Linear Systems (a.k.a. systems of linear equations)

to accompany Ch. 2 of **Data-Driven Modeling & Scientific Computation** by J. Nathan Kutz

- **linear** $\iff$ each term involves an unknown with a constant coefficient
- **affine** $\iff$ includes constant terms
- Start with n equations in n unknowns $x_0, x_1, \ldots, x_{n-1}$:

$$\begin{aligned}
a_{0,0}x_0 + a_{0,1}x_1 &\quad + \ldots + a_{0,n-1}x_{n-1} &&= b_0 \\
a_{1,0}x_0 + a_{1,1}x_1 &\quad + \ldots + a_{1,n-1}x_{n-1} &&= b_1 \\
\vdots &\quad \ddots &&\vdots \\
a_{n-2,0}x_0 + a_{n-2,1}x_1 &\quad + \ldots + a_{n-2,n-1}x_{n-1} &&= b_{n-2} \\
a_{n-1,0}x_0 + a_{n-1,1}x_1 &\quad + \ldots + a_{n-1,n-1}x_{n-1} &&= b_{n-1}
\end{aligned}$$

- Left-hand side (LHS) is a **linear system**; all together, a set of **affine** equations

Nobody wants to write (or typeset) that over and over so...

- Collect variables and RHS as **vectors** (1D arrays)
  - Typical elements x_j, b_i
- Collect coefficients as **matrix** (2D array)
  - Typical element $a_{i,j}$: $i \iff$ row; $j \iff$ column

-Abbreviations:

- $\sum_{j=0}^{n-1} a_{i,j} x_j = b_i$
  - Einstein Summation Convention $\implies$ $a_{i,j} x_j = b_i$ (sum on repeated index)
  - Coordinate free notation: $Ax = b$

Section 2.1 - Direct Solution Methods for $Ax = b$

Start with the original solution method: **Gaussian elimination**.

> Note likely historical misattribution. According to Wikipedia, first introduced in "Western" math by Newton, but the first used in China possibly 2 millenia ago.

- Basic matrix operations provide opportunites to use `for()` loops
- Take advantage of opportunities to develop our execution control skills.
- **Basic idea**:
  - Create **augmented matrix** $[A|b]$ (attach RHS as extra column)
  - Perform elementary row operations until the matrix becomes the identity.
  - At the end of that process, augmented column contains the solution x.
  - Conceptually simple, but not most efficient...

Approach:

- Code up simpler things and work our way up to a full solver. Start by:
- Importing `numpy` (to get access to `array` capabilities)
- Creating a sample matrix (2D array) and vector (array with a single row or column).

Example in text involves matrix $a = [[1, 1, 1], [1, 2, 4], [1, 3, 9]]$.
There are many ways to create a corresponding array in python. The "manual" approach (where you enter the desired values from the keyboard) is straightforward, but perhaps not terribly instructive.

Manually construct array and print array followed by its transpose.

```
import numpy as np #import numpy for access to array capabilities
a = np.array([[1,1,1],[1,2,4],[1,3,9]])
print('a=\n',a, "\ntranspose of a=\n", a.T)
```

```
a=
[[1 1 1]
[1 2 4]
[1 3 9]]
transpose of a=
[[1 1 1]
[1 2 3]
[1 4 9]]
```

Transpose `a.T` suggests algorithmic approach (generalizes to different array sizes). Each row in the transpose contains powers of a sequence of integers starting with 1. Code below sets the matrix size and creates a corresponding array of zeros.

```
n = 3 #set the size of the square matrix
a_transpose = np.zeros([n,n]) # create a 3x3 array initialized with zeros
# reassign the entries according to a_transpose[i,j] = 1+i+j**2
for i in range(n):
    for j in range(n):
        a_transpose[i,j] = (j+1)**i
a = a_transpose.T
print(a)
```

```
[[1. 1. 1.]
[1. 2. 4.]
[1. 3. 9.]]
```

Define function to construct $n \times n$ matrix of this type:

```
def make_a(n):
    """
    construct n x n matrix with columns containing powers of natural numbers
        Inputs: n, integer number of rows/columns
        Returns: 2D array
    """
    a = np.zeros([n,n]) #create array of size needed
    for i in range(n): #loop over rows
        for j in range(n): #loop over columns
            a[i,j] = (j+1)**i #assign value
    return a.transpose()

a = make_a(3); print('a=\n',a) #call function and print result
```

```
a=
[[1. 1. 1.]
[1. 2. 4.]
[1. 3. 9.]]
```

Construct array corresponding to RHS: $b = [1, -1, 1]$:

```
b = np.array([1,-1,1])
print('b=',b)
```

b= [1 -1 1]

Again construct function to produce arrays of given size:

```
def make_b(n):
    a = np.zeros(n)
    for i in range(n):
        a[i]=(-1)**i
    return a

print('b(',n,')=', make_b(7))
```

b(7)= [1. -1. 1. -1. 1. -1. 1.]

> Aside: b looks like a row-vector. Column version can be produced in various ways including:

```
print('transpose version:\n', np.transpose([b]))
print('newaxis version:\n',b[:, np.newaxis])
```

Test them out!

Starting toward a solver

Construct augmented matrix by appending b as an extra column of a

```
a = make_a(3)
b = make_b(3)
auga = np.column_stack([a,b])
# alternate using np.c_ (check docs for details)
# auga= np.c_[a,b]
print('augmented matrix:\n',auga)
```

augmented matrix:

[[1. 1. 1. 1.]

[1. 2. 4. -1.]

[1. 3. 9. 1.]]

Create function for specified size:

```
def augment(A,b):
    '''
    Construct augmented matrix
    Inputs:
        A: 2-dim numpy array of shape (n,n)
        b: 1- numpy array of length n
    Returns:
        n x (n+1) numpy array
    '''
    m,n = A.shape
    if m!=n:
        print("Input A is not square.")
    return np.column_stack[A,b]
auga = augment(a,b)
print('The augmented matrix is:\n', auga)
```

The augmented matrix is: [[1. 1. 1. 1.]

[1. 2. 4. -1.]

[1. 3. 9. 1.]]

