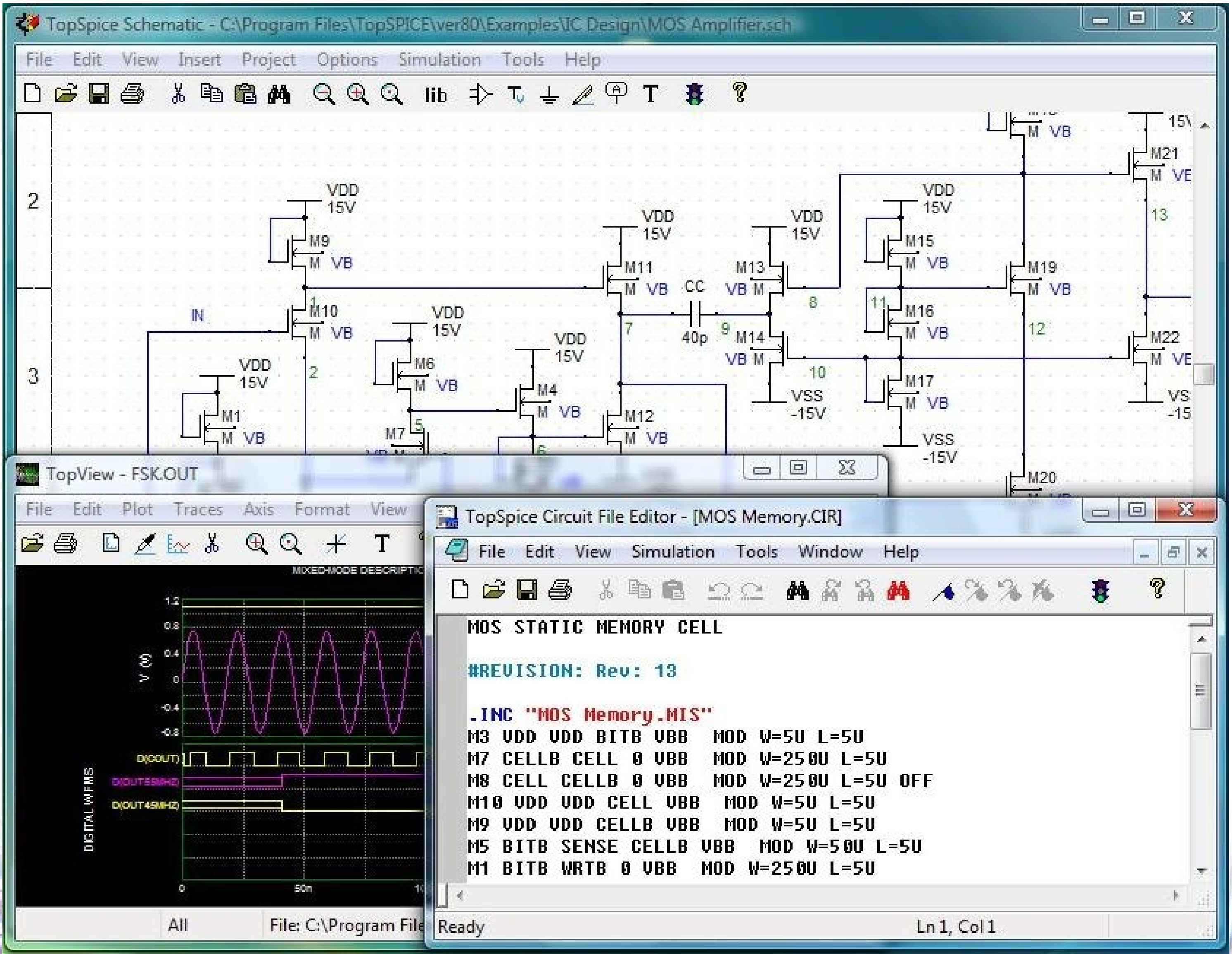
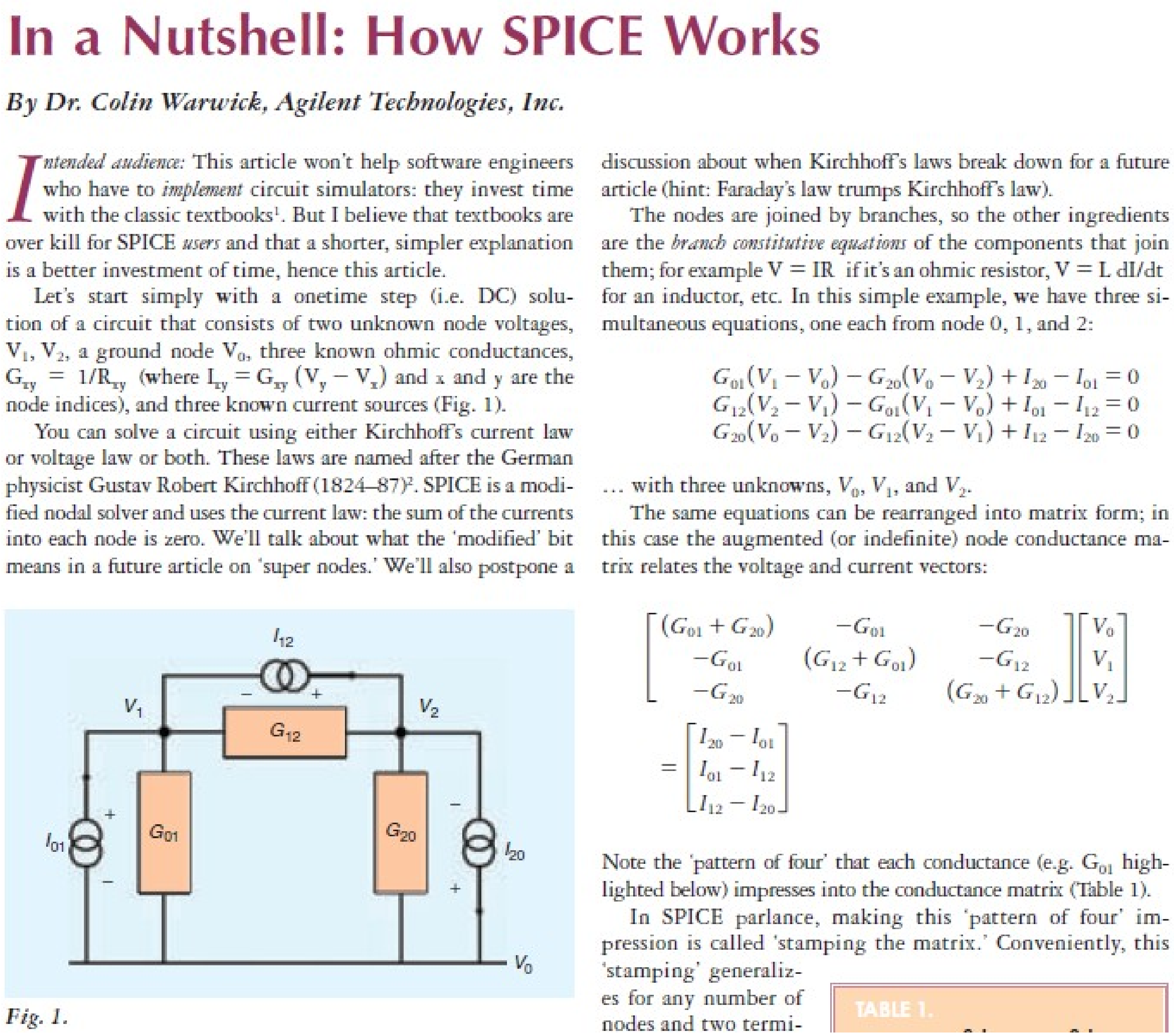
SPICE

