2018

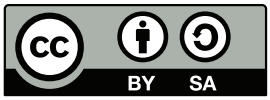
Computational Methods in Civil Engineering

Nir Krakauer
CUNY City College

Lecture notes: CE 33500, Computational Methods in Civil Engineering

Nir Krakauer

June 1, 2018



1 Introduction: Mathematical models, numerical methods, and Taylor series

- Engineering problem → Mathematical model → Numerical method

  Approximations/uncertainties in each step (build in factors of safety and conservative assumptions to make design robust to 'small' errors)

  Example: Beam bending under loading

  Example: Contraction of steel by cooling (Kaw 01.01)

- "Numerical methods" solve math problems, often approximately, using arithmetic operations (add, subtract, multiply, divide) and logic operations (such as checking whether one quantity x is greater than another quantity y – returns true/false). They therefore permit math problems relevant to engineering to be easily handled by computers, which can carry out arithmetic and logic operations very fast.

  On the other hand, "analytical methods", like those learned in calculus, give exact solutions, but are not available for many math problems of engineering interest (such as complicated integrals and differential equations that don't have integration formulas). Computers can help with using such methods also.

- The term "computational methods" may include numerical methods plus setting up the mathematical model as well as representing the model and any needed numerical methods for solving it as a computer program

- Many numerical methods are based on approximating arbitrary functions with polynomials using Taylor's theorem: If f is a function that has $k+1$

continuous derivatives over the interval between a and x, then

$$f(x) = f(a) + (x-a)f'(a) + \frac{(x-a)^2}{2}f''(a) + \ldots + \frac{(x-a)^k}{k!}f^{(k)}(a) + R_{k+1},$$

where the "remainder term" R_{k+1} has the form

$$R_{k+1} = \frac{(x-a)^{k+1}}{(k+1)!}f^{(k+1)}(\zeta)$$

for some ζ between a and x.

We typically don't know the value of R_{k+1} exactly, since the exact value of ζ isn't easily determined. R_{k+1} represents the error in approximating $f(x)$ by the first $k+1$ terms in the right-hand side, which are known.

Taylor's theorem can also be written in a series form:

$$f(x) = \sum_{i=0}^{k} \frac{(x-a)^i}{i!}f^{(i)}(a) + R_{k+1}.$$

If f is infinitely differentiable between a and x, we can write

$$f(x) = \sum_{i=0}^{\infty} \frac{(x-a)^i}{i!}f^{(i)}(a)$$

with an infinite series and no remainder term.

The Taylor series when $a = 0$ is also known as the Maclaurin series.

If we define $h \equiv x - a$, we can also write

$$f(a+h) = f(a) + hf'(a) + \frac{h^2}{2}f''(a) + \ldots + \frac{h^k}{k!}f^{(k)}(a) + R_{k+1}$$

with

$$R_{k+1} = \frac{h^{k+1}}{(k+1)!}f^{(k+1)}(\zeta)$$

for some ζ between a and $a+h$, or more compactly

$$f(a+h) = \sum_{i=0}^{k} \frac{h^i}{i!}f^{(i)}(a) + R_{k+1}$$

or as an infinite series

$$f(a+h) = \sum_{i=0}^{\infty} \frac{h^i}{i!}f^{(i)}(a).$$

- Taylor's theorem can be applied to find series expansions for functions in terms of polynomials:

$$e^x = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \dots$$

$$\sin(x) = x - \frac{x^3}{6} + \frac{x^5}{120} - \dots$$

$$\cos(x) = 1 - \frac{x^2}{2} + \frac{x^4}{24} - \dots$$

$$\frac{1}{1-x} = 1 + x + x^2 + x^3 + \dots$$

(converges for $|x| < 1$)

$$\log(x+1) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \dots$$

(converges for $|x| < 1$)

(In this class, trigonometric functions are always for the argument x in radians, and log means the natural logarithm [base e].)

- Newton's method for estimating x where some function f is equal to 0 can be derived from Taylor's theorem:

$$f(x) = f(a) + (x-a)f'(a) + R_2 \rightarrow$$

$$0 = f(a) + (x-a)f'(a) + R_2 \rightarrow x = a - \frac{f(a)}{f'(a)} - \frac{R_2}{f'(a)} \approx a - \frac{f(a)}{f'(a)},$$

with the approximation good if R_2 is small (small h or small second derivative $f^{(2)}$)

Example: Apply Newton's method to estimate the square root of 26 iteratively. Set $f(x) = x^2 - 26 = 0$, start with $a_0 = 5$, get $a_1 = a_0 - \frac{f(a_0)}{f'(a_0)} = 5.1, a_2 = a_1 - \frac{f(a_1)}{f'(a_1)} = 5.0990196078\dots, a_3 = a_2 - \frac{f(a_2)}{f'(a_2)} = 5.09901951359\dots$, giving a series of increasingly accurate numerical estimates of $\sqrt{26}$

- Centered finite difference for estimating the derivative of some function f at x :

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + R_{3,+}$$

$$f(x-h) = f(x) - hf'(x) + \frac{h^2}{2}f''(x) - R_{3,-}$$

$$f(x+h) - f(x-h) = 2hf'(x) + R_{3,+} + R_{3,-}$$

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} - \frac{R_{3,+} + R_{3,-}}{2h} \approx \frac{f(x+h) - f(x-h)}{2h}$$

where

$$R_{3,+} = \frac{h^3}{6} f^{(3)}(\zeta_+), R_{3,-} = \frac{h^3}{6} f^{(3)}(\zeta_-)$$

for some ζ_+ between x and $x+h$ and ζ_- between $x-h$ and x.

Example: To estimate the derivative of $f(x) = e^x$ at $x = 1$, we can use the approximation derived here with $h = 0.1 : f'(1) \approx \frac{f(1.1)-f(0.9)}{2 \cdot 0.1} = 2.72281456\ldots$

The finite-difference approximation generally becomes more accurate as we make h closer to zero, because the remainder terms get smaller

- Euler's method for numerically approximating the value of $y(x)$ given the differential equation $y'(x) = g(x, y(x))$ and the initial value $y(a) = y_0$:

$$y(x) = y(a) + (x-a)y'(a) + R_2 = y(a) + (x-a)g(a, y_0) + R_2 \approx y(a) + (x-a)g(a, y_0)$$

Example: If we know that $y(0) = 1, y'(x) = y$, we can estimate $y(0.4)$ as $y(0.4) \approx 1 + 0.4 \cdot 1 = 1.4$. To get a more accurate estimate, we can reduce the size of the left-out remainder terms by solving the problem in two steps: $y(0.2) \approx y_1 = 1 + 0.2 \cdot 1 = 1.2$ and $y(0.4) \approx y_2 = 1.2 + 0.2 \cdot 1.2 = 1.44$, or even in four steps: $y(0.1) \approx y_1 = 1 + 0.1 \cdot 1 = 1.1, y(0.2) \approx y_2 = 1.1 + 0.1 \cdot 1.1 = 1.21, y(0.3) \approx y_3 = 1.21 + 0.1 \cdot 1.21 = 1.331, y(0.4) \approx y_4 = 1.331 + 0.1 \cdot 1.331 = 1.4641$. In this case, the true answer can be found analytically to be $e^{0.4} \approx 1.4918$.

2 Error sources and control

- To solve important problems reliably, need to be able to identify, control, and estimate sources of *error*

- Size of error: If the true answer is x^* and the numerical answer is x,

  Absolute error: $|x - x^*|$

  Fractional error:

$$\frac{|x - x^*|}{|x^*|}$$

  Usually for an engineering calculation we want the fractional error introduced during the solution step to be small, say $< 10^{-6}$; sometimes, we also want the absolute error to be under some threshold.

- Approximate (estimated) error: needed in practice, because we generally don't know the exact answer x^*

  There are a few ways to estimate approximately how big the error might be. For example, if we have two different numerical approximations x_1, x_2 of x^*, we can write

  Approximate absolute error: $|x_1 - x_2|$

Approximate fractional error:

$$\frac{|x_1 - x_2|}{|x^+|}$$

where x^+ is the best available approximation (whichever of x_1 or x_2 is believed to be more accurate than the other, or their average if both are believed to be equally accurate)

- Error types in solving math problems: gross error, roundoff error, truncation error

- *Gross error*: Entering the wrong number, using the wrong commands or syntax, incorrect unit conversion ...

  This does happen in engineering practice (but isn't looked on kindly).

  Example: Mars Climate Orbiter spacecraft lost in space in 1999 because Lockheed Martin calculated rocket thrust in lb while NASA thought the results were in N

  Detection methods: Know what answer to expect and check if what you got makes sense; try to find the answer for a few simple cases where you know what it should be (programs to be used to solve important problems should have a *test suite* to do this); compare answers with others who worked independently

- *Roundoff error* in floating-point computation: results from the fact that computers only carry a finite number of decimal places – about 16 for Matlab's default IEEE double-precision format (*machine epsilon* [`eps`] or *unit roundoff* $\epsilon \approx 10^{-16}$)

  Example: `0.3/0.1 - 3` is nonzero in Matlab, but is of order ϵ

  The double-precision format uses 64 bits (binary digits) to represent each number – 1 bit for the sign, 11 bits for the base-2 exponent (which can range between -1022 and 1023), and 52 bits for the *significand* or *mantissa*, which is interpreted as binary $1.bbbb\ldots$. It can represent number magnitudes between about 10^{-308} and 10^{308} (beyond that, numbers *overflow* to infinity or *underflow* to zero)

  Multiplication and division under roundoff are subject to maximum fractional error of ϵ

  Addition of two similar numbers also has maximum roundoff error similar to ϵ, but subtraction of two numbers that are almost the same can have roundoff error much bigger than ϵ (cancellation of significant digits)

  A non-obvious example of subtractive cancellation: computing e^x using the first number of terms in the Taylor series expansion $e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!}$ (In Matlab, `sum(x .^ (0:imax) ./ factorial(0:imax))`) when x is a large negative number, say -20

Another example: Approximating derivatives with finite difference formulas with small increments h

In general, math operations which incur large roundoff error (like subtracting similar numbers) tend to be those that are *ill-conditioned*, meaning that small fractional changes in the numerical values used can change the output by a large fractional amount

Examples of ill-conditioned problems: solving linear systems when the coefficient matrix has a large condition number; trigonometric operations with a large argument (say, $\sin(10^{100})$); the quadratic formula when b is much larger than a and c; finding a polynomial that interpolates a large number of given points

Mitigation: Use extended precision; reformulate problems to avoid subtracting numbers that are very close together

- *Truncation error*: Running an iterative numerical algorithm for only a few steps, whereas convergence to the exact answer requires theoretically an infinite number of steps

  Often can be thought of as only considering the first few terms in the Taylor series

  "Steps" can be terms in the Taylor series, iterations of Newton's method or bisection for root finding, number of subdivisions in the composite trapezoid rule or Simpson rule for numerical integration, etc.

