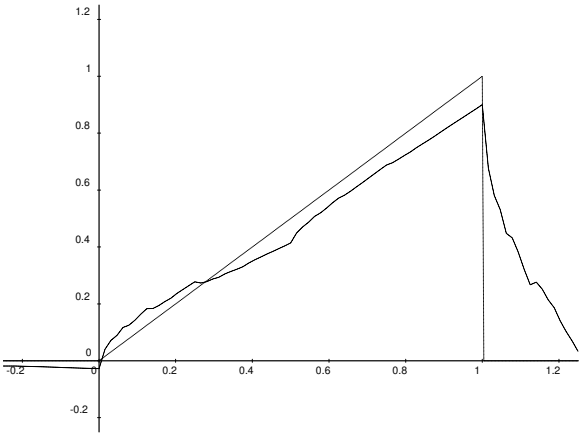


FIGURE 8.2.2. First two terms of wavelet-series

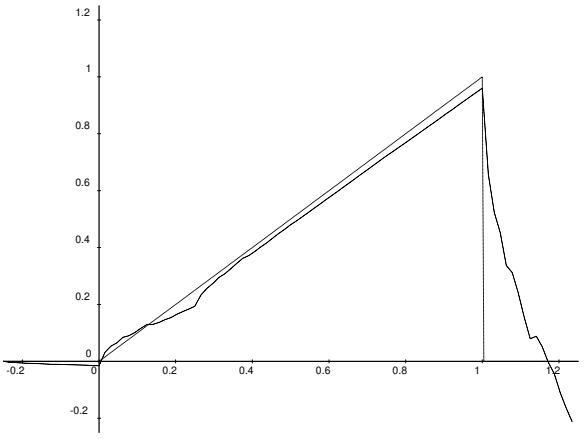


FIGURE 8.2.3. First three terms

The next step involves adding in the $A_{1,*}$-terms

$$S_1(x) = S_0(x) + \left(\frac{275}{16384} - \frac{15\sqrt{3}}{2048}\right) W_4(2x+1) - \left(\frac{339}{16384} + \frac{67\sqrt{3}}{4096}\right) W_4(2x) - \left(\frac{531}{16384} - \frac{485\sqrt{3}}{4096}\right) W_4(2x-2) - \left(\frac{1}{512} + \frac{\sqrt{3}}{1024}\right) W_4(2x-3)$$

Figure 8.2 shows how the wavelet-series begins to approximate $f(x)$.

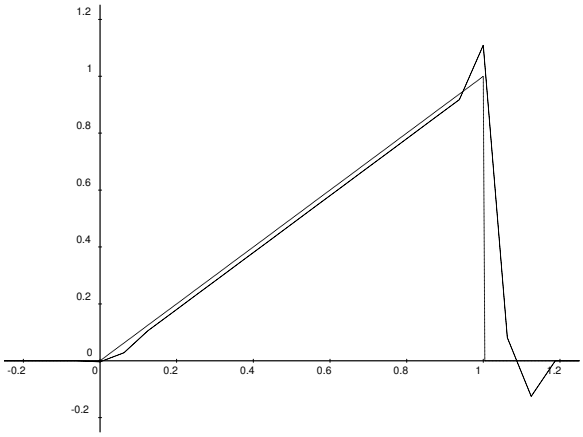


FIGURE 8.2.4. First four terms

The $A_{3,*}$ contribute:

$$S_2(x) = S_1(x) + \left(\frac{23}{4096} - \frac{\sqrt{3}}{512}\right) W_4(4x+1) - \left(\frac{27}{4096} + \frac{3\sqrt{3}}{256}\right) W_4(4x) + \left(\frac{15\sqrt{3}}{512} - \frac{295}{4096}\right) W_4(4x-3) + \left(\frac{315}{4096} - \frac{35\sqrt{3}}{256}\right) W_4(4x-4) - \left(\frac{1}{256} + \frac{\sqrt{3}}{256}\right) W_4(4x-5)$$

As with Fourier series (see equation 4.3.10 on page 64), this series converges in following sense

$$\lim_{n\to\infty} \int_{-\infty}^{\infty} (f(x) - S_n(x))^2 \, dx \to 0 \tag{8.2.3}$$

This means that the *area* of the space *between* the graphs of $f(x)$ and $S_n(x)$ approaches 0 as n approaches ∞. This does not necessarily mean that $S_n(x) \to f(x)$ for all values of x. It is interesting that there are points x_0 where $\lim_{n\to\infty} S_n(x_0) \neq f(x_0)$. For instance, $x = 1$ is such a point[8]. Equation 8.2.3 implies that the total area of this set of points is zero. Luckily, most of the applications of wavelets only require the kind of convergence described in equation 8.2.3.

We will conclude this section with a discussion of the *converse* of the algorithm defined by equation 8.2.1 on page 162 — it is an algorithm that computes partial sums of a wavelet-series, given the coefficients $\{A_{j,k}\}$. Although there is a straightforward method for doing this that involves simply plugging values into the functions $w(2^j x - k)$

[8]This is Gibbs phenomena again!

and plugging these values into the wavelet-series, there is also a faster algorithm for this. This algorithm is very similar to the algorithm for computing the $\{A_{j,k}\}$ in the first place.

We make use of the recursive definition of the scaling and wavelet-functions in equation 8.1.4 on page 154. This leads to the reconstruction-algorithm for wavelets:

$$B_{k+1,i} = \sum_{j=\mathcal{L}(k)}^{\mathcal{U}(k)} \left(\xi_{i-2j} B_{k,j} + (-1)^j \xi_{1-i+2j} A_{k,j}\right) \tag{8.2.4}$$

where i runs from $\mathcal{L}(k+1)$ to $\mathcal{U}(k+1)$, and k runs from -1 to $p+1$. At this point, the reader might wonder:

> *In what sense have we evaluated the wavelet series?* All we've done is compute the $\left\{B_{p+1,i}\right\}$ using the $\left\{A_{k,j}\right\}$(recall that $B_{-1,j} = A_{-1,j}$).

Equation 8.2.2 on page 162 states that $B_{p+1,i} = f(2^{-p-1}i)$, so we have actually reconstructed $f(x)$ — at least at the points $\left\{\frac{j}{2^{p+1}}\right\}$.

8.3. Discussion and Further reading

The defining coefficients for the Daubechies wavelets W_{2n} for $n > 2$ are somewhat complex — see [15] for a general procedure for finding them. For instance, the defining coefficients for W_6 are

$$\begin{aligned}
c_0 =& \frac{\sqrt{5+2\sqrt{10}}}{16} + 1/16 + \frac{\sqrt{10}}{16} \\
c_1 =& \frac{\sqrt{10}}{16} + \frac{3\sqrt{5+2\sqrt{10}}}{16} + \frac{5}{16} \\
c_2 =& 5/8 - \frac{\sqrt{10}}{8} + \frac{\sqrt{5+2\sqrt{10}}}{8} \\
c_3 =& 5/8 - \frac{\sqrt{10}}{8} - \frac{\sqrt{5+2\sqrt{10}}}{8} \\
c_4 =& \frac{5}{16} - \frac{3\sqrt{5+2\sqrt{10}}}{16} + \frac{\sqrt{10}}{16} \\
c_5 =& 1/16 - \frac{\sqrt{5+2\sqrt{10}}}{16} + \frac{\sqrt{10}}{16}
\end{aligned} \tag{8.3.1}$$

In [63], Sweldens and Piessens give formulas for approximately computing coefficients of wavelet-expansions:

$$B_{r,j} = 2^r \int_{-\infty}^{\infty} \phi(2^r x - j) f(x)\, dx$$

(defined in 8.2.1 on page 160). For the Daubechies wavelet $W_4(x)$ the simplest case of their algorithm gives:

$$B_{r,j} \approx f\left(\frac{2j+3-\sqrt{3}}{2^{r+1}}\right)$$

(the accuracy of this formula increases with increasing r). This is more accurate than the estimates we used in the example that appeared in pages 163 to 167 (for instance, it is *exact* if $f(x) = x$). We didn't go into this approximation in detail because it would have taken us too far afield.

The general continuous wavelet-transform of a function $f(x)$ with respect to a wavelet $w(x)$, is given by

$$\mathcal{T}_f(a,b) = \frac{1}{\sqrt{a}}\int_{-\infty}^{\infty} \bar{w}\left(\frac{x-b}{a}\right) f(x)\,dx$$

where $\bar{*}$ denotes complex conjugation. The two variables in this function correspond to the two indices in the wavelet-series that we have been discussing in this section. This definition was proposed by Morlet, Arens, Fourgeau and Giard in [3]. It turns out that we can recover the function $f(x)$ from its wavelet-transform via the formula

$$f(x) = \frac{1}{2\pi C_h}\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} \frac{\mathcal{T}_f(a,b)}{\sqrt{|a|}} w\left(\frac{x-b}{a}\right)\,da\,db$$

where C_h is a suitable constant (the explicit formula for C_h is somewhat complicated, and not essential for the present discussion).

The two equations 8.2.1 on page 162 and 8.2.4 on the preceding page are, together, a kind of wavelet-analogue to the FFT algorithm. In many respects, the fast wavelet transform and its corresponding reconstruction algorithm are simpler and more straightforward than the FFT algorithm. They use basic linear algebra rather than sines and cosines.

Wavelets that are used in image processing are two-dimensional. It is possible to get such wavelets from one-dimensional wavelets via the process of taking the *tensor-product*. This amounts to making definitions like:

$$W(x,y) = w(x)w(y)$$

In fact the image-compression standard called JPEG2000 uses Daubechies wavelet transforms rather than Fourier series — see [64]. There many graphics applications that use it, including the virtual reality system OpenSimulator (and Second Life).

The concept of wavelets predate their "official" definition in [28].

Discrete Wavelet-transforms of images that are based upon tensor-products of the Haar wavelet were known to researchers in image-processing — such transforms are known as *quadtree* representations of images.

Many authors have defined systems of wavelets that remain non-vanishing over the entire x-axis. In every case, these wavelets decay to 0 in such a way that conditions like equation 8.1.2 on page 153 are still satisfied. The wavelets of Meyer decay like $1/x^k$ for a suitable exponent k — see [46].

See [8] for an interesting application of wavelets to *astronomy* — in this case, the determination of large-scale structure of the universe.

See [35] as an excellent general reference.

EXERCISES.

1. Write a program to compute wavelet-coefficients using equation 8.2.1 on page 162.

2. Find a wavelet-series for the function

$$f(x) = \begin{cases} 1 & \text{if } 0 \leq x \leq 1/3 \\ 0 & \text{otherwise} \end{cases}$$

and plot partial sums.

3. Suppose that

$$S_n(x) = \sum_{i=-1}^{n} \sum_{j>-\infty}^{\infty} A_{k,j} w(2^k x - j)$$

is a partial sum of a wavelet-series, as in 8.2.4 on page 168. Show that this partial sum is equal to

$$S_n(x) = \sum_{j>-\infty}^{\infty} B_{n,j} \phi(2^n x - j) \tag{8.3.2}$$

so that wavelet-series correspond to series of *scaling functions*.

4. Write a routine to compute $D_6(x)$ (using the parameters in equation 8.3.1 on page 168) and plot them at dyadic points.

CHAPTER 9

Graph Theory

"The origins of graph theory are humble, even frivolous... The problems which led to the development of graph theory were often little more than puzzles, designed to test the ingenuity rather than the stimulate the imagination."
— Norman L. Biggs ([7]).

9.1. Königsberg bridge problem

Graph theory is the study of combinatorial structures similar to the diagram in figure 7.4.1 on page 134. The idea is that we have points, called *nodes (or vertices)*, connected by arcs, called *edges*, and we are interested with many questions that arise.

The great Leonhard Euler invented graph theory to solve the famous Königsberg bridge problem. See figure 9.1.1.

The Pregel River flows through the city of Königsberg (now called Kaliningrad) and around the two islands of Kneiphof and Lomse. There are seven bridges (marked in figure 9.1.1) connecting the north and south banks of the river and these two islands. The Königsberg bridge problem is:

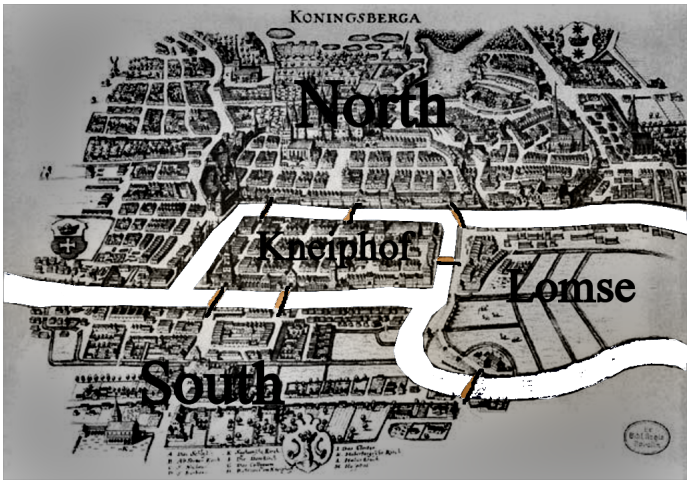


FIGURE 9.1.1. Königsberg (1736)

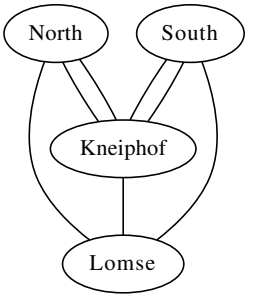


FIGURE 9.1.2. Graph for Königsberg

> How can one visit the two banks and two islands by traversing each of the bridges *exactly once?*

Euler replaced the diagram in figure 9.1.1 on the previous page by the more abstract diagram — a graph — in figure 9.1.2, realizing that the shapes of the respective landmasses were unimportant. The only thing that mattered was how they were interconnected.

At this point, Euler realized that if you enter a node or vertex (synonymous with node) by one edge and leave it by a *different* edge, there must be an *even* number of edges incident on that node. The only exceptions would be the nodes where the path *started* and *ended,* which could have an odd number of incident edges. In the diagram in figure 9.1.2, all four vertices have an *odd* number of edges, so there *cannot* be a path through the diagram that hits each edge once.

A path in a graph that crosses each edge exactly once is called an *Euler path* or *Eulerian path*. Euler showed that a *necessary* condition for the existence of an Eulerian path is that all vertices except two must have an *even* numbers of edges incident on them. He also stated but didn't prove that this condition was *sufficient* for the existence of an Eulerian path. The first complete proof of this latter claim (and an algorithm for finding an Euler path) was published posthumously in 1873 by Carl Hierholzer in [31] — also see [7].

An Eulerian *circuit* or *cycle* is a closed path. For the existence of such, it is necessary and sufficient for for all vertices to have an even number of edges incident on them.

Nowadays, graph theory is used for analyzing networks of all kinds: computer networks, highway systems, electrical grids, etc.

The type of graph in figure 9.1.2 is called a *multigraph* since it has more than one edge connecting the same two nodes. A graph with at most one edge connecting two nodes is said to be *simple*.

This gives us an opportunity to introduce a new Maxima programming construct: *structures.* See section E.10 on page 310.

A *structure* is a kind of list where the entries in the list have *names* and are referred to by those names. We define types of structures with the **defstruct**-command:

```
defstruct(structure_name(name_1,...,name_n));
```

We create structures via the **new-**command

```
z:new(structure_name(value_1,...,value_n));
```

Example:

```
defstruct(dog(legs,eyes,color));
```

This defines a *kind* of structure called 'dog'. We create a *actual* structure (i.e., a 'dog') with

```
fido:new(dog(4,2,"brown"));
```

We access these fields via the '@'-command:

```
fido@legs;
```

and Maxima returns 4. If we simply type fido, Maxima prints

```
dog(legs=4,eyes=2,color="brown");
```

Note that one cannot create a new structure using these '=' commands.

After a *tragic accident*

```
fido@legs:3;
```

and, if we simply type fido, we get

```
dog(legs=3,eyes=2,color="brown");
```

With multigraphs, we can

```
defstruct(mgraph(V, E));
```

where V is set of vertices and E is a set of edges (see section E.11 on page 311 on sets). Given this, we define

```
konigsberg: new(mgraph({A,B,C,D},
                       {[a,{A,B}],
                        [b,{A,B}],
                        [c,{A,C}],
                        [d,{A,C}],
                        [e,{A,D}],
                        [f,{B,D}],
                        [g,{C,D}]}));
```

Each edge is a list

$$[\text{name}, \{\text{set of endpoints}\}]$$

Now

```
konigsberg@V
```

prints

$$\{A, B, C, D\}$$

and

```
konigsberg@E
```

prints

$$\left\{ [a, \{A, B\}], [b, \{A, B\}], [c, \{A, C\}], \right.$$
$$\left. [d, \{A, C\}], [e, \{A, D\}], [f, \{B, D\}], [g, \{C, D\}] \right\}$$

To access the *endpoints* of an edge, we use the **assoc**-command

```
assoc(a, konigsberg@E)
```

which prints out

$$\{A, B\}$$

A *path* through a graph is a sequence of vertices and edges

$$[v_0, e_1, v_1, \ldots, v_{i-1}, e_i, v_i, \ldots, e_n, v_n]$$

where the endpoints of e_i are v_{i-1} and v_i. We can write a function to recognize valid paths through a multigraph:

```
is_path(G, P):= block(
 [result: true],
 for i: 2 step 2 thru (length(P)-1) do (
   result: result and
     is({P[i-1], P[i+1]} = assoc(P[i], G@E))
       ),
 return(result)
     )
```

An Euler path is one for which

(1) no edge is repeated
(2) every edge in the graph appears.

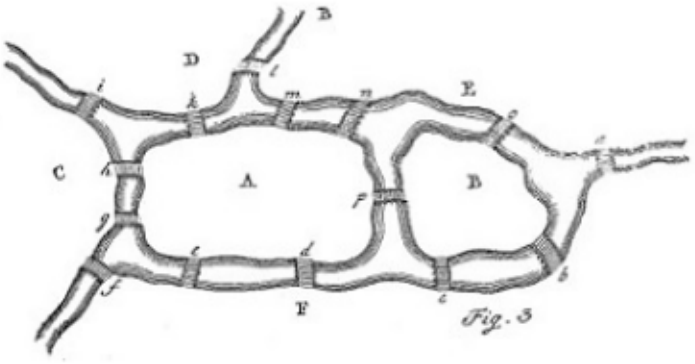


FIGURE 9.1.3. "Eulersberg"

```
edges_unique(G,P):= block(
 [edge_set:{},val:true],
 for i: 2 step 2 thru (length(P)-1) do (
  if(elementp(P[i],edge_set)) then
        (val:false,return(false))
        else edge_set:adjoin(P[i],edge_set)
        ),
 return(val)
     );
```

Note that the **return** statement *in* the **do**-loop *only* exits that *loop*, not the entire **block**.

So: if we know that all edges are unique, we can test whether all edges of the graph are used by checking the length of the path:

```
eulerian(G,P):= (
          is(is_path(G, P)) and edges_unique(G,P)
          and is(length(P) >= 2*length(G@E))
        )
```

In his paper, [21], Euler proposed a variation of Königsberg with extra bridges for which an Eulerian path *does* exist — see figure 9.1.3.

We "declare" Eulersberg via the code in algorithm 1 on the following page. And we define Euler's magic path, via

```
s: "EaFbBcFdAeFfCgAhCiDkAmEnApBoElD";
magic: (eval_string, charlist(s))
```

This allows us to discuss new commands:

Algorithm 1 Declaration of Eulersberg

```
eulersberg: new(mgraph({A,B,C,D,E,F},
                       {[a,{E,F}],
                        [b,{B,F}],
                        [c,{B,F}],
                        [d,{A,F}],
                        [e,{A,F}],
                        [f,{C,F}],
                        [g,{A,C}],
                        [h,{A,C}],
                        [i,{C,D}],
                        [j,{D,E}],
                        [k,{A,D}],
                        [l,{E,D}],
                        [m,{A,E}],
                        [n,{A,E}],
                        [o,{B,E}],
                        [p,{A,B}]
                        }));
```

- ▷ **charlist** — converts a string into a list of characters, which are strings of length 1.
- ▷ **eval_string** — takes a string and *evaluates* it as if you typed it into wxMaxima and clicked "evaluate cell". In this case, it turns strings of length 1 into Maxima variables or symbols. By default, Maxima displays strings without quotes so they look like variables (which can be confusing, since strings are *not* variables). The variable **stringdisp** controls this. By default, it is **false**. Setting it to **true** causes strings to be displayed with double-quotes.
- ▷ **map** — executes **eval_string** on each element of the list, **charlist**(s).

Now we use our function to verify that 'magic' is, indeed, an Eulerian path:

```
eulerian(eulersberg,magic)
```

which returns **true**.

Given an Eulerian mgraph, we'd like a way to construct an Euler tour. We will follow Hierholzer 's algorithm (in [31]), *recursively:*

(1) We select an edge incident on the starting vertex
(2) we locate the *other* end of this edge: newvert
(3) we remove this edge from the graph (or put it on a forbidden list)

Algorithm 2 Euler tour (Hierholzer 's algorithm)

```
euler_path(mgr,vert) := block(
    [sofar:[],n_edges:cardinality(mgr@E),
       temp:[]],
    sofar:partial_path(mgr,vert),
    while (cardinality(forbidden)<n_edges) do (
        for i:1 step 2 thru length(sofar)-1 do (
            newvert:sofar[i],
            temp:partial_path(mgr,newvert),
            s_len:length(sofar),
            if(length(temp)>0) then
            (sofar:append(firstn(sofar,i-1),
              temp,lastn(sofar,s_len-i+1))),
 /* splice the new partial_path into sofar */
            temp:[]
        )
    ),
sofar
);
```

(4) we continue the search from newvert and the remaining edges of the graph (i.e., call euler_path, again)
(5) we adjoin the starting vertex and edge to the path returned by this *recursive* call and *return* the result.
(6) If we get *stuck* before visiting all of the edges in the graph, we rerun steps 1-4 on a vertex of the *partial path* we have constructed. Iterate through the edges of the partial path until all edges are visited.

The code is in figure 2 and 3 on the following page.

Notes:

▷ We get the other end of the current edge via

```
newvert:first(listify(disjoin(vert,last(x)))),
```

Here, **last**(x) extracts the two-element set of endpoints of the edge given by **first**(x), and **disjoin** deletes the starting vertex from it. We now have a one-element set containing the *other* end of that edge. We extract that one element via **listify** and **first**. The function **first** alone may work, but future versions of Maxima might no long support **first** for sets.

The main program partial_path in 3 on the next page.

Algorithm 3 Partial path (Hierholzer 's algorithm)

```
forbidden:{};
partial_path(mgr,vert):=block(
    [x,keepgoing:true,retval:[]],
    for x in mgr@E while (keepgoing) do (
    if (not((elementp(first(x), forbidden) ))
            and elementp(vert, last(x))) then
    (block (
           [newvert:first(listify(disjoin(vert,
                                  last(x)))),
            /* vertex at end of current edge */
                    val],
            forbidden:adjoin(first(x),forbidden),
            /* Put the current edge in forbidden */
            keepgoing:false,
            /* Stop for-loop after an edge located */
            val:partial_path(mgr,newvert),
            /* Continue the search with
               the rest of the graph */
            retval:append( [vert,first(x)],val))
        )
        ),
    retval /* Return with this value */
);
```

9.2. Simple graphs

9.2.1. Introduction. Most graphs used in applications are *simple:* they only have a single edge between two vertices. Maxima has a very comprehensive library of routines for handling these graphs see Appendix G on page 345.

We begin with a few definitions:

(1) *degree of a vertex* — the number of edges attached to it. For a directed graph, we have *in-degree* and *out-degree*.
(2) *walk* — a path through a graph that can contain repeated edges and vertices
(3) *trail* — a walk with no repeated edges.
    (a) *Eulerian trail* — one that contains every edge in the graph. Euler stated but didn't prove that an Eulerian trail exists if and only if the graph has precisely two vertices of odd degree.
(4) *circuit* — a path through a graph that begins and ends at the same vertex, and never repeats an edge.

(a) *Eulerian circuit* — visits every edge of the graph. Euler stated but didn't prove that a graph has an Eulerian circuit if and only if the degree of each vertex is even. This, and the earlier claim of Euler were proved by Carl Hierholzer in 1873, who developed an algorithm for computing Euler paths and circuits — see [7].
(b) *Hamiltonian circuit* — visits every vertex of the graph exactly once.

(5) *cycle* — a circuit that never repeats a vertex. Every cycle is a circuit but the converse is not necessarily true. On the other hand, a Hamiltonian circuit is a cycle.
(6) A *matching* of a graph is a selection of edges, no two of which are incident on the same vertex. It's called a matching because each edge pairs up (i.e. "matches") its endpoints.

We type:

```
load(graphs);
```

Now we illustrate the kinds of problems graph theory can solve.

EXAMPLE 9.2.1. What is the largest set of integers from 1 to 100 with the property that no two integers differ by a *perfect square?*

First, we code a function to recognize a perfect square:

```
square_p[n]:=block(
    [sqval:isqrt(n)],
    is(n=sqval^2)
);
```

Then, we create a graph whose vertices are the integers from 1 to 100 and whose edges connect numbers that differ by a perfect square:

```
sq:make_graph(100,lambda([i,j],square_p[abs(i-j)]));
```

Now we identify an independent set in this graph: an *independent set* is a set of vertices no two of which are connected by an edge in that graph. We actually want a *maximal* independent set in the graph, sq.

```
indep:max_independent_set(sq);
```

At this point, be prepared to wait: it's well-known that the problem of finding a maximum independent set is very hard[1] (even NP-hard — see [48]). We eventually get a list of 24 numbers:

$$[1,3,6,8,14,20,25,27,32,35,38,40,$$
$$46,53,59,66,80,83,85,88,90,93,98,100]$$

[1]Incidentally, *finding* an independent set is easy. The hard part is finding a *maximal* independent set.

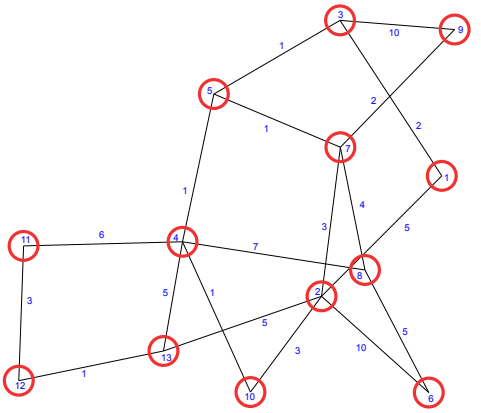


FIGURE 9.2.1. A weighted graph

Now we consider another example: the GPS problem:

EXAMPLE 9.2.2. In this case, the edges of the graph represent highways and streets. It's not enough to represent the network of roads: we must use a *weighted graph* where each edge has a number associated with it called its *weight*. In a GPS system, the weight could represent the time it takes to traverse a road at the posted speed limit. We'll make the simplifying assumption that all roads are two-way.

```
roads:create_graph([1,2,3,4,5,6,7,8,9,10,11,12,13],
[[[1,2],5],[[1,3],2],[[3,9],10],[[2,10],3],
[[3,5],1],[[2,13],5],[[2,7],3],[[4,8],7],
[[5,7],1],[[4,10],1],[[7,8],4],[[6,8],5],[[2,6],10],
[[4,5],1],[[7,9],2],[[4,11],6],
[[11,12],3],[[4,13],5],[[12,13],1]  ]
);
```

We can see what this graph looks like via the command

```
draw_graph(roads,show_id=true,show_weight=true,
    vertex_type=circle,vertex_size=4);
```

which produces figure 9.2.1. Note: **vertex_type** and **vertex_size** are coded because the default display is not very readable. The **show_id** command causes the vertex-numbers to be displayed.

Now, if we want the shortest weighted path[2] from vertex 1 to vertex 11, we type

```
gps:shortest_weighted_path(1,11,roads);
```

[2]There's also a **shortest_path** command that gives the path with the *fewest edges* (regardless of weight). This produces a *different* outcome in this case (try it!).

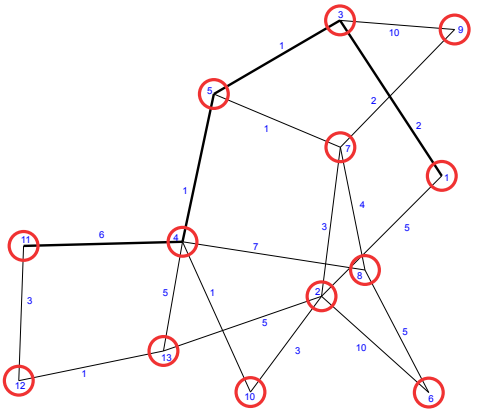


FIGURE 9.2.2. Shortest weighted path

which results in

(9.2.1) $[10, [1, 3, 5, 4, 11]]$

where the first number is just the "distance" or total weight of the path, and the second list is the list of vertices.

We can display this path in the image of the graph via the command

```
draw_graph(roads,show_id=true,show_weight=true,
    show_edges=vertices_to_path(last(gps)),
    show_edge_width=5,
    vertex_type=circle,vertex_size=4);
```

which produces figure 9.2.2.

Notes: the option, **show_edges,** causes some edges to be displayed in a different format than the others, determined by the **show_edge_width** command. The command **last**(gps) extracts the list of vertices from equation 9.2.1, and **vertices_to_path** converts this list of vertices into a list of edges.

EXERCISES.

1. Why does the command for generating the graph in example 9.2.1 on page 179 have a **lambda**-expression? Why couldn't it just be

```
sq:make_graph(100,square_p[abs(i-j)]);
```

?

9.3. The Traveling Salesperson Problem

9.3.1. Introduction. This is a classic problem in graph theory. The background story:

> A salesman must travel to, say, 100 cities around the country pitching his wares. How can this travel be done in a manner that the total distance covered is a *minimum?*

Modern formulation:

> Given a complete weighted graph on n vertices, find the minimum-total-weight Hamiltonian path through it.

This type of problem frequently occurs in transportation networks:

- ▷ the problem of arranging school bus routes to pick up the children in a school district (Vehicle Routing Problem).
- ▷ the scheduling of a machine to drill holes in a circuit board or other object, the cost of travel is the time it takes to move the drill head from one hole to the next.
- ▷ routing workers through a warehouse to assemble orders (for a company like Amazon).

Unfortunately, the traveling salesperson problem is well-known to be NP-complete (see [48]), which means it is computationally very difficult.

We start with a complete graph whose vertices represent 6 cities:

- ▷ New York
- ▷ Chicago
- ▷ New Orleans
- ▷ Los Angeles
- ▷ Phoenix
- ▷ Denver

and weight it by the distances between these cities in miles. We use the command

```
draw_graph(z,
    show_weight=true,
    show_label=true,
    program=fdp,
    vertex_type=circle,
    vertex_size=5,redraw=true)
```

to get figure 9.3.1 on the next page. Recall that 'fdp' is part of the free 'graphviz' package that can be downloaded from `https://graphviz.org`. It does a much better job of displaying the graph.

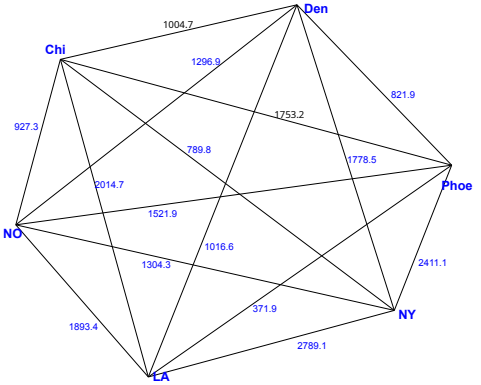


FIGURE 9.3.1. Complete weighted graph

9.3.2. Exhaustive search. The only method guaranteed to produce an optimal solution is exhaustive search. This is easily programmed in Maxima using some set-theoretic commands. The most straightforward code involves generating a list of all permutations of the vertices:

```
paths:permutations(vertices(g));
```

which produces a *set* containing *lists* of permuted vertices. Each represents a possible tour through the complete graph.

For each list of vertices, we compute the total distance traveled

```
total_length(li,g):=block (
          [len:length(li),i,t_len:0],
          for i:1 step 1 thru len-1 do (
            t_len:t_len+
             get_edge_weight([li[i],li[i+1]],g)
          ),
        t_len:t_len+
             get_edge_weight([li[len],li[1]],g),
        return(t_len)
      );
```

Now we merely cycle through all permutations of the vertices of our graph and find the one with the shortest total length. The code for this appears in algorithm 4 on the following page. For our sample graph, this prints out

Algorithm 4 The Traveling salesman Problem

```
show_labels(li,g):=
                for i in li do
                 block(
                  [lab:get_vertex_label(i,g)],
                  display(lab)
                       );

exhaustive(g):=block(
             [paths:permutations(vertices(g)),
                          first_time:true,
                          s_dist:0,
                          s_path:[]],
         for p in paths do block(
         [current_dist:total_length(p,g)],
         if first_time then
              block(s_dist:current_dist,s_path:p),
         first_time:false,
         if current_dist < s_dist then block(
                                 [],
                             s_dist:current_dist,
                             s_path:p
                          )
            ),
             display(s_dist,s_path),
             show_labels(s_path,g)
           );
```

```
s_dist=6009.2
s_path=[0,1,5,2,4,3]
lab="NY"
lab="Chi"
lab="Den"
lab="LA"
lab="Phoe"
lab="NO"
```

so our shortest tour is

$$\text{NY} \rightarrow \text{Chi} \rightarrow \text{Den} \rightarrow \text{LA} \rightarrow \text{Phoe} \rightarrow \text{NO}$$

with a total length of roughly 6009 miles.

Although this program had no trouble solving the traveling salesman problem, there were only 6 cities, giving rise to $6! = 720$ permutations to be checked. We could decrease this number by noting

that *cyclic* permutations of a path give the same path — so we have $6!/6 = 5! = 120$. With n cities, there are $(n-1)!$ possibilities, and for large n this quickly goes beyond the capacity of a computer. For instance, if we have 100 cities, we must still consider 99! cases — a very large number[3]!

It's not hard to see that we can cyclically permute every permutation of

$$[0, \ldots, n-1]$$

to 0 followed by a permutation of

$$[1, \ldots, n-1]$$

so we can generate these $(n-1)!$ tours in this fashion.

EXERCISES.

1. Rewrite the code in this section to only test the $(n-1)!$ permutations: 0 followed by a permutation of

$$[1, \ldots, n-1]$$

2. The code in this section requires all of the permutations to be stored in memory at the same time. The combinatorics package — loaded via **load**("combinatorics") — has a **perm_next**(permutation)-function that generates permutations *one by one.* Rewrite the code in this section to use that, so only one permutation at a time must be in memory. If **perm_next**(permutation) gives you the *next* permutation, what is the *first* one?

9.3.3. An approximation algorithm. *Practical* algorithms for the traveling salesman problem involve *approximation algorithms* for tours, especially those based on minimal spanning trees:

DEFINITION 9.3.1. Given a weighted graph, G, a *minimal spanning tree*, T, of G is a subgraph of G that

- ▷ is a *tree* (i.e., it has no cycles),
- ▷ it *spans* G, i.e., it contains *all* of the vertices of G,
- ▷ has a *minimal total weight* among the (many!) possible spanning trees.

[3]Type it into Maxima!

REMARK. It turns out that there are fast and efficient algorithms for finding minimum spanning trees. Besides the traveling salesman problem, minimum spanning trees are widely used in network design.

In a weighted graph, G, the total weight of a subgraph $H \subset G$ will be called $w(H)$. Suppose $Z \subset G$ is a minimum-weight salesman's tour. Then deleting one edge from Z will form a spanning tree (a special type, with only *one path).* If $T \subset G$ is a minimum spanning tree, it follows that

$$w(T) \leq w(Z)$$

In fact, if $E \subset Z$ is the edge with the greatest weight, then

$$w(T) \leq w(Z) - w(E)$$

since we could delete E from Z to turn it into a spanning tree.

So our strategy for approximating a minimum salesman's tour is:

(1) create a complete graph of all the cities the salesman must visit, weighted by distances between the cities (this has already been done in section 9.3.1 on page 182),
(2) find a minimum spanning tree, T, of this,
(3) double all the edges of this graph to form an mgraph,
(4) find an Euler tour of this (using the algorithm in 2 on page 177 and 3 on page 178). This will have a total weight of $2w(T) < 2w(Z)$,
(5) "short-cutting" this path by eliminating duplicate vertices. This results in a tour whose total weight is $< 2w(T) < 2w(Z)$.

Now we get a minimum spanning tree of this via

```
t:minimum_spanning_tree(z);
```

and draw this via

```
draw_graph(z,
     show_weight=true,
     show_label=true,
     show_edges=edges(t),
     show_edge_width=5,
     program=fdp,
     vertex_type=circle,
     vertex_size=5,redraw=true)
```

to get figure 9.3.2 on the facing page.

At this point, we create an mgraph with *two* edges for each edge of the minimum spanning tree in algorithm 5 on the next page and

```
m:double_up(z,t);
```

returns

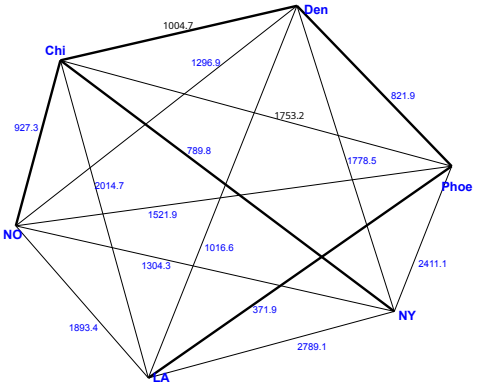


FIGURE 9.3.2. Minimum spanning tree

Algorithm 5 Converting a tree into a multigraph

```
double_up(gra,tree):=block([mgr,count:0],
/* We use 'gra' for vertex-labels
   (so vertices of 'gra' must have labels */
    mgr:new(mgraph({},{})),
    for x in vertices(tree) do
   block(
    [lab],
      lab:get_vertex_label(x,gra),
      mgr@V:adjoin(eval_string(lab),mgr@V)
    ),
for e in edges(tree) do block(
[i1:e[1],i2:e[2],lab1,lab2,name_for,name_back],
    count:count+1,
    lab1:eval_string(get_vertex_label(i1,gra)),
    lab2:eval_string(get_vertex_label(i2,gra)),
    name_for:eval_string(concat("e",count)),
    name_back:eval_string(concat("f",count)),
    mgr@E: adjoin([name_for,set(lab1,lab2)],mgr@E),
    mgr@E: adjoin([name_back,set(lab1,lab2)],mgr@E)
),
    return(mgr)
);
```

```
mgraph(V={Chi,Den,LA,NO,NY,Phoe},
                E={[e1,{Chi,Den}],[e2,{Chi,NO}],
                   [e3,{Den,Phoe}],[e4,{Chi,NY}],
                   [e5,{LA,Phoe}],[f1,{Chi,Den}],
                   [f2,{Chi,NO}],[f3,{Den,Phoe}],
                   [f4,{Chi,NY}],[f5,{LA,Phoe}]})
```

At this point, we can find an Euler Tour of this with

```
euler_path(m,NY);
```

to get

```
[NY, e4 , Chi , e1 , Den , e3 , Phoe , e5 , LA, f5 ,
Phoe , f3 , Den , f1 , Chi , e2 , NO, f2 , Chi , f4 ]
```

Now we *short-cut* this by eliminating duplicate vertices to get our tour

$$\text{NY} \to \text{Chi} \to \text{Den} \to \text{Phoe} \to \text{LA} \to \text{NO} \tag{9.3.1}$$

and back to NY, roughly 6186 miles — 177 miles longer than the optimal tour.

Although this code looks more complicated that that in section 9.3.2 on page 183, it executes much more rapidly when the number of cities is large.

Many algorithms for finding optimal routes start with an *approximate* solution like that in 9.3.1 and improve it by making many small changes — see [29]. These methods were used to get an optimal solution through 15,112 German cities (Applegate, Bixby, Chvátal, and Cook, 2001). The current record is an 85,900-city tour that arose in a chip-design application.

EXERCISES.

3. *Muddy city problem:* Once upon a time there was a city (figure 9.3.3 on the facing page) that had no roads. Getting around the city was particularly difficult after rainstorms because the ground became very muddy, and cars got stuck in the mud, and people got their boots dirty. The mayor of the city decided that some of the streets must be paved but specified two conditions:

- ▷ Enough streets must be paved so that it is possible for everyone to travel from their house to anyone else's house only along paved roads, and
- ▷ The paving should cost as little as possible.

The number of paving stones along each path in this diagram — figure 9.3.3 on the next page — represents the cost of paving that path (the bridge does not need to be paved).

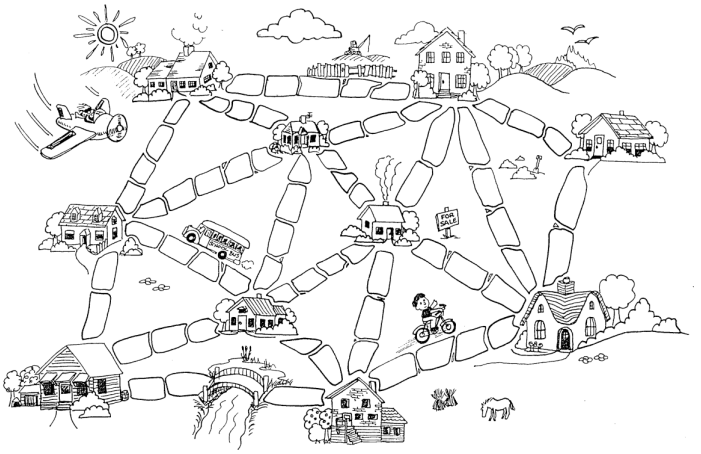


FIGURE 9.3.3. The Muddy City

CHAPTER 10

Calculus of Finite Differences

"In the following exposition of the Calculus of Finite Differences, particular attention has been paid to the connexion of its methods with those of the Differential Calculus — a connexion which in some instances involves far more than a merely formal analogy."
— George Boole, in the introduction to [9].

10.1. A discrete introduction to finite differences

In this chapter, we will consider a *discrete* analogue to differential calculus. Isaac Newton invented it at roughly the same time as he invented calculus and used it to perform exacting numerical computations.

Isaac Newton FRS[a](1642 – 1726/27) was an English polymath active as a mathematician, physicist, astronomer, alchemist, theologian, and author who was described in his time as a natural philosopher. He was a key figure in the Scientific Revolution and the Enlightenment that followed. His pioneering book *Philosophiæ Naturalis Principia Mathematica* (Mathematical Principles of Natural Philosophy), first published in 1687, consolidated many previous results and established classical mechanics. Newton also made seminal contributions to optics, and shares credit with the German mathematician Gottfried Wilhelm Leibniz for developing infinitesimal calculus, though Newton developed calculus years before Leibniz (and gave it the more appropriate name, *fluxions*). He is considered one of the greatest and most influential scientists in history.

[a]Fellow of the Royal Society

As we have seen, Maxima has built-in commands **diff** and **integrate** (among others) for ordinary calculus, while it has *none* for finite-difference calculus. We'll program these commands using a very powerful Maxima programming language feature called *macros.*

What commands do we need to implement? The most basic ones are:

DEFINITION 10.1.1. If $f(x)$ is a function, define

(1) $\mathbf{E}(f)(x) = f(x+1)$

(2) $\mathbf{\Delta} f(x) = f(x+1) - f(x) = \mathbf{E}(f) - f$ or $\mathbf{\Delta} = \mathbf{E} - 1$.
(3) $\mathbf{\Sigma}_a^b f(x) = \sum_{x=a}^{b} f(x)$ (we assume a, b, and x are integers).

REMARK. The operations $\mathbf{\Delta}$ and $\mathbf{\Sigma}$ are approximate inverses because of the *Fundamental Theorems* of finite-difference calculus:

$$\mathbf{\Sigma}_a^b \mathbf{\Delta} f(x) = \mathbf{E}(f)(b) - f(a) \tag{10.1.1}$$
$$\mathbf{\Delta}\mathbf{\Sigma}_a^x f(x) = \mathbf{E}(f)$$

Because of this, $\mathbf{\Sigma}$ is sometimes written as $\mathbf{\Delta}^{-1}$.

One breakthrough in differential calculus occurred when it was discovered that

$$\frac{dx^n}{dx} = nx^{n-1}$$

so that

$$\int x^n dx = \frac{x^{n+1}}{n+1}$$

Something similar happens in finite difference calculus with *Pochhammer* symbols or *falling factorials:*

DEFINITION 10.1.2. If n is an integer and x is a real number, then the *falling factorial* of x is defined by

$$x_{(n)} = \begin{cases} x \cdot (x-1) \cdots (x-n+1) & \text{if } n \geq 1 \\ 1 & \text{if } n = 0 \\ \frac{1}{(x+1)\cdots(x-n)} & \text{if } n < 0 \end{cases}$$

REMARK. Falling factorials were initially only defined for *positive* values of n. This is easily extended by noting that

$$\frac{x_{(n+1)}}{x_{(n)}} = x - n$$

which implies (setting $n = 0$) that $x_{(0)} = 1$, and

$$x_{(-1)} = \frac{1}{x+1} \tag{10.1.2}$$

At the end of section 14.1 on page 245 this definition is extended to arbitrary *complex* values of n.

Leo August Pochhammer (1841 – 1920) was a Prussian mathematician who was known for his work on special functions and introducing the Pochhammer symbol.

PROPOSITION 10.1.3. *Let n be an integer and let $x \in \mathbb{C}$. Then*

$$\mathbf{\Delta} x_{(n)} = n \cdot x_{(n-1)} \tag{10.1.3}$$

Since this has the same structural properties as derivatives of functions like x^n, we can prove the *finite-difference* version of the *Taylor Series:*

$$f(x) = f(a) + \mathbf{\Delta}[f](a)(x-a)_{(1)} + \frac{\mathbf{\Delta}^2[f](a)(x-a)_{(2)}}{2!} \tag{10.1.4}$$

$$+ \frac{\mathbf{\Delta}^3[f](a)(x-a)_{(3)}}{3!} + \cdots \tag{10.1.5}$$

called *Gregory–Newton interpolation formula*. Newton proved its validity for f a polynomial.

James Gregory FRS[a] (1638 – 1675) was a Scottish mathematician and astronomer. His surname is sometimes spelled as Gregorie, the original Scottish spelling. He described an early practical design for the reflecting telescope — the Gregorian telescope — and made advances in trigonometry, discovering infinite series representations for several trigonometric functions.
In his book *Geometriæ Pars Universalis* (1668) Gregory gave both the first published statement and proof of the fundamental theorem of calculus (from a geometric point of view, and only for a special class of curves).

[a]Fellow of the Royal Society

It's not hard to see that, if $a \in \mathbb{R}$ then

$$\mathbf{\Delta}a^x = (a-1)a^x$$

so that

$$\mathbf{\Delta}2^x = 2^x$$

It follows that 2^x is the finite-difference version of e^x in regular calculus.

Let $\mathbf{D}$ denote the differential operator, so

$$(\mathbf{D}f)(x) = f'(x)$$

The *conventional* Taylor series implies that

$$f(x+1) = f(x) + \mathbf{D}f(x)\cdot 1 + \frac{\mathbf{D}^2 f(x)}{2!}1^2 + \cdots$$

or

$$f(x+1) = \left(1 + \mathbf{D} + \frac{\mathbf{D}^2}{2!} + \cdots\right) f(x)$$

Since the bracketed series looks like that for e^x, we conclude

$$\mathbf{E} = e^{\mathbf{D}}$$

and

$$\mathbf{\Delta} = e^{\mathbf{D}} - 1$$

or

$$\mathbf{\Delta} + 1 = e^{\mathbf{D}}$$

from which we conclude

$$\mathbf{D} = \ln(1+\boldsymbol{\Delta}) \tag{10.1.6}$$
$$= \boldsymbol{\Delta} - \frac{\boldsymbol{\Delta}^2}{2} + \frac{\boldsymbol{\Delta}^3}{3} - \frac{\boldsymbol{\Delta}^4}{4} + \cdots$$

It looks as though we have played very fast and loose with these operators, but this equation is valid when applied to any polynomial, and many other functions. One problem is that, even when it converges, it does so very slowly.

The interested reader is referred to Boole's classic, [9], which is still very relevant today.

Incidentally, Boole is famous in his own right for boolean algebras and boolean operations used in designing modern computers.

George Boole (1815 – 1864) was a largely self-taught English mathematician, philosopher, and logician, most of whose short career was spent as the first professor of mathematics at Queen's College, Cork in Ireland. He worked in the fields of differential equations and algebraic logic, and is best known as the author of *The Laws of Thought* (1854) which contains Boolean algebra. Boolean logic is credited with laying the foundations for the Information Age, alongside the work of Claude Shannon.

Now we'll investigate *Harmonic numbers*

$$H(n) = \sum_{i=1}^{n} \frac{1}{n}$$

so $\boldsymbol{\Delta}H(n) = 1/(n+1) = n_{(-1)}$ (see equation 10.1.2 on page 192).

PROPOSITION 10.1.4. *If $k \geq 1$, then*

$$\boldsymbol{\Delta}^k H(n)(1) = (-1)^{k+1}(k-1)!n_{(-k)}(1) = \frac{(-1)^{k+1}}{k(k+1)}$$

PROOF. Induction on k: If true for $k-1$, we have

$$\boldsymbol{\Delta}^{k-1}H(n)(1) = (-1)^k(k-1)!n_{(-k+1)}(1)$$

and

$$\begin{aligned}\boldsymbol{\Delta}^k H(n) &= \boldsymbol{\Delta}\boldsymbol{\Delta}^{k-1}H(n) = -(-1)^k(k-2)!\cdot(k-1)\cdot n_{(-k)}(1)\\ &= (-1)^{k+1}(k-1)!n_{(-k)}(1)\end{aligned}$$

by proposition 10.1.3 on page 192. Definition 10.1.2 on page 192 shows that

$$(-1)^{k+1}(k-1)!n_{(-k)}(1) = \frac{(-1)^{k+1}(k-1)!}{(1+1)(1+2)\cdots(1+k)} = \frac{(-1)^{k+1}}{k(k+1)}$$

□

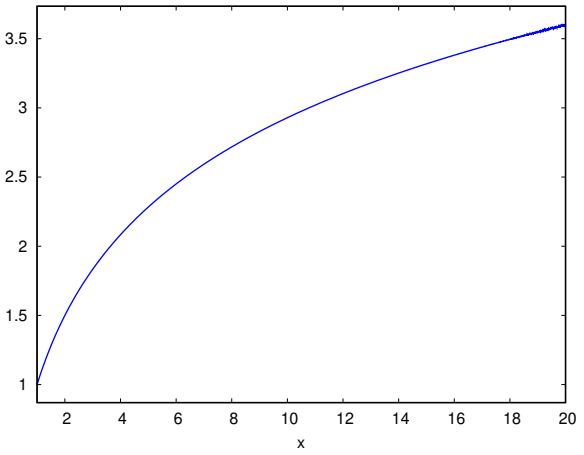


FIGURE 10.1.1. The harmonic numbers

The Gregory-Newton series for harmonic numbers is, therefore, (10.1.7)

$$H(x) = 1 + \frac{x-1}{2} - \frac{(x-1)_{(2)}}{6 \cdot 2!} + \cdots + \frac{(-1)^{k+1}(x-1)_{(k)}}{k(k+1)k!} + \cdots$$

Now we implement falling factorials — and this series:

```
poch[x,n]:=block(
                  [],
                  if n=0 then return (1),
                  if n=1 then return (x),
                  poch[x,n-1]*(x-n+1)
                 );

h(x):=1+sum((-1)^(n+1)*poch[x-1,n]/(n*(n+1)*n!),
                n,1,50);
```

Plotting this with the command

```
plot2d(h(x),[x,1,20]);
```

gives figure 10.1.1. The lack of wild oscillations shows that the harmonic numbers lend themselves to representation by a *polynomial* (compare this with what happens with prime numbers in the next section!). See exercise 4 on page 248 (and its solution) for an extension of $H(x)$ to the entire complex plane.

