python 程式設計

第 十一 講

NumPy 科學運算套件 (一)

NumPy：科學計算套件

- 提供多維陣列物件、相關衍生物件與許多快速處理陣列的特殊功用函式，可應用於數學、邏輯、排序、線性代數、統計等問題
- 提供類似於 **matlab、maple、Mathematica** 等軟體的運算功能方便於數值計算

NumPy 的核心：ndarray object

- 套件以 c 語言撰寫，幾近 c 程式的執行速度
- 陣列長度設定後即不能更動
- 陣列內的元素都是同型別
- **NumPy** 陣列使用向量式運算提供比一般 **python** 程式更簡潔與更數學式的運算方式，例如以下為傳統向量內積運算的比較：

```
# 傳統 python: a b c 為 list 物件
c = [None] * n
for i in range(n) : c[i] = a[i] * b[i]

# NumPy 套件: a b c 為 ndarray 物件
c = a * b
```

NumPy 物件

- **import numpy as np: 簡化 numpy 名稱為 np**

```
>>> import numpy as np
>>> # 一維陣列
>>> a = np.array( [2,5,2,1] )
>>> a
array([2, 5, 2, 1])

>>> # 二維陣列
>>> b = np.array( [ [2.,5,7], [3,1,6.] ] )
>>> b
array([[ 2.,  5.,  7.],
       [ 3.,  1.,  6.]])

>>> # 三維陣列
>>> c = np.array( [ [ [1,2,0],[3,4,0] ], [ [5,6,1],[7,8,2] ] ] )
>>> c
array([[[1, 2, 0],
        [3, 4, 0]],

       [[5, 6, 1],
        [7, 8, 2]]])
```

以上 a、b、c 分別為一、二、三維 ndarray 陣列物件。

np 陣列性質 (一)

- **foo.ndim: foo** 陣列分量數量
- **foo.shape: foo** 陣列各個分量資料量(分量大小)
- **foo.size: foo** 陣列元素個數
- **foo.dtype: foo** 陣列元素型別
- **foo.itemsize: foo** 元素佔用位元組

np 陣列性質 (二)

```
>>> a = np.array( [2,5,2,1] )
>>> a.ndim, a.shape, a.size
(1, (4,), 4)
>>> a.dtype, a.itemsize
(dtype('int64'), 8)

>>> b = np.array( [[2.,5,7],[3,1,6.]] )
>>> b.ndim, b.shape, b.size
(2, (2, 3), 6)
>>> b.dtype, b.itemsize
(dtype('float64'), 8)

>>> c = np.array([[[1,2,0],[3,4,0]],[[5,6,1],[7,8,2]]])
>>> c.ndim, c.shape, c.size
(3, (2, 2, 3), 12)
>>> c.dtype, c.itemsize
(dtype('int64'), 8)
```

ndarray 陣列元素型別 (一)

- 型別

代碼	型別
i	integer
u	unsigned integer
f	floating point number
c	complex
b	boolean
S	string
U	unicode

ndarray 陣列元素型別 (二)

- **dtype**: 設定元素的型別與位元組數

```
>>> np.array([1,2],dtype='f')          # float32: 32 位元浮點數
array([ 1.,  2.], dtype=float32)

>>> np.array([1,2],dtype='f8')         # 使用 8 位元組的浮點數
array([ 1.,  2.])

>>> np.array([1,2],dtype='i')          # int32: 32 位元整數
array([1, 2], dtype=int32)

>>> np.array([1,2],dtype='i8')         # int64: 64 位元整數
array([1, 2], dtype=int64)

>>> np.array([1,2],dtype='c8')         # complex64: 64 位元複數
array([ 1.+0.j,  2.+0.j], dtype=complex64)
```

可以先設定 dtype 物件後再利用

```
>>> dt = np.dtype('i4')
>>> np.array([1,2],dt)
array([1, 2], dtype=int32)
```

ndarray 陣列元素型別 (三)

- **np.nan: 非數字 (not a number mathematically)**
  如 0/0
- **np.inf: 無限大, 如 1/0**

特殊類型物件 (一)

- **np.empty([2,3]):**
  產生 (2,3) 陣列，元素值不設定，預設為浮點數
- **np.ones([2,3] , dtype=int):**
  產生 (2,3) 陣列內存整數 1
- **np.zeros([2,3]):** 同上，但是存浮點數 0.
- **np.ones_like(foo):**
  使用與 foo 同樣的分量大小，但改存 1.
- **np.zeros_like(foo):**
  使用與 foo 同樣的分量大小，但改存 0

特殊類型物件 (二)

```
>>> np.empty([2,2])
array([[  6.92546258e-310,  1.93471601e-316],
       [  0.00000000e+000,  6.92545259e-310]])

>>> np.zeros([2,3])
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])

>>> a = np.zeros([2,3])
>>> np.ones_like(a,dtype=int)
array([[ 1,  1,  1],
       [ 1,  1,  1]])
```

特殊類型物件 (三)

- **np.reshape()**：在元素不變情況下，重新調整陣列各分量個數

```
>>> a = np.ones( [2,3] )
>>> a
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])

>>> a.reshape(3,2)
array([[ 1.,  1.],
       [ 1.,  1.],
       [ 1.,  1.]])
```

特殊類型物件 (四)

- **np.empty(a.shape)：** 設定新陣列與 a 陣列有著相同的分量大小

```
>>> a = np.array([2,3])

>>> b = np.empty(a.shape)                          # b 與 a 同樣 shape
>>> b
array([  1.48219694e-323,  1.97626258e-323])

>>> c = np.array([[2,3],[3,4]])
>>> d = np.empty(c.shape)
>>> d
array([[  6.90866156e-310,  1.84353125e-316],
       [  0.00000000e+000,  4.94065646e-324]])
```

❖ 若使用 np.empty(a)，則新陣列的分量大小是用 a 陣列數值來設定

```
>>> a = np.array([2,3])
>>> b = np.empty(a)
>>> b
array([[  6.90866156e-310,  6.90866156e-310,  2.03493229e-316],
       [  0.00000000e+000,  1.84353205e-316,  2.04903885e-316]])
```

陣列複製 (一)

- 指定方式：陣列多一個名稱，元素沒有複製

```
>>> a = np.array([1,2,3])
>>> b = a                          # b 與 a 佔用同個空間
>>> b[0]= 9                        # 改變 b[0] 等於改變 a[0]
>>> a
array([9, 2, 3])
```

陣列複製 (二)

- **foo.copy()：複製 foo 元素成新陣列**

```
>>> a = np.array([1,2,3])
>>> b = a.copy()
>>> b
array([1, 2, 3])
>>> b[0] = 9                          # 更改 b[0] 不會影響 a
>>> a
array([1, 2, 3])
```

❖ 也可使用 np.copy(foo) 或 np.array(foo,copy=True)

```
>>> a = np.array([1,2,3])
>>> b = np.copy(a)                    # 複製 a 元素成新陣列 b
>>> c = np.array(a,copy=True)         # 複製 a 元素成新陣列 c
```

