실습)
Omega & XI

```
# -----------
# User Instructions
#
# Write a function, doit, that takes as its input an
# initial robot position, move1, and move2. This
# function should compute the Omega and Xi matrices
# discussed in lecture and should RETURN the mu vector
# (which is the product of Omega.inverse() and Xi).
#
# Please enter your code at the bottom.



from math import *
import random


#========================================================================
#
# SLAM in a rectolinear world (we avoid non-linearities)
#
#
#========================================================================



# ------------------------------------------------
#
# this is the matrix class
# we use it because it makes it easier to collect constraints in GraphSLAM
# and to calculate solutions (albeit inefficiently)
#

class matrix:

    # implements basic operations of a matrix class

    # ------------
```

```
    #
    # initialization - can be called with an initial matrix
    #

    def __init__(self, value = [[]]):
        self.value = value
        self.dimx  = len(value)
        self.dimy  = len(value[0])
        if value == [[]]:
            self.dimx = 0

    # ------------
    #
    # makes matrix of a certain size and sets each element to zero
    #

    def zero(self, dimx, dimy = 0):
        if dimy == 0:
            dimy = dimx
        # check if valid dimensions
        if dimx < 1 or dimy < 1:
            raise ValueError("Invalid size of matrix")
        else:
            self.dimx  = dimx
            self.dimy  = dimy
            self.value = [[0.0 for row in range(dimy)] for col in
range(dimx)]

    # ------------
    #
    # makes matrix of a certain (square) size and turns matrix into
identity matrix
    #


    def identity(self, dim):
        # check if valid dimension
        if dim < 1:
            raise ValueError( "Invalid size of matrix")
        else:
```

```
            self.dimx  = dim
            self.dimy  = dim
            self.value = [[0.0 for row in range(dim)] for col in
range(dim)]
            for i in range(dim):
                self.value[i][i] = 1.0
    # ------------
    #
    # prints out values of matrix
    #


    def show(self, txt = ''):
        for i in range(len(self.value)):
            print(txt + '['+ ', '.join('%.3f'%x for x in self.value[i]) +
']' )
        print(' ')

    # ------------
    #
    # defines elmement-wise matrix addition. Both matrices must be of
equal dimensions
    #


    def __add__(self, other):
        # check if correct dimensions
        if self.dimx != other.dimx or self.dimx != other.dimx:
            raise ValueError("Matrices must be of equal dimension to add")
        else:
            # add if correct dimensions
            res = matrix()
            res.zero(self.dimx, self.dimy)
            for i in range(self.dimx):
                for j in range(self.dimy):
                    res.value[i][j] = self.value[i][j] + other.value[i][j]
            return res

    # ------------
    #
```

```
    # defines elmement-wise matrix subtraction. Both matrices must be of
equal dimensions
    #

    def __sub__(self, other):
        # check if correct dimensions
        if self.dimx != other.dimx or self.dimx != other.dimx:
            raise ValueError("Matrices must be of equal dimension to
subtract")
        else:
            # subtract if correct dimensions
            res = matrix()
            res.zero(self.dimx, self.dimy)
            for i in range(self.dimx):
                for j in range(self.dimy):
                    res.value[i][j] = self.value[i][j] - other.value[i][j]
            return res

    # ------------
    #
    # defines multiplication. Both matrices must be of fitting dimensions
    #


    def __mul__(self, other):
        # check if correct dimensions
        if self.dimy != other.dimx:
            raise ValueError("Matrices must be m*n and n*p to multiply")
        else:
            # multiply if correct dimensions
            res = matrix()
            res.zero(self.dimx, other.dimy)
            for i in range(self.dimx):
                for j in range(other.dimy):
                    for k in range(self.dimy):
                        res.value[i][j] += self.value[i][k] *
other.value[k][j]
        return res
```

```
    # ------------
    #
    # returns a matrix transpose
    #


    def transpose(self):
        # compute transpose
        res = matrix()
        res.zero(self.dimy, self.dimx)
        for i in range(self.dimx):
            for j in range(self.dimy):
                res.value[j][i] = self.value[i][j]
        return res

    # ------------
    #
    # creates a new matrix from the existing matrix elements.
    #
    # Example:
    #       l = matrix([[ 1,  2,  3,  4,  5],
    #                   [ 6,  7,  8,  9, 10],
    #                   [11, 12, 13, 14, 15]])
    #
    #       l.take([0, 2], [0, 2, 3])
    #
    # results in:
    #
    #       [[1, 3, 4],
    #        [11, 13, 14]]
    #
    #
    # take is used to remove rows and columns from existing matrices
    # list1/list2 define a sequence of rows/columns that shall be taken
    # is no list2 is provided, then list2 is set to list1 (good for
symmetric matrices)
    #

    def take(self, list1, list2 = []):
        if list2 == []:
```

```
            list2 = list1
        if len(list1) > self.dimx or len(list2) > self.dimy:
            raise ValueError("list invalid in take()")

        res = matrix()
        res.zero(len(list1), len(list2))
        for i in range(len(list1)):
            for j in range(len(list2)):
                res.value[i][j] = self.value[list1[i]][list2[j]]
        return res

    # ------------
    #
    # creates a new matrix from the existing matrix elements.
    #
    # Example:
    #       l = matrix([[1, 2, 3],
    #                   [4, 5, 6]])
    #
    #       l.expand(3, 5, [0, 2], [0, 2, 3])
    #
    # results in:
    #
    #       [[1, 0, 2, 3, 0],
    #        [0, 0, 0, 0, 0],
    #        [4, 0, 5, 6, 0]]
    #
    # expand is used to introduce new rows and columns into an existing
matrix
    # list1/list2 are the new indexes of row/columns in which the matrix
    # elements are being mapped. Elements for rows and columns
    # that are not listed in list1/list2
    # will be initialized by 0.0.
    #

    def expand(self, dimx, dimy, list1, list2 = []):
        if list2 == []:
            list2 = list1
        if len(list1) > self.dimx or len(list2) > self.dimy:
            raise ValueError("list invalid in expand()")
```

```
        res = matrix()
        res.zero(dimx, dimy)
        for i in range(len(list1)):
            for j in range(len(list2)):
                res.value[list1[i]][list2[j]] = self.value[i][j]
        return res

    # ------------
    #
    # Computes the upper triangular Cholesky factorization of
    # a positive definite matrix.
    # This code is based on http://adorio-research.org/wordpress/?p=4560

    def Cholesky(self, ztol= 1.0e-5):
        res = matrix()
        res.zero(self.dimx, self.dimx)

        for i in range(self.dimx):
            S = sum([(res.value[k][i])**2 for k in range(i)])
            d = self.value[i][i] - S
            if abs(d) < ztol:
                res.value[i][i] = 0.0
            else:
                if d < 0.0:
                    raise ValueError("Matrix not positive-definite")
                res.value[i][i] = sqrt(d)
            for j in range(i+1, self.dimx):
                S = sum([res.value[k][i] * res.value[k][j] for k in
range(i)])
                if abs(S) < ztol:
                    S = 0.0
                res.value[i][j] = (self.value[i][j] - S)/res.value[i][i]
        return res

    # ------------
    #
    # Computes inverse of matrix given its Cholesky upper Triangular
    # decomposition of matrix.
    # This code is based on http://adorio-research.org/wordpress/?p=4560
```

```
    def CholeskyInverse(self):
        res = matrix()
        res.zero(self.dimx, self.dimx)

    # Backward step for inverse.
        for j in reversed(range(self.dimx)):
            tjj = self.value[j][j]
            S = sum([self.value[j][k]*res.value[j][k] for k in range(j+1,
self.dimx)])
            res.value[j][j] = 1.0/ tjj**2 - S/ tjj
            for i in reversed(range(j)):
                res.value[j][i] = res.value[i][j] = \
                    -sum([self.value[i][k]*res.value[k][j] for k in \
                              range(i+1,self.dimx)])/self.value[i][i]
        return res

    # ------------
    #
    # comutes and returns the inverse of a square matrix
    #


    def inverse(self):
        aux = self.Cholesky()
        res = aux.CholeskyInverse()
        return res

    # ------------
    #
    # prints matrix (needs work!)
    #


    def __repr__(self):
        return repr(self.value)



# #########################################################################
```

```
# ########################################################################
# ########################################################################


"""
For the following example, you would call doit(-3, 5, 3):
3 robot positions
  initially: -3
  moves by 5
  moves by 3



which should return a mu of:
[[-3.0],
 [2.0],
 [5.0]]
"""
def doit(initial_pos, move1, move2):
    #
    #
    # Add your code here.
    #
    #
    omega = matrix([[2,-1,0],[-1,2,-1],[0,-1,1]])
    xi = matrix([[-8],[2],[3]])
    mu = omega.inverse() * xi
    print("omega:",omega)
    print("xi:",xi)
    print("mu:",mu)

    # oops, I guess I hard coded the whole thing. Let's try again :)

    omega = matrix([[1,0,0],[0,0,0],[0,0,0]])
    xi = matrix([[initial_pos],[0],[0]])
    omega += matrix([[1,-1,0],[-1,1,0],[0,0,0]])
    xi += matrix([[-move1],[move1],[0]])
    omega += matrix([[0,0,0],[0,1,-1],[0,-1,1]])
    xi += matrix([[0],[-move2],[move2]])
    print("omega:",omega)
```

```
    print("xi:",xi)
    print("mu:",mu)

    return mu

doit(-3, 5, 3)
```

```
omega: [[2, -1, 0], [-1, 2, -1], [0, -1, 1]]
xi: [[-8], [2], [3]]
mu: [[-2.999999999999999], [2.0000000000000018], [5.000000000000002]]
omega: [[2, -1, 0], [-1, 2, -1], [0, -1, 1]]
xi: [[-8], [2], [3]]
mu: [[-2.999999999999999], [2.0000000000000018], [5.000000000000002]]
PS C:\work\python_work>
```

