BIS634 - Assignment4

Name: Zhiyuan Cao; NetID: zc347

Exercise 1

Implementation of the gradient descent algorithm

I realize the two-dimensional gradient descent algorithm by

```
def derivative_a(a, b, delta=1e-4):
    return (get_error(a+delta, b) - get_error(a, b)) / delta

def derivative_b(a, b, delta=1e-4):
    return (get_error(a, b+delta) - get_error(a, b)) / delta

def find_minimum(a, b, delta=1e-4, step_len=0.1, stop_thres=1e-4):
    prev_error = get_error(a, b) - 2*stop_thres
    curr_error = get_error(a, b)
    while (abs(curr_error - prev_error) > stop_thres):
        da = derivative_a(a,b,delta)
        db = derivative_b(a,b,delta)
        a -= step_len*da
        b -= step_len*db
        prev_error = curr_error
        curr_error = get_error(a, b)
    return (a, b, curr_error)
```

Explanation of derivative estimation: I calculate the derivative of a using the formula $da = \dfrac{f(a+h,b)-f(a,b)}{h}$. Here I set h as 1×10^{-4} to approximate the derivative. For b, the derivative db is calculated in the same way.

Explanation of numerical choices: Besides setting h as 1×10^{-4}, I set the stopping criteria to be 1×10^{-4}. Also, I set the step length of each iteration to be 0.1.

Justification: h should be set as smaller as possible. After several trails, I think 1×10^{-4} has already met my requirements. For step length, a smaller step length will make the final result more accurate while the speed of algorithm will be slower. After several tests, I choose step length to be 0.1 as a trade-off result. Finally, for stopping threshold, I choose 1×10^{-4} to balance the runtime and accuracy. By running the algorithm using different initial values of a and b and obtaining similar results in a reasonable running speed, I justify that my chioces are reasonable.

Local minimum and global minimum

By vary a and b from 0.1 to 0.9 respectively and considering 81 different initial cases, I obtain different local minimums. Part of the results are shown below

In [54]:
```python
import numpy as np
a_range = [.1, .2, .3, .4, .5, .6, .7, .8, .9]
b_range = [.1, .2, .3, .4, .5, .6, .7, .8, .9]
a, b, c, d = find_global(a_range, b_range)
```

```
Initial (a, b) = (0.1, 0.1):
Local minimum: 1.10011820098 with (a, b) = (0.213861230000456, 0.6783404199999342)
----------
Initial (a, b) = (0.1, 0.2):
Local minimum: 1.10012940829 with (a, b) = (0.21333904000010442, 0.6779398200001061)
----------
Initial (a, b) = (0.1, 0.3):
Local minimum: 1.10013193602 with (a, b) = (0.2126862899996084, 0.6780020300001575)
----------
Initial (a, b) = (0.1, 0.4):
Local minimum: 1.10012143141 with (a, b) = (0.21187035999982165, 0.678783469999766)
----------
Initial (a, b) = (0.1, 0.5):
Local minimum: 1.10015021737 with (a, b) = (0.2095755899989115, 0.6785623599992459)
----------
Initial (a, b) = (0.1, 0.6):
Local minimum: 1.10015934818 with (a, b) = (0.20598997000021982, 0.681309260000495)
----------
Initial (a, b) = (0.1, 0.7):
Local minimum: 1.10015754259 with (a, b) = (0.20349995999958673, 0.6901364799996805)
----------
Initial (a, b) = (0.1, 0.8):
Local minimum: 1.10012176475 with (a, b) = (0.2079819900002832, 0.6965813100000322)
----------
Initial (a, b) = (0.1, 0.9):
Local minimum: 1.10017473551 with (a, b) = (0.20957558000024293, 0.7005525900004465)
----------
Initial (a, b) = (0.2, 0.1):
Local minimum: 1.10017737622 with (a, b) = (0.2155908400001348, 0.6756880200000758)
----------
Initial (a, b) = (0.2, 0.2):
Local minimum: 1.10012249521 with (a, b) = (0.21559083999991274, 0.6779398100007714)
----------
Initial (a, b) = (0.2, 0.3):
Local minimum: 1.10012120431 with (a, b) = (0.21550103999958808, 0.6780020299997134)
```

Then I check these minimums. If the distance between two points (a_1, b_1) and (a_2, b_2) are close enough, i.e. the l2 norm of $(a_1 - a_2,\ b_1 - b_2)$ is smaller than a threshold, I consider them as the same local minimum and keep the smaller one. After this filtering process, there is one local minimum and one golbal minimum left.

In [86]:
```python
e, f = find_different_local(a, b)
for i, item in enumerate(e):
    print(f"One local minimum is {item} with (a, b) = {f[i]}")
print(f"The global minimum is {c} with (a, b) = {d}")
```

```
One local minimum is 1.10011345278 with (a, b) = (0.2264713099998421, 0.6909504999995424)
One local minimum is 1.00000318633 with (a, b) = (0.7114914600000016, 0.1698963799999117)
The global minimum is 1.00000318633 with (a, b) = (0.7114914600000016, 0.1698963799999117)
```

The local minimum is 1.10011345278 with the corresponding (a, b) = (0.2264713099998421, 0.6909504999995424).

