10. Solving systems of linear equations

Summary of last lecture

- Calculating derivatives numerically:
  - Finite differences with error bounds:

    $f'(a) = \frac{f(a+h)-f(a-h)}{2h} + \mathcal{O}(h^2)$

- Automatic differentiation with dual numbers
  - $c + d\epsilon$ represents f

  with $f(a) = c$, $f'(a) = d$

Goals for today

- Solving systems of linear equations

- Iterative methods
- Gaussian elimination: LU factorization

Systems of linear equations

- How can we **solve** the system of linear equations

$$\begin{cases} x + 3y = 7 \\ 2x - y = 0 \end{cases}$$

Systems of linear equations

- How can we **solve** the system of linear equations

$$\begin{cases} x + 3y = 7 \\ 2x - y = 0 \end{cases}$$

- A **solution** is a vector $\mathbf{x} = (x, y)$ satisfying both equations simultaneously

Why solve systems of linear equations?

- Systems of linear equations forms a key part of *many* computations in scientific computing:

Why solve systems of linear equations?

- Systems of linear equations forms a key part of *many* computations in scientific computing:

- Multi-dimensional Newton method

- Boundary-value problems in ODEs
- Discretization of PDEs

- Stationary solutions of Markov chains
- Google Pagerank algorithm

Why solve systems of linear equations?

- Systems of linear equations forms a key part of *many* computations in scientific computing:

- Multi-dimensional Newton method

- Boundary-value problems in ODEs
- Discretization of PDEs

- Stationary solutions of Markov chains
- Google Pagerank algorithm

Matrix notation for linear systems

- notation using matrices:

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

Matrix notation for linear systems

- notation using matrices:

$$\mathsf{A}\,\mathbf{x} = \mathbf{b}$$

- $\mathsf{A} \in \mathbb{R}^{n \times n}$ is an $(n \times n)$ matrix
- With entries $(\mathsf{A})_{i,j} =: a_{i,j}$

Matrix notation for linear systems

- notation using matrices:

$$\mathsf{A}\,\mathbf{x} = \mathbf{b}$$

- $\mathsf{A} \in \mathbb{R}^{n \times n}$ is an $(n \times n)$ matrix
- With entries $(\mathsf{A})_{i,j} =: a_{i,j}$

- And $\mathbf{b} \in \mathbb{R}^n$ is an n-vecvtor

Matrix notation for linear systems

- notation using matrices:

$$\mathsf{A}\,\mathbf{x} = \mathbf{b}$$

- $\mathsf{A} \in \mathbb{R}^{n \times n}$ is an $(n \times n)$ matrix
- With entries $(\mathsf{A})_{i,j} =: a_{i,j}$

- And $\mathbf{b} \in \mathbb{R}^n$ is an n-vecvtor

- We want to find the unknown n-vector $\mathbf{x} \in \mathbb{R}^n$

Matrix notation II

- A $\mathbf{x} = \mathbf{b}$ written out:

$$\begin{gathered}
a_{1,1}x_1 + a_{1,2}x_2 + \cdots + a_{1,n}x_n = b_1 \\
a_{2,1}x_1 + a_{2,2}x_2 + \cdots + a_{2,n}x_n = b_2 \\
\vdots \\
a_{n,1}x_1 + a_{n,2}x_2 + \cdots + a_{n,n}x_n = b_n
\end{gathered}$$

Existence and uniqueness of solutions

- When is it possible to solve $A\,\mathbf{x} = \mathbf{b}$?

Existence and uniqueness of solutions

- When is it possible to solve $A\,\mathbf{x} = \mathbf{b}$?
- Linear algebra: solutions “usually” exist and are unique

Existence and uniqueness of solutions

- When is it possible to solve $\mathsf{A}\,\mathbf{x} = \mathbf{b}$?

- Linear algebra: solutions “usually” exist and are unique
- Namely when the **determinant** det $A \neq 0$

Existence and uniqueness of solutions

- When is it possible to solve $A\,\mathbf{x} = \mathbf{b}$?

