PHY604 Lecture 10

September 23, 2021

Review: Gaussian elimination

- Main general technique for solving **A x = b**
  - Does not involve matrix inversion
  - For “special” matrices, faster techniques may apply
- Involves forward-elimination and back-substitution
- Partial-pivoting:
  - Interchange of rows to move the one with the largest element in the current column to the top
  - (Full pivoting would allow for row and column swaps—more complicated)
- Scaled pivoting
  - Consider largest element relative to all entries in its row
  - Further reduces roundoff when elements vary in magnitude greatly
- Row echelon form: This is the upper-triangular form that the matrix is in after forward elimination

Review: Gaussian elimination for banded matrices

- Only need to do Gaussian elimination steps for *m* nonzero elements below given row (*m* is less than the number of diagonal bands)
- Example:

$$\left(\begin{array}{c}\boxed{\begin{array}{cccc}2 & 1 & 0 & 0\\ 3 & 4 & -5 & 0\end{array}}\\ \begin{array}{cccc}0 & -4 & 3 & 5\\ 0 & 0 & 1 & 3\end{array}\end{array}\right) \rightarrow \left(\begin{array}{cc}2 & \begin{array}{ccc}1 & 0 & 0\end{array}\\ \begin{array}{c}0\\0\end{array} & \boxed{\begin{array}{ccc}2.5 & -5 & 0\\ -4 & 3 & 5\end{array}}\\ 0 & \begin{array}{ccc}0 & 1 & 3\end{array}\end{array}\right) \rightarrow \left(\begin{array}{cc}\begin{array}{cc}2 & 1\\ 0 & 2.5\\ 0 & 0\\ 0 & 0\end{array} & \begin{array}{c}\begin{array}{cc}0 & 0\\ -5 & 0\end{array}\\ \boxed{\begin{array}{cc}-5 & 5\\ 1 & 3\end{array}}\end{array}\end{array}\right) \rightarrow \begin{pmatrix}2 & 1 & 0 & 0\\ 0 & 2.5 & -5 & 0\\ 0 & 0 & -5 & 5\\ 0 & 0 & 0 & 4\end{pmatrix}$$

Review: LU decomposition (Newman Ch. 6)

- Often happens that we would like to solve: $\mathbf{A}\mathbf{x}_i = \mathbf{v}_i$ for the same **A** but many **v**
  - For example, our implementation for the inverse
  - Wasteful to do Gaussian elimination over and over, we will always get the same row echelon matrix, just $\mathbf{v}_i$ will be different
  - Instead, we should keep track of operations we did to $\mathbf{v}_1$ and use them over and over
- For a general 4 x 4 matrix:

$$\mathbf{L}_0 \equiv \frac{1}{a_{00}} \begin{pmatrix} 1 & 0 & 0 & 0 \\ -a_{10} & a_{00} & 0 & 0 \\ -a_{20} & 0 & a_{00} & 0 \\ -a_{30} & 0 & 0 & a_{00} \end{pmatrix}, \quad \mathbf{L}_1 \equiv \frac{1}{b_{11}} \begin{pmatrix} b_{11} & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & -b_{21} & b_{11} & 0 \\ 0 & -b_{31} & 0 & b_{11} \end{pmatrix},$$

$$\mathbf{L}_2 \equiv \frac{1}{c_{22}} \begin{pmatrix} c_{22} & 0 & 0 & 0 \\ 0 & c_{22} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -c_{32} & c_{22} \end{pmatrix}, \quad \mathbf{L}_3 \equiv \frac{1}{d_{33}} \begin{pmatrix} d_{33} & 0 & 0 & 0 \\ 0 & d_{33} & 0 & 0 \\ 0 & 0 & d_{33} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\mathbf{L}_3\mathbf{L}_2\mathbf{L}_1\mathbf{L}_0\mathbf{A} = \mathbf{L}_3\mathbf{L}_2\mathbf{L}_1\mathbf{L}_0\mathbf{v}$$

Today’s lecture: More on linear and nonlinear algebra

- More on LU decomposition
- Iterative methods
- Eigensystems
- Nonlinear algebra: Roots and extrema of multivariable functions

Slightly different formulation of LU decomposition

- From the properties of upper triangular matrices (same holds for lower):
  - Product of two upper triangular matrices is an upper triangular matrix.
  - Inverse of an upper triangular matrix is an upper triangular matrix

- Consider the lower-diagonal matrix **L** and the upper-diagonal matrix **U**:

$$\mathbf{L} = \mathbf{L}_0^{-1}\mathbf{L}_1^{-1}\mathbf{L}_2^{-1}\mathbf{L}_3^{-1}, \quad \mathbf{U} = \mathbf{L}_3\mathbf{L}_2\mathbf{L}_1\mathbf{L}_0\mathbf{A}$$