設定物件初值 (一)

- 使用 **list comprehension**
  - **np.array()**

```
>>> np.array( [ x for x in range(10) ] )
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

>>> a = np.array( [ [x,y] for x in range(3) for y in range(2) ] )
>>> a
array([[0, 0],
       [0, 1],
       [1, 0],
       [1, 1],
       [2, 0],
       [2, 1]])
>>> a.shape
(6, 2)
```

設定物件初值 (二)

- 使用 **np.arange()**
  - **np.arange(a)**：產生 **[0,1,. . .,a-1]** 等數字
  - **np.arange(a,b)**：產生 **[a,a+1,. . .,b-1]** 等數字
  - **np.arange(a,b,d)**：
    產生 **{a, a+d, a+2d, . . . }** 等數字。
    d 為正，末尾數比 **b** 小。**d** 為負，則末尾數比 **b** 大。

```
>>> a = np.arange(3)       # a = np.array([0, 1, 2])     數字為 int
>>> b = np.arange(3.)      # b = np.array([0., 1., 2.]) 數字為 fpn
>>> c = np.arange(1,5,2.) # c = np.array([1.,3.])       數字為 fpn
>>> d = np.arange(5,1,-2) # d = np.array([5,3])         數字為 int
```

設定物件初值 (三)

- 使用 np.linspace()
  - np.linspace(a,b,n):
    在 [a,b] 之間產生 n 個等差浮點數，包含兩端點
  - np.linspace(a,b,n,endpoint=Flase):
    endpoint 為假，則不含末尾點
  - np.linspace(a,b,n,retstep=True):
    retstep 為真，則同時回傳等差數

```
>>> np.linspace(1,2,5)
array([ 1.  ,  1.25,  1.5 ,  1.75,  2.  ])
>>> np.linspace(1,2,5,endpoint=False)
array([ 1. ,  1.2,  1.4,  1.6,  1.8])
>>> foo , dx = np.linspace(1,2,5,endpoint=False,retstep=True)
>>> foo
array([ 1. ,  1.2,  1.4,  1.6,  1.8])
>>> dx
0.2
```

設定物件初值 (四)

- 使用 np.fromfunction()
  - np.fromfunction(fn,foo):
    利用 fn 函式作用於 foo 陣列來產生數據 。fn 可為 lambda 函式或一般函式，fn 的參數為 foo 下標

```
>>> def  f(i,j) : return i+j

# 設定為 (2,3) 分量大小，將下標一一傳入 fn 計算得到數值
>>> np.fromfunction(f,(2,3))
array([[ 0.,  1.,  2.],
       [ 1.,  2.,  3.]])

# 同上
>>> np.fromfunction( lambda i,j:i+j , (2,3) )
array([[ 0.,  1.,  2.],
       [ 1.,  2.,  3.]])

一維陣列
>>> np.fromfunction( lambda i : i*i , (4,) )
array([ 0.,  1.,  4.,  9.])
```

下標截取低維陣列 (一)

- 二維陣列：使用下標取得一維列陣列

```
>>> a = np.array([[1,2,3],[4,5,6],[7,8,9]])
>>> a[0]                                          # 第一列
array([1, 2, 3])

>>> a[2]                                          # 第三列
array([7, 8, 9])
```

下標截取低維陣列 (二)

- 三維陣列：使用下標取得一或二維列陣列

```
>>> a = np.array([[[1,2],[3,4]],[[5,6],[7,8]]])
>>> a
array([[[1, 2],
        [3, 4]],

       [[5, 6],
        [7, 8]]])

>>> a[0]
array([[1, 2],
       [3, 4]])

>>> a[0,1]
array([3, 4])
```

下標截取陣列元素 (一)

- 一維陣列
  - **foo[a:b:c]:** 與字串下標範圍設定方式相同

```
>>> a = np.linspace(1,6,6,dtype=int)    # a = array([1,2,3,4,5,6])
>>> a[1:4:2]
array([2,4])
>>> a[3::-2]
array([4,2])
>>> a[::-1]
array([6,5,4,3,2,1])
```

- 下標截取元素並非產生新元素，只是重新連結到原有元素，更改新陣列值等同更改原陣列

```
>>> a = np.linspace(1,4,4,dtype=int)
>>> b = a[::-1]
>>> b
array([4,3,2,1])
>>> b[0] = 9                            # b = array([9,3,2,1])
>>> a
array([1, 2, 3, 9])
```

下標截取陣列元素 (二)

- 二維陣列
  - 使用冒號代表某分量所有資料，逗點區分各分量

```
>>> a = np.linspace(1,12,12,dtype=int)
        .reshape(4,3)
>>> a
array([[ 1,  2,  3],  # [1,2,3]     --> a[0,:]
       [ 4,  5,  6],  # [4,5,6]     --> a[1,:]
       [ 7,  8,  9],  # [7,8,9]     --> a[2,:]
       [10, 11, 12]]) # [10,11,12] --> a[3,:]

>>> a[2,:]
array([7, 8, 9])

>>> a[:,1]
array([2, 5 ,8 ,11])

>>> a[2:]              # row 2 之後所有列
array([[ 7,  8,  9],
       [10, 11, 12]])
```

```
>>> a[2:4]  # row [2,4) 所有列
array([[ 7,  8,  9],
       [10, 11, 12]])

>>> a[:,:2] # col [0,2) 所有行
array([[ 1,  2],
       [ 4,  5],
       [ 7,  8],
       [10, 11]])

>>> a[1:-1,1:]
array([[5, 6],
       [8, 9]])

>>> a[1::2,::2]
array([[ 4,  6],
       [10, 12]])
```

下標截取陣列元素 (三)

- 多維陣列
  - 使用 . . . 代表相鄰的冒號

```
>>> a = np.linspace(1,12,12,dtype=int).reshape(2,2,3)
>>> a
array([[[ 1,  2,  3],
        [ 4,  5,  6]],

       [[ 7,  8,  9],
        [10, 11, 12]]])

>>> a[0,...]                                  # 等同 a[0,:,:]
array([[1, 2, 3],
       [4, 5, 6]])

>>> a[...,1]
array([[ 2,  5],                               # 等同 a[:,:,1]
       [ 8, 11]])
```

下標截取陣列元素 (四)

- 下標截取方式可用在設定

```
>>> a = np.linspace(1,6,6,dtype=int).reshape(3,2)
>>> a
array([[1, 2],
       [3, 4],
       [5, 6]])

>>> a[:,1] = 0
>>> a
array([[1, 0],
       [3, 0],
       [5, 0]])

>>> a[1,:] = [2]*2
>>> a
array([[1, 0],
       [2, 2],
       [5, 0]])
```

下標截取陣列元素 (五)