- Simulation Program with Integrated Circuit Emphasis


TopSpice Schematic - C:\Program Files\TopSPICE\ver80\Examples\IC Design\MOS Amplifier.sch
File Edit View Insert Project Options Simulation Tools Help
TopView - FSK.OUT
File Edit Plot Traces Axis Format View
TopSpice Circuit File Editor - [MOS Memory.CIR]
File Edit View Simulation Tools Window Help
Ready
Ln 1, Col 1

```
MOS STATIC MEMORY CELL

#REVISION: Rev: 13

.INC "MOS Memory.MIS"
M3 VDD VDD BITB VBB  MOD W=5U L=5U
M7 CELLB CELL 0 VBB  MOD W=250U L=5U
M8 CELL CELLB 0 VBB  MOD W=250U L=5U OFF
M10 VDD VDD CELL VBB  MOD W=5U L=5U
M9 VDD VDD CELLB VBB  MOD W=5U L=5U
M5 BITB SENSE CELLB VBB  MOD W=50U L=5U
M1 BITB WRTB 0 VBB  MOD W=250U L=5U
```

SPICE under the covers



In a Nutshell: How SPICE Works

By Dr. Colin Warwick, Agilent Technologies, Inc.

Intended audience: This article won't help software engineers who have to *implement* circuit simulators: they invest time with the classic textbooks[1]. But I believe that textbooks are over kill for SPICE *users* and that a shorter, simpler explanation is a better investment of time, hence this article.

Let's start simply with a onetime step (i.e. DC) solution of a circuit that consists of two unknown node voltages, V_1, V_2, a ground node V_0, three known ohmic conductances, $G_{xy} = 1/R_{xy}$ (where $I_{xy} = G_{xy}(V_y - V_x)$ and x and y are the node indices), and three known current sources (Fig. 1).

You can solve a circuit using either Kirchhoff's current law or voltage law or both. These laws are named after the German physicist Gustav Robert Kirchhoff (1824–87)[2]. SPICE is a modified nodal solver and uses the current law: the sum of the currents into each node is zero. We'll talk about what the 'modified' bit means in a future article on 'super nodes.' We'll also postpone a discussion about when Kirchhoff's laws break down for a future article (hint: Faraday's law trumps Kirchhoff's law).

The nodes are joined by branches, so the other ingredients are the *branch constitutive equations* of the components that join them; for example V = IR if it's an ohmic resistor, V = L dI/dt for an inductor, etc. In this simple example, we have three simultaneous equations, one each from node 0, 1, and 2:

$$\begin{aligned} G_{01}(V_1 - V_0) - G_{20}(V_0 - V_2) + I_{20} - I_{01} = 0 \\ G_{12}(V_2 - V_1) - G_{01}(V_1 - V_0) + I_{01} - I_{12} = 0 \\ G_{20}(V_0 - V_2) - G_{12}(V_2 - V_1) + I_{12} - I_{20} = 0 \end{aligned}$$

… with three unknowns, V_0, V_1, and V_2.

The same equations can be rearranged into matrix form; in this case the augmented (or indefinite) node conductance matrix relates the voltage and current vectors:

$$\begin{bmatrix} (G_{01}+G_{20}) & -G_{01} & -G_{20} \\ -G_{01} & (G_{12}+G_{01}) & -G_{12} \\ -G_{20} & -G_{12} & (G_{20}+G_{12}) \end{bmatrix} \begin{bmatrix} V_0 \\ V_1 \\ V_2 \end{bmatrix} = \begin{bmatrix} I_{20} - I_{01} \\ I_{01} - I_{12} \\ I_{12} - I_{20} \end{bmatrix}$$

Note the 'pattern of four' that each conductance (e.g. G_{01} highlighted below) impresses into the conductance matrix (Table 1).

In SPICE parlance, making this 'pattern of four' impression is called 'stamping the matrix.' Conveniently, this 'stamping' generalizes for any number of nodes and two termi-

