MATH 350: Introduction to Computational Mathematics

Chapter I: Mathematical Modeling, Taylor Series, Floating-Point Numbers, and MATLAB

Greg Fasshauer

Department of Applied Mathematics
Illinois Institute of Technology

Spring 2011

Outline

1. Introduction
2. Mathematical Modeling
3. Taylor Series
4. Floating-Point Numbers
5. MATLAB

What is “computational mathematics”?

Possible answer:

Definition

“Computational mathematics is concerned with the study of algorithms (or numerical methods) for the solution of computational problems in science and engineering.”

Other names: *numerical analysis* or *scientific computing*

Desirable properties of algorithms:

- accuracy
- efficiency (speed and memory use)
- reliability/stability

Physical problem ⟶ mathematical model ⟶ approximate solution of problem (analytic or numeric)

Example

Growth of bacteria is often modeled using $\frac{dP}{dt} = kP$. The analytic solution is $P(t) = P_0 e^{kt}$. We can also solve the DE numerically (see later).

Why “approximate”?

- model usually idealized/simplified (e.g., infinite resources above; relativity theory applies to large scale problems, quantum mechanics to small scales → want unified theory (string theory?))
- modeling errors possible (e.g., different drag forces below)
- data obtained from physical problem could be inaccurate (measurement errors)
- possible roundoff errors in numerical solutions
- numerical algorithms can contain truncation errors
- programming errors

Physical Problem

A skydiver jumps out of an airplane (from sufficiently high altitude).
What is his *terminal velocity*? (picture below taken from [Prof. Kallend's website])

Mathematical Model

To get a handle on the velocity we use Newton's Second Law of Motion, $F = ma$. This implies that the acceleration $\frac{\mathrm{d}v}{\mathrm{d}t} = a = \frac{F}{m}$.
A very crude model would be to consider only the gravitational force $F_g = mg$, i.e., $\frac{\mathrm{d}v}{\mathrm{d}t} = \frac{F_g}{m} = \frac{mg}{m} = g$.
But then

$$v(t) = v_0 + gt,$$

and since we know about the concept of *terminal velocity* this cannot work.
A refined model also includes a drag force, $F_d = -cv$, due to air resistance. Here c is the *drag coefficient* (measured in kg/s), and v is the velocity.

This leads to the first model we will use:

$$\frac{\mathrm{d}v}{\mathrm{d}t}(t) = \frac{F_g + F_d(t)}{m} = g - \frac{c}{m}v(t). \tag{1}$$

Approximate Solutions

- The ODE
$$\frac{\mathrm{d}v}{\mathrm{d}t}(t) = g - \frac{c}{m}v(t)$$
is linear first-order (also separable) and has the analytical solution (assuming $v(0) = v_0 = 0$)
$$v(t) = \frac{gm}{c}\left(1 - \mathrm{e}^{-(c/m)t}\right). \tag{2}$$
Note: Terminal velocity is obtained by taking $t \to \infty$, so $v_T = \frac{gm}{c}$.
- The simplest method for obtaining a numerical solution of any first-order ODE $y'(t) = f(t, y)$ is Euler's method (approximate $y'(t) \approx \frac{y(t+h)-y(t)}{h}$, where h is some *stepsize* for the time step):
$$y'(t) = f(t, y) \quad \longrightarrow \quad y(t+h) \approx y(t) + hf(t, y)$$

Euler's Method

For our problem the general Euler formulation results in

$$v'(t) = \underbrace{g - \frac{c}{m} v(t)}_{=f(t,v)} \quad \longrightarrow \quad v(t+h) \approx v(t) + h\left(g - \frac{c}{m} v(t)\right).$$

In algorithmic form we have

$$v_{n+1} = v_n + h\left(g - \frac{c}{m} v_n\right), \quad n = 0, 1, 2, \ldots,$$

where h is the stepsize, $v_n = v(t_n)$ with $t_n = nh$, and we assume $v_0 = 0$.