- 下標陣列：截取/設定元素
  - 一維陣列

```
>>> a = np.linspace(1,10,10,dtype=int)
>>> ia = np.array( [ 2*i for i in range(5)] )          # 下標陣列
>>> ia
array([0, 2, 4, 6, 8])

>>> a[ia]
array([1, 3, 5, 7, 9])

>>> a[ia] = 0

>>> a
array([ 0,  2,  0,  4,  0,  6,  0,  8,  0, 10])

以上的 a[ia] 等同 array( [ a[ia[0]] , a[ia[1]] , ... , a[ia[-1]] ] )
```

下標截取陣列元素 (六)

- 二維陣列

```
>>> a = np.linspace(1,8,8,dtype=int).reshape(2,4)
>>> ia = np.array( [ [1,0] , [1,1] ] )
>>> ja = np.array( [ [0,3] , [2,1] ] )
>>> a
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])

>>> a[ia,ib]
array([[5, 4],
       [7, 6]])

>>> a[ia,ib] = 0
>>> a
array([[1, 2, 3, 0],
       [0, 0, 0, 8]])
```

下標截取陣列元素 (七)

- 布林陣列：截取/設定元素
  - 布林陣列可截取對應位置為 **True** 的元素

```
>>> a = np.linspace(-2,2,5,dtype=int)
>>> a
array([-2, -1,  0,  1,  2])

>>> b = a < 0
>>> b
array([ True,  True, False, False, False], dtype=bool)

>>> a[b] = 0
>>> a
array([0, 0, 0, 1, 2])

>>> a[a>0] = 10
>>> a
array([0, 0, 0, 10, 10])
```

內積與外積 (一)

- **foo.dot(bar)**：求陣列內積
- **np.dot(foo,bar)**：同上

```
>>> a = np.array([1,2,3])
>>> b = np.array([2,2,1])
>>> a.dot(b)
9
>>> np.dot(a,b)
9
```

內積與外積 (二)

- **np.cross(foo,bar)：** 求陣列外積

```
>>> a = np.array([1,2,3])
>>> b = np.array([2,2,1])
>>> np.cross(a,b)
array([-4,  5, -2])
```

❖ 二維陣列內積

```
>>> a = np.array( [[1],[2],[3]] )
>>> b = np.array( [[2],[2],[1]] )
>>> a.dot(b)                              # 錯誤，無法執行

>>> np.dot(a.ravel(),a.ravel())
9
```

二維陣列

- 二維陣列：兩一維向量乘積
  - np.outer(a,b)：a 與 b 兩個一維向量相乘為二維陣列 c ，$c_{ij} = a_i b_j \ \forall \ ij$

```
>>> a = np.array([1,2,3])
>>> b = np.array([4,5,3])
>>> np.outer(a,b)
array([[ 4,  5, 3],
       [ 8, 10, 6],
       [12, 15, 9]])
```

❖ 此種乘積型式如同數學上的行向量乘以列向量

向量化運算 (一)

- 向量化運算: **vectorization**
  - **universal functions:**
    可直接接受 **ndarray** 陣列計算後產生陣列

```
>>> x = np.linspace(0,3,4)
>>> x
array([ 0.,  1.,  2.,  3.])

>>> a = x + 1
>>> a
array([ 1.,  2.,  3.,  4.])

>>> b = x**2 + 1
>>> b
array([  1.,  2.,  5.,  10.])
```

```
>>> c = x > 2
>>> c
array([False, False, False, True],
      dtype=bool)

>>> d = np.sin(x)
>>> d
array([ 0.        ,  0.84147098,
        0.90929743,  0.14112001])

>>> # 計算兩陣列各分量的乘積
>>> e = x * (x+1)
>>> e
array([  0.,  2.,  6.,  12.])
```

向量化運算 (二)

- **universal functions:**

add (+)	subtract (-)	multiply (*)	divide (/)
remainder (%)	power (**)	arccos	arccosh
arcsin	arcsinh	arctan	arctanh
cos	cosh	tan	tanh
log10	sin	sinh	sqrt
abs	fabs	floor	ceil
fmod	exp	log	conjugate
max	min		
greater (>)	greater_equal(>=)	equal(==)	
less (<)	less_equal (<=)	not_equal (!=)	
logical_or	logical_xor	logical_not	logical_and
bitwise_or (|)	bitwise_xor (^)	bitwise_not (~)	bitwise_and (&)
>>	<<		

向量化運算 (三)

```
>>> x = np.arange(3)        # x = array([0,1,2])

>>> y = np.add( x , 2 )     # y = array([2,3,4])
                            # 也可寫為: y = x + 2

>>> z = np.sqrt( x )        # z = array([1.4142, 1.732, 2.])

>>> a = np.greater(x,1)     # a = array([False,False,True])
                            # 也可寫為: a = x > 1

>>> b = x << 2              # b = array([0,4,8])

>>> c = np.log10(x)         # c = array([-inf, 0., 0.30103])
```

變更陣列分量大小 (一)

- **foo.flatten()**：將多維陣列 **foo** 打散成新的一維陣列
- **foo.ravel()**：將多維陣列 **foo** 看成一維陣列

```
>>> a = np.array( [[1,2],[3,4]] )
>>> b = a.flatten()                  # b = array([1,2,3,4])
                                     # a 與 b 無關

>>> c = a.ravel()                    # c = array([1,2,3,4])
>>> c[2] = 9                         # c 與 a 佔用同空間
>>> a                                # 更改 c 等同更改 a
array([[1, 2],
       [9, 4]])
```

變更陣列分量大小 (二)

- **foo.resize()**：重新調整 **foo** 陣列各分量，調整後資料量不變

```
>>> a = np.linspace(1,6,6) # a = array( [1.,2.,3.,4.,5.,6.] )
>>> a.resize(2,3)          # a = array( [ [1.,2.,3.], [4.,5.,6.] ] )
>>> a.resize(3,2)          # a = array( [ [1.,2.],[3.,4.],[5.,6.] ] )
>>> a.resize(3,2,1)        # a = array( [ [[1.],[2.]],[[3.],[4.]],[[5.],[6.]] ] )
>>> a.resize(4,2)          # 錯誤，資料數量不對
>>> b = a.resize(2,3)      # 無效的指定，np.resize() 不回傳物件
```

- **foo.reshape()**：調整 **foo** 陣列分量大小回傳物件，原有物件分量大小不變，回傳原來物件

```
>>> a = np.linspace(1,6,6) # a = array( [1.,2.,3.,4.,5.,6.] )
>>> b = a.reshape(2,3)     # b = array( [ [1.,2.,3.], [4.,5.,6.] ] )
>>> b[1,2] = 0             # 更改 b 末尾元素資料
>>> a                      # a b 佔相同空間，a 末尾元素也受影響
array([ 1.,  2.,  3.,  4.,  5.,  0.])
```

以上的 b 物件可直接寫成：

```
>>> b = np.linspace(1,6,6).reshape(2,3)
```

變更陣列分量大小 (三)