- Then trivially: **LU** = **A**, so for **Ax** = **v**,, we can write **LUx** = **v**

Expression for L

- We can confirm that for our 4 x 4 example,

$$\mathbf{L}_0^{-1} = \begin{pmatrix} a_{00} & 0 & 0 & 0 \\ a_{10} & 1 & 0 & 0 \\ a_{20} & 0 & 1 & 0 \\ a_{30} & 0 & 0 & 1 \end{pmatrix}, \quad \mathbf{L}_1^{-1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & b_{11} & 0 & 0 \\ 0 & b_{21} & 1 & 0 \\ 0 & b_{31} & 0 & 1 \end{pmatrix}, \quad \mathbf{L}_2^{-1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & c_{22} & 0 \\ 0 & 0 & c_{32} & 1 \end{pmatrix}, \quad \mathbf{L}_3^{-1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & d_{33} \end{pmatrix}$$

- Multiplying together we get

$$\mathbf{L} = \begin{pmatrix} a_{00} & 0 & 0 & 0 \\ a_{10} & b_{11} & 0 & 0 \\ a_{20} & b_{21} & c_{22} & 0 \\ a_{30} & b_{31} & c_{32} & d_{33} \end{pmatrix}$$

Solving the equation with **L** and **U**

- Break into two steps:
  - 1. **Ly** = **v** can be solved by back substitution:

$$\begin{pmatrix} l_{00} & 0 & 0 & 0 \\ l_{10} & l_{11} & 0 & 0 \\ l_{20} & l_{21} & l_{22} & 0 \\ l_{30} & l_{31} & l_{32} & l_{33} \end{pmatrix} \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \end{pmatrix}$$

  - 2. Now solve **Ux** = **y** by back substitution:

$$\begin{pmatrix} u_{00} & u_{01} & u_{02} & u_{03} \\ 0 & u_{11} & u_{12} & u_{13} \\ 0 & 0 & u_{22} & u_{23} \\ 0 & 0 & 0 & u_{33} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

Some comments about **LU** decomposition

- Very common method for solving simultaneous equations

- Decomposition needs to be done once, then only back substitution is needed for different **v**

- In general, still may need to pivot
  - Every time you swap rows, you have to do the same to **L**
  - Need to perform the same sequence of swaps on **v**

Today’s lecture: More on linear and nonlinear algebra

- More on LU decomposition
- Iterative methods
- Eigensystems
- Nonlinear algebra: Roots and extrema of multivariable functions

Jacobi and Gauss-Seidel iterative methods

- Gaussian elimination is a **direct** method

- We can also use an **iterative** method
    - Choose an initial guess and converge to better and better guesses
    - E.g., Jacobi or Gauss Seidel methods
    - Can be much more efficient for very large systems
    - Often puts restrictions on the form of the matrix for guaranteed convergence

Jacobi iterative method

- Starting with a linear system:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ \vdots \qquad \vdots \qquad\qquad \vdots \quad &\quad \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n \end{aligned}$$

- Pick initial guesses **x**^{k}, solve equation *i* for *i*th unknown to get an improved guess:

$$\begin{aligned} x_1^{k+1} &= -\frac{1}{a_{11}}(a_{12}x_1^k + a_{13}x_2^k + \cdots + a_{1n}x_n^k - b_1) \\ x_2^{k+1} &= -\frac{1}{a_{22}}(a_{21}x_1^k + a_{23}x_2^k + \cdots + a_{2n}x_n^k - b_2) \\ \vdots \qquad & \qquad \vdots \qquad\qquad\qquad \vdots \qquad\qquad \vdots \\ x_n^{k+1} &= -\frac{1}{a_{nn}}(a_{n1}x_1^k + a_{n2}x_2^k + \cdots + a_{n,n-1}x_{n-1}^k - b_n) \end{aligned}$$

Jacobi iterative method

- We can write an element-wise formula for **x**:

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

- Or:

$$\mathbf{x}_i^{k+1} = \mathbf{D}^{-1} \left(\mathbf{b} - (\mathbf{A} - \mathbf{D}) \mathbf{x}^k \right)$$

  - Where **D** is a diagonal matrix constructed from the diagonal elements of **A**

- Convergence is guaranteed if matrix is diagonally dominant (but works in other cases):

$$a_{ii} > \sum_{j=1, j \neq i}^{N} |a_{ij}|$$

Today's lecture:
More on linear and nonlinear algebra

- More on LU decomposition
- Iterative methods
- Eigensystems
- Nonlinear algebra: Roots and extrema of multivariable functions