EXERCISES.

1. Compute

$$\sum_{x=0}^{n} x_{(-i)}$$

where $i \neq 1$, and x is an integer. How do we handle the case where $i = 1$?

2. For $n \geq 0$ an integer, show that x^n can be written as a linear combination of falling factorials. Hint: use induction on n.

3. Prove equation 10.1.3 on page 192.

4. Show that

$$\frac{x_{(n)}}{n!} = \binom{x}{n}$$

5. Show that equation 10.1.4 on page 193 is true for all polynomials.

6. Prove the *Product Formula*

$$\mathbf{\Delta}(fg) = \mathbf{\Delta}(f)\mathbf{E}(g) + f\mathbf{\Delta}(g) \tag{10.1.8}$$

or

$$\mathbf{\Delta}(fg) = f\mathbf{\Delta}g + g\mathbf{\Delta}f + \mathbf{\Delta}f\mathbf{\Delta}g$$

7. Prove *Summation by Parts*

$$\mathbf{\Sigma}_m^n f\mathbf{\Delta}g = \mathbf{E}(fg)(n) - (fg)(m) - \mathbf{\Sigma}_m^n \mathbf{E}(g)\mathbf{\Delta}f \tag{10.1.9}$$

where $\mathbf{E}(g)(x) = g(x+1)$.

8. Is there a way to make equation 10.1.6 on page 194 rigorous?

9. Find a closed-form equation for

$$\sum_{k=0}^{n} k \cdot 2^k$$

Hint: Use summation by parts.

10. Prove *Euler's Formula for harmonic numbers*

$$H_n = \int_0^1 \frac{1-t^n}{1-t} dt$$

This allows us to define H_n for *n not* an integer. For instance

$$H_{1/2} = 2 - 2\log(2) = 0.6137056388801094\ldots$$

10.2. Functional Programming and Macros

In this section, we'll implement the operators defined in the previous section in Maxima and apply them. Ideally, these operators could be implemented in a *functional programming language:*

> A programming language is said to be *functional* if functions can be treated as *data-items:* i.e., they can be passed to other functions as parameters or arguments, and a function can return a *function* as its *value.*
>
> There are many functional languages today: Scheme (an interesting dialect of Lisp), Common Lisp, Python, Haskell, etc. One might wonder why Maxima doesn't simply "inherit" functional programming from Common Lisp. The problem is that Maxima was originally written for Maclisp, an early form of Lisp *without* functional programming.
>
> Oddly, no volunteers have stepped up and offered to rewrite Maxima (from scratch!) in modern Lisp to implement functional programming.
>
> In a *functional* language, we could write

```
Bdelta(f,x):=lambda([f,x],f(x+1)-f(x))
```

> and

```
Bdelta(sin,x)
```

> would return

```
lambda([sin,x],sin(x+1)-sin(x))
```

> whereas Maxima *actually* returns

```
lambda([f,x],f(x+1)-f(x))
```

It turns out we can "fake" features of a functional language using constructs called *macros.* When $f(x,y,z)$ is called, Maxima

(1) evaluates x, y, z
(2) *jumps* to the function-code and plugs those values into the body of f.
(3) then *returns* with the computed values

A *macro* $f(x,y,z)$ superficially resembles a function but it

(1) executes the *body* of the macro on x, y, z and other expressions (doing something like a text-edit), *inserting* it into the code where it was called — i.e., no *jumping* and *returning,*
(2) then it executes the revised expression.

Here's our code for the **Δ**-operator using a macro built in to Maxima called **buildq**:

```
Bdelta(f,x) := buildq([y:x, g:f],
                      lambda([y],g(y+1)-g(y))
                     );
```

The way this executes is:

(1) **buildq** *edits* the **lambda**-expression, replacing y by x and g by f (think of it as a *text-edit*, although the actual algorithm is faster)
(2) then it *executes* it

```
Bdelta(sin,x)
```

produces

```
lambda([x],sin(x+1)-sin(x))
```

and

```
Bdelta(sin,x)(x)
```

produces

```
sin(x+1)-sin(x)
```

Having implemented Δ, we can try to implement Δ^n:

```
Bdeltan(f,x,n):=block([],
                      if n=1 then return (Bdelta(f,x)),
                      Bdelta(Bdeltan(f,x,n-1),x)
                      );
```

For instance, the command

```
Bdeltan(sin,x,3)(x)
```

returns

$$\sin(x+3) - 3\sin(x+2) + 3\sin(x+1) - \sin(x)$$

Now we'll try our hand at the Gregory-Newton series (equation 10.1.4 on page 193)

```
discrete_sin(x):=sum(
                     poch[x,n]*(Bdeltan(sin,x,n)(0))/n!,
                     n,1,20);
```

After doing this, we plot the result with

```
plot2d(discrete_sin(x),[x,0,10])
```

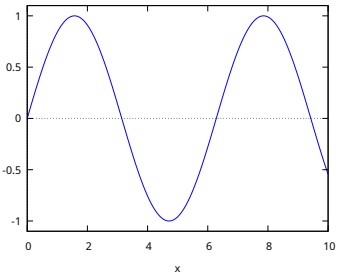


FIGURE 10.2.1. The Gregory-Newton series

to get figure 10.2.1, which shows that the Gregory-Newton series converges for some *non*-polynomial functions.

Now, we will try something more ambitious: a *formula* for prime numbers!

Let

```
primes:[2,3,5,7,11,13,17,19,23,
        29,31,37,41,43,47,53,59,61,67,
        71,73,79,83,89,97,101,103,107,
        109,113,127,131,137,139,149,151,157]
```

denote a bunch of primes. We turn this into a function (because our routines work with functions) via

```
prime_fun(n):=primes[n]
```

and compute a bunch of Δ's via

```
dellist:makelist(Bdeltan(prime_fun,x,n)(1),n,20);
```

— note that we're running this Gregory-Newton series from $a = 1$ rather than 0.

This gives

```
dellist:[1,1,-1,3,-9,23,-53,115,
         -237,457,-801,1213,-1389,
         445,3667,-15081,41335,
         -95059,195769,-370803];
```

Now we define the series itself

```
pol(x):=2+sum(dellist[k]*poch[x-1,k]/k!,k,1,20);
```

This gives a polynomial such that

(1) pol(1)=2
(2) pol(2)=3
(3) pol(7)=17

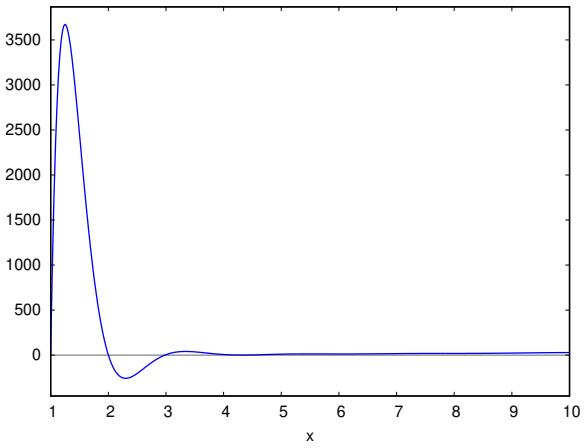


FIGURE 10.2.2. Polynomial giving the first 10 primes

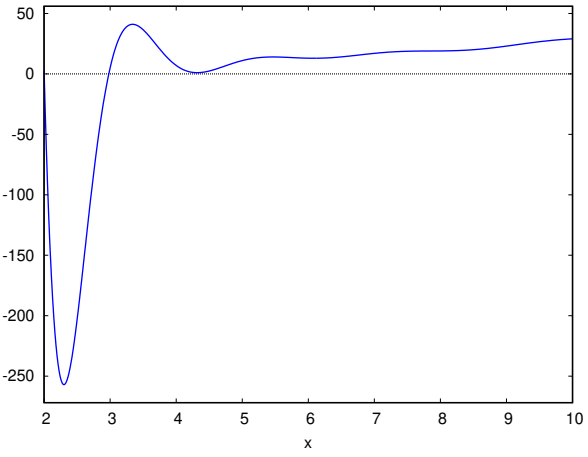


FIGURE 10.2.3. Prime polynomial plot

Now, let's plot it!

```
plot2d(pol(x),[x,1,10])
```

What happened? It looks as though prime numbers don't lend themselves to be expressed by a polynomial. While one can create a polynomial that equals these primes, it oscillates wildly between them.

If we cut out the worst oscillations, we *still* get figure 10.2.3.

See chapter 15 on page 269 for a far more profound discussion of prime numbers.

EXERCISES.

1. Program equation 10.1.6 on page 194 for $\sin(x)$ and $x = 0$.

2. Does the Gregory-Newton series work for *all* analytic functions? Hint: try expanding $\sin(\pi x)$ in a Gregory-Newton series.

3. Find a closed-form equation for $\mathbf{\Delta}^n f(x)$. Hint: write $\mathbf{\Delta}^n = (\mathbf{E} - 1)^n$ and use the Binomial Theorem.

4. Write a non-recursive version of the

Bdeltan

command, using the equation in exercise 3.

10.3. The Euler-Maclaurin Summation formula

In this section, we will explore the difference between

$$\sum_{k=M}^{N} f(k) \text{ and } \int_{M}^{N} f(x)dx$$

In many cases, it's easier to find a closed-form expression for the *integral* but not the *sum* — for instance, if $f(x) = 1/x$.

We begin by defining Bernoulli polynomials:

DEFINITION 10.3.1. If $n > 0$ is an integer, the n^{th} *Bernoulli polynomial*, $B_n(x)$ is defined to be the *unique* polynomial satisfying

$$\int_{x}^{x+1} B_n(t)dt = x^n \tag{10.3.1}$$

In Maxima, the command **bernpoly**(*x, n)* returns the n^{th} Bernoulli polynomial. *Bernoulli numbers* are defined by

$$B_n = B_n(0)$$

and given by the Maxima command **bern**(n).

REMARK. Equation 10.3.1 turns out to imply that

$$B_n'(x) = nB_{n-1}(x)$$

It also turns out that all odd-numbered Bernoulli numbers, except for B_1, *vanish* (this is not obvious!).

Given these definitions, we can state the Euler-Maclaurin summation formula (first published in [19, and, 20]):

$$\sum_{k=M}^{N} f(k) = \int_M^N f(x)dx + \frac{f(M)+f(N)}{2} + \sum_{j=1}^{\nu} \frac{B_{2j}}{(2j)!} f^{(2j-1)}(x)\Big|_M^N + R_{2\nu} \tag{10.3.2}$$

where

$$R_{2\nu} = \frac{1}{(2\nu+1)!}\int_M^N \bar{B}_{2\nu+1}(x) f^{(2\nu+1)}(x)dx \tag{10.3.3}$$

and

$$\bar{B}_{2\nu+1}(x) = B_{2\nu+1}(x - \lfloor x \rfloor)$$

where $\nu > 0$ is some integer. The value of ν is important because the series in equation 10.3.2 does *not* converge for most functions, $f(x)$. The terms get smaller for a time, but ultimately grow without limit. The interesting thing is that the error term, given by equation 10.3.3, is of the same order of magnitude as the first term *omitted* from equation 10.3.2[1]. See [2] for a simplified derivation.

Example: $f(x) = 1/x$. The Euler-Mascheroni constant, γ, is defined by

$$\gamma = \lim_{n\to\infty}\left(\sum_{k=1}^{n}\frac{1}{k} - \log(n)\right) \tag{10.3.4}$$

This process converges *very* slowly, as can be seen via the code

```
cgamma(n):=float(sum(1/k,k,1,n)-log(n));
```

If we compute cgamma(100)−%**gamma**; we get 0.004991666749996071. Now we code up the Euler-Maclaurin equation as

```
em(nu):=float(1/2-
    sum(bern(2*k)/(2*k)!*at(diff(1/x,x,2*k-1),
    x=1),k,1,nu));
```

and em(2)−%**gamma**; gives −0.00221566490153291 which is already more accurate than cgamma(100). The "sweet spot" is em(3)−%**gamma**; which gives 0.001752589066721155 — even more accurate. Larger values of ν give *larger* errors.

Equation 10.3.2 is an example of an *asymptotic expansion* — a divergent series that can be used to *estimate* values of a function. If we could compute the remainder in equation 10.3.3, we'd get an exact answer, but that integral is usually *at least* as hard to evaluate as the original sum.

[1] As with Taylor series.

Since

$$\sum_{n=1}^{N} \frac{1}{k} - \int_1^N \frac{dx}{x} = \sum_{n=1}^{10} \frac{1}{k} - \int_1^{10} \frac{dx}{x} + \sum_{k=10}^{N} \frac{1}{k} - \int_{10}^N \frac{dx}{x} - \frac{1}{10}$$

for $N > 10$, and we can compute γ by *directly* computing

$$\sum_{n=1}^{10} \frac{1}{k} - \log(10) \tag{10.3.5}$$

and using the Euler-Maclaurin formula to compute

$$\lim_{N\to\infty} \sum_{k=10}^{N} \frac{1}{k} - \int_{10}^N \frac{dx}{x} \tag{10.3.6}$$

Since the term $1/10$ appears *twice* in the sum (once in equation 10.3.5 and a second time in equation 10.3.6), we have to subtract off the extra copy.

```
em10(nu):=float(cgamma(10)
      +1/20-sum(bern(2*k)/(2*k)!
        *at(diff(1/x,x,2*k-1),x=10),k,1,nu)-1/10);
```

Since $x = 10$ in the *lower* limit, the derivatives of $1/x$ (and corresponding correction terms) are much smaller[2], and em10(3); is

$$0.5772156649424619$$

which is correct to 10 decimal places. Computing that via equation 10.3.4 on the preceding page would require $n = 10^{10}$.

In [37], Knuth used the Euler-Maclaurin equation to compute γ to 1,271 decimal places. He used $N = 10,000$, and $nu = 250$.

[2]Recall that they are proportional to $1/10^{2k}$.

CHAPTER 11

Nonlinear algebra

> "Cantor illustrated the concept of infinity for his students by telling them that there was once a man who had a hotel with an infinite number of rooms, and the hotel was fully occupied. Then one more guest arrived. So the owner moved the guest in room number 1 into room number 2; the guest in room number 2 into number 3; the guest in 3 into room 4, and so on. In that way room number 1 became vacant for the new guest.
>
> What delights me about this story is that everyone involved, the guests and the owner, accept it as perfectly natural to carry out an infinite number of operations so that one guest can have peace and quiet in a room of his own. That is a great tribute to solitude."
>
> — Smilla Qaavigaaq Jaspersen, in the novel *Smilla's Sense of Snow,* by Peter Høeg (see [32]).

11.1. Introduction

As the previous chapter made clear, complex nonlinear algebraic equations often arise in practical applications.

11.2. Ideals and systems of equations

Define

$$W = \mathbb{C}[X_1, \ldots, X_n]$$

to represent the set of all polynomials in variables $X_1, \ldots, X_n$ with complex coefficients (i.e. coefficients in $\mathbb{C}$)

This mathematical structure is called a *ring*. More formally:

DEFINITION 11.2.1. A *ring*, R, is a set equipped with two binary operations, denoted $+$ and multiplication, $\cdot$, such that, for all $r_1, r_2, r_2 \in R$,

(1) $(r_1 + r_2) + r_3 = r_1 + (r_2 + r_3)$
(2) $(r_1 \cdot r_2) \cdot r_3 = r_1 \cdot (r_2 \cdot r_3)$
(3) $r_1 \cdot (r_2 + r_3) = r_1 \cdot r_2 + r_1 \cdot r_3$
(4) $(r_1 + r_2) \cdot r_3 = r_1 \cdot r_3 + r_1 \cdot r_3$
(5) there exists elements $0, 1 \in R$ such that $r + 0 = 0 + r = r$ and $r \cdot 1 = 1 \cdot r = r$ for all $r \in R$.

(6) For every $r \in R$, there exists an element $s \in R$ such that $r + s = 0$.

The ring R will be called *commutative* if $r_1 \cdot r_2 = r_2 \cdot r_1$ for all $r_1, r_2 \in R$.

A *division ring* is one in which every nonzero element has a multiplicative inverse.

A *subring* $S \subset R$ is a subset of R that is also a ring under the operations $+$ and $\cdot$.

REMARK. We will also regard the set containing only the number 0 as a ring with $0 + 0 = 0 = 0 \cdot 0$ — the *trivial ring* (the multiplicative and additive identities are the same). When an operation is written with a '$+$' sign it is implicitly assumed to be commutative.

EXAMPLE. Perhaps the simplest example of a ring is the integers, $\mathbb{Z}$. This is simple in terms of familiarity to the reader but a detailed analysis of the integers is a very deep field of mathematics in itself (number theory). Its only units are ± 1, and it has no zero-divisors.

We can use the integers to construct:

EXAMPLE. If m is an integer, the numbers modulo m, $\mathbb{Z}_m$ is a ring under addition and multiplication modulo m. In $\mathbb{Z}_6$, the elements 2 and 3 are zero-divisors because $2 \cdot 3 = 0 \in \mathbb{Z}_6$.

EXAMPLE 11.2.2. The rational numbers, $\mathbb{Q}$, are an example of a field. Other examples: the real numbers, $\mathbb{R}$, and the complex numbers, $\mathbb{C}$.

REMARK. We have seen (and worked with) *many* examples of rings before, $\mathbb{Z}$, $\mathbb{Z}_m$, $\mathbb{Q}$, $\mathbb{R}$, and $\mathbb{C}$. The rings $\mathbb{Q}$, $\mathbb{R}$, and $\mathbb{C}$ are *commutative* division rings — also known as *fields.*

DEFINITION 11.2.3. If R is a ring, rings of polynomials $R[X]$ is the ring of polynomials where addition and multiplication are defined

$$\begin{aligned}\left(\sum_{i=0}^{n} a_i X^i\right) + \left(\sum_{i=0}^{m} b_i X^i\right) &= \sum_{i=0}^{\max(n,m)} (a_i + b_i) X^i \\ \left(\sum_{i=0}^{n} a_i X^i\right)\left(\sum_{j=0}^{m} b_j X^j\right) &= \sum_{k=0}^{n+m} \left(\sum_{i+j=k} a_i b_j\right) X^k\end{aligned}$$

with $a_i, b_j \in R$ and $a_i = 0$ if $i > n$ and $b_i = 0$ if $i > m$.

More formally, one can define $R[X]$ as the set of infinite sequences

$$(r_0, \dots, r_i, \dots) \tag{11.2.1}$$

with the property that all but a *finite* number of the r_i vanish, and with addition defined by

$$(r_0, \dots, r_i, \dots) + (s_0, \dots, s_i, \dots) = (r_0 + s_0, \dots, r_i + s_i, \dots)$$

and multiplication defined by

$$(r_0, \dots, r_i, \dots)(s_0, \dots, s_i, \dots) = (t_0, \dots, t_i, \dots)$$

with

$$t_n = \sum_{\substack{i+j=n \\ i\geq 0, j\geq 0}} r_i s_j$$

(the *convolution* of the lists of coefficients — see definition 5.2.1 on page 84). In this case,

$$\sum_{i=0}^{k} r_i X^i$$

becomes the *notation* for the sequence $(r_0, \dots, r_i, \dots, r_k, 0 \cdots)$.

We need one more definition:

DEFINITION 11.2.4. If R is a commutative ring, an *ideal* is a subset closed under addition $\mathfrak{I} \subset R$ such that $x \cdot r \in \mathfrak{I}$ for all $r \in R$.

(1) An ideal, $\mathfrak{I} \subset R$ is *prime* if $a \cdot b \in \mathfrak{I}$ implies that $a \in \mathfrak{I}$ or $b \in \mathfrak{I}$ (or both).
(2) The *ideal generated by* $\alpha_1, \dots, \alpha_n \in R$, denoted $(\alpha_1, \dots \alpha_n) \subseteq R$, is the set of all linear combinations

$$\sum_{k=1}^{n} r_k \cdot \alpha_k \cdot s_k$$

where the r_i and s_i run over all elements of R. The element 0 is an ideal, as well as the whole ring. The set $\alpha_1, \dots, \alpha_n \in R$ is called a basis for the ideal $(\alpha_1, \dots \alpha_n)$.
(3) An ideal $\mathfrak{I} \subset R$ is *maximal* if $\mathfrak{I} \subset \mathfrak{K}$, where $\mathfrak{K}$ is an ideal, implies that $\mathfrak{K} = R$. This is equivalent to saying that for any $r \in R$ with $r \notin \mathfrak{I}$,

$$\mathfrak{I} + (r) = R$$

(4) An ideal generated by a *single element* of R is called a *principal ideal*.
(5) Given two ideals $\mathfrak{a}$ and $\mathfrak{b}$, their *product* is the ideal generated by all products $\{(a \cdot b) | \forall a \in \mathfrak{a}, b \in \mathfrak{b}\}$.

REMARK. We will usually denote ideals by *Fraktur letters.*

EXAMPLE. We claim that the ideals of $\mathbb{Z}$ are just the sets

$$\begin{aligned}
(0) &= \{0\} \\
(2) &= \{\dots, -4, -2, 0, 2, 4, 6, 8, \dots\} \\
(3) &= \{\dots, -6, -3, 0, 3, 6, 9, 12, \dots\} \\
&\vdots \\
(n) &= \{n \cdot \mathbb{Z}\}
\end{aligned}$$

for various values of n. Note that the ideal $(1) = \mathbb{Z}$. An ideal $(n) \subset \mathbb{Z}$ is *prime* if and only if n is a prime number.

Now, suppose $f_1, \dots, f_s \in W = \mathbb{C}[X_1, \dots, X_n]$, and suppose we want to solve the system of algebraic equations

$$\begin{aligned} f_1(X_1, \dots, X_n) &= 0 \\ &\vdots \\ f_s(X_1, \dots, X_n) &= 0 \end{aligned} \tag{11.2.2}$$

If $g_1, \dots, g_t \in W$ is a set of polynomials with the property that

$$(f_1, \dots, f_s) = (g_1, \dots, g_t) = \mathfrak{B}$$

— i.e., the g_j are another basis for the ideal generated by the f_i, then the equations in 11.2.2 are *equivalent* to

$$\begin{aligned} g_1(X_1, \dots, X_n) &= 0 \\ &\vdots \\ g_t(X_1, \dots, X_n) &= 0 \end{aligned}$$

To see that, note that, since the f_i are a *basis* for $\mathfrak{B}$ and the $g_i \in \mathfrak{B}$, we have equations

$$g_i = \sum_{j=1}^{s} a_{i,j} f_j$$

where $a_{i,j} \in W$ for all i and j. It follows that $f_1 = \cdots = f_s = 0$ implies that $g_1 = \cdots = g_t = 0$. Since the g_j are *also* a basis for $\mathfrak{B}$, the *reverse* implication is also true.

EXAMPLE 11.2.5. Suppose we want to find solutions to the system of algebraic equations

$$\begin{aligned} xy &= z^2 \\ xz &= 1 \\ x^2 + y^2 &= 3 \end{aligned}$$

We first make these into equations set to zero

$$\begin{aligned} xy - z^2 &= 0 \\ xz - 1 &= 0 \\ x^2 + y^2 - 3 &= 0 \end{aligned}$$

and find another basis for the *ideal* these polynomials generate. It *turns out*[1] that

$$(xy - z^2, xz - 1, x^2 + y^2 - 3) = (z^8 - 3z^2 + 1, y - z^3, z^7 - 3z + x)$$

[1]This is not at all obvious! Later, we will look at an algorithm for coming to this conclusion.

So our original equations are equivalent to the equations

$$\begin{aligned} z^8 - 3z^2 + 1 &= 0 \\ y - z^3 &= 0 \\ z^7 - 3z + x &= 0 \end{aligned}$$

or

$$\begin{aligned} z^8 - 3z^2 + 1 &= 0 \\ y &= z^3 \\ x &= 3z - z^7 \end{aligned}$$

so that it follows that our original set of equations had *eight* solutions: find 8 roots of the polynomial in z and plug them into the equations for x and y.

It follows that there are applications to finding "simplified" or "improved" bases for ideals in polynomial rings.

11.3. Gröbner bases

One of the most powerful technique for computations in polynomial rings use a special basis for an ideal, called a *Gröbner basis.* Gröbner bases were discovered by Bruno Buchberger.

Bruno Buchberger 1942 –) is Professor of Computer Mathematics at Johannes Kepler University in Linz, Austria. In his 1965 Ph.D. thesis (see [13]), he created the theory of Gröbner bases, and has refined this construction in subsequent papers — see [14, 12]. He named these objects after his advisor Wolfgang Gröbner. Since 1995, he has been active in the Theorema[a] project at the University of Linz.
In 1987 Buchberger founded and chaired the Research Institute for Symbolic Computation (RISC) at Johannes Kepler University. In 1985 he started the Journal of Symbolic Computation, which has now become the premier publication in the field of computer algebra.
Buchberger also conceived Softwarepark Hagenberg in 1989 and since then has been directing the expansion of this Austrian technology park for software.

[a] A system for automatic theorem-proving.

One key idea in the theory of Gröbner bases involves imposing an *ordering* on monomials:

DEFINITION 11.3.1. Define an ordering on the elements of $\mathbb{N}^n$ and an induced ordering on the monomials of $\mathbb{F}[X_1, \ldots, X_n]$ by $\alpha = (a_1, \ldots, a_n) \succ \beta = (b_1, \ldots, b_n)$ implies that

$$\prod X_i^{a_i} \succ \prod X_i^{b_i}$$

The ordering of $\mathbb{N}^n$ must satisfy the conditions:

(1) if $\alpha \succ \beta$ and $\gamma \in \mathbb{N}^n$, then $\alpha + \gamma \succ \beta + \gamma$
(2) $\succ$ is a *well-ordering:* every set of elements of $\mathbb{N}^n$ has a *minimal* element.

For any polynomial $f \in \mathbb{F}[X_1, \dots, X_n]$, let $\mathrm{LT}(f)$ denote its *leading term* in this ordering — the polynomial's highest-ordered monomial with its coefficient.

REMARK. Condition 1 implies that the corresponding ordering of monomials is preserved by multiplication by a monomial. Condition 2 implies that there are no infinite descending sequences of monomials.

DEFINITION 11.3.2. Suppose an ordering has been chosen for the monomials of $\mathbb{F}[X_1, \dots, X_n]$. If $\mathfrak{a} \in \mathbb{F}[X_1, \dots, X_n]$ is an ideal, let $\mathrm{LT}(\mathfrak{a})$ denote the ideal generated by the leading terms of the polynomials in $\mathfrak{a}$.

(1) If $\mathfrak{a} = (f_1, \dots, f_t)$, then $\{f_1, \dots, f_t\}$ is a *Gröbner basis* for $\mathfrak{a}$ if
$$\mathrm{LT}(\mathfrak{a}) = (\mathrm{LT}(f_1), \dots, \mathrm{LT}(f_t))$$
(2) A Gröbner basis $\{f_1, \dots, f_t\}$ is *minimal* if the leading coefficient of each f_i is 1 and for each i
$$\mathrm{LT}(f_i) \notin (\mathrm{LT}(f_1), \dots, \mathrm{LT}(f_{i-1}), \mathrm{LT}(f_{i+1}), \dots \mathrm{LT}(f_t))$$
(3) A Gröbner basis $\{f_1, \dots, f_t\}$ is *reduced* if the leading coefficient of each f_i is 1 and for each i and *no monomial* of f_i is contained in
$$(\mathrm{LT}(f_1), \dots, \mathrm{LT}(f_{i-1}), \mathrm{LT}(f_{i+1}), \dots \mathrm{LT}(f_t))$$

REMARK. There are many different types of orderings that can be used and a Gröbner basis with respect to one ordering will generally not be one with respect to another.

DEFINITION 11.3.3. The two most common orderings used are:

(1) *Lexicographic ordering*[2]. Let $\mathbb{N}$ denote the natural numbers and let $\mathbb{N}^n$ represents sequences of n natural numbers. Let $\alpha = (a_1, \dots, a_n)$, $\beta = (b_1, \dots, b_n) \in \mathbb{N}^n$. Then $\alpha > \beta \in \mathbb{N}^n$ if, in the vector difference $\alpha - \beta \in \mathbb{Z}^n$, the leftmost nonzero entry is positive — and we define
$$\prod X_i^{a_i} \succ \prod X_i^{b_i}$$
so
$$XY^2 \succ Y^3Z^4$$
This is the ordering that we will use.
(2) *Graded reverse lexicographic order.* Here, monomials are first ordered by *total degree* — i.e., the sum of the exponents. Ties are resolved lexicographically (in reverse — higher lexicographic order represents a lower monomial).

[2]Also called *dictionary-ordering.*

REMARK. In Graded Reverse Lexicographic order, we get

$$X^4Y^4Z^7 \succ X^5Y^5Z^4$$

since the total degree is greater. As remarked above, Gröbner bases depend on the ordering, $\succ$: different orderings give different bases and even different *numbers* of basis elements.

Gröbner bases give an algorithmic procedure for deciding whether a polynomial is contained in an ideal and whether two ideals are equal.

11.4. Buchberger's Algorithm

Unfortunately, Buchberger's algorithm can have *exponential* time-complexity — for graded-reverse lexicographic ordering — and *doubly-exponential* (e^{e^n}) complexity for lexicographic ordering (see [45]). This, incidentally, is why we discussed resultants of polynomials: the complexity of computing Gröbner bases (especially with lexicographic ordering, which leads to the Elimination Property) can easily overwhelm powerful computers. Computing resultants is relatively simple (they boil down to computing determinants).

In practice it seems to have a reasonable running time. In special cases, we have:

(1) For a system of *linear* polynomials, Buchberger's algorithm reduces to Gaussian Elimination for putting a matrix in upper triangular form.
(2) For polynomials over a single variable, it becomes *Euclid's algorithm* for finding the greatest common divisor for two polynomials.

In 1950, Gröbner published a paper ([27]) that explored an algorithm for computing Gröbner bases, but could not prove that it ever terminated. One of Buchberger's signal contributions were the introduction of constructs called S-polynomials. For details, see [58, chapter 5].

The main property of Gröbner bases that will interest us is:

PROPOSITION 11.4.1 (Elimination Property). *Suppose* $\{g_1, \dots, g_j\}$ *is a Gröbner basis for the ideal* $\mathfrak{a} \in \mathbb{C}[X_1, \dots, X_n]$, *computed using lexicographic ordering with*

$$X_1 \succ X_2 \succ \cdots \succ X_n$$

If $1 \leq t \leq n$, *then*

$$\mathfrak{a} \cap \mathbb{C}[X_t, \dots, X_n]$$

has a Gröbner basis that is

$$\{g_1, \dots, g_j\} \cap \mathbb{C}[X_t, \dots, X_n]$$

REMARK. This is particularly important in using Gröbner bases to solve systems of algebraic equations. Here, we want to eliminate

variables if possible and isolate other variables. In example 11.2.5 on page 208, we have the ideal

$$\mathfrak{B} = (xy - z^2, xz - 1, x^2 + y^2 - 3)$$

and can find a Gröbner basis for it with lexicographic ordering with $x \succ y \succ z$ of

$$(z^8 - 3z^2 + 1, y - z^3, z^7 - 3z + x)$$

Here, the basis element $z^8 - 3z^2 + 1$ is an element of $\mathfrak{B} \cap \mathbb{C}[z] \subset \mathbb{C}[x, y, z]$ and the variables x and y have been *eliminated* from it. It follows that z, alone, must satisfy

$$z^8 - 3z^2 + 1 = 0$$

and we can solve for x and y in terms of z:

$$y = z^3$$
$$x = 3z - z^7$$

PROOF. See [58, chapter 5]. □

Maxima has a package that computes Gröbner bases using lexicographic ordering. To load it, type

```
load("grobner")
```

Its main commands[3] are

```
poly_grobner(poly-list,var-list)
```

and

```
poly_reduced_grobner(poly-list,var-list)
```

For example:

```
poly_grobner([x^2+y^2,x^3-y^4],[x,y])
```

returns

$$(x^2 + y^2, x^3 - y^4, x^4 + xy^2, y^6 + y^4)$$

— the Gröbner basis with lexicographic order: $x \succ y$. The command

```
poly_reduced_grobner([x^2+y^2,x^3-y^4],[x,y])
```

returns

$$\left[y^2 + x^2, y^4 + xy^2, y^6 + y^4\right]$$

a Gröbner basis with extraneous elements deleted.

[3] The ones that will interest us, anyway.

11.5. Consistency of algebraic equations

In linear algebra it's well known that systems like

$$x + y = 2$$
$$2x + 2y = 3$$

have no solution; they are inconsistent.

Given a system of algebraic equations

$$\begin{aligned} f_1(x_1, \ldots, x_n) &= v_1 \\ &\vdots \\ f_m(x_1, \ldots, x_n) &= v_m \end{aligned} \tag{11.5.1}$$

we construct an ideal

$$\mathfrak{A} = (f_1(x_1, \ldots, x_n) - v_1, \ldots, f_m(x_1, \ldots, x_n) - v_m)$$

and try to find an "improved" basis for it, i.e,

$$\mathfrak{A} = (b_1, \ldots, b_k)$$

so that the original equations are equivalent to

$$b_1 = \cdots = b_k = 0$$

Suppose we discover that

$$1 \in \mathfrak{A}$$

This implies that

$$\mathfrak{A} = (1)$$

and that

$$1 = 0$$

is equivalent to the original set of equations. Since this is clearly impossible, it follows that the original set of equations was *inconsistent.*

A theorem due to David Hilbert, called the Nullstellensatz (see [58], chapter 12, 12.2.3) shows that inconsistent equations *always* imply that $1 \in \mathfrak{A}$.

David Hilbert (1862–1943) was one of the most influential mathematicians in the 19th and early 20th centuries, having contributed to algebraic and differential geometry, physics, and many other fields.

We conclude that

The system of equations in 11.5.1 is inconsistent if and only if $1 \in \mathfrak{A}$*, which happens if and only if*

```
poly_reduced_grobner([f1-v1,...,fm-vm],
                     [x1,...,xn])
```

returns $[1]$.

EXERCISES.

1. Solve the equations

$$x^2 + y^2 = 0$$
$$x^3 - y^4 = 0$$

2. Solve the equations

$$a_1 a_2 - b_1 b_2 + a_1 - 1 = 0$$
$$a_2 b_1 + a_1 b_2 + b_1 - 1/2 = 0$$
$${a_1}^2 + {b_1}^2 - 1 = 0$$
$${a_2}^2 + {b_2}^2 - 1 = 0$$

3. If

$$(a^2 + 1)(b^2 + 1) + 25 = 10(a + b)$$

and

$$ab = 1$$

what is

$$a^3 + b^3$$

equal to?

4. Solve

$$x^2 + xy + y^2 = 39$$
$$y^2 + yz + z^2 = 49$$
$$z^2 + zx + x^2 = 19$$

5. Find a solution to the system

$$a1 \cdot x1^5 + a2 \cdot x2^5 + a3 \cdot x3^5 = 1/6$$
$$a1 \cdot x1^4 + a2 \cdot x2^4 + a3 \cdot x3^4 = 1/5$$
$$a1 \cdot x1^3 + a2 \cdot x2^3 + a3 \cdot x3^3 = 1/4$$
$$a1 \cdot x1^2 + a2 \cdot x2^2 + a3 \cdot x3^2 = 1/3$$
$$a1 \cdot x1 + a2 \cdot x2 + a3 \cdot x3 = 1/2$$
$$a1 + a2 + a3 = 1$$

CHAPTER 12

Robot motion-planning

"Geometry is one and eternal shining in the mind of God. That share in it accorded to men is one of the reasons that Man is the image of God."

— Johannes Kepler, *Conversation with the Sidereal Messenger (an open letter to Galileo Galilei), [52].*

12.1. A simple robot-arm

Suppose we have a robot-arm with two links, as in figure 12.1.1.

If we assume that both links are of length ℓ, suppose the second link were attached to the origin rather than at the end of the second link.

Then its endpoint would be at (see equation 7.5.2 on page 141)

$$\begin{bmatrix} \ell\cos(\phi) \\ \ell\sin(\phi) \\ 1 \end{bmatrix} = \begin{bmatrix} \cos(\phi) & -\sin(\phi) & 0 \\ \sin(\phi) & \cos(\phi) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & \ell \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$
$$= \begin{bmatrix} \cos(\phi) & -\sin(\phi) & \ell\cos(\phi) \\ \sin(\phi) & \cos(\phi) & \ell\sin(\phi) \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

In other words, the effect of moving from the origin to the end of the second link (attached to the origin) is

(1) *displacement* by ℓ — so that $(0,0)$ is moved to $(\ell,0) = (\ell,0,1) \in \mathbb{R}^3$.
(2) *rotation* by ϕ

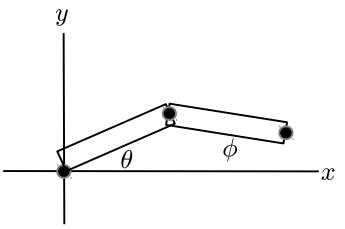


FIGURE 12.1.1. A simple robot arm

This is the effect of the *second* link on all of $\mathbb{R}^2$. If we want to compute the effect of *both* links, *insert* the first link into the system — i.e. rigidly attach the second link to the first, displace by ℓ, and rotate by θ. The effect is equivalent to multiplying by

$$M_2 = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & \ell\cos(\theta) \\ \sin(\theta) & \cos(\theta) & \ell\sin(\theta) \\ 0 & 0 & 1 \end{bmatrix}$$

It is clear that we can compute the endpoint of *any* number of links in this manner — always inserting new links at the *origin* and moving the rest of the chain accordingly.

At this point, the reader might wonder

Where does *algebra* enter into all of this?

The point is that we do not have to deal with trigonometric functions until the very last step. If $a, b \in \mathbb{R}$ are numbers with the property that

$$a^2 + b^2 = 1 \tag{12.1.1}$$

there is a *unique* angle θ with $a = \cos(\theta)$ and $b = \sin(\theta)$. This enables us to replace the trigonometric functions by real numbers that satisfy equation 12.1.1 and derive purely algebraic equations for

(1) the set of points in $\mathbb{R}^2$ reachable by a robot-arm
(2) strategies for reaching those points (solving for explicit angles).

In the simple example above, let $a_1 = \cos(\theta)$, $b_1 = \sin(\theta)$, $a_2 = \cos(\phi)$, $b_2 = \sin(\phi)$ so that our equations for the endpoint of the second link become

$$\begin{aligned} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} &= \begin{bmatrix} a_1 & -b_1 & \ell a_1 \\ b_1 & a_1 & \ell b_1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \ell a_2 \\ \ell b_2 \\ 1 \end{bmatrix} \\ &= \begin{bmatrix} \ell a_1 a_2 - \ell b_2 b_1 + \ell a_1 \\ \ell b_1 a_2 + \ell a_1 b_2 + \ell b_1 \\ 1 \end{bmatrix} \end{aligned}$$

It follows that the points (x, y) reachable by this link are those for which the system of equations

$$\begin{aligned} \ell a_1 a_2 - \ell b_2 b_1 + \ell a_1 - x &= 0 \\ \ell b_1 a_2 + \ell a_1 b_2 + \ell b_1 - y &= 0 \\ a_1^2 + b_1^2 - 1 &= 0 \\ a_2^2 + b_2^2 - 1 &= 0 \end{aligned} \tag{12.1.2}$$

has *real* solutions (for a_i and b_i). Given values for x and y, we can solve for the set of configurations of the robot arm that will *reach* (x, y). We

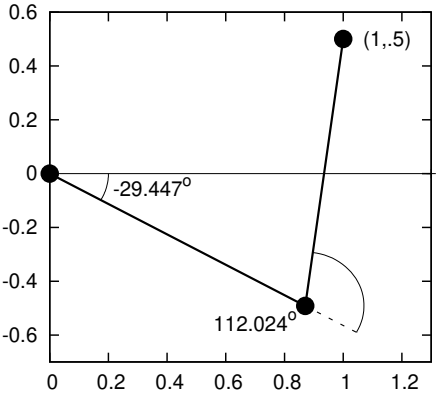


FIGURE 12.1.2. Reaching a point

set the lengths of the robot arms to 1. The system of equations 12.1.2 on the preceding page gives rise to the ideal

$$r = (a_1a_2 - b_1b_2 + a_1 - x, a_2b_1 + a_1b_2 + b_1 - y, {a_1}^2 + {b_1}^2 - 1, {a_2}^2 + {b_2}^2 - 1)$$

in $\mathbb{C}[a_1, a_2, b_1, b_2]$. If we set $x = 1$ and $y = 1/2$, the Gröbner basis of r (using the command 'Basis(r,plex(a_1,b_1,a_2,b_2))' in Maple) is

$$(-55 + 64\,{b_2}^2, 8\,a_2 + 3, 16\,b_2 - 5 + 20\,b_1, -5 - 4\,b_2 + 10\,a_1)$$

from which we deduce that $a_2 = -3/8$ and b_2 can be either $+\sqrt{55}/8$ in which case

$$\begin{aligned} a_1 &= 1/2 + \sqrt{55}/20 \\ b_1 &= 1/4 - \sqrt{55}/10 \end{aligned}$$

or $-\sqrt{55}/8$ in which case

$$\begin{aligned} a_1 &= 1/2 - \sqrt{55}/20 \\ b_1 &= 1/4 + \sqrt{55}/10 \end{aligned}$$

Another question we might ask is:

For what values of x are points on the line $y = 1 - 2x$ reachable?

In this case, we start with the ideal

$$r = (a_1a_2 - b_1b_2 + a_1 - x, a_2b_1 + a_1b_2 + b_1 + 2x - 1, {a_1}^2 + {b_1}^2 - 1, {a_2}^2 + {b_2}^2 - 1)$$

and get the Gröbner basis

```
poly_reduced_grobner([a1*a2-b1*b2+a1-x,
a2*b1+a1*b2+b1+2*x-1,a1^2+b1^2-1,a2^2+b2^2-1],
[a1,b1,a2,b2,x])
```

to get

$$\begin{aligned}[-5x^2+4x+2a_2+1,\\ &-25x^4+40x^3-6x^2-8x-4b_2^2+3,\\ &-5x^3+4x^2+4b_2x+3x+4b_1b_2-2b_2,\\ &-5x^2-5b_1x+4x-b_2+2b_1+a_1-1,\\ &-10x^3-10b_1x^2+13x^2-2b_2x+8b_1x-6x-2b_1+1]\end{aligned}$$

The second line

$$-25x^4+40x^3-6x^2-8x-4b_2^2+3$$

is significant: Since all variables are real, $4b_2^2 \geq 0$, which requires

$$-25x^4+40x^3-6x^2-8x+3 \geq 0 \tag{12.1.3}$$

If we type

```
solve(-25*x^4+40*x^3-6*x^2-8*x+3=0,[x])
```

we get

$$\left[x=-\frac{\sqrt{19}-2}{5}, x=\frac{\sqrt{19}+2}{5}, x=-\frac{i-2}{5}, x=\frac{i+2}{5}\right]$$

Since our variables are all real, we discard the last two solutions. Since the polynomial is >0 for $x=0$,we conclude that solution to the inequality in 12.1.3 is

$$x \in \left[\frac{2-\sqrt{19}}{5}, \frac{2+\sqrt{19}}{5}\right]$$

— so those are the only points on the line $y=1-2x$ that the robot-arm can reach.

12.2. A more complex robot-arm

We conclude this chapter with a more complicated robot-arm in figure 12.2.1 on the next page— somewhat like a Unimation Puma 560[1].

It has:

(1) A base of height ℓ_1 and motor that rotates the whole assembly by ϕ_1 — with 0 being the positive x-axis.
(2) An arm of length ℓ_2 that can be moved forward or backward by an angle of θ_1 — with 0 being straight forward (in the positive x-direction).

[1] In 1985, this type of robot-arm was used to do brain-surgery! See [39].

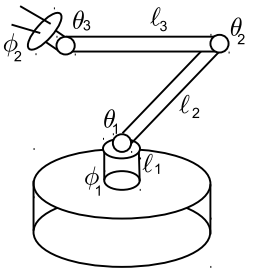


FIGURE 12.2.1. A more complicated robot arm

(3) A second arm of length ℓ_3 linked to the first by a link of angle θ_2, with 0 being when the second arm is in the same direction as the first.
(4) A little "hand" of length ℓ_4 that can be inclined from the second arm by an angle of θ_3 and rotated perpendicular to that direction by an angle ϕ_2.

We do our computations in $\mathbb{R}^4$, start with the hand and work our way to the base. The default position of the hand is on the origin and pointing in the positive x-direction. It displaces the origin in the x-direction by ℓ_2, represented by the matrix

$$D_0 = \begin{bmatrix} 1 & 0 & 0 & \ell_2 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The angle ϕ_1 rotates the arm in the xy-plane, and is therefore represented by

$$\begin{bmatrix} \cos(\phi_1) & -\sin(\phi_1) & 0 & 0 \\ \sin(\phi_1) & \cos(\phi_1) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

or

$$Z_1 = \begin{bmatrix} a_1 & -b_1 & 0 & 0 \\ b_1 & a_1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

with $a_1 = \cos(\phi_2)$ and$b_1 = \sin(\phi_2)$. The "wrist" inclines the hand in the xz-plane by an angle of θ_3, given by the matrix

$$Z_2 = \begin{bmatrix} a_2 & 0 & -b_2 & 0 \\ 0 & 1 & 0 & 0 \\ b_2 & 0 & a_2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

with $a_2 = \cos(\theta_3)$ and $b_2 = \sin(\theta_3)$ and the composite is

$$Z_2 Z_1 D_0 = \begin{bmatrix} a_2 & -b_2 b_1 & -b_2 a_1 & a_2 \ell_4 \\ 0 & a_1 & -b_1 & 0 \\ b_2 & a_2 b_1 & a_2 a_1 & b_2 \ell_4 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The second arm displaces everything by ℓ_3 in the x-direction, giving

$$D_1 = \begin{bmatrix} 1 & 0 & 0 & \ell_3 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

so

$$D_1 Z_2 Z_1 D_0 = \begin{bmatrix} a_2 & -b_2 b_1 & -b_2 a_1 & a_2 \ell_4 + \ell_3 \\ 0 & a_1 & -b_1 & 0 \\ b_2 & a_2 b_1 & a_2 a_1 & b_2 \ell_4 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

so and then inclines it by θ_2 in the xz-plane, represented by

$$Z_3 = \begin{bmatrix} a_3 & 0 & -b_3 & 0 \\ 0 & 1 & 0 & 0 \\ b_3 & 0 & a_3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

so that $Z_3 D_1 Z_2 Z_1 D_0$ is

$$\begin{bmatrix} a_3 a_2 - b_3 b_2 & (-a_3 b_2 - b_3 a_2)\, b_1 & (-a_3 b_2 - b_3 a_2)\, a_1 & (a_3 a_2 - b_3 b_2)\, \ell_4 + a_3 \ell_3 \\ 0 & a_1 & -b_1 & 0 \\ b_3 a_2 + a_3 b_2 & (a_3 a_2 - b_3 b_2)\, b_1 & (a_3 a_2 - b_3 b_2)\, a_1 & (b_3 a_2 + a_3 b_2)\, \ell_4 + b_3 \ell_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Continuing in this fashion, we get a huge matrix, Z. To find the endpoint of the robot-arm, multiply

$$\begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

(representing the origin of $\mathbb{R}^3 \subset \mathbb{R}^4$) by Z to get

$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \tag{12.2.1}$$

$$\begin{bmatrix} ((a_5a_3 + b_5b_4b_3)\,a_2 + (-a_5b_3 + b_5b_4a_3)\,b_2)\,\ell_4 + (a_5a_3 + b_5b_4b_3)\,\ell_3 + a_5\ell_2 \\ ((b_5a_3 - a_5b_4b_3)\,a_2 + (-b_5b_3 - a_5b_4a_3)\,b_2)\,\ell_4 + (b_5a_3 - a_5b_4b_3)\,\ell_3 + b_5\ell_2 \\ (a_4b_3a_2 + a_4a_3b_2)\,\ell_4 + a_4b_3\ell_3 + \ell_1 \\ 1 \end{bmatrix}$$

where $a_3 = \cos(\theta_2)$, $b_3 = \sin(\theta_2)$, $a_4 = \cos(\theta_1)$, $b_4 = \sin(\theta_1)$ and $a_5 = \cos(\phi_1)$, $b_5 = \sin(\phi_1)$. Note that $a_i^2 + b_i^2 = 1$ for $i = 1, \ldots, 5$. We are also interested in the *angle* that the hand makes (for instance, if we want to pick something up). To find this, compute

$$Z\begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix} - Z\begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} = Z\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \tag{12.2.2}$$

$$\begin{bmatrix} (a_5a_3 + b_5b_4b_3)\,a_2 + (-a_5b_3 + b_5b_4a_3)\,b_2 \\ (b_5a_3 - a_5b_4b_3)\,a_2 + (-b_5b_3 - a_5b_4a_3)\,b_2 \\ a_4b_3a_2 + a_4a_3b_2 \\ 0 \end{bmatrix}$$

The numbers in the top three rows of this matrix are the *direction-cosines* of the hand's direction. We can ask what points the arm can reach with its hand aimed in a particular direction.If we set $\ell_1 = \ell_2 = 1$, equation 12.2.1 implies that the endpoint of the robot-arm are solutions to the system

$$\begin{aligned} a_5a_4a_3 - a_5b_4b_3 + a_5a_4 - x &= 0 \\ b_5a_4a_3 - b_5b_4b_3 + b_5a_4 - y &= 0 \\ b_4a_3 + a_4b_3 + b_4 - z &= 0 \\ a_3^2 + b_3^2 - 1 &= 0 \\ a_4^2 + b_4^2 - 1 &= 0 \\ a_5^2 + b_5^2 - 1 &= 0 \end{aligned} \tag{12.2.3}$$

If we want to know which points it can reach with the hand pointing in the direction

$$\begin{bmatrix} 1/\sqrt{3} \\ 1/\sqrt{3} \\ 1/\sqrt{3} \end{bmatrix}$$

use equation 12.2.2 on the previous page to get

$$\begin{aligned}(a_5a_4a_3 - a_5b_4b_3)\,a_2 + (-a_5a_4b_3 - a_5b_4a_3)\,b_2 - 1/\sqrt{3} &= 0\\ (b_5a_4a_3 - b_5b_4b_3)\,a_2 + (-b_5a_4b_3 - b_5b_4a_3)\,b_2 - 1/\sqrt{3} &= 0\\ (b_4a_3 + a_4b_3)\,a_2 + (a_4a_3 - b_4b_3)\,b_2 - 1/\sqrt{3} &= 0\\ a_2^2 + b_2^2 - 1 &= 0\end{aligned} \tag{12.2.4}$$

We regard these terms (in equations 12.2.3 on the preceding page and 12.2.4 as generators of an ideal, $\mathfrak{P}$.