- **foo.transpose()**：回傳 **foo** 陣列的轉制陣列，但原有物件分量不更動，轉制矩陣與原物件資料分享相同空間
- **foo.T** ：同上，為 **transpose()** 的簡寫

```
>>> a = np.linspace(1,6,6).reshape(3,2) # a 為 3x2 矩陣
>>> b = a.transpose()                   # b 為 2x3 矩陣
>>> b
array([[ 1.,  3.,  5.],
       [ 2.,  4.,  6.]])
>>> b[1,0] = 0                          # 改變 b[1,0] 為 0
>>> a                                   # a 在同樣位置資料也變為 0
array([[ 1.,  0.],
       [ 3.,  4.],
       [ 5.,  6.]])
```

變更陣列分量大小 (四)

❖ 可利用轉制陣列取得二維陣列的直行陣列：

```
>>> c = np.array( [[1,2],[3,4],[5,6]] )
>>> c.T[0]
array([1, 3, 5])

>>> c.T[1]
array([2, 4, 6])
```

❖ 一維陣列物件的 transpose() 不改變任何資料

```
>>> d = np.linspace(1,6,6)
>>> d == d.T
array([ True,  True,  True,  True,  True,  True], dtype=bool)
```

變更陣列分量大小 (五)

- 列向量 × 行向量

```
>>> a = np.array([1,2,3])
>>> a
array([1,2,3])
>>> a.T
array([1,2,3])
>>> a.dot(a.T)
14
```

- 行向量 × 列向量

```
>>> a = np.array([1,2,3]).reshape(3,1)
>>> a
array([[1],
       [2],
       [3]])
>>> b = a.T
>>> b
array([[1, 2, 3]])
>>> a.dot(b)
array([[1, 2, 3],
       [2, 4, 6],
       [3, 6, 9]])
```

陣列合併 (一)

- np.vstack((a,b,c)): 垂直方式合併 a b c 陣列成新陣列
- np.hstack((a,b,c)): 水平方式合併 a b c 陣列成新陣列

```
>>> a = np.array([0]*3)
>>> b = np.array([[1]*3,[2]*3])
>>> c = np.array([3]*3)

>>> d = np.vstack((a,b,c))
>>> d
array([[0, 0, 0],
       [1, 1, 1],
       [2, 2, 2],
       [3, 3, 3]])

>>> e = np.hstack((a,c))
>>> e
array([0, 0, 0, 3, 3, 3])
```

陣列合併 (二)

- **np.dstack((a,b,c))**：取用 a b c 陣列同位置的元素合併成新陣列

```
>>> a = np.array([1,2,3])
>>> b = np.array([4,5,6])
>>> c = np.array([7,8,9])

>>> d = np.dstack((a,b))
>>> d
array([[[1, 4],
        [2, 5],
        [3, 6]]])
>>> d.shape
(1, 3, 2)

>>> e = np.array((a,b,c))
>>> e
array([[[1, 4, 7],
        [2, 5, 8],
        [3, 6, 9]]])
>>> e.shape
(1, 3, 3)
```

dstack 也可以用在多維陣列：

```
>>> a = np.array([[1,2],[3,4]])
>>> b = np.array([[5,6],[7,8]])
>>> c = np.dstack((a,b))
>>> c
array([[[1, 5],
        [2, 6]],

       [[3, 7],
        [4, 8]]])
>>> c.shape
(2, 2, 2)
```

陣列分解 (一)

- 分解列：在同個空間上取得列陣列

```
>>> a = np.array( [[1,2],[3,4],[5,6]] )
>>> r1 , r2 , r3 = a
>>> r1 , r3 , r3
(array([1, 2]), array([3, 4]), array([5, 6]))

>>> r1[0] = 9                    # 更改 r1[0] 影響 a[0,0]
>>> a
array([[9, 2],
       [3, 4],
       [5, 6]])
```

也可以使用星號分解：

```
>>> a = np.array( [[1,2],[3,4],[5,6]] )
>>> r1 , *r2 = a
>>> r1
array([1, 2])
>>> r2
[array([3, 4]), array([5, 6])]
```

陣列分解 (二)

- 分解行：在同個空間上取得行陣列

```
>>> a = np.array( [[1,2],[3,4],[5,6]] )
>>> c1 , c2 = a.T
>>> c1
array([1, 3, 5])
>>> c2
array([2, 4, 6])
```

- **np.hsplit(foo,n)**：在同個空間上將 **foo** 陣列水平切割成 **n** 個陣列

```
>>> a = np.linspace(1,6,6)
>>> b = np.hsplit(a,3)
>>> b
[array([ 1.,  2.]), array([ 3.,  4.]), array([ 5.,  6.])]
>>> a[0] = np.inf
>>> b
[array([ inf,  2.]), array([ 3.,  4.]), array([ 5.,  6.])]
```

陣列分解 (二)

- **np.hsplit(foo,index):** 設定一組 **index** 下標切割點分離 **foo**

```
>>> a = np.linspace(1,6,6)
>>> b = np.hsplit(a,(2,3,5))
[array([ 1.,  2.]), array([ 3.]), array([ 4.,  5.]), array([ 6.])]

>>> b[0][1] = 10
>>> b
[array([  1.,  10.]), array([ 3.]), array([ 4.,  5.]), array([ 6.])]
>>> a
array([  1.,  10.,  3.,  4.,  5.,  6.])
```

以上切割 a 步驟等同以下設定方式:

```
>>> b = [ a[:2] , a[2:3] , a[3:5] , a[5:] ]
```

陣列連接

- np.append(foo,bar)：連接 foo 與 bar 陣列成新的一維陣列

```
>>> a = np.array([1,2])
>>> b = np.array([3,4])
>>> c = np.append(a,b)
>>> c
array([1, 2, 3, 4])
>>> a[1] = 9
>>> c
array([1, 2, 3, 4])
```

❖ 如果 foo 與 bar 為多維陣列，則會先將個別陣列展開成一維陣列後連接

```
>>> a = np.array([[[1,2],[3,4],[0,0]]])
>>> b = np.array([[[5,6],[7,8]]])
>>> np.append(a,b)
array([1, 2, 3, 4, 0, 0, 5, 6, 7, 8])
```

網格座標點

- **np.meshgrid(xs,ys)**：將 **X** 軸與 **Y** 軸的區間點 **xs** 與 **ys** 合併產生網格點

```
>>> xs = np.linspace(0,2,3)                    # xs = array([0.,1.,2.])
>>> ys = np.linspace(5,6,2)                    # ys = array([5.,6.])

>>> ptx , pty = np.meshgrid(xs,ys)
>>> ptx
array([[ 0.,  1.,  2.],
       [ 0.,  1.,  2.]])

>>> pty
array([[ 5.,  5.,  5.],
       [ 6.,  6.,  6.]])

>>> # zip 合併分量取得六個座標點
>>> pts = list( zip( ptx.flatten() , pty.flatten() ) )
>>> pts
[(0.0, 5.0), (1.0, 5.0), (2.0, 5.0), (0.0, 6.0), (1.0, 6.0), (2.0, 6.0)]
```