See MATLAB example `SkydiveDemo.m`

Improved Mathematical Model

The dependence of the drag force due to air resistance is actually proportional to the square of the velocity, so $F_d = -\tilde{c}v^2$. Here $\tilde{c}$ is now a different *drag coefficient* (measured in kg/m).

This leads to the second and improved model we will use:

$$\frac{\mathrm{d}v}{\mathrm{d}t}(t) = \frac{F_g + F_d(t)}{m} = g - \frac{\tilde{c}}{m}v^2(t), \quad v(0) = v_0 = 0. \tag{3}$$

- This ODE is **nonlinear** first-order (but still separable). Its analytical solution is (since $\int \frac{\mathrm{d}x}{a^2-x^2} = \frac{1}{a}\tanh^{-1}(\frac{x}{a})$ or $\frac{1}{2a}\ln\left|\frac{x+a}{x-a}\right|$, depending on which table/program you consult)

$$v(t) = \sqrt{\frac{gm}{\tilde{c}}}\tanh\left(\sqrt{\frac{g\tilde{c}}{m}}t\right) = \sqrt{\frac{gm}{\tilde{c}}}\frac{e^{2\sqrt{\frac{g\tilde{c}}{m}}t} - 1}{e^{2\sqrt{\frac{g\tilde{c}}{m}}t} + 1}. \tag{4}$$

The terminal velocity is again obtained for $t \to \infty$, so $v_T = \sqrt{\frac{gm}{\tilde{c}}}$.

Improved Mathematical Model (cont.)

- A corresponding numerical solution via Euler's method is given in algorithmic form as

$$v_{n+1} = v_n + h\left(g - \frac{\tilde{c}}{m}(v_n)^2\right), \quad n = 0, 1, 2, \ldots,$$

where h is the stepsize, and $v_n = v(t_n)$ with $v_0 = 0$ as before.

See the MATLAB example `Skydive2Demo.m`

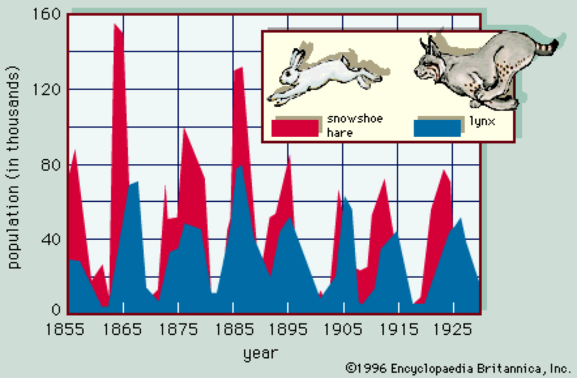
Remark

Note how simple the change in Euler's method is (just square the v-term in `Skydive.m`*), and compare this to the extra effort that is needed to solve the nonlinear ODE analytically.*

Physical Problem

According to records of the Hudson Bay Company, snowshoe hares and Canadian lynx populations have fluctuated as in the figure below

(see also [Marty ’95, Zhang et al. ’07] according to which this situation is **not** a predator-prey problem)



©1996 Encyclopaedia Britannica, Inc.

Mathematical Model

We treat lynx as predators and hares as prey and model their dependence by a Lotka-Volterra system

$$\begin{aligned} \frac{\mathrm{d}H(t)}{\mathrm{d}t} &= aH(t) - bH(t)L(t) \\ \frac{\mathrm{d}L(t)}{\mathrm{d}t} &= -cL(t) + dH(t)L(t) \end{aligned} \tag{5}$$

Here t denotes time, H population of hares, L population of lynx,

- $a = 0.5$ denotes birth rate of hares
- $b = 0.02$ denotes death rate of hares (depends on interaction with lynx “how good are lynx at killing hares”)
- $c = 0.4$ denotes death rate of lynx
- $d = 0.004$ denotes birth rate of lynx (depends on interaction with hares “how well do hares feed lynx”)

Approximate Solution

- Note that here an analytical solution is not available
- The only way to solve these coupled nonlinear ODEs is via a numerical method

