

E4300: A study guide and review V1.2

Marc Spiegelman

March 12, 2008

1 Sources of Error

Numerical methods are fraught with errors that principally arise from

Model Error This is the error introduced in approximating reality with equations. This is potentially the most severe source of error but is not really within the scope of this course.

Truncation Error This is the error introduced by approximating a function by a simpler one, often a polynomial. For Polynomial approximations, understand *Taylor's theorem* for the expansion of a function $f \in C^{n+1}$ around a point x_0

$$f(x) = T_n(x) + R_n(x)$$

where

$$T_n(x, x_0) = \sum_{k=0}^n \frac{f^{(k)}(x_0)(x - x_0)^k}{k!}$$

is the Taylor Polynomial of order n and

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

is the Taylor Remainder term where ζ is an unknown position in the interval $\zeta \in [x_0, x]$.

In the framework of Taylor's theorem, R_n provides a bound/estimate on the truncation error of a function with respect to its Taylor Polynomial. In particular we are interested in the **Absolute Error** between a function $f(x)$ and its approximation $\hat{f}(x)$

$$e = |f(x) - \hat{f}(x)|$$

and the **Relative Error**

$$r = \frac{|f(x) - \hat{f}(x)|}{|f(x)|} = e/|f(x)|$$

In the case of a Taylor Polynomial approximation the absolute error is simply $e = |R_n|$ and the relative error is $r = |R_n|/|f(x)|$. Be able to estimate both relative and absolute truncation errors for simple functions.

Floating Point Error This is the error introduced by approximating real numbers with finite precision floating point numbers of the form

$$f = d_1.d_2d_3d_4 \dots d_p \times \beta^E$$

where d_1 to d_p represent p digits of precision in the *fraction* or *mantissa*, β is the *base* and E is the *exponent*. Important features of floating point systems include

- There is a smallest number below which occurs *underflow* (for IEEE double precision UFL $\sim 2.22 \times 10^{-308}$ numbers smaller than UFL return zero (or subnormal floating point numbers))
- There is a largest number above which generates *overflow* (IEEE DP OFL $\sim 1.79 \times 10^{308}$, numbers greater than OFL return `inf`).
- Floating point numbers are not evenly distributed on the number line.
- Within any log unit of the base (i.e. for a fixed value of E) Floating point numbers are evenly spaced and separated by a number corresponding to 1 *unit-in-the-last-place* or *ULP* which corresponds to the smallest floating point number that can be added to a number to get it to round up.
- there is a number ϵ_{mach} that is the smallest floating point number that can be added to 1 such that $1 + \epsilon_{\text{mach}} > 1$ (for IEEE DP $\epsilon_{\text{mach}} = 2^{-52} \sim 2.22 \times 10^{-16}$). ϵ_{mach} corresponds to 1 ULP in the interval $[1, 2]$ in a binary floating point system.
- Note ϵ_{mach} is much, much larger than UFL and limits the size of two numbers that can be added accurately.
- Subtraction of two numbers that are close to each other can lead to *catastrophic* cancellation.
- In matlab `eps(x)` returns 1 ULP for any number x . `eps(1) = ϵ_{mach}` .

You should know how to do simple calculations in toy floating point systems e.g. 3-digit decimal precision. You should also be able to determine the number of *significant digits* in decimal precision defined by the largest number p such that

$$r < 5 \times 10^{-p}$$

where r is the relative error. For example if $f = \pi$ and $\hat{f} = 3.14$, $r = |\pi - 3.14|/|\pi| = 5.0696 \times 10^{-4}$, then \hat{f} is accurate to 3 decimal digits. If $\hat{f} = 22/7$, $r = 4.025 \times 10^{-4}$ and $\hat{f} \sim 3.142$ is accurate to 4 significant digits.

2 Root Finding: Solving $f(x) = 0$

Convergence of Iterative schemes Understand ideas of convergence rates $|e_{n+1}| = C|e_n|^r$ where C is a constant and r is the convergence rate.

- Linear Convergence $C < 1, r = 1$
- Super Linear Convergence $r > 1$

- Quadratic Convergence $r = 2$

Fixed point iteration a value x is said to be a *fixed-point* of a function $g(x)$ if

$$x = g(x)$$

A *fixed-point iteration* is an iterative scheme starting at some value x_0 and iterating

$$x_{n+1} = g(x_n) \quad (1)$$

Analysis of asymptotic convergence of fixed-point iteration if x^* is a fixed point of $g(x)$ such that $x^* = g(x^*)$, it is straightforward to analyze the convergence behavior of a fixed-point iteration in a neighborhood close to the fixed point. We begin by redefining

$$\begin{aligned} x_{n+1} &= x^* + e_{n+1} \\ x_n &= x^* + e_n \end{aligned}$$

where e_n is the *absolute error* from the fixed-point at iteration n . Given these definitions, Eq. 1 can be written

$$x^* + e_{n+1} = g(x^* + e_n) \quad (2)$$

Now, if we assume we are close to the fixed point, we can expand the RHS around x^* using Taylor's Theorem as

$$g(x^* + e_n) = g(x^*) + g'(x^*)e_n + g''(\zeta)e_n^2/2 \quad (3)$$

where $\zeta \in [x^*, x^* + e_n]$. Using Eq. (3) in Eq. (2) together with $x^* = g(x^*)$ yields

$$e_{n+1} = g'(x^*)e_n + g''(\zeta)e_n^2/2 \quad (4)$$

Now if $g'(x^*) \neq 0$, the leading term in this expansion will be $O(e_n)$ and Eq. (4) leads to the linear convergence relationship

$$|e_{n+1}| = C|e_n| \quad (5)$$

where $C = |g'(x^*)|$. This can be rewritten as

$$|e_n| = C^n|e_0|$$

which implies, that this iterative scheme will only converge to the fixed point (i.e. $|e_n| \rightarrow 0$ as $n \rightarrow \infty$) if $|g'(x^*)| < 1$. If $|g'(x^*)| > 1$ the iteration will diverge and blow up. If $|g'(x^*)| = 1$ it will oscillate without converging.

