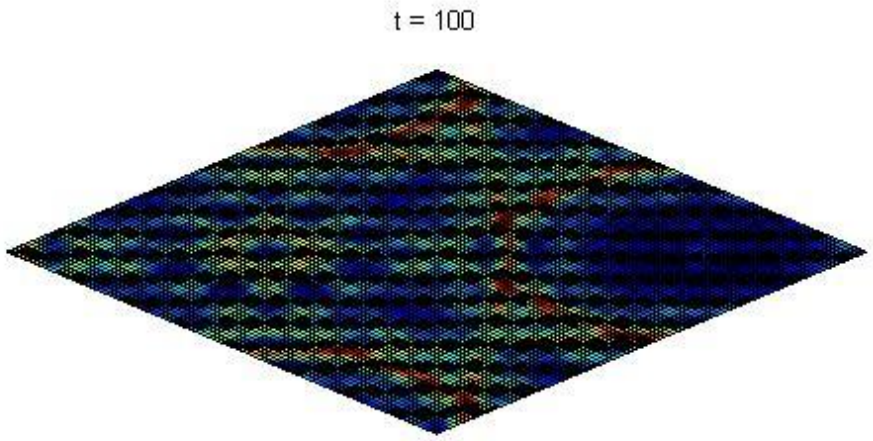
SHALLOW WATER EQUATION ANALYSIS

---By applying Forth-Order Runge –Kutta and Forth-Order Finite Difference Method



Sida Cheng 571224

HASHIM RAZA 651291

Introduction of Shallow Water Equation:

Shallow Water Equations are known as a system of Partial Differential Equations to describe the surface wave of water where the height of the moving liquid is much smaller compare to the wave length of the moving water.

Description of Shallow Water Equation:

$$\frac{dv}{dt} + v \cdot \nabla v = -g\nabla h$$

$$v = [v_x \quad v_y]$$

$$\nabla v = \begin{bmatrix} \frac{dv_x}{dx} & \frac{dv_x}{dy} \\ \frac{dv_y}{dx} & \frac{dv_y}{dy} \end{bmatrix}$$

$$\nabla h = \begin{bmatrix} \frac{dh}{dx} & \frac{dh}{dy} \end{bmatrix}$$

$$\frac{dv}{dt} = \begin{bmatrix} \frac{dv_x}{dt} & \frac{dv_y}{dt} \end{bmatrix}$$

$$v \cdot \nabla v^T = [v_x \quad v_y] \cdot \begin{bmatrix} \frac{dv_x}{dx} & \frac{dv_y}{dx} \\ \frac{dv_x}{dy} & \frac{dv_y}{dy} \end{bmatrix} = \begin{bmatrix} v_x \frac{dv_x}{dx} + v_y \frac{dv_x}{dy} & v_x \frac{dv_y}{dx} + v_y \frac{dv_y}{dy} \end{bmatrix}$$

$$\begin{bmatrix} \frac{dv_x}{dt} & \frac{dv_y}{dt} \end{bmatrix} = -\begin{bmatrix} v_x \frac{dv_x}{dx} + v_y \frac{dv_x}{dy} & v_x \frac{dv_y}{dx} + v_y \frac{dv_y}{dy} \end{bmatrix} - g \begin{bmatrix} \frac{dh}{dx} & \frac{dh}{dy} \end{bmatrix}$$

$$(\mathbf{1}) \qquad \boldsymbol{\frac{dv_x}{dt} = -v_x \frac{dv_x}{dx} - v_y \frac{dv_x}{dy} - g \frac{dh}{dx}}$$

$$(\mathbf{2}) \qquad \boldsymbol{\frac{dv_y}{dt} = -v_x \frac{dv_y}{dx} - v_y \frac{dv_y}{dy} - g \frac{dh}{dy}}$$

$$\frac{dh}{dt} + \nabla \cdot vh = 0$$

$$\nabla \cdot vh = \frac{d(v_x h)}{dx} + \frac{d(v_y h)}{dy}$$

$$(\mathbf{3}) \qquad \boldsymbol{\frac{dh}{dt} = -\frac{d(v_x h)}{dx} - \frac{d(v_y h)}{dy}}$$

Where vx and vy are two components of velocity V in x and y direction and h is the depth of the shallow water.

