Applied Programming

Vector and Matrix Arithmetic Overview

– Vector-Vector Operations

– Matrix-Vector Operations

– Matrix-Matrix Operations

Vector & Matrix Notation

- **Matrices** are denoted by ***capital letters***
  - e.g., ***A,B,C***
- **Vectors** are denoted by ***lower case letters***
  - e.g., ***v,w,x***
- ***Scalars*** will also be denoted by ***lower case***
  - Will be clear from context
- The superscript T denotes ***transpose*** as in $\boldsymbol{A^T}$
- "Matlab" indexing notation:
  - $A(\mathbf{i},:)$ denotes the ***i*th row** of matrix A
  - $A(:,\mathbf{j})$ denotes the ***j*th column** of matrix A

Preliminaries

- An ***nxm* matrix *A*** is an (*n* rows by *m* columns) array of numbers.

- A ***n-vector*** (n-dimensional) is a ***nx1 matrix***
  - n rows, 1 column
  - **A column vector**

$$\begin{matrix} \mathbf{1} \\ \mathbf{2} \\ \mathbf{3} \end{matrix}$$

- Warning: not the same as *1*x*n* matrix !!
  - Row vector

$$1\ 2\ 3$$

Preliminaries

- Addition & subtraction between matrices or vectors are defined element-wise
- ***Multiplication*** between matrices or vectors is ***not element-wise.***

Warning: Multiplication of matrices is not commutative: $AB \neq BA$

Arithmetic Vector Operations

- Vector-scalar multiplication:

$$\boldsymbol{z} = \alpha \boldsymbol{v} = \begin{bmatrix} \alpha v_1 \\ \alpha v_2 \\ \alpha v_3 \end{bmatrix}$$

- Vector addition (3x3 example) :

$$\boldsymbol{z} = \boldsymbol{v} + \boldsymbol{w} = \begin{bmatrix} v_1 + w_1 \\ v_2 + w_2 \\ v_3 + w_3 \end{bmatrix}$$

Element-wise

Special Vector Operations

- Element-wise (Hadamard) Product of two Vectors:

$$\boldsymbol{w} .* \boldsymbol{v} = \begin{bmatrix} w_1 v_1 \\ w_2 v_2 \\ w_3 v_3 \end{bmatrix}$$

Special Vector Operations

- **Inner Product**
  - Scalar Product
  - Dot Product

  of two vectors

$$\boldsymbol{v} \cdot \boldsymbol{w} = \langle \boldsymbol{v}, \boldsymbol{w} \rangle = \boldsymbol{v}^T \boldsymbol{w} = \sum_{i=1}^{n} v_i w_i$$

- Produces a scalar

Inner (dot) Product

- Consider the following vectors.

$$\boldsymbol{a} = \begin{bmatrix} 3 \\ 12 \\ -4 \end{bmatrix}, \quad \boldsymbol{b} = \begin{bmatrix} -2 \\ 3 \\ 4 \end{bmatrix}$$

- Their inner product is

$$a^T = [3 \quad 12 \quad -4]$$

$$\begin{aligned} \langle \boldsymbol{a}, \boldsymbol{b} \rangle &= \boldsymbol{a}^T \boldsymbol{b} = \sum_{i=1}^{3} a_i \, b_i \\ &= 3 \times (-2) + 12 \times 3 + (-4) \times 4 \\ &= 14 \end{aligned}$$

- Produces a SCALAR

Inner (dot) Product: C Code

```
int      k;
double DotProduct = 0.0;
...
/* Inner prod between A(row,:) and B(:,col)*/
/* row and col select ONE pair of vectors */
for (k=0; k < n; k++)                /* k running index */
{
  DotProduct += A[row][k]*B[k][col];
} /* for() */
```

Inner (dot) products involve Multiply ACcumulate (MAC) operations that return a scalar

Inner (dot) Product: Summary

- The inner product operation is composed of a ***sequence of multiplication and addition*** operations, (a.k.a., multiply-accumulate [**MAC**]).
  - MAC is a common operation that most general-purpose *microprocessors* are optimized for.
  - All digital signal processors (DSP) include a *MAC assembly language instruction*.

Important:

- The "*complexity*" of the **inner product** between two *n*-vectors is ***O(n)***

Inner (dot) Product Code

- Simple loop code works, but it can be slow.

- We can improve the performance of inner product computations by unrolling the loop
  - creating independent operations within the loop
- Recode using special MAC assembly instructions
  - **optimized in hardware, DSP processors**

Inner Product Code Unrolled

```
double DotProduct[4] = {0.0, 0.0, 0.0, 0.0};
...
...
for (k=0; k < n; k+=4){
  DotProduct[0] +=     a[row][k+0]*b[k+0][col];
  DotProduct[1] +=     a[row][k+1]*b[k+1][col];
  DotProduct[2] +=     a[row][k+2]*b[k+2][col];
  DotProduct[3] +=     a[row][k+3]*b[k+3][col];
} /* for() */
DotProduct[0] += (   DotProduct[1]
                   + DotProduct[2]
                   + DotProduct[3] ) ;
```

Register Variables

- Register keyword
  - A hint to the compiler that a variable will be used **a lot**
  - Compilers ALWAYS allocate variables to registers
- Most of the time we *will not request register variables*.
  - The compiler generally can make the decision effectively.

```
register double DotProduct = 0.0;
for (k=0; k < n; k++)
  {
   DotProduct += a[row][k]*b[k][col];
  } /* for() */
```

The Saxpy Operation

- A **saxpy** is a **vector "MAC"** , the result is a *vector* (not a scalar)

$$y = a\,x + y$$

- The name comes from "scalar alpha x plus y", where ***x*** and ***y*** are vectors and α a scalar

```
/* Saxpy sample C code */
for (k=0; k < n; k++){
   y[k] = a*x[k] + y[k];
} /* for() */
```

Note: y[k] += a*x[k] is slightly faster

- A **saxpy** is also ***O(n)***

Operations between Matrices

- **Addition and subtraction** of matrices is performed "element-wise". It is ***$O(n^2)$***

- Example (3x3 matrices): A + B = C

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{02} \end{bmatrix} + \begin{bmatrix} b_{00} & b_{01} & b_{02} \\ b_{10} & b_{11} & b_{12} \\ b_{20} & b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} (a_{00}+b_{00}) & (a_{01}+b_{01}) & (a_{02}+b_{02}) \\ (a_{10}+b_{10}) & (a_{11}+b_{11}) & (a_{12}+b_{12}) \\ (a_{20}+b_{20}) & (a_{21}+b_{21}) & (a_{22}+b_{22}) \end{bmatrix}$$

Notes:

- Standard Notation: $A(i,j) = a_{ij}$
- C Notation: $A(i,j) = A[i][j]$

Efficient Matrix Add/Subt

- **C arrays** are stored in **row major form**
  - elements of rows are stored contiguously in memory.
  - The Fortran default is column major.

