Linear Algebra

Numerical Linear Algebra

- We've now seen several places where solving linear systems comes into play
  - Implicit ODE integration
  - Cubic spline interpolation
- We'll see many more, including
  - Solving the diffusion PDE
  - Multivariable root-finding
  - Curve fitting

Numerical Linear Algebra

- The basic problem we wish to solve is: **A x** = **b**
  - We'll start with the most general methods
  - Depending on the form of the matrix (sparse, symmetric, etc.) more efficient methods exist
- Generally speaking, you don't need to write your own linear algebra routines—efficient, robust libraries exist
- We want to explore some of the key methods to understand what they do
- Garcia provides a good introduction to the basic methods
  - We'll add a few other topic along the way

Review of Matrices

- Matrix-vector multiplication
  - **A** is *m*×*n* matrix
  - **x** is *n*×1 (column) vector
  - Result: **b** is *m*×1 (column) vector
  - Simple scaling: O(N^2) operation

$$b_i = (Ax)_i = \sum_{j=1}^{M} A_{i,j} x_j$$

- Matrix-matrix multiplication
  - **A** is *m*×*n* matrix
  - **B** is *n*×*p* matrix
  - (**AB**) is *m*×*p* matrix
  - Direct multiplication: O(N^3) operation.
    - Some faster algorithms exist (make use of organization of sub-matrices for simplication)

$$(AB)_{ij} = \sum_{k=1}^{M} A_{ik} B_{kj}$$

Review of Matrices

- Determinant
  - Encodes some information about the (square) matrix
    - For us: Used in some linear system algorithms (and solution only exists if determinant non-zero)
    - Also gives area of parallelogram or volume of parallelepiped
  - 2×2 and 3×3 are straighforward:

$$\begin{vmatrix} a & b \\ c & d \end{vmatrix} = ad - bc$$

$$\begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} = a \begin{vmatrix} e & f \\ h & i \end{vmatrix} + b \begin{vmatrix} d & f \\ g & i \end{vmatrix} + c \begin{vmatrix} d & e \\ g & h \end{vmatrix}$$

  - Algorithm we were taught in school for larger matrices (sometimes called the *Laplace expansion*) is cumbersome (blackboard)

Review of Matrices

- Inverse
  - $\mathbf{A}^{-1}\mathbf{A} = \mathbf{A}\,\mathbf{A}^{-1} = \mathbf{I}$
  - Formally, the solution to the linear system $\mathbf{Ax} = \mathbf{b}$ is $\mathbf{x} = \mathbf{A}^{-1}\,\mathbf{b}$
    - We'll see that it is less expensive to get the solution without computing the inverse first
  - Singular (non-invertible) if the determinant is 0
- Eigenvalues and eigenvectors
  - We'll encounter these when we do PDEs later
  - Eigenvalues: $|\mathbf{A} - \lambda\mathbf{I}| = 0$
  - Right eigvenvectors: $\mathbf{Ar} = \lambda\mathbf{r}$
  - Left eigenvectors: $\mathbf{lA} = \lambda\mathbf{l}$

Cramer's Rule

- It is instructive to review the techniques we studied analytically to solve linear systems
- Cramer's rule is for solving a linear system **A x** = **b**

$$x_i = \frac{|\mathbf{A}_i|}{|\mathbf{A}|}$$

  - Here, $\mathbf{A}_i$ is the matrix **A** with the *i*th column replaced by the vector **b**
  - Blackboard example
- Note that the methods we use w/ pen and paper are not going to scale for big systems on a computer
  - from Yakowitz & Szidarovszky: *The solution of a 20-variable (m = n = 20) system by the popular Cramer's rule requires about 5 x 10^{19} multiplications*"
    - today's fastest machine (from Top 500 list) is Tianhe-2 in China: peak is 55,000 TFLOP/S
      - this is 252 hours to solve our N = 20 system

Gaussian Elimination

- The main, general technique for solving a linear system **Ax** = **b** is Gaussian-Elimination
  - Doesn't require computing an inverse
  - For special matrices, faster techniques may apply
- Forward-elimination + Back-substitution steps
  - Blackboard example to get a feel…
- Round-off can cause issues and lead to systems that are unsolvable
  - Example from Garcia (blackboard)

$$\begin{aligned} \epsilon x_1 + x_2 + x_3 &= 5 \\ x_1 + x_2 \qquad &= 3 \\ x_1 + \qquad x_3 &= 4 \end{aligned}$$

  - in limit ε → 0
- But note: there is *no truncation in this process*—it's all algebra

Gaussian Elimination

- 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 form that the matrix is in after forward elimination

Gaussian Elimination

- Implementation: matrix **A** and vector **b** passed in
  - Can change **A** “in place” returning the row-echelon matrix
  - Can pass in multiple **b**'s—work is greatly reduced once you've put **A** into row echelon form
  - When you pivot **A** you need to swap rows in **b** as well
- Work ~$O(N^3)$
- Note: swapping rows (pivoting) does not change the order of the **x**'s. (Swapping columns would).
- Important: to test this out, code up a matrix-vector multiply and see if the solution recovers your initial righthand side.

