ENGN2020 – Midterm #2

Problem 1

(a) Class Vertex:

Here is a brief description of Class Vertex, as shown in Table. 1.

Table.1 The data members and methods of the Class Vertex

	Name	Description
Data member	board	Store the current tic-tac-toe board
Method	__init__	The constructor of the class, takes in the board stored in a 3x3 matrix
	isFull	Check whether is board is full
	get_empty	get all empty positions on the board

Please see the code attached in Appendix 1.1.

(b) Method of get_status():

The method get_status() is added to the Vertex class. The brief description of get_status() is shown in Table.2.

Table.2 The description of method get_status()

Name	get_status		
Description	check the result of the current board		
	Name	Type	Description
Input	self	object	The object itself
Output	result	string	The result from 'X wins' , 'O wins' and 'in progress'

In the method get_status(), several steps are used:

Step 1: Check all rows to see if all elements are equal. If so, check if it`s 1, -1 or 0.

Step 2: Check all columns to see if all elements are equal. If so, check if it`s 1, -1 or 0.

Step 3: Check the diagonal line from left top to right bottom to see if all elements are equal. If so, check if it`s 1, -1 or 0.

Step 4: Check the diagonal line from right top to left bottom to see if all elements are equal. If so, check if it`s 1, -1 or 0.

Please see the code attached in Appendix 1.1.

(c) Function of computer_move

The brief description of computer_move () is shown in Table.3.

Table.3 The description of function computer_move ()

Name	computer_move		
Description	Decide the computer`s next move purely based on brute-force probabilites		
	Name	Type	Description
Input	currentBoard	Vertex	the object of Class Vertex storing the current board
	myTurn	string	the input indicate whose turn it is('X' or 'O')
Output	newBoard	Vertex	the object of Class Vertex after the move

A helper function called calculate_probability() is used in computer_move to calculate the winning probability of all possible moves. The brief description of calculate_probability() is shown is Table.4.

Table.4 The description of function calculate_probability ()

Name	calculate_probability		
Description	calculate the win probability of all possible moves using DFS method		
	Name	Type	Description
Input	currentBoard	Vertex	the object of Class Vertex storing the current board
	myTurn	string	the input indicate whose turn it is('X' or 'O')
Output	result	dict	the total outcome and win times of each possible move

Test results of the computer_move() function:

(1) Case 1:

Code:

```
A = np.array([[ 0, 0,-1],
```

```
                    [ 0, 1, 1],
                    [ 1, 0,-1]])
    a = Vertex(A)
    nextMove = computer_move(a,'O')
    print(nextMove.board)
```

Result:

The board after computer`s move is:

```
[[-1  0 -1]
 [ 0  1  1]
 [ 1  0 -1]]
```

The next move of computer is on the top left of the board.

The total number of possible outcomes is 5

The total number of possible winning outcomes is 1

(2) Case 2:

Result:

The board after computer`s move is:

```
[[ 0  0 -1]
 [ 0  1  0]
 [ 0  0  0]]
```

The next move of computer is in the center of the board.

The total number of possible outcomes is 3468

The total number of possible winning outcomes is 1312

(3) Case 3:

Result:

The board after computer`s move is:

```
[[ 1  0  0]
 [-1 -1  0]
 [ 1 -1  1]]
```

The next move of computer is in the center of the board.

The total number of possible outcomes is 6

The total number of possible winning outcomes is 4

(4) Case 4:

Result:

The board after computer`s move is:

```
[[ 1  0  0]
 [ 0 -1  0]
 [ 0  0  0]]
```

The next move of computer is on the top left of the board.

The total number of possible outcomes is 3198

The total number of possible winning outcomes is 792

Please see the code in Appendix 1.2.

(d) The complete tic-tac-toe game

The problem1.py contains the game as well as all necessary modules. Please run the .py file under shell by:

python #full path of the problem1.py

If you are already under python environment, please run the .py file by:

%run #full path of the problem1.py

The game is a pure-text game, the interface is shown in Fig.1.

```
In [41]: %run D:\Brown\Study\2019Spring\ENGN2020\Mid2\problem1.py
Welcome to Tic-tac-toe! I`ll use 'X' and you`ll use 'O', now let`s see who goes first
Generatering a random number to decide who goes first:
0 : I`ll go fisrt
1 : You`ll go fisrt
The number is 0, I`ll go fisrt
My turn:
After move:
  |   |
  | X |
  |   |
Your turn:

Please enter move(in 'row,column' format and starts from 1):1,1
After move:
O |   |
  | X |
  |   |
My turn:
After move:
O |   | X
  | X |
  |   |
Your turn:

Please enter move(in 'row,column' format and starts from 1):3,1
After move:
O |   | X
  | X |
O |   |
My turn:
After move:
O |   | X
X | X |
O |   |
Your turn:
```

Fig 1. The interface of tic-tac-toe game

Please note that the when you enter your move, it should follow the 'row,column' format, and row and column starts with 1.

Please see the complete code in Appendix 1.3.

Problem 2

(a) Answer:

Solve the function by scipy.optimize.fsolve(), when the initial values of [CA, T] differs, three different roots can be calculated:

(1) When the initial values of [CA, T] = [10, 300.0], the roots are CA = 1.9052761218083372, T = 301.894930354838.
(2) When the initial values of [CA, T] = [2.0, 500.0], the roots are CA = 0.2527196071917932, T = 334.9539600568946.
(3) When the initial values of [CA, T] = [200, 3000], the roots are CA = 0.8066255609579893, T = 323.873193247566.

In conclusion, there are three possible solutions:

$$\begin{cases} CA = 1.90527612180833 \\ T = 301.894930354838 \end{cases}, \begin{cases} CA = 0.252719607191793 \\ T = 334.9539600568946 \end{cases} \text{ and } \begin{cases} CA = 0.80662556095798 \\ T = 323.873193247566 \end{cases}$$

Please see the code in Appendix 2.1.