- For ***efficient computations*** in C the ***row index should be in the outer loop.***

```
double a[m][n], b[m][n];
...
for (row=0; row<m; row++){ /* row in outermost loop*/
  for (col=0; col<n;col++)}/* inner loop */
      c[row][col]=a[row][col]+b[row][col];
  } }
```

Matrix-Scalar Multiplication

- Multiplying a matrix by a scalar (e.g., a number) is an "element-wise" operation as well. It is ***O(n²)***

  Example: **B=** *s* **A** (*s* a scalar)

- Sample C code for scalar multiplication:

  (note col index in inner loop for efficiency)

```
int row, col;
double s;
...
for (row=0;row<m;row++){ /* row is outermost loop */
  for (col=0; col<n; col++)
    {
    B[row][col]=s*A[row][col];
    }}
```

Inner Product Matrix Multiplication

- ***Matrix-matrix multiplication*** can be formulated as a sequence of inner products between the rows of the first matrix and the columns of the second matrix
  - The row dimension of the first must match the column dimension of the second
  - E.g (2x**3**) X (**3**x4)

$$AB = \begin{bmatrix} \boxed{a_{00} \quad a_{01} \quad a_{02}} \\ a_{10} \quad a_{11} \quad a_{12} \\ a_{20} \quad a_{21} \quad a_{22} \end{bmatrix} \begin{bmatrix} \boxed{\begin{matrix} b_{00} \\ b_{10} \\ b_{20} \end{matrix}} & \begin{matrix} b_{01} \\ b_{11} \\ b_{21} \end{matrix} & \begin{matrix} b_{02} \\ b_{12} \\ b_{22} \end{matrix} \end{bmatrix}$$

Matrix Multiplication & Inner Products..

$$
\begin{aligned}
AB &= \begin{bmatrix} a_0^T \\ a_1^T \\ a_2^T \end{bmatrix} \begin{bmatrix} b_0 & b_1 & b_2 \end{bmatrix} \\
&= \begin{bmatrix} a_0^T b_0 & a_0^T b_1 & a_0^T b_2 \\ a_1^T b_0 & a_1^T b_1 & a_1^T b_2 \\ a_2^T b_0 & a_2^T b_1 & a_2^T b_2 \end{bmatrix}
\end{aligned}
$$

(3x1) X (1x3)

(3x3)

- ***Each entry*** in the result requires ***one inner product*** of a row and a column, (is *O(n)*)
- For an ***nxn matrix*** we need ***n^2 inner products***
- Therefore, ***matrix multiplication* is *O(n^3)***

Matrix Multiplication

$$A = \begin{bmatrix} 5 & -2 & 6 \\ 0 & 7 & 4 \end{bmatrix} \qquad B = \begin{bmatrix} 8 & 5 & 1 & -3 \\ -4 & 6 & -2 & 7 \\ 9 & -1 & 2 & 3 \end{bmatrix}$$

(2x**3**) (**3**x4)

C=AB

Row 1 x col 1: $5{\times}8 + -2{\times}-4 + 6{\times}9 = 102$

Row 1 x col 2: $5{\times}5 + -2{\times}6 + 6{\times}-1 = 7$

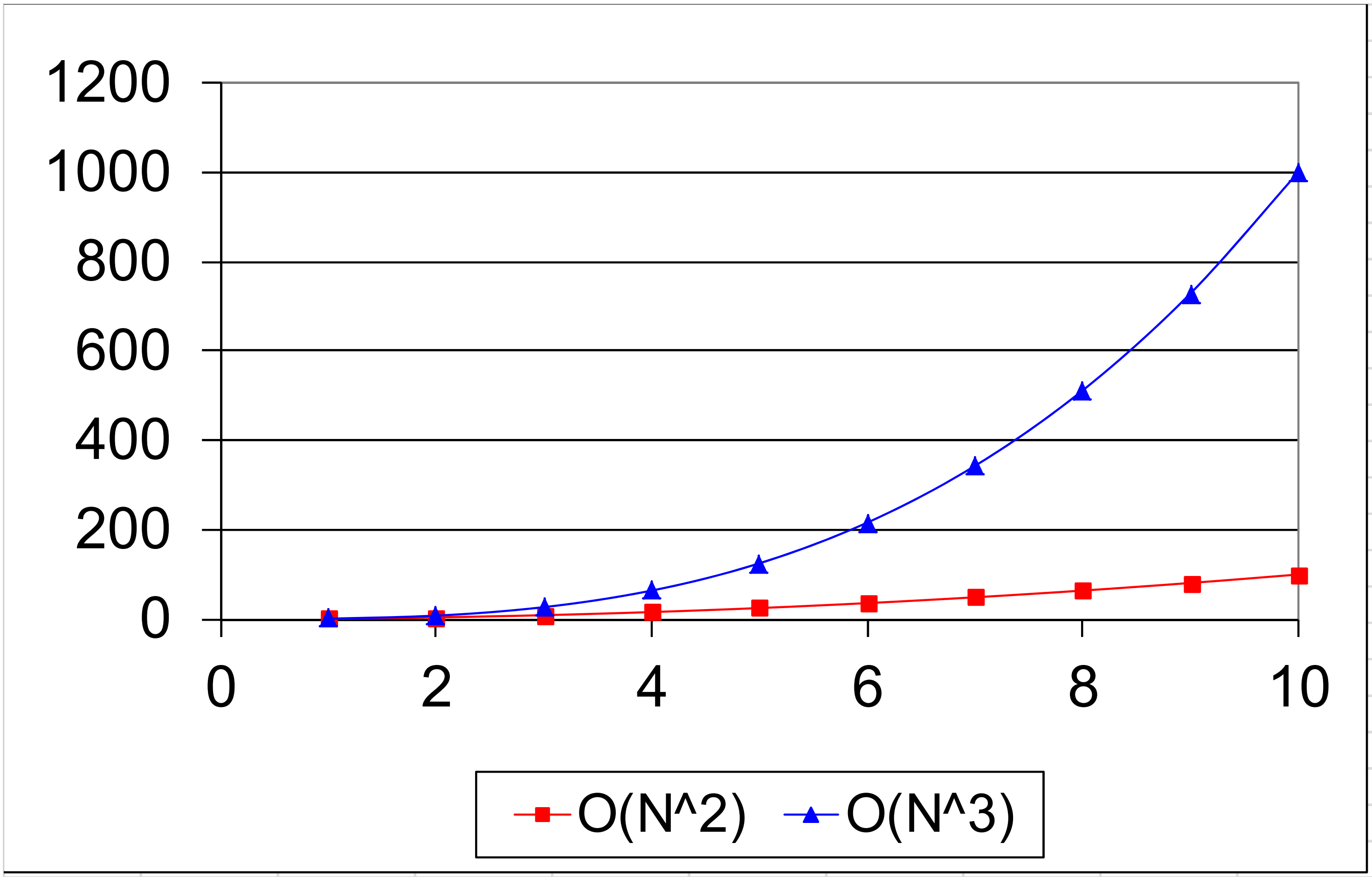
Row 1 x col 3: $5{\times}1 + -2{\times}-2 + 6{\times}2 = 21$