Gaussian Elimination

```
⎡   4.000    3.000    4.000   10.000⎤        ⎡  2.000⎤
⎢   2.000   -7.000    3.000    0.000⎥        ⎢  6.000⎥
⎢  -2.000   11.000    1.000    3.000⎥        ⎢  3.000⎥
⎣   3.000   -4.000    0.000    2.000⎦        ⎣  1.000⎦
pivoted

⎡   3.000   -4.000    0.000    2.000⎤        ⎡  1.000⎤
⎢   0.000   -4.333    3.000   -1.333⎥        ⎢  5.333⎥
⎢   0.000    8.333    1.000    4.333⎥        ⎢  3.667⎥
⎣   0.000    8.333    4.000    7.333⎦        ⎣  0.667⎦
pivoted

⎡   3.000   -4.000    0.000    2.000⎤        ⎡  1.000⎤
⎢   0.000    8.333    4.000    7.333⎥        ⎢  0.667⎥
⎢   0.000    0.000   -3.000   -3.000⎥        ⎢  3.000⎥
⎣   0.000    0.000    5.080    2.480⎦        ⎣  5.680⎦
pivoted

⎡   3.000   -4.000    0.000    2.000⎤        ⎡  1.000⎤
⎢   0.000    8.333    4.000    7.333⎥        ⎢  0.667⎥
⎢   0.000    0.000    5.080    2.480⎥        ⎢  5.680⎥
⎣   0.000    0.000    0.000   -1.535⎦        ⎣  6.354⎦

⎡   3.000   -4.000    0.000    2.000⎤        ⎡  1.000⎤
⎢   0.000    8.333    4.000    7.333⎥        ⎢  0.667⎥
⎢   0.000    0.000    5.080    2.480⎥        ⎢  5.680⎥
⎣   0.000    0.000    0.000   -1.535⎦        ⎣  6.354⎦
```

Let's look at the code

solved x: [6.04615385 2.21538462 3.13846154 -4.13846154]
A.x: [2. 6. 3. 1.]

code: gauss.py, matmul.py, gauss-test.py

Caveats

- Singular matrix
  - If the matrix has no determinant, then we cannot solve the system
  - Common way for this to enter: one equation in our system is a linear combination of some others
  - Not always easy to detect from the start
  - Singular G-E example…

`code: gauss-singular.py`

Gaussian Elimination

- Condition number
  - Measure of how close to singular we are
  - Think of it as a measure of how much **x** would change due to a small change in **b** (large condition number means G-E inaccurate)
  - Formal definition: $\mathrm{cond}(\mathbf{A}) = \|\mathbf{A}\|\|\mathbf{A}^{-1}\|$
    - Further elucidation requires defining the norm
  - Rule of thumb (Yakowitz & Szidarovszky, Eq. 2.31):
    - If $\mathbf{x}^\star$ is the exact solution and **x** is the computed solution, then

$$\frac{\|\mathbf{x}^\star - \mathbf{x}\|}{\|\mathbf{x}^\star\|} \approx \mathrm{cond}(\mathbf{A}) \cdot \epsilon_{\mathrm{machine}}$$

- Simple ill-conditioned example (from G. J. Tee, ACM SIGNUM Newsletter, v. 7, issue 3, Oct. 1972, p. 19) (blackboard)...

Determinants

- Gaussian elimination leaves the matrix in a form where it is trivial to get the determinant
  - If no pivoting was done, then

$$\det(\mathbf{A}) = \prod_{i=1}^{N} A_{ii}^{\text{row-echelon}}$$

    - Where the "row-echelon" superscript indicates that this is done over the matrix in row echelon form
  - Each time you interchange rows, the determinant changes sign, so with pivoting:

$$\det(\mathbf{A}) = (-1)^{\mathrm{N_{pivot}}} \prod_{i=1}^{N} A_{ii}^{\text{row-echelon}}$$