(b) Answer:

Let x_1 , x_2, θ stand for *CA*, *T* and T_{inlet}, respectively. Then the given functions can be written as:

$$f_1(x_1, x_2) = \frac{C_{A,inlet} - x_1}{\tau} - kx_1$$

$$f_2(x_1, x_2, \theta) = \frac{\theta - x_2}{\tau} - \frac{H_{rxn}}{\rho c_p} kx_1$$

$$\mathrm{k} = Ae^{-E_A/Rx_2}$$

Then the Jacobian matrix of the given functions can be written as:

$$J = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} \end{bmatrix} = \begin{bmatrix} -\frac{1}{\tau} - k & -kx_1 \frac{E_A}{Rx_2^2} \\ -\frac{H_{rxn}}{\rho c_p} k & -\frac{1}{\tau} - kx_1 \frac{H_{rxn} E_A}{\rho c_p R x_2^2} \end{bmatrix}$$

Let dev(J) equals to 0 together with $f_1(x_1, x_2) = 0$, we can solve for two roots:

$$\begin{cases} x_1 = 1.595861231544271 \\ x_2 = 312.0553014597565 \end{cases} \text{and} \begin{cases} x_1 = 0.4697772609992 \\ x_2 = 329.45193859762 \end{cases}$$

Use those two roots and $f_2(x_1, x_2, \theta) = 0$ to solve θ:

$$\begin{cases} \theta_1 = 298.8401691775697 \\ \theta_2 = 303.9705942598330 \end{cases}$$

So when $T_{inlet}^* = 298.8402$ or $T_{inlet}^* = 303.9706$, where the number of steady-state solutions changes.

Please see the code in Appendix 2.2.

(c) Answer:

Let ***x₁***, ***x₂*** stand for ***CA*** and ***T***, respectively.

According to Part(a), there are three solutions:

$$\begin{cases} x_1 = 1.90527612180833 \\ x_2 = 301.894930354838 \end{cases}, \begin{cases} x_1 = 0.252719607191793 \\ x_2 = 334.9539600568946 \end{cases} \text{and} \begin{cases} x_1 = 0.80662556095798 \\ x_2 = 323.873193247566 \end{cases}$$

(1) $x_1 = 1.90527612180833$ and $x_2 = 301.894930354838$

The Jacobian matrix is:

$$J = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} \end{bmatrix} = \begin{bmatrix} -0.01749528 & -0.00026148 \\ 0.01657617 & -0.01143592 \end{bmatrix}$$

The eigenvalues are $\lambda_1 = -0.01666667$ and $\lambda_2 = -0.01226453$

Since all Re(λ)<0, the state of this solution is stable.

(2) $x_1 = 0.252719607191793$ and $x_2 = 334.9539600568946$

The Jacobian matrix is:

$$J = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} \end{bmatrix} = \begin{bmatrix} -0.13189849 & -0.00391809 \\ 2.30518719 & 0.06171394 \end{bmatrix}$$

The eigenvalues are $\lambda_1 = -0.05351788$ and $\lambda_2 = -0.01666667$

Since all Re(λ)<0, the state of this solution is stable.

(3) $x_1 = 0.80662556095798$ and $x_2 = 323.873193247566$

The Jacobian matrix is :

$$J = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} \end{bmatrix} = \begin{bmatrix} -0.04132442 & -0.00286226 \\ 0.49327293 & 0.04059223 \end{bmatrix}$$

The eigenvalues are $\lambda_1 = -0.01666667$ and $\lambda_2 = 0.01593448$

Since any Re(λ)>0, the state of this solution is unstable.

Please see the code attached in Appendix 2.3.

(d) Answer:

The ratio of the largest and smallest eigenvalue moduli can be used to judge the stiffness of the differential equations. According to Part(c), in all three solutions, the ratios of the largest and smallest eigenvalue moduli are rather small. Therefore, this system of differential equations is not stiff.

From part(c), there is an unstable state. Therefore, from that point of view, the system is not stiff.

(e) Answer:

(1) $CA_0 = 0, T_0 = 0$

The plot of CA and T versus t is shown in Fig 2.

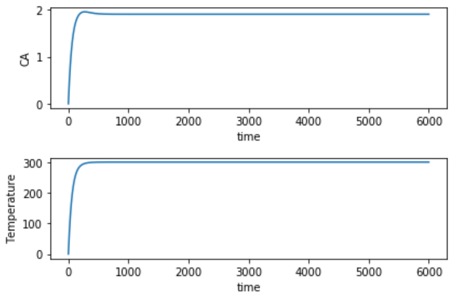


Fig 2. The plot of CA and T versus t, when $CA_0 = 0, T_0 = 0$

At last, the CA converges to CA = 1.9052761218083372, and T converges to T = 301.894930354838

(2) $CA_0 = 0, T_0 = 500$

The plot of CA and T versus t is shown in Fig 3.

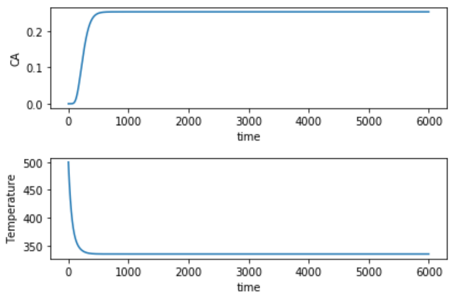


Fig 3. The plot of CA and T versus t, when $CA_0 = 0, T_0 = 500$

At last, the CA converges to CA = 0.252719607191793, and T converges to T = 334.9539600568946

(3) $CA_0 = 10, T_0 = 500$

The plot of CA and T versus t is shown in Fig 4.

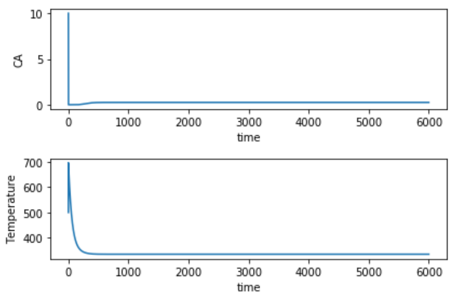


Fig 4. The plot of CA and T versus t, when $CA_0 = 10, T_0 = 500$

At last, the CA converges to CA = 0.252719607191793, and T converges to T = 334.9539600568946

(4) $CA_0 = 2, T_0 = 2500$

The plot of CA and T versus t is shown in Fig 5.

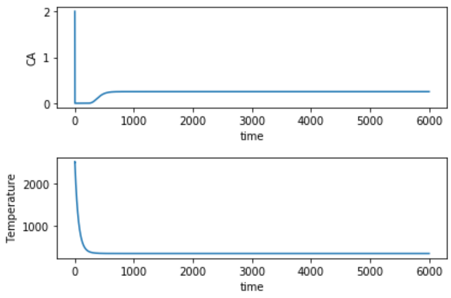


Fig 5. The plot of CA and T versus t, when $CA_0 = 2, T_0 = 2500$

At last, the CA converges to CA = 0.252719607191793, and T converges to T = 334.9539600568946

(5) $CA_0 = 200, T_0 = 0$

The plot of CA and T versus t is shown in Fig 6.