…..

$$C = \begin{bmatrix} 102 & 7 & 21 & -11 \\ 8 & 38 & -6 & 61 \end{bmatrix}$$

(2x4)

Complexity: $O(n^2)$ vs $O(n^3)$



Computing with "dense matrices" is expensive

Summary

- Computing inner products efficiently is fundamental to matrix computations and signal processing

- Most CPUs and all DSPs have ***hardware optimized*** multiply-accumulate (MAC) operations.

- Efficient inner product implementation is a key component of good algorithms to solve simultaneous equations.

Linear Algebra Algorithms

- It is common to describe linear algebra algorithms in ***vectorized notation:***
  G. Golub and C.F. Van Loan, Matrix Computations (Johns Hopkins Press)

- This notation is now called *"Matlab notation"*
  - it was introduced before Matlab even existed

Matlab and Octave

- *MATLAB*® is a high-level commercial language and interactive environment for numerical computation, visualization, and programming
  - **Origin ONE BASED**

- ***GNU Octave*** is a free tool that was designed to be **"compatible" with Matlab.**
  - Sometimes we will prototype numerical algorithms in *MATLAB* (especially if matrix computations are involved)

In this very short introduction you can safely replace Matlab by Octave

Matlab Settings

- Octave – built in to the CE systems
  - I use this most of the time

- If you REQURE real Matlab

  module load Matlab

  matlab

Minimal Matlab - Vectors

- Create a vector “a” & “b”
  - Commas between numbers are optional

- >> a = [3 12 -4]

  a = 3 12 -4

- >> b = [-2, 3, 4]

  b = -2 3 4

Minimal Matlab - Transpose

- Transpose a vector (or matrix)
- >> a = a'

```
a =   3
     12
     -4
```

- Vector dot product
- >> dot(a,b)

```
ans =  14
```

Minimal Matlab - Matrix

- Define a Matrix
  - The semicolon defines the row

- >A=[5 -2 6; 0 7 4]

  = 5 -2 6

  0 7 4

- > **B = [8 5 1 -3; -4 6 -2 7; 9 -1 2 3]**

  = 8 5 1 -3

  -4 6 -2 7

  9 -1 2 3

Minimal Matlab - Mult

- Matrix Multiplication
- >> C=A*B

```
  =   102     7    21   -11
        8    38    -6    61
```

Minimal Matlab - Solve

Given:

$$2x_1 + 8x_2 + 6x_3 = 20$$
$$4x_1 + 2x_2 - 2x_3 = -2$$
$$3x_1 - x_2 + x_3 = 11$$

Solve for x

```
>> A = [2 8 6; 4 2 -2; 3 -1 1];
    A =   2   8   6
          4   2  -2
          3  -1   1
```

```
>> b = [20 -2 11]'
    b =  20
         -2
         11
```

```
>> A\b
    ans =   2
           -1
            4
```

So:

$x_1 = 2$, $x_2 = -1$, $x_3 = 4$

Last Detail: Scripts and Functions

A **Script** is a sequence of Matlab commands organized in a file

Example: **`myscript.m`**

```
%% The variable a must
% already exist in the
% “workspace”
% matlab script
if a>0,
  b=sqrt(pi/3);
else
  b=sqrt(-a);
end
```

A Matlab **function** is usually stored in a file with the same name as the function

Example: **`ex_func.m`**

```
function b=ex_func(a)
%% this text is displayed
% if you type help ex_func
%
if a>0,
  b=sqrt(pi/3);
else
  b=sqrt(-a);
end
```

Tip: When prototyping in Matlab/Octave start with a script; you can make it a function after debugging it.

Vectors and Matrices in Matlab/Octave

- In Matlab the **main data type** is a ***matrix of doubles*** (IEEE double precision)

- Matrices, Rows and Column Vectors

```
>>%Matlab is case sensitive R is different from r
>>% row vectors
>>r1 = [1,2,3]; % commas are not necessary
>>r2 = [4 5 6]; % semicolon at end suppresses output

>>% column vectors
>>c1 = [1;2;3];  % here semicolon separates rows
>>c2 = [4 5 6]'; % can transpose operator

>>% matrices
>>R = [r1;r2]; % constructed by rows
>>C = [c1 c2]; % constructed by columns
```