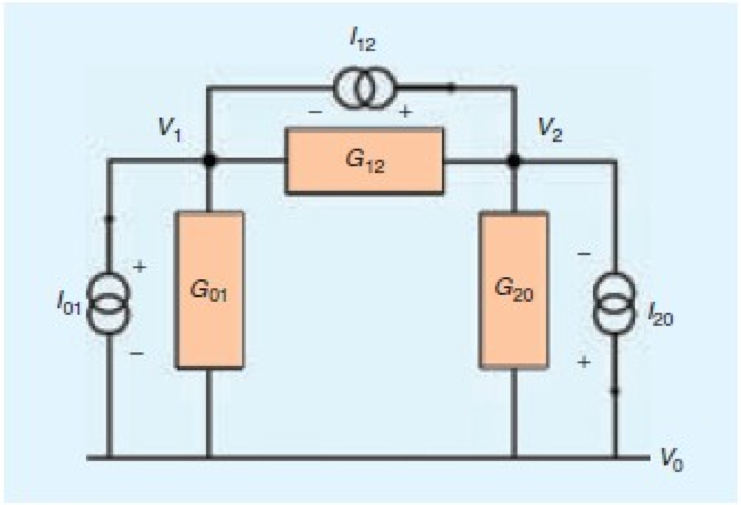
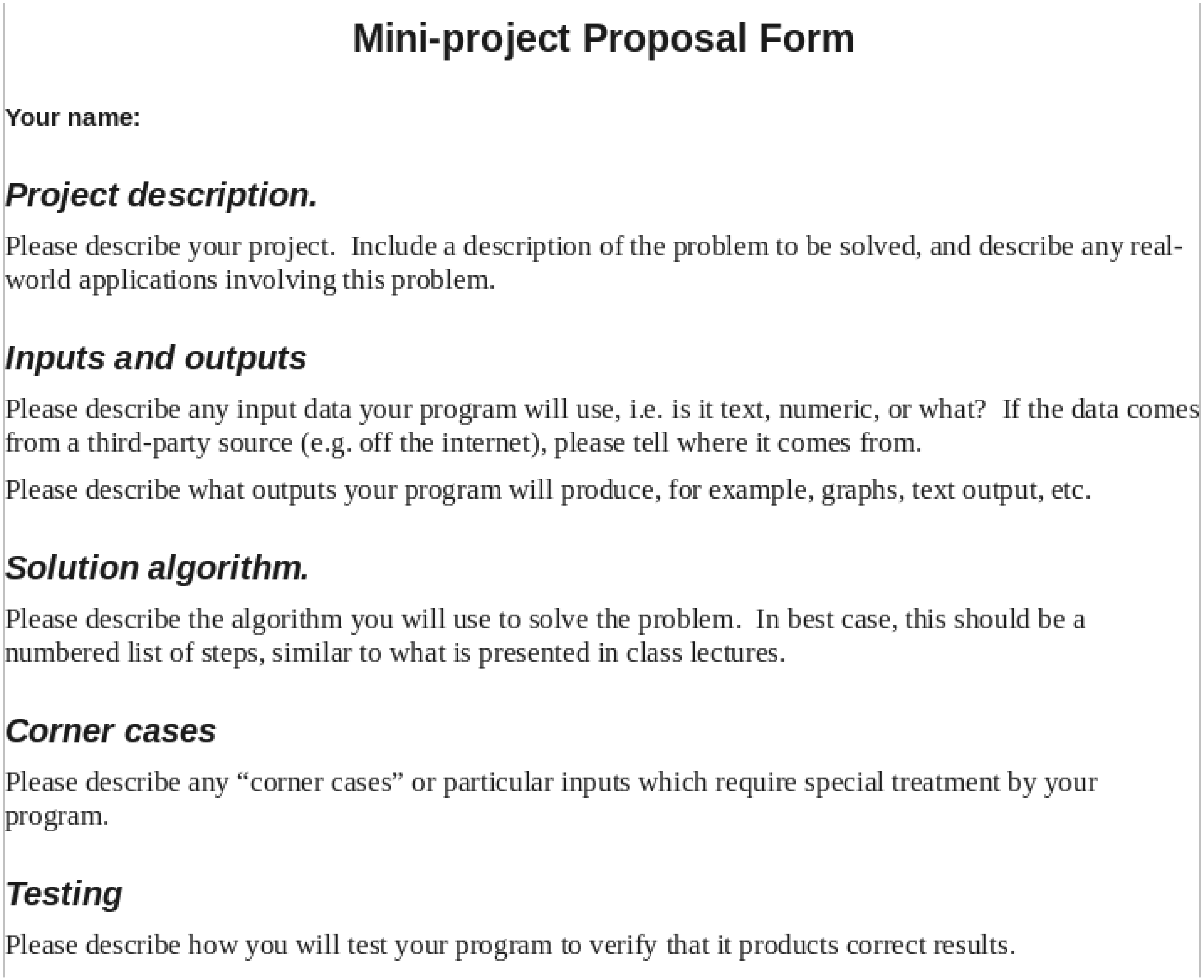


Fig. 1.

TABLE 1.

Mini-Projects

- Some ideas for mini-projects are on Canvas now. Please use them for inspiration.
- Please choose one (or make up your own) and fill out a project proposal form by next week's class.
  - Proposal forms are mandatory, and they are also helpful to you.
- After I approve your mini-project (perhaps after modifications), then do your project and create your presentation.
- Presentations are on Fri, March 3rd.
  - We'll use Matlab to determine order of presentations.



Mini-project Proposal Form

Your name:

Project description.

Please describe your project. Include a description of the problem to be solved, and describe any real-world applications involving this problem.

Inputs and outputs

Please describe any input data your program will use, i.e. is it text, numeric, or what? If the data comes from a third-party source (e.g. off the internet), please tell where it comes from.

Please describe what outputs your program will produce, for example, graphs, text output, etc.

Solution algorithm.

Please describe the algorithm you will use to solve the problem. In best case, this should be a numbered list of steps, similar to what is presented in class lectures.

Corner cases

Please describe any "corner cases" or particular inputs which require special treatment by your program.

Testing

Please describe how you will test your program to verify that it products correct results.

Available on Canvas

A word about your mini-project presentations

- Give presentation from your laptop using PowerPoint or equivalent.
- Rule of thumb: 1 slide every 2 minutes.
- Count on 12 – 14 minutes.
  - 6 or 7 slides max.
  - Allow time for a question.
- Maximum 5 main points (main bullets) per slide.
- Pictures are better than words. Draw a picture to convey your idea if you can.

Mini-projects....

- Please rehearse your presentation before you give it in class.
- Make sure you know how to project using your laptop.
  - Suggestion: try to project using the math department's projector prior to class.
- Put a copy of your presentation (.pdf format) onto a flash drive in case of emergency.

Ideal presentation

- 3 slides introducing your general topic area, and talking about the specific problem you want to solve.
- 2 slides reviewing your computation and algorithm
- 1 slide presenting your results.
- Maybe one more slide to expand on one of the above. Maybe.....

Next topic: Solvers for dense systems of linear equations

$$\begin{aligned} a_{11}x_1 + \ & a_{12}x_2 + \cdots + \ & a_{1n}x_n &= b_1 \\ a_{21}x_1 + \ & a_{22}x_2 + \cdots + \ & a_{2n}x_n &= b_2 \\ \vdots \quad & \quad \vdots & \vdots \quad & \quad \vdots \\ a_{m1}x_1 + \ & a_{m2}x_2 + \cdots + \ & a_{mn}x_n &= b_m . \end{aligned}$$

$$A x = b$$

Linear solver: Problem statement

Given elements of matrix A, and elements of vector b, find the elements of vector x.

$$\begin{aligned}
a_{11}x_1 + \; a_{12}x_2 + \cdots + \; a_{1n}x_n &= b_1 \\
a_{21}x_1 + \; a_{22}x_2 + \cdots + \; a_{2n}x_n &= b_2 \\
\vdots \qquad\quad \vdots \qquad\qquad\quad \vdots \quad &\quad \vdots \\
a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n &= b_m .
\end{aligned}$$

$$\boxed{A\,x = b}$$

Gauss elimination

- Algorithm to solve linear system A x = b
- Named after Gauss (early 19th c. German mathematician)
- Known to Chinese mathematicians as early as 179 AD.
- Example on white board

Johann Carl Friedrich Gauss, 1777-1855

Step 1: Forward elimination

Start with

$$\begin{aligned} -6x+4y+4z&=2\\ 3x+2y-6z&=4\\ 3x-y+z&=1 \end{aligned}$$

$$\begin{array}{l} \\ r_2+\frac{1}{2}r_1 \\ r_3+\frac{1}{2}r_1 \end{array} \left(\begin{array}{ccc|c} -6 & 4 & 4 & 2\\ 3 & 2 & -6 & 4\\ 3 & -1 & 1 & 1 \end{array}\right)$$

Pivot elements in green

$$\begin{array}{l} \\ \\ r_3-\frac{1}{4}r_2 \end{array} \left(\begin{array}{ccc|c} -6 & 4 & 4 & 2\\ 0 & 4 & -4 & 5\\ 0 & 1 & 3 & 2 \end{array}\right)$$

$$\left(\begin{array}{ccc|c} -6 & 4 & 4 & 2\\ 0 & 4 & -4 & 5\\ 0 & 0 & 4 & 3/4 \end{array}\right)$$

After elimination

$$\begin{aligned} -6x+4y+4z&=2\\ 4y-4z&=5\\ 4z&=3/4 \end{aligned}$$

- Use forward elimination to get triangular system

Step 2: Backsubstitution

$$\begin{tikzcd}
\boxed{\begin{array}{r} -6x+4y+4z=2 \\ 4y-4z=5 \\ 4z=3/4 \end{array}} \arrow[r] & \boxed{z=3/16} \arrow[d, "4y=5-4\times(3/16)"] \\
& \boxed{y=23/16} \arrow[d, "-6x=2-4\times(3/16)-4\times(23/16)"] \\
& \boxed{x=3/4}
\end{tikzcd}$$

- Use backsubstitution on triangular system to get x, y, z values.