❖ np.meshgrid 也可產生二維以上座標點，使用方式與上相似。

最大與最小值 (一)

- **foo.max(), foo.min():** 取得陣列最大值與最小值
- **foo.argmax(), foo.argmin():**
  將陣列元素一列排開，取得最大值與最小值的下標位置

```
>>> a = np.array([[0,2,7],[-1,2,5]])
>>> a.max() , a.min()
(7, -1)

>>> a.argmax() , a.argmin()                # 元素一列排開 7 的下標為 2，
(2, 3)                                     -1 的下標為 3

可由 argmax() 與 argmin() 反求 max() 與 min()

  >>> a = np.array([[0,2,7],[-1,2,5]])
  >>> a.ravel()[a.argmax()]               # 等同 a.max()
  7
  >>> a.ravel()[a.argmin()]               # 等同 a.min()
  -1
```

最大與最小值 (二)

- **foo.max(axis=n), foo.min(axis=n):**
  取得各分量的最大與最小值
- **foo.argmax(axis=n), foo.argmin(axis=n):**
  取得各分量的最大與最小值的下標位置

```
>>> a = np.array([[0,1,7],[2,5,3]])
>>> a.max(axis=0)                    # 第一分量的最大值
array([2, 5, 7])

>>> a.max(axis=1)                    # 第二分量的最大值
array([7, 5])

>>> a.min(axis=1)                    # 第二分量的最小值
array([0, 2])

>>> a.argmax(axis=0)                 # [2, 5, 7] 在第二分量的下標位置
array([1, 1, 0])

>>> a.argmax(axis=1)                 # [7, 5] 在第一分量的下標位置
array([2, 1])
```

最大與最小值 (三)

- **np.fmin(a,b)/np.fmax(a,b):**
  取得 **a b** 兩陣列各元素的最小/最大值，忽略 **nan**
- **np.minimum(a,b)/np.maximum(a,b):**
  同上，但直接取用 **nan**

```
>>> a = np.array([3,5,9,1])
>>> b = np.array([8,1,np.nan,4])

>>> np.fmin(a,b)
array([ 3.,  1.,  9.,  1.])
>>> np.minimum(a,b)
array([  3.,  1.,  nan,  1.])

>>> np.fmax(a,b)
array([ 8.,  5.,  9.,  4.])
>>> np.maximum(a,b)
array([  8.,  5.,  nan,  4.])
```

相鄰元素差值

- **np.diff(foo)**：回傳 **foo** 陣列的相鄰元素間差值陣列

```
>>> a = np.array([2,3,8,7])
>>> b = np.diff(a)
>>> b
array([ 1,  5, -1])
```

- **np.diff(foo,axis=n)**：若為多維陣列，可針對第 **n** 個分量計算差值，預設為最後一個分量

```
>>> a = np.array([[2,3,8,7],[9,3,5,9]])
>>> np.diff(a)
array([[ 1,  5, -1],
       [-6,  2,  4]])

>>> np.diff(a,axis=0)
array([[ 7,  0, -3,  2]])
```

❖ np.diff 回傳的陣列與原陣列維度一樣

函式、一次微分、二次微分圖形 (一)

- 函式、一次微分、二次微分圖形：$f(x) = \frac{\cos(\pi x)^2}{\sqrt{x^2+1}}$

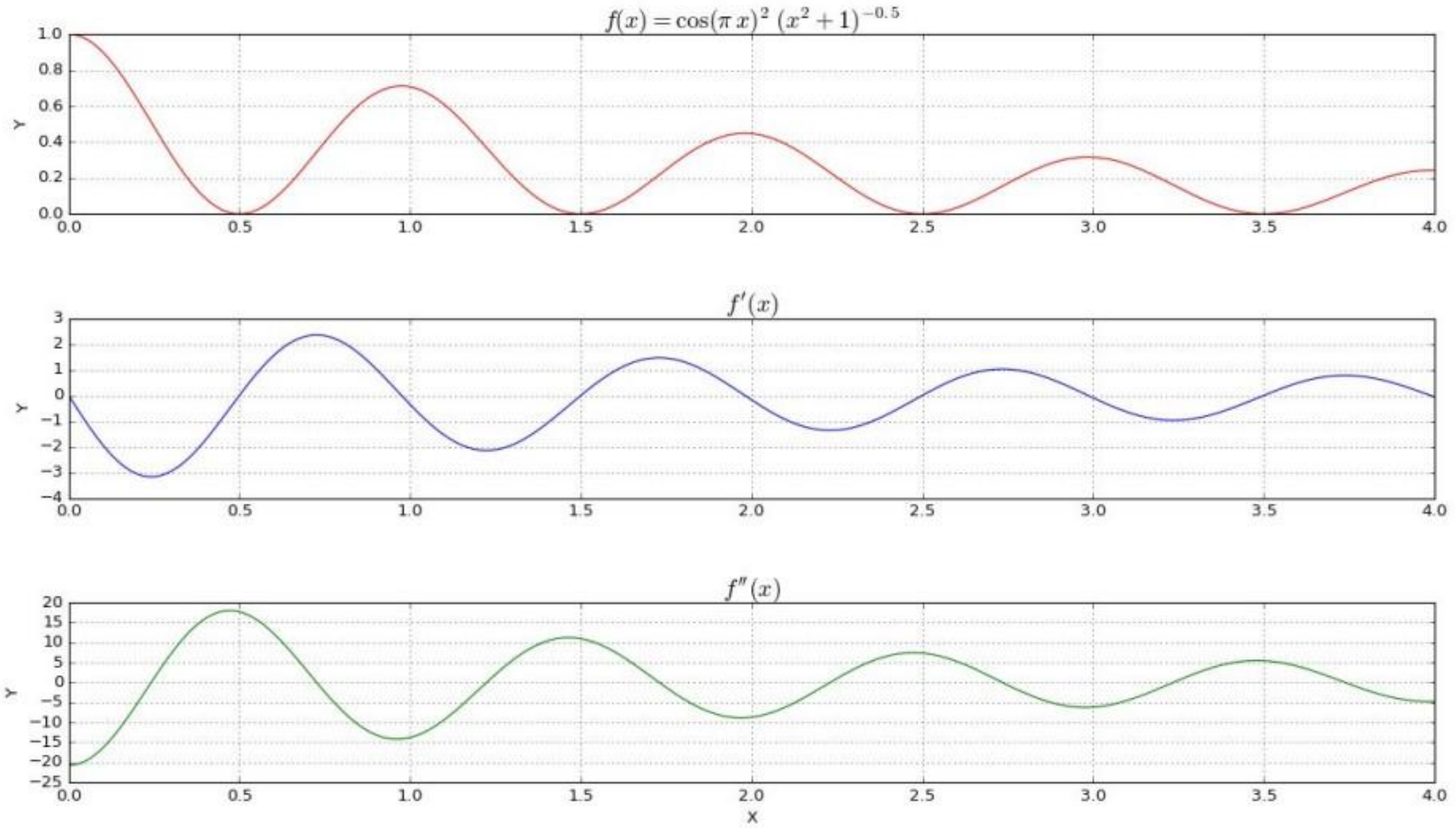
本題利用 **numpy** 的向量式運算，直接計算函式與一二次微分值。由於微分公式複雜，為省卻手算推導過程，程式中直接使用中間差分法估算一次與二次微分，

$$f'(x) \approx \frac{f(x+h)-f(x-h)}{2h}$$

$$f''(x) \approx \frac{f'\left(x+\frac{h}{2}\right)-f'\left(x-\frac{h}{2}\right)}{h}$$

