Numerical Methods Lecture Notes

Elijan Mastnak

2019-2020 Summer Semester

About These Notes

These are my lecture notes from the course *Numerične Metode* (Numerical Methods), an introductory-level elective course offered to second-year physics students at the Faculty of Math and Physics in Ljubljana, Slovenia. The course serves as an introduction to a wide range of numerical methods, including systems of linear equations, root-finding algorithms for non-linear equations, the linear least squares problem, eigenvalues, polynomial interpolation and numerical methods for ODEs. The exact material is specific to the physics program at the University of Ljubljana, but the content is fairly standard for an introductory numerical methods course. I am making the notes publicly available in the hope that they might help others learning the same material—the most recent version can be found on GitHub.

Navigation: For easier document navigation, the table of contents is "clickable", meaning you can jump directly to a section by clicking the colored section names in the table of contents. Unfortunately, *the clickable links do not work in most online or mobile PDF viewers*; you have to download the file first.

On Authorship: The material herein comes directly from the lectures given by Professor Emil Žagar, who accordingly deserves credit for the content in these notes. I take credit for nothing more than typesetting the notes, translating to English, and perhaps adding a additional comment or two for clarity.

Disclaimer: Mistakes—both trivial typos and legitimate errors—are likely. Keep in mind that these are the notes of an undergraduate student in the process of learning the material himself—take what you read with a grain of salt. If you find mistakes and feel like telling me, by Github pull request, email or some other means, I'll be happy to hear from you, even for the most trivial of errors.

Contents

1 Introduction and Review of Linear Algebra

1.1 Brief Summary of Floating Point Numbers

- In numerical calculations, floating point numbers are represented $x = \pm sb^e$
  - $b \in \mathbb{N}$ is the *base*. $b = 2$ in the binary system used by computers.
  - s is the *significand* (aka *mantissa*) and contains the number's significant figures. The maximum number of digits in the significand is the precision p.
  - e is the exponent, an integer value. $e \in \{L, L+1, \dots, U-1, U\}$
  - L and U are the smallest (lower) and largest (upper) possible values of the exponent e. L and U are integer numbers.
- A number system's format is concisely written $P(b, p, L, U)$.

  Double precision: Numbers are represented with 64 bits. $p = 53$, encompassing about 16 decimal digits. In the IEEE 754 standard, sign is one bit, e is 11 bits, and s is 52 bits. e ranges from -1024 to 1023 (signed) or from 0 to 2047 (unsigned).

  Single precision: Numbers are represented with 32 bits. $p = 24$, encompassing about 7 decimal digits. Sign is one bit, the s is 23 bits, and e is 8 bits.
- Bound on floating point error: For $x \in \mathbb{R}$, $\text{float}(x) = x(1+\delta), \quad |\delta| \leq \frac{1}{2}b^{1-p}$

1.2 Review of Linear Algebra Basics

This section briefly summarizes a few concepts from linear algebra we'll need later in the course.

1.2.1 Matrices

- Matrices are two-dimensional grids of numbers with m rows and n columns. A real-valued $m \times n$ matrix is written $\mathbf{A} \in \mathbb{R}^{m \times n}$.

  Matrices are defined element-wise in the form

  $$\mathbf{A} = [\text{A}_{ij}], i \in \{1, 2, \dots, m\}, j \in \{1, 2, \dots, n\}$$

  Matrices may also be defined as a collection of n $m \times 1$ column vectors $\boldsymbol{a}_i \in \mathbb{R}^m$.

  $$\mathbf{A} = [\boldsymbol{a}_1 \ \boldsymbol{a}_2 \ \cdots \ \boldsymbol{a}_n]$$

- Matrix multiplication: The matrices $\mathbf{A} \in \mathbb{R}^{m_a \times n_a}$ and $\mathbf{B} \in \mathbb{R}^{m_b \times n_b}$ can be multiplied only if $n_a = m_b$, i.e. if $\mathbf{A}$ has the same number of columns as $\mathbf{B}$ has rows. In this case, the product $\mathbf{C} = \mathbf{AB} \in \mathbb{R}^{m_a \times n_b}$ has m_a rows and n_b columns.
- Multiplying a matrix $\mathbf{A}$ by a vector $\boldsymbol{b}$ may be thought of as a linear combination of $\mathbf{A}$'s column vectors where the coefficients are the elements of $\boldsymbol{b}$.

  $$\mathbf{A}\boldsymbol{b} = b_1\boldsymbol{a}_1 + b_2\boldsymbol{a}_2 + \cdots + b_n\boldsymbol{a}_n$$

- Some matrix properties:
  - A matrix is *singular* if its determinant is zero. A matrix is *non-singular* if its determinant is non-zero.
  - $\mathbf{A} \in \mathbb{R}^{n\times n}$ is *symmetric* if $\mathbf{A} = \mathbf{A}^T$. Symmetric matrices have all real eigenvalues.
  - $\mathbf{A} \in \mathbb{R}^{n\times n}$ is *positive definite* if $\boldsymbol{x}^T\mathbf{A}\boldsymbol{x} \geq 0$ for all $\boldsymbol{x} \in \mathbb{R}^n$. Equivalently, $\mathbf{A}$ is positive definite if it has all positive eigenvalues.
  - A matrix is *row diagonally dominant* if, for every row in the matrix, the absolute value of the diagonal entry in the row is larger than or equal to the absolute value sum of other elements in the row. In equation form:

$$|a_{ii}| \geq \sum_{j\neq i} |a_{ij}| \quad \text{for all } i \iff \mathbf{A} \in \mathbb{R}^{m\times n} \text{ row diagonally dominant}$$

  - A matrix is *column diagonally dominant* if, for every column in the matrix, the absolute value of the diagonal entry in the column is larger than or equal to the absolute value sum of other elements in the column.

$$|a_{jj}| \geq \sum_{i\neq j} |a_{ij}| \quad \text{for all } j \iff \mathbf{A} \in \mathbb{R}^{m\times n} \text{ column diagonally dominant}$$

1.2.2 Vector Norms

- Vector norms are a way to quantify the "size" of a vector. Formally, a vector norm is a function $\|\cdot\| : \mathbb{R}^{n\times 1} \to \mathbb{R}^+$ mapping vectors to the positive real numbers.
- Vector norms satisfy the following three properties for all vectors $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^n$
  1. Positive definiteness: $\|\boldsymbol{x}\| \geq 0$ and $\|\boldsymbol{x}\| = 0 \iff \boldsymbol{x} = \mathbf{0}$
  2. Homogeneity/scalability: $\|\alpha\boldsymbol{x}\| = |\alpha|\|\boldsymbol{x}\|$
  3. Triangle inequality (subadditivity): $\|\boldsymbol{x} + \boldsymbol{y}\| \leq \|\boldsymbol{x}\| + \|\boldsymbol{y}\|$
- Three important vector norms are:
  1. The Euclidean norm $\|\boldsymbol{x}\|_2 = \sqrt{\sum_{i=1}^{n} |x_i|^2}$, the intuitive distance between two points in Euclidean space.
  2. The one-norm $\|\boldsymbol{x}\|_1 = \sum_{i=1}^{n} |x_i|$, the absolute value sum of $\boldsymbol{x}$'s components.
  3. The infinity-norm $\|\boldsymbol{x}\|_\infty = \max_{i=1}^{n} |x_i|$, the largest component by absolute value.

1.2.3 Matrix Norms

- Matrix norms are a way to quantify the "size" of a matrix. Formally, a matrix norm is a function $\|\cdot\| : \mathbb{R}^{m\times n} \to \mathbb{R}^+$ mapping matrices to the positive real numbers.
- Matrix norms satisfy the following properties for all matrices $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{m\times n}$
  1. $\|\mathbf{A}\| \geq 0$
  2. $\|\mathbf{A}\| = 0 \iff \mathbf{A} = \mathbf{0}$
  3. $\|\alpha\mathbf{A}\| = \alpha\|\mathbf{A}\|$
  4. $\|\mathbf{A} + \mathbf{B}\| \leq \|\mathbf{A}\| + \|\mathbf{B}\|$
  5. Additionally, for square matrices, $\|\mathbf{AB}\| \leq \|\mathbf{A}\|\|\mathbf{B}\|$ (sub-multiplicative)
- Four important matrix norms are:
  1. The 2-norm $\|\mathbf{A}\|_2 = \max_i \sqrt{\lambda_i \in \text{eig}(\mathbf{A}^T\mathbf{A})}$, the square root of the matrix $\mathbf{A}^T\mathbf{A}$'s largest eigenvalue.
  2. The 1-norm $\|\mathbf{A}\|_1 = \max_{1\leq j\leq n} \sum_{i=1}^{m} |a_{ij}|$, $\mathbf{A}$'s largest absolute value column sum.
  3. The ∞-norm $\|\mathbf{A}\|_\infty \max_{1\leq i\leq m} \sum_{j=1}^{n} |a_{ij}|$,$\mathbf{A}$'s largest absolute value row sum.
  4. The Frobenius or F-norm $\|\mathbf{A}\|_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n} |a_{ij}|^2} = \sqrt{\text{tr}(\mathbf{A}^*\mathbf{A})}$, the square root of the sum of the squares of every element in $\mathbf{A}$.

1.2.4 Determinant

Every square matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$ has an associated scalar value called the determinant, denoted by $\det \mathbf{A}$. Some useful properties are:

- The determinant of a diagonal matrix (including upper and lower diagonal matrices) is the product of the diagonal terms
- The determinant of the identity matrix $\mathbf{I}$ is one. Each switch of $\mathbf{I}$'s rows or columns multiplies the determinant of the resulting matrix by one.

2 Systems of Linear Equations

2.1 Introduction

2.1.1 Convention for Linear Systems of Equations

- In matrix form, a system of m linear equations with n unknowns is written

$$\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$$

where $\mathbf{A} \in \mathbb{R}^{m\times n}, \boldsymbol{x} \in \mathbb{R}^{n\times 1}$, and $\boldsymbol{b} \in \mathbb{R}^{m\times 1}$. In general, a system of linear equations can have $0, 1$, or infinite solutions.

In general, the goal is: given the matrix $\mathbf{A}$ and the column vector $\boldsymbol{b}$, find the column vector $\boldsymbol{x}$ satisfying the equation $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$.

- For the majority of this course, we consider only the case $m = n$, corresponding to the same number of equations and unknowns. In this case $\mathbf{A} \in \mathbb{R}^{n\times n}$ and $\boldsymbol{x}, \boldsymbol{b} \in \mathbb{R}^{n\times 1}$, and the corresponding system of n equations for n unknowns has a unique solution if $\operatorname{rank}\mathbf{A} = \operatorname{rank}[\mathbf{A} + \boldsymbol{b}]$.

2.1.2 Condition Number

- Every matrix has an associated condition number $\kappa \geq 1$ measuring the matrix's tendency to propagate error resulting from numerical calculations.
- A matrix's condition number is defined as $\kappa(\mathbf{A}) = \|\mathbf{A}\|\left\|\mathbf{A}^{-1}\right\|$.
- $\kappa \geq 1$ for all matrices, and is equal to 1 for the identity matrix. The definition holds for any norm satisfying $\kappa(\mathbf{A}) = \|\mathbf{A}\|\left\|\mathbf{A}^{-1}\right\| \geq \left\|\mathbf{A}\mathbf{A}^{-1}\right\| = \|\mathbf{I}\| = 1$.
- In general, the more a matrix's columns are linearly dependent, the larger the matrix's condition number.
- **Bound for Error:** For a system of linear equations $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$

$$\frac{\|\delta\boldsymbol{x}\|}{\|\boldsymbol{x}\|} \leq \left(\frac{\kappa(\mathbf{A})}{1 - \kappa(\mathbf{A})\frac{\|\delta\mathbf{A}\|}{\|\mathbf{A}\|}}\right)\left(\frac{\|\delta\mathbf{A}\|}{\|\mathbf{A}\|} + \frac{\|\delta\boldsymbol{b}\|}{\|\boldsymbol{b}\|}\right)$$

Interpretation: For a system of linear equations, the matrix $\mathbf{A}$'s condition number relates small variations in the matrix $\mathbf{A}$ and vector $\boldsymbol{b}$ to the resulting changes in the solution $\boldsymbol{x}$.

2.1.3 Upper and Lower Triangular Systems

- A *lower triangular system* is a matrix equation of the form

$$\mathbf{L}\boldsymbol{x} = \boldsymbol{b}$$

where $\mathbf{L}$ is a $n \times n$ non-singular lower-triangular matrix with elements l_{ij}.

An *upper triangular system* is a matrix equation of the form

$$\mathbf{U}\boldsymbol{x} = \boldsymbol{b}$$

where $\mathbf{U}$ is a $n \times n$ non-singular upper-triangular matrix with elements u_{ij}.

- Lower triangular system are solved with the method of *forward substitution*, which takes advantage of the matrix $\mathbf{L}$'s lower triangular shape. To solve the system $\mathbf{L}\boldsymbol{x} = \boldsymbol{b}$, working from $\mathbf{L}$'s top row to bottom, we find the ith component x_i with the algorithm:

$$x_i = \frac{1}{\mathrm{L}_{ii}} \left(b_i - \sum_{j=1}^{i-1} \mathrm{L}_{ij}\, x_j \right)$$

  The computational complexity is $\mathcal{O}(n^2)$.

- Upper triangular system are solved with the method of *back substitution*, which takes advantage of the matrix $\mathbf{U}$'s upper triangular shape. To solve the system $\mathbf{U}\boldsymbol{x} = \boldsymbol{b}$, working from $\mathbf{U}$'s bottom row to top, we find the ith component x_i with the algorithm:

$$x_i = \frac{1}{\mathrm{U}_{ii}} \left(b_i - \sum_{j=i+1}^{n} \mathrm{U}_{ij}\, x_j \right)$$

  The computational complexity is $\mathcal{O}(n^2)$

2.2 LU Decomposition

LU decomposition is the process of decomposing an arbitrary matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$ into a product of a lower triangular and upper triangular matrix; these are easier to work with than the original matrix. Although in principle LU decomposition works with arbitrary matrices, this document considers only square matrices.

2.2.1 LU Decomposition without Pivoting

- A matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$ has an *LU decomposition* if it can be written in the form $\mathbf{A} = \mathbf{LU}$ where $\mathbf{L}$ and $\mathbf{U}$ are lower and upper-triangular matrices, respectively.