Vectors and Matrices in Matlab/Octave

```
>>% Various special matrices
>>E = [];          % and empty matrix
>>I = eye(3);      % 3x3 Identity Matrix
>>M0 = zeros(4,2);% 4x2 matrix of zeros
>>M1 = ones(7,4); % 7x4 matrix of ones
>>RM = rand(14,16); % 14x16 random matrix

>>% The colon operator
>>idx1=1:10;     % row vector with entries 1,2,…,10
>>idx2=0:2:10;   % row vector with entries 0,2,…,10
>>idx3=0:-4:-10; % row vector with entries 0,-4,-8

>>% Extracting submatrices with the colon operator
>>rm1 = RM(:,2) ;  % extracts 2nd column of RM
>>rm1 = RM(:,2:3); % extracts 2nd and 3rd columns of RM
>>rm2 = RM(3,:) ;  % extracts 3rd row of RM
>>RM3 = RM(2:4,4:end);  % here end is # of cols
>>RM4 = RM(2:end,4:10); % here end is # of rows
```

Vectors and Matrices in Matlab/Octave

```
>>% In this course "vectors" are column vectors
>>% Inner and Outer products
>>v=[1;2;3];   % vector v
>>w=[5;4;3];   % vector w
>>x=[5;4;3;1]; % vector x
>>A=v'*w; % inner product – row * column = scalar
>>B=v*x'; % outer product – column * row = matrix
>>[nr,nc]=size(A);% this is 1x1, a scalar
>>[nr,nc]=size(B);% this is 3x4, a matrix

>>% Warning !! Matrix indexing starts at 1
>>A(2,3); % in C you would write A[1][2]
>>A+B;    % will give an error
>>v+w;    % works
>>v*w;    % produces an error

>>% to list variables in "the workspace" use
>>who   % only list of variables and their type
>>whos  % also shows size
```

FYI - BLAS

- **BLAS** (**B**asic **L**inear **A**lgebra **S**ubroutines) is an ***efficient library for linear algebra***, available for **C** (**CBLAS**) and other languages.
- It is ***organized in levels***, with ***level k*** performing $O(n^k)$ operations, for instance,
  - **Level-1**: Performs vector operations (scaling, saxpy, inner products, etc. all take $O(n^1)$ *FLOPS*)
  - **Level-2**: Performs matrix-vector products, matrix addition, backward and forward subst., etc.
  - **Level-3**: Performs matrix-matrix operations
- A highly optimized interface for these libraries is provided by **ATLAS** http://math-atlas.sourceforge.net/

Applied Programming

Solving Systems of Linear Algebraic Equations

The Gaussian Elimination Approach

Solving Systems of Linear Algebraic Equations

Example: Which system is "easier" to solve ?

a) $$\begin{bmatrix} 2 & 8 & 6 \\ 4 & 2 & -2 \\ 3 & -1 & 11 \end{bmatrix} \boldsymbol{x} = \begin{bmatrix} 20 \\ -2 \\ 11 \end{bmatrix}$$

"dense" full system

b) $$\begin{bmatrix} 2 & 8 & 6 \\ 0 & -14 & -14 \\ 0 & 0 & 5 \end{bmatrix} \boldsymbol{x} = \begin{bmatrix} 20 \\ -42 \\ 20 \end{bmatrix}$$

upper triangular system

c) $$\begin{bmatrix} 2 & 0 & 0 \\ 0 & -14 & 0 \\ 0 & 0 & 5 \end{bmatrix} \boldsymbol{x} = \begin{bmatrix} 4 \\ 14 \\ 20 \end{bmatrix}$$

"sparse" diagonal system – not used

Solving Linear Algebraic Equations by Elimination

- "Elimination" (of variables) is the process of *reducing a "dense" system* of linear equations *to* another one (*upper triangular* or in general *upper row echelon*) that is easier to solve.
- *Gaussian Elimination* is an algorithm that organizes, in a *systematic way*, the process of *elimination of variables*

Note: After completing the elimination process there is still an additional step to actually solve the equations

Solving Linear Algebraic Equations by Elimination

Example: 3 equations in 3 unknowns

$$2x_1 + 8x_2 + 6x_3 = 20 \quad (1)$$

$$4x_1 + 2x_2 - 2x_3 = -2 \quad (2)$$

$$3x_1 - x_2 + x_3 = 11 \quad (3)$$

We will always start by eliminating the first variable x_1 first.

This algorithm is organized in passes, each pass eliminates one variable from the others

Solving Linear Algebraic Equations by Elimination

Pass one: Elimination of x_1

This is called a pivot

$$2x_1 + 8x_2 + 6x_3 = 20 \quad (1)$$
$$4x_1 + 2x_2 - 2x_3 = -2 \quad (2)$$
$$3x_1 - x_2 + x_3 = 11 \quad (3)$$

1. Eliminate x_1 from eqs. (2) and (3)

$$2x_1 + 8x_2 + 6x_3 = 20 \quad (1)$$

(2) $- 4/2 \times$ (1)-> $0x_1 - 14x_2 - 14x_3 = -42$ (2’)

(3) $- 3/2 \times$ (1)-> $0x_1 - 13x_2 - 8x_3 = -19$ (3’)

More detail

$$2x_1 + 8x_2 + 6x_3 = 20 \quad (1)$$
$$4x_1 + 2x_2 - 2x_3 = -2 \quad (2)$$
$$3x_1 - x_2 + x_3 = 11 \quad (3)$$

We want to “subtract” (1) from (2&3) in such a way to cause x_1 in (2&3) to be zero after we add.

e.g. multiply (1) by -4/2 and (1) by -3/2

-4/2 * [$2x_1 + 8x_2 + 6x_3 = 20$] => $-4x_1 - 16x_2 - 12x_3 = -40$ (2’)

-3/2 * [$2x_1 + 8x_2 + 6x_3 = 20$] => $-3x_1 - 12x_2 - 9x_3 = -30$ (3’)

Now add (2’&3’) into (2&3)

$$\begin{array}{rl} 4x_1 + 2x_2 - 2x_3 = -2 & (2) \\ \underline{-4x_1 - 16x_2 - 12x_3 = -40} & \underline{(2')} \\ 0 \quad - 14x_2 - 14x_3 = -42 & \end{array}$$

$$\begin{array}{rl} 3x_1 - x_2 + x_3 = 11 & (3) \\ \underline{-3x_1 - 12x_2 - 9x_3 = -30} & \underline{(3')} \\ 0 \quad -13x_2 - 8x_3 = -19 & \end{array}$$

