

Homework #1 Review

- My notes are committed in git—you may need to “git pull” to sync up before you can push.
- Discussion of the past homework and look over the solutions
 - Problem 2:
 - You need to compare the h to $h/2$ differences to estimate the error—in general you don't have the analytic derivative
 - Richardson extrapolation works from your difference approximations—you don't include the f''' term in your calculations.
 - Problem 3:
 - The coefficient of the leading term in the truncation error is (likely) different between the odd and even cases, so the points won't fall exactly on a 4th-order scaling curve.
 - Problem 4:
 - 5-point simpson means $N=4$ (4 slabs)
 - In Fortran, if doing double precision, you need to do the “d0” for your weights and roots, otherwise the error is too large

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: $Ax = b$
 - 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 $m \times n$ matrix
- x is $n \times 1$ (column) vector
- Result: b is $m \times 1$ (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 $m \times n$ matrix
- B is $n \times p$ matrix
- (AB) is $m \times p$ matrix
- Direct multiplication: $O(N^3)$ operation.
 - Some faster algorithms exist (make use of organization of submatrices for simplification)

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

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 straightforward:

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

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

- Algorithm we were taught in school for larger matrices is cumbersome (blackboard)

Review of Matrices

- Inverse
 - $A^{-1}A = A A^{-1} = I$
 - Formally, the solution to the linear system $Ax = b$ is $x = A^{-1} b$
 - 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: $|A - \lambda I| = 0$
 - Right eigenvectors: $Ar = \lambda r$
 - Left eigenvectors: $lA = \lambda l$

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 $Ax = b$

$$x_i = \frac{|A_i|}{|A|}$$

- Here, A_i is the matrix A with the i^{th} column replaced by the vector b
- **Blackboard example**

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)

$$\epsilon x_1 + x_2 + x_3 = 5$$

$$x_1 + x_2 = 3$$

$$x_1 + x_3 = 4$$

in limit $\epsilon \rightarrow 0$

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 $\sim O(N^3)$
- 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 |
\      /      \      /

/      \      /      \
|  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 |
\      /      \      /

/      \      /      \
|  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 |
\      /      \      /

/      \      /      \
|  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.]

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...

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: $\text{cond}(A) = \|A\| \|A^{-1}\|$
 - Further elucidation requires defining the norm
- Rule of thumb (Yakowitz & Szidarovszky, Eq. 2.31):
 - If x^* is the exact solution and x is the computed solution, then

$$\frac{\|x^* - x\|}{\|x^*\|} \approx \text{cond}(A) \cdot \epsilon_{\text{machine}}$$

- **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

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

- 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:

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

Sparse Matrices / Tridiagonal

- Recall from cubic 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 $a_0 = c_{N-1} = 0$

Matrix Inverse

- We can find the inverse using Gaussian elimination too.

- Start with $A A^{-1} = I$

- Let $x^{(\alpha)}$ represent a column of A^{-1}

- Let $e^{(\alpha)}$ represent a column of I

- Now we just need to solve the α linear systems:

$$Ax^{(\alpha)} = e^{(\alpha)}$$

- We can carry all the $e^{(\alpha)}$ 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...

LU Decomposition

- LU Decomposition is an alternate to Gaussian elimination
 - See Pang/Wikipedia for details
- Basic idea:
 - Find upper and lower triangular matrices such that $A = LU$
 - Express $Ax = b$ as $Ly = b$ and $Ux = y$
 - 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

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 ($\alpha x + y$)
 - Level 2: matrix-vector operations ($\alpha A x + \beta y$)
 - Level 3: matrix-matrix operations ($\alpha A B + \beta C$)
 - 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

- LAPACK

- The standard for linear algebra
- Built upon BLAS
- Routines named in the form $x_{yy}zzz$
 - 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

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 $1 \times N$ or $N \times 1$ matrices
 - Matrix objects cannot be $> \text{rank } 2$
 - 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...

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

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

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

$$\begin{array}{lcl} x_1^{(k+1)} & = & -\frac{1}{a_{11}} \left(a_{12}x_2^{(k)} + a_{13}x_3^{(k)} \dots a_{1n}x_n^{(k)} - b_1 \right) \\ x_2^{(k+1)} & = & -\frac{1}{a_{22}} \left(a_{21}x_1^{(k)} + a_{23}x_3^{(k)} \dots a_{2n}x_n^{(k)} - b_2 \right) \\ & & \vdots \\ x_n^{(k+1)} & = & -\frac{1}{a_{nn}} \left(a_{n1}x_1^{(k)} + a_{n2}x_2^{(k)} \dots a_{n,n-1}x_{n-1}^{(k)} - b_n \right) \end{array}$$

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...

