PRACTICAL 4

Question 1. (a) Write a program to construct a dot plot for the alignment of **human and chicken hemoglobin beta chain**. Identify the segments, which are same in both sequences. (b) Construct the dot plot manually for the **residues 1-20** and verify with the plot obtained using program

Solution. (a) Below is the code for the program to construct a dot plot for the alignment.

```
import matplotlib.pyplot as plt
import seaborn as sns

# Function to create the dot plot for two sequences
def dot_plot(seq_a, seq_b):
    dot_plot = [[0 for i in range(len(seq_b))] for j in range(len(seq_a))]
    for i in range(len(seq_a)):
        for j in range(len(seq_b)):
            if seq_a[i] == seq_b[j]:
                dot_plot[i][j] = 1
    return dot_plot

seq_human = """
   MVHLTPEEKSAVTALWGKVNVDEVGGEALGRLLVVYPWTQRFFESFGDLSTPDAVMGNPKVKAHGKKVLGAFS
DGLAHLDNLKGTFATLSELHCDKLHVDPENFRLLGNVLVCVLAHHFGKEFTPPVQAAYQKVVAGVANALAHKYH"""
seq_chick = """
   MVHWTAEEKQLITGLWGKVNVAECGAEALARLLIVYPWTQRFFASFGNLSSPTAILGNPMVRAHGKKVLTSFG
DAVKNLDNIKNTFSQLSELHCDKLHVDPENFRLLGDILIIVLAAHFSKDFTPECQAAWQKLVRVVAHALARKYH"""

matrix = dot_plot(seq_human, seq_chick)
sns.heatmap(matrix)
plt.show()
```

LISTING 1. Code to generate dot plot

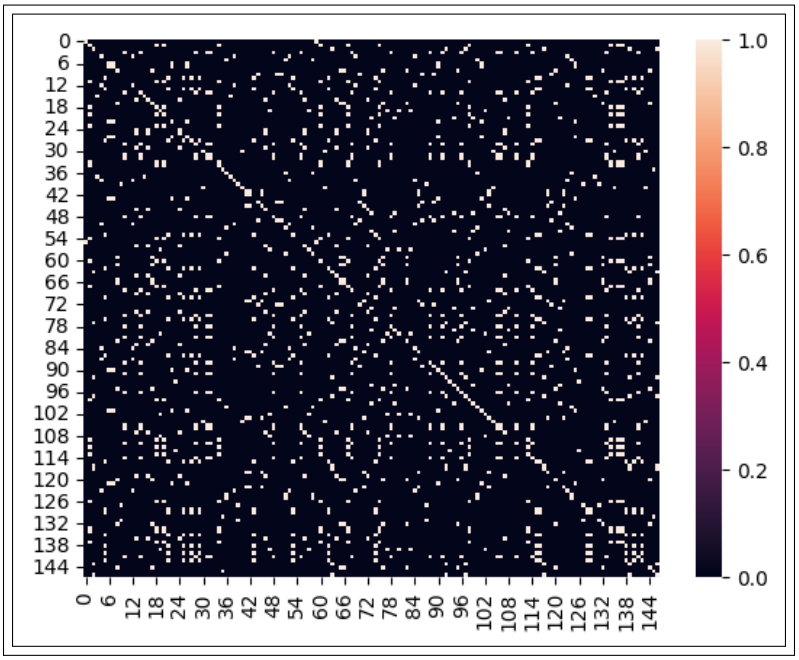


FIGURE 1. Dot plot for alignment of Human and Chicken hemoglobin chain

In the above figure, the numbers on the left and the bottom correspond to the **residue number** of the corresponding hemoglobin chain. The left denotes the **human hemoglobin beta chain**. The bottom denotes the **chicken hemoglobin beta chain**. **White denotes a match**, while the **dark blue denotes a mismatch**.
Observing the dot plot, we can see that a lot of **white points lie along the diagonal**. These correspond to the **same residue positions** on either sequence. This is indicative of the fact that the **hemoglobin beta chains** of humans and chickens have a **lot of residues in common** and in **corresponding positions**.
MVH-T-EEK—T-LWGKVNV-E-G-EAL-RLL-VYPWTQRFF-SFG-LS-P-A–GNP-V-AHGKKVL–F-D—-LDN-K-TF–LSELHCDKLHVDPENFRLLG–L–VLA-HF-K-FTP–QAA-QK-V–VA-ALA-KYH
Given above is the sequence where **-** indicates dissimilarity, while the rest positions are occupied by the respective amino acids.
(b) Considering **dot plots (both manually and program generated)** for residues **1–20** of the above **human and chicken hemoglobin beta chains**.