- A square matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$ has a unique LU decomposition, if and only if, all of its leading principle minors are nonzero. In symbols:

$$\mathbf{A} = \mathbf{LU} \iff \det \mathrm{sub}_{ii}[\mathbf{A}] \neq 0 \quad \text{for all } i = 1, 2, \ldots, n$$

  Interpretation: The condition ensures no zeros occur along $\mathbf{A}$'s main diagonal at any step in the decomposition process. An equivalent condition to non-zero leading principle minors is strict diagonal dominance.

- Assuming the LU decomposition exists, it is found using *LU decomposition without pivoting* algorithm.

  Algorithm: For a matrix $\mathbf{A} \in \mathbf{R}^{n\times n}$ with an LU decomposition, let $\mathbf{A}^{(j)}$ be the matrix $\mathbf{A}$ on the algorithm's jth iteration. Start with $j = 1$ and $\mathbf{A}^{(j)} = \mathbf{A}$.

  1. Perform row subtraction on the rows $j + 1$ to n of $\mathbf{A}^{(j)}$ to make the jth column have zeros in the rows $j + 1$ to n. *Record the row subtraction coefficient in the matrix* $\mathbf{L}$ *by writing each row subtraction coefficient in corresponding row of the* j*th column.*

The coefficients are α_k satisfying $\mathbf{A}^{(j)}_{k,j} - \alpha \mathbf{A}^{(j)}_{j,j} = 0$ where $k \in [j+1, n]$. In human language: perform row subtraction but only on the principle sub-matrix of size $(n-j+1) \times (n-j+1)$. On the first iteration $j = 1$ this is the full $n \times n$ matrix, on the second iteration $j = 2$, the principle sub-matrix of size $(n-1) \times (n-1)$ and so on. The row subtraction coefficient is the number you multiplied the top row by to make the target row vanish.

2. Increment j by one and repeat the algorithm, using the matrix from the previous step as $\mathbf{A}^{(j+1)}$.

To finish: The matrix $\mathbf{L}$ has ones on its main diagonal, and its lower non-zero elements are the row-subtraction coefficients recorded in the last iteration $\mathbf{A}^{(n-1)}$. The matrix $\mathbf{U}$ has upper non-zero elements matching the upper non-zero elements of $\mathbf{A}^{(n-1)}$. The algorithm's time cost is $\mathcal{O}(n)$.

2.2.2 LU Decomposition with Partial Pivoting

- An LUP decomposition is a generalization of the LU decomposition in which a permutation matrix $\mathbf{P}$ that re-arranges $\mathbf{A}$'s rows so that $\mathbf{PA}$ has a unique LU decomposition.

- Theorem: For all non-singular matrices $\mathbf{A} \in \mathbb{R}^{n \times n}$, there exists a permutation matrix $\mathbf{P} \in \mathbb{R}^{n \times n}$ such that $\mathbf{PA}$ has a unique LU decomposition. The permutation matrix is not unique, and many $\mathbf{P}$ may exist for a given $\mathbf{A}$; however, for each $\mathbf{P}$, the matrix $\mathbf{PA}$ has a unique LU decomposition.

  Interpretation: The permutation matrix $\mathbf{P}$ just represents the way to rearrange $\mathbf{A}$'s rows in such a way that the rearranged $\mathbf{A}$ has an LU decomposition.

- **Algorithm**: Let $\mathbf{A}^{(j)}$ be the matrix $\mathbf{A}$ on the jth iteration of the algorithm.

  Starting with $j = 1$, $\mathbf{A}^{(j)} = \mathbf{A}$ and $\mathbf{P} = \mathbf{I}$:

  1. Find the largest element by absolute value in column j from rows j to n, i.e. $\max\limits_{j \le i \le n} \left| a^{(j)}_{ij} \right|$. If the largest element occurs in multiple rows, use the row closest to the top.
  2. Switch the row containing the maximum element with the jth row. *Record the row switch in the permutation matrix* $\mathbf{P}$ by switching the corresponding rows in $\mathbf{P}$.
  3. Perform row subtraction on the rows $j+1$ to n of the switched-row matrix to make the jth column have zeros in the rows $j+1$ to n. *Record the row subtraction coefficient in the matrix* $\mathbf{L}$. (By convention write the coefficients in the jth column in the corresponding rows $j+1$ to n).
     - Coefficients are elements in row $j+1, \ldots, n$ divided by element in row j.
  4. Increment j by one and repeat the algorithm, using the matrix from the previous step as $\mathbf{A}^{(j+1)}$.

2.3 Applications of the LU Decomposition

- *Solving Systems of Linear Equations*: To solve the system $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$

  1. Find $\mathbf{A}$'s LU decomposition
  2. Write the system as

  $$\mathbf{A} = \mathbf{LU} \quad \implies \quad (\mathbf{LU})\boldsymbol{x} = \boldsymbol{b} \quad \implies \quad \mathbf{L}(\mathbf{U}\boldsymbol{x}) = \boldsymbol{b}$$

  This step uses $\mathbf{A}$'s LU decomposition and the associativity of matrix multiplication.

  3. Let $\boldsymbol{y} \coloneqq \mathbf{U}\boldsymbol{x}$ and solve the lower-triangular system $\mathbf{L}\boldsymbol{y} = \boldsymbol{b}$ with direct substitution to find $\boldsymbol{y}$.
  4. With the known value of $\boldsymbol{y}$, solve the upper-triangular system $\mathbf{U}\boldsymbol{x} = \boldsymbol{y}$ with backward substitution to find $\boldsymbol{x}$.

- *Solving Matrix Equations*: To find $\mathbf{X}$ solving $\mathbf{AX} = \mathbf{B}$ where $\mathbf{A} \in \mathbb{R}^{n\times n}$, $\mathbf{X}, \mathbf{B} \in \mathbb{R}^{n\times p}$.

  1. Find $\mathbf{A}$'s LU decomposition
  2. Write the system in column form, i.e. $\mathbf{A}[\boldsymbol{x}_1, \boldsymbol{x}_2, \dots, \boldsymbol{x}_n] = [\boldsymbol{b}_1, \boldsymbol{b}_2, \dots, \boldsymbol{b}_n]$.
  3. Solve the p vector systems $\mathbf{A}\boldsymbol{x}_j = \boldsymbol{b}_j, j = 1, 2, \dots, p$ using the technique for solving vector systems in the previous section.

- *To Find a Matrix's Inverse*: To find the inverse of the matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$, solve the matrix equation $\mathbf{AX} = \mathbf{I}$ where $\mathbf{X} = \mathbf{A}^{-1}$.

- *Calculating Determinants* The LU decomposition of $\mathbf{A}$ with partial pivoting leads to the formula

  $$\det \mathbf{A} = (-1)^s \det \mathbf{U}$$

  where s is the number of row switches in the permutation matrix $\mathbf{P}$.

  Derivation: Applying the multiplicative nature of the determinant to the *LU* decomposition leads to

  $$\mathbf{PA} = \mathbf{LU} \quad \implies \quad \det \mathbf{P} \det \mathbf{A} = \det \mathbf{L} \det \mathbf{U}$$

  The determinant of a diagonal matrix is the product of the diagonal terms and the matrix $\mathbf{L}$ has ones along its diagonal, so $\det \mathbf{L} = 1$. The determinant of $\mathbf{P}$ starts as the determinant of the identity matrix $\mathbf{I}$ and is multiplied by -1 for every row switch. Setting $\det \mathbf{L} = 1, \det \mathbf{P} = (-1)^s$ and dividing by $\det \mathbf{P}$ produces the above determinant formula.

2.4 Methods for Special Linear Systems

2.4.1 The Cholesky Decomposition

- Symmetric, positive definite matrices have a special decomposition called the *Cholesky decomposition* of the form

  $$\mathbf{A} = \mathbf{L}\mathbf{L}^T$$

where $\mathbf{L}$ is a non-singular lower triangular matrix.

The Cholesky decomposition is an efficient way to solve the system $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$ if $\mathbf{A} \in \mathbb{R}^{n\times n}$ is a symmetric, positive definite matrix. The resulting system $\mathbf{L}\mathbf{L}^T\boldsymbol{x} = \boldsymbol{b}$ is solved analogously to $\mathbf{L}\mathbf{U}\boldsymbol{x} = \boldsymbol{b}$

- *Algorithm*: For $k = 1$ initialize $\mathbf{L}_1 = \sqrt{\mathrm{A}_{11}}$. For $k = 2, \dots, n$, solve $\mathbf{L}_{k-1}\boldsymbol{l}_k = \boldsymbol{a}_k$ for $\boldsymbol{l}_k$. Solve $\mathrm{L_{kk}} = \sqrt{\mathrm{A}_{kk} - \boldsymbol{l}_k^T\boldsymbol{l}_k}$. Assemble $\mathbf{L}_k = \begin{bmatrix} \mathbf{L}_{k-1} & \mathbf{0}_k \\ \boldsymbol{l}_k^T & \mathrm{L}_{kk} \end{bmatrix} \in \mathbb{R}^{k\times k}, \mathbf{0}_k \in \mathbb{R}^k$. Finish with $\mathbf{L} = \mathbf{L}_n$.
  Notation: Matrix $\mathbf{L}_k$: $k \times k$ principle sub-matrix of $\mathbf{L}$. Vectors $\boldsymbol{a}_k, \boldsymbol{l}_k$: first $k-1$ entries in column k of $\mathbf{A}$ and $\mathbf{L}^T$.

  The time cost is $\frac{1}{3}n^3 + \mathcal{O}(n^2)$. Calculated with

$$\sum_{k=1}^{n}(2(k-1) + (n-k)(2k-1)) = \frac{1}{3}n^3 + \mathcal{O}(n^2)$$

- *Verifying positive definiteness* To efficiently determine if a matrix is positive definite: Perform Cholesky decomposition on the matrix; if the process produces square roots of negative numbers, the matrix is not positive definite. Otherwise it is.

2.4.2 Tridiagonal Systems

- A *tridiagonal matrix* has nonzero elements along its main diagonal and on the diagonals immediately above and below the main diagonal; it is zero elsewhere. It looks like this:

$$\mathbf{T} = \begin{bmatrix} a_1 & b_1 & & & \\ b_1 & a_1 & b_2 & & \\ & \ddots & \ddots & \ddots & \\ & & b_{n-2} & a_{n-1} & b_{n-1} \\ & & & b_{n-1} & a_n \end{bmatrix}$$

- *Theorem:* If a tridiagonal matrix $\mathbf{T} \in \mathbb{R}^{n\times n}$ is diagonally dominant (and strictly diagonally column dominant in at least one column) is has a unique LU decomposition, which can be found without pivoting. In this case, $\mathbf{L}$ and $\mathbf{U}$ take the form

$$\mathbf{L} = \begin{bmatrix} 1 & & & \\ l_1 & 1 & & \\ & \ddots & \ddots & \\ & & l_{n-1} & 1 \end{bmatrix} \qquad \mathbf{U} = \begin{bmatrix} u_1 & b_1 & & \\ & \ddots & \ddots & \\ & & u_{n-1} & b_{n-1} \\ & & & u_n \end{bmatrix}$$

- The computational complexity of solving a tridiagonal system without pivoting is only $\mathcal{O}(n)$.

3 Non-Linear Equations

Solving an equation can be interpreted as, given a function $f : \mathbb{R} \to \mathbb{R}$, finding the values of x satisfying

$$f(x) = 0$$

The values of x are called the roots or zeros of the function f.

3.1 Bisection

- First the theoretical basis: If a continuous function $f : [a, b] \to \mathbb{R}$ has opposite signs at its endpoints a and b, the function has at least on zero on $[a, b]$. In symbols, if $f(a) \cdot f(b) < 0$, there exists at least on $x \in [a, b]$ for which $f(x) = 0$.

  Interpretation: if f is continuous and changes sign on the interval $[a, b]$, it must have crossed through zero at least once, and thus has at least one zero.

- The *bisection method* is a reliable but relatively slow method for finding the roots of any continuous function $f : [a, b] \subset \mathbb{R} \to \mathbb{R}$.

- To find zeros of a continuous function $f : [a, b] \to \mathbb{R}$ on the interval $[a, b]$, the algorithm uses a variable left endpoint α, variable right endpoint β, a midpoint c, and tolerance ϵ for the interval width. Starting with $\alpha = a, \beta = b$,

  1. Calculate the midpoint with $c = \alpha + \frac{\beta - \alpha}{2}$
  2. Contract the interval width. If $\operatorname{sign} f(\alpha) = \operatorname{sign} f(c)$, let $\alpha = c$; if $\operatorname{sign} f(\alpha) \neq \operatorname{sign} f(c)$, let $\beta = c$.

  Repeat until $|\beta - \alpha| \leq \epsilon$, then return root $x_0 = \alpha + \frac{\beta - \alpha}{2}$.

- Some notes

  - The algorithm doesn't evaluate exact function values but only checks the function's sign, which is faster than calculating the exact function value. $c = \alpha + \frac{\beta - \alpha}{2}$ is used to find midpoint instead of $c = \frac{\alpha + \beta}{2}$ to reduce risk of overflow.
  - If f has multiple zeros on $[a, b]$, the algorithm behaves unpredictably. It will converge to one of the zeros, but we cannot predetermine which one.
  - The interval $[\alpha, \beta]$ halves in width during each iteration. After k iterations, the interval has width $\ell = \frac{b-a}{2^k}$

    A tolerance ϵ for interval width requires at least $k \geq \log_2\left(\frac{b-a}{\epsilon}\right)$ iterations to converge.

3.2 Fixed-Point Iteration

- Start with the equation $f(x) = 0$ where $f : [a, b] \to \mathbb{R}$ and manipulate this expression into the form $x = g(x)$. In other words, isolate the linear term x and put all other forms of x on the other side of the equation. The function g is called the *iteration function*.

For a given equation $f(x) = 0$, there are in general multiple ways to express $x = g(x)$; the iteration function $g(x)$ is not unique. For example, for the polynomial root problem $f(x) = x^4 - 2x^3 + 3x = 0$, possible iteration functions are

$$x_a = \frac{2x^3 - x^4}{3} \qquad x_b = \sqrt[3]{\frac{x^4 + 3x}{2}} \qquad x_c = \sqrt[4]{2x^3 - 3x}$$

- The algorithm is simple. Choose an initial guess $x_0 \in [a, b]$ and tolerance ϵ, then iterate with the formula $x_{i+1} = g(x_i)$ until the x_{i+1} and x_i differ by less than ϵ.