Eigenvalues and eigenvectors

- Very common matrix problem in physics
- Mostly concerned with real symmetric matrices, or Hermitian matrices
- For a symmetric matrix **A**, an eigenvector **v**$_i$ satisfies:

$$\mathbf{A}\mathbf{v}_i = \lambda_i \mathbf{v}_i$$

  - λ_i are the eigenvalues
- Eigenvectors are orthogonal, and we will assume they are normalized:

$$\mathbf{v}_i \cdot \mathbf{v}_j = \delta_{ij}$$

- Combining eigenvectors into matrix **V**, and eigenvalues into diagonal matrix **D**:

$$\mathbf{AV} = \mathbf{VD}$$

QR algorithm for calculating eigenvalues/eigenvectors

- We will focus on real, symmetric, square **A**
- Makes use of QR decomposition to obtain **V** and **D**
    - Same idea as LU decomposition
    - Write **A** as a product of orthogonal matrix **Q**, and upper-triangular matrix **R**
    - Any square matrix can be written that way

- 1. Break **A** down into QR decomposition: $\mathbf{A} = \mathbf{Q}_1\mathbf{R}_1$
- 2. Multiply on the left by $\mathbf{Q}_1^{\mathrm{T}}$:

$$\mathbf{Q}_1^{\mathrm{T}}\mathbf{A} = \mathbf{Q}_1^{\mathrm{T}}\mathbf{Q}_1\mathbf{R}_1 = \mathbf{R}_1$$

- Note that since **Q** is orthogonal, $\mathbf{Q}^{\mathrm{T}}=\mathbf{Q}^{-1}$

QR decomposition

- 3. Now we define a new matrix, product of **Q**$_1$ and **R**$_1$ in reverse order:

$$\mathbf{A}_1 = \mathbf{R}_1\mathbf{Q}_1$$

  - Combine with step 2 to get:

$$\mathbf{A}_1 = \mathbf{Q}_1^{\mathrm{T}}\mathbf{A}\mathbf{Q}_1$$

- 4. Repeat the process, find QR decomposition of **A**$_1$:

$$\mathbf{A}_2 = \mathbf{R}_2\mathbf{Q}_2 = \mathbf{Q}_2^{\mathrm{T}}\mathbf{A}_1\mathbf{Q}_2 = \mathbf{Q}_2^{\mathrm{T}}\mathbf{Q}_1^{\mathrm{T}}\mathbf{A}\mathbf{Q}_1\mathbf{Q}_2$$

  - And so on:

$$\begin{aligned}
\mathbf{A}_1 &= \mathbf{Q}_1^{\mathrm{T}}\mathbf{A}\mathbf{Q}_1 \\
\mathbf{A}_2 &= \mathbf{Q}_2^{\mathrm{T}}\mathbf{Q}_1^{\mathrm{T}}\mathbf{A}\mathbf{Q}_1\mathbf{Q}_2 \\
\mathbf{A}_3 &= \mathbf{Q}_3^{\mathrm{T}}\mathbf{Q}_2^{\mathrm{T}}\mathbf{Q}_1^{\mathrm{T}}\mathbf{A}\mathbf{Q}_1\mathbf{Q}_2\mathbf{Q}_3 \\
&\vdots \\
\mathbf{A}_k &= (\mathbf{Q}_k^{\mathrm{T}}\ldots\mathbf{Q}_1^{\mathrm{T}})\mathbf{A}(\mathbf{Q}_1\ldots\mathbf{Q}_k)
\end{aligned}$$

Eigenvalues and eigenvectors from QR decomposition

- If you continue this process long enough, the matrix **A**$_k$ will eventually become diagonal:

$$\mathbf{A}_k \simeq \mathbf{D}$$

- Continue until the off-diagonal elements are below some accuracy
- Eigenvector matrix is given by:

$$\mathbf{V} = \mathbf{Q}_1\mathbf{Q}_2\mathbf{Q}_3 \ldots \mathbf{Q}_k = \prod_{i=1}^{k} \mathbf{Q}_i$$

- **V** Orthogonal since the product of orthogonal matrices is orthogonal. Then:

$$\mathbf{D} = \mathbf{A}_k = \mathbf{V}^{\mathrm{T}}\mathbf{A}\mathbf{V}$$

- So:

$$\mathbf{A}\mathbf{V} = \mathbf{V}\mathbf{D}$$

How do we do the QR decomposition?