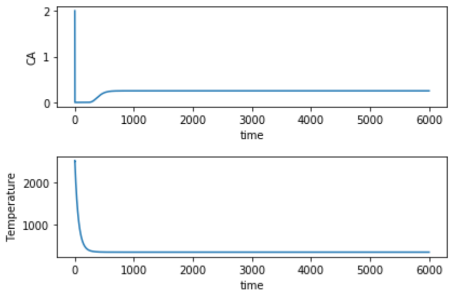


Fig 6. The plot of CA and T versus t, when $CA_0 = 200, T_0 = 0$

At last, the CA converges to CA = 1.9052761218083372, and T converges to T = 301.894930354838

Basically, when the initial T is relatively high, the system tends to converge to the stable state of:

$$\begin{cases} CA = 0.252719607191793 \\ T = 334.9539600568946 \end{cases}$$

When the initial T is relatively low, the system tends to converge to the stable state of:

$$\begin{cases} CA = 1.90527612180833 \\ T = 301.894930354838 \end{cases}$$

Please see the attached code in Appendix.2.4.

(f) Answer:

The map that could be used to determine from a given initial condition that which steady state will be reached is shown in Fig.7.

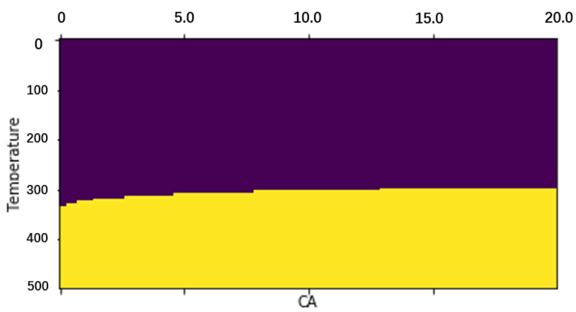


Fig 7. The map of steady state based on input initial condition

The purple part indicates the stable state of:

$$\begin{cases} CA = 1.90527612180833 \\ T = 301.894930354838 \end{cases}$$

The yellow part indicates the stable state of:

$$\begin{cases} CA = 0.252719607191793 \\ T = 334.9539600568946 \end{cases}$$

We can draw the similar conclusion with part(e), when the initial T is relatively high, the system tends to converge to the stable state of:

$$\begin{cases} CA = 0.252719607191793 \\ T = 334.9539600568946 \end{cases}$$

When the initial T is relatively low, the system tends to converge to the stable state of:

$$\begin{cases} CA = 1.90527612180833 \\ T = 301.894930354838 \end{cases}$$

Please see the code attached in Appendix 2.5.

Problem 3

(a) Answer:

The noise free contour is shown in Fig.8.

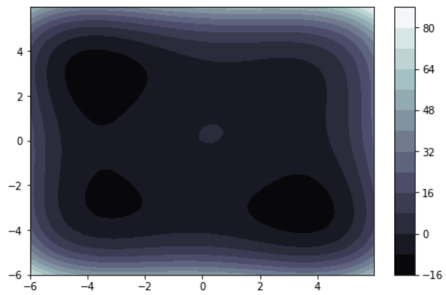
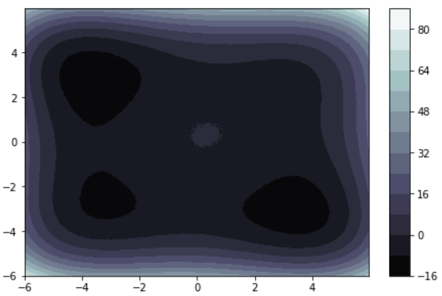
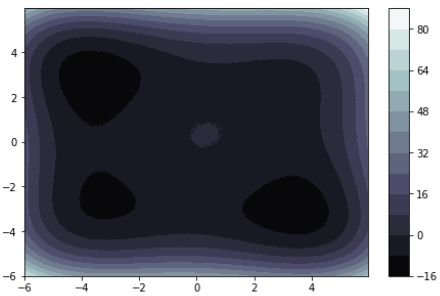


Fig 8. The map of steady state based on input initial condition

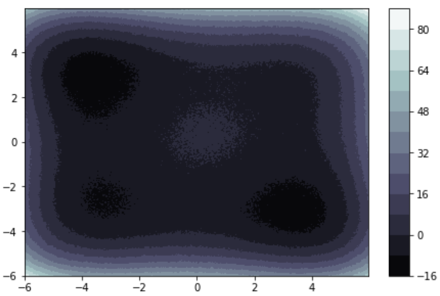
The contour plots at different of sigma is shown is Fig 9.



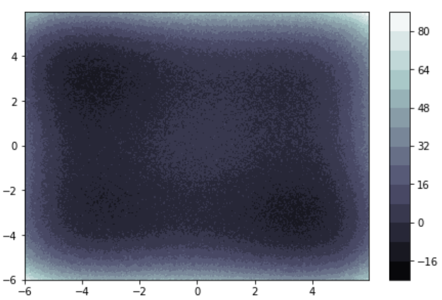
(a) $\sigma = 0.01$



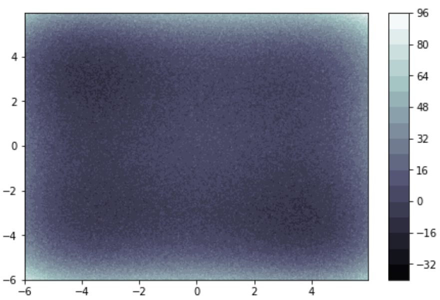
(b) $\sigma = 0.1$



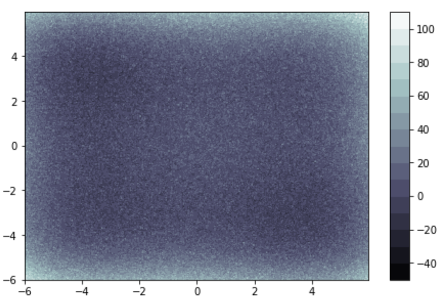
(c) $\sigma = 1$



(d) $\sigma = 3$



(e) $\sigma = 5$



(f) $\sigma = 10$

Fig 9. Contour plots at different of sigma

From Fig.9. we can see when σ increases, the influence becomes more and more significant. When $\sigma = 5$, the influence becomes quite significant. When $\sigma = 10$, the noise nearly washes out the signal.