  Detection: Estimate truncation error (and roundoff error) by comparing the results of different numerical methods, or the same method run for different numbers of steps on the same problem

  Mitigation: Run for more steps (at the cost of more computations); use a more accurate numerical method, if available

- Non-mathematical error sources should also be considered in formulating and interpreting a problem, including measurement errors, variations in material properties, uncertainties as to the loadings that will be faced, simplifications in constructing math models .... They may have large effects, but will be discussed more in other courses

  Models are always only approximations (like maps) that hopefully represent the main aspects of interest in an engineering problem

3 Linear systems

3.1 Properties

- Linear systems arise directly from engineering problems (stresses, circuits, pipes, traffic networks ...) as well as indirectly via numerical methods, for example finite difference and finite element methods for solving differential equations

- Any system of m linear equations in n unknowns $x_1, x_2, x_3, \ldots x_n$ (where each equation looks like $a_1x_1 + a_2x_2 + a_3x_3 + \ldots + a_nx_n = b$, with different a and b coefficients) can be written in a standard matrix form $\mathbf{Ax} = \mathbf{b}$, where

  $\mathbf{A}$ is the $m \times n$ matrix with each row containing the (known) coefficients of the unknowns in one linear equation,

  $\mathbf{x}$ is the $n \times 1$ vector of unknowns,

  $\mathbf{b}$ is the $m \times 1$ vector of (known) constant terms in the equations.

  Can also write the system as an $m \times (n+1)$ "augmented matrix" $\mathbf{A}|\mathbf{b}$ (with $\mathbf{x}$ implied)

- Does a solution exist? For square systems ($m = n$), there is a unique solution equal to $\mathbf{A}^{-1}\mathbf{b}$ if $\mathbf{A}$ has an inverse.

  If $\mathbf{A}$ has no inverse (is *singular*), then there will be either no solution or infinitely many solutions.

  $\mathbf{A}$ has no inverse when the equations of a linear system with $\mathbf{A}$ as the coefficient matrix are not linearly independent of each other. E.g.: the 2 given equations in 2 unknowns are $x_1 + 2x_2 = 3, 2x_1 + 4x_2 = 6$ so that the coefficient matrix is

$$\begin{pmatrix} 1 & 2 \\ 2 & 4 \end{pmatrix}$$

- Solution accuracy measures

  Error size: $||\mathbf{x} - \mathbf{x}^*||$, where $\mathbf{x}^*$ is the true solution and $\mathbf{x}$ is computed;

  Residual size: $||\mathbf{Ax} - \mathbf{b}||$ (usually easier to calculate than the error)

- Note: *norms*, denoted by $||\mathbf{x}||$, measure the size of a vector or matrix (analogous to absolute value, $|x|$, for a scalar)

  2-norm of a vector $\mathbf{v}$:

$$||\mathbf{v}||_2 = \sqrt{\sum_i v_i^2}$$

  2-norm of a matrix $\mathbf{A}$: $||\mathbf{A}||_2$ is defined as the maximum $||\mathbf{Av}||_2$ over all vectors $\mathbf{v}$ such that $||\mathbf{v}||_2 = 1$.

  Similar in magnitude to the largest element of the matrix

- If a matrix $\mathbf{A}$ has an inverse but is very close to a non-invertible matrix, then

  In exact math, any linear system $\mathbf{Ax} = \mathbf{b}$ has a unique solution

  However, a small change in $\mathbf{A}$ or $\mathbf{b}$ can change the solution $\mathbf{x}$ by a lot (ill-conditioning)

The equations can become linearly dependent if we change the coefficients slightly

Another way of putting this is that there are vectors $\mathbf{x}$ far from the true solution $\mathbf{x}^*$ (large error compared to $\mathbf{x}^*$) where $\mathbf{Ax}$ is nevertheless close to $\mathbf{b}$ (small residual).

Roundoff errors in the computation have an effect similar to perturbing the coefficient matrix $\mathbf{A}$ or $\mathbf{b}$ slightly, potentially giving a very wrong solution (large error), although it will generally have a small residual.

We measure how close a matrix is to a singular matrix by the condition number, defined as $\text{cond}(\mathbf{A}) = ||\mathbf{A}|| \times ||\mathbf{A}^{-1}||$, which ranges from 1 to infinity. A matrix with no inverse has infinite condition number. The condition number gives the factor by which small changes to the coefficients, due to measurement or roundoff error, can multiply to give a larger error in $\mathbf{x}$. A large condition number means that linear systems with this coefficient matrix will be ill-conditioned – changing the numerical values by a small fraction could change the solution by a large percentage.

3.2 Solution methods

- An *upper triangular* matrix has only zero entries for columns less than the row number, i.e. $A_{i,j} = 0$ whenever $j < i$. ($A_{i,j}$ [or A_{ij}] here refers to the element in row i and column j of the matrix $\mathbf{A}$.)

- If the coefficient matrix of a square linear system $\mathbf{A}$ is upper triangular, then it can generally be solved for the unknown x_i by *back substitution*:

  $x_n = b_n / A_{n,n}$

  for $i = n-1, n-2, \ldots$ to 1

  $x_i = \left(b_i - \sum_{j=i+1}^{n} A_{i,j} x_j\right) / A_{i,i}$

- Similarly, a *lower triangular* matrix has only zero entries for columns more than the row number, i.e. $A_{i,j} = 0$ whenever $j > i$.

- If the coefficient matrix of a square linear system $\mathbf{A}$ is lower triangular, then it can generally be solved for the unknown x_i by *forward substitution*:

  $x_1 = b_1 / A_{1,1}$

  for $i = 2, 3, \ldots$ to n

  $x_i = \left(b_i - \sum_{j=1}^{i-1} A_{i,j} x_j\right) / A_{i,i}$

- Any square linear system can generally be transformed into an upper triangular one with the same solution through *Gauss elimination*:

  for $j = 1, 2, \ldots$ to n

  $p = A_{j,j}$ (pivot element; j is the pivot row or pivot equation)

  for $i = j+1, j+2, \ldots$ to n

$M = A_{i,j}/p$ (multiplier)
$A_{i,:} = A_{i,:} - M \times A_{j,:}$
$b_i = b_i - M \times b_j$

- Example: to solve

$$\begin{pmatrix} -1 & 2 & 2 \\ 1 & 1 & 1 \\ 1 & 3 & 2 \end{pmatrix} \mathbf{x} = \begin{pmatrix} 8 \\ 1 \\ 4 \end{pmatrix}$$

for $\mathbf{x}$:

$$\left(\begin{array}{ccc|c} -1 & 2 & 2 & 8 \\ 1 & 1 & 1 & 1 \\ 1 & 3 & 2 & 4 \end{array}\right) \text{(augmented matrix)} \rightarrow \left(\begin{array}{ccc|c} -1 & 2 & 2 & 8 \\ 0 & 3 & 3 & 9 \\ 0 & 5 & 4 & 12 \end{array}\right) \rightarrow \left(\begin{array}{ccc|c} -1 & 2 & 2 & 8 \\ 0 & 3 & 3 & 9 \\ 0 & 0 & -1 & -3 \end{array}\right)$$

and employ back substitution to get

$$\mathbf{x} = \begin{pmatrix} -2 \\ 0 \\ 3 \end{pmatrix}$$

- Gauss elimination requires a total of $\frac{1}{3}n^3$ each subtraction and multiplication operations; forward and back substitution require only some n^2.
- Gauss elimination would fail if any of the pivots p is zero, even when the linear system actually has a unique solution. If a pivot is not zero but very small, the Gauss elimination wouldn't fail in exact math, but any roundoff error from previous steps could become large when we divide by the very small p.
- Solution: *row pivoting* – rearrange the rows (equations) so that the pivot is as large as possible (in absolute value). This avoids dividing by a pivot element that is zero or small in absolute value.
- Algorithm for Gauss elimination with row pivoting:

  for $j = 1, 2, \ldots$ to n

  Row pivoting:

  Find row l such that $|A_{l,j}|$ is the maximum of all the elements in column j that are at or below row j.

  Interchange rows l and j in the matrix $\mathbf{A}$. (This is the same as multiplying $\mathbf{A}$ (and then $\mathbf{b}$) by a permutation matrix $\mathbf{Q}$, which is an identity matrix with rows l and j interchanged.)

  Interchange elements l and j in the vector $\mathbf{b}$.