Newton's method as fixed point iteration Newton's method for finding roots of a function $f(x)$ can be considered as a special case of a fixed-point iteration. Newton's method can be written concisely as

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \quad (6)$$

where now

$$g(x) = x - \frac{f(x)}{f'(x)} \quad (7)$$

Clearly if x is a **root** of f such that $f(x) = 0$, then x is also a fixed point of $g(x)$ (as long as $f'(x)$ is bounded).

Asymptotic Convergence of Newton's Method Thus we can use the above analysis to analyze the asymptotic convergence behavior of Newton's method near a root. Given Eq. (7) we can now calculate

$$g'(x) = 1 - \frac{f'}{f} + \frac{f f''}{(f')^2} = \frac{f(x) f''(x)}{(f'(x))^2} \quad (8)$$

and

$$g''(x) = \frac{f''}{f'} + \frac{f f'''}{(f')^2} - 2 \frac{f (f'')^2}{(f')^3} \quad (9)$$

however, at the fixed point x^* , $f(x^*) = 0$ therefore $g'(x^*) = 0$ and

$$g''(x^*) = \frac{f''(x^*)}{f'(x^*)} \quad (10)$$

in which case Eq. (4) becomes

$$|e_{n+1}| = C |e_n|^2 \quad (11)$$

where $C = |f''/2f'|$ (and we have assumed that we are sufficiently close to the fixed point that $\zeta \approx x^*$). Thus Newton is quadratically convergent (close to a simple root).

Convergence of Newton for p multiple roots With similar analysis, it is easy to show that near a multiple root where $f(x) \propto (x - x^*)^p$ to leading order (i.e. there are p multiple roots such that the first $p - 1$ derivatives of f vanish at x^* (i.e. $f^{(n)}(x^*) = 0$ for $n = 0, 1, \dots, p - 1$) then Newton's method is only **linearly convergent** with convergence factor $C = (p - 1)/p$. I.e. a double root converges with error that decreases by $1/2$ at each iteration. A triple root converges more slowly as $(2/3)^n$.

1-D Root Finding Algorithms

- Bisection (Linear $C = 1/2$)
- Newton $x_{n+1} = x_n - f(x_n)/f'(x_n)$ (Quadratic... but see above)
- Secant $x_{n+1} = x_n - \frac{f(x_n)(x_n - x_{n-1})}{f(x_n) - f(x_{n-1})}$ (Superlinear)
- Also useful to know is Inverse Interpolation (see Lagrange interpolation)
- Brent's algorithm: guaranteed to converge if given an initial bracket.

3 Interpolation

Polynomial Interpolation Understand that there is a **unique** interpolating polynomial $P_n(x)$ of degree n that passes exactly through $n + 1$ points (x_i, y_i) for $i = 1, 2, \dots, n + 1$. The polynomial will be unique if all of the x_i 's are distinct. A more complete set of notes on Interpolation can be found on the Course Website in the interpolation section.

You should understand how to find the interpolating polynomial of degree n points using different bases.

- **monomial basis** $1, x, x^2 \dots x^n$ (use Vandermonde matrix, or polyfit)

- **Lagrange polynomial basis** $L_k(x)$. Know how to calculate Lagrange polynomials given a set of unevenly spaced points $x_0 \dots x_n$. The general formula is

$$L_k(x) = \prod_{i=0, i \neq k}^n \frac{(x - x_i)}{(x_k - x_i)} \quad (12)$$

which is the product of the fractional distances of the point x from the point x_k . For example, the three Lagrange Polynomials defined by the nodes x_0, x_1 and x_2 are

$$L_0(x) = \frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)} \quad L_1(x) = \frac{(x - x_0)(x - x_2)}{(x_1 - x_0)(x_1 - x_2)} \quad L_2(x) = \frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)} \quad (13)$$

As shown in the notes, the interpolating polynomial is easily defined in the Lagrange basis as

$$P(x) = \sum_{k=0}^{n-1} y_k L_k(x)$$

which is a linear combination of the lagrange polynomials, simply weighted by the values at the nodes.

(You might also want to be comfortable with linear interpolation and piecewise cubic interpolation using Lagrange polynomials).

Some properties of the Lagrange Polynomials

- for n points $L_k(x)$ are all polynomials of degree $n - 1$ (i.e. in the example above all the polynomials are quadratics)
- the lagrange polynomials have the crucial property that

$$L_i(x_j) = \delta_{ij} \quad (14)$$

where δ_{ij} is the Kronecker delta ($= 1$ if $i = j$ or 0 for $i \neq j$).

- **Newton Polynomials:** know them but I won't emphasize them
- Understand the pitfalls of using high-order polynomials on regularly spaced points. Particularly Runge effects and that using the Chebyshev points will minimize the error in polynomial interpolation on a fixed interval.

Piecewise Polynomial Interpolation These problems are a bit too hard to do by hand (but easy in matlab). Understand both 1 and 2-D ideas for different levels of continuity

- C_0 : piecewise continuous: just use local n points.. e.g. *piecewise linear just connects the dots*
- C_1 : Hermite Cubic Polynomials: (matlab pchip)
- C_2 : cubic splines (matlab spline)

4 Numerical Quadrature and differentiation

Basic ideas

- All quadrature rules can be written as

$$I \equiv \int_a^b f(x)dx \approx (b-a) \sum_{i=1}^n w_i f(x_i)$$

plus an error term that depends on derivatives of f and powers of the step-size $h = (b-a)$.

- If the positions x_i are known, the quadrature weights can be calculated using the *method of undetermined coefficients* which involves a Vandermonde matrix if you use a monomial basis.

1-step Newton-Cotes formulas (and their errors) you should be able to derive all of the following from either undetermined coefficients, or *using the Lagrange interpolating polynomial*. To estimate the errors you need to use Taylor's series to see which terms are canceling.

