Comp Prelim Study Material

Trevor Squires
tsquire@clemson.edu

Begin from 8/23/2019
Last updated July 19, 2021

Contents

Part I

List of Topics

Below are a list of topics for Math 8600 and 8610 taken directly from the school's webpage.

1 Math 8600

1. Scientific computing
   - Floating point number system
   - Floating point arithmetic
   - Sensitivity and conditioning
2. Systems of Linear Equations
   - Back solving and forward solving
   - Gauss transformation
   - LU decomposition
   - Cholesky decomposition
   - Band matrix
   - Vector norm, matrix norm, and condition number
   - Sensitivity of a solution of a linear system
3. Linear Least Squares
   - Existence of a solution of a linear least squares problem
   - Normal equations
   - QR decomposition
   - Singular value decomposition
4. Nonlinear equations
   - Rate of convergence
   - Bisection method
   - Regular falsi method
   - Fixed point iteration
   - Newton's method
   - Secant method
5. Interpolation
   - Polynomial interpolation
     - method of undetermined coefficients
     - Lagrange interpolation
     - Neville's algorithm
     - error analysis
   - Piecewise polynomial interpolation
     - hermite cubic interpolation
     - cubic spline interpolation
6. Numerical Integration and Differentiation
   - Newton-Cotes quadrature
   - Gaussian quadrature
   - Composite and adaptive quadrature
   - Richardson's extrapolation

- Romberg integration

7. Initial Value Problems for Ordinary Differential Equations
    - Introduction
    - One step method
        - Euler method
        - Taylor method
        - Runge-Kutta method
        - order of accuracy and error analysis
    - Multi-step method
        - Adams methods
        - predictor-corrector method
    - Stability
    - Stiff equation

2 Math 8610

1. Conditioning and Stability
    - Condition and condition number
    - Forward and backward stability
    - Growth factor and stability of LU and other similar factorizations
2. QR factorization and linear least squares
    - QR factorization by modified Gram-Schmidt
    - QR factorization by Householder reflectors and Givens rotations
    - Linear least squares problem and solution algorithms
3. Singular value decompositions (SVD)
    - Definition of SVD, and its important relations and properties
    - Golub-Khan bidiagonalization and the equivalence of SVD on two symmetric eigenvalue problems
    - Applications such as low-rank approximation
4. Eigenvalue problems and algorithms
    - Diagonalization, complex and real Schur form
    - Reduction to upper Hessenberg/tridiagonal form
    - Shifted QR iteration and its important relations and properties
    - Simultaneous iteration and Arnoldi/Lanczos method for computing several eigenvalues
5. Iterative methods for large sparse linear systems
    - Conjugate gradient (CG) method
    - Generalized minimial residual (GMRES) method
    - Preconditioned linear systems
6. Iteration complexity for all non-iterative algorithms

Part II

8600 Notes

This part contains a condensed set of notes from 8600. The notes are primarily motivated by Ascher and Grief the textbook for 8600. These are not intended to be a replacement to the comp courses, but rather as a supplement. Motivation, big ideas, and clarity will be emphasized, but will be fairly light on details. The fundamentals will be covered, but one is advised to attempt the problems in later sections for full exposure. The listed topics in the previous section will be the backbone of these notes.

Chapter 1

Scientific Computing Overview

We define **scientific computing** as the development and studying of numerical algorithms for solving mathematical problems. In a standard setting, one might model a naturally occurring problem using a mathematical model. This model, usually continuous, is sometimes difficult (and often outright impossible) to describe finitely. A natural step from here is to approximate the continuous (infinite) model with a finite dimensional one - that can be solved on the computer. **Numerical Analysis** is the study of of such approximations and resulting algorithms.

1 Error

Definition. Error is unavoidable in scientific computing. We quantify it with absolute and relative error. Let u be the true value of some quantity and v be the approximation. Then we define relative error as

$$\mathrm{Err}_r(u, v) := \frac{|u - v|}{|u|}$$

and absolute error as

$$\mathrm{Err}_A(u, v) := |u - v|$$

In addition to quantifying error, it may be helpful to describe them. We can identify three main sources of error

1. Modeling - A mathematical formulation rarely exactly describes a real life phenomenon. In practice, instruments do not record exact measurements and human error is omnipresent. These issues give us modeling errors.
2. Approximation - Although many problems can be described using an infinite process, this is not always easy to work with. An replacement of a infinite process with a finite one introduces approximation errors. Even further, we can say that
   - Discretization errors arise from discretizations of a continuous process
   - Convergence errors arise from the termination or truncation of an infinite process
3. Roundoff - Because we desire to use computers to solve our models, we must work in finite precision. This finite precision leads to roundoff errors.

For this note, we will primarily be concerned with roundoff errors.

2 Solving Problems and Algorithm Properties

Once a mathematical model is formulated, an algorithm is sought after to solve the model. We may describe an algorithm by a few characteristics

1. Accuracy - the ability to provide an accurate solution upon termination

2. Efficiency - the effort required to provide an accurate solution. This is usually measured in flops of number of function evaluations

3. Robustness - the reliability of an algorithm. A good algorithm will work in most cases and be able to describe the situations in which it doesn't. It will be reliable and stable.

On the other hand, there are issues that are problem dependent and not algorithm dependent. The most important of these is conditioning.

Definition. Problem conditioning refers to the propensity of change of a solution given a change in input data.

We usually describe conditioning using one of two straight-forward terms

1. Ill-conditioned - a small perturbation in the data would produce a large difference in the result

2. Well-conditioned - the solution is resistant to small changes in the input data

Loosely speaking, we say that the condition number is $\frac{\text{relative output}}{\text{relative input}}$. A well conditioned problem has a condition number close to one. This reasoning has an additional meaning for functions. Note that

$$\text{cond number} = \frac{\frac{f(x)-f(\hat{x})}{f(x)}}{\frac{x-\hat{x}}{x}} = f'(\eta)\frac{x}{f(x)} \leq \max f'(x)\frac{x}{f(x)}$$

Chapter 2

Roundoff Errors

In this section, we discuss the aforementioned roundoff errors. While the details are of importance and can be followed closely for an in-depth analysis, the main take away from this section is when roundoff errors become an issue.

1 Representation

A real number $x \in \mathbb{R}$ can be written as

$$x = \pm(1.d_1 d_2 d_3 \dots d_t) \cdot \beta^e$$

where $d_i \in \{0, 1\}$, e is the integer exponent, and β is the base of representation. Letting $\beta = 2$, this is referred to as the standard binary format. Digits after the decimal point are called the mantissa. That is, to refer to a number on a computer (in binary), we need to know

- the sign
- the mantissa
- the exponent

Note here that the first digit is always a 1 for normalization. Given such a representation, we say that this represents

$$(x) = \pm(\frac{d_1}{\beta^0} + \cdots + \frac{d_t}{\beta^{t-1}}) \cdot \beta^e$$

However, it is far more common to work in the IEEE standard than anything else. For this reason, future discussion will focus on the IEEE standard (with $\beta = 2$).

Definition. The IEEE standard provides specific usage of the 64 bits of precision provided by most modern computers. It breaks them down in the following way

1. 1 bit for the sign of the number
2. 52 bits for the mantissa
3. 11 bits for the exponent

2 Consequences of Finite Number System

There are a number of issues that stem from this representation. A few are listed below

- There are largest and smallest numbers
- There are a finite number of representable quantities
- The absolute difference between two closest is not the same for all numbers
- Every number has a potential error of $\varepsilon_{\text{mach}} = 2^{-52} \approx 10^{-16}$. This error can accumulate if not accounted for and ruin algorithm accuracy.

Possible error sources

Some operations are more prone to significant errors than others (all operations have errors, but not all errors are meaningful). We list a few of these below

1. Adding large and small numbers. In general, the smaller a number is, the more likely it is affected by a large relative error. Adding a large and smaller number, i.e. $1 + 10^{-16}$, is likely to produce large relative error.

2. Product/division of numbers close to 0. If $y << 1$ then xy and $\frac{x}{y}$ can have large relative and absolute error.

3. Subtraction of similarly sized numbers. If $x \approx y$, then $x - y \approx 0$, but due to cancellation error, can be as large as $2\varepsilon_{\mathrm{mach}}$. The relative error is even worse.

Chapter 3

Roots to Nonlinear Equations

In this section we discuss methods of finding roots to nonlinear equations. Per the results of the Abel Ruffini Theorem, there exists no algebraic solution to a general polynomial of degree 5 or higher. As such, we should not expect our solutions to be finite processes, but rather iterative ones. With this in mind, it makes sense to first consider the standard nuances of iterative algorithms: stopping criterion and desired properties.

When solving for roots of nonlinear equations, there are a few stopping criterion that may become useful. Let $\varepsilon > 0$ be some specialized tolerance and $\{x_k\}_{k=1}^N$ be the iterates of a nonlinear solver used to solve $f(x) = 0$.

1. Absolute tolerance
$$|x_n - x_{n-1}| < \varepsilon$$
2. Relative tolerance
$$|x_n - x_{n-1}| < \varepsilon \, |x_n|$$
3. Functional tolerance
$$|f(x_n)| < \varepsilon$$

It is important to note that there is no dominance among these 3 criteria. We simply list them all because some methods of convergence analysis may be more suitable for finding bounds of specific stopping criterion. Furthermore, we would like our algorithm to (in addition to the previous discussion) satisfy some of the following

- Small requirements on the smoothness of f
- Little dependence on function evaluations
- Generalizes easy
- Robust

Last but not least, we more accurately describe what it means for an algorithm to be slow (or fast).

Definition. There are 3 basic types of convergence for iterative methods

1. Linear Convergence
$$|x_{n+1} - x^*| < C \, |x_n - x^*|$$
for $C < 1$.
2. Superlinear Convergence
$$|x_{n+1} - x^*| < p_n \, |x_n - x^*|$$
where $\rho_n \to 0, n \to \infty$.
3. Quadratic Convergence
$$|x_{n+1} - x^*| < M \, |x_n - x^*|^2$$

With these in mind, we look at a few methods in particular

1 Bisection Method

The motivation behind the bisection method is quite simple. For a continuous function f, if $f(a) < 0$ and $f(b) > 0$, then by the IVT, there must exist a c between these points such that $f(c) = 0$. By querying f at the right places, we can reduce the search space. That is, if a and b are the points from above and $f(c) < 0$, then our search interval is reduced from $[a, b]$ to $[c, b]$. Otherwise, it becomes $[a, c]$. In order to minimize the expected interval change, we should choose c such that $c = \frac{a+b}{2}$. Then with each iteration (one function evaluation), our search interval decreases by a factor of 2.

That is, with each iteration, the absolute error decreases by a factor of 2 since

$$|x_{n+1} - x^*| < 0.5\,|x_n - x^*| < (0.5)^n\,|x_0 - x^*|$$

That is, the Bisection Method is a linearly convergent algorithm. Thus, to achieve an $\varepsilon > 0$ tolerance, we need N to satisfy

$$N > \log(\frac{x_0 - x^*}{\varepsilon}) > \log(\frac{b_0 - a_0}{\varepsilon})$$

We summarize the Bisection Method with the following pros and cons

Pros

- Robust (Globally convergent)
- Only uses function evaluations
- Only needs a fixed number of iterations

Cons

- Not very efficient
- Does not generalize

2 Fixed Point Iteration

Fixed Point Iteration, or FPI, is a method of solving $g(x) = x$ for some function g. You may wonder why this discussion belongs in the root solving section of these notes. Hopefully it is not too hard to see that for any equation $f(x) = 0$, we can find a (not necessarily unique) equation $g(x) = x$ such that x^* solves the former if and only if x^* also solves the latter. FPI considers the sequence $\{x_n\}$ where $x_{n+1} = g(x_n)$. If this sequence converges, then we have found a point x_n such that $g(x_n) = x_n$. The conditions on which this sequence converge are demonstrated in the following theorem. If $g \in C([a, b])$ and $a \leq g(x) \leq b$ for all $x \in [a, b]$, then there is a fixed point x^* of g in $[a, b]$. Additionally, if $|g'(x)| < \rho$ for some $\rho < 1$ and any $x \in [a, b]$, then x^* is unique.

Proof. Hint: Consider $\phi(x) = g(x) - x$ and use IVT. □

For convergence, note that

$$\begin{aligned} |x_{k+1} - x^*| &= |g(x_k) - g(x^*)| \\ &\leq |g'(c_k)|\,|x_k - x^*| \\ &\leq |\rho|\,|x_k - x^*| \end{aligned}$$

That is, FPI is a linearly convergent method with $C = \rho$ from the theorem.

Pros

- Easy to generalize
- Only uses function evaluations

Cons

- Not very efficient
- May be difficult to find a good function g

3 Newton's Method

Newton's method is derived from the Taylor expansion of f. Instead of looking for roots of f why not find a root of a linear approximation of f which is a considerably easier task? The linear approximation is available via the Taylor expansion

$$f(x) = f(x_k) + f'(x_k)(x - x_k) + f''(\eta_k)(x - x_k)^2$$

for some η_k. Letting $x = x^*$ and ignoring the error term, this becomes

$$0 \approx f(x_k) + f'(x_k)(x^* - x_k)$$

or that

$$x^* \approx x_k - \frac{f(x_k)}{f'(x_k)}$$

Of course, this is only an approximation so we instead adopt the scheme

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

We can prove that under suitable conditions, this is a quadratically convergent algorithm. Let $f(x^*) = 0, f'(x^*) \neq 0, f \in C^2$. Then for x_0 chosen close enough to x^*, Newton's method converges at least quadratically.

Proof. Consider the Taylor expansion of $f(x^*)$ at $f(x_k)$

$$0 = f(x_k) + f'(x_k)(x^* - x_k) + \frac{1}{2}f''(c_k)(x^* - x_k)^2$$

Rearranging we obtain

$$-\frac{f(x_k)}{f'(x_k)} = x^* - x_k + \frac{f''(c_k)}{2f'(x_k)}(x^* - x_k)^2$$

or that

$$\frac{|e_{k+1}|}{|e_k|^2} = \frac{x_{k+1} - x^*}{(x^* - x_k)^2} = \frac{f''(c_k)}{2f'(x_k)} = M$$

which is our definition of quadratically convergent. □

Pros

- Easy to generalize
- Extremely fast

Cons

- Requires gradient
- Only locally convergent

4 Secant Method

One of the major cons of Newton's method is that it requires us to compute gradient values at each step. The secant method tackles this problem directly by using the secant line to approximate the gradient. The update for the secant method is

$$x_{k+1} = x_k - f(x_k)\frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})}$$

Unfortunately, this approximation loses the quadratic convergence. However, the secant method still achieves a superlinear convergence.

Pros

- Easy to generalize
- Super linear
- No gradient computations

Cons

- Slower than Newton's
- Only locally convergent

Chapter 4

Direct Methods of Solving Linear Systems

Systems of linear equations arise in every mathematical field on a consistent basis. This section overviews a few basic techniques of solving such systems directly.