- Think of the matrix as a set of *N* columns:

$$\mathbf{A} = \begin{pmatrix} | & | & | & \dots \\ \mathbf{a}_0 & \mathbf{a}_1 & \mathbf{a}_2 & \dots \\ | & | & | & \dots \end{pmatrix}$$

- Now define two new sets of vectors:

$$\mathbf{u}_0 = \mathbf{a}_0, \qquad \mathbf{q}_0 = \frac{\mathbf{u}_0}{|\mathbf{u}_0|}$$

$$\mathbf{u}_1 = \mathbf{a}_1 - (\mathbf{q}_0 \cdot \mathbf{a}_1)\mathbf{q}_0, \qquad \mathbf{q}_1 = \frac{\mathbf{u}_1}{|\mathbf{u}_1|}$$

$$\mathbf{u}_2 = \mathbf{a}_2 - (\mathbf{q}_0 \cdot \mathbf{a}_2)\mathbf{q}_0 - (\mathbf{q}_1 \cdot \mathbf{a}_2)\mathbf{q}_1, \qquad \mathbf{q}_1 = \frac{\mathbf{u}_2}{|\mathbf{u}_2|}$$

$$\vdots \qquad \qquad \vdots$$

(Gram-Schmidt orthogonalization!)

How do we do the QR decomposition?

- General formula for **u**$_i$ and **q**$_i$:

$$\mathbf{u}_i = \mathbf{a}_i - \sum_{j=0}^{i-1} (\mathbf{q}_j \cdot \mathbf{a}_i)\mathbf{q}_j, \qquad \mathbf{q}_i = \frac{\mathbf{u}_i}{|\mathbf{u}_i|}$$

- We can show that the **q** vectors are orthonormal:

$$\mathbf{q}_i \cdot \mathbf{q}_j = \delta_{ij}$$

- Now we rearrange the definitions of the vectors:

$$\begin{aligned}
\mathbf{a}_0 &= |\mathbf{u}_0|\mathbf{q}_0, \\
\mathbf{a}_1 &= |\mathbf{u}_1|\mathbf{q}_1 + (\mathbf{q}_0 \cdot \mathbf{a}_1)\mathbf{q}_0 \\
\mathbf{a}_2 &= |\mathbf{u}_2|\mathbf{q}_2 + (\mathbf{q}_0 \cdot \mathbf{a}_2)\mathbf{q}_0 + (\mathbf{q}_1 \cdot \mathbf{a}_2)\mathbf{q}_1
\end{aligned}$$

How do we do the QR decomposition?

- Finally write all the equations as a single matrix equation:

$$\mathbf{A} = \begin{pmatrix} | & | & | & \dots \\ \mathbf{a}_0 & \mathbf{a}_1 & \mathbf{a}_2 & \dots \\ | & | & | & \dots \end{pmatrix} = \begin{pmatrix} | & | & | & \dots \\ \mathbf{q}_0 & \mathbf{q}_1 & \mathbf{q}_2 & \dots \\ | & | & | & \dots \end{pmatrix} \begin{pmatrix} |\mathbf{u}_0| & \mathbf{q}_0 \cdot \mathbf{a}_1 & \mathbf{q}_0 \cdot \mathbf{a}_2 & \dots \\ 0 & |\mathbf{u}_1| & \mathbf{q}_1 \cdot \mathbf{a}_2 & \dots \\ 0 & 0 & |\mathbf{u}_2| & \dots \end{pmatrix}$$

- Our QR decomposition is thus

$$\mathbf{Q} = \begin{pmatrix} | & | & | & \dots \\ \mathbf{q}_0 & \mathbf{q}_1 & \mathbf{q}_2 & \dots \\ | & | & | & \dots \end{pmatrix}, \qquad \mathbf{R} = \begin{pmatrix} |\mathbf{u}_0| & \mathbf{q}_0 \cdot \mathbf{a}_1 & \mathbf{q}_0 \cdot \mathbf{a}_2 & \dots \\ 0 & |\mathbf{u}_1| & \mathbf{q}_1 \cdot \mathbf{a}_2 & \dots \\ 0 & 0 & |\mathbf{u}_2| & \dots \end{pmatrix}$$

- **Q** is orthogonal since the columns are orthonormal
- **R** is upper triangular

QR decomposition algorithm:

- For a give *N* x *N* starting matrix **A**:

- 1. Create an *N* x *N* array to hold **V**; initialize as identity
- 2. Calculate QR decomposition **A** = **QR**
- 3. Update **A** with new value **A** = **RQ**
- 4. Multiply **V** on the RHS with **Q**
- 5. Check off-diagonal elements of **A**. If they are less than some tolerance, we are done. Otherwise go back to 2.