為了顯示由上而下共三個函式的圖形，程式使用了 **subplot("abc")** 函式用來設定子圖形排列方式，以此為例，垂直方向、水平方向各有 **a** 與 **b** 個子圖形， **c** 是由上到下的編號，由 **1** 開始。

函式、一次微分、二次微分圖形 (二)



函式、一次微分、二次微分圖形 (三)

```
import matplotlib.pyplot as plt
import numpy as np

# 函式: cos(pi*x)^2 / sqrt(x^2+1)
def f( x ) :
    return np.cos(np.pi*x)**2/(x**2+1)**0.5

# 一次導數: 使用函式的中間差分
def df( x , h = 1.e-7 ) :
    return ( f(x+h) - f(x-h) )/(2*h)

# 二次導數: 使用一次導數的中間差分
def df2( x , h = 1.e-7 ) :
    return ( df(x+h/2) - df(x-h/2) ) / h
```

```
# 設定圖形周邊區域為 白色
fig = plt.figure(facecolor='white')

a , b , dx = 0 , 4 , 0.01

# 點數量
n = (b-a)//dx

# x 座標點
xs = np.linspace(a,b,n+1)

# 向量處理: 函式, 一次導數, 二次導數
fs = f(xs)
dfs = df(xs)
df2s = df2(xs)
```

函式、一次微分、二次微分圖形 (四)

```
# 311 圖形: 直向為 3 , 橫向為 1 , 第 1 個子圖
fig1 = plt.subplot("311")
fig1.plot(xs,fs,'r')
fig1.xaxis.grid()
fig1.yaxis.grid()
fig1.set_ylabel('Y')
fig1.set_title(r"$f(x) = \cos(\pi\,x)^2\;(x^2+1)^{-0.5}$",fontsize=18)

# 312 圖形: 直向為 3 , 橫向為 1 , 第 2 個子圖
fig2 = plt.subplot("312")
fig2.plot(xs,dfs,'b')
fig2.xaxis.grid()
fig2.yaxis.grid()
fig2.set_ylabel('Y')
fig2.set_title(r"$f'(x)$",fontsize=18)
```

函式、一次微分、二次微分圖形 (五)

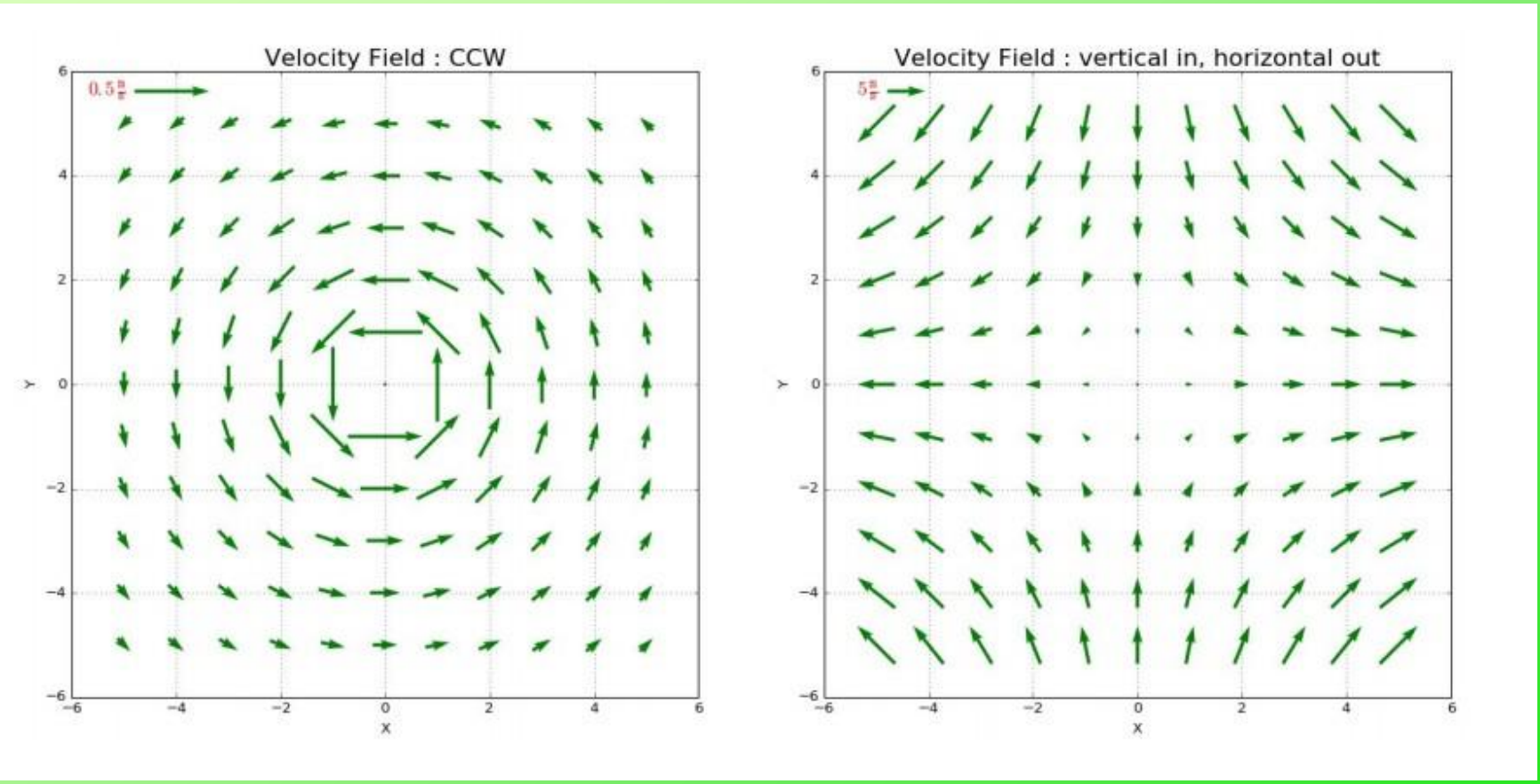
```
# 313 圖形: 直向為 3 , 橫向為 1 , 第 3 個子圖
fig3 = plt.subplot("313")
fig3.plot(xs,df2s,'g')
fig3.xaxis.grid()
fig3.yaxis.grid()
fig3.set_xlabel('X')
fig3.set_ylabel('Y')
fig3.set_title(r"$f''(x)$",fontsize=18)

# 自動調整各子圖間的位置
plt.tight_layout()

plt.show()
```