Sparse Matrices / Tridiagonal

- Recall from cublic splines, we had this linear system:

$$\begin{pmatrix} 4\Delta x & \Delta x & & & & & \\ \Delta x & 4\Delta x & \Delta x & & & & \\ & \Delta x & 4\Delta x & \Delta x & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & \ddots & \ddots & \ddots & \\ & & & & \Delta x & 4\Delta x & \Delta x \\ & & & & & \Delta x & 4\Delta x \end{pmatrix} \begin{pmatrix} p_1'' \\ p_2'' \\ p_3'' \\ \vdots \\ \vdots \\ p_{n-2}'' \\ p_{n-1}'' \end{pmatrix} = \frac{6}{\Delta x} \begin{pmatrix} f_0 - 2f_1 + f_2 \\ f_1 - 2f_2 + f_3 \\ f_2 - 2f_3 + f_4 \\ \vdots \\ \vdots \\ f_{n-3} - 2f_{n-2} + f_{n-1} \\ f_{n-2} - 2f_{n-1} + f_n \end{pmatrix}$$

- Gaussian elimination as we discussed doesn't take advantage of all the zeros

Tridiagonal Solve

- Tridiagonal systems come up very often in physical systems
- Efficient, O(N), algorithm derived from looking at the Gaussian elimination sequence (see e.g. Wikipedia):
  - Standard data layout. Let's look at the code…

$$\begin{pmatrix} b_0 & c_0 & & & & & \\ a_1 & b_1 & c_1 & & & & \\ & a_2 & b_2 & c_2 & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & \ddots & \ddots & \ddots & \\ & & & & a_{N-2} & b_{N-2} & c_{N-2} \\ & & & & & a_{N-1} & b_{N-1} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ \vdots \\ x_{N-2} \\ x_{N-1} \end{pmatrix} = \begin{pmatrix} d_0 \\ d_1 \\ d_2 \\ \vdots \\ \vdots \\ d_{N-2} \\ d_{N-1} \end{pmatrix}$$

Note that $a_0 = c_{N-1} = 0$

Linear Algebra

Class Logistics

- Homework #3 was assigned
- **If you use Matlab or Mathematica, please add a plain text (.txt) file of the output to your git repo**
- I'm mostly finished grading homework #1
  - there will be a `grades.txt` file in your repo with my comments
- Start the homework early
  - if you run into trouble, push your work, and I can pull and look at it

Matrix Inverse

- We can find the inverse using Gaussian elimination too.
  - Start with $\mathbf{A}\mathbf{A}^{-1} = \mathbf{I}$
    - Let $\mathbf{x}^{(\alpha)}$ represent a column of $\mathbf{A}^{-1}$
    - Let $\mathbf{e}^{(\alpha)}$ represent a column of $\mathbf{I}$
  - Now we just need to solve the α linear systems:

$$\mathbf{A}\mathbf{x}^{(\alpha)} = \mathbf{e}^{(\alpha)}$$

  - We can carry all the $\mathbf{e}^{(\alpha)}$ together as we do the forward elimination, greatly reducing the computational expense.
- Again, test it by doing a multiply at the end to see if you get the identity matrix
- Let's look at the code…

Matrix Inverse

(based on Garcia)

- We can also use iterative methods, instead of direct methods
  - We can use a Newton-like method to find the inverse of **A**, as:

$$\mathbf{X}^{(k+1)} = 2\mathbf{X}^{(k)} - \mathbf{X}^{(k)}\mathbf{A}\mathbf{X}^{(k)}$$

    - This mirrors the result that Newton's method would give for the scalar equation ax – 1 = 0
    - Here $\mathbf{X}^k$ is the current guess to the inverse
- We need a good initial guess of else this will diverge rapidly.

LU Decomposition

- LU Decomposition is an alternate to Gaussian elimination
  - See Newman/Pang/Wikipedia for details
- Basic idea:
  - Find upper and lower triangular matrices such that **A** = **LU**
  - Express **Ax** =**b** as **Ly** = **b** and **Ux** = **y**
    - These are easy to solve since they are triangular
  - Most of the work in the solve is in doing the decomposition, so if you need to solve for many different **b**'s, this is more efficient
- This is often used in stiff integration packages

LU Decomposition

- Consider:

$$\mathbf{A} = \begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix}$$

- The first step of Gaussian elimination can be expressed as:

$$\underbrace{\frac{1}{a_{00}} \begin{pmatrix} 1 & 0 & 0 & 0 \\ -a_{10} & a_{00} & 0 & 0 \\ -a_{20} & 0 & a_{00} & 0 \\ -a_{30} & 0 & 0 & a_{00} \end{pmatrix}}_{\mathbf{L}_0} \begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix}$$

LU Decomposition

- Then define the result as **B**:

$$\mathbf{B} = \mathbf{L}_0 \mathbf{A} = \begin{pmatrix} 1 & b_{01} & b_{02} & b_{03} \\ 0 & b_{11} & b_{12} & b_{13} \\ 0 & b_{21} & b_{22} & b_{23} \\ 0 & b_{31} & b_{32} & b_{33} \end{pmatrix}$$