New Equation:

$$\begin{aligned} 2x_1 + 8x_2 + 6x_3 &= 20 \\ -14x_2 - 14x_3 &= -42 \\ -13x_2 - 8x_3 &= -19 \end{aligned}$$

Solving Linear Algebraic Equations by Elimination

Pass two: Elimination of x_2

This is the new pivot

$$
\begin{aligned}
2x_1 + 8x_2 + 6x_3 &= 20 \quad (1)\\
-14x_2 - 14x_3 &= -42 \quad (2)\\
-13x_2 - 8x_3 &= -19 \quad (3)
\end{aligned}
$$

2. Eliminate x_2 from eq. (3)

$$
\begin{aligned}
2x_1 + 8x_2 + 6x_3 &= 20 \quad (1)\\
-14x_2 - 14x_3 &= -42 \quad (2)
\end{aligned}
$$

(3) $-13/14 \times$ (2): $\mathbf{5x_3 = 20}$ (3)

Elimination Completed, system is in triangular form

More detail

$$2x_1 + 8x_2 + 6x_3 = 20 \quad (1)$$
$$-14x_2 - 14x_3 = -42 \quad (2)$$
$$-13x_2 - 8x_3 = -19 \quad (3)$$

We want to "subtract" (2) from (3) in such a way to cause x_1 in (2&3) to be zero after we add. e.g. multiply (2) by -13/14

$$-13/14 * [-14x_2 - 14x_3 = -42] \Rightarrow +13x_2 + 13x_3 = 39 \quad (3')$$

Now add (3') into (3)

$$
\begin{array}{rrrl}
-13x_2 & - \ 8x_3 & = -19 & (3) \\
\underline{+13x_2} & \underline{+13x_3} & \underline{= 39} & \underline{(3')} \\
0 & 5x_3 & = 20 &
\end{array}
$$

Final triangular system

$$
\begin{array}{rrrl}
2x_1 + & 8x_2 + & 6x_3 = & 20 \quad (1) \\
& -14x_2 - & 14x_3 = & -42 \quad (2) \\
& & 5x_3 = & 20 \quad (3)
\end{array}
$$

Back Substitution

Final triangular system

$$\begin{aligned} 2x_1 + 8x_2 + \quad 6x_3 &= \quad 20 \qquad (1) \\ -14x_2 - 14x_3 &= -42 \qquad (2) \\ 5x_3 &= \quad 20 \qquad (3) \end{aligned}$$

Solve by **Back-substitution:**

$$x_3 = \frac{20}{5} \qquad = 4$$

$$x_2 = \frac{-42 - (-14x_3)}{-14} \qquad = -1$$

$$x_1 = \frac{20 - (8x_2 + 6x_3)}{2} \qquad = 2$$

Simple Back Substitution Algorithm

$$x_3 = \frac{20}{5} = 4$$

$$x_2 = \frac{-42 - (-14x_3)}{-14} = -1$$

$$x_1 = \frac{20 - (8x_2 + 6x_3)}{2} = 2$$

$$\begin{bmatrix} 2 & 8 & 6 \\ 0 & -14 & -14 \\ 0 & 0 & 5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 20 \\ -42 \\ 20 \end{bmatrix}$$

A x = b

- Notice:
  - Starts from the bottom row, up
  - Requires an extra solution "x" vector (more later)
  - The denominator is the pivot
  - The numerator contains the partial sum of the previous terms taken from 'b' for that row

Simple Back Substitution Algorithm

- Initialize the solution “x” vector to zero
- Process all the rows from the bottom to the top
  - Initialize a sum term to the current b vector row
  - Process the partial columns for this row (rows + 1)
    - Subtract the product of the remaining columns in the matrix from the solution
- Divide the final sum by the current pivot

Back Substitution Choices

- Simple back substitution
  - Requires an extra "x" vector to hold the answer
  - Process a row at a time and then each column
- **In-Place substitution**
  - No extra "x" vector to hold the answer
  - Process a column at a time then rows

"In-place" back Substitution

$$\begin{bmatrix} 2 & 8 & 6 \\ 0 & -14 & -14 \\ 0 & 0 & 5 \end{bmatrix} \begin{bmatrix} x1 \\ x2 \\ x3 \end{bmatrix} = \begin{bmatrix} 20 \\ -42 \\ 20 \end{bmatrix}$$

Solve for x3 = 20/5 = 4 (now we don't't' need the last "20")

$$\begin{bmatrix} 20 \\ -42 \\ 4 \end{bmatrix}$$

Use x3 in row 2: -42 –(-14*4) =14 Use x3 in row 3: 20 –(6*4) = **-4**

$$\begin{bmatrix} \mathbf{-4} \\ \mathbf{14} \\ \mathbf{4} \end{bmatrix}$$

Solve for x2 = 14/-14 = **-1**

$$\begin{bmatrix} \mathbf{-4} \\ \mathbf{-1} \\ \mathbf{4} \end{bmatrix}$$

Use x2 in row 1: -4 – (8*(-1)) = **4**

$$\begin{bmatrix} \mathbf{4} \\ \mathbf{-1} \\ \mathbf{4} \end{bmatrix}$$

Solve for $x1 = \frac{4}{2} = \mathbf{2}$

$$\begin{bmatrix} \mathbf{2} \\ \mathbf{-1} \\ \mathbf{4} \end{bmatrix}$$

Final answers for x

Process:

1) Solve for a root
2) Partially process the remaining data
3) Repeat until you are at the top.

In-place Back Substitution Algorithm

- Process the columns from right to left
  - New partial "b" column is the current divided by the pivot

  - Process partial rows for this column
    - New partial "b" is the current less the current multiplied by "M" entry for this row and column

- Solve the final entry b[0]

“in-place” pseudo-code

- Back substitution (solving $\mathbf{U}\boldsymbol{x} = \boldsymbol{c}$)
  - Requires: ($\mathbf{U}$ $n \times n$ upper triangular)

```
% j is row index, j=n,n-1, …,1
for j= n:-1:2                  % row index
    c(j) = c(j)/U (j,j)        % select pivot
   rows=1:j-1                  % works like a mini for loop
    c(rows) = c(rows) –c(j)U(rows,j)
end
c(1) = c(1)/U (1,1)            % The final entry
```

(Reference: Golub and Van Loan, Matrix Computations)

Note: This algorithms stores the solution in $\boldsymbol{c}$, (e.g., overwrites $\boldsymbol{c}$) and is origin 1

Complexity of Back substitution

- For an upper triangular n x n matrix the complexity of Back Substitution is:

$$\mathrm{FLOP} = 1 + \sum_{k=2}^{n} (2k - 1)$$
$$= n^2 + 2n + 1 = \mathcal{O}(n^2)$$

- **Gaussian Elimination + BS**

$$\mathrm{FLOP} = \frac{2}{3}n^3 + \frac{1}{2}n^2 - \frac{5}{6}n + 1 = \mathcal{O}\left(\frac{2}{3}n^3\right)$$

Note: After Gaussian Elimination is completed Back substitution is essentially "free" – Why ?

