Lecture 5

Matrix, Vector Operations

T. Gambill

Department of Computer Science
University of Illinois at Urbana-Champaign

February 14, 2012

Goals:

- recall linear algebra ouch!
- motivation: why do we need to solve a linear system of equations?
- cost analysis of basic operations
- identify basic solution schemes to systems

Prereq

Linear Algebra is a prerequisite of the course!

- look at Lecture 5a notes

Why this is important:

- matrix problems arise in many areas of CS (information sciences, graphics, design, etc)
- Basic Linear Algebra Subprograms (BLAS) is an interface standard for operations
- simple systems set the stage for further development: avoiding error, avoiding large costs

Motivation: Newton's Method in higher dimensions

Given an initial guess $\mathbf{x_0}$ we compute the matrix $\mathbf{J}(\mathbf{x_0})$ is called the Jacobian,

$$\mathbf{J_{ij}} = \frac{\partial f_i(\mathbf{x_0})}{\partial x_j}.$$

and we solve the following system of equations for $\mathbf{x_1}$,

$$\mathbf{x_1} = \mathbf{x_0} - \mathbf{J}^{-1}(\mathbf{x_0}) * \mathbf{f}(\mathbf{x_0})$$

Although, we will later see why we should (and can) avoid computing the inverse of the Jacobian and instead solve the system of equations,

$$\mathbf{J}(\mathbf{x_0}) * (\mathbf{x_1} - \mathbf{x_0}) = -\mathbf{f}(\mathbf{x_0})$$

We check to see if $\mathbf{x_1}$ is a root and if not then we continue to iterate.

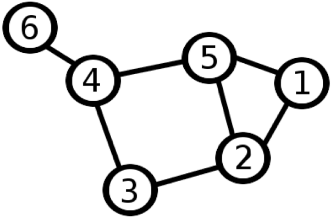
$$\mathbf{J}(\mathbf{x_k}) * (\mathbf{x_{k+1}} - \mathbf{x_k}) = -\mathbf{f}(\mathbf{x_k})$$

We may save calculations of $\mathbf{J}(\mathbf{x_k})$ by using the same value of $\mathbf{J}(\mathbf{x_k})$ over several iterations.

Motivation: Graph Theory

- Given a graph, we can construct an associated square matrix A, called the Adjacency Matrix
- If we denote $A = [a_{ij}]$ then

$$a_{ij} = \begin{cases} 1 & \text{if node i is connected to node j} \\ 0 & \text{otherwise} \end{cases} \tag{1}$$



$$\Rightarrow \begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

Motivation: Graph Theory

A path (walk) of length l from node i to node j in a graph is a sequence of l edges of the graph that starts at node i and terminates at node j.

Counting paths

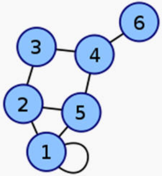
Given an adjacency matrix A corresponding to a graph then the number of different paths of length $l > 0$ from node i to node j equals the value b_{ij} where $[b_{ij}] = B = A^l$.

What is the cost of computing A^l?

Motivation: Graph Theory

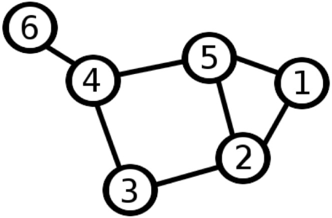
- Given a graph, we can construct an associated square matrix D, called the Degree matrix
- If we denote $D = [d_{ij}]$ then

$$d_{ij} = \begin{cases} k & \text{if i = j and node i has k edges incident(connected) to node i} \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

Vertex labeled graph	Degree matrix
	$\begin{pmatrix} 4 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$

Motivation: Graph Theory

- Given a graph, construct associated square matrix $\mathcal{L}$, called the graph Laplacian
- $\mathcal{L} = D - A$ where D is the Degree Matrix and A is the Adjacency Matrix for the graph.



$$\Rightarrow \begin{bmatrix} 2 & -1 & 0 & 0 & -1 & 0 \\ -1 & 3 & -1 & 0 & -1 & 0 \\ 0 & -1 & 2 & -1 & 0 & 0 \\ 0 & 0 & -1 & 3 & -1 & -1 \\ -1 & -1 & 0 & -1 & 3 & 0 \\ 0 & 0 & 0 & -1 & 0 & 1 \end{bmatrix}$$

Motivation: Graph Theory

Graph is Laplacian useful for

- Calculating spanning trees
- Partitioning a graph evenly
- and many more....

