Numerical Linear Algebra – Solving Linear Systems

CS370 Lecture 29 – March 24, 2017

Numerical Linear Algebra

The study of algorithms for performing linear algebra operations ***numerically*** (i.e., approximately; on a computer with floating point).

Matrix/vector arithmetic, solving linear systems of equations, taking norms, factoring matrices, inverting matrices, finding eigenvalues, etc.

Due to floating point, the behaviour of our *numerical* (i.e. approximate) methods differ from exact/analytical counterparts.

Solving Linear Systems of Equations

Many many many practical problems rely on solving systems of linear equations of the form

$$Ax = b$$

where A is a matrix, b is a right-hand-side (column) vector, and x is a (column) vector of unknowns.

Applications

Application areas include:

- Fitting polynomials and splines.
- Implicit time integration.
- Optimization problems.
- Machine learning, statistics.
- Engineering.
- Computational finance.
- Computational biology.
- Image processing.
- Data mining & search.
- Computer vision.
- etc!

Nearly everywhere numerical computation is used, numerical linear algebra plays some role.

Example: Animating Fluids

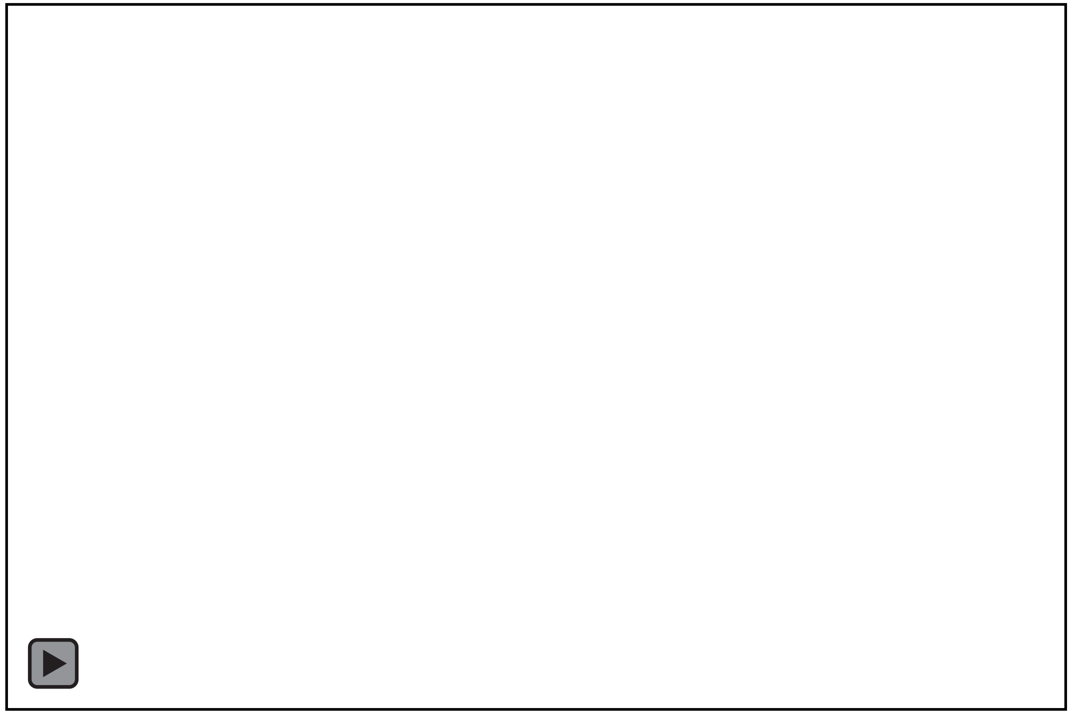
Computing one frame of animation requires solving a linear system with > **one million unknowns**.

- i.e. matrix A has dimensions > $1{,}000{,}000 \times 1{,}000{,}000$.

Must be done once per frame; animations are usually played back at 30 frames / second.

- e.g. for 10 seconds of video, must solve ~300 linear systems with size $1{,}000{,}000^2$.

So: We need methods to solve linear systems **efficiently** and **accurately**.



Review: Gaussian Elimination

In your linear algebra class, you would have seen *Gaussian Elimination*.

This involves:
- eliminating variables via row operations, until only one remains.
- back-substituting to recover the value of all the other variables.

This was done by applying combinations of:
(1) Multiplying a row by a constant.
(2) Swapping rows.
(3) Adding a multiple of one row to another row.

Gaussian Elimination: $Ax = b$

(Some) numerical algorithms use Gaussian elimination, too. But it is interpreted differently…

Our view will be the following:

1. **Factor** matrix A into $A = LU$, where L and U are *triangular*.
2. **Solve** $Lz = b$ for intermediate vector z.
3. **Solve** $Ux = z$ for x.

(Later: We may also need to reorder (*permute*) the equations, which leads to the modified factorization $PA = LU$.)

Gaussian Elimination as Factorization

Solve $\begin{bmatrix} 1 & 1 & 1 \\ 1 & -2 & 2 \\ 1 & 2 & -1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 4 \\ 2 \end{bmatrix}$ for the vector $\vec{x}$.

Algorithm – LU Factorization (Decomposition)

For $k = 1, .., n$ //Iterate over all rows.

  For $i = k + 1, ..., n$ //Iterate over each row i *beneath* row k.