Lanczos method (see Pang Sec. 5.9)

- Iterative scheme that works especially well for sparse matrices, or when we only need a few eigenvalues/vectors
  - Often used for “exact diagonalization” calculations in condensed matter physics
- Assume that **H** is an *n* x *n* real symmetric matrix
- In a similar was as we discussed for QR decomposition, we can “tridiagonalize” *m* x *m* subsets of the matrix via:

$$\mathbf{O}^{\mathrm{T}}\mathbf{H}\mathbf{O} = \widetilde{\mathbf{H}}$$

- Where **O** is an *n* x *m* matrix with columns:

$$\mathbf{v}_k = \frac{\mathbf{u}_k}{|\mathbf{u}_k|}$$

Can be random (normalized) vector for first step

0 for first step

- And: $\mathbf{u}_{k+1} = \mathbf{H}\mathbf{v}_k - (\mathbf{v}_k^{\mathrm{T}}\mathbf{H}\mathbf{v}_k)\mathbf{v}_k - (\mathbf{v}_{k-1}^{\mathrm{T}}\mathbf{H}\mathbf{v}_k)\mathbf{v}_{k-1}$

Lanczos method (see Pang Sec. 5.9)

- The eigenvalues of $\widetilde{\mathbf{H}}$ can be shown to be approximations of the ones of **H** with the largest magnitude
- Use standard methods to diagonalize: $\widetilde{\mathbf{H}}\widetilde{\mathbf{x}}_k = \lambda_k \widetilde{\mathbf{x}}_k$
- Approximate eigenvectors of **H** are: $\mathbf{x}_k \simeq \mathbf{O}\widetilde{\mathbf{x}}_k$
- Approximation can be improved by constructing a new initial state:

$$\mathbf{u}_0 = \sum_{k=1}^{m} c_k \widetilde{\mathbf{x}}_k$$

Need to choose c_k

- Iterative process will eventually lead to m eigenvectors of **H** corresponding to the eigenvalues with largest magnitude

Lanczos for many-body quantum systems (see Pang Sec. 5.9)

- Say that we have some basis functions, and express the Hamiltonian as a matrix in that basis
    - We know that the Hilbert space increases exponentially
    - But we may not be interested in all the eigenvalues, just a few low energy ones

- We introduce the matrix: $\mathbf{G} = (\mathbf{H} - \mu\mathbf{I})^{-1}$

- Solve this with the Lanczos method to get eigenvectors with eigenvalues near μ:

$$\mathbf{G}\mathbf{x}_k = \frac{1}{\lambda_k - \mu}\mathbf{x}_k$$

Libraries for linear algebra: BLAS (basic linear algebra subroutines)

- These are the standard building blocks (API) of linear algebra on a computer (Fortran and C)
- Most linear algebra packages formulate their operations in terms of BLAS operations
- Three levels of functionality:
  - Level 1: vector operations ($\alpha\mathbf{x} + \mathbf{y}$)
  - Level 2: matrix-vector operations ($\alpha\mathbf{A}\,\mathbf{x} + \beta\,\mathbf{y}$)
  - Level 3: matrix-matrix operations ($\alpha\mathbf{A}\,\mathbf{B} + \beta\,\mathbf{C}$)
- Available on pretty much every platform
  - Some compilers provide specially optimized BLAS libraries (-lblas) that take great advantage of the underlying processor instructions
  - ATLAS: automatically tuned linear algebra software

Libraries for linear algebra: LAPACK

- The standard for linear algebra
- Built upon BLAS
- Routines named in the form xyyzzz
  - x refers to the data type (s/d are single/double precision floating, c/z are single/double complex)
  - yy refers to the matrix type
  - zzz refers to the algorithm (e.g. sgebrd = single precision bi-diagonal reduction of a general matrix)

- Routines: http://www.netlib.org/lapack/

Libraries for linear algebra: Python

- Basic methods in numpy.linalg (based on BLAS and LAPACK)
  - https://numpy.org/doc/stable/reference/routines.linalg.html
  - Has a matrix type built from the array class
  - * operator works element by element for arrays but does matrix product for matrices
  - As of python 3.5, @ operator will do matrix multiplication for NumPy arrays
  - Vectors are automatically converted into 1×N or N×1 matrices
  - Matrix objects cannot be > rank 2
  - Matrix has .H (or .T), .I, and .A attributes (transpose, inverse, as array)

- More general stuff in SciPy (scipy.linalg)
  - http://docs.scipy.org/doc/scipy/reference/linalg.html

Today's lecture: More on linear and nonlinear algebra

- More on LU decomposition
- Iterative methods
- Eigensystems
- Nonlinear algebra: Roots and extrema of multivariable functions