Initial Conditions:

The range of x and y are both 100 and the program will be implemented to be running from time=0 to time=100s.

```
x_min=0.00;
```

```
x_max=100.00;
y_min=0.00;
y_max=100.00;
t_min=0.00;
t_max=100.00;
g=9.81;
delta_x=1;
delta_y=1;
delta_t=0.1;
x=x_min:delta_x:x_max;
y=y_min:delta_y:y_max;
t=t_min:delta_t:t_max;
N_x=length(x);
N_y=length(y);
N_t=length(t);


%allocate arrays
h=zeros(N_x,N_y);
vx=zeros(N_x,N_y);
vy=zeros(N_x,N_y);
```

The initial condition of vx and vy are 0 for all of the point at time t=0. The initial height condition h at different point are calculated based on its location in the x-y coordinates.

```
%set initial condition

for i=1:N_x
    for j=1:N_y
        h (i,j)=1+0.5*exp((-1/25)*(((x(i)-30).^2)+((y(j)-30).^2)));
    end
end

vx(:,:)=0;
vy(:,:)=0;
```

Forth order central finite difference derivation:

First use the finite difference formula:

$$\frac{d^n\emptyset}{dx^n}|x_i = \frac{1}{\Delta x^n}\left(\sum_{m=1}^{M} a_{-m}\emptyset_{i-m} + a_0\emptyset_i + \sum_{m=1}^{M} a_{+m}\emptyset_{i+m}\right)$$

For the first derivative:

$$\frac{d\emptyset}{dx}|x_i = \frac{1}{\Delta x}\left(\sum_{m=1}^{M} a_{-m}\emptyset_{i-m} + a_0\emptyset_i + \sum_{m=1}^{M} a_{+m}\emptyset_{i+m}\right)$$

And if M=2:

$$\frac{d\emptyset}{dx}|x_i = \frac{1}{\Delta x}(a_{-2}\emptyset_{i-2} + a_{-1}\emptyset_{i-1} + a_0\emptyset_i + a_{+1}\emptyset_{i+1} + a_{+2}\emptyset_{i+2})$$

After apply Tyler series:

$$\emptyset_{i-1} = \emptyset(x_i - \Delta x) = \emptyset_i - (\Delta x)\frac{d\emptyset}{dx} + \frac{(\Delta x)^2}{2!}\frac{d^2\emptyset}{dx^2} - \frac{(\Delta x)^3}{3!}\frac{d^3\emptyset}{dx^3} + \frac{(\Delta x)^4}{4!}\frac{d^4\emptyset}{dx^4} \dots \dots$$

$$\emptyset_{i+1} = \emptyset(x_i + \Delta x) = \emptyset_i + (\Delta x)\frac{d\emptyset}{dx} + \frac{(\Delta x)^2}{2!}\frac{d^2\emptyset}{dx^2} + \frac{(\Delta x)^3}{3!}\frac{d^3\emptyset}{dx^3} + \frac{(\Delta x)^4}{4!}\frac{d^4\emptyset}{dx^4} \dots \dots$$

$$\emptyset_{i-2} = \emptyset(x_i - 2\Delta x) = \emptyset_i - (2\Delta x)\frac{d\emptyset}{dx} + \frac{(2\Delta x)^2}{2!}\frac{d^2\emptyset}{dx^2} - \frac{(2\Delta x)^3}{3!}\frac{d^3\emptyset}{dx^3} + \frac{(2\Delta x)^4}{4!}\frac{d^4\emptyset}{dx^4} \dots \dots$$

$$\emptyset_{i+2} = \emptyset(x_i + 2\Delta x) = \emptyset_i - (2\Delta x)\frac{d\emptyset}{dx} + \frac{(2\Delta x)^2}{2!}\frac{d^2\emptyset}{dx^2} - \frac{(2\Delta x)^3}{3!}\frac{d^3\emptyset}{dx^3} + \frac{(2\Delta x)^4}{4!}\frac{d^4\emptyset}{dx^4} \dots \dots$$