$p = A_{j,j}$ (pivot element; j is the pivot row)
for $i = j+1, j+2, \ldots$ to n
$M = A_{i,j}/p$ (multiplier)
$A_{i,:} = A_{i,:} - M \times A_{j,:}$
$b_i = b_i - M \times b_j$

- Example:

$$\left(\begin{array}{ccc|c} 0 & 2 & 2 & -2 \\ 1 & -2 & -1 & 2 \\ 4 & -2 & 4 & 4 \end{array}\right) \rightarrow \left(\begin{array}{ccc|c} 4 & -2 & 4 & 4 \\ 1 & -2 & -1 & 2 \\ 0 & 2 & 2 & -2 \end{array}\right) \rightarrow \left(\begin{array}{ccc|c} 4 & -2 & 4 & 4 \\ 0 & 2 & 2 & -2 \\ 0 & -\frac{3}{2} & -2 & 1 \end{array}\right) \rightarrow \left(\begin{array}{ccc|c} 4 & -2 & 4 & 4 \\ 0 & 2 & 2 & -2 \\ 0 & 0 & -\frac{1}{2} & -\frac{1}{2} \end{array}\right)$$

Giving

$$\mathbf{x} = \begin{pmatrix} -1 \\ -2 \\ 1 \end{pmatrix}$$

- LU (=lower/upper triangular) decomposition: A matrix $\mathbf{A}$ can be written as the matrix product $\mathbf{LU}$, where $\mathbf{L}$ is unit lower triangular (*unit* meaning that all elements on the *main diagonal* – where the row and column numbers are the same – are equal to 1) and $\mathbf{U}$ is upper triangular.

- The LU decomposition can be obtained based on Gauss elimination, as follows:

for $j = 1, 2, \ldots$ to n
$p = A_{j,j}$ (pivot element; j is the pivot row)
for $i = j+1, j+2, \ldots$ to n
$M = A_{i,j}/p$ (multiplier)
Save M as $L_{i,j}$
$A_{i,:} = A_{i,:} - M \times A_{j,:}$

Then $\mathbf{U}$ is the transformed $\mathbf{A}$ (which is upper triangular) and $\mathbf{L}$ is the matrix built up from the multipliers M, with ones added along the main diagonal.

Example:

$$\begin{pmatrix} -1 & 2 & 2 \\ 1 & 1 & 1 \\ 1 & 3 & 2 \end{pmatrix} \rightarrow \begin{pmatrix} -1 & 2 & 2 \\ -1| & 3 & 3 \\ -1| & 5 & 4 \end{pmatrix} \rightarrow \begin{pmatrix} -1 & 2 & 2 \\ -1| & 3 & 3 \\ -1 & 5/3| & -1 \end{pmatrix}$$

This gives us the factors in the form $(\mathbf{L}\backslash\mathbf{U})$, which can be expanded to

$$\mathbf{L} = \begin{pmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ -1 & 5/3 & 1 \end{pmatrix}, \mathbf{U} = \begin{pmatrix} -1 & 2 & 2 \\ 0 & 3 & 3 \\ 0 & 0 & -1 \end{pmatrix}$$

You can then check that $\mathbf{LU}$ is in fact equal to the original matrix.

- LU decomposition is useful in solving linear systems because once we have the decomposition of a matrix $\mathbf{A}$, we can solve any system with the coefficient matrix $\mathbf{A}$ using forward and back substitution and only around n^2 operations instead of n^3:

  Given $\mathbf{Ax} = \mathbf{b}$ with unknown $\mathbf{x}$ and the LU decomposition $\mathbf{A} = \mathbf{LU}$,

  Solve the lower triangular system $\mathbf{Ly} = \mathbf{b}$ for $\mathbf{y}$.

  Now that $\mathbf{y}$ is known, solve the upper triangular system $\mathbf{Ux} = \mathbf{y}$ for the unknown $\mathbf{x}$.

- Thus, if we need to solve several problems with the same coefficient matrix $\mathbf{A}$ and different vectors $\mathbf{b}$, it's more efficient to find the LU decomposition of $\mathbf{A}$ and use that to solve for the different unknown vectors, instead of repeating the Gauss elimination for each problem.

- What about row pivoting?

  If we include row pivoting in the LU decomposition above, what we will get is a *permuted LU decomposition* with factors $\mathbf{L}$ and $\mathbf{U}$ such that $\mathbf{PA} = \mathbf{LU}$, where $\mathbf{P}$ is a permutation matrix that represents the row interchanges:

  Start with $\mathbf{P} = \mathbf{I}$, an $n \times n$ identity matrix.

  for $j = 1, 2, \ldots$ to n

  Row pivoting:

  Find row l such that $|A_{l,j}|$ is the maximum of all the elements in column j that are at or below row j.

  Interchange rows l and j in the matrix $\mathbf{A}$. (This is the same as multiplying $\mathbf{A}$ by a permutation matrix $\mathbf{Q}$, which is an identity matrix with rows l and j interchanged.)

  Update $\mathbf{P} \leftarrow \mathbf{QP}$. (Interchange rows l and j of $\mathbf{P}$)

  $p = A_{j,j}$ (pivot element; j is the pivot row)

  for $i = j + 1, j + 2, \ldots$ to n

  $M = A_{i,j}/p$ (multiplier)

  Save M as $L_{i,j}$

  $A_{i,:} = A_{i,:} - M \times A_{j,:}$

Then $\mathbf{U}$ is the transformed $\mathbf{A}$ (which is upper triangular) and $\mathbf{L}$ is the matrix built up from the multipliers M, with ones added along the main diagonal.

- Example:

$$\left(\begin{array}{cccc} 0 & 2 & 2 & (1) \\ 1 & -2 & -1 & (2) \\ 4 & -2 & 4 & (3) \end{array}\right) \rightarrow \left(\begin{array}{cccc} 4 & -2 & 4 & (3) \\ 1 & -2 & -1 & (2) \\ 0 & 2 & 2 & (1) \end{array}\right) \rightarrow \left(\begin{array}{cccc} 4 & -2 & 4 & (3) \\ \frac{1}{4}| & -\frac{3}{2} & -2 & (2) \\ 0| & 2 & 2 & (1) \end{array}\right) \rightarrow$$
$$\left(\begin{array}{cccc} 4 & -2 & 4 & (3) \\ 0| & 2 & 2 & (1) \\ \frac{1}{4}| & -\frac{3}{2} & -2 & (2) \end{array}\right) \rightarrow \left(\begin{array}{cccc} 4 & -2 & 4 & (3) \\ 0| & 2 & 2 & (1) \\ \frac{1}{4} & -\frac{3}{4}| & -\frac{1}{2} & (2) \end{array}\right)$$

Giving the factors

$$\mathbf{L} = \left(\begin{array}{ccc} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \frac{1}{4} & -\frac{3}{4} & 1 \end{array}\right), \mathbf{U} = \left(\begin{array}{ccc} 4 & -2 & 4 \\ 0 & 2 & 2 \\ 0 & 0 & -\frac{1}{2} \end{array}\right).$$

The product $\mathbf{LU}$ is equal to the original matrix with the rows permuted (switched). In this case the permutation matrix $\mathbf{P}$ is

$$\left(\begin{array}{ccc} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{array}\right)$$

- To solve a linear system given a permuted LU decomposition $\mathbf{PA} = \mathbf{LU}$:

  Given $\mathbf{Ax} = \mathbf{b}$ with unknown $\mathbf{x}$ and the LU decomposition $\mathbf{PA} = \mathbf{LU}$

  (Note that $\mathbf{PAx} = \mathbf{Pb}$, so $\mathbf{LUx} = \mathbf{Pb}$.)

  Form $\mathbf{y} = \mathbf{Ux}$

  Solve the lower triangular system $\mathbf{Ly} = \mathbf{Pb}$ for the unknown $\mathbf{y}$.

  Now that $\mathbf{y}$ is known, solve the upper triangular system $\mathbf{Ux} = \mathbf{y}$ for the unknown $\mathbf{x}$.

- In addition to the LU decomposition, a *symmetric* matrix (one that's the same as its transpose) may also has a *Cholesky decomposition*, for which the upper triangular factor is the transpose of the lower triangular factor (specifically, the Cholesky decomposition of a symmetric matrix exists if the matrix is *positive definite*). This decomposition can be computed with about half as many arithmetic operations as LU decomposition, and can used to help solve linear systems with this coefficient matrix just like the LU decomposition can.

  Given a symmetric positive definite matrix $\mathbf{A}$, its lower triangular Cholesky factor $\mathbf{L}$ ($\mathbf{A} = \mathbf{LL}^T$) can be computed as:

for $j = 1, 2, \ldots$ to n

$$L_{j,j} = \sqrt{A_{j,j} - \sum_{k=1}^{j-1} L_{j,k}^2}$$

for $i = j+1, j+2, \ldots$ to n

$$L_{i,j} = \frac{A_{i,j} - \sum_{k=1}^{j-1} L_{i,k} L_{j,k}}{L_{j,j}}$$

4 Eigenvalues and eigenvectors

- A vector $\mathbf{v}$ is an *eigenvector* of a matrix $\mathbf{A}$ if $\mathbf{Av} = \lambda\mathbf{v}$ for some nonzero scalar λ. λ is then an *eigenvalue* of $\mathbf{A}$.

- Newton's second law for a system with forces linear in the displacements (as in ideal springs connecting n different masses, or a discretized approximation to a linear beam or to a multistory building) can be written as

  $\mathbf{Mx}'' = -\mathbf{Kx}$ (where $\mathbf{x}''$ stands for the acceleration, $\frac{d^2x}{dt^2}$) – or

  $$\mathbf{x}'' = -\mathbf{Ax},$$

  where $\mathbf{A} \equiv \mathbf{M}^{-1}\mathbf{K}$.

  For this system, if $\mathbf{Av} = \lambda\mathbf{v}$, then $\mathbf{x}(t) = \mathbf{v}\sin(\sqrt{\lambda}t)$ and $\mathbf{x}(t) = \mathbf{v}\cos(\sqrt{\lambda}t)$ are solutions.