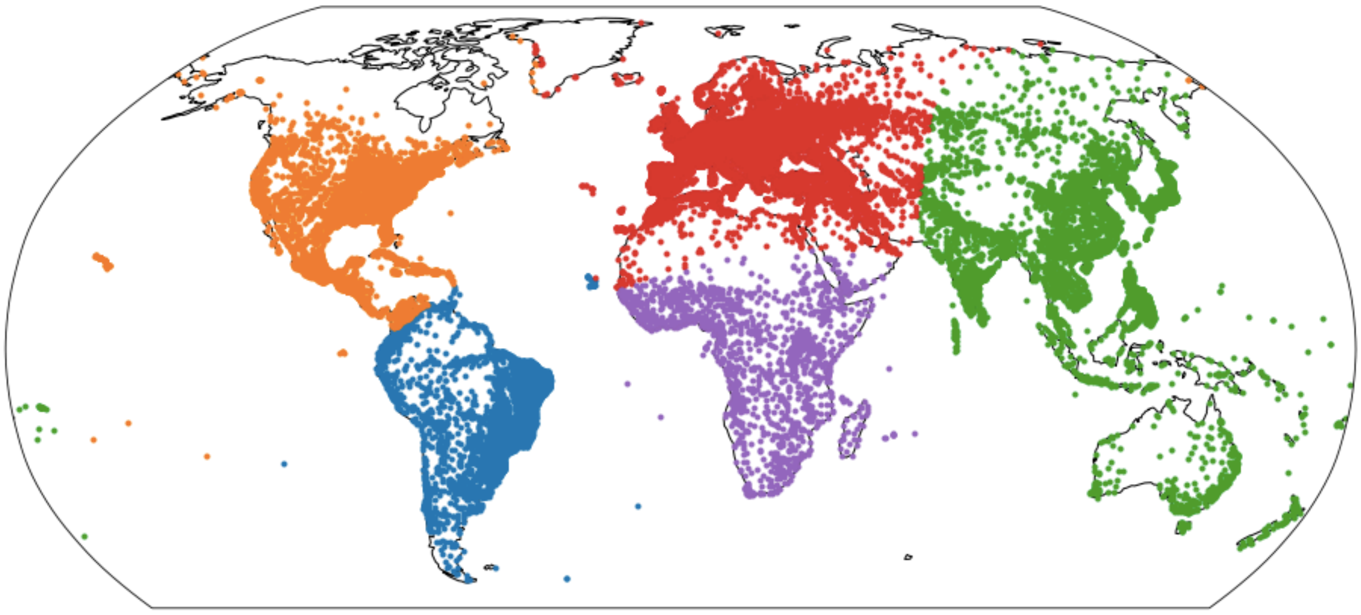
The global minimum is 1.00000318633 with the corresponding (a, b) = (0.7114914600000016, 0.1698963799999117).

Exercise 2

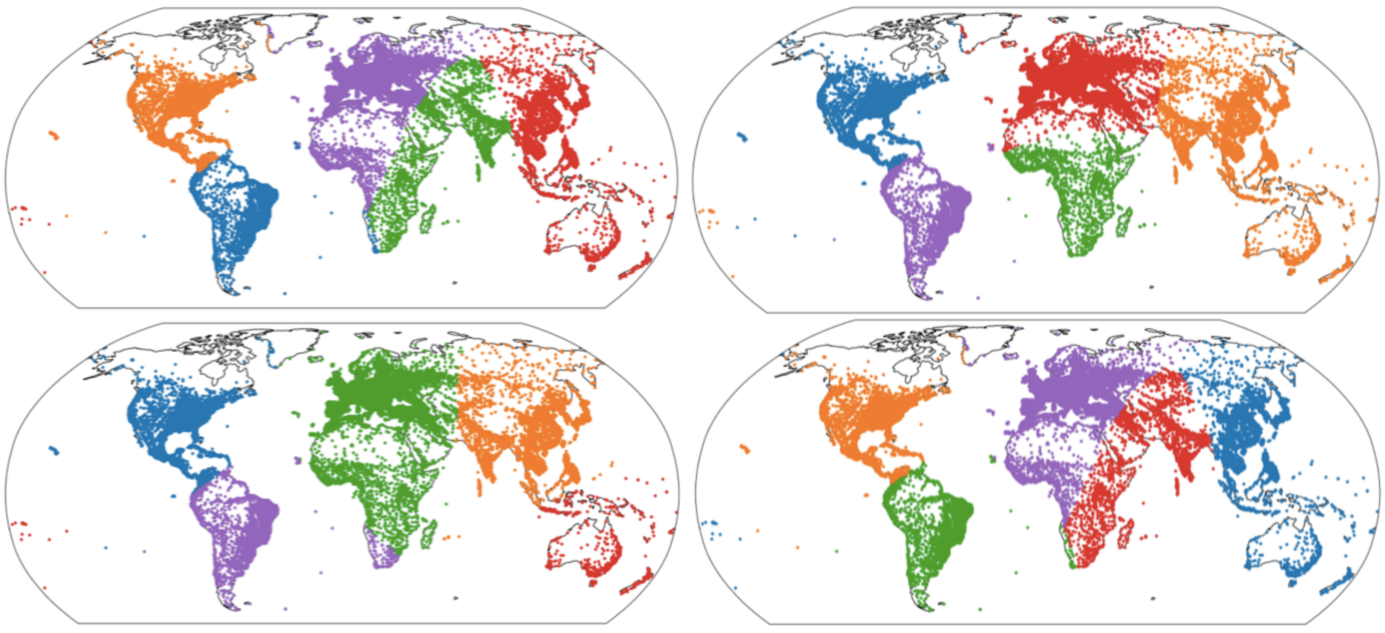
I modified the k-means code from slides by using the Haversine metric as distance. See my code for detailed implementation.

Then I visualize my result with a color-coded scatter plot using an appropriate map projection cartopy.