실습)
Expands

```
# -----------
# User Instructions
#
# Modify your doit function to incorporate 3
# distance measurements to a landmark(Z0, Z1, Z2).
# You should use the provided expand function to
# allow your Omega and Xi matrices to accomodate
# the new information.
#
# Each landmark measurement should modify 4
# values in your Omega matrix and 2 in your
# Xi vector.
#
# Add your code at line 356.

from math import *
import random

#============================================================
#
# SLAM in a rectolinear world (we avoid non-linearities)
#
```

```
#
#=========================================================================



# ----------------------------------------------
#
# this is the matrix class
# we use it because it makes it easier to collect constraints in GraphSLAM
# and to calculate solutions (albeit inefficiently)
#

class matrix:

    # implements basic operations of a matrix class

    # ------------
    #
    # initialization - can be called with an initial matrix
    #

    def __init__(self, value = [[]]):
        self.value = value
        self.dimx  = len(value)
        self.dimy  = len(value[0])
        if value == [[]]:
            self.dimx = 0

    # ------------
    #
    # makes matrix of a certain size and sets each element to zero
    #

    def zero(self, dimx, dimy = 0):
        if dimy == 0:
            dimy = dimx
        # check if valid dimensions
        if dimx < 1 or dimy < 1:
            raise ValueError( "Invalid size of matrix")
        else:
            self.dimx  = dimx
```

```
            self.dimy  = dimy
            self.value = [[0.0 for row in range(dimy)] for col in
range(dimx)]

    # ------------
    #
    # makes matrix of a certain (square) size and turns matrix into
identity matrix
    #


    def identity(self, dim):
        # check if valid dimension
        if dim < 1:
            raise ValueError( "Invalid size of matrix")
        else:
            self.dimx  = dim
            self.dimy  = dim
            self.value = [[0.0 for row in range(dim)] for col in
range(dim)]
            for i in range(dim):
                self.value[i][i] = 1.0
    # ------------
    #
    # prints out values of matrix
    #


    def show(self, txt = ''):
        for i in range(len(self.value)):
            print (txt + '['+ ', '.join('%.3f'%x for x in self.value[i]) +
']' )
        print(' ')

    # ------------
    #
    # defines elmement-wise matrix addition. Both matrices must be of
equal dimensions
    #
```

```
    def __add__(self, other):
        # check if correct dimensions
        if self.dimx != other.dimx or self.dimx != other.dimx:
            raise ValueError( "Matrices must be of equal dimension to
add")
        else:
            # add if correct dimensions
            res = matrix()
            res.zero(self.dimx, self.dimy)
            for i in range(self.dimx):
                for j in range(self.dimy):
                    res.value[i][j] = self.value[i][j] + other.value[i][j]
            return res

    # ------------
    #
    # defines elmement-wise matrix subtraction. Both matrices must be of
equal dimensions
    #

    def __sub__(self, other):
        # check if correct dimensions
        if self.dimx != other.dimx or self.dimx != other.dimx:
            raise ValueError( "Matrices must be of equal dimension to
subtract")
        else:
            # subtract if correct dimensions
            res = matrix()
            res.zero(self.dimx, self.dimy)
            for i in range(self.dimx):
                for j in range(self.dimy):
                    res.value[i][j] = self.value[i][j] - other.value[i][j]
            return res

    # ------------
    #
    # defines multiplication. Both matrices must be of fitting dimensions
    #
```

```
    def __mul__(self, other):
        # check if correct dimensions
        if self.dimy != other.dimx:
            raise ValueError( "Matrices must be m*n and n*p to multiply")
        else:
            # multiply if correct dimensions
            res = matrix()
            res.zero(self.dimx, other.dimy)
            for i in range(self.dimx):
                for j in range(other.dimy):
                    for k in range(self.dimy):
                        res.value[i][j] += self.value[i][k] *
other.value[k][j]
        return res


    # ------------
    #
    # returns a matrix transpose
    #


    def transpose(self):
        # compute transpose
        res = matrix()
        res.zero(self.dimy, self.dimx)
        for i in range(self.dimx):
            for j in range(self.dimy):
                res.value[j][i] = self.value[i][j]
        return res

    # ------------
    #
    # creates a new matrix from the existing matrix elements.
    #
    # Example:
    #       l = matrix([[ 1,  2,  3,  4,  5],
    #                   [ 6,  7,  8,  9, 10],
    #                   [11, 12, 13, 14, 15]])
```

```
    #
    #       l.take([0, 2], [0, 2, 3])
    #
    # results in:
    #
    #       [[1, 3, 4],
    #        [11, 13, 14]]
    #
    #
    # take is used to remove rows and columns from existing matrices
    # list1/list2 define a sequence of rows/columns that shall be taken
    # is no list2 is provided, then list2 is set to list1 (good for
symmetric matrices)
    #

    def take(self, list1, list2 = []):
        if list2 == []:
            list2 = list1
        if len(list1) > self.dimx or len(list2) > self.dimy:
            raise ValueError( "list invalid in take()")

        res = matrix()
        res.zero(len(list1), len(list2))
        for i in range(len(list1)):
            for j in range(len(list2)):
                res.value[i][j] = self.value[list1[i]][list2[j]]
        return res

    # ------------
    #
    # creates a new matrix from the existing matrix elements.
    #
    # Example:
    #       l = matrix([[1, 2, 3],
    #                   [4, 5, 6]])
    #
    #       l.expand(3, 5, [0, 2], [0, 2, 3])
    #
    # results in:
    #
```

```
    #        [[1, 0, 2, 3, 0],
    #         [0, 0, 0, 0, 0],
    #         [4, 0, 5, 6, 0]]
    #
    # expand is used to introduce new rows and columns into an existing
matrix
    # list1/list2 are the new indexes of row/columns in which the matrix
    # elements are being mapped. Elements for rows and columns
    # that are not listed in list1/list2
    # will be initialized by 0.0.
    #

    def expand(self, dimx, dimy, list1, list2 = []):
        if list2 == []:
            list2 = list1
        if len(list1) > self.dimx or len(list2) > self.dimy:
            raise ValueError( "list invalid in expand()")

        res = matrix()
        res.zero(dimx, dimy)
        for i in range(len(list1)):
            for j in range(len(list2)):
                res.value[list1[i]][list2[j]] = self.value[i][j]
        return res

    # ------------
    #
    # Computes the upper triangular Cholesky factorization of
    # a positive definite matrix.
    # This code is based on http://adorio-research.org/wordpress/?p=4560

    def Cholesky(self, ztol= 1.0e-5):


        res = matrix()
        res.zero(self.dimx, self.dimx)

        for i in range(self.dimx):
            S = sum([(res.value[k][i])**2 for k in range(i)])
            d = self.value[i][i] - S
```

```
            if abs(d) < ztol:
                res.value[i][i] = 0.0
            else:
                if d < 0.0:
                    raise ValueError( "Matrix not positive-definite")
                res.value[i][i] = sqrt(d)
            for j in range(i+1, self.dimx):
                S = sum([res.value[k][i] * res.value[k][j] for k in
range(i)])
                if abs(S) < ztol:
                    S = 0.0
                res.value[i][j] = (self.value[i][j] - S)/res.value[i][i]
        return res

    # ------------
    #
    # Computes inverse of matrix given its Cholesky upper Triangular
    # decomposition of matrix.
    # This code is based on http://adorio-research.org/wordpress/?p=4560

    def CholeskyInverse(self):
    # Computes inverse of matrix given its Cholesky upper Triangular
    # decomposition of matrix.
        # This code is based on
http://adorio-research.org/wordpress/?p=4560

        res = matrix()
        res.zero(self.dimx, self.dimx)

    # Backward step for inverse.
        for j in reversed(range(self.dimx)):
            tjj = self.value[j][j]
            S = sum([self.value[j][k]*res.value[j][k] for k in range(j+1,
self.dimx)])
            res.value[j][j] = 1.0/ tjj**2 - S/ tjj
            for i in reversed(range(j)):
                res.value[j][i] = res.value[i][j] = \
                    -sum([self.value[i][k]*res.value[k][j] for k in \
                                range(i+1,self.dimx)])/self.value[i][i]
        return res
```

```
    # ------------
    #
    # comutes and returns the inverse of a square matrix
    #


    def inverse(self):
        aux = self.Cholesky()
        res = aux.CholeskyInverse()
        return res

    # ------------
    #
    # prints matrix (needs work!)
    #


    def __repr__(self):
        return repr(self.value)



# ######################################################################
# ######################################################################
# ######################################################################


"""
For the following example, you would call doit(-3, 5, 3, 10, 5, 2):
3 robot positions
  initially: -3 (measure landmark to be 10 away)
  moves by 5 (measure landmark to be 5 away)
  moves by 3 (measure landmark to be 2 away)



which should return a mu of:
[[-3.0],
 [2.0],
```

```
 [5.0],
 [7.0]]
"""
def doit(initial_pos, move1, move2, Z0, Z1, Z2):
    Omega = matrix([[1.0, 0.0, 0.0],
                    [0.0, 0.0, 0.0],
                    [0.0, 0.0, 0.0]])
    Xi = matrix([[initial_pos],
                 [0.0],
                 [0.0]])

    Omega += matrix([[1.0, -1.0, 0.0],
                     [-1.0, 1.0, 0.0],
                     [0.0, 0.0, 0.0]])
    Xi += matrix([[-move1],
                  [move1],
                  [0.0]])

    Omega += matrix([[0.0, 0.0, 0.0],
                     [0.0, 1.0, -1.0],
                     [0.0, -1.0, 1.0]])
    Xi += matrix([[0.0],
                  [-move2],
                  [move2]])

    #
    #
    # Add your code here.
    #
    #
    Omega = Omega.expand(4,4,[0,1,2],[0,1,2])
    Xi = Xi.expand(4,1,[0,1,2],[0])

    Omega += matrix([[1.0, 0.0, 0.0, -1.0],
                     [0.0, 0.0, 0.0, 0.0],
                     [0.0, 0.0, 0.0, 0.0],
                     [-1.0, 0.0, 0.0, 1.0]])
    Xi += matrix([[-Z0],
                  [0.0],
                  [0.0],
```

```
                  [Z0]])
    Omega += matrix([[0.0, 0.0, 0.0, 0.0],
                     [0.0, 1.0, 0.0, -1.0],
                     [0.0, 0.0, 0.0, 0.0],
                     [0.0, -1.0, 0.0, 1.0]])
    Xi += matrix([[0.0],
                  [-Z1],
                  [0.0],
                  [Z1]])
    Omega += matrix([[0.0, 0.0, 0.0, 0.0],
                     [0.0, 0.0, 0.0, 0.0],
                     [0.0, 0.0, 1.0, -1.0],
                     [0.0, 0.0, -1.0, 1.0]])
    Xi += matrix([[0.0],
                  [0.0],
                  [-Z2],
                  [Z2]])
    Omega.show('Omega: ')
    Xi.show('Xi:    ')
    mu = Omega.inverse() * Xi
    mu.show('Mu:    ')

    return mu

doit(-3, 5, 3, 10, 5, 2)
```

```
Omega: [3.000, -1.000, 0.000, -1.000]
Omega: [-1.000, 3.000, -1.000, -1.000]
Omega: [0.000, -1.000, 2.000, -1.000]
Omega: [-1.000, -1.000, -1.000, 3.000]

Xi:    [-18.000]
Xi:    [-3.000]
Xi:    [1.000]
Xi:    [17.000]

Mu:    [-3.000]
Mu:    [2.000]
Mu:    [5.000]
Mu:    [7.000]
```