- Linear algebra: solutions "usually" exist and are unique
- Namely when the **determinant** $\det A \neq 0$

- Or when the columns of A are **linearly independent**
- And several other equivalent conditions

Collaboration I

Let's temporarily suspend any prior knowledge we might have about how to solve linear equations.

Solving linear systems

1. With ideas from the course so far, how could you try to solve the linear system from two slides ago?
2. Can you see a way to possibly improve the resulting method?

Iterative methods for linear systems

- Create an **iterative** method

Iterative methods for linear systems

- Create an **iterative** method
- Idea: Solve the ith equation for x_i in terms of the others:

$$x_i = \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij} x_j \right)$$

Iterative methods for linear systems

- Create an **iterative** method
- Idea: Solve the ith equation for x_i in terms of the others:

$$x_i = \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij} x_j \right)$$

- How can we convert this into an iterative method?

Iterative methods for linear systems

- Create an **iterative** method
- Idea: Solve the ith equation for x_i in terms of the others:

$$x_i = \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij} x_j \right)$$

- How can we convert this into an iterative method?
- **Jacobi method**:

$$x_i^{(n+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij} x_j^{(n)} \right)$$

Jacobi method: Matrix version

- The Jacobi method corresponds to *splitting* A

Jacobi method: Matrix version

- The Jacobi method corresponds to *splitting* A
- Separate diagonal and non-diagonal elements: $\mathsf{A} = \mathsf{D} + \mathsf{L} + \mathsf{U} = \mathsf{D} + \mathsf{A}'$
- Where:
  - D are the diagonal entries a_{ii}
  - L are the strictly lower-triangular entries a_{ij} with $i > j$
  - U are the strictly upper-triangular entries a_{ij} with $i < j$

Jacobi method: Matrix version

- The Jacobi method corresponds to *splitting* A
- Separate diagonal and non-diagonal elements:
  $\mathsf{A} = \mathsf{D} + \mathsf{L} + \mathsf{U} = \mathsf{D} + \mathsf{A}'$
- Where:
  - D are the diagonal entries a_{ii}
  - L are the strictly lower-triangular entries a_{ij} with $i > j$
  - U are the strictly upper-triangular entries a_{ij} with $i < j$

- We need to solve $\left(\mathsf{D} + \mathsf{L} + \mathsf{U}\right)\mathbf{x} = \mathbf{b}$

Jacobi method: Matrix version II

- We need to solve $\left(\mathrm{D} + \mathrm{A}'\right)\mathbf{x} = \mathbf{b}$

Jacobi method: Matrix version II

- We need to solve $\left(\mathsf{D} + \mathsf{A}'\right) \mathbf{x} = \mathbf{b}$

- So try

$$\mathsf{D}\, \mathbf{x}_{n+1} = \mathbf{b} - \mathsf{A}'\, \mathbf{x}_n$$

Jacobi method: Matrix version II

- We need to solve $\left(\mathsf{D} + \mathsf{A}'\right)\mathbf{x} = \mathbf{b}$

- So try

$$\mathsf{D}\,\mathbf{x}_{n+1} = \mathbf{b} - \mathsf{A}'\,\mathbf{x}_n$$

- So

$$\mathbf{x}_{n+1} = \mathsf{D}^{-1}\left[\mathbf{b} - \mathsf{A}'\mathbf{x}_n\right]$$

Jacobi method: Matrix version II

- We need to solve $\left(\mathsf{D} + \mathsf{A}'\right) \mathbf{x} = \mathbf{b}$
- So try

$$\mathsf{D}\, \mathbf{x}_{n+1} = \mathbf{b} - \mathsf{A}'\, \mathbf{x}_n$$

- So

$$\mathbf{x}_{n+1} = \mathsf{D}^{-1} \left[\mathbf{b} - \mathsf{A}' \mathbf{x}_n\right]$$

- The point is that D is *easy to invert*!

Jacobi method: Convergence

- When should we stop the iteration?

