

# PHY604 Lecture 12

October 17, 2023

# Review: Solving the equation with L and U

- Break into two steps:
  - 1.  $\mathbf{Ly} = \mathbf{v}$  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  $\mathbf{Ux} = \mathbf{y}$  by back substitution:

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

# Review: Jacobi iterative method

- We can write an element-wise formula for  $\mathbf{x}$ :

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

- Or:

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

- Where  $\mathbf{D}$  is a diagonal matrix constructed from the diagonal elements of  $\mathbf{A}$
- Convergence is guaranteed if matrix is diagonally dominant (but works in other cases):

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

# Review: QR algorithm for calculating eigenvalues/eigenvectors

- We will focus on real, symmetric, square  $\mathbf{A}$
- Makes use of **QR decomposition** to obtain  $\mathbf{V}$  and  $\mathbf{D}$ 
  - Same idea as LU decomposition
  - Write  $\mathbf{A}$  as a product of **orthogonal matrix  $\mathbf{Q}$** , and **upper-triangular matrix  $\mathbf{R}$**
  - Any square matrix can be written that way
- 1. Break  $\mathbf{A}$  down into QR decomposition:  $\mathbf{A} = \mathbf{Q}_1 \mathbf{R}_1$
- 2. Multiply on the left by  $\mathbf{Q}_1^T$  :

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

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

# Review:QR decomposition

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

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

- Combine with step 2 to get:

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

- 4. Repeat the process, find QR decomposition of  $\mathbf{A}_1$ :

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

- And so on:

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

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

$$\mathbf{A}_3 = \mathbf{Q}_3^T \mathbf{Q}_2^T \mathbf{Q}_1^T \mathbf{A} \mathbf{Q}_1 \mathbf{Q}_2 \mathbf{Q}_3$$

⋮

$$\mathbf{A}_k = (\mathbf{Q}_k^T \dots \mathbf{Q}_1^T) \mathbf{A} (\mathbf{Q}_1 \dots \mathbf{Q}_k)$$

# Review: Eigenvalues and eigenvectors from QR decomposition

- If you continue this process long enough, the matrix  $\mathbf{A}_k$  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:

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

- $\mathbf{V}$  Orthogonal since the product of orthogonal matrices is orthogonal.  
Then:

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

- So:

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

# Today's lecture:

## Linear/Nonlinear algebra and FFTs

- Finish discussing QR decomposition
- Nonlinear algebra: Roots and extrema of multivariable functions

# 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:

$$\mathbf{u}_0 = \mathbf{a}_0,$$

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

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

⋮

$$\mathbf{q}_0 = \frac{\mathbf{u}_0}{|\mathbf{u}_0|}$$

$$\mathbf{q}_1 = \frac{\mathbf{u}_1}{|\mathbf{u}_1|}$$

$$\mathbf{q}_2 = \frac{\mathbf{u}_2}{|\mathbf{u}_2|}$$

⋮

(Gram-Schmidt orthogonalization!)

# How do we do the QR decomposition?

- General formula for  $\mathbf{u}_i$  and  $\mathbf{q}_i$ :

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

- We can show that the  $\mathbf{q}$  vectors are orthonormal:

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

- Now we rearrange the definitions of the vectors:

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

$$\mathbf{a}_1 = |\mathbf{u}_1| \mathbf{q}_1 + (\mathbf{q}_0 \cdot \mathbf{a}_1) \mathbf{q}_0$$

$$\mathbf{a}_2 = |\mathbf{u}_2| \mathbf{q}_2 + (\mathbf{q}_0 \cdot \mathbf{a}_2) \mathbf{q}_0 + (\mathbf{q}_1 \cdot \mathbf{a}_2) \mathbf{q}_1$$

# 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

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

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

# QR decomposition algorithm:

- For a give  $N \times N$  starting matrix  $\mathbf{A}$ :
  - 1. Create an  $N \times N$  array to hold  $\mathbf{V}$ ; initialize as identity
  - 2. Calculate QR decomposition  $\mathbf{A} = \mathbf{Q}\mathbf{R}$
  - 3. Update  $\mathbf{A}$  with new value  $\mathbf{A} = \mathbf{R}\mathbf{Q}$
  - 4. Multiply  $\mathbf{V}$  on the RHS with  $\mathbf{Q}$
  - 5. Check off-diagonal elements of  $\mathbf{A}$ . If they are less than some tolerance, we are done. Otherwise go back to 2.