Simplify using elementary row operations

- Swap rows
- Multiply row by a non-zero constant
- Add multiple of one row to another

Elementary row ops leave solution unchanged

- After implementation, apply elementary row ops to systematically transform/simplify until we can readily solve
- Be a little clever and implement them all in a single function

```
def row_op(A,c,i1,i2):
    """
    perform elementary row operations on 2D array A
    if i1==i2, multiply row by constant c
    if i1!=i2, add c*(row i1) to row i2
    Args:
        A: 2D numpy array representing a matrix
        c: float multiplicative constant
        i1,i2: int row indices
    """
    m,n = A.shape #number of rows and columns
    if  i1<0 or i2<0 or i1>=m or i2>=m:
        print("WARNING: Invalid index specifications. Each index value i must satisfy 0<=i<#rows.")
    if i1==i2: #repeated index -> multiply row by constant
        for j in range(n):
            A[i1,j] *= c #Equiv. to A[i1,j] = c*A[i1,j]
    else: # add c*row i1 to row i2
        for j in range(n):
            A[i2,j] += c * A[i1,j] # Equiv. to A[i2,j] = A[i2,j] + c*A[i1,j]
    return
```

Simplification scheme

Use row ops to obtain **triangular** form (entries below main diagonal are 0). How?

- Choose a diagonal element `A[p,p]` as a "pivot".
- Divide row `p` by pivot (to produce `A[p,p]` = 1 at pivot on diagonal)
- Perform row operations to zero out each coefficient beneath it in column `p`
  - Subtract `A[i,p]` times pivot row `p` from row `i`.
- If preceding entries in a row are already zero, they remain zero during elementary row ops.
- Proceed systematically across the columns:
  - Zero out below diagonal to produce upper triangular system.

Implement: Write `cancel_below_diagonal()` to zero out elements in lower triangle

```
def cancel_below_diagonal(A, pivot):
    """
    insert docstring here
    """
    SMALL_VALUE = 1E-8 # check for possible overflow
    m,n = A.shape
    if  pivot<0 or pivot>=m:
        print("WARNING: Invalid index specification. Index value pivot must satisfy 0<=pivot<#rows.")
    if abs(A[pivot,pivot]) < SMALL_VALUE:
        print("WARNING: Division by near-zero pivot value.")
    else:
        # row_op(A,1./A[pivot,pivot], pivot, pivot) #divide by pivot value so value at pivot position becomes 1
        for i in range(pivot+1,m):
            row_op(A,-A[i,pivot]/A[pivot,pivot],pivot,i)
```

```
#test the function by zeroing the first column below the diagonal
auga = augment(a,b)
cancel_below_diagonal(auga,0)
print("Original matrix A = \n", a)
print("Augmented matrix after processing 1st pivot = \n", auga)
```

Original matrix A =

[[1. 1. 1.]

[1. 2. 4.]

[1. 3. 9.]]

Augmented matrix after processing 1st pivot =

[[1. 1. 1. 1.]

[0. 1. 3. -2.]

[0. 2. 8. 0.]]

```
#continue by zeroing the second column below the diagonal
cancel_below_diagonal(auga,1)
print("Augmented matrix after processing 2nd pivot = \n", auga)
```

Augmented matrix after processing 2nd pivot =

[[1. 1. 1. 1.]

[0. 1. 3. -2.]

[0. 0. 2. 4.]]

Successfully achieve upper triangular form (with 0 below main diagonal `A[i,i]`)

Define function `upper_tri()` to triangularize in single call:

```
def upper_tri(A):
    m,n = A.shape
    for i in range(m):
        cancel_below_diagonal(A,i)

auga = augment(a,b)
upper_tri(auga)
print("Inspect to verify that augmented array has been triangularized:\n", auga)
```

Inspect to verify that augmented array has been triangularized:

[[1. 1. 1. 1.]

[0. 1. 3. -2.]

[0. 0. 2. 4.]]

Backsolve/backsubstitution

- Given triangular matrix, the last row corresponds to a linear equation in 1 variable that can be solved immediately: $x_2 = A_{3,4}/A_{3,3} = 4./2. = 2.$
- Back-substitute value of x_2 into rows above.
- Next row up gives equation to solve for x_1.
- Plug in above; solve for x_0 to complete solution.

Are you done at that point?

Check your result.

How? Compute the residual $r = Ax - b$; should be close to $\vec{0}$.

Implement a `back_sub()` function to execute the back substitution process:

```
def back_sub(augU):
    """
    Insert suitable docstring here.
    """
    m,n = augU.shape
    x = np.zeros(m)
    for i in range(m):
        x[m-1-i]= augU[m-1-i,-1] #Initialize solution entry with value from RHS of tri. system
        for j in range(m-i,m): #For each entry of the row right of the main diagonal
            x[m-i-1] -= augU[m-i-1,j]*x[j] #Subtract coeff. * (known/larger-index entry in solution)
        x[m-1-i] /= augU[m-1-i, m-1-i] #Divide by pivot to get the new entry in the solution
    return x
```

Test functions for triangularizing and back-substituting:

```
a = make_a(3)
b = make_b(3)
auga = augment(a,b)
upper_tri(auga)
print("Triangular aumented matrix:\n", auga)
soln = back_sub(auga)
print("Solution obtained by back-substitution:", soln)
print("Check that solution satisfies the equations:\nResidual = ", np.dot(a, soln) - b)
```

Triangular aumented matrix:

[[1. 1. 1. 1.]

[0. 1. 3. -2.]

[0. 0. 2. 4.]]

Solution obtained by back-substitution: [7. -8. 2.]

Check that solution satisfies the equations:

Residual = [0. 0. 0.]

Modify test problem with more pivots not equal to 1:

```
a = np.array([[3., 3., 3.], [2., 4., 8.], [1., 3., 9.]])
b = np.array([3., -2., 1.])
auga = augment(a,b)
upper_tri(auga)
print("Triangular aumented matrix:\n", auga)
soln = back_sub(auga)
print("Solution obtained by back-substitution:", soln)
residual = np.dot(a, soln) - b
print("Residual = ", residual, ", Norm of the residual = ", np.linalg.norm(residual))
```

Triangular aumented matrix:

[[3. 3. 3. 3.]

[0. 2. 6. -4.]

[0. 0. 2. 4.]]

Solution obtained by back-substitution: [7. -8. 2.]

Residual = [0. 0. 0.], Norm of the residual = 0.0

```
# check that Ax agrees with b to within a threshold
# print the entries in Ax-b
print("Ax-b = ", a.dot(soln)-b)
# use numpy's `allclose` function to check for numerical agreement
# to within a specified absolute and/or relative tolerance
print("Solution checks? : ", np.allclose(np.dot(a,soln),b, atol = 1e-10))
```

```
Ax-b = [0. 0. 0.]
Solution checks? : True
```

This approach basically works (at least in the cases we have seen so far), but the whole process needs to run again to solve with a different right-hand side `b`. Instead of repeating the computation, it is more convenient to rephrase the problem in terms of matrix factorization. The particular factorization of interest here is called the **LU Factorization** because it involves rewriting A as the product of a lower-triangular matrix L and the upper-triangular U that was computed above by row reduction.

