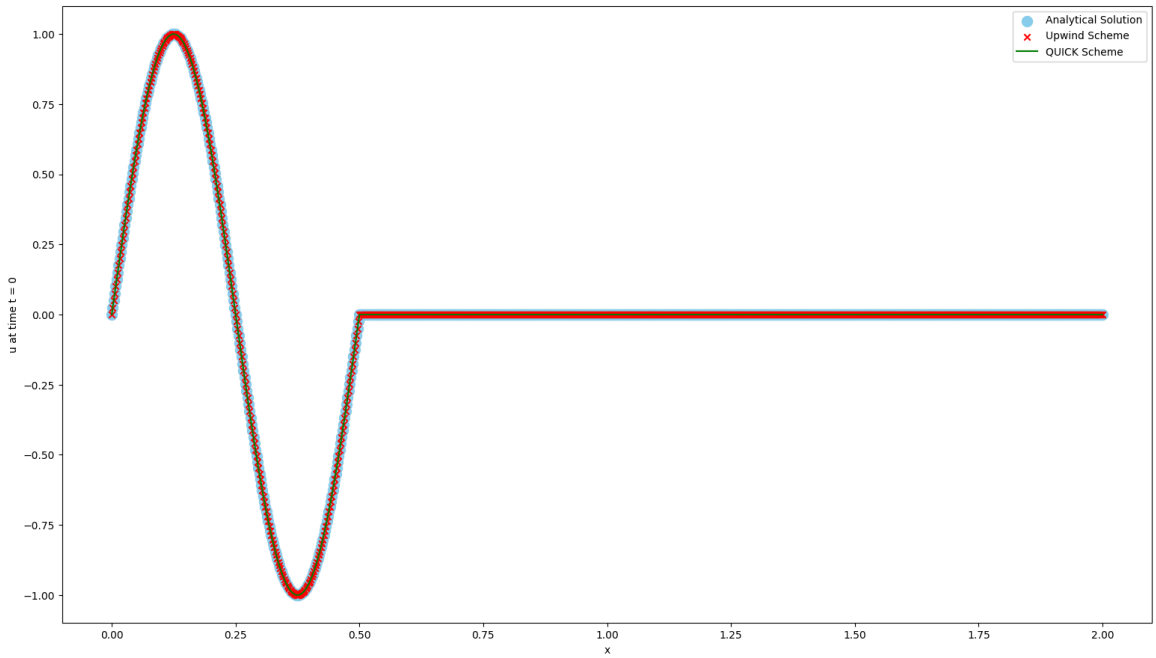
ID5310 - Assignment 2

Rishabh Upadhayay
Roll : NA20B052

1. (a)

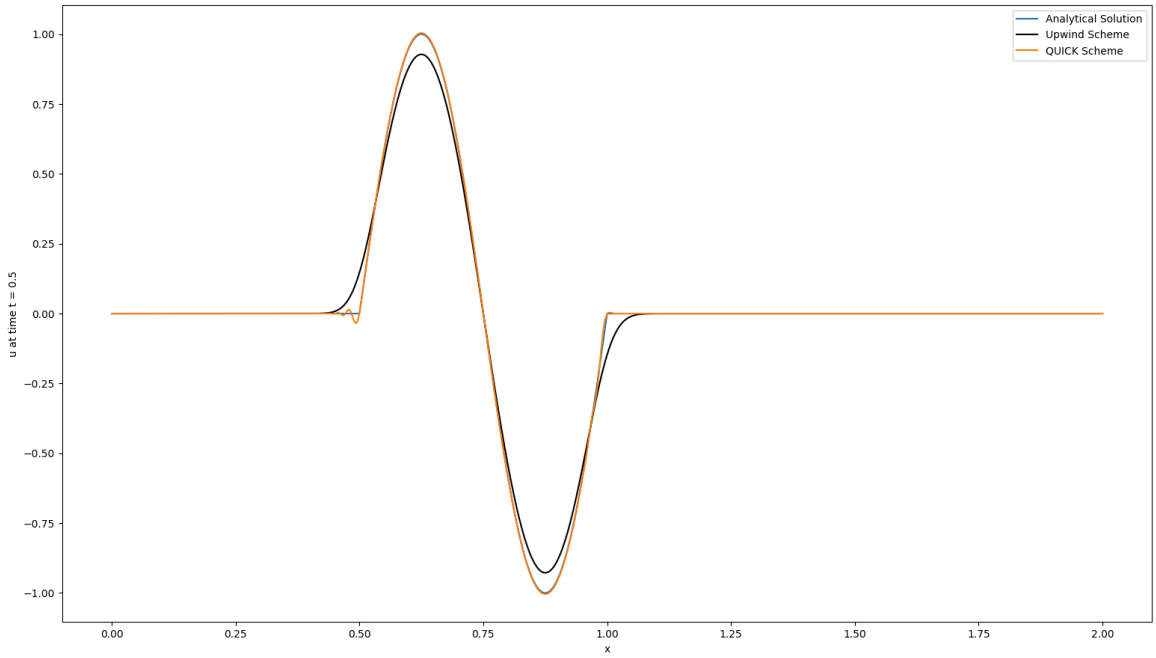
```python
def analytical_sol(x, t):
    ans = np.zeros(len(x))
    ans[int(c*t/dx) :int((0.5 + c*t)/dx)] = np.sin(4 * math.pi * x[:int(0.5/dx)])
    return ans

def upwind_scheme(u):
    u_new = u.copy()
    for i in range(1, len(u) - 1):
        u_new[i] = u[i] - c * (dt / dx) * (u[i] - u[i-1])
    return u_new

def QUICK_scheme(u):
    u_new = u.copy()
    # Boundary condition using upwind scheme
    u_new[1] = u[1] - c * (dt / dx) * (u[1] - u[0])
    for i in range(2, len(u) - 1):
        u_new[i] = u[i] - c * (dt / dx) * ((3/8) * u[i] - (7/8) * u[i-1] + (1/8) * u[i-2] + (3/8) * u[i+1])
    return u_new
```