To use the graph Laplacian, you need to solve $\mathcal{L}x = b$ for many different vectors, b.

Motivation: Graph Theory (Multiple Right Hand Sides)

- Solve $Ax = b$ for many different b vectors
- For k different b vectors, Gaussian Elimination costs $\mathcal{O}(kn^3)$
- We can do better: LU factorization

Matrix Inverse

Let A be a square (i.e. $n \times n$) with real elements. The *inverse* of A is designated A^{-1}, and has the property that

$$A^{-1}A = I \qquad \text{and} \qquad AA^{-1} = I$$

The **formal solution** to $A\mathbf{x} = \mathbf{b}$ is $\mathbf{x} = A^{-1}\mathbf{b}$.

$$\begin{aligned} A\mathbf{x} &= \mathbf{b} \\ A^{-1}A\mathbf{x} &= A^{-1}\mathbf{b} \\ I\mathbf{x} &= A^{-1}\mathbf{b} \\ \mathbf{x} &= A^{-1}\mathbf{b} \end{aligned}$$

Matrix Inverse

- formal solution to $A\mathbf{x} = \mathbf{b}$ is $\mathbf{x} = A^{-1}\mathbf{b}$

Matrix Inverse

- formal solution to $A\mathbf{x} = \mathbf{b}$ is $\mathbf{x} = A^{-1}\mathbf{b}$
- BUT it is *bad* evaluate $\mathbf{x}$ this way

Matrix Inverse

- formal solution to $A\mathbf{x} = \mathbf{b}$ is $\mathbf{x} = A^{-1}\mathbf{b}$
- BUT it is *bad* evaluate $\mathbf{x}$ this way
- why?

Matrix Inverse

- formal solution to $A\mathbf{x} = \mathbf{b}$ is $\mathbf{x} = A^{-1}\mathbf{b}$
- BUT it is *bad* evaluate $\mathbf{x}$ this way
- why?
- we will not form A^{-1}, but solve for $\mathbf{x}$ directly using Gaussian elimination.

Why do we care as Numerical Analysts?

Open questions:

- How *expensive* is it to solve $A\mathbf{x} = \mathbf{b}$?
- What problems (errors) will we encounter solving $A\mathbf{x} = \mathbf{b}$?
- Some matrices are easy/cheap to use: diagonal, tridiagonal, etc.
  - are there others? what makes something a ”good” matrix numerically?
  - are there bad ones? how do we identify them numerically?
- what do actual numerical analysts, engineers, developers, etc use?!?!

Formal Solution when A is $n \times n$

The *formal solution* to $A\mathbf{x} = \mathbf{b}$ is

$$\mathbf{x} = A^{-1}\mathbf{b}$$

where A is $n \times n$.
If A^{-1} exists then A is said to be **nonsingular**.
If A^{-1} does not exist then A is said to be **singular**.

Formal Solution when A is $n \times n$

If A^{-1} exists then

$$A\mathbf{x} = \mathbf{b} \qquad \Longrightarrow \qquad \mathbf{x} = A^{-1}\mathbf{b}$$

but

Do not compute the solution to $A\mathbf{x} = \mathbf{b}$ by finding A^{-1}, and then multiplying $\mathbf{b}$ by A^{-1}!

We see: $\mathbf{x} = A^{-1}\mathbf{b}$

We do: **Solve $A\mathbf{x} = \mathbf{b}$ by Gaussian elimination or an equivalent algorithm**

Singularity of A

If an $n \times n$ matrix, A, is **singular** then

- the columns of A are linearly dependent
- the rows of A are linearly dependent
- $\text{rank}(A) < n$
- $\det(A) = 0$
- A^{-1} does not exist
- a solution to $A\mathbf{x} = \mathbf{b}$ may not exist
- If a solution to $A\mathbf{x} = \mathbf{b}$ exists, it is not unique

Summary of Requirements for Solution of $Ax = b$

Given the $n \times n$ matrix A and the $n \times 1$ vector, $\mathbf{b}$

- the solution to $A\mathbf{x} = \mathbf{b}$ exists and is unique for any $\mathbf{b}$ if and only if $\mathrm{rank}(A) = n$.