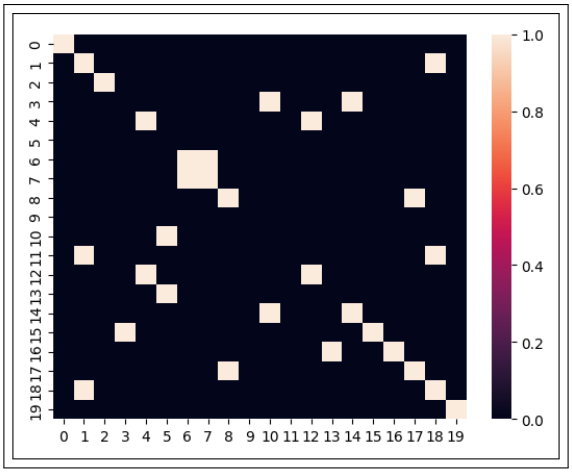


FIGURE 2. Dot plot code for alignment of first 20 residues

On comparing both plots, we can notice that both plots are **exactly the same**, thereby verifying the **correctness of the box plot code**.

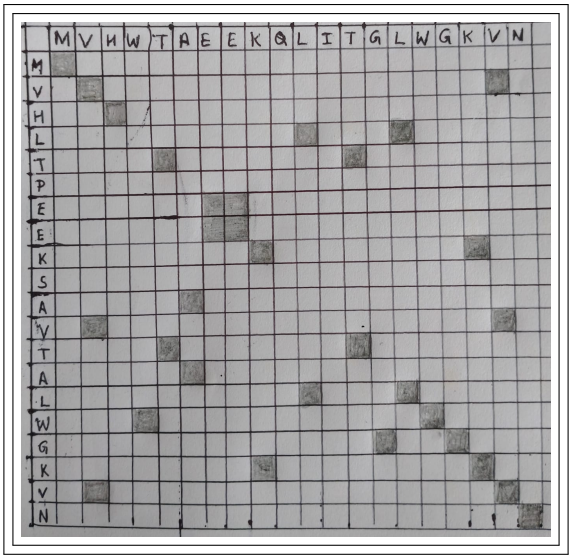


FIGURE 3. Dot plot manual for alignment of first 20 residues

Question 2. Calculate the score for the following alignments using code:
AATCTATA
AAG- -ATA
Assume that the **match score is 1**, the **mismatch score is 0**, the **origination penalty is -2**, and the **length penalty is -1**.

Solution. Below is the code for the program to compute the alignment score between two sequences. There are a set of assumptions here.

- The first sequence is always **complete without any gaps**, otherwise the concept of computing origination penalty will change relatively drastically.
- The second sequence is given **with the gaps enclosed** (if any).
- The **length of first sequence and second sequence are equal**.

```
def calculate_score(seq_1, seq_2, match_score, mismatch_score, origin_penalty
    , len_penalty):
    # Considering length of seq_1 equal to seq_2 after insertion of gaps.
    origin=True
    score = 0
    for i in range(len(seq_1)):
        if (seq_2[i] == "-" and origin == True):
            score += origin_penalty + len_penalty
            origin=False
        elif(seq_2[i] == "-" and origin == False):
            score += len_penalty
        elif(seq_2[i] == seq_1[i]):
            score += match_score
            origin = True
        else:
            score += mismatch_score
            origin = True
    return score

seq_1 = "AATCTATA"
seq_2 = "AAG--ATA"
match_score = +1
mismatch_score = 0
origin_penalty = -2
len_penalty = -1
score = calculate_score(seq_1, seq_2, match_score, mismatch_score,
   origin_penalty, len_penalty)
print(score)
```

Listing 2. Code to compute alignment score

Alignment Score

AATCTATA
AAG- -ATA
The alignment score for above two sequences is +1

Apart from the above two sequences, I have tried computing the alignment scores between various other sequences, which were discussed in class. Below are the results of aligning those sequences.