Plots @ T = 0

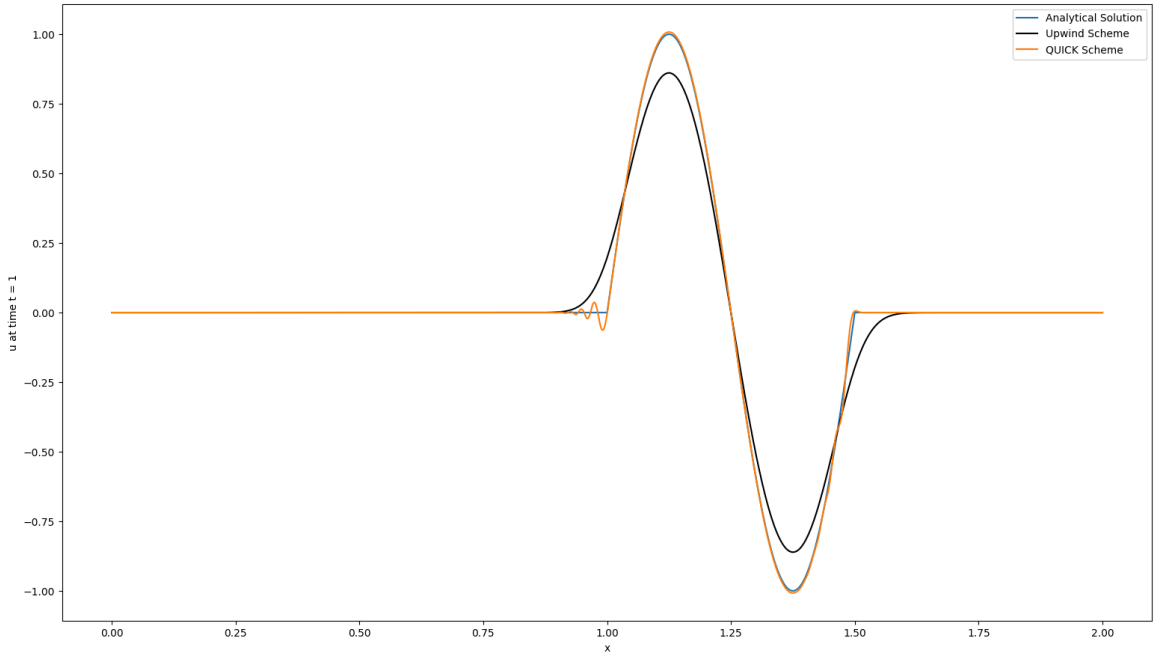


All these plots overlap each other at the start. All of these plots are equal to u0.

T = 0.5 :