- The next step of Gaussian elimination works on the second row, and can be expressed as

$$\underbrace{\frac{1}{b_{11}} \begin{pmatrix} b_{11} & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & -b_{21} & b_{11} & 0 \\ 0 & -b_{31} & 0 & b_{11} \end{pmatrix}}_{\mathbf{L}_1} \begin{pmatrix} 1 & b_{01} & b_{02} & b_{03} \\ 0 & b_{11} & b_{12} & b_{13} \\ 0 & b_{21} & b_{22} & b_{23} \\ 0 & b_{31} & b_{32} & b_{33} \end{pmatrix}$$

LU Decomposition

- This can continue, defining

$$\mathbf{C} = \mathbf{L}_1\mathbf{B} = \mathbf{L}_1\mathbf{L}_0\mathbf{A} = \begin{pmatrix} 1 & c_{01} & c_{02} & c_{03} \\ 0 & 1 & c_{12} & c_{13} \\ 0 & 0 & c_{22} & c_{23} \\ 0 & 0 & c_{32} & c_{33} \end{pmatrix}$$

- Eventually, after all the steps, we wind up with an upper triangle matrix:

$$\mathbf{U} \equiv \mathbf{L}_3\mathbf{L}_2\mathbf{L}_1\mathbf{L}_0\mathbf{A}$$

  - Our system is then: **Ux** = **Ub**
  - Writing this as **A** = **LU**, we see that

$$\mathbf{L} = \mathbf{L}_0^{-1}\mathbf{L}_1^{-1}\mathbf{L}_2^{-1}\mathbf{L}_3^{-1}$$

LU Decomposition

- The inverse of each $\mathbf{L}_x$ can be found more or less by inspection, and then the product is (see your text, e.g.)

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

- Some notes:
  - In general, you still need to pivot. If you want to reuse the **L** and **U** for multiple solves, you need to store the pivots too, since you'll need to *replay* them on an righthand side you pass in
  - There is not a unique decompostion. Different texts put all 1s on the diagonal in **L** or **U**

`code: lu_decomp.py`

BLAS/LINPACK/LAPACK

- 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

BLAS/LINPACK/LAPACK

- LINPACK
  - Older standard linear algebra package from 1970s designed for architectures available then
  - Superseded by LAPACK

See Top 500 list

- 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)
  - Ex: single precision routines: http://www.netlib.org/lapack/single/

Table 2.1: Matrix types in the LAPACK naming scheme

BD	bidiagonal
DI	diagonal
GB	general band
GE	general (i.e., unsymmetric, in some cases rectangular)
GG	general matrices, generalized problem (i.e., a pair of general matrices)
GT	general tridiagonal
HB	(complex) Hermitian band
HE	(complex) Hermitian
HG	upper Hessenberg matrix, generalized problem (i.e a Hessenberg and a triangular matrix)
HP	(complex) Hermitian, packed storage
HS	upper Hessenberg
OP	(real) orthogonal, packed storage
OR	(real) orthogonal
PB	symmetric or Hermitian positive definite band
PO	symmetric or Hermitian positive definite
PP	symmetric or Hermitian positive definite, packed storage
PT	symmetric or Hermitian positive definite tridiagonal
SB	(real) symmetric band
SP	symmetric, packed storage
ST	(real) symmetric tridiagonal
SY	symmetric
TB	triangular band
TG	triangular matrices, generalized problem (i.e., a pair of triangular matrices)
TP	triangular, packed storage
TR	triangular (or in some cases quasi-triangular)
TZ	trapezoidal
UN	(complex) unitary
UP	(complex) unitary, packed storage

http://www.netlib.org/lapack/lug/node24.html

Python Support for Linear Algebra

- Basic methods in numpy.linalg
  - http://docs.scipy.org/doc/numpy/reference/routines.linalg.html
- More general stuff in SciPy (scipy.linalg)
  - http://docs.scipy.org/doc/scipy/reference/linalg.html
- Let's look at some code
  - No pure tridiagonal solver, but instead banded matrices (see our old cubic spline example...)

