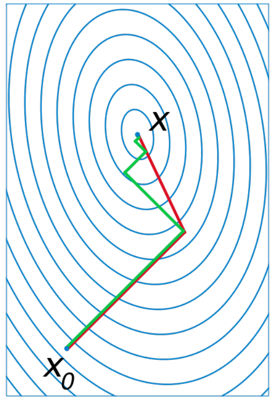
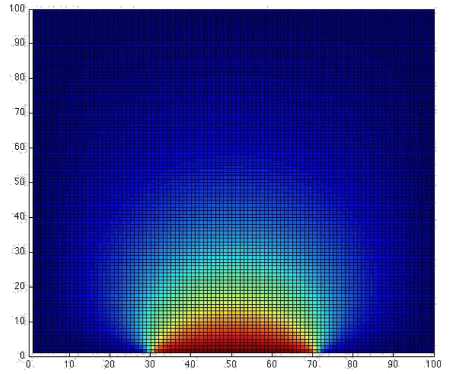
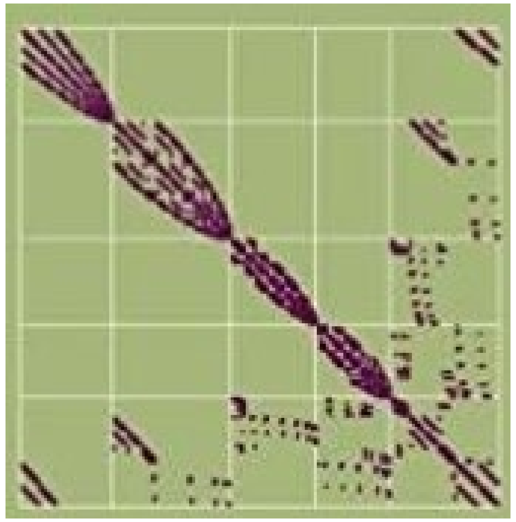




Lecture 09:

# Iterative Solvers of Ax = b
# The Jacobi and CG Methods 2

# Numerical Stability



CS 111: Intro to Computational Science

Spring 2023

Ziad Matni, Ph.D.

Dept. of Computer Science, UCSB

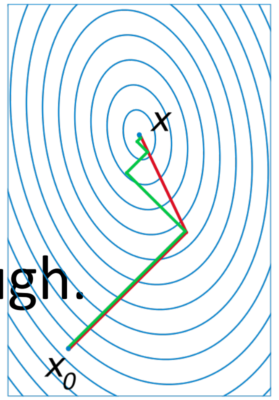
Carl Gustav Jacob Jacobi (1804 – 1851)

## Administrative

- Current homework due today
- New homework out later today

- Quiz 3 on Wednesday
  - Lectures 5, 6, 7, and 8
    - SPD, Cholesky, QR, the Temp. Problem, Jacobi

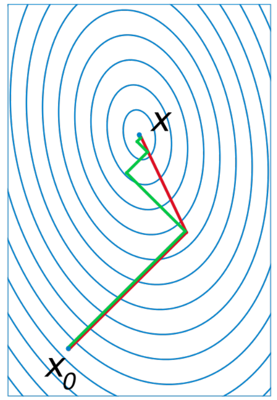
## Conjugate Gradient Method



- Another (more efficient) iterative algorithm to solve $Ax = b$.
  - Start with a guess $x^{(0)}$, then compute $x^{(1)}$, $x^{(2)}$, ....
  - Stop when you think (or when your error measure says) you're close enough.