Jacobi method: Convergence

- When should we stop the iteration?
- When we are “sufficiently close” to the solution

Jacobi method: Convergence

- When should we stop the iteration?
- When we are "sufficiently close" to the solution

- Define the **residual** $\mathbf{r}^{(n)} := A\mathbf{x}^{(n)} - \mathbf{b}$

Jacobi method: Convergence

- When should we stop the iteration?
- When we are "sufficiently close" to the solution

- Define the **residual** $\mathbf{r}^{(n)} := \mathsf{A}\mathbf{x}^{(n)} - \mathbf{b}$
- Continue the iteration until the residual is sufficiently small (below a pre-assigned tolerance)

Jacobi method: Convergence

- When should we stop the iteration?
- When we are "sufficiently close" to the solution

- Define the **residual** $\mathbf{r}^{(n)} := \mathsf{A}\mathbf{x}^{(n)} - \mathbf{b}$
- Continue the iteration until the residual is sufficiently small (below a pre-assigned tolerance)

- We need a way to define the "size" of a vector, i.e. a **norm**
- E.g. the 2-norm: $\|\mathbf{x}\|_2 := \sqrt{\sum_i x_i^2}$

Jacobi method: Convergence II

- The Jacobi method does *not* always converge

Jacobi method: Convergence II

- The Jacobi method does *not* always converge
- Look at the distance $\boldsymbol{\delta}^{(n)} := \mathbf{x}^{(n+1)} - \mathbf{x}^{(n)}$

Jacobi method: Convergence II

- The Jacobi method does *not* always converge
- Look at the distance $\boldsymbol{\delta}^{(n)} := \mathbf{x}^{(n+1)} - \mathbf{x}^{(n)}$
- We get

$$\boldsymbol{\delta}_n = -(D^{-1}A')\,\boldsymbol{\delta}_{n-1}$$

Jacobi method: Convergence II

- The Jacobi method does *not* always converge
- Look at the distance $\boldsymbol{\delta}^{(n)} := \mathbf{x}^{(n+1)} - \mathbf{x}^{(n)}$
- We get

$$\boldsymbol{\delta}_n = -(D^{-1}A')\,\boldsymbol{\delta}_{n-1}$$

- Hence to study convergence we need to understand iterated multiplication by the same matrix
- A *sufficient* condition for convergence is that A is **diagonally dominant**:

$$|a_{ii}| \geq \sum_{j \neq i} |a_{ij}|$$

Collaboration II

Improving the Jacobi method

1. Is there a way to modify Jacobi to use the new information that is generated?

Gauss–Seidel method

- Jacobi uses only components of $\mathbf{x}^{(n)}$ (previous generation)

Gauss–Seidel method

- Jacobi uses only components of $\mathbf{x}^{(n)}$ (previous generation)

- The **Gauss–Seidel** method modifies the Jacobi method
- We use newly-generated $x_i^{(n+1)}$ that are already available:

$$x_i^{(n+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j<i} a_{ij} x_j^{(n+1)} - \sum_{j>i} a_{ij} x_j^{(n)} \right)$$

Gauss–Seidel method

- Jacobi uses only components of $\mathbf{x}^{(n)}$ (previous generation)

- The **Gauss–Seidel** method modifies the Jacobi method
- We use newly-generated $x_i^{(n+1)}$ that are already available:

$$x_i^{(n+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j<i} a_{ij} x_j^{(n+1)} - \sum_{j>i} a_{ij} x_j^{(n)} \right)$$

- In matrix language this corresponds to

$$\mathsf{L}\, \mathbf{x}_{n+1} = \mathbf{b} - \left(\mathsf{D} + \mathsf{U}\right) \mathbf{x}_n$$

- L is again easy to invert (by forward substitution)

Computational complexity

- The **running time** of an algorithm is the number of "operations" that it uses

Computational complexity

- The **running time** of an algorithm is the number of “operations” that it uses
- We will only want the dominant term in the running time and its dependence on the input size

Computational complexity

- The **running time** of an algorithm is the number of "operations" that it uses
- We will only want the dominant term in the running time and its dependence on the input size