(b) Answer:

Given the initial simplex of [(0,0),(-1,0),(0,1)], the Nelder-Mead algorithm will find the minimum on the noise free function. The process is shown in Fig.10.

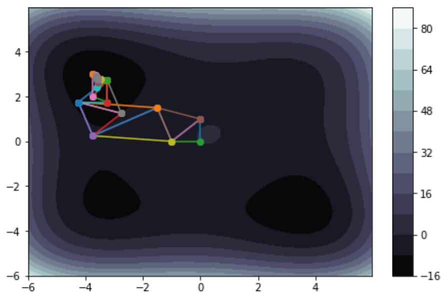


Fig.10. The process that Nelder-Mead algorithm finds the minimum given the initial simplex

The position of the this local minima is (-3.63460063, 2.91071661)

Besides, there are three additional local minimas in the function. By using different initial values, we can find positions of those local minimas:

(-3.25733016, -2.64544543)

(2.98019805, 2.31968217)

(3.43582195, -3.0972582)

The number of times out of 100 that converges to the same local minima and the number of times out of 100 that converges to any local minima based on different σ is shown in Table.5.

Table.5 The number of converging result

Sigma σ	Number of same minima	Number of any minima
0	1000	1000
0.5	632	892
1	204	422
1.5	56	176
2	32	85
2.5	18	46
3	1	15
3.5	4	17
4	3	14
4.5	2	7
5	1	4

The plots of those two kinds of number versus sigma is shown in Fig.11.

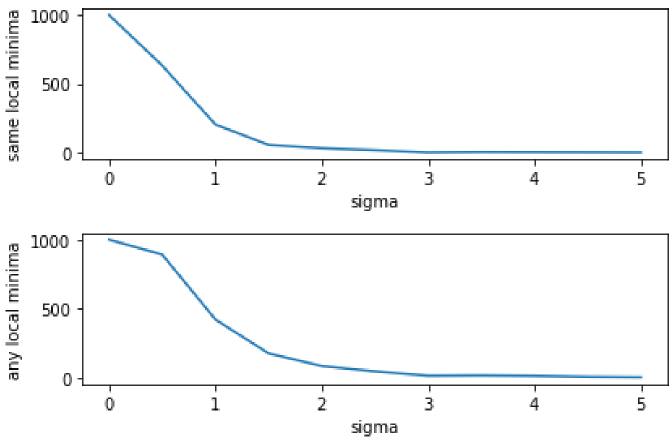


Fig.11. The number of converging to the same local minima and the number of converging to any local minima.

(c) Answer:

(1) use Nelder-Mead method in scipy.optimize.minimize

The number of times out of 100 that converges to the same local minima and the number of times out of 100 that converges to any local minima based on different σ is shown in Table.6.

(d) **Table.6** The number of converging results

Sigma σ	Number of same minima	Number of any minima
0	1000	1000
0.5	614	996
1	326	776
1.5	208	495
2	109	296
2.5	77	225
3	40	127
3.5	36	102
4	19	79
4.5	17	59
5	15	50

The plots of those two kinds of number versus sigma is shown in Fig.12.

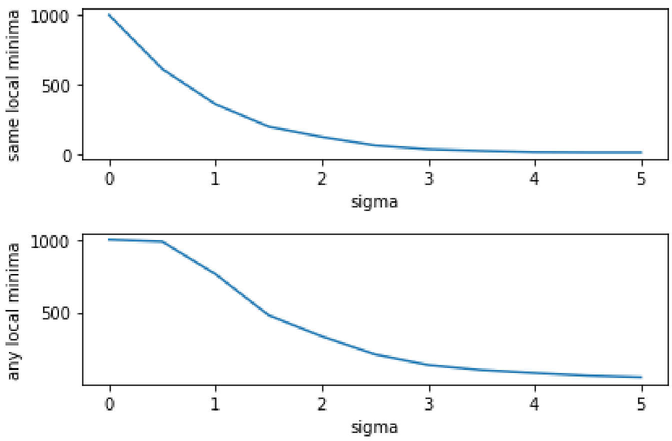


Fig.12. The number of converging to the same local minima and the number of converging to any local minima.

(2) use BFGS method in scipy.optimize.minimize

The number of times out of 100 that converges to the same local minima and the number of times out of 100 that converges to any local minima based on different σ is shown in Table.7.

(e) **Table.7** The number of converging results

Sigma σ	Number of same minima	Number of any minima
0	1000	1000
0.5	799	799
1	668	668
1.5	495	495
2	362	362
2.5	239	239
3	211	211
3.5	140	140
4	125	125
4.5	100	100
5	72	72

The plots of those two kinds of number versus sigma is shown in Fig.13.

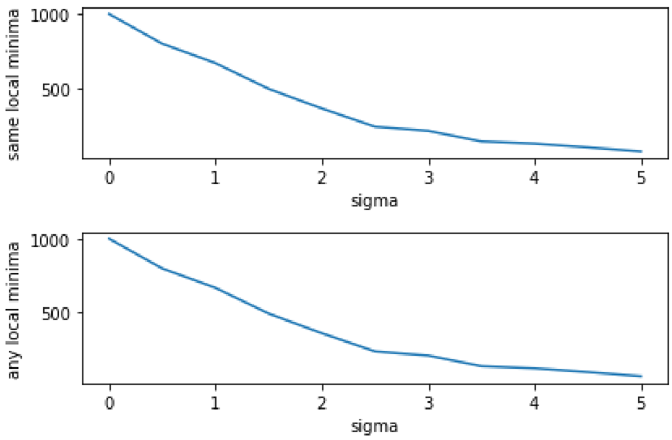


Fig.12. The number of converging to the same local minima and the number of converging to any local minima.

Among the built in Nelder-Mead, home-made Nelder-Mead and BFGS, BFGS is definitely the best algorithm. The speed is extremely fast and the algorithm doesn`t converge to other local minimas. Also, it`s accuracy is better than the other two algorithms with existence of noise.