Analytical Solution
Upwind Scheme
QUICK Scheme
u at time t = 0.5
x

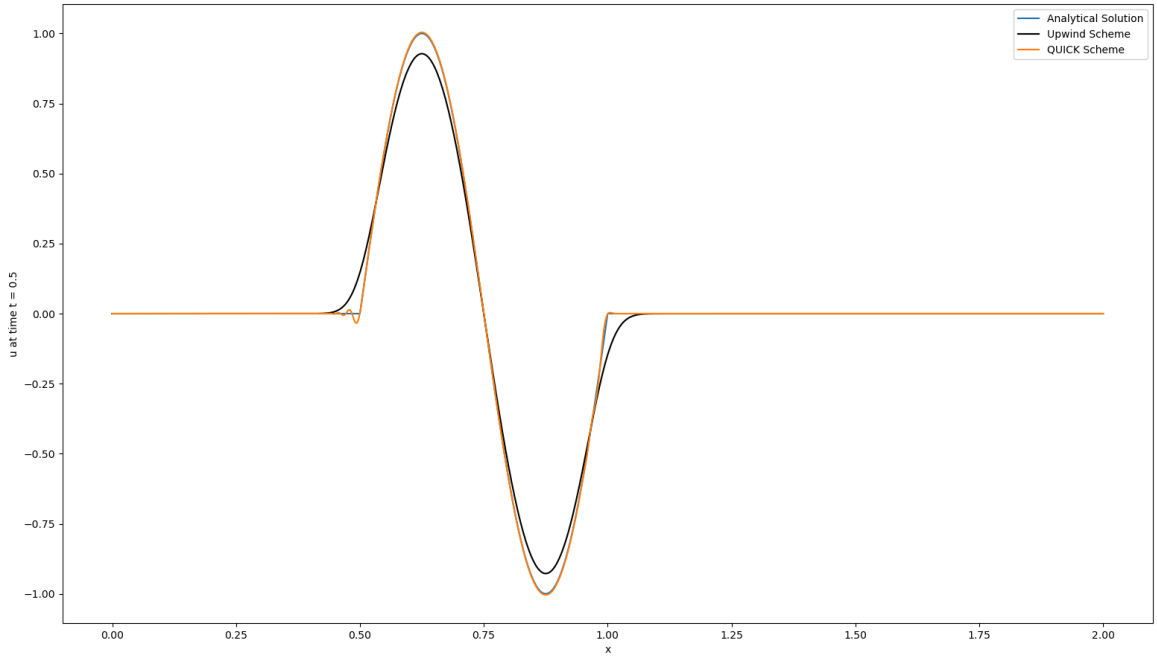
T = 1 :


Analytical Solution
Upwind Scheme
QUICK Scheme
u at time t = 1
x

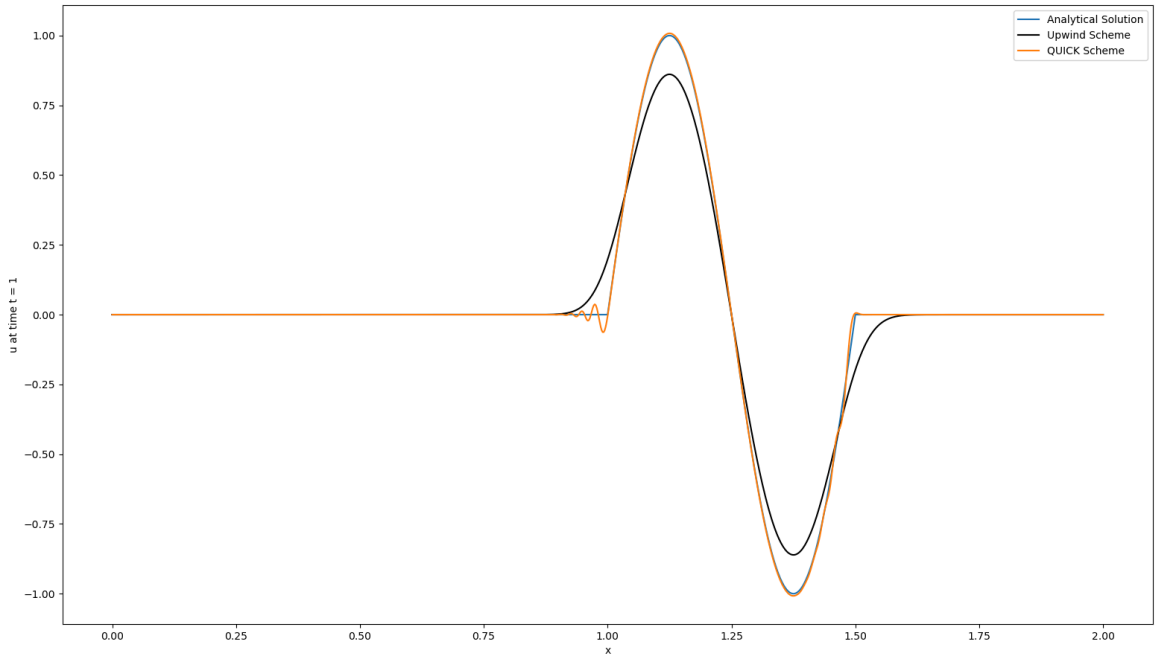
1 (b)

```python
def upwind_scheme(u,left):
    u_new = u.copy()
    if rank == 0:
        u_new[0] = 0
    if rank > 0:
        u_new[0] = u[0] - c * (dt / dx) * (u[0] - left[0])
    for i in range(1, len(u)):
        u_new[i] = u[i] - c * (dt / dx) * (u[i] - u[i-1])
    return u_new


def QUICK_scheme(u,left, right):
    u_new = u.copy()
    if rank == 0:
        u_new[0] = 0
        u_new[1] = u[1] - c * (dt / dx) * (u[1] - u[0])
    else:
        u_new[0] = u[0] - c * (dt / dx) * ((3/8) * u[0] - (7/8) * left[-1] + (1/8) * left[-2] + (3/8) * u[1])
        u_new[1] = u[1] - c * (dt / dx) * ((3/8) * u[1] - (7/8) * u[0] + (1/8) * left[-1] + (3/8) * u[2])
    if rank != size-1:
        u_new[-1] = u[-1] - c * (dt / dx) * ((3/8) * u[-1] - (7/8) * u[-2] + (1/8) * u[-3] + (3/8) * right[0])

    for i in range(2, len(u) - 1):
        u_new[i] = u[i] - c * (dt / dx) * ((3/8) * u[i] - (7/8) * u[i-1] + (1/8) * u[i-2] + (3/8) * u[i+1])
    return u_new
```