Alignment Score

AATCTATA
AA–G–ATA
The alignment score for above two sequences is -1

Alignment Score

AATCTATA
AAG–AT–A
The alignment score for above two sequences is -3

Question 3. Verify the Q2 **manually**.

Solution. Below is an image where I have manually calculated the alignment score for the example given in Q2. The so manually obtained alignment score matches with the score obtained via code, which is **+1**.

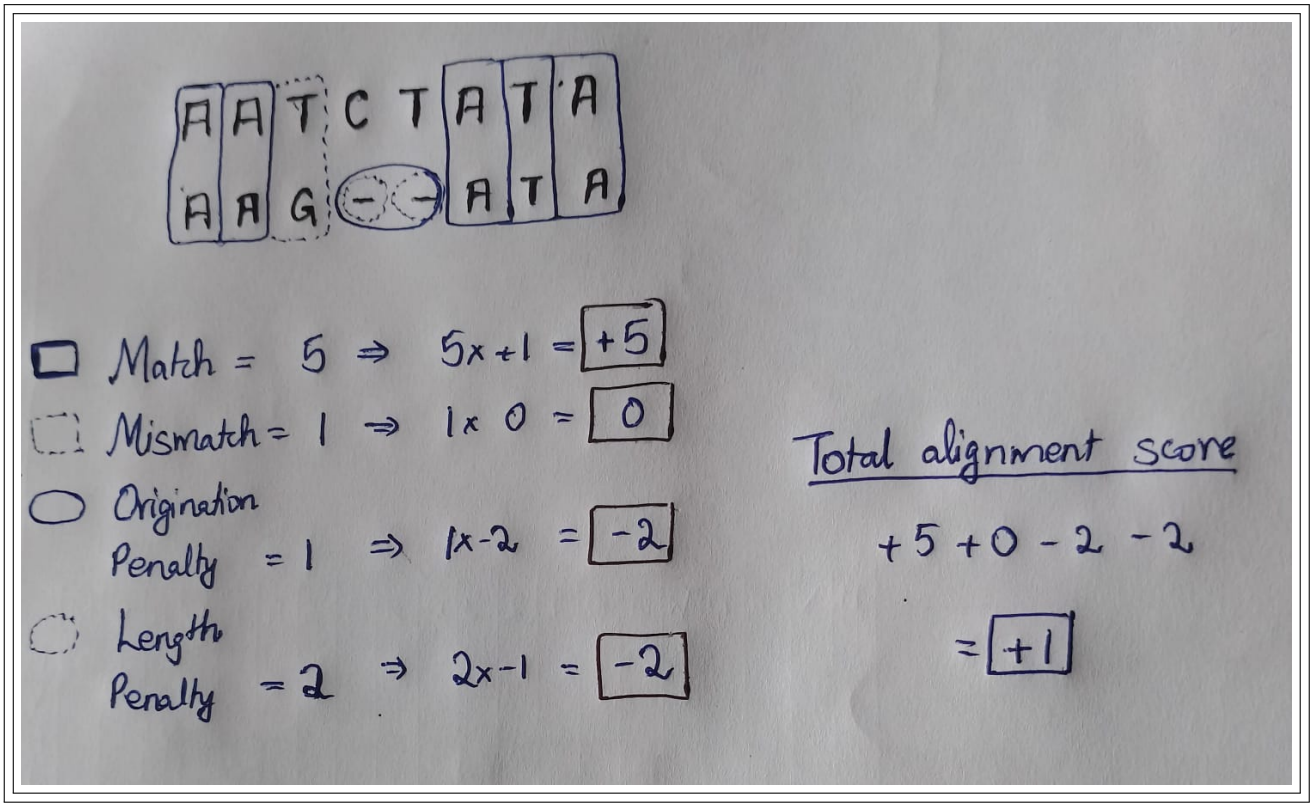


FIGURE 4. Manually calculated alignment score

Question 4. Using the **Needleman and Wunsch dynamic programming method**, construct the partial alignment score table and align the following two sequences (using code):
ACAGTCGAACG and ACCGTCCG
use the scoring parameters: match score: +2; mismatch score: -1; and gap penalty: -2