Once the factorization is computed, the solution for any right-hand side can be found by solving the 2 triangular systems $Ly = b$ and then $Ux = y$.

> Note that the 2 equations together are equivalent to the original system:
>
> $$Ly = b \wedge Ux = y \implies L(Ux) = b \iff (LU)x = b \iff Ax = b.$$

Next goal: Write function to compute the LU factorization and to solve given L, U, b.

First, some context:

Previously discussed solution by **Gaussian Elimination**:

- Perform elementary row ops to triangularize
- Solve triangular system by back substitution
- Bascally works (at least for simple cases seen so far)
- What if we need to solve again with different b on RHS?

Instead of starting over, "record" row ops for triangularization.

Change our perspective from row ops to matrix multiplication:

- What matrix multiplication is needed to undo the row operation?
- Keep both the updated matrix and the matrix that undoes the update (to preserve the initial matrix as the product)

Back to example:

```
a = make_a(3)
b = make_b(3)
auga = augment(a,b)
print(auga)
```

[[1. 1. 1. 1.]

[1. 2. 4. -1.]

[1. 3. 9. 1.]]

First triangularization step: Subtract Row 0 from Rows 1 and 2 to zero out the subdiagonal entries in Column 0.

What operation undoes the cancellation? Add Row 0 to Row 1 and Row 2.

How do we write that in the language of linear algebra?

When we multiply matrix A by column vector x, each entry in the output is obtained by multiplying a row by the entries in x and summing.

Net result: Multiply a matrix by a column vector (on the right) to produce a linear combination of the columns of A with coefficients in x.

Here we want linear combinations of rows, so we transpose:

- Pre-multiply the matrix by a row of coefficients to produce the corresponding linear combination of the rows.
- Stack the rows to form a matrix.

$$\left\{\begin{array}{lcl} R_1 & \leftarrow & R_1 \\ R_2 & \leftarrow & 1 * R_1 + R_2 \\ R_3 & \leftarrow & 1 * R_1 + R_3 \end{array}\right\} \iff \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix} A$$

Do the factorization described to clear the first column below the diagonal:

$$A = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 2 & 4 \\ 1 & 3 & 9 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 3 \\ 0 & 2 & 8 \end{pmatrix}$$

Factorize rightmost matrix to clear the second column (below diagonal)

$$A = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 2 & 4 \\ 1 & 3 & 9 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 2 & 1 \end{pmatrix} \begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 3 \\ 0 & 0 & 2 \end{pmatrix}$$

Multiply the first 2 matrices together (noting that the first column is preserved):

$$A = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 2 & 4 \\ 1 & 3 & 9 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 2 & 1 \end{pmatrix} \begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 3 \\ 0 & 0 & 2 \end{pmatrix} = LU$$

And A is now factored into lower triangular L and upper triangular U.

More realistic example with fewer 1's:

Follow convention - Leave the pivots on diagonal in U with 1's on diagonal of L:

$$A = \begin{pmatrix} 4 & 3 & 2 \\ 16 & 14 & 9 \\ 12 & 13 & 13 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 4 & 1 & 0 \\ 3 & 0 & 1 \end{pmatrix} \begin{pmatrix} 4 & 3 & 2 \\ 0 & 2 & 1 \\ 0 & 4 & 7 \end{pmatrix}$$

$$A = \begin{pmatrix} 4 & 3 & 2 \\ 16 & 14 & 9 \\ 12 & 13 & 13 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 4 & 1 & 0 \\ 3 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 2 & 1 \end{pmatrix} \begin{pmatrix} 4 & 3 & 2 \\ 0 & 2 & 1 \\ 0 & 0 & 5 \end{pmatrix}$$

$$A = \begin{pmatrix} 4 & 3 & 2 \\ 16 & 14 & 9 \\ 12 & 13 & 13 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 4 & 1 & 0 \\ 3 & 2 & 1 \end{pmatrix} \begin{pmatrix} 4 & 3 & 2 \\ 0 & 2 & 1 \\ 0 & 0 & 5 \end{pmatrix} = LU$$

L is lower triangular with 1's on the diagonal

U is upper triangular with pivot's on main diagonal

Can be stored together in a single $n \times n$ array (in-place factorization):

- Upper triangle stores U (with pivots on the diagonal)
- Off-diagonal elements of L stored below the diagonal (diagonal of 1's need not be stored explicitly)
- product of pivots $= \sum_{i=0}^{n-1} a_{i,i} = Det(A)$

```
# check that the factorization and det
you = np.array([[4,3,2],[0,2,1],[0,0,5]])
ell = np.array([[1,0,0],[4,1,0],[3,2,1]])
product_of_factors = np.dot(ell, you)
pivots = np.diag(you)
print("Product of factors = \n", product_of_factors, "\nU = \n", you)
print("Pivots = ", pivots)
print("Product of pivots = ", np.prod(pivots))
print("Det(A) = ", np.linalg.det(product_of_factors))
```

Product of factors =

[[4 3 2]

[16 14 9]

[12 13 13]]

U =

[[4 3 2]

[0 2 1]

[0 0 5]]

Pivots = [4 2 5]

Product of pivots = 40

Use factorization to solve linear system as 2 triangular solves:

$Ax = b \iff LUx = b \iff Ly = b$ where $Ux = y$

- Factor: $A \to LU$
- Solve $Ly = b$
- Solve $Ux = y$

Code up LU factorization and triangular solvers

```
def LU_factor(A):
    m,n = A.shape
    if m != n:
        print("WARNING: Non-square input matrix")
        return
    mult = 0
    U = np.copy(A) #make a copy of the array
    #Note that U=A just makes another name for A, not a new copy of the array
    L = np.eye(n) #numpy's name for the identity matrix is "eye"
    for i in range(n): # for each row i
        for j in range(i+1,n): # for each row below row i
            mult = U[j,i]/U[i,i]
            L[j,i] = mult
            for k in range(i,n): # for each entry beyond the i^th diagonal entry
                U[j,k] = U[j,k] - mult*U[i,k] # for entries to the right, subtract multiple of term in row i
    return L,U
```

```
#test the LU_factor function
L,U = LU_factor(make_a(3))
print("L= \n", L,"\nU=\n",U)
```

L=

[[1. 0. 0.]

[1. 1. 0.]

[1. 2. 1.]]

U=

[[1. 1. 1.]

[0. 1. 3.]