- We call this the **(computational) complexity** of the algorithm

Collaboration III

Computational complexity of iterative methods

1. What is the computational complexity of one iteration of Jacobi and Gauss–Seidel?

Computational complexity of iterative methods

- We have an $(n \times n)$ matrix A

Computational complexity of iterative methods

- We have an $(n \times n)$ matrix A
- Each row has 1 division, n multiplications and n additions

Computational complexity of iterative methods

- We have an $(n \times n)$ matrix A
- Each row has 1 division, n multiplications and n additions
- So approximately $2n$ operations per row
- Or $2n^2$ operations in total

Computational complexity of iterative methods

- We have an $(n \times n)$ matrix A
- Each row has 1 division, n multiplications and n additions
- So approximately $2n$ operations per row
- Or $2n^2$ operations in total

- Hence the complexity is $\mathcal{O}(n^2)$

Gaussian elimination

Elimination – row reduction

- How can we “exactly” solve a linear system in a finite number of operations?

Elimination – row reduction

- How can we “exactly” solve a linear system in a finite number of operations?
- **Idea**: **Eliminate** one variable
- **Solve** 1 equation for 1 variable in terms of other variables

Elimination – row reduction

- How can we "exactly" solve a linear system in a finite number of operations?
- **Idea**: **Eliminate** one variable
- **Solve** 1 equation for 1 variable in terms of other variables
- **Substitute** that variable into the other equations
- This gives $n-1$ equations in $n-1$ unknowns

Elimination – row reduction

- How can we "exactly" solve a linear system in a finite number of operations?
- **Idea**: **Eliminate** one variable
- **Solve** 1 equation for 1 variable in terms of other variables
- **Substitute** that variable into the other equations
- This gives $n-1$ equations in $n-1$ unknowns
- Recursively solve to reduce by one equation each time
- Until we end up with a single equation in a single unknown

Elimination – row reduction

- How can we “exactly” solve a linear system in a finite number of operations?
- **Idea**: **Eliminate** one variable
- **Solve** 1 equation for 1 variable in terms of other variables

- **Substitute** that variable into the other equations
- This gives $n-1$ equations in $n-1$ unknowns

- Recursively solve to reduce by one equation each time
- Until we end up with a single equation in a single unknown

Elimination II

- Number equations as E_1, E_2, etc.

Elimination II

- Number equations as E_1, E_2, etc.

- Eliminate x by adding multiple of E_1 to E_2
- Form new equation $E_2' := E_2 + \alpha E_1$

Elimination II

- Number equations as E_1, E_2, etc.

- Eliminate x by adding multiple of E_1 to E_2
- Form new equation $E_2' := E_2 + \alpha E_1$

- Choose α to make coefficient of x in the result equal to 0
- Gives **equivalent** system

Elimination II

- Act on matrix A *and* right-hand side **b** in *same* way

Elimination II

- Act on matrix A *and* right-hand side **b** in *same* way
- Form **augmented matrix** by **adjoining b** to A:

$$\left[\begin{array}{cc|c} 1 & 3 & 7 \\ 2 & -1 & 0 \end{array}\right]$$

Elimination II

- Act on matrix A *and* right-hand side **b** in *same* way
- Form **augmented matrix** by **adjoining b** to A:

$$\left[\begin{array}{cc|c} 1 & 3 & 7 \\ 2 & -1 & 0 \end{array}\right]$$

- Take

$$\alpha = -\frac{a_{2,1}}{a_{1,1}}$$

so that $a'_{2,1} = a_{2,1} + \alpha\, a_{1,1} = 0$

Elimination III

- Applying the above **row operation** gives

$$\left[\begin{array}{cc|c} 1 & 3 & 7 \\ 0 & -7 & -14 \end{array}\right]$$

Elimination III

- Applying the above **row operation** gives

$$\left[\begin{array}{cc|c} 1 & 3 & 7 \\ 0 & -7 & -14 \end{array}\right]$$

- 2nd row shows that $-7y = -14$, so $y = 2$
- Then **backsubstitute** to find x:

  $x + 3y = 7$, so $x + 6 = 7$, so $x = 1$

Elimination IV

- Generalise to matrix of any size

Elimination IV

- Generalise to matrix of any size

- Apply row operations until get **upper-triangular matrix** U
- i.e. all entries below main diagonal are 0

Elimination IV

- Generalise to matrix of any size

- Apply row operations until get **upper-triangular matrix** U
- i.e. all entries below main diagonal are 0

- Backsubstitution effectively does row operations to introduce zeros in upper triangular part

Several right-hand sides

- Suppose that we want to solve $\mathrm{A}\,\mathbf{x} = \mathbf{b}$ with the *same* matrix A for several right-hand sides $\mathbf{b}_i$
- We will execute the *same sequence* of row operations to reduce to upper-triangular form

Several right-hand sides

- Suppose that we want to solve $\mathrm{A}\,\mathbf{x} = \mathbf{b}$ with the *same* matrix A for several right-hand sides $\mathbf{b}_i$
- We will execute the *same sequence* of row operations to reduce to upper-triangular form
- Can we use this by recording only the row operations?

Row operations as elementary matrices

- Applying a row operation to an augmented matrix A produces a new augmented matrix
- Express as $L_1 A$ for a suitable matrix L_1:

Row operations as elementary matrices

- Applying a row operation to an augmented matrix A produces a new augmented matrix
- Express as L_1 A for a suitable matrix L_1:
- E.g. the row operation $E_2 \leftarrow E_2 + \alpha E_1$ is

$$\mathsf{L}_1 = \begin{pmatrix} 1 & 0 \\ \alpha & 1 \end{pmatrix}$$

Sequence of row operations

- Row reduction of A to an upper-triangular matrix U is a sequence of n row operations:

$$\mathsf{L}_n \mathsf{L}_{n-1} \cdots \mathsf{L}_1 A = \mathsf{U}$$

Sequence of row operations

- Row reduction of A to an upper-triangular matrix U is a sequence of n row operations:

$$\mathsf{L}_n \mathsf{L}_{n-1} \cdots \mathsf{L}_1 A = \mathsf{U}$$

- Hence

$$\mathsf{A} = \mathsf{L}_1^{-1} \mathsf{L}_2^{-1} \cdots \mathsf{L}_n^{-1} \mathsf{U}$$

- So $\mathsf{A} = \mathsf{L}\,\mathsf{U}$ where L is lower-triangular

Sequence of row operations

- Row reduction of A to an upper-triangular matrix U is a sequence of n row operations:

$$\mathsf{L}_n \mathsf{L}_{n-1} \cdots \mathsf{L}_1 A = \mathsf{U}$$

- Hence

$$\mathsf{A} = \mathsf{L}_1^{-1} \mathsf{L}_2^{-1} \cdots \mathsf{L}_n^{-1} \mathsf{U}$$

- So $\mathsf{A} = \mathsf{L}\,\mathsf{U}$ where L is lower-triangular
- Note that $\mathsf{L}_1^{-1} = \begin{pmatrix} 1 & 0 \\ -\alpha & 1 \end{pmatrix}$

Structure of L

- L_k has 1s on main diagonal
- And nonzero entries below diagonal only on kth column

$$(\mathsf{L}_k)_{i,k} = -\frac{a_{i,k}^{(k-1)}}{a_{k,k}^{(k-1)}}$$

where $\mathsf{A}^{(k-1)}$ is matrix after $(k-1)$ steps

Structure of L

- L_k has 1s on main diagonal
- And nonzero entries below diagonal only on kth column

$$(\mathsf{L}_k)_{i,k} = -\frac{a_{i,k}^{(k-1)}}{a_{k,k}^{(k-1)}}$$

where $\mathsf{A}^{(k-1)}$ is matrix after $(k-1)$ steps

- $\mathsf{L} = \mathsf{L}_n \cdots \mathsf{L}_1$
- Below-diagonal entries are those of the individual Ls!