流場計算與顯示 (一)

- 流體力學的數值運算經常需要顯示速度分佈，速度方向與大小通常以調整箭頭角度與長度來表示。在 **matplotlib** 有 **quiver** 函式是專門用來顯示流速分佈圖，此函式需要的參數很直觀，即是各位置的 **x** 與 **y** 座標與速度的兩分量。此外，也可利用 **quiverkey** 來設定圖示，圖示位置是依圖形的比例位置來設定，左下角為 **(0,0)** ，右上角為 **(1,1)** 。

流場計算與顯示 (二)

下圖為兩速度場，圖左為逆時鐘旋轉，圖右流體由上下流入，左右流出。兩圖的圖示位置都設定在圖形的 (0.1,0.97) 位置。



流場計算與顯示 (三)

```
import matplotlib.pyplot as plt
import numpy as np

# 速度場: 逆時鐘旋轉
def vel_field1(x,y) :

    r = np.sqrt(x**2 + y**2)
    r[r==0] = 1e-10     # 讓 r 不為零

    cos , sin = x/r , y/r
    # 離圓心越遠, 速度越小
    u , v = cos / (1+r) , sin/(1+r)
    return ( -v , u )

# 速度場: 上下流入, 左右流出
def vel_field2(x,y) :

    r = np.sqrt(x**2 + y**2)
    r[r==0] = 1e-10     # 讓 r 不為零

    cos , sin = x/r , y/r
    u , v = r * cos , r * sin
    return ( u , -v )
```

```
# 區域
xmin , xmax = -5 , 5
ymin , ymax = -5 , 5

ds , n = 2 , 11

xs = np.linspace(xmin,xmax,n)
ys = np.linspace(ymin,ymax,n)

# 格點
ptx , pty = np.meshgrid( xs , ys )

# 設定圖形周邊區域為 白色
plt.figure(facecolor='white')
```

流場計算與顯示 (四)

```
for i in range(2) :

    # 圖形擺設為左右兩圖
    fig = plt.subplot( "12" + str(i+1) )

    # 計算流場數值
    if i == 0 :
        u , v = vel_field1(ptx,pty)
        ar = 0.5
        title = 'Velocity Field : CCW'
    else :
        u , v = vel_field2(ptx,pty)
        ar = 5
        title = 'Velocity Field :
                 vertical in,
                 horizontal out'

    # 在 xs , ys 位置上畫速度場 u , v ,
    # 綠色箭頭,箭頭中心在格點上
    q = fig.quiver( xs, ys, u, v,
        color='green',pivot='mid' )
    # 箭頭圖示
    astr = r'$\mathtt{' + str(ar) +
                      r'\;\frac{m}{s}}$'
    # 在畫面比例位置 (0.1,0.97) 畫速度值為
    # ar 的箭頭, 文字放在箭頭的 West 側
    fig.quiverkey( q, 0.1, 0.97, ar,
                   astr , labelpos='W',
                   labelcolor='red',
                   fontproperties=
                   {'weight':'bold',
                    'size':16} )

    # 重新調整 X 與 Y 的顯示範圍
    fig.axis([xmin-ds, xmax+ds ,
              ymin-ds, ymax+ds])

    # 設定圖形標頭文字
    fig.set_title(title, fontsize=20)

    # 設定 X 軸與 Y 軸文字
    fig.set_xlabel('X')
    fig.set_ylabel('Y')

    # 顯示格網
    fig.grid()

plt.show()
```

prey-predator 模型數值模擬 (一)

- 在一封閉區域內，獵物(prey)與獵食動物(predator)數量通常可用以下的微分方程式來模擬：

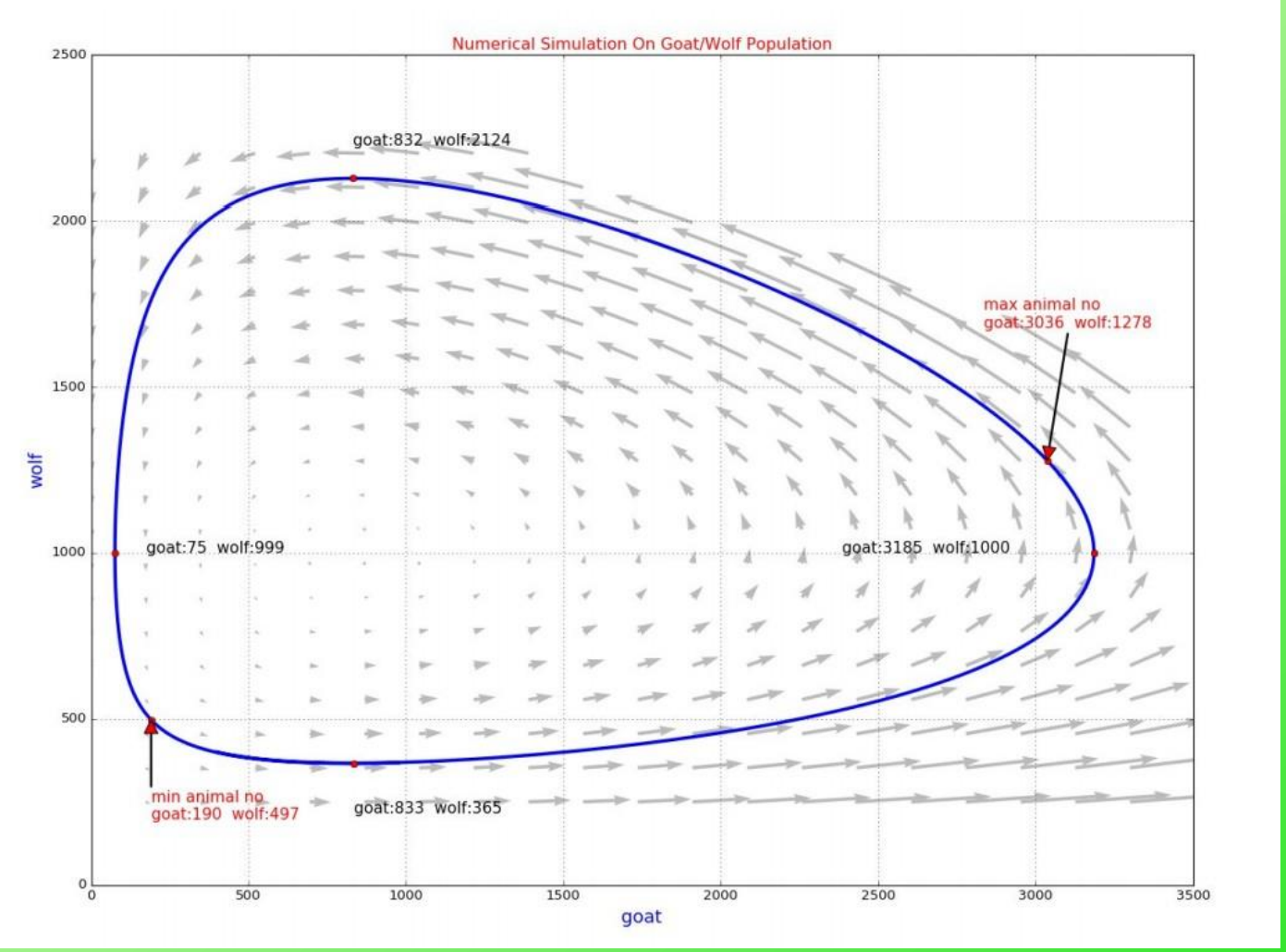
$$\frac{du}{dt} = (k_1 - k_2 v)u$$
$$\frac{dv}{dt} = (k_3 u - k_4)v$$