실습)
Confident measurement

```
# -----------
# User Instructions
#
# Modify the previous code to adjust for a highly
# confident last measurement. Do this by adding a
# factor of 5 into your Omega and Xi matrices
# as described in the video.



from math import *
import random


#==========================================================
#
# SLAM in a rectolinear world (we avoid non-linearities)
#
#
#==========================================================



# ----------------------------------------------
#
# this is the matrix class
# we use it because it makes it easier to collect constraints in GraphSLAM
# and to calculate solutions (albeit inefficiently)
#

class matrix:

    # implements basic operations of a matrix class

    # ------------
    #
    # initialization - can be called with an initial matrix
    #
```

```
    def __init__(self, value = [[]]):
        self.value = value
        self.dimx  = len(value)
        self.dimy  = len(value[0])
        if value == [[]]:
            self.dimx = 0

    # ------------
    #
    # makes matrix of a certain size and sets each element to zero
    #

    def zero(self, dimx, dimy = 0):
        if dimy == 0:
            dimy = dimx
        # check if valid dimensions
        if dimx < 1 or dimy < 1:
            raise ValueError("Invalid size of matrix")
        else:
            self.dimx  = dimx
            self.dimy  = dimy
            self.value = [[0.0 for row in range(dimy)] for col in
range(dimx)]

    # ------------
    #
    # makes matrix of a certain (square) size and turns matrix into
identity matrix
    #


    def identity(self, dim):
        # check if valid dimension
        if dim < 1:
            raise ValueError( "Invalid size of matrix")
        else:
            self.dimx  = dim
            self.dimy  = dim
```

```
            self.value = [[0.0 for row in range(dim)] for col in
range(dim)]
            for i in range(dim):
                self.value[i][i] = 1.0
    # ------------
    #
    # prints out values of matrix
    #


    def show(self, txt = ''):
        for i in range(len(self.value)):
            print (txt + '['+ ', '.join('%.3f'%x for x in self.value[i]) +
']' )
        print(' ')

    # ------------
    #
    # defines elmement-wise matrix addition. Both matrices must be of
equal dimensions
    #


    def __add__(self, other):
        # check if correct dimensions
        if self.dimx != other.dimx or self.dimx != other.dimx:
            raise ValueError( "Matrices must be of equal dimension to
add")
        else:
            # add if correct dimensions
            res = matrix()
            res.zero(self.dimx, self.dimy)
            for i in range(self.dimx):
                for j in range(self.dimy):
                    res.value[i][j] = self.value[i][j] + other.value[i][j]
            return res

    # ------------
    #
```

```
    # defines elmement-wise matrix subtraction. Both matrices must be of
equal dimensions
    #

    def __sub__(self, other):
        # check if correct dimensions
        if self.dimx != other.dimx or self.dimx != other.dimx:
            raise ValueError( "Matrices must be of equal dimension to
subtract")
        else:
            # subtract if correct dimensions
            res = matrix()
            res.zero(self.dimx, self.dimy)
            for i in range(self.dimx):
                for j in range(self.dimy):
                    res.value[i][j] = self.value[i][j] - other.value[i][j]
            return res

    # ------------
    #
    # defines multiplication. Both matrices must be of fitting dimensions
    #


    def __mul__(self, other):
        # check if correct dimensions
        if self.dimy != other.dimx:
            raise ValueError( "Matrices must be m*n and n*p to multiply")
        else:
            # multiply if correct dimensions
            res = matrix()
            res.zero(self.dimx, other.dimy)
            for i in range(self.dimx):
                for j in range(other.dimy):
                    for k in range(self.dimy):
                        res.value[i][j] += self.value[i][k] *
other.value[k][j]
        return res
```

```
    # ------------
    #
    # returns a matrix transpose
    #


    def transpose(self):
        # compute transpose
        res = matrix()
        res.zero(self.dimy, self.dimx)
        for i in range(self.dimx):
            for j in range(self.dimy):
                res.value[j][i] = self.value[i][j]
        return res

    # ------------
    #
    # creates a new matrix from the existing matrix elements.
    #
    # Example:
    #       l = matrix([[ 1,  2,  3,  4,  5],
    #                   [ 6,  7,  8,  9, 10],
    #                   [11, 12, 13, 14, 15]])
    #
    #       l.take([0, 2], [0, 2, 3])
    #
    # results in:
    #
    #       [[1, 3, 4],
    #        [11, 13, 14]]
    #
    #
    # take is used to remove rows and columns from existing matrices
    # list1/list2 define a sequence of rows/columns that shall be taken
    # is no list2 is provided, then list2 is set to list1 (good for
symmetric matrices)
    #

    def take(self, list1, list2 = []):
        if list2 == []:
```

```
            list2 = list1
        if len(list1) > self.dimx or len(list2) > self.dimy:
            raise ValueError( "list invalid in take()")

        res = matrix()
        res.zero(len(list1), len(list2))
        for i in range(len(list1)):
            for j in range(len(list2)):
                res.value[i][j] = self.value[list1[i]][list2[j]]
        return res

    # ------------
    #
    # creates a new matrix from the existing matrix elements.
    #
    # Example:
    #       l = matrix([[1, 2, 3],
    #                   [4, 5, 6]])
    #
    #       l.expand(3, 5, [0, 2], [0, 2, 3])
    #
    # results in:
    #
    #       [[1, 0, 2, 3, 0],
    #        [0, 0, 0, 0, 0],
    #        [4, 0, 5, 6, 0]]
    #
    # expand is used to introduce new rows and columns into an existing
matrix
    # list1/list2 are the new indexes of row/columns in which the matrix
    # elements are being mapped. Elements for rows and columns
    # that are not listed in list1/list2
    # will be initialized by 0.0.
    #

    def expand(self, dimx, dimy, list1, list2 = []):
        if list2 == []:
            list2 = list1
        if len(list1) > self.dimx or len(list2) > self.dimy:
            raise ValueError( "list invalid in expand()")
```

```
        res = matrix()
        res.zero(dimx, dimy)
        for i in range(len(list1)):
            for j in range(len(list2)):
                res.value[list1[i]][list2[j]] = self.value[i][j]
        return res

    # ------------
    #
    # Computes the upper triangular Cholesky factorization of
    # a positive definite matrix.
    # This code is based on http://adorio-research.org/wordpress/?p=4560

    def Cholesky(self, ztol= 1.0e-5):


        res = matrix()
        res.zero(self.dimx, self.dimx)

        for i in range(self.dimx):
            S = sum([(res.value[k][i])**2 for k in range(i)])
            d = self.value[i][i] - S
            if abs(d) < ztol:
                res.value[i][i] = 0.0
            else:
                if d < 0.0:
                    raise ValueError( "Matrix not positive-definite")
                res.value[i][i] = sqrt(d)
            for j in range(i+1, self.dimx):
                S = sum([res.value[k][i] * res.value[k][j] for k in
range(i)])
                if abs(S) < ztol:
                    S = 0.0
                res.value[i][j] = (self.value[i][j] - S)/res.value[i][i]
        return res

    # ------------
    #
    # Computes inverse of matrix given its Cholesky upper Triangular
```

```
    # decomposition of matrix.
    # This code is based on http://adorio-research.org/wordpress/?p=4560

    def CholeskyInverse(self):
    # Computes inverse of matrix given its Cholesky upper Triangular
    # decomposition of matrix.
        # This code is based on
http://adorio-research.org/wordpress/?p=4560

        res = matrix()
        res.zero(self.dimx, self.dimx)

    # Backward step for inverse.
        for j in reversed(range(self.dimx)):
            tjj = self.value[j][j]
            S = sum([self.value[j][k]*res.value[j][k] for k in range(j+1,
self.dimx)])
            res.value[j][j] = 1.0/ tjj**2 - S/ tjj
            for i in reversed(range(j)):
                res.value[j][i] = res.value[i][j] = \
                    -sum([self.value[i][k]*res.value[k][j] for k in \
                              range(i+1,self.dimx)])/self.value[i][i]
        return res

    # ------------
    #
    # comutes and returns the inverse of a square matrix
    #


    def inverse(self):
        aux = self.Cholesky()
        res = aux.CholeskyInverse()
        return res

    # ------------
    #
    # prints matrix (needs work!)
    #
```

```
    def __repr__(self):
        return repr(self.value)


# ########################################################################
# ########################################################################
# ########################################################################


# Including the 5 times multiplier, your returned mu should now be:
#
# [[-3.0],
#  [2.179],
#  [5.714],
#  [6.821]]



############## MODIFY CODE BELOW ##################

def doit(initial_pos, move1, move2, Z0, Z1, Z2):
    Omega = matrix([[1.0, 0.0, 0.0],
                    [0.0, 0.0, 0.0],
                    [0.0, 0.0, 0.0]])
    Xi    = matrix([[initial_pos],
                    [0.0],
                    [0.0]])

    Omega += matrix([[1.0, -1.0, 0.0],
                     [-1.0, 1.0, 0.0],
                     [0.0, 0.0, 0.0]])
    Xi    += matrix([[-move1],
                     [move1],
                     [0.0]])

    Omega += matrix([[0.0, 0.0, 0.0],
                     [0.0, 1.0, -1.0],
                     [0.0, -1.0, 1.0]])
    Xi    += matrix([[0.0],
```

```
                    [-move2],
                    [move2]])

    Omega = Omega.expand(4, 4, [0, 1, 2], [0, 1, 2])
    Xi =    Xi.expand(4, 1, [0, 1, 2], [0])

    Omega += matrix([[1.0, 0.0, 0.0, -1.0],
                    [0.0, 0.0, 0.0, 0.0],
                    [0.0, 0.0, 0.0, 0.0],
                    [-1.0, 0.0, 0.0, 1.0]])
    Xi    += matrix([[-Z0],
                    [0.0],
                    [0.0],
                    [Z0]])

    Omega += matrix([[0.0, 0.0, 0.0, 0.0],
                    [0.0, 1.0, 0.0, -1.0],
                    [0.0, 0.0, 0.0, 0.0],
                    [0.0, -1.0, 0.0, 1.0]])
    Xi    += matrix([[0.0],
                    [-Z1],
                    [0.0],
                    [Z1]])

    Omega += matrix([[0.0, 0.0, 0.0, 0.0],
                    [0.0, 0.0, 0.0, 0.0],
                    [0.0, 0.0, 1.0*5, -1.0*5],
                    [0.0, 0.0, -1.0*5, 1.0*5]])
    Xi    += matrix([[0.0],
                    [0.0],
                    [-Z2*5],
                    [Z2*5]])

    Omega.show('Omega: ')
    Xi.show('Xi:    ')
    mu = Omega.inverse() * Xi
    mu.show('Mu:    ')

    return mu
```

```
doit(-3, 5, 3, 10, 5, 1)
```

```
Xi:    [-18.000]
Xi:    [-3.000]
Xi:    [-2.000]
Xi:    [20.000]

Mu:    [-3.000]
Mu:    [2.179]
Mu:    [5.714]
Mu:    [6.821]
```

실습)