  Since this is a system of linear differential equations, any linear combination of solutions is also a solution. (There are normally n eigenvalue-eigenvector pairs $\lambda_i, \mathbf{v}_i$ of $\mathbf{A}$, and we need $2n$ initial conditions [i.e. the values of $\mathbf{x}(0)$ and $\mathbf{x}'(0)$] to find a unique solution $\mathbf{x}(t)$.)

  The general solution is therefore $\sum_{i=1}^{n} c_i\mathbf{v}_i \sin(\sqrt{\lambda_i}t) + d_i\mathbf{v}_i \cos(\sqrt{\lambda_i}t)$, or equivalently $\sum_{i=1}^{n} a_i\mathbf{v}_i e^{i\sqrt{\lambda_i}t} + b_i\mathbf{v}_i e^{-i\sqrt{\lambda_i}t}$, where c_i, d_i or a_i, b_i can be determined from the initial conditions

  The eigenvectors are modes of oscillation for the system, and the eigenvalues are the squared frequencies for each mode.

  In analyzing vibrational systems, the first few modes (the ones with the lowest frequencies/eigenvalues) are usually the most important because they are the most likely to be excited and the slowest to damp. The *fundamental mode* is the one with lowest frequency.

  Modes of e.g. a beam or structure can be found experimentally by measuring the responses induced by vibrations with different frequencies (modal analysis).

- How do we find eigenvalues and eigenvectors?

  For diagonal or (upper/lower) triangular matrices, the eigenvalues are just the diagonal elements

  For general 2×2 matrices, we can solve a quadratic equation for the eigenvalues λ – generally, there will be two (they may be complex). For each eigenvalue, we can then solve a linear system to get the eigenvector.

  Example:

  If

$$\mathbf{A} = \begin{pmatrix} -1 & 2 \\ 1 & 1 \end{pmatrix},$$

  eigenvalues λ and eigenvectors $\mathbf{v}$ must be solutions to

  $\mathbf{Av} = \lambda\mathbf{v}$, or $(\mathbf{A} - \lambda\mathbf{I})\mathbf{v} = \mathbf{0}$.

  Assuming that $\mathbf{v}$ isn't a zero vector, this implies that $(\mathbf{A} - \lambda\mathbf{I})$ is not invertible, so its determinant must be zero. But the determinant of $(\mathbf{A} - \lambda\mathbf{I})$ is $(-1 - \lambda)(1 - \lambda) - 2$, so we have the *characteristic polynomial*

  $\lambda^2 - 3 = 0 \rightarrow \lambda = \pm\sqrt{3}$.

  To find the eigenvectors corresponding to each λ, we solve the linear system to find $\mathbf{v}$. We have $\mathbf{Av} = \pm\sqrt{3}\mathbf{v}$, or

$$\begin{pmatrix} -1 \mp \sqrt{3} & 2 \\ 1 & 1 \mp \sqrt{3} \end{pmatrix} \mathbf{v} = \mathbf{0},$$

  where the second row is a multiple of the first, so there is not a unique solution. We have

$$\mathbf{v} = \begin{pmatrix} 1 \\ (1 \pm \sqrt{3})/2 \end{pmatrix}$$

  or any multiples thereof.

- In general, an $n \times n$ matrix has n (complex) eigenvalues, which are the roots of a characteristic polynomial of order n. We could find the eigenvalues by writing and solving for this characteristic polynomial, but more efficient numerical methods exist, for example based on finding a QR factorization of the matrix (which we won't cover in this class).

- A conceptually simple numerical method for finding the largest (in absolute value) eigenvalue of any given square matrix is the *power method*. It involves the iteration:

  Start with an $n \times 1$ vector $\mathbf{v}$

  Do until convergence:

  $\mathbf{v} \leftarrow \mathbf{Av}$

  $\mathbf{v} \leftarrow \frac{\mathbf{v}}{||\mathbf{v}||_2}$

- A symmetric real matrix will have only real eigenvalues. Otherwise, eigenvalues of a real matrix, being the roots of a polynomial, may also come in complex conjugate pairs.

- For any square matrix,

  The product of the eigenvalues is equal to the determinant

  The sum of the eigenvalues is equal to the sum of the elements on the matrix main diagonal (called the *trace*)

- For any matrix $\mathbf{A}$, the square root of the ratio of the largest to smallest eigenvalue of $\mathbf{A}\mathbf{A}^T$ is equal to the (2-norm) condition number of $\mathbf{A}$

- A linear system with damping, $\mathbf{M}\mathbf{x}''+\mathbf{C}\mathbf{x}'+\mathbf{K}\mathbf{x} = \mathbf{0}$ where $\mathbf{C}$ is a matrix of damping coefficients, has the general solution $\mathbf{x}(t) = \sum_{i=1}^{2n} c_i \mathbf{v}_i e^{\lambda_i t}$, where $\lambda_i, \mathbf{v}_i$ are generalized eigenvalue-eigenvector pairs that solve the *quadratic eigenvalue problem* $(\lambda^2\mathbf{M} + \lambda\mathbf{C} + \mathbf{K})\mathbf{v} = \mathbf{0}$ and the coefficients c_i can be set to match the initial conditions $\mathbf{x}(0), \mathbf{x}'(0)$.

5 Differentiation

- Finite difference (centered) to approximate $f'(x_0)$:

$$f'(x_0) \approx \frac{f(x_0 + \Delta x) - f(x_0 - \Delta x)}{2\Delta x}$$

  This approximation is 'second-order accurate,' meaning that truncation error is proportional to $(\Delta x)^2$ (and to $f'''(x)$) (derived previously from Taylor's theorem)

  However, can't make Δx very small because roundoff error will increase

- Richardson extrapolation

  Starting with some fairly large Δ_0, define D_i^0 to be the centered finite-difference estimate obtained with $\Delta x = \frac{\Delta_0}{2^i}$:

$$D_i^0(f, x_0, \Delta_0) \equiv \frac{f(x_0 + \Delta x) - f(x_0 - \Delta x)}{2 \cdot \Delta x}$$

  Then $D_i^1 \equiv \frac{4}{3} D_{i+1}^0 - \frac{1}{3} D_i^0$ will typically be much more accurate than D_{i+1}^0

  Can extend to higher $j \geq 1$ up to some $j_{\max}$:

$$D_i^j \equiv \frac{4^j}{4^j - 1} D_{i+1}^{j-1} - \frac{1}{4^j - 1} D_i^{j-1}$$

The difference between two estimates

$$(\text{at a given } j_{\max}, |D_0^{j_{\max}} - D_1^{j_{\max}-1}|)$$

can give an estimate of uncertainty, which may be used as a criterion for convergence. $j_{\max}$ often doesn't need to be large to get a very accurate estimate, which makes roundoff error less of a problem.

Example of estimates obtained:

$$(f(x) = \sin(x),\ x_0 = 0.5,\ \Delta_0 = 0.2, j_{\max} = 3)$$

	$j=0$	$j=1$	$j=2$	$j=3$
$i=0$	0.871743701440879	0.877579640095606	0.877582561716377	0.877582561890374
$i=1$	0.876120655431924	0.877582379115078	0.877582561887656	
$i=2$	0.877216948194290	0.877582550464370		
$i=3$	0.877491149896850			

where the answer is actually $\cos(0.5) = 0.8775825618903727\ldots$

- Second-order accurate finite-difference approximations to higher derivatives (which can also be derived from Taylor's theorem) are

$$f''(x) \approx \frac{f(x+\Delta x) - 2f(x) + f(x-\Delta x)}{(\Delta x)^2}$$

$$f'''(x) \approx \frac{f(x+2\Delta x) - 2f(x+\Delta x) + 2f(x-\Delta x) - f(x-2\Delta x)}{2(\Delta x)^3}$$

$$f^{(4)}(x) \approx \frac{f(x+2\Delta x) - 4f(x+\Delta x) + 6f(x) - 4f(x-\Delta x) + f(x-2\Delta x)}{(\Delta x)^4}$$

- Non-centered (forward or backward) finite-difference approximations can be derived which are useful for estimating a derivative at the edge of a function's range. For example, a second-order accurate forward finite-difference approximation for the first derivative is

$$f'(x) \approx \frac{-f(x+2\Delta x) + 4f(x+\Delta x) - 3f(x)}{2\Delta x}$$

- Cubic spline interpolation [see under Interpolation]

  Good when only some given function values (not necessarily equally spaced) are available

  Fit an interpolating cubic spline $S(x)$ to the given points, and estimate $f'(x)$ as $S'(x)$

6 Integration

6.1 Introduction

- Some applications of integrals

  Average function value between a and b: $\frac{1}{b-a}\int_a^b f(x)dx$

  Center of mass (in 1-D):

$$\frac{\int_a^b x\rho(x)dx}{\int_a^b \rho(x)dx}$$

  (ρ = density)

  Moment of inertia about $x = x_0$ (in 1-D): $\int_a^b (x-x_0)^2\rho(x)dx$

  Net force produced by a distributed loading: $\int_a^b w(x)dx$ (w = force per unit length)

  Net moment about $x = x_0$ produced by a distributed loading:

$$\int_a^b (x-x_0)w(x)dx$$

- Typical situations where we need to approximate an integral $I = \int_a^b f(x)dx$ numerically:

  The function f doesn't have an analytic integral

  No mathematical expression for function is available – we can only measure values or get them from a computation.

