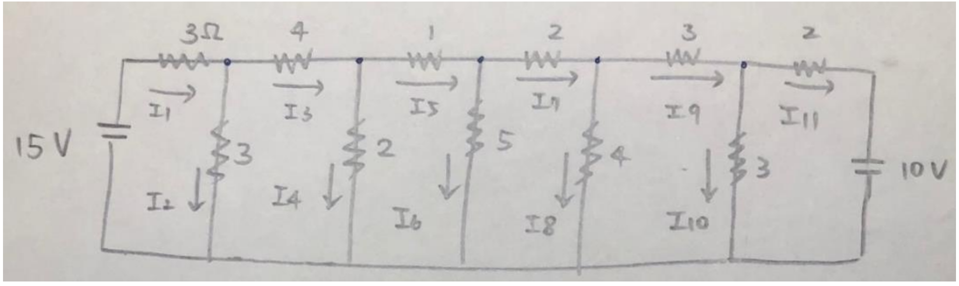
HW2

104070038 陳羿先

My circuit：



My matrix A,b for Ax = b：

1、KCL

$I_1 = I_2 + I_3$

$I_3 = I_4 + I_5$

$I_5 = I_6 + I_7$

$I_7 = I_8 + I_9$

$I_9 = I_{10} + I_{11}$

2、KVL

$3I_1 + 3I_2 = 15$

$3I_2 - 4I_3 - 2I_4 = 0$

$2I_4 - 5I_6 - I_5 = 0$

$5I_6 - 2I_7 - 4I_8 = 0$

$4I_8 - 3I_9 - 3I_{10} = 0$

$3I_{10} - 2I_{11} = 10$

Let $x^T = [I_1, I_2, \ldots, I_{11}]$

$AX = b$

$$A = \begin{bmatrix} 1 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & -1 \\ 3 & 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & -4 & -2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & -1 & -5 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 5 & -2 & -4 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 4 & -3 & -3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 & -2 \end{bmatrix}_{11\times 11} \qquad b = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 15 \\ 0 \\ 0 \\ 0 \\ 0 \\ 10 \end{bmatrix}$$

Discussion：
隨著 n*n 的矩陣變大，
1. 比較 np.linalg.det、 mydet 的時間

首先，我利用實際執行 3*3、5*5、11*11(題目所要求的 circuit)矩陣，來討論隨著 n*n 矩陣變大，兩種計算 determinent 所需的時間差異。

(1). n = 3
a. np.linalg.det

```
determinant using numpy:
-11.000000000000002
Time to solve determinent using np.linalg.det is  0:00:00.025020  seconds.
```

b. mydet

```
determinant using mydet:
-11
Time to solve determinent using mydet is  0:00:00.002001  seconds.
```

(2). n = 5
a. np.linalg.det

```
determinant using numpy:
-229.99999999999983
Time to solve determinent using np.linalg.det is  0:00:00.018013  seconds.
```

b. mydet

```
determinant using mydet:
-230
Time to solve determinent using mydet is  0:00:00.001000  seconds.
```

(3). 如題目要求在 n = 11 時，
a. np.linalg.det

```
determinant using numpy:
-78531.00000000004
Time to solve determinent using np.linalg.det is  0:00:00.018014  seconds.
```

b. mydet

```
determinant using mydet:
-78531
Time to solve determinent using mydet is  0:04:06.021025  seconds.
```

我們可以發現，當 n 值較小時，利用 numpy 內建的 np.linalg.det、與使用我們自己所寫的 mydet 來計算 determinent 所需時間相差不多，但隨著 n 值變大，mydet 所需的時間會急速上升，無法像 np.linalg.det 一樣仍保持極快的運算速度。

以我所寫的 mydet(A)來說，因為會跑一個從 i=0 到 range(n)的 for 迴圈，而迴圈又會 recursively 的呼叫 $mydet(M_{ij})$ $O(n)$ 次（ Minor matrix 從(n-1)*(n-1)到 1*1），因此這個 function 可以寫成：

$$T(n) = n \times T(n-1) + (2n-1)$$

透過演算法的運算，我們最終可得到這樣的推導：

```
T(n) = n · T(n-1) + 2n-1
     = n · (n-1) · T(n-2) + n · (n-1)
     = n · ( (n-1) · ( (n-2) · (...) + n-3 ) + n-2 ) + n-1
     = 2n-1 + n · (2(n-1)-1) + n · (n-1) · (2(n-2)-1) + ... + n!
     < 2 · n + 2 · n · (n-1) + 2 · n · (n-1) · (n-2) + ... + 2 · n! + n!
     = 2 · (n + n · (n-1) + ... + n!/2) + 3 · n!
     < 2 · (n!/(n-1)! + n!/(n-2)! + ... + n!/2!) + 3 · n!
```