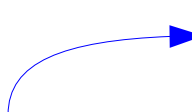
Multivariate Newton's Method

- Imagine a vector function: $\mathbf{f}(\mathbf{x})$

- $\mathbf{f}(\mathbf{x}) = (f_1(\mathbf{x}) \ f_2(\mathbf{x}) \ f_3(\mathbf{x}) \ \dots \ f_N(\mathbf{x}))^T$
- Column vector of unknowns: $\mathbf{x} = (x_1 \ x_2 \ x_3 \ \dots \ x_N)^T$

- We want to find the zeros:

- Initial guess: $\mathbf{x}^{(0)}$
- Taylor expansion:


$$f_i(\mathbf{x}^{(0)} + \delta \mathbf{x}) \approx 0 = f_i(\mathbf{x}^{(0)}) + \sum_{j=1}^N \underbrace{\frac{\partial f_i}{\partial x_j}}_{\text{This is the Jacobian}} \delta x_j + \dots$$

N equations + N
unknowns

This is the 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: $\mathbf{J} \delta \mathbf{x}^{(k)} = -\mathbf{f}(\mathbf{x}^{(k)})$, $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \delta \mathbf{x}^{(k)}$

Example: Lorenz Model

- Lorenz simplified model for global weather (model of convection):

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

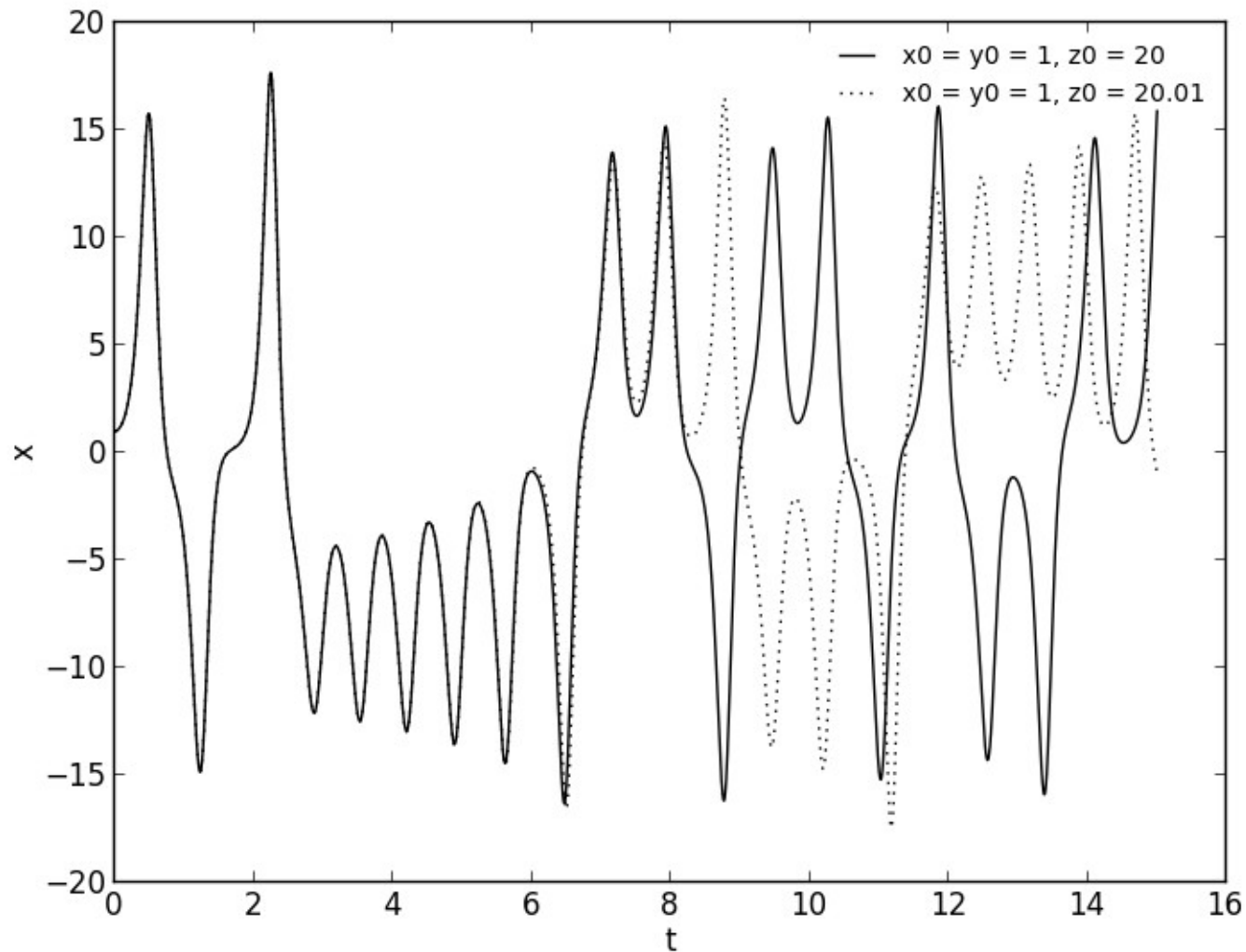
$$\frac{dy}{dt} = rx - y - xz$$

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

- Here, r , σ , b are constants
 - See Garcia (sections 3.4 and 4.3) and Lorenz paper linked on website
- This system is nonlinear
- Easy to solve with Runge-Kutta

Example: Lorenz Model

- Following Garcia, we choose $\sigma = 10$, $b = 8/3$, $r = 28$



- Chaos is simply a sensitivity to initial conditions

Example: Lorenz Model

130

JOURNAL OF THE ATMOSPHERIC SCIENCES

VOLUME 20

Deterministic Nonperiodic Flow¹

EDWARD N. LORENZ

Massachusetts Institute of Technology

(Manuscript received 18 November 1962, in revised form 7 January 1963)

ABSTRACT

