LU Factorization (without Pivots)

There are a few "textbook" techniques for computing the LU factorization of an arbitrary dense square matrix.

$$A = LU.$$

They are the *Crout Method* and *Doolittle's Method*. They vary slightly in terms of which matrix receives the 1's on the diagonal and the operation count. But, effectively each method performs an inversion of the $L \times U$ matrix multiplication operation to recover *L* and *U*. Many implementations that you might find perform the factorization "in place" meaning they do not explicitly allocate new matrices for *L* and *U*, but rather return *A* with *L* in the lower triangle and *U* in the upper triangle.

These textbook methods are generally not appropriate for arbitrary applications as they will breakdown for some matrices (e.g. those that are ill-conditioned, require pivoting, or are singular). However, if you know you are working "nice" matrices that will not exhibit those features (such as in pedagogical settings like this), then these methods are perfectly adequate.

In this notebook, we provide an implementation of Doolittle's Method for computing the *LU* factorization of a matrix. This is provided through a class that will generate a random $n \times n$ matrix, factorize it, then provide an interactive visualization showing the order of the reads and writes for an "in place" factorization.

The algorithm for Doolittle's method implemented here provides *L* and *U* in the following forms:

$$L = \begin{pmatrix} 1 & & & & & \\ l_{1,0} & \ddots & & & 0 & \\ & \ddots & 1 & & & \\ \vdots & \dots & l_{j+1,j} & 1 & & \\ & & \vdots & \ddots & \ddots & \\ l_{n-1,0} & \dots & l_{n-1,j} & \dots & l_{n-1,n-2} & 1 \end{pmatrix},$$

$$U = \begin{pmatrix} u_{0,0} & u_{0,1} & \dots & u_{0,j} & \dots & u_{0,n-1} \\ & u_{1,1} & & \vdots & & \\ & & \ddots & u_{i-1,i} & & \vdots \\ & & & & \ddots & \\ & & & u_{i,i} & & \\ & 0 & & & \ddots & u_{n-2,n-1} \\ & & & & & u_{n-1,n-1} \end{pmatrix},$$

and Doolittle's algorithm is:

```
for i = 0 to n-1
    for j = 0 to i - 1
```

$$l_{i,j} = \left(a_{i,j} - \sum_{k=0}^{j-1} l_{i,k} u_{k,j} \right) \Big/ u_{j,j}$$

```
    end

    for j = i to n-1
```

$$u_{i,j} = \left(a_{i,j} - \sum_{k=0}^{i-1} l_{i,k} u_{k,j} \right)$$

```
    end
end
```

In [1]:

```
#import packages and define class (MODIFY AT YOUR OWN RISK!)
import numpy as np
import matplotlib.pyplot as plt
from ipywidgets import *
from IPython.display import clear_output

#Define a class for LU factorization and visualization of matrix accesses
class LUFactorizer:
    def __init__(self,n):
        self.n = n
        self.A = np.random.rand(self.n,self.n)
        self.LU= np.zeros((self.n,self.n))
        self.reads = np.zeros((2,int(2/3*self.n**3 + self.n**2/2 - 1/6*self.n)))
        self.writes = np.zeros((2,self.n*self.n))
        self.access = [None]*self.reads.shape[1]*self.writes.shape[1]
        self.naccess=0
        self.rcounter=0
        self.wcounter=0

    def factorize(self):
        self.LU=self.A.copy()
        for i in range(0,self.n):
            for j in range(0,i):

                self.reads[:,self.rcounter]=[i,j]; self.rcounter+=1;
                self.access[self.naccess]='r'; self.naccess+=1
                alpha = self.LU[i,j]

                for k in range(0,j):
                    self.reads[:,self.rcounter]=[i,k]; self.rcounter+=1
                    self.access[self.naccess]='r'; self.naccess+=1
                    self.reads[:,self.rcounter]=[k,j]; self.rcounter+=1
                    self.access[self.naccess]='r'; self.naccess+=1
                    alpha = alpha - self.LU[i,k]*self.LU[k,j]

                self.reads[:,self.rcounter]=[j,j]; self.rcounter+=1
                self.access[self.naccess]='r'; self.naccess+=1
                self.writes[:,self.wcounter]=[i,j]; self.wcounter+=1
                self.access[self.naccess]='w'; self.naccess+=1
                self.LU[i,j] = alpha/self.LU[j,j]
```

```
        for j in range(i,self.n):

            self.reads[:,self.rcounter]=[i,j]; self.rcounter+=1
            self.access[self.naccess]='r'; self.naccess+=1
            alpha = self.LU[i,j]

            for k in range(0,i):
                self.reads[:,self.rcounter]=[i,k]; self.rcounter+=1
                self.access[self.naccess]='r'; self.naccess+=1
                self.reads[:,self.rcounter]=[k,j]; self.rcounter+=1
                self.access[self.naccess]='r'; self.naccess+=1
                alpha = alpha - self.LU[i,k]*self.LU[k,j]

            self.writes[:,self.wcounter]=[i,j]; self.wcounter+=1
            self.access[self.naccess]='w'; self.naccess+=1
            self.LU[i,j] = alpha

def getL(self):
    L=np.zeros((self.n,self.n))
    for i in range(0,self.n):
        L[i,i]=1
        for j in range(0,i):
            L[i,j]=self.LU[i,j]
    return L

def getU(self):
    U=np.zeros((self.n,self.n))
    for i in range(0,self.n):
        for j in range(i,self.n):
            U[i,j]=self.LU[i,j]
    return U

def verify(self):
    LU=np.matmul(self.getL(),self.getU())
    return np.linalg.norm(np.subtract(LU,self.A))

def __plot(self,w):

    nwrite=self.access[0:w].count('w')
    istt=0
    istp=w-1
    if nwrite > 0:
        for i in range(w-2,-1,-1):
            if self.access[i] == 'w':
                istt=self.access[0:i+1].count('r')
                istp=istt+self.access[i:w].count('r')
                break
    else:
        istp=istt+w

    if LU.access[w-1] == 'w':
        r_marker='x'
    else:
        r_marker='o'

    self.rplot.set_data(self.reads[1,np.arange(istt,istp)]+0.5,
                        self.reads[0,np.arange(istt,istp)]+0.5)
    self.wplot.set_data(self.writes[1,np.arange(0,nwrite)]+0.5,
                        self.writes[0,np.arange(0,nwrite)]+0.5)
```

