Project 3 on Mathematics in Al

Subject: Square Root

Name: Hesam Mousavi

Student number: 9931155

```
import numpy as np
from my_io import generate_dataset
import warnings

warnings.filterwarnings('ignore')

matrix_size = 5
```

First Idea

Using eigenvalue and eigenvector(eigendecomposition)

Let's find the eigenvalue and eigenvector of \sqrt{A}

Just as with the real numbers, a real matrix may fail to have a real square root, but have a square root with **complex-valued** entries. Some matrices have **no square root**.

Positive semidefinite(P.S.D) matrices

When matrix M is P.S.D? when M is **symmetric** matrix and $\forall x \to x^\top M x \geqslant 0$ then call M as P.S.D matrix

A square real matrix is positive semidefinite if and only if $A=B^{\top}B$ for some matrix B. There can be many different such matrices B. A positive semidefinite matrix A can also have many matrices B such that A=BB. However, A always has precisely one square root B(**principal**, **non-negative**, or **positive square root**) that is positive semidefinite (and hence symmetric).

The principal square root of a real positive semidefinite matrix is real. The principal square root of a positive definite matrix is positive definite; more generally, the **rank** of the principal square root of A is the same as the rank of A.

An n×n matrix with n distinct nonzero eigenvalues has 2^n square roots. Such a matrix, A, has an **eigendecomposition** $VD^{1/2}V^{-1}$ where V is the matrix whose columns are eigenvectors of A and D is the diagonal matrix whose diagonal elements are the corresponding n eigenvalues λ_i . Thus the **square roots** of A are given by $VD^{1/2}V^{-1}$, where $D^{1/2}$ is any square root matrix of D, which, for distinct eigenvalues, must be diagonal with diagonal elements equal to square roots of the diagonal elements of D; since there are two possible choices for a square root of each

diagonal element of D, there are 2^n choices for the matrix $D^{1/2}$.(notes: 1- D may have complex values. 2- A must has n linearly independent eigenvectors)

So
$$\sqrt{A}=A^{1/2}=VD^{1/2}V^{-1}$$
 because $\left(VD^{rac{1}{2}}V^{-1}
ight)^2=VD^{rac{1}{2}}\left(V^{-1}V
ight)D^{rac{1}{2}}V^{-1}=VDV^{-1}=A$

In [4]: error = 0for _ in range(100): B = generate dataset(matrix size) # B = create_random_walk_matrix(matrix_size) # print(f'B:\n{B}\n') # Eigendecomposition of a matrix eigen_values, eigen_vectors = np.linalg.eig(B) eigen_values = np.diag(eigen_values) eigen values square = np.sqrt(eigen values, dtype=np.complex) A = (eigen_vectors @ eigen_values_square @ np.linalg.pinv(eigen_vectors)) # A = A.astype(float)reconstruct_B_with_rooted_square = np.round((A @ A), 3) reconstruct B with rooted square = \ reconstruct_B_with_rooted_square.astype(float) # print(np.sum(A, axis=1)) # print(f'A @ A:\n{reconstruct B with rooted square}\n') error += np.linalg.norm(B - reconstruct B with rooted square, 'fro') print(f'The average error in 100 random matrices is {error/100}') print('pause')

The average error in 100 random matrices is 0.09996095845172331 pause

But it has some issues, what if the eigenvectors weren't independent? Then we don't have an inverse for it and we can't use pseudo inverse

In some cases, even when it had inverse, the error was around 32

Second Idea

Using alternate search method

In each step I select a row of the matrix (vertex in the graph) and choose two elements of that row (two edges connected to our vertex) because it's a random walk, the sum of each row should be 1 so if I decrease one element by x, I must add other element x. to find which x should we use to decrease error(optimize our variable) I use two different methods:

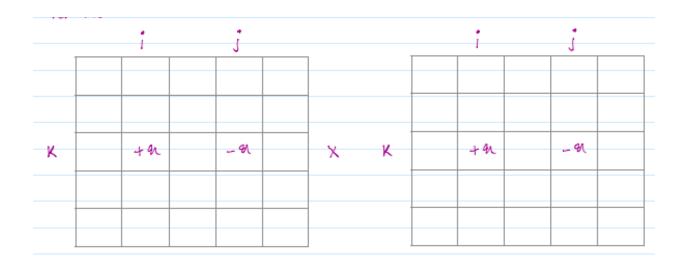
1. #### Adaptive learning rate

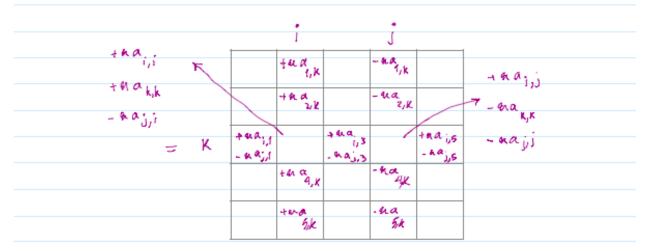
Adaptive learning rate(x)

In this method simply consider the learning rate to be 1 at the first and then solve the problem until there is no improvement (after some specific iterations) then divide the learning rate by two until the learning rate leads to zero and by choosing deviation by two, all the best learning rate can be achievable (all numbers are the sum of 2-powers)