Python Support for Linear Algebra

- NumPy has a matrix type built from the array class
  - * operator works element by element for arrays but does matrix product for matrices
  - Vectors are automatically converted into 1×N or N×1 matrices
  - Matrix objects cannot be > rank 2
  - Matrix has .H, .I, and .A attributes (transpose, inverse, as array)
  - See http://www.scipy.org/NumPy_for_Matlab_Users for more details
- Some examples…
  - http://www.scipy.org/Tentative_NumPy_Tutorial
- As of python 3.5, @ operator will do matrix multiplication for NumPy arrays

Jacobi and Gauss-Seidel Iteration

- So far the methods we have been exploring are direct
- Iterative methods allow approximation solutions to be found
  - Can be much more efficient for large system
  - Puts some restrictions on the matrix, but we'll see that these can usually be met when studying PDEs

Jacobi Iteration

(Yakowitz & Szidarovszky)

- Starting point: our linear system

$$\begin{array}{ccccccc} a_{11}x_1 & + & a_{12}x_2 & + & \dots & a_{1n}x_n & = & b_1 \\ a_{21}x_1 & + & a_{22}x_2 & + & \dots & a_{2n}x_n & = & b_2 \\ & & & & \vdots & & & \\ a_{n1}x_1 & + & a_{n2}x_2 & + & \dots & a_{nn}x_n & = & b_n \end{array}$$

- Pick an initial guess: $x^{(k)}$
- Solve each equation i for the i^{th} unknown to get improved guess

$$x_1^{(k+1)} = -\frac{1}{a_{11}}(a_{12}x_2^{(k)} + a_{13}x_3^{(k)} \dots + a_{1n}x_n^{(k)} \quad -b_1)$$

$$x_2^{(k+1)} = -\frac{1}{a_{22}}(a_{21}x_1^{(k)} + a_{23}x_3^{(k)} \dots + a_{2n}x_n^{(k)} \quad -b_2)$$

$$\vdots$$

$$x_n^{(k+1)} = -\frac{1}{a_{nn}}(a_{n1}x_1^{(k)} + a_{n2}x_2^{(k)} \dots + a_{n,n-1}x_{n-1}^{(k)} - b_n)$$

Jacobi and Gauss-Seidel Iteration

- This is an iterative method: each iteration improves our guess.
- Convergence is guaranteed if the matrix is diagonally dominant

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

  - Frequently works if the “>” above is replaced by “≥”
  - You may need to swap equations around until this is satisfied
  - We'll see these methods again when solving Poisson's equation
- Note: you actually don't need to write out the matrix for this solve
- Gauss-Seidel iteration is almost the same
  - The only change is that as you get the new estimate for each x , you use that new value immediately in the subsequent equations

Jacobi and Gauss-Seidel Iteration

- When do we stop?
  - Fixed # of iterations (this doesn't give you an error measure)
  - Monitor the maximum change in the x 's from one iteration to the next
  - Compute the residual (how well do we satisfy **Ax** = **b** ?)

Sparse Matrices / Tridiagonal

- Let's try this on our cubic spline system...

$$\begin{pmatrix} 4\Delta x & \Delta x & & & & & \\ \Delta x & 4\Delta x & \Delta x & & & & \\ & \Delta x & 4\Delta x & \Delta x & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & \ddots & \ddots & \ddots & \\ & & & & \Delta x & 4\Delta x & \Delta x \\ & & & & & \Delta x & 4\Delta x \end{pmatrix} \begin{pmatrix} p_1'' \\ p_2'' \\ p_3'' \\ \vdots \\ \vdots \\ p_{n-2}'' \\ p_{n-1}'' \end{pmatrix} = \frac{6}{\Delta x} \begin{pmatrix} f_0 - 2f_1 + f_2 \\ f_1 - 2f_2 + f_3 \\ f_2 - 2f_3 + f_4 \\ \vdots \\ \vdots \\ f_{n-3} - 2f_{n-2} + f_{n-1} \\ f_{n-2} - 2f_{n-1} + f_n \end{pmatrix}$$

- This is clearly diagonally dominant.
- Switch to Gauss-Seidel...

`code: cubic-spline-iterate.py`

Multivariate Newton's Method

- Imagine a vector function: $\mathbf{f}(\mathbf{x})$
  - $\mathbf{f}(\mathbf{x}) = (f_1(\mathbf{x})\ f_2(\mathbf{x})\ f_3(\mathbf{x})\ \ldots\ f_N(\mathbf{x}))^T$
  - Column vector of unknowns: $\mathbf{x} = (x_1\ x_2\ x_3\ \ldots\ x_N)^T$
- We want to find the zeros:
  - Initial guess: $\mathbf{x}^{(0)}$
  - Taylor expansion:

$$f_i(\mathbf{x}^{(0)} + \delta\mathbf{x}) \approx 0 = f_i(\mathbf{x}^{(0)}) + \sum_{j=1}^{N} \frac{\partial f_i}{\partial x_j}\delta x_j + \ldots$$