Re-arrange the equations:

$$\frac{d\emptyset}{dx}|x_i = \frac{1}{\Delta x}\begin{bmatrix} (a_{-2} + a_{-1} + a_0 + a_{+1} + a_{+2})\emptyset_i \\ (a_{-2}(-2) + a_{-1}(-1) + a_0(0) + a_{+1}(1) + a_{+2}(2))\Delta x\frac{d\emptyset}{dx} \\ (a_{-2}(\frac{(-2)^2}{2!}) + a_{-1}(\frac{(-1)^2}{2!}) + a_0(0) + a_{+1}(\frac{(1)^2}{2!}) + a_{+2}(\frac{(2)^2}{2!}))\Delta x\frac{d^2\emptyset}{dx^2} \\ (a_{-2}(\frac{(-2)^3}{3!}) + a_{-1}(\frac{(-1)^3}{3!}) + a_0(0) + a_{+1}(\frac{(1)^3}{3!}) + a_{+2}(\frac{(2)^3}{3!}))\Delta x\frac{d^3\emptyset}{dx^3} \\ (a_{-2}(\frac{(-2)^4}{4!}) + a_{-1}(\frac{(-1)^4}{4!}) + a_0(0) + a_{+1}(\frac{(1)^4}{4!}) + a_{+2}(\frac{(2)^4}{4!}))\Delta x\frac{d^4\emptyset}{dx^4} \end{bmatrix}$$

We will get a matrix based on the re-arrangement of the equation make $\Delta x = 1$

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ -2 & -1 & 0 & 1 & 2 \\ 2 & \frac{1}{2} & 0 & \frac{1}{2} & 2 \\ -\frac{4}{3} & -\frac{1}{6} & 0 & \frac{1}{6} & \frac{4}{3} \\ \frac{2}{3} & \frac{1}{24} & 0 & \frac{1}{24} & \frac{2}{3} \end{bmatrix}\begin{bmatrix} a_{-2} \\ a_{-1} \\ a_0 \\ a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Solve for the matrix by MATLAB:

$$\begin{bmatrix} a_{-2} \\ a_{-1} \\ a_0 \\ a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} \frac{1}{12} \\ -\frac{2}{3} \\ 0 \\ \frac{2}{3} \\ -\frac{1}{12} \end{bmatrix}$$

So the forth order central difference equation is:

$$\boldsymbol{\frac{d\emptyset}{dx}|x_i = \frac{1}{12\Delta x}(\emptyset_{i-2} - 8\emptyset_{i-1} + 8\emptyset_{i+1} - \emptyset_{i+2})}$$

Stability analysis of the forth order RH method:

From the equation:

$$\emptyset^{l+1} = \emptyset^{l} + \frac{\Delta t}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

Where:

$$K_1 = f(\emptyset^l, t^l)$$

$$K_2 = f\left(\emptyset^l + \frac{\Delta t}{2}K_1, t^l + \frac{\Delta t}{2}\right)$$

$$K_3 = f\left(\emptyset^l + \frac{\Delta t}{2}K_2, t^l + \frac{\Delta t}{2}\right)$$

$$K_4 = f\left(\emptyset^l + \Delta t K_3, t^l + \frac{\Delta t}{2}\right)$$

If make :

$$K_1 = f(\emptyset^l, t^l) = \lambda\emptyset^l$$

We will get:

$$k_1 = \lambda\emptyset^l$$

$$k_2 = \lambda(\emptyset^l + \frac{\Delta t}{2}(\lambda\emptyset^l)) = \lambda\emptyset^l + \frac{1}{2}\Delta t\lambda^2\emptyset^l$$

$$k_3 = \lambda(\emptyset^l + \frac{\Delta t}{2}(\lambda\emptyset^l + \frac{1}{2}\Delta t\lambda^2\emptyset^l)) = \lambda\emptyset^l + \frac{1}{2}\Delta t\lambda^2\emptyset^l + \frac{1}{4}\Delta t^2\lambda^3\emptyset^l$$

$$k_4 = \lambda(\emptyset^l + \Delta t(\lambda\emptyset^l + \frac{1}{2}\Delta t\lambda^2\emptyset^l + \frac{1}{4}\Delta t^2\lambda^3\emptyset^l)) = \lambda\emptyset^l + \Delta t\lambda^2\emptyset^l + \frac{1}{2}\Delta t^2\lambda^3\emptyset^l + \frac{1}{4}\Delta t^3\lambda^4\emptyset^l$$