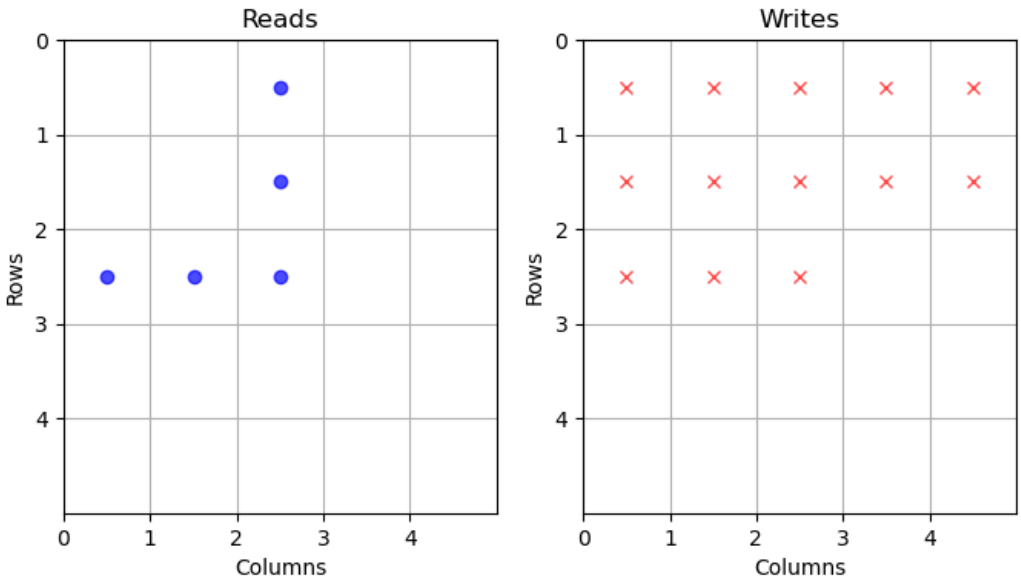
```
        self.fig.canvas.draw_idle()

    def visualize(self):
        self.fig,axs = plt.subplots(1, 2,figsize=(1600./200,800./200), dpi=100)
        # fig.subplots_adjust(wspace=0.3)
        axs[0].set_title('Reads')
        axs[1].set_title('Writes')
        axs[0].set_ylabel('Rows')
        axs[0].set_xlabel('Columns')
        axs[1].set_ylabel('Rows')
        axs[1].set_xlabel('Columns')
        axs[0].set_xlim(0,self.n)
        axs[1].set_xlim(0,self.n)
        axs[0].set_ylim(self.n,0)
        axs[1].set_ylim(self.n,0)

        axs[0].set_xticks(np.arange(0,self.n))
        axs[0].set_yticks(np.arange(0,self.n))
        axs[1].set_xticks(np.arange(0,self.n))
        axs[1].set_yticks(np.arange(0,self.n))
        axs[0].grid()
        axs[1].grid()

        self.nwrites=0
        self.last_update=0
        self.rplot,= axs[0].plot([], [], 'o',color='blue',alpha=0.7)
        self.wplot,= axs[1].plot([], [], 'x',color='red',alpha=0.7)


        interact(self.__plot,w=IntSlider(value=0,min=1,max=self.naccess,step=1));
```

Visualization of LU factorization

- Feel free to modify the input for the matrix size in the cell below.
- After running the cell use the slider to see how the algorithm accesses the data in the matrix
- The verify routine checks that the LU factorization returns the original matrix by computing $||LU - A||$.
  The resulting number should be less than the double precision tolerance $O\left(10^{-15}\right)$.

In [2]:

```
%matplotlib notebook
LU = LUFactorizer(5)
LU.factorize()
LU.visualize()
LU.verify()
```



Out[2]: 3.444376352465766e-16