Appendix

1. Code of Problem 1

(1) part a and b

```
class Vertex:
	'''
	* @name: __init__
	* @description: the constructor of the class, save the input board
	* @param board: the input board which is saved in a 3X3 matrix
	'''
	def __init__(self,board):
		self.board = board
	'''
	* @name: get_status
	* @description: check the result of the current board
	* @return: the result('X wins', 'O wins' or 'in progress')
	'''
	def get_status(self):
		#check all rows
		for i in range(3):
			#if the current row has three same element
			if self.board[i][0] == self.board[i][1] and self.board[i][0] == self.board[i][2]:
				if self.board[i][0] == 1:
					return "X wins"
				if self.board[i][0] == -1:
					return "O wins"
		#check all cols
		for i in range(3):
			#if the current col has three same element
			if self.board[0][i] == self.board[1][i] and self.board[0][i] == self.board[2][i]:
				if self.board[0][i] == 1:
					return "X wins"
				if self.board[0][i] == -1:
					return "O wins"
		#check diagonal
		if self.board[0][0] == self.board[1][1] and self.board[0][0] == self.board[2][2]:
			if self.board[0][0] == 1:
				return "X wins"
			if self.board[0][0] == -1:
				return "O wins"
		#check diagonal
		if self.board[0][2] == self.board[1][1] and self.board[0][2] == self.board[2][0]:
			if self.board[0][2] == 1:
				return "X wins"
			if self.board[0][2] == -1:
				return "O wins"

		return 'in progress'
	'''
	* @name: get_children
	* @description: get all possible board with one additional move from the current board
	* @return: the result('X wins', 'O wins' or 'in progress')
	'''
	def get_children(self,index):
		#get the current row and col
		row = index // 3
		col = index % 3
		#the 8 neighbors
		offsetRow = [-1,-1,-1, 0, 0, 1, 1, 1]
		offsetCol = [-1, 0, 1,-1, 1,-1, 0, 1]
		#the list to save final results
		result = []

		for i in range(8):
			y = row + offsetRow[i]
			x = col + offsetCol[i]

			if y>=0 and y<3 and x>=0 and x<3:
				if self.board[y][x] == 0:
					result.append(3*y+x)
		return result
	'''
	* @name: isFull
	* @description: check whether the current board is full
	* @return: boolean, whether the current boad is full
	'''
	def isFull(self):
		for row in range(3):
			for col in range(3):
```

```
                        if self.board[row][col] == 0:
                                return False
                return True
        '''
        * @name: get_empty
        * @description: get all empty positinos on the board
        * @return: list, the list storing all position indexes
        '''
        def get_empty(self):
                result = []

                for row in range(3):
                        for col in range(3):
                                if self.board[row][col] == 0:
                                        result.append(row*3+col)

                return result
        '''
        * @name: print_board
        * @description: print the board with "X" and "O"
        '''
        def print_board(self):
                matrix = []
                for i in range(3):
                        row = []
                        for j in range(3):
                                if self.board[i][j] == 0:
                                        row.append(" ")
                                elif self.board[i][j] == 1:
                                        row.append("X")
                                else:
                                        row.append("O")
                        matrix.append(row)

                for row in range(3):
                        s=''
                        s = s+matrix[row][0]+" | "+matrix[row][1]+" | "+matrix[row][2]
                        print(s)
```

(2) part c

```
'''
* @name: calculate_probability
* @description: calculate the win probability of all possible moves
* @param currentBoard: the object of Class Vertex storing the current board
* @param myTurn: the input indicate whose turn it is('X' or 'O')
* @return: dict, the total outcome and win times of each possible move
'''
def calculate_probability(currentBoard,myTurn):
        emptyPositions = currentBoard.get_empty()
        emptyNum = len(emptyPositions)

        #initialize the final result
        result = []

        #base condition
        if emptyNum == 0:
                return result


        #get the number to be placed
        myNumber = 0
        if myTurn == 'X':
                myNumber = 1
                otherMove = 'O'
        else:
                myNumber = -1
                otherMove = 'X'

        #loop for all possiblities
        for position in emptyPositions:

                #intialize the total possiblities and win outcomes of current move
                totalResult = 0
                winResult = 0

                #get the row and col of the current empty position
                row = position//3
                col = position%3
                #get the current board
                board = np.copy(currentBoard.board)
                #set the current empty position by my Number
                board[row][col] = myNumber
```

```
            #create a new object of class Vertex based on the modified board
            newBoard = Vertex(board)
            #get the output of next move
            outcome = newBoard.get_status()

            #if win
            if outcome.startswith(myTurn):
                totalResult = 1
                winResult = 1
                result.append({"move":position,"total":totalResult, 'win':winResult})
            #if no direct result
            elif outcome == 'in progress':
                #if the board is full
                if newBoard.isFull():
                    totalResult = 1
                    winResult = 0
                #if the board is not full
                else:
                    #loop all empty positions after the new move
                    newEmptyMoves = newBoard.get_empty()

                    for newPosition in newEmptyMoves:
                        tempBoard = np.copy(newBoard.board)
                        tempRow = newPosition//3
                        tempCol = newPosition%3
                        #move a move as the other player
                        tempBoard[tempRow][tempCol] = -myNumber

                        oppositeMove = Vertex(tempBoard)
                        #if the other player wins
                        if oppositeMove.get_status().startswith(otherMove):
                            totalResult = totalResult+1
                        elif oppositeMove.isFull():
                            totalResult = totalResult+1
                        else:
                            tempResult = calculate_probability(oppositeMove,myTurn)
                            if len(tempResult)!= 0:
                                for item in tempResult:
                                    totalResult = totalResult + item['total']
                                    winResult = winResult + item['win']

                result.append({"move":position,"total":totalResult, 'win':winResult})
            else:
                totalResult = 1
                winResult = 0
                result.append({"move":position,"total":totalResult, 'win':winResult})

    return result
'''
* @name: computer_move
* @description: make the move based on calculation of winning probability of all possible moves
* @param currentBoard: the object of Class Vertex storing the current board
* @param myTurn: the input indicate whose turn it is('X' or 'O')
* @return: Vertex, the object of Class Vertex after the move
'''
def computer_move(currentBoard,myTurn):

    currentStatus = currentBoard.get_status()

    if currentStatus == 'O wins' or currentStatus == 'X wins':
        print(currentStatus)
        return currentBoard
    elif currentStatus == 'in progress' and currentBoard.isFull():
        print('It`s a tie!')
        return currentBoard

    possiblities = calculate_probability(currentBoard,myTurn)

    maxPossiblity = -1
    finalMove = -1
    totalWins = 0
    totalNum = 0

    board = np.copy(currentBoard.board)

    for move in possiblities:
        winRate = move['win']/move['total']
        if winRate>maxPossiblity:
            maxPossiblity = winRate
            finalMove = move['move']
            totalWins = move['win']
            totalNum = move['total']
```

```
    #get the row and col of the current empty position
    row = finalMove//3
    col = finalMove%3

    #get the number to be placed
    myNumber = 0
    if myTurn == 'X':
        myNumber = 1
    else:
        myNumber = -1

    board[row][col] = myNumber

    newBoard = Vertex(board)

    #print("The total number of possible outcome is "+str(totalNum))
    #print("The total number of possible wins is "+str(totalWins))

    return newBoard
```