Plot @ T = 0.5 :



T = 1 :



1(c)

The quick scheme follows the actual solution more accurately, whereas the wave in the upwind scheme tends to increase in width with time.

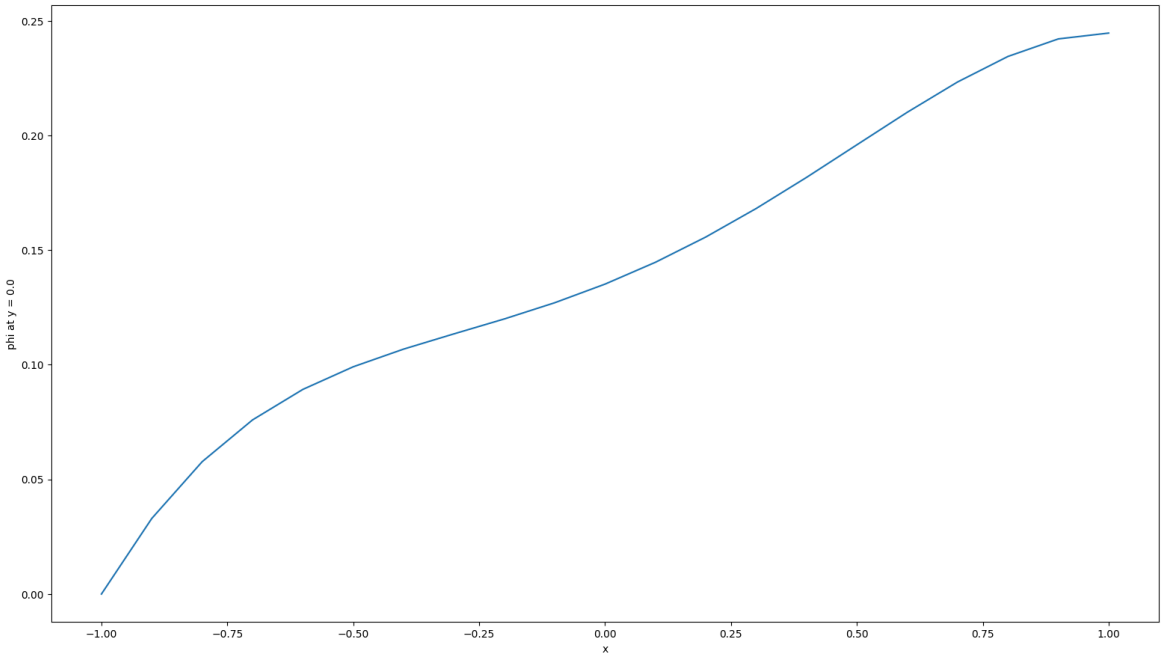
There are some noises accumulated near the boundary of the wave in the quick scheme.

2(a) Jacobi Iteration :