3.2.1 Some Theory on Convergence of Fixed Point Iteration

- A *fixed point* of a function is an element of the function's domain that is mapped to itself, i.e. α is a fixed point of the function g if $g(\alpha) = \alpha$.

- Let α be the fixed point and let $I = [\alpha - \delta, \alpha + \delta]$ be an interval centered at α. If the iterative function g satisfies the Lipschitz continuity condition on I; i.e there exists real number $\theta \in [0, 1)$ such that

  $$|g(x) - g(y)| \leq \theta |x - y| \quad \text{for all } x, y, \in I$$

  then the iterative sequence $x_{i+1} = g(x_i)$ converges to α for all initial $x_0 \in I$.

  More so, we have the following bounds:

  $$|x_j - \alpha| \leq \theta^j |x_0 - \alpha| \qquad |x_{j+1} - \alpha| \leq \frac{\theta}{1 - \theta} |x_j - x_{j+1}|$$

- Some theory: consider the function $g : [a, b] \to \mathbb{R}$. The point $\alpha \in [a, b]$ is a *fixed point* of g if for all x, y in the interval $I = [\alpha - \delta, \alpha + \delta]$ centered at α

  $$|g(x) - g(y)| \leq m|x - y|, \qquad m \in [0, 1)$$

  In this case, for all initial values $x_0 \in I$ and $k = 0, 1, 2, \ldots$

  - The sequence $x_{k+1} = g(x_k)$ converges to α
  - $|x_k - \alpha| \leq m^k |x_0 - \alpha|$
  - $|x_{k+1} - \alpha| \leq \frac{m}{1-m} |x_j - x_{j-1}|$

- If α is a fixed point of $g : [a, b] \to \mathbb{R}$ and g is continuously differentiable at α and the derivative is bounded by $|g'(\alpha)| < 1$, there exists a neighborhood $I = [\alpha - \delta, \alpha + \delta]$ of the point α on which the sequence $x_{k+1} = g(x_k)$ converges to α for all initial values $x_0 \in I$ and $k = 0, 1, 2, \ldots$

  Interpretation: In practice, this is just an alternate, more practical condition for convergence of an iteration function than the Lipschitz continuity condition. It's usually easier to evaluate the first derivative than to prove Lipschitz continuity.

- **Rate of Convergence** Let the iterative function g be p-times continuously differentiable at the fixed point α and let

$$g^{(p)}(\alpha) = 0 \quad \text{for } j = 1, 2, \dots, p-1$$

and $g^{(p)}(\alpha) \neq 0$. In this case, the iterative sequence $x_{j+1} = g(x_j)$ has order of convergence p in a neighborhood of α.

Basically the order of continuous differentiability determines the order of convergence.

- **Definition of order of convergence:** Let the sequence $\left(x_j\right)_{j=0}^{\infty}$ converge to the limit L. The sequence's order of convergence is p if there exists finite constant M such that

$$\lim_{j\to\infty} \frac{|x_{j+1} - L|}{|x_j - L|^p} < M$$

3.2.2 Newton's Method

- Newton's method is an example of fixed point iteration that uses the tangent line to find the next point. The next approximation is the intersection of the tangent line at the previous approximation with the x axis.

- *Algorithm:* To find a zero of the continuously differentiable function $f : [a, b] \to \mathbb{R}$ choose a tolerance ϵ and initial guess x_0. Calculate successive approximations with $x_{j+1} = x_j - \frac{f(x_j)}{f'(x_j)}$ and repeat until $|x_{j+1} - x_j| < \epsilon$.

Note that Newton's method is a form of fixed-point iteration with the iteration function $g(x) = x - \frac{f(x)}{f'(x)}$

- *Convergence*: Newton's method converges converges at least quadratically to any simple zero x_0, i.e. a zero for which $f'(x_0) \neq 0$. Convergence is quadratic if $f''(x_0) \neq 0$ and at least cubic if $f''(x_0) = 0$.

For higher-order zeros, i.e. zeros for which $f'(x_0) = 0$, Newton's method converges at least linearly.

- Theory: If α is a simple zero of the twice-continuously differentiable function $f : [a, b] \to \mathbb{R}$, there exists a constant $C \in \mathbb{R}$ and neighborhood $I = [\alpha-\delta, \alpha+\delta]$ of the zero α such that Newton's method converges for all $x_0 \in I$ and the approximations x_i satisfy the bound

$$|x_{j+1} = \alpha| \leq C(x_i - \alpha)^2$$

- Sort of interesting note: If f is a twice-continuously differentiable, increasing, convex function on the interval $I = [a, \infty)$ with a zero $\alpha \in I$, then Newton's method converges to α for all initial guesses $x_0 \in I$.

Interpretation: For example, a parabola or an exponential function satisfy these criterion. The conditions are in my opinion too strict to be useful, but the theorem is interesting because it guarantees convergence for all values in an infinitely large interval.

3.2.3 Secant Method

- The second method is a finite-difference approximation to Newton's method. Newton's method requires the derivative f', which is not always known. When the derivative f' of a function is not known, the secant method replaces Newton's method.
- The secant method is a form of fixed-point iteration with the iteration function

$$x_{j+1} = x_j - f(x_j)\frac{x_j - x_{j-1}}{f(x_j) - f(x_{j-1})}$$

  The algorithm is analogous to Newton's method and other fixed-point methods.
- The downside to the secant method is the need to store two previous approximations x_j and x_{j-1} in memory.
- The rate of convergence is super-linear with $p \approx 1.62$.

3.3 Roots of Polynomial Equations

- The general problem is to find all roots of an nth degree polynomial

$$p(x) = a_n x^n + \cdots + a_1 x + a_0$$

- Although any of the methods listed above work for polynomial equations, there are more efficient methods designed specifically for polynomials.
- All polynomial root methods can implement *deflation*. This means that, once you find a zero x_k of p, you divide p by $(x - x_k)$ and find the zeros of the resulting polynomial. This way you don't re-find a zero you already found and you reduce the complexity of the polynomial you are working with. In general, it is best do deflate in order of increasing magnitude of the zeros.

3.3.1 Companion Matrix

The roots of the polynomial equation $p(x) = a_n x^n + \cdots + a_1 x + a_0$ are the eigenvalues of the associated $(n \times n)$ *companion matrix*

$$\mathbf{A}_n = \begin{bmatrix} 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \\ -\frac{a_0}{a_n} & -\frac{a_1}{a_n} & -\frac{a_2}{a_n} & \cdots & -\frac{a_{n-1}}{a_n} \end{bmatrix}$$

The zeros of p are the eigenvalues $\text{eig}(\mathbf{A}_p)$.

3.3.2 Laguerre Method

To find the roots of the polynomial equation $p(x) = a_n x^n + \cdots + a_1 x + a_0$, make an initial guess x_0 and choose a tolerance ϵ. For $j = 0, 1, 2, \ldots$

1. Calculate $G = \frac{p'(x_j)}{p(x_j)}$ and $H = G^2 - \frac{p''(x_j)}{p(x_j)}$
2. Calculate the approximation $\alpha = \frac{n}{G \pm \sqrt{(n-1)(nH - G^2)}}$ choosing the $\pm$ sign so the denominator has maximum absolute value.
3. Repeat the iteration with $x_{j+1} = x_j - a$

Repeat until $\alpha \leq \epsilon$; the root is x_j. The Laguerre method converges cubically in the neighborhood of a simple zero.

3.4 Systems of Non-Linear Equations

Given a system of non-linear functions $\boldsymbol{F} = (f_1, f_2, \ldots, f_n)$ written as a vector function $\boldsymbol{F} : \mathbb{R}^n \to \mathbb{R}^n$, find the vectors $\boldsymbol{x} \in \mathbb{R}^n$ satisfying the vector equation:

$$\boldsymbol{F}(\boldsymbol{x}) = \boldsymbol{0}$$

Note that this can be written in the form

$$\begin{aligned} f_1(x_1, x_2, \ldots, x_n) &= 0 \\ f_2(x_1, x_2, \ldots, x_n) &= 0 \\ &\vdots \\ f_n(x_1, x_2, \ldots, x_n) &= 0 \end{aligned}$$

In this case $\boldsymbol{F} = (f_1, f_2, \ldots, f_n)^T$ and $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)^T$
These methods are mostly beyond the scope of this course; we take only a brief look at fixed-point iteration.

3.4.1 Fixed-Point Iteration

- Just a higher-dimensional analog of fixed-point iteration for higher dimensions in which the Jacobian determinant replaces the role of the derivative.
- Manipulate $\boldsymbol{F}(\boldsymbol{x}) = 0$ into the form $\boldsymbol{x} = \boldsymbol{G}(\boldsymbol{x})$ and use fixed point iteration of the form $\boldsymbol{x}_{j+1} = \boldsymbol{G}(\boldsymbol{x}_j)$
- The Jacobi matrix $\mathbf{J}_{\mathbf{G}}(\boldsymbol{x})$ of the function $\boldsymbol{G}$ at the point $\boldsymbol{x} \in \mathbb{R}^n$ is

$$\mathbf{J}_{\mathbf{G}}(\boldsymbol{x}) = \begin{bmatrix} \frac{\partial g_1}{\partial x_1}(\boldsymbol{x}) & \cdots & \frac{\partial g_1}{\partial x_n}(\boldsymbol{x}) \\ \vdots & \ddots & \vdots \\ \frac{\partial g_n}{\partial x_1}(\boldsymbol{x}) & \cdots & \frac{\partial g_n}{\partial x_n}(\boldsymbol{x}) \end{bmatrix}$$

  The spectral radius $\rho(\mathbf{A})$ of a matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ is $\mathbf{A}$'s largest eigenvalue by absolute value.

- On convergence: If there exists a region $\Omega \subset \mathbb{R}^n$ for which $\boldsymbol{G}(\boldsymbol{x}) \in \Omega$ and $\rho(\mathbf{J}_{\mathbf{G}}(\boldsymbol{x})) < 1$ for all $\boldsymbol{x} \in \Omega$, the sequence $\boldsymbol{x}_{j+1} = \boldsymbol{G}(\boldsymbol{x}_j)$ converges to a unique solution $\boldsymbol{\alpha} \in \mathbb{R}^n$ for all initial guesses $\boldsymbol{x}_0 \in \Omega$

- Implication for convergence: The sequence $\boldsymbol{x}_{j+1} = \boldsymbol{G}(\boldsymbol{x}_j)$ converges to a unique solution $\boldsymbol{\alpha} \in \mathbb{R}^n$ for all initial guesses $\boldsymbol{x}_0 \in \Omega$ if $\boldsymbol{G}(\boldsymbol{x}) \in \Omega$ for all $\boldsymbol{x} \in \Omega$ and for $m < 1$ and $k = 1, 2, \dots, n$

$$\sum_{j=1}^{n} \left| \frac{\partial g_k}{\partial x_j}(\boldsymbol{x}) \right| \leq m$$

  In this case $\|\boldsymbol{x}_j - \boldsymbol{\alpha}\|_\infty \leq \frac{m^j}{1-m} \|\boldsymbol{x}_1 - \boldsymbol{x}_0\|_\infty$ for $j = 0, 1, 2, \dots$

3.4.2 Newton's Method

- Analogous to Newton's method in one dimension, with the Jacobi matrix replacing the derivative. The iteration function is

$$\boldsymbol{G}(\boldsymbol{x}) = \boldsymbol{x} - \mathbf{J}_F(\boldsymbol{x})^{-1} \boldsymbol{F}(\boldsymbol{x})$$

  And for $j = 0, 1, 2, \dots$, the iteration reads

$$\boldsymbol{x}_{j+1} = \boldsymbol{x}_j - \mathbf{J}_F(\boldsymbol{x}_j)^{-1} \boldsymbol{F}(\boldsymbol{x}_j)$$

- In practice, instead of calculating the inverse $\mathbf{J}_F(\boldsymbol{x}_j)^{-1}$, we solve the system

$$\mathbf{J}_F(\boldsymbol{x}_j)(\boldsymbol{x}_{j+1} - \boldsymbol{x}_j) = -F(\boldsymbol{x}_j)$$

- Formal conditions for convergence of the higher-dimensional Newton's method exist, but are hard to verify in practice. Instead, we settle for a good initial guess and usually things work out.

4 Linear Least Squares

Given many data points and a model function described by a few parameters (much fewer than the number of data points), find the values of the parameters for which the function best models the data points. Our actors are:

- m data points $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_m)$.
- n parameters $a_1, a_2, \dots, a_n$
- A model function $f(x, a_1, a_2, \dots, a_n)$

Some notes

- The linear least squares problem can be written in matrix form as follows: Given a matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ and a vector of data points $\boldsymbol{b} \in \mathbb{R}^m$, find the vector of parameters $\boldsymbol{x} \in \mathbb{R}^n$ minimizing the expression $\|\mathbf{A}\boldsymbol{x} - \boldsymbol{b}\|_2$. The entries A_{ij} of the matrix are the coefficients of the jth parameter for the ith data point, $i = 1, \dots, m$ and $j = 1, \dots, n$.
- In this formulation, the goal is to find the vector $\boldsymbol{x}$ minimizing the expression

  $$\|\boldsymbol{b} - \mathbf{A}\boldsymbol{x}\|_2^2 = (\boldsymbol{b} - \mathbf{A}\boldsymbol{x})^T \cdot (\boldsymbol{b} - \mathbf{A}\boldsymbol{x}) = (b_1 - \mathbf{A}_1^T\boldsymbol{x})^2 + \cdots + (b_m - \mathbf{A}_m^T\boldsymbol{x})^2$$

  where $\mathbf{A}_m^T$ is the matrix $\mathbf{A}$'s mth row, or equivalently the matrix $\mathbf{A}^T$'s mth column.
- Such a system is called an *overdetermined system* because there are more (often many more!) equations than unknowns. Each data point produces an equation, while the number of unknowns (number of parameters) is usually small in comparison.
- We assume in this text $\operatorname{rank}\mathbf{A} = n$; i.e. $\mathbf{A}$ has full rank. If $\mathbf{A}$ does not have full rank, the solution to the system is not unique.