Forward elimination

- Start with system:

$$\begin{aligned} -6x+4y+4z&=2 \\ 3x+2y-6z&=4 \\ 3x-y+z&=1 \end{aligned}$$

- Create augmented matrix

$$\left(\begin{array}{ccc|c} -6 & 4 & 4 & 2 \\ 3 & 2 & -6 & 4 \\ 3 & -1 & 1 & 1 \end{array}\right)$$

- Identify pivot element

$$\left(\begin{array}{ccc|c} \colorbox{yellow}{-6} & 4 & 4 & 2 \\ 3 & 2 & -6 & 4 \\ 3 & -1 & 1 & 1 \end{array}\right)$$

Forward elimination

- Multiply rows by constants then subtract to clear first column.

$$\left(\begin{array}{ccc|c} -6 & 4 & 4 & 2 \\ 3 & 2 & -6 & 4 \\ 3 & -1 & 1 & 1 \end{array}\right)$$

$$r_2 + \frac{1}{2} r_1$$

$$r_3 + \frac{1}{2} r_1$$

$$\left(\begin{array}{ccc|c} -6 & 4 & 4 & 2 \\ 0 & 4 & -4 & 5 \\ 0 & 1 & 3 & 2 \end{array}\right)$$

Forward elimination

- Identify next pivot element

$$\left(\begin{array}{ccc|c} -6 & 4 & 4 & 2 \\ 0 & 4 & -4 & 5 \\ 0 & 1 & 3 & 2 \end{array}\right)$$

- Multiply rows by constants then subtract to clear next column.

$$\left(\begin{array}{ccc|c} -6 & 4 & 4 & 2 \\ 0 & 4 & -4 & 5 \\ 0 & 1 & 3 & 2 \end{array}\right) \xrightarrow{r_3 - \frac{1}{4} r_2} \left(\begin{array}{ccc|c} -6 & 4 & 4 & 2 \\ 0 & 4 & -4 & 5 \\ 0 & 0 & 4 & 3/4 \end{array}\right)$$

Result of forward elimination

- We now have upper triangular LHS

$$\left(\begin{array}{ccc|c} -6 & 4 & 4 & 2 \\ 0 & 4 & -4 & 5 \\ 0 & 0 & 4 & 3/4 \end{array}\right)$$

- Equivalent linear system

$$\begin{aligned} -6x+4y+4z&=2 \\ 4y-4z&=5 \\ 4z&=3/4 \end{aligned}$$

- All operations performed in forward elimination preserve the solution, so this system has the same solution as the original

Back substitution

- Equivalent linear system

$$\begin{aligned} -6x+4y+4z&=2 \\ 4y-4z&=5 \\ 4z&=3/4 \end{aligned}$$

- Backsubstitute bottom row to get z

$$4z=3/4 \Rightarrow z=3/16$$

- Backsubstitute: Work upwards to get remaining rows

$$4y=5-4\times(3/16)$$
$$\Rightarrow y=23/16$$

$$-6x=2-4\times(3/16)-4\times(23/16)$$
$$x=3/4$$

Remarks about Gauss elimination

- Two stages to Gauss elimination:
  - Forward elimination
  - Back substitution
- Useful for dense matrices.
  - Use other solvers for sparse.
- *Important*: Gauss elimination is $O(N^3)$ overall.
  - Forward elimination $O(N^3)$
  - Back substitution $O(N^2)$
  - Demonstration on whiteboard.

Gaussian elimination code

```
function x = naive_gauss(A,b);
  % Get length of input vector b.
  n = length(b);

  % preallocate output x vector.
  x = zeros(n,1);

  % Perform forward elimination to create triangular matrix.
  fprintf('-------  Forward elimination  -------\n')
  for k=1:n-1     % Iterate over pivots
    for i=k+1:n   % Iterate over rows
      xmult = A(i,k)/A(k,k);  % Divide by pivot
      for j=1:n     % Third inner loop over cols -- algorithm is O(N^3)
        A(i,j) = A(i,j)-xmult*A(k,j);
      end
      b(i) = b(i)-xmult*b(k);
    end
  end

  % Now do back substitution to get x.
  fprintf('-------  Backward substitution  -------\n')
  x(n) = b(n)/A(n,n);
  for i=n-1:-1:1
    sum = b(i);
    for j=i+1:n
      sum = sum-A(i,j)*x(j);
    end
    x(i) = sum/A(i,i);
  end
end
```

Code is on Blackboard in Class 5 folder.