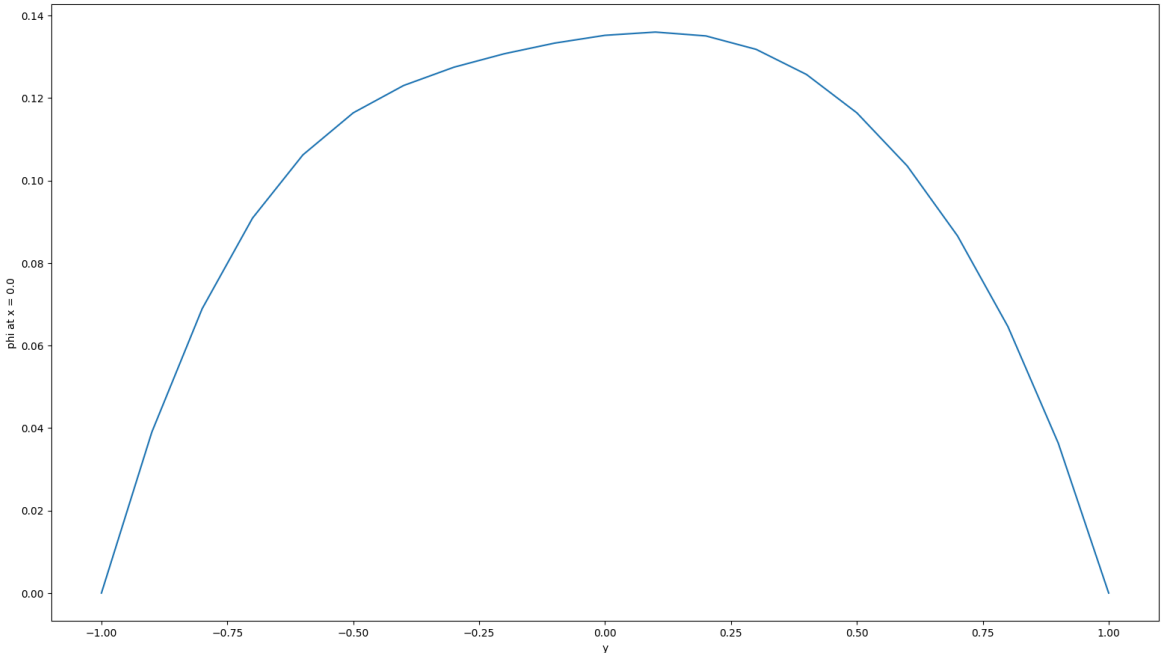
```python
# Function to calculate the value of phi at each point
def jacobi_update(phi, q):
    phi_new = phi.copy()
    change = 0
    for i in range(1, len(phi) - 1):
        for j in range(1, len(phi[0]) - 1):
            phi_new[i, j] = 0.25 * (phi[i+1, j] + phi[i-1, j] + phi[i, j+1] + phi[i, j-1] + (dx**2)*q[i, j])
            change = max(change, abs(phi_new[i, j] - phi[i, j]))
    for j in range(len(phi[-1])):
        phi_new[-1,j] = (4*phi_new[-2,j] - phi_new[-3,j])/3
        change = max(change, abs(phi_new[-1,j] - phi[-1,j]))

    return phi_new, change
```

The code took **387** iterations in total to converge for (dx = dy = 0.1).

Phi at y = 0.0 wrt x (dx = dy = 0.1)



Phi at x = 0.0 wrt y (dx = dy = 0.1)



2(b) Jacobi Iteration parallel : (column-wise block decomposition)