Multivariate Newton’s method

- We can generalize Newton’s method for equations with several variables
  - Can be used when we no longer have a linear system
  - Cast the problem as one of root finding
- Consider the vector function: $\mathbf{f}(\mathbf{x}) = \begin{bmatrix} f_1(\mathbf{x}) & f_1(\mathbf{x}) & \ldots & f_N(\mathbf{x}) \end{bmatrix}$
- Where the unknowns are: $\mathbf{x} = \begin{bmatrix} x_1 & x_1 & \ldots & x_N \end{bmatrix}$
- Revised guess from initial guess **x**$^{(0)}$: $\mathbf{x}_1 = \mathbf{x}_0 - \mathbf{f}(\mathbf{x}_0)\mathbf{J}^{-1}(\mathbf{x}_0)$
  - **J**$^{-1}$ is the inverse of the Jacobian matrix:

$$J_{ij}(\mathbf{x}) = \frac{\partial f_i(\mathbf{x})}{\partial x_i}$$

- To avoid taking the inverse at each step, solve with Gaussian substitution:

$$\mathbf{J}\delta\mathbf{x}^k = -\mathbf{f}(\mathbf{x}^k)$$

Example: Lorenz model (Garcia Sec. 4.3)

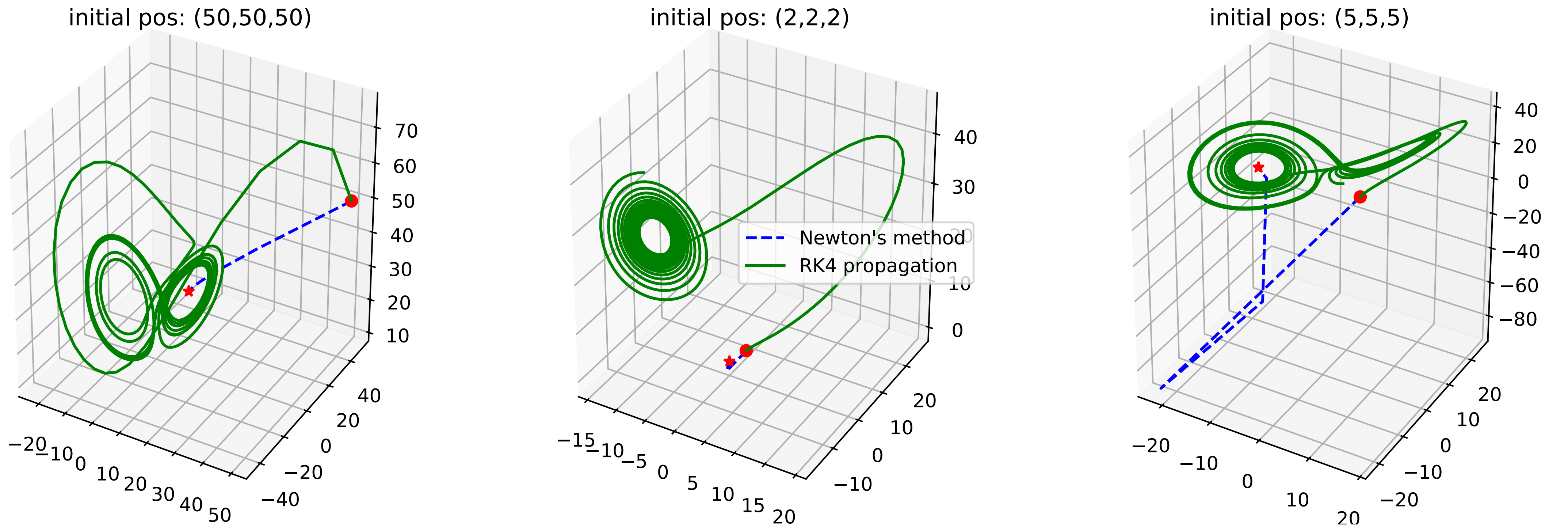
- Lorenz system:

$$\frac{dx}{dt} = \sigma(y - x)$$
$$\frac{dy}{dt} = rx - y - xz$$
$$\frac{dz}{dt} = xy - bz$$

- σ, *r*, and *b* are positive constants
- If we want steady-state, we can propagate with, e.g., 4th order RK
- Steady-state directly given by roots of Lorenz system:

$$\mathbf{f}(x,y,z) = \begin{pmatrix} \sigma(y-x) \\ rx - y - xz \\ xy - bz \end{pmatrix} = 0 \qquad \mathbf{J} = \begin{pmatrix} -\sigma & \sigma & 0 \\ r - z & -1 & -x \\ y & x & -b \end{pmatrix}$$