[0. 0. 2.]]

```
# check that the factorization reproduces the input matrix
np.dot(L,U)
```

array([[1., 1., 1.],

[1., 2., 4.],

[1., 3., 9.]])

```
def upper_tri_solve(U,b):
    """
    insert docstring here
    """
    m,n = U.shape #matrix has m rows and n columns
    x=np.zeros(m) #create an array to store the solution (init to zeros)
    for i in range(m):
        row = m-i-1
        accum=0 #variable to store sum of coeffs times known entries in solution
        for j in range(i):
            accum+=U[row,j]*x[j]
        x[row]=(b[row]-accum)/U[i,i] #solve for i^th entry in solution
    return x
```

```
def lower_tri_solve(L,b):
    """
    insert docstring here
    """
    m,n = L.shape #matrix has m rows and n columns
    # should really check for compatible size
    y=np.zeros(m) #create an array to store the solution (init to zeros)
    for i in range(m):
        row = i
        accum=0
        for j in range(i):
            accum+=L[row,j]*y[j]
        y[row]=(b[row]-accum)/L[i,i] #solve for i^th entry in solution
    return y
```

```
# check lower_tri_solve
b = make_b(3)
y = lower_tri_solve(L,b)
print("y=",y)
print("residual = ", np.dot(L,y)-b)
```

y= [1. -2. 4.]

residual = [0. 0. 0.]

```
def LU_solve(L,U,b):
    y = lower_tri_solve(L,b)
    x = upper_tri_solve(U,y)
    return x, y
```

```
N = 20
A,b = make_a(N), make_b(N)
L,U = LU_factor(A)
x,y = LU_solve(L,U,b)
residual = np.dot(A,x)-b
np.set_printoptions(precision=2)
print("x=",x, "\ny=", y, "\nresidual = ", residual)
print("Norm of residual = ", np.linalg.norm(residual))
```

```
x= [ 1.05e+06 -3.66e+06 5.56e+06 -4.96e+06 2.94e+06 -1.24e+06 3.90e+05
-9.36e+04 1.75e+04 -2.57e+03 2.99e+02 -2.77e+01 2.04e+00 -1.18e-01
5.31e-03 -1.82e-04 4.58e-06 -7.98e-08 8.60e-10 -4.31e-12]
y= [ 1.00e+00 -2.00e+00 4.00e+00 -8.00e+00 1.60e+01 -3.20e+01 6.40e+01
-1.28e+02 2.56e+02 -5.12e+02 1.02e+03 -2.05e+03 4.10e+03 -8.19e+03
1.64e+04 -3.28e+04 6.55e+04 -1.31e+05 2.62e+05 -5.24e+05]
residual = [ 1.04e-10 -2.01e-10 9.83e-09 4.74e-08 -5.19e-07 -3.04e-06 -1.61e-05
-3.97e-05 -8.90e-05 -2.43e-04 -3.00e-04 -1.35e-03 -2.22e-03 -8.02e-03
-3.26e-02 -3.74e-02 3.03e-02 -2.28e-02 1.47e-01 1.06e+00]
```

- Residual has entries ≈ 1
- Norm of residual = 1.068 which is NOT small.
- What went wrong?
- Problem becomes ill-conditioned as the matrix becomes larger.
- A (not reliable) indicator is that $Det(A)$ becomes large.

```
d = 1
for i in range(N):
    d *= U[i,i]
print("Det(A[", N, "]) = ", d)
```

Det(A[20]) = 5.238587393322858 4e+137

Counter example ($Det(A) >> 1$ but matrix solves nicely)

Return to consider reliable indicator: **condition number**

Cramer's rule and Laplace expansion

Can immediately write down the solution for any variable in the linear system $Ax = b$:

$$x_i = \frac{Det(A_i)}{Det(A)}$$

- A_i is a with the i^{th} column replaced by b - - Can evaluate the determinants recursively by Laplace expansion:

$$Det(A) = \sum_{i=0}^{n-1} (-1)^{i+j} Det(A_{i,j})$$

- $A_{i,j}$ is A with row i and column j removed.
- Why not just compute the solution this way? What is the computing cost?

- Cramer's rule is handy for very small problems of theoretical results, BUT...
- **Using Cramer's rule (or matrix inverse) to COMPUTE the solution of a linear system is only efficient for...**

- Cramer's rule is handy for very small problems of theoretical results, BUT...
- **Using Cramer's rule (or matrix inverse) to COMPUTE the solution of a linear system is only efficient for..convincing someone that you do not know much about numerical methods.**
- It is expensive and ill-conditioned.
- Whenever possible use factorization or iteration methods (coming up next)

Section 2.2 - Iterative Solution Methods for $Ax = b$

There is an entire class of iterative linear solvers.
Basic idea is a 3-step iterative scheme:

1. Make an initial estimate of the solution $x^{(0)}$.
2. Given a guess $x^{(k)}$, compute an improved estimate $x^{(k+1)}$.
3. Repeat until a stopping criterion is reached:

- Change between successive estimates is sufficiently small
- Residual error is sufficiently small
- Number of iterations reaches a specified limit

> Superscript in parentheses indicates iteration number, not an exponent.

Jacobi Iteration (simplest iterative method)

- Solve each equation/row for the corresponding solution entry
  (i.e. solve the i^{th} equation for x_i for `i in range(n)`)
- Update by plugging previous estimate into the "row-wise" solution.

The symbolic version involves the matrix D which is the diagonal part of A.
In $Ax = b$, substitute $A \to D - (D - A)$

$$(D - (D - A))x = b$$
$$Dx - (D - A)x = b$$
$$x = D^{-1}((D - A)x + b)$$

Evaluate RHS using previous iterate to get the Jacobi iteration update formula:

$$x^{(k+1)} = D^{-1}((D - A)x^{(k)} + b) = D^{-1}Dx^{(k)} + D^{-1}(b - Ax^{(k)})$$

or in terms of the **residual** $r^{(k)} = (b - Ax^{(k)})$:

$$x^{(k+1)} = x^{(k)} + D^{-1}r^{(k)}$$

Jacobi iteration formula: $x^{(k+1)} = x^{(k)} + D^{-1}(b - Ax^{(k)})$

- Requires computing a matrix inverse, but **methods based on computing inverses are inefficient and ill-behaved**.
- Does that present a serious limitation for using Jacobi iteration?
- Here the matrix to invert is diagonal, so we just have to do n divisions and we are OK provided that no diagonal element is close to zero.