To understand possible configurations of the robot-arm, we compute a Gröbner basis of $\mathfrak{P}$ with lexicographic ordering. Unfortunately, we run up against the e^{e^n}-execution time: This Maxima program

```
poly_grobner([a5*a4*a3-a5*b4*b3+a5*a4-x,
b5*a4*a3-b5*b4*b3+b5*a4-y,
b4*a3+a4*b3+b4-z,
(a5*a4*a3-a5*b4*b3)*a2+(-a5*a4*b3-a5*b4*a3)*b2
        -1/sqrt(3),
(b5*a4*a3-b5*b4*b3)*a2+(-b5*a4*b3-b5*b4*a3)*b2
        dis-1/sqrt(3),
(b4*a3+a4*b3)*a2+(a4*a3-b4*b3)*b2-1/sqrt(3),
a2^2+b2^2-1,
a3^2+b3^2-1,
a4^2+b4^2-1,
a5^2+b5^2-1],
[a5,a4,a3,a2,b5,b4,b3,b2,x,y,z]);
```

runs for several hours and crashes due to memory issues. We can try another piece of free software, Macaulay 2, which can compute Gröbner bases (among other things). The program

```
R=QQ[a_5,a_4,a_3,a_2,b_5,b_4,b_3,b_2,w,x,y,z,
      MonomialOrder=>Lex]
Z=ideal(a_5*a_4*a_3-a_5*b_4*b_3+a_5*a_4-x,
b_5*a_4*a_3-b_5*b_4*b_3+b_5*a_4-y,
b_4*a_3+a_4*b_3+b_4-z,
(a_5*a_4*a_3-a_5*b_4*b_3)*a_2+
(-a_5*a_4*b_3-a_5*b_4*a_3)*b_2-w,
(b_5*a_4*a_3-b_5*b_4*b_3)*a_2+
(-b_5*a_4*b_3-b_5*b_4*a_3)*b_2-w,
(b_4*a_3+a_4*b_3)*a_2+(a_4*a_3-b_4*b_3)*b_2-w,
a_2^2+b_2^2-1,
a_3^2+b_3^2-1,
a_4^2+b_4^2-1,
a_5^2+b_5^2-1
)
P=gb Z
```

```
gens P
```

runs for several hours and crashes when it tries to allocate more than 4 gigabytes of memory (my computer has more, but the system seems reluctant to allocate it).

We finally use the free system, Singular, from the University of Karlsruhe in Germany. It uses newer and greatly improved algorithms for computing Gröbner bases. It is, perhaps, the most advanced such program in the world.

The cryptic Singular program

```
option(redSB); // Causes Singular to
               //compute a reduced
               // Groebner basis
ring R=(0),(a_5,a_4,a_3,a_2,b_5,b_4,b_3,b_2,
                    x,y,z,w),lp;
// numeric 0=rational coefficients
// lp=lexicographic ordering
ideal s = a_5*a_4*a_3-a_5*b_4*b_3+a_5*a_4-x,
b_5*a_4*a_3-b_5*b_4*b_3+b_5*a_4-y,
b_4*a_3+a_4*b_3+b_4-z,
(a_5*a_4*a_3-a_5*b_4*b_3)*a_2+
     (-a_5*a_4*b_3-a_5*b_4*a_3)*b_2-w,
(b_5*a_4*a_3-b_5*b_4*b_3)*a_2+
     (-b_5*a_4*b_3-b_5*b_4*a_3)*b_2-w,
(b_4*a_3+a_4*b_3)*a_2+(a_4*a_3-b_4*b_3)*b_2-w,
a_2^2+b_2^2-1,
a_3^2+b_3^2-1,
a_4^2+b_4^2-1,
a_5^2+b_5^2-1;
slimgb(s);  // Causes Singular to try to use the
            // smallest expressions
```

immediately comes back with the results in appendix A on page 285. Here '0' is the digit zero, which represents *rational coefficients* for the polynomials. Since $1/\sqrt{3}$ is not rational, we make it into a *variable.*

Incidentally, the commercial computer algebra software, Maple 16, also comes back with an answer quickly, but an *incorrect* one. This author has not had the opportunity to test the other major commercial system: Mathematica.

Several things leap out at us from the long list in appendix A on page 285:

(1) 3*w^2-1
(2) 2*b_5^2-1
(3) a_5-b_5
(4) x-y

Since all elements of the Gröbner basis are implicitly set to 0, this means $w = \pm 1/\sqrt{3}$; no other values are possible[2]. The second line implies that $x = y$ — a *very* severe restriction on where the robot-arm can reach (when the hand is pointed in that direction).

This example was meant to illustrate what is called "the combinatorial explosion" where even fast computers can be overwhelmed by the complexities of a problem.

Maxima will be able to handle all of the other examples we do (and the exercises!).

Suppose we want to reach the point (1,1,1) and we don't care how the hand is aligned. We try the Maxima commands

```
poly_reduced_grobner([a5*a4*a3-a5*b4*b3+a5*a4-1,
b5*a4*a3-b5*b4*b3+b5*a4-1,
b4*a3+a4*b3+b4-1,
a2^2+b2^2-1,
a3^2+b3^2-1,
a4^2+b4^2-1,
a5^2+b5^2-1],
[a5,a4,a3,a2,b5,b4,b3,b2]);
```

and Maxima comes back (immediately!) with

$$[b2^2 + a2^2 - 1, 12b4^2 - 12b4 + 1, 4b3^2 - 3,$$
$$2b3b4 - b3 + a5, -6b3b4 + 4b3 - 3a4, 1 - 2a3, b5 + 2b3b4 - b3]$$

Conclusions:

- ▷ $a3 = 1/2$
- ▷ $b3 = \pm\sqrt{3/4}$
- ▷ **solve**(12*b4^2-12*b4+1=0,b4) gives $\left[\text{b4} = -\frac{\sqrt{6}-3}{6}, \text{b4} = \frac{\sqrt{6}+3}{6}\right]$
- ▷ b5, a4, and a5 are uniquely determined by b3 and b4: $a5 = b3 - 2b3b4$, $b5 = b3 - 2b3b4 = a5$, $a4 = 4b3/3 - 2b3b4$
- ▷ a2 and b2 are free, subject only to the equation $a2^2 + b2^2 = 1$. They rotate the hand and have no effect on its position.

So there an infinite number of solutions and, for each choice of a2 and b2, there are 4 solutions for the other variables:

(1) $a3 = 1/2$, $b3 = \sqrt{3/4}$, $\text{b4} = -\frac{\sqrt{6}-3}{6} = .09175117$, $a4 = \frac{\sqrt{3}\sqrt{6}+\sqrt{3}}{6} = .995781$,$a5 = 1/\sqrt{2} = b5$

(2) $a3 = 1/2$, $b3 = -\sqrt{3/4}$,$\text{b4} = -\frac{\sqrt{6}-3}{6} = .09175117$, $a4 = -\frac{\sqrt{3}\sqrt{6}+\sqrt{3}}{6} = -.995781$,$a5 = -1/\sqrt{2} = b5$

[2]We could've concluded this from geometric reasoning: this is the length of a unit-vector manipulated by orthogonal transformations that leave length unchanged.

(3) $a3 = 1/2$, $b3 = \sqrt{3/4}$, $b4 = \frac{\sqrt{6}+3}{6} = .9082482$, $a4 = \frac{\sqrt{3}\cdot\sqrt{6}-\sqrt{3}}{6} = .4184316$, $a5 = 1/\sqrt{2} = b5$

(4) $a3 = 1/2$, $b3 = -\sqrt{3/4}$, $b4 = \frac{\sqrt{6}+3}{6} = .9082482$, $a4 = -\frac{\sqrt{3}\cdot\sqrt{6}-\sqrt{3}}{6} = -.4184316$, $a5 = -1/\sqrt{2} = b5$

EXERCISES.

1. How far can the robot-arm in figure on page 219 reach? Hint: add $x^2 + y^2 + z^2 - r^2$ to the list of expressions representing the robot-arm and find reasonable values of r.

2. How can the robot-arm in figure on page 219 reach the point $(1/2, -1/3, 1)$?

3. For the robot-arm in figure on page 219, why are there always an *even* number of solutions (ignoring a2 and b2)? (Hint: consider its geometry).

CHAPTER 13

Differential Game Theory, a Drive-by

"Everything has been thought of before, but the problem is to think of it again."
— Johann Wolfgang von Goethe.

13.1. Dances with Limousines

Steering with his right hand and sipping a martini[1] with his left, Dr. Evil is pursuing James Bond in a limousine. Bond is on foot so the limousine is much faster but has a large turning radius.

The limousine is controlled by its steering wheel, and Bond can run or jump in any direction. We will set up a system of differential equations to simulate this situation and solve them with the **rk**-command.

Fixed parameters:

(1) limo_turning_radius
(2) limo_speed
(3) bond_speed
(4) kill_distance

State variables:

limo_x The limousine's x-coordinate

limo_y The limousine's y-coordinate

limo_theta The angle the limousine makes with respect to the x-axis. Between $-\pi$ and π.

limo_steering Position of the steering wheel. Between $-\pi$ and π. It's the direction we want to go.

limo_dtheta The rate of change of limo_theta with respect to time, t — determined by the steering wheel and turning_radius. Since curvature is rate of change with respect to distance, s and since

$$\frac{ds}{dt} = \text{limo\_speed}$$

we get

$$-\frac{1}{\text{turning\_radius}} \leq \frac{\text{limo\_dtheta}}{\text{limo\_speed}} \leq \frac{1}{\text{turning\_radius}}$$

[1] Stirred not shaken, of course!

or

$$-\frac{\text{limo\_speed}}{\text{turning\_radius}} \leq \text{limo\_dtheta} \leq \frac{\text{limo\_speed}}{\text{turning\_radius}}$$

so

$$\text{limo\_dtheta} = \min\left(\max\left(-\frac{\text{limo\_speed}}{\text{turning\_radius}}, \text{limo\_steering}\right), \frac{\text{limo\_speed}}{\text{turning\_radius}}\right)$$

limo_dx The limousine's speed in the x-direction — limo_speed · cos(limo_theta)

limo_dy The limousine's speed in the y-direction — limo_speed · sin(limo_theta)

bond_x

bond_y

bond_dx

bond_dy

The idea of this algorithm is that the limo's direction is determined by the angle **limo_theta** and its derivative with respect to arc-length is given by the steering wheel. We determine the angle of a line connecting the limo to bond (using the **atan2**-function) and increase or decrease the steering wheel accordingly (subject to the turning radius).

```
limo_speed:50;
limo_turning_radius:30;
time_step:.01;

limo_dx(limo_theta):=limo_speed*cos(limo_theta);
limo_dy(limo_theta):=limo_speed*sin(limo_theta);

normalize(theta):=block([],
if(theta>=%pi) then return (theta-2*%pi),
if(theta<-%pi) then return (theta+2*%pi),
theta);

/* Aim straight for Bond! */
limo_steering(bond_x,bond_y,limo_x,limo_y,
        limo_theta)
    :=block(
    [angle:atan2(bond_y-limo_y,bond_x-limo_x)],
angle-limo_theta);

/* Take turning radius into account. */
limo_dtheta(limo_steering):=block(
[lim:limo_speed/limo_turning_radius],
```

```
min(max(-lim,limo_steering),lim)
);

/* Bond's strategies */

/* Matador strategy: don't move
   unless the limo comes close! */
bond_dx(bond_x,bond_y,limo_x,limo_y,limo_theta)
  :=block([distance],
distance=sqrt((bond_x-limo_x)^2+(limo_y-bond_y)^2),
if(distance<5) then return
    (cos(limo_theta+%pi/2)/time_step),
0);

bond_dy(bond_x,bond_y,limo_x,limo_y,limo_theta)
   :=block([distance],
distance=sqrt((bond_x-limo_x)^2+(limo_y-bond_y)^2),
if(distance<5) then return
   (sin(limo_theta+%pi/2)/time_step),
0);
/* End of Bond's strategies */

/* Run the simulation! */
pursuit:rk([
       limo_dx(limo_theta),
       limo_dy(limo_theta),
       limo_dtheta(limo_steering(bond_x,bond_y,
                          limo_x,limo_y,limo_theta)),
       bond_dx(bond_x,bond_y,limo_x,limo_y,
           limo_theta),
       bond_dy(bond_x,bond_y,limo_x,limo_y,
           limo_theta)
               ],
    [limo_x,limo_y,limo_theta,bond_x,bond_y],
    [0,0,0,10,10],
    [t,0,10,time_step]
);
```

Unfortunately, this reasonable-looking approach fails *miserably*[2].

Suppose we abandon angles and reference a *unit-vector* giving the direction of the limo's motion. If $\mathbf{u}(t)$ is a unit vector then

$$\mathbf{u} \bullet \mathbf{u} = 1$$

[2]Try running it!

so

$$\frac{d(\mathbf{u} \bullet \mathbf{u})}{dt} = 2\mathbf{u} \bullet \frac{d\mathbf{u}}{dt} = 0$$

It follows that

$$\mathbf{u} \perp \frac{d\mathbf{u}}{dt}$$

So the vector we pick for $d\mathbf{u}/dt$ must be perpendicular to $\mathbf{u}$. In the following program $\mathbf{u}$ is called **limo_direction** and $d\mathbf{u}/dt$ is bond_at$_\perp$ (compare with definition 7.3.4 on page 117):

```
limo_speed:50;
limo_turning_radius:30;
time_step:.01;
max_angle:limo_speed/limo_turning_radius;

direction_d(bond_position,limo_position,
    limo_direction):=block(
     [bond_at,distance,dot_prod,perp,d_vect,
          p_comp],
     bond_at:float(bond_position-limo_position),
    distance:float(sqrt(bond_at.bond_at)),
    dot_prod:float(bond_at.limo_direction),
    perp:float(bond_at - dot_prod*limo_direction),
/* Compute the projection of bond_at
   onto limo_direction
 Now compute the perpendicular vector, perp */
    perp_length:float(sqrt(perp.perp)),
    if(perp_length<.01) then return (0),
/*We're pointed in the right direction */

    /* Impose the turning-radius restrictions */
    if(perp_length>max_angle) then
       return ((max_angle/perp_length)*perp),
    perp
    );

limo_d(limo_direction):= limo_speed*limo_direction;

/* Bond's strategies */

/* Matador strategy:
    don't move unless the limo comes close! */
bond_d(bond_position,limo_position,
     limo_direction):=block(
    [bond_at:bond_position-limo_position,distance],
         distance:float(sqrt(bond_at.bond_at)),
       bond_hit:(distance<1),
```

```
      /* We're done. Bond is dead */

    if(distance < 3) then
      return (100*matrix([-limo_direction[1,2],
        limo_direction[1,1]])),
0);
/* End of Bond's strategies */

/* Initial conditions */
limo_position:matrix([0,0]);
bond_position:matrix([20,-20]);
limo_direction:matrix([-1,0]);
path:[];
bpath:[];
bond_hit:false;
timeout:false;

/* Run the simulation! */
for t:0 step time_step unless (bond_hit or timeout)
  do(
    timeout:(t>100),
    limo_position:limo_position
      +.01*limo_d(limo_direction),
    bond_position:bond_position
      +.01*bond_d(bond_position,
         limo_position,limo_direction),
    limo_direction:limo_direction
      +.01*direction_d(bond_position,
         limo_position,limo_direction),

    /* Record the limo's location:
       place the limo's coords
       at the end of its path.*/
    path:endcons([limo_position[1,1],
                  limo_position[1,2]], path),

    /* Normalize limo_direction
       so it remains a unit-vector */
    limo_direction:
     float(limo_direction
           /sqrt(limo_direction.limo_direction)),

    /* Record Bond's location:
       place Bond's coords at the
       end of his path.*/
```

```
    bpath:endcons([bond_position[1,1],
                   bond_position[1,2]],bpath)
        ); /* End of for-loop */

/* Watch the dance! */
plot2d([[discrete,path],[discrete,bpath]],
     [style,[lines,1],[points,1]],
     [legend,"Limo","Bond"]);
```

Unfortunately, the **rk**-command requires its arguments to be *real scalars* (argh!) and fails *silently* if they aren't.

We dispense with the **rk**-command and use a *finite-difference* approximation to derivatives (see Euler's method, equation 4.1.5 on page 52)

$$\frac{d\text{limo\_direction}}{dt} \sim \frac{\Delta\text{limo\_direction}}{\Delta t}$$

which becomes more accurate the smaller Δt is. We pick $\Delta t = .01$. Since this is only an approximate derivative, **limo_direction** will *grow* with each iteration of the **for**-loop. We reset it to being a unit-vector via

```
  limo_direction:
    float(limo_direction
          /sqrt(limo_direction.limo_direction)),
```

Take note of the command

```
    /* Record the limo's location:
       place the limo's coords
       at the end of its path.*/
    path:endcons([limo_position[1,1],
                  limo_position[1,2]], path),
```

for keeping a record of the limo's position and compare to Note 7.1.1 on page 106.

If we run the code as given above, we get figure 13.1.1 on the facing page, which shows that it's easy to evade a limo with a large turning radius that is close by — just stand in one spot!

If we set the turning radius to 10, we get figure 13.1.2 on the next page, where Bond has to jump around a bit.

Note that this program works in n-dimensions, whereas the program using angles would only work in two, at best[3].

If Bond stands far away (x=100, for instance) even a large turning radius limo manages to catch him — see figure 13.1.3 on page 234. This suggests a strategy to use when the turning radius is large.

[3]Although finding 100-dimensional limos might be hard!

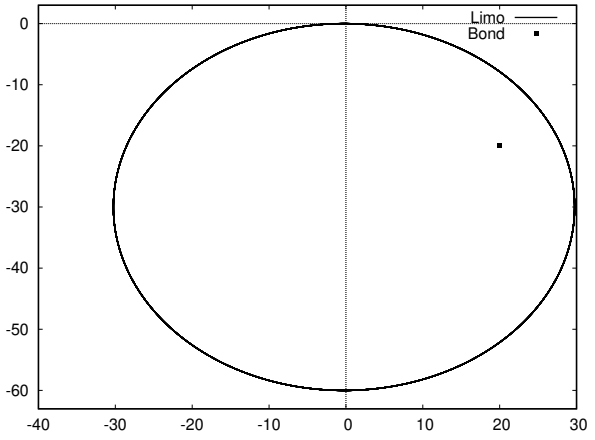


FIGURE 13.1.1. Turning radius=30

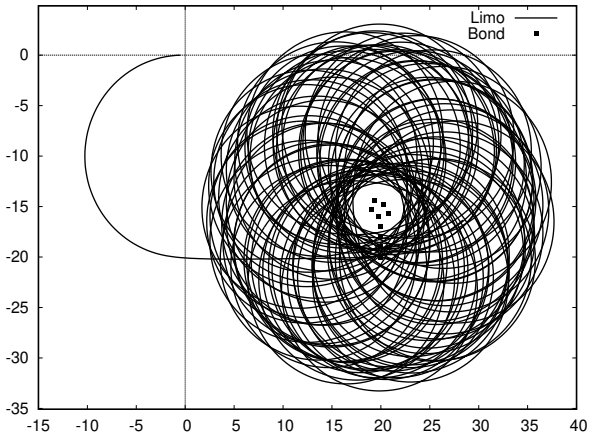


FIGURE 13.1.2. Turning radius=10

Note that the limo starts by facing away from Bond. This is the initial condition

```
limo_direction:matrix([-1,0]);
```

Something strange happens if Bond starts out on the positive x-axis, say at (10,0). We get the plot in figure 13.1.4 on the next page, where the limo frantically runs *away* from Bond! This is what is called the *Gimbel Problem*: certain angles cause algorithms to lock up and execute incorrectly. This problem occurs because we assume that, if **perp**=0, we must be facing Bond whereas we might be facing away from him.

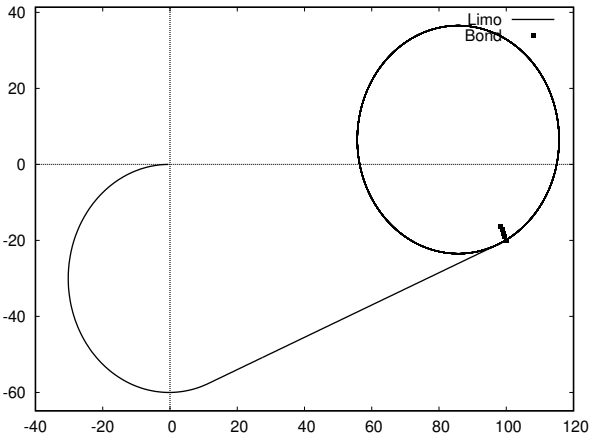


FIGURE 13.1.3. Bond far away

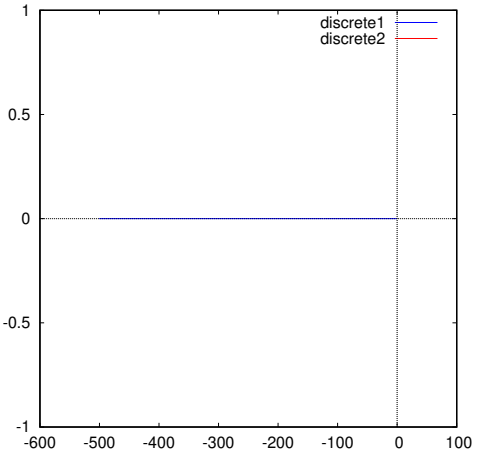


FIGURE 13.1.4. Gimbel problem

Equation 7.3.1 on page 116 suggests the solution: if dot_prod > 0, we are facing Bond, and if dot_prod < 0, we are facing away. We amend the limo-code by replacing

```
    perp_length:float(sqrt(perp.perp)),
    if(perp_length<.01) then return (0),
/*We're pointed in the right direction */
```

with

```
    perp_length:float(sqrt(perp.perp)),
```

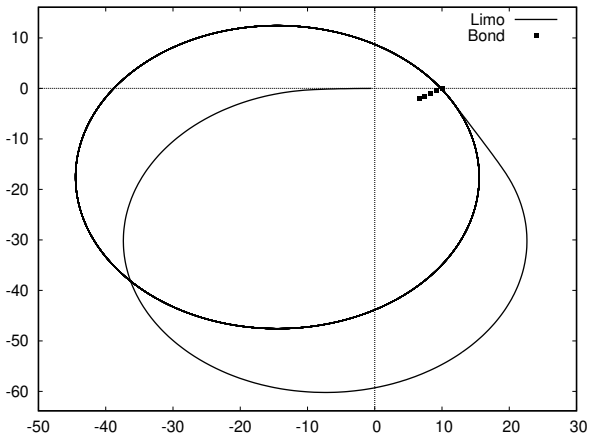


FIGURE 13.1.5. Solution to the Gimbel Problem

```
    if((perp_length<.01) and (dot_prod >0))
      then return (0),
/*We're pointed in the right direction */

    if(perp_length<.01) then return
    (matrix([-limo_direction[1,2],
             limo_direction[1,1]])),
/* Jump sideways, so we're no longer lined up
   with Bond */
```

and this gives us figure 13.1.5.

EXERCISES.

1. Why does the initial program that uses *angles* and **atan2** *fail?*

2. Why did we need the **timeout** logical variable?

3. Dr. Evil is afraid of spilling his martini if he makes a left turn[4]. Rewrite the limo program so it only makes *right* turns.

4. Rewrite the limo program so it is capable of hitting Bond even if its turning radius is large and Bond is nearby. Hint: If Bond is nearby, put the limo into *back-off* mode and drive away. When it's far enough away, put it back into *attack* mode.

[4]Reminiscent of an old road-sign in Ireland: "Don't drink while you drive; you might spill some."

5. In the program given, Bond always jumps in a direction *perpendicular* to the direction of the limo. Experiment with other angles.

6. Notice that Bond never gets tired and always jumps a huge distance (100). Rewrite the program so Bond get tired after each jump, and the distance jumped decreases until he is unable to jump at all.

13.2. Rock, Paper, Rocket

In this section we consider the opposite problem from the last:

Is it possible to hit a *fast* object with a *slow* one?

Clearly, if the slow object merely chases the fast one, it will miss. We suppose the fast object is headed to a target and the slow one starts out near the target. We'll begin by assuming that the fast object heads in a straight line to the target.

We begin by coding the main loop and the rocket's equations of motion:

```
rocket_speed:100;
rocket_start:matrix([100,3]);
target_location:matrix([0,0]);
time_step:.01;
timeout:false;
target_hit:false;
rocket_location:copymatrix(rocket_start);
rocket_dir:matrix([1,0]);
rocket_path:[];

 rocket_unit(rocket_location,target_location):=
              block(
          [vector_to_target,dist_to_target],
          vector_to_target:target_location
             - rocket_location,
          dist_to_target:
             float(sqrt(vector_to_target
             .vector_to_target)),
          if(dist_to_target<1.0) then
             return(matrix([0,0])),
          float(vector_to_target/dist_to_target)
          );

for t:0 step time_step unless
        (timeout or target_hit)
    do (
```

```
        timeout:t>100,
        rocket_dir:rocket_unit(rocket_location,
            target_location),
        target_hit:rocket_dir.rocket_dir<.5,

        rocket_location:rocket_location
          +time_step*rocket_speed*rocket_dir,
        rocket_path:endcons([rocket_location[1,1],
                             rocket_location[1,2]],
                             rocket_path)
        );
plot2d([discrete,rocket_path],
     [style,[lines,1],[points,1]]);
```

Now we add the code for the rock. We'll begin by using the simplest code possible: simply head in the direction of the rocket[5]. We have

```
rock_unit(rocket_location,target_location,
            rock_location):=block(
          [vector_to_rocket,dist_to_rocket],
          vector_to_rocket:rocket_location
             - rock_location,
          dist_to_rocket:
              float(sqrt(vector_to_rocket
             .vector_to_rocket)),
          if(dist_to_rocket<1.0) then
             return(matrix([0,0])),
          float(vector_to_rocket/dist_to_rocket)
          );
```

and combining it with the previous code gives

```
rocket_speed:100;
rocket_start:matrix([100,3]);
target_location:matrix([0,0]);
time_step:.01;
timeout:false;
target_hit:false;
rocket_hit:false;
rocket_location:copymatrix(rocket_start);
rocket_dir:matrix([1,0]);

rock_speed:10;
rock_offset:matrix([5,5]);
rocket_path:[];
rock_location:target_location+rock_offset;
```

[5] So our "rock" can be steered!

```
rock_path:[];

 rocket_unit(rocket_location,target_location):=block(
          [vector_to_target,dist_to_target],
          vector_to_target:target_location
             - rocket_location,
          dist_to_target:float(sqrt(vector_to_target
             .vector_to_target)),
          if(dist_to_target<1.0) then
             return(matrix([0,0])),
          float(vector_to_target/dist_to_target)
          );
rock_unit(rocket_location,
              rock_location):=block(
          [vector_to_rocket,dist_to_rocket],
          vector_to_rocket:rocket_location
             - rock_location,
          dist_to_rocket:float(sqrt(vector_to_rocket
             .vector_to_rocket)),
          if(dist_to_rocket<1.0) then
             return(matrix([0,0])),
          float(vector_to_rocket/dist_to_rocket)
          );

for t:0 step time_step
        unless (timeout or target_hit or rocket_hit)
    do (
       timeout:t>100,
       rocket_dir:rocket_unit(rocket_location,
           target_location),
       target_hit:rocket_dir.rocket_dir<.5,

       rock_dir:rock_unit(rocket_location,
           rock_location),
       rocket_hit:rock_dir.rock_dir<.5,

       rocket_location:rocket_location
         +time_step*rocket_speed*rocket_dir,
       rock_location:rock_location
         +time_step*rock_speed*rock_dir,
       rocket_path:endcons([rocket_location[1,1],
                     rocket_location[1,2]],
                   rocket_path),
       rock_path:endcons([rock_location[1,1],
                     rock_location[1,2]],rock_path)
```

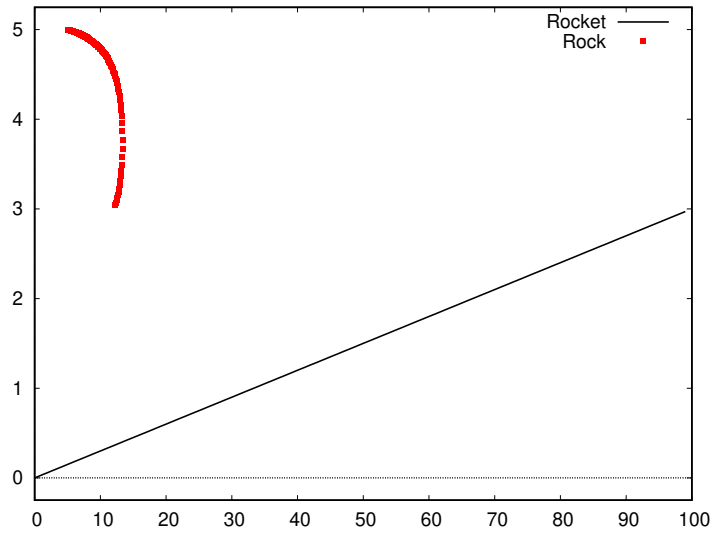


FIGURE 13.2.1. Naive pursuit algorithm

```
        );
plot2d([[discrete,rocket_path],[discrete,rock_path]],
       [style,[lines,1],[points,1]]);
```

This produces the plot in figure 13.2.1. Clearly, we have failed to stop the rocket! If we increase the rock speed to 30, we get the plot in figure 13.2.2 on the next page

As it nears the rocket, the rock simply chases it and will always lose.

We will try a slightly more sophisticated algorithm:

(1) Determine how far away the rocket is,
(2) Determine how long it would take to reach that point (at the speed the *rock* travels),
(3) Estimate the possible future location of the rocket at that time, using the rocket's direction,
(4) Aim for *that* location instead of the rocket's present location.

```
/* Predictive Rocket vs. Rock program */
rocket_speed:100;
rocket_start:matrix([100,3]);
target_location:matrix([0,0]);
time_step:.01;
timeout:false;
target_hit:false;
rocket_hit:false;
```

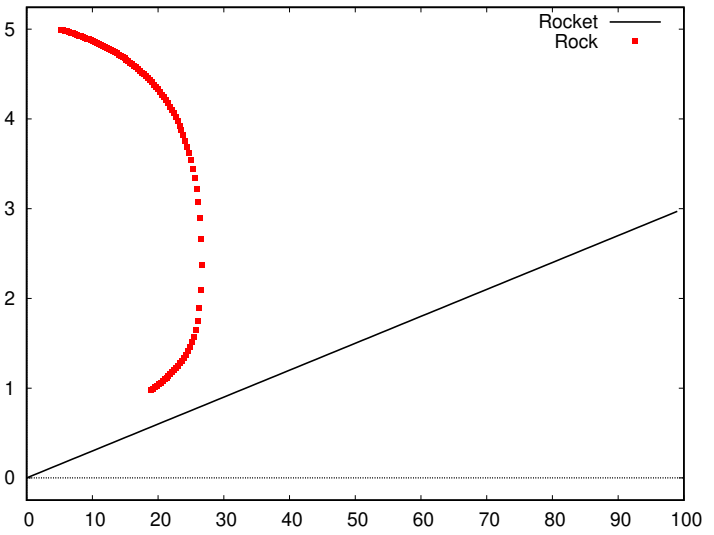


FIGURE 13.2.2. Rock speed 30

```
rocket_location:copymatrix(rocket_start);
rocket_dir:matrix([1,0]);

rock_speed:30;
rock_offset:matrix([5,5]);
rocket_path:[];

/* The rock is not located at the target,
   but offset from it */
rock_location:target_location+rock_offset;
rock_path:[];

/* The rocket's direction of travel */
 rocket_unit(rocket_location,
       target_location):=block(
          [vector_to_target,dist_to_target],
          vector_to_target:target_location
             - rocket_location,
          dist_to_target:float(sqrt(vector_to_target
            .vector_to_target)),
          if(dist_to_target<1.0) then
             return(matrix([0,0])),
          float(vector_to_target/dist_to_target)
          );
```

```
/*The rock's direction of travel*/
     rock_unit(rocket_location,rock_location,
          rocket_dir):=block(
          [vector_to_rocket,dist_to_rocket,
           rocket_future,vector_to_future,
           dist_to_future],
          vector_to_rocket:rocket_location
             - rock_location,
          dist_to_rock:float(sqrt(vector_to_rocket
             .vector_to_rocket)),
          if(dist_to_rock<1.0) then
             return(matrix([0,0])),

/* The rocket's predicted future position */
          rocket_future:rocket_location+
             rocket_dir*rocket_speed/rock_speed,
          vector_to_future:rocket_future
               -rock_location,
          dist_to_future:
             sqrt(vector_to_future.vector_to_future),
/* Aim at the future position! */
          float(vector_to_future/dist_to_future)
          );

for t:0 step time_step
        unless (timeout or target_hit or rocket_hit)
    do (
       timeout:t>100,
       rocket_dir:rocket_unit(rocket_location,
           target_location),
       target_hit:rocket_dir.rocket_dir<.1,
       rock_dir:rock_unit(rocket_location,
           rock_location,rocket_dir),
       rocket_hit:rock_dir.rock_dir <.1,
       if(rocket_hit) then print("Rocket hit"),
       rocket_location:rocket_location
         +time_step*rocket_speed*rocket_dir,
       rock_location:rock_location
         +time_step*rock_speed*rock_dir,
       rocket_path:endcons([rocket_location[1,1],
                    rocket_location[1,2]],
                   rocket_path),
       rock_path:endcons([rock_location[1,1],
                    rock_location[1,2]],rock_path)
```

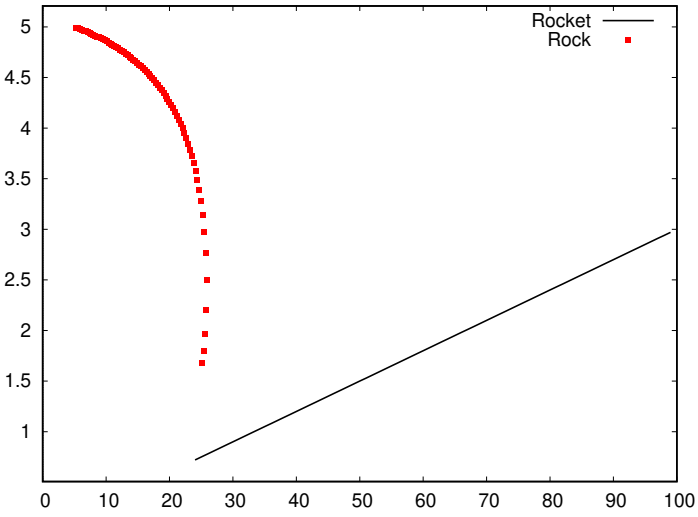


FIGURE 13.2.3. Predictive algorithm with rock speed 30

```
        );
plot2d([[discrete,rocket_path],[discrete,rock_path]],
       [style,[lines,1],[points,1]],
       [legend,"Rocket","Rock"]);
```

We will also print out "Rocket hit" if this happens. We get the plot in figure 13.2.3, the "Rocket hit" phrase is printed, and the rocket doesn't get within 20 units of the target, even though it is traveling more than 3 times the rock's speed!

EXERCISES.

1. Experiment with the predictive program and determine the minimum speed the rock must be traveling to guarantee that the rocket does not get within 10 distance-units of the target.

2. Experiment with different offsets of the rock-launch-point.

3. Try the program where the rocket follows a *curved* path rather than a straight line.

4. Consider possible improvements to the predictive algorithm and program them. Our program uses the *first* derivative of the rocket

motion (its tangent vector). Consider how one could estimate its *second* derivative and use that (and the first) to estimate its future location. Hint: consider how much the rocket's unit-vector changes between a given call to the rock-routine and the *previous* call.

5. Suppose the rocket has random fluctuations in its direction to try to confuse the rock. These fluctuations must decrease to 0 as the rocket approaches the target, or it won't have a chance of hitting the target. Program this!

CHAPTER 14

Special Functions

"Certain functions appear so often that it is convenient to give them names. These are collectively called special functions. There are many examples and no single way of looking at them can illuminate all examples or even all the important properties of a single example of a special function."
— Richard Askey.

14.1. The Gamma Function

In an attempt to define factorials over non-integers, Bernoulli defined the function

$$\Gamma(z) = \int_0^\infty e^{-t} t^{z-1} dt \quad \text{for } \Re(z) > 0 \tag{14.1.1}$$

Integration by parts shows that

$$\Gamma(z+1) = z\Gamma(z) \tag{14.1.2}$$

Since $\Gamma(1) = 1$, an easy induction shows that

$$\Gamma(n) = (n-1)!$$

for n a positive integer.

Daniel Bernoulli (1700 – 1782) was a Swiss mathematician and physicist and was one of the many prominent mathematicians in the Bernoulli family from Basel. He is particularly remembered for his applications of mathematics to mechanics, especially fluid mechanics, and for his pioneering work in probability and statistics. His name is commemorated in the Bernoulli's principle, a particular example of the conservation of energy, which describes the mathematics of the mechanism underlying the operation of two important technologies of the 20th century: the carburetor and the airplane wing.

Maxima implements the Γ-function via the **gamma**-command. For instance

```
map('gamma,[1,2,3,4,5,6,7,8,9]);
```

returns

$$[1, 1, 2, 6, 24, 120, 720, 5040, 40320]$$

We can plot $\Gamma(x)$ on the real line via

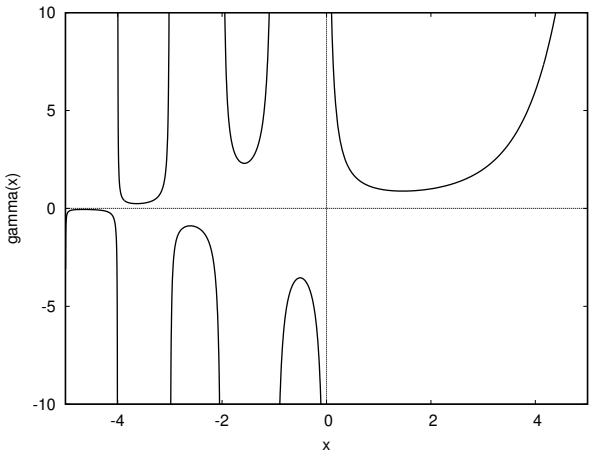


FIGURE 14.1.1. The Γ-function

```
plot2d(gamma(x),[x,-5,5],[y,-10,10],[style,
[lines,2,5]],[grid,10,20]);
```

to get figure 14.1.1.

We also have the *Euler Reflection equation* (see [1, chapter 5]).

$$\Gamma(1-z)\Gamma(z) = \frac{\pi}{\sin(\pi z)}$$

from which it is easy to derive the famous equation

$$\Gamma(1/2) = \sqrt{\pi}$$

or

$$\left(-\frac{1}{2}\right)! = \sqrt{\pi}$$

and

$$\frac{1}{2}! = \frac{\sqrt{\pi}}{2}$$

Equation 14.1.2 on the preceding page allows us to extend the gamma function over the whole complex plane, except for zero and negative integers via

$$\Gamma(z) = \frac{\Gamma(z+1)}{z}$$

and the *Euler reflection formula*:

$$\Gamma(z) \cdot \Gamma(1-z) = \frac{\pi}{\sin(\pi z)} \tag{14.1.3}$$

— see [1, chapter 5].

The command

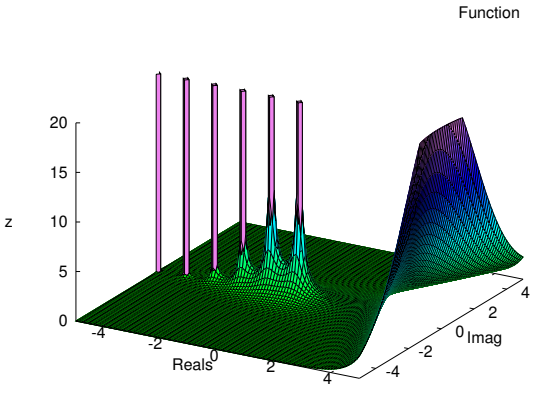


FIGURE 14.1.2. Plot of $|\Gamma(z)|$

```
plot3d (cabs(gamma(u+%i*v)), [u, -5, 5], [v, -5, 5],
[z,0,20],[xlabel,"Reals"],[ylabel,"Imag"],
[grid,100,100]);
```

produces figure 14.1.2.

Maxima also "knows" other functions of the gamma-function that are used in number-theory.

- ▷ **log_gamma**(z) — self-explanatory.
- ▷ **gamma_incomplete_lower**$(a,z) = \int_0^z e^{-t}t^{a-1}dt$
- ▷ **gamma_incomplete**$(a,z) = \int_z^\infty e^{-t}t^{a-1}dt$
- ▷ **gamma_incomplete_regularized**(a,z) =**gamma_incomplete**(a,z)/ **gamma**(a)
- ▷ **beta**(a, b) — The **beta** function is defined as **gamma**(a)***gamma**(b)/**gamma**(a+b)
- ▷ **psi**[n](x) — The derivative of **log**(**gamma**(x)) of order $n+1$. Thus, **psi**[0](x) is the first derivative, **psi**[1](x) is the second derivative, etc. Unfortunately this is only defined for *real values* of x.

There's a form of some of these functions valid for complex arguments. They must be loaded by **load**(bffac). The following functions become available:

- ▷ **bffac**(x,n) — Bigfloat version of the factorial (shifted gamma) function. The second argument is how many digits to retain and return, it's a good idea to request a couple of extra.

▷ **bfpsi**(n, z, p) — bigfloat version of **psi**[n](z), where p is the number of digits of precision. Although the documentation says it's only valid for real values of z, it appears to work for complex ones as well. **bfpsi0**(z, p)=**bfpsi**$(0, z, p)$

EXERCISES.

1. If x is a number and m is a positive integer, recall the *falling factorial* or *Pochhammer symbol*s $(x)_m$ from chapter 10 on page 191 defined by

$$(x)_m = x \cdot (x-1) \cdots (x-m+1)$$

Extend this definition to all complex values of x and m using the gamma function. Note that, if x and m are integers and $m > x$, then we get a factor $(x - x) = 0$ so the result is 0.

2. Write a Maxima function to compute falling factorials $(x)_n$ for all complex values of x and n.

3. A simple induction shows that for any positive integer, k,

$$\frac{d^k x^n}{dx^k} = n(n-1) \cdots (n-k+1)x^{n-k}$$

Extend this to arbitrary complex values of k, so one could define (for instance)

$$\frac{d^{1/2}x}{dx^{1/2}}$$

4. If

$$H(x) = \sum_{n=1}^{x} \frac{1}{n}$$

is the harmonic sum, show that

$$H(x) = \gamma + \frac{d}{dx}\left(\log\Gamma(x+1)\right) = \gamma + \frac{\Gamma'(x+1)}{\Gamma(x+1)}$$

which can be programmed

load(bffac); H(x):=**bfloat**(**%gamma**+**bfpsi0**(x+1,15))

and plotted via

```
plot3d(cabs('H(x+%i*y)),[x,-5,5],[y,-5,5],[z,0,10]);
```

to get figure 14.1.3 on the next page.

This allows $H(x)$ to be defined over the entire complex plane, except for negative integers (where it becomes ∞).

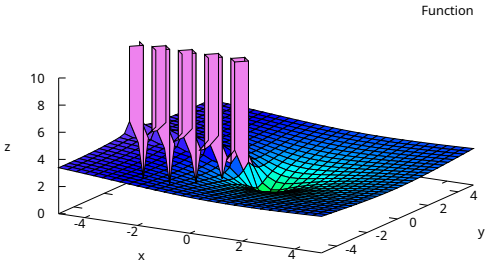


FIGURE 14.1.3. Harmonic sum on the complex plane

5. It turns out that the series

$$h(z) = \sum_{n=1}^{\infty} \left(\frac{1}{n} - \frac{1}{n+z} \right) = z \sum_{n=1}^{\infty} \frac{1}{nz+n^2}$$

converges (slowly!) for all complex values of z other than $z = -1, -2, \ldots$. Show that $h(z) = H(z)$, for z a positive integer (it also turns out to be true for all other values of z).

6. Compute

$$\frac{d^{1/2} e^x}{dx^{1/2}}$$

by computing that for each term of its Taylor series. Plot this function and e^x.

14.2. Elliptic integrals and elliptic functions

Elliptic integrals and functions have a long and complex history that impinges on numerous other areas of mathematics, including complex analysis, number theory, and algebraic geometry.

The general elliptic integral is anything of the form

$$\int_c^x R\left(t, \sqrt{P(t)}\right) \, dt$$

where c is a constant, $R(u, v)$ is a rational function, and $P(t)$ is a polynomial of degree 3 or 4 with no repeated roots. Any elliptic integral can be transformed into a linear combination of an integral of a rational function and Legendre elliptic integrals of the *first, second, and third kinds* — see [69].

Carl Gustav Jacob Jacobi (1804 – 1851) was a German mathematician who made fundamental contributions to elliptic functions, dynamics, differential equations, determinants, and number theory. His name is occasionally written as Carolus Gustavus Iacobus Iacobi in his Latin books, and his first name is sometimes given as Karl.

We will focus on elliptic integrals and Jacobi's elliptic *functions*, which initially arose in in an effort to parametrize the arc-length of an ellipse. The *incomplete elliptic integral of the first kind* is

$$F(x,k) = \int_0^x \frac{dt}{\sqrt{(1-t^2)(1-k^2t^2)}} = \int_0^\phi \frac{dt}{\sqrt{1-k^2\sin^2 t}}$$

where $0 \leq x \leq 1$, $0 \leq \phi \leq \pi/2$, and the quantity $0 \leq k \leq 1$ is called the *modulus* of the elliptic integral. The *complete* elliptic integral simply has $x = 1$ (or $\phi = \pi/2$).

The incomplete elliptic integral of the first kind is also sometimes written

$$F(\phi,k) = \int_0^{\sin\phi} \frac{dt}{\sqrt{(1-t^2)(1-k^2t^2)}}$$

This is the form Maxima implements:

```
elliptic_f(phi, m)
```

where $m = k^2$.

If we type

```
elliptic_f(x, 0)
```

we get

```
x
```

and if we type

```
elliptic_f(x, 1)
```

we get

$$\log\left(\tan\left(\frac{x}{2}+\frac{\pi}{4}\right)\right)$$

EXAMPLE 14.2.1. We will discuss an application of the elliptic integral of the first kind. Consider the pendulum in figure 14.2.1 on the facing page

The tangential force on the bob perpendicular to the rod is $-mg\sin\theta$, where m is the bob's mass and g is the acceleration of gravity. This force is also equal to

$$m\frac{dv_T}{dt} = m\frac{d}{dt}\left(R\frac{d\theta}{dt}\right) = mR\frac{d^2\theta}{dt^2}$$

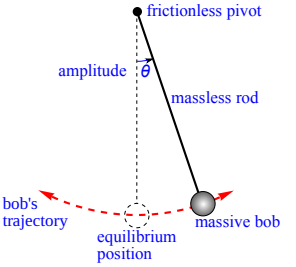


FIGURE 14.2.1. Pendulum

Equating these quantities gives

$$mR\frac{d^2\theta}{dt^2} = -mg\sin\theta$$

so the equation of motion becomes

$$\frac{d^2\theta}{dt^2} + \frac{g}{R}\sin\theta = 0 \tag{14.2.1}$$

Compare this with equation 5.3.2 on page 88 for a harmonic oscillator and figure 5.3.1 on page 88. The sine-term adds considerable complexity.

Now we multiply by $d\theta/dt$ to get

$$\left(\frac{d^2\theta}{dt^2}\right)\frac{d\theta}{dt} + \frac{g}{R}\sin\theta\frac{d\theta}{dt} = 0$$

and integrate with respect to t to get

$$\frac{1}{2}\left(\frac{d\theta}{dt}\right)^2 - \frac{g}{R}\cos\theta = C$$

where C is the constant of integration. If we set $d\theta/dt = 0$, we get $C = -\frac{g}{R}\cos\theta$, which means $\theta = \theta_0$, the pendulum's *initial* angle. Our equation becomes

$$\begin{aligned}\frac{d\theta}{dt} &= \sqrt{2\left(\frac{g}{R}\cos\theta - \frac{g}{R}\cos\theta_0\right)} \\ &= \sqrt{\frac{2g}{R}}\sqrt{\cos\theta - \cos\theta_0}\end{aligned}$$

Now we apply the identity

$$\cos\theta = 1 - 2\sin^2\left(\frac{\theta}{2}\right)$$

to get

$$\frac{d\theta}{dt} = 2\sqrt{\frac{g}{R}}\sqrt{\sin^2\left(\frac{\theta_0}{2}\right) - \sin^2\left(\frac{\theta}{2}\right)}$$

so

$$\frac{dt}{d\theta} = \frac{1}{2}\sqrt{\frac{R}{g}}\frac{1}{\sqrt{k^2 - \sin^2\left(\frac{\theta}{2}\right)}}$$

where $k = \sin(\theta_0/2)$. The complete *period* of the pendulum is

$$T = 4 \cdot \frac{1}{2}\sqrt{\frac{R}{g}} \cdot \int_0^{\theta_0} \frac{d\theta}{\sqrt{k^2 - \sin^2\left(\frac{\theta}{2}\right)}} \tag{14.2.2}$$

Now we do a u-substitution, setting $\sin(\theta/2) = k\sin(u)$, which implies that

$$\cos\left(\frac{\theta}{2}\right)\frac{d\theta}{2} = k\cos(u)du$$

or

$$\begin{aligned} d\theta &= \frac{2k\cos(u)}{\cos\left(\frac{\theta}{2}\right)}du = \frac{2k\sqrt{1-\sin^2(u)}}{\sqrt{1-k^2\sin^2(u)}}du \\ &= \frac{2\sqrt{k^2 - \sin^2(\theta/2)}}{\sqrt{1-k^2\sin^2(u)}}du \end{aligned} \tag{14.2.3}$$

Now we plug equation 14.2.3 into equation 14.2.2 to get the complete elliptic integral of the first kind

$$T = 4\sqrt{\frac{R}{g}} \cdot \int_0^{\pi/2} \frac{du}{\sqrt{1-k^2\sin^2(u)}} \tag{14.2.4}$$

where we note that when $\theta = \theta_0$, $k = \sin(\theta_0/2) = \sin(\theta_0/2)\sin(u)$, so $\sin(u) = 1$, and $u = \pi/2$.

The *inverse* of the elliptic integral of the first kind is the elliptic *function* $\mathrm{sn}(x,k)$, a generalization of the sine-function. Indeed, $\mathrm{sn}(x,0) = \sin x$. These were first studied by Jacobi.

Maxima implements these things via

```
jacobi_sn(u, m);
```

where $m = k^2$.

See figure 14.2.2 on the next page[1], which is a plot of the Jacobi ellipse

$$x^2 + \frac{y^2}{b^2} = 1$$

[1]This beautiful diagram was taken from Wikimedia, without attribution.