- Midpoint rule: $I \approx hf(h/2) + O(f''h^3)$
- Trapezoidal rule: $I \approx h[f(0)/2 + f(h)/2] + O(f''h^3)$
- Simpson's Rule: $I \approx h[f(0)/6 + (2/3)f(h/2) + f(h)/6] + O(f^{iv}h^5)$ Note: $S(f) = (2/3)M(f) + (1/3)T(f)$

or as 3-point stencils

- Midpoint: $M = h[0 \quad 1 \quad 0]$
- Trapezoidal: $M = h[1/2 \quad 0 \quad 1/2]$
- Simpsons: $M = h[1/6 \quad 2/3 \quad 1/6]$

Degree of Precision A quadrature rule is said to possess a *degree of precision* p such that it is exact for integrating polynomials up to degree p . For example, the degree of precision for both Midpoint and trapezoidal rules is $p = 1$ while Simpson's rule has $p = 3$. In general, the degree of precision for Newton-Cotes formulas with odd numbers of points is $p = n$ while for even numbers of points it is $p = n - 1$

Gauss-Legendre Quadrature For integrals that can be transformed into

$$I = \int_{-1}^1 f(x)dx$$

Gauss-Legendre n -point quadrature rules give exact integrals for all polynomials up to degree $2n - 1$ (i.e. the degree of precision is $p = 2n - 1$) when the quadrature points are taken to be the roots of the Legendre Polynomials. E.g

- 2-point rule $P_2(x) = (3x^2 - 1)/2$, $x_i = (-1/\sqrt{3}, 1/\sqrt{3})$, $w_i = (1, 1)$
- 3-point rule $P_3(x) = (5x^3 - 3x)/2$, $x_i = (-\sqrt{3/5}, 0, \sqrt{3/5})$, $w_i = (5/9, 8/9, 5/9)$.

To use Gauss-Legendre quadrature for the integral

$$I(f) = \int_a^b f(y)dy$$

we need to transform $y \in [a, b]$ to $x \in [-1, 1]$ via

$$y = \frac{(b-a)x + (b+a)}{2}$$

and therefore

$$I(f) = \frac{(b-a)}{2} \int_{-1}^1 g(y(x))dx \approx \frac{(b-a)}{2} \sum_{i=1}^n w_i g\left(\frac{(b-a)x + (b+a)}{2}\right)$$

Extended Newton Cotes formulas The final practical quadrature rules concern regular tiling of the interval $(b-a)$ with $N+1$ points giving N panels of width of width $h = (b-a)/N$. The two most useful extended formulas and their error estimates are

- Extended Trapezoidal rule

$$T_N = h \left(\frac{f_0}{2} + f_1 + f_2 + \dots + f_{N-1} + \frac{f_N}{2} \right) + O\left(\frac{f''(b-a)^3}{N^2}\right)$$

- Extended Simpsons rule

$$S_N = h \left(\frac{1}{3}f_0 + \frac{4}{3}f_1 + \frac{2}{3}f_2 + \frac{4}{3}f_3 + \dots + \frac{2}{3}f_{N-2} + \frac{4}{3}f_{N-1} + \frac{1}{3}f_N \right) + O\left(\frac{H^5}{N^4}\right)$$

which is easier to calculate using multiple evaluations of the trapezoidal rule at different resolution via

$$S_{2N} = \frac{4}{3}T_{2N} - \frac{1}{3}T_N$$

- Romberg Integration: The above combined rule is an example of Romberg integration which combines estimates of the integral with known error behavior as a function of panel width h and extrapolates to $h = 0$ to give an estimate of the quadrature with infinitely many points. The next higher combination of Extended Simpsons rules gives

$$Q_{2N} = S_{2N} + \frac{1}{15}(S_{2N} - S_N) + O(h^6)$$

(this happens to have the name of Booles' Rule) which is equivalent to a three-term Romberg integration using extended trapezoidal rule and is the quadrature rule that Matlab uses in *adaptive Simpson's quadrature*

Numerical Differentiation: Finite difference stencils Numerical operators for differentiation of functions can also be derived from polynomial interpolation. Very simply, given n points with an interpolating polynomial $P(x) = \sum_{i=0}^{n-1} f_i L_i(x)$. The derivative of this polynomial at any point x is simply $P'(x) = \sum_{i=0}^{n-1} f_i L'_i(x)$ where $L'_i(x)$ is the derivative of the Lagrange interpolating polynomial. For equally spaced points, standard 2nd order finite difference stencils can be written as

- centered first derivative: $\frac{df}{dx}(x_2) \approx (1/2h) \begin{bmatrix} -1 & 0 & 1 \end{bmatrix}$
- left sided first derivative $\frac{df}{dx}(x_1) = (1/2h) \begin{bmatrix} -3 & 4 & -1 \end{bmatrix}$
- right sided first derivative $\frac{df}{dx}(x_3) = (1/2h) \begin{bmatrix} 1 & -4 & 3 \end{bmatrix}$
- 2nd derivative (all points in the interpolant) $\frac{d^2f}{dx^2} = (1/h^2) \begin{bmatrix} 1 & -2 & 1 \end{bmatrix}$

Note: each stencil forms a row in a general difference matrix D such that given a discrete n point approximation $\mathbf{f} = f(x_i)$ at points x_1, \dots, x_n then we can write the approximation to the derivative as $\frac{df}{dx} \approx D\mathbf{f}$.

5 Ordinary Differential Equations

Basic Ideas Most explicit ordinary differential equation **Initial Value problems** can be written concisely as the dynamical system of first-order equations

$$\frac{d\mathbf{u}}{dt} = \mathbf{f}(t, \mathbf{u}) \quad \mathbf{u}(0) = \mathbf{u}_0$$

where \mathbf{u} is a vector of *state variables* (e.g positions and velocities of orbiting bodies, compositions of radioactive elements, stock prices etc.)