Implementing SLAM

```
# ------------
# User Instructions
#
# In this problem you will implement SLAM in a 2 dimensional
# world. Please define a function, slam, which takes five
# parameters as input and returns the vector mu. This vector
# should have x, y coordinates interlaced, so for example,
# if there were 2 poses and 2 landmarks, mu would look like:
#
#  mu =  matrix([[Px0],
#                [Py0],
#                [Px1],
#                [Py1],
#                [Lx0],
#                [Ly0],
#                [Lx1],
#                [Ly1]])
#
# data - This is the data that is generated with the included
#        make_data function. You can also use test_data to
#        make sure your function gives the correct result.
#
# N -    The number of time steps.
#
# num_landmarks - The number of landmarks.
#
```

```
# motion_noise - The noise associated with motion. The update
#                strength for motion should be 1.0 / motion_noise.
#
# measurement_noise - The noise associated with measurement.
#                     The update strength for measurement should be
#                     1.0 / measurement_noise.
#
#
# Enter your code at line 509

# --------------
# Testing
#
# Uncomment the test cases at the bottom of this document.
# Your output should be identical to the given results.


from math import *
import random


#=============================================================
#
# SLAM in a rectolinear world (we avoid non-linearities)
#
#
#=============================================================


# ----------------------------------------------
#
# this is the matrix class
# we use it because it makes it easier to collect constraints in GraphSLAM
# and to calculate solutions (albeit inefficiently)
#

class matrix:

    # implements basic operations of a matrix class
```

```
    # ------------
    #
    # initialization - can be called with an initial matrix
    #

    def __init__(self, value = [[]]):
        self.value = value
        self.dimx  = len(value)
        self.dimy  = len(value[0])
        if value == [[]]:
            self.dimx = 0

    # ------------
    #
    # makes matrix of a certain size and sets each element to zero
    #

    def zero(self, dimx, dimy):
        if dimy == 0:
            dimy = dimx
        # check if valid dimensions
        if dimx < 1 or dimy < 1:
            raise ValueError( "Invalid size of matrix")
        else:
            self.dimx  = dimx
            self.dimy  = dimy
            self.value = [[0.0 for row in range(dimy)] for col in
range(dimx)]

    # ------------
    #
    # makes matrix of a certain (square) size and turns matrix into
identity matrix
    #

    def identity(self, dim):
        # check if valid dimension
        if dim < 1:
            raise ValueError( "Invalid size of matrix")
        else:
```

```
            self.dimx  = dim
            self.dimy  = dim
            self.value = [[0.0 for row in range(dim)] for col in
range(dim)]
            for i in range(dim):
                self.value[i][i] = 1.0
    # ------------
    #
    # prints out values of matrix
    #

    def show(self, txt = ''):
        for i in range(len(self.value)):
            print (txt + '['+ ', '.join('%.3f'%x for x in self.value[i]) +
']' )
        print (' ')

    # ------------
    #
    # defines elmement-wise matrix addition. Both matrices must be of
equal dimensions
    #

    def __add__(self, other):
        # check if correct dimensions
        if self.dimx != other.dimx or self.dimx != other.dimx:
            raise ValueError( "Matrices must be of equal dimension to
add")
        else:
            # add if correct dimensions
            res = matrix()
            res.zero(self.dimx, self.dimy)
            for i in range(self.dimx):
                for j in range(self.dimy):
                    res.value[i][j] = self.value[i][j] + other.value[i][j]
            return res

    # ------------
    #
```

```
    # defines elmement-wise matrix subtraction. Both matrices must be of
equal dimensions
    #

    def __sub__(self, other):
        # check if correct dimensions
        if self.dimx != other.dimx or self.dimx != other.dimx:
            raise ValueError( "Matrices must be of equal dimension to
subtract")
        else:
            # subtract if correct dimensions
            res = matrix()
            res.zero(self.dimx, self.dimy)
            for i in range(self.dimx):
                for j in range(self.dimy):
                    res.value[i][j] = self.value[i][j] - other.value[i][j]
            return res

    # ------------
    #
    # defines multiplication. Both matrices must be of fitting dimensions
    #

    def __mul__(self, other):
        # check if correct dimensions
        if self.dimy != other.dimx:
            raise ValueError( "Matrices must be m*n and n*p to multiply")
        else:
            # multiply if correct dimensions
            res = matrix()
            res.zero(self.dimx, other.dimy)
            for i in range(self.dimx):
                for j in range(other.dimy):
                    for k in range(self.dimy):
                        res.value[i][j] += self.value[i][k] *
other.value[k][j]
        return res


    # ------------
```

```
    #
    # returns a matrix transpose
    #

    def transpose(self):
        # compute transpose
        res = matrix()
        res.zero(self.dimy, self.dimx)
        for i in range(self.dimx):
            for j in range(self.dimy):
                res.value[j][i] = self.value[i][j]
        return res

    # ------------
    #
    # creates a new matrix from the existing matrix elements.
    #
    # Example:
    #       l = matrix([[ 1,  2,  3,  4,  5],
    #                   [ 6,  7,  8,  9, 10],
    #                   [11, 12, 13, 14, 15]])
    #
    #       l.take([0, 2], [0, 2, 3])
    #
    # results in:
    #
    #       [[1, 3, 4],
    #        [11, 13, 14]]
    #
    #
    # take is used to remove rows and columns from existing matrices
    # list1/list2 define a sequence of rows/columns that shall be taken
    # if no list2 is provided, then list2 is set to list1 (good for
    # symmetric matrices)
    #

    def take(self, list1, list2 = []):
        if list2 == []:
            list2 = list1
        if len(list1) > self.dimx or len(list2) > self.dimy:
```

```
            raise ValueError( "list invalid in take()")

        res = matrix()
        res.zero(len(list1), len(list2))
        for i in range(len(list1)):
            for j in range(len(list2)):
                res.value[i][j] = self.value[list1[i]][list2[j]]
        return res

    # ------------
    #
    # creates a new matrix from the existing matrix elements.
    #
    # Example:
    #       l = matrix([[1, 2, 3],
    #                   [4, 5, 6]])
    #
    #       l.expand(3, 5, [0, 2], [0, 2, 3])
    #
    # results in:
    #
    #       [[1, 0, 2, 3, 0],
    #        [0, 0, 0, 0, 0],
    #        [4, 0, 5, 6, 0]]
    #
    # expand is used to introduce new rows and columns into an existing
matrix
    # list1/list2 are the new indexes of row/columns in which the matrix
    # elements are being mapped. Elements for rows and columns
    # that are not listed in list1/list2
    # will be initialized by 0.0.
    #

    def expand(self, dimx, dimy, list1, list2 = []):
        if list2 == []:
            list2 = list1
        if len(list1) > self.dimx or len(list2) > self.dimy:
            raise ValueError( "list invalid in expand()")

        res = matrix()
```

```
        res.zero(dimx, dimy)
        for i in range(len(list1)):
            for j in range(len(list2)):
                res.value[list1[i]][list2[j]] = self.value[i][j]
        return res

    # ------------
    #
    # Computes the upper triangular Cholesky factorization of
    # a positive definite matrix.
    # This code is based on http://adorio-research.org/wordpress/?p=4560
    #

    def Cholesky(self, ztol= 1.0e-5):

        res = matrix()
        res.zero(self.dimx, self.dimx)

        for i in range(self.dimx):
            S = sum([(res.value[k][i])**2 for k in range(i)])
            d = self.value[i][i] - S
            if abs(d) < ztol:
                res.value[i][i] = 0.0
            else:
                if d < 0.0:
                    raise ValueError( "Matrix not positive-definite")
                res.value[i][i] = sqrt(d)
            for j in range(i+1, self.dimx):
                S = sum([res.value[k][i] * res.value[k][j] for k in
range(i)])
                if abs(S) < ztol:
                    S = 0.0
                res.value[i][j] = (self.value[i][j] - S)/res.value[i][i]
        return res

    # ------------
    #
    # Computes inverse of matrix given its Cholesky upper Triangular
    # decomposition of matrix.
    # This code is based on http://adorio-research.org/wordpress/?p=4560
```

```
    #

    def CholeskyInverse(self):

        res = matrix()
        res.zero(self.dimx, self.dimx)

        # Backward step for inverse.
        for j in reversed(range(self.dimx)):
            tjj = self.value[j][j]
            if tjj ==0:
                print("tjj = 0 at j =",j)
            S = sum([self.value[j][k]*res.value[j][k] for k in range(j+1,
self.dimx)])
            res.value[j][j] = 1.0/ tjj**2 - S/ tjj
            for i in reversed(range(j)):
                res.value[j][i] = res.value[i][j] = \
                    -sum([self.value[i][k]*res.value[k][j] for k in \
                                range(i+1,self.dimx)])/self.value[i][i]
        return res

    # ------------
    #
    # computes and returns the inverse of a square matrix
    #
    def inverse(self):
        aux = self.Cholesky()
        res = aux.CholeskyInverse()
        return res

    # ------------
    #
    # prints matrix (needs work!)
    #
    def __repr__(self):
        return repr(self.value)

# ------------------------------------------------
#
# this is the robot class
```

```
#
# our robot lives in x-y space, and its motion is
# pointed in a random direction. It moves on a straight line
# until is comes close to a wall at which point it turns
# away from the wall and continues to move.
#
# For measurements, it simply senses the x- and y-distance
# to landmarks. This is different from range and bearing as
# commonly studied in the literature, but this makes it much
# easier to implement the essentials of SLAM without
# cluttered math
#

class robot:

    # --------
    # init:
    #   creates robot and initializes location to 0, 0
    #

    def __init__(self, world_size = 100.0, measurement_range = 30.0,
                 motion_noise = 1.0, measurement_noise = 1.0):
        self.measurement_noise = 0.0
        self.world_size = world_size
        self.measurement_range = measurement_range
        self.x = world_size / 2.0
        self.y = world_size / 2.0
        self.motion_noise = motion_noise
        self.measurement_noise = measurement_noise
        self.landmarks = []
        self.num_landmarks = 0


    def rand(self):
        return random.random() * 2.0 - 1.0

    # --------
    #
    # make random landmarks located in the world
    #
```

```
    def make_landmarks(self, num_landmarks):
        self.landmarks = []
        for i in range(num_landmarks):
            self.landmarks.append([round(random.random() *
self.world_size),
                                   round(random.random() *
self.world_size)])
        self.num_landmarks = num_landmarks


    # --------
    #
    # move: attempts to move robot by dx, dy. If outside world
    #       boundary, then the move does nothing and instead returns
failure
    #

    def move(self, dx, dy):

        x = self.x + dx + self.rand() * self.motion_noise
        y = self.y + dy + self.rand() * self.motion_noise

        if x < 0.0 or x > self.world_size or y < 0.0 or y >
self.world_size:
            return False
        else:
            self.x = x
            self.y = y
            return True


    # --------
    #
    # sense: returns x- and y- distances to landmarks within visibility
range
    #        because not all landmarks may be in this range, the list of
measurements
    #        is of variable length. Set measurement_range to -1 if you
want all
```

```
    #         landmarks to be visible at all times
    #

    def sense(self):
        Z = []
        for i in range(self.num_landmarks):
            dx = self.landmarks[i][0] - self.x + self.rand() *
self.measurement_noise
            dy = self.landmarks[i][1] - self.y + self.rand() *
self.measurement_noise
            if self.measurement_range < 0.0 or abs(dx) + abs(dy) <=
self.measurement_range:
                Z.append([i, dx, dy])
        return Z

    # --------
    #
    # print robot location
    #

    def __repr__(self):
        return 'Robot: [x=%.5f y=%.5f]'  % (self.x, self.y)

####################################################

# --------
# this routine makes the robot data
#

def make_data(N, num_landmarks, world_size, measurement_range,
motion_noise,
              measurement_noise, distance):


    complete = False

    while not complete:

        data = []
```

```
        # make robot and landmarks
        r = robot(world_size, measurement_range, motion_noise,
measurement_noise)
        r.make_landmarks(num_landmarks)
        seen = [False for row in range(num_landmarks)]

        # guess an initial motion
        orientation = random.random() * 2.0 * pi
        dx = cos(orientation) * distance
        dy = sin(orientation) * distance

        for k in range(N-1):

            # sense
            Z = r.sense()

            # check off all landmarks that were observed
            for i in range(len(Z)):
                seen[Z[i][0]] = True

            # move
            while not r.move(dx, dy):
                # if we'd be leaving the robot world, pick instead a new
direction
                orientation = random.random() * 2.0 * pi
                dx = cos(orientation) * distance
                dy = sin(orientation) * distance

            # memorize data
            data.append([Z, [dx, dy]])

        # we are done when all landmarks were observed; otherwise re-run
        complete = (sum(seen) == num_landmarks)

    print (' ')
    print ('Landmarks: ', r.landmarks)
    print (r)


    return data
```

```
##################################################

# -------------------------------
#
# print the result of SLAM, the robot pose(s) and the landmarks
#

def print_result(N, num_landmarks, result):
    print
    print ('Estimated Pose(s):')
    for i in range(N):
        print ('    ['+ ', '.join('%.3f'%x for x in result.value[2*i]) +
', ' \
            + ', '.join('%.3f'%x for x in result.value[2*i+1]) +']')
    print
    print ('Estimated Landmarks:')
    for i in range(num_landmarks):
        print ('    ['+ ', '.join('%.3f'%x for x in result.value[2*(N+i)])
+ ', ' \
            + ', '.join('%.3f'%x for x in result.value[2*(N+i)+1]) +']')

# -------------------------------
#
# slam - retains entire path and all landmarks
#

############## ENTER YOUR CODE BELOW HERE ###################

def slam(data, N, num_landmarks, motion_noise, measurement_noise):
    #
    #
    # Add your code here!
    #
    #
    total = N + num_landmarks
    mtc = motion_noise  # motion confidence
    msc = measurement_noise  # measurement confidence
    omegax = [[0.0 for row in range(total)] for col in range(total)]
    xix = [[0.0 for row in range(1)] for col in range(total)]
```

```
    omegay = [[0.0 for row in range(total)] for col in range(total)]
    xiy = [[0.0 for row in range(1)] for col in range(total)]
    omegax[0][0] += 1.
    xix[0][0] += 50.
    omegay[0][0] += 1.
    xiy[0][0] += 50.
    for locI in range(len(data)):
        for measI in range(len(data[locI][0])):
            lmI = data[locI][0][measI][0] + N
            msx = data[locI][0][measI][1] * msc
            msy = data[locI][0][measI][2] * msc
            omegax[locI][locI] += msc
            omegax[lmI][lmI] += msc
            omegax[locI][lmI] += -msc
            omegax[lmI][locI] += -msc
            omegay[locI][locI] += msc
            omegay[lmI][lmI] += msc
            omegay[locI][lmI] += -msc
            omegay[lmI][locI] += -msc
            xix[locI][0] += -msx
            xix[lmI][0] += msx
            xiy[locI][0] += -msy
            xiy[lmI][0] += msy
        dx = data[locI][1][0] * mtc
        dy = data[locI][1][1] * mtc
        omegax[locI][locI] += mtc
        omegax[locI+1][locI+1] += mtc
        omegax[locI+1][locI] += -mtc
        omegax[locI][locI+1] += -mtc
        omegay[locI][locI] += mtc
        omegay[locI+1][locI+1] += mtc
        omegay[locI][locI+1] += -mtc
        omegay[locI+1][locI] += -mtc
        xix[locI][0] += -dx
        xix[locI+1][0] += dx
        xiy[locI][0] += -dy
        xiy[locI+1][0] += dy

    mux = matrix(omegax).inverse() * matrix(xix)
    muy = matrix(omegay).inverse() * matrix(xiy)
```

```
    mu = []
    for i in range(mux.dimx):
        mu.append(mux.value[i])
        mu.append(muy.value[i])
    mu = matrix(mu)
    return mu # Make sure you return mu for grading!

############### ENTER YOUR CODE ABOVE HERE ###################

# ------------------------------------------------------------------------
# ------------------------------------------------------------------------
# ------------------------------------------------------------------------
#
# Main routines
#


num_landmarks      = 5        # number of landmarks
N                  = 20       # time steps
world_size         = 100.0    # size of world
measurement_range  = 50.0     # range at which we can sense landmarks
motion_noise       = 2.0      # noise in robot motion
measurement_noise  = 2.0      # noise in the measurements
distance           = 20.0     # distance by which robot (intends to) move
each iteratation

data = make_data(N, num_landmarks, world_size, measurement_range,
motion_noise, measurement_noise, distance)
result = slam(data, N, num_landmarks, motion_noise, measurement_noise)
print_result(N, num_landmarks, result)

# -------------
# Testing
#
# Uncomment one of the test cases below to compare your results to
# the results shown for Test Case 1 and Test Case 2.

test_data1 = [[[[1, 19.457599255548065, 23.8387362100849], [2,
-13.195807561967236, 11.708840328458608], [3, -30.0954905279171,
15.387879242505843]], [-12.2607279422326, -15.801093326936487]], [[[2,
```

```
-0.4659930049620491, 28.088559771215664], [4, -17.866382374890936,
-16.384904503932]], [-12.2607279422326, -15.8010933269364871]], [[[4,
-6.202512900833806, -1.8234032102746391]], [-12.2607279422326,
-15.8010933269364871]], [[[4, 7.412136480918645, 15.388585962142429]],
[14.008259661173426, 14.274756084260822]], [[[4, -7.526138813444998,
-0.45639424297178491]], [14.008259661173426, 14.274756084260822]], [[[2,
-6.299793150150058, 29.047830407717623], [4, -21.93551130411791,
-13.21956810989039]], [14.008259661173426, 14.274756084260822]], [[[1,
15.796300959032276, 30.65769689694247], [2, -18.64370821983482,
17.380022987031367]], [14.008259661173426, 14.274756084260822]], [[[1,
0.40311325410337906, 14.169429532679855], [2, -35.069349468466235,
2.4945558982439957]], [14.008259661173426, 14.274756084260822]], [[[1,
-16.71340983241936, -2.7770002695438341]], [-11.006096015782283,
16.699276945166858]], [[[1, -3.611096830835776, -17.9540192267639581]],
[-19.693482634035977, 3.4880856845730481]], [[[1, 18.398273354362416,
-22.7051023325509471]], [-19.693482634035977, 3.4880856845730481]], [[[2,
2.789312482883833, -39.737201931213241]], [12.849049222879723,
-15.3265108249729831]], [[[1, 21.26897046581808, -10.1210297990409151], [2,
-11.917698965880655, -23.177116626020971], [3, -31.81167947898398,
-16.79856730233311]], [12.849049222879723, -15.3265108249729831]], [[[1,
10.48157743234859, 5.692957082575485], [2, -22.31488473554935,
-5.389184118551409], [3, -40.81803984305378, -2.47033297902381181]],
[12.849049222879723, -15.3265108249729831]], [[[0, 10.591050242096598,
-39.20517989671131], [1, -3.5675572049297553, 22.849456408289125], [2,
-38.39251065320351, 7.2889903060295111]], [12.849049222879723,
-15.3265108249729831]], [[[0, -3.6225556479370766, -25.580068652355121]],
[-7.8874682868419965, -18.3790055232610921]], [[[0, 1.9784503557879374,
-6.5025974151499]], [-7.8874682868419965, -18.3790055232610921]], [[[0,
10.050665232782423, 11.026385307998742]], [-17.82919359778298,
9.0620006429471421]], [[[0, 26.526838150174818, -0.225633932324256211, [4,
-33.70303936886652, 2.8803398410136771]], [-17.82919359778298,
9.0620006429471421]]]
test_data2 = [[[[0, 26.543274387283322, -6.2625381603126721, [3,
9.9373968257997551, -9.1285403608676891]], [18.92765331253674,
-6.4609550439866831]], [[[0, 7.7065447397229611, -3.7584672154457481, [1,
17.039544119489371, 31.7054899385534381, [3, -11.617312887774971,
-6.649640967164161]], [18.92765331253674, -6.4609550439866831]], [[[0,
-12.351305071363781, 2.5851191042392491, [1, -2.5635345361653131,
38.221596578383691, [3, -26.9612368047409351, -0.48023126261415251]],
[-11.167066095509824, 16.5920654174974551]], [[[0, 1.4138633151721272,
```

```
-13.912454837810632], [1, 8.087721200818589, 20.51845934354381], [3,
-17.091723454402302, -16.521500551709707], [4, -7.414211721400232,
38.091916026744391]], [-11.167066095509824, 16.592065417497455]], [[[0,
12.886743222179561, -28.703968411636318], [1, 21.660953298391387,
3.4912891084614914], [3, -6.401401414569506, -32.321583037341625], [4,
5.034079343639034, 23.102207946092893]], [-11.167066095509824,
16.592065417497455]], [[[1, 31.126317672358578, -10.036784369535214], [2,
-38.70878528420893, 7.4987265861424595], [4, 17.977218575473767,
6.150889254289742]], [-6.595520680493778, -18.881183939392651]], [[[1,
41.82460922922086, 7.847527392202475], [3, 15.711709540417502,
-30.346336599128181]], [-6.595520680493778, -18.881183939392651]], [[[0,
40.18454208294434, -6.710999804403755], [3, 23.019508919299156,
-10.121108672906041]], [-6.595520680493778, -18.881183939392651]], [[[3,
27.18579315312821, 8.067219022708391]], [-6.595520680493778,
-18.881183939392651]], [[], [11.492663265706092, 16.36822198838621]], [[[3,
24.57154567653098, 13.461499960708197]], [11.492663265706092,
16.36822198838621]], [[[0, 31.61945290413707, 0.4272295085799329], [3,
16.97392299158991, -5.274596836133088]], [11.492663265706092,
16.36822198838621]], [[[0, 22.407381798735177, -18.035000683792591], [1,
29.642444125196995, 17.3794951934614], [3, 4.7969752441371645,
-21.075053616399691], [4, 14.726069092569372, 32.759994223000781]],
[11.492663265706092, 16.36822198838621]], [[[0, 10.705527984670137,
-34.589764174299596], [1, 18.58772336795603, -0.201097081647877651], [3,
-4.839806195049413, -39.922087423051051], [4, 4.18824810165454,
14.146847823548889]], [11.492663265706092, 16.36822198838621]], [[[1,
5.878492140223764, -19.955352450942357], [4, -7.059505455306587,
-0.97408492805505851]], [19.628527845173146, 3.836781806574671]], [[[1,
-11.150789592446378, -22.736641053247872], [4, -28.832815721158255,
-3.9462962046291388]], [-19.841703647091965, 2.5113335861604362]], [[[1,
8.64427397916182, -20.286336970889053], [4, -5.036917727942285,
-6.3117399938683361]], [-5.946642674882207, -19.095482211697871]], [[[0,
7.151866679283043, -39.561032326163691], [1, 16.01535401373368,
-3.7809953451940271], [4, -3.048013318321371, 13.697362774960865]],
[-5.946642674882207, -19.095482211697871]], [[[0, 12.872879480504395,
-19.707592098123207], [1, 22.236710716903136, 16.331770792606406], [3,
-4.841206109583004, -21.246044358512421], [4, 4.27111163223552,
32.253097486141841]], [-5.946642674882207, -19.095482211697871]]]

##  Test Case 1
##
```

```
##  Estimated Pose(s):
##      [49.999, 49.999]
##      [37.971, 33.650]
##      [26.183, 18.153]
##      [13.743, 2.114]
##      [28.095, 16.781]
##      [42.383, 30.900]
##      [55.829, 44.494]
##      [70.855, 59.697]
##      [85.695, 75.540]
##      [74.010, 92.431]
##      [53.543, 96.451]
##      [34.523, 100.078]
##      [48.621, 83.951]
##      [60.195, 68.105]
##      [73.776, 52.932]
##      [87.130, 38.536]
##      [80.301, 20.506]
##      [72.797, 2.943]
##      [55.244, 13.253]
##      [37.414, 22.315]
##
##  Estimated Landmarks:
##      [82.954, 13.537]
##      [70.493, 74.139]
##      [36.738, 61.279]
##      [18.696, 66.057]
##      [20.633, 16.873]


##  Test Case 2
##
##  Estimated Pose(s):
##      [49.999, 49.999]
##      [69.180, 45.664]
##      [87.742, 39.702]
##      [76.269, 56.309]
##      [64.316, 72.174]
##      [52.256, 88.151]
##      [44.058, 69.399]
```

```
##       [37.001, 49.916]
##       [30.923, 30.953]
##       [23.507, 11.417]
##       [34.179, 27.131]
##       [44.154, 43.844]
##       [54.805, 60.919]
##       [65.697, 78.544]
##       [77.467, 95.624]
##       [96.801, 98.819]
##       [75.956, 99.969]
##       [70.199, 81.179]
##       [64.053, 61.721]
##       [58.106, 42.626]
##
##  Estimated Landmarks:
##       [76.778, 42.885]
##       [85.064, 77.436]
##       [13.546, 95.649]
##       [59.448, 39.593]
##       [69.262, 94.238]



### Uncomment the following three lines for test case 1 ###

result = slam(test_data1, 20, 5, 2.0, 2.0)
print_result(20, 5, result)
#print result


### Uncomment the following three lines for test case 2 ###

result = slam(test_data2, 20, 5, 2.0, 2.0)
print_result(20, 5, result)
#print result
```

```
Landmarks:  [[22, 35], [13, 95], [20, 48], [86, 10], [92, 51]]
Robot: [x=84.42877 y=28.95365]
Estimated Pose(s):
    [50.000, 50.000]
    [60.357, 65.609]
    [71.438, 80.889]
    [81.297, 96.730]
    [80.277, 75.612]
    [79.706, 54.359]
    [79.017, 33.133]
    [79.104, 13.827]
    [64.413, 28.486]
```

```
    [85.993, 29.790]
Estimated Landmarks:
    [21.942, 34.548]
    [13.220, 94.865]
    [19.721, 47.706]
    [84.847, 9.845]
    [91.217, 50.794]
Estimated Pose(s):
    [49.999, 49.999]
    [37.972, 33.651]
    [26.184, 18.154]
    [13.744, 2.115]
    [28.096, 16.782]
    [42.383, 30.900]
    [55.830, 44.495]
```