6.2 Basic integration rules

- Trapezoid rule: $I = \int_a^b f(x)dx \approx T = (b-a)\frac{f(a)+f(b)}{2}$ (exact if $f(x)$ is a straight line [first-degree polynomial])
- Midpoint rule: $I = \int_a^b f(x)dx \approx M = (b-a)f(\frac{a+b}{2})$ (also exact if $f(x)$ is a straight line [first-degree polynomial])
- Simpson's rule: combines the trapezoid and midpoint rules in such a way as to be exact if $f(x)$ is up to a third-degree polynomial.

$$S = (T+2M)/3 = \frac{b-a}{6}\left(f(a) + 4f(\frac{a+b}{2}) + f(b)\right)$$

  (1-4-1 weighting of the function values)

6.3 Composite rules

- For numerical approximations of an integral, we can take advantage of the fact that integral are additive – if we divide the integration interval $[a, b]$ into $[a, c]$ and $[c, b]$, the integral $I = \int_a^b f(x)dx$ is equal to $I_1 = \int_a^c f(x)dx$ plus $I_2 = \int_c^b f(x)dx$.

  Composite forms of the trapezoid, midpoint, or Simpson rules divide $[a, b]$ up into n subintervals, apply the rule to each subinterval, and then add up the results to get an approximation for the whole integral. Generally, this improves the accuracy of the approximation compared to applying the rule to the whole interval.

  For the composite Simpson's rule when the interval has equally spaced points $a = x_0, x_1, x_2, ...x_n = b$, we get a 1-4-2-4-2-. . . 4-2-4-1 weighting (For the trapezoid rule, it's 1-2-2-2-. . . 2-1)

  Adaptive subdivision can be used to achieve high accuracy for numerical integration with fewer computations.

  An analogue to Taylor's theorem can be used to derive absolute error bounds for the composite rules with equally spaced evaluation points for a function f that is smooth in $[a, b]$: $\frac{(b-a)^3}{12}\frac{K_2}{n^2}$ for the trapezoid rule, $\frac{(b-a)^3}{24}\frac{K_2}{n^2}$ for the midpoint rule, and $\frac{(b-a)^5}{180}\frac{K_4}{n^4}$ for Simpson's rule, where K_2 is the maximum of $|f''(x)|$ in (a, b) and K_4 is the maximum of $|f''''(x)|$ in (a, b). Although we often don't know the maximum value of derivatives of f, these bounds are nevertheless useful for estimating how the error will decrease as a result of increasing the number of intervals n.

- To integrate functions whose values are only available at certain points (which may be unequally spaced), there are a few options:

  Composite trapezoid rule works even for unequally spaced intervals

  Can interpolate the points with an easily integrable function (such as a polynomial or cubic spline) and integrate the interpolating function.

6.4 More complex numerical methods

- Romberg integration

  Let $R_i^0, i = 0, 1, 2, \ldots$ be the estimated integrals obtained by applying the composite trapezoid rule with 2^i equal-width subintervals:

$$R_i^0 \equiv \frac{b-a}{2^i}\left(\frac{f(a)+f(b)}{2} + \sum_{j=1}^{2^i-1} f\left(a + (b-a)\frac{j}{2^i}\right)\right)$$

$$= \frac{R_{i-1}^0}{2} + \frac{b-a}{2^i}\sum_{\substack{j=1 \\ j \text{ odd}}}^{2^i-1} f\left(a + (b-a)\frac{j}{2^i}\right) \quad [\text{if } i > 0]$$

Because the error from the composite trapezoid rule decreases as the number of subintervals squared, The error of R_{i+1}^0 is expected to be about $1/4$ that of R_i^0, and in the same direction

We exploit this by coming up with a generally more accurate estimate $R_i^1 \equiv \frac{4}{3}R_{i+1}^0 - \frac{1}{3}R_i^0$.

Can continue, with $R_i^j \equiv \frac{4^j}{4^j-1}R_{i+1}^{j-1} - \frac{1}{4^j-1}R_i^{j-1}$ for any $j \geq 1$, to obtain generally even more accurate estimates.

Difference between two estimates can give an estimate of uncertainty, which may be used as a criterion for convergence. For smooth functions, i often doesn't need to be large to get a very accurate estimate.

Algorithm:

for $j = 0, 1, \ldots j_{\max}$:

Evaluate the function at $2^j + 1$ equally spaced points, including the endpoints a and b (giving 2^j equal-width subintervals), and obtain R_j^0

Find $R_{j-i}^i, i = 1, \ldots j$ using the formula $R_i^j \equiv \frac{4^j}{4^j-1}R_{i+1}^{j-1} - \frac{1}{4^j-1}R_i^{j-1}$

For $j \geq 1$, if $|R_0^j - R_1^{j-1}|$ is less than our required absolute error tolerance, can stop (convergence reached)

As an example, consider $f(x) = e^x - 4x$, $a = 0$, $b = 1$, $j_{\max} = 3$:

	$j=0$	$j=1$	$j=2$	$j=3$
$i=0$	-0.14086	-0.28114	-0.28172	-0.28172
$i=1$	-0.24607	-0.28168	-0.28172	
$i=2$	-0.27278	-0.28172		
$i=3$	-0.27948			

The best estimate from this is R_0^3 or -0.28172.

- Gauss quadrature

Estimate an integral based on the function value at specific non-equally spaced points within the interval (more points closer to the edges)

Select the sample points and weights based on approximating the function as a polynomial of degree $2n-1$, where n is the number of points

In practice, tabulated values of the sample points x_i and weights w_i for the standard integration interval $[-1, 1]$ are available

To approximate $I = \int_{-1}^{1} f(x)dx$, use $G_n = \sum_{i=1}^{n} w_i f(x_i)$

To approximate $I = \int_a^b f(x)dx$, use

$$G_n = \frac{b-a}{2}\sum_{i=1}^{n} w_i \cdot f\left(a + \frac{b-a}{2}(x_i + 1)\right)$$

Can give very accurate numerical integral estimates with few function evaluations (small n).

With given n, could divide the integration interval into parts and apply Gauss quadrature to each one in order to get increased accuracy

- Both Romberg integration and Gauss quadrature are only applicable if we can find the function value at the desired points. Also, they may not be more accurate than simpler methods if the function is not smooth (e.g. has discontinuities). Essentially this is because they both rely on approximating the function by the first terms in its Taylor series.

7 Ordinary differential equations: Initial value problems

- Initial-value ODE first-order problem: Given $\frac{dy}{dt} = f(y,t)$ and $y(a) = y_0$, find $y(b)$ – this is a generalization of the definite integral problem considered previously

- Most numerical methods for ODE IVPs consider a sequence of points $a = t_0, t_1, t_2, \ldots t_N = b$, similar to the idea of composite integration rules, and construct estimates of y at those points: $y(a) = y_0, y_1, y_2, \ldots, y_N \approx y(b)$

- Euler method (explicit): $y_{i+1} = y_i + hf(y_i, t_i)$, where h is the step size, $t_{i+1} - t_i$ (iterate to h to get from $t_0 = a$ to $t_N = b$)

  Approximates $\bar{f}$ with the value of f at the beginning of the interval $[t_i, t_i + h]$

- Initial-value problem for an ODE first-order *system*: Given $\frac{dy_i}{dt} = f(y_1, y_2, \ldots, y_n, t)$ and $y_i(a) = y_{i,0}$ for $i = 1, 2, \ldots n$, find all $y_i(b)$

  In vector notation: Given $\frac{d\mathbf{y}}{dt} = \mathbf{f}(\mathbf{y}, t)$ and $\mathbf{y}(a) = \mathbf{y}_0$, where $\mathbf{y}(t)$ is an $n \times 1$ vector and $\mathbf{f}$ is a function that returns an $n \times 1$ vector, find $\mathbf{y}(b)$

  In this notation, Euler's method for a system can be written compactly as $\mathbf{y}_{i+1} = \mathbf{y}_i + h\mathbf{f}(\mathbf{y}_i, t_i)$

- Any ODE system (even one with higher derivatives) can be converted to this first-order form by setting the derivatives of lower order than the highest one that appears as additional variables in the system

  Example: pendulum motion (with friction and a driving force),

$$\frac{d^2\theta}{dt^2} + c\frac{d\theta}{dt} + \frac{g}{L}\sin(\theta) = a\sin(\Omega t)$$

  (second-order equation: highest-order derivative of θ is 2)

Can be written as

$$\frac{dy_1}{dt} = y_2$$
$$\frac{dy_2}{dt} = -cy_2 - \frac{g}{L}\sin(y_1) + a\sin(\Omega t)$$

where $y_1 = \theta, y_2 = \frac{d\theta}{dt}$.

- Implicit Euler method: $y_{i+1} = y_i + hf(y_{i+1}, t_{i+1})$ – implicit because the unknown y_{i+1} appears on both sides of the equation.

  Solving for this y_{i+1} may require a numerical nonlinear equation solving (root-finding) method, depending on how f depends on y.