4.1 Normal System

- As system of equations of the form $\mathbf{A}^T\mathbf{A}\boldsymbol{x} = \mathbf{A}^T\boldsymbol{b}$ is called a *normal system* of equations. The normal system arises naturally in the linear least squares problem, where $\boldsymbol{b} \in \mathbb{R}^m$ is the vector of data points, $\boldsymbol{x} \in \mathbb{R}^n$ is the vector of parameters, and $\mathbf{A} \in \mathbb{R}^{m \times n}$ is the matrix of parameter coefficients.

  The system is called normal because it specifies that the residual $\boldsymbol{x}$ must be normal (orthogonal) to every vector in $\operatorname{span}\mathbf{A}$.
- Solving the normal system of equations for the parameter vector $\boldsymbol{x}$ is equivalent to solving by the method of least squares; the parameters $\boldsymbol{x}$ are those minimizing the sum of the squared residuals.
- The stuff goes like this: define $f(\boldsymbol{x}) = \|\boldsymbol{b} - \mathbf{A}\boldsymbol{x}\|_2^2 = (\boldsymbol{b} - \mathbf{A}\boldsymbol{x})^T \cdot (\boldsymbol{b} - \mathbf{A}\boldsymbol{x})$, which we want to minimize. It turns out that

  $$\nabla f(\boldsymbol{x}) = 2\mathbf{A}^T\mathbf{A}\boldsymbol{x} - 2\mathbf{A}^T\boldsymbol{b}$$

So solving $\nabla f(\boldsymbol{x}) = \mathbf{0}$ is equivalent to solving the normal system $\mathbf{A}^T\mathbf{A}\boldsymbol{x} = \mathbf{A}^T\boldsymbol{b}$.

In general, $\boldsymbol{x}$ could be a minimum, maximum, or saddle point when $\nabla f(\boldsymbol{x}) = 0$. However, $\mathbf{A}^T\mathbf{A}$ is positive semi-definite which implies that $\boldsymbol{x}$ solving $\nabla f(\boldsymbol{x}) = 0$ is a global minimum of the function $f(\boldsymbol{x})$.

- Since $\mathbf{A}^T\mathbf{A}$ is positive semi-definite, it can be decomposed into the form $\mathbf{L}\mathbf{L}^T$ with the Cholesky decomposition. The Cholesky decomposition is the most efficient way to solve the normal system. However, the method becomes unstable if $\mathbf{A}^T\mathbf{A}$'s columns are close to being linearly dependent.

4.2 QR Decomposition

- The *QR decomposition* of a matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$ is $\mathbf{A} = \mathbf{QR}$ where $\mathbf{Q} \in \mathbb{R}^{m\times n}$ has orthonormal columns and $\mathbf{R} \in \mathbb{R}^{n\times n}$ is upper triangular.

- The QR decomposition can be used to solve the linear least squares problem. If the parameter coefficient matrix has the QR decomposition $\mathbf{A} = \mathbf{QR}$, then the vector of parameters $\boldsymbol{x}$ is

$$\boldsymbol{x} = \left(\mathbf{A}^T\mathbf{A}\right)^{-1}\mathbf{A}^T\mathbf{b} = \mathbf{R}^{-1}\mathbf{Q}^T\boldsymbol{b}$$

  The proof is a direct application of $\mathbf{Q}$'s orthogonality, which implies $\mathbf{Q}^T\mathbf{Q} = \mathbf{I}$.

$$\begin{aligned}\left(\mathbf{A}^T\mathbf{A}\right)^{-1}\mathbf{A}^T &= \left((\mathbf{QR})^T(\mathbf{QR})\right)^{-1}(\mathbf{QR})^T = \left(\mathbf{R}^T\mathbf{Q}^T\mathbf{QR}\right)^{-1}\mathbf{R}^T\mathbf{Q}^T \\ &= \left(\mathbf{R}^T\mathbf{IR}\right)^{-1}\mathbf{R}^T\mathbf{Q}^T = \left(\mathbf{R}^T\right)^{-1}\mathbf{R}^{-1}\mathbf{R}^T\mathbf{Q}^T \\ &= \mathbf{R}^{-1}\left(\mathbf{R}^T\right)^{-1}\mathbf{R}^T\mathbf{Q}^T = \mathbf{R}^{-1}\mathbf{I}\mathbf{Q}^T = \mathbf{R}^{-1}\mathbf{Q}^T\end{aligned}$$

  It follows that $\boldsymbol{x} = \left(\mathbf{A}^T\mathbf{A}\right)^{-1}\mathbf{A}^T\mathbf{b} = \mathbf{R}^{-1}\mathbf{Q}^T\boldsymbol{b}$.

- If we known the QR decomposition $\mathbf{A} = \mathbf{QR}$, we can solve the least squares problem by solving the upper triangular system

$$\mathbf{R}\boldsymbol{x} = \mathbf{Q}^T\boldsymbol{b}$$

  Solving a least squares by QR decomposition is more stable than directly solving the corresponding normal system.

- Theorem: Every matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$ with $\operatorname{rank}\mathbf{A} = n$ (where $m \geq n$) has a unique QR decomposition $\mathbf{A} = \mathbf{QR}$ where $\mathbf{Q} \in \mathbb{R}^{m\times n}$ has orthonormal columns and $\mathbf{R} \in \mathbb{R}^{n\times n}$ is upper triangular.

4.3 Methods of QR Decomposition

4.3.1 Gram-Schmidt Orthonormalization

The Gram-Schmidt orthonormalization algorithm is used to convert a system of column vectors $\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_n$ into a normalized, mutually orthogonal set of vectors $(\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots, \boldsymbol{u}_n)$ where $\|\boldsymbol{u}_i\| = 1$ for $i = 1, \ldots, n$ and $\boldsymbol{u}_i \perp \boldsymbol{u}_j$ for all $i \neq j$.

Algorithm The Gram-Schmidt algorithm is recursive. To use it, one iterates over each basis vector and first normalizes it, then makes it orthogonal to every basis vector that came before it using the formula for an orthogonal projection. Given a set of vectors $\boldsymbol{v}_1, \boldsymbol{v}_2, \dots, \boldsymbol{v}_n$

1. Normalize the first vector $\boldsymbol{v}_1$ to get the normalized vector $\boldsymbol{u}_1 = \frac{\boldsymbol{v}_1}{\|\boldsymbol{v}_1\|}$
2. Orthogonalize the second vector $\boldsymbol{v}_2$ relative to $\boldsymbol{u}_1$ to get the vector $\boldsymbol{v}_2'$ using
$$\boldsymbol{v}_2' = \boldsymbol{v}_2 - \langle \boldsymbol{v}_2, \boldsymbol{u}_1 \rangle \, \boldsymbol{u}_1$$
Normalize $\boldsymbol{v}_2'$ to get the second orthonormalized vector $\boldsymbol{u}_2 = \frac{\boldsymbol{v}_2'}{\|\boldsymbol{v}_2'\|}$
3. Orthogonalize the third vector $\boldsymbol{v}_3$ relative to $\boldsymbol{u}_1$ and $\boldsymbol{u}_2$ to get the vector $\boldsymbol{v}_3'$ using
$$\boldsymbol{v}_3' = \boldsymbol{v}_3 - \langle \boldsymbol{v}_3, \boldsymbol{u}_2 \rangle \, \boldsymbol{u}_2 - \langle \boldsymbol{v}_2, \boldsymbol{u}_1 \rangle \, \boldsymbol{u}_1$$
Normalize $\boldsymbol{v}_3'$ to get the third orthonormalized vector $\boldsymbol{u}_3 = \frac{\boldsymbol{v}_3'}{\|\boldsymbol{v}_3'\|}$.
4. Continue recursively using the general formula
$$\boldsymbol{v}_k' = \boldsymbol{v}_k - \langle \boldsymbol{v}_k, \boldsymbol{u}_{k-1} \rangle \, \boldsymbol{u}_{k-1} - \cdots - \langle \boldsymbol{v}_k, \boldsymbol{u}_1 \rangle \, \boldsymbol{u}_1 \qquad \text{and} \qquad \boldsymbol{u}_k = \frac{\boldsymbol{v}_k'}{\|\boldsymbol{v}_k'\|}$$

4.3.2 Modified Gram-Schmidt

The modified Gram-Schmidt algorithm gives the same answer as the classic Gram-Schmidt algorithm when performed in exact arithmetic, but gives slightly more stable/accurate results in floating point arithmetic.
In the classic Gram-Schmidt algorithm, the kth orthonormal vector $\boldsymbol{u}_k$ is found with

$$\boldsymbol{u}_k = \boldsymbol{v}_k - \text{proj}_{\boldsymbol{u}_1}(\boldsymbol{v}_k) - \text{proj}_{\boldsymbol{u}_2}(\boldsymbol{v}_k) - \cdots - \text{proj}_{\boldsymbol{u}_{k-1}}(\boldsymbol{v}_k)$$

where $\text{proj}_{\boldsymbol{u}}(\boldsymbol{v}) = \frac{\langle \boldsymbol{u}, \boldsymbol{v} \rangle}{\langle \boldsymbol{u}, \boldsymbol{u} \rangle} \boldsymbol{u}$. In The modified version, the algorithm reads

$$\begin{aligned} \boldsymbol{u}_k^{(1)} &= \boldsymbol{v}_k - \text{proj}_{\boldsymbol{u}_1}(\boldsymbol{v}_k) \\ \boldsymbol{u}_k^{(2)} &= \boldsymbol{u}_k^{(1)} - \text{proj}_{\boldsymbol{u}_2}(\boldsymbol{u}_k^{(1)}) \\ &\vdots \\ \boldsymbol{u}_k^{(k-1)} &= \boldsymbol{u}_k^{(k-2)} - \text{proj}_{\boldsymbol{u}_{k-1}}(\boldsymbol{u}_k^{(k-2)}) \end{aligned}$$

and finishes with $\boldsymbol{u}_k = \frac{\boldsymbol{u}_k^{(k-1)}}{\|\boldsymbol{u}_k^{(k-1)}\|}$

4.3.3 Gram-Schmidt For QR Decomposition

Suppose you start with the matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ with column vectors $\boldsymbol{A} = (\boldsymbol{a}_1, \dots, \boldsymbol{a}_n)$. Use the Gram-Schmidt process to orthonormalize $(\boldsymbol{a}_1, \dots, \boldsymbol{a}_n)$ into $(\boldsymbol{q}_1, \dots, \boldsymbol{q}_n)$. In this case

$$\mathbf{Q} = [\boldsymbol{q}_1, \dots, \boldsymbol{q}_n] \quad \text{and} \quad \mathbf{R} = \begin{bmatrix} \langle \boldsymbol{e}_1, \boldsymbol{a}_1 \rangle & \langle \boldsymbol{e}_1, \boldsymbol{a}_2 \rangle & \langle \boldsymbol{e}_1, \boldsymbol{a}_3 \rangle & \cdots & \langle \boldsymbol{e}_1, \boldsymbol{a}_n \rangle \\ 0 & \langle \boldsymbol{e}_2, \boldsymbol{a}_2 \rangle & \langle \boldsymbol{e}_2, \boldsymbol{a}_3 \rangle & \cdots & \langle \boldsymbol{e}_2, \boldsymbol{a}_n \rangle \\ 0 & 0 & \langle \boldsymbol{e}_3, \boldsymbol{a}_3 \rangle & \cdots & \langle \boldsymbol{e}_3, \boldsymbol{a}_n \rangle \\ \vdots & \vdots & \cdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \langle \boldsymbol{e}_n, \boldsymbol{a}_n \rangle \end{bmatrix} \quad \text{where } \boldsymbol{e}_i = \frac{\boldsymbol{q_i}}{\|q_i\|}$$

4.3.4 Givens Rotations

Excellent link here.

- The rotation of the vector $\boldsymbol{x} \in \mathbb{R}^2$ about the origin by the angle ϕ in the counter-clockwise (positive) direction is encoded by the rotation matrix

$$\mathbf{R}_\phi = \begin{bmatrix} \cos\phi & -\sin\phi \\ \sin\phi & \cos\phi \end{bmatrix}$$

  and the rotated vector is $\boldsymbol{x}' = \mathbf{R}\boldsymbol{x}$.

- For rotation in a planar subspace of $\mathbf{R}^n$ spanned by the basis vectors $\boldsymbol{e}_i, \boldsymbol{e}_j$, the corresponding rotation matrix $\mathbf{R}_{ij}$ is the $n \times n$ identity matrix with the elements in the j and ith column of the jth row replaced by $\cos\phi$ and $-\sin\phi$ and the elements in the j and ith column of the ith row replaced by $\cos\phi$ and $\sin\phi$.

- A Given's rotation is a matrix of the form

$$\mathbf{G}_{ij}^T(\phi) = \begin{bmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & c & \cdots & -s & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & s & \cdots & c & \cdots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{bmatrix}^T$$

  where $c = \cos\phi$ and $s = \sin\phi$.

  For $\boldsymbol{x} \in \mathbb{R}^n$, the product $\mathbf{G}_{ij}(\phi)\boldsymbol{x}$ yields a counterclockwise rotation of the vector $\boldsymbol{x}$ by an angle of ϕ radians in the planar subspace spanned by the basis vectors $\boldsymbol{e}_i$ and $\boldsymbol{e}_j$.

- When a Given's rotation $\mathbf{G}_{ij}^T \in \mathbb{R}^{n\times n}$ multiplies another matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$ from the left, only the i and jth rows of $\mathbf{A}$ are affected.

- In two dimensions, for a vector $\begin{bmatrix} a \\ b \end{bmatrix}$, the goal is to find c and s such that

$$\begin{bmatrix} c & -s \\ s & c \end{bmatrix}^T \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}$$

  where $r = \sqrt{a^2 + b^2}$. The solution is $c = \frac{a}{r}$ and $s = \frac{b}{r}$

- In n dimensions:

$$\begin{bmatrix} 1 & & & & & & & & \\ & \ddots & & & & & & & \\ & & 1 & & & & & & \\ & & & c & & & -s & & \\ & & & & 1 & & & & \\ & & & & & \ddots & & & \\ & & & & & & 1 & & \\ & & & s & & & c & & \\ & & & & & & & 1 & \\ & & & & & & & & \ddots \\ & & & & & & & & & 1 \end{bmatrix}^T \begin{bmatrix} \times \\ \vdots \\ \times \\ a \\ \times \\ \vdots \\ \times \\ b \\ \times \\ \vdots \\ \times \end{bmatrix} = \begin{bmatrix} \times \\ \vdots \\ \times \\ r \\ \times \\ \vdots \\ \times \\ 0 \\ \times \\ \vdots \\ \times \end{bmatrix}$$

  So, to zero the element at the ith row and jth column (*assuming* $i > j$) of a matrix $\mathbf{A}$, create a Given's rotation matrix $\mathbf{G}_{ij}^T$ with the jth and ith column of the jth row replaced by $\cos\phi$ and $-\sin\phi$ and the elements in the jth and ith column of the ith row replaced by $\cos\phi$ and $\sin\phi$.