Again, the simplest numerical method for first-order IVPs is Euler's method. Here

$$\begin{aligned}
\frac{\mathrm{d}H(t)}{\mathrm{d}t} &= aH(t) - bH(t)L(t) \quad \rightarrow \quad H_{n+1} = H_n + h\left(aH_n - bH_nL_n\right) \\
\frac{\mathrm{d}L(t)}{\mathrm{d}t} &= -cL(t) + dH(t)L(t) \quad \rightarrow \quad L_{n+1} = L_n + h\left(-cL_n + dH_nL_n\right)
\end{aligned}$$

with H_0 and L_0 the initial populations.

This is now a *system of ODEs*, but the MATLAB code is the same (see `LynxHareDemo.m`)

Projectile Motion

This example is discussed at

http://blog.wolfram.com/2010/09/27/do-computers-dumb-down-math-education/

Load `matheducation.nb` into Mathematica and play with it!

The TED talk mentioned in the document is here:

http://www.ted.com/talks/lang/eng/conrad_wolfram_teaching_kids_real_math_with_computers.html

From YouTube

Modeling Summary

There are many other kinds of mathematical modeling situations such as

- *data fitting* (e.g., find the best approximation – from a certain linear/nonlinear function class – to given measurement data)
- *parameter estimation* (e.g., find the best parameters for one of the models used earlier – drag coefficient, birth/death rate, etc.)
- *statistical/probabilistic modeling* (e.g., non-deterministic models in finance or weather prediction)
- *discrete modeling* (e.g., determining the best location of a fire department or hospital)
- *geometric modeling* (e.g., used for CAD systems)
- *asymptotic modeling* (focus on extreme or limiting cases, can usually be done analytically)

An entertaining overview of the field of mathematical modeling is provided by Charlie's activities on the TV show *NUMB3RS*.

Modeling Summary (cont.)

Remark

Even if an analytical solution is available for a (simple) mathematical model, perhaps a numerical method can be used to solve a more realistic (and more complicated) model.

For example, the skydiving model could be further improved by including a gravitational “constant” g that depends on the altitude x according to Newton’s inverse square law of gravitational attraction

$$g(x) = g(0)\frac{R^2}{(R+x)^2},$$

where $R \approx 6.37 \times 10^6(\mathrm{m})$ denotes the earth’s radius, and $g(0) = 9.81(\mathrm{m}/\mathrm{s}^2)$ denotes the values of the gravitational constant at the earth’s surface (see Chapter 7).

Why do we need to approximate functions?

Since many “simple” functions are difficult to evaluate without a calculator, certain approximation methods were developed early on to aid in this task.

One of the simplest (and most useful) is approximation by Taylor polynomials.

The central idea is to match a given function locally by some (low-degree) polynomial, and then evaluate this polynomial instead.

Example

Match $f(x) = \sqrt{x}$ at $x_0 = 1$ by a quadratic polynomial, i.e., find constants a_0, a_1, a_2 such that

$$p_2(x) = a_0 + a_1 x + a_2 x^2 \approx f(x) \tag{6}$$

for values of x near $x_0 = 1$.

◂ Return

Solution

We will determine the coefficients a_0, a_1, a_2 by matching derivatives of f at $x_0 = 1$, i.e., we will enforce (3 conditions for 3 coefficients)

$$\begin{aligned} p_2(1) &= f(1) = 1 \\ p_2'(1) &= f'(1) = \frac{1}{2} \\ p_2''(1) &= f''(1) = -\frac{1}{4} \end{aligned}$$

since we know $f'(x) = \frac{1}{2\sqrt{x}}$, $f''(x) = -\frac{1}{4x^{3/2}}$.

In fact, in many cases we will not actually know the functions f, f', f'', etc., but only their values at the specified point.
Note that this is not the most efficient way to obtain the Taylor approximation (but it illustrates where it comes from).