Basic stepping Schemes To understand and derive most of the simple single step algorithms, it's useful to consider the integral of the dynamical system for one time interval $[0, h]$ i.e.

$$\mathbf{u}(t+h) = \mathbf{u}(t) + \int_t^{t+h} \mathbf{f}(\tau, \mathbf{u}(\tau)) d\tau$$

or

$$\mathbf{u}(t+h) = \mathbf{u}(t) + \mathbf{k}$$

where $\mathbf{k}(h)$ is integral of the right-hand side function over one timestep. If $f(t)$ were known explicitly we could just approximate k using any of the quadrature rules we've already developed. As it is, we'll still use quadrature but we need to "feel" our way forward in time to evaluate the integral. But if we write a basic 1-step scheme to get from value \mathbf{u}_n at time t_n to value \mathbf{u}_{n+1} at time $t_{n+1} = t_n + h$ then we can readily derive the following 1-step explicit schemes (and their errors)

- Forward Euler (calculate k by a 1st order rectangle rule)

$$\begin{aligned} \mathbf{k}_1 &= h\mathbf{f}(t_n, \mathbf{u}_n) \\ \mathbf{u}_{n+1} &= \mathbf{u}_n + \mathbf{k}_1 + O(h^2) \end{aligned}$$

- Mid-point scheme (calculate \mathbf{k} by iterated mid-point rule)

$$\begin{aligned} \mathbf{k}_1 &= h\mathbf{f}(t_n, \mathbf{u}_n) \\ \mathbf{k}_2 &= h\mathbf{f}(t_n + h/2, \mathbf{u}_n + \mathbf{k}_1/2) \\ \mathbf{u}_{n+1} &= \mathbf{u}_n + \mathbf{k}_2 + O(h^3) \end{aligned}$$

- 4th order Runge-Kutta scheme (calculate \mathbf{k} by Simpson's rule with 4 function evaluations)

$$\begin{aligned}\mathbf{k}_1 &= h\mathbf{f}(t_n, \mathbf{u}_n) \\ \mathbf{k}_2 &= h\mathbf{f}(t_n + h/2, \mathbf{u}_n + \mathbf{k}_1/2) \\ \mathbf{k}_3 &= h\mathbf{f}(t_n + h/2, \mathbf{u}_n + \mathbf{k}_2/2) \\ \mathbf{k}_4 &= h\mathbf{f}(t_n + h, \mathbf{u}_n + \mathbf{k}_3) \\ \mathbf{u}_{n+1} &= \mathbf{u}_n + \frac{1}{6}(\mathbf{k}_1 + 2(\mathbf{k}_2 + \mathbf{k}_3) + \mathbf{k}_4) + O(h^5)\end{aligned}$$

Implicit methods In addition to explicit schemes that use only current (or past information), implicit schemes use future information. This tends to greatly improve the stability and allowable time-step of these schemes (particularly for stiff systems), however, they do it at the expense of often developing systems of non-linear equations (see below). Two basic first and second order implicit schemes are

- Backwards euler: do euler step but evaluate the function at time $t + h$ (still a first-order scheme)

$$\mathbf{u}_{n+1} = \mathbf{u}_n + h\mathbf{f}(t_{n+1}, \mathbf{u}_{n+1}) + O(h^2)$$

- Trapezoidal: use trapezoidal rule for quadrature and use the average of f at the old and new time. Second order.

$$\mathbf{u}_{n+1} = \mathbf{u}_n + \frac{h}{2}(\mathbf{f}(t_n, \mathbf{u}_n) + \mathbf{f}(t_{n+1}, \mathbf{u}_{n+1}))$$

Example of forward and backwards Euler for 1-D non-linear problem Consider the first-order ode initial value problem (which doesn't have an analytic solution)

$$y' = y \sin(y) \quad y(0) = 1$$

Forward euler for this problem is the explicit non-linear update

$$y_{n+1} = y_n + hy_n \sin(y_n)$$

whereas the implicit backwards euler problem is

$$y_{n+1} = y_n + hy_{n+1} \sin(y_{n+1})$$

which can be written as a non-linear equation for y_{n+1} such that $g(y_{n+1}) = 0$ or

$$y_{n+1} - hy_{n+1} \sin(y_{n+1}) - y_n = 0$$

letting $x = y_{n+1}$ the problem becomes finding the roots of

$$f(x) = x - hx \sin(x) - y_n = 0$$

If we use Newton's method, the $k + 1$ th Newton iteration would look like

$$x_{k+1} = x_k - \frac{x_k(1 - h \sin(x_k)) - y_n}{1 - h(\sin(x_k) + x_k \cos(x_k))}$$

Stability of explicit and implicit schemes: eigenvalues! Consider the general *linear* dynamical system

$$\frac{d\mathbf{u}}{dt} = A\mathbf{u} \quad \mathbf{u}(0) = \mathbf{u}_0$$

where A is an arbitrary $n \times n$ matrix. An Euler scheme update for this system look like

$$\mathbf{u}_{n+1} = \mathbf{u}_n + hA\mathbf{u}_n = (I + hA)\mathbf{u}_n$$

this is a simple iterative system (think power method) that can be rewritten as

$$\mathbf{u}_n = (I + hA)^n \mathbf{u}_0$$

which will only stay bounded if the spectral radius of the *iteration matrix* $\rho(I + hA) \leq 1$. In general, this puts constraints on the possible size of the time-step h . By using the shift and scaling rules, if λ is an eigenvalue of A then $1 + h\lambda$ is an eigenvalue of $(I + hA)$. Therefore, for this euler step to be stable, it requires that $|1 + h\lambda_{max}| \leq 1$. If the eigenvalues of A are real and negative, this implies that $h < 1/|\lambda_{max}|$ which can put large constraints on the time step if there are some very large eigenvalues. However, the backwards Euler scheme for this problem looks like

$$\mathbf{u}_{n+1} = \mathbf{u}_n + hA\mathbf{u}_{n+1}$$

or rearranging

$$(I - hA)\mathbf{u}_{n+1} = \mathbf{u}_n \quad \text{OR} \quad \mathbf{u}_{n+1} = (I - hA)^{-1}\mathbf{u}_n$$

or as an iterative scheme

$$\mathbf{u}_n = [(I - hA)^{-1}]^n \mathbf{u}_0$$

which has eigenvalues $1/(1 - h\lambda)$ and thus if $\Re\lambda < 0$ the overall scheme will be unconditional stable for all time steps h

Accuracy and adaptive time-stepping Stability is not the same thing as accuracy and a good ODE solver will attempt to adjust the time-step h such that the *relative truncation error* between two different techniques with **known** error behavior is within some specified tolerance.