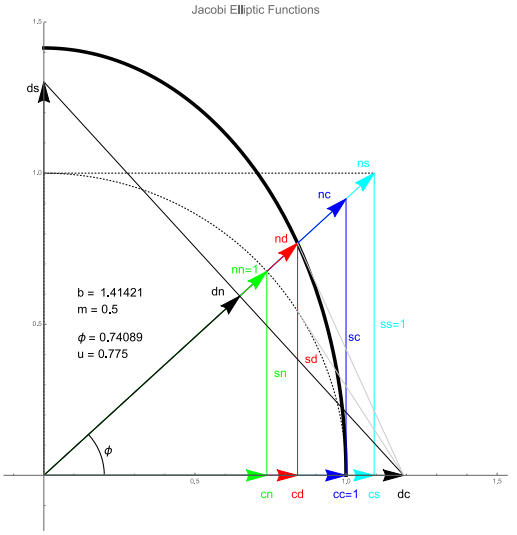


FIGURE 14.2.2. Jacobi functions

with b real. The quantities m and b are related via

$$b^2 = \frac{1}{m}$$

The twelve Jacobi functions, **vw**(u,m), are shown, where $w, v = s, c, d, n$ and any function of the form **ww** is defined to be 1 for the sake of completeness. In general

$$\mathbf{wv}(u, m) = \frac{1}{\mathbf{vw}(u, m)}$$

In figure 14.2.2 note that

$$\sin\phi = \mathrm{sn}(u, m)$$
$$\cos\phi = \mathrm{cn}(u, m)$$

where u is arc-length along the ellipse (from the point (1,0)). This implies that

$$\mathrm{sn}^2(u, m) + \mathrm{cn}^2(u, m) = 1$$

for all u, m.

Maxima "knows" some properties of these functions:

```
jacobi_sn(u, 0);
```

produces $\sin(u)$ and typing

```
jacobi_sn(u, 1);
```

produces $\tanh(u)$.

The command

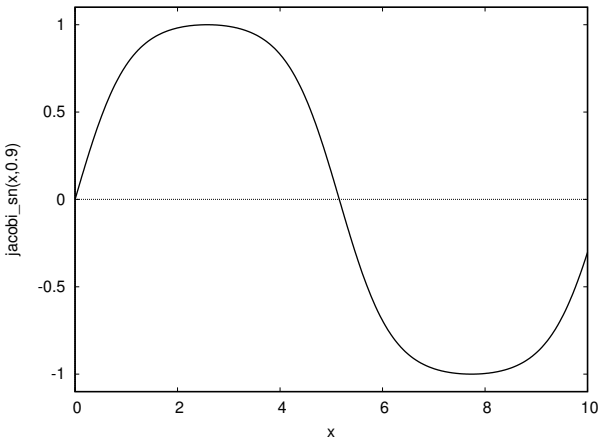


FIGURE 14.2.3. Plot of sn$(x, .9)$

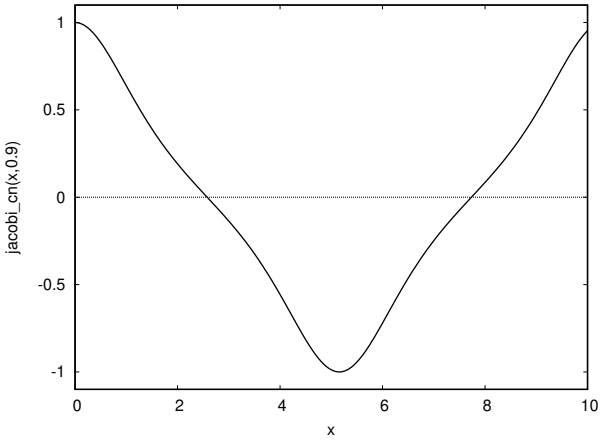


FIGURE 14.2.4. Plot of cn$(x, .9)$

```
plot2d(jacobi_sn(x,.9),[x,0,10],[style,[lines,2,5]]);
```

produces the plot in figure 14.2.3.

Notice that the graph of sn$(x, .9)$ looks like a sine curve that has been "rounded".

Figure 14.2.4 shows the complementary function.

Other elliptic functions that Maxima implements include

```
jacobi_dn(u, m);
jacobi_ns(u,m)=1/jacobi_sn(u,m);
```

```
jacobi_sc(u,m)=jacobi_sn(u,m)/jacobi_cn(u,m);
jacobi_cs(u,m)=jacobi_cn(u,m)/jacobi_sn(u,m);
jacobi_nd(u,m)=1/jacobi_dn(u, m);
jacobi_ds(u,m)=jacobi_dn(u, m)/jacobi_sn(u,m);
jacobi_dc(u,m)=jacobi_dn(u,m)/jacobi_cs(u,m);
```

Maxima also implements the *inverses* of all of these functions, although the inverse of $\mathrm{sn}(u,m)$ is technically the elliptic integral of the first kind. Their names are the word '**inverse_**' followed by the Maxima name of the function.

Now we will consider the elliptic integrals of the *second*

$$E(\phi,m)=\int_0^{\phi}\sqrt{1-m\sin^2\theta}d\theta=\int_0^{\sin\phi}\frac{\sqrt{1-mt^2}}{\sqrt{1-t^2}}dt$$

and *third* kinds

$$\Pi(n,\varphi,m)=\int_0^{\sin\varphi}\frac{dt}{(1-nt^2)\sqrt{(1-mt)^2(1-t^2)}}$$

The number n is called the *characteristic* and can take on any value. These are coded into Maxima, respectively, as

```
elliptic_e (phi, m)
```

and

```
elliptic_pi (n, phi, m)
```

EXERCISES.

1. Using elliptic functions, give a parametric plot of an ellipse with $m = .9$. Hint: Closely examine figure 14.2.2 on page 253.

2. What's the period of a pendulum with $R = 1$ foot, $g = 32$ feet/second^2, and a starting angle of 45°?

3. In a pendulum, as the starting angle approaches 180° the period approaches ∞. What is going on?

14.3. Bessel functions

Bessel functions were first defined by Daniel Bernoulli and generalized by Friedrich Bessel as solutions to Bessel's differential equation

$$x^2\frac{d^2y}{dx^2}+x\frac{dy}{dx}+(x^2-\alpha^2)y=0 \tag{14.3.1}$$

for a complex number, α, which is called the *order* of the Bessel function. The most significant values for α are integer and half-integer values. The integer values arise from converting the two-dimensional wave equation (see section 4.6.4 on page 77) to polar or cylindrical coordinates, and the half-integer values arise from converting it to spherical coordinates.

The general solution to equation 14.3.1 on the previous page is

$$AJ_\alpha(x) + BY_\alpha(x)$$

where $J_\alpha(x)$ and $Y_\alpha(x)$ are called, respectively, Bessel functions of the first and second kind.

> Friedrich Wilhelm Bessel (1784 – 1846) was a German astronomer, mathematician, and physicist. He was the first astronomer who determined reliable values for the distance from the sun to another star by the method of parallax. Certain important mathematical functions were named Bessel functions after Bessel's death, though they had originally been discovered by Daniel Bernoulli before being generalized by Bessel.

To see how Bessel's equation arises, we convert the wave equation into polar coordinates.

In polar coordinates, (r, θ), we have

$$\frac{\partial^2\psi}{\partial x^2} + \frac{\partial^2\psi}{\partial y^2} = \frac{1}{r}\frac{\partial}{\partial r}\left(r\frac{\partial\psi}{\partial r}\right) + \frac{1}{r^2}\frac{\partial^2\psi}{\partial\theta^2} = \frac{1}{c^2}\frac{\partial^2\psi}{\partial t^2}$$

We write $\psi(r, \theta, t) = \tau(t)\Phi(r, \theta)$ to get

$$\tau\frac{1}{r}\frac{\partial}{\partial r}\left(r\frac{\partial\Phi}{\partial r}\right) + \tau\frac{1}{r^2}\frac{\partial^2\Phi}{\partial\theta^2} = \Phi\frac{1}{c^2}\frac{\partial^2\tau}{\partial t^2}$$

or

$$\frac{1}{\Phi r}\frac{\partial}{\partial r}\left(r\frac{\partial\Phi}{\partial r}\right) + \frac{1}{\Phi r^2}\frac{\partial^2\Phi}{\partial\theta^2} = \frac{1}{\tau c^2}\frac{\partial^2\tau}{\partial t^2} = -\lambda$$

where λ is a constant (the only way a function of t could equal a function of other, independent, variables). So we get

$$\frac{d^2\tau}{dt^2} + \lambda\tau = 0$$

$$\frac{1}{r}\frac{\partial}{\partial r}\left(r\frac{\partial\Phi}{\partial r}\right) + \frac{1}{r^2}\frac{\partial^2\Phi}{\partial\theta^2} + \lambda\Phi = 0$$

Now, we write $\Phi(r, \theta) = R(r)\Xi(\theta)$ and get

$$\Xi\frac{1}{r}\frac{\partial}{\partial r}\left(r\frac{\partial R}{\partial r}\right) + R\frac{1}{r^2}\frac{\partial^2\Xi}{\partial\theta^2} + \lambda R\Xi = 0$$

and dividing by $R(r)\Xi(\theta)$ and multiplying by r^2 gives

$$\frac{r}{R}\frac{\partial}{\partial r}\left(r\frac{\partial R}{\partial r}\right) + \frac{1}{\Xi}\frac{\partial^2\Xi}{\partial\theta^2} + \lambda r^2 = 0$$

or

$$\frac{r}{R}\frac{\partial}{\partial r}\left(r\frac{\partial R}{\partial r}\right)+\lambda r^2=-\frac{1}{\Xi}\frac{\partial^2\Xi}{\partial\theta^2}$$

Again, we are faced with a situation where a function of one variable is equal to one of another, so they both equal the same constant

$$\frac{r}{R}\frac{d}{dr}\left(r\frac{dR}{dr}\right)+\lambda r^2=\kappa$$

$$\frac{1}{\Xi}\frac{d^2\Xi}{d\theta^2}=-\kappa$$

The top equation is equal to

$$r^2\frac{d^2R}{dr^2}+r\frac{dR}{dr}+R\lambda r^2=\kappa R$$

or

$$r^2\frac{d^2R}{dr^2}+r\frac{dR}{dr}+R(\lambda r^2-\kappa)=0$$

A change of scale ($\bar{r}=r\sqrt{\lambda}$) allows us to get rid of λ, and we get Bessel's differential equation. Maxima implements Bessel functions. The first kind is called **bessel_j**(v,z) with mathematical notation $J_v(z)$. It can be defined via

$$J_v(z)=\sum_{k=0}^{\infty}\frac{(-1)^k 2^{v-2k} z^{v+2k}}{k!\Gamma(v+k+1)}$$

or an integral representation

$$J_v(z)=\frac{1}{\pi}\int_0^{\pi}\cos\left(vt-z\sin t\right)dt$$

If we type

```
plot2d([bessel_j(0,x),bessel_j(1,x),bessel_j(2,x)],
[x,0,10],[style,[lines,2,5],
[points,1,4,5],[lines,2,1]]);
```

gives us figure 14.3.1 on the following page.

The second kind is called **bessel**_y(v,z) with mathematical notation $Y_v(z)$. These functions have a singularity at 0, going to $-\infty$. We can graph them

```
plot2d([bessel_y(0,x),bessel_y(1,x),bessel_y(2,x)],
[x,.1,10],[y,-4,1],[style,[lines,2,5],
[points,1,4,5],[lines,2,1]]);
```

to get figure 14.3.2 on the next page.

These are related to the J-Bessel functions via the equation

$$Y_k(z)=\frac{\cos(\pi k)J_k(z)-J_{-k}(z)}{\sin(\pi k)}$$

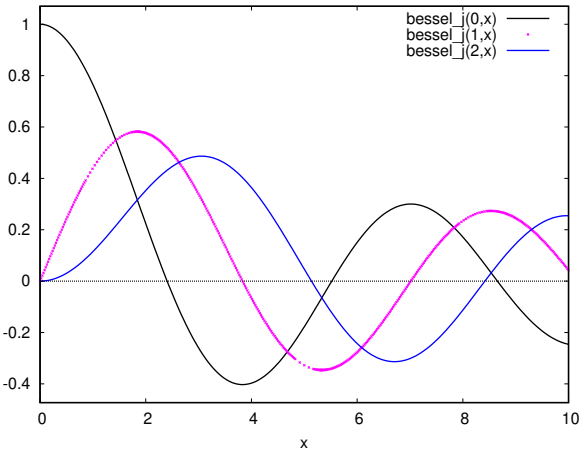


FIGURE 14.3.1. First three Bessel J-functions

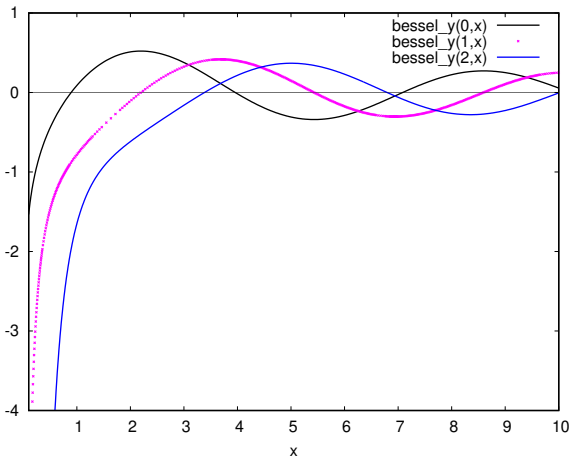


FIGURE 14.3.2. First three Bessel Y-functions

when k is *not* an integer. When it *is* (an integer), take the limit as k approaches its integer value.

Bessel functions are used for

- ▷ Electromagnetic waves in a cylindrical waveguide.
- ▷ Pressure amplitudes of inviscid rotational flows.
- ▷ Heat conduction in a cylindrical object.
- ▷ Modes of vibration of a thin circular or annular acoustic membrane or thicker plates such as sheet metal.
- ▷ Solutions to the radial Schrödinger equation (in spherical and cylindrical coordinates) for a free particle.

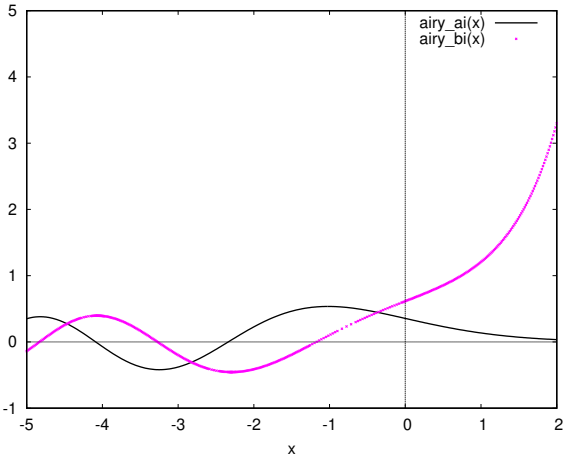


FIGURE 14.4.1. The Airy Functions

- ▷ Solving for patterns of acoustical radiation.
- ▷ Frequency-dependent friction in circular pipelines.
- ▷ Dynamics of floating bodies.
- ▷ Diffraction from helical objects, including DNA.
- ▷ Probability density function of product of two normally distributed random variables.

14.4. Airy functions

The two linearly independent Airy functions $\mathrm{Ai}(x)$ and $\mathrm{Bi}(x)$ are solutions to Airy's differential equation

$$\frac{d^2y}{dx^2} - xy = 0$$

These functions have the interesting property that they switch from being oscillatory when $x < 0$ to being exponential when $x > 0$. Their Maxima definitions are, respectively,

```
airy_ai(x)
```

and

```
airy_bi(x)
```

Their "switching" behavior is clear from figure 14.4.1.

For real values of x we have the integral formulas

$$\mathrm{Ai}(x) = \frac{1}{\pi}\int_0^\infty \cos\left(\frac{t^3}{3} + xt\right) dt$$

$$\mathrm{Bi}(x) = \frac{1}{\pi}\int_0^\infty \left(e^{-t^3/3+xt} + \sin\left(\frac{t^3}{3} + xt\right)\right) dt$$

Maxima also implements *first derivatives* of the Airy functions

```
airy_dai(x)
```

and

```
airy_dbi(x)
```

The Airy function is the solution to the time-independent Schrödinger equation for a particle confined within a triangular potential well and for a particle in a one-dimensional constant force field.

> Sir George Biddell Airy (1801 – 1892) was an English mathematician and astronomer, and the seventh Astronomer Royal from 1835 to 1881. His many achievements include work on planetary orbits, measuring the mean density of the Earth, a method of solution of two-dimensional problems in solid mechanics and, in his role as Astronomer Royal, establishing Greenwich as the location of the prime meridian.

Forming the Fourier transform of Airy's differential equation gives us the Fourier transform of $\mathrm{Ai}(x)$:

$$\mathcal{F}(\mathrm{Ai}(x))(s) = e^{(2\pi s)^3/3}$$

14.5. Logarithmic and exponential integrals

We begin with the *logarithmic integral*, defined via

$$\mathrm{li}(x) = \int_0^x \frac{dt}{\log t} \tag{14.5.1}$$

The Maxima command for this is

```
expintegral_li(x)
```

The command

```
plot2d(expintegral_li(x),[x,0,5],[y,-5,5]);
```

produces the plot in figure 14.5.1 on the next page.

The function $1/\log(x)$ has a singularity at $x = 1$, in which case, we interpret equation 14.5.1 as the *Cauchy principal value*:

$$\mathrm{li}(x) = \lim_{\epsilon\to 0^+}\left(\int_0^{1-\epsilon} \frac{dt}{\log t} + \int_{1+\epsilon}^x \frac{dt}{\log t}\right) \tag{14.5.2}$$

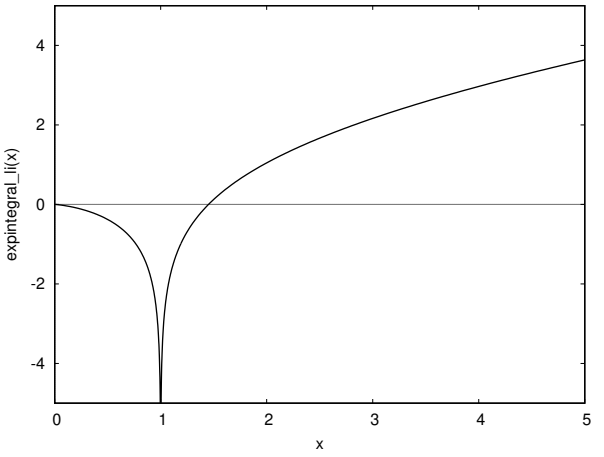


FIGURE 14.5.1. The li-function

whenever $x > 1$.

We also have

$$\mathrm{Li}(x) = \int_2^x \frac{dt}{\log t}$$

for $x \geq 2$, which avoids the singularity at $x = 1$. We have the relation

$$\mathrm{Li}(x) = \mathrm{li}(x) - \mathrm{li}(2)$$

Note: the Maxima documentation *incorrectly* states that

```
expintegral_li(x)
```

computes $\mathrm{Li}(x)$, but it actually[2] computes $\mathrm{li}(x)$. We also have *dilogarithms* and *polylogarithms*

$$\mathrm{li}_k(z) = \sum_{n=1}^{\infty} \frac{z^n}{n^k}$$

In Maxima, this is coded as

```
li[k](z)
```

We also have the *exponential integrals*:

(1) There is the integral

$$\mathrm{Ei}(x) = -\int_{-x}^{\infty} \frac{e^{-t}}{t} dt$$

and the Maxima command is

```
expintegral_ei(x)
```

[2]As its name implies.

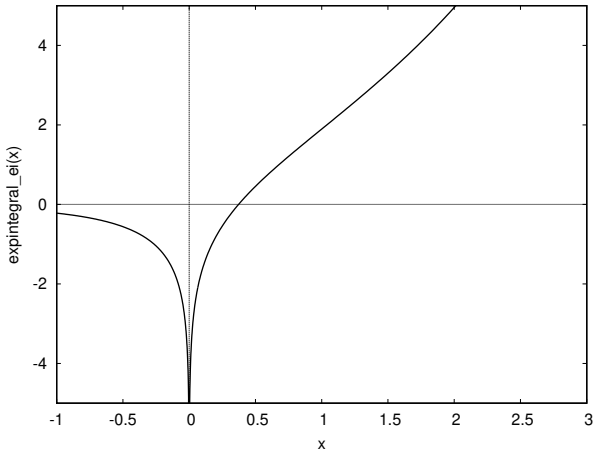


FIGURE 14.5.2. The Ei-function

where the singularity at $t = 0$ is handled via the Cauchy principal value (as in equation 14.5.2 on page 260). This function is related to $\mathrm{li}(x)$ via

$$\mathrm{li}(e^x) = \mathrm{Ei}(x) \tag{14.5.3}$$

and

```
plot2d(expintegral_ei(x),[x,-1,3])
```

produces figure 14.5.2. Note,

```
expintegral_li(x)
```

can produce incorrect results if x is a complex number[3]. In this case, equation 14.5.3 gives the correct value.

(2) The integral

$$\mathrm{E}_1(x) = \int_x^\infty \frac{e^{-t}}{t} dt$$

with a Maxima command

```
expintegral_e1(x)
```

that produces figure 14.5.3 on the facing page.

(3) The *sine integral*

$$\mathrm{Si}(x) = \int_0^x \frac{\sin t}{t} dt$$

coded in Maxima as

[3]Maxima documentation states that it's only defined for real values of $x > 1$.

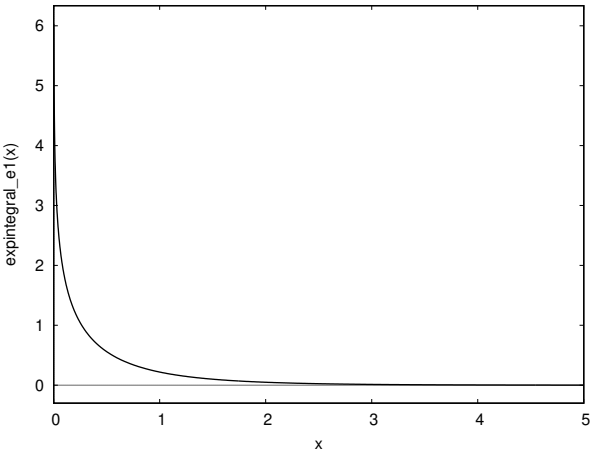


FIGURE 14.5.3. E_1-function

expintegral_si(x)

which is plotted in figure 14.5.4 on the next page. It is well-known that

$$\lim_{x\to\infty} \mathrm{Si}(x) = \frac{\pi}{2}$$

It turns out that the sine-integral is closely related to *Gibbs phenomena,* mentioned on page 63. The size of the jump (overshoot or undershoot) is approximately 9% and is exactly given by

$$\frac{\mathrm{Si}(\pi)}{\pi} - \frac{1}{2} = 0.08948987223608362\ldots$$

called the *Wilbraham-Gibbs constant* — see [30]. Incidentally, the phenomena was discovered by Wilbraham fifty years before Gibbs mentioned it, but it's *named* after Gibbs (as per mathematical tradition!).

(4) The *cosine-integral*

$$\mathrm{Ci}(x) = -\int_x^\infty \frac{\cos t}{t} dt$$

coded in Maxima as

expintegral_ci(x)

and plotted in figure 14.5.5 on the following page.

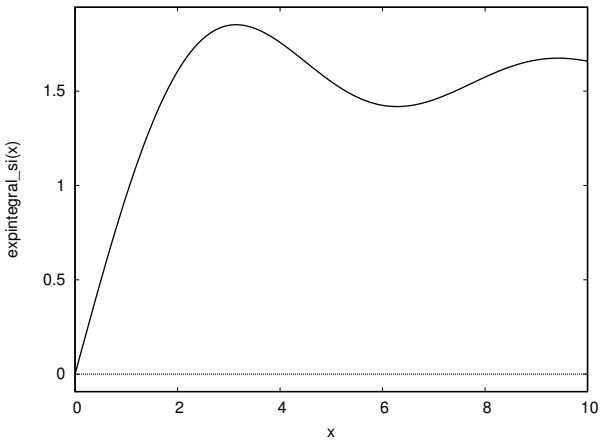


FIGURE 14.5.4. The sine-integral

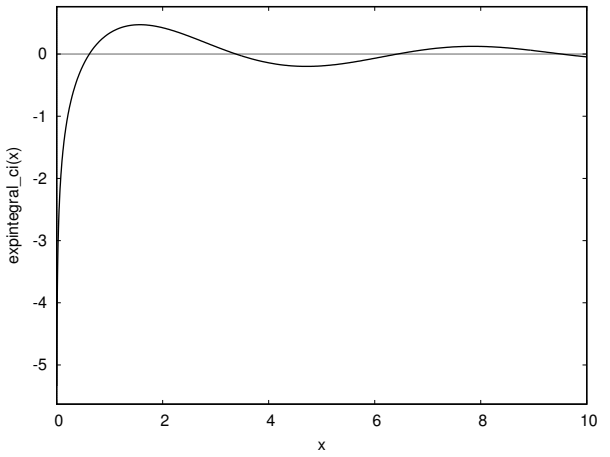


FIGURE 14.5.5. The cosine-integral

(5) The *hyperbolic sine integral*:

$$\mathrm{Shi}(x) = \int_0^x \frac{\sinh t}{t} dt$$

coded in Maxima via

```
expintegral_shi(x)
```

(6) And the *hyperbolic cosine integral*:

$$\mathrm{Chi}(x) = \gamma + \log x + \int_0^x \frac{\cosh t - 1}{t} dt$$

coded in Maxima via

```
expintegral_chi(x)
```

Here γ is the Euler–Mascheroni constant, coded in Maxima as **%gamma**.

EXERCISES.

1. Write a function to compute $\mathrm{li}(x)$ from equation 14.5.2 on page 260 (i.e., without using Maxima's **expintegral_li**-function).

2. Find a relation between the functions $\mathrm{E}_1(x)$, $\mathrm{Si}(x)$, and $\mathrm{Ci}(x)$.

14.6. Lambert functions

The *Lambert-W function* is the inverse function to

$$xe^x$$

— which is sometimes called the *product logarithm*, because the conventional logarithm is the inverse function to e^x. Another name for it is the *omega function*. Like the logarithm, the W-function has many branches, denoted $W_n(z)$, and

$$y = xe^x$$

if and only if

$$x = W_n(y)$$

for some integer n. Unlike the logarithm, these branches are not equally spaced.

Two of them, $W_0(z)$ and $W_{-1}(z)$ (often written $W_m(z)$) are real-valued for real values of z and the others are complex valued. If z is real-valued, then $W_0(z)$ is well-defined for $0 \leq z$, and $W_{-1}(z)$ is defined for $-1/e \leq z \leq 0$.

The command

```
lambert_w(x)
```

computes $W_0(x)$ and the command

```
plot2d(lambert_w(x),[x,0,10]);
```

produces the plot in figure 14.6.1 on the following page.

The other branches, $W_k(z)$, of the Lambert function are given by the command

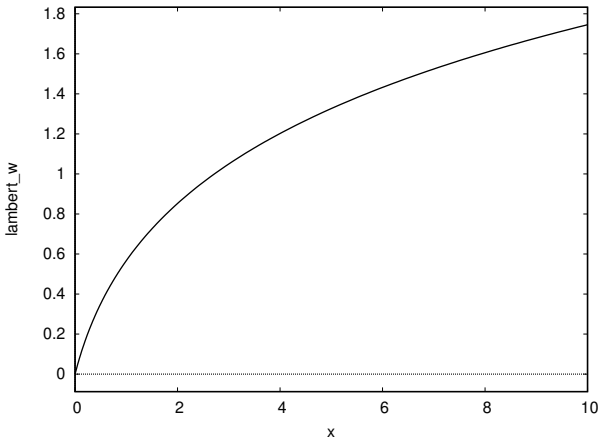


FIGURE 14.6.1. The Lambert W function

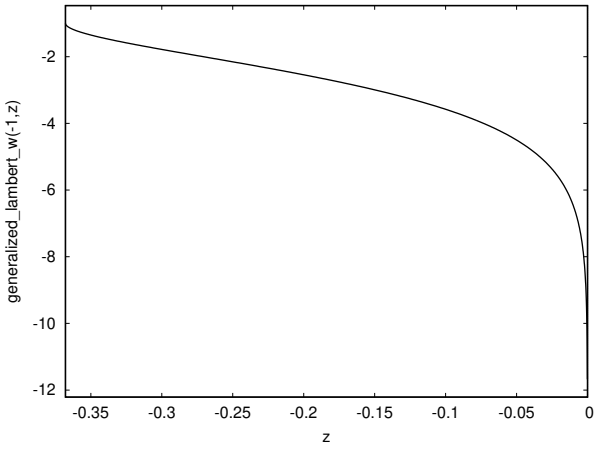


FIGURE 14.6.2. The Lambert function $W_{-1}(z)$

```
generalized_lambert_w(k, z)
```

where

```
generalized_lambert_w(0, z)=lambert_w(z)
```

The command

```
plot2d(generalized_lambert_w(-1, z),[z,-1/%e,0])
```

produces the plot in figure 14.6.2.

The Lambert function has many applications in physics — see the paper [66].

Johann Heinrich Lambert (1728 – 1777) was a polymath from the Republic of Mulhouse (today part of the province of Alsace, in France), who made important contributions to the subjects of mathematics, physics (particularly optics), philosophy, astronomy, and map projections.

The Lambert functions all satisfy the differential equation

$$\frac{dW(z)}{dz} = \frac{W(z)}{z(1+W(z))} = \frac{1}{z+e^{W(z)}}$$

which implies that it has applications to Michaelis–Menten kinetics involving the biochemistry of enzyme-catalysed reactions of one substrate and one product. it also has applications to ecology and evolution — see [40].

The Lambert function also has applications to the problem of the *infinite power tower*:

$$y = x^{x^{x^{\cdot^{\cdot^{\cdot}}}}}$$

One might ask what this even means. Define a sequence

$$\{t_k(x)\}$$

by

$$\begin{aligned} t_1(x) &= x \\ t_{k+1}(x) &= x^{t_k(x)} \qquad \text{for } k \text{ an integer } \geq 1 \end{aligned}$$

and

$$y = t_\infty(x) = \lim_{k\to\infty} t_k(x) \tag{14.6.1}$$

if this limit exists.

Since the tower is *infinite*, we have

$$t_{\infty+1}(x) = t_\infty(x)$$

or

$$y = x^y = e^{y\log(x)} \tag{14.6.2}$$

so that

$$ye^{-y\log(x)} = 1$$

If we multiply by $-\log(x)$, we get

$$-\log(x)ye^{-y\log(x)} = -\log(x)$$

so

$$-\log(x)y = W(-\log(x))$$

and

$$y = -\frac{W(-\log(x))}{\log(x)} = t_\infty(x) \tag{14.6.3}$$

Surprisingly, this limit exists for many values of x.

If we take the $1/y$ power of equation 14.6.2 on the preceding page, we get

$$y^{1/y} = x \tag{14.6.4}$$

which shows that equation 14.6.3 actually solves for y satisfying equation 14.6.4, given x.

EXERCISES.

1. Compute $t_\infty(\sqrt{2})$ as in equations 14.6.1 on the previous page and 14.6.3.

2. Estimate the largest finite value of d such that $t_\infty(\sqrt{2}+d)$ is well-defined. Hint: start with $d = .01$. If the sequence $\{t_k(x)\}$ diverges, equation 14.6.3 will give *complex numbers* for $t_\infty(x)$. What is the limiting value (i.e., the largest finite value you can get) for $t_\infty(x)$?

3. If we define

```
tow(x):=-lambert_w(-log(x))/log(x);
```

we can compute

```
x:tow(3)
```

Why is this well-defined?

4. If

$$y = x^x$$

with $x > 1$, find a formula for x as a function of y using Lambert functions. Hint: use equation 14.6.3.

CHAPTER 15

The Zeta function

"God may not play dice with the universe, but something strange is going on with prime numbers. "
— taken from a talk given by Carl Pomerance on the Erdős-Kac theorem, San Diego in January 1997.

15.1. Properties of the ζ-function

In this section, we will discuss Riemann's groundbreaking research (see [51] and the English translation, [70]) on Euler's Zeta function — a function so special it rates its own chapter. This involves the theory of functions of a complex variable, so the reader needs to be familiar with that — see [1].

Georg Friedrich Bernhard Riemann (1826–1866) was an influential German mathematician who made contributions to many fields including analysis, number theory and differential geometry. Riemann's work in differential geometry provided the mathematical foundation for Einstein's Theory of General Relativity (see [47]).

It all begins with Euler's *zeta-function*:

$$\zeta(s) = \sum_{n=1}^{\infty} \frac{1}{n^s} \tag{15.1.1}$$

Maxima has a command to compute this

```
zeta(x)
```

when x is an integer and when exact values when they are known, so

```
zeta(2)
```

gives

$$\frac{\pi^2}{6}$$

In general (i.e., for arbitrary complex numbers), the command

```
bfzeta(x,n)
```

computes $\zeta(x)$ as a bigfloat with n digits.

Riemann first noted

$$\int_0^\infty e^{-nx}x^{s-1}dx = \frac{\Gamma(s)}{n^s}$$

which you can confirm using Maxima (and answering questions about n and s). It follows that

$$\int_0^\infty \left(e^{-x} + e^{-2x} + e^{-3x} + \cdots\right) x^{s-1}dx = \Gamma(s)\left(\frac{1}{1^s} + \frac{1}{2^s} + \frac{1}{3^s} + \cdots\right)$$

The infinite series

$$e^{-x} + e^{-2x} + e^{-3x} + \cdots = e^{-x} + \left(e^{-x}\right)^2 + \left(e^{-x}\right)^3 + \cdots$$

is the geometric series equal to

$$e^{-x}\frac{1}{1-e^{-x}} = \frac{e^x}{e^x}\frac{e^{-x}}{1-e^{-x}} = \frac{1}{e^x-1}$$

so we get

$$\Gamma(s)\zeta(s) = \int_0^\infty \frac{x^{s-1}}{e^x-1}dx$$

It's interesting that the mere subtraction of 1 in the denominator transforms this integral from $\Gamma(s)$ to $\Gamma(s)\zeta(s)$.

Next, Riemann extends the definition of $\zeta(s)$ to the whole complex plane by evaluating the integral

$$\int_C \frac{(-z)^{s-1}}{e^z-1}dz$$

where C is the contour in figure 15.1.1 on the facing page, and the branch of $\log(-z)$ used in calculating $(-z)^{s-1}$ is the one where the logarithm is real for negative values of z.

We will ultimately take the limit as $\epsilon \to 0$ and the circle around the origin has a radius that goes to 0 as well. We split the contour, C, into three parts. Note that $-z = e^{\pm\pi i}\cdot z$ and our three parts are

(1) γ_1, the path above the real axis from ∞ to ϵ. Since the argument of z goes from $-\pi$ to $+\pi$, we will start with $-z = e^{-\pi i}z$

$$\begin{aligned}\int_{\gamma_1} \frac{(-z)^{s-1}}{e^z-1}dz &= \int_\infty^\epsilon e^{-(s-1)\pi i}\frac{z^{s-1}}{e^z-1}dz\\ &= e^{-s\pi i}e^{\pi i}\int_\infty^\epsilon \frac{z^{s-1}}{e^z-1}dz\\ &= e^{-s\pi i}\int_\epsilon^\infty \frac{z^{s-1}}{e^z-1}dz\end{aligned}$$

(2) γ_2, the small circle around the origin. In this case

$$\frac{1}{e^z-1} \sim \frac{1}{z}$$

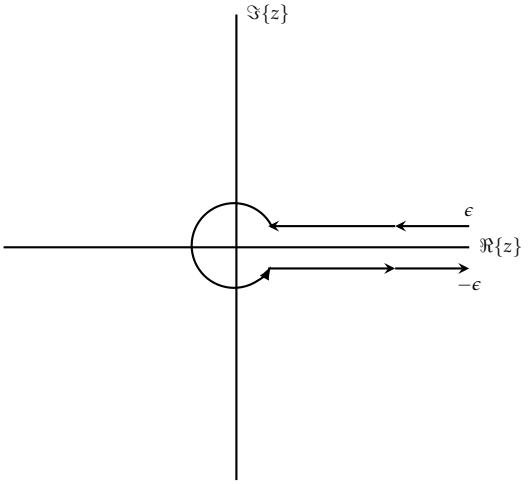


FIGURE 15.1.1. The contour, C

as the circle shrinks to the origin, so

$$\frac{z^{s-1}}{e^z-1} \sim z^{s-2}$$

which has a residue of 0 unless $s = 1$ — but we already know $\zeta(s)$ has a singularity at $s = 1$.

(3) γ_3, the path below the real axis from ϵ to ∞. Here $-z = e^{+\pi i}z$ (since we have wrapped around the origin in a positive direction) so

$$\begin{aligned}\int_{\gamma_3} \frac{(-z)^{s-1}}{e^z-1}dz &= \int_\epsilon^\infty e^{+(s-1)\pi i}\frac{z^{s-1}}{e^z-1}dz \\ &= e^{s\pi i}e^{-\pi i}\int_\epsilon^\infty \frac{z^{s-1}}{e^z-1}dz \\ &= -e^{s\pi i}\int_\epsilon^\infty \frac{z^{s-1}}{e^z-1}dz\end{aligned}$$

We conclude (as Riemann did) that

$$(e^{-\pi i s} - e^{\pi i s})\Gamma(s)\zeta(s) = \int_C \frac{(-z)^{s-1}}{e^z-1}dz$$

or

$$2\sin(\pi s)\Gamma(s)\zeta(s) = i\int_C \frac{(-z)^{s-1}}{e^z-1}dz \tag{15.1.2}$$

for $s \neq 1$. Note that $\sin(\pi s)\Gamma(s)$ is finite for $s < 0$. Even though $\Gamma(s)$ has singularities at all negative integers, $\sin(\pi s)$ vanishes for all

integers, so it "cancels out" these singularities. The Euler reflection formula 14.1.3 on page 246 shows that

$$2\sin(\pi s)\Gamma(s) = \frac{2\pi}{\Gamma(1-s)}$$

which implies that it's also nonzero for $s < 0$.

If we traverse the contour, C, in a *negative* direction, this inverts the sign of the result. If we do this and invert the interior of this contour (i.e., regard it as enclosing the *exterior* of the narrow strip around the positive real-axis) the integral is inverted again — i.e., its *original* value is restored.

We get a formula

$$2\sin(\pi s)\Gamma(s)\zeta(s) = i\int_{\overline{-C}} \frac{(-z)^{s-1}}{e^z - 1}dz$$

where $-C$ is the contour C traversed in the opposite direction and $\overline{-C}$ represents the result of regarding it as enclosing the entire complex plane *outside* the narrow strip around the positive real-axis.

Evaluating this integral is straightforward, using the calculus of residues. Nonzero residues occur when $s = 2n\pi i$ where n is a nonzero integer, and they are equal to $(-2n\pi i)^{s-1}\cdot(-2\pi i)$.

We get the formula, well-defined for $s < 0$:

$$2\sin(\pi s)\Gamma(s)\zeta(s) = (2\pi)^s \sum_{n=1}^{\infty} n^{s-1}\left(i^{s-1} + (-i)^{s-1}\right)$$

which is a kind of complement to equation 15.1.1 on page 269. It immediately implies that $\zeta(-2n) = 0$ for n a positive integer. At this point, Riemann observed that the Euler reflection formula for the gamma function (equation 14.1.3 on page 246) implies one for the zeta function:

$$\Gamma\left(\frac{s}{2}\right)\pi^{-\frac{s}{2}}\zeta(s) = \Gamma\left(\frac{1-s}{2}\right)\pi^{-\frac{1-s}{2}}\zeta(1-s) \tag{15.1.3}$$

which we can call the *Riemann reflection formula*. Note that when $s = 1/2$, both sides of this equation become identical. This suggested to Riemann that the line $\Re(s) = 1/2$ is critical.

Starting with[1]

$$\int_0^\infty e^{-n^2\pi x}x^{\frac{s}{2}-1}dx = \frac{1}{n^s}\Gamma\left(\frac{s}{2}\right)\pi^{-\frac{s}{2}}$$

so that

$$\Gamma\left(\frac{s}{2}\right)\pi^{-\frac{s}{2}}\zeta(s) = \int_0^\infty \psi(x)x^{\frac{s}{2}-1}dx \tag{15.1.4}$$

[1] You can easily check this with Maxima!

where

$$\psi(x) = \sum_{n=1}^{\infty} e^{-n^2 \pi x}$$

for $x > 0$.

Riemann then cited the remarkable result

$$2\psi(x) + 1 = x^{-1/2}\left(2\psi\left(\frac{1}{x}\right) + 1\right) \tag{15.1.5}$$

and gave a reference to a paper of Jacobi which doesn't contain this formula. It appears at the bottom of page 307 in *another* paper of Jacobi (see [34]) which attributes the result to unpublished work of Poisson. See appendix C on page 293 for a detailed proof.

We rewrite the integral in equation 15.1.4 on the preceding page as

$$\Gamma\left(\frac{s}{2}\right)\pi^{-\frac{s}{2}}\zeta(s) = \int_1^{\infty} \psi(x) x^{\frac{s}{2}-1} dx + \int_0^1 \psi(x) x^{\frac{s}{2}-1} dx \tag{15.1.6}$$

and we rewrite equation 15.1.5 as

$$\psi(x) = x^{-1/2}\left(\psi\left(\frac{1}{x}\right) + \frac{1}{2}\right) - \frac{1}{2} \tag{15.1.7}$$

and use it to reformulate the term from 0 to 1 in equation 15.1.6 as

$$\begin{aligned}\int_0^1 \psi(x) x^{\frac{s}{2}-1} dx &= \int_0^1 \left(x^{-1/2}\left(\psi\left(\frac{1}{x}\right) + \frac{1}{2}\right) - \frac{1}{2}\right) x^{\frac{s}{2}-1} dx \\ &= \int_0^1 x^{-1/2} x^{\frac{s}{2}-1} \psi\left(\frac{1}{x}\right) dx \\ &\quad + \int_0^1 x^{\frac{s}{2}-1}\left(\frac{x^{-1/2}}{2}\right) dx - \int_0^1 x^{\frac{s}{2}-1}\frac{1}{2} dx\end{aligned}$$

so we get

$$\begin{aligned}\Gamma\left(\frac{s}{2}\right)\pi^{-\frac{s}{2}}\zeta(s) &= \int_1^{\infty} \psi(x) x^{\frac{s}{2}-1} dx + \int_0^1 \psi\left(\frac{1}{x}\right) x^{\frac{s-3}{2}} dx \\ &\quad + \frac{1}{2}\int_0^1 \left(x^{\frac{s-3}{2}} - x^{\frac{s}{2}-1}\right) dx\end{aligned}$$

Now we do a u-substitution on the second of these three terms on the right

$$u = \frac{1}{x}$$

so

$$dx = -\frac{1}{u^2} du$$

$$\begin{aligned}\int_0^1 \psi\left(\frac{1}{x}\right) x^{\frac{s-3}{2}} dx &= -\int_\infty^1 \psi(u) u^{-\left(\frac{s-3}{2}\right)} u^{-2} du \\ &= \int_1^\infty \psi(u) u^{\frac{3-s}{2}-2} du \\ &= \int_1^\infty \psi(u) u^{-\frac{1+s}{2}} du\end{aligned}$$

and we finally get (replacing u in the integral above by x)

$$\Gamma\left(\frac{s}{2}\right) \pi^{-\frac{s}{2}} \zeta(s) = \frac{1}{s(s-1)} + \int_1^\infty \psi(x) \left(x^{\frac{s}{2}-1} + x^{-\frac{1+s}{2}}\right) dx$$

Now, Riemann sets $s = \frac{1}{2} + ti$ and

$$\Gamma\left(\frac{s}{2}\right) \pi^{-\frac{s}{2}} \zeta(s)(s-1) = \xi(t) \tag{15.1.8}$$

so that the line $\Re(s) = \frac{1}{2} \subset \mathbb{C}$ gets transformed to the real axis and Maxima easily computes this last transformation.

```
z:x^(s/2-1)+x^(-(1+s)/2);
```

and

```
subst(1/2+t*%i,s,z);
```

gives

$$x^{\frac{\%i \cdot t + \frac{1}{2}}{2} - 1} + x^{\frac{-\%i \cdot t - \frac{3}{2}}{2}}$$

Now, we type

```
assume(x>0);
```

and

```
rectform(z);
```

gives

$$\frac{2\cos\left(\frac{t\log(x)}{2}\right)}{x^{\frac{3}{4}}}$$

We ultimately get

$$\xi(t) = \frac{1}{2} - \left(t^2 + \frac{1}{4}\right) \int_1^\infty \psi(x) x^{-\frac{3}{4}} \cos\left(\frac{t}{2}\log(x)\right) dx \tag{15.1.9}$$

Now we type

```
integrate((t^2+1/4)*x^(-3/4)*cos((t/2)*log(x)),x);
```

to get

$$\frac{4\left(t^2 + \frac{1}{4}\right) x^{\frac{1}{4}} \left(2t\sin\left(\frac{t\log(x)}{2}\right) + \cos\left(\frac{t\log(x)}{2}\right)\right)}{4t^2+1}$$

and

```
ratsimp(%)
```

gives

$$x^{\frac{1}{4}}\left(2t\sin\left(\frac{t\log(x)}{2}\right)+\cos\left(\frac{t\log(x)}{2}\right)\right)$$

which means we can integrate the right side of equation 15.1.9 on the facing page by parts to get

$$\xi(t)=\frac{1}{2}+\int_1^\infty \psi'(x)x^{\frac{1}{4}}\left(2t\sin\left(\frac{t\log(x)}{2}\right)+\cos\left(\frac{t\log(x)}{2}\right)\right)dx + \psi(1)$$

Now we write $x^{1/4}=x^{3/2}\cdot x^{-5/4}$ and integrate by parts a second time.

```
integrate(x^(-5/4)*(2*t*sin((t*log(x))/2)
+cos((t*log(x))/2)),x);
```

gives

$$4\left(\frac{2t\sin\left(\frac{t\log(x)}{2}\right)-\cos\left(\frac{t\log(x)}{2}\right)}{(4t^2+1)\,x^{\frac{1}{4}}}+\frac{2t\left(-\sin\left(\frac{t\log(x)}{2}\right)-2t\cos\left(\frac{t\log(x)}{2}\right)\right)}{(4t^2+1)\,x^{\frac{1}{4}}}\right)$$

and

```
expand(%)
```

gives

$$-\frac{16t^2\cos\left(\frac{t\log(x)}{2}\right)}{4t^2x^{\frac{1}{4}}+x^{\frac{1}{4}}}-\frac{4\cos\left(\frac{t\log(x)}{2}\right)}{4t^2x^{\frac{1}{4}}+x^{\frac{1}{4}}}$$

A final

```
ratsimp(%)
```

gives

$$-\frac{4\cos\left(\frac{t\log(x)}{2}\right)}{x^{\frac{1}{4}}}$$

So our second integration by parts gives

$$\xi(t)=\frac{1}{2}+\psi(1)+\psi'(1)+4\int_1^\infty \frac{d}{dx}\left(x^{\frac{3}{2}}\psi'(x)\right)x^{-\frac{1}{4}}\cos(t\log(x)/2)dx$$

Differentiating equation 15.1.7 on page 273 by x (and plugging in $x=1$) gives

$$\frac{1}{2}+\psi(1)+\psi'(1)=0$$

so

$$\xi(t)=4\int_1^\infty \frac{d}{dx}\left(x^{\frac{3}{2}}\psi'(x)\right)x^{-\frac{1}{4}}\cos(t\log(x)/2)dx \tag{15.1.10}$$

It's interesting that, in his paper [51], Riemann goes from equation 15.1.9 on page 274 to equation 15.1.10 on the previous page in a *single step*[2].

This is an *entire* function of t (i.e., no singularities) and Riemann conjectured that all of its zeroes lie on the real line. This is equivalent to saying all of the zeroes of $\zeta(z)$ that aren't of the form $-2n$ for $n \in \mathbb{Z}^+$ lie on the line $\Re(z) = \frac{1}{2}$ — which is the famous *Riemann hypothesis*. No one has found a counterexample to it or succeeded in proving it. Experiments have verified it up to $t < 12,363,153,437,138$.

Since this is an entire function, Riemann conjectured that it can be written as an "infinite polynomial"

$$\xi(t) = \xi(0)\prod_{i=1}^{\infty}\left(1 - \frac{t}{\rho_i}\right) \tag{15.1.11}$$

where the ρ_i run over all the zeroes of $\xi(t)$. Although this formula wasn't rigorous at the time, Weierstrass later proved that such infinite polynomials could exist and equal entire functions — under the right conditions. Later, Hadamard proved that the right conditions exist in this case, so Riemann's formula is correct.

Since $\xi(t)$ is actually a function of t^2, every zero, ρ, has a complementary one, $-\rho$, we usually write formula 15.1.11 as

$$\xi(t) = \xi(0)\prod_{i=1}^{\infty}\left(1 - \frac{t}{\rho_i}\right)\left(1 - \frac{t}{-\rho_i}\right) = \xi(0)\prod_{i=1}^{\infty}\left(1 - \frac{t^2}{\rho_i^2}\right) \tag{15.1.12}$$

15.2. A "formula" for prime numbers

Recall Euler's original formula for the Zeta function, equation 15.1.1 on page 269 for $x > 1$, which he immediately rewrote as

$$\zeta(x) = \prod_{p \text{ prime}} \frac{1}{1 - p^{-x}} \tag{15.2.1}$$

The way to see this is to recall the infinite series

$$\begin{aligned}\frac{1}{1 - p^{-x}} &= 1 + p^{-x} + p^{-2x} + \cdots \\ &= 1 + p^{-x} + (p^2)^{-x} \cdots\end{aligned}$$

The product

$$\prod_{p \text{ prime}} \left(1 + p^{-x} + (p^2)^{-x} \cdots\right)$$

will be a sum of terms of the form

$$\frac{1}{\left(p_1^{n_1} \cdots p_k^{n_k}\right)^x}$$

[2]Isn't it *obvious?*

with all possible primes raised to all possible powers, i.e. the series in equation 15.1.1 on page 269.

If we take the logarithm of equation 15.2.1 on the preceding page, we get

$$\log \zeta(x) = -\sum_{p \text{ prime}} \log(1 - p^{-x})$$

and plug in the Taylor series[3] for $\log(1 - p^{-x})$ to get

$$\log \zeta(x) = \sum_{p \text{ prime}} p^{-x} + \frac{p^{-2x}}{2} + \frac{p^{-3x}}{3} \cdots$$

Now we define a function that *counts* primes

$$\pi(x) = \begin{cases} \text{number of primes} < x & \text{if } x \text{ is not a prime} \\ 1/2 + \text{number of primes} < x & \text{if } x \text{ is a prime} \end{cases} \tag{15.2.2}$$

This is a discontinuous function whose value at each jump-point is the *mean* of the value before the jump and the one after.