So

$$\emptyset^{l+1} = \emptyset^{l} + \frac{\Delta t}{6}(k_1 + 2k_2 + 2k_3 + k_4) = \emptyset^l(1 + \lambda\Delta t + \frac{1}{2}\Delta t^2\lambda^2 + \frac{1}{6}\Delta t^3\lambda^3 + \frac{1}{24}\Delta t^4\lambda^4)$$

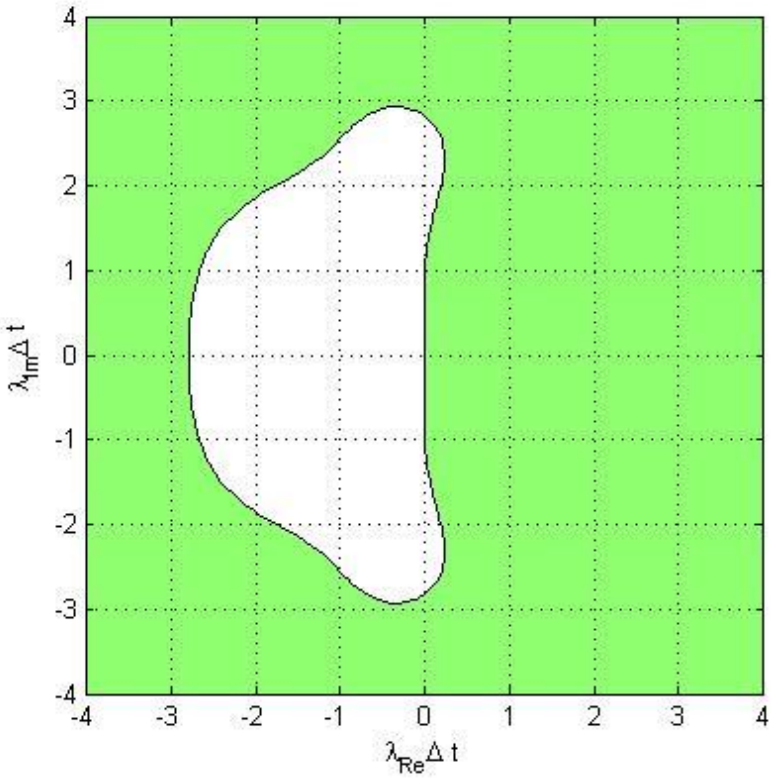
Get the equation below from previous steps:

$$\textbf{Sigma=}1 + \lambda\Delta t + \tfrac{1}{2}\Delta t^2\lambda^2 + \tfrac{1}{6}\Delta t^3\lambda^3 + \tfrac{1}{24}\Delta t^4\lambda^4$$

The stability region could be plotted by MATLAB based on the code:

```
function RTstability()
    clear all; close all; clc;
    [X, Y]          = meshgrid(-4:0.1:4, -4:0.1:4);
    Z               = X + i*Y;
```

```
sigma          = abs(1 + Z + (Z.^2)/2 + (Z.^3)/6 + (Z.^4)/24);
figure('WindowStyle', 'docked');
contourf(X, Y, sigma, [1 1],'-k');
hold on;
axis('equal', [-4 4 -4 4]);
grid on;
xlabel('\lambda_{Re}\Delta t');
ylabel('\lambda_{Im}\Delta t');
```



The stable region is the white region inside the green unstable region

Error analysis of forth order RH method:

If λ only has the imaginary component for our analysis:

Sigma=$1+\lambda\ \mathrm{img}\Delta t+\frac{1}{2}\Delta t^2\lambda\ \mathrm{img}^2+\frac{1}{6}\Delta t^3\lambda\ \mathrm{img}^3+\frac{1}{24}\Delta t^4\lambda\ \mathrm{img}^4$