u 為獵物數量，v 為獵食動物數量，t 為時間，四個 k_i 為常數。這個數值模型為非線性的微分方程式，但使用最簡單的 Euler 方法來求解也異常簡單，數值公式為：

$$u_{i+1} = u_i + \Delta t(k_1 - k_2 v_i)u_i$$
$$v_{i+1} = v_i + \Delta t(k_3 u_i - k_4)v_i$$

u_i 與 v_i 分別為 u 與 v 在時間為 i × Δt 的計算值。u_0 與 v_0 為起始已知。本題核心程式很短，大部份的程式碼是用來畫圖的指令。在程式中，獵物為羊，獵食動物為狼，起始條件為兩種動物各為 400 隻。在程式中，我們以箭頭來顯示兩種動物的數量變化。

prey-predator 模型數值模擬 (二)



prey-predator 模型數值模擬 (三)

```
import numpy as np
import matplotlib.pyplot as plt

k1 , k2 = 2 , 0.002
k3 , k4 = 0.0006 , 0.5

# 羊數量變化方程式
def  fn1( goat , wolf ) :
    global k1 , k2 , k3 , k4
    u = ( k1 - k2 * wolf ) * goat
    return u

# 狼數量變化方程式
def  fn2( goat , wolf ) :
    global k1 , k2 , k3 , k4
    v = ( k3 * goat - k4 ) * wolf
    return v

# 運算次數
n = 10000

plt.figure(facecolor='white')

t1 , t2 = 0 , 8
ts , dt = np.linspace(t1,t2,n,
          retstep=True)

gs = np.empty(ts.size)
ws = np.empty(ts.size)

max_ano , min_ano = 0 , 1000000
```

```
# gs : goat , ws : wolf
gs[0] , ws[0] = 400 , 400

# Euler 計算法
for i in range(n-1) :

    if gs[i] + ws[i] > max_ano :
        max_n = i
        max_ano = gs[i] + ws[i]

    if gs[i] + ws[i] < min_ano :
        min_n = i
        min_ano = gs[i] + ws[i]

    u = fn1(gs[i],ws[i])
    v = fn2(gs[i],ws[i])

    gs[i+1] = gs[i] + dt * u
    ws[i+1] = ws[i] + dt * v

plt.plot(gs,ws,linewidth=3)

wimax , wimin = ws.argmax() , ws.argmin()
gimax , gimin = gs.argmax() , gs.argmin()

wmax , wmin = ws[wimax] , ws[wimin]
gmax , gmin = gs[gimax] , gs[gimin]
```

prey-predator 模型數值模擬 (四)

```
# 畫出羊與狼的最大/最小數量處
plt.plot(gs[wimax],ws[wimax],'or')
plt.plot(gs[wimin],ws[wimin],'or')
plt.plot(gs[gimax],ws[gimax],'or')
plt.plot(gs[gimin],ws[gimin],'or')

# 畫出羊與狼的數量和的最大/最小處
wnmax , gnmax = ws[max_n] , gs[max_n]
plt.plot(gnmax,wnmax,'or')

wnmin , gnmin = ws[min_n] , gs[min_n]
plt.plot(gnmin,wnmin,'or')

ds = 100
```

prey-predator 模型數值模擬 (五)

```
# 顯示羊與狼的最大/最小數量值
plt.annotate('goat:'+ str(int(gs[wimax]))+"  wolf:"+str(int(ws[wimax])),
             xy=(gs[wimax], ws[wimax]), fontsize=14, color='black',
             xytext=(gs[wimax], ws[wimax]+ds))

plt.annotate('goat:'+ str(int(gs[wimin]))+"  wolf:"+str(int(ws[wimin])),
             xy=(gs[wimin], ws[wimin]), fontsize=14, color='black',
             xytext=(gs[wimin], ws[wimin]-1.5*ds))

plt.annotate('goat:'+ str(int(gs[gimax]))+"  wolf:"+str(int(ws[gimax])),
             xy=(gs[gimax], ws[gimax]), fontsize=14, color='black',
             xytext=(gs[gimax]-7*ds, ws[gimax]))

plt.annotate('goat:'+ str(int(gs[gimin]))+"  wolf:"+str(int(ws[gimin])),
             xy=(gs[gimin], ws[gimin]), fontsize=14, color='black',
             xytext=(gs[gimin]+ds, ws[gimin]))

# 顯示羊與狼數量和的最大/最小值
plt.annotate('max animal no\ngoat:'+ str(int(gnmax))+"  wolf:"+str(int(wnmax)),
             xy=(gnmax, wnmax), fontsize=14, color='red',
             xytext=(gnmax-2*ds, wnmax+4*ds),
             arrowprops=dict(facecolor='red',width=1))

plt.annotate('min animal no\ngoat:'+ str(int(gnmin))+"  wolf:"+str(int(wnmin)),
             xy=(gnmin, wnmin), fontsize=14, color='red',
             xytext=(gnmin, wnmin-3*ds),
             arrowprops=dict(facecolor='red',width=1))
```

prey-predator 模型數值模擬 (六)

```
# 畫羊與狼的變化率
xmin , xmax = 0 , 3300
ymin , ymax = 250 , 2200

m = 20

xs2 = np.linspace(xmin,xmax,m)
ys2 = np.linspace(ymin,ymax,m)

ptx , pty = np.meshgrid(xs2,ys2)

ptx2 = ptx[ptx+3*pty<=8000]
pty2 = pty[ptx+3*pty<=8000]

u , v = fn1(ptx2,pty2) , fn2(ptx2,pty2)

plt.quiver(ptx2,pty2,u,v,color="#bbbbbb",pivot="tail")

plt.grid()
plt.xlabel('goat', color='blue', fontsize=16 )
plt.ylabel('wolf', color='blue', fontsize=16 )
plt.title('Numerical Simulation On Goat/Wolf Population' , color='red' )

plt.show()
```