- In theory, **CG** can be used to solve *any* system $Ax = b$, provided "only" that **A is SPD**.
  - There is a related method called BiCG that can solve for a general (i.e. non-symmetric) **A**

## Conjugate Gradient Method



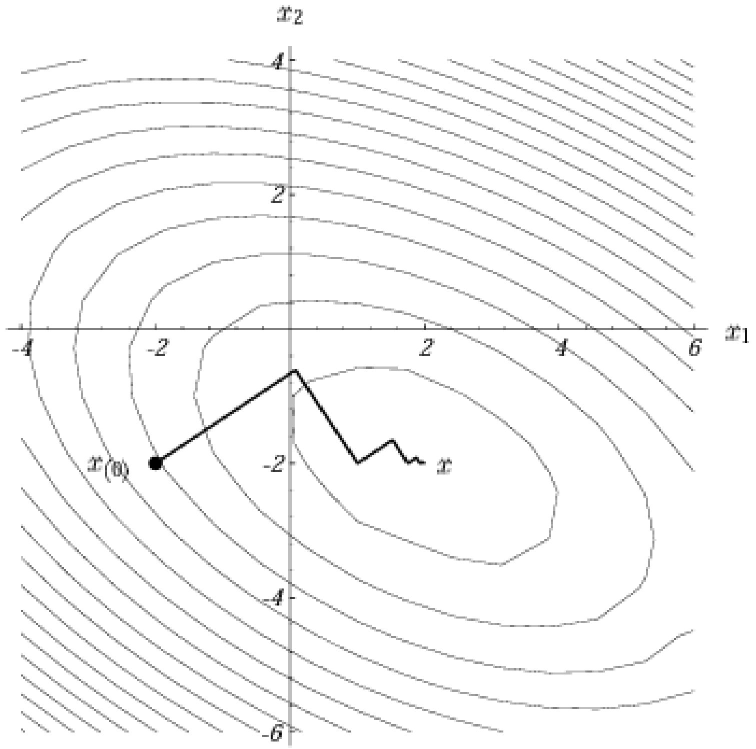
- In practice, how well **CG** works depends on specifics of **A** in subtle ways, involving eigenvalues and the *condition number*.
- QUESTION:
  Has the same requirements as Cholesky’s Method (**A** is SPD), so why use CG??
- ANS:
  It’s more efficient computationally for very large, very sparse matrices

# Main Idea of CG

- In A**x** = b, where A is an SPD, solving for **x** is like finding the minimizing point of a quadratic equation in *n*-dim space (so, more general than just a 2 dimensional approach)

- Related to the method of **Steepest Descent** – finding the minimum of a function by taking a series of *optimal "steep gradients"* to get to it

- Each time a step is taken, we *re-adjust* for the next descent by examining *error factors*



Here, the method of Steepest Descent starts at $[-2, -2]^T$ and converges at $[2, -2]^T$.