  The terms c and s are found using $c = \frac{a}{r}$ and $s = \frac{b}{r}$ where a is the element $\mathbf{A}$'s jth column in row j and b is the element $\mathbf{A}$'s jth column in row i.

- To use Given rotations to QR decompose the matrix $\mathbf{A}$, working from left to right column, use Givens rotations to create zeros in $\mathbf{A}$'s columns from the bottom row to the top. Note that the Givens matrix is always square, but the matrix $\mathbf{A}$ does not have to be.

- The upper-triangular matrix $\mathbf{R}$ is the result of applying all of the Givens rotations to $\mathbf{A}$ until $\mathbf{A}$ is upper-triangular. In general, $\mathbf{R}$ can be quasi-upper triangular.

  Meanwhile, $\mathbf{Q}$ is the product of all of the $\mathbf{G}_{ij}$ matrices (*not* the transposed $\mathbf{G}_{ij}^T$). The left-most term in the product should be the last givens matrix calculated, and the rightmost term should be the first Givens matrix (left column, bottom row).

4.3.5 Householder Reflections

Householder reflections are yet another way to find a matrix's QR decomposition, which is in turn, remember, used to solve a linear least squares problem.

- A *Householder reflection matrix* is a matrix of the form

$$\mathbf{P} = \mathbf{I} - \frac{2}{\boldsymbol{v}^T \cdot \boldsymbol{v}} \boldsymbol{v} \cdot \boldsymbol{v}^T$$

  where $\boldsymbol{w}$ is a non-zero vector in $\mathbb{R}^n$.

  The matrix $\mathbf{P}$ is symmetric and orthogonal ($\mathbf{P}\mathbf{P}^T = \mathbf{I}$) by construction.

- The product $\mathbf{P}\boldsymbol{x}$ represents a reflection of the vector $\boldsymbol{x} \in \mathbb{R}^n$ about the hyperplane normal to the vector $\boldsymbol{v}$ used to define $\mathbf{P}$.

- The general problem: Start with a vector $\boldsymbol{x} \in \mathbb{R}^n$ and construct a Householder reflection $\mathbf{P} \in \mathbb{R}^{m\times n}$ such that $\mathbf{P}\boldsymbol{x} = \alpha\boldsymbol{e}_1$. From

$$\|\mathbf{P}\boldsymbol{x}\|_2 = \|\boldsymbol{x}\|_2 \qquad \text{and} \qquad \|\alpha\boldsymbol{e}_1\|_2 = |\alpha|\|\boldsymbol{e}_1\|_2 = |\alpha|$$

  it follows that $|\alpha| = \|\boldsymbol{x}\|_2$.

Algorithm: To use Householder reflections to find the QR decomposition of the matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$, we construct on Householder reflection for each of $\mathbf{A}$'s columns.

1. Start with $\mathbf{A}^{(1)} = \mathbf{A}$ and find the vector $\boldsymbol{v}^{(1)}$ defining $\mathbf{P}^{(1)}$ such that $\mathbf{P}^{(1)}\boldsymbol{a}_1 = \alpha\boldsymbol{e}_1$ where $\boldsymbol{a}_1$ is $\mathbf{A}^{(1)}$'s first column. Use the formula

$$\boldsymbol{v}^{(1)} = \begin{bmatrix} \boldsymbol{a}_{1_1} + \operatorname{sgn}(\boldsymbol{a}_{1_1})\alpha \\ \boldsymbol{a}_{1_2} \\ \vdots \\ \boldsymbol{a}_{1_n} \end{bmatrix}$$

   Note that $\boldsymbol{a}_{1_1}$ and α are chosen to have the same sign to avoid cancellation i.e. to avoid $\boldsymbol{a}_{1_1} + \alpha \approx 0$.

   Apply $\mathbf{P}^{(1)}$ to $\mathbf{A}^{(1)}$ and get a matrix $\mathbf{A}^{(2)} = \mathbf{P}^{(1)}\mathbf{A}^{(1)}$ with an orthogonal first column.

2. Next, with the first column zeroed out, we turn our attention to $\mathbf{A}^{(2)}$'s first $(n-1)\times(n-1)$ principle sub-matrix. Find the next vector $\boldsymbol{v}^{(2)}$ so that the associated Householder reflection $\tilde{\mathbf{P}}^{(2)}$ turns the first column of the principle sub-matrix into $\alpha\boldsymbol{e}_1$ where $\boldsymbol{e}_1 \in \mathbb{R}^{n-1}$. Note that $\tilde{\mathbf{P}}^{(2)} \in \mathbb{R}^{n-1\times n-1}$ because the first sub-matrix is one dimension smaller! Then construct the $(n-1)\times(n-1)$ matrix $\mathbf{P}^{(2)} = \begin{bmatrix} 1 & 0 \\ 0 & \tilde{\mathbf{P}}_2 \end{bmatrix}$.

   Apply $\mathbf{P}^{(2)}$ to $\mathbf{A}^{(2)}$ and get a matrix $\mathbf{A}^{(3)} = \mathbf{P}^{(2)}\mathbf{A}^{(2)}$ with an upper-triangular first and second column.

3. Continue in the same manner along $\mathbf{A}^{(j)}$'s principle sub-matrices for $j = 1, \dots, n$. You should have to find n reflections for each of $\mathbf{A}$'s n columns.

   Should look something like this: for $j = 1, 2, \dots, n$, $\mathbf{P}^{(j)} = \begin{bmatrix} \mathbf{I}_{j-1} & 0 \\ 0 & \tilde{\mathbf{P}}_j \end{bmatrix}$ where $\mathbf{I}_{j-1}$ is the $(j-1)\times(j-1)$ identity matrix.

4. We then apply a new $\mathbf{P}$ to the next sub-matrix of $\mathbf{A}$ and repeat the process. We get:

$$\mathbf{P}_n \dots \mathbf{P}_2\mathbf{P}_1\mathbf{A} = \begin{bmatrix} \mathbf{R} \\ \mathbf{0} \end{bmatrix}$$

   where $\mathbf{R}$ is upper triangular. The orthogonal matrix $\mathbf{Q}$ is

$$\mathbf{Q} = (\mathbf{P}_n\mathbf{P}_{n-1}\dots\mathbf{P}_1)^T = \dots = \mathbf{P}_1\mathbf{P}_2\dots\mathbf{P}_n$$

   where the last equality uses the orthogonality of the $\mathbf{P}$ reflection matrices.

5. The computational complexity of using Householder reflections to solve an over-determined system found in the linear least squares problem is approximately $2mn^2 - \frac{2}{3}n^3$.

   The computational complexity of using Householder reflections to take the QR decomposition of an $n \times n$ matrix is approximately $n^3 + n^2$.

4.3.6 Comparison of Computational Complexities

To solve the over-determined system $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$ where $\mathbf{A} \in \mathbb{R}^{m\times n}, \boldsymbol{x} \in \mathbb{R}^n$ and $\boldsymbol{b} \in R^m$ where $m \gg n$, for example like in the linear least squares problem.

- Solving the normal system takes mn^2 but is often unstable
- Modified Gram Schmidt is $2mn^2$; this is more stable than the normal system
- Givens takes $3mn^2 - n^3$ and is quite stable
- Householder takes $2mn^2 - \frac{2}{3}n^3$

To solve the square system $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$ where $\mathbf{A} \in \mathbb{R}^{n\times n}$ and $\boldsymbol{x}, \boldsymbol{b} \in R^n$.

- LU decomposition takes $\frac{2}{3}n^3$
- Givens takes $2n^3$
- Householder takes $\frac{4}{3}n^3$.

Although LU is faster, Householder and Givens are more stable.

4.3.7 Singular Decomposition

- For all matrices $\mathbf{A} \in \mathbb{R}^{m\times n}$ where $m \geq n$ there exists a *singular decomposition*

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$$

  where $\mathbf{U} \in \mathbb{R}^{m\times m}$ and $\mathbf{V} \in \mathbb{R}^{n\times n}$ are orthogonal and $\mathbf{\Sigma} \in \mathbb{R}^{m\times n}$ is a quasi-diagonal matrix (basically diagonal, but not square. Nonzero elements on what would be the main diagonal and then a few totally zero rows at the bottom) with elements $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_n$. The elements σ_j are called *singular values*. The columns of the matrix $\mathbf{U}$ are called left-singular vectors and the columns of $\mathbf{V}$ are called right-singular vectors.

- For $\mathbf{A} \in \mathbb{R}^{m\times n}$ for which rank $\mathbf{A} = n$ with singular decomposition $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$, the minimum $\|\mathbf{A}\boldsymbol{x} - \boldsymbol{b}\|_2$ occurs for

$$\boldsymbol{x} = \sum_{j=1}^{n} \frac{\boldsymbol{u}_j^T \boldsymbol{b}}{\sigma_j} \boldsymbol{v}_j$$

5 Eigenvalues and Eigenvectors

5.1 Some Eigenvalue Theory

- The vector $\boldsymbol{x} \in \mathbb{C}^n$ and scalar $\lambda \in \mathbb{C}$ is called an *eigenvector-eigenvalue pair* of the matrix $\mathbf{A} \in \mathbb{C}^{n\times n}$ if $\mathbf{A}\boldsymbol{x} = \lambda\boldsymbol{x}$. The scalar λ is the eigenvalue and the vector $\boldsymbol{x}$ is the corresponding right-sided eigenvector.

- The vector $\boldsymbol{y} \in \mathbb{C}^n$ is a *left eigenvector* of $\mathbf{A}$ if $\boldsymbol{y}^H\mathbf{A} = \lambda\boldsymbol{y}^H$ where $\cdot^H$ denotes the hermitian transpose. The eigenvalue λ is the same for both the right and left eigenvectors.

- The eigenvectors of a symmetric matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$ corresponding to different eigenvalues $\lambda, \mu \in \mathbb{C}$ are mutually orthogonal.

- Formally, the eigenvalues of a matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$ are the zeros of the matrix's associated *characteristic polynomial* $p_{\mathbf{A}}(\lambda) = \det(\mathbf{A} - \lambda\mathbf{I})$. However. there are more efficient ways to find a matrix's eigenvalues than from the zeros of the characteristic polynomial.

  Eigenvalue-finding algorithms can be further optimized for sparse and symmetric matrices.

- For every square matrix $\mathbf{A} \in \mathbb{C}^{n\times n}$ there exist unitary matrix $\mathbf{Q} \in \mathbb{C}^{n\times n}$ and upper triangular matrix $\mathbf{U} \in \mathbb{C}^{n\times n}$, called a *Schur form* of $\mathbf{A}$, such that $\mathbf{Q}^H\mathbf{A}\mathbf{Q} = \mathbf{U}$. The decomposition $\mathbf{A} = \mathbf{Q}\mathbf{U}\mathbf{Q}^H$ is called the *Schur decomposition* of $\mathbf{A}$.

  Analogously, for every real square matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$ there exists an orthogonal matrix $\mathbf{Q} \in \mathbb{R}^{n\times n}$ and (possibly quasi-) upper triangular matrix $\mathbf{U} \in \mathbb{R}^{n\times n}$ such that $\mathbf{Q}^T\mathbf{A}\mathbf{Q} = \mathbf{U}$. $\mathbf{U}$ is quasi-diagonal if $\mathbf{A}$ has complex eigenvalues, in which case the corresponding diagonal element is replaced by a 2×2 block.

5.2 Power Method and Reduction

5.2.1 The Power Method

The power method reliably finds a matrix's largest eigenvalue by absolute value.

- The *dominant eigenvalue* of a matrix is the matrix's largest eigenvalue by absolute value.

- *Algorithm:* Starting with a square matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$, choose an initial vector $\boldsymbol{y}_0 \in \mathbb{R}^n$ (which can be random in practice) and a tolerance ϵ. For $k = 0, 1, \dots$ calculate
$$\boldsymbol{y}_{k+1} = \frac{\mathbf{A}\boldsymbol{y}_k}{\|\mathbf{A}\boldsymbol{y}_k\|}$$
and stop iteration when $\|\mathbf{A}\boldsymbol{y}_k - \rho_k\boldsymbol{y}_k\| \leq \epsilon$, where $\rho(\boldsymbol{x}, \mathbf{A}) = \frac{\boldsymbol{x}^T\mathbf{A}\boldsymbol{x}}{\boldsymbol{x}^T\boldsymbol{x}}$.

  ρ_k is the approximation for $\mathbf{A}$'s dominant eigenvalue and $\boldsymbol{y}_k$ is the approximation for an eigenvector corresponding to $\mathbf{A}$'s dominant eigenvalue.

  The normalization avoids the expression $\boldsymbol{y}_{k+1} = \mathbf{A}^k\boldsymbol{y}_0$ diverging for large k.

- *On convergence:* The power method converges to a dominant eigenvalue-eigenvector pair as long as the initial vector $\boldsymbol{y}_0$ has a non-zero component in the position corresponding to the dominant eigenvalue's direction. As long as $\boldsymbol{y}_0$ has non-zero elements, the power method converges.

  In practice, because of numerical error, we never actually encounter exact zeros in computer arithmetic, and numerical error would theoretically recover the convergence process even for a zero component in the position of the dominant eigenvalue.

- *On stopping iteration:* The goal is to stop when $\boldsymbol{y}_k$ and $\boldsymbol{y}_{k+1}$ have an arbitrary similar *direction*. Instead of comparing $\boldsymbol{y}_k$ and $\boldsymbol{y}_{k+1}$ with a norm, we use the Rayleigh quotient, defined for the $\boldsymbol{x} \in \mathbb{R}^n$ and matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ as

  $$\rho(\boldsymbol{x}, \mathbf{A}) = \frac{\boldsymbol{x}^T \mathbf{A} \boldsymbol{x}}{\boldsymbol{x}^T \boldsymbol{x}}$$

  When $\boldsymbol{x}$ is close to one of $\mathbf{A}$'s eigenvectors, ρ is close to the corresponding eigenvalue.