The general idea is straightforward. Suppose we have two methods of different accuracies that estimate the value of the dynamical system \mathbf{u}_{n+1} after a single step of size h_0 . We can write the first method as

$$\mathbf{u}_{n+1} = \mathbf{u}_n + \mathbf{k} + O(h^p)$$

and the second as

$$\mathbf{u}_{n+1}^* = \mathbf{u}_n + \mathbf{k}^* + O(h^q)$$

if method 1 is more accurate than method 2 then $p \geq q$. Either way, the *relative truncation error* between the two methods will be dominated by the cruder estimate i.e.

$$\Delta_0 = \|\mathbf{u}_{n+1} - \mathbf{u}_{n+1}^*\| \propto h_0^q$$

If we really wanted an error Δ_1 good to some specified tolerance tol , then there is some step-size h_1 such that $\Delta_1 \propto h_1^q$. Thus

$$\frac{\Delta_1}{\Delta_0} = \left(\frac{h_1}{h_0}\right)^q$$

which gives us a way for predicting the next step-size h_1 as

$$h_1 = h_0 \left(\frac{\Delta_1}{\Delta_0} \right)^{1/q}$$

The only real issue is how to choose Δ_1 . One simple option is just to use the relative error

$$r = \frac{\|\mathbf{u}_{n+1} - \mathbf{u}_{n+1}^*\|}{\|\mathbf{u}_n\|} = \frac{\Delta_1}{\|\mathbf{u}_n\|} \leq \text{tol}$$

and set it to some tolerance tol . One needs to be a bit careful around zero crossings however. If one does this, however, then a simple formula for estimating the appropriate new timestep is

$$h_1 = h_0 \left(\frac{\text{tol}}{\Delta_0} \right)^{1/q}$$

Some possible pairings There are many possible pairings of techniques that yield different values of q . Some examples are

- Step-doubling 4th-order Runge Kutta: here we just take two steps, one of size h and one of size $h/2$. Both estimates will still be $O(h^5)$ so $q = 5$
- Embedded 4-5th order RK. The embedded schemes uses two different combinations of function evaluations to generate the estimates. The first is $O(h^5)$ and the second is $O(h^6)$ so $q = 5$ again.
- low order schemes: you could also do this with something like mid-point schemes $O(h^3)$ and Euler schemes $O(h^2)$ which would yield something with $q = 2$.

6 Solving Systems of non-linear Equations: Newton's method

Systems of linear equations are well understood and can be solved using the techniques of numerical linear algebra. Systems of non-linear equations can be considerably harder to solve although the basic techniques are simple enough to understand. In general, solving systems of non-linear equations are the n -Dimensional extension to root-finding for equations of one variable, however, the basic problem of solving $f(x) = 0$, now generalizes to solving $\mathbf{F}(\mathbf{x}) = \mathbf{0}$ where \mathbf{F} is a vector valued function of a vector of unknowns \mathbf{x} that maps $\mathbb{R}^n \rightarrow \mathbb{R}^n$. For example, two non-linear equations in two unknowns $\mathbf{x} = (x_1, x_2)$ could be written

$$F_1(\mathbf{x}) = x_1^2 + 2x_2^2 - 10 \quad (15)$$

$$F_2(\mathbf{x}) = x_1 + 5x_2 - 2 \quad (16)$$

$$(17)$$

where

$$\mathbf{F}(\mathbf{x}) = \begin{bmatrix} F_1(\mathbf{x}) \\ F_2(\mathbf{x}) \end{bmatrix}$$

In general, systems of non-linear equations are much harder to analyze and solve than linear systems and not much can be said generally about existence and uniqueness of solutions. Moreover, many of the standard techniques for rootfinding in 1 dimension do not generalize to higher-dimensional systems. One that does, however, is **Newton's Method**.

As a reminder, Newton's method for equations of 1-variable can be written as a fixed point method as

$$x_{n+1} = x_n - f(x_n)/f'(x_n)$$

which will converge quadratically to the solution if started either very close to the root or for regions near the root where the system is well approximated by a linear problem. Newton is exact if $f(x)$ is a straight-line.

Similarly, for systems of non-linear equations, Newton's method can be derived by expanding $\mathbf{F}(\mathbf{x})$ in a Truncated Taylor series as

$$\mathbf{F}(\mathbf{x} + \delta) \approx \mathbf{F}(\mathbf{x}) + J(\mathbf{x})\delta$$

where $J(\mathbf{x})$ is the *Jacobian* of \mathbf{F} whose i, j th component is given by

$$J_{ij} = \frac{\partial F_i}{\partial x_j}$$

For the simple 2-D example given above, the Jacobian is

$$J(\mathbf{x}) = \begin{bmatrix} 2x_1 & 4x_2 \\ 1 & 5 \end{bmatrix}$$

Given the truncated Taylor series, we set $\mathbf{F}(\mathbf{x} + \delta) = 0$ and solve

$$J\delta = -\mathbf{F}(\mathbf{x})$$

for the **vector** δ . We then update \mathbf{x} to $\mathbf{x} + \delta$ and repeat. Like 1-D newton, this can be succinctly written as a vector-valued fixed point iteration for the next estimate of \mathbf{x} as

$$\mathbf{x}_{n+1} = \mathbf{x}_n - J(\mathbf{x}_n)^{-1}\mathbf{F}(\mathbf{x}_n)$$

starting from some initial guess \mathbf{x}_0 . **Note:** symbolically we can write the problem using J^{-1} however, in reality we would use Gaussian elimination (e.g. \backslash), or even an approximate solver) to solve $J\delta = -\mathbf{F}$ for the correction vector δ . Some important properties/comments about Newton's Method.

Convergence of n -D Newton Like its 1-D counterpart, it can be shown that the asymptotic convergence of n -D Newton is quadratic if started close enough to the fixed point. But you do need to be very close for this to work.