Now define

$$R(x) = \pi(x) + \frac{\pi(x^{1/2})}{2} + \frac{\pi(x^{1/3})}{3} + \cdots \tag{15.2.3}$$

Note that

$$p^{-x} = x \int_p^\infty s^{-x-1} ds$$

$$p^{-2x} = x \int_{p^2}^\infty s^{-x-1} ds$$

etc.

so we conclude that

$$\frac{\log \zeta(x)}{x} = \int_1^\infty R(s) s^{-x-1} ds$$

After several steps using Fourier transforms, Riemann got

$$R(x) = \frac{1}{2\pi i} \int_{a-i\infty}^{a+i\infty} \frac{\log \zeta(s)}{s} x^s ds$$

Now we use formulas from the preceding section to estimate $\log \zeta(s)/s$, namely equations 15.1.8 on page 274 and 15.1.12 on the preceding page.

$$\log \zeta(s) = \log \xi(s) - \log \Gamma\left(\frac{s}{2}\right) + \frac{s}{2} \log(\pi) - \log(s-1) \tag{15.2.4}$$

If we try to integrate this in a straightforward fashion, several terms in the integral diverge (particularly the term $\frac{s}{2}\log(\pi)$). We choose to integrate by parts: assuming that x^s was a derivative of something with respect to s. Note that

[3] For instance, type **taylor(log**(1-s),s,0,20);

```
integrate(x^s,s);
```

produces

$$\frac{x^s}{\log(x)}$$

Since

$$\lim_{T\to\pm\infty}\frac{\log\zeta(a+iT)}{(a+iT)}x^{a+iT}=0$$

since x^{a+iT} oscillates as $T\to\infty$ and $\log\zeta(a+iT)$ remains bounded, so the denominator kills off the fraction.

It follows that integrating by parts gives

$$R(x)=-\frac{1}{2\pi i\log x}\int_{a-i\infty}^{a+i\infty}\frac{d}{ds}\left(\frac{\log\zeta(s)}{s}\right)x^s ds \tag{15.2.5}$$

We will start by computing

$$-\frac{1}{2\pi i\log x}\int_{a-i\infty}^{a+i\infty}\frac{d}{ds}\left(\frac{\log(s-1)}{s}\right)x^s ds$$

since it is probably the largest term. We compute

$$G(\beta)=-\frac{1}{2\pi i\log x}\int_{a-i\infty}^{a+i\infty}\frac{d}{ds}\left(\frac{\log(s/\beta-1)}{s}\right)x^s ds$$

and consider the limit as $\beta\to 1$. This integral is well-defined if $a>\Re(\beta)$ because

$$\begin{aligned}\left|\frac{d}{ds}\left(\frac{\log(s/\beta-1)}{s}\right)\right|&=\left|\frac{1}{s\,(s-\beta)}-\frac{\log(s-\beta)}{s^2}\right|\\&\le\frac{1}{|s\,(s-\beta)\,|}+\frac{|\log(s-\beta)|}{|s|^2}\end{aligned}$$

which are integrable. We can differentiate $G(\beta)$ with respect to β:

$$G'(\beta)=\frac{1}{2\pi i\log x}\int_{a-i\infty}^{a+i\infty}\frac{d}{ds}\left(\frac{1}{\beta\,(s-\beta)}\right)x^s ds$$

Now integration by parts gives

$$G'(\beta)=-\frac{1}{2\pi i}\int_{a-i\infty}^{a+i\infty}\frac{1}{\beta\,(s-\beta)}x^s ds$$

This resists brute-force efforts to integrate it (i.e., plug in $s=a+iT$, and integrate with T running from $-\infty$ to ∞). We use a bit of *finesse:* Regard the path from $a-i\infty$ to $a+i\infty$ as a *circle* on the Riemann sphere. It encloses the part of $\mathbb{C}$ to the left of it, i.e. the half plane $\Re(z)<a$. The integral is determined by its *residue* at the singularity $s=\beta$ (see [1] for information on the Residue Theorem). We get

$$G'(\beta)=\begin{cases}\frac{x^\beta}{\beta}=\int_0^x u^{\beta-1}du & \text{if } a>\Re(\beta)\\ 0 & \text{if } a<\Re(\beta)\end{cases}$$

So

$$\begin{aligned} G(\beta) &= \int \left\{ \int_0^x u^{\beta-1} du \right\} d\beta \\ &= \int_0^x \left\{ \int u^{\beta-1} d\beta \right\} du \\ &= \int_0^x \left\{ \frac{u^{\beta-1}}{\log u} \right\} du \qquad \text{now set } v = u^\beta \\ &= \frac{1}{\beta} \int_0^{x^\beta} \frac{dv}{\log v^{1/\beta}} \\ &= \int_0^{x^\beta} \frac{dv}{\log v} \end{aligned}$$

We get[4]

$$F(x) = \mathrm{li}(x^\beta) + \text{constant}(x)$$

Riemann then argues that this constant is identically 0. If $\beta = 1$, we get the statement of the Prime Number Theorem

$$\pi(x) \sim \mathrm{li}(x)$$

Now we can introduce the remaining terms of equation 15.2.4 on page 277 into equation 15.2.5 to get

$$R(x) = \mathrm{li}(x) - \sum_{\xi(\rho)=0} \left(\mathrm{li}\left(x^{\frac{1}{2}+i\rho}\right) + \mathrm{li}\left(x^{\frac{1}{2}-i\rho}\right) \right) + \int_x^\infty \frac{du}{(u^2-1)u\log u} + \log \xi(0)$$

We can plot this function, using table on the next page of the first few values of ρ. The following function uses this table:

```
zeros: matrix(
[14.1347250,21.022039,25.010857,30.424876,
32.935061,37.586178,40.918719,43.327073,
48.005150,49.773832]
);

R(x):=block ([accum:expintegral_li(x)],
    for i :1 step 1 thru 10 do (
        accum:accum-
          expand(float(expintegral_ei(
              log(x)*(1/2+%i*zeros[1,i])))
          +float(expintegral_ei(
              log(x)*(1/2-%i*zeros[1,i]))))
        ),
```

[4] If we integrate it the other way, i.e. as the integral of n^beta/beta, we get -**gamma_incomplete**(0,-beta*__log__(x)), which *equals* $\mathrm{li}(x^\beta)$.

14.134725
21.022039
25.010857
30.424876
32.935061
37.586178
40.918719
43.327073
48.005150
49.773832

TABLE 15.2.1. First few zeros of $\xi(t)$

```
        realpart(accum)
    )
```

The command

```
plot2d(R(x),[x,2,100])
```

produces the plot in figure on the facing page. It is well-known that there are 25 primes < 100, so the plot is fairly accurate[5].

The reader will doubtless have several questions:

(1) why do we have **expintegral_ei**? This is because **expintegral_li** produces incorrect results for complex arguments, so we use equation 14.5.3 on page 262. It's also more efficient to compute

$$\log(x) \cdot (1/2 + i \cdot \text{zeros}_{1,i})$$

than

$$\log\left(x^{1/2+i\cdot\text{zeros}_{1,i}}\right)$$

(2) Why do we have **expand** and **float**? This is because Maxima wants to do exact symbolic computations. We have to force it to do numeric computations.

(3) $R(x)$, as defined above counts primes and powers of primes. How do we get a formula that only counts primes[6]? This is answered below.

Recall equations 15.2.2 on page 277 and 15.2.3 on page 277. The Möbius Inversion Theorem (see [38]) states that we can *invert* equation 15.2.3 on page 277 to get

$$\pi(x) = \sum_{k=1}^{\infty} \mu(k) \frac{R(x^{1/k})}{k} \tag{15.2.6}$$

[5]It's counting *powers* of primes as well as primes themselves.

[6]I.e., an *equation* for primes!

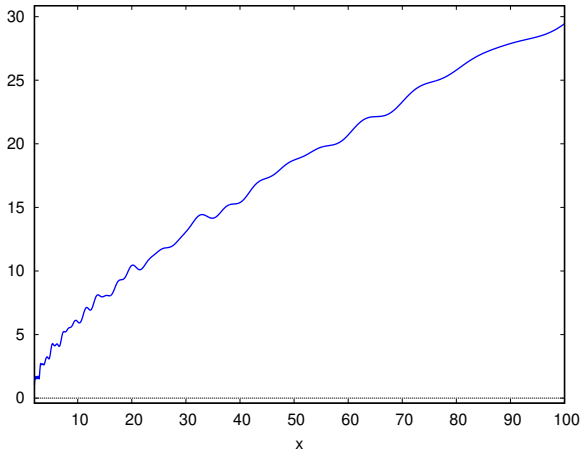


FIGURE 15.2.1. Plot of $R(x)$

where $\mu(k)$ is the *Möbius function*, defined by

$$\mu(k) = \begin{cases} 0 & \text{if } k \text{ is divisible by the square of any prime} \\ (-1)^\ell & \text{if } k = p_1 \cdots p_\ell \text{ where they are all distinct primes} \end{cases}$$

Maxima implements the Möbius function via the command

```
moebius(x)
```

August Ferdinand Möbius (1790–1868) was a German mathematician and astronomer popularly known for his discovery of the Möbius strip (although he made many other contributions to mathematics, including Möbius transformations, the Möbius function in combinatorics and the Möbius inversion formula).

In our case we can define

```
pi(x):=R(x)-R(x^(1/2))/2-R(x^(1/3))/3
       -R(x^(1/5))/5+R(x^(1/6))/6
```

and (see section F.5 on page 336) the command

```
draw(
    gr2d(line_width=2,color=black,
        explicit(
            pi(x),
            x,2,20
                ),line_type=dots,
        parametric(3,t,t,0,8),
        parametric(5,t,t,0,8),
```

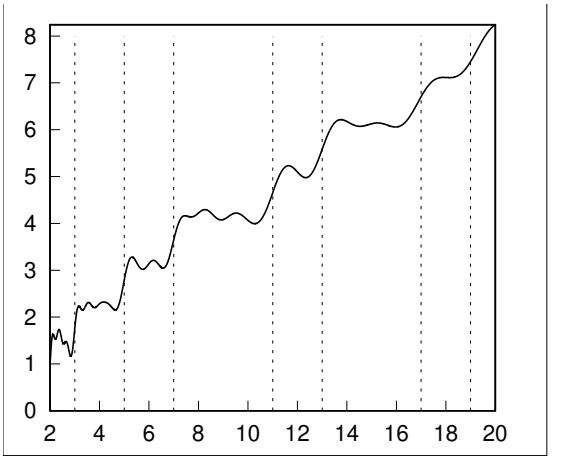


FIGURE 15.2.2. Approximate $\pi(x)$

```
        parametric(7,t,t,0,8),
        parametric(11,t,t,0,8),
        parametric(13,t,t,0,8),
        parametric(17,t,t,0,8),
        parametric(19,t,t,0,8)
                )
        );
```

produces figure 15.2.2, which jumps every time it passes a prime.

We can also plot this to 100 to get figure 15.2.3 on the next page, which correctly estimates the number of primes < 100 as 25.

EXERCISES.

1. For a given value of x, show that equation 15.2.3 on page 277 and equation 15.2.6 on page 280 are finite sums.

2. Verify the Möbius Inversion Theorem by plugging equation 15.2.3 on page 277 into equation 15.2.6 on page 280.

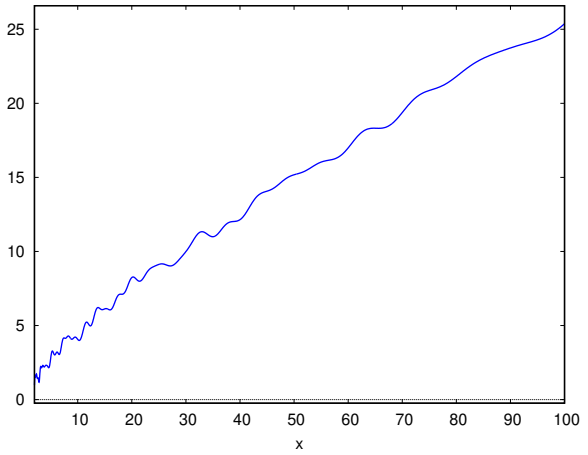


FIGURE 15.2.3. First 100 primes (approximately)

APPENDIX A

Gröbner basis for the robotic motion problem

(1) 3*w^2−1
(2) x−y
(3) 576*b_2^4*y^4+576*b_2^4*y^2*z^2+144*b_2^4*z^4+192*b_2^2*y^6+768*b_2^2*y^5*z+96*b_2^2*y^4*z^2−768*b_2^2*y^4+768*b_2^2*y^3*z^3−768*b_2^2*y^3*z−48*b_2^2*y^2*z^4−576*b_2^2*y^2*z^2+192*b_2^2*y*z^5−384*b_2^2*y*z^3−24*b_2^2*z^6−96*b_2^2*z^4+144*y^8+288*y^6*z^2−384*y^6−384*y^5*z+216*y^4*z^4−480*y^4*z^2+256*y^4−384*y^3*z^3+512*y^3*z+72*y^2*z^6−192*y^2*z^4+384*y^2*z^2−96*y*z^5+128*y*z^3+9*z^8−24*z^6+16*z^4
(4) 48*b_3*y^5+24*b_3*y^4*z+48*b_3*y^3*z^2−64*b_3*y^3+24*b_3*y^2*z^3−96*b_3*y^2*z+12*b_3*y*z^4−48*b_3*y*z^2+6*b_3*z^5−8*b_3*z^3+144*b_2^3*y^4*w+144*b_2^3*y^2*z^2*w+36*b_2^3*z^4*w+120*b_2*y^6*w+192*b_2*y^5*z*w+132*b_2*y^4*z^2*w−288*b_2*y^4*w+192*b_2*y^3*z^3*w−288*b_2*y^3*z*w+42*b_2*y^2*z^4*w−216*b_2*y^2*z^2*w+48*b_2*y*z^5*w−144*b_2*y*z^3*w+3*b_2*z^6*w−36*b_2*z^4*w
(5) 16*b_3*b_2*y+8*b_3*b_2*z+24*b_2^2*y^2*w+12*b_2^2*z^2*w−12*y^4*w−12*y^2*z^2*w+16*y^2*w+16*y*z*w−3*z^4*w+4*z^2*w
(6) 2*b_3*b_2^2*z^2−48*b_3*y^4−48*b_3*y^2*z^2+64*b_3*y^2+64*b_3*y*z−12*b_3*z^4+16*b_3*z^2−144*b_2^3*y^3*w+72*b_2^3*y^2*z*w−72*b_2^3*y*z^2*w+36*b_2^3*z^3*w−120*b_2*y^5*w−132*b_2*y^4*z*w−120*b_2*y^3*z^2*w+288*b_2*y^3*w−132*b_2*y^2*z^3*w+144*b_2*y^2*z*w−30*b_2*y*z^4*w+216*b_2*y*z^2*w−33*b_2*z^5*w+108*b_2*z^3*w
(7) 4*b_3^2+4*y^4+4*y^2*z^2−8*y^2+z^4−4*z^2

(8) 576*b_4*y*z^2−576*b_4*z^3+2880*b_3*b_2^3*z*w−1080*b_3*b_2*z^3*w−1152*b_3*b_2*z*w+576*b_2^4*y^2+288*b_2^4*z^2+192*b_2^2*y^4−432*b_2^2*y^3*z−1320*b_2^2*y^2*z^2−768*b_2^2*y^2−216*b_2^2*y*z^3+1536*b_2^2*y*z−708*b_2^2*z^4+672*b_2^2*z^2+144*y^6−360*y^5*z+396*y^4*z^2−384*y^4−360*y^3*z^3+864*y^3*z+288*y^2*z^4−96*y^2*z^2+256*y^2−90*y*z^5−504*y*z^3−512*y*z+63*z^6+444*z^4−320*z^2

(9) 64*b_4*y^2+64*b_4*y*z−128*b_4*z^2+576*b_3*b_2^3*w−216*b_3*b_2*z^2*w−192*b_3*b_2*w−240*b_2^2*y^3−264*b_2^2*y^2*z−120*b_2^2*y*z^2+384*b_2^2*y−132*b_2^2*z^3+192*b_2^2*z−72*y^5+36*y^4*z−72*y^3*z^2+224*y^3+36*y^2*z^3−48*y^2*z−18*y*z^4−24*y*z^2−128*y+9*z^5+100*z^3−64*z

(10) 2592*b_4*b_2^2*z^3+1944*b_4*z^5−2592*b_4*z^3−6912*b_3*b_2^5*z*w+12096*b_3*b_2^3*z*w+972*b_3*b_2*z^5*w−648*b_3*b_2*z^3*w−3456*b_3*b_2*z*w+1728*b_2^6*y^2+864*b_2^6*z^2+3456*b_2^4*y^3*z+4752*b_2^4*y^2*z^2−2304*b_2^4*y^2+1728*b_2^4*y*z^3−5760*b_2^4*y*z+2376*b_2^4*z^4−3168*b_2^4*z^2+240*b_2^2*y^6−480*b_2^2*y^5*z−1464*b_2^2*y^4*z^2−384*b_2^2*y^4+4056*b_2^2*y^3*z^3−4272*b_2^2*y^3*z−24*b_2^2*y^2*z^4−2664*b_2^2*y^2*z^2+768*b_2^2*y^2+2148*b_2^2*y*z^5−4728*b_2^2*y*z^3+8832*b_2^2*y*z+384*b_2^2*z^6−4476*b_2^2*z^4+4224*b_2^2*z^2−144*y^8−432*y^7*z+252*y^6*z^2+384*y^6−324*y^5*z^3+888*y^5*z+432*y^4*z^4+1164*y^4*z^2−256*y^4−1704*y^3*z^3+1312*y^3*z+171*y^2*z^6+1200*y^2*z^4−2256*y^2*z^2+27*y*z^7−750*y*z^5+1384*y*z^3−2304*y*z+18*z^8−1425*z^6+2732*z^4−1152*z^2

(11) 4608*b_4*b_2^2*y*z−11520*b_4*b_2^2*z^2−5184*b_4*z
^4+6912*b_4*z^2+41472*b_3*b_2^5*w−41472*b_3*b_2^3*w
−5832*b_3*b_2*z^4*w+1728*b_3*b_2*z^2*w+9216*b_3*b_2
*w−18432*b_2^4*y^3−21888*b_2^4*y^2*z−9216*b_2^4*y
*z^2+27648*b_2^4*y−10944*b_2^4*z^3+13824*b_2^4*z
−5568*b_2^2*y^5+96*b_2^2*y^4*z−15504*b_2^2*y^3*z
^2+28032*b_2^2*y^3−5304*b_2^2*y^2*z^3+11712*b_2^2*y
^2*z−6360*b_2^2*y*z^4+16320*b_2^2*y*z^2−27648*b_2^2*
y−2676*b_2^2*z^5+19104*b_2^2*z^3−13824*b_2^2*z−288*y
^7−720*y^6*z−2376*y^5*z^2+2496*y^5−108*y^4*z^3+96*
y^4*z−2160*y^3*z^4+9408*y^3*z^2−7424*y^3+432*y^2*z
^5−768*y^2*z^3+1152*y^2*z−522*y*z^6+2136*y*z^4−5952*
y*z^2+6144*y+153*z^7+3804*z^5−7648*z^3+3072*z

(12) 144*b_4*b_3*z^2+432*b_4*b_2*z^3*w−384*b_3*b_2^4*z
+192*b_3*b_2^2*z+96*b_3*y^4*z−96*b_3*y^3+96*b_3*y^2*
z^3−80*b_3*y^2*z−176*b_3*y*z^2+128*b_3*y+24*b_3*z
^5−152*b_3*z^3+64*b_3*z+288*b_2^5*y^2*w+144*b_2^5*z
^2*w+96*b_2^3*y^4*w+1152*b_2^3*y^3*z*w+384*b_2^3*y
^2*z^2*w−672*b_2^3*y^2*w+576*b_2^3*y*z^3*w−960*b_2
^3*y*z*w+168*b_2^3*z^4*w−672*b_2^3*z^2*w+72*b_2*y
^6*w+384*b_2*y^5*z*w+300*b_2*y^4*z^2*w−432*b_2*y^4*
w+384*b_2*y^3*z^3*w−1440*b_2*y^3*z*w+246*b_2*y^2*z
^4*w−504*b_2*y^2*z^2*w+704*b_2*y^2*w+96*b_2*y*z^5*w
−720*b_2*y*z^3*w+704*b_2*y*z*w+57*b_2*z^6*w−576*b_2*
z^4*w+464*b_2*z^2*w

(13) 16*b_4*b_3*y+8*b_4*b_3*z−48*b_4*b_2*y*z*w+120*b_4*b_2
*z^2*w−144*b_3*b_2^4+48*b_3*b_2^2+36*b_3*y^4+36*
b_3*y^2*z^2−40*b_3*y^2−64*b_3*y*z+9*b_3*z^4−16*b_3
*z^2+288*b_2^3*y^3*w+144*b_2^3*y^2*z*w+144*b_2^3*
y*z^2*w−288*b_2^3*y*w+72*b_2^3*z^3*w−144*b_2^3*z*
w+144*b_2*y^5*w+72*b_2*y^4*z*w+144*b_2*y^3*z^2*w
−384*b_2*y^3*w+72*b_2*y^2*z^3*w−96*b_2*y^2*z*w+36*
b_2*y*z^4*w−144*b_2*y*z^2*w+96*b_2*y*w+18*b_2*z^5*w
−168*b_2*z^3*w+48*b_2*z*w

(14) 192*b_4*b_3*b_2+384*b_4*y*w−432*b_4*z^3*w+192*b_4*z*w+384*b_3*b_2^3*z−144*b_3*b_2*z^3−288*b_3*b_2*z−288*b_2^4*y^2*w−144*b_2^4*z^2*w−96*b_2^2*y^4*w−864*b_2^2*y^3*z*w−528*b_2^2*y^2*z^2*w+96*b_2^2*y^2*w−432*b_2^2*y*z^3*w+960*b_2^2*y*z*w−240*b_2^2*z^4*w+384*b_2^2*z^2*w−72*y^6*w−144*y^5*z*w−36*y^4*z^2*w+48*y^4*w−144*y^3*z^3*w+576*y^3*z*w+18*y^2*z^4*w−24*y^2*z^2*w+256*y^2*w−36*y*z^5*w+144*y*z^3*w−704*y*z*w+9*z^6*w+336*z^4*w−272*z^2*w

(15) 288*b_4^2*z−192*b_4*y*z−96*b_4*z^2−1728*b_3*b_2^3*w+648*b_3*b_2*z^2*w+384*b_3*b_2*w+720*b_2^2*y^3+792*b_2^2*y^2*z+360*b_2^2*y*z^2−1536*b_2^2*y+396*b_2^2*z^3−672*b_2^2*z+216*y^5−108*y^4*z+216*y^3*z^2−672*y^3−108*y^2*z^3−48*y^2*z+54*y*z^4+72*y*z^2+512*y−27*z^5−108*z^3+160*z

(16) 288*b_4^2*y−768*b_4*y*z+480*b_4*z^2−1728*b_3*b_2^3*w+648*b_3*b_2*z^2*w+960*b_3*b_2*w+720*b_2^2*y^3+792*b_2^2*y^2*z+360*b_2^2*y*z^2−1248*b_2^2*y+396*b_2^2*z^3−384*b_2^2*z+216*y^5−108*y^4*z+216*y^3*z^2−672*y^3−108*y^2*z^3−48*y^2*z+54*y*z^4+360*y*z^2+416*y−27*z^5−396*z^3+256*z

(17) 24*b_4^2*b_3−48*b_4*b_3*z+48*b_4*b_2*y*z*w−120*b_4*b_2*z^2*w+144*b_3*b_2^4−72*b_3*b_2^2−36*b_3*y^4−36*b_3*y^2*z^2+48*b_3*y^2+48*b_3*y*z−9*b_3*z^4+36*b_3*z^2−8*b_3−288*b_2^3*y^3*w−144*b_2^3*y^2*z*w−144*b_2^3*y*z^2*w+288*b_2^3*y*w−72*b_2^3*z^3*w+144*b_2^3*z*w−144*b_2*y^5*w−72*b_2*y^4*z*w−144*b_2*y^3*z^2*w+384*b_2*y^3*w−72*b_2*y^2*z^3*w+120*b_2*y^2*z*w−36*b_2*y*z^4*w+144*b_2*y*z^2*w−96*b_2*y*w−18*b_2*z^5*w+180*b_2*z^3*w−96*b_2*z*w

(18) 18*b_4^4−12*b_4^2*b_2^2−12*b_4^2−96*b_4*b_2^2*y+64*b_4*y−72*b_4*z^3+32*b_4*z−72*b_3*b_2*z*w+18*b_2^4−24*b_2^2*y^2−48*b_2^2*y*z−36*b_2^2*z^2−12*b_2^2−36*y^4−36*y^2*z^2+80*y^2−16*y*z+45*z^4−28*z^2+2

(19) 108*b_5*z^4−144*b_5*z^2+144*b_4*b_3*z−432*b_4*b_2*y*z*w+1080*b_4*b_2*z^2*w−1296*b_3*b_2^4+864*b_3*b_2^2+204*b_3*y^4−36*b_3*y^3*z+276*b_3*y^2*z^2−272*b_3*y^2−18*b_3*y*z^3−152*b_3*y*z+87*b_3*z^4−188*b_3*z^2+2232*b_2^3*y^3*w+1368*b_2^3*y^2*z*w+1116*b_2^3*y*z^2*w−2592*b_2^3*y*w+684*b_2^3*z^3*w−1296*b_2^3*z*w+996*b_2*y^5*w+228*b_2*y^4*z*w+1212*b_2*y^3*z^2*w−3168*b_2*y^3*w+444*b_2*y^2*z^3*w−504*b_2*y^2*z*w+357*b_2*y*z^4*w−1260*b_2*y*z^2*w+1728*b_2*y*w+165*b_2*z^5*w−1440*b_2*z^3*w+864*b_2*z*w

(20) 96*b_5*y−72*b_5*z^3+48*b_4*b_3+120*b_3*b_2^2*z+24*b_3*y^3−48*b_3*y^2*z+12*b_3*y*z^2−80*b_3*y−24*b_3*z^3−64*b_3*z+72*b_2^3*y^2*w+36*b_2^3*z^2*w+60*b_2*y^4*w−144*b_2*y^3*z*w−84*b_2*y^2*z^2*w−288*b_2*y^2*w−72*b_2*y*z^3*w+192*b_2*y*z*w−57*b_2*z^4*w−12*b_2*z^2*w

(21) 64*b_5*b_2*z^2+128*b_4*y*z*w−224*b_4*z^2*w+192*b_3*b_2^3−72*b_3*b_2*z^2−128*b_3*b_2−240*b_2^2*y^3*w−264*b_2^2*y^2*z*w−120*b_2^2*y*z^2*w+384*b_2^2*y*w−132*b_2^2*z^3*w+192*b_2^2*z*w−72*y^5*w+36*y^4*z*w−72*y^3*z^2*w+288*y^3*w+36*y^2*z^3*w+16*y^2*z*w−18*y*z^4*w−56*y*z^2*w−256*y*w+9*z^5*w+196*z^3*w−128*z*w

(22) 288*b_5*b_2^2*z+72*b_5*z^3−96*b_5*z+96*b_4*b_3+288*b_4*b_2*y*w−288*b_4*b_2*z*w−120*b_3*b_2^2*z−24*b_3*y^3+48*b_3*y^2*z−12*b_3*y*z^2+80*b_3*y+24*b_3*z^3−80*b_3*z−72*b_2^3*y^2*w−36*b_2^3*z^2*w−60*b_2*y^4*w+144*b_2*y^3*z*w+84*b_2*y^2*z^2*w+144*b_2*y^2*w+72*b_2*y*z^3*w−480*b_2*y*z*w+57*b_2*z^4*w+228*b_2*z^2*w

(23) 384*b_5*b_2^3−128*b_5*b_2+288*b_4^3*w−480*b_4*b_2^2*w−576*b_4*y*z*w+576*b_4*z^2*w−96*b_4*w−1728*b_3*b_2^3+648*b_3*b_2*z^2+384*b_3*b_2+2160*b_2^2*y^3*w+2376*b_2^2*y^2*z*w+1080*b_2^2*y*z^2*w−4608*b_2^2*y*w+1188*b_2^2*z^3*w−1536*b_2^2*z*w+648*y^5*w−324*y^4*z*w+648*y^3*z^2*w−2016*y^3*w−324*y^2*z^3*w−144*y^2*z*w+162*y*z^4*w+216*y*z^2*w+1536*y*w−81*z^5*w−612*z^3*w+576*z*w

(24) b_5*b_3+3*b_5*b_2*z*w+b_4*y−b_4*z−y*z+z^2

(25) 96*b_5*b_4*z−48*b_5*z^2−24*b_4*b_3*z+48*b_4*b_2*y*z*w
−120*b_4*b_2*z^2*w+144*b_3*b_2^4−144*b_3*b_2^2−36*
b_3*y^4−36*b_3*y^2*z^2+72*b_3*y^2+48*b_3*y*z−9*b_3*z
^4+48*b_3*z^2−32*b_3−288*b_2^3*y^3*w−144*b_2^3*y^2*z
*w−144*b_2^3*y*z^2*w+288*b_2^3*y*w−72*b_2^3*z^3*w
+144*b_2^3*z*w−144*b_2*y^5*w−72*b_2*y^4*z*w−144*b_2*
y^3*z^2*w+528*b_2*y^3*w−72*b_2*y^2*z^3*w+192*b_2*y
^2*z*w−36*b_2*y*z^4*w+216*b_2*y*z^2*w−384*b_2*y*w
−18*b_2*z^5*w+216*b_2*z^3*w−240*b_2*z*w

(26) 4*b_5*b_4*b_2−4*b_5*b_2*z−3*b_4^2*w+6*b_4*z*w−3*b_2
^2*w−3*z^2*w+w

(27) 48*b_5*b_4^2+48*b_5*b_2^2−16*b_5−24*b_4*b_3*z+48*b_4*
b_2*y*z*w−120*b_4*b_2*z^2*w−96*b_4*b_2*w+144*b_3*b_2
^4−144*b_3*b_2^2−36*b_3*y^4−36*b_3*y^2*z^2+72*b_3*y
^2+48*b_3*y*z−9*b_3*z^4+48*b_3*z^2−32*b_3−288*b_2^3*y
^3*w−144*b_2^3*y^2*z*w−144*b_2^3*y*z^2*w+288*b_2^3*
y*w−72*b_2^3*z^3*w+144*b_2^3*z*w−144*b_2*y^5*w−72*
b_2*y^4*z*w−144*b_2*y^3*z^2*w+528*b_2*y^3*w−72*b_2*
y^2*z^3*w+192*b_2*y^2*z*w−36*b_2*y*z^4*w+216*b_2*y*z
^2*w−384*b_2*y*w−18*b_2*z^5*w+216*b_2*z^3*w−144*b_2
*z*w

(28) 2*b_5^2−1

(29) a_2−2*b_5*b_2+3*b_4*w−3*z*w

(30) 2*a_3−2*y^2−z^2+2

(31) 48*a_4−96*b_5*b_4−72*b_5*z^3+96*b_5*z+48*b_4*b_3+120*
b_3*b_2^2*z+24*b_3*y^3−48*b_3*y^2*z+12*b_3*y*z^2−80*
b_3*y−24*b_3*z^3−64*b_3*z+72*b_2^3*y^2*w+36*b_2^3*z
^2*w+60*b_2*y^4*w−144*b_2*y^3*z*w−84*b_2*y^2*z^2*w
−288*b_2*y^2*w−72*b_2*y*z^3*w+192*b_2*y*z*w−57*b_2*z
^4*w−12*b_2*z^2*w+144*b_2*w

(32) a_5−b_5

APPENDIX B

Predefined values.

Constant	Maxima name	Approximate value
Last result	%	
π	%pi	3.14159265358979
e	%e	2.71828182845905
γ	%gamma	0.577215664901533
φ	%phi	1.61803398874989
$\sqrt{-1}$	%i	
$+\infty$	inf	
$-\infty$	minf	
F	false	
T	true	
0^+	zeroa	
0^-	zerob	
__ (two underscores)		Last expressions being evaluated

APPENDIX C

Functional equation

C.1. Poisson summation

In this section, we will prove the functional equation of the ψ-function used in section 15 on page 269.

Baron Siméon Denis Poisson FRS FRSE (1781 – 1840) was a French mathematician and physicist who worked on statistics, complex analysis, partial differential equations, the calculus of variations, analytical mechanics, electricity and magnetism, thermodynamics, elasticity, and fluid mechanics.

THEOREM C.1.1 (Poisson Summation formula). *If $f(x)$ is a smooth function such that*

$$\int_{-\infty}^{\infty} |f(x)|dx$$

is well-defined and finite and

$$\hat{f}(u) = \int_{-\infty}^{\infty} f(x)e^{-2\pi ixu}dx$$

is its Fourier transform, then

$$\sum_{n>-\infty}^{\infty} f(n) = \sum_{k>-\infty}^{\infty} \hat{f}(k) \tag{C.1.1}$$

PROOF. Create a periodic function with period 1:

$$F(x) = \sum_{n>-\infty}^{\infty} f(x+n)$$

and expand it into a Fourier series

$$F(x) = \sum_{n>-\infty}^{\infty} a_n e^{2\pi inx} \tag{C.1.2}$$

where the Fourier coefficients are given by

$$\begin{aligned} a_n &= \int_{-\infty}^{\infty} F(x)e^{-2\pi inx}dx \\ &= \sum_{m>-\infty}^{\infty} \int_0^1 f(x+m)e^{-2\pi inx}dx \\ &= \sum_{m>-\infty}^{\infty} \int_m^{m+1} f(y)e^{-2\pi in(y-m)}dy \\ &\sum_{m>-\infty}^{\infty} e^{2\pi inm} \int_m^{m+1} f(y)e^{-2\pi iny}dy \\ &= \int_{-\infty}^{\infty} f(x)e^{-2\pi iny}dy = \hat{f}(-n) \end{aligned}$$

since $e^{2\pi inm} = 1$.

If we set $x = 0$ in equation C.1.2 on the previous page, we get equation C.1.1 on the preceding page. □

C.2. The main result

Suppose

$$G(x) = \sum_{n>-\infty}^{\infty} e^{-\pi n^2 x} = 1 + 2\psi(x)$$

We can regard this as the sum of values of a function

$$f(u) = e^{-\pi u^2 x}$$

and take its Fourier transform. Here, u is the independent variable, x is a parameter, and y is the new variable (introduced by the Fourier transform). Typing

```
integrate(%e^(-%pi*u^2*x)*%e^(2*%pi*%i*u*y),
                    u,minf,inf)
```

gives

$$\frac{e^{-\frac{\pi \cdot y^2}{x}}}{\sqrt{x}}$$

so

$$\sum_{n>-\infty}^{\infty} e^{-\pi n^2 x} = \frac{1}{\sqrt{x}} \sum_{n>-\infty}^{\infty} e^{-\pi n^2/x}$$

and

$$G(x) = \frac{1}{\sqrt{x}} G\left(\frac{1}{x}\right) \tag{C.2.1}$$

APPENDIX D

Fermat factorization

D.1. The algorithm

Suppose N is an odd integer greater than 1 and $N = u \cdot v$ where u and v are (odd) integers. Fermat factorization is based on the equation

$$N = \left(\frac{u+v}{2}\right)^2 - \left(\frac{u-v}{2}\right)^2$$

or

$$\left(\frac{u+v}{2}\right)^2 - N = \left(\frac{u-v}{2}\right)^2$$

Since u and v are both odd, their sum and differences are even, so dividing by 2 gives us integers.

Oddly enough, this gives us a way to find u or v if the difference between them isn't too great.

We compute

$$k^2 - N$$

for $k = \left\lceil \sqrt{N} \right\rceil, \left\lceil \sqrt{N} \right\rceil + 1, \left\lceil \sqrt{N} \right\rceil + 2, \ldots$ and test whether the difference is a perfect square. If it *is*, say m^2, then $u = k \pm m$ and we have found a factor of N.

Here $\lceil x \rceil$ is the **ceiling** function, the least integer that is $\geq x$. In Maxima, it is coded via

```
ceiling(x)
```

It is a complement to the **floor**-command

```
floor(x)
```

which returns the greatest integer $\leq x$.

In this example, we will use the **isqrt**-command which computes *integer* square roots of very large numbers — i.e., **isqrt**(n) computes the largest integer, k, such that $k^2 \leq n$.

EXAMPLE D.1.1. Let $u = 1000003$ and v be the next prime, which is 1000033. We compute

$$u \cdot v = 1000008000015$$

and

```
ceiling(float(sqrt(1000008000015)))
```

is

```
1000018
```

and

```
1000018^2-n;
```

gives

$$225 = 15^2$$

so we *immediately* get both factors of n: $1000018 - 15 = 1000003$ and $1000018 + 15 = 1000033$.

The "lucky accident" in this example (where the answer occurs on the *first iteration*) turns out to always occur if $\frac{1}{2q}\left(\frac{p-q}{2}\right)^2 < 1$, where p is the larger of the two primes — see section D.2 on the next page below.

We can write a Maxima program implementing this algorithm. First, we need a function to determine whether a number is a perfect square:

```
square_p(n):=block(
    [sqval:isqrt(n)],
    is(n=sqval^2)
);
```

Now for the main function:

```
one_factor(n):=block(
   [start:isqrt(n),test,val],
 for x:0 step 1 thru 100 do
  (
 /* Note: for-loops cannot handle very large numbers
    so we only iterate through the increments to
    ceiling(sqrt(n)) */
  val:x+start,
  test:val^2-n,
  if(square_p(test)) then
     return(val-isqrt(test))
/* If test is a perfect square, then
   isqrt is equal to its square root */
  )
);
```

For instance

```
one_factor(2251644881930449333);
```

returns

1500450271

after 3 iterations of the loop.

If

$n = 895649414291294604941588381871244924626104121562042227318384494381723497514540860474803494041479529$

p:one_factor(n) instantly comes back with

29927402397991286489627837734179186385188296382227

a prime factor of n, and n/p produces

29927402397991286489627904551843385490310576382227

the other prime factor.

D.2. Derivation of the upper bound for the number of iterations

Suppose $0 < q < p$ and $N = pq$. We have

$$N = \left(\frac{p+q}{2}\right)^2 - \left(\frac{p-q}{2}\right)^2$$

or

$$\left(\frac{p+q}{2}\right)^2 - N = \left(\frac{p-q}{2}\right)^2$$

$$\left(\frac{p+q}{2} - \sqrt{N}\right)\left(\frac{p+q}{2} + \sqrt{N}\right) = \left(\frac{p-q}{2}\right)^2$$

Note that

$$\left(\frac{p+q}{2} + \sqrt{N}\right) > 2q$$

so dividing by $2q$ will give a *larger* result than dividing by $\left(\frac{p+q}{2} + \sqrt{N}\right)$ (to get $\left(\frac{p-q}{2}\right)^2$). We get

$$0 < \frac{p+q}{2} - \sqrt{N} < \frac{1}{2q}\left(\frac{p-q}{2}\right)^2$$

and the quantity on the right is an approximate upper bound to the number of iterations required.

EXERCISES.

1. Factor 15241580725499173.

APPENDIX E

The Maxima Programming language

E.1. Introduction

Maxima implements a powerful programming language that is used for the functions in its libraries. We have already seen how to program a function and the **block**-construct. This is not an exhaustive treatment of the language by any means; it should be enough to understand the programs in the book.

E.2. Arithmetic commands

Aside from the familiar operations like '+', '*', '/', and '^', we have

- ▷ **abs**(x) — computes the absolute value of a real number.
- ▷ **cabs**(x) — computes the absolute value of a real or complex number.
- ▷ **carg**(x) — computes the argument of a complex number.
- ▷ **ceiling**(x) — returns the smallest integer $\geq x$.
- ▷ **denom**(x)— returns the denominator of a rational fraction or function, x.
- ▷ **float(**x) — converts a number to floating point defined by the computer hardware. This is of the form $m \times 10^e$, where m is a number with a decimal point.
- ▷ **floor**(x) — the largest integer $\leq x$.
- ▷ **bfloat(**x) — converts a number to software-implemented floating point. Slower than **float**(x) but its precision is only limited by the amount of memory. The quantity **fpprec** sets the number of digits of accuracy. For instance: **fpprec**:100;
- ▷ **factor**(n) — factors an integer.
- ▷ **imagpart**(z) — returns the imaginary part of the complex number, z.
- ▷ **isqrt**(n) — returns the "integer" square root of the absolute value of n. Essentially, **isqrt**(x)=**floor**(**sqrt**(x)).
- ▷ **num**(x) — returns the numerator of a rational fraction or function, x.
- ▷ **sqrt**(x) — returns the square root of the absolute value of x.

▷ **sum**$(f(x), x, lower, upper)$ — computes the sum

$$\sum_{x=lower}^{upper} f(x)$$

▷ **polarform**(z) — converts the complex number, z, into polar form: For instance **polarform**(1+2*%i) produces

$$\sqrt{5}\%e^{\%i\,\mathrm{atan}(2)}$$

▷ **product**$(f(x), x, lower, upper)$ — computes the product

$$\prod_{x=lower}^{upper} f(x)$$

▷ **realpart**(z) — returns the real part of the complex number, z.

▷ **rectform**(z) — converts the complex number, z, into the rectangular form: a+%i*b.

▷ Following a symbolic expression with **,numer** causes Maxima to try to convert it into a *numeric* form. Example

```
sqrt(2),numer
```

returns

1.414213562373095

E.3. Commands for functions and equations

Recall that we code functions via

*f(args)***:=***code*;

and a *memoized* function (usually, of a *single* argument although multiple integer arguments are possible) via

*f[arg]***:=***code*;

Non-memoized functions can be nested.

We can access the arguments of a function-call via the **args** command:

args(f(a,b,c))
[a,b,c]

and the function-name via the **op**-command:

op(f(a,b,c))
f

fundef(F) — returns the definition of the function or macro F.

funmake$(F, [a_1, \ldots, a_n])$ — returns $F(a_1, \ldots, a_n)$ without calling F.

Given an equation, $a = b$, we can isolate the sides of the equation via the **rhs** and **lhs** commands:

rhs(a=b)
b
lhs(a=b)
a

We have the **subst**-command or. The command's format is

```
subst(new_value, old_variable, expression)
```

or

```
subst(old_variable=new_value, expression)
```

This command can also take a list of expressions and it performs the substitution in each of them.

A block is the word **block** followed by a comma-separated sequence in parentheses

(1) The first element is a list of local variables or an empty list.
(2) The remaining entries (before the last one) are expressions.
(3) The last entry is a (numeric or symbolic) value.
(4) **block** statements can be nested to any depth.

One exits the **block** by either

(1) dropping through the last entry, or
(2) a **return** statement. Note: this *only* jumps out of the **block** containing it, not necessarily out of the *function* in which it appears[1]. If there are several *nested* **blocks**, this must be taken into account.

at (expr, [eqn_1, ..., eqn_n]) — Given the expression expr, plugs the assignments in the list of equations eqn_1,...,eqn_n
at (expr, eqn) — the **at**-command with only one variable.

E.4. Trigonometric functions

Maxima implements all of the common trig functions and their inverses:

(1) **sin**(x), **asin**(x) — the sine and its inverse.
(2) **cos**(x), **acos**(x) — the cosine and its inverse.
(3) **tan**(x), **atan**(x) — the tangent and its inverse. We also have the **atan2**(y,x) function, which computes the angle between the positive x-axis and a line from the origin and the point (x, y). Note that **atan2**(0,x) is 0 if $x > 0$ and π is $x < 0$. **atan2**(0,0) is undefined.

E.5. Logical Operations

We have already seen the **if** (something) **then** do_something construct. Variables can take on logical values: **true**, **false**. We also have operations

- ▷ **and**
- ▷ **or**
- ▷ **not**

[1] Which is somewhat atypical in programming languages.

▷ **is** (*expression*) — determines whether *expression* is true. Returns **true** or **false**.

EXAMPLE. **if**((a<3) **and** (b>4)) **then** *do_something*

E.6. Looping constructs

Many programs have loops that perform sequences of computations over and over again. Maxima has several of these:

▷ **for** *variable*: *initial_value* **step** *increment* **thru** *limit* **do** (*body*)
Example: **for** t:0 **step** .01 **thru** 10 **do** (*stuff*)

▷ **for** *variable*: *initial_value* **step** *increment* **while** *logical_condition* **do** (*body*)
for x:1 **step** 10 **while** keep_going **do** (*stuff,more_stuff,etc.,*keep_going:(a>5))

▷ **for** *variable*: *initial_value* **step** *increment* **unless** *logical_condition* **do** (*body*)
for x:1 **step** -1 **unless** time_to_stop **do** (*stuff,more_stuff,etc.,*time_to_stop:**true**)

▷ **while** *logical_condition* **do** (*stuff*)

▷ **for** variable **in** L end_tests **do** body Here L can be a *list* or *set* and end_tests is optional

E.7. Predicates

Predicate functions all end with the letter 'p' (except for **equal**) and test *properties* of objects:

evenp — tests whether an integer is even.

integerp — tests whether an integer is present.

oddp — tests whether an integer is odd.

listp — tests whether an object is a list.

orderlessp — a predicate that takes two arguments and tests whether the first is less than the second. This uses an ordering that Maxima establishes for all identifiers and expressions (so *all* pairs of objects are comparable). This is not necessarily the same as numeric comparison (even between numbers).

ordergreatp — reverses **orderlessp**.

ordermagnitudep — compares numbers numerically (as <) and everything else like **orderlessp**.

equal(n, m) — tests whether n and m are numerically equal.

subvarp — returns **true** if its parameter is subscripted.

E.8. Lists

Since Maxima is written in Lisp, it has all of the powerful list-handling features of Lisp. A Maxima *list* is a comma-separated list[2] of data-items enclosed in square brackets. Examples:

[2]Recursive definition!

- ▷ L1:[1,2,3]
- ▷ L2:[[1],[[2]],x^2-1]
- ▷ E:[]

Elements are accessed by indices that start with 1, in square brackets.

- ▷ L1[2]=2
- ▷ L2[3]=x^2-1
- ▷ L2[2][1]=[2]

We have many other list-operations:

- ▷ **append** returns the concatenation of all of the lists that occurs as its arguments. if list_1=[x_1,...x_n], list2=[y_1,...,y_m],...,list_t=[w_1,...,w_k] then **append**(list_1,...,list_t)=[x_1,...x_n,y_1,...,y_m,...,w_1,...,w_k]
- ▷ **assoc** (*key, e, default*) *or* **assoc** *(key, e*) — **assoc** searches for *key* as the first part of an argument of *e* and returns the second part of the first match, if any.
  - *key* may be any expression. *e* must be a nonatomic expression, and every argument of *e* must have exactly two parts. **assoc** returns the second part of the first matching argument of *e*. Matches are determined by **is**(*key* = **first**(*a*)) where *a* is an argument of *e*.
  - If there are two or more matches, only the first is returned. If there are no matches, *default* is returned, if specified. Otherwise, **false** is returned.

  Examples:
  - **assoc** (f(x), foo(g(x) = y, f(x) = z + 1, h(x) = 2*u));
    z + 1
  - **assoc** (yy, [xx = 111, yy = 222, yy = 333, yy = 444]);
    222
  - If there are *no* matches, default is returned, if specified. Otherwise, **false** is returned.
    assoc (abc, [[x, 111], [y, 222], [z, 333]], none);
    none
  - **assoc** (abc, [[x, 111], [y, 222], [z, 333]]);
    false
- ▷ **atom**(x) — returns **true** if x is an atomic expression, **false** otherwise.
- ▷ **cons**(*e*,*L*) returns a new list with *e* as the first element followed by the elements of *L*. Since it doesn't modify the list, *L*, one must write *L*:**cons**(e, L) to put a new element onto the list. This function can also be used where the second argument is other than a list, which might be useful. In this case, **cons** (*expr_1, expr_2*) returns an expression with same operator as *expr_2* but with argument **cons**(*expr_1, args(expr_2)*).
  Examples:

- **cons**(a,[b,c,d]);
  [a, b, c, d]
- **cons**(a,f(b,c,d));
  f(a, b, c, d)

▷ **create_list** — **create_list** (*form, x_1, list_1, . . . , x_n, list_n*) Create a list by evaluating *form* with *x_1* bound to each element of *list_1*, and for each such binding bind *x_2* to each element of *list_2*, . . . The number of elements in the result will be the product of the number of elements in each list. Each variable *x_i* must actually be a symbol — it will not be evaluated. The list arguments will be evaluated once at the beginning of the iteration.

- **create_list** (x^i, i, [1, 3, 7]);
  [x,x,^3 , x^7]
- **create_list** ([i, j], i, [a, b], j, [e, f, h]);
  [[a, e], [a, f], [a, h], [b, e], [b, f], [b, h]]

▷ **copylist**(*list*) Does what the name implies.

▷ **delete**(*expr_1, expr_2*) removes from *expr_2* any arguments of its top-level operator which are the same (as determined by "=") as *expr_1*. Note that "=" tests for formal equality, not equivalence. Note also that arguments of subexpressions are not affected. Examples:

- Removing elements from a list.
  (%i1) **delete** (y, [w, x, y, z, z, y, x, w]);
  (%o1) [w, x, z, z, x, w]
- Removing terms from a sum.
  (%i1) **delete** (sin(x), x + sin(x) + y);
  (%o1) y + x
- Removing factors from a product.
  (%i1) **delete** (u - x, (u - w)*(u - x)*(u - y)*(u - z));
  (%o1) (u - w) (u - y) (u - z)
- Removing arguments from an arbitrary expression.
  (%i1) **delete** (a, foo (a, b, c, d, a));
  (%o1) foo(b, c, d)
- Limiting the number of removed arguments.
  (%i1) **delete** (a, foo (a, b, a, c, d, a), 2);
  (%o1) foo(b, c, d, a)

▷ **endcons**(*e*,*L*) returns a new list with all of the elements of *L* followed by *e*. The second argument (*L*) can also be an expression. Since it doesn't modify the list, *L*, one must write *L*:**endcons**(e, L) to put a new element onto the front of the list. Examples:

- (%i1) **endcons**(a,[b,c,d]);
  (%o1) [b, c, d, a]

- (%i2) **endcons**(a,f(b,c,d));
  (%o2) f(b, c, d, a)

▷ **first**(*expr*) — Returns the first part of *expr* which may result in the first element of a list, the first row of a matrix, the first term of a sum, etc.

▷ **firstn** (*expr, count*) — Returns the first *count* arguments of *expr*, if *expr* has at least count arguments. Returns *expr* if *expr* has less than *count* arguments.

▷ **join** (*L, m*) — Creates a new list containing the elements of lists *L* and *m,* interspersed. The result has elements [*L*[1], *m*[1], *L*[2], *m*[2], . . .]. The lists *L* and *m* may contain any type of elements.

▷ **last** (*expr*) — Returns the last part (term, row, element, etc.) of the *expr*.

▷ **lastn** (*expr, count*) — — Returns the last *count* arguments of *expr*, if *expr* has at least count arguments. Returns *expr* if *expr* has less than *count* arguments.

▷ **length** (*expr*) — Returns (by default) the number of parts in the external (displayed) form of *expr*. For lists this is the number of elements, for matrices it is the number of rows, and for sums it is the number of terms.

▷ **map**(*f,list*) — returns a list with the function *f* applied to the members of *list*.

▷ **listp** (*expr*) — Returns **true** if *expr* is a list else **false**.

▷ **lreduce**(*F,s*) — extends the binary function, *F*, to all of the list, *s*, by composition from the left. Example
lreduce(*F,s*)=*F*(. . . *F*(*s_1 ,s_2*),*s_3*),*s_4*,. . . ,*s_n*).