Now let's see Jacobi iteration in action...

```
# Apply Jacobi iteration to solve text Eq.(2,2,4)
A1 = np.array([[4,-1,1],[4,-8,1],[-2,1,5]])
b1 = np.array([7,-21,15])
print(A1,b1)
```

A1 = [[4 -1 1]

[4 -8 1]

[-2 1 5]]

b1 = [7 -21 15]

```
#Verify that the exact solution is [2,4,3]
A1DotSol = np.dot(A1, np.array([2,4,3]))
np.allclose(A1DotSol, b1), A1DotSol, b1
```

(True, array([7, -21, 15]), array([7, -21, 15]))

Write an update function based on solving line i for entry i

```
#based on Eqs.(2.2.1)
def update_1(x,y,z):
    x_new = (7+y-z)/4.
    y_new = (-21 -4*x - z)/(-8.)
    z_new = (15+2*x-y)/5.
    return x_new,y_new,z_new
```

```
update_1(1.,2.,2.)
```

(1.75, 3.375, 3.0)

```
iters = 6 # Reproduce some results in Table 2.1
data = np.zeros([iters,4])
x,y,z =  1., 2., 2.
for k in range(iters):
    data[k,0] = k
    data[k,1] = x
    data[k,2] = y
    data[k,3] = z
    x,y,z = update_1(x,y,z)
print(data)
```

[[0. 1. 2. 2.]

[1. 1.75 3.38 3.]

[2. 1.84 3.88 3.02]

[3. 1.96 3.92 2.96]

[4. 1.99 3.98 3.]

[5. 1.99 4. 3.]

[6. 2. 4. 3.]] (2 decimal accuracy at iteration 6; notebook $\rightarrow$ 15 places at iteration 20)

Do things always work out so nicely?

- Rework the same system swapping top and bottom equations
- Results should be the same except for ordering

```
#based on Eqs.(2.2.4) with first and last equations swapped
def update_2(x,y,z):
    x_new = (y+5*z-15)/2.
    y_new = (21+4*x+z)/8.
    z_new = y-4*x+7
    return x_new,y_new,z_new
```

```
iters = 7
data = np.zeros([iters,4])
x,y,z =  1., 2., 2.
for k in range(iters):
    data[k,0] = k
    data[k,1] = x
    data[k,2] = y
    data[k,3] = z
    x,y,z = update_2(x,y,z)
print(data)
```

```
[[ 0. 1. 2. 2. ]
[ 1. -1.5 3.38 5. ]
[ 2. 6.69 2.5 16.38]
[ 3. 34.69 8.02 -17.25]
[ 4. -46.62 17.81 -123.73]
[ 5. -307.93 -36.15 211.28]]
```

If the values in this iteration blow up, why did the first case converge nicely?

- **Diagonal dominance**
  - Magnitude of each diagonal term is larger than the sum of the magnitudes of the other coefficients in the row
  - Guarantees convergence (as in the case of `update_1`)

To deal with problems outside this narrow category, a variety of embellished algorithms have been concocted.

Gauss-Seidel iteration (next simplest scheme)

Same as Jacobi iteration but use updated values as they become available. (See Eq. (2.2.9) for details.)

- Gauss-Seidel improves rate of convergence
- Can G-S prevent divergence in the non-diagonally-dominant case?

```
#based on Eqs.(2.2.4) with first and last equations swapped
def update_2gs(x,y,z):
    x_new = x + (1/5.)*(15-(-2*x+y+5*z))
    y_new = y + (-1/8.)*(-21-(4*x_new-8*y+z))
    z_new = z + (1/4.)*(7-(4*x_new-y_new+z))
    return x_new,y_new,z_new
```

```
# Compare with the results from update_2 where the first and last equations are swapped.
iters = 7
data = np.zeros([iters,4])
x,y,z =  1., 2., 2.
for k in range(iters):
    data[k,0] = k
    data[k,1] = x
    data[k,2] = y
    data[k,3] = z
    x,y,z = update_2gs(x,y,z)
print(data)
```

```
[[ 0. 1. 2. 2. ]
 [ 1. 2. 3.88 2.22]
 [ 2. 2.81 4.31 1.68]
 [ 3. 4.38 5.03 -0.11]
 [ 4. 8.24 6.73 -4.9 ]
 [ 5. 18.09 11.06 -17.25]
 [ 6. 43.37 22.15 -49.02]]
```

Does G-S iteration converge to solution?

- Not in this case; but it does diverge more slowly
- Check for diagonal dominance or see if it can be attained by permuting rows

> Resume here Tuesday 17 January 2023

Aside on Generalized Derivatives (inspired by a post by Michael Orlitzky)

More details of differentiation in Ch. 4, but let's introduce some ideas here to set up the discussion of Gradient/Steepest Descent solvers.

Start with some things everyone should have seen previously:

- Scalar function: $f : \mathbb{R} \to \mathbb{R}$
- Taylor series expansion of f about x: $f(x+h) = f(x) + f'(x)h + \dots$
- Derivative:

$$f'(x) = \frac{df}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

Derivative gives the rate of change of the function
Linear approximation of f near x is $f(x+h) \approx f(x) + f'(x)\ h$

Generalize derivative to functions with input/output vectors: $f : \mathbb{R}^m \to \mathbb{R}^n$

Linear approximation still written as $f(x+h) \approx f(x) + f'(x)\ h$
Now use multi-variable Taylor serieswith $x, h \in \mathbb{R}^m$ (a.k.a. column vectors of length m)
Each entry in f has a linear term in each element of h so we need an $n \times m$ array of partial derivatives called the ***Jacobian matrix***:

$$f(x+h) \approx f(x) + \begin{bmatrix} \frac{\partial f_0}{\partial x_0} & \frac{\partial f_0}{\partial x_1} & \dots & \frac{\partial f_0}{\partial x_{m-1}} \\ \vdots & \vdots & \dots & \vdots \\ \frac{\partial f_{n-1}}{\partial x_0} & \frac{\partial f_{n-1}}{\partial x_1} & \dots & \frac{\partial f_{n-1}}{\partial x_{m-1}} \end{bmatrix} \begin{bmatrix} h_0 \\ h_1 \\ \vdots \\ h_{m-1} \end{bmatrix} = f(x) + J(f;x)\ h$$

$\implies$ Generalized derivative: $f'(x) = J(f;x)$

Here we are interested in $f : \mathbb{R}^{\mathbb{m}} \to \mathbb{R}$, i.e. $m = 1$ (scalar function of m variables)

f has the single component f_0 so drop the n subscript

$$f(x+h) \approx f(x) + \begin{pmatrix} \frac{\partial f}{\partial x_0} & \frac{\partial f}{\partial x_1} & \cdots & \frac{\partial f}{\partial x_{m-1}} \end{pmatrix} \begin{bmatrix} h_0 \\ h_1 \\ \vdots \\ h_{m-1} \end{bmatrix}$$

$$f(x) + f'(x) \cdot h = f(x) + J(f;x)\, h \to f'(x) = (\nabla f)^T$$

Derivative of inner product: $f(x) = x\, b = < x, b > = b^T x = x^T b = \sum_{k=1}^{m} b_k x_k$

$$f'(x) = \begin{pmatrix} \frac{\partial f}{\partial x_0} & \frac{\partial f}{\partial x_1} & \cdots & \frac{\partial f}{\partial x_{m-1}} \end{pmatrix} = \begin{pmatrix} b_0 & b_1 & \cdots & b_{m-1} \end{pmatrix} = b^T$$