# 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 (<http://www.netlib.org/blas/>)
  - See ([https://en.wikipedia.org/wiki/Basic\\_Linear\\_Algebra\\_Subprograms](https://en.wikipedia.org/wiki/Basic_Linear_Algebra_Subprograms))
  - 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  $1 \times N$  or  $N \times 1$  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:

## Linear/Nonlinear algebra and FFTs

- Finish discussing QR decomposition
- Nonlinear algebra: Roots and extrema of multivariable functions

# Nonlinear algebra

- We have discussed systems of equations of the form  $\mathbf{Ax} = \mathbf{b}$ 
  - $\mathbf{A}$  is a matrix of constants
- Nonlinear algebra: More generally equations of the form  $\mathbf{f(x)} = \mathbf{b}$
- What would we like to do?
  - Solve equations: Often cast them as finding the root of  $\mathbf{f(x)} - \mathbf{b} = \mathbf{0}$
  - Find minima and maxima: Could be cast as roots of derivatives of  $\mathbf{f}$

# 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:  $\mathbf{f}(\mathbf{x}) = [f_1(\mathbf{x}) \quad f_2(\mathbf{x}) \quad \dots \quad f_N(\mathbf{x})]$
- Where the unknowns are:  $\mathbf{x} = [x_1 \quad x_2 \quad \dots \quad x_N]$
- Revised guess from initial guess  $\mathbf{x}^{(0)}$ :  $\mathbf{x}_1 = \mathbf{x}_0 - \mathbf{f}(\mathbf{x}_0) \mathbf{J}^{-1}(\mathbf{x}_0)$ 
  - $\mathbf{J}^{-1}$  is the inverse of the Jacobian matrix:

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

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

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

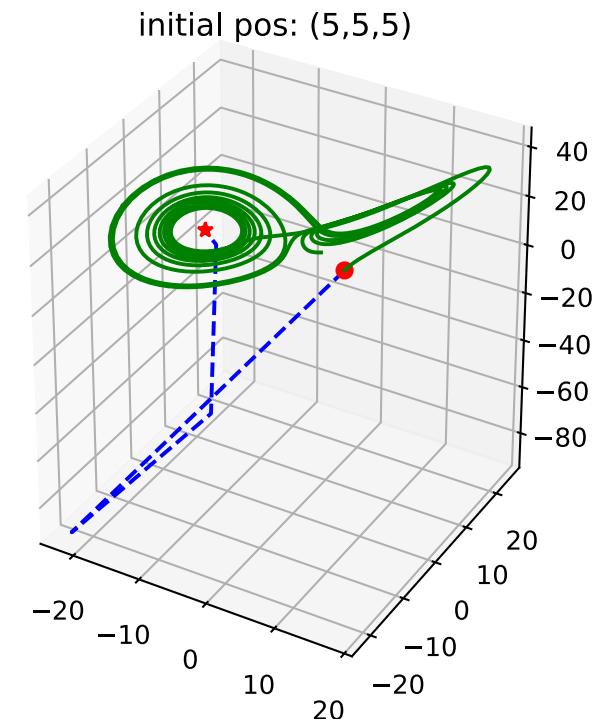
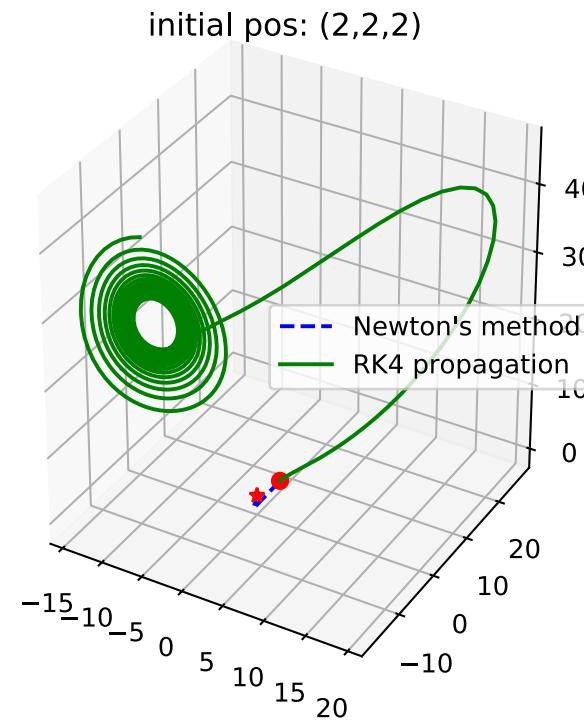
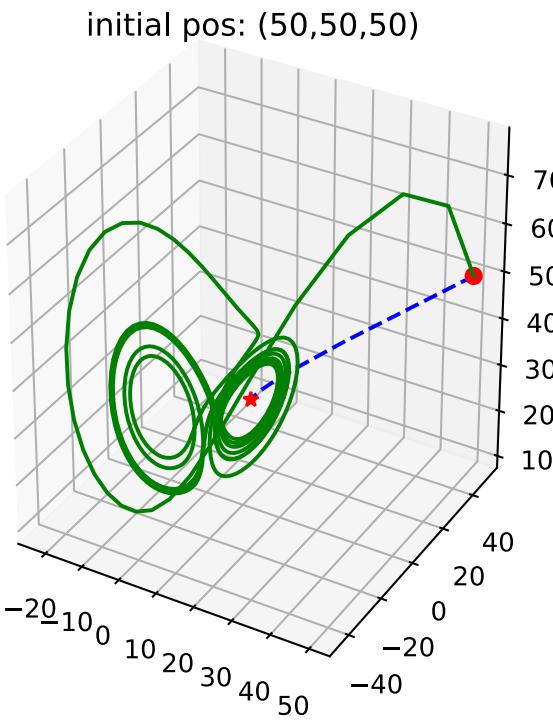
# Example: Lorenz model