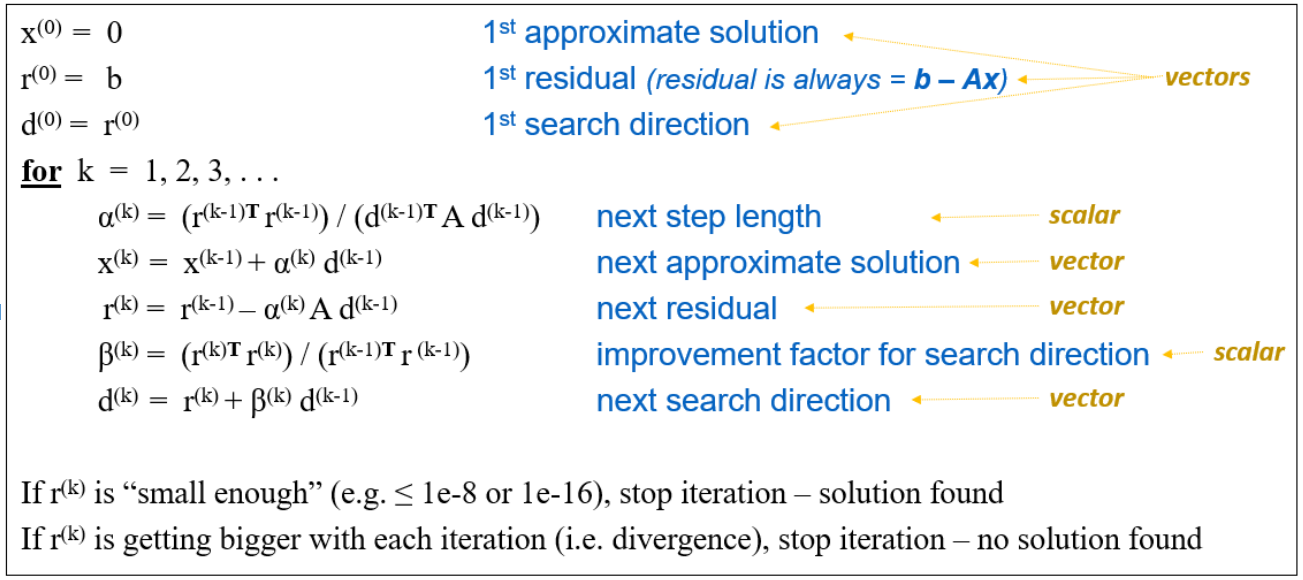
# CG Iterative Algorithm

```
$x^{(0)} = 0$                  1st approximate solution ← vectors
$r^{(0)} = b$                  1st residual *(residual is always = **b – Ax**)* ← vectors
$d^{(0)} = r^{(0)}$            1st search direction ← vectors
**for** $k = 1, 2, 3, \ldots$
    $\alpha^{(k)} = (r^{(k-1)\mathbf{T}} r^{(k-1)}) / (d^{(k-1)\mathbf{T}} A\, d^{(k-1)})$     next step length ← scalar
    $x^{(k)} = x^{(k-1)} + \alpha^{(k)} d^{(k-1)}$     next approximate solution ← vector
    $r^{(k)} = r^{(k-1)} - \alpha^{(k)} A\, d^{(k-1)}$     next residual ← vector
    $\beta^{(k)} = (r^{(k)\mathbf{T}} r^{(k)}) / (r^{(k-1)\mathbf{T}} r^{(k-1)})$     improvement factor for search direction ← scalar
    $d^{(k)} = r^{(k)} + \beta^{(k)} d^{(k-1)}$     next search direction ← vector
```

If $r^{(k)}$ is "small enough" (e.g. $\leq$ 1e-8 or 1e-16), stop iteration – solution found

If $r^{(k)}$ is getting bigger with each iteration (i.e. divergence), stop iteration – no solution found

# CG Efficiency



```
$x^{(0)} = 0$                      1st approximate solution
$r^{(0)} = b$                      1st residual (residual is always = b – Ax)
$d^{(0)} = r^{(0)}$                1st search direction
                                                                      vectors
for k = 1, 2, 3, . . .
    $\alpha^{(k)} = (r^{(k-1)T} r^{(k-1)}) / (d^{(k-1)T} A d^{(k-1)})$    next step length                          scalar
    $x^{(k)} = x^{(k-1)} + \alpha^{(k)} d^{(k-1)}$                          next approximate solution                 vector
    $r^{(k)} = r^{(k-1)} - \alpha^{(k)} A d^{(k-1)}$                        next residual                             vector
    $\beta^{(k)} = (r^{(k)T} r^{(k)}) / (r^{(k-1)T} r^{(k-1)})$            improvement factor for search direction   scalar
    $d^{(k)} = r^{(k)} + \beta^{(k)} d^{(k-1)}$                            next search direction                     vector
```