Solution. The formula for computing the **Needleman and Wunsch dynamic programming method** is as follows:

$$s_{(i,j)} = \max \begin{cases} s_{(i-1,j)} + w(a_i, -) \\ s_{(i,j-1)} + w(-, b_j) \\ s_{(i-1,j-1)} + w(a_i, b_j) \end{cases} \qquad w(a_i, b_j) = \begin{cases} match\_score & \text{if } a_i = b_j \\ mismatch\_score & \text{if } a_i != b_j \\ gap\_penalty & \text{if } a_i = - \text{ or } b_j = - \end{cases}$$

This algorithm is widely used for optimal global alignment. The code for **Needleman and Wunsch algorithm** is given below:

```
# Calculate the matrix for Needleman-Wunsch algorithm
def needleman_wunsch_matrix(seq_1, seq_2, match_score, mismatch_score,
    gap_score):
    matrix = [[0 for i in range(len(seq_2)+1)] for j in range(len(seq_1)+1)]
    for i in range(len(seq_2)+1):
        matrix[0][i] = gap_score*i
    for i in range(len(seq_1)+1):
        matrix[i][0] = gap_score*i
    for i in range(1,len(seq_1)+1):
        for j in range(1, len(seq_2)+1):
            val_1 = matrix[i-1][j] + gap_score
            val_2 = matrix[i][j-1] + gap_score
            val_3 = matrix[i-1][j-1] + match_score if seq_1[i-1] ==
                    seq_2[j-1] else matrix[i-1][j-1] + mismatch_score
            matrix[i][j] = max(val_1, val_2, val_3)
    return matrix

# Backtrack to find the aligned sequences
def backtrack(matrix, seq_1, seq_2, match_score, mismatch_score, gap_score):
    seq_align_1 = ""
    seq_align_2 = ""
    i = len(seq_1)
    j = len(seq_2)
    while(i>0 and j>0):
        if(matrix[i-1][j] == matrix[i][j] - gap_score):
            seq_align_1 += seq_1[i-1]
            seq_align_2 += "-"
            i -= 1
        elif(matrix[i][j-1] == matrix[i][j] - gap_score):
            seq_align_2 += seq_2[j-1]
            seq_align_1 += "-"
            j -= 1
        else:
            seq_align_1 += seq_1[i-1]
            seq_align_2 += seq_2[j-1]
            i -= 1
            j -= 1
    if(i == 0 and j != 0):
        for k in range(j-1,-1,-1):
            seq_align_2 += seq_2[k]
            seq_align_1 += "-"
    elif(j == 0 and i != 0):
        for k in range(i-1,-1,-1):
            seq_align_1 += seq_1[k]
            seq_align_2 += "-"
    seq_align_1 = seq_align_1[::-1]
    seq_align_2 = seq_align_2[::-1]

    return seq_align_1, seq_align_2

seq_1 = "ACAGTCGAACG"
seq_2 = "ACCGTCCG"
match_score = +2
```

```
mismatch_score = -1
gap_score = -2
matrix = needleman_wunsch_matrix(seq_1, seq_2, match_score, mismatch_score,
    gap_score)
seq1, seq2 = backtrack(matrix, seq_1, seq_2, match_score, mismatch_score,
    gap_score)
print(seq1)
print(seq2)

# Output results of seq1 and seq2
ACAGTCGAACG
ACCGTC---CG
```

LISTING 3. Code to compute sequences in Needleman Wunsch Algorithm

The partial alignment score table for the above two sequences in the question is given below. Also enclosed below is the code to generate the **partial alignment score table**. The **matrix** in the code is taken from the above code:

```
seq_1 = "ACAGTCGAACG"
seq_2 = "ACCGTCCG"
sns.heatmap(matrix, annot=True)
plt.xticks(ticks=np.arange(len(matrix[0])) + 0.5, labels=["-", "A", "C", "C",
    "G", "T", "C", "C" , "G"])
plt.yticks(ticks=np.arange(len(matrix)) + 0.5, labels=["-", "A", "C", "A", "G
    ", "T", "C", "G", "A", "A", "C", "G"])
plt.show()
```