Elimination step: Three nested loops

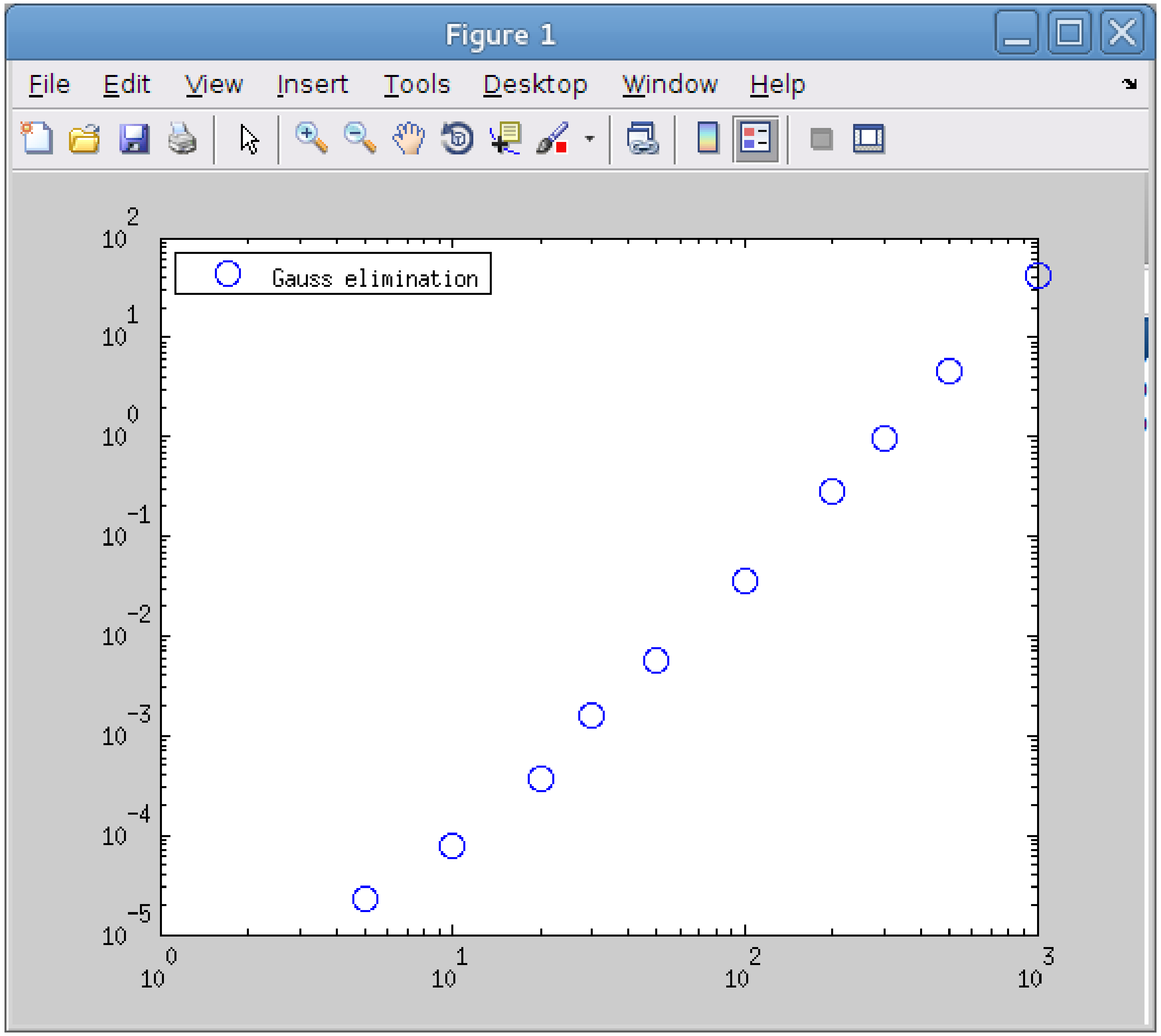
Back substitution step: Two nested loops.

Demo

~/Class4/NaiveGauss

```
>> naive_gauss(A, b)
-------  Forward elimination  -------
After iteration 1,
A =
    0.1952   -0.6336    0.0297   -1.9038
         0   -0.1459   -1.0608   -3.4057
         0   -0.0281    0.0671   -0.3557
         0    6.2837    0.0578   17.7575

b =
    0.6970
    2.0223
    0.0976
   -6.5570

After iteration 2,
A =
    0.1952   -0.6336    0.0297   -1.9038
         0   -0.1459   -1.0608   -3.4057
         0         0    0.2717    0.3013
         0         0  -45.6376 -128.9532

b =
    0.6970
    2.0223
   -0.2925
   80.5599

After iteration 3,
A =
    0.1952   -0.6336    0.0297   -1.9038
         0   -0.1459   -1.0608   -3.4057
         0         0    0.2717    0.3013
         0         0    0.0000  -78.3423

b =
    0.6970
    2.0223
   -0.2925
   31.4191
```

Timing of naive_gauss



```
>> time_naive_gauss
Testing [3, 3] matrix
Testing [5, 5] matrix
Testing [10, 10] matrix
Testing [20, 20] matrix
Testing [30, 30] matrix
Testing [50, 50] matrix
Testing [100, 100] matrix
Testing [200, 200] matrix
Testing [300, 300] matrix
Testing [500, 500] matrix
Testing [1000, 1000] matrix
Gauss elimination is O(2.990)
```

- Gauss elimination: $O(N^3)$

Pivoting

- Consider trying to solve this system using Gaussian elimination:

$$\begin{pmatrix} 0 & 3 & 4 \\ 2 & -6 & 1 \\ -1 & 7 & -3 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}=\begin{pmatrix} -1 \\ 3 \\ 2 \end{pmatrix}$$

- System is non-singular, well conditioned.
- But we can't do Gaussian elimination due to 0 in upper left position.
- However, we ***can*** solve this system:

$$\begin{pmatrix} 2 & -6 & 1 \\ 0 & 3 & 4 \\ -1 & 7 & -3 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}=\begin{pmatrix} 3 \\ -1 \\ 2 \end{pmatrix}$$

Same as above, except with rows 1, 2 swapped.

Pivoting and stability

- Consider simple system:

$$\begin{pmatrix} \text{1e-20} & 1 \\ 1 & 1 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$$

- By observation, solution is close to

$$\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

- But what is the result of Gaussian elimination on this system? (Matlab demo: pivot_demo)

Pivoting

- Pivoting: Means putting largest possible value into “pivot element” before doing elimination.
  - Partial pivoting: Row permutations only
  - Complete pivoting: Both row and column permutations
- Pivoting is important for two reasons:
  - Removing zeros
  - Numerical stability
- If you can't eliminate zeros using pivoting, your matrix is probably singular.

Next: LU decomposition

- Derived from Gauss elimination
- Recall LU decomposition

L = non-zeros on lower triangle

U = non-zeros on upper triangle

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}.$$

- Why is it good? Decompose matrix once, use for many different b vectors

$$Ax = b$$

Hard to solve

$$LUx = b \quad \Rightarrow \quad Ly = b$$

Easy to solve

$$Ux = y$$

Easy to solve

LU decomposition walk-through

- Put 1 on diagonal of L.
- Put neg of Gauss elimin coeffs into off-diags of L.

$$\begin{array}{ccc} & U & L \\ \begin{array}{r} \\ r_2+\frac{1}{2}r_1 \\ r_3+\frac{1}{2}r_1 \end{array} & \begin{pmatrix} -6 & 4 & 4 \\ 3 & 2 & -6 \\ 3 & -1 & 1 \end{pmatrix} & \begin{pmatrix} 1 & 0 & 0 \\ -1/2 & & \\ -1/2 & & \end{pmatrix} \\ \begin{array}{r} \\ \\ r_3-\frac{1}{4}r_2 \end{array} & \begin{pmatrix} -6 & 4 & 4 \\ 0 & 4 & -4 \\ 0 & 1 & 3 \end{pmatrix} & \begin{pmatrix} 1 & 0 & 0 \\ -1/2 & 1 & \\ -1/2 & 1/4 & \end{pmatrix} \\ & \begin{pmatrix} -6 & 4 & 4 \\ 0 & 4 & -4 \\ 0 & 0 & 4 \end{pmatrix} & \begin{pmatrix} 1 & 0 & 0 \\ -1/2 & 1 & 0 \\ -1/2 & 1/4 & 1 \end{pmatrix} \end{array}$$

Check LU decomposition

$$LU = \begin{pmatrix} 1 & 0 & 0 \\ -1/2 & 1 & 0 \\ -1/2 & 1/4 & 1 \end{pmatrix} \begin{pmatrix} -6 & 4 & 4 \\ 0 & 4 & -4 \\ 0 & 0 & 4 \end{pmatrix}$$

$$= \begin{pmatrix} -6 & 4 & 4 \\ 3 & 2 & -6 \\ 3 & -1 & 1 \end{pmatrix}$$

We get original matrix back.