(Garcia Sec. 4.3)

- Lorenz system:  
$$\frac{dx}{dt} = \sigma(y - x)$$
  
$$\frac{dy}{dt} = rx - y - xz$$
  
$$\frac{dz}{dt} = xy - bz$$
- $\sigma$ ,  $r$ , and  $b$  are positive constants
- If we want steady-state, we can propagate with, e.g., 4<sup>th</sup> order RK
- **OR:** 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 4<sup>th</sup> order RK



# 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  $\mathbf{x}_n$ , next step is:

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \alpha_n \nabla f(\mathbf{x}_n)$$

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

$$\frac{d}{d\alpha_n} f[\mathbf{x}_{n+1}(\alpha_n)] = -\nabla f(\mathbf{x}_{n+1}) \cdot \nabla f(\mathbf{x}_n) = 0$$

- Find root of function of  $\alpha$ :

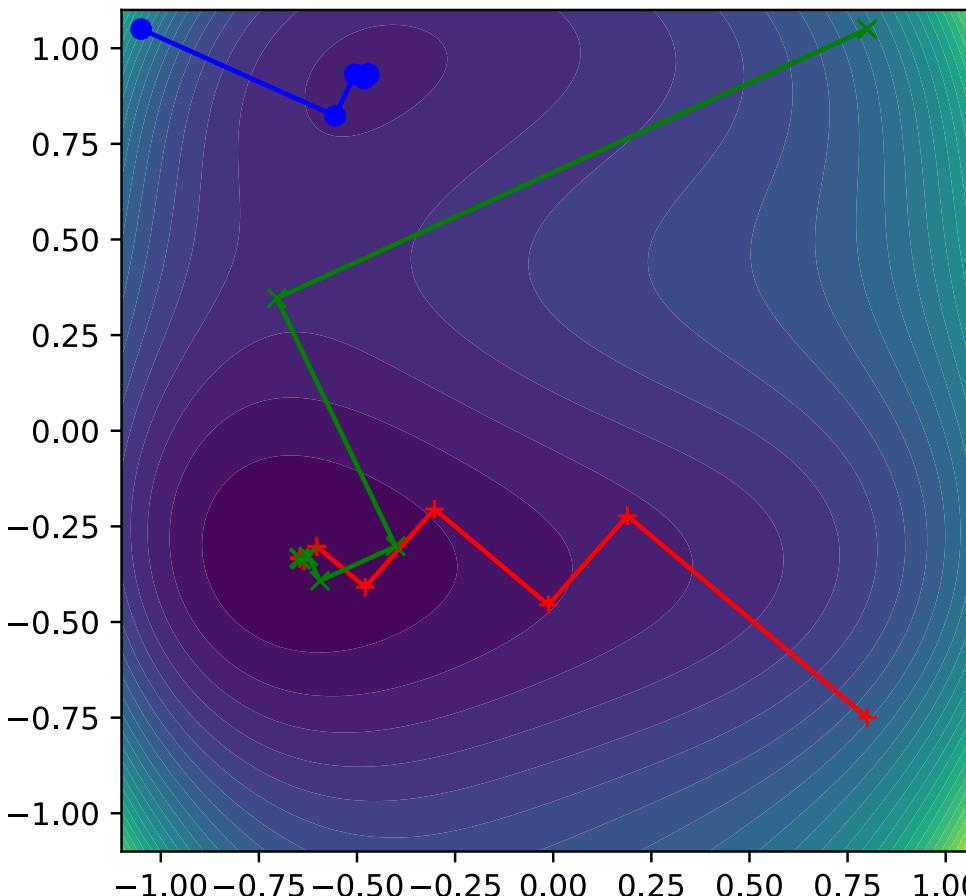
$$g(\alpha) = \nabla f[\mathbf{x}_{n+1}(\alpha)] \cdot \nabla f(\mathbf{x}_n) = 0$$