Derivative of a matrix times a vector: $f(x+h) \approx f(x) + f'(x)\, h$

$$f(x) = Ax \implies f(x+h) = A(x+h) = Ax + Ah \implies f'(x) = A$$

Derivative of Quadratic Form: $ f(x) = x^T A x$ where $x \in \mathbb{R}^m$ and A is an $m \times m$ matrix

Let $y = Ax$ so $f(x, y(x)) = x^T y(x) = y(x)^T x$

Total derivative:

$$\begin{aligned} f'(x, y(x)) &= \frac{\partial f}{\partial x} && + \frac{\partial f}{\partial y} y'(x) \\ &= y(x)^T && + x^T\, A \\ &= (Ax)^T && + x^T\, A = x^T A^T + x^T A = \boxed{x^T (A^T + A) = (x^T A x)'} \end{aligned}$$

> Use derivative to proceed from first discussion of iterative methods (Jacobi, G-S)

§ 2.3 - Gradient (Steepest) Descent for $Ax = b$

Alternate approach to linear systems underlying some of the more sophisticated solvers

- Bi-Conjugate Gradient Descent or **bicstab**
- generalized method of residuals or **gmres**
- construct scalar function s.t. linear system solution $\iff$ critical point of function

Critical point $\iff$ generalized derivative vanishes: $(\nabla f)^T = 0 \implies \nabla f = 0 \, (= 0^T)$
Vanishing gradient $\iff$ linear system (degree 1) $\rightarrow$ function is quadratic (degree 2).

Generic quadratic form: $f(x) = \frac{1}{2} x^T A x - b^T x + c$

has the gradient: $\nabla f(x) = \frac{1}{2} A^T x + \frac{1}{2} A x - b$

Symmetric matrix $A \;\rightarrow\; A^T = A$
Condition at the critical point becomes: $\nabla f(x) = Ax - b = 0$
Critical point of the quadratic coincides with the solution of symmetric linear system.

Critical point of the quadratic $f(x) = \frac{1}{2}x^T Ax - b^T x + c$ coincides with the solution of symmetric linear system $Ax = b$.

If A is also ***positive definite***, i.e $x^T Ax > 0 \; \forall x \in \mathbb{R}^n \setminus 0$, critical point is a minimum.

Gradient (or Steepest) Descent:
Seek solution by systematically moving "downhill" toward the minimum.

If A is not symmetric, then gradient descent seeks the solution of $\frac{1}{2}(A^T + A)x = b$.

For non-square A, can multiply by A^T to get **normal equations** $A^T Ax = A^T b$

Normal matrix A^TA is symmetric, positive definite $\rightarrow$ descent methods can be applied:

$\Longrightarrow$ **Least Squares solution** (minimixes the least squared error $||Ax-b||^2$)

> **CAUTION**: There are numerical reasons to be careful about how you use the normal equations.
> Generally avoid actually doing the matrix multiplication to compute A^TA because:
>
> - Multiplication of large matrices is much more "expensive" (in terms of computing operations) than matrix-vector multiplication.
> - Multiplying by A^T tends to increase the condition number and detract from the accuracy of the numerical solution.

Example: Compute best-fit line to following data set:

`t = [2,3,5,7,9], y=[4,5,7,10,15]`

The line is defined by an equation of the form `y = slope*t+intercept`

Gives equations of the form `t[i]*slope + intercept*1 = y[i]`

in the 2 unknowns `[slope, intercept]`.

Can write as a linear system $A_0 x = b_0$ where:

1. the first column of A_0 is t
2. the second column of A_0 is all 1s
3. the unknown vector to solve for is: `x = [slope, intercept]`
4. the right-hand side is y

```
#python example
t = np.array([2,3,5,7,9])
y = np.array([4,5,7,10,15])
A0 = np.column_stack([t,np.ones([5])])
print(A0, y)
```

(array([[2., 1.],

[3., 1.],

[5., 1.],

[7., 1.],

[9., 1.]]),

array([4, 5, 7, 10, 15]))

For such a small problem, we can multiply by A^T to convert this overconstrained, non-square system to the square **normal** system:

```
A = np.dot(A0.T,A0)
b = np.dot(A0.T,y)
print(A, b)
```

(array([[168., 26.],
[26., 5.]]),
array([263., 41.]))

Can solve with one of our earlier codes (for elimination or iteration) or make our first use of a **library function** `np.linalg.solve`

```
soln = np.linalg.solve(A, b)
slope, intercept = soln[0], soln[1]
print("Best fit line: y = %6.2f * t + %6.2f" % (slope, intercept))
```

Best fit line: y = 1.52 * t + 0.30

The normal matrix is square symmetric (by construction)

$\Longrightarrow$ should also be able to find the solution by minimizing the quadratic form:

$$f(x) = \frac{1}{2} x^T M x - b^T x \text{ where } M = A^T A, b = A^T y$$

Below is sample python code for computing the quadratic function and its gradient.
(The "manual" definition of the function, `fxy`, is a convenience to support plotting.)

A sample problem (from Kutz p. 40)

$$A = \begin{pmatrix} 3 & 2 \\ 2 & 6 \end{pmatrix}, \quad b = \begin{pmatrix} 2 \\ -8 \end{pmatrix}, \quad c = 0$$

$$f(x) = \frac{1}{2} \begin{pmatrix} x_1 & x_2 \end{pmatrix} \begin{pmatrix} 3 & 2 \\ 2 & 6 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} - \begin{pmatrix} 2 & -8 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} - 0$$

$$= \frac{3}{2} x_1^2 + 2x_1 x_2 + 3x_2^2 - 2x_1 + 8x_2$$

The gradient of this quadratic function is:

$$\nabla f(x) = \begin{pmatrix} 3x_1 + 2x_2 - 2 \\ 2x_1 + 6x_2 + 8 \end{pmatrix}$$

and the critical point condition is:

$$\nabla f(x) = \begin{pmatrix} 3x_1 + 2x_2 - 2 \\ 2x_1 + 6x_2 + 8 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

The critical point condition

$$\nabla f(x) = \begin{pmatrix} 3x_1 + 2x_2 - 2 \\ 2x_1 + 6x_2 + 8 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

is equivalent to the linear system:

$$Ax = b \iff \begin{pmatrix} 3 & 2 \\ 2 & 6 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 2 \\ -8 \end{pmatrix}$$

That sets the theoretical foundation; now let's implement a gradient descent solver.