又因為：

$$\frac{2 \times \mathrm{n}!}{k!} \leq \frac{n!}{(k-1)!} \; for \; all \; k \geq 2$$

```
  n!/(n-1)! + n!/(n-2)! + n!/(n-3)! + ... + n!/2!
≤ n!/(n-2)! + n!/(n-2)! + n!/(n-3)! + ... + n!/2!
≤ n!/(n-3)! + n!/(n-3)! + ... + n!/2!
≤ n!/(n-4)! + ... + n!/2!
≤ ...
≤ n!/2! + n!/2!
≤ n!
```

因此，$T(n) = n \times T(n-1) + (2n-1) = O(n!)$

Complexity of mydet 可以用 $O(n!)$ 來表示，因此當 n 值增大時，計算所需時間上升非常快，無法像 np.linalg.det 計算來的迅速。

```
det = 0
for i in range(n):
    # compute minor M(0,i)
    ### you can use np.concatenate((A,B), axis=1) to merge two matrices
    if i==0:
        M = A[1:n,1:n]
    else:
        X = A[1:n,0:i]
        Y = A[1:n,i+1:n]
        M = np.concatenate((X,Y), axis=1)
    # compute cofactor A(0,i) = (-1)^{i} det(M(0,i))

    if i%2==0:
        C = mydet(M)
    else:
        C = -mydet(M)
    det = det + A[0][i] * C
    ###   call mydet recursively to compute det(M)
```

2. 比較 np.linalg.solve、 mysolve_adj、 mysolve_cramer 的時間

(1). n = 3

a. np.linalg.solve

```
Time to solve Ax=b using np.linalg.solve is  0:00:00.246176  seconds.
rediduals of Ax=b using np.linalg.solve:
[[0.0000000e+00]
 [4.4408921e-16]
 [4.4408921e-16]]
```

b. mysolve_adj

```
Execution Time using adjoint matrix =  0:00:00.000999  seconds.

rediduals of Ax=b using adjoint matrix:
[[ 0.0000000e+00]
 [-4.4408921e-16]
 [ 0.0000000e+00]]
```

c. mysolve_cramer

```
Execution Time using Cramer's rule =  0:00:00  seconds.

rediduals of Ax=b using Cramer's rule:
[[ 1.11022302e-16]
 [ 0.00000000e+00]
 [-2.22044605e-16]]
```

(2). n = 5

a. np.linalg.solve

```
Time to solve Ax=b using np.linalg.solve is  0:00:00.001001  seconds.
rediduals of Ax=b using np.linalg.solve:
[[-2.22044605e-16]
 [ 2.22044605e-16]
 [ 0.00000000e+00]
 [ 0.00000000e+00]
 [ 0.00000000e+00]]
```

b. mysolve_adj

```
Execution Time using adjoint matrix =  0:00:00.005002  seconds.

rediduals of Ax=b using adjoint matrix:
[[-2.22044605e-16]
 [-4.44089210e-16]
 [ 0.00000000e+00]
 [ 0.00000000e+00]
 [ 0.00000000e+00]]
```

c. mysolve_cramer

```
Execution Time using Cramer's rule =  0:00:00.004000  seconds.

rediduals of Ax=b using Cramer's rule:
[[0.00000000e+00]
 [2.22044605e-16]
 [0.00000000e+00]
 [0.00000000e+00]
 [0.00000000e+00]]
```