1)

간단한 그래프 **SLAM** 노드 및 엣지 생성

문제**:**

로봇의 위치를 나타내는 노드를 생성하고, 두 노드 사이의 이동을 나타내는 엣지를 추가하는 간단한 그래프를 파이썬으로 구현하세요. 로봇은 처음 위치 (0, 0)에서 시작하여 (1, 0)으로 이동합니다.

```python
import numpy as np

# 노드 생성
node_0 = np.array([0.0, 0.0])  # 초기 위치
node_1 = np.array([1.0, 0.0])  # 이동 후 위치

# 엣지 생성 (이동 정보)
edge_0_1 = node_1 - node_0  # 이동 벡터

print("Node 0 위치:", node_0)
print("Node 1 위치:", node_1)
print("Edge 0->1 이동:", edge_0_1)
```

```
Node 0 위치: [0. 0.]
Node 1 위치: [1. 0.]
Edge 0->1 이동: [1. 0.]
```

- 노드는 로봇의 위치를 나타내는 좌표로 표현됩니다.
- 엣지는 두 노드 사이의 상대적인 이동을 나타내며, 이동 벡터로 계산됩니다.

2)

노드와 엣지를 활용한 그래프 생성

문제:

세 개의 노드로 구성된 그래프를 생성하세요. 로봇은 위치 (0, 0)에서 시작하여 (1, 0)으로 이동한 후, (1, 1)로 이동합니다. 각 이동은 엣지로 표현됩니다.

```python
import numpy as np

# 노드 생성
nodes = {
    0: np.array([0.0, 0.0]),
    1: np.array([1.0, 0.0]),
    2: np.array([1.0, 1.0])
}

# 엣지 생성
```

```
edges = {
    (0, 1): nodes[1] - nodes[0],
    (1, 2): nodes[2] - nodes[1]
}

print("노드 정보:")
for idx, pos in nodes.items():
    print(f"노드 {idx}: 위치 {pos}")

print("\n엣지 정보:")
for (i, j), vec in edges.items():
    print(f"엣지 {i}->{j}: 이동 {vec}")
```

```
노드 정보:
노드 0: 위치 [0. 0.]
노드 1: 위치 [1. 0.]
노드 2: 위치 [1. 1.]

엣지 정보:
엣지 0->1: 이동 [1. 0.]
엣지 1->2: 이동 [0. 1.]
```

- 노드는 딕셔너리를 사용하여 관리되며, 키는 노드의 인덱스입니다.
- 엣지도 딕셔너리로 관리되며, 키는 노드 쌍의 튜플입니다.
- 각 엣지는 시작 노드와 종료 노드 사이의 이동 벡터로 나타냅니다.