1 Gaussian Elimination

The most naive approach to solving a system $Ax = b$ is applying a series of row reductions Q such that $QAx = Ix = Qb$. In this way, the solution can be immediately read off as $x = Qb$. One may notice that these row operations Q represent the operation of an inverse for A. Indeed, this method simply computes $x = A^{-1}b$. The computational cost for such an approach is $\mathcal{O}(n^3)$ for the construction of Q and then an addition $\mathcal{O}(n^2)$ for the matrix-vector multiplication Qb. We will see that this is far from the best approach.

2 Forward/Backward Substitution

An equivalently naive approach is instead of reducing A to the identity matrix, why not simply reduce it to an upper (or lower) triangular matrix and then solve the corresponding system? That is, row operations R are performed to transform A into an upper triangular matrix T. Once in upper triangular form, the system $Tx = Rb$ can be solved in $\mathcal{O}(n^2)$. The construction, however, again takes $\mathcal{O}(n^3)$.

3 LU Factorization

A more sustainable method would be to decompose A into upper and lower triangular matrices U and L. In this way, the system $Ax = b$ becomes $LUx = b$. Accordingly, one could solve $Ly = b$ and then $Ux = y$, i.e. one forward substitution and one backward substitution. As it turns out, this takes the same amount of work as the reduction to an upper triangluar matrix **and** is reusable for future $Ax = b$ solves. Furthermore, LU factorization takes less time than finding an inverse, and is more numerically stable.

To compute an LU factorization, simply reduce A to an upper triangular matrix U. For the lower triangular matrix L, the entry L_{ij} $i > j$ is simply the multiplication factor used in the Gaussian elimination process. That is, if $A_{ii} \cdot l_{ij} + A_{ij} = 0$, then $L_{ij} = -l_{ij}$. The diagonal elements are 1 and the remaining elements are computed during the elimination scheme!

4 Pivoting Strategies

Gaussian Elimination is not always stable. For instance, if $A_{ii} \approx \varepsilon_{\text{mach}}$, then a resulting decomposition could be incredibly prone to errors. To avoid this, we may want to swap rows before doing the elimination at every step. That is, interchange rows such that the diagonal element is as large as it can be. This will reduce the instability previously mentioned. However, it no longer is true that $LU = A$. To keep up with the row changes, we must introduce a permutation matrix P such that $LU = PA$. We then simply solve $LU = Pb$ instead. This decomposition is called LU factorization with partial pivoting.

Additional Comments

As the name suggests, partial pivoting is not the only pivoting strategy, nor does it guarantee stability. However, in practice, it tends to perform perfectly fine and is faster than methods that do guarantee stability (complete pivoting). One should always consider the tradeoffs involved when deciding on algorithm. Furthermore, there exists classes of matrices (diagonally dominant, SPD, etc) that are innately resistant to numerical error in decomposition techniques and thus only require basic LU factorization.

5 Condition Number

So far, we have looked at direct methods for solving $Ax = b$. But in the presence of roundoff errors, we should not expect even a direct method to produce a perfectly accurate solution. We would like some cheap method to estimate the relative error $\frac{||x-\hat{x}||}{||x||}$. One such quantity is the residual

$$\hat{r} = b - A\hat{x}$$

It can be shown that even with a small residual, the error can still be incredibly large. That being said, how can we know how good our solution ever is? Well note that

$$\hat{r} = b - A\hat{x} = Ax - A\hat{x} = A(x - \hat{x})$$

so

$$x - \hat{x} = A^{-1}\hat{r}$$

and by Cauchy Schwarz, we have

$$||x - \hat{x}|| \leq \left|\left|A^{-1}\right|\right| \, ||\hat{r}||$$

Furthermore, since $||b|| \leq ||A|| \, ||x||$, it follows that

$$\frac{||x - \hat{x}||}{||x||} \leq \kappa(A)\frac{||\hat{r}||}{||b||}$$

where $\kappa(A) = ||A|| \left|\left|A^{-1}\right|\right|$. In words, the relative error in the solution is bounded by the condition number of the matrix A times the relative error in the residual! Thus, for an ill-conditioned matrix, even a good residual will not guarantee a small relative error.

Another approach to error analysis is **backward** error analysis. The computed solution $\hat{x}$ of $Ax = b$ can be viewed as the exact solution to a slightly perturbed problem $(A + \delta A)x = b + \delta b$. In this way, we see that

$$\hat{r} = b - A\hat{x} = (\delta A)\hat{x} - \delta b$$

Plugging this into the previous equation (and assuming that the perturbation is small, i.e. $||\delta A|| < 1/\left|\left|A^{-1}\right|\right|$) we see that

$$\frac{||x - \hat{x}||}{||x||} \leq \frac{\kappa(A)}{1 - \kappa(A)(||\delta A|| \, / \, ||A||} \left(\frac{||\delta b||}{||b||} + \frac{||\delta A||}{||A||}\right)$$

In summary, a stable algorithm is responsible for producing a small residual. This will yield an acceptably small error in the solution if the problem is well conditioned, i.e. has a small condition number.

Chapter 5

Least Squares Problem

Last section, algorithms for solving $Ax = b$ were introduced. Now consider the following scenario: we seek a solution to $Ax = b$, but in this case, the number of rows of A are greater than the number of columns, i.e. $A \in \mathbb{R}^{m\times n}$ and $m > n$. Here, the system is overdetermined. If $b \notin \text{col}(A)$, then there exists no feasible solution.

In many disciplines, a solution to $Ax \approx b$ would still be desirable if $Ax = b$ cannot be solved. There are many ways to describe a "good" approximation, but the way we will proceed with is rather than solving $Ax = b$, we may instead solve $x = \text{argmin}\, ||Ax - b||_2$. Noting that multiplication by a constant factor and squaring does not change the minimizer (only the minimum value), this is equivalent to

$$x^* = \underset{x}{\text{argmin}} \frac{1}{2} \, ||Ax - b||_2^2$$

which is what we will call the least squares problem.

1 An Analytical Solution

Let $\phi(x) = \frac{1}{2}\,||Ax - b||_2^2$. Then, our least squares problem is of the form $\min \phi(x)$, an optimization formulation. From optimization, we know that since ϕ is a convex quadratic function, necessary and sufficient conditions for x^* to be a minimizer are $\nabla\phi(x^*) = 0$ and $\nabla^2\phi(x^*) \succ 0$. It is easy to compute $\nabla\phi(x) = A^TAx - A^Tb$ and $\nabla^2\phi(x) = A^TA$. We may assume A to be full rank (if not, then the rows of A contain redundant information) and so $x^TA^TAx = ||Ax|| > 0$ for $x \neq 0$ and so A^TA is positive definite. Thus, the Hessian of ϕ is positive definite everywhere. Consequently, our sufficient and necessary condition for x^* to solve the least squares problem is

$$A^TAx = A^Tb$$

As it turns out, solving $Ax \approx b$ is equivalent to solving another linear system $A^TAx = A^Tb$. This small derivation is the backbone of many areas of data science and machine learning and also allows us to transfer any knowledge of solving linear systems to solving least squares problems.

2 Naive Numerical Solution

Hopefully by now, we know better than to simply compute $x = (A^TA)^{-1}A^Tb$ and call it a day. Yet, even more stable methods such as LU factorization with partial pivoting discussed previously may have trouble solving this system. To investigate, we must look at the conditioning of this problem.

Recall that the conditioning of solving a linear system is strongly correlated with the condition number of the matrix. We need to compute $\kappa_2(A^TA)$ to fully understand the difficulties. Let $A = U\Sigma V^T$ be a singular value decomposition of A. Then

$$\kappa_2(A^TA) = \kappa_2(V\Sigma^T\Sigma V^T) = \frac{\sigma_1^2}{\sigma_2^2} = \kappa_2(A)^2$$

We see here that the conditioning of solving this linear system is the squared condition number of solving a linear system of just A. In many scenarios, A is a matrix of data and cannot be assumed to be well-conditioned. If A is poorly conditioned, i.e. $\kappa_2(A) \approx \sqrt{\varepsilon_{\text{mach}}}$, then any solution computed via $x = (A^TA)^{-1}A^Tb$ will have no meaningful digits. We may rely on this approach for small systems with small condition numbers, but for larger matrices, we must find another approach.

3 Other Numerical Methods

The two suggestions here rely on QR and SVD decomposition methods. Properties and computation of these factorizations will be covered later, but we only rely on the basics here. For QR factorization, the solution via normal equations reduces to be

$$x = (A^TA)^{-1}A^Tb = (R^TQ^TQR)^{-1}R^TQ^Tb = (R^TR)^{-1}R^TQ^Tb = R^{-1}Q^Tb$$

Now suppose $R = U\Sigma V^T$ is an SVD of R. Then $A = QR = (QU)\Sigma V^T$ is a valid SVD for A. Thus, the singular values of R are the same as those of A and therefore $\kappa_2(R) = \kappa_2(A)$. Indeed, solving the least squares problem in this manner has condition number $\kappa_2(A)$ instead of $\kappa_2(A)^2$.

Alternatively, let $A = U \begin{bmatrix} \Sigma \\ 0 \end{bmatrix} V^T$ be a full SVD of A. Then by properties of orthogonal matrices

$$\begin{aligned} ||Ax - b|| &= \left|\left| U \begin{bmatrix} \Sigma \\ 0 \end{bmatrix} V^T x \right|\right| \\ &= \left|\left| \begin{bmatrix} \Sigma \\ 0 \end{bmatrix} V^T x - U^T b \right|\right| \end{aligned}$$

Letting $U^Tb = \begin{bmatrix} y \\ z \end{bmatrix}$, it follows that

$$||Ax - b|| = \left|\left| y - \Sigma V^T x \right|\right| + ||z||$$

Thus, in order to solve the least squares problem, it suffices to minimize $\left|\left| y - \Sigma V^T x \right|\right|$ instead. This yields system

$$\Sigma V^T x = y$$

to solve. Again, note that $I\Sigma V^T$ is a valid SVD for the matrix in question so it has the same singular values as A. Thus, $\kappa_2(A) = \kappa_2(\Sigma V^T)$. This is yet another more stable approach to solving the least squares problem. Note that these decompositions of A will take time to compute and are thus more costly operations wise but yield more stable numerical solutions.

Chapter 6

Polynomial interpolation

In this section we build up the foundation for the next few sections. Many topics in numerical analysis such as ODE's and integration rely on polynomial interpolation to work their magic. Here, we describe some basic techniques and a few results that will come in handy for future sections.

Function approximation can roughly be broken down into two categories: data fitting and approximating functions. They are different, but the distinction is subtle. **Data Fitting** is the process of finding a function that "fits" some data points. We use the term fit here loosely because there are different ways a function can fit a dataset. One such example is interpolation, i.e. the function passes through each point exactly. However, a function may fit in the least squares sense where the function is simpler, but may not interpolate exactly as is the case with linear regressions. **Approximating functions** are exactly that - functions that approximate other functions. It should be noted that the difference between these two is that the latter is identical to those of data fitting once we specify the data points.

For approximating functions, we generally assume a linear form

$$v(x) = \sum_{j=0}^{n} c_j \phi_j(x)$$

where $\{c_j\}_{j=0}^n$ are unkown coefficients and $\{\phi_j\}_{j=0}^n$ basis functions for the space we wish to approximate in. Note that $v(x)$ is linear wrt the basis functions and not x itself. Furthermore, we assume that the basis functions are linearly independent.

By default, we assume that the number of data points and the number of basis functions used are equal. If there are fewer basis functions than data, then the resulting system to solve for the coefficients c_j would be overdetermined, and the problem would be reduced to a least squares one previously covered.

Suppose we have data points $\{x_j, y_j\}_{j=0}^n$ and basis functions $\{\phi_j\}_{j=0}^n$. Then the coefficients can be found by solving the linear system

$$\begin{bmatrix} \phi_0(x_0) & \dots & \phi_n(x_0) \\ \vdots & \ddots & \vdots \\ \phi_0(x_n) & \dots & \phi_n(x_n) \end{bmatrix} \begin{bmatrix} c_0 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} y_0 \\ \vdots \\ y_n \end{bmatrix}$$

Again, we have reduced our current problem (polynomial interpolation) to one that we have already solved (linear systems). Thus, the same principles from previous chapters, such as conditioning and stability, are present here.

There is one additional point to be made about interpolation. Although we make the argument above that polynomial interpolation is immediately understood via solutions of linear systems, there is also the point of application. A polynomial interpolant isn't constructed to sit and be observed. It is generally evaluated. That is, there are two steps to polynomial interpolation: construction and evaluation. Most of construction is inherited from solutions of linear systems. The next few sections will analyze different choice of basis functions for construction with evaluation being considered on a "as necessary" basis.

1 Monomial interpolation

Perhaps the simplest basis for the $n+1$ degree polynomial space is the monomial basis, $\phi_j := x^j$. An interpolating polynomial p would then have to satisfy $p(x_i) = \sum_{j=0}^{n} c_j x^j$ for some c_j. Assuming (for now) that $x_i \neq x_j$ for $i \neq j$, the corresponding linear system for coefficients c_j is

$$\begin{bmatrix} 1 & \dots & x_0^n \\ \vdots & \ddots & \vdots \\ 1 & \dots & x_n^n \end{bmatrix} \begin{bmatrix} c_0 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} y_0 \\ \vdots \\ y_n \end{bmatrix}$$

The coefficient matrix X in question here is known as Vandermonde matrix. From linear algebra, we know that

$$\det X = \prod_{i=0}^{n-1} \prod_{j=i+1}^{n} x_j - x_i$$

that is, under our distinct data point assumption, the determinant is non-zero and the corresponding interpolant is unique as described in the following theorem. For any real data points $\{x_j, y_j\}_{j=0}^{n}$ with distinct abscissae x_i there exists a unique polynomial $p(x)$ of degree at most n which satisfies the interpolation conditions

$$p(x_i) = y_i, i = 0, 1, \dots, n$$

We can summarize the monomial interpolation technique in just a few points

1. The coefficients computed may completely change if we only slightly modify the interpolation problem (more on this later)
2. The data matrix X is often ill-conditioned as n grows large or as the data points themselves spread out.
3. The construction stage requires $\mathcal{O}(n^3)$ steps, but the evaluation can be done as quickly as $\mathcal{O}(n)$ with only roughly $2n$ flops per point.

It is important to point out that the latter two disadvantages are not prevalent in small datasets. Monomial interpolation is a perfectly acceptable method for such cases.

2 Lagrange Interpolation

Monomial interpolation is quite straight-forward. It would be the first approach of any naive attempt. In fact, one of its major upsides is that it is easy to understand. Lagrange interpolation, in contrast, is not so. Nonetheless, let us proceed as intuitively as possible.

The main computational drawback of monomial interpolation is solving the linear system. What if we found a polynomial basis such that $c_j = y_j$? Then, such a representation would be easy to manipulate and significantly reduce the construction costs. These polynomial bases are the Lagrange polynomials $L_j(x)$ which satisfy

$$L_j(x_i) = \begin{cases} 0 & i \neq j \\ 1 & i = j \end{cases}$$

Thus, by letting

$$p(x) = \sum_{j=0}^{n} y_j L_j(x)$$

we have formed our polynomial interpolant. Indeed, it satisfies the interpolation condition because

$$p(x_i) = \sum_{j=0}^{n} y_j L_j(x_i) = y_i$$

for any i.