Recall: rank $=$ # of linearly independent rows or columns

Recall: $Range(A)$ $=$ set of vectors y such that $A\mathbf{x} = \mathbf{y}$ for some $\mathbf{x}$

Solving a system

$$A\mathbf{x} = \mathbf{b}$$

Three situations:

1. A is nonsingular: There exists a unique solution $\mathbf{x} = A^{-1}\mathbf{b}$
2. A is singular and $\mathbf{b} \in Range(A)$: There are infinite solutions.
3. A is singular and $\mathbf{b} \notin Range(A)$: There are no solutions.

1. $A = \begin{bmatrix} 2 & 0 \\ 0 & 4 \end{bmatrix}$ $\mathbf{b} = \begin{bmatrix} 1 \\ 8 \end{bmatrix}$, then $\mathbf{x} = \begin{bmatrix} 1/2 \\ 2 \end{bmatrix}$.
2. $A = \begin{bmatrix} 2 & 0 \\ 0 & 0 \end{bmatrix}$ $\mathbf{b} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$, then infinitely many solutions. $\mathbf{x} = \begin{bmatrix} 1/2 \\ \alpha \end{bmatrix}$.
3. $A = \begin{bmatrix} 2 & 0 \\ 0 & 0 \end{bmatrix}$ $\mathbf{b} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$, then no solutions.

Solving a system, Cramer's Rule

The Determinant

Given a matrix $A_{nxn} = (a_{ij})$ the determinant is defined as,

$$det(A) = \sum_{i=0}^{n} (-1)^{i+j} * a_{ij} * det(A_{i,j})$$

where j(the column) is fixed and $A_{i,j}$ represents the "reduced" matrix of A with it's i^{th} row and j^th column removed.

The above definition is called the column sum expansion. There is also a row sum expansion and other ways to compute the determinant. Note that the det function maps nxn (square) matrices into $\mathbb{R}$ the real numbers.

Solving a system, Cramer's Rule

The det function has the following properties:

1. $det(AB) = det(A)det(B)$
2. $det(I) = 1$ where I is the nxn identity matrix
3. $det(A^T) = det(A)$
4. $det(A) = 0$ if and only if A is singular
5. $|det(A)| =$the volume of the parallelepiped(parallelogram for $n = 2$) formed by the columns of A

Examples

- $det\left(\begin{bmatrix}1 & 2\\ 2 & 4\end{bmatrix}\right) = 0$
- $det\left(\begin{bmatrix}2 & 0 & 0\\ 0 & 2 & 0\\ 0 & 0 & 2\end{bmatrix}\right) = 8$

Use the Python $numpy.linalg.det$ function to compute determinants. Better to use the Python $numpy.linalg.cond$ function to determine singular matrices.

Solving a system, Cramer's Rule

The solution of the system of equations $A\mathbf{x} = \mathbf{b}$ is given by,

Cramer's Rule

$$x_k = \frac{det(A|_k\mathbf{b})}{det(A)} \tag{3}$$

where $A|_k\mathbf{b}$ denotes the matrix A with the k^{th} column replaced by $\mathbf{b}$.

but **Cramer's Rule is bad!!!**

What's the big deal? ....cost

Consider the time it takes to compute one of the determinants. Denote $A' = (a_{ij}) = A|_k\mathbf{b}$ for a fixed k and expand the determinant down the first column,

$$det((a_{ij})) = a_{11}(-1)^{1+1}det(A'_{1,1}) + a_{21}(-1)^{2+1}det(A'_{2,1}) + \ldots + a_{n1}(-1)^{n+1}det(A'_{n,1}) \tag{4}$$

so the cost is greater than n multiplications and $n-1$ additions plus the cost of performing the n determinants $det(A'_{i,1})$ for $i = 1, \ldots, n$. We can write a formula for a lower bound on the cost of computing the determinant of an nxn matrix A' as,

$$cost(det(A')) = n * cost(det(A'')) \tag{5}$$

where A'' is a matrix of size $(n-1)x(n-1)$. Since for a matrix A of size $1x1$ we have $cost(det(A)) = 1$ we can write, for a lower bound on the cost,

$$cost(det(A')) = n! \tag{6}$$

O(n!)