Finite systems of deterministic ordinary nonlinear differential equations may be designed to represent forced dissipative hydrodynamic flow. Solutions of these equations can be identified with trajectories in phase space. For those systems with bounded solutions, it is found that nonperiodic solutions are ordinarily unstable with respect to small modifications, so that slightly differing initial states can evolve into considerably different states. Systems with bounded solutions are shown to possess bounded numerical solutions.

A simple system representing cellular convection is solved numerically. All of the solutions are found to be unstable, and almost all of them are nonperiodic.

The feasibility of very-long-range weather prediction is examined in the light of these results.

Example: Lorenz Model

To verify the existence of deterministic nonperiodic flow, we have obtained numerical solutions of a system of three ordinary differential equations designed to represent a convective process. These equations possess three steady-state solutions and a denumerably infinite set of periodic solutions. All solutions, and in particular the periodic solutions, are found to be unstable. The remaining solutions therefore cannot in general approach the periodic solutions asymptotically, and so are nonperiodic.

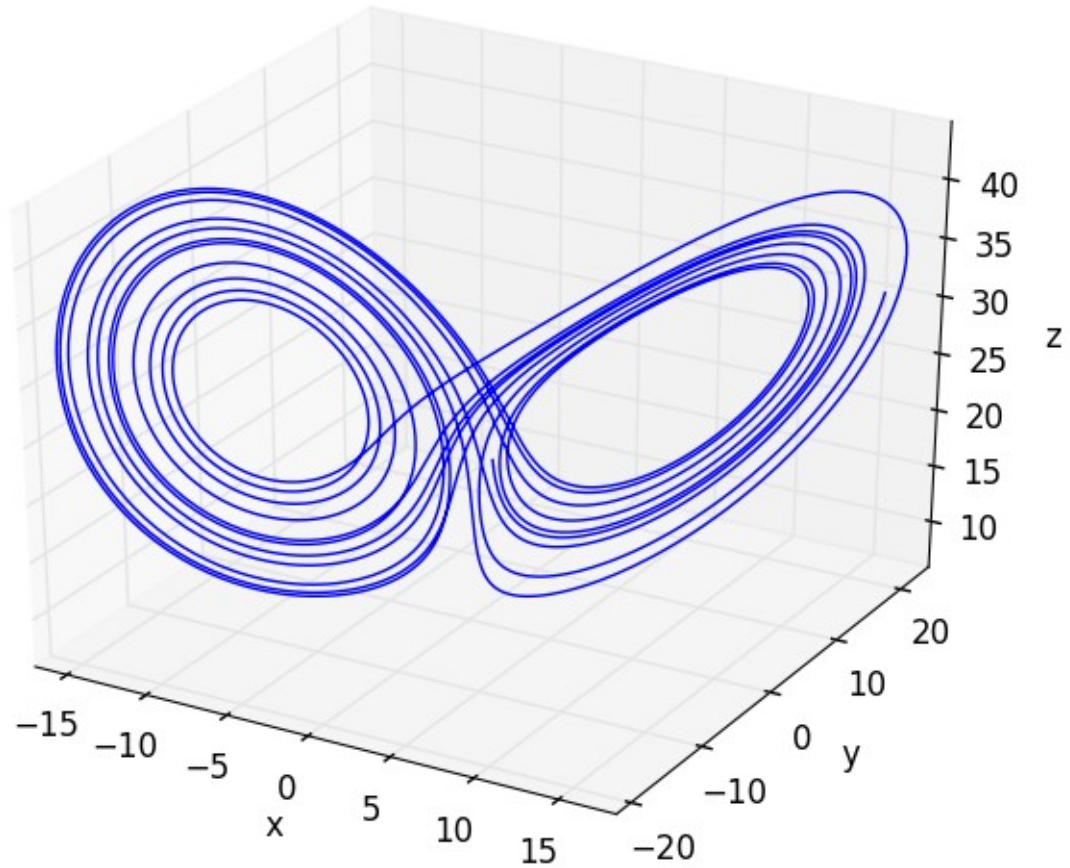
When our results concerning the instability of non-periodic flow are applied to the atmosphere, which is ostensibly nonperiodic, they indicate that prediction of the sufficiently distant future is impossible by any method, unless the present conditions are known exactly. In view of the inevitable inaccuracy and incompleteness of weather observations, precise very-long-range forecasting would seem to be non-existent.