With the construction stage a mere formality, what left is there to do? This is where evaluation becomes important. Let us look at the Lagrange polynomials.

$$L_j(x) = \prod_{\substack{i=0 \\ i\neq j}}^{n} \frac{(x-x_i)}{(x_j-x_i)}$$

Indeed, the n roots of the polynomial must be x_i for $i \neq j$. To ensure $L_j(x_j) = 1$, we must also divide out by $x_j - x_i$ for every $i \neq j$. Thus, giving us the representation above. With all of this in hand, we are ready to do evaluation.

Notice that we can construct the denominators of the $n+1$ Lagrange polynomials without the use of an evaluation point x. Let us compute

$$\rho_j = \prod_{i\neq j} (x_j - x_i), w_j = \frac{1}{\rho_j}$$

This requires roughly n^2 flops. We call the w_j **barycentric weights**. For evaluation, we may define the function

$$\psi(x) = \prod_{i=0}^{n} x - x_i$$

to obtain the interpolant

$$p(x) = \psi(x) \sum_{j=0}^{n} \frac{w_j y_j}{x - x_j}$$

For any given argument x, the above takes roughly $5n$ flops.

We can simplify this slightly further. Note that the function $f(x) \equiv 1$, it must be that y_j for all j. Since f is a degree 0 polynomial, it must be the degree n interpolating polynomial for any n. Thus,

$$1 = \psi(x) \sum_{j=0}^{n} \frac{w_j \cdot 1}{x - x_j}$$

That is, ψ can be computed using quantities that are already used. This brings us to the final representation used for evaluation

$$p(x) = \frac{\sum_{j=0}^{n} \frac{w_j y_j}{x - x_j}}{\sum_{j=0}^{n} \frac{w_j}{x - x_j}}$$

for any x.

3 Newton Polynomial Interpretation

The previous two basis functions fail to be flexible with respect to a growing dataset and also do not make it very easy to compute error in the interpolant. The Newton polynomial basis approach, however, does. We can view the Newton polynomial basis as a compromise of monomial and Lagrange: set

$$\phi_j(x) = \prod_{i=0}^{j-1} x - x_i$$

for $j = 0, 1, \ldots, n$. Here we see that by construction of the basis functions, an interpolant constructed in this way is adaptive. That is, to compute the kth coefficient, only the first k data points are required. This allows us the flexibility of not having all data at once as is often the case with lab experiments.

Furthermore, the coefficients are themselves solutions of a lower triangular system. One could theoretically form such a system and solve for the coefficients, or use the following: let

$$\begin{aligned} f[x_i] &= f(x_i) \\ f[x_i, \ldots, x_j] &= \frac{f[x_{i+1}, \ldots, x_j] - f[x_i, \ldots, x_{j-1}]}{x_j - x_i} \end{aligned}$$

and then set $f[x_0, \dots, f_j] = c_j$. The c_j's are known as divided differences. Note that the textbook these notes are transcribed from (and the corresponding lecturer notes) show how one can form a divided difference table recursively, but is no more than fancy bookkeeping and will not be shown here.

4 Error in Polynomial Interpolation

Let us briefly discuss possible error in the approximations. We define an error function of an interpolant $p_n(x)$ as

$$e_n(x) = f(x) - p_n(x)$$

Newton's approach allows us to cleverly compute such an error. The error at a new point x is simply the difference between the polynomial interpolant already computed evaluated at x and the polynomial interpolant that includes x. Mathematically, that is

$$f(x) = p_{n+1}(x) = p_n(x) + f[x_0, \dots, x_n, x]\phi_n(x)$$

and so

$$e_n(x) = f(x) - p_n(x) = f[x_0, \dots, x_n, x]\phi_n(x)$$

Furthermore, because the interpolating polynomial is unique, this is precisely the error no matter which basis is used. Unfortunately, this depends on the data and the evaluation point x. We continue our search for a more general error bound.

Assuming f is smooth enough, we may replace the divided differences by their corresponding derivatives to yield

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

for some ξ. The only remaining unknowns are $\phi_n(x)$ and $f^{(n+1)}(\xi)$. By upper bounding these, we obtain the following result. If p_n interpolates f at the $n+1$ points $x_0, \dots, x_n$ and f has $n+1$ bounded derivatives on an interval $[a, b]$ containing these points, then for each $x \in [a, b]$ there exists a point $\xi = \xi(x) \in [a, b]$ such that

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

with error bound

$$\max_{a \le x \le b} |f(x) - p_n(x)| \le \frac{1}{(n+1)!} \max_{a \le t \le b} \left| f^{(n+1)}(t) \right| \max_{a \le s \le b} \prod_{i=0}^{n} |s - x_i| \tag{6.1}$$ eq:polyErrBound

This error bound is so important that it is one of the few equations in this monograph that will earn an equation number.

Minimizing Error

Suppose that we are tasked with approximating a function, but are allowed to choose our interpolating points ourselves. Notice that from the error bound computed previously, this is our only hope at minimizing the error anyways. It suffices to choose points such that the product of distances between points is minimized. Such a choice is provided by the **Chebyshev points**. These points are defined on the interval $[-1, 1]$ by

$$x_i = \cos\left(\frac{2i+1}{2(n+1)}\pi\right)$$

for $i = 0, \dots, n$. For a general interval $[a, b]$, apply the transformation

$$x_i \leftarrow a + \frac{b-a}{2}(x_i + 1)$$

For points in the $[-1, 1]$ interval, the interpolant using the Chebyshev points satisfies[1]

$$\max_{-1 \le x \le 1} |f(x) - p_n(x)| \le \frac{1}{2^n (n+1)!} \max_{-1 \le t \le 1} \left| f^{(n+1)(t)} \right|$$

[1]see appendix for Chebyshev polynomial discussion

Chapter 7

Piecewise Polynomial Interpolation

The contrast between the error of an ordinary polynomial interpolant and that of one with Chebyshev points should indicate that polynomial interpolation can be quite bad if we are not allowed to choose our points cleverly. In such a case, equation (6.1) shows us that by decreasing the size of the interval, we can also reduce the error[1]. Piecewise polynomial interpolation accomplishes this by reducing the overarching problem into many smaller ones. This decomposition will also allow our interpolant to be globally flexible. That is, if one were to change a data realization y_i, only the subinterval containing the corresponding x_i will be affected. This is not the case for polynomial interpolation covered in the previous section.

That is, we may divide an interval $[a, b]$ into a smaller number of subintervals by the partition

$$a = t_0 < t_1 < \cdots < t_r = b$$

and use a low degree polynomial interpolant for each subinterval. Call these interpolants $s_i(x)$. These are then patched together to form a continuous global interpolating curve which satisfies

$$v(x) = s_i(x)$$

for any $t_i \leq x \leq t_{i+1}, i = 0, \ldots, r-1$.

1 Broken line and piecewise Hermite interpolation

The simplest piecewise interpolation is piecwise linear, "broken line" interpolation. In this approach, the $s_i' s$ are simply linear functions. We can even compute the global error on the interval $[a, b]$ using equation (6.1). Let $v(x)$ represent the global interpolant. For any argument $t_i \leq x \leq t_{i+1}$, the error is obtained by $f(x) - v(x) = f(x) - s_i(x)$. For a linear interpolant, this error from (6.1) is

$$f(x) - s_i(x) = \frac{f''(\xi)}{2!}(x - t_i)(x - t_{i+1})$$

The maximum value of the RHS is achieved at $x = \frac{t_i + t_{i+1}}{2}$. Letting $h = \max\limits_{1 \leq i \leq r} t_{i+1} - t_i$, it follows that

$$|f(x) - v(x)| \leq \frac{h^2}{8} \max_{a \leq \xi \leq b} |f''(\xi)|$$

Unfortunately, only enforcing continuity forces us to give up the hopes of differentiability. In some cases, this is not so desirable. Let us look at techniques that ensure differentiability.

2 Piecewise cubic interpolation

The reason we cannot guarantee differentiability is that a linear function only has two degrees of freedom. With n interpolants, that is only $2n$ degrees of freedom. The interpolating conditions are $s_i(x_i) = f(x_i)$

[1]Note that a simple rescaling of the x-axis will not work. Convince yourself why

and $s_i(x_i + 1) = f(x_i + 1)$ for each i which is a total of $2n$ conditions. Thus, there is no room for differentiability. The natural way to alleviate this is to move to a higher degree polynomial. By far the most common choice outside of linear is cubic. By writing $s_i(x) = a_i + b_i(x - t_i) + c_i(x - t_i)^2 + d_i(x - t_i)^3$, there are a total of $4n$ degrees of freedom to enforce constraints for. The next two sections will cover ways to choose these conditions.

Piecewise cubic Hermite interpolation

Recall that the continuity conditions take up $2n$ of the $4n$ degrees of freedom available. If the values $f'(t_i)$ are provided, we can further ensure that the derivative of our interpolant $v'(x)$ is also continuous in an analogous way. This provides $2n$ more conditions which totals to our degrees of freedom. These conditions written explicitly are of course

$$s_i'(t_i) = f'(t_i) \text{ and } s_i'(t_i + 1) = f'(t_{i+1})$$

This interpolation technique is known as Hermite cubic interpolation. One very handy property of such an interpolant is that each piece can be computed independently. That is, the approximation is completely local. Furthermore, one should expect that using more points does indeed provide us with a better error estimate. Indeed, it would do the reader well to prove the following theorem. Let v interpolate f at the $n + 1$ points $x_0 < \cdots < x_n$ and define $h = \max_{1 \le i \le n} x_i - x_{i-1}$ and assume f has as many bounded derivatives as necessary for the bounds below on an interval $[a, b]$ containing these points. Then, using a local constant, linaer, or Hermite cubic interpolation, for each $x \in [a, b]$ the interpolation error is bounded by

$$\begin{aligned}
|f(x) - v(x)| &\le \frac{h}{2} \max_{a \le \xi \le b} |f'(\xi)| && \text{piecewise constant} \\
|f(x) - v(x)| &\le \frac{h^2}{8} \max_{a \le \xi \le b} |f''(\xi)| && \text{piecewise linear} \\
|f(x) - v(x)| &\le \frac{h^4}{384} \max_{a \le \xi \le b} |f''''(\xi)| && \text{piecewise cubic Hermite}
\end{aligned}$$

3 Cubic Spline Interpolation

Of course, the main disadvantage with Hermite cubics is the need for derivative values. As with everything in numerical analysis, it is best to have a methodology for when these are not available. Enter cubic splines. Our setting is again the same: approximate each interval with an interpolating cubic. However, with cubic splines, the remaining conditions are used to ensure that $v \in C^3$. That is, the conditions to be satisfied by the cubic spline are

$$\begin{aligned}
&s_i(x_i) = f(x_i) && i = 0, \ldots, n-1 \\
&s_i(x_{i+1} = f(x_{i+1}) && i = 0, \ldots, n-1 \\
&s_i'(x_{i+1}) = s_{i+1}'(x_{i+1}) && i = 0, \ldots, n-2 \\
&s_i''(x_{i+1}) = s_{i+1}''(x_{i+1}) && i = 0, \ldots, n-2
\end{aligned}$$

Note that the latter two equations only provide $2n - 2$ conditions because they only apply to the interior intersections. Thus there are even more variants that handle these remaining conditions differently.

1. The free boundary approach, giving a **natural spline**:

$$v''(x_0) = v''(x_n) = 0$$

   Although popular, there is no reason to believe that this additional constraint improves the interpolation. It is widely used for its simplicity.

2. If f' is available on interval ends (such as with some boundary value problems) then the clamped boundary may be considered, specified by

$$v'(x_0) = f'(x_0), v'(x_n) = f'(x_n)$$

   The interpolant here is known as the **complete spline**.

3. The third alternative is called **not-a-knot**. This approach ensures continuity of the third derivative the spline interpolant at the the nearest interior break points x_1 and x_{n-1}. That is,

$$s_0'''(x_1) = s_1'''(x_1) \text{ and } s_{n-2}'''(x_{n-1}) = s_{n-1}'''(x_{n-1})$$

Constructing the cubic spline

The trade-off of ensuring differentiability is that now each interpolant depends on those immediately next to it. The effect diminishes significantly the further out we go, but one serious downside is that the construction is not so simple anymore. Unlike before with Hermite cubics, the coefficients in the splines must be computed simultaneously. The derivation for the following algorithm is not given, but one is encouraged to try and explain it on your own. To obtain coefficients $\{a_i, b_i, c_i, d_i\}_{i=0}^{n-1}$, simply

1. Set $a_i = y_i = f(x_i)$.

2. Construct and solve a tridiagonal system of equations for the unknowns $c_0, \dots, c_n$ using the two boundary conditions of your choosing and the equations

$$f[x_{i-1}, x_i] - h_{i-1}(2c_{i-1} + c_i) + 2h_{i-1}c_{i-1} + h_{i-1}(c_i - c_{i-1}) = f[x_i, x_{i+1}] - \frac{h_i}{3}(2c_i + c_{i+1})$$

3. Set the coefficients d_i for $i = 0, \dots, n-1$ by

$$d_i = \frac{c_{i+1} - c_i}{3h_i}$$

4. Set the coefficients b_i for $i = 0, \dots, n-1$ by

$$b_i = f[x_i, x_{i+1}] - \frac{h_i}{3}(2c_i + c_{i+1})$$

The corresponding interpolant for $x_i \le x \le x_{i+1}$ is

$$s_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3$$

Chapter 8

Numerical Differentiation

In my opinion, this is perhaps the easiest and most straight-forward section in this document. Here, we discuss the topic of numerical differentiation. Although a very routine and often easy task for early level college students, differentiation is not so trivial numerically. As a powerful tool in ODE's (a later chapter), numerical differentiation is a topic that is essential for any scientific computing text. However, most of the ingredient have already been discussed. For this reason, this section is rather short, but do not mistake this for irrelevance.

1 Taylor Series Approximations

Let f be a sufficiently smooth function with x_0 existing within its domain. We seek a cheap, accurate approximation to $f'(x_0)$, preferably through functional evaluations. As with many things in computational courses, a solid first approach is to consider a Taylor series expansion. Indeed, note that

$$f(x_0+h) = f(x_0) + hf'(x_0) + \frac{h^2}{2}f''(\xi)$$

for some $\xi \in (x_0, x_0+h)$. Solving for $f'(x_0)$ gives

$$f'(x_0) = \frac{f(x_0+h) - f(x_0)}{h} + \frac{h}{2}f''(\xi) \approx \frac{f(x_0+h) - f(x_0)}{h}$$

which will refer to as a one-sided, two point formula. We say one-sided since the function evaluations are only points larger (or smaller) than x_0 and two point because there are two function evaluations involved. This particular approximation is commonly known as the forward difference formula for $f'(x_0)$ and has truncation error $\mathcal{O}(h)$.