Lorenz model steady-state: Newton versus 4th order RK



Newton's method: Extrema of multivariable functions

- To get extrema of *g*(**x**), Must solve the nonlinear equation:

$$\mathbf{f}(\mathbf{x}) = \nabla g(\mathbf{x}) = 0$$

- Need to ensure that *g*(**x**) continually decreases if we want the minima, or continually increases if we want the maximum, modify the Jacobian in Newton's method

$$J_{ij}(\mathbf{x}) = \frac{\partial f_i(\mathbf{x})}{\partial x_i} + \mu\delta_{ij}$$

  - μ is small and positive to make sure **A** is positive definite:

- Popular scheme involves updating μ with each step: $\mathbf{w}^{\mathrm{T}}\mathbf{A}\mathbf{w} \geq 0 \quad \forall\, \mathbf{w} \neq 0$

$$\mathbf{A}_k = \mathbf{A}_{k-1} + \frac{\mathbf{y}\mathbf{y}^{\mathrm{T}}}{\mathbf{y}^{\mathrm{T}}\mathbf{w}} - \frac{\mathbf{A}_{k-1}\mathbf{w}\mathbf{w}^{\mathrm{T}}\mathbf{A}_{k-1}}{\mathbf{w}^{\mathrm{T}}\mathbf{A}_{k-1}\mathbf{w}}, \quad \mathbf{w} = \mathbf{x}_k - \mathbf{x}_{k-1}, \quad \mathbf{y} = \mathbf{f}_k - \mathbf{f}_{k-1}$$

- BFGS method (Broyden, Fletcher, Goldfarb, Shanno)

Steepest descent

- Used for finding roots, minima, or maxima of functions of several variables
- Based on the idea of moving downhill with each iteration, i.e., opposite to the gradient
  - If current position is **x**$_n$, next step is:

$$x_{n+1} = -x_n - \alpha_n \nabla f(x_n)$$

- Determine the step size α such that we reach the line minimum in direction of the gradient:

$$\frac{d}{d\alpha_n} f[x_{n+1}(\alpha_n)] = -\nabla f(x_{n+1}) \cdot \nabla f(x_n) = 0$$

- Find root of function of α :

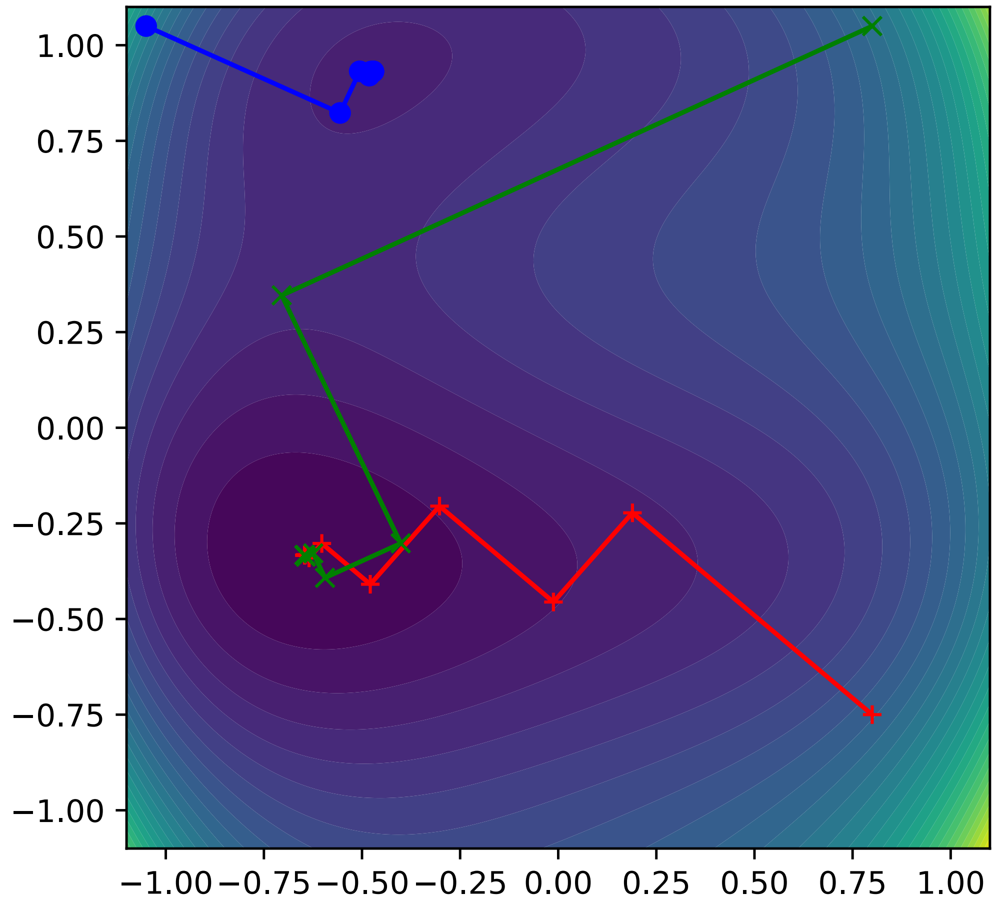
$$g(\alpha) = \nabla f[x_{n+1}(\alpha)] \cdot \nabla f(x_n) = 0$$