N equations + N unknowns

Jacobian

  - Update to initial guess is: $\delta\mathbf{x} = -\mathbf{J}^{-1}\mathbf{f}(\mathbf{x}^{(0)})$
    - Cheaper to solve the linear system than to invert the Jacobian
  - Iterate:

$$\mathbf{J}\delta\mathbf{x}^{(k)} = -\mathbf{f}(\mathbf{x}^{(k)})\ , \quad \mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \delta\mathbf{x}^{(k)}$$

Example: Lorenz Model

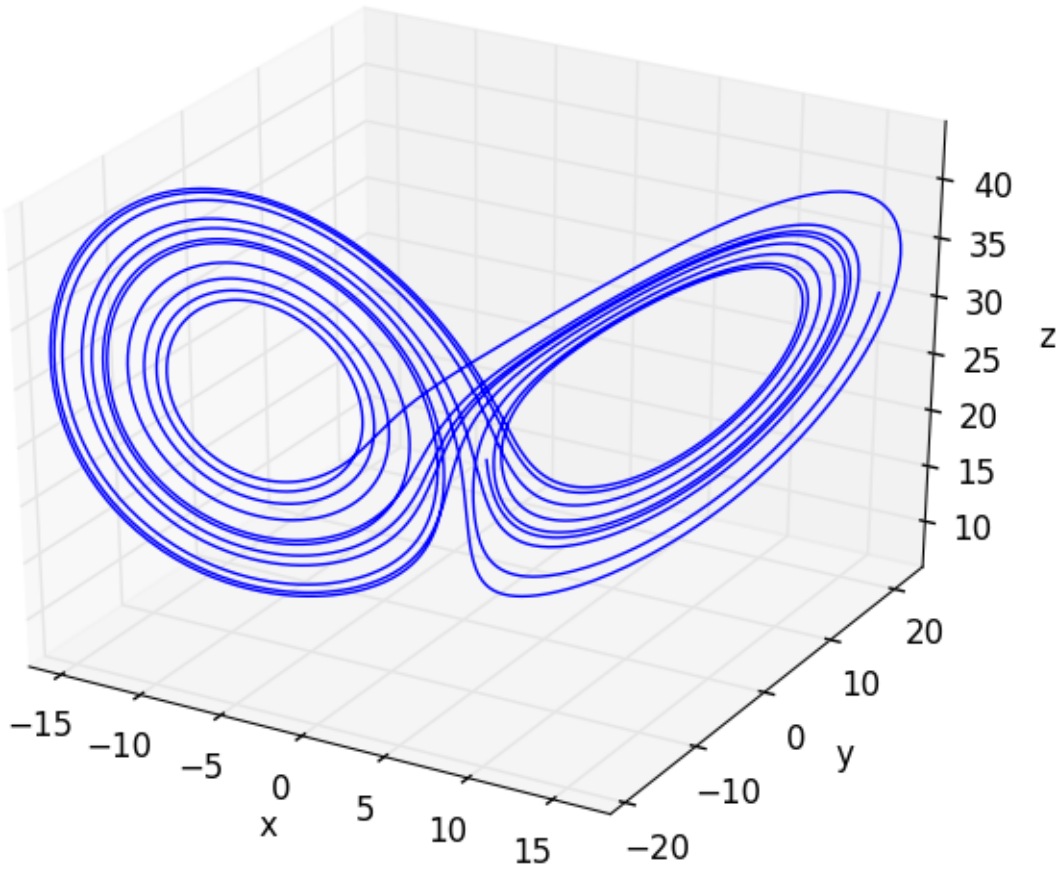
- Lorenz system:

$$\frac{dx}{dt} = \sigma(y - x)$$

$$\frac{dy}{dt} = rx - y - xz$$

$$\frac{dz}{dt} = xy - bz$$

- Phase-space solution
  - In dynamical systems theory, this is said to be an attractor
  - As points pass through the center, they are sent to either left or right lobes
    - Initially close points get widely separated