There is nothing particular about the Taylor series expansion chosen here, barring the fact that $f'(x_0)$ lies readily available. Indeed, we may arrive at a different approximation in a similar way using a two point, centered formula. Let

$$\begin{aligned} f(x_0+h) &= f(x_0) + hf'(x_0) + \frac{h^2}{2}f''(x_0) + \frac{h^3}{6}f'''(\xi_1) \\ f(x_0-h) &= f(x_0) - hf'(x_0) + \frac{h^2}{2}f''(x_0) - \frac{h^3}{6}f'''(\xi_2) \end{aligned}$$

be two Taylor expansions. Then subtracting the second from the first and solving for $f'(x_0)$, we obtain

$$f'(x_0) = \frac{f(x_0+h) - f(x_0-h)}{2h} - \frac{h^2}{6}f'''(\xi)$$

Here, we used the intermediate value theorem in (**??**) to arrive at our error term. Note that we only used two evaluations of f, but still obtained a second order accurate approximation. A downside to this is that the method is two-sided and the points are more spread apart than those in the forward difference formula.

As long as our tolerance for long, redundant Taylor series expansions does not falter, we may continue to compute both higher order approximations as well as approximations of higher orders. However, we will not spend too much time here, only noting the possible approach.

2 Richardson Extrapolation

Rather than deal with complicated Taylor series expansions, which will unfortunately come back to us, let us take a look at another approach for generating higher order approximations. Suppose we have two order q approximation formulas for $f'(x_0)$, $g_1(x_0, h)$ and $g_2(x_0, h)$ with errors $e_1 h^q$ and $e_2 h^q$. Then for constants c_1, c_2 such that $c_1 e_1 + c_2 e_2 = 0$, the approximation

$$f'(x_0) = \frac{c_1 g_1(x_0, h) + c_2 g_2(x_0, h)}{2}$$

has error of order $\mathcal{O}(h^{q+1})$. This technique is known as Richardson extrapolation.

If the Taylor series approach was not compelling enough to convince you that the generation of highly accurate approximation formulas is quite simple, then Richardson's extrapolation should leave you satisfied. As with anything else in this text though, it does not come without a cost. Such formulas proposed here require "sufficiently smooth" functions. For high order formulas, we required many nicely bounded derivatives of f, which may not always be possible. Furthermore, approximations through Richardson extrapolation is the formulas are not compact, and often use points from a wider area than necessary.

3 Lagrange Polynomial Approximations

It would be a shame if we did all this polynomial interpolation work previously and never put it to use. Fear not, Lagrange interpolation provides us with remarkably easy to produce differentiation formulas - so much so that you may wonder why we even bothered with the first two sections. Recall that the degree n interpolating polynomial through point $x_0, \ldots, x_n$ is given by

$$p_n(x) = \sum_{i=0}^{n} f(x_i) L_i(x)$$

with error term $e_n(x) = \prod_{i=0}^{n} (x - x_i) f[x_0, \ldots, x_n]$. Thus, by differentiating, we get

$$f'(x) = \sum_{i=0}^{n} f(x_i) L_i'(x)$$

with error term $e_n'(x)$. Evaluating at x_0 gives the almost-too-clean-to-be-comp expression

$$f'(x_0) = \sum_{i=0}^{n} f(x_i) L_i'(x_0)$$

with error $e_n'(x_0)$. As you may have guessed it, neither the error term nor the derivatives of the Lagrange polynomials are particularly easy to evaluate in the sense that I certainly don't want to include their expressions. However, rest assured that it can be done. Last but not least, under an equidistant point assumption, the error is $\mathcal{O}(h^n)$.

Chapter 9

Numerical Integration

Just as the previous chapter, numerical integration is also crucial in the development of numerical ODE's discussed next. Furthermore, much of this chapter is motivated by simple polynomial interpolation covered previously. However, unlike the previous section, analysis in numerical integration is not as easily derived. This comes from the simple fact that it is in general much easier to differentiate symbolically than it is to integrate. Consequently, motivation will be key here while direct computation will be left as exercises.

1 Basic Quadrature Rules

Continuing with the assumptions of smoothness asserted previously, our problem is now to find a numerical solutions to $\int_a^b f(x)dx$. Similar to before, note that $\int_a^b p_n(x)dx$ approximates $\int_a^b f(x)dx$. Using our Lagrange interpolation form, we have that

$$\int_a^b f(x)dx \approx \int_a^b p_n(x)dx = \int_a^b \sum_{i=0}^n f(x_i)L_i(x)dx = \sum_{i=0}^n f(x_i)\int_a^b L_i(x)dx$$

A numerical integral of this form is called a quadrature with $a_i = \int_a^b L_i(x)dx$ and x_i called the quadrature weights and nodes, respectively. Here, we may choose our x_i freely and use our Lagrange interpolation polynomials to compute the quadrature weights. It is important to note here that once an a and b are fixed, the quadrature weights are computed once, and only once. That is, they do not depend on the integrand itself. Let us show a few examples.

Example 9.1 (Trapezoidal Rule)**.** Set $n = 1$ and interpolate at the ends $x_0 = a$ and $x_1 = b$. Then

$$L_0(x) = \frac{x-b}{a-b}, L_1(x) = \frac{x-a}{b-a}$$

with weights

$$\int_a^b \frac{x-b}{a-b}dx = \frac{b-a}{2}$$
$$\int_a^b \frac{x-a}{b-a}dx = \frac{b-a}{2}$$

The resulting interpolant is known as the trapezoidal rule and has the following form

$$I_f \approx Q_f = \frac{b-a}{2}\left[f(a) + f(b)\right]$$

Example 9.2 (Simpson Rule)**.** Letting $n = 2, x_0 = a, x_1 = \frac{a+b}{2}, x_2 = b$ gives us the Simpson rule given by

$$I_f \approx Q_f = \frac{b-a}{6}\left[f(a) + 4f\left(\frac{b+a}{2}\right) + f(b)\right]$$

A quadrature rule based on polynomial interpolation at equidistant abscissae are referred to as Newton-Cotes forms. These two examples have the special property that $f(a)$ and $f(b)$ are explicitly used in the computation. Quadrature rules satisfying this are called closed formulas. An example of an open formula is the midpoint rule given by

$$I_f \approx Q(f) = (b-a)f\left(\frac{a+b}{2}\right)$$

2 Quadrature Error

The error discussion of quadrature rules is where this section differs most drastically from the previous. Recall that the error induced by quadrature rules based on polynomial interpolation takes the form

$$E(f) = \int_a^b f(x)dx - \int_a^b p_n(x)dx = \int_a^b f(x) - p_n(x)dx = \int_a^b f[x_0, x_1, \ldots, x_n, x]\prod_{i=0}^{n}(x - x_0)dx$$

In the past, there were simple cases where $E(f)$ could be computed at least up to a constant. Unfortunately, the best we can muster here is on a case-by-case basis. The interested reader should attempt to replicate the results below (see exercises for example). Nonetheless, we present the error bounds without proof

$$\begin{aligned} E_{\text{trap}}(f) &= \frac{f''(\eta)}{12}(b-a)^3 \\ E_{\text{simp}}(f) &= \frac{f''''(\zeta)}{90}(\frac{b-a}{2})^5 \\ E_{\text{mid}}(f) &= \frac{f''(\eta)}{24}(b-a)^3 \end{aligned}$$

In addition to error, another quantity used to measure quadrature rules is the precision, or degree of accuracy. If a quadrature formula satisfies $E(f) = 0$ for all polynomials of degree p or less, then we say that the quadrature $Q(f)$ has precision p. For example, trapezoidal and midpoint have precision 1, and the Simpson rule has precision 3 [1]. If the success of the midpoint rule catches your attention, then you have a healthy amount of skepticism. Indeed, the midpoint rule only uses one function compared to the trapezoidal's two, but maintains the same precision and only slightly higher error (a constant of 2). The trapezoidal rule makes its presence known in composite numerical integration.

3 Composite Numerical Integration

In order to decrease the error of a quadrature, we have two realistic options. One is simply use a quadrature formula with a higher ordered error term. This approach tends to suffer greatly because in order to achieve such, more evaluation points are needed. Should this not bother you, it is suggested that you go and read the chapter on polynomial interpolation again. The only remaining option is to decrease the interval size. That is, apply the same technique we did with polynomial interpolation. However, here, there is no need to enforce smoothness conditions and as such the composite numerical integration formulas are a seamless transition from their single interval counterparts.

Thus, in its simplest form, divide the interval $[a, b]$ into r equal subintervals of length $h = \frac{b-a}{r}$ each. Then by addititivity of integration,

$$\int_a^b f(x)dx = \sum_{i=1}^{r}\int_{a+(i-1)h}^{a+ih} f(x)dx = \sum_{i=1}^{r}\int_{t_{i-1}}^{t_i} f(x)dx$$

where $t_i = a + ih$. Consequently, the error is simply the error committed by each individual subinterval. Although what follows may have a different form than before, make no mistake, it is simply the quadrature

[1]In fact, it is possible to derive all these rules independently by simply enforcing these degrees of accuracy along with the abscissae

applied to each individual subinterval. We proceed nevertheless. The composite trapezoid rule yields

$$\begin{aligned}\int_a^b f(x)dx &\approx \frac{h}{2}\sum_{i=1}^{r}[f(t_{i-1}+f(t_i)] \\ &= \frac{h}{2}\left[f(a)+f(b)+2\sum_{i=1}^{r-1}f(t_i)\right]\end{aligned}$$

with error

$$E(f)=\sum_{i=1}^{r}\left(-\frac{f''(\eta_i)}{12}h^3\right)=-\frac{f''(\eta)}{12}(b-a)h^2$$

Before deriving the composite Simpson and midpoint rules, note the beauty of closed form quadrature rules here. The function evaluations on the beginning and end of each subinterval are repeated. That is, although the trapezoidal rule requires 2 function evaluations per interval, the composite version only requires $r+1$ evaluations for r subintervals. An open point formula, such as midpoint, will not be able to reuse evaluations and is less efficient in the generalization. Indeed, we continue with both the composite Simpson and midpoint rules, respectively

$$\int_a^b f(x)dx \approx \frac{h}{3}\left[f(a)+f(b)+2\sum_{k=1}^{r/2-1}f(t_{2k})+4\sum_{k=1}^{r/2}f(t_{2k-1})\right]$$

$$\int_a^b f(x)dx \approx h\sum_{i=1}^{r}f(a+(i-1/2)h)$$

You are free to compute their errors.

4 Gaussian Quadratures

Up until this point we have been considering Newton-Cotes quadrature formulas that are based on equidistant evaluation points. But we have (hopefully) learned that such a choice of abscissae can perform quite poorly. Indeed, recall that the error of a quadrature based on polynomial interpolation is

$$E(f)=\int_a^b f[x_0,x_1,\ldots,x_n,x]\prod_{i=0}^{n}(x-x_i)dx$$

Suppose that f is a polynomial of degree $m\le n$. Then

$$f[x_0,x_1,\ldots,x_n,x]=\frac{f^{(n+1)}(\xi)}{(n+1)!}=0$$

That is, for any $n+1$, we will generate a formula of precision n. We would optimistically hope that by choosing the $n+1$ parameters carefully, we can obtain $2n+1$ precision. To attempt this, recall that the Legendre polynomials $\{\phi_i(x)\}_{i=0}^{n+1}$ form an orthogonal basis for polynomials of degree at most n in the sense that

$$\int_a^b \phi_i(x)\phi_j(x)dx=0, i\neq j$$

and

$$\int_a^b g(x)\phi_{n+1}(x)dx=0$$

for any polynomial g of degree less than or equal to n. By setting x_i to be the i^{th} root of the Legendre polynomial $\phi_{n+1}(x)$, we have that

$$E(f)=\int_a^b Kf[x_0,x_1,\ldots,x_n,x]\phi_{n+1}(x)dx$$

for an appropriate constant K. Since for a function f of degree m, $f[x_0, x_1, \ldots, x_n, x]$ is of degree $m-n-1$, it follows that

$$E(f) = \int_a^b Kf[x_0, x_1, \ldots, x_n, x]\phi_{n+1}(x) = 0$$

for any polynomial f that satisfies $m - n - 1 \leq n \Leftrightarrow m \leq 2n+1$. That is, this quadrature rule has precision $2n+1$, the most we can logically expect[2]. This is known as the Gaussian quadrature since the roots of the Legendre polynomials are called Gauss points. Note that such a procedure requires us to compute both the weights and the nodes, but, just as with the weights, the nodes only need to computed for a single interval $[a, b]$ and degree n - not for a particular integrand.

Furthermore, it is often the case that we wish to approximate $\int_a^b f(x)w(x)dx$ for some weight function w. Rather than directly apply the above techniques, suppose we seek a solution of the form $\sum_{i=0}^n a_i f(x_i)$, i.e. the evaluations are only on f. We may follow the same procedure as the Gauss quadrature above, except the nodes must be the roots of an orthogonal polynomial basis with respect to $w(x)$. That is, if $\{\alpha_i\}_{i=0}^{n+1}$ is a basis for polynomials of degree n or less and

$$\int_a^b \alpha_i(x)\alpha_j(x)w(x)dx = 0, i \neq j$$

then letting the nodes of our quadrature formula be the roots of α_{n+1} will produce a rule with precision $2n+1$ for approximating $\int_a^b f(x)w(x)dx$. We may use Gram-Schmidt to compute such an orthogonal basis.

5 Adaptive Quadrature

We can certainly conjure up examples where a function is wild over a subinterval $[c, d]$, but moderately mild everywhere else in $[a, b]$. In such a case, we would be required to use many subintervals in composite quadrature techniques to ensure a small error over the wild interval. But this would be significant overkill for the rest of the interval $[a, b]$. A natural solution to this is to simply only use as many subintervals as necessary to achieve a particular error goal. In order to achieve such an optimistic task, we need to know when a particular quadrature estimate is good or bad. We can do this by approximating its error.

Consider a rule with error written as $E(f) = E(f; h) = Kh^{q-1} + \mathcal{O}(h^q)$ for some constant K. Let us compute two approximations R_1 and R_2 using the same quadrature rule with h and $h/2$. The first error $I_f - R_1$ is approximately Kh^q while the second $I_f - R_2$ is $K(h/2)^q = Kh^q/2^q$. Then we may compute the errors in terms of R_1 and R_2 as

$$\begin{aligned} I_f - R_1 &\approx \frac{2^q}{2^q - 1}(R_2 - R_1) \\ I_f - R_2 &\approx \frac{1}{2^q - 1}(R_2 - R_1) \end{aligned}$$

Since R_2 is the better approximation, if $I_f - R_2 < \varepsilon$ for some specified tolerance ε, then our approximation is good. If not, then we can simply split our bad interval in 2 and recursively perform the same quadrature rule over the subintervals until they satisfy the tolerance. This allows us to use many subintervals where necessary, and few when not. There are a few subtleties here that a robust program will address[3], but hopefully the idea is clear enough to understand.

6 Romberg Integration

Lastly, we present a procedure known as Romberg integration. It can be thought of as the integration version of Richardson extrapolation both in results and derivation. It can be shown (and perhaps one should) that the error for the trapezoidal rule[4] can be written as

$$E(f; h) = K_1 h^2 + \cdots + K_s h^{2s} + \mathcal{O}(h^{2s+1})$$

[2]It is important to point out that these nodes are chosen to maximize the precision of a rule and not necessarily the error, should it have any.

[3]such as the error being only an approximation, making sure to reuse function evaluations, etc

[4]Consider tracking our optimism with the trapezoidal rule throughout this chapter. It is an excellent example of the difference between mathematically feasible and computationally feasible approaches.