- Crank-Nicholson method (uses average of slopes from original (explicit) and implicit Euler methods to estimate $\bar{f}$, and so should be more accurate then either one – analogous to trapezoid rule)

  $y_{i+1} = y_i + h(f(y_i, t_i) + f(y_{i+1}, t_{i+1}))/2$

  Again, an implicit method, which makes each step substantially more complicated than in the original (explicit) Euler method (unless, for example, f is a linear function of y)

- Modified (second-order accurate) Euler method is a 2nd-order 'Runge-Kutta' method (RK2; also known as Heun's method); it's analogous to trapezoid rule, but unlike Crank-Nicholson method it's explicit, so no root-finding is required:

  Two stages at each timestep, based on estimating the derivative at the end as well as the beginning of each subinterval:

  $$y_{i+1} = y_i + \frac{K_1 + K_2}{2},$$

  with $K_1 = hf(y_i, t_i)$ (as in Euler method) and $K_2 = hf(y_i + K_1, t_i + h)$

- The classic Runge-Kutta method of order 4 (RK4) is one that is often used in practice for solving ODEs. Each step involves four stages, and can be written as:

  $$y(t_{i+1}) = y(t_i) + \frac{1}{6}(K_1 + 2K_2 + 2K_3 + K_4),$$

  with $K_1 = hf(y(t), t)$ (as in Euler method), $K_2 = hf(y(t_i) + K_1/2, t_i + h/2)$, $K_3 = hf(y(t_i) + K_2/2, t_i + h/2)$, $K_4 = hf(y(t_i) + K_3, t_i + h)$ (Notice that the 1-2-1 ratio of the weights is similar to the Simpson rule)

- Local truncation error of a numerical method: error of each step of length h

For Euler's method, the local error is bounded by

$$\frac{h^2}{2}K_2,$$

where K_2 is the maximum of $|y''|$ between t and $t+h$

- Global truncation error: error for far end of interval, after $n = |b-a|/h$ steps

  The global error may depend in a complicated way on the local errors at each step, but usually can be roughly approximated as the number of steps times the local truncation error for each step

  For Euler's method, the estimated global error using this approximation is $|b-a| \cdot h/2 \cdot y''(c)$ for some c between a and b. Thus, the estimated global error is proportional to h (first-order accuracy).

  The RK2 local truncation error is bounded by

$$\frac{h^3}{12}K_3,$$

  where K_3 is the maximum of $|y^{(3)}|$ between t and $t+h$ (same as for the trapezoid rule in integration). Thus, the estimated global error is proportional to h^2 (second order).

  The RK4 local truncation error is bounded by

$$\frac{h^5}{2880}K_5,$$

  where K_5 is the maximum of $|y^{(5)}|$ between t and $t+h$ (same as for Simpson's rule in integration). Thus, the estimated global error is proportional to h^4 (fourth order). Usually, it will be much more accurate than first- or second-order numerical methods (such as Euler or RK2) for the same step size h.

- The Euler method as well as the other explicit and implicit methods can be extended readily to systems of ODEs – just run through all the equations in the system at each timestep to go from $\mathbf{y}(t)$ to $\mathbf{y}(t+h)$ (but for the implicit methods, will need to solve a system of equations at each timestep)

- In practice, how do we estimate the local truncation error (since we don't generally know the values of higher derivatives of y)? The usual method is to compare two estimates of $y(t+h)$, obtained by different numerical methods or different step sizes.

  The estimated error can be used to adjust the step size h *adaptively* so that the global truncation error is within the given tolerance: if the estimated error is very small the step size is increased so that fewer computations are needed, while if the estimated error is too large the step size is decreased so that the answer is accurate enough.

8 Ordinary differential equations: Boundary value problems

- In initial-value ODE problems (which is the type we've been doing so far), the conditions are all given at one value of the independent variable t, say $t = a$. By contrast, in boundary-value problems, the conditions are spread out between different values of t. Therefore, we can't use methods that start at the known initial condition and take steps away from it, as in the Euler method and Runge-Kutta methods, to find a numerical solution.

- Example of a boundary-value problem: A beam supported at both ends with nonlinear deflection, so that

$$\frac{y''}{(1+(y')^2)^{3/2}} - \frac{T}{EI}y = \frac{wx(L-x)}{2EI},$$

where T is the tension in the beam, E is the beam modulus of elasticity, I is the beam moment of inertia, and w is the uniform loading intensity on the beam. The boundary conditions are $y(0) = 0$ and $y(L) = 0$.

Linearized form:

$$y'' - \frac{T}{EI}y = \frac{wx(L-x)}{2EI}.$$

- Another example (linear):

Steady-state 1-D groundwater flow

$$\frac{\mathrm{d}}{\mathrm{d}x}\left(\mathrm{K}\frac{\mathrm{d}y}{\mathrm{d}x}\right) = 0,$$

where $y(x)$ is the groundwater head and K the hydraulic conductivity, with upstream and downstream boundary conditions $y(x_u) = y_u, y(x_d) = y_d$.

- *Finite-difference* is one method of numerically solving boundary value problems. The idea is to find y on an equally spaced grid, in the beam example case between $x = 0$ and $x = L$, where the points on the grid are designated $0 = x_0, x_1, x_2, \ldots x_n = L$ and the corresponding y values are $y_0, y_1, y_2, \ldots y_n$. We get one equation at each x_i. For the beam example, this is

$$\frac{y''(x_i)}{(1+(y'(x_i))^2)^{3/2}} - \frac{T}{EI}y_i = \frac{wx(L-x_i)}{2EI}.$$

To continue we need expressions for the derivatives of $y(x)$ at each x_i in terms of the x_i and y_i. We approximate these by finite-difference formulas, for example (all these formulas are second-order-accurate):

$$y'(x_i) \approx \frac{y_{i+1} - y_{i-1}}{2h}$$

$$y''(x_i) \approx \frac{y_{i+1} - 2y_i + y_{i-1}}{h^2}$$

$$y'''(x_i) \approx \frac{y_{i+2}/2 - y_{i+1} + y_{i-1} - y_{i-2}/2}{h^3}$$

$$y''''(x_i) \approx \frac{y_{i+2} - 4y_{i+1} + 6y_i - 4y_{i-1} + y_{i-2}}{h^4}$$

Applying these for i from 1 to $n-1$, this gives us $n-1$ generally nonlinear equations for the $n-1$ unknown y_i (from the boundary conditions, we already know that $y_0 = y(0) = 0$ and $y_n = y(L) = 0$). If the ODE system is linear (as in the linearized form of the beam example, or in the groundwater example), then we actually have $n-1$ linear equations in the $n-1$ unknowns (or $n+1$ if we include the boundary conditions as additional equations), which we can solve using Gauss elimination or LU decomposition to find the unknown y_i, and hence (in this example) the shape of the deflected beam.

- Note: For the finite-difference method, there's no need to rewrite higher-order ODEs as a system of first-order ODEs.

- Non-centered finite-difference formulas could be used for conditions near the boundaries. Alternatively, fictitious points beyond the boundaries, such as y_{-1} and y_{n+1}, could be added to the system to allow using the centered formulas.

- Example: Use finite differences to approximate $y(x)$ given $y'' + y' = x, y(0) = 1, y'(15) = 2$:

  Start by establishing a grid of points to solve at that spans the domain over which there are boundary conditions: say $n = 5$ equally spaced intervals, with endpoints indexed $0, 1, \ldots 5$ ($h = 3$)

  Next, write the differential equation for each interior grid point, plus the boundary conditions, with each derivative replaced by a finite-difference approximation (here, ones with second-order accuracy are used):

  $y_0 = 1$
  $\frac{1}{h^2} y_0 - \frac{2}{h^2} y_1 + \frac{1}{h^2} y_2 - \frac{1}{2h} y_0 + \frac{1}{2h} y_2 = 3$
  $\frac{1}{h^2} y_1 - \frac{2}{h^2} y_2 + \frac{1}{h^2} y_3 - \frac{1}{2h} y_1 + \frac{1}{2h} y_3 = 6$
  $\frac{1}{h^2} y_2 - \frac{2}{h^2} y_3 + \frac{1}{h^2} y_4 - \frac{1}{2h} y_2 + \frac{1}{2h} y_4 = 9$
  $\frac{1}{h^2} y_3 - \frac{2}{h^2} y_4 + \frac{1}{h^2} y_5 - \frac{1}{2h} y_3 + \frac{1}{2h} y_5 = 12$
  $\frac{1}{2h} y_3 - \frac{2}{h} y_4 + \frac{3}{2h} y_5 = 2$ [backward finite-difference approximation for the first derivative at the upper boundary]

The resulting system of algebraic equations for the approximate y values at the grid points is

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ -\frac{1}{18} & -\frac{2}{9} & \frac{5}{18} & 0 & 0 & 0 \\ 0 & -\frac{1}{18} & -\frac{2}{9} & \frac{5}{18} & 0 & 0 \\ 0 & 0 & -\frac{1}{18} & -\frac{2}{9} & \frac{5}{18} & 0 \\ 0 & 0 & 0 & -\frac{1}{18} & -\frac{2}{9} & \frac{5}{18} \\ 0 & 0 & 0 & \frac{1}{6} & -\frac{2}{3} & \frac{1}{2} \end{pmatrix} \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{pmatrix} = \begin{pmatrix} 1 \\ 3 \\ 6 \\ 9 \\ 12 \\ 2 \end{pmatrix}$$

which gives, to 5 decimal places,

$$\mathbf{y} = \begin{pmatrix} 1 \\ 325.32 \\ 270.85 \\ 173.22 \\ 116.80 \\ 102 \end{pmatrix}.$$

9 Interpolation

- Interpolation: fitting a function of a given type, say f, through some given points (x_i, y_i) so that for all the given points, $f(x_i) = y_i$.

- Why interpolate data?

  We may have measured responses at a few points, need responses at intermediate points.

  We may have evaluated a complicated function (such as the solution to a differential equation) at a few points, and want estimates for its value at many other points.

  We may want to estimate the average value, integral, derivative, ... of a measured quantity or difficult-to-evaluate function whose value is known only at some points.

- Polynomial interpolation

  For any set of n points (x_i, y_i) with distinct x_i, there's a unique polynomial p of degree $n-1$ such that $p(x_i) = y_i$.

  For $n > 5$ or so, polynomial interpolation tends to in most cases give oscillations around the given points, so the interpolating polynomial often looks unrealistic.

  Polynomial interpolation is quite *nonlocal* and *ill-conditioned*, especially for larger n: if you change slightly one of the points to interpolate, the whole curve will often change substantially.