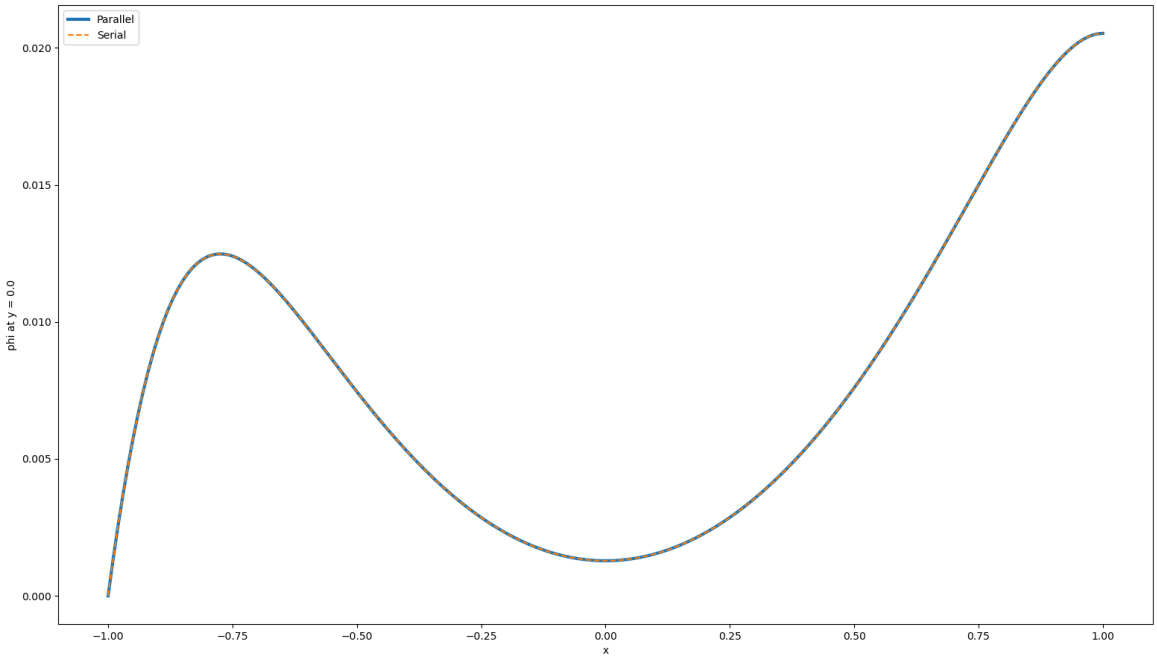
```
# Function to calculate the value of phi at each point
def jacobi_update(phi, left, right, q):
    phi_new = phi.copy()
    change = 0

    for i in range(1, len(phi) - 1):
        for j in range(1, len(phi[0]) - 1):
            phi_new[i, j] = 0.25 * (phi[i+1, j] + phi[i-1, j] + phi[i, j+1] + phi[i, j-1] + (dx**2)*q[i, j])
            change = max(change, abs(phi_new[i, j] - phi[i, j]))

    if rank == size-1:
        for j in range(len(phi[-1])):
            phi_new[-1,j] = (4*phi_new[-2,j] - phi_new[-3,j])/3
            change = max(change, abs(phi_new[-1, j] - phi[-1, j]))

    else :
        for j in range(1, len(phi[-1])-1):
            phi_new[-1, j] = 0.25 * (right[j] + phi[-2, j] + phi[-1, j+1] + phi[-1, j-1] + (dx**2)*q[-1, j])
            change = max(change, abs(phi_new[-1, j] - phi[-1, j]))

    if rank != 0:
        for j in range(1, len(phi[0])-1):
            phi_new[0, j] = 0.25 * (phi[1, j] + left[j] + phi[0, j+1] + phi[0, j-1] + (dx**2)*q[0, j])
            change = max(change, abs(phi_new[0, j] - phi[0, j]))


    return phi_new, change
```

```
while True:
    if rank > 0:
        comm.send(phi[0, :], dest=rank-1, tag=1)
    if rank < size-1:
        right = comm.recv(source=rank+1, tag=1)
    if rank < size-1:
        comm.send(phi[-1, :], dest=rank+1, tag=2)
    if rank > 0:
        left = comm.recv(source=rank-1, tag=2)
    comm.barrier()

    phi, change = jacobi_update(phi, left, right, q)

    change = comm.allreduce(change, op=MPI.MAX)
    comm.Barrier()
    iterations += 1

    if change < tolerance:
        print(rank, change, iterations)
        break
```

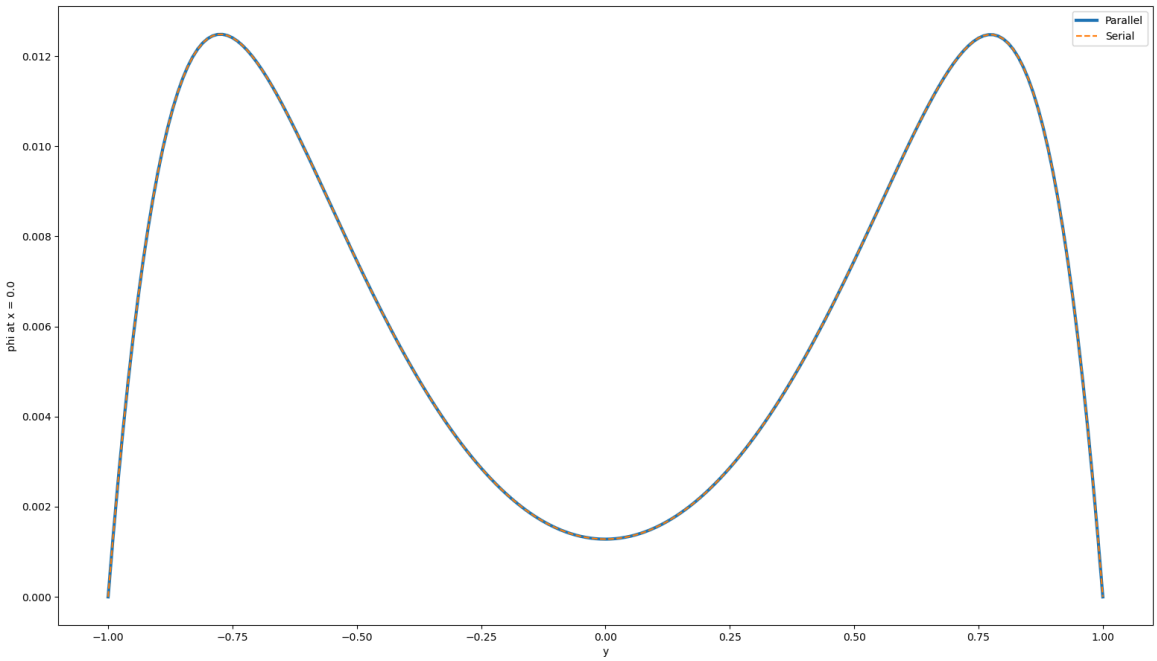
The code took **1013** iterations in total to converge for (dx = dy = 0.01).