Failure of Newton Like the 1-D case, Newton can fail for values of \mathbf{x} such that the Jacobian is singular or even ill-conditioned.

1-D newton is still Newton Actually, the 1-D problem is simply a special case of the general n -D problem for a system with 1 equation in 1 unknown. In this case, the Jacobian is 1×1 , $J = [f'(x)]$, $J^{-1} = [1/f']$ and the n -D algorithm simply reduces to the 1-D algorithm

n -D Newton is exact for linear systems In the special case that the systems of equations can be expressed as a linear system (i.e. $\mathbf{F}(\mathbf{x}) = A\mathbf{x} - \mathbf{b}$), then it is easy to show that the Jacobian is given simply by $J = A$ and then Newton's method can be shown to converge to the exact

answer $\mathbf{F}(\mathbf{x}) = \mathbf{0}$ for $\mathbf{x} = A^{-1}\mathbf{b}$ in one iteration. To show this let \mathbf{x}_0 be any initial guess and $J = A$. Then the first iteration of Newton is

$$\mathbf{x}_1 = \mathbf{x}_0 - J^{-1}\mathbf{F}(\mathbf{x}_0)$$

or

$$\mathbf{x}_1 = \mathbf{x}_0 - A^{-1}(A\mathbf{x}_0 - \mathbf{b}) = \mathbf{x}_0 - \mathbf{x}_0 + A^{-1}\mathbf{b} = A^{-1}\mathbf{b}$$

This is equivalent to the statement that 1-D newton converges exactly if $f(x)$ is a straight line.

7 Numerical Linear Algebra

A core problem that occurs in many aspects of numerical methods is the numerical solution of linear algebra problems. There are three fundamental problems that can be succinctly describe by

- $A\mathbf{x} = \mathbf{b}$: Square systems of linear equations
- $A^T A\mathbf{x} = A^T \mathbf{b}$: Least-square problems
- $A\mathbf{x} = \lambda\mathbf{x}$: Eigen Problems

Matlab provides a particularly clean syntax and interface for solving these problems with “backslash” (`\` e.g. `x=A\b`) providing solvers for the first two problems via the *LU* decomposition with partial pivoting for square systems and the *QR* factorization for overdetermined, least-squares problems. `eig(A)` provides general eigenvalue, eigenvector results for dense matrices. The rest of this section just fleshes out the basic important features of these fundamental algorithms as well as numerical issues involved in solving numerical linear algebra.

Vector and Matrix Norms In solving numerical linear algebra problems it is often important to discuss how good a solution we’ve attained (or how well *conditioned* a problem is). To do this we need to define the *norm* of vectors and matrices. Here we just use the vector p norms and the matrix p norms induced by the vector norms. Definitions

Vector p norm

$$\|\mathbf{x}\|_p = \left[\sum_{i=1}^n |x_i|^p \right]^{1/p}$$

is a scalar that returns the “length” of a vector with different weightings. Standard values of p are

- $p = 1$: 1-norm or “Manhattan distance”, is simply `sum(abs(x))`
- $p = 2$: 2-norm or Euclidean norm, standard definition of length
- $p = \infty$ “infinity” norm, which is simply `max(abs(x))`

For vectors, the p norms satisfy the following relationships

- $\|\cdot\|_1 \geq \|\cdot\|_2 \geq \|\cdot\|_\infty \geq 0$
- $\|\alpha\mathbf{x}\| = |\alpha|\|\mathbf{x}\|$, for any norm

- $\|\mathbf{x} + \mathbf{y}\| \leq \|\mathbf{x}\| + \|\mathbf{y}\|$ for any norm

Matrix p -norm Given these definitions and properties, the induced matrix p -norms are defined as

$$\|A\|_p = \max_{\text{all } \mathbf{x}} \frac{\|A\mathbf{x}\|_p}{\|\mathbf{x}\|_p}$$

and can be thought of as the *maximum* distortion of the *unit-sphere* in any given norm (the unit-”sphere” is simply the set of all unit vectors $\|\mathbf{x}\| = 1$ for any given norm). The matrix p -norms for $p = 1, 2, \infty$ can be computed as follows.

- $p = 1$: $\|A\|_1$ is the maximum value of the 1-norms of the *columns* of A
- $p = 2$: harder to calculate, $\|A\|_2$ is the maximum singular value σ_{\max} from the singular value decomposition
- $p = \infty$: is the maximum value of the 1-norms of the *rows* of A (or is simply $\|A^T\|_1$).

Most of the properties of the matrix p -norms follow from the underlying vector norms. In particular

- $\|A\| \geq 0$ in all norms
- $\|\alpha A\| = |\alpha| \|A\|$ for all scalar α
- $\|AB\| \leq \|A\| \|B\|$ (product rule)
- $\|A + B\| \leq \|A\| + \|B\|$ triangle inequality

Other useful properties are

- if D is a diagonal matrix $\|D\| = \max D_i$ in all norms
- $\|I\| = 1$ for the identity matrix in all norms.
- $\|Q\|_2 = 1$ for all orthonormal matrices Q in the 2-norm.

Condition Number The main purpose of defining the matrix p -norms is to define the *condition number*

$$\text{cond}(A) = \|A\| \|A^{-1}\|$$

which is a measure of how close to singular a matrix is. Definitions and properties

- $\text{cond}(I) = 1$ in all norms. The best conditioned matrices have $\text{cond}(A) = 1$
- Singular matrices are defined to have $\text{cond}(A) = \infty$
- $\text{cond}_2(A) = \sigma_{\max}/\sigma_{\min}$
- For $A\mathbf{x} = \mathbf{b}$ The condition number describes how a small change in the right-hand side $\Delta\mathbf{b}$ affects a change in the solution $\Delta\mathbf{x}$ by

$$\frac{\|\Delta\mathbf{x}\|}{\|\mathbf{x}\|} \leq \text{cond}(A) \frac{\|\Delta\mathbf{b}\|}{\|\mathbf{b}\|}$$

- It also can relate the size of the relative forward error to the relative residual

$$\frac{\|\Delta\mathbf{x}\|}{\|\mathbf{x}\|} \leq \text{cond}(A) \frac{\|\mathbf{b} - A\mathbf{x}\|}{\|\mathbf{b}\|}$$

which shows that a small-residual $\mathbf{r} = \mathbf{b} - A\mathbf{x}$ only implies a small error in the solution if the matrix is well conditioned.