Example: Lorenz Model

- Steady states—the zeros of the righthand side

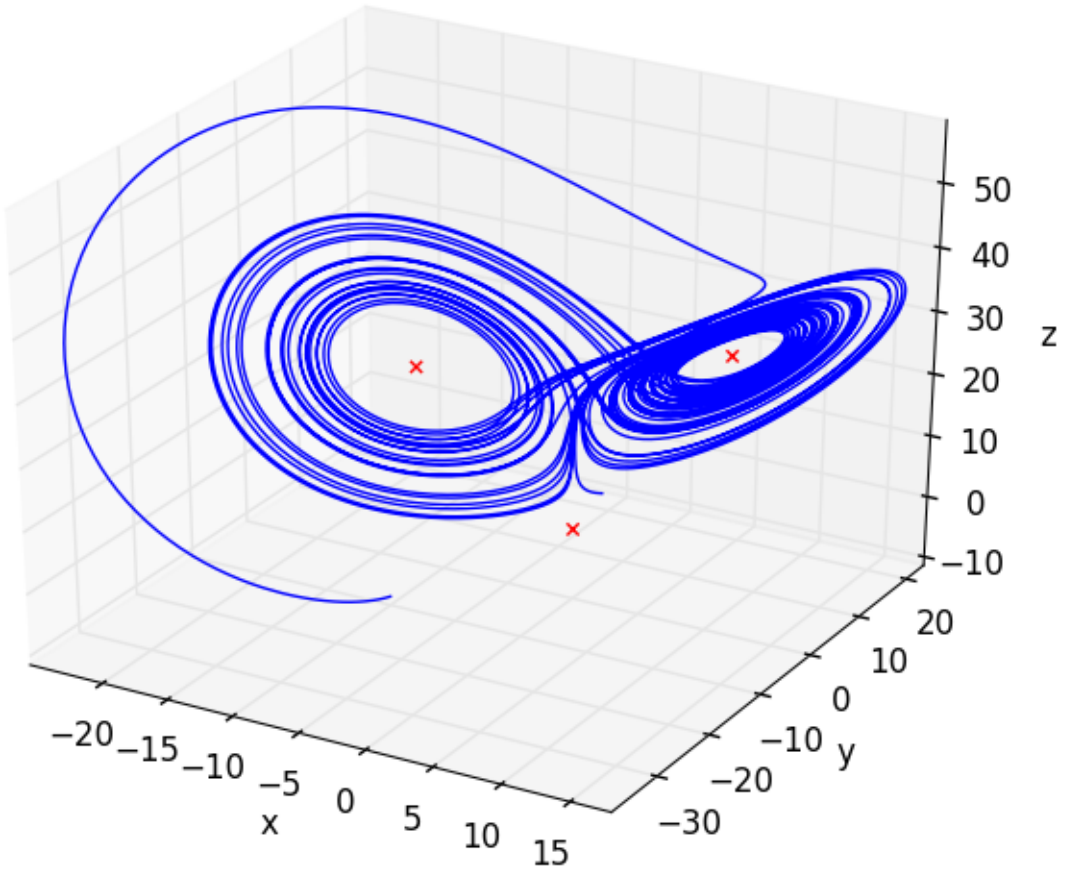
$$\mathbf{f}(\mathbf{x}) = \begin{pmatrix} \sigma(y-x) \\ rx - y - xz \\ xy - bz \end{pmatrix} = 0$$

$$\mathbf{J} = \begin{pmatrix} -\sigma & \sigma & 0 \\ r - z & -1 & -x \\ y & x & -b \end{pmatrix}$$

- Given an initial guess, we can arrive at a root using the multivariate Newton method
  - Steady state gives a cubic—three stationary points
- Look at the code…

`code: lorenz-roots.py`

Example: Lorenz Model



Other Numerical Linear Algebra…

- Extremes of a multivariate function
  - Special methods exist that build off of the multivariate Newton method
  - Pang briefly discusses the BFGS method

Eigenvalues

- Eigenvalues and eigenvectors
  - Many specialized methods exist
  - Built into LAPACK and python/SciPy
- QR method is the “gold standard”
  - Pang and Newman give an overview
  - We'll consider the case of real symmetric (or Hermitian) matrices
  - Get all the eigenvalues at once!

QR Method

- We'll follow the discussion from Newman, Ch 6
- Matrix eigenvectors / eigenvalues satisfy: $\mathbf{A}\mathbf{v} = \lambda\mathbf{v}$
  - Real, symmetric matrix: eigenvectors are orthogonal
  - Assume eigenvectors are normalized

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

- Consider a matrix of eigenvectors:

$$\mathbf{V} = \begin{pmatrix} | & | & \cdots & | \\ \mathbf{v}_1 & \mathbf{v_2} & \ddots & \mathbf{v_N} \\ | & | & \cdots & | \end{pmatrix}$$