Closed form

In this method, I write a quadratic equality equation based on error by changing x and find the best x by solving it





```
In [2]:
    from copy import deepcopy
    import numpy as np
    from my_io import generate_dataset
    import warnings
    warnings.filterwarnings('ignore')
```

matrix_size = 5
decimal_point = 4
random test = 100

```
def err_improve(B, A, i, j, k, x):
   Ap = deepcopy(A)
   Ap[k, i] += x
   Ap[k, j] = x
   return np.linalg.norm(B - A@A, 'fro') - np.linalg.norm(B - Ap@Ap, 'fro')
def find_root_square(B, method, decimal_point):
   A = deepcopy(B)
   matrix size = B.shape[0]
   old_err = 1000
   new_err = 100
   no improve = 0
   best err = 1000
   best_A = None
   learning rate = 1
    iteration = 0
    if(method == 'adaptive_lerning_rate'):
       while(learning_rate >= 10**-decimal_point):
           iteration += 1
           A2 = A@A
           E = B - A2
            i, j, k = np.random.randint(0, matrix size, 3)
           while(i == j):
                i, j, k = np.random.randint(0, matrix size, 3)
            if(err_improve(B, A, i, j, k, learning_rate) > 0
                  and A[k, j] >= learning rate):
                A[k, i] += learning_rate
                A[k, j] -= learning_rate
            if(err_improve(B, A, i, j, k, -learning_rate) > 0
                    and A[k, i] >= learning rate):
                A[k, i] += -learning rate
                A[k, j] -= -learning_rate
            new_err = np.linalg.norm(B - A@A, 'fro')
            # print(new err)
            if(no improve > 100):
                learning rate /= 2
                learning_rate = np.round(learning_rate, decimal_point)
                no improve = 0
            if(best_err - new_err >= 10**-decimal_point):
                best err = new err
                no improve = 0
               best A = deepcopy(A)
            else:
                no improve += 1
    if(method == 'closed_form'):
```

```
while(no improve < 1000):</pre>
            iteration += 1
            A2 = A@A
            E = B - A2
            i, j, k = np.random.randint(0, matrix size, 3)
            while(i == j):
                i, j, k = np.random.randint(0, matrix_size, 3)
            # i, j, k = 1, 3, 1
            C = np.zeros((matrix_size, matrix_size))
            C[k][i], C[k][j] = 1, -1
            a4, a3, a2, a1, a0 = [0]*5
            sum_new_val = 0
            for row in range(matrix_size):
                for col in range(matrix_size):
                    new row = A[row, :] + C[row, :]
                    new_col = A[:, col] + C[:, col]
                    new_val = sum(new_row * new_col)
                    sum_new_val += new_val
                    delta_val = new_val - A2[row, col]
                    a2 += delta_val**2
                    a1 += 2 * delta_val * E[row, col]
                    a0 += E[row, col]**2
            roots = np.roots((a2, a1, a0))
            if(roots[0] != 0):
                root = roots[0]
            else:
                root = roots[1]
            if(type(root) == np.complex_):
                root = root.astype(float)
            A[k][i] = root
            A[k][j] += root
            old err = new err
            new err = np.linalg.norm(B - A@A, 'fro')
            # if(new err > old err):
                 print(new err)
            if(best_err - new_err >= 10**-decimal_point):
                best_err = new_err
                no improve = 0
                best A = deepcopy(A)
            else:
                no improve += 1
    return best A, best err, iteration
avg err adaptive lerning rate = 0
avg err closed form = 0
avg iter adaptive lerning rate = 0
avg iter closed form = 0
for in range(random test):
   B = generate_dataset(5, decimal=decimal_point)
```

```
best A, adaptive lerning rate err, iteration adaptive lerning rate =\
        find_root_square(B, 'adaptive_lerning_rate', decimal_point)
    best_A, closed_form_err, iteration_closed_form =\
        find_root_square(B, 'closed_form', decimal_point)
    avg_err_adaptive_lerning_rate += adaptive_lerning_rate_err
    avg_iter_adaptive_lerning_rate += iteration_adaptive_lerning_rate
    avg err closed form += closed form err
    avg_iter_closed_form += iteration_closed_form
avg_err_adaptive_lerning_rate /= random_test
avg_iter_adaptive_lerning_rate /= random_test
avg err closed form /= random test
avg_iter_closed_form /= random_test
print(
    f'The average error in {random_test}',
    f'random matrices is by adaptive lerning rate is {avg err adaptive lerning r
print(f'The average iteration in {random_test} random matrices by adaptive lerni
print(
    f'The average error in {random test}',
    f'random matrices is by closed form rate is {avg_err_closed_form}')
print(f'The average iteration in {random_test} random matrices by closed form ra
print('pause')
The average error in 100 random matrices is by adaptive lerning rate is 0.162374
21294503832
The average iteration in 100 random matrices by adaptive lerning rate is 2605.01
```

The average error in 100 random matrices is by closed form rate is 0.10707642347

The average iteration in 100 random matrices by closed form rate is 4573.09 pause

Conclusion

Based on my results I think alternate search with adaptive learning rate is the best method in my homework and it's not too slow because in eigendecomposition we should have a positive definite matrix so it's not always working and adaptive learning rate method use less iteration but closed-form has less error with more iteration

Thank you very much for taking the time to read