There remains the question as to whether our results really apply to the atmosphere. One does not usually regard the atmosphere as either deterministic or finite, and the lack of periodicity is not a mathematical certainty, since the atmosphere has not been observed forever.

(from Lorenz
conclusions)

Example: Lorenz Model

- 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

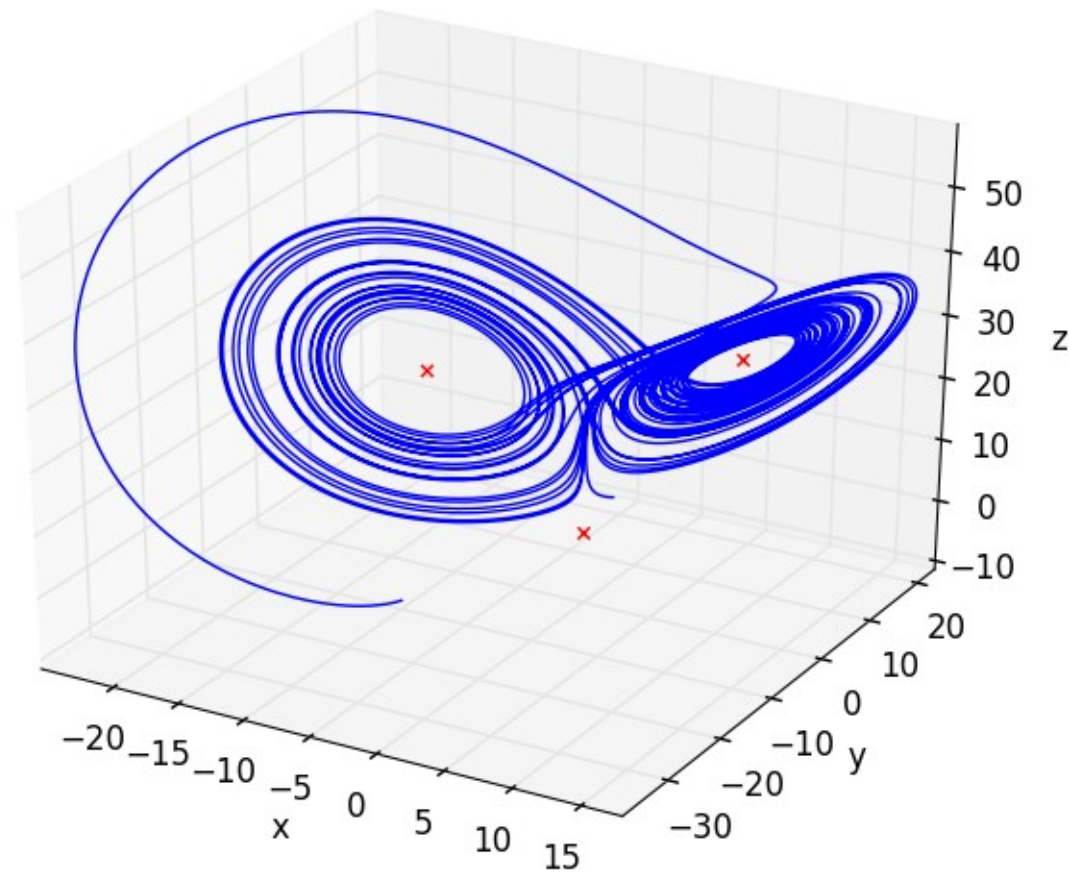
- Steady states—the zeros of the righthand side

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

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

- 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...

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 and eigenvectors
 - Many specialized methods exist
 - Built into LAPACK and python/SciPy
 - Pang gives a brief overview

Conjugate Gradient

- Iterative method for symmetric, positive definite ($\mathbf{x}^T \mathbf{A} \mathbf{x} > 0$) 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*