LU – the main points

- We want to solve Ax = b
- Forward iteration to get A = LU. $O(N^3)$
- Now we have LUx = b. Use back substitution to get x = U\(L\b). $O(N^2)$
- This is useful for problems where A remains the same, but b changes.
- Generally used for dense matrices.
- Matlab command: [L, U] = lu(A)
- Default Matlab solver for Ax=b when A is square.

LU demo

```
>> A

A =

    -6     4     4
     3     2    -6
     3    -1     1

>> [L, U] = lu(A)

L =

    1.0000         0         0
   -0.5000    1.0000         0
   -0.5000    0.2500    1.0000


U =

    -6     4     4
     0     4    -4
     0     0     4
```

$$A=\begin{pmatrix}-6 & 4 & 4\\ 3 & 2 & -6\\ 3 & -1 & 1\end{pmatrix}$$

$$L=\begin{pmatrix}1 & 0 & 0\\ -1/2 & 1 & 0\\ -1/2 & 1/4 & 1\end{pmatrix}$$

$$U=\begin{pmatrix}-6 & 4 & 4\\ 0 & 4 & -4\\ 0 & 0 & 4\end{pmatrix}$$

Another demo

- Note: to get LU form in general, you need to pay attention to the permutation matrix returned along with decomposition.

What happened here?!?

```
>> A = randn(4,4)

A =

   -0.1953   -1.2125    0.5279    1.0228
    0.8734    0.4541    0.6959    0.5313
   -0.4659    1.0691   -1.1500   -1.5655
   -0.8709    0.2928   -0.6465   -0.9258

>> [L,U] = lu(A)

L =

   -0.2236   -0.8472    0.0483    1.0000
    1.0000         0         0         0
   -0.5335    1.0000         0         0
   -0.9971    0.5686    1.0000         0


U =

    0.8734    0.4541    0.6959    0.5313
         0    1.3114   -0.7788   -1.2821
         0         0    0.4902    0.3329
         0         0         0    0.0393
```

How to pivot? Permutation matrices

- Upon multiplication a permutation matrix will swap rows/cols. Examples:

$$P=\begin{pmatrix}1&0&0&0\\0&0&1&0\\0&1&0&0\\0&0&0&1\end{pmatrix}$$

$$PA=\begin{pmatrix}1&0&0&0\\0&0&1&0\\0&1&0&0\\0&0&0&1\end{pmatrix}\begin{pmatrix}1&2&3&4\\5&6&7&8\\9&10&11&12\\13&14&15&16\end{pmatrix}=\begin{pmatrix}1&2&3&4\\9&10&11&12\\5&6&7&8\\13&14&15&16\end{pmatrix}$$

- PA – swap rows
- AP – swap columns.

Make a permutation matrix in Matlab

```
>> P = eye(5)

P =

     1     0     0     0     0
     0     1     0     0     0
     0     0     1     0     0
     0     0     0     1     0
     0     0     0     0     1

>> P = P(:, [1 4 3 2 5])

P =

     1     0     0     0     0
     0     0     0     1     0
     0     0     1     0     0
     0     1     0     0     0
     0     0     0     0     1
```

Row swap

```
>> A

A =

     1     1     1     1     1
     1     2     3     4     5
     1     3     6    10    15
     1     4    10    20    35
     1     5    15    35    70

>> P*A

ans =

     1     1     1     1     1
     1     4    10    20    35
     1     3     6    10    15
     1     2     3     4     5
     1     5    15    35    70
```

Column swap

```
>> A

A =

     1     1     1     1     1
     1     2     3     4     5
     1     3     6    10    15
     1     4    10    20    35
     1     5    15    35    70

>> A*P

ans =

     1     1     1     1     1
     1     4     3     2     5
     1    10     6     3    15
     1    20    10     4    35
     1    35    15     5    70
```

Permutation matrices

- When we do Gaussian elimination, we do pivoting by hand.
- When Matlab does pivoting, it can return a permutation matrix along with the matrix decomposition.

```
>> A = randn(4,4)

A =

   -0.1953   -1.2125    0.5279    1.0228
    0.8734    0.4541    0.6959    0.5313
   -0.4659    1.0691   -1.1500   -1.5655
   -0.8709    0.2928   -0.6465   -0.9258

>> [L,U] = lu(A)

L =

   -0.2236   -0.8472    0.0483    1.0000
    1.0000         0         0         0
   -0.5335    1.0000         0         0
   -0.9971    0.5686    1.0000         0


U =

    0.8734    0.4541    0.6959    0.5313
         0    1.3114   -0.7788   -1.2821
         0         0    0.4902    0.3329
         0         0         0    0.0393
```

```
>> A

A =

   -0.1953   -1.2125    0.5279    1.0228
    0.8734    0.4541    0.6959    0.5313
   -0.4659    1.0691   -1.1500   -1.5655
   -0.8709    0.2928   -0.6465   -0.9258

>> [L,U, P] = lu(A)

L =

    1.0000         0         0         0
   -0.5335    1.0000         0         0
   -0.9971    0.5686    1.0000         0
   -0.2236   -0.8472    0.0483    1.0000


U =

    0.8734    0.4541    0.6959    0.5313
         0    1.3114   -0.7788   -1.2821
         0         0    0.4902    0.3329
         0         0         0    0.0393


P =

     0     1     0     0
     0     0     1     0
     0     0     0     1
     1     0     0     0
```

Ask for
permutation matrix
as part of return

L

U

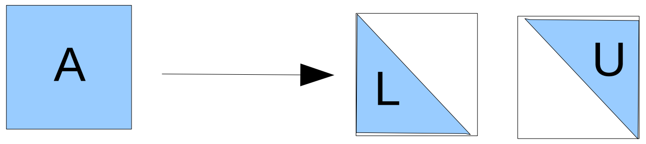
Permutation matrix

$$P*A = L*U$$

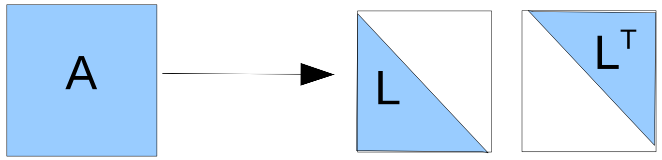
$$A = P^T*L*U$$

Next: Cholesky decomposition

- The story so far:
  - Gauss elim is O(N^3)
  - LU is O(N^3) for decomposition, O(N^2) to re-use the decomposition for different b vectors.