After re arrange with $\lambda\ \mathrm{img}=i\lambda$

If **sigma=Z$e^{i\theta}$**

$$\boldsymbol{sigma}=\left(1-\frac{1}{2}\Delta t^2\lambda^2+\frac{1}{24}\Delta t^4\lambda^4\right)+i\left(\lambda\Delta t-\frac{1}{6}\Delta t^3\lambda^3\right)$$

The amplitude error Z could be calculated as:

$$Z = \sqrt{\left(1 - \frac{1}{2}\Delta t^2\lambda^2 + \frac{1}{24}\Delta t^4\lambda^4\right)^2 + \left(\lambda\Delta t - \frac{1}{6}\Delta t^3\lambda^3\right)^2}$$

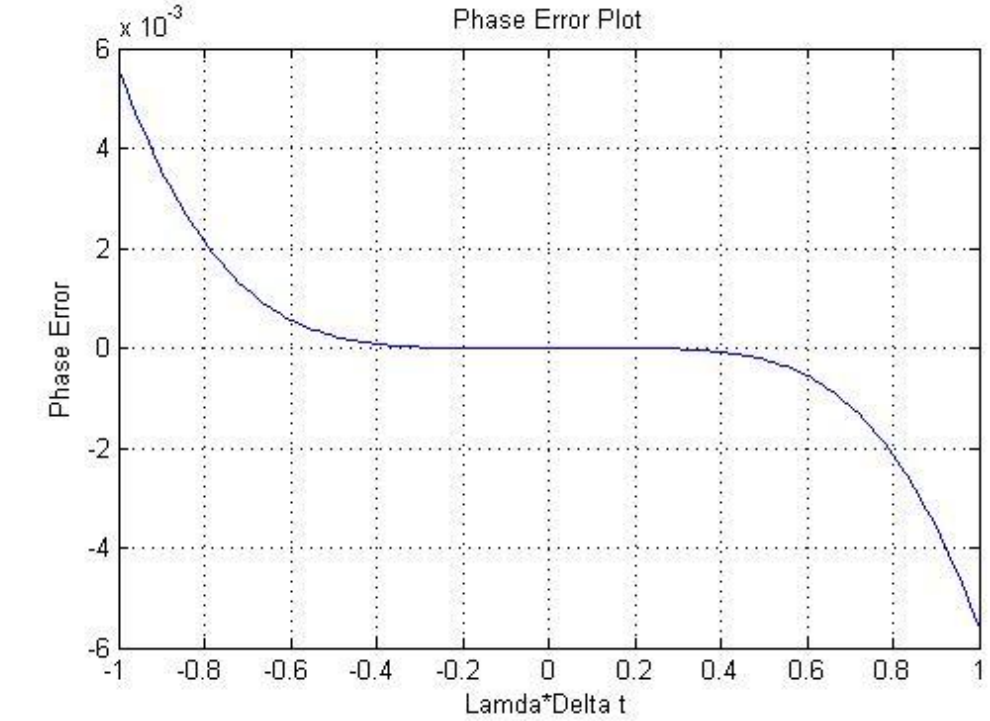
And the phase error θ is calculated as:

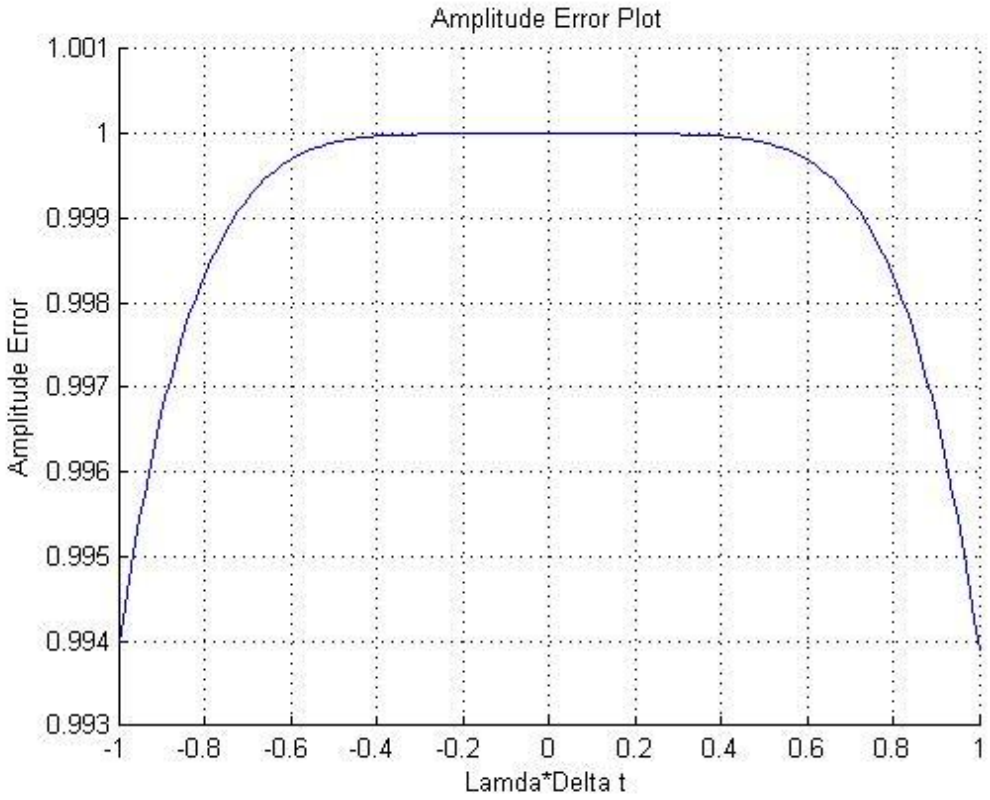
$$\theta = tan^{-1}\left(\frac{\lambda\Delta t - \frac{1}{6}\Delta t^3\lambda^3}{1 - \frac{1}{2}\Delta t^2\lambda^2 + \frac{1}{24}\Delta t^4\lambda^4}\right)$$

And the plot of these two errors could be done in the matlab:

```
function Erroranalysis()
clear all; close all; clc;
x=-1:0.1:1;
Amp_error=sqrt((1-(x.^2)/2+(x.^4)/24).^2+(x-(x.^3)/6).^2);
Phase_error=atan((x-(x.^3)/6)./(1-(x.^2)/2+(x.^4)/24))-x;
hold on
figure(1)
plot(x,Amp_error);
xlabel('Lamda*Delta t');
ylabel('Amplitude Error');
grid on
title('Amplitude Error Plot');
figure(2)
plot(x,Phase_error);
xlabel('Lamda*Delta t');
ylabel('Phase Error');
grid on
title('Phase Error Plot');
```

The results are plotted as:





Matlab Code implementation:

$$\frac{dv_x}{dt} = -v_x \frac{dv_x}{dx} - v_y \frac{dv_x}{dy} - g \frac{dh}{dx}$$

$$\frac{dv_x}{dt} = -\frac{v_x}{12\Delta x}\left(v_{x_{i-2,j}} - 8v_{x_{i-1,j}} + 8v_{x_{i+1,j}} - v_{x_{i+2,j}}\right) - \frac{v_y}{12\Delta y}\left(v_{x_{i,j-2}} - 8v_{x_{i,j-1}} + 8v_{x_{i,j+1}} - v_{x_{i,j+2}}\right) - \frac{g}{12\Delta x}\left(h_{i-2,j} - 8h_{i-1,j} + 8h_{i+1,j} - h_{i+2,j}\right)$$

$$\frac{dv_y}{dt} = -v_x\frac{dv_y}{dx} - v_y\frac{dv_y}{dy} - g\frac{dh}{dy}$$

$$\frac{dv_y}{dt} = -\frac{v_x}{12\Delta x}\left(v_{y_{i-2,j}} - 8v_{y_{i-1,j}} + 8v_{y_{i+1,j}} - v_{y_{i+2,j}}\right) - \frac{v_y}{12\Delta y}\left(v_{y_{i,j-2}} - 8v_{y_{i,j-1}} + 8v_{y_{i,j+1}} - v_{y_{i,j+2}}\right) - \frac{g}{12\Delta y}\left(h_{i,j-2} - 8h_{i,j-1} + 8h_{i,j+1} - h_{i,j+2}\right)$$