# Steepest descent example

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

- Consider the function:

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



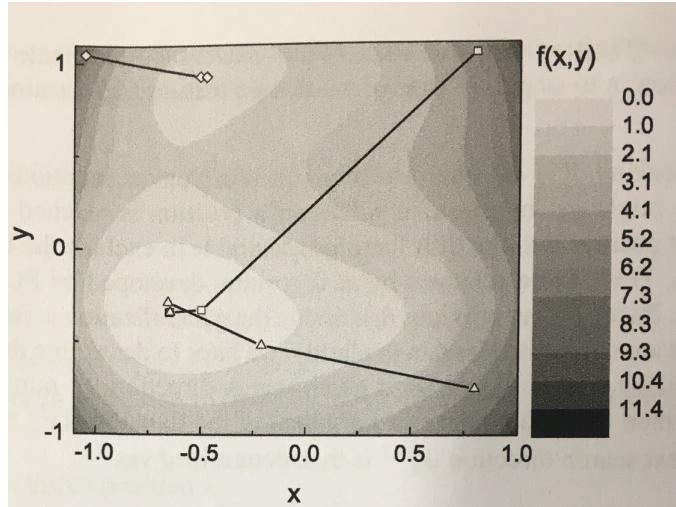
# Comments on steepest descent

- 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

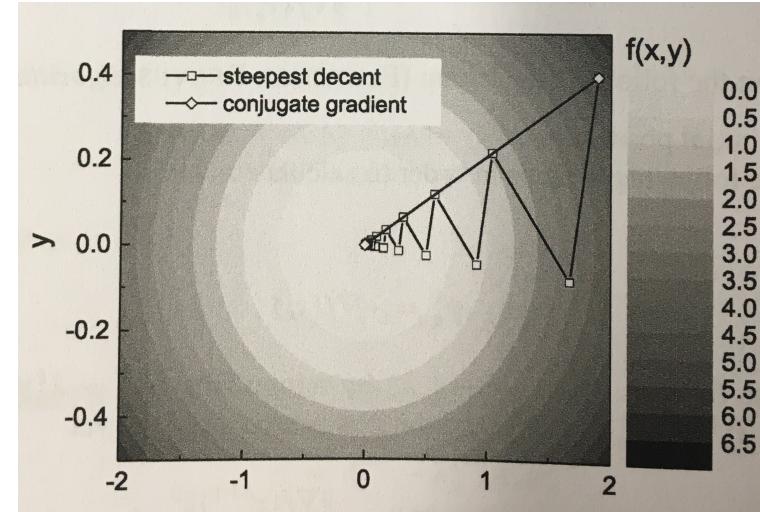
# Conjugate gradients method

- Based on the definition of  $N$  orthogonal search directions in  $N$  dimensional space
- Consider function in “quadratic” form:  $f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T \mathbf{A}\mathbf{x} - \mathbf{b}^T \mathbf{x} + c$
- 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



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



# After class tasks

- Homework 3 due Oct. 26
- Readings:
  - Linear/nonlinear equations:
    - Newman Ch. 6
    - Garcia Ch. 4
    - Pang Ch. 5
    - “An Introduction to the Conjugate Gradient Method Without the Agonizing Pain,”  
Jonathan Richard Shewchuk