- For symmetric, positive-definite matrices: Cholesky decomposition
  - Still O(N^3), but 1/2 time required by LU



$$A = L L^T$$

Derivation

Note symmetric

$$A = L L^T$$

Cholesky decomposition – SPD matrix only

$$\begin{pmatrix} a_{11} & a_{21} & a_{31} \\ a_{21} & a_{22} & a_{32} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} = \begin{pmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{pmatrix} \begin{pmatrix} l_{11} & l_{21} & l_{31} \\ 0 & l_{22} & l_{32} \\ 0 & 0 & l_{33} \end{pmatrix}$$

$$= \begin{pmatrix} l_{11}^2 & same & same \\ l_{21} l_{11} & l_{21}^2 + l_{22}^2 & same \\ l_{31} l_{11} & l_{31} l_{21} + l_{32} l_{22} & l_{31}^2 + l_{32}^2 + l_{33}^2 \end{pmatrix}$$

- Walk through on white board

Working backwards....

$$\begin{pmatrix} a_{11} & a_{21} & a_{31} \\ a_{21} & a_{22} & a_{32} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} = \begin{pmatrix} l_{11}^2 & same & same \\ l_{21} l_{11} & l_{21}^2 + l_{22}^2 & same \\ l_{31} l_{11} & l_{32} l_{21} + l_{32} l_{22} & l_{31}^2 + l_{32}^2 + l_{33}^2 \end{pmatrix}$$

$$\begin{tikzcd}
l_{11}=\sqrt{a_{11}} \arrow[d] & & \\
l_{21}=a_{21}/l_{11} \arrow[r] & l_{22}=\sqrt{a_{22}-l_{21}^2} \arrow[dl] & \\
l_{31}=a_{31}/l_{11} \arrow[r] & l_{32}=(a_{32}-l_{31}\,l_{21})/l_{22} \arrow[r] & l_{33}=\sqrt{a_{33}^2-l_{31}^2-l_{32}^2}
\end{tikzcd}$$

- Each element depends only upon previously computed values.

Cholesky algorithm

1. Initialize by computing $\longrightarrow$ $l_{11} = \sqrt{a_{11}}$
2. Loop on i = 2:N
3. Loop on j = 1:i-1 (lower triangle)
4. Compute off diagonals $\longrightarrow$ $l_{ij} = \left(\frac{1}{l_{jj}}\right)\left(a_{ij}^2 - \sum_{k=1}^{j-1} l_{ik} l_{jk}\right)$
5. End j
6. Compute diagonal entry $\longrightarrow$ $l_{ii} = \sqrt{a_{ii}^2 - \sum_{j=1}^{i-1} l_{ij}^2}$
7. End i
8. Return lower triangular matrix $\begin{pmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{pmatrix}$

Simple Matlab implementation

```
function L = mychol(A)
  % My simple implementation of the basic
  % Cholesly factorization algorithm.
  % Written for Numerical Analysis 1, Spring 2016.
  % Note: This version is not vectorized. Performance
  % may stink.

  N = size(A,1);
  L = zeros(size(A));

  fprintf('Computing [%d, %d]\n', 1, 1)
  L(1,1) = sqrt(A(1,1));
  for i=2:N
    % Do sum giving lower off-diagonals.
    for j=1:(i-1)
      s = 0;
      for k=1:(j-1)
        s = s + L(i,k)*L(j,k);
      end
      fprintf('Computing [%d, %d]\n', i, j)
      L(i,j) = (A(i,j) - s)/L(j,j);
    end

    % Do sum giving diagonal piece.
    s = 0;
    for j=1:(i-1)
      s = s + L(i,j)*L(i,j);
    end
    fprintf('Computing [%d, %d]\n', i, i)
    L(i,i) = sqrt(A(i,i) - s);

  end

end
```

```
>> B = 100*randn(4,4)

B =

 -135.5593  -71.7969   -4.3032   52.2341
   45.9567   85.5801  219.6252  -55.9906
   44.2279  155.6688  -75.9326  -63.3317
  -25.6306  -38.9228  134.5036  -46.2980

>> A = B'*B
```

This creates SPD matrix

```
A =

   1.0e+04 *

    2.3101    2.1548    0.3871   -1.1268
    2.1548    3.8226    0.2049   -1.6599
    0.3871    0.2049    7.2111   -1.3940
   -1.1268   -1.6599   -1.3940    1.2018
```

```
>> L = mychol(A)
Computing [1, 1]
Computing [2, 1]
Computing [2, 2]
Computing [3, 1]
Computing [3, 2]
Computing [3, 3]
Computing [4, 1]
Computing [4, 2]
Computing [4, 3]
Computing [4, 4]

L =

  151.9914         0         0         0
  141.7728  134.6364         0         0
   25.4675  -11.5992  267.0723         0
  -74.1381  -45.2172  -47.0899   47.5316

>> L*L' - A

ans =

   1.0e-10 *

         0         0         0         0
         0         0         0         0
         0         0   -0.1455         0
         0         0         0         0
```

Test

```
function test_mychol()

  basetol = 2e-11;
  pass = 0;
  fail = 0;

  % Generate representative sample of different sized matrices for test.
  for n = 3:13:200
    tol = n*basetol;

    % Generate random SPD matrix of size nxn.
    B = 100*randn(n,n);
    A = B'*B;

    % Now send to mychol
    L = mychol(A);

    % Now check decomposition
    diff = norm(L*L' - A);
    fprintf('Testing [%d, %d] matrix...  diff = %e, tol = %e ', n, n, diff, tol)
    if (diff < tol)
      fprintf('Test passed!\n')
      pass = pass+1;
    else
      fprintf('Test failed!\n')
      fail = fail+1;
    end
  end

  if (fail > 0)
    fprintf('At end, at least one test failed -- general failure\n')
  else
    fprintf('All tests pass!\n')
  end

end
```