LU factorization

- Any square matrix has an **LU factorization**
- To solve $A\mathbf{x} = \mathbf{b}$:
  - Factorise to get L and U with $LU = A$
  - We want to solve $LU\,\mathbf{x} = \mathbf{b}$
  - Solve $L\,\mathbf{y} = \mathbf{b}$
  - Then solve $U\,\mathbf{x} = \mathbf{y}$

LU factorization

- Any square matrix has an **LU factorization**
- To solve $A\mathbf{x} = \mathbf{b}$:
  - Factorise to get L and U with $LU = A$
  - We want to solve $LU\,\mathbf{x} = \mathbf{b}$
  - Solve $L\,\mathbf{y} = \mathbf{b}$
  - Then solve $U\,\mathbf{x} = \mathbf{y}$
- Solving triangular systems is easy using forward- or back-substitution

Pivoting

- In elimination algorithm may need to divide by $a_{i,i} = 0$
- Clearly we cannot do this

Pivoting

- In elimination algorithm may need to divide by $a_{i,i} = 0$
- Clearly we cannot do this
- Solution: **pivot**: swap with row with largest $a_{j,i}$

Pivoting

- In elimination algorithm may need to divide by $a_{i,i} = 0$
- Clearly we cannot do this
- Solution: **pivot**: swap with row with largest $a_{j,i}$
- We track which rows are swapped using a **permutation matrix** P:

$$\mathsf{PA} = \mathsf{LU}$$

Pivoting

- In elimination algorithm may need to divide by $a_{i,i} = 0$
- Clearly we cannot do this
- Solution: **pivot**: swap with row with largest $a_{j,i}$
- We track which rows are swapped using a **permutation matrix** P:

$$\mathsf{PA} = \mathsf{LU}$$

- In fact for numerical stability we should *always* pivot

Complexity of Gaussian elimination

- We subtract n elements of the 1st row from each $(n-1)$ other row, wth

Complexity of Gaussian elimination

- We subtract n elements of the 1st row from each $(n-1)$ other row, wth
- We subtract $n-1$ elements of the 1st row from each $(n-2)$ other rows

Complexity of Gaussian elimination

- We subtract n elements of the 1st row from each $(n-1)$ other row, wth
- We subtract $n-1$ elements of the 1st row from each $(n-2)$ other rows

- So the total number of operations is

$$2[n(n-1)+(n-1)(n-2)+(n-2)(n-3)+\cdots+2\cdot 1]$$

Complexity of Gaussian elimination

- We subtract n elements of the 1st row from each $(n-1)$ other row, wth
- We subtract $n-1$ elements of the 1st row from each $(n-2)$ other rows

- So the total number of operations is

$$2[n(n-1)+(n-1)(n-2)+(n-2)(n-3)+\cdots+2\cdot 1]$$

- This is approximately $\frac{2}{3}n^3$ (exercise)

Should we use direct or iterative methods?

- Gaussian elimination gives, in principle, the exact result
- Up to round-off error, but see also later

Should we use direct or iterative methods?

- Gaussian elimination gives, in principle, the exact result
- Up to round-off error, but see also later
- Iterative methods give an approximation, but possibly much faster: $\mathcal{O}(n^2)$, instead of $\mathcal{O}(n^3)$

Should we use direct or iterative methods?

- Gaussian elimination gives, in principle, the exact result
- Up to round-off error, but see also later
- Iterative methods give an approximation, but possibly much faster: $\mathcal{O}(n^2)$, instead of $\mathcal{O}(n^3)$

- Bear in mind these very different methods and their properties

Summary

- Solving linear systems

- **Iterative** methods: Jacobi & Gauss–Seidel
- With complexity $\mathcal{O}(n^2)$

- **Direct** method: Gaussian elimination
- Equivalent to factorization $\mathsf{A} = \mathsf{LU}$ or $\mathsf{PA} = \mathsf{LU}$
- Solve $\mathsf{A}\,\mathbf{x} = \mathbf{b}$ by solving $\mathsf{L}\,\mathbf{y} = \mathbf{b}$ and $\mathsf{U}\,\mathbf{x} = \mathbf{y}$