▷ **makelist** — **makelist** (), creates the empty list, [].

- **makelist** (*expr, n*), **makelist** (*expr*), creates a list with *expr* as its single element. **makelist** (*expr, n*) creates a list of *n* copies of *expr*.
- **makelist** (*expr(i), i, i_max*) creates a list with *expr* evaluated at *i*=1 to *i*=*imax*, stepped by 1 each time.
- **makelist** (*expr(i), i, i_0, i_max*) creates a list with *expr* evaluated at *i*=*i0* to *i*=*imax*, stepped by 1 each time.
- **makelist** (*expr(i), i, i_0, i_max, step*) creates a list with *expr* evaluated at *i*=*i0* to *i*=*imax*, stepped by *step* each time.
- **makelist** (*expr(x), x, list*) creates a list with *expr(x)* evaluated **x** equal to successive elements of *list*.

▷ **member** *(expr_1, expr_2*) Tests whether *expr_1* is a member of *expr_2*, which may be a list or expression.

▷ **pop** (*list*) — removes and returns the first element of *list*. Note: it modifies *list* in the process.

▷ **push** (*item, list*) — puts *item* as the first member of *list* and returns a copy of the new list.

▷ **rest** — **rest** (*expr, n*) **rest** (expr) Returns *expr* with its first *n* elements removed if *n* is positive and its last - *n* elements removed if *n* is negative. If *n* is 1 it may be omitted. The first argument *expr* may be a list, matrix, or other expression. Applying **rest** to expression such as *f(a,b,c)* returns *f(b,c)*. In general, applying **rest** to a non-list doesn't make sense. For example, because '^' requires two arguments, **rest**(a^b) results in an error message. The functions **args** and **op** may be useful as well, since **args**(a^b) returns [a,b] and **op**(a^b) returns ^.

▷ **reverse** (*list*) — Reverses the order of the members of the *list* (not the members themselves). **reverse** also works on general expressions, e.g. **reverse**(a=b); gives b=a.

▷ **rreduce**(*F,s*) — Like **lreduce** but it works from the right rather than the left. Example:
rreduce(*F,s*)=*F*(*s_1*,F(*s_2*,...F(*s_(n-2)*,F(*s_(n-1)*,*s_n*)...).

▷ **sort**(*list,predicate*) — returns a list that is the result of sorting *list* in ascending order, using the *predicate* to compare pairs of items. If *predicate* is omitted, then **orderlessp** is used (which can compare *any* two Maxima objects or expressions). To sort in descending order, use **ordergreatp** as the *predicate*.
The *predicate* may be specified as the name of a function or binary infix operator, or as a lambda expression. If specified as the name of an operator, the name must be enclosed in double quotes.

- **sort** ([1, a, b, 2, 3, c], '**orderlessp**);
  [1, 2, 3, a, b, c]
- **sort** ([1, a, b, 2, 3, c], '**ordergreatp**);
  [c, b, a, 3, 2, 1]
- L : [%pi, 3, 4, %e, %gamma];
  [%pi, 3, 4, %e, %gamma]
  sort (L, ">");
  [4, %pi, 3, %e, %gamma]
- **ordermagnitudep** orders numbers, constants, and constant expressions the same as <, and all other elements the same as **orderlessp**.
  L: [%i, 1+%i, 2*x, minf, inf, %e, sin(1), 0,1,2,3, 1.0, 1.0b0];
  [%i, %i + 1, 2 x, minf, inf, %e, sin(1), 0, 1, 2, 3, 1.0, 1.0b0]
  sort (L, **ordermagnitudep**);
  [minf, 0, sin(1), 1, 1.0, 1.0b0, 2, %e, 3, inf, %i, %i + 1, 2 x]

▷ **sublist** (*list, p*) — Returns the list of elements of *list* for which the predicate *p* returns true.
L: [1, 2, 3, 4, 5, 6];
[1, 2, 3, 4, 5, 6]

sublist (L, **evenp**);
[2, 4, 6]

▷ **sublist_indices** (L, P) — Returns the indices of the elements x of the list L for which the predicate **maybe**(*P(x)*) returns true; this excludes unknown as well as false. P may be the name of a function or a lambda expression. L must be a literal list.

- **sublist_indices** ('[a, b, b, c, 1, 2, b, 3, b], **lambda** ([x], x='b));
  [2, 3, 7, 9]

▷ **tree_reduce** — **tree_reduce** (*F, s*), **tree_reduce** (*F, s, s_0*) Extends the binary function F to an n-ary function by composition, where s is a set or list.

- **tree_reduce** is equivalent to the following: Apply F to successive pairs of elements to form a new list [*F(s_1, s_2), F(s_3, s_4), ...*], carrying the final element unchanged if there are an odd number of elements. Then repeat until the list is reduced to a single element, which is the return value.
- When the optional argument *s_0* is present, the result is equivalent **tree_reduce**(*F,* **cons**(*s_0, s*)).

▷ **unique** (*L*) — Returns the unique elements of the list L.

- When all the elements of L are unique, unique returns a shallow copy of L, not L itself.
- If L is not a list, **unique** returns L.

E.9. Strings

By default, Maxima displays strings without quotes so they look like symbols or variables (which can be confusing, since strings are *not* variables). The variable **stringdisp** controls this. By default, it is **false**. Setting it to **true** causes strings to be displayed with double-quotes.

Many of the commands listed here are in the 'stringproc' package, which is *automatically* loaded whenever one calls one of these functions.

▷ **charat**(s, n) — Returns the n-th character of s, numbering them from 1 on.

▷ **charlist**(s) — returns a list of the characters in s. Characters are regarded as strings of length 1.

▷ **concat**$(a_1, a_2, \dots)$ — Concatenates its arguments. The arguments must evaluate to atoms. The return value is a symbol if the first argument is a symbol and a string otherwise.

▷ **eval_string**(s) — Parse the string s as a Maxima expression and evaluate it. It turns a character-string into a *symbol,* i.e., a *variable.*

- ▷ **sconcat**$(a_1, a_2, \dots)$ — Concatenates its arguments into a string. Unlike **concat**, the arguments do not need to be atoms.
- ▷ **scopy**(s) — returns a new string that is a copy of s.
- ▷ **sdowncase**(s) — converts all characters to lowercase. Variations: **sdowncase**(s, n) starts with character n, and **sdowncase**(s, n, m) runs from character n to m.
- ▷ **string**(e) — converts the Maxima expression e into a string. In a manner of speaking, it's the *opposite* of **eval_string.**
- ▷ **sequal**(s_1, s_2) — returns **true** if s_1 and s_2 are the same length and contain the same characters.
- ▷ **sequalignore**(s_1, s_2) — Like **sequal** but ignores case.
- ▷ **simplode**(L) — takes a list, L, of expressions and concatenates them into a string. Variation: **simplode**(L, d) — like the previous case, but it uses the string d as a delimiter.
- ▷ **sinsert**(t, s, p) — returns the result of inserting string t into string s at position p.
- ▷ **sinvertcase**(s) — returns the result of converting uppercase letters of s to lowercase and vice-versa. Variations: **sinvertcase**(s, n) and **sinvertcase**(s, n, m) — inverts case of letters starting with position n to position m (or the end of the string).
- ▷ **slength**(s) — returns the number of characters in s.
- ▷ **smake**(n, c) — returns a string that is n copies of the character c. One wonders what application the originator of *this* command had in mind ☺.
- ▷ **smismatch**(s_1, s_2) — returns the index of the first character in string s_1 that doesn't match the corresponding character in string s_2. Variation: **smismatch**(s_1, s_2, t), where t is a test used to compare characters. The default is **sequal** but you can also specify **sequalignore** to ignore case.
- ▷ **split**(s) — Returns the list of all tokens in s. Each token is an unparsed string. Variations: **split**(s, d), **split**(s, d, m), **split** uses d as delimiter. If d is not given, the space character is the default delimiter. m is a boolean variable which is **true** by default. Multiple delimiters are read as one. This is useful if tabs are saved as multiple space characters. If m is set to **false**, each delimiter is noted.
- ▷ **sposition**(c, s) — Returns the position of the first character in string, s, which matches character, c. The first character in s is in position 1. For matching characters ignoring case see **ssearch**.
- ▷ **sremove**(u, s) — Returns a string like s but without all substrings matching u. Variations: **sremove**(u, s, t), **sremove**(u, s, t, m), **sremove**(u, s, t, m, n) Default test

function, t, for matching is **sequal**. If **sremove** should ignore case while searching for u, use **sequalignore** as t. Use m and n to limit searching. Note that the first character in string is in position 1.

▷ **sremovefirst**(u, s) — Like **sremove** but it only removes the first occurrence of u. Variations: **sremovefirst**(u, s, t), **sremovefirst**(u, s, t, m), **sremovefirst**(u, s, t, m, n).

▷ **sreverse**(s) — returns s reversed.

▷ **ssearch**(u, s) — returns the position of the first substring of s that matches the string u. Variations: **ssearch**(u, s, t), **ssearch**(u, s, t, m), **ssearch**(u, s, t, m, n). Default test function, t, for matching is **sequal**. If **ssearch** should ignore case, use **sequalignore** as t. Use m and n to limit searching. Note that the first character in string is in position 1.

▷ **ssort**(s) — Variation: **ssort**(s, t), where t is a test. Returns a string that contains the characters of s sorted. Possible tests:

- 'clessp The default. Sorts in ascending order.
- 'clesspignore Sorts in ascending order, ignoring case.
- 'cgreaterp Sorts in descending order..
- 'cgreaterpignore Sorts in descending order, ignoring case.

▷ **ssubst**(u, v, s) — edits string s, replacing substrings v by u. Variations: **ssubst**(u, v, s, t), **ssubst**(u, v, s, t, m), **ssubst**(u, v, s, t, m, n). Here t is a test and m, n control the range of characters in s where the edits take place.

▷ **ssubstfirst**(u, v, s) — like **ssubst**(u, v, s) but only replaces the *first* occurrence of v. Variations: **ssubstfirst**(u, v, s, t), **ssubstfirst**(u, v, s, t, m), **ssubstfirst**(u, v, s, t, m, n). Here t is a test and m, n control the range of characters in s where the edits take place.

▷ **strim**(u, s) — returns a string like s, but with all characters that appear in the string u removed from both ends. Variations: **striml**(u, s) Like **strim** except that only the *left* end of string is trimmed. **strimr**(u, s) Like **strim** except that only the *right* end of string is trimmed.

▷ **substring**(s, m), **substring**(s, m, n) — Returns the substring of s beginning at position m and ending at position n. The character at position n is not included. If n is not given, the substring contains the rest of the string.

▷ **supcase**(s), **supcase**(s, m), **supcase**(s, m, n) — returns s except that lowercase characters from position m to n are replaced by the corresponding uppercase ones. If n is not given, all lowercase characters from start to the end of string are replaced.

▷ **tokens**(s), **tokens**(s,t) — returns a list of tokens, which have been extracted from s. The tokens are substrings whose characters satisfy a certain test function, t. If t is not given, **constituent** is used as the default test. The test functions
- **constituent** — the characters: ! " # % ' ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ? @ A B C D E F G H I J K L M N O P Q R S T U V W X Y Z [\ ] ^ _ ` a b c d e f g h i j k l m n o p q r s t u v w x y z { | } ~
- **'alphacharp** — alphabetic
- **'digitcharp** — digit
- **'lowercasep** — self-explanatory
- **'uppercasep** — self-explanatory
- **'charp** — a character
- **'alphanumericp** — self-explanatory

E.10. Structures

A *structure* is a kind of list where the entries in the list have *names* and are referred to by those names. We define types of structures with the **defstruct**-command:

```
defstruct(structure_name(name_1,...,name_n));
```

We create structures via the **new-**command

```
z:new(structure_name(value_1,...,value_n));
```

Example:

```
defstruct(dog(legs,eyes,color));
```

This defines a *kind* of structure called 'dog'. We create a actual structure with

```
fido:new(dog(4,2,"brown"));
```

We access these fields via the '@'-command:

```
fido@legs;
```

and Maxima returns 4. If we simply type fido, Maxima prints

```
dog(legs=4,eyes=2,color="brown");
```

Note that one cannot create a new structure using these '=' commands.

After a *tragic accident*

```
fido@legs:3;
```

and, if we simply type fido, we get

```
dog(legs=3,eyes=2,color="brown");
```

If we don't assign a value to an entry, it simply displays as its name:

```
rover:new(dog);
```

results in

```
dog(legs,eyes,color)
```

Having created a structure using the **new** command, we can destroy it via the **kill** command

```
kill(rover);
```

and Maxima replies

done

One can also delete entry-values in a structure using the **kill** command:

```
kill(fido@legs)
```

results in

```
dog(legs,eyes=2,color="brown");
```

E.11. Sets

E.11.1. Introduction. A set is a list in which each element is unique — uniqueness being determined by **is**(x=y). Maxima provides several commands for creating sets. The most basic is:

```
set(a,b,c,a)
```

and Maxima responds with

```
{a,b,c}
```

eliminating the duplicate 'a'. The **set**-command above is completely equivalent to writing

```
{a,b,c,a}
```

with curly brackets instead of straight ones. Of course

```
set()
```

produces the empty set

```
{}
```

The command **setify** converts a list to a set

```
setify([a,b,c,a])
```

produces

```
{a,b,c}
```

The **listify**-command does the opposite.

The **map**-command can be used to iterate over elements of a set (just as it does for a list):

```
map(f,{a,b,c})
```

gives

```
{f(a),f(b),f(c)}
```

You also iterate via

```
for x in s do stuff
```

or

```
for x in s while keep_going do stuff
```

E.11.2. Set-operations.

- ▷ **adjoin**(x,a) — returns the set-union $a \cup \{x\}$. The set a is unchanged.
- ▷ **disjoin**(x,a) — the opposite of **adjoin**, returns the set-difference $a \setminus \{x\}$. The set a is unchanged.
- ▷ **belln**(n) — returns the n^{th} *Bell number*[3] — the number of partitions of a set of size n. A *partition* of a set S is defined as a family of nonempty, pairwise disjoint subsets of S whose union is S. For example, $B_3 = 5$ because the 3-element set $\{a, b, c\}$ can be partitioned in 5 distinct ways:

$$\{\{a\},\{b\},\{c\}\}$$
$$\{\{a\},\{b,c\}\}$$
$$\{\{b\},\{a,c\}\}$$
$$\{c\},\{a,b\}\}$$
$$\{\{a,b,c\}$$

  This is also the number of distinct equivalence-relations that can exist on this set.
- ▷ **cardinality**(a) — returns the number of elements in the set a.
- ▷ **cartesian_product**(b_1, ... , b_n) — Returns a set of lists of the form $[x_1, \ldots, x_n]$, where $x_i \in b_i$, respectively.
  cartesian_product({0,1}) produces

$$\{[0],[1]\}$$

  cartesian_product ({0,1},{0,1}) produces

$$\{[0,0],[0,1],[1,0],[1,1]\}$$

[3] In an example of Stigler's law of eponymy, they are named after Eric Temple Bell, who wrote about them in the 1930s. These numbers go back much further, being mentioned in the medieval Japanese novel, [57].

▷ **disjointp**(a, b) — returns **true** if the sets a and b are *disjoint*.
▷ **divisors**(n) — returns the set of divisors of n, including 1 and n. **divisors** distributes over equations, lists, matrices, and sets:

```
(%i1) divisors(a = b);
(%o1)                 divisors(a) = divisors(b)
(%i2) divisors([a, b, c]);
(%o2)[divisors(a), divisors(b), divisors(c)]
(%i3) divisors(matrix ([a, b], [c, d]));
                [ divisors(a)  divisors(b) ]
(%o3)           [                          ]
                [ divisors(c)  divisors(d) ]
(%i4) divisors({a, b, c});
(%o4){divisors(a), divisors(b), divisors(c)}
```

▷ **elementp**(x, a) — returns **true** if $x \in a$.
▷ **emptyp**(a) — returns **true** if $a = \emptyset$, or if a is the empty list.
▷ **equiv_classes**(s, F) — if s is a set and F is a function of two variables returning **true** or **false** (that is an equivalence relation), this computes the set of equivalence classes of elements of s. Example:

```
(%i1) equiv_classes({1, 2, 3, 4, 5, 6, 7},
      lambda([x, y],
          remainder(x - y, 3) = 0));
(%o1)          {{1, 4, 7}, {2, 5}, {3, 6}}
```

▷ **every**(F, s) (*set* form) — returns **true** if the predicate F is **true** for all given arguments (elements of the set s). Example:

```
(%i1) every(integerp, {1, 2, 3, 4, 5, 6});
(%o1)                          true
(%i2) every(atom, {1, 2, sin(3), 4,
              5 + y, 6});
(%o2)                          false
```

▷ **every**(F, L_1 ,..., L_k) (*list* form) — returns **true** if the predicate F (taking k parameters) is **true**, where its i^{th} parameter is taken from L_i. Example:

```
(%i1) every ("=", [a, b, c], [a, b, c]);
(%o1)                          true
(%i2) every ("#", [a, b, c], [a, b, c]);
(%o2)                          false
```

▷ **extremal_subset**(s, f, max) — returns the subset of s on which the function f takes on its *maximum* values.

▷ **extremal_subset**(s, f, min) — returns the subset of s on which the function f takes on its *minimum* values.

▷ **flatten** (expr) — Collects arguments of subexpressions which have the same operator as expr and constructs an expression from these collected arguments. Subexpressions in which the operator is different from the main operator of expr are copied without modification, even if they, in turn, contain some subexpressions in which the operator is the same as for expr. Examples:
Applied to a list, **flatten** gathers all list elements that are lists.

```
(%i1) flatten([a, b, [c, [d, e], f],
                        [[g, h]], i, j]);
(%o1)        [a, b, c, d, e, f, g, h, i, j]
```

Applied to a set, **flatten** gathers all members of set elements that are sets.

```
(%i1) flatten({a, {b}, {{c}}});
(%o1)                          {a, b, c}
(%i2) flatten({a, {[a], {a}}});
(%o2)                          {a, [a]}
```

▷ **full_listify** (a) — Replaces every set operator in *a* by a list operator, and returns the result. **full_listify** replaces set operators in nested subexpressions, even if the main operator is not set.

▷ **fullsetify** (a) — When *a* is a list, it replaces the list operator with a set operator, and applies **fullsetify** to each member which is a set. When *a* is not a list, it is returned unchanged.

▷ **identity**(a) — Returns a for any argument a.

▷ **integer_partitions**(n) — Returns a set of integer partitions of *n*, that is, lists of integers which sum to *n*. The numbers in each list are sorted from highest to lowest. Alternate form: **integer_partitions**(n,len) — Returns a set of partitions of length len.

```
(%i1) integer_partitions(3);
(%o1)                {[1, 1, 1], [2, 1], [3]}
(%i2) s: integer_partitions(25);
(%i3) cardinality(s);
(%o3)                          1958
(%i4) map (lambda ([x], apply ("+", x)), s);
(%o4)                          {25}
(%i5) integer_partitions(5, 3);
(%o5) {[2, 2, 1], [3, 1, 1],
          [3, 2, 0], [4, 1, 0], [5, 0, 0]}
```

```
(%i6) integer_partitions(5, 2);
(%o6)                    {[3, 2], [4, 1], [5, 0]}
```

We can also find partitions that satisfy a *condition*

```
(%i1) s:integer_partitions(10);
(%i2) cardinality(s);
(%o2)                                42
(%i3) xprimep(x) := integerp(x)
        and (x > 1) and primep(x);
(%i4) subset (s,
      lambda ([x], every (xprimep, x)));
(%o4) {[2, 2, 2, 2, 2], [3, 3, 2, 2],
        [5, 3, 2], [5, 5], [7, 3]}
```

▷ **intersection**(a_1, ..., a_n) —returns the intersection of the sets a_1 ... a_n. Alternate form: **intersect**(a_1, ..., a_n)

▷ **makeset**(expr, x, s) — similar to **makelist**, but for sets. Returns a set with members generated from the expression expr, where x is a list of variables in expr, and s is a set or list of lists. To generate each set member, expr is evaluated with the variables x bound in parallel to a member of s.
Each member of s must have the same length as x. The list of variables x must be a list of symbols, without subscripts. Even if there is only one symbol, x must be a list of one element, and each member of s must be a list of one element. Example:

```
makeset(i/j,[i,j],[[1,a],[2,b],[3,c],[4,d]]);
```

returns

$$\left\{\frac{1}{a}, \frac{2}{b}, \frac{3}{c}, \frac{4}{d}\right\}$$

and

```
makeset(sin(x), [x], {[1], [2], [3]});
```

returns

$$\{\sin(1), \sin(2), \sin(3)\}$$

▷ **multinomial_coeff**$(a_1, \ldots, a_n)$ — returns

$$\frac{(a_1 + \cdots + a_n)!}{a_1! \cdots a_n!}$$

▷ **num_distinct_partitions**(n) — returns the number of partitions of n (in the sense of the **integer_partitions**-command) where the numbers in the partitions are all *distinct*.

num_distinct_partitions(n,list), with the keyword **list** in it, returns a *list:*

[**num_distinct_partitions** (1),..., **num_distinct_partitions**(n)]

- ▷ **num_partitions**(n) — returns the number of partitions of n (in the sense of the **integer_partitions**-command), i.e. the *cardinality* of the set **integer_partitions**(n). num_partitions(n,list), with the keyword **list** in it, returns a *list:*

[**num_partitions** (1),..., **num_partitions**(n)]

- ▷ **partition_set**(a,F) — given a set, a, it returns a list of two sets:

$[\{e \in a \text{ with } F(e) = \textbf{false}\}, \{e \in a \text{ with } F(e) = \textbf{true}\}]$

For instance:

```
s:{1,2,3,4,5,6,7,8};
partition_set(s,evenp)
```

returns

$$[\{1,3,5,7\},\{2,4,6,8\}]$$

- ▷ **permutations**(a); — returns a set of all permutations of the elements of the list or set a.
- ▷ **powerset**(a) — returns the set of all subsets of the set, a. **powerset**(a,n) returns the set of all subsets of cardinality n.
- ▷ **random_permutation**(a) — returns a random permutation of the set or list, a, as constructed by the Knuth shuffle algorithm: Given an array of items numbered from 0, the algorithm can be defined as follows (in Python):

```
from random import randrange

def knuth_shuffle(x):
    for i in range(len(x)-1, 0, -1):
        j = randrange(i + 1)
        x[i], x[j] = x[j], x[i]

x = list(range(10))
knuth_shuffle(x)
print("shuffled:", x)
```

- ▷ **setdifference**(a,b) — returns the set $a \setminus b$.
- ▷ **setequalp**(a,b) — returns **true** if sets a and b have the same number of elements and **is**$(x = y)$ is true for $x \in a$ and $y \in b$, considered in the order determined by **listify**. Otherwise, **setequalp** returns false.

- ▷ **setp**(a) — returns **true** if a is a set, **false** otherwise.
- ▷ set_partitions(a) — returns the set of all partitions of a, or a subset of that set. set_partitions(a, n) returns a set of all partitions of a into n nonempty subsets.
- ▷ **some**(F, a) — returns **true** if **is**$F(x)$ is **true** for some $x \in a$. **some**$(F, L_1, ..., L_n)$ Given one or more lists as arguments, **some**$(F, L_1, ..., L_n)$ returns true if **is**$(F(x_{1,j}, ..., x_{n,j}))$ returns **true** for some j where $x_{i,j} \in L_i$ is the j^{th}-element in the i^{th} list for all i.
- ▷ **stirling1**(n, m) — the *Stirling number of the first kind*. It's the coefficient of x^m in the Pochhammer falling factorial

$$x_{(n)} = x(x-1)\cdots(x-n+1)$$

- ▷ **stirling2**(n, m) — the *Stirling number of the second kind*. It's the coefficient, $S(n, m)$, of $x_{(m)}$ in the expansion

$$x^n = \sum_{m=0}^{n} S(n, m) x_{(i)} \tag{E.11.1}$$

- ▷ **subset**(a, F) — the subset of elements $x \in a$ such that $F(x) =$ **true**.
- ▷ **subsetp**(a, b) — returns **true** if $a \subseteq b$.
- ▷ **symmdifference**$(a_1, \ldots, a_n)$ — returns the set of elements in all of the a_i that are *not* in any of the other sets.
- ▷ **union**$(a_1, \ldots, a_n)$ — returns the union of the sets.

E.12. Macros

To describe what a macro does, we must first analyze how a *function* is called. When a function, $f(x, y, z)$, is called, Maxima

(1) evaluates x, y, z
(2) *jumps* to the function-code and plugs those values into the body of f.
(3) then *returns* with the computed values

A *macro* $f(x, y, z)$ superficially resembles a function but it

(1) executes the *body* of the macro on x, y, z and other expressions (doing something like a *text-edit), inserting* it into the code that called it — i.e., no *jumping* and *returning,*
(2) then it *executes* the rewritten expression.

A *user-defined* macro uses the ::= operation:

```
zz(x):=x/10;
```

The right side of this is *edited* into the expression where

```
zz(x)
```

appears. In other words

```
p:a+b+zz(top)+d
```

is rewritten as

```
p:a+b+top/10+d
```

before it is executed. As such, the effect of this macro is very similar to a function-call.

The main macro *built in to* Maxima is **buildq.**

It has the form

```
buildq([x1:vq, x2:v2,...,xn:vn],
                        stuff involving x1,...,xn
      );
```

It does a kind of text-edit of *stuff* replacing xi by vi for $i = 1,\ldots,n$, and then it *executes* the edited code. If an xi has *no* corresponding vi, whatever value it was assigned in the past is used. So

```
b:29;
buildq ([a: x, b], a + b + c);
```

results in

```
x + c + 29
```

The command

```
 buildq ([e: [a, b, c]], foo (x, e, y));
```

results in

```
foo(x, [a, b, c], y)
```

which is then executed.

Note that the substitutions a **buildq** command are carried out in *parallel*, so

```
 buildq ([a:x,b:a,c,b], foo (x, e, y));
```

produces

```
foo (x, a, b);
```

If they had been carried out *sequentially* (from left to right, as **subst** does), we would've gotten

```
foo (x, x, x);
```

Within a **buildq** command, the **splice**-command (whose argument is a list) *interpolates* that list into a larger list. So, whereas,

```
 buildq ([e: [a, b, c]], foo (x, e, y));
```

produces

```
foo(x, [a, b, c], y)
```

the **splice** command

```
buildq ([e: [a, b, c]], foo (x, splice(e), y));
```

produces

```
foo(x, a, b, c, y)
```

Outside of a **buildq** command, the **splice**-command is merely an error.

Here's an example that combines several language features:

```
show_values ([L]) ::= buildq ([L],
        map ("=", 'L, L));
```

If we have statements

```
a:3;
b:2;
c:1000;
print(show_values(a,b,c));
```

is interpreted in several steps. First, it rewrites the **print** statement as

```
print(buildq ([a,b,c],
      map ("=", '[a,b,c], [a,b,c])));
```

Next, we get

```
print(map ("=", '[a,b,c], [a,b,c]));
```

Since the **print** command is a function rather than a macro, it *executes* the **map**-command to get

```
print([a=2,b=3,c=1000]);
```

and prints it.

Other macro-related commands, include

macroexpand — this expands a macro but doesn't execute it. If the macro calls *other* macros, they are also expanded.

macroexpand1 — this expands a macro but doesn't execute it. If the macro calls *other* macros, they are *not* expanded — it only expands the *topmost* level of nested macro-calls.

E.13. Input and Output

The most widely-used forms of output involve *plotting* and *drawing pictures.* These operations are so important, an entire appendix is devoted to them — see appendix F on page 323.

Simple (and useful for debugging) output-commands are:

- ▷ **display** (*expr_1, expr_2, ...*) — displays equations whose left side is *expr_i* unevaluated, and whose right side is the value of the expression centered on the line. This function is useful in blocks and for statements in order to have intermediate results displayed. The arguments to display are usually atoms, subscripted variables, or function calls.
- ▷ **print** (*expr_1, ..., expr_n*) — Evaluates and displays *expr_1, ..., expr_n* one after another, from left to right, starting at the left edge of the console display. The value returned by **print** is the value of its last argument. **print** does not generate intermediate expression labels.
  Note: ___ (two underscores) represents the input expression currently being evaluated. That is, while an input expression *expr* is being evaluated, ___ is *expr*. Example:
  - **print** ("My name is ", ___);
    My name is **print**(My name is, ___)
  - zztop (___);
    zztop(zztop(___))
- ▷ **appendfile** (*filename*) — Appends a console transcript to *filename*. **appendfile** is the same as **writefile**, except that the transcript file, if it exists, is always appended.
- ▷ **batchload** (*filename*) — Reads Maxima expressions from *filename* and evaluates them, without displaying the input or output expressions and without assigning labels to output expressions. Printed output (such as produced by print or describe) is displayed, however.
- ▷ **closefile**() closes the transcript file opened by **appendfile** or **writefile**.
- ▷ **file_search** (*filename*) — **file_search** searches for the file *filename* and returns the path to the file (as a string) if it can be found; otherwise **file_search** returns false. **file_search** (*filename*) searches in the default search directories, which are specified by the **file_search_maxima**, **file_search_lisp**, and **file_search_demo** variables.
  file_search first checks if the actual name passed exists, before attempting to match it to "wildcard" file search patterns.
  - **file_search_maxima** Option variable,
  - **file_search_lisp** Option variable,
  - **file_search_demo** Option variable,

- **file_search_usage** Option variable,
- **file_search_tests**.

▷ **load** — **load** (*filename*) Evaluates expressions in *filename*, thus bringing variables, functions, and other objects into Maxima. The binding of any existing object is clobbered by the binding recovered from *filename*. To find the file, **load** calls **file_search** with **file_search_maxima** and **file_search_lisp** as the search directories. If **load** succeeds, it returns the name of the file. Otherwise **load** prints an error message.

- **load** works equally well for Lisp code and Maxima code. Files created by **save**, **translate_file**, and **compile_file**, which create Lisp code, and **stringout**, which creates Maxima code, can all be processed by **load**. **load** calls **loadfile** to load Lisp files and **batchload** to load Maxima files.

▷ **directory** — **directory** (*path*) Returns a list of the files and directories found in *path* in the file system. **path** may contain wildcard characters (i.e., characters which represent unspecified parts of the path), which include at least the asterisk on most systems, and possibly other characters, depending on the system.

▷ **printfile** — **printfile** (*path*) Prints the file named by path to the console. *path* may be a string or a symbol; if it is a symbol, it is converted to a string. If *path* names a file which is accessible from the current working directory, that file is printed to the console. Otherwise, **printfile** attempts to locate the file by appending path to each of the elements of **file_search_usage** via **filename_merge**. **printfile** returns path if it names an existing file, or otherwise the result of a successful filename merge.

▷ **save** —This takes several forms:

- **save** (*filename, name_1, name_2, name_3,* ...) Stores the current values of *name_1, name_2, name_3,* ..., in *filename*. The arguments are the names of variables, functions, or other objects. If a name has no value or function associated with it, it is ignored. **save** returns *filename.* **save** stores data in the form of Lisp expressions. If *filename* ends in .lisp the data stored by save may be recovered by **load** (*filename*). See **load**.
- **save** (*filename, values, functions, labels,* ...) — stores the items named by *values, functions, labels,* etc. The names may be any specified by the variable **infolists.values** comprises all user-defined variables.

- **save** (*filename, [m, n]*) — stores the values of input and output labels *m* through *n*. Note that *m* and *n* must be literal integers. Input and output labels may also be stored one by one, e.g., **save** ("foo.1", %i42, %o42).
- **save** (*filename,* **labels**) stores all input and output labels. When the stored labels are recovered, they clobber existing labels.
- **save** (*filename*, **all**) — stores the current state of Maxima.
- **save** (*filename, name_1=expr_1, name_2=expr_2, ...*) stores the values of *expr_1, expr_2, ...*, with names *name_1, name_2, ...* It is useful to apply this form to input and output labels, e.g., **save** ("foo.1", aa=%o88). The right-hand side of the equality in this form may be any expression, which is evaluated.

▷ **stringout** — Similar to **save** above but stores information in Maxima form rather than Lisp form. This occurs in several forms (see **save**, above):

- **stringout** (filename, expr_1, expr_2, expr_3, ...)
- **stringout** (filename, [m, n])
- **stringout** (filename, input)
- **stringout** (filename, functions)
- **stringout** (filename, values)

▷ **with_stdout** — **with_stdout** (*s, expr_1, expr_2, expr_3, ...*) Evaluates *expr_1, expr_2, expr_3, ...* and writes any output thus generated to a file *f* or output stream *s.* The evaluated expressions are not written to the console. **with_stdout** redirects output commands that normally print on the console to the file. Output may be generated by **print** and **display** among other functions.

```
with_stdout ("tmp.out", for i:5 thru 10 do
print (i, "! yields", i!))$
(%i2) printfile ("tmp.out")$
5 ! yields 120
6 ! yields 720
7 ! yields 5040
8 ! yields 40320
9 ! yields 362880
10 ! yields 3628800
```

▷ **writefile** — **writefile**(*filename*) Begins writing a transcript of the Maxima session to *filename*. All interaction between the user and Maxima is then recorded in this file, just as it appears on the console.

APPENDIX F

Visual outputs

F.1. Plotting

F.1.1. Basic plot-commands. Basic plotting in wxMaxima is done by external software called 'gnuPlot', which provides the commands **plot2d** and **plot3d** as well as several others. The basic command varies depending on whether you are doing a single plot or multiple superimposed plots.

With a single plot, the format looks like (z=2,3)

```
plotzd(function_specification,
[x,lower_lim,upper_lim]
{,[y, lower_lim,upper_lim]}
{,[style,plot_options]}{,terminal_spec}
{,file_spec})
```

(Items in curly brackets are optional).

With multiple superimposed plots (only with **plot2d**) we have

```
plot2d([f1,...,fn],[x,lower_lim,upper_lim]
{,[y, lower_lim,upper_lim]},
{[style, plot_option_1,...,plot_option_n]}
{,terminal_spec}{,file_spec})
```

F.1.2. Function specifications. Descriptions of functions to be plotted take many forms. The most basic function-specification is simply to name the function. Examples:

- ▷ x^2-3*x
- ▷ sin(cos(x))

In many cases, functions are defined by Maxima code that has an **if**-statement that tests the independent variable. In these cases, the function must be quoted with a *single quote.* Example:

- ▷ 'myfunct(x)

With functions like this, the **if**-statement is evaluated once by Maxima and the outcome remains fixed throughout the plot. The solution is to *quote* the function, so the function itself is sent to GnuPlot.

- ▷ With functions only defined at certain points, we can define a function to be *discrete.* This consists of the keyword **discrete**

in a list with a list of points. For example:
f:[**discrete**,[[1,2],[2,3],[3,5],[4,7],[5,11],[6,13],[7,17],
[8,19],[9,23],[10,29]]];
defines a function equal to the first 10 prime numbers. In general, a discrete function can take two forms:
[**discrete**,[[x1,y1],...,[xn,yn]]] *or*
[**discrete**,[[x1,...,xn],[y1,...,yn]]]

- ▷ An *equation* defines an *implicit function,* and these can be plotted. One must provide ranges of values for *all* the variables. Example: **plot2d**(x^2+y^4=1,[x,-1,1],[y,-1,1]).

F.1.3. High-level plot styles. These are specified with the word '**style**' followed in a list that indicates the style:

- ▷ **lines** — this is the default. Points are plotted and connected with lines.
- ▷ **points** — plots isolated points.
- ▷ **linespoints** — plots lines but also highlights the original points.
- ▷ **dots** — plots isolated dots.

Example:

[**style**,**lines,lines,points**] if are plotting three functions and want these three respective styles.

We also specify colors of the various plots via:
[**color**,name_1,...,name_n]
for n plots. The acceptable colors-names are

- ▷ red
- ▷ green
- ▷ blue
- ▷ magenta
- ▷ cyan
- ▷ yellow
- ▷ orange
- ▷ violet
- ▷ brown
- ▷ gray
- ▷ black
- ▷ white
- ▷ # followed by six hexadecimal digits: two for the red component, two for green component and two for the blue component. This expression must be *quoted* with double-quotes. Example: "#3d01f2". This produces a bluish violet color.
- ▷ If the name of a given color is *unknown, black* is used

If there are more curves or surfaces than colors, the colors are repeated in sequence. For instance, if you write [**color**,black] *all* of the plots will

be black. If no colors are specified, defaults are used (blue, red, green, etc.).

If we have points in the styles, we can specify the *point type* via the statement

[**point_type**,typename_1...,typename_n]

The acceptable point-type-names are:

- ▷ bullet
- ▷ circle
- ▷ plus
- ▷ times
- ▷ asterisk
- ▷ box
- ▷ square
- ▷ triangle
- ▷ delta
- ▷ wedge
- ▷ nabla
- ▷ diamond
- ▷ lozenge

If there are more sets of points than objects in this list, they will be repeated sequentially.

Here's an example:

```
plot2d([x^2,x^3,x^4,[discrete,[[.1,.2],
          [.2,.3],[.3,.5],
[.4,.7],[.5,.11],[.6,.13],[.7,.17],
          [.8,.19],[.9,.23],
[1,.29]]],[x,-1,1],[style,lines,lines,
           lines,points],
[color,blue,red,green],[point_type,nabla])
```

This produces the plot in figure F.1.1 on the next page.

Another example: Here we plot two *implicit* functions, one *explicit* one (x^2) and a *discrete* function that is defined at a single point. Note that discrete functions expect a *list* of points even if there's only a *single point* in it:

```
plot2d([x=1,y=3,x^2,[discrete,[[1,3]]]],[x,-1,3],
[y,-1,10],[style,lines,lines,lines,points],
[point_type, asterisk]);
```

We get the plot in figure F.1.2 on the following page.

F.1.4. Slightly lower-level plot styles. These are slightly harder to use than the high-level options above. Their main (only?) advantage is that you can specify the *size* of each object. In this case, the styles are *lists* rather than simple names:

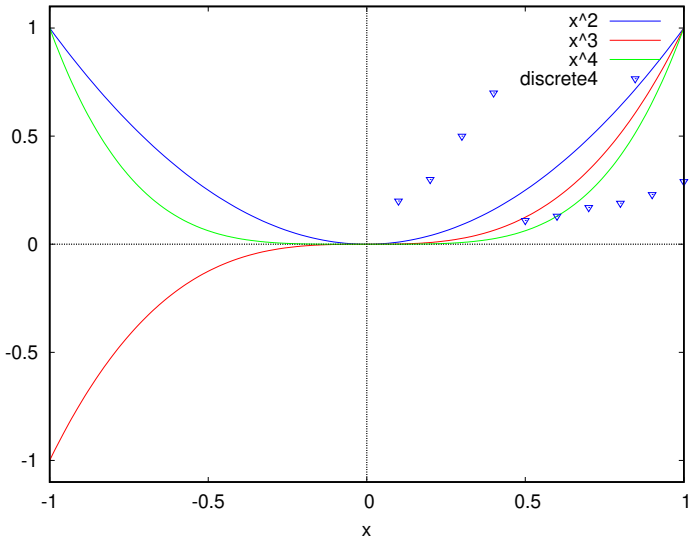


FIGURE F.1.1. High-level plot example

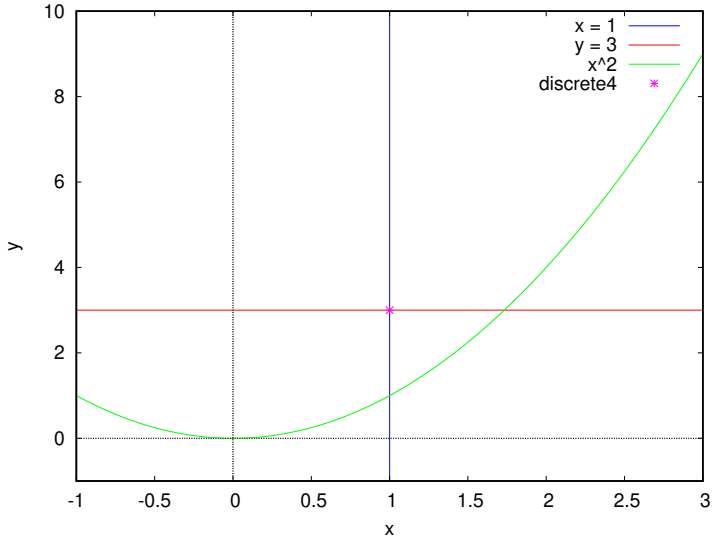


FIGURE F.1.2. Mixed plot-types

▷ [**lines**,line_width{,color}] — this is the default. Points are plotted and connected with lines.
▷ [**points**,radius_of_points{,color,type_of_point}] — plots isolated points. See table F.1.1 on the next page.

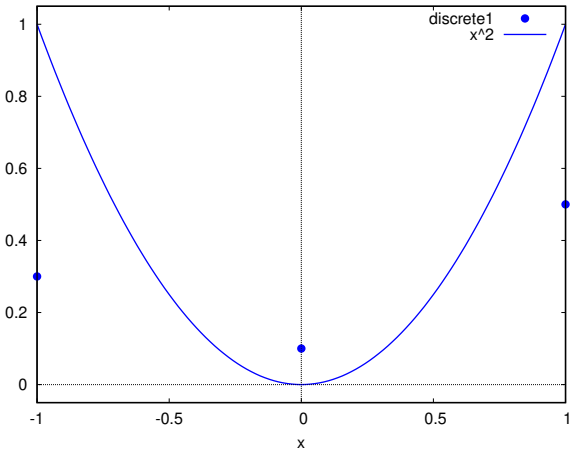


FIGURE F.1.3. Plot example 1

Code	Description	Code	Description	Code	Description
−1	None	4	Square	9	Filled up-triangle
0	Dot	5	Filled square	10	Down-triangle
1	Plus	6	Circle	11	Filled down-triangle
2	Multiply	7	Filled circle	12	Diamond
3	Asterisk	8	Up-triangle	13	Filled diamond

TABLE F.1.1. type_of_point codes

- ▷ [**linespoints**,line_width{,color}] — plots lines but also highlights the original points.
- ▷ [**dots**] — plots isolated tiny dots.

For example:

```
plot2d([[discrete,[[-1,.3],[0,.1],[1,.5]]],x^2],
    [x,-1,1],[style,[points,4,7,1],[lines,2,1]]);
```

produces the plot in figure F.1.3.

If we leave out the style command, the style defaults to '**lines**' and the plot looks like figure F.1.4 on the next page.

Colors (X11 term)

1: blue, 2: red, 3: magenta, 4: orange, 5: brown, 6: lime and 7: aqua

F.1.5. Other options. There are several other options that affect how the plot looks. These are all optional.

- ▷ [**nticks**,nn] — the number of initial points used to calculate the curve (the default is 10). GnuPlot starts evaluating the

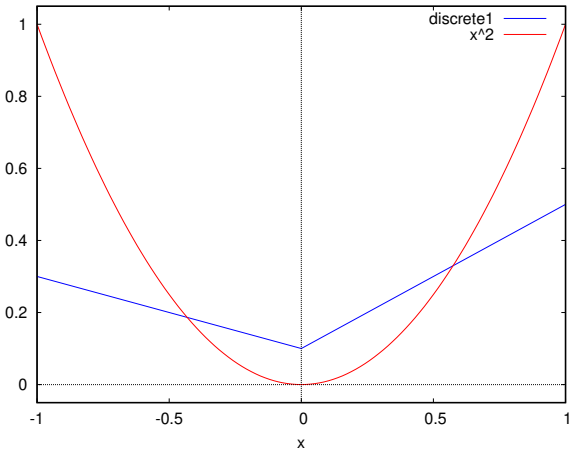


FIGURE F.1.4. Plot example 2

function to be plotted with this number of points and subdivides its intervals where the function changes rapidly, the goal being to produce a smooth plot.

- ▷ [**xlabel**,"text"] — label for the x-axis.
- ▷ [**ylabel**,"text"] — label for the y-axis.
- ▷ [**legend**,"text1",...,"textn"] — labels for n plots (must be in the same order as the plotted functions!).
- ▷ [**grid**,nx,ny] — number of grid-lines.
- ▷ [**yxratio**, nn] — defines the shape of the rectangle in which the plot is drawn.
- ▷ [**title**,"string"] — the title of the plot.
- ▷ **logx**,**logy** — not enclosed in square brackets. Causes logarithmic scales to be used.
- ▷ **same_xy** — causes the aspect ratio of the *plot* to be 1.

NOTE. Some editors uses "styled" curly brackets for the quote character. These are not recognized as quotes by wxMaxima (although they *display* as straight quotes in wxMaxima!) and can lead to mysterious errors. Of course, using wxMaxima as your text-editor produces the right kinds of quotes!

F.1.6. Parametric plots. Many geometric objects are defined parametrically. These can be plotted with **plot2d** and a list headed with the keyword **parametric**. The general form is

```
plot2d ([parametric, two-functions_of_parameter,
range_of_parameter]);
```

For instance, the command

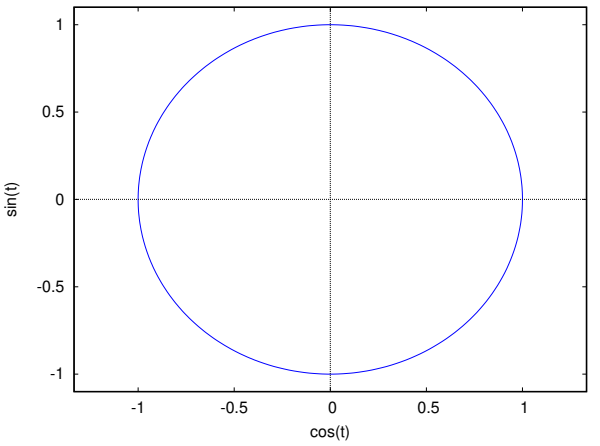


FIGURE F.1.5. Parametric plot 1

```
plot2d ([x^2+2, [parametric, cos(t), sin(t),
[t, -5, 5]]], [x, -3, 3]);
```

FIGURE F.1.6. Code for mixed parametric plot

```
plot2d ([parametric, cos(t), sin(t), [t,-%pi,%pi]],
[x, -4/3, 4/3]);
```

produces the plot in figure F.1.5.

Parametric plots can be mixed with other types. For instance, the code in figure F.1.6 produces the plot in figure F.1.7 on the following page.

F.1.7. Contour plots. This involves plotting contour lines for a function of two variables — these are like isobars on a weather map. The function to be plotted appears in a list whose first element is the keyword **contour**. Example:

```
plot2d ([contour, sin(y) * cos(x)^2],
[x, -4, 4], [y, -4, 4]);
```

which produces the plot in figure F.1.8 on the next page. This shows the level-sets for the function $f(x,y) = \sin(y)\cos(x)^2$ at $f = -0.5, 0, 0.5$.

F.2. plot3d

Many of the options for **plot2d** also apply to **plot3d**. Other options include **nomesh_lines**.

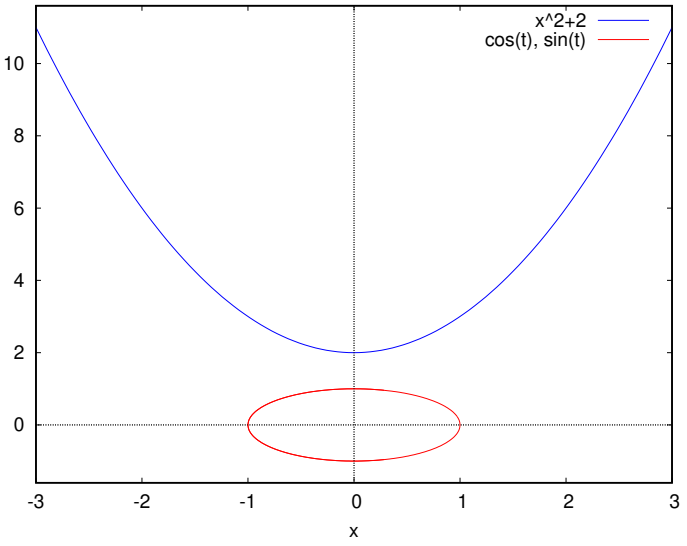


FIGURE F.1.7. Mixed parametric plot

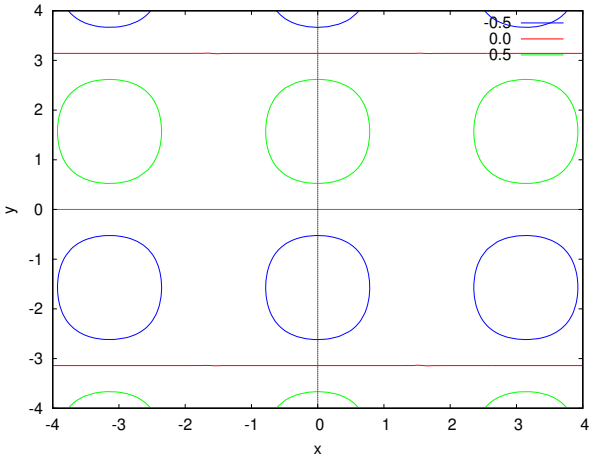


FIGURE F.1.8. Contour plot

F.2.1. Height plots. plot3d (*expr*, *x_range*, *y_range*, . . . , *options*,. . .)
Example:

```
plot3d (u^2 - v^2, [u, -2, 2], [v, -3, 3],
      [grid, 100, 100],
      nomesh_lines);
```

This produces the plot in figure F.2.1 on the facing page.

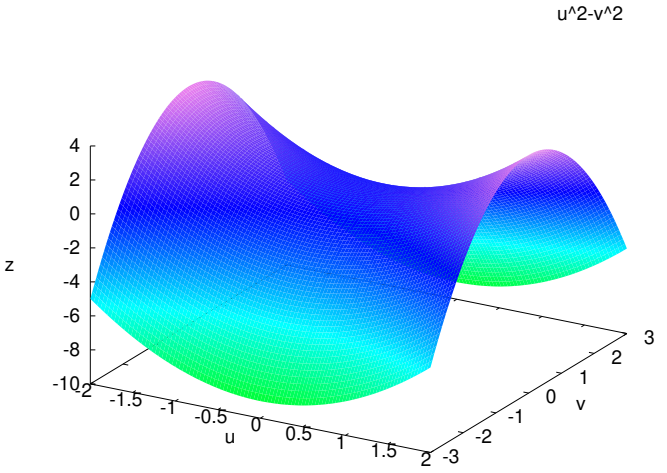


FIGURE F.2.1. A 3d plot

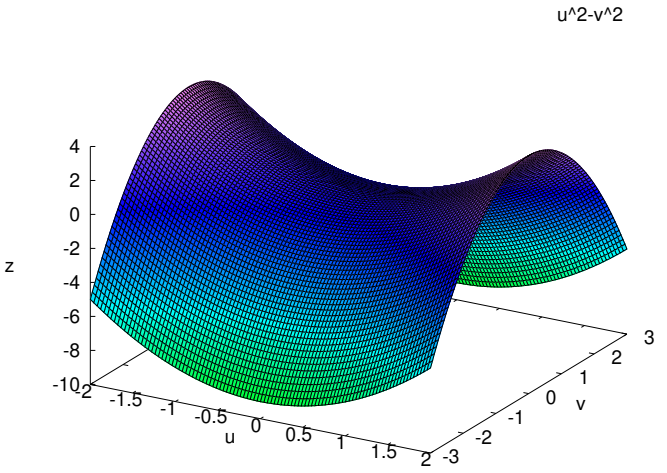


FIGURE F.2.2. A 3d plot with mesh

If we leave out the **nomesh_lines**, we get figure F.2.2 (so mesh lines are the default).