Testing different matrix sizes

```
>> test_mychol
Testing [3, 3] matrix...  diff = 2.943187e-12, tol = 6.000000e-11 Test passed!
Testing [16, 16] matrix...  diff = 3.382943e-11, tol = 3.200000e-10 Test passed!
Testing [29, 29] matrix...  diff = 8.032349e-11, tol = 5.800000e-10 Test passed!
Testing [42, 42] matrix...  diff = 1.963549e-10, tol = 8.400000e-10 Test passed!
Testing [55, 55] matrix...  diff = 2.601481e-10, tol = 1.100000e-09 Test passed!
Testing [68, 68] matrix...  diff = 3.717185e-10, tol = 1.360000e-09 Test passed!
Testing [81, 81] matrix...  diff = 3.559407e-10, tol = 1.620000e-09 Test passed!
Testing [94, 94] matrix...  diff = 5.008336e-10, tol = 1.880000e-09 Test passed!
Testing [107, 107] matrix...  diff = 5.222256e-10, tol = 2.140000e-09 Test passed!
Testing [120, 120] matrix...  diff = 7.884064e-10, tol = 2.400000e-09 Test passed!
Testing [133, 133] matrix...  diff = 9.766660e-10, tol = 2.660000e-09 Test passed!
Testing [146, 146] matrix...  diff = 1.187644e-09, tol = 2.920000e-09 Test passed!
Testing [159, 159] matrix...  diff = 1.042439e-09, tol = 3.180000e-09 Test passed!
Testing [172, 172] matrix...  diff = 1.229054e-09, tol = 3.440000e-09 Test passed!
Testing [185, 185] matrix...  diff = 1.499971e-09, tol = 3.700000e-09 Test passed!
Testing [198, 198] matrix...  diff = 2.454740e-09, tol = 3.960000e-09 Test passed!
All tests pass!
```

```
>> B = 100*randn(4,4)

B =

  -43.7288 -177.1440  -15.6100  -84.0567
  -58.3422  -68.3055 -131.0864 -140.8051
    5.8492  -85.7203   44.9550  155.4949
 -121.6094   59.1517 -179.1292 -162.2765

>> L = mychol(B)
Computing [1, 1]
Computing [2, 1]
Computing [2, 2]
Computing [3, 1]
Computing [3, 2]
Computing [3, 3]
Computing [4, 1]
Computing [4, 2]
Computing [4, 3]
Computing [4, 4]

L =

   0.0000 + 6.6128i   0.0000 + 0.0000i   0.0000 + 0.0000i   0.0000 + 0.0000i
   0.0000 + 8.8227i   3.0877 + 0.0000i   0.0000 + 0.0000i   0.0000 + 0.0000i
   0.0000 - 0.8845i -30.2894 + 0.0000i   0.0000 +29.5247i   0.0000 + 0.0000i
   0.0000 +18.3901i  71.7046 + 0.0000i   0.0000 -66.9436i   0.0000 +22.0041i

>> L*L' - B

ans =

   1.0e+04 *

    0.0087    0.0235    0.0010    0.0206
    0.0117    0.0156    0.0030    0.0524
   -0.0012   -0.0016    0.1745   -0.4320
    0.0243    0.0324   -0.3986    1.0608
```

What if input matrix is not SPD?

Returned L does not satisfy $B = L\,L^T$

Another decomposition: QR

- Decompose matrix into product of orthogonal and upper triangular matrices.

$$A = Q R$$

- Upper triangular R:

$$R = \begin{pmatrix} r_{11} & r_{12} & r_{13} & \cdots & r_{1\mathrm{N}} \\ 0 & r_{22} & r_{23} & \cdots & r_{2\mathrm{N}} \\ 0 & 0 & r_{33} & \cdots & r_{3\mathrm{N}} \\ & & \vdots & & \\ 0 & 0 & 0 & \cdots & r_{NN} \end{pmatrix}$$

- Orthogonal matrix Q spanning space of A.

We will use the QR decomposition later....

Orthogonal matrix

- Orthogonal matrix: columns are vectors orthogonal to each other.

$$Q = \begin{pmatrix} \vdots & \vdots & \vdots & & \vdots \\ e_1 & e_2 & e_3 & \cdots & e_N \\ \vdots & \vdots & \vdots & & \vdots \end{pmatrix}$$

- Nice property of orthogonal matrices:

$$Q^T Q = I \quad \Leftrightarrow \quad Q^{-1} = Q^T$$

- But how to create an orthogonal matrix?

Gram-Schmidt procedure

- Details in handout on Canvas
- Goal: Given matrix A, create set of orthonormal basis vectors e_i spanning space of A.
- The idea is to take each column of input matrix A, and shave off non-orthogonal components of each column.
- Picture on white board.

Gram-Schmidt procedure

Start with: $A = (a_1 | a_2 | a_3 | \cdots | a_N)$

First vector: $u_1 = a_1$ $\qquad e_1 = \dfrac{u_1}{\|u_1\|}$

$$u_2 = a_2 - (a_2 \cdot e_1) e_1 \qquad e_2 = \frac{u_2}{\|u_2\|}$$

$$u_3 = a_3 - (a_3 \cdot e_2) e_2 - (a_3 \cdot e_1) e_1 \qquad e_3 = \frac{u_3}{\|u_3\|}$$

Gram-Schmidt code

```
function e = gram_schmidt(A)
  % This fcn returns the orthogonalization of the
  % matrix A computed using Gram-Schmidt

  N = size(A, 2);
  e = zeros(size(A));

  % Initialize computation by computing u and e for first col.
  un = A(:, 1);
  e(:, 1) = un/norm(un);

  % Iterate over remaining columns of A
  for k = 2:N
    an = A(:, k);
    un = an;
    % Iterate over previous e values and subtract off
    % component parallel to each ei
    for i = k-1:-1:1
      un = un - dot(an,e(:, i))*e(:, i);
    end
    % Compute next col of e
    e(:, k) = un/norm(un);

  end
end
```

QR algorithm – Gram-Schmidt

$$A = (a_1 | a_2 | a_3 | \cdots | a_N)$$

Start by considering A as collection of column vectors.

$$= (e_1 | e_2 | e_3 | \cdots | e_N) \begin{pmatrix} a_1 \cdot e_1 & a_2 \cdot e_1 & a_2 \cdot e_1 & \cdots & a_N \cdot e_1 \\ 0 & a_2 \cdot e_2 & a_2 \cdot e_2 & \cdots & a_N \cdot e_2 \\ 0 & 0 & a_3 \cdot e_3 & \cdots & a_N \cdot e_3 \\ & & \vdots & & \\ 0 & 0 & 0 & \cdots & a_N \cdot e_N \end{pmatrix}$$

Use Gram-Schmidt to compute orthonormal basis set e_i from a_i vectors. This forms orthogonal matrix Q.

Use e_i and a_i to compute elements of R matrix.

$$= QR$$

Code implementing QR decomposition

```
function [Q, R] = my_qr(A)
  % This fcn implements the QR algorithm using Gram-Schmidt

  e = gram_schmidt(A);

  N = size(A, 1);
  Q = e;
  R = zeros(size(A));
  for r = 1:N
    for c = r:N
      R(r, c) = dot(A(:, c), e(:, r));
    end
  end

end
```

Two steps:

1. Compute the e vectors from A.
2. Compute the R elements from e and A.

Remarks

- You can do a QR decomposition on a non-square matrix.
  - Important in linear regression.
- Gram-Schmidt is numerically unstable. Better methods use:
  - Householder transformation
  - Givens Rotations

Topics covered in this session

$$A x = b$$

- Gauss elimination
  - Works for every type of (nonsingular) matrix
  - $O(N^3)$
- LU decomposition
  - $O(N^3)$ decomposition, $O(N^2)$ re-use.
  - Pivoting and permutation matrices
- Cholesky
  - $O(N^3)$ but faster... for SPD matrices only.
- QR decomposition
  - Can do for non-square