Since our assumption ▸ (6) implies

$$\begin{aligned} p_2'(x) &= a_1 + 2a_2 x, \\ p_2''(x) &= 2a_2 \end{aligned}$$

we obtain a system of three linear equations in the three unknowns a_0, a_1 and a_2:

$$\begin{aligned} p_2(1) &= a_0 + a_1 + a_2 = 1 \\ p_2'(1) &= a_1 + 2a_2 = \frac{1}{2} \\ p_2''(1) &= 2a_2 = -\frac{1}{4}. \end{aligned}$$

Solving this triangular system we get $a_2 = -\frac{1}{8}$, $a_1 = \frac{3}{4}$, and $a_0 = \frac{3}{8}$ so that

$$p_2(x) = \frac{3}{8} + \frac{3}{4}x - \frac{1}{8}x^2.$$

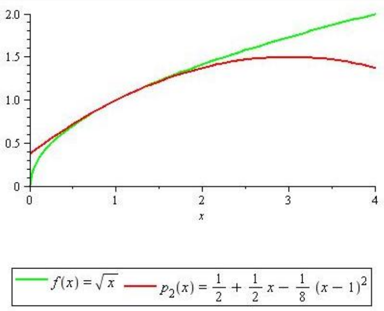
A more convenient representation of this polynomial is

$$p_2(x) = 1 + \frac{1}{2}(x-1) - \frac{1}{8}(x-1)^2$$

since this corresponds to

$$p_2(x) = f(1) + f'(1)(x-1) + \frac{f''(1)}{2}(x-1)^2$$

and shows how we use our “data” (the value of f and its derivatives at $x_0 = 1$).



Taylor Polynomials

In general, we can use Taylor's formula to obtain an n-th degree polynomial which matches the first n derivatives of f at some number x_0:

$$\begin{aligned} f(x) \approx p_n(x) &= f(x_0) + f'(x_0)(x-x_0) + \frac{f''(x_0)}{2}(x-x_0)^2 + \\ &\quad \frac{f'''(x_0)}{6}(x-x_0)^3 + \ldots + \frac{f^{(n)}(x_0)}{n!}(x-x_0)^n \\ &= \sum_{k=0}^{n} \frac{f^{(k)}(x_0)}{k!}(x-x_0)^k \end{aligned} \tag{7}$$

The polynomial in (7) is called the *n*-th degree Taylor polynomial for f at x_0.

Example

Let $f(x) = \mathrm{e}^x$ and find $p_n(x)$ for $x_0 = 0$.

Solution

Since $f^{(k)}(x) = \mathrm{e}^x$, $k = 0, 1, 2, \ldots, n$, we get

$$\begin{aligned} p_n(x) &= \sum_{k=0}^{n} \frac{f^{(k)}(x_0)}{k!}(x - x_0)^k \\ &= \sum_{k=0}^{n} \frac{\mathrm{e}^0}{k!}(x - 0)^k \\ &= \sum_{k=0}^{n} \frac{x^k}{k!} \\ &\approx \mathrm{e}^x = f(x). \end{aligned}$$

What is the error when approximating f by p_n?

Theorem (Taylor's Theorem)

Assume f is $n+1$ times continuously differentiable on an interval I containing the point x_0. Then there exists a number ξ between x and x_0 such that

$$f(x) = p_n(x) + \underbrace{\frac{f^{(n+1)}(\xi)}{(n+1)!}(x-x_0)^{n+1}}_{=E_{n+1}(x)}.$$

$E_{n+1}(x)$ is called the pointwise error at x or remainder at x.

The problem is that ξ is somewhere between x and x_0, but we don't know exactly where. Therefore we may obtain estimates for the error by examining certain "worst cases" of $E_{n+1}(x)$.

How to use Taylor's theorem?

Example

Let $f(x) = e^x$ and $x_0 = 0$. How accurate is $p_n(\frac{1}{2})$? More precisely, how large should n be so that the error $E_{n+1}(\frac{1}{2}) = \sqrt{e} - p_n(\frac{1}{2}) < 10^{-4}$?