> Note: the python version calls the variables `x,y` or `x[0],x[1]` instead of x_1, x_2.

```
def f(x,A,b):
    """
    Compute the quadratic function (1/2)x'.A.x - b'.x

    Inputs:
        x: 1D numpy array of size n
        A: 2D
    """
    y = np.dot(A,x)
    return 0.5*np.dot(x,y) - np.dot(b,x)

# This version in terms of the scalar components is a convenience for plotting...
def fxy(x,y):
    return (1.5*x**2 + 2*x*y + 3*y**2) - 2*x + 8*y

def grad_f(x):
    g0 = 3*x[0] + 2 * x[1] - 2
    g1 = 2*x[0] + 6*x[1] - 8
    return np.array([g0,g1])
```

```
# Visualize the quadratic function
# %matplotlib inline - This "magic" command is no longer necessary...
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (12,6)
x = np.linspace(0, 4, 10)
y = np.linspace(-4, 0, 10)
X, Y = np.meshgrid(x, y)
Z = fxy(X, Y)
fig = plt.figure()
ax = plt.axes(projection='3d')
ax.plot_wireframe(X, Y, Z, color='black')
ax.set_xlabel('x[0]')
ax.set_ylabel('x[1]')
ax.set_zlabel('f')
ax.set_zlim(bottom=-20,top=20)
ax.set_title('wireframe');
plt.show(fig)
```

Execute code to see the plot

Constructed quadratic whose gradient is $Ax - b \implies$ quadratic has stationary point at solution of linear system.

Positive definite matrix (which is true for $A^T A$) $\implies$ stationary point is a **minimum**

Gradient (Steepest) Descent Method

1. Make an initial guess at the solution, `x[0]`
2. For a given estimate `x[i]`
3. Compute the gradient `grad_f(x[i])` (to determine steepest direction)
4. Move downhill (toward the minimum) to obtain improved solution,
   e.g. `x[i+1] = x[i] - c *grad_f(x[i])`
   (Note: $\exists$ better ways to update estimate & move efficiently toward the minimum.)
5. Termination: If `np.norm(x[i+1]-x[i]) > tol` repeat from step 2; else `return x[i+1]`

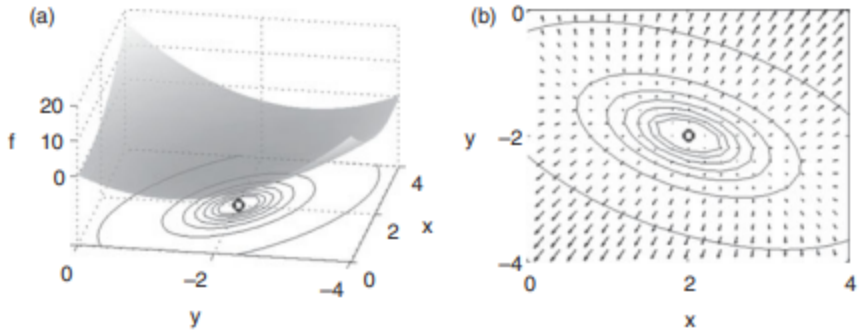


Figure 2.1: Graphical depiction of the gradient descent algorithm. The quadratic form surface $f = (3/2)x_1^2 + 2x_1x_2 + 3x_2^2 - 2x_1 + 8x_2$ is plotted along with its contour lines. In (a), the surface is plotted along with the contour plot beneath. The objective of gradient descent is to find the bottom of the paraboloid. In (b), the gradient is calculated and shown with a quiver plot.

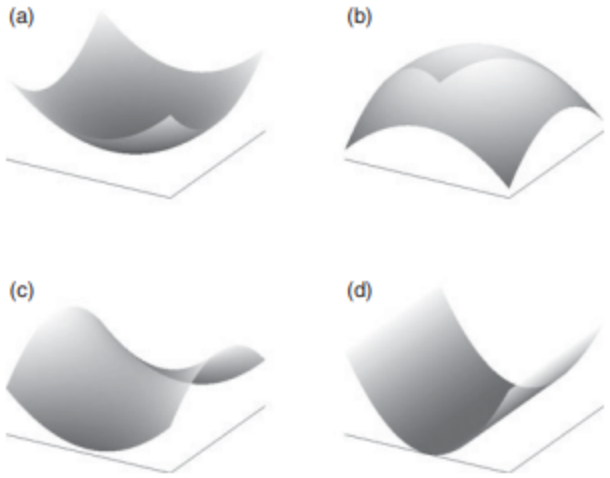


Figure 2.2: Four characteristic matrix forms: (a) positive definite, (b) negative definite, (c) saddle and (d) singular matrix. Intuition would suggest that a descent algorithm will only work for a positive definite matrix.

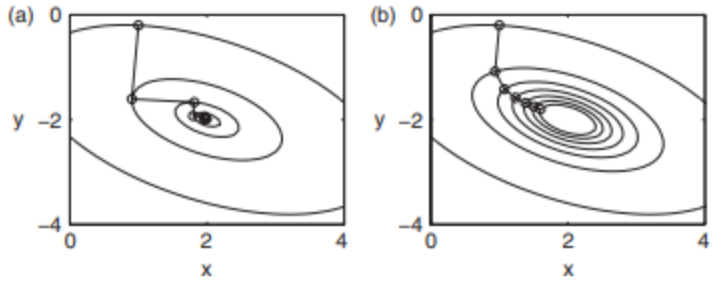


Figure 2.3: Gradient descent algorithm applied to the function (quadratic form) $f = (3/2)x_1^2 + 2x_1x_2 + 3x_2^2 - 2x_1 + 8x_2$. In both panels, the contours are plotted for each successive value (x, y) in the iteration algorithm given the initial guess $(x, y) = (1, -0.2)$. For (a), the optimal τ value is calculated so that each successive gradient in the steepest descent algorithm is orthogonal to the previous step. In (b), a prescribed value of $\tau = 0.1$ is used. In this case, the iteration still converges to the solution, but at a much slower rate since the descent steps are not chosen in an optimal way.

Return to look at the example:

```
x0 = np.array([0,0])
print(A, b, f(x0,A,b), grad_f(x0))
```

(array([[168., 26.],

[26., 5.]]),

array([263., 41.]),

0.0,

array([-2, -8]))

```
def descent_update_const(x, c):
    return x - c*grad_f(x)
```

```
# previously solved to get [1.52, 0.30]
c = 0.01
n = 20
x = x0
for i in range(n):
    x = descent_update_const(x, c)
    print(x)
```

[0.02 0.08]

[0.04 0.15]

[0.05 0.22]

[0.07 0.29]

[0.08 0.35]

[0.09 0.41]

[0.1 0.46]

[0.11 0.51]

[0.11 0.56]

[0.12 0.6] (Slooooowly converging...)