$\mathcal{O}(h^2)$	$\mathcal{O}(h^4)$	$\mathcal{O}(h^6)$	$\ldots$	$\mathcal{O}(h^{2s})$
$R_{1,1}$				
$R_{2,1}$	$R_{2,2}$			
$R_{3,1}$	$R_{3,2}$	$R_{3,3}$		
$\vdots$	$\vdots$	$\vdots$	$\ddots$	
$R_{s,1}$	$R_{s,2}$	$R_{s,3}$	$\ldots$	$R_{s,s}$

Figure 9.1: Romberg Table of Integration

⟨table:romberg⟩

for some constants K_i. As with Richardson extrapolation, we can also extrapolate the trapezoidal rule for higher order approximations. Consider approximations $R_{1,1}$ and $R_{2,1}$ based on $h_1 = h = b - a$ and $h_2 = \frac{h}{2}$. The errors are

$$\begin{aligned} E_{1,1} &= K_1 h^2 + K_2 h^4 + \ldots \\ E_{2,1} &= K_1 (h/2)^2 + K_2 (h/2)^4 + \ldots \end{aligned}$$

We can see that $E_{1,1} - 4E_{2,1} = \mathcal{O}(h^4)$. Coming back to our quadratures, it follows that

$$E_{1,1} - 4E_{2,1} = (4R_{2,1} - R_{1,1}) - 3I_f$$

Therefore,

$$I_f = \frac{4R_{2,1} - R_{1,1}}{3} + \mathcal{O}(h^4)$$

That is, $R_{2,2} = R_{2,1} + \frac{R_{2,1} - R_{1,1}}{3}$ is a $\mathcal{O}(h^4)$ accurate approximation to I_f. Not unlike Richardson extrapolation, we can continue this process to theoretically achieve an arbitrary small error. In fact, such a process is done cleanly using a Romberg table shown in Table 9.1. Note that the table can be generated each row at a time. That is, we may generated approximations in an adaptive manner. Much of the same problems persist, however. For very small h, roundoff errors start to dominate. Also, for rough functions, the lower order terms may not necessarily dominate as proposed since the constants themselves depend on high order derivatives of f. Thus, Romberg integration is most reliably applied to smooth functions with a need for high degree accuracy.

Chapter 10

Differential Equations

The materials presented in this chapter have an acquired taste. On one hand, all of the build in the previous chapters takes form in differential equations making it a particularly strong section to serve as a general review. On the other hand, much of the analysis becomes less elegant and more "left to the reader". Furthermore, there is quite a bit of content in just simple ordinary differential equations - so much so that in this chapter we often present a logical derivation or example of an idea and leave natural extensions of it to the problem sets to follow. Nonetheless, differential equations are without a doubt one of the most important tools an applied mathematician can have.

1 Euler Methods

1.1 Preliminaries

In this chapter we seek a solution to $y'(t) = f(t, y)$ for $a \leq t \leq b$. Here, we are looking for a particular function y, or rather $y(t)$ for all $t \in [a, b]$. We typically refer to the independent variable as t since a majority of differential equations are time based, but this need not be the case. Let us look at two examples.

Example 10.1. Consider the function $f(t, y) = t - y$ defined over $t \geq 0$. This gives the ODE

$$y'(t) = -y(t) + t$$

Like many numerical solutions, it is much easier to verify a solution than to find it. Indeed, please verify that this has solution $y(t) = t - 1 + \alpha e^{-t}$ for any scalar α. In order for the solution to be unique, we would require one additional piece of information to fix α. One such condition is $y(0) = c$ which enforces $\alpha = c + 1$. This type of specification is typically referred to as an initial value, or a trajectory.

For most of this chapter, we will only consider scalar ODE's simply because of their simplicity. However, it is entirely possible, and likely, that this is not the case in practice. For most systems of ODE's, they can be handled in a very similar manner as scalar ones.

Example 10.2. Consider a tiny ball of mass 1 attached at the end of a rigid, massless rod of length $r = 1$. At the other end the rod's position is fixed. Denote θ the angle between the pendulum and the negative vertical axis. A simplified model of the motion of the pendulum is governed by

$$\theta'' = -g \sin(\theta)$$

where $g = 9.81$. We can write this ODE as a first order system. Let

$$y_1(t) = \theta(t), y_2(t) = \theta'(t)$$

Then $y_1' = y_2$ and $y_2' = -g \sin(y_1)$. This defines

$$y = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}, f(t, y) = \begin{bmatrix} y_2 \\ -g \sin(y_1) \end{bmatrix}, c = \begin{bmatrix} \theta(0) \\ \theta'(0) \end{bmatrix}$$

We will use the simplest numerical method for approximately solving initial value ODE's, Euler's method, to introduce necessary terminology and core fundamentals. In later sections, where applicable, we will covered more advanced machinery as we build our theory. Consider searching for an approximate solution via equidistant abscissae defined as $t_0 = a, t_i = a + ih$ with $h = \frac{b-a}{N}$. Recall our forward difference formula for approximating a derivative

$$y'(t_i) = \frac{y(t_{i+1}) - y(t_i)}{h} - \frac{h}{2}y''(\xi)$$

From the differential equation, it follows that

$$y(t_{i+1}) = y(t_i) + hf(t_i, y(t_i)) + \frac{h^2}{2}y''(\xi_1)$$

Let us denote y_i an approximate solution to $y(t_i)$. It then makes sense to approximate y_{i+1} via

$$y_{i+1} = y_i + hf(t_i, y_i)$$

which is the celebrated forward Euler method. This simple method allows us to step through time and compute an approximation based on the previous one. Note that the initial value is necessary to do so. But who's to stay that we can't replicate such a method with the backward difference formula instead? Indeed, using

$$y'(t_{i+1}) = \frac{y(t_{i+1}) - y(t_i)}{h} - \frac{h}{2}y''(\xi_2)$$

we arrive at

$$y_{i+1} = y_i + hf(t_{i+1}, y_{i+1})$$

At first glance, one may not have much an issue with this method. However, it belongs to the class of implicit equations. That is, solving for y_{i+1}, the unknown value, is not so easy anymore. The ease of computing the forward Euler is due to it being an explicit formula, while backward Euler is an implicit method. For now, it suffices to say that the implicit method has added an unnecessary headache. We will further discuss implicit methods in sections to follow.

Example 10.3. This example will need to be showcased somewhere so lets take a look at it now. Consider the differential equation $y' = \lambda y, y(0) = 1$ for some scalar λ. It can be verified that the true solution is $y = e^{\lambda t}$. Applying forward Euler we obtain

$$\begin{aligned} y_{i+1} &= y_i + hf(t_i, y_i) \\ &= y_i + h\lambda y_i \\ &= (1 + h\lambda)^{i+1} y(0) \\ &= (1 + h\lambda)^{i+1} \end{aligned}$$

We will show later that as $h \to 0$, this method converges to the true solution under a few assumptions.

1.2 Method Errors

There are two crucial types of errors for an approximation in this chapter. The first is the local truncation error d_i which is the amount by which the exact solution fails to satisfy the difference equation. The order of accuracy q is the smallest positive integer such that the local truncation error is $\mathcal{O}(h^q)$. The second is the global error e_i, the amount an approximation differs from the true solution at a given point, defined by

$$e_i = y(t_i) - y_i$$

For example, by construction of the forward Euler method, we have

$$d_i = \frac{y(t_{i+1}) - y(t_i)}{h} - f(t_i, y(t_i)) = \frac{h}{2}y''(\xi)$$

Thus, it is first order accurate with $q = 1$.

We say that a method converges if the maximum global error tends to 0 as h tends to 0. Consider the forward Euler global error. By subtracting

$$\begin{aligned} d_i &= \frac{y(t_{i+1}) - y(t_i)}{h} - f(t_i, y(t_i)) \\ 0 &= \frac{y_{i+1} - y_i}{h} - f(t_i, y_i) \end{aligned}$$

we obtain

$$d_i = \frac{e_{i+1} - e_i}{h} - [f(t_i, y(t_i)) - f(t_i, y_i)]$$

If $y''(t)$ is bounded by some M over $[a, b]$ and $f(t, y)$ is L-Lipschitz in y, then

$$\begin{aligned} e_{i+1} &= e_i + h[f(t_i, y(t_i)) - f(t_i, y_i)] + hd_i \\ &\leq e_i + hLe_i + \frac{Mh}{2} \\ &\leq \cdots \leq (1 + hL)^{i+1} e_0 + \frac{Mh}{2} \sum_j^i (1 - hL)^j \\ &= \frac{Mh}{2} \sum_j^i (1 - hL)^j \end{aligned}$$

which tends to 0 as $h \to 0$. Thus, under some moderate conditions, the forward Euler method is convergent.

1.3 Stability

We must make one last remark regarding stability before moving to more advanced methods. Consider the test equation ODE introduced previously $y' = \lambda y$. When $\lambda < 0$ the solution $e^{-|\lambda| t}$ decays as $t \to \infty$. Thus, it is reasonable to expect for any approximate solution y_{i+1}, we should have

$$|y_{i+1}| \leq |y_i|$$

Recall that for forward Euler applied to this ODE, we have

$$y_{i+1} = (1 + h\lambda) y_i$$

Thus, in order for our seemingly harmless condition to hold, it must be that

$$h \leq \frac{2}{\lambda}$$

which is an awfully strict condition! Any ODE that requires more stringent conditions for stability than for accuracy is called strict.

One should know by now that if the entire story of forward vs backward Euler methods were told in the previous two sections, then we would have never bothered to introduce backward Euler in the first place. Applying backward Euler to the test equation, this imposed constraint simplifies to

$$\frac{1}{1 - h\lambda} \leq 1$$

which holds for any $h > 0, \lambda < 0$. Indeed, the implicit nature of the backward Euler allows for a much more flexible selection of h. Although the test equation may appear a bit arbitrary, this analysis is quite fundamental for determining the stability of a method. We will explore more on stability later.

2 Runge-Kutta Methods

With both Euler methods being only first order accurate, there is a definite need for higher order methods. One such family is the Runge-Kutta (RK) methods. In short, consider integrating the differential equation to obtain

$$y(t_{i+1}) = y(t_i) + \int_{t_i}^{t_{i+1}} f(t, y(t)) dt$$

RK methods are built around the idea of approximating the integral using some methods derived in previous the previous chapter. For example, the implicit trapezoidal method gives

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

Unfortunately, as an implicit method, this has some serious drawbacks (most notably being slow). We can "remedy" these issues by introducing the explicit trapezoidal method. It is obtained by, you guessed it, approximating y_{i+1} on the right hand side of the implicit trapezoid method by an explicit method, forward Euler. Thus

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

gives us an explicit (order 2) method. It should come as no surprise that the explicit trapezoid rule is only conditionally stable, however[1]. We will quickly present the remaining RK methods based on our favorite quadrature rules the midpoint and Simpson rules. For midpoint we have implicit midpoint given by

$$y_{i+1} = y_i + hf(t_{i+1/2}, y_{i+1/2})$$

where $t_{i+1/2} = 0.5(t_i + t_{i+1})$ and $y_{i+1/2} = 0.5(y_i + y_{i+1})$, and its explicit counterpart

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

The RK method based on the Simpson rule is the most popular of the RK methods and even named RK4. It is typically written as the explicit procedure

$$\begin{aligned}
Y_1 &= y_i \\
Y_2 &= y_i + \frac{h}{2}f(t_i, Y_1) \\
Y_3 &= y_i + \frac{h}{2}f(t_{i+1/2}, Y_2) \\
Y_4 &= y_i + \frac{h}{2}f(t_{i+1/2}, Y_3) \\
y_{i+1} &= y_i + \frac{h}{6}\left(f(t_i, Y_1) + 2f(t_{i+1/2}, Y_2) + 2f(t_{i+1/2}, Y_3) + f(t_{i+1}, Y_4)\right)
\end{aligned}$$

[1]There is no free lunch.

Part III

8610 Notes

This part is a culmination of notes from Xue's 8610 class and draw much comparison to Trefethen's Numerical Linear Algebra textbook. Similar to the 8600 part, the primary emphasis will be on motivation and intuition with detailed analysis left to homeworks and practice questions. The first two chapters were scribed while enrolled in the class and thus contain more detailed explanations/examples. However, do not draw the wrong conclusion from the brevity of the final three chapters. They are fundamental to numerical linear algebra and make up the bulk of the course.

Chapter 11

Conditioning and Stability

In this chapter, we turn to a systematic discussion of two fundamental issues of numerical analysis. **Conditioning** is the perturbation behavior of a mathematical problem. It is, loosely speaking, the sensitivity of the solution (output) to a mathematical problem with respect to the problem data (input). It is an intrinsic property of a problem. **Stability** pertains to the perturbation behavior of an algorithm used to solve that problem on a computer. Likewise, stability is the ability of an algorithm to produce a "reasonable" computed solution to a math problem under a small perturbation of input data. It is an intrinsic property of an algorithm.

1 Conditioning of a Problem

Consider the mapping $f : X \to Y$ where X, Y are normed vector spaces and $X = \{\text{all valid input data}\}, Y = \{\text{all possible solutions}\}$. Assume f is continuous and typically differentiable.

Given a problem data point x, let $y = f(x)$ be the corresponding solution. Now let the problem data change from x to $x + \Delta x$ and the solution becomes $f(x + \Delta x)$. The absolute condition number of the problem at x is

$$\kappa_a = \lim_{\delta \to 0} \sup_{||\Delta x|| \leq \delta} \frac{||f(x+\Delta x) - f(x)||}{||\Delta x||} = \sup_{\Delta x} \frac{||f(x+\Delta x) - f(x)||}{||\Delta x||}$$

Likewise, the relative condition number is defined as

$$\kappa_r = \lim_{\delta \to 0} \sup_{||\Delta x|| \leq \delta} \frac{||x||\,||f(x+\Delta x) - f(x)||}{||f(x)||\,||\Delta x||} = \sup_{\Delta x} \frac{||x||\,||f(x+\Delta x) - f(x)||}{||f(x)||\,||\Delta x||}$$

From now on, assume that both x and $f(x)$ are vectors and $\frac{\partial f_i}{\partial x_j}$ exist for all i, j. Then

$$f(x+\Delta x) - f(x) = J_f(x)\Delta x + \mathcal{O}(\Delta x)^2 \approx J_f(x)\Delta x$$

In this instance, the absolute condition number is

$$\kappa_a = \sup_{\Delta x} \frac{\left|\left|J_f(x)\Delta x + \mathcal{O}(\Delta x)^2\right|\right|}{||\Delta x||} = \sup_{\Delta x} \frac{||J_f(x)\Delta x||}{||\Delta x||} = ||J_f(x)||$$

Similarly, one can show that the corresponding relative condition number is

$$\kappa_r = \sup_{\Delta x} \frac{||J_f(x)||\,||x||}{||f(x)||}$$

Examples

Example 11.1. Let $f(x) = \alpha x$.

$$\kappa_r = \frac{|\alpha|\,|x|}{|\alpha x|} = 1$$

So this function is well conditioned by any standard.