Square Linear Systems Direct solution of Square linear problems is almost always done by *Gaussian Elimination with Partial Pivoting* which can be succinctly written as $PA = LU$ where

- P is a *permutation* matrix that reorders the rows of A and \mathbf{b}
- L is a lower triangular matrix with 1's on the diagonal and the multipliers ℓ_{ij} in the i, j position.
- U is an upper triangular systems with pivots on the diagonal.

In general, Gaussian Elimination takes $A \rightarrow U$ and $\mathbf{b} \rightarrow \mathbf{c}$ where $L\mathbf{c} = \mathbf{b}$. To solve the linear system $A\mathbf{x} = \mathbf{b}$ we first factorize $PA = LU$ then solve

- $L\mathbf{c} = P\mathbf{b}$ for \mathbf{c} by *forward substitution*
- $U\mathbf{x} = \mathbf{c}$ by *back substitution*

Operation costs for LU of Dense $n \times n$ matrices are $O(n^3/3)$ for factorization and $O(n^2)$ for forward and back substitution. *Tri-diagonal* systems can be solved in $O(n)$ operations.

Partial Pivoting is essential for numerical stability and (almost) always guarantees a small residual $\mathbf{r} = \mathbf{b} - A\mathbf{x}$.

Least Squares Problems For overdetermined problems of form $A\mathbf{x} = \mathbf{b}$ where $A \in \mathbb{R}^{m \times n}$, $\mathbf{b} \in \mathbb{R}^m$ and $m > n$, the linear least-squares problem finds the smallest residual in the 2-norm, i.e. finds a solution $\hat{\mathbf{x}}$ that minimizes

$$\|\mathbf{r}\|_2 = \|\mathbf{b} - A\hat{\mathbf{x}}\|_2$$

There are basically two-different methods for finding the same least-squares solution

- **Normal Equations:** Form the *normal Equations* $A^T A \hat{\mathbf{x}} = A^T \mathbf{b}$ which is a square, symmetric positive definite system which can be solved by the *LU* decomposition¹. The normal equations *always have at least one solution* but will not be unique if A is not full-column rank (which implies that $A^T A$ is singular).
- **Orthogonalization Techniques and the QR factorization** A better way to solve these problems is through the *QR* factorization which finds $A = QR$ where Q is an orthonormal matrix whose columns form an orthonormal basis for the columns space of A and R is an upper triangular matrix. The basic approach is a two-step process
 - Factor $A = QR$ (by various techniques)
 - Solve $R\hat{\mathbf{x}} = Q^T \mathbf{b}$ by back substitution

We have discussed two principal algorithms for calculating the *QR* factorization

Modified Gram-Schmidt This algorithm transforms $A \rightarrow Q$ by a successive set of projections and calculates R as a by-product. As an example, given three vectors

¹Actually you can solve it about two times faster using the *Cholesky Factorization* for symmetric Positive Definite matrices, but I didn't teach this

$\mathbf{a}_1, \mathbf{a}_2, \mathbf{a}_3$ in \mathbb{R}^m ($m > 3$) that form the columns of A we transform them into three orthonormal vectors $\mathbf{q}_1, \mathbf{q}_2, \mathbf{q}_3$ in the following steps

$$\begin{aligned}\mathbf{q}_1 &= \mathbf{a}_1 / \|\mathbf{a}_1\|_2 \\ \mathbf{b}_2 &= \mathbf{a}_2 - \mathbf{q}_1(\mathbf{q}_1^T \mathbf{a}_2) \\ \mathbf{q}_2 &= \mathbf{b}_2 / \|\mathbf{b}_2\|_2 \\ \mathbf{b}_3 &= \mathbf{a}_3 - \mathbf{q}_1(\mathbf{q}_1^T \mathbf{a}_3) - \mathbf{q}_2(\mathbf{q}_2^T \mathbf{a}_3) \\ \mathbf{q}_3 &= \mathbf{b}_3 / \|\mathbf{b}_3\|_2\end{aligned}$$

Given Q , we can calculate R simply as $R = Q^T A$. This can be written succinctly as an algorithm in matlab to calculate Q in the place of A and the non-zero components of R as

```
for i=1:n % loop over columns
    R(i,i) = norm(A(:,i));

    % create the i'th unit vector q_i

    A(:,i) = A(:,i)/R(i,i);

    % now remove the projection of q_i from
    %the rest of the columns

    for j = i+1:n
        R(i,j) = A(:,i)'*A(:,j); % dot product q_i'*a_j
        A(:,j) = A(:,j) - R(i,j)*A(:,i); % remove projection
    end
end

Q=A; % copy into output matrix Q
```

Orthogonalization by Householder Reflection matrices This method is more closely related to the LU decomposition and directly transforms $A \rightarrow R$ and $\mathbf{b} \rightarrow Q^T \mathbf{b}$. The key ingredient is the orthogonal *Householder reflection matrices* defined by

$$H = I - \rho \mathbf{v} \mathbf{v}^T$$

with $\rho = 2/\mathbf{v}^T \mathbf{v}$

which are used in a similar manner to elementary elimination matrices to put zeros below the diagonal in A . This algorithm is rather hard to implement by hand but can be written reasonably succinctly in matlab as follows.

```
for k = 1:n % loop over columns
    % Introduce zeros below the diagonal in the k-th column.
    % Use Householder transformation, I - rho*v*v'.
    i = k:m; % just use the rows from k to m
    v = A(i,k);
```

```

alpha = norm(v);

% Skip transformation if column is already zero.
if alpha ~= 0
    % calculate v and rho
    alpha = -sign(v(1))*alpha;
    v(1) = v(1) + alpha;
    rho = 2/norm(v);

    % Update the k-th column.
    A(i,k) = 0;
    A(k,k) = alpha;

    % Apply the transformation to remaining columns of A.
    j = k+1:n;
    u = rho*(v'*A(i,j)); % this is a row vector
    A(i,j) = A(i,j) - v*u; %v*u is a rank one matrix

    % Apply the transformation to b.

    tau = rho*(v'*b(i));
    b(i) = b(i) - tau*v;
end
end

```

Eigen Problems In the interest of time... we're going to skip eigen problems for the moment.