Results for k = 5

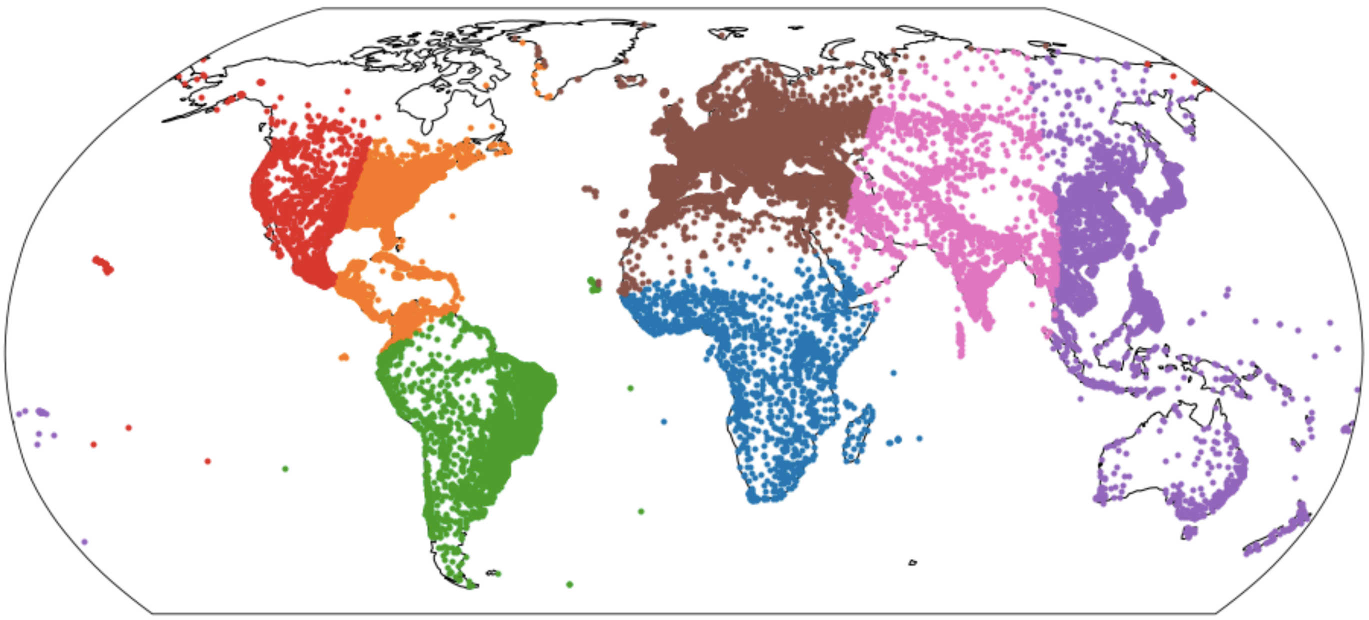


Variation for k = 5:

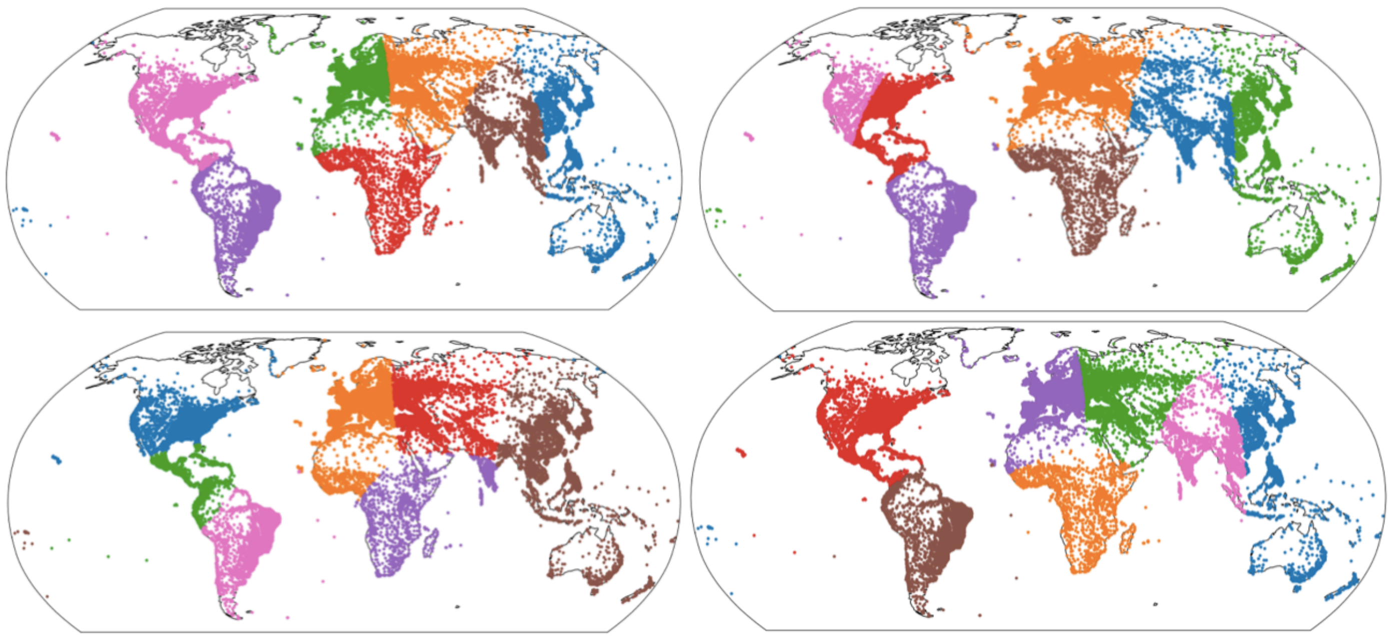


Comments on diversity: I plotted five graphs for k=5. From the results, I find that the initialization of centers do have an non-ignorable influence on the final result. For the American continent, algorithms will always split the points into two clusters. For the other continents, the clusters have more diversity.