LISTING 4. Code to generate alignment table in Needleman Wunsch Algorithm

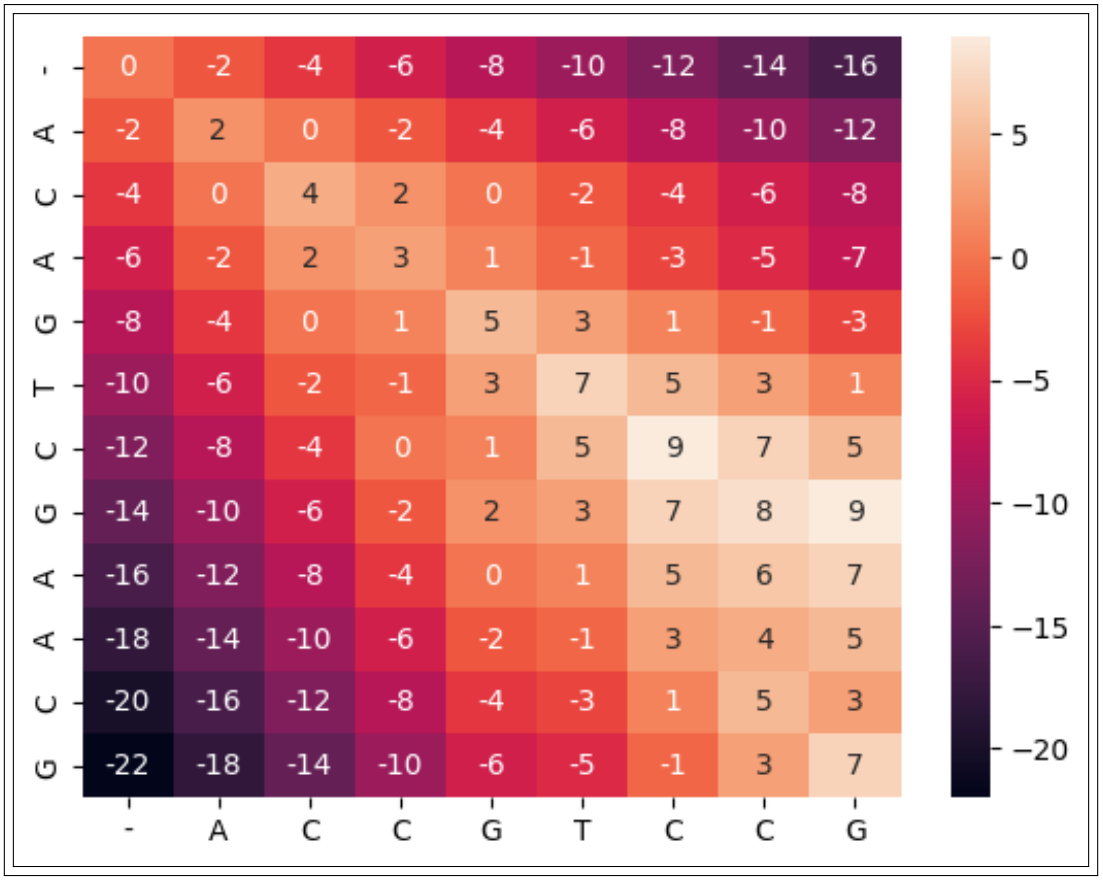
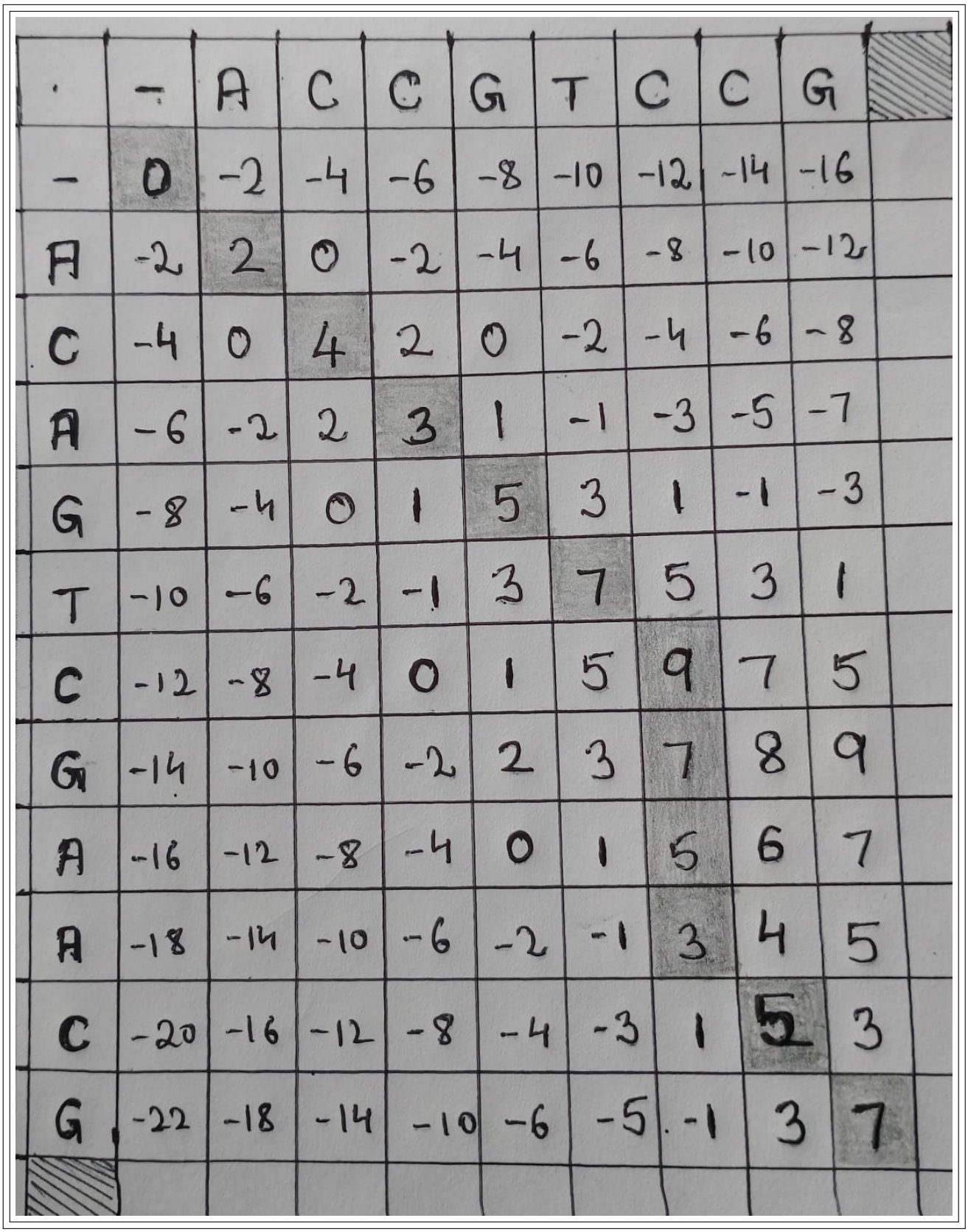