(3). 如題目要求在 n = 11 時，

a. np.linalg.solve

```
Time to solve Ax=b using np.linalg.solve is  0:00:00.036025  seconds.
rediduals of Ax=b using np.linalg.solve:
[[ 0.00000000e+00]
 [-1.24900090e-16]
 [ 1.11022302e-16]
 [ 0.00000000e+00]
 [ 0.00000000e+00]
 [ 0.00000000e+00]
 [ 0.00000000e+00]
 [ 0.00000000e+00]
 [ 0.00000000e+00]
 [ 0.00000000e+00]
 [ 0.00000000e+00]]
```

b. mysolve_adj

```
Execution Time using adjoint matrix =  0:54:14.173832  seconds.

rediduals of Ax=b using adjoint matrix:
[[ 2.22044605e-16]
 [-5.55111512e-17]
 [-5.55111512e-17]
 [ 0.00000000e+00]
 [-4.44089210e-16]
 [ 0.00000000e+00]
 [ 8.88178420e-16]
 [ 0.00000000e+00]
 [ 0.00000000e+00]
 [ 0.00000000e+00]
 [ 0.00000000e+00]]
```

c. mysolve_cramer

```
Execution Time using Cramer's rule =  0:52:24.990416  seconds.

rediduals of Ax=b using Cramer's rule:
[[-2.22044605e-16]
 [-4.16333634e-17]
 [ 5.55111512e-17]
 [ 0.00000000e+00]
 [ 0.00000000e+00]
 [ 0.00000000e+00]
 [ 0.00000000e+00]
 [ 0.00000000e+00]
 [ 0.00000000e+00]
 [ 0.00000000e+00]
 [ 0.00000000e+00]]
```

比較 np.linalg.solve、mysolve_adj、mysolve_cramer，可以發現在 n 值小時(ex：n=3)，三種方法所需時間都差不多(1 秒內可以計算出)，但隨著 n 值增加，np.linalg.solve 的優勢便漸漸顯露，在 n=5 時可以發現其計算時間比 mysolve_adj、mysolve_cramer 快；在 n=11 時更加明顯，np.linalg.solve 仍可維持 1 秒內的計算速度，但 mysolve_adj、mysolve_cramer 需要的時間都超過了 30 分鐘。

mysolve_adj 實作方法：

以 mysolve_adj 而言，我利用了雙層 while 迴圈來得到 minor matrix M(透過 np.concatenate()合成所需的 elements)與 inverse matrix of A ： A^{-1} (命名為 adj)，而此兩者的關係式為：

$$adj[j][i] = (-1)^{i+j} \times \mathrm{mydet}(M_{ij}) \div mydet(A)$$

在得到A^{-1}後，利用$\mathrm{x} = A^{-1} \cdot b$，即可得到解 x。

因為在雙層迴圈中需要不斷呼叫 mydet，因此我認為 complexity 可寫成：

$$O(n!) \times O(n^2) = O(n^2 \times n!)。$$

```
def mysolve_adj(A, b, detA):
    # TODO 1. solve Ax=b using adjoint matrix (using mydet)
    n = A.shape[0]
    ans = np.empty((n,1))
    # M = inverse matrix of A
    if n == 2:
        adj = [[A[1][1]/detA, -1*A[0][1]/detA],
               [-1*A[1][0]/detA, A[0][0]/detA]]
    else:
        M = np.empty((n-1,n-1))
        adj = np.empty((n,n))
        i = 0

        while i<n:
            j = 0
            while j<n:
                if i==0:
                    X = A[1:n,0:j]
                    Y = A[1:n,j+1:n]
                    M = np.concatenate((X,Y), axis=1)
                elif i<n-1:
                    X = A[0:i,0:j]
                    Y = A[0:i,j+1:n]
                    N = np.concatenate((X,Y), axis=1)
                    U = A[i+1:n,0:j]
                    V = A[i+1:n,j+1:n]
                    P = np.concatenate((U,V), axis=1)
                    M = np.concatenate((N,P), axis=0)
                elif i == n-1:
                    X = A[0:n-1,0:j]
                    Y = A[0:n-1,j+1:n]
                    M = np.concatenate((X,Y), axis=1)
                sign = (-1)**(i+j)
                adj[j][i] = sign * mydet(M) / detA
                j = j + 1
            i = i + 1
    # x = A^-1 * b
    ans = np.dot(adj, b)
    return ans
```

mysolve_cramer 實作方法：

以 mysolve_cramer 而言，我透過 for 迴圈計算解 x 中每個 x_i 需要的 matrix M (把第 i 個 column 用 b 取代，可透過 np.concatenate()來合成所需的 elements)，因此存在以下關係式：