Example 11.2. Let $f(x) = \sqrt{x}, n \in \mathbb{N}^+, x > 0)$.

$$\kappa_r = \frac{\left|\frac{1}{n}x^{1/n-1}\right| |x|}{\left|x^{1/n}\right|} = \frac{1}{n}, \text{ but } \kappa_a = \frac{1}{n}\left|x^{1/n-1}\right|$$

In this example, we see that the relative condition number is quite reasonable, but the absolute conditioning of the function is quite poor when $x << 1$.

Example 11.3. Let $f(x) = \tan(x)$ and $x = 10^{20}$. Then

$$\kappa_r = \frac{\sec^2(x)\,|x|}{|\tan(x)|} = \frac{1+\tan^2(x)}{|\tan(x)|}\,|x| \geq 2\,|x| = 2 \cdot 10^{20}$$

which is a terrible relative error! Conversely, $f(x) = \tan^{-1}(x)$ is perfectly conditioned.

Example 11.4. Let $f(x) = x_1 - x_2$. Then

$$\kappa_r = \frac{||J_f(x)||\,||x||}{||f(x)||} = \frac{2 \cdot \max(x_1, x_2)}{|x_1 - x_2|}$$

under the inf $-$norm since $J_f(x) = [1, -1]$. We see here that if $x_1 \approx x_2$ and x_1, x_2 are not close to 0, then the relative conditioning of the subtraction operation is very ill-conditioned.

Example 11.5. Let $f(x) = Ax$.

$$\kappa_r = ||A||\,\frac{||x||}{||Ax||}$$

If we assume A to be square and nonsingular, then

$$\kappa_r = ||A||\,\frac{\left|\left|A^{-1}Ax\right|\right|}{||Ax||} \leq ||A||\,\left|\left|A^{-1}\right|\right|$$

Similarly, if we were to let $f(x) = A^{-1}x, \kappa_r \leq \left|\left|A^{-1}\right|\right|\,||A||$. That is to say, the sensitivity of solving $Ay = b$ for slightly perturbed y or b is bounded by the same condition number.

2 Conditioning of a System of Equations

A natural question extending from the previous section is how sensitive is $f(x) = Ax$ to changes in the coefficient matrix A? Let y be the solution to $Ay = b$ for some fixed $b \in \mathbb{R}^n$ and $y + \Delta y$ be the solution to a slightly perturbed problem $(A + \Delta A)(y + \Delta y) = b$. From the latter, we have that

$$b = Ay + \Delta Ay + A\Delta y + \Delta A\Delta y \approx Ay + +\Delta Ay + A\Delta y$$

since we can drop the double infinitesimal term in the limit. Now substituting $Ay = b$ we have that $\Delta Ay = -A\Delta y$ or that $\Delta y = -A^{-1}\Delta Ay$. Taking norms and applying Cauchy-Schwarz provides us with $\Delta y \leq \left|\left|A^{-1}\right|\right|\,||\Delta A||\,||y||$. Thus,

$$\kappa_r = \sup_{\Delta A}\frac{||\Delta y||\,/\,||y||}{||\Delta A||\,/\,||A||} \leq \left|\left|A^{-1}\right|\right|\,||A||$$

3 Conditioning of Eigenvalues of Matrices

Consider the matrix $A = \begin{bmatrix} 1 & \frac{1}{\varepsilon} \\ 0 & 1 \end{bmatrix}$. It is not difficult to show that $\lambda_1 = \lambda_2 = 1$. However, through a small perturbation we may represent the above as $\hat{A} = \begin{bmatrix} 1 & \frac{1}{\varepsilon} \\ \varepsilon & 1 \end{bmatrix}$. In this case, we see that $\lambda_1 = 0, \lambda_2 = 2$, a significant difference. In general, for non-symmetric matrices, certain eigvenalues could be very sensitive. Let λ be a simple eigenvalue of A and v and w be the corresponding right and left eigenvectors. That is, $Av = \lambda v$ and $w^H A = \lambda w^H$. Furthermore, set E to be a small perturbation of A such that $(A + E)(v + \Delta v) = (\lambda + \Delta\lambda)(v + \Delta v)$. Then, $|\Delta\lambda| = \frac{||E||_2}{\cos\angle(v,w)}$. In the symmetric case, $v = w$ in direction and therefore $\kappa_a = 1$.

4 Conditioning of Roots of a Polynomial

Recall from before that to interpolate a set of points using monomial basis is terrible as it requires the solve of a dense $n \times n$ matrix that has high condition number. We will further reinforce the idea that the monomial basis should very rarely be used.

Consider the solving $x^2 - 2x + 1 = 0$. Clearly $x_1 = x_2 = 0$. Now, for the slightly perturbed problem $x^2 - 2x + 1 - \delta^2$, we have $x_1 = 1 - \delta, x_2 = 1 + \delta$ in exact arithmetic. However, if $\delta < \sqrt{\varepsilon}$, then the roots become $x_1 = x_2 = 1$ as the original problem is unchanged in double precision ($\delta^2 < \varepsilon_{mach}$). Here, a relative perturbation in one coefficient of magnitude $\mathcal{O}(\delta^2)$ produces a perturbation in the roots of magnitude $\mathcal{O}(\delta)$. So condition number of the roots are $\lim\limits_{\delta \to 0} \frac{\mathcal{O}(\delta)}{\mathcal{O}(\delta^2)} = \infty$.

In general, if a polynomial in the monomial basis has a repeated root of multiplicity m, then a perturbation in absolute value in the polynomial coefficients of $\mathcal{O}(\delta^m)$ is enough to warrant an error of $\mathcal{O}(\delta)$ in the roots.

Example 11.6. Let's take a look at a more general example. Consider the Wilkinson Polynomial $p(x) = (x-1)(x-2)\dots(x-23)(x-24) = x^{24} + \dots + a_1 x + a_0$. How sensitive are the roots to perturbations in coefficients? Consider p as a function of the coefficients **and** x. It follows by Taylor expansion that

$$\begin{aligned} 0 &= p(x_j + \Delta x_j; a_0, \dots, a_i + \Delta a_i, \dots, a_{23}) - p(x_j; a_0, \dots, a_23) \\ &= p(x_j; a_0, \dots, a_{23}) + \frac{\partial P}{\partial x_j} \mid_{(x_j; a_0, \dots, a_{23})} \Delta x_j + \frac{\partial P}{\partial a_i} \mid_{(x_j; a_0, \dots, a_{23})} \Delta a_i - p(x_j; a_0, \dots, a_{23}) \end{aligned}$$

Therefore,

$$\Delta x_j = -\frac{\frac{\partial P}{\partial a_i} \Delta a_i}{\frac{\partial P}{\partial x_j}} = -\frac{x_j^i \Delta a_i}{p'(x_j)}$$

And the relative condition is consequently

$$\kappa_r = \lim_{\Delta a_i \to 0} \frac{|\Delta x_j / x_j|}{|\Delta a_i / a_i|} = \frac{x_j^{i-1} a_i}{p'(x_j)}$$

For instance, if $i = j$, then $\kappa_r = 3.54 \cdot 10^{15}$. That is, a perturbation of machine precision on this coefficient makes it practically impossible to find the corresponding root!

5 Algorithm Stability

When solving a problem numerically, the conditioning of the problem is only half the battle. We also need to ensure that our algorithm is stable. Let $y = f(x)$ be the true solution and $\hat{y} = \hat{f}(x)$ be the computed solution by some numerical algorithm.

We naturally would like $\frac{||f(x) - \hat{f}(x)||}{||f(x)||}$ to be small. An algorithm is called accurate if this can be achieved for any valid input. However, if the problem is ill-conditioned, such an expectation is a bit ambitious. In this case, a reasonable expectation is that the algorithm is stable. That is, for each input x, there exists some Δx such that

$$\frac{\left|\left|\hat{f}(x) - f(x + \Delta x)\right|\right|}{||f(x + \Delta x)||} = \mathcal{O}(\varepsilon_{mach}) \text{ and } \frac{||\Delta x||}{||x||} = \mathcal{O}(\varepsilon_{mach})$$

The difference between the computed solution and the true solution is referred to as the forward error. Another stronger type of stability is called backward stability. In backward stability, we require that the computed solution is **exactly** the solution to a slightly perturbed problem. That is, $\hat{f}(x) = f(x + \Delta x)$ for some Δx satisfying $\frac{||\Delta x||}{||x||} = \mathcal{O}(\varepsilon_{mach})$.

Example 11.7. Let's look at an example of an algorithm that is not backward stable: Gaussian Elimination/LU factorization without pivoting. Let $A = \begin{bmatrix} \frac{\varepsilon}{2} & -1 \\ 1 & 1 \end{bmatrix}$. Its LU factorization in exact arithmetic is

$$A = \begin{bmatrix} 1 & 0 \\ \frac{2}{\varepsilon} & 1 \end{bmatrix} \begin{bmatrix} \frac{\varepsilon}{2} & -1 \\ 0 & 1 + \frac{2}{\varepsilon} \end{bmatrix}$$

However, when we perform the same task numerically, we obtain

$$\hat{A} = \begin{bmatrix} 1 & 0 \\ \frac{2}{\varepsilon} & 1 \end{bmatrix} \begin{bmatrix} \frac{\varepsilon}{2} & -1 \\ 0 & \frac{2}{\varepsilon} \end{bmatrix} = \begin{bmatrix} \frac{\varepsilon}{2} & -1 \\ 1 & 0 \end{bmatrix}$$

Therefore, $\Delta A = A - \hat{A} \implies \frac{||\Delta A||_\infty}{||A||_\infty} = \frac{1}{2} >> \mathcal{O}(\varepsilon)$, i.e. the algorithm is not backward stable.

6 Stability of Linear Solvers

In this section, we will consider the stability of a few algorithms to solve systems of linear equations ($Ax = b$). We will take a look at

1. Gaussian Elimination/LU factorization with partial pivoting
2. Cramer's Rule
3. Compute A^{-1} then $x = A^{-1}b$
4. QR factorization: $x = A^{-1}b = R^{-1}Q^{-1}b = R^{-1}(Q^Tb)$

Let $\hat{x}$ be the computed solution generated by these algorithms. We know that the forward error $\frac{||\hat{x}-x||}{||x||}$ could be large if A is ill-conditioned. We will instead explore the backward error. We define the backward error as the solution to $\min\limits_{\Delta A} \frac{||\Delta A||}{||A||}$ s.t. $(A + \Delta A)\hat{x} = b$. Our question now is how can we solve such a problem?

m:backwardstable⟩ Let $r = b - A\hat{x}$ be the residual of the computed solution. Then

$$\min_{\Delta A} \frac{||\Delta A||_2}{||A||_2} = \frac{||r||_2}{||A||_2\, ||\hat{x}||_2}$$

To complete the proof, we will need to make use of the following lemma. Let $u, v \in \mathbb{R}^n$. Then $||uv^T||_2 = ||u||_2\, ||v||_2$.

Proof. Let $u, v \in \mathbb{R}^n$. To prove the lemma, first set $\tilde{u}, \tilde{v}$ to be unit vectors in the direction of u, v respectively, i.e. $\tilde{u}\, ||u||_2 = u, \tilde{v}\, ||v||_2 = v$. Set U, V to be orthonormal basis extensions of u, v. Denote E to be the zero $n \times n$ matrix with a 1 in the first entry. Then $\tilde{u}\tilde{v}^T = \tilde{A} = UEV^T$. Scaling up to A, we see that $||u||_2\, ||v||_2$ is the only nontrivial singular value of A. Thus, it must be the 2-norm. □

Proof of Theorem 6. Let $r = b - A\hat{x}$. From $(A + \Delta A)\hat{x} = b$, we get simplify to $r = \Delta A\hat{x}$. Therefore, $||r||_2 = ||b - A\hat{x}||_2 \leq ||\Delta A||_2\, ||\hat{x}||_2$. It directly follows that

$$\frac{||r||_2}{||A||_2\, ||\hat{x}||_2} \leq \frac{||\Delta A||_2}{||A||_2}$$

To achieve equality, consider the 1 rank matrix $\Delta A = \frac{r\hat{x}^T}{\hat{x}^T\hat{x}}$. By the lemma,

$$||\Delta A||_2 = \frac{||r||_2\, ||\hat{x}||_2}{||\hat{x}||_2^2} = \frac{||r||_2}{||\hat{x}||_2}$$

And so the relative backward error is

$$\frac{||\Delta A||_2}{||A||_2} = \frac{||r||_2}{||A||_2\, ||\hat{x}||_2}$$

which completes our proof. □

Example 11.8. Recall the previously mentioned algorithms. Let A be the Hilbert matrix of order 10. Set $x = [1, \dots, 1]^T, b = Ax$. We now solve for $\hat{x}$ using the above algorithms and compute $\min\limits_{\Delta A} \frac{||\Delta A||_2}{||A||_2}$. The results are summarized in the table below

Algorithm	**Relative Backward Error**	**Relative Forward Error**
GEPP/LUPP	10^{-16}	3.3×10^{-4}
Cramer's Rule	10^{-6}	5.7×10^{-4}
$A^{-1}b$	10^{-5}	10^{-3}
QR Factorization	10^{-16}	1.2×10^{-3}

Clearly, Cramer's Rule and solving via inverses are not backward stable. However, these two methods appear to perform decently well in terms of forward error. We say that all of these algorithms are forward stable. A loose definition of forward stability is an algorithm is forward stable if it produces a forward error similar to that of the forward error produced by a backward stable algorithm!

7 Conditioning of GE/LU factorization

Continuing with our stability exploration of linear solvers, let A be a nonsingular square matrix of order n and assume that no zero pivot arises during factorization in exact arithmetic such that $A = LU$. Then for sufficiently small $\varepsilon_{\text{mach}}$, the factorization can also be completed successfully in floating point arithmetic. Furthermore, let $\hat{L}$ and $\hat{U}$ be the computed factors of LU decomposition. Then, it can be shown that

$$\hat{L}\hat{U} = \hat{A} = A + \Delta A$$

where ΔA satisfies $\frac{||\Delta A||}{||\hat{L}||||hU||} = \mathcal{O}(\varepsilon)$. Similarly, let $|A|$ be the matrix obtained by component-wise absolute value operation. It can be shown that

$$|\Delta A| \leq \frac{n\varepsilon_{\text{mach}}}{1 - n\varepsilon_{\text{mach}}} \cdot \left|\hat{L}\right| \left|\hat{U}\right|$$

holds component-wise. Furthermore, suppose these factors were used in the forward/backward substitutions of solving $Ax = b$. Then $\hat{x}$ satisfies

$$(A + \Delta A)\hat{x} = b \text{ where } |\Delta A| \leq \tfrac{3n\varepsilon_{\text{mach}}}{1-3n\varepsilon_{\text{mach}}} \left|\hat{L}\right| \left|\hat{U}\right|$$

Thus, for backwards stability of both factorizing A and solving $Ax = b$, we need

$$\frac{||\Delta A||}{||A||} = \mathcal{O}(\varepsilon) \text{ or } |\Delta A| \leq \mathcal{O}(\varepsilon)\,|A|$$

Therefore, the stability is dependent on whether or not we have $\left|\left|\hat{L}\right|\right| \left|\left|\hat{U}\right|\right| \leq C_n \,||A||$ or $\left|\hat{L}\right| \left|\hat{U}\right| \leq C_n \,|A|$. Note that this is equivalent to C_n 'not being too large' because combining the above yields

$$\frac{|\Delta A|}{|A|} \leq \frac{n\varepsilon_{\text{mach}}}{1 - n\varepsilon_{\text{mach}}} C_n$$

To explore how large C_n would be, let $A^{(k)}$ be the intermediate matrix during factorization. Define

$$\rho_n := \frac{\max \left|a_{ij}^{(k)}\right|}{\max |a_{ij}|}$$

Example 11.9. Let's look at an example for computing ρ_n. Consider the factorization

$$A = \begin{bmatrix} \varepsilon & -1 \\ 1 & 1 \end{bmatrix} \longrightarrow \begin{bmatrix} \varepsilon & -1 \\ \frac{1}{\varepsilon} & \frac{1}{\varepsilon} + 1 \end{bmatrix}$$

after one pivot. Thus,

$$L = \begin{bmatrix} 1 & 0 \\ \frac{1}{\varepsilon} & 1 \end{bmatrix}, U = \begin{bmatrix} \varepsilon & -1 \\ 0 & \frac{1}{\varepsilon} + 1 \end{bmatrix}$$

It follows that $\rho_n \leq 1 + \frac{1}{\varepsilon} = \mathcal{O}(\frac{1}{\varepsilon})$ which is very bad.