Results for k = 7

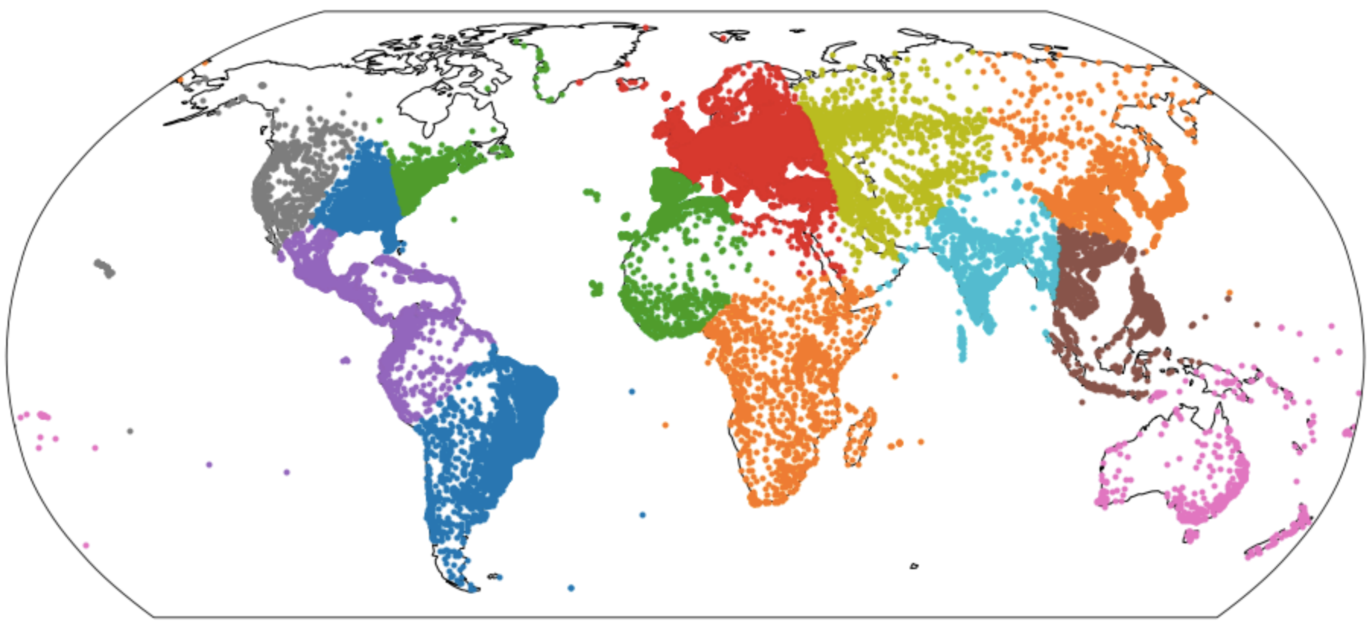


Variation for k = 7:

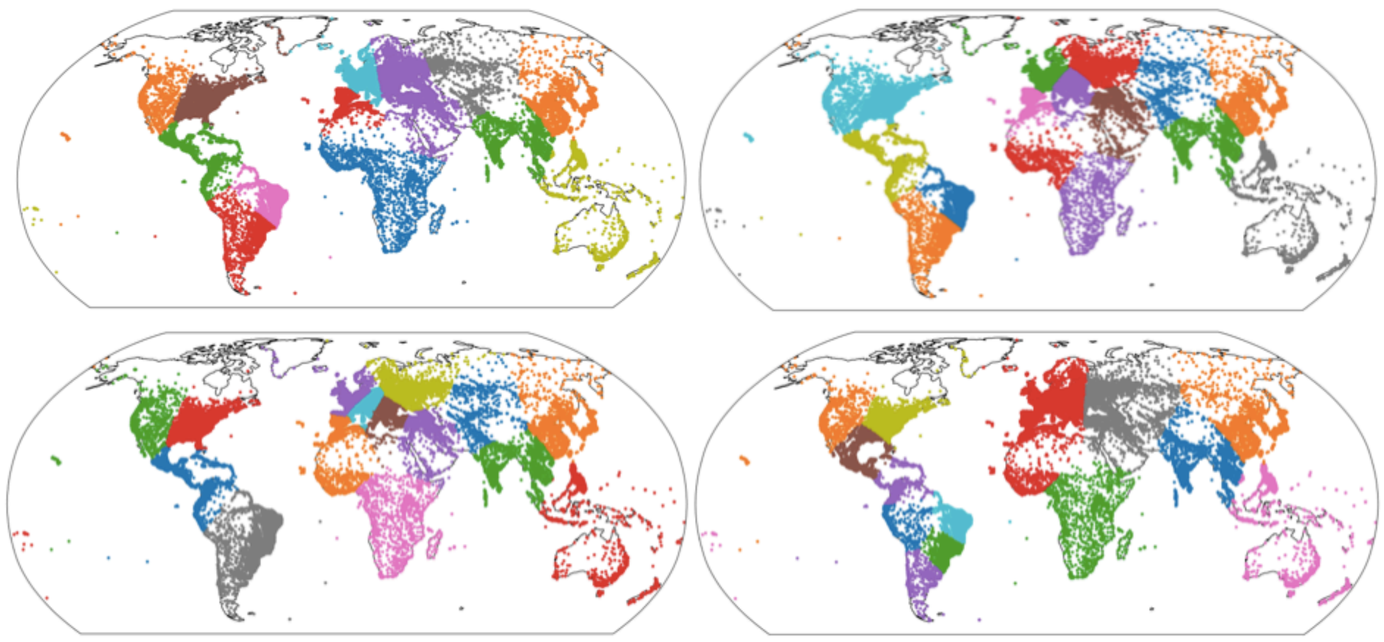


Comments on diversity: Similarly, I plotted five graphs for k=7. From the results, I find that the initialization of centers do have a larger influence on the final result. For the American continent, algorithms will always split the points into two or three clusters. For the other continents, the clusters have more diversity.