Solution

From Taylor's theorem we have

$$E_{n+1}(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!}(x - x_0)^{n+1}$$

with ξ somewhere between x and x_0, i.e., $\xi \in [0, \frac{1}{2}]$.
We also know $f^{(n+1)}(x) = e^x$ for all x. Thus

$$E_{n+1}\left(\frac{1}{2}\right) = \frac{e^\xi}{(n+1)!}\left(\frac{1}{2} - 0\right)^{n+1} = \frac{e^\xi}{2^{n+1}(n+1)!}.$$

Solution (cont.)

We concluded above that $0 \le \xi \le \frac{1}{2}$, so we get (since the exponential function is increasing)

$$\frac{1}{2^{n+1}(n+1)!} \le E_{n+1}(\frac{1}{2}) = \frac{e^{\xi}}{2^{n+1}(n+1)!} \le \frac{e^{1/2}}{2^{n+1}(n+1)!}.$$

The whole point of the exercise is to approximate the value of $\sqrt{e} = e^{1/2}$, so we need to use a *known* upper bound above. Since we know that $2 < e < 3$, we can safely estimate

$$\frac{e^{1/2}}{2^{n+1}(n+1)!} < \frac{2}{2^{n+1}(n+1)!} = \frac{1}{2^{n}(n+1)!}$$

Solution (cont.)

Therefore, to ensure $E_{n+1}(\frac{1}{2}) < 10^{-4}$ we want to pick n such that

$$\frac{e^{1/2}}{2^{n+1}(n+1)!} < \frac{1}{2^n(n+1)!} \overset{!}{<} 10^{-4} \quad \Longrightarrow \quad 10^4 \overset{!}{<} 2^n(n+1)!.$$

This implies $n = 5$ (since $2^4 5! = 1920$ and $2^5 6! = 23040$).

Taylor Series

A Taylor series is obtained by taking the degree of the Taylor polynomial to infinity:

$$f(x) = \sum_{k=0}^{\infty} \frac{f^{(k)}(x_0)}{k!}(x - x_0)^k.$$

Of course, the equality holds only if the Taylor remainder $E_{n+1}(x)$ goes to zero as $n \to \infty$, i.e.,

$$\lim_{t \to \infty} E_{n+1}(x) = 0.$$

Note that the remainder depends on the point x of evaluation, and that in many cases the Taylor series will converge only for certain values of x near the point x_0 (within a ball/interval whose radius is called the radius of convergence). See the Maple worksheet `Taylor.mw`.

Alternate formulation of Taylor's theorem

For our purposes it will often be better to use Taylor's theorem in the following form:

Theorem

Assume f *is* $n+1$ *times continuously differentiable on an interval* I *containing both* x_0 *and* x_0+h *for some (small) number* h*. Then there exists a number* ξ *somewhere between* x_0 *and* x_0+h *such that*

$$f(x_0+h) = \sum_{k=0}^{n} \frac{f^{(k)}(x_0)}{k!} h^k + \frac{f^{(n+1)}(\xi)}{(n+1)!} h^{n+1}$$

Note that we get this formulation from the previous one by replacing x by x_0+h so that $x-x_0=h$.

In this new representation we can say

$$E_{n+1}(x_0) = \mathcal{O}(h^{n+1}), \quad \text{as } h \to 0,$$

which means $|E_{n+1}(x_0)| \leq C|h|^{n+1}$ for some constant C.

Remark

From the alternate form of Taylor's theorem we can get the important estimates

$$f(x+h) = f(x) + \mathcal{O}(h) \tag{8}$$
$$f(x+h) = f(x) + f'(x)h + \mathcal{O}(h^2). \tag{9}$$

Estimate (9) implies

$$f'(x) = \frac{f(x+h) - f(x)}{h} + \mathcal{O}(h),$$

which plays a crucial role in our understanding of many numerical methods (e.g., Euler's method).

Alternating series

Remark

The alternating series test from calculus may become useful if we need to estimate the error for a series whose terms have alternating signs.