Solving Linear Algebraic Equations via "Gaussian Elimination"

- The numerical solution of any "dense" system of linear equations proceeds in ***two steps***
  1. Gaussian Elimination to a triangular system
  2. Back Substitution

Note: ***Gauss-Jordan Elimination***

- Reduces the original matrix to *diagonal*. This approach ***should not be used***

Gaussian Elimination: Matrix Setup

$$
\begin{aligned}
2x_1 + 8x_2 + 6x_3 &= 20 \quad (1)\\
4x_1 + 2x_2 - 2x_3 &= -2 \quad (2)\\
3x_1 - x_2 + x_3 &= 11 \quad (3)
\end{aligned}
$$

Start by representing the system of equations in *matrix form*: **A** ***x*** = **b**

$$
\underbrace{\begin{bmatrix} 2 & 8 & 6 \\ 4 & 2 & -2 \\ 3 & -1 & 1 \end{bmatrix}}_{\mathbf{A}} \times \underbrace{\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}}_{\boldsymbol{x}} = \underbrace{\begin{bmatrix} 20 \\ -2 \\ 11 \end{bmatrix}}_{\mathbf{b}}
$$

A is the coefficient matrix (n by n), **b** is coefficient vector (n by 1) and ***x*** is the vector of unknowns (n by 1)

Classical Gaussian Elimination

Form the ***auGmented*** *matrix* **G = [A | b]**

$$G = \left[\begin{array}{rrr|r} 2 & 8 & 6 & 20 \\ 4 & 2 & -2 & -2 \\ 3 & -1 & 1 & 11 \end{array}\right]$$

Apply the Gaussian Elimination process to ***reduce*** the **A** submatrix of **G** ***to "row echelon form"*** (e.g., upper triangle)

Note: With programming hints

Space-efficient Gaussian Elimination

- The Gaussian Elimination process *introduces many zeros* in a matrix being reduced
  - It is a waste of space to store zeros, ***don't store zeros***

- For efficient memory use it is common to implement the Gaussian Elimination process (and most linear algebra algorithms) ***"in-place"***

- **In-place** means that we will ***overwrite the original matrix*** entries, *where elimination introduces zeros*

Classical Gaussian Elimination (GE) (in-place)

- Illustration: solve the system

$$\begin{aligned} 2x_1 + 8x_2 + 6x_3 &= 20 \quad (1) \\ 4x_1 + 2x_2 - 2x_3 &= -2 \quad (2) \\ 3x_1 - x_2 + x_3 &= 11 \quad (3) \end{aligned}$$

- Form Augmented Matrix

$$G = \begin{bmatrix} 2 & 8 & 6 & 20 \\ 4 & 2 & -2 & -2 \\ 3 & -1 & 1 & 11 \end{bmatrix}$$

Example: GE in place

Pass one

- Choose pivot: 2

$$\begin{bmatrix} 2 & 8 & 6 & 20 \\ 4 & 2 & -2 & -2 \\ 3 & -1 & 1 & 11 \end{bmatrix}$$

- Find “multipliers”, store them “in place”

$$\begin{bmatrix} 2 & 8 & 6 & 20 \\ 4/2 & 2 & -2 & -2 \\ 3/2 & -1 & 1 & 11 \end{bmatrix}$$

- Complete elimination

$$\begin{bmatrix} 2 & 8 & 6 & 20 \\ 2 & -14 & -14 & -42 \\ 3/2 & -13 & -8 & -19 \end{bmatrix}$$

Multipliers are stored “in-place”

Reminder

$$2x_1 + 8x_2 + 6x_3 = 20 \quad (1)$$
$$4x_1 + 2x_2 - 2x_3 = -2 \quad (2)$$
$$3x_1 - x_2 + x_3 = 11 \quad (3)$$

$-4/2 * [2x_1 + 8x_2 + 6x_3 = 20] \Rightarrow -4x_1 - 16x_2 - 12x_3 = -40 \; (2')$

$-3/2 * [2x_1 + 8x_2 + 6x_3 = 20] \Rightarrow -3x_1 - 12x_2 - 9x_3 = -30 \; (3')$

Now add (2'&3') into (2&3)

$$\begin{array}{rrrrl} 4x_1 & + 2x_2 & - 2x_3 & = -2 & (2) \\ \underline{-4x_1} & \underline{- 16x_2} & \underline{- 12x_3} & \underline{= -40} & \underline{(2')} \\ 0 & - 14x_2 & - 14x_3 & = -42 & \end{array}$$

$$\begin{array}{rrrrl} 3x_1 & - x_2 & + x_3 & = 11 & (3) \\ \underline{-3x_1} & \underline{- 12x_2} & \underline{- 9x_3} & \underline{= -30} & \underline{(3')} \\ 0 & -13x_2 & - 8x_3 & = -19 & \end{array}$$

New Equation:

$$\begin{aligned} 2x_1 + 8x_2 + 6x_3 &= 20 \\ -14x_2 - 14x_3 &= -42 \\ -13x_2 - 8x_3 &= -19 \end{aligned}$$

Example: GE in place

Pass two:

- Choose pivot: -14

$$\begin{bmatrix} 2 & 8 & 6 & 20 \\ 2 & \mathbf{-14} & -14 & -42 \\ 3/2 & -13 & -8 & -19 \end{bmatrix}$$

- Find scaling

$$\begin{bmatrix} 2 & 8 & 6 & 20 \\ 2 & -14 & -14 & -42 \\ 3/2 & 13/14 & -8 & -19 \end{bmatrix}$$