In general, we have the following result (that we present without proof) for LU factorization without pivoting.

$$|||L|\,|U|||_\infty \leq [1 + 2(n^2 - n)\rho_n]\,||A||_\infty$$

If ρ_n is small, $|L|\,|U|$ will also be small. For sufficiently small $\hat{L}, \hat{U}$ satisfy a similar relation. Therefore, the magnitude of ρ_n determines the backwards stability of GE/LU. To achieve backwards stability, however, we must use pivoting to control ρ_n. At the very least, we need to ensure ρ_n depends on n only. To accomplish this, we use partial pivoting.

Let $PA = LU$ be the exact LU factorization of A with partial pivoting. Then, the computed factors satisfy $\hat{L}\hat{U} = \hat{P}A + \Delta A$ where $\frac{||\Delta A||}{||A||} = \mathcal{O}(\rho_n \varepsilon_{\text{mach}})$ and $\rho_n \leq 2^{n-1}$. In general, if GEPP is used to solve $Ax = b$, then the computed solution satisfies

$$(A + \Delta A)\hat{x} = b \text{ where } ||\Delta A||_\infty \leq \frac{3n^2 \rho_n \varepsilon_{\text{mach}}}{1 - 3n\varepsilon_{\text{mach}}}\,||A||_\infty$$

While this may look promising at first, note that $\rho_n \leq 2^{n-1}$ is in many ways unacceptable. As n grows large, even partial pivoting cannot guarantee backward stability. However, in practice, this upper bound is far from attained. In fact, only for very constructed examples does the growth factor ever reach unacceptable levels. With this in mind, we instead say that GEPP is backwards stable for a fixed n.

The above paragraph is not to say that all matrices suffer this unfortunate fate. We conclude this section by reiterating the idea that matrices with special structure (SPD, row/diagonal dominance) do not require pivoting at all to be backwards stable.

Chapter 12

QR Factorization

We begin the next chapter with a look into one of (if not the) most important concepts in all of numerical linear algebra: QR factorization.

1 Properties of QR

Given a matrix $A \in \mathbb{R}^{m \times n} (m \geq n)$, the *reduced* QR factorization produces $Q_1 \in \mathbb{R}^{m \times n}$ (same size as A) and $R_1 \in \mathbb{R}^{n \times n}$ (upper triangular) such that $A = Q_1 R_1$ where Q_1 has orthonormal columns (i.e. $Q_1^T Q_1 = I_n$).

Furthermore, we can also compute a full QR factorization of the form $A = \begin{bmatrix} Q_1 & Q_2 \end{bmatrix} \begin{bmatrix} R_1 \\ 0 \end{bmatrix}$ where

$$\begin{aligned} \begin{bmatrix} Q_1 & Q_2 \end{bmatrix}^T \begin{bmatrix} Q_1 & Q_2 \end{bmatrix} &= I_m \\ Q_1^T Q_1 &= I_n \\ Q_2^T Q_2 &= I_{m-n} \\ Q_1^T Q_2 &= O_{n \times (m-n)} \end{aligned}$$

Let $A = QR$ be a reduce QR factorization of A. Multiplying each on the right by e_k, we obtain the kth columns. Thus, we have

$$a_k = Ae_k = QRe_k = \sum_{i=1}^{k} q_i r_{ik}$$

That is, the columns of A can be written as a linear combination of those in Q. Does the reverse direction hold? Theorem 1 sheds light on this topic.

thm:rnonsingular⟩ Let $A = QR$ be a reduced QR factorization of A. If A has full column rank, then $r_{jj} \neq 0$ for all j and consequently R is nonsingular. Therefore, $AR^{-1} = Q$, i.e. the columns of Q can be written as linear combinations of of those in A.

Proof. Assume for the sake of contradiction that $r_{kk} = 0$ for some $1 \leq k \leq n$. Then R is singular. Thus the column space of QR cannot be n, contradicting the fact that A has linearly independent columns. □

In general, if A is of rank k and the first $l \leq k$ columns of A are linearly independent, but the $(l+1)$st column is a linear combinations of the first l columns of A, then the top left submatrix of R is nonsingular, but $r_{l+1,l+1} = 0$. Thus, q_{l+1} cannot be written as a linear combination of $A_1, \dots A_{l+1}$.

Proof. For a proof, consider the proof of Theorem 1 restricted to only the first l columns. □

2 QR Factorization via Gram-Schmidt

We will now consider methods in which we can compute the QR factorization. Directly enforcing the requirements of $A = QR$, we can derive $A_1 = r_{11}Q_1 \implies q_1 = A_1/r_{11}$. Continuing in this manner, we can write the k step as $Q_k = \dfrac{A_k - \sum_{i=1}^{k} Q_i r_{ik}}{r_{kk}}$. This is in fact just the Gram-Schmidt algorithm applied to the columns of A. The steps of this Gram-Schmidt Process are summarized in the Algorithm 12.1. A

Algorithm 12.1 Classical Gram-Schmidt

⟨alg:classicGS⟩
```
for $k = 1, \dots, n$ do
  for $i < k$ do
    $r_{ik} = Q_i^T A_k$
  end for
  $\tilde{A}_k = A_k - \sum_{i=1}^{k-1} Q_i r_{ik}$
  $|r_{kk}| = \left|\left|\tilde{A}_k\right|\right|_2$
  $Q_k = \tilde{A}_k r_{kk}$
end for
```

few remarks about the algorithm,

1. Notice that we set the modulus of $|r_{kk}|$, but not the vector itself. That is, the QR factorization is unique up to scaling by a complex number with modulus 1.
2. Recall that if A does not have full column rank, then we will have some $r_{jj} = 0$. To continue in this case, we may choose any unit vector $u \notin \operatorname{span}\{Q_1, \dots, Q_{j-1}\}$ and orthogonalize it against $\{Q_1, \dots, Q_{j-1}\}$.

This process is referred to as the Classical Gram-Schmidt algorithm. It is wildly unstable in practice. Because of this, we from now on use Modified Gram-Schmidt detailed in Algorithm 12.2. For Modified

Algorithm 12.2 Modified Gram-Schmidt

⟨alg:modifiedGS⟩
```
for $k = 1, \dots, n$ do
  $Q_i = A_i$
end for
for $k = 1, \dots, n$ do
  $r_{kk} = ||Q_k||_2$
  $Q_k = Q_k / r_{kk}$
  for $j = k+1, \dots, n$ do
    $r_{kj} = Q_k^T Q_j$
    $Q_j = Q_j - r_{kj} Q_k$
  end for
end for
```

GS, we have the following result

$$\left|\left|Q^T Q - I_n\right|\right| = \mathcal{O}(\kappa_2(A)\varepsilon_{\text{mach}})$$

From this result, we see that we may not be able to guarantee that the columns of our output are orthogonal, the most important property of GS. It is for this reason that common practice is to simply perform GS twice for reliable orthogonality.

GS Computational Cost

We will now turn our attention to the computation cost of QR factorization via Gram-Schmidt. Recall that at step k of Gram-Schmidt, we compute the following

$$\begin{aligned}\tilde{A}_k &= A_k - \sum_{i=1}^{k-1} Q_j(Q_j^T A_k) \\ A_k &= \tilde{A}_k / \left|\left|\tilde{A}_k\right|\right|\end{aligned}$$

The computational cost of such is

$$[m + (m-1) + m + m](k-1) + 3m$$

Thus, for all k, we have

$$\begin{aligned}\text{cost} &= \sum_{k=1}^{n} [4m-1](k-1) + 3m \\ &= (4m-1)(\frac{(n-1)(n)}{2}) + 3mn \\ &\approx 2mn^2\end{aligned}$$

And again, for increased reliability, we perform this operation twice.

3 QR Factorization via Orthogonal Transformations

The recommended approach for QR factorization is by orthogonal transformations, i.e. Householder transformations and Givens rotations. We will begin with Householder. While our task may be motivated by QR, let us first consider a slightly simpler problem: given a vector x, project x onto the space spanned by e_1. Define $v = x - ||x||_2 e_1$ and note that

$$x - 2\,\text{proj}_v(x) = x - v = ||x||_2 e_1$$

where $\text{proj}_v(x) = \frac{v^T x}{v^T v} v$. Substituting this into the previous gives

$$x - 2v\frac{v^T x}{v^T v} = (I - 2\frac{vv^T}{v^T v})x = ||x||_2 e_1$$

The matrix $H(x) = (I - \frac{vv^T}{v^T v})x$ is called the Householder reflector. $H(x)$ has many useful properties such as

1. Symmetry. $H^T = H, H^* = H$
2. Involutory. $H^2 = I - 4\frac{vv^T}{v^T v} + 4\frac{vv^T}{v^T v} = I$
3. Orthogonal. $H^T H = H^2 = I$

Property (3) is perhaps the most important of the Householder reflector. Orthogonality plays a major role in both QR factorization and SVD decomposition. A few nice consequences of orthogonality of a matrix Q are

- Multiplication by Q does not affect norm. $||Qv||^2 = (Qv)^T Qv = v^T Q^T Qv = v^T v = ||v||^2$
- Eigenvalues have modulus 1. For an eigenvalue/eigenvector pair λ, v, $||v|| = ||Qv|| = ||\lambda v|| = \lambda\,||v|| \implies |\lambda| = 1$
- Perfect conditioning. $\kappa_2(Q) = \frac{\text{largest eigenvalue}}{\text{smallest eigenvalue}} = 1$

With the help of Householder transformations, we can compute QR factorizations. Recall that $H(x)x = ||x||_2 e_1$. Let $A \in \mathbb{R}^{m\times n}$ be a fully dense matrix with $m > n$. Then applying $H(A_1)A$ eliminates the lower triangular portion of the first column, i.e.

$$A = \begin{bmatrix} x & x & x \\ \vdots & \vdots & \vdots \\ x & x & x \end{bmatrix} \overset{Q_1=H(A_1)A}{\rightarrow} \begin{bmatrix} x & x & x \\ 0 & x & x \\ \vdots & \vdots & \vdots \\ 0 & x & x \end{bmatrix}$$

We then apply Householder again to the first column of the first principle minor of the transformed matrix. We now have $Q_2 = \begin{bmatrix} I_1 & O_{1\times n} \\ O_{n\times 1} & H([O_{n\times 1} I_n]) \end{bmatrix}$ and

$$A = \begin{bmatrix} x & x & x \\ 0 & x & x \\ 0 & x & x \\ \vdots & \vdots & \vdots \\ 0 & x & x \end{bmatrix} \overset{Q_2}{\rightarrow} \begin{bmatrix} x & x & x \\ 0 & x & x \\ 0 & 0 & x \\ \vdots & \vdots & \vdots \\ 0 & 0 & x \end{bmatrix}$$

We may continue in this fashion until we have transformed A into an upper triangular matrix R. Since $AQ_1Q_2\dots Q_n = R$, R is the finished product of our transformations of A and Q is the orthogonal matrix $Q_nQ_{n-1}\cdots Q_1$.

In addition to the derivation of Householder reflectors, we must also consider the practical side of using Householder transformations to compute QR factors. Below we list some of the more computational ideas associated with the algorithm.

- When applying the Householder reflectors, the Householder reflector should never be explicitly formed, e.g.
$$HU = (I - 2\frac{vv^T}{v^Tv})U = U - v(\frac{2}{v^Tv}(v^TU))$$
- Note that $Q_nQ_{n-1}\dots Q_1A = \begin{bmatrix} R \\ 0 \end{bmatrix} \implies A = Q_1\dots Q_n \begin{bmatrix} R \\ 0 \end{bmatrix}$. Denoting Q to be the product of the individual Q_i factors, we see that $A = Q\begin{bmatrix} R \\ 0 \end{bmatrix} = [Q_LQ_R]\begin{bmatrix} R \\ 0 \end{bmatrix} = Q_LR$. Thus, for a reduced QR factorization, we only need to compute Q_L! Observe that
$$A = Q\begin{bmatrix} R \\ 0 \end{bmatrix} = Q\begin{bmatrix} I \\ 0 \end{bmatrix}$$
Therefore, to explicitly compute Q_L, simply apply $Q_n, \dots Q_1$ to the left of I_n.
- In many problems, such as linear least squares, explicit computation of Q_L is not necessary! Consider solving $\min ||b - Ax||_2$ by QR factorization where A is full rank. We have previous seen that either $x = (A^TA)^{-1}A^Tb$ or $x = R^{-1}Q^Tb$ would suffice. However, the later in Householder reduces to $x = R^{-1}([I_n0_{n\times(m-n)}]Q_n\dots Q_1b)$. Thus, there is no need to explicitly form Q_L either.

Householder Computational Cost

As always, it is essential to know the computational cost of an algorithm. Algorithm 12.3 showcases some pseudocode of the process. We can enumerate the flops necessary for each steps as in the table below

Step	**Flops Required**
$\|\|x_k\|\|_2$	$2(m-k+1)$
$\|\|x_k\|\|_2$	1
$\|\|x_k\|\|_2 - x_k$	$(m-k+1)$
$v_k/\|\|v_k\|\|_2$	$3(m-k+1)$
$2v_k^TA(k:m,j)$	$2(m-k+1)+1$
$A(k:m,j) - v_k(2v_k^TA(k:m,j)$	$2(m-k+1)+1)$

where the latter two steps are done $n-k$ times. In total, this is roughly

$$\begin{aligned}\sum_{k=1}^{n}\{6(m-k+1)+1+[4(m-k+1)+1](n-k)\} &\approx \sum_{k=1}^{n}4(m-k+1)+1](n-k)\\ &= 4(m-n+1)\sum_{k=1}^{n}n-k+4\sum_{k=1}^{n}(n-k)^2\\ &= 4(m-n+1)\frac{(n-1)n}{2}+4\frac{(n-1)n(2n-1)}{6}\\ &\approx 2mn^2-\frac{2}{3}n^3\end{aligned}$$

A few comments must be made here

- This is only for the R factor. We still need to do additional work to get our Q factor.
- In certain problems, like Linear Least Squares mentioned above, we can avoid doing this work.
- Note that when $m = n$, the cost of solving Linear Least Squares using QR factorization via Householder transformations costs $\frac{4}{3}n^3$ as opposed to $\frac{2}{3}n^3$ for GEPP.
- Modified GSQR is even worse since it requires $2mn^2$ flops.