Results for k = 15



Variation for k = 15:



Comments on diversity: Finally, I plotted five graphs for k=15. From the results, I find that the performance become worse when k turns to 15. Many of the clusters do not seems to be reasonable enough. Also, the clusters becomes more randomly distributed.

Exercise 3

I implement `Fibonacci` and `Fibonacci_cache` as follows:

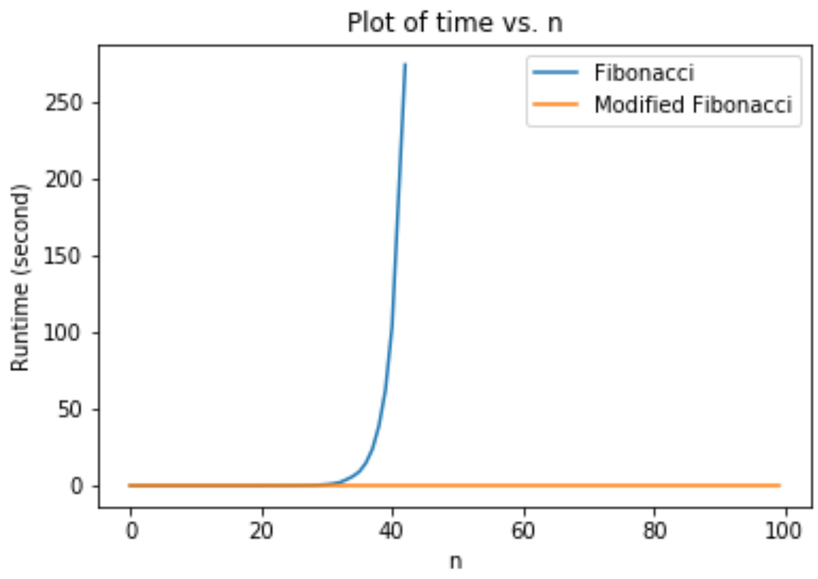
```python
def Fibonacci(n):
    if (n == 0 or n == 1 or n == 2):
        return math.ceil(n/2)
    else:
        return Fibonacci(n-1) + Fibonacci(n-2)
```

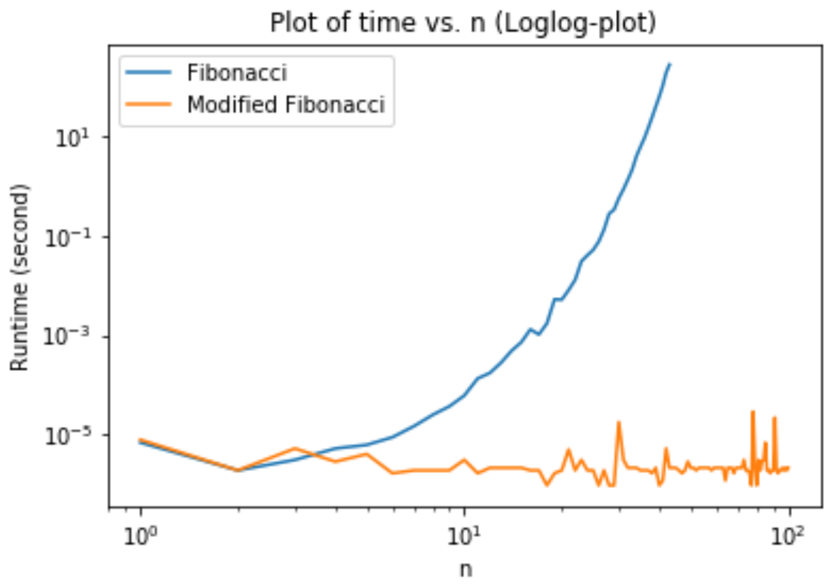
```python
def Fibonacci_lru_cache(n, cache):
    if (n == 0 or n == 1 or n == 2):
        return math.ceil(n/2)
    elif (not n in cache):
        cache[n] = Fibonacci_lru_cache(n-1, cache) + Fibonacci_lru_cache(n-2, cache)
    return cache[n]
```

Then I test these function to make sure that I implement them correctly.

I print the first 30 elements of Fibonacci sequence and find that the result goes as expected.

Then I test the runtime of these two algorithms with different n. For `Fibonacci`, I use n from 0 to 43, because larger n will make the runtime extremely slow, while for `Fibonacci_cache`, I choose n from 0 to 100, because Fibonacci_cache runs in a relatively constant and fast speed. Then I plot the graph of time vs. n. The result is shown below.





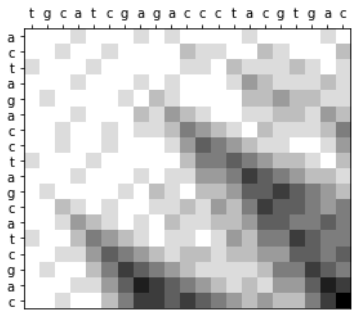
From the result, we can observe that after using lru_cache, the speed of modified `Fibonacci_cache` is much more faster than `Fibonacci`, because `Fibonacci_cache` stores all the values in a dictionary which makes the whole process faster.