Steepest descent example

(From Stickler and Schachinger: Basic Concepts in Computational Physics)

- Consider the function:

$$f(x, y) = \cos(2x) + \sin(4y) + \exp(1.5x^2 + 0.7y^2) + 2x$$



Comments on steepest descent

- Rather slow due to orthogonality of subsequent search directions

- Can only find local minimum closest to starting point
  - Not global minimum

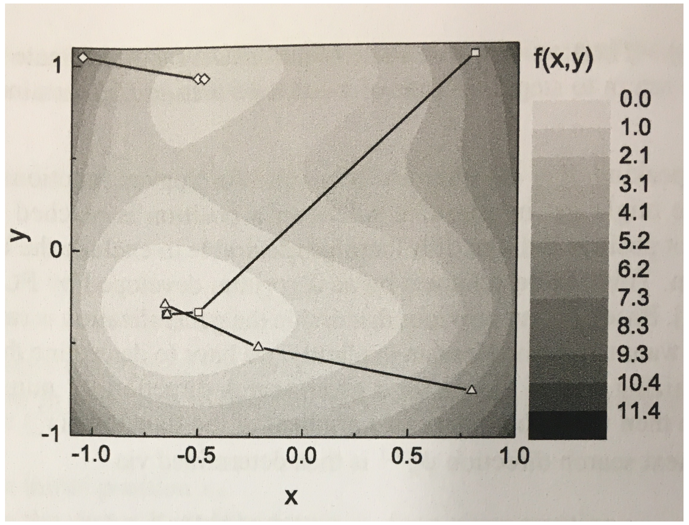
- Convergence rate is highly affected by choice of initial position

- Very simple method, works in space of arbitrary dimensions

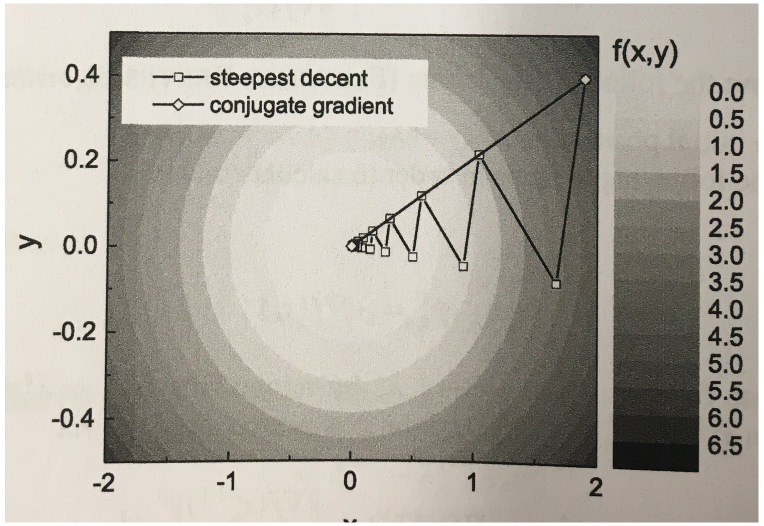
Conjugate gradients method

- Based on the definition of *N* orthogonal search directions in *N* dimensional space
- Consider function in "quadratic" form: $f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^{\mathrm{T}}\mathbf{A}\mathbf{x} - \mathbf{b}^{\mathrm{T}}\mathbf{x} + c$
- For functions in this form, CG method will converge in at most *N* steps
  - More steps for general functions, still more efficient than steepest descent
- Formulation is a bit complex, see readings

Previous slide example



$f(x,y) = x^2 + 10y^2$



Stickler and Schachinger

After class tasks

- Homework 2 posted due Sept. 30
- No office hours today

- Readings:
  - Newman Ch. 6
  - Garcia Ch. 4
  - Pang Ch. 5

  - “An Introduction to the Conjugate Gradient Method Without the Agonizing Pain,” Jonathan Richard Shewchuk