  We stop power iteration when $\|\mathbf{A}\boldsymbol{y}_k - \rho_k \boldsymbol{y}_k\| \leq \epsilon$ in which case ρ_k is the approximation for $\mathbf{A}$'s largest eigenvalue.

- *How the algorithm works:* A very rough idea is that because dominant eigenvalue is larger than the other eigenvalues, the product $\boldsymbol{y}_{k+1} = \mathbf{A}^k \boldsymbol{y}_0$ converges to a corresponding eigenvector after many multiplications.

- Power iteration is a very simple algorithm but may converge slowly. The most time-consuming operation is the matrix multiplication step, so the algorithm can be implemented efficiently for sparse matrices where multiplication is simple.

  The order of convergence is linear, and the method converges faster when the largest eigenvalue is much larger than the second-largest.

5.2.2 Reduction

- Reduction allows us to use power iteration to find all of a matrix's eigenvalues, not just the largest one. Suppose we've use the power method to find $\mathbf{A}$'s dominant eigenvalue-eigenvector pair $\lambda_1, \boldsymbol{x}_1$.

  The general idea of reduction is to modify $\mathbf{A}$ by somehow "removing" the dominant eigenvalue. We could then use power method on the modified matrix to find $\mathbf{A}$'s second-largest eigenvalue, and repeat the reduction-power iteration process for all of $\mathbf{A}$'s eigenvalues.

  Formally, the goal of reduction is to create a new matrix $\mathbf{B}$ whose eigenvalues match all of $\mathbf{A}$'s eigenvalues except the dominant eigenvalue, which should be replaced by zero. With the dominant eigenvalue replaced by zero, there is no chance of it being found again by the power method (because every other eigenvalue will be larger in magnitude.)

5.2.3 Reduction of Symmetric Matrices

Reduction is easy for symmetric matrices; we take advantage of the fact that symmetric matrices have orthogonal eigenvectors. Reduction of symmetric matrices is sometimes called *Hotelling deflation*.

- For a symmetric matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$ with a dominant eigenvalue-eigenvector pair $\lambda_1, \boldsymbol{x}_1$, where $\boldsymbol{x}_1$ *must be normalized* (i.e. $\|\boldsymbol{x}_1\|_2 = 1$) the formula for the reduced matrix $\mathbf{B}$ is then
$$\mathbf{B} = \mathbf{A} - \lambda_1 \boldsymbol{x}_1 \boldsymbol{x}_1^T$$
In this case, if $\lambda_j, \boldsymbol{x}_j$ are $\mathbf{A}$'s eigenvalue-eigenvector pairs for $j = 1, \dots, n$ with $\lambda_1, \boldsymbol{x}_1$ the dominant pair, then $\mathbf{B}\boldsymbol{x}_1 = \mathbf{0}$ and $\mathbf{B}\boldsymbol{x}_j = \lambda_j \boldsymbol{x}_j$ for $j \neq 1$.

- *Quick Proof:* To show $\mathbf{B}\boldsymbol{x}_1 = \mathbf{0}$, use the definition of $\mathbf{B}$, the eigenvalue identity $\mathbf{A}\boldsymbol{x}_1 = \lambda_1 \boldsymbol{x}_1$, and the condition $\|\boldsymbol{x}_1\|_2 = 1$ to get
$$\begin{aligned}\mathbf{B}\boldsymbol{x}_1 &= \mathbf{A}\boldsymbol{x}_1 - \lambda_1(\boldsymbol{x}_1\boldsymbol{x}_1^T)\boldsymbol{x}_1 = \lambda_1\boldsymbol{x}_1 - \lambda_1\boldsymbol{x}_1(\boldsymbol{x}_1^T\boldsymbol{x}_1) \\ &= \lambda_1\boldsymbol{x}_1 - \lambda_1\boldsymbol{x}_1\left(\|\boldsymbol{x}_1\|_2\right) = \lambda_1\boldsymbol{x}_1 - \lambda_1\boldsymbol{x}_1 = 0\end{aligned}$$
To show $\mathbf{B}\boldsymbol{x}_j = \lambda_j\boldsymbol{x}_j$ for $j \neq 1$, use the fact that symmetric matrices have orthogonal eigenvectors, so $\boldsymbol{x}_i \cdot \boldsymbol{x}_j = \delta_{i,j}$ for the eigenvectors $\boldsymbol{x}_{i,j}$, meaning $\boldsymbol{x}_1 \cdot \boldsymbol{x}_j = 0$ for $j \neq 1$.
$$\begin{aligned}\mathbf{B}\boldsymbol{x}_j &= \mathbf{A}\boldsymbol{x}_j - \lambda_1(\boldsymbol{x}_1\boldsymbol{x}_1^T)\boldsymbol{x}_j = \lambda_j\boldsymbol{x}_j - \lambda_1\boldsymbol{x}_1(\boldsymbol{x}_1^T\boldsymbol{x}_j) \\ &= \lambda_j\boldsymbol{x}_j - \lambda_1\boldsymbol{x}_1(0) = \lambda_j\boldsymbol{x}_j\end{aligned}$$

- Applying the power method to $\mathbf{B}$ would produce $\mathbf{A}$'s second-largest eigenvalue-eigenvector pair. Because $\mathbf{B}\boldsymbol{y} = \mathbf{A}\mathbf{y} - \lambda_1(\boldsymbol{x}_1^T\boldsymbol{y})\boldsymbol{x}_1$, we don't need to explicitly calculate $\mathbf{B}$.

5.2.4 Reduction of Non-Symmetric Matrices

- For an arbitrary real matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$ with a dominant eigenvalue-eigenvector pair $\lambda_1, \boldsymbol{x}_1$, where $\boldsymbol{x}_1$ *must be normalized* (i.e. $\|\boldsymbol{x}_1\|_2 = 1$) the formula for the reduced matrix $\mathbf{B}$ is
$$\mathbf{B} = \mathbf{Q}\mathbf{A}\mathbf{Q}^T$$
where the orthogonal matrix $\mathbf{Q}$ is a Householder reflection mapping $\boldsymbol{x}_1$ to $\alpha\boldsymbol{e}_1$, i.e. $\mathbf{Q}\boldsymbol{x}_1 = \alpha\boldsymbol{e}_1$. In this case, $\mathbf{B}$ has the form
$$\mathbf{B} = \begin{bmatrix}\lambda_1 & \boldsymbol{b}^T \\ \mathbf{0} & \mathbf{C}\end{bmatrix}$$
for some vector $\boldsymbol{b} \in \mathbb{R}^{n-1}$ and matrix $\mathbf{C} \in \mathbb{R}^{n-1\times n-1}$ whose eigenvalues match $\mathbf{A}$'s eigenvalues $\lambda_2, \dots, \lambda_n$.

5.2.5 Inverse Iteration

- Inverse iteration is used to find the *smallest* eigenvalue-eigenvector pair of a matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$. Inverse iteration works on the principle of the eigenvalue-eigenvector pair $\lambda_j, \boldsymbol{x}_j$ of the matrix $\mathbf{A}$ corresponding to the pair $\frac{1}{\lambda_j}, \boldsymbol{x}_j$ of $\mathbf{A}$'s inverse.

- *Quick proof:* For an arbitrary eigenvalue-eigenvector pair $\lambda_i, \boldsymbol{x}_i$ from the matrix $\mathbf{A}$, start with the trivial equality $\boldsymbol{x}_i = \boldsymbol{x}_i$, use the fact that $\mathbf{A}^{-1}\mathbf{A}\boldsymbol{x}_i = \mathbf{I}\boldsymbol{x}_i = \boldsymbol{x}_i$, and apply the eigenvalue property $\mathbf{A}\mathbf{x}_i = \lambda_i \boldsymbol{x}_i$

  $$\boldsymbol{x}_i = \boldsymbol{x}_i \implies \mathbf{A}^{-1}\mathbf{A}\boldsymbol{x}_i = \boldsymbol{x}_i \implies \mathbf{A}^{-1}\lambda_i\boldsymbol{x}_i = \boldsymbol{x}_i \implies \mathbf{A}^{-1}\boldsymbol{x}_i = \frac{1}{\lambda_i}\boldsymbol{x}_i$$

  If follows that $\mathbf{A}$'s eigenvalue-eigenvector pair $\lambda_i, \boldsymbol{x}_i$ corresponds to the pair $\frac{1}{\lambda_i}, \boldsymbol{x}_i$ of $\mathbf{A}$'s inverse $\mathbf{A}^{-1}$.

- *Algorithm:* To find the *smallest* eigenvalue-eigenvector pair of a matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$, use modified power iteration on the inverse matrix $\mathbf{A}^{-1}$ as follows:

  For efficiency (to avoid calculating the inverse), iterate using $\boldsymbol{y}_{k+1}$ by solving the system $\mathbf{A}\boldsymbol{y}_{k+1} = \boldsymbol{y}_k$ instead of proceeding directly with $\boldsymbol{y}_{k+1} = \mathbf{A}^{-1}\boldsymbol{y}_k$.

  The power iteration will produce the eigenvalue λ_j, the largest eigenvalue of $\mathbf{A}^{-1}$. $\mathbf{A}$'s smallest eigenvalue is then $\frac{1}{\lambda_j}$.

5.2.6 QR Iteration

QR iteration is the most efficient way to find all eigenvalues of a general, non-symmetric square matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$.

- *Algorithm:* Start with $\mathbf{A}_0 = \mathbf{A}$. For $k = 0, 1, \ldots$

  1. Find the QR decomposition $\mathbf{A}_k = \mathbf{Q}_k\mathbf{R}_k$.
  2. Calculate the next matrix with $\mathbf{A}_{k+1} = \mathbf{R}_k\mathbf{Q}_k$

- If $\mathbf{A}$ has only real eigenvalues, then for large k, $\mathbf{A}_k$ becomes upper-triangular and $\mathbf{A}_k$'s diagonal entries approach $\mathbf{A}$'s eigenvalues. If $\mathbf{A}$ has unique eigenvalues by absolute value, $\mathbf{A}$ converges to its Schur form.

  If $\mathbf{A}$ has m complex eigenvalues (where $m < n$), then for large k, $\mathbf{A}_k$ becomes quasi-upper triangular, $\mathbf{A}_k$'s diagonal entries approach $\mathbf{A}$'s real eigenvalues, and a $m\times m$ sub-matrix whose eigenvalues are $\mathbf{A}$'s complex eigenvalues remains on $\mathbf{A}_k$'s diagonal.

5.3 Eigenvalues of Symmetric Matrices

Eigenvalue methods for symmetric methods make use of the principle that every symmetric matrix is orthogonally similar to a tridiagonal matrix. The methods in this section also assume a priori the tridiagonal matrix is irreducible, i.e. has no zeros on upper tridiagonal. If it had zeros, the matrix could be split into two principle sub-matrix blocks to eliminate the zero while preserving the eigenvalues.

5.3.1 Tridiagonalization

- Because we can change any symmetric matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$ into a tridiagonal matrix $\mathbf{T} = \mathbf{PAP}^T$ with similarity transformations.
- More so, as long as $\mathbf{A}$ is symmetric $\mathbf{T}$ is also symmetric, so we assume for this section that we start each method with a symmetric tridiagonal matrix.

5.3.2 Sturm Sequence

- Start with the tridiagonal, irreducible, symmetric matrix $\mathbf{T}$ of the form

$$\mathbf{T} = \begin{bmatrix} a_1 & b_1 & & & \\ b_1 & a_1 & b_2 & & \\ & \ddots & \ddots & \ddots & \\ & & b_{n-2} & a_{n-1} & b_{n-1} \\ & & & b_{n-1} & a_n \end{bmatrix}$$

  and let $\mathbf{T}_j$ be the principle $j\times j$ sub-matrix of $\mathbf{T}$. $\mathbf{T}_j$'s characteristic polynomial is

$$f_j(\lambda) = \det(\mathbf{T}_j - \lambda \mathbf{I}_j)$$

  where f_j is a Sturm-sequence as defined immediately below.

- Formally, sequence of polynomials p_j for $j = 0, 1, \ldots, n$ is a *Sturm sequence* if
  1. $f_0(\lambda) \neq 0$ for all λ, i.e. the first polynomial in the sequence has no zeros
  2. In the interior of the sequence (for $j < n$), if $f_j(\lambda_0) = 0$, then $f_{j-1}(\lambda_0) \cdot f_{j+1}(\lambda_0) < 0$.
  3. If $f_n(\lambda_0) = 0$, then $f_{n-1}(\lambda_0) \cdot f_n'(\lambda_0) < 0$.
- We can recursively construct the characteristic polynomials of all of the principle sub-matrices $\mathbf{T}_j$ in terms of the elements a_j, b_j with the Sturm function sequence

$$f_{j+1}(\lambda) = (a_{j+1} - \lambda) f_j(\lambda) - b_j^2 f_{j-1}(\lambda)$$

  where $f_0(\lambda) = 1$ and $f_1(\lambda) = a_1 - \lambda$. The Sturm sequence is an easy way to construct $\mathbf{T}$'s characteristic polynomial $f_j(\lambda) = \det(\mathbf{T}_j - \lambda I)$ instead of by definition involving a determinant.

  Note that if λ_0 is root of $f_n(\lambda)$, then the last polynomial in the Sturm sequence equals zero when evaluated at λ_0. This is important for determining if λ_0 is an eigenvalue of $\mathbf{T}$.

- *Note:* $\mathbf{T}$ has unique eigenvalues because all zeros of the Sturm sequence are simple zeros. More generally, any irreducible tridiagonal symmetric matrix has unique eigenvalues.
- *Lemma: On Counting Sign Agreements*
  For a chosen λ_0, let $s(\lambda_0)$ denote the number of sign agreements between successive terms in the sequence $f_s(\lambda_0)$ for $s = 0, 1, \ldots, n$, using the following ruls

  - Two adjacent terms with the same sign count as one sign agreement
  - Interior zeros count as one sign agreement, zeros at the sequence's border do not.
  - Some examples:

$+\ +$	1 sign agreement
$-\ +\ +\ 0\ -\ -$	3 sign agreements
$+\ -\ -\ +\ -\ +\ +\ -\ 0$	2 sign agreements
$+\ +\ -\ -\ +\ +\ 0\ +\ -\ -$	5 sign agreements

- *Theorem:* The number of sign agreements $s(\lambda_0)$ is the number of $\mathbf{T}$'s eigenvalues that are strictly larger than λ_0. (In the usual sense, including sign, not larger by absolute value).