Exercise 4

I implemented the Smith-Waterman algorithm by scratch. See my code for details.

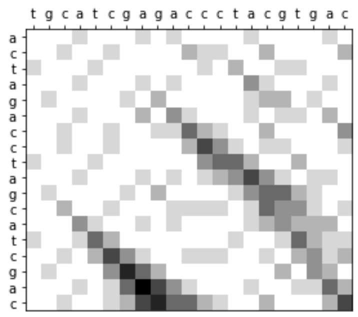
I first test my code using ***'tgcatcgagaccctacgtgac'*** and ***'actagacctagcatcgac'*** with match=1, gap_penalty=1, mismatch_penalty=1. The result is shown below.

```
In [321]:  align('tgcatcgagaccctacgtgac', 'actagacctagcatcgac')
Out[321]: ('agacccta-cgt-gac', 'aga-cctagcatcgac', 8.0)
```



Then I test my code using ***'tgcatcgagaccctacgtgac'*** and ***'actagacctagcatcgac'*** but change the gap_penalty as 2, i.e. match=1, gap_penalty=2, mismatch_penalty=1. The result is shown below.

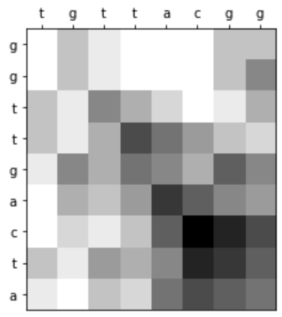
```
In [305]: align('tgcatcgagaccctacgtgac', 'actagacctagcatcgac', gap_penalty=2)
Out[305]: ('gcatcga', 'gcatcga', 7.0)
```



We can observe that the matrix becomes fainter.

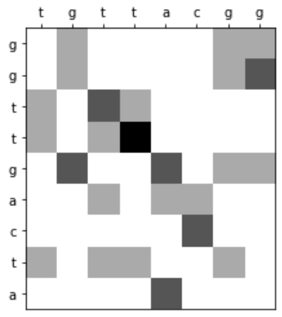
After that I test my code using ***'tgttacgg'*** and ***'ggttgacta'*** but change the gap_penalty as 2, i.e. match=3, gap_penalty=2, mismatch_penalty=3. The result is shown below.

```
In [306]: align('tgttacgg', 'ggttgacta', match=3, mismatch_penalty=3, gap_penalty=2)
Out[306]: ('gtt-ac', 'gttgac', 13.0)
```



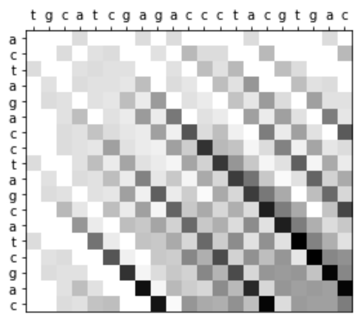
Then I change the gap_penalty to be 100. The expected result is ***'gtt'***.

```
In [307]: align('tgttacgg', 'ggttgacta', match=3, mismatch_penalty=3, gap_penalty=100)
Out[307]: ('gtt', 'gtt', 9.0)
```



Finally I change match and gap penalty. The result is exactly what we expect.

```
In [326]: align('tgcatcgagaccctacgtgac', 'actagacctagcatcgac', match=10, gap_penalty=100)
Out[326]: ('cgagaccctacgt', 'ctagacctagcat', 75.0)
```



Appendix

Exercise 1

```
import requests
def get_error(a, b):
    return float(requests.get(f"http://ramcdougal.com/cgi-bin/error_function.py?a=
{a}&b={b}",
                              headers={"User-Agent": "MyScript"}).text)

def derivative_a(a, b, delta=1e-4):
    return (get_error(a+delta, b) - get_error(a, b)) / delta

def derivative_b(a, b, delta=1e-4):
    return (get_error(a, b+delta) - get_error(a, b)) / delta

def find_minimum(a, b, delta=1e-4, step_len=0.1, stop_thres=1e-4):
    prev_error = get_error(a, b) - 2*stop_thres
    curr_error = get_error(a, b)
    while (abs(curr_error - prev_error) > stop_thres):
        da = derivative_a(a,b,delta)
        db = derivative_b(a,b,delta)
        a -= step_len*da
        b -= step_len*db
        prev_error = curr_error
        curr_error = get_error(a, b)
    return (a, b, curr_error)

# Helper function 1 to distinguish different local minimum
def dist_ab(a, b):
    return np.linalg.norm(tuple(map(lambda i, j: i - j, a, b)))

# Helper function 2 to distinguish different local minimum
def dist_ab_check(x, ls, thres=.1):
    for item in ls:
        if dist_ab(x, item) < thres:
            return False
    return True
```

```
from tqdm import tqdm
def find_global(a_range, b_range, delta=1e-4, step_len=0.2, stop_thres=1e-3):
    x, y, z = find_minimum(a_range[0], b_range[0], delta, step_len, stop_thres)
    local_min = [z]
    local_min_ab = [(x, y)]
    global_min = local_min[0]
    global_min_ab = local_min_ab[0]
    for i in a_range:
        for j in b_range:
            x, y, z = find_minimum(i, j)
            local_min.append(z)
            local_min_ab.append((x, y))
            print(f"Initial (a, b) = ({i}, {j}): ")
            print(f"Local minimum: {z} with (a, b) = ({x}, {y})")
            print("-"*10)
            if (z <= global_min):
                global_min = z
                global_min_ab = (x, y)
    return (local_min, local_min_ab, global_min, global_min_ab)
```

```
def find_different_local(local_min, local_min_ab):
    # Find local minimums that are different
    local_min_different = []
    local_min_ab_different = []
    for i, item in enumerate(local_min_ab):
        if (len(local_min_different) == 0):
            local_min_different.append(local_min[i])
            local_min_ab_different.append(item)
        else:
            if (dist_ab_check(item, local_min_ab_different, thres=.1)):
                local_min_different.append(local_min[i])
                local_min_ab_different.append(item)
            else:
                for j, itemj in enumerate(local_min_ab_different):
                    if (dist_ab(item, itemj) < .1 and local_min[i] <
local_min_different[j]):
                        local_min_different[j] = local_min[i]
                        local_min_ab_different[j] = item
                        break
    return (local_min_different, local_min_ab_different)

xx, yy, zz = find_minimum(.4, .2)
print(f"Initial (a, b) = (0.4, 0.2): ")
print(f"Local minimum: {zz} with (a, b) = ({xx}, {yy})")
```

```
import numpy as np
a_range = [.1, .2, .3, .4, .5, .6, .7, .8, .9]
b_range = [.1, .2, .3, .4, .5, .6, .7, .8, .9]
a, b, c, d = find_global(a_range, b_range)
```

```
e, f = find_different_local(a, b)
for i, item in enumerate(e):
    print(f"One local minimum is {item} with (a, b) = {f[i]}")
print(f"The global minimum is {c} with (a, b) = {d}")
```