    $mult := a_{ik}/a_{kk}$ //Determine row i's multiplicative factor.

    $a_{ik} := mult$ //Store this factor (instead of a zero).

    For $j = k + 1, ..., n$ //Iterate over all (non-zero) columns in the row.

      $a_{ij} := a_{ij} - mult * a_{kj}$ //Subtract the scaled row data.

    EndFor

  EndFor

EndFor

//Note: Resulting factors are stored back
//into A matrix for sake of space.

Gaussian Elimination: $Ax = b$

(Some) numerical algorithms use Gaussian elimination, too. But it is interpreted differently…

Our view will be the following:

1. **Factor** matrix A into $A = LU$, where L and U are *triangular*.
2. **Solve** $Lz = b$ for intermediate vector z.
3. **Solve** $Ux = z$ for x.

(Later: We may also need to reorder (*permute*) the equations, which leads to the factorization $PA = LU$.)

Gaussian Elimination: $Ax = b$

(Some) numerical algorithms use Gaussian elimination, too. But it is interpreted differently…

Our view will be the following:

1. **Factor** matrix A into $A = LU$, where L and U are *triangular*.
2. **Solve** $Lz = b$ for intermediate vector z.
3. **Solve** $Ux = z$ for x.

(Later: We may also need to reorder (*permute*) the equations, which leads to the factorization $PA = LU$.)

Factorization and Triangular Solves

Given the factorization $A = LU$, we can rapidly solve $Ax = b$. How?

This is the same as solving $LUx = b$. Rewrite it as $L(Ux) = b$.

Define $z = Ux$, and rewrite this as two separate solves:

First: Solve $Lz = b$ for z.

Then: Solve $Ux = z$ for x.

But why are two solves better than one (i.e., our original system)?

Triangular Solves – Advantage?

Our two solves are:

$Lz = b$ for z. "Forward Solve"

$Ux = z$ for x. "Backward Solve"

L and U are both **triangular**: all entries above, or below, the diagonal are zero, respectively.

This makes them easier (i.e., more ***efficient***) to solve. (More later.)

Backward Solve

For example, the "backward solve" $Ux = z$ is just the back-substitution step from standard Gaussian elimination.

e.g. Our example last time required back-substitution on:

$$\begin{bmatrix} 1 & 1 & 1 \\ 0 & -3 & 1 \\ 0 & 0 & -5/3 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 4 \\ 10/3 \end{bmatrix}$$

Forward Solve

The “forward solve”, $Lz = b$ gives us the same RHS as we had after our row operations.

e.g. In the earlier example, $Lz = b$ is

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & -1/3 & 1 \end{bmatrix} \begin{bmatrix} z_0 \\ z_1 \\ z_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 4 \\ 2 \end{bmatrix}$$

Solving it gives $z = \begin{bmatrix} 0 \\ 4 \\ 10/3 \end{bmatrix}$, exactly the RHS we had after row operations on the augmented system!

Triangular Solves (Backward & Forward)

$Lz = b$ is called the forward solve (or forward substitution).

e.g., $L = \begin{bmatrix} 1 & 0 & 0 \\ X & 1 & 0 \\ X & X & 1 \end{bmatrix}$

For $i = 1, \ldots, n$
$\quad z_i := b_i$
$\quad$For $j = 1, \ldots, i-1$
$\quad\quad z_i := z_i - l_{ij} * z_j$
$\quad$EndFor
EndFor

$Ux = z$ is called the back(ward) solve (or back(ward) substitution).

e.g. $U = \begin{bmatrix} X & X & X \\ 0 & X & X \\ 0 & 0 & X \end{bmatrix}$

For $i = n, \ldots, 1$
$\quad x_i := z_i$
$\quad$For $j = i+1, \ldots, n$
$\quad\quad x_i := x_i - u_{ij} * x_j$
$\quad$EndFor
$\quad x_i := x_i / u_{ii}$
EndFor

Advantage: Different Right-Hand-Sides

The RHS vector b is not needed to factor A into $\mathrm{A} = LU$.

Factoring work can be reused for different b's!

Only have to redo the (cheaper) forward/backward solves.

e.g., Let $A = LU$ with $L = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & -1/3 & 1 \end{bmatrix}$ and $U = \begin{bmatrix} 1 & 1 & 1 \\ 0 & -3 & 1 \\ 0 & 0 & -5/3 \end{bmatrix}$

(i.e. *Same factorization* from earlier example.)

Solve $Ax = b$ for $b = \begin{bmatrix} 6 \\ 3 \\ 2 \end{bmatrix}$, by first solving $Lz = b$ and then $Ux = z$.

Different RHS: Example Application

e.g. Polynomial fitting of $y = p(x)$.

If you change *only* the y-values being interpolated, you don't have to refactor the Vandermonde matrix.

$$V = \begin{bmatrix} 1 & x_1 & \ldots & x_1^{n-1} \\ 1 & x_2 & \ldots & x_2^{n-1} \\ & & \ldots & \\ 1 & x_n & \ldots & x_n^{n-1} \end{bmatrix}, \quad \vec{c} = \begin{bmatrix} c_1 \\ \ldots \\ c_n \end{bmatrix} \quad \text{and} \quad \vec{y} = \begin{bmatrix} y_1 \\ \ldots \\ y_n \end{bmatrix}$$