FIGURE 5. Partial alignment score table

Question 5. Verify Q4 **manually**

Solution. The partial alignment score table obtained in manually computing is shown in the image below. The table so obtained is found to be **exactly same** as the one computed by the code above, thereby verifying the correctness of the **Needleman Wunsch algorithm code**.



·	-	A	C	C	G	T	C	C	G	
-	0	-2	-4	-6	-8	-10	-12	-14	-16	
A	-2	2	0	-2	-4	-6	-8	-10	-12	
C	-4	0	4	2	0	-2	-4	-6	-8	
A	-6	-2	2	3	1	-1	-3	-5	-7	
G	-8	-4	0	1	5	3	1	-1	-3	
T	-10	-6	-2	-1	3	7	5	3	1	
C	-12	-8	-4	0	1	5	9	7	5	
G	-14	-10	-6	-2	2	3	7	8	9	
A	-16	-12	-8	-4	0	1	5	6	7	
A	-18	-14	-10	-6	-2	-1	3	4	5	
C	-20	-16	-12	-8	-4	-3	1	5	3	
G	-22	-18	-14	-10	-6	-5	-1	3	7	

FIGURE 6. Manual Needleman Wunsch Algorithm

Question 6. Using the **Smith-Waterman method**, construct the partial alignment scoring table and align the following two sequences (using code):
ACGTATCGCGTATA and **GATGCGTATCG**
scoring parameters: **match score: +2**; **mismatch score: -1**; and **gap penalty: -2**.