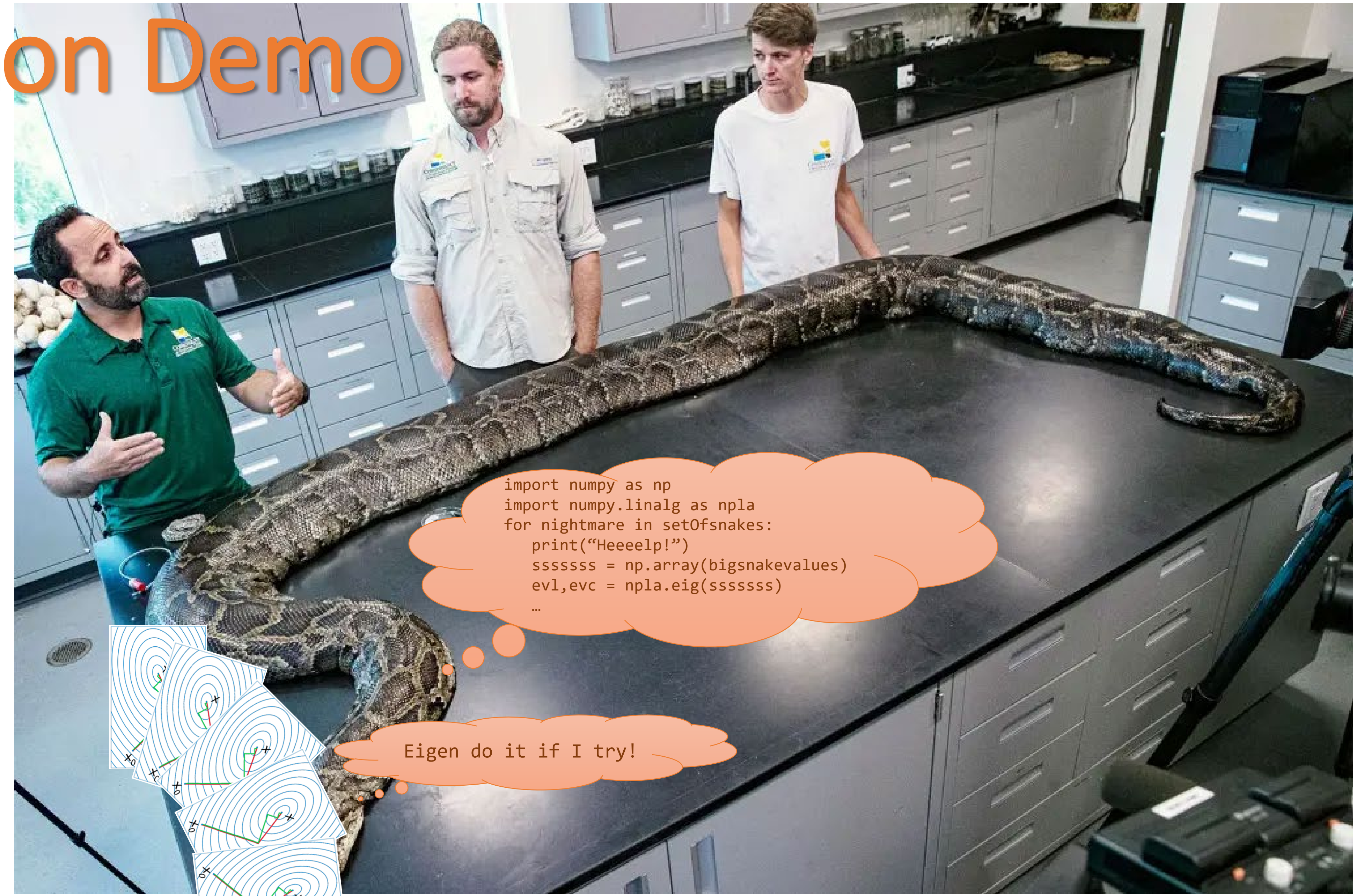
If $r^{(k)}$ is "small enough" (e.g. ≤ 1e-8 or 1e-16), stop iteration – solution found

If $r^{(k)}$ is getting bigger with each iteration (i.e. divergence), stop iteration – no solution found

Note that in each iteration, we have:

- Vector dot products
  - Example: 2 inside $\alpha^{(k)}$, 1 inside $\beta^{(k)}$
  - Each vector dot product has n multiplications and n-1 additions ➔ Time = O(n)
- Matrix-vector multiplication
  - Example: **A**$\mathbf{d}^{(k-1)}$ inside $\alpha^{(k)}$
  - Each matrix-vector multiplication has $n^2$ multiplications and n(n-1) additions ➔ Time = $O(n^2)$
  - BUT if **A** is a sparse matrix, then Time ≅ O(*m*) where: *m* = # of non-zeros in **A**
    - This is generally better than $O(n^2)$
    - More accurately, it's $O(m\sqrt{\kappa})$ where $\kappa$ = condition number
    - We generally want $\kappa$ to be close to 1 because it means **A** is a "stable matrix"
    - What's this "*condition number*"??? More on that next time! ☺