(3) part d

```
def game():
    #initialize the new board:
    board = np.array([[ 0, 0, 0],
                       [ 0, 0, 0],
                       [ 0, 0, 0]])
    initialBoard = Vertex(board)
    #print some welcome message
    print("Welcome to Tic-tac-toe! I`ll use ‘X’ and you`ll use 'O', now let`s see who goes first")
    #generate a random number between 0 and 1
    print("Generatering a random number to decide who goes first:")
    print("0 : I`ll go fisrt")
    print("1 : You`ll go fisrt")

    a = random.randint(0,1)
    if a == 0:
        msg = "I`ll go fisrt"
    else:
        msg = "You`ll go fisrt"

    print("The number is "+str(a)+", "+msg)

    #if the user starts first,wait for first move
    if a == 1:
        #print the initial board
        print("Here is the current board:")
        initialBoard.print_board()

        #check whether the input is valid
        isValid = False

        #loop until the input is valid
        while not isValid:
            #get user`s input coordinate
            move = input("Please enter move(in 'row,column' format and starts from 1):")
            move = np.array(move.split(','),dtype=int)
            row = move[0]-1
            col = move[1]-1

            index = 3*row+col

            #check if it`s in the valid positions list
            emptyPositions = initialBoard.get_empty()

            if index in emptyPositions:
                isValid = True
            else:
                print("Invalid input, please input again!")

        #set the board based on user`s input
        newMove = np.copy(initialBoard.board)
        newMove[row][col] = -1

        #declare a new Vertex object
        newBoard = Vertex(newMove)
        #print the board after move
        print("After move:")
        newBoard.print_board()
    else:
        #the initial board is our input
```

```
        newMove = np.copy(initialBoard.board)
        newBoard = Vertex(newMove)


    #get the status of current board, only process if the status is in process
    result = newBoard.get_status()

    while result == 'in progress':
        #computer move
        print("My turn:")
        newBoard = computer_move(newBoard,'X')

        #get the new status
        result = newBoard.get_status()

        if newBoard.isFull():
            break

        if result != 'in progress':
            break

        #if still in process,print the board and wait for user`s input
        print("After move:")
        newBoard.print_board()


        print("Your turn:")

        #check whether the input is valid
        isValid = False
        #loop until the input is valid
        while not isValid:
            #get user`s input coordinate
            move = input("Please enter move(in 'row,column' format and starts from 1):")
            move = np.array(move.split(','),dtype=int)
            row = move[0]-1
            col = move[1]-1

            index = 3*row+col
            #check if it`s in the valid positions list
            emptyPositions = newBoard.get_empty()

            if index in emptyPositions:
                isValid = True
            else:
                print("Invalid input, please input again!")

        #set the board based on user`s input
        newMove = np.copy(newBoard.board)
        newMove[row][col] = -1
        #declare a new Vertex object
        newBoard = Vertex(newMove)
        #print the board after move
        print("After move:")
        newBoard.print_board()
        #get the new status
        result = newBoard.get_status()

        if newBoard.isFull():
            break
    #print the final result and the final board
    print("Final results:")
    newBoard.print_board()

    if result == 'in progress':
        print('Nobody won!')
    elif result == 'O wins':
        print('Good game! You won!')
    else:
        print("I won!")
```

2. Code of Problem 2

(1) part a

```
import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt
from scipy.optimize import fsolve
import math

Cinlet = 2
Tinlet = 300
```

```
Hrxn = -83700
Rhocp = 4184
tao = 60
A = 4.3e18
EA = 125500
R = 8.314

#part a
'''
* @name: f1
* @description: get the result of the system
* @param y: the input value, in format of [CA,T]
* @return: list, [dCA/dt,dT/dt]
'''
def f1(y):

    k = getK(y[1])
    #dCA/dt
    dy0 = ((Cinlet-y[0])/tao-k*y[0])
    #dT/dt
    dy1 = ((Tinlet-y[1])/tao-Hrxn*k*y[0]/Rhocp)

    return [dy0,dy1]
'''
* @name: getK
* @description: get k based on input temperature T
* @param T: the input temperture
* @return: k, the cofficient
'''
def getK(T):
    return A*math.exp(-EA/(R*T))

x, y =   fsolve(f1, [2.0, 500])
x, y =   fsolve(f1, [200, 3000])
x, y =   fsolve(f1, [10, 300])
```

(2) part b

```
#part b
'''
* @name: getDev
* @description: get the value of dev(Jacobian matrix) and f1 based on input value
* @param p: the input value, in format of [CA,T]
* @return: list, [dev(J),f1]
'''
def getDev(p):

    Y,T = p

    k = getK(T)

    a = -1/tao - k
    b = -k*Y*EA/R/T/T
    c = -Hrxn/Rhocp*k
    d = -1/tao - Hrxn/Rhocp*k*Y*EA/R/T/T

    return [a*d-b*c,(Cinlet-Y)/tao-k*Y]

x, y =   fsolve(getDev, (2, 500))
x, y =   fsolve(getDev, (10, 300))
'''
* @name: getTinlet
* @description: get Tinlet based on CA and T
* @param Y: the input value of CA
* @param T: the input value of T
* @return: float, the value of Tinlet
'''
def getTinlet(Y,T):

    k = getK(T)

    Tinlet = Hrxn*k*Y/Rhocp*tao+T

    return Tinlet

getTinlet(0.46977726099921147,329.45193859762)
getTinlet(1.5958612315442713,312.0553014597565)
```

(3) part c

```
'''
* @name: getDevMatrix
* @description: get Jacobian Matrix based on CA and T
* @param Y: the input value of CA
```

```
* @param T: the input value of T
* @return: float, the value of Tinlet
'''
def getDevMatrix(Y,T):
    k = getK(T)

    a = -1/tao - k
    b = -k*Y*EA/R/T/T
    c = -Hrxn/Rhocp*k
    d = -1/tao - Hrxn/Rhocp*k*Y*EA/R/T/T

    devMatrix = np.array([[a,b],[c,d]])

    return devMatrix

a = getDevMatrix(1.90527612180833,301.894930354838)
B = np.linalg.eig(a)
print(B[0])

a = getDevMatrix(0.252719607191793,334.9539600568946)
B = np.linalg.eig(a)
print(B[0])

a = getDevMatrix(0.80662556095798,323.873193247566)
B = np.linalg.eig(a)
print(B[0])
```