Plots :

Phi at y = 0.0 wrt x



Phi at x = 0.0 wrt y



In both the plots, serial and parallel implementations of jacobi coincide with each other.

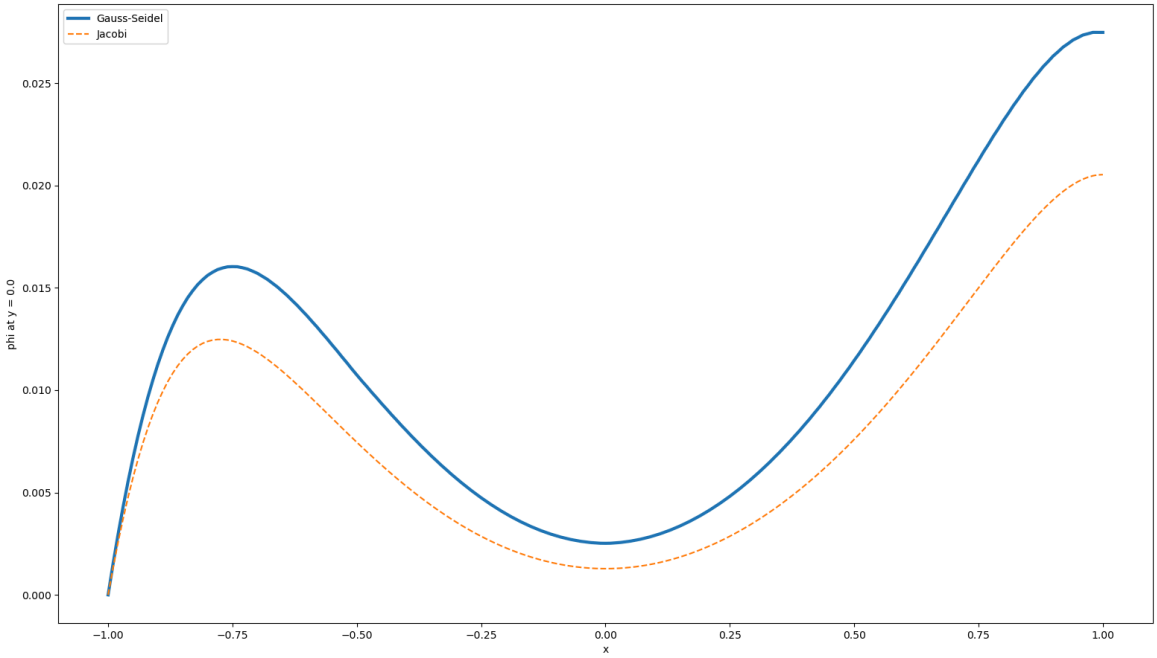
2(c) Gauss-Seidel red-black coloring approach :

```
# Function to calculate the value of phi at each point
def gs_update(phi, left, right, q, alpha):
    change = 0
    for i in range(1, len(phi) - 1):
        for j in range(1, len(phi[0]) - 1):
            if (i+j)%2 == alpha:
                continue
            x = phi[i,j]
            phi[i, j] = 0.25 * (phi[i+1, j] + phi[i-1, j] + phi[i, j+1] + phi[i, j-1] + (dx**2)*q[i, j])
            change = max(change, abs(phi [i, j] - x))

    if rank == size-1:
        for j in range(len(phi[-1])):
            if (j + len(phi) - 1 )%2 == alpha:
                continue
            x = phi[-1, j]
            phi[-1,j] = (4*phi[-2,j] - phi[-3,j])/3
            change = max(change, abs(phi[-1, j] - x))

    else :
        for j in range(1, len(phi[-1])-1):
            if (j + len(phi) - 1 )%2 == alpha:
                continue
            x = phi[-1, j]
            phi[-1, j] = 0.25 * (right[j] + phi[-2, j] + phi[-1, j+1] + phi[-1, j-1] + (dx**2)*q[-1, j])
            change = max(change, abs(phi[-1, j] - x))

    if rank != 0:
        for j in range(1, len(phi[0])-1):
            if (j + len(phi) - 1 )%2 == alpha:
                continue
            x = phi[0, j]
            phi[0, j] = 0.25 * (phi[1, j] + left[j] + phi[0, j+1] + phi[0, j-1] + (dx**2)*q[0, j])
            change = max(change, abs(phi[0, j] - x))
    return change
```