Solution. The formula for computing the **Smith and Waterman dynamic programming method** is as follows:

$$s_{(i,j)} = \max \begin{cases} s_{(i-1,j)} + w(a_i, -) \\ s_{(i,j-1)} + w(-, b_j) \\ s_{(i-1,j-1)} + w(a_i, b_j) \\ 0 \end{cases} \qquad w(a_i, b_j) = \begin{cases} match\_score & \text{if } a_i = b_j \\ mismatch\_score & \text{if } a_i != b_j \\ gap\_penalty & \text{if } a_i = - \text{ or } b_j = - \end{cases}$$

This algorithm is widely used for optimal local alignment. The code for **Smith and Waterman algorithm** is given below:

```
# Calculate the matrix for Smith-Waterman algorithm
def smith_waterman_matrix(seq_1, seq_2, match_score, mismatch_score,
    gap_score):
    matrix = [[0 for i in range(len(seq_2)+1)] for j in range(len(seq_1)+1)]
    for i in range(len(seq_2)+1):
        matrix[0][i] = gap_score*i
    for i in range(len(seq_1)+1):
        matrix[i][0] = gap_score*i
    for i in range(1,len(seq_1)+1):
        for j in range(1, len(seq_2)+1):
            val_1 = matrix[i-1][j] + gap_score
            val_2 = matrix[i][j-1] + gap_score
            val_3 = matrix[i-1][j-1] + match_score if seq_1[i-1] ==
            seq_2[j-1] else matrix[i-1][j-1] + mismatch_score
            matrix[i][j] = max(val_1, val_2, val_3, 0)
    return matrix

# Backtrack to find the local aligned sequence
def backtrack(matrix, seq_1, seq_2, match_score, mismatch_score, gap_score):
    seq_align_1 = ""
    seq_align_2 = ""
    i = 0
    j = 0
    maximum = 0
    for r in range(len(seq_1)+1):
        for c in range(len(seq_2)+1):
            if matrix[r][c] > maximum:
                maximum = matrix[r][c]
                i = r
                j = c

    while(matrix[i][j] != 0):
        if(matrix[i-1][j] == matrix[i][j] - gap_score):
            seq_align_1 += seq_1[i-1]
            seq_align_2 += "-"
            i -= 1
        elif(matrix[i][j-1] == matrix[i][j] - gap_score):
            seq_align_2 += seq_2[j-1]
            seq_align_1 += "-"
            j -= 1
        else:
```

```
            seq_align_1 += seq_1[i-1]
            seq_align_2 += seq_2[j-1]
            i -= 1
            j -= 1

    seq_align_1 = seq_align_1[::-1]
    seq_align_2 = seq_align_2[::-1]

    index_start_1 = seq_1.index(seq_align_1)
    index_start_2 = seq_2.index(seq_align_2)

    index_end_1 = index_start_1 + len(seq_align_1)
    index_end_2 = index_start_2 + len(seq_align_2)

    prefix_1 = "-"*(max(index_start_1, index_start_2) - index_start_1) +
                seq_1[:index_start_1]
    prefix_2 = "-"*(max(index_start_1, index_start_2) - index_start_2) +
                seq_2[:index_start_2]

    suffix_1 = seq_1[index_end_1:] + "-"*(max(len(seq_1) - index_end_1,
               len(seq_2) - index_start_2) - (len(seq_1) - index_end_1 + 1))
    suffix_2 = seq_2[index_end_2:] + "-"*(max(len(seq_1) - index_end_1,
               len(seq_2) - index_start_2) - (len(seq_2) - index_end_2 + 1))

    seq_align_1 = prefix_1 + seq_align_1 + suffix_1
    seq_align_2 = prefix_2 + seq_align_2 + suffix_2

    return seq_align_1, seq_align_2

seq_1 = "ACGTATCGCGTATA"
arr_seq_1 = list("-"+seq_1)
seq_2 = "GATGCGTATCG"
arr_seq_2 = list("-"+seq_2)
match_score = +2
mismatch_score = -1
gap_score = -2
matrix = smith_waterman_matrix(seq_1, seq_2, match_score, mismatch_score,
   gap_score)
seq1, seq2 = backtrack(matrix, seq_1, seq_2, match_score, mismatch_score,
   gap_score)
print(seq1)
print(seq2)

# Output results of seq1 and seq2
---ACGTATCGCGTATA
GATGCGTATCG------
....CGTATCG...... # The matching local alignment
```