(4) part e

```
def f(y,t):

    k = getK(y[1])

    dy0 = ((Cinlet-y[0])/tao-k*y[0])
    dy1 = ((Tinlet-y[1])/tao-Hrxn*k*y[0]/Rhocp)

    return [dy0,dy1]

def solve(y0):
    t = np.linspace(0,6000,10000)
    #y0 = [0.8, 200.0]
    y = odeint(f, y0, t)

    print(y[-1])

    fig, axs = plt.subplots(2, 1)
    axs[0].plot(t,y[:,0],label='CA')
    axs[0].set_xlabel('time')
    axs[0].set_ylabel('CA')

    axs[1].plot(t,y[:,1],label='T')
    axs[1].set_xlabel('time')
    axs[1].set_ylabel('Temperature')

    fig.tight_layout()
    plt.show()

    return y[-1]

solve(y0 = [0,0])
solve(y0 = [0,500])
solve(y0 = [10,500])
solve(y0 = [2,2500])
solve(y0 = [200,0])
```

(5) part (f)

```
def mapInitialValues():
    Y = np.arange(0, 20, 0.1)
    lenY = Y.shape[0]
    T = np.arange(0, 500, 5)
    lenT = T.shape[0]
    Y_mesh, T_mesh = np.meshgrid(Y, T)

    result = np.zeros((Y_mesh.shape))

    case1 = [1.90527612180833,301.894930354838]
    case2 = [0.252719607191793,334.9539600568946]
    case3 = [0.80662556095798,323.873193247566]
    for i in range(lenY*lenT):
        row = i//lenY
        col = i%lenY

        temp = solve([Y_mesh[row][col],T_mesh[row][col]])
```

```
        distance1 = (case1[0]-temp[0])**2+(case1[1]-temp[1])**2
        distance2 = (case2[0]-temp[0])**2+(case2[1]-temp[1])**2
        distance3 = (case3[0]-temp[0])**2+(case3[1]-temp[1])**2

        if distance1<1:
            result[row][col] = 0.
        elif distance2<1:
            result[row][col] = 1.
        elif distance3<1:
            result[row][col] = -1.

    #plt.scatter(T, Y, s=result)
    #plt.show()

    return result

result = mapInitialValues()

fig, axs = plt.subplots(1, 1)
axs.matshow(result)
axs.set_xlabel('CA')
axs.set_ylabel('Temperature')

Y = np.arange(0, 20, 0.1)
T = np.arange(0, 500, 5)

plt.xscale('linear',0.2)
plt.yscale('linear',5)

plt.show()
```

3. Code of Problem 3

(1) part (a)

```
'''
* @name: drawNoisyContour
* @description: draw the contour with noise based on input sigma
* @param sigma: the sigma of normal distribution of noise
'''
def drawNoisyContour(sigma):
    delta = 0.025

    x = y = np.arange(-6.0, 6.0, delta)
    X, Y = np.meshgrid(x, y)

    Z = 0.045*X**4-X**2+0.5*X+0.065*Y**4-Y**2+0.5*Y+0.3*X*Y+np.random.normal(0,sigma,X.shape)

    Z = np.ma.array(Z)

    origin = 'lower'

    fig1, ax2 = plt.subplots(constrained_layout=True)

    CS = ax2.contourf(X, Y, Z, 15, cmap=plt.cm.bone, origin= origin)
    cbar = fig1.colorbar(CS)
    plt.plot()

drawNoisyContour(0.01)
drawNoisyContour(0.1)
drawNoisyContour(1)
drawNoisyContour(5)
drawNoisyContour(10)
```

(2) part (b)

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import minimize

def get_f(x1, x2, sigma):
    return (0.045 * x1**4 - x1**2 + 0.5 * x1 + 0.065 * x2**4 - x2**2 + 0.5 * x2 + 0.3 * x1 * x2)+np.random.normal(0,sigma)

def step_neldner(f,vertices,sigma):
    #define type {type, x, y, z}
    dtype = [('index', int), ('x1', float), ('x2', float),('z',float)]
    #use an array to store input vertices
    results = np.empty(3,dtype)
    #set the element in the results
    for i in range(3):
        results[i] = (i,vertices[i,0],vertices[i,1],get_f(vertices[i,0],vertices[i,1],sigma))

    #sort by z value, which will be in accent order
    results = np.sort(results, order='z')
```

```
    #get x0 by using best two points
    x0 = (results[0]['x1']+results[1]['x1'])/2.0
    y0 = (results[0]['x2']+results[1]['x2'])/2.0
    z0 = get_f(x0,y0,sigma)

    #use x0 and worst point x2 to get xr, x0 = x0 + alpha*(x0-x2), where alpha = 1
    xr = 2*x0-results[2]['x1']
    yr = 2*y0-results[2]['x2']
    zr = get_f(xr,yr,sigma)

    #if point r is better than second best, but worse than the best, just replace x2 and return the new vertices
    if zr>=results[0]['z'] and zr<results[1]['z']:
        results[2]['x1'] = xr
        results[2]['x2'] = yr
        results[2]['z'] = zr
    #if point r is better than best, need expansion
    elif zr<results[0]['z']:
        #expand the vertices by calculate point e, where xe = x0 + gamma*(xr-x0), where gamma = 2
        xe = 2*xr-x0
        ye = 2*yr-y0
        ze = get_f(xe,ye,sigma)

        #if ze<zr, use point e to replace x2
        if ze<zr:
            results[2]['x1'] = xe
            results[2]['x2'] = ye
            results[2]['z'] = ze
        else:
            #use xr to replace x2
            results[2]['x1'] = xr
            results[2]['x2'] = yr
            results[2]['z'] = zr
    #if zr is worse than the second worst,need contraction
    elif zr>=results[1]['z']:
        #xc = x0 + row*(x2-x0),where row = 0.5
        xc = (x0+results[2]['x1'])/2
        yc = (y0+results[2]['x2'])/2
        zc = get_f(xc,yc,sigma)

        if zc<results[2]['z']:
            results[2]['x1'] = xc
            results[2]['x2'] = yc
            results[2]['z'] = zc
    #shrink
    else:
        #xi = x0+sigma(xi-x0),where sigma = 0.5
       x1 =   (results[0]['x1']+results[1]['x1'])/2.0
       y1 =   (results[0]['x2']+results[1]['x2'])/2.0
       x2 =   (results[0]['x1']+results[2]['x1'])/2.0
       y2 =   (results[0]['x2']+results[2]['x2'])/2.0
       results[1]['x1'] = x1
       results[1]['x2'] = y1
       results[1]['z'] = get_f(x1,y1,sigma)
       results[2]['x1'] = x2
       results[2]['x2'] = y2
       results[2]['z'] = get_f(x2,y2,sigma)