- *Algorithm:* We choose an initial value $\lambda_0^{(0)}$ and calculate the number of sign agreements $s^{(0)}(\lambda_0^{(0)})$ as described above. Starting with $j = 0$:

  1. If $f_n(\lambda_0^{(j)}) = 0$, then $\lambda_0^{(j)}$ is an eigenvalue of $\mathbf{T}$, and we end the method.
  2. In the more likely case $f_n(\lambda_0^{(j)}) \neq 0$, proceed with bisection method logic. Choose a larger guess $\lambda_0^{(j+1)}$ on the next step. If the number $u(\lambda_0^{(j+1)})$ changes by 1, then there is precisely one eigenvalue between $\lambda_0^{(j)}$ and $\lambda_0^{(j+1)}$.

  The guesses $\lambda_0^{(j)}$ are approximations for $\mathbf{T}$'s eigenvalues, and the sign agreements $s(\lambda_0^{(j)})$ are a reference for how close the guesses are to the true eigenvalues. If $s(\lambda_j - \epsilon) = j$ and $s(\lambda_j + \epsilon) = j - 1$, the λ_j is an approximation for $\mathbf{T}$'s jth eigenvalue.

- *On Initial Guesses* How to choose the initial guess for λ_0 for efficient convergence? Use the infinity norm of $\mathbf{A}$ as a reference to establish range in which eigenvalues fall.

$$\text{eig}(A) \in \big[-\|\mathbf{A}\|_\infty, +\|\mathbf{A}\|_\infty \big]$$

  Choose a in initial guess $\lambda_0^{(0)}$ in the range $[-\|\mathbf{A}\|_\infty, +\|\mathbf{A}\|_\infty]$.

5.4 Jacobi Iteration for Eigenvalues

- Jacobi iteration uses the principle of all symmetric matrices being diagonalizable, which the diagonal elements precisely the matrix's eigenvalues.

- Jacobi iteration uses left and right Givens rotations to transform the symmetric matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ into a diagonal form; the diagonal elements are $\mathbf{A}$'s eigenvalues.

  The goal is to find $\mathbf{R} = \begin{bmatrix} c & -s \\ s & c \end{bmatrix}$ such that application to a symmetric matrix yields

$$\mathbf{R}^T \begin{bmatrix} a_{ii} & a_{ij} \\ a_{ij} & a_{jj} \end{bmatrix} \mathbf{R} = \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix}$$

In this case, the Givens rotation matrix values are

$$c = \frac{1}{\sqrt{1+t^2}} \qquad s = ct$$

where

$$t = \frac{\text{sgn}(\tau)}{|\tau| + \sqrt{1+\tau^2}} \qquad \tau = \frac{a_{ii} - a_{jj}}{2a_{ij}}$$

For matrices larger than 2×2 the Givens rotation matrix is a modified identity matrix with the 2x2 Givens matrix plugged on the diagonal. To zero the element in $\mathbf{A} \in \mathbb{R}^{n \times n}$ at the point (i, j), (assuming $i > j$) modify the $n \times n$ identity matrix by plugging in c at (j, j), $-s$ at (j, i), s at (i, j), and c at (i, i), as in the Givens rotation section.

- *Definition:* The *offset value* of a matrix $\mathbf{A} \in \mathbb{R}^{n\times}$ is $\mathbf{A}$'s Frobenius norm excluding the diagonal elements.

$$\text{off}(\mathbf{A}) = \sqrt{\sum_{j \neq k}^{n} |a_{jk}|^2}$$

The offset value measures how close to diagonal a matrix is. The closer $\text{off}(\mathbf{A})$ is to zero, the closer $\mathbf{A}$ is to being diagonal.

- In Jacobi iteration, $\mathbf{A}$'s offset value decreases after each Givens rotation. This makes sense, because each iteration zeros a non-diagonal element. For Jacobi iteration:

$$\left(\text{off } \mathbf{A}^{(n+1)}\right)^2 = \left(\text{off } \mathbf{A}^{(n)}\right)^2 - 2a_{ij}^2$$

where a_{ij} is the off-diagonal element that was made to vanish with the Given's rotation on step n.

- *Algorithm Idea:* To find the eigenvalues of a symmetric matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ with Jacobi iteration, perform the above Givens rotations and turning non-diagonal elements a_{ij} into zero until the offset value $\text{off}(A)$ becomes smaller than chosen tolerance ϵ.

  Once $\mathbf{A}$ is nearly diagonal, the diagonal elements approximate $\mathbf{A}$'s eigenvalues.

- In passing, note there are different ways to choose which off-diagonal element to make zero in Jacobi iteration. Some methods are

  - Classical method: zero the largest off-diagonal element by absolute value
  - Cyclic method: zero all off-diagonal elements in a chosen order
  - Prague method: zero all off-diagonal elements in a chosen order, as long as the element is larger by absolute value than a small boundary value.

6 Polynomial Interpolation

6.1 Introduction

- Given points (x_i, y_i) where $i = 0, 1, \dots, n$ and no two x_i values are equal, the goal is to find a polynomial of lowest possible degree such that

$$p(x_i) = y_i$$

for all i. In other words, the polynomial must pass through all points in the data set.

- Polynomial interpolation rests on the following theorem: for the data points (x_i, y_i) where $i = 0, 1, \dots, n$ and no two x_i values are equal, there exists a unique interpolation polynomial p_n of degree less than or equal to n.

- The interpolation polynomial is generally used to estimate values the data would take on at points $x \neq x_i$.

- In practice, classic interpolation polynomials are useful only for small n; for larger sets of data points, *spline interpolation* is used instead, in which low-order polynomials are "glued" together into a piecewise polynomial called a spline.

6.1.1 Classic Form

Find a polynomial of degree $\leq n$ passing through the data points (x_i, y_i), $i = 0, 1, \dots, n$.

- The interpolation polynomial is written in standard form

$$p_n(x) = a_n x^n + \cdots + a_1 x + a_0$$

The goal is to find the values of the coefficients $a_0, a_1, \dots, a_n$ for which p_n passes through (x_i, y_i).

- Plug all $(n+1)$ data points (x_i, y_i) into the polynomial p_n to get a system of $n+1$ linear equations for the coefficients a_i. This system takes the form

$$\begin{aligned} a_n x_0^n + \cdots + a_1 x_0 + a_0 &= y_0 \\ a_n x_1^n + \cdots + a_1 x_1 + a_0 &= y_1 \\ &\vdots \\ a_n x_n^n + \cdots + a_1 x_n + a_0 &= y_n \end{aligned}$$

- The system of equations for the coefficients can be written in matrix form

$$\mathbf{V}\boldsymbol{a} = \boldsymbol{y}, \qquad \mathbf{V} = \begin{bmatrix} x_0^n & x_0^{n-1} & \cdots & x_0 & 1 \\ x_1^n & x_1^{n-1} & \cdots & x_1 & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ x_n^n & x_n^{n-1} & \cdots & x_n & 1 \end{bmatrix}$$

where $\mathbf{V}$ is the matrix of x_i values, $\boldsymbol{a}$ is unknown vector of coefficients and $\boldsymbol{y}$ are the known function values. $\mathbf{V}$ is often called the *Vandermonde matrix*.

- The Vandermonde matrix is non-singular ($\det \mathbf{V} = \prod_{i<j}(x_i - x_j) \neq 0$) since $x_i \neq x_j$ for $i \neq j$, so the system has a unique solution for the vector of coefficients $\boldsymbol{a}$.
- The classical method of finding the $(n+1)$ polynomial coefficients a_1 directly from a system of linear equation only works for small n. Problems are:
  1. Calculation quickly grows to costly and cumbersome
  2. Large sensitivity values κ for large n
  3. Interpolation polynomial begins to oscillate wildly between subsequent points for large n (Runge's phenomenon)

  Solution is to use splines. In this case the interpolant is a chain of several lower-degree polynomials.

6.1.2 Lagrange Polynomial Interpolation

Instead of using the standard polynomial basis $1, x, x^2, \dots, x^n$, Lagrange interpolation constructs the interpolation polynomial using a basis of *Lagrange polynomials* l_n.

- Given data points $(x_i, y_i), i = 0, 1, \dots, n$ we construct the Lagrange polynomials via
$$l_i(x) = \prod_{j \neq i}^{n} \frac{x - x_j}{x_i - x_j}$$
The polynomial l_i is constructed to have zeros at all x_j values except at x_i, for which $l_i(x_i) = 1$.
- The interpolation polynomial p_n is a linear combination of the Lagrange polynomials
$$p_n(x) = \sum_{i=0}^{n} y_i l_i(x)$$
Because the l_i are constructed so $l_i(x_j) = \delta_{i,j}$, the polynomial equals y_i at each x_i.
- In terms of $\omega(x) = (x - x_0) \cdots (x - x_n)$, then the Lagrange polynomials are
$$l_i = \frac{\omega(x)}{(x - x_i)\omega'(x_i)}$$
- *Error Bound:* Consider the function $f \in C^{n+1}([a, b])$ and the unique points $x_0, x_1, \dots, x_n \in [a, b]$. For each $x \in [a, b]$ there exists $\xi \in [a, b]$ for which
$$f(x) - p_n(x) = \frac{f^{(n+1)}(\zeta)}{(n+1)!}\omega(x)$$
This theorem provides a theoretical bound to the error of the interpolation polynomial for estimating function values at $x \neq x_i$ if the original function is known.
$$\max_{x \in [a,b]} |f(x) - p_n(x)| \leq \frac{1}{(n+1)!} \max_{\xi \in [a,b]} \left| f^{(n+1)}(\xi) \right| \max_{x \in [a,b]} |\omega(x)|$$

The theorem is not particularly useful in practice since it provides no information on the location of the value ξ and the original function is rarely known a priori.

6.1.3 Newton Form of Polynomial Interpolation

Newton's form of polynomial interpolation makes it possible to interpolate derivative as well as function values. The method uses the concept of *divided differences*, which we introduce first.

- The *divided difference* of the function f for the points $x_0, x_1, \dots, x_n$ is the leading coefficient of the interpolation polynomial p_n interpolating f at the points $x_0, x_1, \dots, x_n$. The divided difference is denoted by $[x_0, x_1, \dots, x_n]f$.

- Calculating divided differences using the interpolation polynomial is inefficient. Instead, we find divided differences with the recursive formula

$$[x_0, x_1, \dots, x_n]f = \frac{[x_1, \dots, x_n]f - [x_0, x_1, \dots, x_{n-1}]f}{x_n - x_0}$$

  where f's divided difference for single point x_i is $[x_i]f = f(x_i)$.

- Divided differences are related to a function's derivative by the following formula

$$\lim_{x_1 \to x_0} [x_0, x_1]f = \lim_{x_1 \to x_0} \frac{f(x_1) - f(x_0)}{x_1 - x_0} = f'(x_0)$$

  which leads to the generalized definition of divided difference which works even when the points $x_0, \dots x_n$ are equal

$$[x_0, \dots, x_n]f = \begin{cases} \dfrac{f^{(n)}(x_0)}{n!}, & x_0 = \dots = x_n \\ \dfrac{[x_1, \dots, x_n]f - [x_0, \dots, x_{n-1}]f}{x_n - x_0}, & \text{otherwise} \end{cases}$$

- The Newton polynomial interpolating the function f at the points $x_0, x_1, \dots, x_n$ is

$$p_n(x) = [x_0]f + (x - x_0)[x_0, x_1]f + \dots + (x - x_0)(x - x_1) \cdots (x - x_{n-1})[x_0, x_1, \dots x_n]f$$

  If we use multiple occurrences of the same interpolation point, the polynomial will also interpolate the function's derivatives. So if we repeat x_i say n times, the Newton polynomial will interpolate $f(x_i), f'(x_i), \dots, f^{(n-1)}(x_i)$.

- Newton polynomial interpolation has an analogous error bound to Lagrange interpolation, except that the Newton form permits multiple occurrences of the same x_i values. If we define

$$\omega(x) = (x - x_1)^{k_1}(x - x_2)^{k_2} \cdots (x - x_n)^{k_n}$$

  where k_i denotes the number of occurrences of the point x_i, then

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

for some ξ in the interpolation interval. As with Lagrange interpolation, the associated error bound is

$$\max_{x\in[a,b]} |f(x) - p_n(x)| \leq \frac{1}{(n+1)!} \max_{\xi\in[a,b]} \left|f^{(n+1)}(\xi)\right| \max_{x\in[a,b]} |\omega(x)|$$

6.2 Numerical Differentiation

The general goal is to numerically calculate the derivative values of a continuously differentiable function $f : [a, b] \to R$ at points $c \in [a, b]$. In general, numerical differentiation is highly sensitive to numerical errors and is not well suited to numerical methods.
A numerical derivative can be found using the method of undetermined coefficients or from the derivative of an interpolation polynomial or spline.

6.2.1 Undetermined Coefficients

- The goal in the undetermined coefficient method is to calculate the derivative $f'(c)$ in terms of a linear combination of function values in a neighborhood of c.
- Goal is for the formula for approximation to work for polynomials of as high order as possible.
- Choose a basis for the space of polynomials of degree n, where $n + 1$ is the number of terms in the linear combination. So e.g. an approximation with three points and function values would have $n = 2$, i.e. a second-order polynomial.

For a three point approximation of x_1 using the neighboring points x_0 and x_2, we get to the symmetric difference quotient

$$f'(x_1) = \frac{f(x_2) - f(x_0)}{2h} = \frac{f(x_1 + h) - f(x_1 - h)}{2h}$$

This is generally written

$$f'(x) = \frac{f(x + h) - f(x - h)}{2h}$$

Problem: In numerical computation, for very small h, the approximation doesn't work well because of numerical error. The problem is that as $h \to 0$, $f(x + h) \to f(x - h)$, and we are subtracting two values that are very close together. Possible error because we can only represent 16 decimal places, and likewise we divide this by a small number which blows up the error even more!
Error: A Taylor expansion gives

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

Note that the $f''(x)$ term vanishes because of $\pm h$.
The Taylor error is $\propto Ah^2$ and the numerical rounding error (given/told) adds a term to the error $\propto \frac{u}{h}$ where $u \approx 10^{-16}$ is the rounding error. The total error in the numerical approach is then $\propto Ah^2 + \frac{u}{h}$. This is minimized not as $h \to 0$ but for an intermediate value.