How large is $n!$?

Sterling's Formula

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \tag{7}$$

- humans: milliFLOPS
- hand calculators: 10 FLOPS
- desktops: a few GFLOPS (10^9 FLOPS)
- Intel Core i7 980 XE 107.55 GFLOPS
- ATI Radeon HD4800 1 TERAFLOP (10^{12} FLOPS)
- Tianhe-I 2.5 petaflops (10^{15} FLOPS)

Example: $n!$, for $n = 100$

$$100! \approx 9.3 * 10^{157} \tag{8}$$

At 10^{12} FLOPS = 1 TERAFLOPS it would take $9.3 * 10^{157}/10^{12}$ seconds $\approx 3 * 10^{138}$ years where the age of the universe is ONLY $\approx 1.4 * 10^{10}$ years!!!

Remember Big-O ?

How to measure the impact of n on algorithmic cost?

$\mathcal{O}(\cdot)$

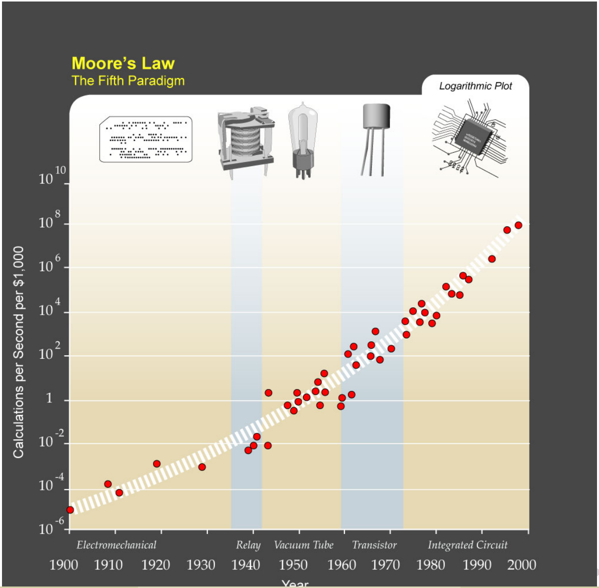
Let $g(n)$ be a function of n. Then define

$$\mathcal{O}(g(n)) = \{f(n) \mid \exists c, n_0 > 0 \,:\, 0 \leqslant f(n) \leqslant cg(n),\ \forall n \geqslant n_0\}$$

That is, $f(n) \in \mathcal{O}(g(n))$ if there is a constant c such that $0 \leqslant f(n) \leqslant cg(n)$ is satisfied.

- assume non-negative functions (otherwise add $|\cdot|$) to the definitions
- $f(n) \in \mathcal{O}(g(n))$ represents an asymptotic upper bound on $f(n)$ up to a constant
- example: $f(n) = 3\sqrt{n} + 2\log n + 8n + 85n^2 \in \mathcal{O}(n^2)$

Moore...


Moore's Law
The Fifth Paradigm
Logarithmic Plot
Calculations per Second per $1,000
10^10
10^8
10^6
10^4
10^2
1
10^-2
10^-4
10^-6
Electromechanical
Relay
Vacuum Tube
Transistor
Integrated Circuit
1900 1910 1920 1930 1940 1950 1960 1970 1980 1990 2000
Year

BLAS

Basic Linear Algebra Subprograms (BLAS) interface introduced APIs for common linear algebra tasks

- Level 1: vector operations (dot products, vector norms, etc) e.g.

$$\mathbf{y} \leftarrow \alpha \mathbf{x} + \mathbf{y} \tag{9}$$

$$\mathbf{y} \leftarrow \mathbf{x} * \mathbf{y} \tag{10}$$

$$\mathbf{y} \leftarrow ||\mathbf{x}|| \tag{11}$$

- Level 2: matrix-vector operations, e.g.