     #sort the result in z`s order
    results = np.sort(results, order='z')

    #take out the x and y coordinate
    ans = np.zeros((3,2))

    for i in range(3):
        ans[i,0] = results[i]['x1']
        ans[i,1] = results[i]['x2']


    return ans

def nelderMead(sigma):
    delta = 0.025

    x = y = np.arange(-6.0, 6.0, delta)

    X, Y = np.meshgrid(x, y)

    Z = 0.045*X**4-X**2+0.5*X+0.065*Y**4-Y**2+0.5*Y+0.3*X*Y

    Z = np.ma.array(Z)
```

```
    vertices = np.array([[0.,0.],
                         [0.,1],
                         [-1,0]])

    minValue = -10000
    finalX = 0
    finalY = 0

    for i in range(50):
        #plot vertices
        x1, y1 = [vertices[0,0], vertices[1,0]], [vertices[0,1], vertices[1,1]]
        x2, y2 = [vertices[1,0], vertices[2,0]], [vertices[1,1], vertices[2,1]]
        x3, y3 = [vertices[2,0], vertices[0,0]], [vertices[2,1], vertices[0,1]]
        z = np.zeros((3,1))
        for j in range(3):
            z[j,0] = get_f(vertices[j,0],vertices[j,1],sigma)

        std = np.std(z)

        if std<0.1:
            minValue= np.mean(z)
            finalX = (vertices[0,0]+vertices[1,0]+vertices[2,0])/3
            finalY = (vertices[0,1]+vertices[1,1]+vertices[2,1])/3
            break

        #call function to calculate new vertices
        vertices = step_neldner(f=get_f, vertices=vertices,sigma=sigma)

    return {'min':minValue,'x':finalX,'y':finalY}
'''
* @name: NoisyTest
* @description: get the result based on different sigmas
'''
def NoisyTest():
    localMinima = np.array([[-3.63460063, 2.91071661],
                            [-3.25733016, -2.64544543],
                            [3.43582195, -3.0972582],
                            [2.98019805, 2.31968217]])
    count1 = []
    count2 = []
    sigma = np.linspace(0, 5, 11)
    for item in sigma:
        cur_count1 = 0
        cur_count2 = 0

        for i in range(1000):
            result = nelderMead(item)

            distance = []

            for j in range(4):
                temp = (result['x']-localMinima[j][0])**2+(result['y']-localMinima[j][1])**2
                distance.append(temp)

            if distance[0] < 1:
                cur_count1 = cur_count1+1
                cur_count2 = cur_count2+1
            elif distance[1]<1 or distance[2]<1 or distance[3]<1:
                cur_count2 = cur_count2+1

        count1.append(cur_count1)
        count2.append(cur_count2)


    fig, axs = plt.subplots(2, 1)
    axs[0].plot(sigma,count1)
    axs[0].set_xlabel('sigma')
    axs[0].set_ylabel('same local minima')

    axs[1].plot(sigma,count2)
    axs[1].set_xlabel('sigma')
    axs[1].set_ylabel('any local minima')

    fig.tight_layout()
    plt.show()
    #print(count1)
    #print(count2)

    return count1,count2


a,b=NoisyTest()
```

(3) part (c)

```
'''
* @name: get_NoisyF_Sigma
* @description: return a function based the input sigma
* @param y: the input value, in format of [CA,T]
* @return: list, [dCA/dt,dT/dt]
'''
def get_NoisyF_Sigma(sigma):
    a = sigma
    v = lambda x: (0.045 * x[0]**4 - x[0]**2 + 0.5 * x[0] + 0.065 * x[1]**4 - x[1]**2 + 0.5 * x[1] + 0.3 * x[0] * x[1])+np.random.normal(0,a)
    return v
'''
* @name: NoisyTest
* @description: get the result based on different sigmas
'''
'''
* @name: getDiff
* @description: return a function based the input sigma
* @param y: the input value, in format of [CA,T]
* @return: list, [dCA/dt,dT/dt]
'''
def getDiff(x):
    diff1 = 0.18*x[0]*x[0]*x[0]-2*x[0]+0.5+0.3*x[1]
    diff2 = 0.26*x[1]*x[1]*x[1]-2*x[1]+0.5+0.3*x[0]
    return np.array([diff1,diff2])

def NoisyTest():
    localMinima = np.array([[-3.63460063, 2.91071661],
                            [3.43582195, -3.0972582],
                            [-3.25733016, -2.64544543],
                            [2.98019805, 2.31968217]])
    count1 = []
    count2 = []
    sigma = np.linspace(0, 5, 11)


    vertices = np.array([[0.,0.],
                         [0.,1],
                         [-1,0]])
    for item in sigma:
        cur_count1 = 0
        cur_count2 = 0

        for i in range(1000):

            a = get_NoisyF_Sigma(item)

            res = minimize(a,[-0.3,0.3],method = 'BFGS',jac = getDiff)

            #res = minimize(a,[-0.3,0.3],method = 'Nelder-Mead',tol=0.00001,options={'initial_simplex':vertices})

            result = res.x

            distance = []

            for j in range(4):
                temp = (result[0]-localMinima[j][0])**2+(result[1]-localMinima[j][1])**2
                distance.append(temp)

            if distance[0] < 1:
                cur_count1 = cur_count1+1
                cur_count2 = cur_count2+1
            elif distance[1]<1 or distance[2]<1 or distance[3]<1:
                cur_count2 = cur_count2+1

        count1.append(cur_count1)
        count2.append(cur_count2)


    fig, axs = plt.subplots(2, 1)
    axs[0].plot(sigma,count1)
    axs[0].set_xlabel('sigma')
    axs[0].set_ylabel('same local minima')

    axs[1].plot(sigma,count2)
    axs[1].set_xlabel('sigma')
    axs[1].set_ylabel('any local minima')

    fig.tight_layout()
    plt.show()
    print(count1)
```

```
        print(count2)

        return count1,count2

a,b = NoisyTest()
print(a)
print(b)
```