# Python Demo



```
import numpy as np
import numpy.linalg as npla
for nightmare in setOfsnakes:
   print(“Heeeelp!”)
   sssssss = np.array(bigsnakevalues)
   evl,evc = npla.eig(sssssss)
   …
```

*Credit: The News-Press-USA TODAY NETWORK*

# A Landscape of **Ax=b** Solvers

| | **Direct Methods**<br>**A = LU**<br>*More robust, more storage, typically higher complexity ( $O(n^3)$ )* | **Iterative Methods**<br>$\mathbf{y^{(k+1)} = Ay^{(k)}}$<br>*Less storage (good with large, sparse **A**), can be lower complexity* |
|---|---|---|
| ***General type*** | Pivoting LU<br>QR* | Jacobi**<br>BiConjugate Gradient |
| ***SPD type*** | Cholesky | Conjugate Gradient |

** QR Method works best with real matrices (LU and Cholesky can work with complex matrices)*
*** Jacobi's Method works a lot better if the matrix **A** is diagonally dominant*

# Numerical Stability

**A generally desirable property of numerical algorithms**

- Consider **f(x) = y**   (a *mathematical* definition)
- We calculate it, using some computation, to be **y***
  - **y*** is a ***deviation*** from the "true" solution **y**   (it's close in value to **y**, but not exactly the same)
  - This can happen because of round-off errors and/or truncation errors

**DEFINITIONS:**

- **Forward Error**: $\Delta y = y^* - y$
- **Backward Error**: Smallest $\Delta x$ such that $f(x + \Delta x) = y^*$
- **Relative Error**: $|\Delta x| \,/\, |x|$

**We ideally want a small Δx to give us a small Δy**

## Numerical Stability

- In a computational matrix **A** we often want to examine:

**how small changes in it can lead to**
***either small or large changes in calculations***

  - You want small changes to yield small changes – that's "**stability**"…

**<u>Example:</u>**

$$\mathbf{A} = \begin{bmatrix} 1 & 1000 \\ 0 & 1 \end{bmatrix} \quad \text{and } \mathbf{Ax} = \mathbf{b} \text{ where } \mathbf{b} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

$$\mathbf{A} = \begin{bmatrix} 1 & 1000 \\ 0 & 1 \end{bmatrix} \quad \text{and } \mathbf{Ax} = \mathbf{b} \text{ where } \mathbf{b} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

## Numerical Stability

- We're going to use **A** and **b**, to solve for **x**
- What happens if **x** deviates a little bit (call the new value, **x***)?
  - Then, **Ax* = b*** *should not* be too far from **b…**
- We will look for something called the **Condition Number**
  - Measures how much the **output** value of a *function* can **change** for a ***small change*** in the **input** argument
  - **This is inherent to the function (i.e. to the matrix)**

## Numerical Stability

- **Forward** and **backward** error are related by the **condition number**
- Large condition number ➔ the matrix is ***not*** numerically stable
  - This is called "**ill-conditioned**"
- Small condition number (the closer to 1, the better)
  ➔ the matrix **is** numerically stable
  - This is called "**well-conditioned**"
- Usually, symmetrical and/or normal matrices are "well-conditioned"

## Condition Number

**Demo…**

The condition number of matrix **M** is defined as:

$$||M|| \times ||M^{-1}||$$

*i.e. norm of M multiplied by the norm of the inverse of M*

In Python, you can use the `.cond()` function in `linalg`:

**`numpy.lingalg.cond(M, ‘fro’)`**

Where **`‘fro’`** indicates the Frobenius Norm, that is the norm defined as the square root of the sum of the absolute squares of its elements (most common way)

# Your TO DOs!

- Assignment 04 due tonight

- Quiz 3 on Wednesday
  - Lectures 5, 6, 7, and 8
    - SPD, Cholesky, QR, the Temp. Problem, Jacobi

</LECTURE>