$$\mathbf{y} \leftarrow \alpha A * \mathbf{x} + B * \mathbf{y}$$

- Level 3: matrix-matrix operations, e.g.

$$C \leftarrow \alpha A * B + \beta C$$

- optimized versions of the reference BLAS are used everyday: ATLAS, etc.

vec-vec, mat-vec, mat-mat

- inner product of $\mathbf{u}$ and $\mathbf{v}$ both $[n \times 1]$

$$\sigma = \mathbf{u}^T\mathbf{v} = u_1 v_1 + \cdots + u_n v_n$$

- $\rightarrow n$ multiplies, $n-1$ additions
- $\rightarrow \mathcal{O}(n)$ flops

vec-vec, mat-vec, mat-mat

- mat-vec of A ($[n \times n]$) and u ($[n \times 1]$)

```
for i = 1,...,n
   for j = 1,...,n
      v(i) = a(i,j)u(j) + v(i)
   end
end
```

- $\rightarrow n^2$ multiplies, n^2 additions
- $\rightarrow \mathcal{O}(n^2)$ flops

vec-vec, mat-vec, mat-mat

- mat-mat of A ($[n \times n]$) and B ($[n \times n]$)

```
for j = 1,...,n
   for k = 1,...,n
      for i = 1,...,n
         C(k,j) = A(k,i)B(i,j) + C(k,j)
      end
   end
end
```

- $\rightarrow n^3$ multiplies, n^3 additions
- $\rightarrow \mathcal{O}(n^3)$ flops

vec-vec, mat-vec, mat-mat

Operation	FLOPS
$u^T v$	$\mathcal{O}(n)$
Au	$\mathcal{O}(n^2)$
AB	$\mathcal{O}(n^3)$

Gaussian Elimination

- Solving Diagonal Systems
- Solving Triangular Systems
- Gaussian Elimination Without Pivoting
  - Hand Calculations
  - Cartoon Version
  - The Algorithm
- Gaussian Elimination with Pivoting
  - Row or Column Interchanges, or Both
  - Implementation
- Solving Systems with the Backslash Operator

Solving Diagonal Systems

The system defined by $A\mathbf{x} = \mathbf{b}$, where

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 5 \end{bmatrix} \qquad b = \begin{bmatrix} -1 \\ 6 \\ -15 \end{bmatrix}$$

Solving Diagonal Systems

The system defined by $A\mathbf{x} = \mathbf{b}$, where

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 5 \end{bmatrix} \qquad b = \begin{bmatrix} -1 \\ 6 \\ -15 \end{bmatrix}$$

is equivalent to

$$\begin{array}{ccccc} x_1 & & & = & -1 \\ & 3x_2 & & = & 6 \\ & & 5x_3 & = & -15 \end{array}$$

Solving Diagonal Systems

The system defined by $A\mathbf{x} = \mathbf{b}$, where

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 5 \end{bmatrix} \qquad b = \begin{bmatrix} -1 \\ 6 \\ -15 \end{bmatrix}$$

is equivalent to

$$\begin{array}{rrrcr} x_1 & & & = & -1 \\ & 3x_2 & & = & 6 \\ & & 5x_3 & = & -15 \end{array}$$

The solution is

$$x_1 = -1 \qquad x_2 = \frac{6}{3} = 2 \qquad x_3 = \frac{-15}{5} = -3$$

Solving Diagonal Systems

Listing 1: Diagonal System Solution

```
given A, b
for i = 1...n
    x_i = b_i/a_{i,i}
end
```

In Python:

```
>>> A = ...            #  A is a diagonal matrix
>>> b = ...            #  b is a row vector
>>> x = b/numpy.diag(A)
```

This is the *only* place where element-by-element division (/) has anything to do with solving linear systems of equations.

Operations?

Try...

Sketch out an operation count to solve a diagonal system of equations...

Operations?

Try...