Algorithm 12.3 Householder Transformation Phase 1

⟨alg:household⟩
```
$v$ =zeros$(m,n)$
for $k = 1,\dots,n$ do
    $x_k = A(k:m,k)$;
    $v_k = ||x_k||_2\, e_1 - x_k$;
    $v_k = v_k/\,||v_k||_2$
    $A(k:m,k) = ||x_k||_2\, e_1$
    $A(k:m,k+1:n) = A(k:m,k+1:n) - v_k 2(v_k^T(A(k:m,k+1:n)))$
end for
```

Lastly, for QR factorization via Householder reflections, we have Theorem 3 regarding the stability of Householder presented without proof.

:houseBackStable⟩ Let $\hat{Q},\hat{R}$ be the computed factors of A by Householder such that $\hat{Q}\hat{R} = \hat{A} = A + \Delta A$. Then, $\frac{||\Delta A||}{||A||} = \mathcal{O}(\varepsilon_{\text{mach}})$ and $\left|\left|\hat{Q}^T\hat{Q} - I_n\right|\right| = \mathcal{O}(\varepsilon_{\text{mach}})$.

4 QR Factorization via Given's Rotation

Although Householder transformations are the defacto method of performing QR factorization, there is occasionally a more efficient method, Given's rotation. For a given vector $\begin{bmatrix}a\\b\end{bmatrix} \in \mathbb{R}^2$, define $G = \begin{bmatrix}c & s\\ -s & c\end{bmatrix}, c = \frac{a}{\sqrt{a^2+b^2}}, s = \frac{b}{\sqrt{a^2+b^2}}$. As a result, $G\begin{bmatrix}a\\b\end{bmatrix} = \begin{bmatrix}ca+sb\\ -sa+cb\end{bmatrix} = \begin{bmatrix}\sqrt{a^2+b^2}\\ 0\end{bmatrix}$. If $a = b = 0, G = I_2$. It is easy to see that $G^TG = I$. Thus, this is an orthogonal transformation.

5 Given's Rotation Computational Cost

We can use this transformation to compute a QR factorization. We simply need to apply the Given's Rotation to all nonzero entries of the lower triangular part of A.

Step	**Flops Required**
Construction of Given's Transformation	6
$G\begin{bmatrix}a\\b\end{bmatrix}$ multiplication	6
Apply G to a row	$6+6(n-1)$
Apply G to entire matrix	$(6+6(n-1))(m-1)$

Thus, summing over all columns, we get a total cost of

$$\begin{aligned} 6\sum_{k=1}^{n-2}(n-k)(m-k-1) &= 6\sum_{k=0}^{n-2}(n-k)[(n-k)(m-n-1)] \\ &= 6\sum_{k=0}^{n-2}(n-k)^2 + 6(m-n-1)\sum_{k=0}^{n-2} n-k \\ &= 6\left(\frac{n(n+1)(2n+1)}{6} - 1\right) + 6(m-n-1)\frac{(n+2)(n-1)}{2} \\ &\approx 2n^3 + 3(m-n)n^2 \\ &= 3mn^2 - n^3 \end{aligned}$$

flops. Compare this with Householder transformations. It is roughly 50% worse! However, consider the case when A is upper Hessenberg. It can be shown that only $3n^2$ flops are needed. Similarly, if A is tridiagonal, we only need roughly $14n$ flops for (phase 1) QR factorization. That is, Given's rotation actually outperforms Householder transformations matrices with special structures.

6 Linear Least Squares (seems out of place)

Consider $Ax \approx b$ where $A \in \mathbb{R}^{m\times n}$ is full rank. We look for $x \in \mathbb{R}^n$ such that $||r||_2 = ||b - Ax||_2$ is minimized. To solve this, we use the following x is the solution to the above linear least squares problem if and only if $r = b - Ax \perp \text{range}(A)$.

Proof. Let x be the solution, but assume that $r = b - Ax \not\perp \text{range}(A)$. Let P be the orthogonal projection of b onto range(A) such that $\exists \hat{x} \in \mathbb{R}^n$ satisfying $Pb = A\hat{x}$ and $b - A\hat{x} = b - Pb \perp \text{range}(A)$. Then $||b - Ax||_2 = ||b - A\hat{x} + A\hat{x} - Ax||_2 = ||b - A\hat{x}|| + ||A(\hat{x} - x)|| > ||b - Ax||_2$, which is a contradiction. For the other direction, assume x satisfies $b - Ax \perp \text{range}(A)$. Then r is perpendicular to each column of A, i.e. $A^Tr = A^T(b - Ax) = 0 \equiv A^TAx = A^Tb$. We know the solution x is such that $Ax = Pb$. Then $A = QR, P = QQ^T$. Thus, $QRx = QQ^Tb \implies Rx = Q^Tb$. So $x = R\backslash Q^Tb = R^{-1}Q^Tb$. □

7 A dual problem to LSQ

Assume that $Ax = b, A \in \mathbb{R}^{m\times n}$ is full (row) rank. Find the solution with minimum 2-norm.

Solution. Let $A^T = QR$ be a reduced QR factorization of A^T. Then $A = R^TQ$ and the linear system we seek to solve is $R^TQ^Tx = b \implies Q^Tx = R^{-T}b$. Let us decompose x into $QQ^Tx + (I - QQ^T)x$, i.e. we can decompose x into the sum of elements in a particular projector and its complementary projector. From this decomposition, we must have that $x_1 \perp x_2$. It follows that $Q^T(QQ^Tx + (I - QQ^T)x = R^{-T}b \implies Q^Tx + 0 = R^{-T}b$. Thus, we can select x_2 to be anything. To minimize the norm, we will want $x = x_1$ and $x_2 = 0$. Then $x = x_1 = QR^{-T}b \implies x = R^TQb$.

Chapter 13

Singular Value Decomposition

Like QR factorizations, singular value decompositions play an important role in numerical linear algebra. A very natural and reliable way of solving virtually any numerical linear algebra problem is simply by asking: what if we take the SVD?

1 SVD Review

Assume that $A \in \mathbb{R}^{m\times n}$ with $m \geq n$. Then a singular value decomposition (SVD) of A is $A = U\Sigma V^T$ where $U \in \mathbb{R}^{m\times m}$, $\Sigma \in \mathbb{R}^{m\times n}$, $V \in \mathbb{R}^{n\times n}$, U and V are orthogonal matrices, $\sigma_i \geq 0$. We can rewrite this as $A = \begin{bmatrix} U_L & U_R \end{bmatrix} \begin{bmatrix} \Sigma \\ 0 \end{bmatrix} V^T = U_L \Sigma V^T$ to obtain a reduced SVD.

Every matrix $A \in \mathbb{R}^{m\times n}$ has a full SVD and the singular values are uniquely determined by A itself.

Proof. If A is the zero matrix, then the result is trivially satisfied with identities. If $A \neq 0$, then $||A|| > 0$. Let $v_1 \in \mathbb{R}^n, ||v_1||_2 = 1$ be such that $||Av_1|| = \frac{||Av_1||}{||v_1||} = ||A||$. Define $\sigma_1 := ||A|| > 0$ and $u_1 = \frac{1}{\sigma_1}Av_1$. It follows that $||u_1|| = 1$. Now suppose that $V_2 \in \mathbb{R}^{n\times(n-1)}$, $U \in \mathbb{R}^{m\times(m-1)}$ be such that $\begin{bmatrix} v_1 & V_2 \end{bmatrix}, \begin{bmatrix} u_1 & U_1 \end{bmatrix}$ are orthogonal matrices. Consider $A_1 = \begin{bmatrix} u_1 & U_2 \end{bmatrix}^T A \begin{bmatrix} v_1 & V_2 \end{bmatrix} = \begin{bmatrix} u_1^T A v_1 & u_1^T A v_2 \\ V_2^T A v_1 & U_2^T A v_2 \end{bmatrix} = \begin{bmatrix} \sigma_1 & 0 \\ 0 & B \end{bmatrix}$. Inducting on B gives the full SVD. □

The SVD has a few nice properties.

1. $\text{rank}(A) =$ number of non-zero singular values.
2. Assume $\text{rank}(A) = r \leq n$. Then $\text{range}(A) = \text{span}(u_1, \ldots, u_r)$ and $\text{null}(A) = \text{span}(v_{r+1}, \ldots, v_n)$.
3. $||A|| = \sigma_1, ||A||_F = \sqrt{\sigma_1^2 + \cdots + \sigma_r^2}$.
4. $\lambda_i(A^TA) = \sigma_i^2(A)$.
5. If $A = A^T$, then $|\lambda_i| = \sigma_i$.
6. $|\det A| = \left|\det U\Sigma V^T\right| = |\det U|\,|\det \Sigma|\,\left|\det V^T\right| = \prod \sigma_i$.

1.1 Low Rank Approximations

For a rank r matrix A, we have that

$$A = \begin{bmatrix} u_1, \ldots, u_r \end{bmatrix} \Sigma \begin{bmatrix} v_r^T \\ \vdots \\ v_n^T \end{bmatrix}$$

Assume with loss of generality that $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r$ and define $A_k = \sum_{i=1}^{k} \sigma_i u_i v_i^T$ where $k \leq r$. Then we the following theorem characterizes low rank approximations of A. A_k is the best rank k approximation to A in the 2-norm, i.e.

$$||A - A_k||_2 = \inf_{B \in \mathbb{R}^{m \times n}} ||A - B||_2 = \sigma_{k+1}$$

Proof. Suppose for the sake of contradiction that there exists some B of rank k such that $||A - B||_2 < ||A - A_k||_2 = \sigma_{k+1}$. Then B has a $n-k$ dimensional null space W and so for any $w \in W$, it follows that

$$||Aw|| = ||Aw - Bw|| = ||(A - B)w|| < ||w|| \, \sigma_{k+1}$$

Now consider the space V' spanned by the first $k+1$ right singular vectors of A. For any $v \in V'$,

$$||Av|| \geq \sigma_{k+1} \, ||v||$$

Since the sum of these dimensions is greater than n, there must be some nonzero element in both, a contradiction. Thus, A_k is the minimizer that we seek. □

2 Computing an SVD

The last topic for this short chapter is an incomplete one. We have previously discussed many ways to utilize an SVD, but no methods of computing such a form. The reason for this is simple: it is difficult to do well.

2.1 Naive Idea

Indeed, one could note that for a matrix A with SVD $A = U\Sigma V^T$,

$$A^T A = V \Sigma^2 V'$$

That is, the singular values of A are the square roots of the eigenvalues of $A^T A$. Then this problem is reduced to an eigenvalue decomposition (which will be covered in depth). Unfortunately, such a method fails to be stable. A backward stable algorithm for computing singular values would obtain $\hat{\sigma}_k$ satisfying

$$\hat{\sigma}_k = \sigma_k(A + \delta A), \frac{||\delta A||}{||A||} = \mathcal{O}(\varepsilon_{\text{mach}})$$

which, combined with previous perturbation analysis $|\sigma_k(A + \delta A) - \sigma(A)| \leq ||\delta A||_2$ gives

$$|\hat{\sigma}_k - \sigma_k| = \mathcal{O}(\varepsilon_{\text{mach}} \, ||A||)$$

Now finding $\lambda_k(A^T A)$ in the same fashion gives

$$\left| \hat{\lambda}_k - \lambda_k \right| = \mathcal{O}(\varepsilon_{\text{mach}} \, \left| \left| \mathbb{A}^T A \right| \right|) = \mathcal{O}(\varepsilon_{\text{mach}} \, ||A||^2)$$

Taking square roots gives

$$|\hat{\sigma}_k - \sigma_k| = \mathcal{O}(\varepsilon_{\text{mach}} \, ||A||^2 / \sigma_k)$$

which is noticeably worse by a factor of $||A|| / \sigma_k$. That is, algorithms aside, computing eigenvalues of $A^T A$ to give singular values of A is a bad idea.

However, all hope is not lost. Consider the matrix $H = \begin{bmatrix} 0 & H^T \\ H & 0 \end{bmatrix}$. Since $A = U\Sigma V^T$ implies $AV = U\Sigma$ and $A^T U = V\Sigma$, we have

$$\begin{bmatrix} 0 & H^T \\ H & 0 \end{bmatrix} \begin{bmatrix} V & V \\ U & -U \end{bmatrix} = \begin{bmatrix} V & V \\ U & -U \end{bmatrix} \begin{bmatrix} \Sigma & 0 \\ 0 & -\Sigma \end{bmatrix}$$

which is an eigenvalue decomposition of H. Thus, the singular values of A are the absolute value of the eigenvalues of H. Please convince yourself that this method is more stable. Unfortunately, the matrix H is a square matrix of dimension $m+n$. When $m >> n$, this is not very troubling, but still a consideration. The standard SVD algorithims are based around this approach, but never form such matrix explicitly.

2.2 Golub-Kahan Bidiagonalization

The algorithm by Golub-Kahan arrives at an SVD through two steps aptly named Phase 1 and Phase 2. Phase 1 attempts to reduce the matrix into a bidiagonal form through orthogonal transformations while Phase 2 takes the bidiagonal matrix into a diagonal one through an iterative process (also via orthogonal transformations). The Phase 2 operation is very finicky and is not covered.

The Phase 1 procedure, however, is not difficult to understand at all. We seek to transform a $m \times n$ matrix A into bidiagonal form. Since an SVD is of the form $A = U\Sigma V^T$, we have the freedom of letting the orthogonal operations perform on the left and right of A be different. That is to say, we can continuously apply distinct Householder transformations to the left and right of A to zero out columns and rows. Applied to a 4×3 matrix, this looks like

$$A = \begin{bmatrix} x & x & x \\ x & x & x \\ x & x & x \\ x & x & x \end{bmatrix} \overset{U_1^T\cdot}{\rightarrow} \begin{bmatrix} x & x & x \\ 0 & x & x \\ 0 & x & x \\ 0 & x & x \end{bmatrix} \overset{\cdot V_1}{\rightarrow} \begin{bmatrix} x & x & 0 \\ 0 & x & x \\ 0 & x & x \\ 0 & x & x \end{bmatrix} \overset{U_2^T\cdot}{\rightarrow} \begin{bmatrix} x & x & 0 \\ 0 & x & x \\ 0 & 0 & x \\ 0 & 0 & x \end{bmatrix} \overset{U_3^T\cdot}{\rightarrow} \begin{bmatrix} x & x & 0 \\ 0 & x & x \\ 0 & 0 & x \\ 0 & 0 & 0 \end{bmatrix}$$

At the end, we will have applied $n = 3$ reflectors on the left and $n - 2 = 1$ reflectors on the right. The total cost is effectively twice that of QR factorization since the process is just two QR schemes applied to A and A^T. Thus, the total cost is roughly $4mn^2 - \frac{4}{3}n^3$.