  Finding the interpolating polynomial through given points:

Lagrange form:

$$\sum_{i=1}^{n} y_i \prod_{\substack{j=1 \\ j\neq i}}^{n} \frac{x - x_j}{x_i - x_j}$$

Newton form: P_n, where $P_1 = y_1$ and

$$P_i = P_{i-1} + (y_i - P_{i-1}(x_i)) \prod_{j=1}^{i-1} \frac{x - x_j}{x_i - x_j}$$

for $i = 2, \ldots n$

Example: if `x = [0 1 -2 2 -1]'` and `y = [-3 -2 1 -4 1]'`, the Lagrange form of the interpolating polynomial is

$P_L(x) = -\frac{3}{4}(x-1)(x+2)(x-2)(x+1) + \frac{1}{3}x(x+2)(x-2)(x+1) + \frac{1}{24}x(x-1)(x-2)(x+1)$
$-\frac{1}{6}x(x-1)(x+2)(x+1) - \frac{1}{6}x(x-1)(x+2)(x-2)$

The Newton form is

$$P_N(x) = -3 + x + x(x-1) - \frac{5}{8}x(x-1)(x+2) - \frac{17}{24}x(x-1)(x+2)(x-2)$$

The two forms are equivalent; thus, both give $P(\frac{1}{2}) = -\frac{387}{128}$, and both will expand to $P(x) = -\frac{17}{24}x^4 + \frac{1}{12}x^3 + \frac{77}{24}x^2 + \frac{19}{12}x - 3$

- Spline interpolation

  Idea: interpolating function $S(x)$ is a piecewise polynomial of some low degree d; the derivative of order d is discontinuous at *nodes* or *knots*, which are the boundaries between the polynomial pieces (for our purposes, these are set to be at the data points).

  Linear spline: $d = 1$ – connects points by straight lines (used for `plot`) – first derivative isn't continuous.

  Quadratic spline: $d = 2$ – connects lines by segments of parabolas – second derivative isn't continuous (but first derivative is).

  Cubic spline: $d = 3$ – commonly used – minimizes curvature out of all the possible interpolating functions with continuous second derivatives (third derivative is discontinuous)

  Uses $n-1$ cubic functions to interpolate n points ($4n-4$ coefficients total)

  Need 2 additional conditions to specify coefficients uniquely. Commonly, we use *natural* boundary conditions, where the second derivative is zero at the endpoints. Another possibility is *not-a-knot* boundary conditions, where the first two cubic functions are the same and last two cubic functions are the same.

The spline coefficients that interpolate a given set of points can be found by solving a linear system. For a cubic spline:

Let the $n+1$ given points be (x_i, y_i), for $i = 0, 1, \ldots n$

Each piecewise cubic polynomial (which interpolates over the interval $[x_{i-1}, x_i]$) can be written as $S_i(x) = a_i(x - x_{i-1})^3 + b_i(x - x_{i-1})^2 + c_i(x - x_{i-1}) + d_i$, where $i = 1, 2, \ldots n$

The $4n-2$ conditions for the piecewise polynomials to form a cubic spline that interpolates the given points are
$S_i(x_{i-1}) = y_{i-1}, i = 1, 2, \ldots n$
$S_i(x_i) = y_i, i = 1, 2, \ldots n$
$S_i'(x_i) = S_{i+1}'(x_i), i = 1, 2, \ldots n-1$
$S_i''(x_i) = S_{i+1}''(x_i), i = 1, 2, \ldots n-1$

We can find the b_i by solving a linear system that includes the following $n-2$ equations, plus two more from the boundary conditions:
$h_{i-1}b_{i-1} + 2(h_{i-1} + h_i)b_i + h_i b_{i+1} = 3(\Delta_i - \Delta_{i-1})$, for $i = 2, 3, \ldots n-1$
where $h_i \equiv x_i - x_{i-1}, \Delta_i \equiv \frac{y_i - y_{i-1}}{h_i}$

For natural boundary conditions, the two additional equations are:
$b_1 = 0$
$h_{n-1}b_{n-1} + 2(h_{n-1} + h_n)b_n = 3(\Delta_n - \Delta_{n-1})$

For not-a-knot boundary conditions, we can introduce an additional unknown b_{n+1}, and the three additional equations needed are:
$h_2 b_1 - (h_1 + h_2)b_2 + h_1 b_3 = 0$
$h_n b_{n-1} - (h_{n-1} + h_n)b_n + h_{n-1}b_{n+1} = 0$
$h_{n-1}b_{n-1} + 2(h_{n-1} + h_n)b_n + h_n b_{n+1} = 3(\Delta_n - \Delta_{n-1})$

Once the b coefficients are found, we can find the a coefficients as
$a_i = \frac{b_{i+1} - b_i}{3h_i}, i = 1, 2, \ldots n-1$
and $a_n = \frac{-b_n}{3h_n}$ (natural boundary conditions) or $a_n = a_{n-1}$ (not-a-knot)

For the remaining coefficients,
$c_i = \Delta_i - h_i b_i - h_i^2 a_i$,
$d_i = x_{i-1}$.

For example, with (x, y) pairs $(1, 3), (2, 4), (3, 6), (4, 7)$,
the cubic spline with natural boundary conditions has the pieces
$S_1(x) = \frac{1}{3}(x-1)^3 + \frac{2}{3}(x-1) + 3$
$S_2(x) = -\frac{2}{3}(x-2)^3 + (x-2)^2 + \frac{5}{3}(x-2) + 4$
$S_3(x) = \frac{1}{3}(x-3)^3 - (x-3)^2 + \frac{5}{3}(x-3) + 6.$
The cubic spline with not-a-knot boundary conditions has the pieces
$S_1(x) = -\frac{1}{3}(x-1)^3 + \frac{3}{2}(x-1)^2 - \frac{1}{6}(x-1) + 3$
$S_2(x) = -\frac{1}{3}(x-2)^3 + \frac{1}{2}(x-2)^2 + \frac{11}{6}(x-2) + 4$
$S_3(x) = -\frac{1}{3}(x-3)^3 - \frac{1}{2}(x-3)^2 + \frac{11}{6}(x-3) + 6$
(In fact, when, as in this example, $n = 3$, the cubic spline with not-a-knot boundary conditions is the same as the interpolating polynomial.)

- Piecewise cubic Hermite polynomial interpolation (also known as monotone cubic interpolation)

Piecewise cubic (like cubic spline)

Differences from cubic spline:

Second derivative not continuous (so less smooth)

Extreme points only at the nodes (so never goes above or below the range formed by the two adjacent points – good when the values should stay within an acceptable range)

10 Regression

- Least squares fitting is a kind of *regression*

  Regression: fit a function of a given type, say f, *approximately* through some given points (x_i, y_i) so that for the given points, $f(x_i) \approx y_i$.

  If the points are given as $n \times 1$ vectors $\mathbf{x}, \mathbf{y}$, the *residual* vector of the fitted function is $\mathbf{r} = \mathbf{y} - f(\mathbf{x})$ (i.e. $r_i = y_i - f(x_i)$)

  The *least squares* criterion: Out of all the functions in the given type, minimize the residual sum of squares, RSS $= \mathbf{r}^T\mathbf{r} = \sum_{i=1}^{n} r_i^2$

- Suppose the function type is such that we can write $f(\mathbf{x}) = \mathbf{A}\boldsymbol{\beta}$, where $\mathbf{A}$ is a known $n \times m$ *design matrix* for the given $\mathbf{x}$ while $\boldsymbol{\beta}$ is an unknown $m \times 1$ vector of 'parameters'. That is, $f(x_i) = A_{i,1}\beta_1 + A_{i,2}\beta_2 + \ldots A_{i,m}\beta_m$.

  Example: the function type is a straight line, $f(x) = ax + b$. Then row i of $\mathbf{A}$ consists of $(x_i, 1)$, and $\boldsymbol{\beta}$ is $[a, b]^T$ $(m = 2)$.

  Example: the function type is a quadratic with zero intercept, $f(x) = cx^2$. Then row i of $\mathbf{A}$ consists of (x_i^2), and $\boldsymbol{\beta}$ is $[c]$ $(m = 1)$.

- In that case, the residual sum of squares (RSS) $\mathbf{r}^T\mathbf{r} = ||\mathbf{r}||_2^2$ is equal to $(\mathbf{A}\boldsymbol{\beta} - \mathbf{y})^T(\mathbf{A}\boldsymbol{\beta} - \mathbf{y})$. Under least squares, we want to choose $\boldsymbol{\beta}$ so that this quantity is as small as possible.

  To find a minimum of the residual sum of squares, we take its derivative with respect to $\boldsymbol{\beta}$, which works out to be

  $$2\mathbf{A}^T(\mathbf{A}\boldsymbol{\beta} - \mathbf{y}).$$

  If we set this equal to zero, we then get for $\boldsymbol{\beta}$ the $m \times m$ linear system

  $$\mathbf{A}^T\mathbf{A}\boldsymbol{\beta} = \mathbf{A}^T\mathbf{y}$$

  Solving this linear system of *normal equations* gives us the parameters $\boldsymbol{\beta}$ that, for the given function type and data points, is the best fit (has smallest residual sum of squares).

  Minimizing RSS is the same as maximizing the *coefficient of determination* $R^2 = 1 - \frac{\text{RSS}}{\text{TSS}}$, where the total sum of squares TSS is $(\mathbf{y} - \bar{y})^T(\mathbf{y} - \bar{y})$, with $\bar{y}$ the average value of $\mathbf{y}$

- An important question in regression is *which* function class we should choose to fit given data, for example what degree polynomial to use. In general, function classes with many parameters (like high-degree polynomials) can fit any given data better (smaller RSS), but are not always better at predicting new data that were not included in the fit. Some ways to decide which function class to choose are:

  Graphical inspection: Plot each fitted function (as a line) together with the given points (as a scatter). Select the function class with fewest unknown parameters that seems to follow the overall trend of the data.