3)

그래프 **SLAM**에서 오도메트리 노이즈 추가

문제**:**

위의 문제 **2**에서 사용된 이동에 오도메트리 노이즈를 추가하세요. 노이즈는 평균이 **0**이고 표준편차가 **0.1**인 정규분포를 따릅니다. 노이즈가 추가된 엣지를 출력하세요.

```
import numpy as np



# 노드 생성 (동일)
```

```python
nodes = {
    0: np.array([0.0, 0.0]),
    1: np.array([1.0, 0.0]),
    2: np.array([1.0, 1.0])
}

# 엣지 생성 및 노이즈 추가
edges = {}
np.random.seed(42)  # 재현성을 위한 시드 설정
for (i, j) in [(0, 1), (1, 2)]:
    true_motion = nodes[j] - nodes[i]
    noise = np.random.normal(0, 0.1, size=2)
    noisy_motion = true_motion + noise
    edges[(i, j)] = noisy_motion

print("노이즈가 추가된 엣지 정보:")
for (i, j), vec in edges.items():
    print(f"엣지 {i}->{j}: 이동 {vec}")
```

```
노이즈가 추가된 엣지 정보:
엣지 0->1: 이동 [ 1.04967142 -0.01382643]
엣지 1->2: 이동 [0.06476885 1.15230299]
```

- `np.random.normal` 함수를 사용하여 노이즈를 생성합니다.
- 노이즈는 각 이동 벡터에 추가되어 실제 이동과 측정된 이동 간의 차이를 모델링합니다.

4)

간단한 그래프 **SLAM** 최적화

문제**:**

노이즈가 추가된 이동 정보를 사용하여 그래프 SLAM 최적화를 수행하세요. 최적화를 위해 각 노드의 위치를 조정하여 엣지의 오차를 최소화하세요. 여기서는 간단하게 평균 위치를 계산하여 노드 위치를 업데이트하세요.

```
import numpy as np

# 초기 노드 위치 (노이즈 없음)
nodes = {
    0: np.array([0.0, 0.0]),
    1: np.array([1.0, 0.0]),
    2: np.array([1.0, 1.0])
}

# 노이즈가 추가된 엣지 (문제 3에서 가져옴)
edges = {
    (0, 1): np.array([1.0, 0.0]) + np.random.normal(0, 0.1, size=2),
    (1, 2): np.array([0.0, 1.0]) + np.random.normal(0, 0.1, size=2)
}

# 최적화 수행 (여기서는 간단히 노드 위치를 조정)
# 실제로는 비선형 최적화 알고리즘을 사용하지만, 여기서는 평균으로 근사
nodes_optimized = nodes.copy()
for i in nodes:
    connected_edges = [e for e in edges if i in e]
    if connected_edges:
        positions = []
        for (a, b) in connected_edges:
            if a == i:
                neighbor = nodes_optimized[b] - edges[(a, b)]
            else:
                neighbor = nodes_optimized[a] + edges[(a, b)]
            positions.append(neighbor)
        nodes_optimized[i] = np.mean(positions, axis=0)
```

```
print("최적화된 노드 위치:")
for idx, pos in nodes_optimized.items():
    print(f"노드 {idx}: 위치 {pos}")
```

```
최적화된 노드 위치:
노드 0: 위치 [-0.01368716 -0.16020866]
노드 1: 위치 [0.95865755 0.01595685]
노드 2: 위치 [1.04134245 0.98404315]
```

- 실제 그래프 SLAM 최적화는 비선형 최적화 알고리즘을 사용하지만, 여기서는 간단히 평균 위치로 노드를 업데이트
- 각 노드는 연결된 엣지와 이웃 노드의 정보를 사용하여 위치를 조정합니다.

5)

그래프 **SLAM**에서의 루프 폐쇄

문제**:**

로봇이 (0, 0)에서 시작하여 (1, 0), (1, 1), (0, 1) 순으로 이동한 후, 다시 시작 지점 (0, 0)으로 돌아온다고 가정합니다. 이 경로를 따라 노드와 엣지를 생성하고, 노이즈를 추가하세요. 그런 다음 그래프 SLAM 최적화를 수행하여 로봇의 경로를 복원하세요.

```
import numpy as np

# 노드 생성
nodes = {
    0: np.array([0.0, 0.0]),
    1: np.array([1.0, 0.0]),
    2: np.array([1.0, 1.0]),
    3: np.array([0.0, 1.0]),
    4: np.array([0.0, 0.0])  # 시작 지점으로 복귀
}

# 엣지 생성 및 노이즈 추가
edges = {}
np.random.seed(42)
for (i, j) in [(0, 1), (1, 2), (2, 3), (3, 4)]:
```

```
    true_motion = nodes[j] - nodes[i]
    noise = np.random.normal(0, 0.1, size=2)
    noisy_motion = true_motion + noise
    edges[(i, j)] = noisy_motion

# 루프 폐쇄 엣지 추가 (0번 노드와 4번 노드 동일)
edges[(4, 0)] = np.zeros(2)  # 실제로는 노이즈가 있는 측정값

# 그래프 최적화 수행 (여기서는 간단한 방법 사용)
nodes_optimized = nodes.copy()
for iteration in range(10):  # 반복 수행
    for i in nodes:
        connected_edges = [e for e in edges if i in e]
        if connected_edges:
            positions = []
            for (a, b) in connected_edges:
                if a == i:
                    neighbor = nodes_optimized[b] - edges[(a, b)]
                else:
                    neighbor = nodes_optimized[a] + edges[(a, b)]
                positions.append(neighbor)
            nodes_optimized[i] = np.mean(positions, axis=0)

print("최적화된 노드 위치:")
for idx, pos in nodes_optimized.items():
    print(f"노드 {idx}: 위치 {pos}")
```

```
최적화된 노드 위치:
노드 0: 위치 [-0.0111959  -0.01725315]
노드 1: 위치 [ 0.98869017 -0.06944216]
노드 2: 위치 [1.00366675 1.0444999 ]
노드 3: 위치 [-0.06954115  0.98272597]
노드 4: 위치 [0.03859211 0.02110815]
```

- 루프 폐쇄는 시작 지점으로 돌아오는 경로를 의미하며, 이는 그래프에서 추가적인 제약 조건을 제공합니다.
- 반복적인 평균 계산을 통해 노드 위치를 점진적으로 업데이트하여 최적화를 수행했습니다.
- 실제로는 가우스-뉴턴 같은 알고리즘 최적화 방법을 사용.

6)

시각화를 통한 로봇 경로 및 노드 위치 표시

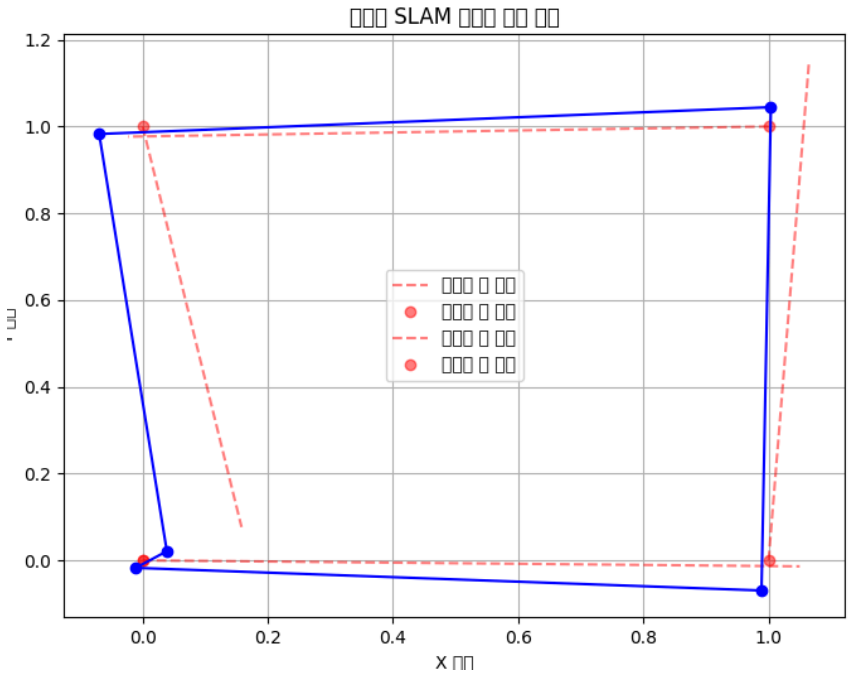
문제:

문제 5에서 생성한 그래프의 노드와 엣지를 matplotlib를 사용하여 시각화하세요. 최적화 전과 후의 노드 위치를 서로 다른 색으로 표시하여 비교하세요.

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

# (문제 5의 코드 사용)
# ... (위의 코드 생략)

# 최적화 전 노드 위치
nodes_before = nodes.copy()

# 그래프 최적화 수행 (위의 코드)

# 시각화
plt.figure(figsize=(8, 6))

# 노드 및 엣지 표시 (최적화 전)
for (i, j), motion in edges.items():
    start = nodes_before[i]
    end = nodes_before[i] + motion
    plt.plot([start[0], end[0]], [start[1], end[1]], 'r--', alpha=0.5)
    plt.plot(start[0], start[1], 'ro', alpha=0.5)

# 노드 및 엣지 표시 (최적화 후)
for (i, j), motion in edges.items():
    start = nodes_optimized[i]
    end = nodes_optimized[j]
    plt.plot([start[0], end[0]], [start[1], end[1]], 'b-')
    plt.plot(start[0], start[1], 'bo')

plt.title('그래프 SLAM 최적화 전후 비교')
plt.xlabel('X 위치')
```

```
plt.ylabel('Y 위치')
plt.legend(['최적화 전 엣지', '최적화 전 노드', '최적화 후 엣지', '최적화 후 노드'])
plt.grid(True)
plt.show()
```

- 루프 폐쇄는 시작 지점으로 돌아오는 경로를 의미하며, 이는 그래프에서 추가적인 제약 조건을 제공합니다.
- 반복적인 평균 계산을 통해 노드 위치를 점진적으로 업데이트하여 최적화를 수행했습니다.
- 실제로는 가우스-뉴턴 알고리즘 같은 최적화 방법을 사용합니다.



7)

시각화를 통한 **2D** 그래프 **SLAM** 시뮬레이션

문제**:**

로봇이 2D 공간에서 랜덤하게 이동하며 랜드마크를 관측하는 시뮬레이션을 생성하세요. 랜드마크는 5개이며, 로봇은 20개의 위치로 이동합니다. 노이즈가 있는 오도메트리와 랜드마크 관측을 사용하여 그래프 SLAM을 수행하고, 최적화 결과를 시각화하세요.