Other options:

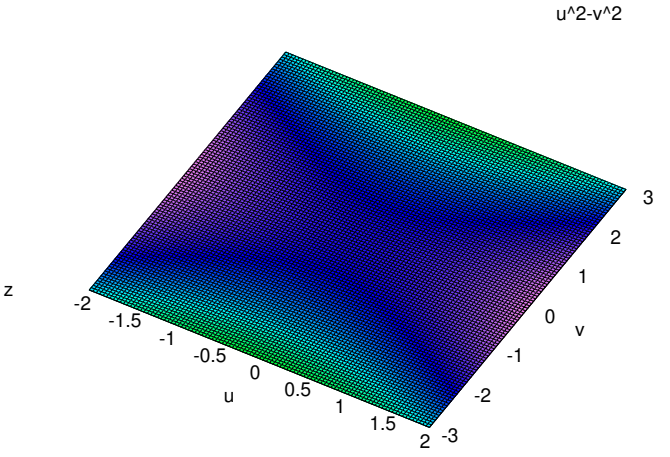


FIGURE F.2.3. Plot with elevation 0

```
plot3d (u^2 - v^2, [u, -2, 2], [v, -3, 3],
   [palette, [gradient, red, orange,
        yellow, green]],
   [grid, 100, 100]);
```

FIGURE F.2.4. Plot using palette

- ▷ **elevation** — the z-values are compressed or expanded to this range. Setting [**elevation**,0] smashes the three-dimensional plot to two dimensions. See figure F.2.3.
- ▷ **palette** — the set of colors to use in the plot. For instance, the command in figure F.2.4 produces the plot in figure F.2.5 on the facing page. The **gradient** option causes the colors to smoothly transition from low z-values (red) to high z-values (green). There is also a **color_bar** option that draws the palette and gives one an idea of what the colors mean. Adding that option gives the plot in figure F.2.6 on page 333.

F.2.2. Parametric plots

. **plot3d** ([x(u,v),y(u,v),z(u,v)],[u,umin,umax],[v,vmin,vmax],options...)

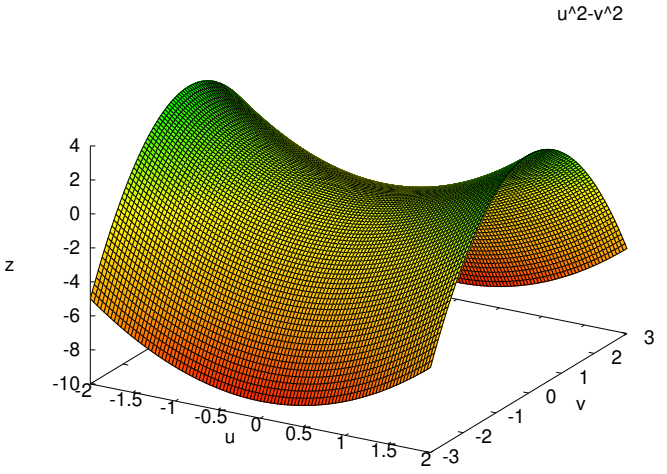


FIGURE F.2.5. Plot with a color palette

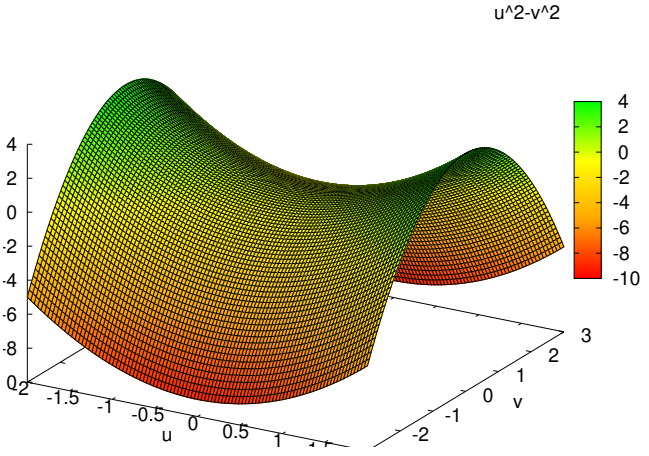


FIGURE F.2.6. Plot with a color-bar

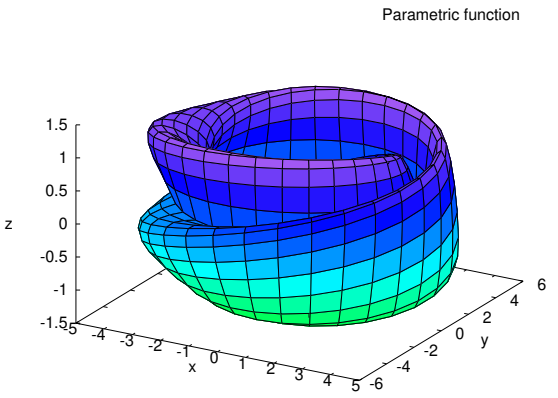


FIGURE F.2.7. A Klein Bottle

Example:

```
a:4;
x(u,v,a):=(a + cos(v/2) * sin(u) -
        sin(v/2) * sin(2*u)) * cos(v);
y(u,v,a):= (a + cos(v/2) * sin(u) -
        sin(v/2) * sin(2*u)) * sin(v);
z(u,v,a):= sin(v/2) * sin(u)
        + cos(v/2) * sin(2*u);
plot3d([x(u,v,a),y(u,v,a),z(u,v,a)],
        [u,-%pi,%pi],[v,-%pi,%pi]);
```

produces the plot in figure F.2.7

F.3. Standalone commands

We have two specialized plot commands that can to create graphic objects.

- ▷ **julia** (x, y, ...options...) — creates a Julia set for the complex number $x + iy$. Each pixel in the grid is given a color corresponding to the number of iterations it takes the sequence that starts at that point to move out of the convergence circle of radius 2 centered at the origin. The number of pixels in the grid is controlled by the **grid** plot option (default 30 by 30). The maximum number of iterations is set with the option **iterations**.
- ▷ **mandelbrot** (options). — Draws a Mandelbrot set with the same options as Julia.

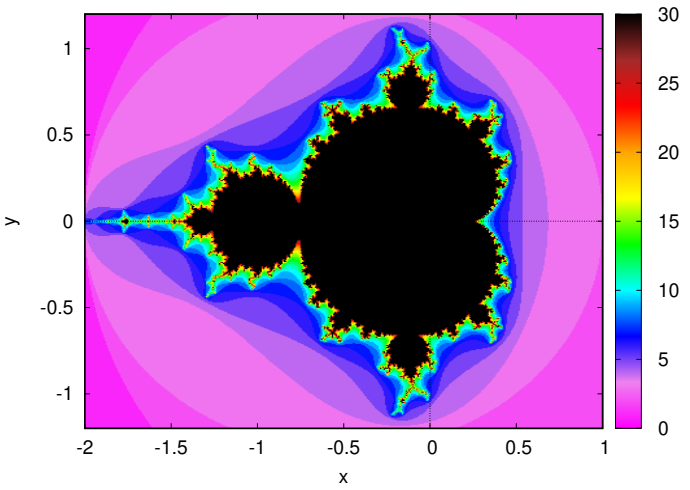


FIGURE F.3.1. The Mandelbrot set

For instance, the command

```
mandelbrot ([iterations, 30], [x, -2, 1],
            [y, -1.2, 1.2],
            [grid,400,400])
```

produces the plot in figure F.3.1.

F.4. Plot-outputs

Gnuplot generates plots on 'terminals'. If no terminal is specified, it defaults to 'x11term', the computer screen. Other terminals include:

- ▷ dumb — tries to do the plot in *ASCII-art!*
- ▷ ps — creates the plot in Postscript, suitable for printing. All of the plots in this book were done on the ps terminal.
- ▷ svg — Scalable Vector Graphics, best for displaying on web pages. These images can be easily resized without losing information (which is why they're called 'scalable').
- ▷ png — Portable Network Graphics, a bitmapped format for web pages.
- ▷ jpg — A lossy, compressed format, suitable for web pages (jpg files for an image are much smaller than png ones for the same image — sometimes by a factor of 100). This format uses two-dimensional, discrete Fourier transforms — see section 5.1 on page 81.
- ▷ pdf — Portable document format, also good for printing.

```
with_slider_draw(
        t, /*variable to attach to the slider */
        makelist(j,j,0,100), /*list of integers */
        explicit(psi_n(100,x,.01*t),x,-%pi,%pi),
        /*plot */
        yrange= [0,1.2] /*optional
            graphic command */
        ); /* end of with_slider_draw-command */
```

FIGURE F.5.1. The with_slider_draw command

```
draw(
    global_options,
    scene_1,
    scene_2,
    ...
    scene_n
);
```

FIGURE F.5.2. Basic **draw**-command

This is specified via the command:

[**gnuplot_term**,"terminal_name"]

One can also specify an output *file* in square brackets and quotes, like the terminal. The command:

[**gnuplot_out_file**,"filename"]

Example:

```
plot2d(x^2,[x,-5,5],[gnuplot_term, "svg"],
     [gnuplot_out_file,"zz.svg"])
```

F.5. The draw commands

The **draw**-library contains commands that are much more complex than those of **plot2d** and **plot3d**, but allow more complete access to gnuPlot's features. We have already seen one command of this library in section 4.5 on page 69, namely figure F.5.1

Although the **draw**-library is supposed to be loaded via the

load("draw")

command, this author has found that the draw-commands work without explicitly loading this library, suggesting that it is loaded by default.

The basic command appears in figure F.5.2.

Global options apply to all of the scenes and include:

▷ **terminal** — the type of output. The terminals in the draw-library are:

- **screen** (default) — the computer screen. Can, optionally, be coded as [**screen**,*nn*], where *nn* is a number. This allows multiple windows with plots to be opened at the same time.
- **png** — Portable Network Graphics. A bitmapped format.
- **pngcairo** — Portable Network Graphics using the Cairo library (if it's present). A bitmapped format that uses antialiasing to produce a clearer image.
- **jpg** — A compressed, lossy format.
- **gif** — A compressed format with a limited number of colors.
- **eps** — Encapsulated postscript in black and white (if your output is meant to be printed, it will probably be in black and white anyway). eps is like postscript except that the file has the size of the image specified so it can be displayed with a minimum of white space around it. The postscript files generated by the **plot** commands are actually eps.
- **eps_color** — Encapsulated postscript in all its colorful glory.
- **epslatex** — Encapsulated postscript and LaTeX code to insert it into a LaTeX document.
- **epslatex_standalone** — Encapsulated postscript and a LaTeX document to display it.
- **svg** — Scalable vector graphics. Ideal for web pages.
- **canvas** — Produces an html file with the graphics, using the Canvas and gnuPlot javascript libraries. Good for web pages, although most browsers support SVG.
- **dumb** — ASCII art!
- **dumb_file** — ASCII art in a *file!*
- **pdf** — produces a PDF document.
- **pdfcairo** — same as above but uses the Cairo library to produce an antialiased image.
- **wxt** — an alternate drawing library. On many systems, this is a synonym for **screen**.
- **animated_gif** — the gif format allows for animated images.
- **multipage_pdfcairo** — like **pdfcairo** but spanning multiple pages.
- **multipage_pdf** — like **pdf** but spanning multiple pages.
- **multipage_eps** — like **eps** but spanning multiple pages.

- **multipage_eps_color** — like **eps_color** but spanning multiple pages.
- **aquaterm** — aquaterm is a graphics terminal for the Macintosh running MacOS X. Can, optionally, be coded as [**aquaterm**,*nn*], where *nn* is a number. This allows multiple windows with plots to be opened at the same time.

▷ **file_name** — as mentioned in section F.4 on page 335, except that the draw library adds its own extension to the name, like 'pdf', or 'eps' so you don't have to.

▷ **columns** — each scene produces a plot, and multiple plots are normally stacked on top of each other. **columns**>1 allows multiple plots to appear side by side.

These are coded as option=value rather than in a list. Values that are not arbitrary character strings do *not* need to be quoted[1] unless they are used as variables in the rest of your program[2] in which case they can be quoted with a single, *single* quote like 'eps. Example:

```
draw(
    gr2d(
        key="sin (x)",grid=[2,2],
        explicit(
            sin(x),
            x,0,2*%pi
        )
    ),
    gr2d(
        title="zztop",
        key="cos (x)",grid=[2,2],
        explicit(
            cos(x),
            x,0,2*%pi
        )
    )
);
```

This produces two smaller plots, one on top of the other — see figure F.5.4 on the facing page. If we set the global option columns=2, and type the command in figure F.5.3 on the next page, we get the plots side-by-side and compressed *horizontally* in figure F.5.5 on the facing page.

Scenes may take one of two forms:

[1] They are predefined data-items.

[2] Generally a bad idea!

```
draw(
     columns=2,
     gr2d(
          key="sin (x)",grid=[2,2],
          explicit(
               sin(x),
               x,0,2*%pi
          )
     ),
     gr2d(
          title="zztop",
          key="cos (x)",grid=[2,2],
          explicit(
               cos(x),
               x,0,2*%pi
          )
     )
 );
```

FIGURE F.5.3. Two-column plot

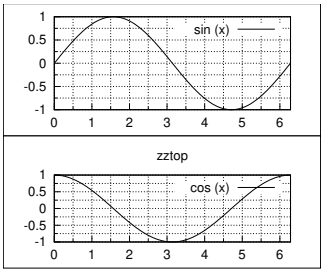


FIGURE F.5.4. Drawing two plots in one command

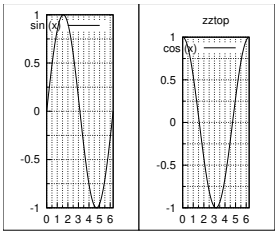


FIGURE F.5.5. Drawings with two columns

▷ **gr2d**(*options*) — for a two-dimensional scene. The available options (besides a function to be plotted) are

- **bars** ([x1,h1,w1], [x2,h2,w2, . . .]) — draws bars centered at values x1, x2,. . . with heights h1, h2,. . . and widths w1, w2, . . . Options: **color** (see table F.5.1 on page 343 for color-names).
- **ellipse** (xc, yc, a, b, ang1, ang2) — plots an ellipse centered at [xc, yc] with horizontal and vertical semi axis a and b, respectively, starting at angle ang1 and ending at angle ang2. Options: **transparent** (= **true** *or* **false**), **fill_color**, **border** (= **true** *or* **false**), **line_width**, **key**, **line_type** and **color** (see table F.5.1 on page 343 for color-names).
- **explicit**
- image
- **implicit**
- **key** — the draw-version of "legend." **key** = "*string*".
- **parametric** (xfun,yfun,par,parmin,parmax).
- **points** ([[x1,y1], [x2,y2],. . .]) — Options: **point_size**, **point_type**, **points_joined** (= **true** *or* **false**), **line_width**, **key**, **line_type** and **color** (see table F.5.1 on page 343 for color-names). Point_types: **bullet**, **circle**, **plus**, **times**, **asterisk**, **box**, **square**, **triangle**, **delta**, **wedge**, **nabla**, **diamond**, **lozenge**.
- **polar** (*radius,ang,minang,maxang*) Options **color**, **line_width**, (see table F.5.1 on page 343 for color-names).
- **polygon** ([x1, x2,. . .], [y1, y2, . . .]) *or* **polygon** ([[x1, y1], [x2, y2], . . .]) — Options **color**, **fill_color**, **line_width**, **border** (= **true** *or* **false**) (see table F.5.1 on page 343 for color-names).
- **quadrilateral**([x1, y1], [x2, y2], [x3, y3], [x4, y4]) — Options: **transparent** (= **true** *or* **false**), **fill_color**, **border** (= **true** *or* **false**), **line_width**, **key**, **xaxis_secondary**, **yaxis_secondary**, **line_type**, **transform** and **color** (see table F.5.1 on page 343 for color-names).
- **title** = "*string*".
- **rectangle** ([x1,y1], [x2,y2]) — The points are opposite vertices. Options: **transparent** (= **true** *or* **false**), **fill_color**, **border** (= **true** *or* **false**), **line_width**, **key**, **line_type** and **color** (see table F.5.1 on page 343 for color-names).
- **triangle** ([x1,y1], [x2,y2], [x3,y3]) — Options: **transparent** (= **true** *or* **false**), **fill_color**, **border** (= **true** *or* **false**),

```
draw(
    gr2d(
        explicit(
            sin(x),
            x,0,2*%pi
        )
        explicit(
            cos(x),
            x,0,2*%pi
        )
    )
);
```

FIGURE F.5.6. Multiple functions in the same scene

line_width, **key**, **line_type** and **color** (see table F.5.1 on page 343 for color-names).

- **vector**([x,y], [dx,dy]) — plots **vector** [dx,dy] with origin in [x,y]. Options: **head_both** (= **true** *or* **false**), **head_length**, **head_angle**, **head_type**, **line_width**, **line_type**, **key** and **color** (see table F.5.1 on page 343 for color-names).

The options **explicit** or **implicit** are required if *functions* are plotted. **plot2d** automatically decided whether a plot is explicit or implicit by how one codes it. The **draw**-library requires you to specify. Multiple functions can be plotted in the *same* scene (as the code in figure F.5.6 shows), in which case they will overlap each other (see figure 15.2.2 on page 282 for an example of this).

▷ **gr3d**(options) — for a three-dimensional scene. The available options are:

- **cylindrical**(radius, z, minz, maxz, azi, minazi, maxazi) — plots the function radius(z, azi) defined in cylindrical coordinates, with variable z taking values from minz to maxz and azimuth azi taking values from minazi to maxazi. Options: **xu_grid**, **yv_grid**, **line_type**, **key**, **wired_surface**, **enhanced3d** and **color** (see table F.5.1 on page 343 for color-names).
- **elevation_grid** (mat,x0,y0,width,height) — draws matrix mat in 3D space. z values are taken from mat, the abscissas range from x0 to x0 + width and ordinates from y0 to y0 + height. Element a(1,1) is projected on point (x0,y0+height), a(1,n) on (x0+width,y0+height), a(m,1) on (x0,y0), and a(m,n) on (x0+width,y0). Options: **line_type**, **line_width**, **key**, **wired_surface**,

enhanced3d and **color** (see table F.5.1 on the next page for color-names).

- **explicit**
- **implicit**
- label
- **mesh** (*row_1*,*row_2*,...) — Argument *row_i* is a list of n 3D points of the form [[x_i1,y_i1,z_i1], ...,[x_in,y_in,z_in]], and all rows are of equal length. All these points define an arbitrary surface in 3D. It's a generalization of the **elevation_grid** object. Options: **line_type**, **line_width**, **color**, **key**, **wired_surface**, **enhanced3d** and **transform** (see table F.5.1 on the facing page for color-names).
- parametric
- parametric_surface
- points
- quadrilateral
- spherical
- triangle
- **tube** (*xfun*,*yfun*,*zfun*,*rfun*,*p*,*pmin*,*pmax*). Draws a tube in 3D. (*xfun*,*yfun*,*zfun*,*rfun*) is the parametric curve with parameter *p* taking values from pmin to pmax. Circles of radius *rfun(p)* are placed with their centers on the parametric curve and perpendicular to it. Options: **xu_grid**, **yv_grid**, **line_type**, **line_width**, **key**, **wired_surface**, **enhanced3d**, **color** and **capping** (see table F.5.1 on the next page for color-names). The command

```
draw3d(
       enhanced3d = true,
       xu_grid =100,
       tube((3+cos(3*t))*cos(2*t),
         (3+cos(3*t))*sin(2*t),
         sin(3*t),.5,t,-%pi,%pi)
);
```

  produces the image of an overhand knot in figure F.5.7 on the facing page.
- **vector** ([x,y,z], [dx,dy,dz]) — Draws the vector [dx,dy,dz] starting from [x,y,z]. Options: **head_both**, **head_length**, **head_angle**, **head_type**, **line_width**, **line_type**, key and **color** (see table F.5.1 on the next page for color-names).

The options **explicit** or **implicit** are required if functions are plotted. **plot2d** automatically decided whether a plot is explicit or implicit by how one codes it. The **draw**-library requires you to specify.

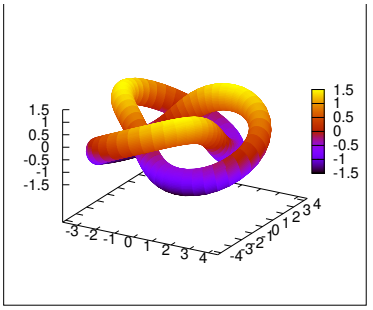


FIGURE F.5.7. The trefoil knot

white	black	gray0	grey0
gray10	grey10	gray20	grey20
gray30	grey30	gray40	grey40
gray50	grey50	gray60	grey60
gray70	grey70	gray80	grey80
gray90	grey90	gray100	grey100
gray	grey	light_gray	light_grey
dark_gray	dark_grey	red	light_red
dark_red	yellow	light_yellow	dark_yellow
green	light_green	dark_green	spring_green
forest_green	sea_green	blue	light_blue
royalblue	skyblue	cyan	light_cyan
dark_cyan	magenta	light_magenta	dark_magenta
turquoise	light_turquoise	dark_turquoise	pink
light_pink	dark_pink	coral	light_coral
orange_red	salmon	light_salmon	dark_salmon
aquamarine	khaki	dark_khaki	goldenrod
light_goldenrod	dark_goldenrod	gold	beige
brown	orange	dark_orange	violet
dark_violet	plum	purple	"x*hhhhhh*"

TABLE F.5.1. Colors in the **draw**-library

Within a scene, plots of functions (not all plots are of functions!) are either

- ▷ **explicit** — plots of functions that you list, or
- ▷ **implicit** — plots defined by *equations.*

As the examples in figures F.5.4 and F.5.5 on page 339 show, it is usually better to draw *one* scene at a time. The following abbreviations make this easier:

- ▷ **draw2d**(*stuff*)=**draw**(**gr2d**(*stuff*)) Its options (including global ones) are identical to those of **gr2d**.

▷ **draw3d**(*stuff*)=**draw**(**gr3d**(*stuff*)) Its options (including global ones) are identical to those of **gr2d**.

APPENDIX G

Graph-theoretic commands

All of these commands are implemented in the graph-theoretic library, loaded via

load(graphs)

G.1. Graph-creation and display

We create graphs via the commands

▷ **create_graph**(v_list, e_list) creates a graph data-structure with vertices equal to the list v_list and edges equal to e_list, where each is a list [v1,v2], where v1 and v2 are the end-vertices of the edge. A *weighted edge* is a list [[v1,v2],w] where w is the *weight* of the edge. For example, we could have

```
z:create_graph([0,1], [[0,1]])
```

to create a graph with two vertices connected by one edge. Having done so, we could print it out with the command

```
print_graph(z)
```

which would produce

```
Graph on 2 vertices with 1 edges.
Adjacencies:
  0 :  1
  1 :  0
```

or

```
draw_graph(z,show_id=true)
```

which produces the image in figure G.1.1 on page 347. 'show_id' causes the identities of the vertices to be displayed. This command takes other options, like

- **vertex_size** (in pixels)
- **vertex_color**
- **edge_width**
- **edge_color**
- **show_edges**
- **show_edge_width**

- **show_edge_color**
- **show_vertices**=v_list: display vertices in the list v_list using a different color
- **show_vertex_type**=type: defines how vertices in **show_vertices** are displayed.
- **show_vertex_size**=size: the size of vertices in **show_vertices**.
- **show_vertex_color**=c: color used for displaying vertices in the **show_vertices** list.
- **show_label**=**true** — displays the *labels* attached to vertices in a graph. See page 346. Note: If **show_id** and **show_label** are both **true**, **show_id** takes precedence.
- **head_angle**=angle: the angle for the arrows displayed on arcs (in directed graphs). Default value: 15.
- **head_length**=len: the length for the arrows displayed on arcs (in directed graphs). Default value: 0.1.
- **terminal**
- **file_name**
- **program** — by default, wxMaxima uses a program called 'spring_embedding' to format graphs. There's the option of using other programs, especially those of the free graphviz package (which must be installed separately). The programs in this package include 'dot', 'neato', and 'twopi', which are especially suited for very large and complex graphs. The graphs in figure 9.1.2 on page 172 and 7.4.1 on page 134 were generated using 'dot'.
- **redraw**=**true** — normally, when a graph is redrawn, wxMaxima saves the locations of the vertices so additional graphs will visually correspond to the original graph (useful in indicating the shortest path in figure 9.2.2 on page 181, for instance). The option, **redraw**=**true**, causes each redraw of the graph to start from scratch.

See the section on the **draw**-commands — F.5 on page 336 for the other options.

We also have a variant of **create_graph**:

```
create_graph(n, [edges])
```

where vertices are numbered from 0 to $n-1$. In both instances of **create_graph**, we have the option of specifying **directed**=**true**, which creates a *directed graph* (= a *digraph*), in which each edge has a *direction*. You can also attach *labels* to vertices of a graph:

```
g:create_graph([[0,"Zero"], [1, "One"]], [[0,1]])
```

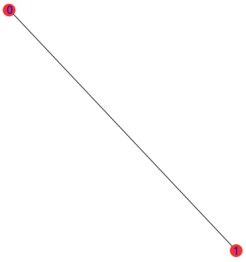


FIGURE G.1.1. A *very* simple graph

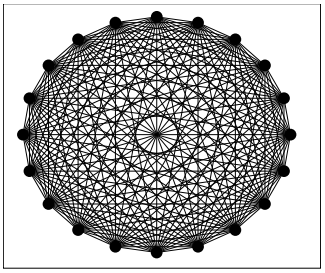


FIGURE G.1.2. Complete graph on 20 vertices

▷ An especially powerful command is

```
make_graph(n, f)
```

where f is a *predicate function,* a function that takes a pair of vertices as parameters and returns **true** or **false**. Every pair of vertices that causes f to return **true** gets an edge connecting them, and these are the *only* edges. The vertices on this graph are numbered from 1 to n. We have already seen this in example 9.2.1 on page 179. If '**directed**=**true**' in the command, it creates a digraph.

▷ **empty_graph**(n) — creates a graph with vertices 0 through n-1 and *no* edges.

▷ **complete_graph**(n) — creates a graph with vertices 0 through n-1 and edges between *every* pair of vertices. For instance

```
z:complete_graph(20);
draw_graph(z)
```

produces figure G.1.2.

▷ **complete_bipartite_graph**(m,n) — creates a complete *bipartite graph* with m and n vertices. A bipartite graph is one whose vertices can be divided into two sets in such a way

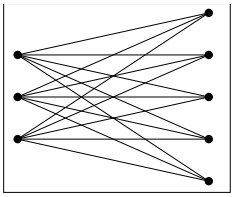


FIGURE G.1.3. Complete bipartite graph (3,5)

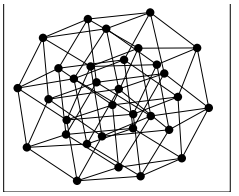


FIGURE G.1.4. 5-dimensional cube

that all the edges connect vertices in one set to vertices in the other (and there are no edges *within* either of these sets).

For instance, the commands

```
z:complete_bipartite_graph(3,5);
draw_graph(z)
```

produce the graph in figure G.1.3.

▷ **cube_graph**(n) — produces a graph representing an n-dimensional cube.

The commands

```
z:cube_graph(5);
draw_graph(z)
```

produce the graph in figure G.1.4.

▷ **circulant_graph**$(n,[a_1,\ldots,a_k])$ — generates a graph with vertices $0,\ldots,n-1$ where vertex i is connected to the $2k$ vertices.

$$i \pm a_1 \bmod n,\ldots,i \pm a_k \bmod n$$

for all $0 \leq i < n$.

The commands

```
z:circulant_graph(7,[2,3]);
draw_graph(z)
```

produce the graph in figure G.1.5 on the facing page.

▷ **cycle_graph**(n) — returns the cycle on n vertices.

▷ **cycle_digraph**(n) — returns the directed cycle on n vertices.

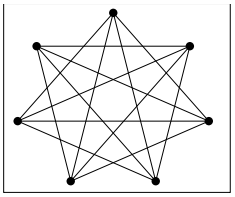


FIGURE G.1.5. Circulant graph

- ▷ **empty_graph**(n) —returns the graph with n vertices and *no* edges.
- ▷ **grid_graph**(n, m) — returns an $n \times m$ grid.
- ▷ **path_graph**(n) — returns a graph that consists of a single path on n vertices.
- ▷ **path_digraph**(n) — returns a graph that consists of a single directed path on n vertices.
- ▷ **random_graph**(n, p) — returns a random graph on n vertices where each edge appears with probability p.
- ▷ **random_digraph**(n, p) — returns a random directed graph on n vertices where each edge appears with probability p.
- ▷ **random_bipartite_graph**(a, b, p) — returns a random bipartite graph on a vertices and b vertices where each edge appears with probability p.
- ▷ **random_regular_graph**(n, d) — a random graph on n vertices where each vertex has degree d. If d is omitted, it is assumed to be 3.
- ▷ **random_graph1**(n, m) — create a graph on n vertices and m random edges.
- ▷ **random_network**(n, p, w) — returns a random network on $n + 2$ vertices where edges appear with probability p and are weighted with random weights between 0 and w. Think of a network as one of water pipes where the capacity of each pipe is given by its *weight.* One vertex is designated as the *source,* where the water enters the network, and another is the *sink,* where water exits it. The command

```
[net,source,sink]:random_network(n,p,w)
```

returns a list, where net is the weighted digraph defining the network, and source and sink are the corresponding vertex-numbers. Example:

```
[net,source,sink]:random_network(10,.2,5)
```

produces

[DIGRAPH(12 vertices, 32 arcs),10,11]

and

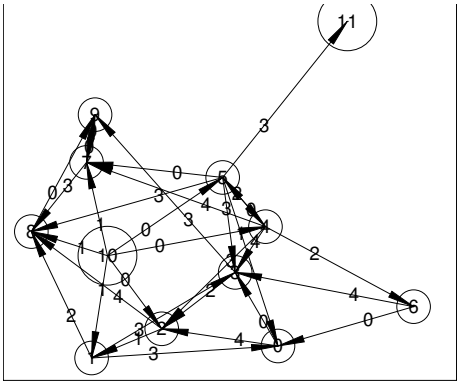


FIGURE G.1.6. Random network

```
draw_graph(net,show_id=true,
     show_weight=true,
     show_vertices=[10,11],
     show_vertex_size=7,
     head_size=.005,
     head_angle=7,
     show_vertex_type=circle,
     vertex_type=circle,vertex_size=4);
```

produces figure G.1.6.

▷ **random_tournament**(n) — returns a random tournament on n vertices. A *tournament* is a complete graph with directed edges. Think of it as representing a round-robin tournament where each vertex is a participant, and a directed edge points from the *winner* to the *loser* of the competition.

▷ **random_tree**(n) — a *tree* is a graph with no cycles. For instance:

```
t:random_tree(10);
```

generates a random tree, and the command

```
draw_graph(t,show_id=true,
     vertex_type=circle,vertex_size=4);
```

produces figure G.1.7 on the facing page.

▷ **underlying_graph**(g) — if g is a digraph, it returns the corresponding undirected graph.

▷ **wheel_graph**(n) — returns the "wagon wheel" shaped graph with n vertices on the outer rim and one extra vertex for the wheel's hub.

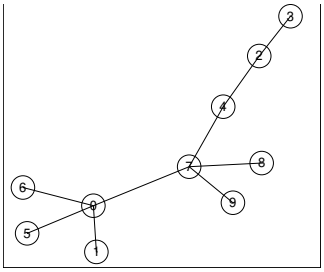


FIGURE G.1.7. Random tree

G.2. Operations with graphs

- ▷ **add_edge**(e, gr) — Adds the edge e to the graph gr. Related command: **add_edges**(e_list, gr) does the corresponding thing with a list of edges.
- ▷ **add_vertex**(v, gr) — does what the name implies: it adds vertex v to graph gr. Related command: **add_vertices**(list,gr) does the corresponding thing with a *list* of vertices.
- ▷ **clear_edge_weight**(e, g) — removes the weight from edge e in graph g. Returns the edge-weight that was cleared.
- ▷ **clear_vertex_label**(v, g) — removes the label from vertex v in graph.
- ▷ **degree_sequence**(g) — r returns a list of the degrees of the vertices of g.
- ▷ **edge_coloring**(g) — returns an optimal edge-coloring of the graph g in the format

  [number_colors,[[edge1,color1],[edge2,color2],...]]
- ▷ **get_edge_weight**(e, g) — returns the weight of the edge e in the graph g. The edge is represented by a list of two vertices.
- ▷ **get_vertex_label**(v, g) — does what its name implies.
- ▷ **copy_graph**(g) — does what its name implies.
- ▷ **complement_graph**(g) — returns the complement of g. Every graph is a subgraph of the complete graph on its set of vertices. The complement is what is left after g becomes deducted from the complete graph.
- ▷ **contract_edge**(e, g) — collapses edge e in graph g, identifying its endpoints.
- ▷ **remove_edge**(e, g) — removes edge e from graph g.
- ▷ **remove_vertex**(v, g) — removes vertex v from gr and all incident edges.
- ▷ **graph_product**(g_1, g_2) — forms the product of the two graphs.
- ▷ **set_edge_weight**(e, w, g) — sets the weight of edge e in graph g to w.

▷ **set_vertex_label**(v, l, g) — sets the label of vertex v in graph g to l.

G.3. Graph properties

▷ **adjacency_matrix**(g) — returns the adjacency matrix of g. If g has n vertices, this is an $n \times n$ matrix, A, where $A_{i,j} = 1$ if an edge connects vertex i with vertex j and 0 otherwise.

▷ **average_degree** (g) — does what its name implies.

▷ **edges**(g) — returns a list of the edges of g.

▷ **girth**(g) — the length of the shortest cycle in g.

▷ **biconnected_components**(g) — returns lists of vertices that span the maximal biconnected subgraphs of g. An *undirected* graph is *biconnecte* if it remains connected after any one vertex is deleted. A *directed* graph is biconnected if any two vertices can be connected by two paths that do not share any vertices except the endpoints.

▷ **bipartition**(g) — attempts to partition the vertices of g into two groups so that all edges connect vertices in one group to vertices in the other. If this is impossible (so the graph is not naturally bipartite), it returns an error.

▷ **chromatic_number**(g) — the minimum number of colors needed to color the vertices of g so that no two adjacent vertices are the same color. There's the famous *Four Color Problem*[1] that says

> "The chromatic number of a *planar loopless* graph is 4"

or, phrased differently,

> "A map with finite-length boundaries can be colored with only four colors"

see [71] for the fascinating history behind this problem.

▷ **chromatic_index**(g) — the minimum number of colors needed to color the *edges* of g so that no two *edges* incident on the same vertex are the same color. This is the chromatic number of the *line-graph* of g.

▷ **connected_components**(g) — a graph is *connected* if there exists paths connecting every vertex to every other vertex. This command returns the maximal connected subgraphs of g.

▷ **graph_charpoly**(g, x) — returns the characteristic polynomial — see definition 7.4.1 on page 124 — of the *adjacency matrix* of the graph g in the variable x.

▷ **graph_eigenvalues**(g) — returns the eigenvalues[2] — see section 7.4 on page 124 — of the *adjacency matrix* of the graph g.

[1] That took more than 100 years to solve!

[2] Since the adjacency matrix is real-symmetric, the eigenvalues will all be real — see corollary 6.2.91 of [58].

- ▷ **induced_subgraph**(V, g) — is g is a graph and V is a subset of its vertices, this produces the graph whose vertices are V and whose edges are those of g connecting vertices in V.
- ▷ **is_biconnected**(g) — An *undirected* graph is *biconnected* if it remains connected after any one vertex is deleted. A *directed* graph is biconnected if any two vertices can be connected by two paths that do not share any vertices except the endpoints.
- ▷ **is_bipartite**(g) — a *bipartite graph* is one whose vertices can be divided into two groups such that every edge in g connects a vertex in one group to one in the other. In other words, this is a graph whose chromatic number is 2.
- ▷ **is_digraph**(g) — self-explanatory.
- ▷ **is_connected**(g) — a graph is *connected* if there exists paths connecting every vertex to every other vertex.
- ▷ **is_edge_in_graph**(e, g) — self-explanatory.
- ▷ **is_graph**(g) — self-explanatory.
- ▷ **is_graph_or_digraph**(g) — self-explanatory.
- ▷ **is_isomorphic**(g_1, g_2) —
- ▷ **line_graph**(g) — returns the line-graph of the graph, g. This is a graph whose vertices correspond to the *edges* of g and two vertices are incident if the corresponding edges in g have a common vertex.
- ▷ **max_independent_set**(g) — It's the *maximum* set of vertices such that no vertex in the set is adjacent to any other.
- ▷ **max_matching**(g) — returns a maximum matching of the graph g. A *matching* of a graph is a selection of edges, no two of which are incident on the same vertex. It's called a matching because each edge pairs up (i.e. "matches") its endpoints. This is a maximum independent set of the line-graph of g. Example:

```
z:circulant_graph(7,[2,3]);
m:max_matching(z);
```

produces

$$[[2,6],[1,5],[0,4]]$$

and

```
draw_graph(z,show_id=true,
    show_edges=m,
    show_edge_width=7,
    vertex_type=circle,vertex_size=4);
```

produces figure G.3.1 on the next page.

- ▷ **hamilton_cycle**(g) — returns a Hamiltonian cycle of g if one exists and the empty list otherwise. A *Hamilton cycle* is one that includes each vertex exactly once.

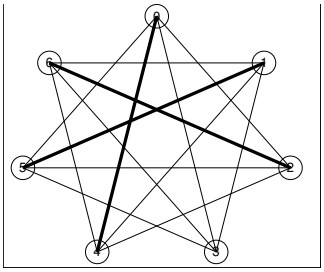


FIGURE G.3.1. Maximum matching

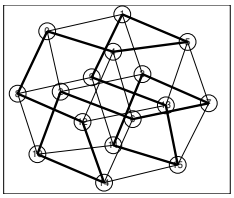


FIGURE G.3.2. A Hamilton cycle

```
f:cube_graph(4);
h:hamilton_cycle(f);
```

returns

$$[15, 11, 3, 7, 6, 2, 10, 14, 12, 8, 0, 4, 5, 1, 9, 13, 15]$$

and

```
draw_graph(f,show_id=true,
    show_edges=vertices_to_path(h),
    show_edge_width=7,
    vertex_type=circle,vertex_size=4);
```

produces figure G.3.2.

- ▷ **is_planar**(g) — a graph is *planar* if it can be embedded in a plane without edges crossing each other.
- ▷ **is_vertex_in_graph**(v, g) — self-explanatory.
- ▷ **is_tree**(g) — a *tree* is a graph without cycles.
- ▷ **laplacian_matrix**(g) — If graph g has n vertices, then the Laplacian, L, is given by $L = D - A$, where D is an $n \times n$ diagonal matrix whose diagonal entries are the *degrees* of the vertices of g and A is the *adjacency matrix* of g.
- ▷ **max_clique**(g) — a *clique* in a graph is a subgraph with the property that every vertex in it is connected to every other vertex in it. In other words the clique is a complete graph on

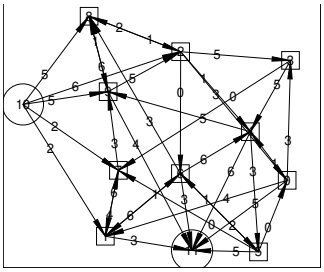


FIGURE G.3.3. Network

its vertices. The maximum clique is the one with the largest number of vertices.

▷ **max_degree**(g) — the maximum number of edges incident on any vertex.

▷ **max_flow**(n, s, t) — if n is a network (weighted, directed graph) and s is a source vertex and t is a sink vertex, this gives the maximum flow possible from s to t through the network, where the flow through each edge is given by its weight. Given the network in figure G.3.3, the command

```
max_flow(net,10,11)
```

produces the output [flow,list_of_edges] where each edge is listed with the flow through it. In this case, the flow is 17.

▷ **min_degree**(g) — the minimum number of edges incident on any vertex.

▷ **min_edge_cut**(g) — the minimum number of edges whose removal will disconnect the graph g.

▷ **min_vertex_cut**(g) — the minimum number of vertices whose removal will disconnect the graph g.

▷ **min_vertex_cover**(g) — a *vertex cover* of a graph is a set of vertices that includes at least one endpoint of every edge of the graph. A *minimum* vertex cover is the smallest such set.

▷ **minimum_spanning_tree**(g) — returns a minimum spanning tree of g — see definition 9.3.1 on page 185.

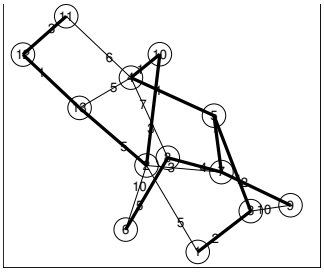


FIGURE G.3.4. Minimal spanning tree

For instance, the command

```
roads:create_graph([1,2,3,4,5,6,7,8,
                    9,10,11,12,13],
[[[1,2],5],[[1,3],2],[[3,9],10],[[2,10],3],
[[ 3,5],1],[[2,13],5],[[2,7],3],[[4,8],7],
[[5,7],1],[[4,10],1],[[7,8],4],[[6,8],5],
[[2,6],10],[[4,5],1],[[7,9],2],
[[4,11],6],[[11,12],3],[[4,13],5],
                  [[12,13],1]  ]
);
```

creates a weighted graph, and

```
t:minimum_spanning_tree(roads);
```

Then

```
draw_graph(roads,show_id=true,
    show_weight=true,
    show_edges=edges(t),
    show_edge_width=7,
    vertex_type=circle,vertex_size=4);
```

produces figure G.3.4.

- ▷ **neighbors**(v, g) — the neighbors of vertex v in graph g.
- ▷ **odd_girth**(g) — returns the length of the shortest cycle of *odd* length.
- ▷ **out_neighbors**(v, g) — returns a list of the vertices of a digraph, g, that lie at the ends of edges pointing *away* from vertex v.
- ▷ **radius**(g) — returns the minimum eccentricity of any vertex in graph g. See the **vertex_eccentricity**-command.
- ▷ **planar_embedding**(g) — when a graph is embedded in the plane (with no edges crossing each other), it has cycles that

subdivide the plane into disjoint regions. This command returns a list of these cycles. Note: The graph g must be biconnected. It g is not planar, this command returns **false**.

- ▷ **shortest_path**(u, v, g) — the shortest (by number of edges) path from vertex u to vertex v in graph g.
- ▷ **shortest_weighted_path**(u, v, g) — the shortest (by total weight of its edges) path from vertex u to vertex v in the weighted graph g. If no path exists (i.e., in a directed graph) the total weight is listed as ∞.
- ▷ **topological_sort**(g) — gives a topological sort of the directed acyclic graph, g. Suppose the vertices of g represent tasks, and whenever we have

Task A must be completed before Task B can begin. A *topological sort* of g lists the tasks in the order they must be performed (from left to right), "linearizing" the graph of dependencies. For instance:

```
g:create_graph(
          [1,2,3,4,5],
          [
           [1,2], [2,5], [5,3],
           [5,4], [3,4], [1,3]
          ],
          directed=true)
```

creates the directed graph in figure G.3.5 on the following page, and the command

```
topological_sort(g);
```

results in

$$[1, 2, 5, 3, 4]$$

If g is not acyclic[3], this command returns an empty list.

- ▷ **vertex_coloring**(g) — returns an optimal coloring of the vertices of g in the form

  [number_of_colors,[[v1,c1],[v2,c2],...]]

- ▷ **vertex_eccentricity**(v, g) — returns the *maximum distance* of vertex v from any other vertex in graph g. Distance is defined to be the length of the shortest path.
- ▷ **vertex_in_degree**(v, g) — number of edges pointing *toward* vertex v in digraph g.

[3] So some tasks could not be begun until after they were completed!

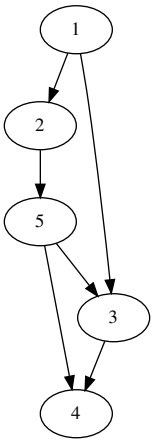


FIGURE G.3.5. A directed acyclic graph

- ▷ **vertex_out_degree**(v, g) — number of edges pointing *away from* vertex v in digraph g.
- ▷ **vertices**(g) — returns a list of the vertices of g.
- ▷ **wiener_index**(g) — returns the *Wiener Index* of the graph g. This is the sum of distances between *all* (not just adjacent) pairs of vertices in g. This is used in chemical graph theory to distinguish isomers of molecules.

G.4. Special graphs

These are graphs with special theoretic properties.

- ▷ **clebsch_graph**() — In the mathematical field of graph theory, the Clebsch graph is either of two complementary graphs on 16 vertices, a 5-regular graph with 40 edges and a 10-regular graph with 80 edges. The 80-edge graph is the dimension-5 halved cube graph; it was called the Clebsch graph by Seidel (1968) because of its relation to the configuration of 16 lines on the quartic surface discovered in 1868 by the German mathematician Alfred Clebsch.
- ▷ **dodecahedron_graph**() — the Platonic graph corresponding to the connectivity of the vertices of a dodecahedron.
- ▷ **flower_snark**(n) — returns the flower snark on $4n$ vertices and $6n$ edges.
- ▷ **frucht_graph**() — the Frucht graph is a cubic graph with 12 vertices, 18 edges, and no nontrivial symmetries (that is, every vertex can be distinguished topologically from every other vertex). It was first described by Robert Frucht in 1939.
- ▷ **grotzch_graph**() — the Grötzsch graph is a triangle-free graph with 11 vertices, 20 edges, chromatic number 4, and

crossing number 5. It is named after German mathematician Herbert Grötzsch, who used it as an example in connection with his 1959 theorem that planar triangle-free graphs are 3-colorable.

- ▷ **heawood_graph**() — This graph is cubic, and all cycles in the graph have six or more edges. Every smaller cubic graph has shorter cycles, so this graph is the smallest cubic graph of girth 6.
- ▷ **icosahedron_graph**() — the Platonic graph corresponding to the connectivity of the vertices of a icosahedron.
- ▷ **mycielski_graph**(g) — given an undirected graph, g, this returns the Mycielskian, $\mu(g)$, of g. The construction preserves the property of being triangle-free but increases the chromatic number. By applying the construction repeatedly to a triangle-free starting graph, Mycielski showed that there exist triangle-free graphs with arbitrarily large chromatic number.
- ▷ **petersen_graph**(n, d) — Returns the Petersen graph, $P_{n,d}$, formed by connecting the vertices of a regular n-gon to the corresponding vertices of a star polygon with Schläfli symbol $\{n/d\}$. **petersen_graph**() uses the default values $n = 5, d = 2$.
- ▷ **tutte_graph()** — the Tutte graph is a 3-regular graph with 46 vertices and 69 edges named after W. T. Tutte. It has chromatic number 3, chromatic index 3, girth 4 and diameter 8. The Tutte graph is a cubic polyhedral graph, but is non-hamiltonian. Therefore, it is a counterexample to Tait's conjecture that every 3-regular polyhedron has a Hamiltonian cycle.

G.5. Input/Output of graphs

The dimacs export and import algorithms handle the widest variety of graphs and digraphs, but the graph6 and sparse6 algorithms produce much more compact results.

- ▷ **dimacs_export**(g, f) — writes the graph, digraph, or weighted digraph g to the file, f, in the dimacs format. The result is a text file. The filename f can be optionally followed by a sequence of comma-separated comment-strings that will be recorded in the file.
- ▷ **dimacs_import**(f) — returns the graph, digraph, or network in file f.
- ▷ **graph6_decode**(s) — decodes the string s and returns the undirected graph.
- ▷ **graph6_encode**(g) — returns a string that encodes the undirected graph g.

- ▷ **graph6_export**(L, f) — converts the graphs in the list L into the graph6 format and writes them to file f.
- ▷ **graph6_import**(f) — returns a list of graphs imported from the file f.
- ▷ **sparse6_decode**(s) — decodes the string s and returns the undirected graph.
- ▷ **sparse6_encode**(g) — returns a string that encodes the undirected graph g.
- ▷ **sparse6_export**(L, f) — converts the graphs in the list L into the graph6 format and writes them to file f.
- ▷ **sparse6_import**(f) — returns a list of graphs imported from the file f.

Solutions to Selected Exercises

Chapter 1, 1.1 Exercise 1 (p. 7) The number of 5-card sets one can form from a 52-card deck is **binomial**(52,5), which is 2598960.

Chapter 1, 1.1 Exercise 2 (p. 7) The command **rectform**(2/(3+%i)); gives

$$\frac{3}{5} - \frac{i}{5}$$

Chapter 1, 1.1 Exercise 3 (p. 7) The command **rectform**(3*%i+1/(1−%i)); gives

$$\frac{1}{2} + \frac{7i}{2}$$

Chapter 1, 1.1 Exercise 4 (p. 7) According to the Binomial Theorem

$$\begin{aligned}\cos n\theta + i\sin n\theta &= (\cos\theta + i\sin\theta)^n \\ &= \cos^n\theta + n\cos^{n-1}\theta\cdot i\sin\theta + \frac{n(n-1)}{2!}\cos^{n-2}\theta\cdot i^2\sin^2\theta + \cdots \\ &\qquad + n\cos\theta\cdot i^{n-1}\sin^{n-1}\theta + i^n\sin^n\theta\end{aligned}$$

so the real part of this equation gives

$$\cos n\theta = \cos^n\theta - \frac{n(n-1)}{2!}\cos^{n-2}\theta\sin^2\theta + \frac{n(n-1)(n-2)(n-3)}{4!}\cos^{n-4}\theta\sin^4\theta + \cdots$$

and the imaginary part gives

$$\sin n\theta = n\cos^{n-1}\theta\sin\theta - \frac{n(n-1)(n-2)}{3!}\cos^{n-3}\theta\sin^3\theta + \cdots$$

Chapter 2, 2.1 Exercise 1 (p. 15) Proposition 2.1.6 on page 12 shows that $\gcd(n,m) = 1$ if and only if n and m have no primes in common in their prime-power factorizations.

Chapter 2, 2.1 Exercise 2 (p. 15) Define a function $f\colon \mathbb{Z}_{n\cdot m}^{\times} \to \mathbb{Z}_n^{\times} \times \mathbb{Z}_m^{\times}$ by mapping $x \in \mathbb{Z}_{n\cdot m}^{\times}$ to $(x \bmod n, x \bmod m) \in \mathbb{Z}_n^{\times} \times \mathbb{Z}_m^{\times}$. The Chinese Remainder Theorem implies that this is 1-1.

Chapter 2, 2.2 Exercise 1 (p. 16) Because $\phi(100) = 40$ (use the **totient**-command), and $40|1000$, so $7^{1000} \equiv 1 \pmod{100}$.

Chapter 2, 2.3 Exercise 1 (p. 17) We write

```
s:integer_partitions(100,2);
 xprimep(x) := integerp(x)
        and (x > 1) and primep(x);
 subset (s,
      lambda ([x], every (xprimep, x)));
```

and we get

$$\{[53,47],[59,41],[71,29],[83,17],[89,11],[97,3]\}$$

so there are 6 ways to write 100 as a sum of two primes.