Consider $\sum_{k=1}^{\infty}(-1)^k a_k$ *with* $a_k \geq 0$*. If the sequence* $\{a_k\}$ *is decreasing and* $\lim_{k\to\infty} a_k = 0$*, then the series converges. Moreover,*

$$E_{n+1} = \left| \underbrace{\sum_{k=1}^{\infty}(-1)^k a_k}_{=S} - \underbrace{\sum_{k=1}^{n}(-1)^k a_k}_{=S_n} \right| \leq a_{n+1},$$

i.e., the truncation error is bounded by the next (unused) term.

Most computer programming languages (such as C/C++/C#, Java, Fortran, or MATLAB) use floating-point arithmetic. Even though we usually don't have to worry much about this in everyday computing, it is good to have a basic understanding of floating-point numbers for those rare occasions when something unexpected happens.

Here is what might happen if we don't understand what we're doing.

First, we need to realize that the set of floating-point numbers is discrete:

- there are only finitely many of them,
- and they possess only finite precision.

Most technical computing environments (including MATLAB) use the IEEE standard for floating-point arithmetic. In particular, MATLAB uses the IEEE double-precision format[1] which uses a word length of 64 bits to represent a number (see also the details in Chapter 1.7 of [NCM]).

---

[1] and since MATLAB 7 also single-precision

Normalized Floating-Point Numbers

Numbers are represented as

$$x = \pm(1 + f) \cdot 2^e,$$

where $0 \leq f < 1$ is the fraction or mantissa, and the exponent $-1022 \leq e \leq 1023$ is an integer.

Of the 64 bits reserved to store floating-point numbers in the IEEE standard, f uses 52, e uses 11, and one bit is used to store the sign (positive or negative).

- Finite f implies finite precision (i.e., discrete spacing of floating point numbers),
- finite e implies finite range (there is a minimum and maximum representable number).

The IEEE Standard

The machine epsilon `eps` represents the distance from 1 to the next larger floating-point number and comes out to be 2^{-52} in the IEEE standard.

In the IEEE double-precision format we have

	binary	decimal
`eps`	2^{-52}	$2.2204 \cdot 10^{-16}$
`realmin`	2^{-1022}	$2.2251 \cdot 10^{-308}$
`realmax`	$(2 - \mathtt{eps}) \cdot 2^{1023}$	$1.7977 \cdot 10^{308}$

The machine epsilon defines the roundoff level, i.e., when following the IEEE standard, numbers can generally be represented with about 16 accurate decimal digits.

Exceptions: Numbers larger than `realmax` will cause *overflow*, while those smaller than `realmin` will lead to *underflow*. The number zero is also treated as an exception.

Example

Assume we have a computer that provides only 4 bits to represent floating-point numbers (1 for sign, 1 for fraction, 2 for exponent).
List all floating-point numbers that can be represented in this computer.

Solution

$t = 1$ bit for f: $\{0, 1\} \overset{\text{normalize}}{\Longrightarrow} f = \{0, 1\}/2^t = \{0, 1/2\}$

2 bits for e: $\{00, 01, 10, 11\}_2 = \{0, 1, 2, 3\}_{10} \overset{\text{center}}{\Longrightarrow} e = \{-2, -1, 0, 1\}$

So possible numbers, $x = \pm(1 + f) \cdot 2^e$, are:

$$\begin{array}{llll}
\pm(1+0)\cdot 2^{-2} & = \pm 1/4 & \pm(1+1/2)\cdot 2^{-2} & = \pm 3/8 \\
\pm(1+0)\cdot 2^{-1} & = \pm 1/2 & \pm(1+1/2)\cdot 2^{-1} & = \pm 3/4 \\
\pm(1+0)\cdot 2^{0} & = \pm 1 & \pm(1+1/2)\cdot 2^{0} & = \pm 3/2 \\
\pm(1+0)\cdot 2^{1} & = \pm 2 & \pm(1+1/2)\cdot 2^{1} & = \pm 3
\end{array}$$

Note the “hole around zero”.