7 Numerical Methods for Ordinary Differential Equations

7.1 First Order Differential Equations

7.1.1 Overview

- The general problem is to solve the first-order initial value problem

$$y' = \frac{\mathrm{d}y}{\mathrm{d}x} = f(x, y) \qquad y(x_0) = y_0$$

on the interval $x \in [a, b]$

- A first-order initial value problem is solved numerically by dividing the interval $[a, b]$ with $n + 1$ partition points $\{x_i\}_{i=0}^{n}$ where

$$a = x_0 < x_1 < \cdots < x_{n-1} < x_n = b$$

In general, spacing between individual points could be arbitrary. Often, however, the points are separated uniformly by the distance $h = x_{i+1} - x_i$.

- The goal is to find numerical approximations to the solution y at each point x_i. So basically to find a value y_i at each partition point x_i, and then approximate the solution y by the values y_i at the partition points. In general, the more partition points, the more approximations y_i and the better the solution.

- The approximations y_i are found either with an *explicit* or *implicit*. An explicit method starts with the initial value y_0 and calculates successive approximations directly on the basis of the previous approximations, for example

$$y_{i+1} = \Phi(h, x_i, y_{i-k}, \ldots, y_{i-1}, y_i)$$

Typically we use only the previous approximation y_i and have $y_{i+1} = \Phi(h, x_i, y_i)$.

An implicit method calculates approximations by solving a (generally nonlinear) equation involving previous approximations and the next approximation, e.g.

$$y_{i+1} = \Phi(h, x_i, y_{i-k}, \ldots, y_i, y_{i+1})$$

- Both explicit and implicit methods are either *single step* or *multi-step*. In single step methods, the next approximation y_{i+1} is found only from the previous approximation y_i. In multi-step methods, the next approximation is found with multiple previous approximations $y_i, y_{i-1}, \ldots$

7.1.2 Local and Global Error

- *Local error* of a numerical method at a point x_i is the difference between the approximation y_i and the value of the analytic solution at that point, i.e.

$$\text{local error at } x_i = y_i - \psi(x_{i+1})$$

where ψ is the analytic solution to the initial value problem.

- *Global error* is the cumulative sum of the local errors at each data point $\{x_i\}_{i=0}^n$
- Local error is of the order $p \in \mathbb{N}$ if

$$y_{i+1} - z_{x_{i+1}} = Ch^{p+1} + \mathcal{O}(h^{p+2})$$

where C is a constant independent of the step size h.

7.1.3 Explicit Euler's Method

- To solve the initial value problem $y' = f(x, y), y(x_0) = y_0$ in the interval $[a, b]$, choose a step size h and split $[a, b]$ into the points

$$a = x_0 < x_1 < \cdots < x_{n-1} < x_n = b$$

where $x_{i+1} - x_i = h$.

- Starting with $y(x_0) = y_0$, calculate the next point using the formula

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

for $i = 0, 1, \ldots, n$.

7.1.4 Runge-Kutta Methods

There is a large range of Runge-Kutta methods. The example below is the fourth-order RK4 method.

- As usual, to solve the initial value problem $y' = f(x, y), y(x_0) = y_0$ in the interval $[a, b]$, choose a step size h and split $[a, b]$ into the points

$$a = x_0 < x_1 < \cdots < x_{n-1} < x_n = b$$

where $x_{i+1} - x_i = h$.

- Starting with $y(x_0) = y_0$, calculate the coefficients

$$\begin{aligned} k_1 &= hf(x_i, y_i) \\ k_2 &= hf\big(x_i + \tfrac{1}{2}h, y_i + \tfrac{1}{2}k_1\big) \\ k_3 &= hf\big(x_i + \tfrac{1}{2}h, y_i + \tfrac{1}{2}k_2\big) \\ k_4 &= hf(x_i + h, y_i + k_3) \end{aligned}$$

and calculate the next term with the formula

$$y_{i+1} = y_i + \frac{1}{6}k_1 + \frac{1}{3}k_2 + \frac{1}{3}k_3 + \frac{1}{6}k_4 + \mathcal{O}(h^5)$$

for $i = 0, 1, \ldots, n$.

- Runge-Kutta methods are more stable than the Euler method, but calculating the coefficients $k_1 - k_4$ makes RK methods significantly more costly.
- The coefficients are chosen so as many terms as possible vanish in the Taylor series expansion, which minimizes error. The coefficients must add up to one for the method to behave well for constant functions.

7.1.5 Single Step Implicit Methods

- Implicit single step methods take the general form

$$y_{i+1} = \Phi(h, x_i, y_i, y_{i+1}, f)$$

  where the value y_{i+1} occurs implicitly and generally non-linearly. The approximations y_{i+1} are found by solving the resulting non-linear equation, often with iterative methods.

- Fixed-point iteration for $y_{i+1} = \Phi(h, x_i, y_i, y_{i+1}, f)$ converges if

$$\left| \frac{\partial \Phi}{\partial y}(h, x_i, y_i, y, f) \right| \leq 1, \qquad y \approx y_{i+1}$$

  Implicit methods can generally be made to converge by choosing a small step size h.

- A simple implicit method is the trapezoid method, in which new approximations are found with

$$y_{i+1} = y_i + \frac{h}{2}\big[f(x_i, y_i) + f(x_{i+1}, y_{i+1})\big]$$

7.1.6 Systems of First-Order Linear Differential Equations

- The problem is to solve the system of first order LDEs

$$\begin{aligned} y_1' &= f_1(x, y_1, \dots, y_n) \\ y_2' &= f_2(x, y_1, \dots, y_n) \\ &\vdots \\ y_n' &= f_1(x, y_1, \dots, y_n) \end{aligned}$$

  given the n initial conditions $y_1(x_0) = y_{1_0}, y_2(x_0) = y_{2_0}, \dots, y_n(x_0) = y_{n_0}$.

- We use the vector notation $\boldsymbol{Y} = (y_1, \dots, y_n)^T$, $\boldsymbol{F} = (f_1, \dots, f_n)^T$ and $\boldsymbol{Y}_0 = (y_{1_0}, y_{2_0}, \dots, y_{n_0})$and write the system in vector form

$$\boldsymbol{Y}' = \boldsymbol{F}(x, \boldsymbol{Y}), \qquad \boldsymbol{Y}(x_0) = \boldsymbol{Y}_0$$

- The system is solved analogously to a single linear differential equation, with each equation solved individually. For example, the explicit Euler method in vector form reads

$$\boldsymbol{Y}_{i+1} = \boldsymbol{Y}_i + h\boldsymbol{F}(x_i, \boldsymbol{Y}_i)$$

7.1.7 Higher-Order Linear Differential Equations

- The goal is numerically approximation the solution a single nth-order linear differential equation

$$y^{(n)} = f(x, y, y', \dots, y^{(n-1)})$$

  with the initial conditions $y(x_0) = y_0, y'(x_0) = y_0', \dots, y^{(n-1)}(x_0) = y_0^{(n-1)}$.

- The equation is solved by converting the single nth-order equation into a system of n linear differential equations. This is done recursively with the new variables $u_1 = y$, $u_2 = y', \dots, u_n = y^{(n-1)}$. It follows that $u_n' = y^{(n)}$, which leads to the recursive system of first-order LDEs

$$\begin{aligned} u_1' &= u_2 \\ u_2' &= u_3 \\ &\vdots \\ u_n' &= y^{(n)} = f(x, y, y', \dots, y^{(n-1)}) \end{aligned}$$

  with initial conditions $u_1(x_0) = y_0, \dots, u_n(x_0) = y^{(n-1)}(x_0)$

- The resulting system of linear equations is solved using the earlier methods, and the solution y is found from $y = u_1$.

7.2 Linear Second-Order Boundary Problems

- The general form of a linear second-order boundary problem on the interval $[a, b]$ is

$$\begin{aligned} &y'' + p(x)y' + q(x)y = r(x) \\ &y(a) = \alpha, y(b) = \beta \end{aligned}$$

  where $p, q, r : [a, b] \to \mathbb{R}$ are (generally continuous) functions and $\alpha, \beta \in \mathbb{R}$ are constants. Note that the equation is linear in y.

- If y_1 and y_2 solve the second-order boundary problem, the linear combination

$$y = \lambda y_1 + (1 - \lambda)y_2, \qquad \lambda \in \mathbb{R}$$

  also solves the equation $y'' + p(x)y' + q(x)y = r(x)$, and $y(a) = \alpha$.

- This principle forms the basis for the following method: First, solve two initial-value problems of the form

$$y'' + p(x)y' + q(x)y = r(x)$$

  with the two conditions $y(a) = \alpha, y'(a) = \delta_1$ and $y(a) = \alpha, y'(a) = \delta_2$ and denote the solutions y_1 and y_2.

- In this case, the function

$$y = \lambda y_1 + (1 - \lambda)y_2, \qquad \lambda \in \mathbb{R}$$

  solves the equation and satisfies the boundary condition $y(a) = \alpha$. The condition $y(b) = \beta$ is satisfied if

$$\lambda = \frac{\beta - y_2(b)}{y_1(b) - y_2(b)}$$

  In the possible but unlikely event that the constants $\delta_{1,2}$ where chosen so that $y_1(b) = y_2(b)$, we must choose a new constant e.g. δ_3 to replace either δ_1 or δ_2 and repeat the problem.

7.2.1 Finite Difference Method

- The finite difference method is used to solve the general second-order boundary problem

$$y'' + p(x)y' + q(x)y = r(x)$$
$$y(a) = \alpha, y(b) = \beta$$

on the interval $[a, b]$.

- Choose a step size h and split $[a, b]$ into $n + 2$ points

$$a = x_0 < x_1 < \cdots < x_n < x_{n+1} = b$$

where $x_{i+1} - x_i = h$. Note that, in general, the partition points need not be equidistant.

- Find approximations for $y''(x)$, $y'(x)$ and $y(x)$ using the formula

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

for $i = 1, \ldots, n$ and for $j = 0, 1, \ldots, n + 1$.

- Denoting $p(x_i), q(x_i), r(x_i) = p_i, q_i, r_i$ and applying the boundary conditions $y(a) = \alpha$ and $y(b) = \beta$ leads to the system of equations

$$y_2(p_1h + 2) + 2y_1(h^2q_1 - 2) - \alpha(p_1h - 2) = 2r_1h^2$$
$$y_{i+1}(p_ih + 2) + 2y_i(h^2q_i - 2) - y_{i-1}(p_ih - 2) = 2r_ih^2$$
$$\beta(p_nh + 2) + 2y_n(h^2q_n - 2) - y_{n-1}(p_nh - 2) = 2r_nh^2$$

for $i = 2, 3, \ldots, n - 1$.

- The system can be written in matrix form $\mathbf{A}\boldsymbol{y} = \boldsymbol{r}$, where $\mathbf{A} \in \mathbb{R}^{n\times n}$ is a tridiagonal matrix of coefficients, $\boldsymbol{y} \in \boldsymbol{R}^n$ is the numerical approximation for the solution, and $\boldsymbol{r} \in \mathbb{R}^n$ is the right-hand vector.

  As long as A is diagonally dominant and strictly diagonally dominant in at least one row or column, the system can be efficiently solved with the Thomas algorithm, with computational cost $\mathbf{O}(n^2)$.

7.3 Non-Linear Second-Order Boundary Value Problems

The goal is to solve a boundary value problem of the form

$$y'' = f(x, y, y')$$

on the interval $[a, b]$ with the boundary conditions $y(a) = \alpha, y(b) = \beta$. Because f is in general not linear in y, the problem can be difficult to solve.

7.3.1 General Finite Difference Method

- As before, split the $[a, b]$ into $n + 2$ partition points

$$a = x_0 < x_1 < \cdots < x_n < x_{n+1} = b$$

with step size $h = x_{i+1} - x_i$.

- Use the same approximations for $y''(x)$, $y'(x)$ and $y(x)$ as before, namely

$$\begin{aligned} y''(x_i) &= \frac{y(x_{i+1}) - 2y(x_i) + y(x_{i-1})}{h^2} \approx \frac{y_{i+1} - 2y_i + y_{i-1}}{h^2} \\ y'(x_i) &= \frac{y(x_{i+1}) - y(x_{i-1})}{2h} \approx \frac{y_{i+1} - y_{i-1}}{2h} \\ y(x_j) &\approx y_j \end{aligned}$$

for $i = 1, \ldots, n$ and for $j = 0, 1, \ldots, n + 1$.

- Inserting the approximations into the differential equation gives

$$y_{i-1} - 2y_i + y_{i+1} = h^2 f\left(x_i, y_i, \frac{y_{i+1} - y_{i-1}}{2h}\right)$$

for $i = 1, 2, \ldots, n$.

- Applying the boundary conditions $y(a) = \alpha, y(b) = \beta$ leads to s system of n non-linear equations with n unknowns y_i. The system could be solved, for instance, with iterative methods for systems of non-linear equations.

7.3.2 Shooting Method

- As before, the goal is to solve the non-linear boundary value problem

$$y'' = f(x, y, y')$$

on the interval $[a, b]$ with the boundary conditions $y(a) = \alpha, y(b) = \beta$.

- The idea is to solve the equation as an initial value problem with $y(a) = \alpha$ and $y'(a) = k_0$. This leads to the solution $y(x; k_0)$. Naturally, because $y'(a) = k_0$ is chosen randomly $y(x; k_0)$ will not satisfy the boundary condition $y(b; k_0) = \beta$. The idea is to keep making initial guesses for $y'(a) = k_i$ so that $y(x; k_i)$ come closer and closer to satisfying the boundary condition $y(b) = \beta$.

- The goal is to find a value $k = y'(a)$ for which the corresponding initial-value solution $y(x; k)$ satisfies the boundary condition $y(b; k) = \beta$. If we define $F(k) = y(b; k) - \beta$, the goal is to find k for which $F(k) = 0$,

  The solution to the non-linear equation $F(k) = 0$ is often found with the secant method

$$k_{i+2} = k_{i+1} - F(k_{i+1}) \cdot \frac{k_{i+1} - k_i}{F(k_{i+1}) - F(k_i)}, \qquad i = 0, 2, \ldots$$

  On each step, we must solve the initial value problem $y'(a) = k_i$ to determine agreement with the boundary condition $y(b; k) = \beta$.