Sketch out an operation count to solve a diagonal system of equations...

cheap!

one division n times $\longrightarrow \mathcal{O}(n)$ FLOPS

Triangular Systems

The generic lower and upper triangular matrices are

$$L = \begin{bmatrix} l_{11} & 0 & \cdots & 0 \\ l_{21} & l_{22} & & 0 \\ \vdots & & \ddots & \vdots \\ l_{n1} & & \cdots & l_{nn} \end{bmatrix}$$

and

$$U = \begin{bmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ 0 & u_{22} & & u_{2n} \\ \vdots & & \ddots & \vdots \\ 0 & & \cdots & u_{nn} \end{bmatrix}$$

The triangular systems

$$L\mathbf{y} = \mathbf{b} \qquad U\mathbf{x} = \mathbf{c}$$

are easily solved by **forward substitution** and **backward substitution**, respectively

Solving Triangular Systems

The system defined by $A\mathbf{x} = \mathbf{b}$, where

$$A = \begin{bmatrix} 4 & 0 & 0 \\ -2 & 3 & 0 \\ 2 & 1 & -2 \end{bmatrix} \qquad b = \begin{bmatrix} 8 \\ -1 \\ 9 \end{bmatrix}$$

Solving Triangular Systems

The system defined by $A\mathbf{x} = \mathbf{b}$, where

$$A = \begin{bmatrix} 4 & 0 & 0 \\ -2 & 3 & 0 \\ 2 & 1 & -2 \end{bmatrix} \qquad b = \begin{bmatrix} 8 \\ -1 \\ 9 \end{bmatrix}$$

is equivalent to

$$\begin{array}{rcrcrcr} 4x_1 & & & & & = & 8 \\ -2x_1 & + & 3x_2 & & & = & -1 \\ 2x_1 & + & x_2 & + & -2x_3 & = & 9 \end{array}$$

Solving Triangular Systems

The system defined by $A\mathbf{x} = \mathbf{b}$, where

$$A = \begin{bmatrix} 4 & 0 & 0 \\ -2 & 3 & 0 \\ 2 & 1 & -2 \end{bmatrix} \qquad b = \begin{bmatrix} 8 \\ -1 \\ 9 \end{bmatrix}$$

is equivalent to

$$\begin{array}{rcrcrcr} 4x_1 & & & & & = & 8 \\ -2x_1 & + & 3x_2 & & & = & -1 \\ 2x_1 & + & x_2 & + & -2x_3 & = & 9 \end{array}$$

Solve in forward order (first equation is solved first)

$$x_1 = \frac{8}{4} = 2 \qquad x_2 = \frac{1}{3}(-1 + 2x_1) = \frac{3}{3} = 1$$

$$x_3 = \frac{1}{-2}(9 - x_2 - 2x_1) = \frac{4}{-2} = -2$$

Solving Triangular Systems

Solving for $x_1, x_2, \ldots, x_n$ for a lower triangular system is called **forward substitution**.

```
given L, b
x_1 = b_1/ℓ_11
for i = 2...n
    s = b_i
    for j = 1...i-1
        s = s - ℓ_{i,j} x_j
    end
    x_i = s/ℓ_{i,i}
end
```

Solving Triangular Systems

Solving for $x_1, x_2, \ldots, x_n$ for a lower triangular system is called **forward substitution**.

```
given L, b
x_1 = b_1/ℓ_11
for i = 2...n
    s = b_i
    for j = 1...i-1
        s = s - ℓ_{i,j} x_j
    end
    x_i = s/ℓ_{i,i}
end
```

Using forward or backward substitution is sometimes referred to as performing a **triangular solve**.

Operations?

Try...

Sketch out an operation count to solve a triangular system of equations...

Operations?

Try...

Sketch out an operation count to solve a triangular system of equations...

cheap!

- begin in the upper corner: 1 div
- row 2: 1 mult, 1 add, 1 div, or 3 FLOPS
- row 3: 2 mult, 2 add, 1 div, or 5 FLOPS
- row 4: 3 mult, 3 add, 1 div, or 7 FLOPS
- $\vdots$
- row k: $2k - 1$ FLOPS

Total FLOPS? $\sum_{k=1}^{n} 2k - 1 = 2\frac{n(n+1)}{2} - n$ or $\mathcal{O}(n^2)$ FLOPS