`floatgui` with `t` $= 1$, `emin` $= -2$, `emax` $= 1$

A (perhaps surprising) weakness of the binary (or hexadecimal) computer representation of numbers is the representation of the decimal number $1/10$.

In fact we have,

$$\begin{aligned}\frac{1}{10} &= \frac{1}{2^4}+\frac{1}{2^5}+\frac{0}{2^6}+\frac{0}{2^7}+\frac{1}{2^8}+\frac{1}{2^9}+\frac{0}{2^{10}}+\frac{0}{2^{11}}+\frac{1}{2^{12}}+\ldots\\ &= \frac{1}{2^4}\left(1+\frac{1}{2}+\frac{0}{2^2}+\frac{0}{2^3}+\frac{1}{2^4}+\frac{1}{2^5}+\frac{0}{2^6}+\frac{0}{2^7}+\frac{1}{2^8}+\ldots\right)\\ &= \frac{1}{16}\left(1+\frac{9}{16}+\frac{9}{16^2}+\frac{9}{16^3}+\frac{9}{16^4}+\ldots\right)\end{aligned}$$

Thus, the decimal number $1/10$ has to be truncated on a digital computer. This leads to roundoff error.

See "disasters due to bad numerical computing".

Example

Solve the following linear system with MATLAB

$$\begin{aligned} 17x_1 + 5x_2 &= 22 \\ 1.7x_1 + 0.5x_2 &= 2.2. \end{aligned}$$

Solution

Note that the system is singular (since the second equation is just a multiple of the first), and has infinitely many solutions.
However, MATLAB offers a unique solution (see `RoundoffDemo.m`).
MATLAB is "tricked" by the fact that the multiplier 1.7/17=1/10, whose truncation produces numerically independent equations!

The system

$$\begin{aligned} x_1 + 2x_2 &= 2 \\ 2x_1 + 4x_2 &= 4 \end{aligned}$$

causes no such problems (see also `RoundoffDemo.m`).

Example

Evaluate $f(x) = \sqrt{x^2+1} - 1$ in MATLAB for $x = 10^{-n}$, $n = 0, 1, \ldots, 5$ using both double-precision and single-precision.

Solution

The “exact” answers (obtained in Maple with much higher precision) are

x	$\sqrt{x^2+1}$	$f(x)$
1	$\sqrt{2} = 1.4142135623730950488$	0.4142135623730950488
0.1	$\sqrt{1.01} = 1.0049875621120890270$	0.0049875621120890270
0.01	$\sqrt{1.0001} = 1.0000499987500624961$	0.0000499987500624961
0.001	$\sqrt{1.000001} = 1.0000004999998750001$	0.0000004999998750001
0.0001	$\sqrt{1.00000001} = 1.0000000049999999875$	0.0000000049999999875
0.00001	$\sqrt{1.0000000001} = 1.0000000000500000000$	0.0000000000500000000

Use `LossOfSignificanceDemo.m`.

How can we prevent this?

Solution

We rewrite the expression $f(x)$ *before we code it*:

$$\begin{aligned} f(x) &= \sqrt{x^2+1}-1 \\ &= \left(\sqrt{x^2+1}-1\right)\frac{\sqrt{x^2+1}+1}{\sqrt{x^2+1}+1} \\ &= \frac{x^2+1-1}{\sqrt{x^2+1}+1} \\ &= \frac{x^2}{\sqrt{x^2+1}+1} \end{aligned}$$

Continue `LossOfSignificanceDemo.m` (can even improve double-precision this way).

MATLAB Guides

Quite a bit of introductory material is posted online at http://math.iit.edu/∼fass/100.html.

This includes

- Getting Started in MATLAB (Some very basic info to get you off the ground — similar to the following slides)
- MATLAB's built-in help: *Video*, *Demos*, or *Getting Started*
- The introductory MATLAB scripts on the handouts page for this class

A *Very Elementary MATLAB Tutorial* is available directly from The MathWorks.