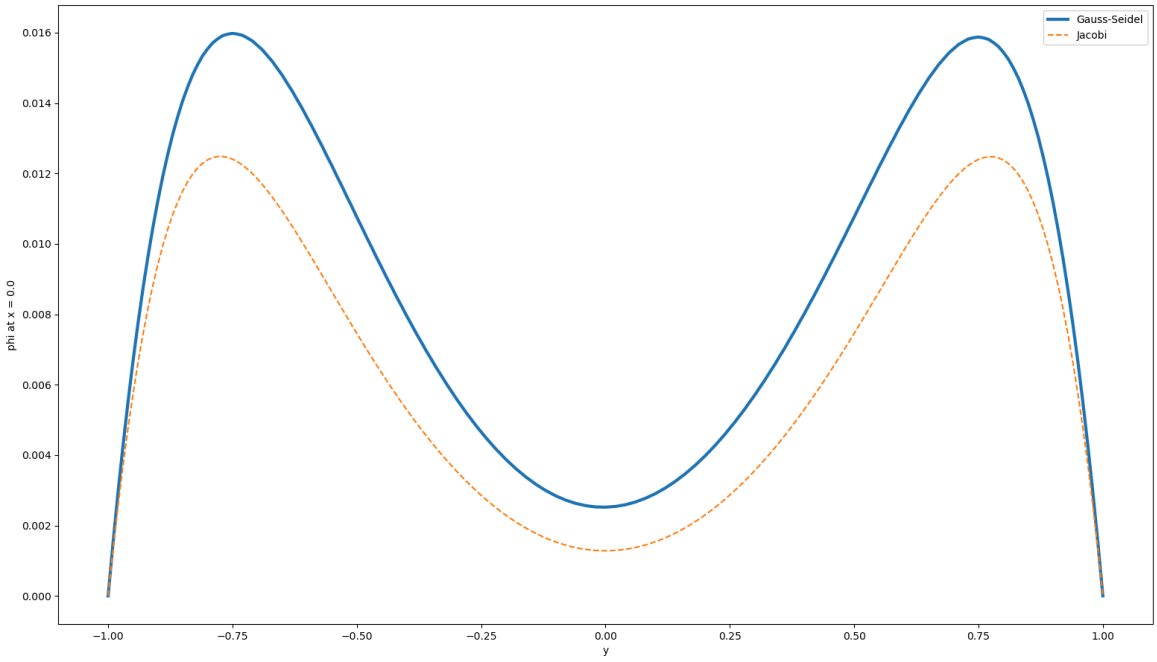
Gauss Seidel approach took 720 iterations (1440 half iterations) whereas jacobi iteration took 1013 iterations for the same grid size dx = dy = 0.01 .

Plots :

Phi at y = 0.0 wrt x (gauss seidel vs jacobi)



Phi at x = 0.0 wrt y (gauss seidel vs jacobi)



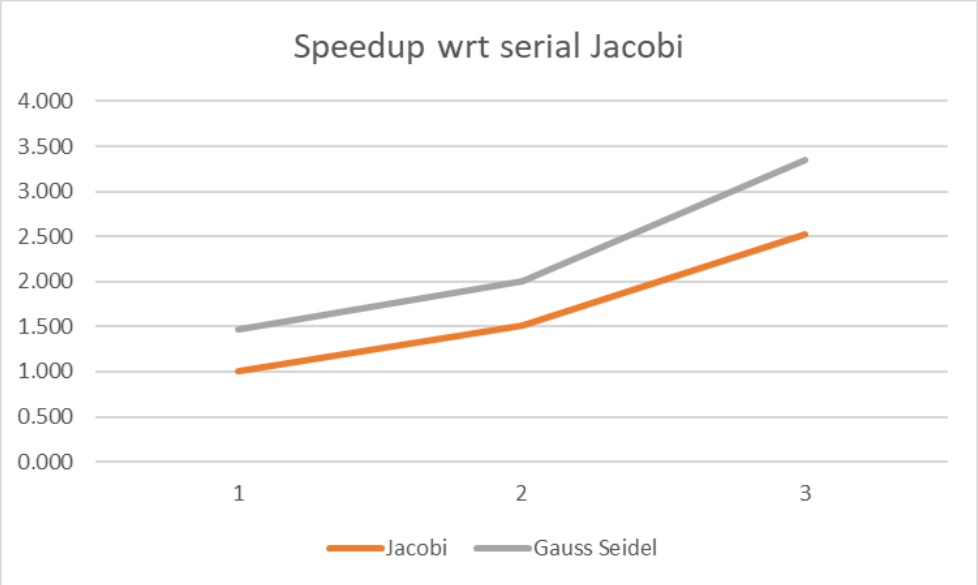
2(d) Time taken :

dx= dy =0.05

Threads	Jacobi	Gauss Seidel
1	3.902	3.093
2	2.510	2.338
4	2.159	1.680

dx= dy =0.01

Threads	Jacobi	Gauss Seidel
1	129.835	88.457
2	85.853	64.911
4	51.303	38.789



Gauss Seidel looks promising and can perform better with problems of higher degree. ** I was not able to plot more points as my laptop has only 4 threads.