Exercise 2

```
import pandas as pd
# import plotnine as p9
import cartopy.crs as ccrs
import matplotlib.pyplot as plt
import random
import numpy as np
from math import radians, cos, sin, asin, sqrt

## THE FOLLOWING CODE ARE FROM https://stackoverflow.com/questions/4913349/haversine-
formula-in-python-bearing-and-distance-between-two-gps-points
def haversine(x, y):
    """
    Calculate the great circle distance in kilometers between two points
    on the earth (specified in decimal degrees)
    """

    lon1 = x[0]
    lat1 = x[1]
    lon2 = y[0]
    lat2 = y[1]
    # convert decimal degrees to radians
    lon1, lat1, lon2, lat2 = map(radians, [lon1, lat1, lon2, lat2])

    # haversine formula
    dlon = lon2 - lon1
    dlat = lat2 - lat1
    a = sin(dlat/2)**2 + cos(lat1) * cos(lat2) * sin(dlon/2)**2
    c = 2 * asin(sqrt(a))
    r = 6371 # Radius of earth in kilometers. Use 3956 for miles. Determines return
value units.
    return c * r

def k_means_cities(k):
  df = pd.read_csv("worldcities.csv")
  df = df[['lng', 'lat']]

  pts = [np.array(pt) for pt in zip(df['lng'], df['lat'])]
  centers = random.sample(pts, k)
  old_cluster_ids, cluster_ids = None, [] # arbitrary but different
  while cluster_ids != old_cluster_ids:
      old_cluster_ids = list(cluster_ids)
      cluster_ids = []
      for pt in pts:
          min_cluster = -1
          min_dist = float('inf')
          for i, center in enumerate(centers):
              dist = haversine(pt, center)
              if dist < min_dist:
                  min_cluster = i
                  min_dist = dist
          cluster_ids.append(min_cluster)
      df['cluster'] = cluster_ids
      cluster_pts = [[pt for pt, cluster in zip(pts, cluster_ids) if cluster == match]
for match in range(k)]
      centers = [sum(pts)/len(pts) for pts in cluster_pts]
  fig = plt.figure(figsize=(16,8))
  ax = fig.add_subplot(1, 1, 1, projection=ccrs.Robinson())
  ax.coastlines()

  for i in range(k):
    df_temp = df[df['cluster'] == i]
    ax.plot(df_temp['lng'], df_temp['lat'], "o", transform=ccrs.PlateCarree(),
markersize=3)
```

```
k_means_cities(5)
```

Exercise 3

```
import math
import matplotlib.pyplot as plt
import numpy as np
import time
from tqdm import tqdm

def Fibonacci(n):
    if (n == 0 or n == 1 or n == 2):
        return math.ceil(n/2)
    else:
```

```python
        return Fibonacci(n-1) + Fibonacci(n-2)

def Fibonacci_lru_cache(n, cache):
    if (n == 0 or n == 1 or n == 2):
        r5eturn math.ceil(n/2)
    elif (not n in cache):
        cache[n] = Fibonacci_lru_cache(n-1, cache) + Fibonacci_lru_cache(n-2, cache)
    return cache[n]

cache = {0:0, 1:1, 2:1}
result1 = []
result2 = []
for i in range(30):
    result1.append(Fibonacci(i+1))
    result2.append(Fibonacci_lru_cache(i+1, cache))
print(f"The result of Fibonacci (n <= 30): {result1}")
print(f"The result of modified Fibonacci (n <= 30): {result2}")
```

```python
def time_tester(n, f, *cache):
    start = time.time()
    if len(cache) == 0:
        f(n)
    else:
        f(n, cache[0])
    return time.time() - start

time1 = []
for i in tqdm(range(43)):
    time1.append(time_tester(i+1, Fibonacci))
cache = {0:0, 1:1, 2:1}
time2 = []
for j in tqdm(range(100)):
    time2.append(time_tester(j+1, Fibonacci_lru_cache, cache))

plt.plot(range(43), time1, label="Fibonacci")
plt.plot(range(100), time2, label="Modified Fibonacci")
plt.legend()
plt.xlabel("n")
plt.ylabel("Runtime (second)")
plt.title("Plot of time vs. n")
plt.savefig("README_img/EX3_1.png")
plt.show()

plt.loglog(range(1, 44), time1, label="Fibonacci")
plt.loglog(range(1, 101), time2, label="Modified Fibonacci")
plt.legend()
plt.xlabel("n")
plt.ylabel("Runtime (second)")
plt.title("Plot of time vs. n (Loglog-plot)")
plt.savefig("README_img/EX3_2.png")
plt.show()
```