- MATLAB is widely used in many areas of applied mathematics and engineering.
- MATLAB stands for MATrix LABoratory and the software uses vectors and matrices as basic building blocks.
- We have to learn to think “the MATLAB way” if we want to take full advantage.
- In addition to its computational engine MATLAB provides a powerful graphical interface that allows us to produce both 2D and 3D plots.
- In addition to its interactive mode, MATLAB is also one of the easiest programming languages for solving mathematical problems.
- MATLAB’s basic capabilities can be extended by calling functions defined in additional *toolboxes*.

- All IIT computer labs should have MATLAB installed. You can also purchase the Student Version for about $100.
- Usually we use MATLAB via its windows-based interface, and start it like any other program.
- Important MATLAB windows:
  - Command window: where you work in interactive mode (at the » command prompt), or run programs (M-files)
  - Editor window: where you write your program code, and then save it to your hard drive (other text editors are also allowed)
  - Help window: where you can get online help (can also type `help` or `help <command name>` at the command prompt)
- Other MATLAB windows:
  - Command History window
  - Current Directory window
  - Workspace window (provides information about all the variables in use)

Other important things

- In an emergency (such as a run-away loop) you can interrupt MATLAB by typing Ctrl-C (note that sometimes it may take MATLAB a while to “come back” from heavy calculations).
- Once you have finished your work you can exit MATLAB by either typing `quit` at the prompt (») in the Command window, by going to the File→Exit menu, or by closing the Command window in the usual way.
- In addition to the windows-based interface with all its bells and whistles MATLAB also has a command-line interface that can be invoked by using additional switches such as `matlab -nodesktop`.

- While you can enter individual MATLAB commands interactively in the Command window, you will often want to combine a sequence of commands into a program (also called a script file or function file).
- You need to write such programs in a separate editor (see above).
- If the Editor does not open by itself when you start MATLAB you can invoke it via the File→New→M-File menu (for a new file) or File→Open menu (for an existing file).
- Basic use of the editor is straightforward.
- Many advanced features are also available (such as adding breakpoints to your code for debugging purposes).

How to save and run a MATLAB program — M-file

While typing your code in the editor, no commands will be performed!
In order to run a program do the following:

- In the Editor save your code as an M-file with some filename you pick. (MATLAB should automatically add the `.m` extension that is required for the file to be recognized as a MATLAB program file).
- Go to the Command window. Make sure the folder your Command window is looking at is the same one you saved your program in!
- Run the program by entering its name (without the `.m` extension) at the command prompt.
- If your code contained an error, MATLAB will interrupt execution of the program and provide you with an error message. You can click on the error message, and will be taken to the corresponding place in the code in the Editor.

References I

T. A. Driscoll.
Learning MATLAB.
SIAM, Philadelphia, 2009.

D. J. Higham and N. J. Higham.
MATLAB Guide.
SIAM (2nd ed.), Philadelphia, 2005.

C. Moler.
Numerical Computing with MATLAB.
SIAM, Philadelphia, 2004.
Also http://www.mathworks.com/moler/index_ncm.html.

C. Moler.
Experiments with MATLAB.
Free download at http://www.mathworks.com/moler/exm/chapters.html.

S. Marty.
The lynx and the hare.
Canadian Geographic Magazine, Sept./Oct. (1995), 28–37.

References II

Z. Zhang, Y. Tao, and Z. Li.
Factors affecting hare-lynx dynamics in the classic time series of the Hudson Bay Company, Canada.
Climate Research **34** (2007), 83–89.

The MathWorks.
MATLAB 7: Getting Started Guide.
http://www.mathworks.com/access/helpdesk/help/pdf_doc/matlab/getstart.pdf.

M. Gockenbach.
Practical Introduction to MATLAB(for Vesrion 5).
http://www.math.mtu.edu/∼msgocken/intro/intro.html.

J. Kallend.
John Kallend's Skydiving Stuff.
http://www.iit.edu/∼kallend/skydive/.