- Combine rows

$$\begin{bmatrix} 2 & 8 & 6 & 20 \\ 2 & -14 & -14 & -42 \\ 3/2 & \boxed{13/14} & 5 & 20 \end{bmatrix}$$

Multipliers stored "in-place"

Reminder

$$2x_1 + 8x_2 + 6x_3 = 20 \quad (1)$$
$$-14x_2 - 14x_3 = -42 \quad (2)$$
$$-13x_2 - 8x_3 = -19 \quad (3)$$

$-13/14 * [-14x_2 - 14x_3 = -42] \Rightarrow +13x_2 + 13x_3 = 39$ (3')

Now add (3') into (3)

$$\begin{array}{rrl} -13x_2 & - 8x_3 = -19 & (3) \\ \underline{+13x_2} & \underline{+13x_3 = 39} & \underline{(3')} \\ 0 & 5x_3 = 20 & \end{array}$$

Final triangular system

$$\begin{aligned} 2x_1 + 8x_2 + 6x_3 &= 20 \quad (1) \\ -14x_2 - 14x_3 &= -42 \quad (2) \\ 5x_3 &= 20 \quad (3) \end{aligned}$$

Final Result of GE in place

$$\left[\begin{array}{ccc|c} 2 & 8 & 6 & 20 \\ 2 & -14 & -14 & -42 \\ 3/2 & 13/14 & 5 & 20 \end{array}\right]$$

- The solution can now be obtained by back substitution, *e.g.*, solving **U** ***x*** = ***c***

$$\underbrace{\begin{bmatrix} 2 & 8 & 6 \\ 0 & -14 & -14 \\ 0 & 0 & 5 \end{bmatrix}}_{\mathbf{U}} \underbrace{\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}}_{\boldsymbol{x}} = \underbrace{\begin{bmatrix} 20 \\ -42 \\ 20 \end{bmatrix}}_{\boldsymbol{c}}$$

Algorithm GE.1

- Gaussian Elimination ($n \times n$ **A** matrix)

  (vectorized "in-place" pseudo-code)

```
% initialize GE: form augmented matrix G
G = [A b];
% k is "pass" index, j=1…n-1
for k= 1: n-1   % k^th pass loop
    pivot = G(k,k)          % select k^th pivot
    rows = k+1:n            % index of entries below pivot
    cols = k+1:n+1          % index of entries right of pivot
   % Store scaling factors "in-place", below k^th pivot
    G(rows,k) = G(rows,k) /pivot
   % Reduce submatrix below and right of k^th pivot
    G(rows,cols) = G(rows,cols)- G(rows,k) G(k,cols)
end
```

(Reference: Golub and Van Loan, Matrix Computations)

Complexity Estimates

- A classical estimate of the complexity is given by the number of FLoating-Point Operations (FLOP):

- For a *square n x n matrix* the *complexity of GE is*

$$
\begin{aligned}
\mathrm{FLOP} &= \sum_{k=1}^{n-1} \left(n - k + 2\,(n-k)^2\right) \\
&= \sum_{k=1}^{n-1} \left(2\,k^2 - (1+4\,n)\,k + n(1+2\,n)\right) \\
&= \frac{2}{3}n^3 - \frac{1}{2}n^2 - \frac{1}{6}n
\end{aligned}
$$

- *The complexity of Gaussian Elimination* $\boxed{\mathcal{O}(\frac{2}{3}n^3)}$

Complexity and Computing Time

- A related quantity of computer performance is the ***FLOPS*** (***FL***oating-point ***O***perations ***P***er ***S***econd),

- An estimate of a FLOPS is

$$\text{cores} \times \text{clock} \times \left(\frac{\text{FLOP}}{\text{cycle}}\right)$$

- Today most microprocessors can perform *4 FLOP per clock cycle*.
- For example, a single-core 2.5 GHz processor has a theoretical performance of

$$1 \times 2.5E6 \times 4 = 10\text{Giga FLOPS}$$

Complexity and Computing Time

Given: GE ~ $\mathcal{O}(\frac{2}{3}n^3)$

- About how long does it take a **`1 GFLOPS`** (10^9) computer to solve a system of *100 linear equations in 100 variables* (n=100)?

$$t \approx \frac{\text{Complexity in FLOP}}{\texttt{FLOPS}} = \frac{100^3}{10^9} = 10^{-3}\text{sec}$$

Ans: 1 millisecond

Gaussian Elimination Theory

- Gaussian Elimination reduces a system of linear equations to a row-echelon form (**U**) without changing its solution.
- Only the following elementary operations are allowed:
  1. Scaling a row by a non-zero constant
  2. Adding any multiple of a row to another row
  3. Interchanging rows
- Operations 1 and 2 can be encoded in a unit (ones on the diagonal) lower triangular matrix **L**
- Operation 3 can be represented by a permutation matrix **P** and encoded in a permutation vector (more later)

Problem 1

- Given: x = [1 2 3] y = [1 2 3]
- Find the inner (dot) product z=x•y

- Sum of the vector products,
  - Matlab: z = dot(x,y)

$$= \sum_{i=1}^{3} a_i \, b_i$$

- z = 1*1+2*2+3*3 = 14

Problem 2

- Transpose $Y = \begin{bmatrix} 1 & 0 \\ 2 & 0 \\ 3 & 0 \end{bmatrix}$

- Column 1 becomes row 1, column 2 becomes row 2.

$$Y' = Y^T = \begin{bmatrix} 1 & 2 & 3 \\ 0 & 0 & 0 \end{bmatrix}$$

Problem 3

- Given: $X = \begin{bmatrix} 1 & 1 \\ 2 & 1 \\ 3 & 1 \end{bmatrix}$ $Y' = Y^T = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 2 & 2 \end{bmatrix}$
- If we calculate xy^T (or xy', or the outer product), what will be the size of the resulting matrix?