8 Two-point ODE-Boundary Value problems

The final class of problems, which bring many of the previous techniques together is the solution of two-point boundary value problems for ODE's. Our classic example was the linear ODE

$$-u'' + u = f(x) \quad 0 < x < L$$

with boundary conditions

$$u(0) = \alpha \quad u(L) = \beta$$

The goal is to find an accurate *discrete* approximation to the continuous function $u(x)$. We introduced two (three) basic methods for solving these problems.

Shooting Methods In shooting methods we combine two well-understood algorithms, ODE integrators for initial value problems (e.g. 4th order Runge-Kutta, `ode45` etc) with a root finding algorithm such as Brent's method (e.g. `fzero`). The basic idea is to turn our ODE into a dynamical system

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

where $\mathbf{y} = (u, u')$ and treat it as an initial value problem from one edge of the domain. I.e. we use an ODE integrator to shoot from one of the boundaries with the initial condition

$u(0) = \alpha$ and a guessed initial slope $u'(0) = u'_0$ and adjust the initial slope u'_0 to drive the difference between the value at the right hand boundary and the boundary condition to zero. If we write the error on the right hand side as

$$r(u'_0) = u(0) + \int_{x=0}^L \mathbf{f}(\zeta, \mathbf{y}; u'_0) - \beta$$

then we can use a 1-d rootfinding routine to bracket and find the correct slope u'_0 such that $r = 0$. In matlab this requires writing several problem specific functions

- `r=residualFunction(u0prime)` to be passed to `fzero`. This routine calls `odeXX` to integrate the IVP given u'_0 and returns the residual.
- `rhsFunction` to be passed to `odeXX` to return the right hand side of the dynamical system.

These problems are hard to do by hand and in general can be quite sensitive to the initial guess.

Finite Difference methods These techniques approximate the continuous function $u(x)$ as a finite dimensional vector \mathbf{u} evaluated at N points in a vector \mathbf{x} . Derivatives of the function are approximated by finite differences or other techniques of Numerical Differentiation (see above). In general, the operation of the derivative operator on a continuous function can be approximated as a matrix-vector product

$$\frac{du}{dx} \approx D\mathbf{u}$$

where D is a sparse differentiation matrix whose rows include the stencils derived from Numerical differentiation of local polynomial interpolants. For example, if u is sampled on a regularly spaced mesh with mesh spacing h , then the second order centered derivative at point i can be approximated as

$$\left. \frac{du}{dx} \right|_{x_i} \approx \frac{1}{2h} [u_{i+1} - u_{i-1}]$$

or in stencil notation as

$$\left. \frac{du}{dx} \right|_{x_i} \approx \frac{1}{2h} [-1 \ 0 \ 1] u_i$$

Likewise, combining consistent 1-sided 2nd order derivatives, on a toy grid of 5 points we can approximate the first derivative operator as

$$\frac{du}{dx} \approx D_1\mathbf{u}$$

where D is a sparse matrix with structure

$$D_1 = \frac{1}{2h} \begin{bmatrix} -3 & 4 & -1 & 0 & 0 \\ -1 & 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 & 1 \\ 0 & 0 & 1 & -4 & 3 \end{bmatrix}$$

Note, this matrix is singular such that $D\mathbf{c} = 0$ for any constant vector.

We can also define a second order finite difference second derivative Matrix $d^2u/dx^2 \approx D_2\mathbf{u}$ where

$$D_2 = \frac{1}{h^2} \begin{bmatrix} 1 & -2 & 1 & 0 & 0 \\ 1 & -2 & 1 & 0 & 0 \\ 0 & 1 & -2 & 1 & 0 \\ 0 & 0 & 1 & -2 & 1 \\ 0 & 0 & 1 & -2 & 1 \end{bmatrix}$$

which is also clearly singular.

Putting all of these together with the boundary conditions our continuous Boundary Value problem

$$-u'' + u = f(x) \quad 0 < x < L$$

with boundary conditions

$$u(0) = \alpha \quad u(L) = \beta$$

can be approximated as

$$-D_2^*\mathbf{u} + \mathbf{u} = \mathbf{f}(\mathbf{x})$$

where D_2^* is a modified version of D_2 that accounts for the fixed boundary conditions. The whole problem can be written as the linear system $A\mathbf{u} = \mathbf{f}$

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ -1/h^2 & 2/h^2 + 1 & -1/h^2 & 0 & 0 \\ 0 & -1/h^2 & 2/h^2 + 1 & -1/h^2 & 0 \\ 0 & 0 & -1/h^2 & 2/h^2 + 1 & -1/h^2 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \end{bmatrix} = \begin{bmatrix} \alpha \\ f_2 \\ f_3 \\ f_4 \\ \beta \end{bmatrix}$$

which is a tri-diagonal linear system that can be solved in $O(n)$ operations using gaussian elimination.

Non-linear ODE's These approaches can be extended to handle non-linear ODE BVP's such as

$$-u'' + u^p = f(x) \quad 0 < x < L$$

with boundary conditions

$$u(0) = \alpha \quad u(L) = \beta$$

Shooting methods with explicit ODE integrators can handle non-linear equations relatively easily and very little change in procedure is necessary (although the problem may become much more difficult to solve)

Finite Difference methods also remain pretty much the same, however instead of reducing to a system of linear equation, it reduces to a system of non-linear algebraic equations that can be recast as $\mathbf{F}(\mathbf{u}) = \mathbf{0}$ and solved for \mathbf{u} using a non-linear solver such as Newton.

Collocation methods In the interest of time and the final... I will post what we have here and put up collocation methods for completeness later.

9 Your Notes Here