$$x_i = \mathrm{mydet(M)} \div mydet(A)$$

就可以得到我們需要的解 x。其 complexity 因為迴圈中也須不斷呼叫 mydet(A)，因此也可寫成：

$$O(n!) \times O(n) = O(n! \times n)。$$

總結來說，從 complexity 分析和實際實驗結果兩方面加總來看，計算速度快慢可表示成：

np.linalg.solve ＞ mysolve_cramer ＞ mysolve_adj

```python
def mysolve_cramer(A, b, detA):
    # TODO 2. solve Ax=b using Cramer's rule (using mydet)
    # return ?
    n = A.shape[0]
    ans = np.empty((n,1))
    for i in range(n):
        if i==0:
            X = A[0:n,1:n]
            M = np.concatenate((b,X), axis=1)
            ans[i] = mydet(M)/detA
        else:
            X = A[0:n,0:i]
            Z = np.concatenate((X,b), axis=1)
            Y = A[0:n,i+1:n]
            M = np.concatenate((Z,Y), axis=1)
            ans[i] = mydet(M)/detA

    return ans
```

Residuals 比較：

透過 np.subtract(np.dot(A, x), b)，我們將矩陣 A 乘上利用不同方法所解出的 x，並與 b 相減來得到每種方法解出來的誤差值$\|A \cdot x - b\|$。可以發現，以 n=11 為例，利用 numpy 內建的 np.linalg.solve 算出的 residuals 只有兩個 x_i 存在誤差值，且誤差值都落在10^{-16}左右，可以說是算得又快又準。

mysolve_cramer 的結果也不差，大概有三項 x_i存在誤差，誤差相減結果也是落在10^{-16}左右，雖然速度慢，但是精準度還是很夠。

mysolve_adj 的計算結果有大概 5 項 x_i有誤差，但也都是10^{-16}左右，因此我認為我實作的這兩種方法計算出來的結果仍然非常準確，就是輸在了時間，如此更可以瞭解內建 np.linalg.solve 的強大。

Bonus：

1. np.linalg.det、 np.linalg.solve 為什麼能算得這麼快?

np.linalg.det、np.linalg.solve 的運算建立在 LU 分解下，對一個 n*n 的矩陣 A 來說，我們可以將 A 分解成一個下三角矩陣 L 和一個上三角矩陣 U 的乘積，也就是 $A = LU$。因此如果適當的改變 A 的行的順序或列的順序，就可以將 A 做 LU 分解。LU 分解在本質上是 Gaussian Elimination 的一種表達形式，實質上，是將 A 通過運算變成一個上三角矩陣，那麼 A 的 transpose matrix 就是一個單位下三角矩陣，這正是所謂的 Doolittle algorithm。
以我們設計的 11*11 矩陣來說，因為大部分的 A[i][j]都為 0，因此 A 可以說是一個稀疏矩陣(代表大部分值為 0)，而 LU 分解對於階數很大的稀疏矩陣，存在特別的簡便算法，因為在 A 為稀疏矩陣的情況下，其找出的 L、U 也會是稀疏矩陣，理論上來說，此時算法的複雜度約等於非零係數的個數，而不是矩陣的大小階數，這些算法通過運用行、列的交換，使得過程中零係數因為操作而變成非零係數的次數減到最少，因此此時計算不需要完全的跑過 n*n 的矩陣(也就是如我們所寫跑過雙層迴圈)，而是簡化過的(只計算非零係數的地方)，因此時間相較於我們所設計的 mysolve_adj、mysolve_cramer 來說，當然簡化許多。
a. np.linalg.det
以 determinent 來說，因為$A = LU$ ，而 determinent of A 可寫成$\det(A) = \det(L) \times \det(U)$ ，因為 L、U 都是 triangle matrix，而 triangle matrix 的 determinent 就是對角線 element 的乘積，也就是

$$\det(A) = \det(L) \times \det(U) = \prod_{i=1}^{n} l_{ii} \times \prod_{j=1}^{n} u_{jj}$$

因此，如此計算 determinent 的方式會比直接找出所有 minor matrix 來說迅速許多。
b. np.linalg.solve
對於解出 $Ax = b$ 來說，我們利用 LU 分解，寫成$Ax = LUx = b$，要解出 x，可以進行一下步驟：
(1). 首先，解方程式 $Ly = b$ 得到 y
(2). 然後解方程式 $Ux = y$ 得到 x。
在兩次的求解中，我們遇到的都是三角矩陣，因此運用向前（向後）替代法就可以簡潔地求解，而不需要用到 Gaussian Elimination，因此也可以達到加速的效果。