Describe performance (or lack thereof)...

What does the quadratic look like?

Implement better approach with line search:

How far should we really move along the descent direction?
Line search along `x[i]-c*grad_f(x[i])` to find `c` that minimizes `f(x[i+1])`.

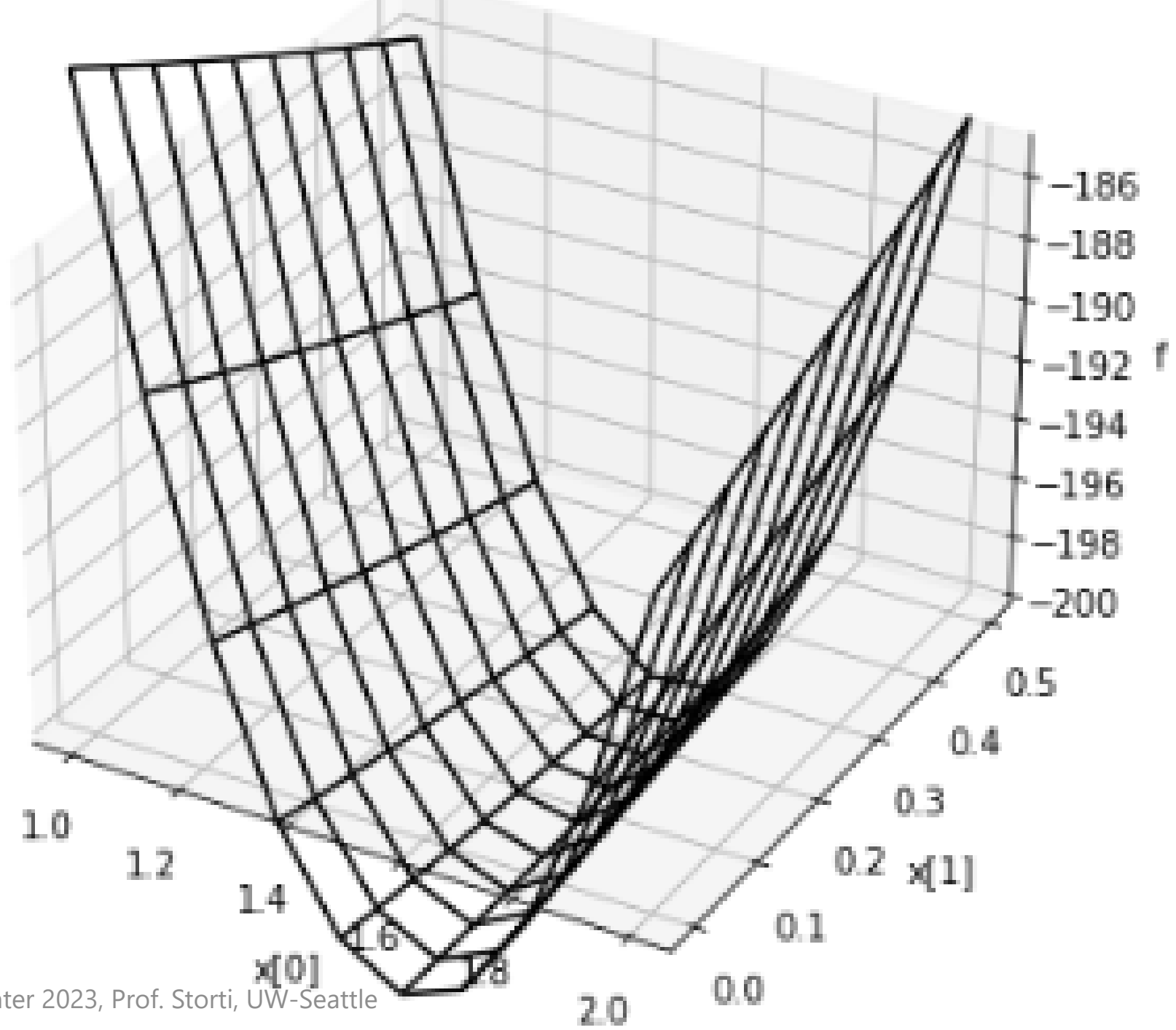
Choose more efficient search directions:
Once the quadratic function has been minimized along a direction, future searches should go in orthogonal (conjugate) directions $\rightarrow$ **conjugate gradient method**

If you are interested in learning more deatils about this topic please see **"An Introduction to the Conjugate Gradient Method Without the Agonizing Pain Edition** $1\frac{1}{4}$**"** by Jonathan Richard Shewchuk.

```
def fxy(x,y):
    return 0.5*(168*x**2+52*x*y+5*y**2)-(263*x+41*y)

def grad_f(x):
    g0 = 168*x[0] + 26 * x[1] - 263
    g1 = 26*x[0] + 5*x[1] - 41
    return np.array([g0,g1])

import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (12,6)
x = np.linspace(1,2, 10)
y = np.linspace(0,0.5, 10)
X, Y = np.meshgrid(x, y)
Z = fxy(X, Y)
fig = plt.figure()
ax = plt.axes(projection='3d')
ax.plot_wireframe(X, Y, Z, color='black')
ax.set_xlabel('x[0]')
ax.set_ylabel('x[1]')
ax.set_zlabel('f')
ax.set_zlim(bottom=-200,top=-185)
ax.set_title('wireframe');
plt.show(fig)
```


−186
−188
−190
−192
−194
−196
−198
−200
f
0.5
0.4
0.3
0.2
x[1]
0.1
0.0
1.0
1.2
1.4
x[0]
2.0

Section 2.4 - Eigenvalues, Eigenvectors, and Stability

We have not spent time discussing when a linear system is actually solvable.
How do we determine solvability?

Square systems: $\exists!$ solution if the matrix is non-singular, i.e. $Det(A) \neq 0$.

So we cannot compute solutions if A is singular.

Fredholm alternative theorem:
For solvable $Ax = b$, b must be orthogonal to the null space of the adjoint A^{*T}.

Can we effectively compute solutions when A is non-singular? Sometimes...

Introduce **Condition number** as a measure of the ratio of relative residual

$$\frac{||Ax - b||}{||b||}$$

to relative error:

$$\frac{||x - x_{exact}||}{||x_{exact}||}$$

All of these things relate to eigenvalues and eigenvectors: How?

Consider iterative computation of eigenvalues and eigenvectors.

Ratio of eigenvalues provides bounds for condition number...