Listing 5. Code for Smith and Waterman Algorithm

The partial alignment score table for the above two sequences in the question is given below. Also enclosed below is the code to generate the **partial alignment score table**. The **matrix** in the code is taken from the above code:

```
seq_1 = "ACGTATCGCGTATA"
arr_seq_1 = list("-"+seq_1)
seq_2 = "GATGCGTATCG"
```

```
arr_seq_2 = list("-"+seq_2)

# Plot the heatmap
sns.heatmap(matrix, annot=True)
plt.xticks(ticks=np.arange(len(matrix[0])) + 0.5, labels=arr_seq_2)
plt.yticks(ticks=np.arange(len(matrix)) + 0.5, labels=arr_seq_1)
plt.show()
```

LISTING 6. Code to generate alignment table in Smith and Waterman algorithm

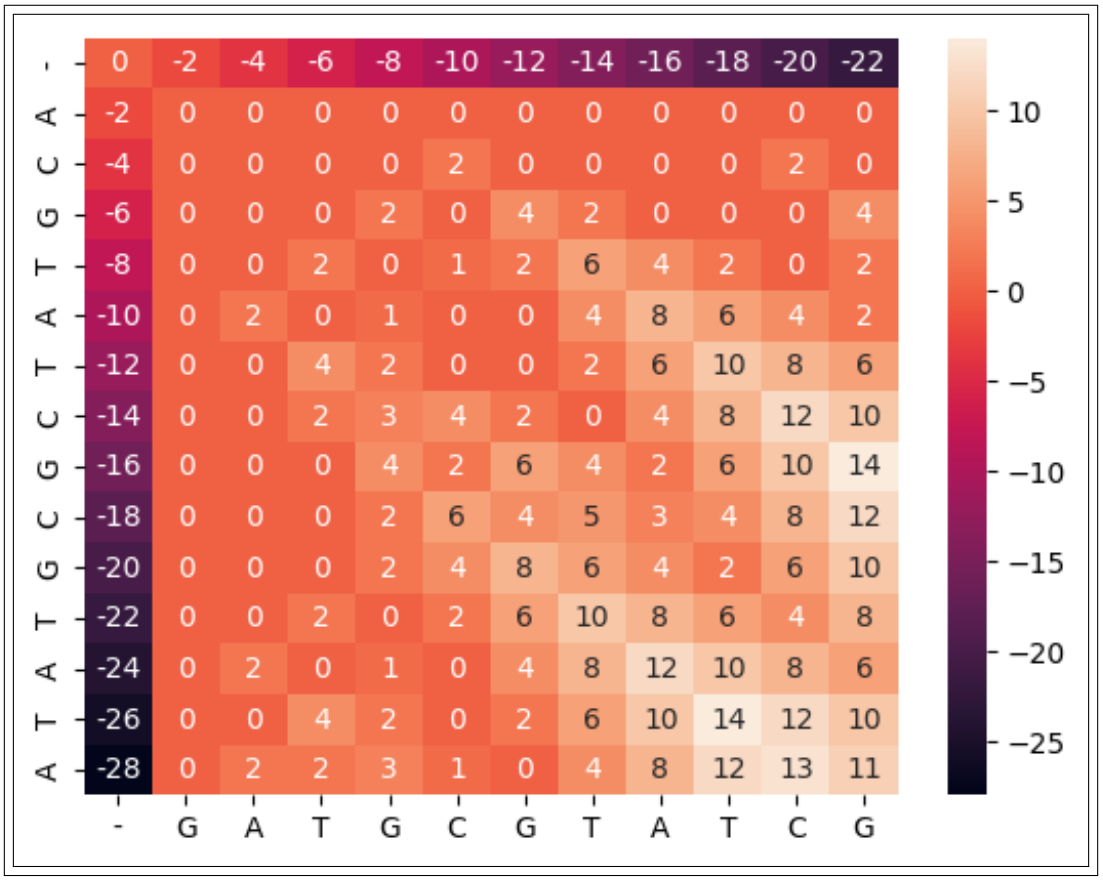


FIGURE 7. Partial alignment score table