  Out-of-sample validation: Divide the available data into "training" and "validation" subsets. Fit the parameters for each model using only the training data. Choose the model with lowest RSS for the validation data.

  Numerical criteria:

  Adjusted R^2 :

$$R_a^2 = 1 - \frac{n-1}{n-m}\frac{\mathrm{RSS}}{\mathrm{TSS}},$$

  where n is the number of data points and m is the number of unknown parameters in each model. Each model is fitted to the data and R_a^2 is computed, and the model with highest R_a^2 is chosen as likely to be best at predicting the values of new points.

  Another commonly used rule is the Akaike information criterion (AIC), which can be given as follows for linear least squares:

$$\mathrm{AIC} = n\log(\mathrm{RSS}/n) + 2mn/(n-m-1)$$

  (log designates natural logarithm). Each model is fitted to the data and AIC is computed, and the model with lowest AIC is chosen as likely to be best at predicting the values of new points.

- In *nonlinear* least squares fitting, the function form is such that finding the least squares parameter values for it requires solving a system of nonlinear equations. In general this is a more difficult numerical problem.

11 Root finding (solving nonlinear equations)

- General form of a nonlinear equation in x: $g(x) = h(x)$, e.g. $x^3 + x = 3$. Unlike non-singular linear systems, such equations can have many (or no) solutions

- Standard form of a nonlinear equation: $f(x) = 0$, where $f(x) = g(x) - h(x)$, e.g. $x^3 + x - 3 = 0$. Any solution to this equation is called a 'root' of f.

- *Newton's method*: The iteration

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

converges to a *root* x^* of f if the initial value x_0 is close to x^*.

e.g. the above example with $x_0 = 1$

Only needs one iteration to find x^* if $f(x)$ is linear (straight line)

Sometimes doesn't converge if $f(x)$ is strongly nonlinear (curved) between x_0 and $x*$. Most likely to converge if x_0 is close to $x*$.

Like many numerical methods, Newton's method (stopped after finitely many iterations) will usually only give an approximate answer, but is easily implementable on a computer.

- One possible "stopping criterion" would be to check when the (absolute or relative) difference between x_n and x_{n-1} is small enough, as an estimate of the error relative to the unknown true value x^*.

- Another possible stopping criterion would be for $f(x_n)$ to be close enough to $f(x^*) = 0$ is small enough. This difference is called the "residual" and is another measure of the error in our estimate of x^* after n iterations.

- Newton's method can be generalized for a function of a vector, $\mathbf{f}(\mathbf{x})$, where the "Jacobian" matrix of partial derivatives $\mathbf{J}(\mathbf{x})$ has the elements $J_{ij}(\mathbf{x}) = \frac{\partial f_i}{\partial x_j}$: $\mathbf{x}_{i+1} = \mathbf{x}_i - \mathbf{J}^{-1}(\mathbf{x}_i)\mathbf{f}(\mathbf{x}_i)$

- Disadvantages of Newton's method are that it requires the derivative of f and may not converge if the function is too nonlinear (so that it has 'flat' regions where the derivative is much closer to zero than elsewhere).

- *Bisection* is another iterative (approximate) numerical root-finding method. It is slower to converge (more iterations typically required for high accuracy) but very reliable.

- Algorithm:

Start by finding two values a and b on either side of the root, such that $f(a) < 0$ and $f(b) > 0$.

For each iteration:

Find $f(c)$, where c is halfway between a and b ($c \leftarrow (a+b)/2$)

If $f(c) = 0$, return c as the root

If $f(c) > 0$, set $b \leftarrow c$

If $f(c) < 0$, set $a \leftarrow c$

Assuming that the exact root is not found, each iteration makes the interval $[a, b]$ half as wide as before

Possible stopping criteria (for some specified error tolerances tol, ϵ):

Number of iterations

Uncertainty in the root: $\frac{|b-a|}{2} < \text{tol}$ (absolute uncertainty) or $\frac{|b-a|}{|b+a|} < \text{tol}$ (fractional uncertainty)

Small residual, $|f(c)| < \epsilon$

- Example for $f(x) = x^2 - 2$:

  Initialize: $a \leftarrow 1, b \leftarrow 2$

  Iteration 1: $c \leftarrow 1.5, f(c) > 0, b \leftarrow c$

  Iteration 2: $c \leftarrow 1.25, f(c) < 0, a \leftarrow c$

  Iteration 3: $c \leftarrow 1.375, f(c) < 0, a \leftarrow c$

- The *secant method* is similar to Newton's method in that it's based on locally approximating the function as a straight line, but it doesn't require us to be able to compute the derivative, instead estimating it from the difference in function value between two points.

  Like Newton's method, this method converges fast (few iterations) if the function f is locally pretty close to a straight line, but may not converge at all if the function is very nonlinear.

- Algorithm:

  Start with two initial points a and b close to the root (can be the same as the starting points for bisection)

  For each iteration:

  Find $f(a)$ and $f(b)$

  Estimate the function's slope (first derivative) as

$$s = \frac{f(b) - f(a)}{b - a}$$

  Compute $c \leftarrow b - f(b)/s$

  Set $a \leftarrow b$, $b \leftarrow c$.

  Possible stopping criteria:

  Number of iterations

  Estimated uncertainty in the root, $|f(b)/s| < \text{tol}$

  Small residual, $|f(c)| < \epsilon$

- Example for $f(x) = x^2 - 2$:

  Initialize: $a \leftarrow 1, b \leftarrow 2$

  Iteration 1: $s = 3, c \leftarrow 1.333, a \leftarrow b, b \leftarrow c$

  Iteration 2: $s = 3.333, c \leftarrow 1.4, a \leftarrow b, b \leftarrow c$

  Iteration 3: $s = 2.733, c \leftarrow 1.4146, a \leftarrow b, b \leftarrow c$

- The *false position* method chooses c in each iteration the same way as in the secant method, but then finds a new bracketing interval for the next iteration as in bisection. This is more reliable than the secant method in that it should converge even for functions with 'flat' parts.

- Algorithm:

  Start by finding two points a and b on either side of the root, such that $f(a) < 0$ and $f(b) > 0$ (as in bisection). Set $c \leftarrow (a + b)/2$.

  For each iteration:

  Find $f(a)$ and $f(b)$

  Estimate the function's slope (first derivative) as

  $$s = \frac{f(b) - f(a)}{b - a}$$

  Compute $c \leftarrow a - f(a)/s$

  Find $f(c)$

  If $f(c) = 0$, return c as the root

  If $f(c) > 0$, set $b \leftarrow c$

  If $f(c) < 0$, set $a \leftarrow c$

  Possible stopping criteria: As in bisection, or if c changes by less than some tolerance between iterations.

- Example for $f(x) = x^2 - 2$:

  Initialize: $a \leftarrow 1, b \leftarrow 2$

  Iteration 1: $s = 3, c \leftarrow 1.4167, f(c) > 0, b \leftarrow c$

  Iteration 2: $s = 2.417, c \leftarrow 1.4138, f(c) < 0, a \leftarrow c$

  Iteration 3: $s = 2.831, c \leftarrow 1.4142, f(c) < 0, a \leftarrow c$

12 Optimization

- Choose x to maximize or minimize a function $f(x)$; examples: minimize $f(x) = -x^2 + 2x$; maximize $f(\theta) = 4\sin(\theta)(1 + \cos(\theta))$

  max $f(x)$ is equivalent to min $-f(x)$

- Local vs. global optimums (maximums or minimums)

- First step should always be to graph the function and assess where the optimum might be

- Some optimization methods:

1) Golden section search (similar in robustness and convergence rate to bisection; finds a local optimum) – named after the golden section ratio $\phi = \frac{\sqrt{5}+1}{2}$ – here given for finding a local minimum

Start with a bracketing interval $[a, b]$ that contains a minimum of $f(x)$, and with $c \leftarrow a + (\phi - 1)(b - a)$. Find $f(a), f(b), f(c)$

Iterate the following until convergence (e.g. until $|b - a| <$ tol, where tol is the maximum allowable error):

Find $d \leftarrow a + (\phi - 1)(c - a)$ and $f(d)$

If $f(d) < f(c)$, set $b \leftarrow c$ and $c \leftarrow d$

Otherwise, set $a \leftarrow b$ and $b \leftarrow d$ (so that c stays the same for the next iteration)

Example: Find minimum of $f(x) = -\sin(x)(1 + \cos(x))$ for x in $[-\pi\, \pi]$:

#	a	b	c	d	$f(c)$	$f(d)$
1	$-\pi$	π	0.74162942	−0.74162942	−1.17357581	1.17357581
2	π	−0.74162942	0.74162942	1.65833381	−1.17357581	−0.90908007
3	−0.74162942	1.65833381	0.74162942	0.17507496	−1.17357581	−0.34570127
4	1.65833381	0.17507496	0.74162942	1.09177934	−1.17357581	−1.29647964
5	1.65833381	0.74162942	1.09177934	1.30818389	−1.29647964	−1.21641886
6	0.74162942	1.30818389	1.09177934	0.95803397	−1.29647964	−1.28855421
7	1.30818389	0.95803397	1.09177934		−1.29647964	

So after 6 iterations, the location of the minimum is narrowed down to $[0.95\, 1.31]$ and the minimum value is under -1.296

2) Root finding for derivative, either analytically or with any numerical method (requires f to be differentiable)

Example: For $f(x) = -\sin(x)(1+\cos(x))$, we have $f'(x) = -(\cos(x)+\cos(2x))$ and $f''(x) = \sin(x)+2\sin(2x)$. Applying Newton's method to find where $f'(x) = 0$ beginning with $x_0 = 1$ gives the sequence 1, 1.04667383, 1.04719747, 1.04719755, 1.04719755 (converging to the minimum, where $f(x) = -1.29903811$).