Exercise 4

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

# Function s(a, b) to evaluate the similarity score of two sequences
def func_s(seq_matrix, seq1, seq2, i, j, match, mismatch_penalty):
    if (seq2[i-1] == seq1[j-1]): # If two charcters to be compared are the same
        return seq_matrix[i-1][j-1] + match
    else: # If two charcters to be compared are different
        return seq_matrix[i-1][j-1] - mismatch_penalty

# Function max(H_{i-k, j} - W_k). Here W_k set to be linear, i.e. W_k = k*W_1
def func_gap_i(seq_matrix, seq1, seq2, i, j, gap_penalty):
    max_val = seq_matrix[i-1][j] - 1*gap_penalty
    for k in range(1, i):
        val = seq_matrix[i-k][j] - k*gap_penalty
        if (val > max_val):
            max_val = val
    return max_val

# Function max(H_{i, j-k} - W_k). Here W_k set to be linear, i.e. W_k = k*W_1
def func_gap_j(seq_matrix, seq1, seq2, i, j, gap_penalty):
    max_val = seq_matrix[i][j-1] - 1*gap_penalty
    for k in range(1, j):
        val = seq_matrix[i][j-k] - k*gap_penalty
        if (val > max_val):
            max_val = val
    return max_val

# Generate the sequence matrix
def gene_seq_matrix(seq1, seq2, match, gap_penalty, mismatch_penalty):
    seq_matrix = np.zeros((len(seq2)+1, len(seq1)+1))
    direction_matrix = np.zeros((len(seq2)+1, len(seq1)+1), dtype='str')
    direction_matrix.fill('e')
    for i, char1 in enumerate(seq2):
        for j, char2 in enumerate(seq1):
            potential_num = [func_s(seq_matrix, seq1, seq2, i+1, j+1, match,
mismatch_penalty),
                             func_gap_i(seq_matrix, seq1, seq2, i+1, j+1, gap_penalty),
                             func_gap_j(seq_matrix, seq1, seq2, i+1, j+1, gap_penalty),
                             0]
            seq_matrix[i+1][j+1] = max(potential_num)
            indx = np.argmax(potential_num)
            num2dir = {0:'d', 1:'u', 2:'l', 3: 'e'}
            direction_matrix[i+1][j+1] = num2dir[indx]
    return seq_matrix, direction_matrix

def find_maximum_indices(seq_matrix):
    max_val = seq_matrix.max()
    max_indices = []
    for i in range(seq_matrix.shape[0]):
        for j in range(seq_matrix.shape[1]):
            if (seq_matrix[i][j] == max_val):
                max_indices.append([i, j])
    return max_indices

# I only choose one case if multiple correct answers are encountered
def trace_back(seq_matrix, direction_matrix, i, j, seq1, seq2, subseq1='', subseq2=''):
    score = seq_matrix.max()
    if seq_matrix[i][j] == 0:
        return subseq1, subseq2, score
    else:
        if direction_matrix[i][j] == 'd':
            return trace_back(seq_matrix, direction_matrix, i-1, j-1, seq1, seq2, 
seq1[j-1]+subseq1, seq2[i-1]+subseq2)
        elif direction_matrix[i][j] == 'l':
            return trace_back(seq_matrix, direction_matrix, i, j-1, seq1, seq2, seq1[j-
1]+subseq1, '-'+subseq2)
        elif direction_matrix[i][j] == 'u':
            return trace_back(seq_matrix, direction_matrix, i-1, j, seq1, seq2, '-
'+subseq1, seq2[i-1]+subseq2)
        else:
            return trace_back(seq_matrix, direction_matrix, i-1, j-1, seq1, seq2, 
seq1[j-1]+subseq1, seq2[i-1]+subseq2)

def plot_matrix(matrix, seq1, seq2):
    plt.xticks(range(len(seq1)), labels=seq1)
    plt.yticks(range(len(seq2)), labels=seq2)
    plt.imshow(matrix[1:, 1:], cmap='binary')
    plt.gca().xaxis.tick_top()


def align(seq1, seq2, match=1, gap_penalty=1, mismatch_penalty=1):
    seq_matrix, direction_matrix = gene_seq_matrix(seq1, seq2, match, gap_penalty, 
mismatch_penalty)
    plot_matrix(seq_matrix, seq1, seq2)
    subseq1, subseq2, score = trace_back(seq_matrix, direction_matrix, 
find_maximum_indices(seq_matrix)[0][0], find_maximum_indices(seq_matrix)[0][1], seq1, 
seq2)
    # subseq1, subseq2, score = trace_back(seq_matrix, find_maximum_indices(seq_matrix)
[0][0], find_maximum_indices(seq_matrix)[0][1], seq1, seq2, match, gap_penalty)
    if (subseq1[0] == '-' or subseq2[0] == '-'):
        subseq1 = subseq1[1:]
        subseq2 = subseq2[1:]
    return subseq1, subseq2, score

align('tgcatcgagaccctacgtgac', 'actagacctagcatcgac')
# align('tgcatcgagaccctacgtgac', 'actagacctagcatcgac', gap_penalty=2)
# align('tgttacgg', 'ggttgacta', match=3, mismatch_penalty=3, gap_penalty=2)
```