Chapter 3, 3.1 Exercise 1 (p. 31) If we set r:**allroots**(x^5+2*x-5); we get equations 3.1.2 on page 30. Now do

map(**rhs**,r) or ('**rhs**,r) and you will get

[1.208917813386895,
0.9409544200647337*%i-1.167042002184508,
-0.9409544200647337*%i-1.167042002184508,
1.234436184384533*%i+0.5625830954910601,
0.5625830954910601-1.234436184384533*%i]

with *no* x=.

Chapter 3, 3.2 Exercise 1 (p. 40) This is easily coded:

```
c(n):=block(
          [],
          if equal(n,1) then return(1),
          if evenp(n) then return (n/2),
          3*n+1
         );
```

Chapter 3, 3.4 Exercise 1 (p. 43) The implicit equation is

$$x^2+2yx^2+2y^2x^2+yx-y^2x-y=0$$

Chapter 3, 3.4 Exercise 2 (p. 43) The resultant is

$$r=4y^2x^2-x^2+y^2$$

so the implicit equation is $r=0$.

Chapter 3, 3.4 Exercise 3 (p. 43) The implicit equation is

$$-2y+1-2x-2yx^2+x^2=0$$

Chapter 3, 3.4 Exercise 4 (p. 43) The resultant in question is

$$x^4+2x^3+x^2-4x=x(x-1)(x^2+3x+4)$$

It follows that x can have one of the 4 values

$$\left\{0,1,\frac{-3\pm i\sqrt{7}}{2}\right\}$$

Each of these x-values turns out to correspond to a *unique* y-value. Our four solutions are

$$(x,y)=\left\{(0,1),(1,0),\left(\frac{-3-i\sqrt{7}}{2},\frac{3-i\sqrt{7}}{2}\right),\left(\frac{-3+i\sqrt{7}}{2},\frac{3+i\sqrt{7}}{2}\right)\right\}$$

The **solve**-command will also reach this solution.

Chapter 3, 3.4 Exercise 5 (p. 43) We get

$$\begin{aligned}\mathrm{Res}(s+t-x,s^2-t^2-y,s) &= -2xt+x^2-y\\ \mathrm{Res}(s^2-t^2-y,2s-3t^2-z,s) &= 9t^4+6t^2z-4t^2-4y+z^2\\ \mathrm{Res}(s+t-x,2s-3t^2-z,s) &= -3t^2-2t+2x-z\end{aligned}$$

and

$$R = \text{Res}(-2xt + x^2 - y, -3t^2 - 2t + 2x - z, t) = \\ -3x^4 + 4x^3 + 6x^2y - 4x^2z + 4yx - 3y^2$$

so the implicit equation is

$$3x^4 - 4x^3 - 6x^2y + 4x^2z - 4yx + 3y^2 = 0$$

If we compute the resultant of $9t^4 + 6t^2z - 4t^2 - 4y + z^2$ and $-2xt + x^2 - y$ we get

$$9x^8 - 36x^6y + 24x^6z - 16x^6 + 54x^4y^2 \\ - 48x^4yz - 32x^4y + 16x^4z^2 \\ - 36x^2y^3 + 24x^2y^2z - 16x^2y^2 + 9y^4$$

which turns out to be a multiple of R.

Chapter 4, 4.1 Exercise 1 (p. 54) Use the command

```
desolve([ 'diff(x(t),t)=3*x(t)-4*y(t),
'diff(y(t),t)=2*x(t)+3*y(t)],[x(t),y(t)]);
```

to get

$$x(t) = -\frac{2\,y(0)\%e^{3t}\sin\left(2^{\frac{3}{2}}t\right) - \sqrt{2}\,x(0)\%e^{3t}\cos\left(2^{\frac{3}{2}}t\right)}{\sqrt{2}}$$

$$y(t) = \frac{x(0)\%e^{3t}\sin\left(2^{\frac{3}{2}}t\right) + \sqrt{2}\,y(0)\%e^{3t}\cos\left(2^{\frac{3}{2}}t\right)}{\sqrt{2}}$$

Chapter 4, 4.1 Exercise 2 (p. 55) Write

$$\frac{dy}{dx} = \text{yprime}$$

$$\frac{d\,\text{yprime}}{dx} = 3 * \text{yprime}^3 - 2 * y$$

Chapter 4, 4.1 Exercise 3 (p. 55) First, notice that the **ode2**-command gives **False**, admitting defeat in solving this differential equation exactly. We solve the exercise by typing:

```
plotdf(x-y^2,[xfun,"sqrt(x);-sqrt(x)"],
[trajectory_at,-1,3],
[direction,forward],
[y,-5,5], [x,-4,16]);
```

It's interesting to note that the solution-curve approaches $\sqrt{x}$ in the limit as $x \to \infty$. Clicking on other points on the plot shows that this is almost *always* the case.

Chapter 4, 4.1 Exercise 4 (p. 55) The code

```
plotdf([v,-k*z/m], [z,v],
[parameters,"m=2,k=2"],
[sliders,"m=1:5"],
[trajectory_at,6,0])
```

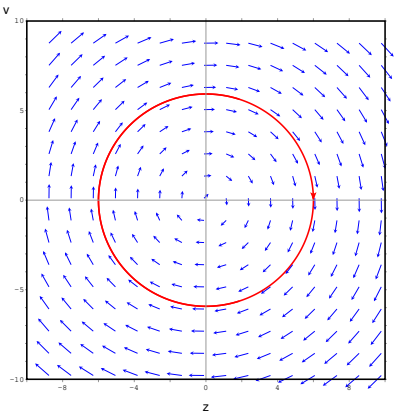


FIGURE G.5.1. Output of exercise plot

produces figure G.5.1.

If we run this with [**versus_t**,1], we also get plots of z and v.

Chapter 4, 4.1 Exercise 5 (p. 55) Try **expand**(%) and **ratsimp** a second time.

Chapter 4, 4.2 Exercise 1 (p. 58) If **v** is the velocity-vector, we have

$$\mathbf{v} = \begin{bmatrix} v_h \\ v_v \end{bmatrix}$$

its vertical and horizontal components. We have equations

$$\begin{aligned} m\frac{dv_v}{dt} &= -9.8m \\ m\frac{dv_h}{dt} &= 0 \end{aligned} \tag{G.5.1}$$

where m is the mass of the cannonball. At time zero $v_v = 1000\sin 30^\circ = 500m/sec$, and $v_h = 1000\cos 30^\circ = 866.025$. We get

$$\begin{aligned} v_v &= 500 - 9.8t \\ y &= 500t - 4.9t^2 \end{aligned}$$

The apogee occurs when $v_v = 0$ or at time $500/9.8$. This gives a value of y equal to 12755.10204081632.

Chapter 4, 4.2 Exercise 2 (p. 58) If $\mathbf{r}$ is the vector representing the air-resistance, we must have $\mathbf{r} = -\alpha\mathbf{v}$, where $\mathbf{v}$ is the velocity vector of the cannonball and $\alpha > 0$ is a scalar. The constant in equation 4.2.1 on page 58 is $\frac{1}{2}C_D\rho A = .057575$ and we have $|\mathbf{r}| = .057575|\mathbf{v}|^2$ so

$$\mathbf{r} = -.057575|\mathbf{v}|\mathbf{v}$$

Our equations of motion (equation G.5.1) become

$$\begin{aligned} m\frac{dv_v}{dt} &= -9.8m - .057575v_v\sqrt{v_h^2 + v_v^2} \\ m\frac{dv_h}{dt} &= -.057575v_h\sqrt{v_h^2 + v_v^2} \end{aligned} \tag{G.5.2}$$

where the mass is very significant now[4]. Dividing by the mass of 4kg gives

$$\frac{dv_v}{dt} = -9.8 - 0.01439375 v_v \sqrt{v_h^2 + v_v^2}$$

$$\frac{dv_h}{dt} = -0.01439375 v_h \sqrt{v_h^2 + v_v^2} \tag{G.5.3}$$

$$\frac{dx}{dt} = v_h \tag{G.5.4}$$

$$\frac{dy}{dt} = v_v \tag{G.5.5}$$

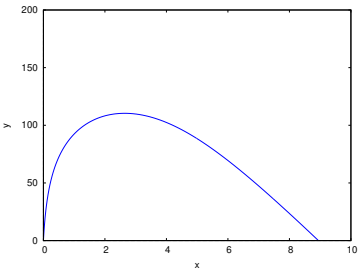
The first two equations are highly nonlinear, so we must use **rk** to solve them.

```
cannon:rk([-9.8-0.01439375*vv*sqrt(vv^2+vh^2),
-0.01439375*vh*sqrt(vv^2+vh^2),vv,vh],[vv,vh,y,x],
[500,866.025,0,0],[t,0,10,.01]);
```

Examining the output show that the apogee occurs at time 2.64 seconds and that y (the altitude) is 110.389 meters. We can reformat the output-list to graph the cannonball's progress:

```
height:makelist([elt[1],elt[4]],elt,cannon)
plot2d([discrete,height],[y,0,200])
```

We get



Chapter 4, 4.6 Exercise 2 (p. 80) Since x and y are independent variables

$$\int_{-\pi}^{\pi}\int_{\pi}^{\pi} \sin(nx)\sin(my)\cdot\sin(\bar{n}x)\sin(\bar{m}y)\,dxdy$$
$$= \left(\int_{-\pi}^{\pi} \sin(nx)\sin(\bar{n}x)\,dx\right)\left(\int_{-\pi}^{\pi} \sin(my)\sin(\bar{m}y)\,dy\right)$$

so the conclusion follows from equation 4.3.5 on page 60. This is why two-dimensional Fourier series work; indeed it is why n-dimensional Fourier series work.

Chapter 5, 5.2 Exercise 1 (p. 86) We could just use definition 5.2.1 on page 84 and straight computation (ugh!). *Or* we could use equation 5.2.4 on page 85 and the fact that *multiplication* is commutative and associative.

Chapter 5, 5.2 Exercise 2 (p. 86) Because the Fast Fourier Transform algorithm requires the length of the sequences to be a power of 2.

[4]This is why a feather falls more slowly than a cannonball in the Earth's atmosphere.

Chapter 5, 5.2 Exercise 4 (p. 86) Since the result will be of degree 9, the sequence of coefficients will be of length 10. The next higher power of 2 is $2^4 = 16$. Let

```
load ("fft");
fpprintprec : 4;  /* number of digits to print */
A : [2,-4,1,-1,0,0,0,0,0,0,0,0,0,0,0,0];
ia:inverse_fft(A);
ia3:ia^3; /* element by element operation */
A3:realpart(fft(ia3));
[8.0,-48.0,108.0,-124.0,102.0,-72.0,31.0,-15.0,3.0,-1.0,
-2.132*10^-14,2.842*10^-14,-2.132*10^-14,
1.421*10^-14,0,-3.553*10^-15]
```

The terms after the 10th are due to round-off errors. The result is

$$8 - 48x + 108x^2 - 124x^3 + 102x^4 - 72x^5 + 31x^6 - 15x^7 + 3x^8 - x^9$$

which you can verify by (tedious!) direct computation.

Chapter 7, 7.1 Exercise 2 (p. 113) In Manifold Theory and Differential Geometry (see [5]), volumes have an *orientation.* In $\mathbb{R}^n$ the standard orientation is

$$\{x_1, \ldots, x_n\}$$

and if you swap any pair of axes, the result has a *negative* orientation.

Chapter 7, 7.2 Exercise 3 (p. 116) Just form the matrix

$$P = \begin{bmatrix} 8 & -1 & 2 \\ 4 & 0 & 1 \\ 3 & -1 & 1 \end{bmatrix}$$

and compute

$$(P^{-1}).A.P = \begin{bmatrix} -27 & 5 & -8 \\ 53 & -10 & 15 \\ 137 & -25 & 40 \end{bmatrix}$$

Chapter 7, 7.4 Exercise 1 (p. 128)

$$\text{(G.5.6)} \quad A^n = \begin{bmatrix} -2^{n+2} - 3^{n+1} + 8 & 3^n - 1 & -3^n - 2^n + 2 \\ 2^{n+3} - 8 & 1 & 2^{n+1} - 2 \\ 5 \cdot 2^{n+2} + 4 \cdot 3^{n+1} - 32 & 4 - 4 \cdot 3^n & 4 \cdot 3^n + 5 \cdot 2^n - 8 \end{bmatrix}$$

Chapter 7, 7.4 Exercise 2 (p. 128) Just plug $n = 1/2$ into equation G.5.6 to get

$$\sqrt{A} = \begin{bmatrix} -4\sqrt{2} - 3\sqrt{3} + 8 & \sqrt{3} - 1 & -\sqrt{3} - \sqrt{2} + 2 \\ 8\sqrt{2} - 8 & 1 & 2\sqrt{2} - 2 \\ 20\sqrt{2} + 12\sqrt{3} - 32 & 4 - 4\sqrt{3} & 4 \cdot \sqrt{3} + 5\sqrt{2} - 8 \end{bmatrix}$$

Chapter 7, 7.5 Exercise 2 (p. 139) We know

$$\begin{aligned} {}_x\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} &= e^{\left(\log(x) \cdot \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}\right)} \\ &= e^{\begin{bmatrix} \log(x) & 2\log(x) \\ 3\log(x) & 4\log(x) \end{bmatrix}} \end{aligned}$$

We type

```
z:matrix([log(x),2*log(x)],[3*log(x),4*log(x)]);
a:matrixexp(z);
```

and get a very messy and convoluted expression that none of our normal simplification commands improve until we type

```
radcan(a);
```

which gives

$$\begin{bmatrix} -\frac{\left(\sqrt{3}\sqrt{11}-11\right)x^{\sqrt{3}\sqrt{11}+2}+\left(-\sqrt{3}\sqrt{11}-11\right)x^2}{22x^{\frac{\sqrt{3}\sqrt{11}-1}{2}}} & \frac{2\sqrt{3}\sqrt{11}x^{\sqrt{3}\sqrt{11}+2}-2\sqrt{3}\sqrt{11}x^2}{33x^{\frac{\sqrt{3}\sqrt{11}-1}{2}}} \\ \frac{\sqrt{3}\sqrt{11}x^{\sqrt{3}\sqrt{11}+2}-\sqrt{3}\sqrt{11}x^2}{11x^{\frac{\sqrt{3}\sqrt{11}-1}{2}}} & \frac{\left(\sqrt{3}\sqrt{11}+11\right)x^{\sqrt{3}\sqrt{11}+2}+\left(11-\sqrt{3}\sqrt{11}\right)x^2}{22x^{\frac{\sqrt{3}\sqrt{11}-1}{2}}} \end{bmatrix}$$

We can further simplify this by hand.

Chapter 7, 7.6 Exercise 1 (p. 145) We begin with j=number of jackets and p=number of pants. The objective function is

$$3p + 2j$$

Our constraints are

$$8p + 4j \leq 60$$

and

$$4p + 8j \leq 48$$

So we formulate this as

```
maximize_lp (3*p+2*j , [8*p+4*j <=60,4*p+8*j <=48])
```

and Maxima comes back with

$$[24, [p = 6, j = 3]]$$

Chapter 7, 7.6 Exercise 2 (p. 145) It's a good idea to restart Maxima (using that command on the Maxima menu) and reload the **simplex** library. We set p=potatoes,c=corn. The profit (objective function) is

$$150p + 50c$$

and the constraints are

$$20p + 60c \leq 3000$$
$$p + c \leq 70$$

As before, we write

```
maximize_lp(150*p+50*c , [20*p+60*c<=3000,p+c<=70])
```

and get

Problem not bounded!

Potatoes are *so profitable*, it's worthwhile to grow "negative" corn to increase the production of potatoes. We must insist that negative potatoes and corn do not exist!

```
maximize_lp(150*p+50*c , [20*p+60*c<=3000,p+c<=70,p>=0,c>=0])
```

and get

$$[10500, [p = 70, c = 0]]$$

so there's no point in planting *any* corn!

Chapter 8, 8.1 Exercise 1 (p. 159) Here is a program for computing these functions:

```
c0 : (1+sqrt(3))/4;
c1 : (3+sqrt(3))/4;
c2 : (3-sqrt(3))/4;
c3 : (1-sqrt(3))/4;
p1 : (1+sqrt(3))/2;
p2 : (1-sqrt(3))/2;
d4[x] := block(
            [],
             if x <= 0 then return(0),
             if 3 <= x then return(0),
             if x = 1 then return(p1),
             if x = 2 then return(p2),
       ratsimp(expand(c0*d4[2*x]+c1*d4[2*x-1]+
                      c2*d4[2*x-2]+c3*d4[2*x-3]))
       );
w4(x) := ratsimp(expand(c3*d4[2*x+2]-c2*d4[2*x+1]+
                      c1*d4[2*x]-c0*d4[2*x-1]))
```

Note: *memoizing* d4 (see page 71) speeds this up *immensely!* Although Maxima is rather slow, it has the advantage that it performs *exact* calculations, so there is no roundoff error.

Chapter 8, 8.1 Exercise 2 (p. 159) First, we generate a list of all the x-values of the plot with

```
xpoints:makelist(j/2^10,j,0,3*2^10,1)
```

and then generate the points to be plotted via

```
ppoints:makelist([x,d4[x]],x,xpoints);
```

Then, we plot it via

```
plot2d([discrete,ppoints]);
```

which will give us a plot that looks like figure 8.1.3 on page 157.

Chapter 8, 8.3 Exercise 4 (p. 170) The values of ϕ at integer values must satisfy

$$\phi(i/2^{k+1}) = \sum_{-\infty<m<\infty} \xi_m \phi(\frac{i}{2^k} - m) \tag{G.5.7}$$

or

$$\begin{aligned}
\phi(1) &= c_0\phi(2) + c_1\phi(1) \\
\phi(2) &= c_0\phi(4) + c_1\phi(3) + c_2\phi(2) + c_3\phi(1) \\
\phi(3) &= c_2\phi(4) + c_3\phi(3) + c_4\phi(2) + c_5\phi(1) \\
\phi(4) &= c_4\phi(4) + c_5\phi(3)
\end{aligned}$$

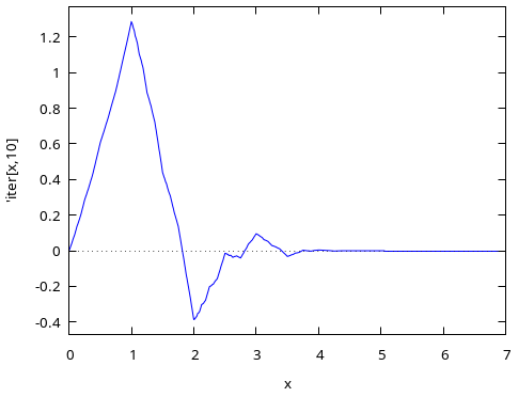


FIGURE G.5.2. Plot of D6

and the eigenvector corresponding to eigenvalue 1 can be converted to float numbers. We get

```
par : sqrt(5+2*sqrt(10));
c0 : (1+par+sqrt(10))/16;
c1 : (5+sqrt(10)+3*par)/16;
c2 : (5-sqrt(10)+par)/8;
c3 : (5-sqrt(10)-par)/8;
c4 : (5-3*par+sqrt(10))/16;
c5 : (1-par+sqrt(10))/16;
p1 : 1.286335069425677;
p2 : -0.3858376494301308;
p3 : 0.09526780086916785;
p4 : 0.004234433942314734;
d6[x] := block(
            [],
             if x <= 0 then return(0),
             if 5 <= x then return(0),
             if x = 1 then return(p1),
             if x = 2 then return(p2),
             if x = 3 then return (p3),
             if x = 4 then return (p4),
       ratsimp(expand(c0*d6[2*x]+c1*d6[2*x-1]+
                      c2*d6[2*x-2]+c3*d6[2*x-3]+c4*d6[2*x-4]
                       +c5*d6[2*x-5]));
```

Note: *memoizing* d6 (see page 71) speeds this up *immensely!* Although Maxima is rather slow, it has the advantage that it performs *exact* calculations, so there is no roundoff error.

We get the plot in figure G.5.2.

Chapter 9, 9.2 Exercise 1 (p. 181) Because the **make_graph** command requires a function of *two* variables.

Chapter 9, 9.3 Exercise 3 (p. 188) A graph with the fewest edges such that every vertex can reach every other is a *tree*, so[5]: Create a weighted graph where each house becomes a vertex and each path becomes a edge weighted by the cost of paving that path. The solution is a minimal spanning tree of that graph.

Chapter 10, 10.1 Exercise 1 (p. 196) Equation 10.1.1 on page 192 implies that

$$\mathbf{\Sigma} x_{(-i)} = \mathbf{E}\frac{x_{(1-i)}}{1-i}$$

In the case where $i = 1$, we get

$$\mathbf{\Sigma}_0^n x_{(-1)} = 1 + \frac{1}{2} + \cdots + \frac{1}{n+1} = H_{n+1}$$

the $n+1$st *Harmonic number*. This shows that the harmonic numbers play a part in finite-difference calculus similar to that of the *logarithm* in regular calculus.

Chapter 10, 10.1 Exercise 2 (p. 196) There's nothing to prove if $n = 0, 1$. Assume it has been proved for all values of $n < k$, and we want to proved it for x^k. The *lowest-degree* falling factorial that contains a term of x^k is $x_{(k)}$, and it contains a *single term* of x^k. It follows that

$$x^k - x_{(k)}$$

consists of a linear combination of powers x^i with $i < k$. By the induction hypothesis

$$x^k - x_{(k)} = \sum_{i=0}^{k-1} a_i x_{(i)}$$

with $a_i \in \mathbb{Z}$, and

$$x^k = x_{(k)} + \sum_{i=0}^{k-1} a_i x_{(i)}$$

See equation E.11.1 on page 317.

Chapter 10, 10.1 Exercise 3 (p. 196) Straight (somewhat tedious) computation.

Chapter 10, 10.1 Exercise 5 (p. 196) Since the equation is linear, proving it for some class of functions proves it for all possible *linear combinations* of these functions. It's sufficient to prove for $f(x) = x^n$ for all integers $n \geq 0$. Since these functions are linear combinations of falling factorials (exercise 2 on page 196), it suffices to prove it for all falling factorials $x_{(n)}$ with n an integer ≥ 0. If we set $a = 0$ in equation 10.1.4 on page 193, we get

$$S = 0 + \mathbf{\Delta}[x_{(n)}](0)(x)_{(1)} + \frac{\mathbf{\Delta}^2[x_{(n)}](0)(x)_{(2)}}{2!} + \frac{\mathbf{\Delta}^3[x_{(n)}](0)(x)_{(3)}}{3!} + \cdots$$

Now

$$\mathbf{\Delta}^i[x_{(n)}](0)(x)_{(i)} = n(n-1)\cdots(n-i+1)x_{(n-i)}(0) = \begin{cases} 0 & \text{if } i < n \\ n! & \text{if } i = n \\ 0 & \text{if } i > n \end{cases}$$

so $S = x_{(n)}$.

[5] If it has a loop, delete one edge, and every vertex will *still* be able to reach every other — maybe going the *long* way around.

Chapter 10, 10.1 Exercise 6 (p. 196) Just write $\mathbf{\Delta} = \mathbf{E} - 1$ and plug it in:

$$\begin{aligned}(\mathbf{E}-1)(fg) &= \mathbf{E}(f)\mathbf{E}(g) - fg \\ &= (\mathbf{E}(f) - f)\,\mathbf{E}(g) + f\mathbf{E}(g) - fg \\ &= (\mathbf{E}(f) - f)\,\mathbf{E}(g) + f\cdot(\mathbf{E}g - g) \\ &= \mathbf{\Delta}f\mathbf{E}(g) + f\mathbf{\Delta}g\end{aligned}$$

Chapter 10, 10.1 Exercise 7 (p. 196) Just plug equation 10.1.8 on page 196 into $\mathbf{\Sigma}_a^b$ and rearrange terms.

Chapter 10, 10.1 Exercise 8 (p. 196) Yes. Consider the vector-space spanned by $\{x^n\}$ for n an integer ≥ 0. In this vector-space, the operators $\mathbf{E}$, $\mathbf{\Delta}$, and $\mathbf{D}$ are (infinite) matrices and equation 10.1.6 on page 194 is literally true.

Chapter 10, 10.1 Exercise 9 (p. 196) Set $f(x) = x$ and $g(x) = 2^x$. Then $\mathbf{\Delta}g = g$, $\mathbf{\Delta}f = 1$, and we get

$$\begin{aligned}\mathbf{\Sigma}_0^n f\mathbf{\Delta}g &= \mathbf{E}(fg)(n) - (fg)(0) - \mathbf{\Sigma}_0^n\mathbf{E}(g)\mathbf{\Delta}f \\ &= (n+1)2^{n+1} - \sum_{k=0}^{n} 2^{k+1'} \\ &= (n+1)2^{n+1} - 2\sum_{k=0}^{n} 2^k \\ &= (n+1)2^{n+1} - 2(2^{n+1}-1) \\ &= (n+1)2^{n+1} - 2^{n+2} + 2\end{aligned}$$

Chapter 10, 10.1 Exercise 10 (p. 196) This is easier than it looks. Code

```
z(n):=ratsimp(1-t^n)/(1-t))
```

Then

```
z(3)
```

produces

$$t^2 + t + 1$$

It's not hard to see that for $n > 0$ an integer,

$$\frac{1-t^n}{1-t} = 1 + t + \cdots + t^{n-1}$$

Integrating this gives the conclusion.

Chapter 10, 10.2 Exercise 4 (p. 201) You can just write

```
Bdeltan(f,x,m) := buildq([y:x, g:f, n:m],
                    lambda([y],
                    sum((-1)^(k+1)*binomial(n,k)*g(y+k),
                              k,0,n))
                    );
```

or (more efficiently)

```
Bdeltan(f,x,m) := buildq([y:x, g:f, n:m],
                    lambda([y],
                    -sum((-1)^k*binomial(n,k)*g(y+k),
                              k,0,n))
```

```
);
```

Chapter 11, 11.5 Exercise 1 (p. 214) Typing

```
poly_reduced_grobner([x^2+y^2,x^3-y^4],[x,y])
```

returns

$$\left[y^2+x^2, y^4+xy^2, y^6+y^4\right]$$

so the original equations are equivalent to

$$y^2+x^2=0$$
$$y^4+xy^2=0$$
$$y^6+y^4=0$$

The last equation implies that $y = 0, \pm i$. The first equation implies that, if $y = 0$, then $x = 0$. The second equation implies that, if $y = \pm i$, then $x = 1$. So the solutions to the original equations are

$$x = y = 0,$$
$$x = 1, y = \pm i$$

So there are a total of three solutions.

Chapter 11, 11.5 Exercise 2 (p. 214) Define an ideal and type

```
poly_reduced_grobner([a1*a2-b1*b2+a1-1,
a2*b1+a1*b2+b1-1/2,a1^2+b1^2-1,a2^2+b2^2-1],[a1,b1,a2,b2])
```

to get

$$\left[64b2^2-55, -4b2+10a1-5, -16b2-20b1+5, 8a2+3\right]$$

from which we deduce that $a_2 = -3/8$ and b_2 can be either $+\sqrt{55}/8$ in which case

$$\begin{aligned} a_1 &= 1/2+\sqrt{55}/20 \\ b_1 &= 1/4-\sqrt{55}/10 \end{aligned}$$

or $-\sqrt{55}/8$ in which case

$$\begin{aligned} a_1 &= 1/2-\sqrt{55}/20 \\ b_1 &= 1/4+\sqrt{55}/10 \end{aligned}$$

The **solve**-command also works in this case.

Chapter 11, 11.5 Exercise 3 (p. 214) Make this into an ideal problem:

Create an ideal

$$\left((a^2+1)(b^2+1)+25-10(a+b), ab-1, a^3+b^3-z\right)$$

and find a Gröbner basis with $a \succ b \succ z$:

Typing

```
poly_reduced_grobner([(a^2+1)*(b^2+1)+25-10*(a+b),
a^3+b^3-z,a*b-1],[a,b,z]);
```

gives

$$\left[-bz+72b^2-250b+72,-z^2+220z-12100,-z+72b+72a-250\right]$$

and

```
solve(-z^2+220*z-12100=0,z);
```

gives $z = 110$.

Chapter 11, 11.5 Exercise 4 (p. 214) Type

```
poly_reduced_grobner([x^2+x*y+y^2-39,
y^2+y*z+z^2-49,z^2+z*x+x^2-19],[x,y,z]);
```

to get

$$\left[-7z^3+58z+3y,-7z^4+64z^2-9,7z^3-67z+6x\right]$$

and type

```
solve(-7*z^4+64*z^2-9,z);
```

to get

$$\left[z=-\frac{1}{\sqrt{7}},z=\frac{1}{\sqrt{7}},z=-3,z=3\right]$$

Now you can solve for the x and y values that go with these z-values:

- ▷ If $z = -1/\sqrt{7}$, then $y = 19/\sqrt{7}$, $x = -11/\sqrt{7}$.
- ▷ If $z = 1/\sqrt{7}$,then $y = -19/\sqrt{7}$, $x = 11/\sqrt{7}$.
- ▷ If $z = 3$, then $y = 5$, $x = 2$.
- ▷ If $z = -3$, then $y = -5$, $x = -2$.

Chapter 11, 11.5 Exercise 5 (p. 214) Type

```
poly_reduced_grobner([a1*x1^5 + a2*x2^5 + a3*x3^5 - 1/6,
a1*x1^4 + a2*x2^4 + a3*x3^4 - 1/5,
a1*x1^3 + a2*x2^3 + a3*x3^3 - 1/4,
a1*x1^2 + a2*x2^2 + a3*x3^2 - 1/3,
a1*x1 + a2*x2 + a3*x3 - 1/2,
a1 + a2 + a3 - 1],
[x1,x2,x3,a1,a2,a3]);
```

to get a basis that contains the term

$$162\cdot a2^2-117\cdot a2+20$$

The command

```
solve(162*a2^2-117*a2+20=0,a2);
```

produces

$$\left[a2=\frac{5}{18},a2=\frac{4}{9}\right]$$

Adding a2-4/9 to the original list produces a Gröbner basis of

$$\left[9\cdot a2-4,5-18\cdot a3,1-2\cdot x2,5-18\cdot a1,-x3-x1+1,-10\cdot x3^2+10\cdot x3-1\right]$$

From which it is straightforward to get all the other values.

Chapter 12, 12.2 Exercise 1 (p. 225) Run the Maxima command

```
poly_reduced_grobner([a5*a4*a3-a5*b4*b3+a5*a4-x,
b5*a4*a3-b5*b4*b3+b5*a4-y,
b4*a3+a4*b3+b4-z,
x^2+y^2+z^2-r^2,
a2^2+b2^2-1,
a3^2+b3^2-1,
a4^2+b4^2-1,
a5^2+b5^2-1],
[a5,a4,a3,a2,b5,b4,b3,b2,x,y,z,r]);
```

and, in the long list of expressions in the Gröbner basis there is

$$-r^4 + 4 * r^2 - 4 * b3^2$$

Since $b3^2 \geq 0$, we must have $-r^4 + 4r^2 \geq 0$ or $r \leq 2$. We could have come to the same conclusion by the fact that the robot-arm has two links of length 1!

Chapter 14, 14.1 Exercise 1 (p. 248) We get

$$(x)_m = \frac{\Gamma(x+1)}{\Gamma(x-m+1)}$$

If x and m are integers and $m > x$, then the denominator is Γ evaluated at 0 or a negative integer, which is infinite, so the quotient will be 0.

Chapter 14, 14.1 Exercise 2 (p. 248) Unfortunately, the **gamma** command just crashes if it's fed an argument that is a nonpositive integer (i.e., it doesn't return **inf**). We must first check that case and return 0. We get

```
ff(x,m) :=block(
                [a:imagpart(x-m+1),b:realpart(x-m+1)],
                if(a = 0) and (b <=0 ) and /*Test for
                                        a negative
                                        real integer*/
                   abs(mod(b,1))<.000001
                then return (0),
                  gamma(x+1)/gamma(x-m+1)
                );
```

If x and m are integers and $m > x$, then the denominator is Γ evaluated at 0 or a negative integer, which is infinite, so the quotient will be 0.

Chapter 14, 14.1 Exercise 3 (p. 248) Use the Γ-function! Our formula in the exercise becomes

$$\frac{d^k x^n}{dx^k} = \frac{\Gamma(n+1)}{\Gamma(n-k+1)} x^{n-k}$$

and

$$\frac{d^{1/2} x}{dx^{1/2}} = \frac{x^{1/2}}{\sqrt{\pi}/2} = \frac{2x^{1/2}}{\sqrt{\pi}}$$

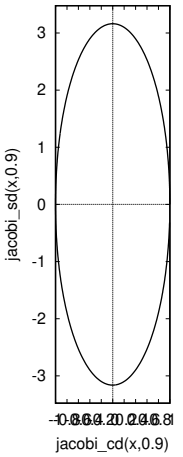


FIGURE G.5.3. Plot of ellipse,**same_xy**

Chapter 14, 14.1 Exercise 4 (p. 248) Define $L(x) = \frac{d}{dx}(\log\Gamma(x+1))$. Then

$$\begin{aligned} L(x+1) &= \frac{d}{dx}(\log\Gamma(x+2) = \frac{d}{dx}(\log((x+1)\Gamma(x+1)) \\ &= \frac{d}{dx}(\log(x+1) + \frac{d}{dx}(\log(\Gamma(x+1)) \\ &= \frac{1}{x+1} + L(x) \end{aligned}$$

So this function satisfies the same functional equation as $H(x)$. It follows that $H(x) - L(x)$ is a constant when x is a positive integer. Since

```
psi[0](1)=-%gamma
```

it follows that this constant difference is γ.

Chapter 14, 14.1 Exercise 6 (p. 249) Use the Γ-function! Our formula in the exercise becomes

$$\frac{d^{1/2}x^n}{dx^{1/2}} = \frac{n!}{\Gamma(n+1/2)}x^{n-1/2}$$

so

$$\frac{d^{1/2}e^x}{dx^{1/2}} = \sum_{n=1}^{\infty}\frac{x^{n-1/2}}{\Gamma(n+1/2)}$$

We assume that

$$\frac{d^{1/2}1}{dx^{1/2}} = 0$$

Chapter 14, 14.2 Exercise 1 (p. 255) Use the command

```
plot2d([parametric,jacobi_cd(x,.9),jacobi_sd(x,.9),
[x,0,10]],same_xy,[style,[lines,2,5]]);
```

to get the plot in figure G.5.3. If we leave out the '**same_xy**' we get the well-proportioned but distorted plot in figure G.5.4 on the next page.

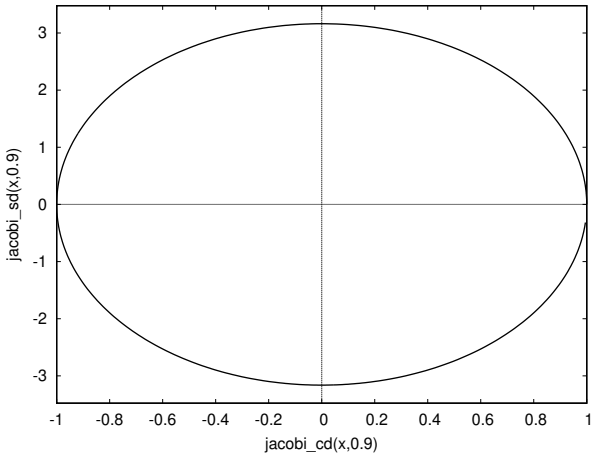


FIGURE G.5.4. Plot without **same_xy**

Chapter 14, 14.2 Exercise 3 (p. 255) If the starting angle is 180°, the pendulum is *vertical.* This is an unstable equilibrium (very!), and the pendulum can theoretically remain in this position forever.

Chapter 14, 14.5 Exercise 2 (p. 265) Using Euler's formula

$$e^{ix} = \cos x + i \sin x$$

we get

$$\mathrm{E}_1(ix) = i\left(-\frac{\pi}{2} + \mathrm{Si}(x)\right) - \mathrm{Ci}(x)$$

Chapter 14, 14.6 Exercise 2 (p. 268) First, code a function

```
tow(x):=-lambert_w(-log(x))/log(x);
```

Basic experimentation shows that

```
float(sqrt(2)+.030454298636671))
```

is still real, and that its value is

$$2.718281776395266$$

which leads one to suspect that the maximum finite value of $t_\infty(x)$ is e. Note, this is not a *local* maximum, so taking the derivative of $t_\infty(x)$ and setting it to 0 will not work.

The value where this occurs is

$$v = \sqrt{2} + .030454298636671 = 1.444667861009766$$

If we randomly play around with this value, we eventually find

$$\frac{1}{\log(v)} = 2.718281828459046$$

so

$$v = e^{\frac{1}{e}}$$

And, if we type

```
tow(%e^(1/%e))
```

we get

```
%e
```

as implied by equation 14.6.4 on page 268.

Chapter 14, 14.6 Exercise 3 (p. 268) Typing

```
x:tow(3)
```

gives

$$-\frac{\text{lambert\_w}(-\log(3))}{\log(3)}$$

and

```
float(%)
```

gives

$$-0.9102392266268373 \cdot (1.391335054072608 \cdot \%i - 0.2524062904251475)$$

The command

```
rectform(%)
```

gives

$$y = 0.2297501065923352 - 1.26644774359786 \cdot \%i$$

What has happened here? Clearly the infinite power tower

$$3^{3^{3^{\cdot^{\cdot}}}} \to \infty$$

isn't well-defined. The point is that we have solved the equation

$$y = 3^y$$

which is *still* well-defined even when the power-tower is *not*. The solution, y, is a value with the property that

$$y^{1/y} = 3$$

If you type

```
y^(1/y)
```

you get

$$(0.2297501065923352 - 1.26644774359786 \cdot \%i)^{\frac{1}{0.2297501065923352 - 1.26644774359786 \cdot \%i}}$$

and

```
rectform(%)
```

gives

$$6.661338147750939 \cdot 10^{-16} \cdot \%i + 3.0$$

where the imaginary part is round-off error.

Chapter 14, 14.6 Exercise 4 (p. 268) If

$$v = u^{1/u}$$

then equation 14.6.3 on page 268 implies that

$$u = -\frac{W(-\log(v))}{\log(v)}$$

Now set $u = 1/x$ so that we get

$$v = \left(\frac{1}{x}\right)^x = \frac{1}{x^x}$$

and set

$$y = 1/v$$

We get

$$x = \frac{\log(y)}{W(\log(y))}$$

Chapter 15, 15.2 Exercise 2 (p. 282) Since

$$p(x) = \pi(x) + \pi(x^{1/2})/2 + \pi(x^{1/3})/3 + \cdots$$

we get

$$p(x) - p(x^{1/2})/2 = \pi(x) + \pi(x^{1/3})/3 + \pi(x^{1/5})/5 \cdots$$

and

$$p(x) - p(x^{1/2})/2 - p(x^{1/3})/3 - p(x^{1/5})/5 + p(x^{1/6})/6 = \pi(x) + \pi(x^{1/7})/7 + \cdots$$

where, at each step the sum on the left consists of terms with the primes already covered and the sum on the right consists of *none* of those primes.

Index

Bibliography

[1] Lars Ahlfors. *Complex Analysis*. McGraw-Hill Education; 3rd edition, 1979.

[2] Tom Apostol. "An Elementary View of Euler's Summation Formula." In: *The American Mathematical Monthly* 106.5 (1999), pp. 409–418.

[3] G. Arens et al. "Wave propagation and sampling theory." In: *Geophysics* 47 (1982), pp. 203–236.

[4] W. W. Rouse Ball. *A short account of the history of mathematics*. E-book number 31246.
http://www.gutenberg.org: Project Gutenberg, 2010.

[5] Dennis Barden and Charles B Thomas. *An Introduction To Differential Manifolds*. ICP, 2003. ISBN: 1860943551.

[6] Étienne Bézout. "Sur le degré des équations résultantes de l'évanouissement des inconnues et sur les moyens qu'il convient d'employer pour trouver ces équations." In: *Histoire de l'académie royale des sciences* (1764), pp. 288–338.

[7] Norman L. Biggs, E. K. Lloyd, and Robin J. Wilson. *Graph Theory: 1736-1936*. Oxford: Clarendon Press, 1976.

[8] A. Bijaoui, G. Mars, and E. Slezak. "Identification of Structures from Galaxy Counts — Use the Wavelet Transform." In: *Astronomy and Astrophysics* 227.2 (1990), pp. 301–316.

[9] George Boole. *A Treatise on the Calculus of Finite Differences*. Cosimo Classics, 2007. ISBN: 978-1602063044.

[10] Sergey Brin and Lawrence Page. *The Anatomy of a Large-Scale Hypertextual Web Search Engine*. Google. URL: `http://infolab.stanford.edu/~backrub/google.html`.

[11] Kurt Bryan and Tanya Leise. *The $25,000,000,000 Eigenvector; The Linear Algebra Behind Google*. English. Rose-Hulman Institute of Technology. URL: `https://www.rose-hulman.edu/~bryan/googleFinalVersionFixed.pdf`.

[12] Bruno Buchberger. "A Criterion for Detecting Unnecessary Reductions in the Construction of Gröbner Bases." In: *Proceedings of the International Symposium on Symbolic and Algebraic Manipulation (EUROSAM '79)*. 1979.

[13] Bruno Buchberger. "Ein Algorithmus zum Auffinden der Basiselemente des Restklassenrings nach einem

nulldimensionalen Polynomideal." PhD thesis. Johannes Kepler University of Linz (JKU), RISC Institute., 1965.

[14] Bruno Buchberger. "Some properties of Gröbner bases for polynomial ideals." In: *ACM SIGSAM Bulletin* 10 (1976), pp. 19–24.

[15] Ingrid Daubechies. "Orthonormal bases of compactly supported wavelets." In: *Comm. Pure Appl. Math.* 41 (1988), pp. 909–996.

[16] Whitfield Diffie and Martin Hellman. "New directions in cryptography." In: *IEEE Transactions on Information Theory* 22.6 (1976).

[17] Euclid. *The Thirteen Books of the Elements, Vol. 2*. Dover Publications, 1956.

[18] Euclid. *The Thirteen Books of the Elements, Vol. 3*. Dover Publications, 1956.

[19] L. Euler. "Methodus uniuersalis seriemm convergentium summas quam proxime inueniendi." In: *Commentarii academie scientiarum Petropolitana* 8 (1736), pp. 3–9.

[20] L. Euler. "Methodus universalis series summandi ulterius promota." In: *Methodus universalis series summandi ulterCommentarii academie scientiarum Petropolitana* 8 (1736), pp. 147–158.

[21] Leonhard Euler. "Solutio problematis ad geometriam situs pertinentis." In: *Commentarii academiae scientiarum Petropolitanae* 7 (1741), pp. 1–10.

[22] F.A. Ficken. *The Simplex Method of Linear Programming*. Dover Books on Mathematics, 2015. ISBN: 978-0486796857.

[23] Johann Carl Friedrich Gauss. *Disquisitiones Arithmeticæ*. Translated by Arthur A. Clarke. Original available online at http://resolver.sub.uni-goettingen.de/purl?PPN235993352. Yale University Press, 1965.

[24] Walter Gautschi. "Leonhard Euler: His Life, the Man, and His Works." In: *SIAM Review* (2008). URL: `https://www.cs.purdue.edu/homes/wxg/Euler/EulerLect.pdf`.

[25] W. M. Gentleman and S. C. Johnson. "The Evaluation of Determinants by Expansion by Minors and the General Problem of Substitution." In: *Mathematics of Computation* 28.126 (Apr. 1974), pp. 543–548.

[26] Sophie Germain. *Recherches sur la théorie des surfaces élastiques*. Available online from http://books.google.com. Courcier, 1821.

[27] Wolfgang Gröbner. "Über die Eliminationstheorie." In: *Monatshefte für Mathematik* 54 (1950), pp. 71–78.

[28] P. Gropillaud, A. Grossman, and J. Morlet. "Cycle-octave and related transforms in seismic signal analysis." In: *Geoexploration* 23 (1984), pp. 85–102.

[29] Keld Helsgun. *An effective implementation of the Lin-Kernighan traveling salesman heuristic*. Tech. rep. 86. Download: `http://webhotel4.ruc.dk/~keld/research/LKH/LKH-`

2 . 0 / DOC / LKH _ REPORT . pdf. Roskilde Universitet, Department of Computer Science, 1999.

[30] R.E. Hewitt and E. Hewitt. "The Gibbs-Wilbraham phenomenon: An episode in fourier analysis." In: *Arch. Hist. Exact Sci.* 21 (1979), pp. 129–160.

[31] Carl Hierholzer. "Ueber die Möglichkeit, einen Linienzug ohne Wiederholung und ohne Unterbrechung zu umfahren." In: *Mathematische Annalen* 6.1 (1873), pp. 30–32.

[32] Peter Høeg. *Smilla's Sense of Snow*. Delta; Reprint edition, 1995.

[33] Jean-Pierre Tignol. *Galois' Theory of Algebraic Equations*. Singapore: World Scientific, 2001.

[34] Carl Jacobi. "Suite des notices sur les fonctions elliptiques." In: *Journal für die reine und angewandte Mathematik* 3 (1828), pp. 303–308.

[35] Palle E. T. Jorgensen. *Analysis and Probability: Wavelets, Signals, Fractals*. Vol. 234. Graduate Texts in Mathematics. Springer, 2006. ISBN: 978-0387295190.

[36] Shen Kangshen and John Crossley. *The Nine Chapters on the Mathematical Art: Companion and Commentary*. Oxford University Press, 2000. ISBN: 978-0198539360.

[37] Donald Knuth. "Euler's constant to 1271 places." In: *Math. of Computation* 16 (1962), pp. 275–281-.

[38] Joseph P.S. Kung. "Möbius inversion." In: *Encyclopedia of Mathematics*, (2001).

[39] Y. S. Kwoh et al. "A robot with improved absolute positioning accuracy for CT guided stereotactic brain surgery." In: *IEEE Trans. Biomed. Engng.* 35 (1988), pp. 153–161.

[40] Jussi Lehtonen. "The Lambert W function in ecological and evolutionary models." In: *Methods in Ecology and Evolution* 7 (2016), pp. 1110–1118.

[41] P. H. Leslie. "The use of matrices in certain population mathematics." In: *Biometrika* 33.3 (1945), pp. 183–212.

[42] Claude Lobry and Tewfik Sari. "Migrations in the Rosenzweig-MacArthur model and the "atto-fox" problem." In: *ARIMA Journa* 20 (2015), pp. 95–125.

[43] Arnold N. Lowan. *On the use of Chebyshev polynomials in numerical analysis*. University of Michigan Library, 1958.

[44] S. G. Mallat. "A theory for multiresolution signal decomposition: the wavelet representation." In: *IEEE Trans. Pattern Anal. and Machine Intel.* 11 (1989), pp. 674–693.

[45] Ernst W. Mayr. "Some complexity results for polynomial ideals." In: *Journal of Complexity* 13 (1997), pp. 301–384.

[46] Y. Meyer. *Ondelettes et Opérateurs*. Paris, FRANCE: Hermann.

[47] Charles W. Misner, Kip S. Thorne, and John Archibald Wheeler. *Gravitation*. Physics. W. H. Freeman, 1973.

[48] Ramaswami Mohandoss. *What is the P vs NP problem?: A complete explanation of the biggest unsolved problem in Computer Science*. Orange Window Publishing, 2022. ISBN: 978-0998974613.
[49] C. D. Olds. *Continued Fractions*. Random house, 1963.
[50] Alan V. Oppenheim, Ronald W. Schafer, and John R. Buck. *Discrete-time signal processing*. Also available at https://d1.amobbs.com/bbs_upload782111/files_24/ourdev_523225.pdf. Upper Saddle River, N.J.: Prentice Hall., 1999. ISBN: 0-13-754920-2.
[51] Bernhard Riemann. "Über die Anzahl der Primzahlen unter einer gegebenen Grösse." In: *Monatsberichte der Berliner Akademie* (1859).
[52] Edward Rosen. *Kepler's Conversation with Galileo's Sidereal messenger. First Complete Translation, with an Introduction and notes.* Johnson Reprint Corp., 1965.
[53] M. Rosenzweig and R. MacArthur. "Graphical representation and stability conditions of predator-prey interactions." In: *American Naturalist* 97.895 (1963).
[54] Walter Rudin. *Real and Complex Analysis*. MC GRAW HILL INDIA, 1987.
[55] H. E. Salzer and R. Zucker. "Table of zeros and weight factors of the first fifteen Laguerre polynomials." In: *Bulletin of the American Mathematical Society* 55.10 (1949), pp. 1004–1012.
[56] Percy Bysshe Shelley. *Queen Mab: A Philosophical Poem, With Notes to Which Is Added a Brief Memoir of the Author*. Forgotten Books, 2012.
[57] Murasaki Shikibu. *The Tale of Genji*. Ed. by Royall Tyler (translator). Penguin Classics; Reprint edition, 2002.
[58] Justin R. Smith. *Abstract Algebra*. Five Dimensions Press, 2019. ISBN: I978-1070799605.
[59] Justin R. Smith. *Bloodline*. Ebook. ASIN : B07BTDNTP5. Silver Leaf Books, 2018.
[60] Justin R. Smith. *Introduction to Algebraic Geometry*. Five Dimensions Press, 2014.
[61] W. J. Stewart. *An Introduction to the Numerical Solution of Markov Chains*. Princeton University Press, 1994.
[62] Gilbert Strang. "Wavelets and dilation equations: A brief introduction." In: *SIAM J. Comput.* 31.4 (1989), pp. 614–627.
[63] Wim Sweldens and Robert Piessens. *An Introduction to Wavelets and Efficient Quadrature Formulae for the Calculation of the Wavelet Decomposition*. TW 159. Celestijnenlaan 200A — B-3001 Leuven Belgium: Department of Computer Science Katholieke Universiteit Leuven, 1991.

[64] David Taubman and Michael Marcellin. *JPEG2000 Image Compression Fundamentals, Standards and Practice*. Science and Business Media. Springer, 2012. ISBN: 9781461507994.

[65] Morris Tenenbaum and Harry Pollard. *Ordinary Differential Equations*. Dover Publications, 1985. ISBN: 9780486649405.

[66] Darko Veberič. "Lambert W function for applications in physics." In: *Computer Physics Communications* 183.12 (2012), pp. 2622–2628.

[67] I. M. Vinogradov. *Elements of Number Theory*. Dover Publications, 2003. ISBN: 978-0-486-49530-9.

[68] Wolfgang Sartorius von Waltershausen. *Gauss zum Gedächtnis*. Reprinted 2012. Edition am Gutenbergplatz Leipzig, 1856. ISBN: 9783937219578.

[69] Eric Weisstein. *Elliptic Integral*. From Mathworld — A Wolfram Web Resource. URL: https://mathworld.wolfram.com/EllipticIntegral.html.

[70] David R. Wilkins. *On the Number of Prime Numbers less than a Given Quantity*. English. Clay Mathematics Institute. Dec. 1998. URL: https://www.claymath.org/wp-content/uploads/2023/04/Wilkins-translation.pdf.

[71] Robin J. Wilson. *Four Colors Suffice: How the Map Problem Was Solved*. Revised edition (November 10, 2013). Vol. 30. Princeton Science Library. Princeton University Press, 2013. ISBN: 978-0691158228.