- X is 3x2, Y is 2x3 so the resulting matrix is 3x3

Problem 4

- Given: $x = \begin{bmatrix} 1 & 1 \\ 2 & 1 \\ 3 & 1 \end{bmatrix}$ $y' = y^T = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 2 & 2 \end{bmatrix}$
- Find xy^T (or xy', or the outer product)
- Reminder: sum of: row * col
- Outer product or '*' in Matlab: z = x*y'

$$z = \begin{matrix} 1*1+1*2 & 1*2+1*2 & 1*3+1*2 \\ 2*1+1*2 & 2*2+1*2 & 2*3+1*2 \\ 3*1+1*2 & 3*2+1*2 & 3*3+1*2 \end{matrix} \quad \text{or} \quad \begin{matrix} 3 & 4 & 5 \\ 4 & 6 & 8 \\ 5 & 8 & 11 \end{matrix}$$

Problem 5

- Given: $x = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$ $A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix}$ $B = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix}$
- Find $Q = A^T B x$ (inner or dot products)
  - Hint, use two steps: $Z = A^T B$, $Q = Zx$

Problem 5

$$A = \begin{matrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{matrix} \quad \begin{matrix} \text{swap} \\ \text{columns} \\ \text{with rows} \end{matrix} \quad A' = \begin{matrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 1 \end{matrix}$$

Z = A'B or Z = A'*B (rows x columns)

$$\begin{matrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 1 \end{matrix} * \begin{matrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{matrix} = \begin{matrix} 1*1+0+0 & 1*2+0+0 & 1*3+0+0 \\ 0 & 0 & 0 \\ 0+1*1+1*1 & 0+1*2+1+2 & 0+1*3+1*3 \end{matrix}$$

$$Z = \begin{matrix} 1 & 2 & 3 \\ 0 & 0 & 0 \\ 2 & 4 & 6 \end{matrix}$$

Problem 5

$$Q = Zx = \begin{matrix} 1 & 2 & 3 \\ 0 & 0 & 0 \\ 2 & 4 & 6 \end{matrix} * \begin{matrix} 1 \\ 2 \\ 3 \end{matrix} = \begin{matrix} 1+4+9 \\ 0 \\ 2+8+18 \end{matrix}$$

$$Q = \begin{matrix} 14 \\ 0 \\ 28 \end{matrix}$$

Problem 6

- Given: $x = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$ $A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix}$
- Find $x^T A$ (or x'A)

Ans. x is a column vector, so x' is a row vector.
A is a matrix

$$x'*A = [1\ 2\ 3] * \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix} = [\ 1 \quad 0 \quad 2*1+3*1]$$

$$x'*A = [\ 1\ 0\ 5]$$

Problem 7

- Given: $x=\begin{bmatrix}1\\2\\3\end{bmatrix}$ $C=\begin{bmatrix}1 & 2 & 3\\0 & 2 & 3\\0 & 0 & -1\end{bmatrix}$ Solve Cz=x using GE

Ans. The matrix is already in the upper triangle form, so just back substitute.

Cz= x => $\begin{bmatrix}1 & 2 & 3\\0 & 2 & 3\\0 & 0 & -1\end{bmatrix}\begin{bmatrix}z_1\\z_2\\z_3\end{bmatrix}=\begin{bmatrix}1\\2\\3\end{bmatrix}$

$z_3 = -3$

$2z_2+3z_3 = 2z_2-9 = 2 \Rightarrow z_2 = 11/2 = 5.5$

$z_1+2z_2+3z_3 = z_1+11-9 = z_1+2 = 1 \Rightarrow z_1 = -1$

$z = \begin{bmatrix}-1\\5.5\\-3\end{bmatrix}$

Problem 8

- Solve the system of linear equations using Gaussian elimination

$$x1 - x2 + 3x3 = 2$$

$$x1 + x2 = 4$$

$$3x1 - 2x2 + 3x3 = 1$$

Ans. Convert to standard form:

$$1x1 - 1x2 + 3x3 = 2$$

$$1x1 + 1x2 + 0x3 = 4$$

$$3x1 - 2x2 + 3x3 = 1$$

Problem 8

Ax=b => $\begin{bmatrix} 1 & -1 & 3 \\ 1 & 1 & 0 \\ 3 & -2 & 3 \end{bmatrix} \begin{bmatrix} x1 \\ x2 \\ x3 \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \\ 1 \end{bmatrix}$

gives augmented matrix

$$G = \begin{bmatrix} 1 & -1 & 3 & 2 \\ 1 & 1 & 0 & 4 \\ 3 & -2 & 3 & 1 \end{bmatrix}$$

row 2 = row 2 - 1/1(row 1) = [1 1 0 4] - [1 -1 3 2] = [0 2 -3 2]

row 3 = row3 - 3/1(row 1) = [3 -2 3 1] - 3 [1 -2 3 2] = [0 1 -6 -5]

$$G = \begin{bmatrix} 1 & -1 & 3 & 2 \\ 0 & 2 & -3 & 2 \\ 0 & 1 & -6 & -5 \end{bmatrix}$$

Problem 8

$$G = \begin{bmatrix} 1 & -1 & 3 & 2 \\ 0 & 2 & -3 & 2 \\ 0 & 1 & -6 & -5 \end{bmatrix}$$

row 3 = row3 - 4/2(row 2) = [0 1 -6 -5] +1/2[0 2 -3 2] = [0 0 -12 -1]

$$G = \begin{bmatrix} 1 & -1 & 3 & 2 \\ 0 & 2 & -3 & 2 \\ 0 & 0 & -4.5 & -6 \end{bmatrix}$$

$$Ux = c = \begin{bmatrix} 1 & -1 & 3 \\ 0 & 2 & -3 \\ 0 & 0 & -4.5 \end{bmatrix} \begin{bmatrix} x1 \\ x2 \\ x3 \end{bmatrix} = \begin{bmatrix} 2 \\ 2 \\ -6 \end{bmatrix}$$

x3 = 6/4.5 = 1.333

2x2-3x3 = 2x2-3(1.333) = 2 => x2 = 3

x1-x2+3x3 = x1-(3)+3(1.333) = 2 => x1 = 1

$$x = \begin{bmatrix} 1 \\ 3 \\ 1.333 \end{bmatrix}$$

Appendix

Appendix: Useful Formulas for Complexity Analysis

$$\sum_{k=0}^{n} k = \frac{1}{2}n^2 + \frac{1}{2}n = \frac{n(n+1)}{2}$$

$$\sum_{k=0}^{n} k^2 = \frac{1}{3}n^3 + \frac{1}{2}n^2 + \frac{1}{6}n$$