  - Then $\mathbf{AV} = \mathbf{\Lambda V}$, where **Λ** is a diagonal matrix of the eigenvalues
  - Note that for a real, symmetric matrix **A**, **V** is orthogonal:

$$\mathbf{V}^\intercal\mathbf{V} = \mathbf{V}\mathbf{V}^\intercal = \mathbf{I}$$

QR Method

- QR decomposition: break A into the product of an orthogonal matrix, **Q**, and an upper triangle matrix, **R**: $\mathbf{A} = \mathbf{Q}_1\mathbf{R}_1$
- Now multiply by $\mathbf{Q}_1^\intercal$: $\mathbf{Q}_1^\intercal\mathbf{A} = \mathbf{Q}_1^\intercal\mathbf{Q}_1\mathbf{R}_1 = \mathbf{R}_1$
- Define a new matrix, $\mathbf{A}_1$: $\mathbf{A_1} = \mathbf{R}_1\mathbf{Q}_1 = \mathbf{Q}_1^\intercal\mathbf{A}\mathbf{Q}_1$
- Do QR decomposition on this new matrix: $\mathbf{A}_1 = \mathbf{Q}_2\mathbf{R}_2$
- Keep repeating the process:

$$\mathbf{A}_1 = \mathbf{Q}_1^\intercal\mathbf{A}\mathbf{Q}_1$$
$$\mathbf{A}_2 = \mathbf{Q}_2^\intercal\mathbf{Q}_1^\intercal\mathbf{A}\mathbf{Q}_1\mathbf{Q}_2$$
$$\vdots$$
$$\mathbf{A}_k = (\mathbf{Q}_k^\intercal\ldots\mathbf{Q}_2^\intercal\mathbf{Q}_1^\intercal)\mathbf{A}(\mathbf{Q}_1\mathbf{Q}_2\ldots\mathbf{Q}_k)$$

- This will converge, with $\mathbf{A}_k$ becoming $\mathbf{\Lambda}$

QR Method

- What about the eigenvectors?
- Recall: $\mathbf{AV} = \mathbf{\Lambda V}$
  - Then: $\mathbf{V}^\intercal \mathbf{AV} = \mathbf{\Lambda}$
- This looks like the end result of our iteration, so we can then identify:

$$\mathbf{V} = \mathbf{Q}_1 \mathbf{Q}_2 \ldots \mathbf{Q}_k$$

- Remaining issues:
  - How do we construct the QR decomposition?
  - How many iterations do we do?
    - Compare diagonal terms to off diagonal (which should → 0), and define an error

QR Method

- Constructing the decomposition **Q**:
- Consider matrix **A** as set of N column vectors:

$$A = \begin{pmatrix} | & | & & | \\ \mathbf{a}_0 & \mathbf{a}_1 & \dots & \mathbf{a}_{N-1} \\ | & | & & | \end{pmatrix}$$

- Define two new sets of vectors, $\mathbf{u}_k$ and $\mathbf{q}_k$, with k = 0, ..., N-1, as

$$\mathbf{q}_k = \mathbf{u}_k / |\mathbf{u}_k|$$

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

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

$$\dots$$

$$\mathbf{u}_k = \mathbf{a}_k - \sum_{j=0}^{k-1} (\mathbf{q}_j \cdot \mathbf{a}_k)\mathbf{q}_j$$

QR Method

- These **q** are orthonormal: $\mathbf{q}_i \cdot \mathbf{q}_j = \delta_{ij}$
- And we can solve for the original matrix **A** columns as:

$$\mathbf{a}_0 = |\mathbf{u}_0|\mathbf{q}_0$$
$$\mathbf{a}_1 = |\mathbf{u}_1|\mathbf{q}_1 + (\mathbf{q}_0 \cdot \mathbf{a}_1)\mathbf{q_0}$$
$$\mathbf{a}_2 = |\mathbf{u}_2|\mathbf{q}_2 + (\mathbf{q}_0 \cdot \mathbf{a}_2)\mathbf{q_0} + (\mathbf{q}_1 \cdot \mathbf{a}_2)\mathbf{q_1}$$
$$\ldots$$

- Then the QR decomposition is:

$$\mathbf{Q} = \begin{pmatrix} | & | & & | \\ \mathbf{q}_0 & \mathbf{q}_1 & \cdots & \mathbf{q}_{N-1} \\ | & | & & | \end{pmatrix} \quad \mathbf{R} = \begin{pmatrix} |\mathbf{u}_0| & \mathbf{q}_0 \cdot \mathbf{a}_1 & \mathbf{q}_0 \cdot \mathbf{a}_2 & \ldots \\ 0 & |\mathbf{u}_1| & \mathbf{q}_1 \cdot \mathbf{a}_2 & \ldots \\ 0 & 0 & |\mathbf{u}_2| & \ldots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix}$$

- Let's look at the code...

`code: eigen.py`

Conjugate Gradient

- Iterative method for symmetric, positive definite ($\mathbf{x}^\mathsf{T}\mathbf{A}\mathbf{x} > 0$) matrices
  - Very efficient for sparse systems
  - Often used in combination with other methods (e.g. multigrid, which we'll study later)
- Good introduction posted on course page: *An Introduction to the Conjugate Gradient Method Without the Agonizing Pain*