```
import numpy as np
```

```
import matplotlib.pyplot as plt



# 시뮬레이션 파라미터

num_landmarks = 5

num_steps = 20

np.random.seed(42)



# 랜드마크 생성

landmarks = {'L{}'.format(i): np.random.uniform(-10, 10, size=2) for i in
range(num_landmarks)}



# 로봇의 실제 경로 생성

true_positions = [np.array([0.0, 0.0])]

for i in range(1, num_steps):

    motion = np.random.uniform(-1, 1, size=2)

    true_positions.append(true_positions[-1] + motion)



# 오도메트리 측정 (노이즈 추가)

motions = []

for i in range(1, num_steps):

    motion = true_positions[i] - true_positions[i-1] + np.random.normal(0,
0.1, size=2)

    motions.append(motion)
```

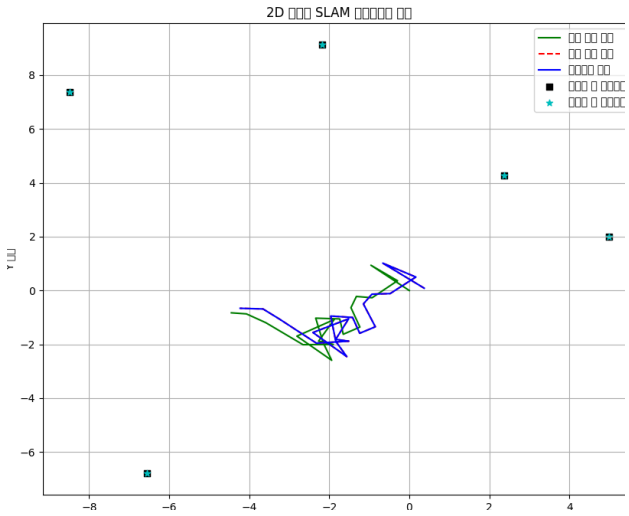
```
# 랜드마크 관측 (노이즈 추가)

measurements = {}

for i in range(num_steps):

    for l_key, l_pos in landmarks.items():

        if np.random.rand() < 0.3:  # 30% 확률로 랜드마크 관측

            measurement = l_pos - true_positions[i] + np.random.normal(0,
0.1, size=2)

            measurements[(i, l_key)] = measurement



# 그래프 생성

nodes = {i: true_positions[i] + np.random.normal(0, 0.5, size=2) for i in
range(num_steps)}  # 초기 노드 위치 (노이즈 있음)

edges = {(i, i+1): motions[i] for i in range(num_steps -1)}  # 오도메트리
엣지



# 그래프 SLAM 최적화 (랜드마크 포함)

def optimize_full_graph(nodes, edges, landmarks, measurements,
iterations=5):

    all_nodes = list(nodes.keys()) + list(landmarks.keys())

    num_nodes = len(all_nodes)

    for iter in range(iterations):

        H = np.zeros((2*num_nodes, 2*num_nodes))

        b = np.zeros(2*num_nodes)

        # 오도메트리 엣지로부터 업데이트

        for (i, j), z in edges.items():
```

```
            xi = nodes[i]
            xj = nodes[j]
            e = (xj - xi) - z
            Ai = -np.eye(2)
            Aj = np.eye(2)
            idx_i = all_nodes.index(i)
            idx_j = all_nodes.index(j)
            H[2*idx_i:2*idx_i+2, 2*idx_i:2*idx_i+2] += Ai.T @ Ai
            H[2*idx_i:2*idx_i+2, 2*idx_j:2*idx_j+2] += Ai.T @ Aj
            H[2*idx_j:2*idx_j+2, 2*idx_i:2*idx_i+2] += Aj.T @ Ai
            H[2*idx_j:2*idx_j+2, 2*idx_j:2*idx_j+2] += Aj.T @ Aj
            b[2*idx_i:2*idx_i+2] += Ai.T @ e
            b[2*idx_j:2*idx_j+2] += Aj.T @ e
        # 랜드마크 관측으로부터 업데이트
        for (i, l), z in measurements.items():
            xi = nodes[i]
            xl = landmarks[l]
            e = (xl - xi) - z
            Ai = -np.eye(2)
            Al = np.eye(2)
            idx_i = all_nodes.index(i)
            idx_l = all_nodes.index(l)
            H[2*idx_i:2*idx_i+2, 2*idx_i:2*idx_i+2] += Ai.T @ Ai
```

```
            H[2*idx_i:2*idx_i+2, 2*idx_l:2*idx_l+2] += Ai.T @ Al
            H[2*idx_l:2*idx_l+2, 2*idx_i:2*idx_i+2] += Al.T @ Ai
            H[2*idx_l:2*idx_l+2, 2*idx_l:2*idx_l+2] += Al.T @ Al
            b[2*idx_i:2*idx_i+2] += Ai.T @ e
            b[2*idx_l:2*idx_l+2] += Al.T @ e
        dx = -np.linalg.solve(H, b)
        for idx, node_idx in enumerate(all_nodes):
            if node_idx in nodes:
                nodes[node_idx] += dx[2*idx:2*idx+2]
            else:
                landmarks[node_idx] += dx[2*idx:2*idx+2]
    return nodes, landmarks


nodes_optimized, landmarks_optimized = optimize_full_graph(
    nodes.copy(), edges, landmarks.copy(), measurements, iterations=5)


# 시각화
plt.figure(figsize=(10, 8))


# 실제 경로
true_positions_array = np.array(true_positions)
plt.plot(true_positions_array[:,0], true_positions_array[:,1], 'g-',
label='실제 로봇 경로')
```

```
# 초기 추정 경로

nodes_before_array = np.array([nodes[i] for i in range(num_steps)])

plt.plot(nodes_before_array[:,0], nodes_before_array[:,1], 'r--',
label='초기 추정 경로')



# 최적화된 경로

nodes_after_array = np.array([nodes_optimized[i] for i in
range(num_steps)])

plt.plot(nodes_after_array[:,0], nodes_after_array[:,1], 'b-',
label='최적화된 경로')



# 랜드마크 위치

landmarks_before_array = np.array([landmarks[key] for key in
landmarks.keys()])

plt.scatter(landmarks_before_array[:,0], landmarks_before_array[:,1],
c='k', marker='s', label='최적화 전 랜드마크')



landmarks_after_array = np.array([landmarks_optimized[key] for key in
landmarks_optimized.keys()])

plt.scatter(landmarks_after_array[:,0], landmarks_after_array[:,1], c='c',
marker='*', label='최적화 후 랜드마크')



plt.title('2D 그래프 SLAM 시뮬레이션 결과')

plt.xlabel('X 위치')

plt.ylabel('Y 위치')

plt.legend()
```

```
plt.grid(True)

plt.show()
```



- 랜덤한 이동과 랜드마크 관측을 통해 시뮬레이션 데이터를 생성했습니다.
- 노이즈가 추가된 오도메트리와 랜드마크 관측을 사용하여 그래프를 생성하고, 그래프 SLAM 최적화를 수행했습니다.
- 시각화를 통해 실제 경로, 초기 추정 경로, 최적화된 경로를 비교하고, 랜드마크의 위치 추정이 어떻게 개선되었는지 확인했습니다.

8)

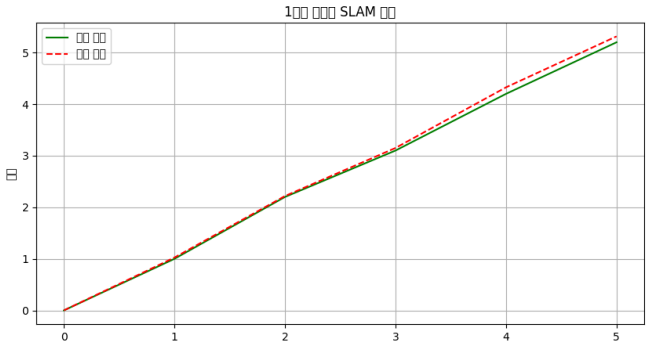
1차원 그래프 **SLAM** 기초 구현

문제**:**

로봇이 1차원 선형 경로를 따라 이동한다고 가정합니다. 로봇은 총 5번 이동하며, 각 이동 거리는 다음과 같습니다: `[1.0, 1.2, 0.9, 1.1, 1.0]`. 하지만 오도메트리 측정에는 노이즈가 포함되어 있습니다. 각 측정에 평균 0, 표준편차 0.05의 가우시안 노이즈가 추가됩니다. 그래프 SLAM을 사용하여 로봇의 위치를 추정하세요.

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

# 실제 이동 거리
true_movements = np.array([1.0, 1.2, 0.9, 1.1, 1.0])

# 오도메트리 측정 (노이즈 추가)
np.random.seed(42)
odometry_measurements = true_movements + np.random.normal(0, 0.05,
size=true_movements.shape)

# 노드 수
num_nodes = len(odometry_measurements) + 1

# 정보 행렬과 정보 벡터 초기화
information_matrix = np.zeros((num_nodes, num_nodes))
information_vector = np.zeros(num_nodes)

# 이동에 대한 제약 조건 추가
for i in range(num_nodes - 1):
    information_matrix[i, i] += 1 / 0.05**2
    information_matrix[i, i+1] -= 1 / 0.05**2
    information_matrix[i+1, i] -= 1 / 0.05**2
    information_matrix[i+1, i+1] += 1 / 0.05**2

    information_vector[i] -= odometry_measurements[i] / 0.05**2
    information_vector[i+1] += odometry_measurements[i] / 0.05**2

# 초기 위치 고정 (0으로)
information_matrix[0, 0] += 1e6  # 매우 큰 값으로 고정
information_vector[0] += 0 * 1e6  # 시작 위치를 0으로 고정

# 선형 시스템 풀기
estimated_positions = np.linalg.solve(information_matrix,
information_vector)

# 실제 위치 계산
true_positions = np.zeros(num_nodes)
for i in range(1, num_nodes):
    true_positions[i] = true_positions[i-1] + true_movements[i-1]
```

```python
# 결과 출력
print("추정된 로봇 위치:")
for i, pos in enumerate(estimated_positions):
    print(f"노드 {i}: 위치 {pos:.4f}")

# 시각화
plt.figure(figsize=(10, 5))
plt.plot(range(num_nodes), true_positions, 'g-', label='실제 위치')
plt.plot(range(num_nodes), estimated_positions, 'r--', label='추정 위치')
plt.xlabel('노드 인덱스')
plt.ylabel('위치')
plt.title('1차원 그래프 SLAM 결과')
plt.legend()
plt.grid(True)
plt.show()
```



- 오도메트리 측정 생성: 실제 이동 거리에 노이즈를 추가하여 오도메트리 측정값을 생성합니다.
- 정보 행렬과 정보 벡터 구성: 각 이동에 대한 제약 조건을 정보 행렬과 벡터에 추가합니다.
- 초기 위치 고정: 시작 위치를 0으로 고정하기 위해 매우 큰 값을 사용하여 정보 행렬과 벡터에 제약을 추가합니다.
- 선형 시스템 해결: `np.linalg.solve`를 사용하여 위치를 추정합니다.
- 결과 시각화: 실제 위치와 추정된 위치를 그래프로 비교하여 표시합니다.

9)

2D 환경에서의 간단한 그래프 **SLAM**

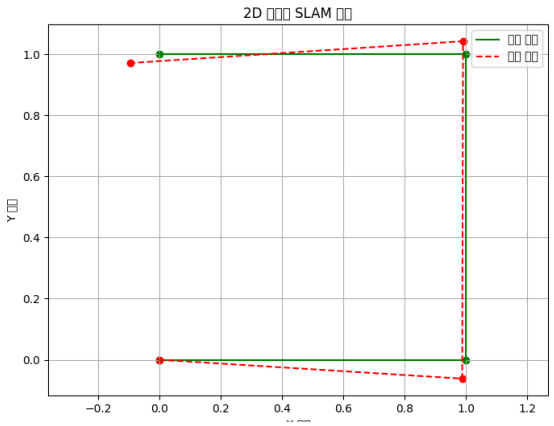
문제**:**

로봇이 2D 평면에서 네 개의 위치를 순서대로 방문합니다: (0,0), (1,0), (1,1), (0,1). 이동 중에 각 위치에서 다음 위치로의 이동에 대한 오도메트리 측정을 수행하지만, 각 측정에는 평균이 0이고 표준편차가 0.1인 가우시안 노이즈가 포함됩니다. 그래프 SLAM을 사용하여 로봇의 위치를 추정하세요.

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

# 실제 위치
true_positions = np.array([
    [0.0, 0.0],
    [1.0, 0.0],
    [1.0, 1.0],
    [0.0, 1.0]
])

# 오도메트리 측정 생성 (노이즈 추가)
np.random.seed(42)
odometry_measurements = []
for i in range(len(true_positions) - 1):
    motion = true_positions[i+1] - true_positions[i]
    noisy_motion = motion + np.random.normal(0, 0.1, size=2)
    odometry_measurements.append(noisy_motion)
odometry_measurements = np.array(odometry_measurements)

# 노드 수
num_nodes = len(true_positions)

# 정보 행렬과 정보 벡터 초기화
information_matrix = np.zeros((2*num_nodes, 2*num_nodes))
information_vector = np.zeros(2*num_nodes)

# 이동에 대한 제약 조건 추가
for i in range(num_nodes - 1):
    idx_i = 2 * i
    idx_j = 2 * (i + 1)
```

```
    # 정보 행렬 업데이트
    information_matrix[idx_i:idx_i+2, idx_i:idx_i+2] += np.eye(2) / 0.1**2
    information_matrix[idx_i:idx_i+2, idx_j:idx_j+2] -= np.eye(2) / 0.1**2
    information_matrix[idx_j:idx_j+2, idx_i:idx_i+2] -= np.eye(2) / 0.1**2
    information_matrix[idx_j:idx_j+2, idx_j:idx_j+2] += np.eye(2) / 0.1**2

    # 정보 벡터 업데이트
    information_vector[idx_i:idx_i+2] -= odometry_measurements[i] / 0.1**2
    information_vector[idx_j:idx_j+2] += odometry_measurements[i] / 0.1**2

# 루프 폐쇄 제약 조건 추가 (마지막 노드와 첫 번째 노드 연결)
idx_i = 2 * (num_nodes - 1)
idx_j = 0

information_matrix[idx_i:idx_i+2, idx_i:idx_i+2] += np.eye(2) / 0.1**2
information_matrix[idx_i:idx_i+2, idx_j:idx_j+2] -= np.eye(2) / 0.1**2
information_matrix[idx_j:idx_j+2, idx_i:idx_i+2] -= np.eye(2) / 0.1**2
information_matrix[idx_j:idx_j+2, idx_j:idx_j+2] += np.eye(2) / 0.1**2

# 루프 폐쇄 측정값 (노이즈 포함)
loop_closure_measurement = true_positions[0] - true_positions[-1] +
np.random.normal(0, 0.1, size=2)

information_vector[idx_i:idx_i+2] -= loop_closure_measurement / 0.1**2
information_vector[idx_j:idx_j+2] += loop_closure_measurement / 0.1**2

# 초기 위치 고정
information_matrix[0:2, 0:2] += np.eye(2) * 1e6  # 매우 큰 값으로 고정
information_vector[0:2] += np.array([0.0, 0.0]) * 1e6  # 시작 위치를
(0,0)으로 고정

# 선형 시스템 풀기
estimated_positions_vector = np.linalg.solve(information_matrix,
information_vector)
estimated_positions = estimated_positions_vector.reshape(-1, 2)

# 결과 출력
print("추정된 로봇 위치:")
for i, pos in enumerate(estimated_positions):
    print(f"노드 {i}: 위치 ({pos[0]:.4f}, {pos[1]:.4f})")
```

```
# 결과 시각화
plt.figure(figsize=(8, 6))
plt.plot(true_positions[:, 0], true_positions[:, 1], 'g-', label='실제
경로')
plt.plot(estimated_positions[:, 0], estimated_positions[:, 1], 'r--',
label='추정 경로')
plt.scatter(true_positions[:, 0], true_positions[:, 1], c='g')
plt.scatter(estimated_positions[:, 0], estimated_positions[:, 1], c='r')
plt.xlabel('X 위치')
plt.ylabel('Y 위치')
plt.title('2D 그래프 SLAM 결과')
plt.legend()
plt.grid(True)
plt.axis('equal')
plt.show()
```



- 오도메트리 측정 생성: 각 이동에 대해 노이즈가 포함된 오도메트리 측정값을 생성합니다.
- 정보 행렬과 정보 벡터 구성: 이동 제약 조건과 루프 폐쇄 제약 조건을 포함하여 정보를 구성합니다.
- 루프 폐쇄 제약 조건 추가: 마지막 위치에서 시작 위치로의 이동을 제약으로 추가합니다.
- 선형 시스템 해결: `np.linalg.solve`를 사용하여 위치를 추정합니다.
- 결과 시각화: 실제 경로와 추정된 경로를 그래프로 비교하여 표시합니다.

10)

랜드마크를 이용한 그래프 **SLAM**

문제**:**

로봇이 2D 환경에서 이동하며, 두 개의 랜드마크를 관측합니다. 랜드마크 위치는 L1: (2,2), L2: (4,0)입니다. 로봇은 위치 (0,0)에서 시작하여 (2,0)으로 이동한 후, (2,2)로 이동합니다. 각 이동에는 노이즈가 포함되어 있으며, 랜드마크 관측에도 노이즈가 있습니다. 그래프 SLAM을 사용하여 로봇의 위치와 랜드마크의 위치를 동시에 추정하세요.

```
import numpy as np

import matplotlib.pyplot as plt



# 실제 로봇 위치

true_robot_positions = np.array([

    [0.0, 0.0],

    [2.0, 0.0],

    [2.0, 2.0]

])



# 실제 랜드마크 위치

true_landmarks = {

    'L1': np.array([2.0, 2.0]),

    'L2': np.array([4.0, 0.0])

}



# 오도메트리 측정 생성 (노이즈 추가)
```

```
np.random.seed(42)

odometry_measurements = []

for i in range(len(true_robot_positions) - 1):

    motion = true_robot_positions[i+1] - true_robot_positions[i]

    noisy_motion = motion + np.random.normal(0, 0.1, size=2)

    odometry_measurements.append(noisy_motion)

odometry_measurements = np.array(odometry_measurements)



# 랜드마크 관측 생성 (노이즈 추가)

observations = {

    (0, 'L2'): true_landmarks['L2'] - true_robot_positions[0] +
np.random.normal(0, 0.1, size=2),

    (1, 'L2'): true_landmarks['L2'] - true_robot_positions[1] +
np.random.normal(0, 0.1, size=2),

    (2, 'L1'): true_landmarks['L1'] - true_robot_positions[2] +
np.random.normal(0, 0.1, size=2)

}



# 노드 수 계산

num_robot_nodes = len(true_robot_positions)

num_landmark_nodes = len(true_landmarks)

num_total_nodes = num_robot_nodes + num_landmark_nodes



# 노드 인덱스 매핑
```

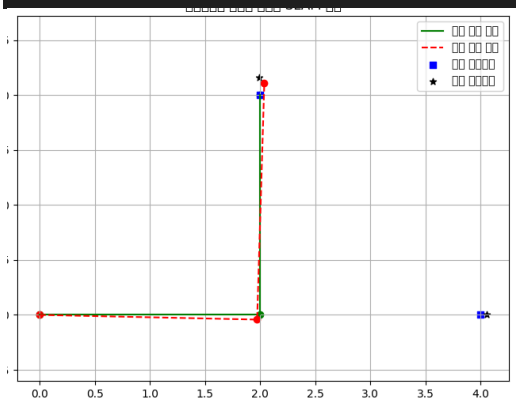
```
node_indices = {}
for i in range(num_robot_nodes):
    node_indices[i] = i  # 로봇 위치 노드
for idx, landmark in enumerate(true_landmarks.keys()):
    node_indices[landmark] = num_robot_nodes + idx  # 랜드마크 노드

# 정보 행렬과 정보 벡터 초기화
information_matrix = np.zeros((2*num_total_nodes, 2*num_total_nodes))
information_vector = np.zeros(2*num_total_nodes)

# 오도메트리 제약 조건 추가
for i in range(num_robot_nodes - 1):
    idx_i = 2 * node_indices[i]
    idx_j = 2 * node_indices[i+1]

    # 정보 행렬 업데이트
    information_matrix[idx_i:idx_i+2, idx_i:idx_i+2] += np.eye(2) / 0.1**2
    information_matrix[idx_i:idx_i+2, idx_j:idx_j+2] -= np.eye(2) / 0.1**2
    information_matrix[idx_j:idx_j+2, idx_i:idx_i+2] -= np.eye(2) / 0.1**2
    information_matrix[idx_j:idx_j+2, idx_j:idx_j+2] += np.eye(2) / 0.1**2

    # 정보 벡터 업데이트
    information_vector[idx_i:idx_i+2] -= odometry_measurements[i] / 0.1**2
```

```
    information_vector[idx_j:idx_j+2] += odometry_measurements[i] / 0.1**2


# 랜드마크 관측 제약 조건 추가
for (i, landmark), measurement in observations.items():
    idx_i = 2 * node_indices[i]
    idx_l = 2 * node_indices[landmark]

    # 정보 행렬 업데이트
    information_matrix[idx_i:idx_i+2, idx_i:idx_i+2] += np.eye(2) / 0.1**2
    information_matrix[idx_i:idx_i+2, idx_l:idx_l+2] -= np.eye(2) / 0.1**2
    information_matrix[idx_l:idx_l+2, idx_i:idx_i+2] -= np.eye(2) / 0.1**2
    information_matrix[idx_l:idx_l+2, idx_l:idx_l+2] += np.eye(2) / 0.1**2

    # 정보 벡터 업데이트
    information_vector[idx_i:idx_i+2] -= measurement / 0.1**2
    information_vector[idx_l:idx_l+2] += measurement / 0.1**2


# 초기 로봇 위치 고정
information_matrix[0:2, 0:2] += np.eye(2) * 1e6  # 매우 큰 값으로 고정
information_vector[0:2] += np.array([0.0, 0.0]) * 1e6  # 시작 위치를
(0,0)으로 고정


# 선형 시스템 풀기
```

```
estimated_positions_vector = np.linalg.solve(information_matrix,
information_vector)

estimated_positions = estimated_positions_vector.reshape(-1, 2)



# 결과 출력

print("추정된 로봇 위치:")

for i in range(num_robot_nodes):

    idx = node_indices[i]

    pos = estimated_positions[idx]

    print(f"노드 {i}: 위치 ({pos[0]:.4f}, {pos[1]:.4f})")



print("\n추정된 랜드마크 위치:")

for landmark in true_landmarks.keys():

    idx = node_indices[landmark]

    pos = estimated_positions[idx]

    print(f"랜드마크 {landmark}: 위치 ({pos[0]:.4f}, {pos[1]:.4f})")



# 결과 시각화

plt.figure(figsize=(8, 6))



# 실제 로봇 위치

plt.plot(true_robot_positions[:, 0], true_robot_positions[:, 1], 'g-',
label='실제 로봇 경로')

plt.scatter(true_robot_positions[:, 0], true_robot_positions[:, 1], c='g')
```

```
# 추정된 로봇 위치
estimated_robot_positions = estimated_positions[:num_robot_nodes]
plt.plot(estimated_robot_positions[:, 0], estimated_robot_positions[:, 1], 'r--', label='추정 로봇 경로')
plt.scatter(estimated_robot_positions[:, 0], estimated_robot_positions[:, 1], c='r')

# 실제 랜드마크 위치
true_landmark_positions = np.array(list(true_landmarks.values()))
plt.scatter(true_landmark_positions[:, 0], true_landmark_positions[:, 1], c='b', marker='s', label='실제 랜드마크')

# 추정된 랜드마크 위치
estimated_landmark_positions = estimated_positions[num_robot_nodes:]
plt.scatter(estimated_landmark_positions[:, 0], estimated_landmark_positions[:, 1], c='k', marker='*', label='추정 랜드마크')

plt.xlabel('X 위치')
plt.ylabel('Y 위치')
plt.title('랜드마크를 이용한 그래프 SLAM 결과')
plt.legend()
plt.grid(True)
plt.axis('equal')
```

```
plt.show()
```



오도메트리 측정 생성: 로봇의 이동에 노이즈를 추가하여 측정값을 생성합니다.
랜드마크 관측 생성: 랜드마크 관측에도 노이즈를 추가하여 측정값을 생성합니다.
노드 인덱스 매핑: 로봇 위치 노드와 랜드마크 노드의 인덱스를 매핑하여 관리합니다.
정보 행렬과 정보 벡터 구성: 오도메트리 제약 조건과 랜드마크 관측 제약 조건을 모두 포함하여 정보를 구성합니다.
선형 시스템 해결: `np.linalg.solve`를 사용하여 로봇의 위치와 랜드마크의 위치를 동시에 추정합니다.
결과 시각화: 실제 로봇 경로, 추정된 로봇 경로, 실제 랜드마크, 추정된 랜드마크를 그래프로 비교하여 표시합니다.