$$\frac{dh}{dt} = -\frac{d(v_x h)}{dx} - \frac{d(v_y h)}{dy}$$

$$\frac{dh}{dt} = -\frac{1}{12\Delta x}\left(v_{x_{i-2,j}}h_{i-2,j} - 8v_{x_{i-1,j}}h_{i-1,j} + 8v_{x_{i+1,j}}h_{i+1,j} - v_{x_{i+2,j}}h_{i+2,j}\right) - \frac{1}{12\Delta y}\left(v_{y_{i,j-2}}h_{i,j-2} - 8v_{y_{i,j-1}}h_{i,j-1} + 8v_{y_{i,j+1}}h_{i,j+1} - v_{y_{i,j+2}}h_{i,j+2}\right)$$

After apply the finite difference method to the three of the equations in the shallow water equations system, the boundary condition is chosen to be ***periodic***. That means every discrete point of vx, vy and h, their first derivate of time (t) could be calculated from their four neighbor points. For example $\frac{dv_x}{dx}$ could be calculated based on points $v_{x_{i-2,j}}\ v_{x_{i-1,j}}\ v_{x_{i+1,j}}\ v_{x_{i+2,j}}$. If ***i*** is 1 for this case, the four points can be chosen to be $v_{x_{N_x-1,j}}\ v_{x_{Nx,j}}\ v_{x_{2,j}}\ v_{3,j}$. Then the function used to calculate **kx, ky and kh** values which are going to be used in forth-order RH method are implemented by the three equations above with applying the **periodic** boundary condition.

```
function [kx,ky,kh]=f(vx,vy,h)

    global  delta_x delta_y N_x N_y g;
    kx=zeros(N_x,N_y);
    ky=zeros(N_x,N_y);
    kh=zeros(N_x,N_y);

    for i=1:N_x;
```

```
    x_a=i-2;
    x_b=i-1;
    x_c=i+1;
    x_d=i+2;
    if x_a==-1
        x_a=N_x-1;
    else if x_a==0
        x_a=N_x;
        end
    end
    if x_b==0
        x_b=N_x;
    end
    if x_c==N_x+1;
        x_c=1;
    end
    if x_d==N_x+2;
        x_d=2;
    else if x_d==N_x+1;
            x_d=1;
        end
    end
for j=1:N_y;

    y_e=j-2;
    y_f=j-1;
    y_g=j+1;
    y_h=j+2;

    if y_e==-1
        y_e=N_y-1;
    else if y_e==0
        y_e=N_y;
        end
    end
    if y_f==0
        y_f=N_y;
    end
    if y_g==N_y+1;
        y_g=1;
    end
    if y_h==N_y+2;
        y_h=2;
    else if y_h==N_y+1;
```

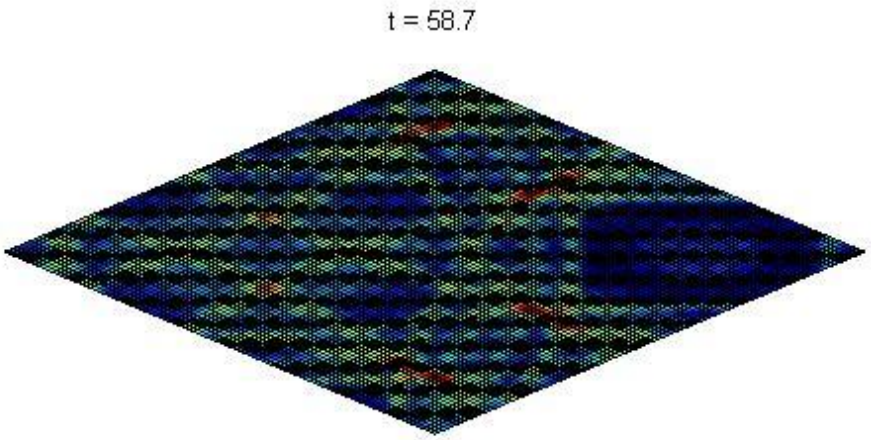
```
                y_h=1;
            end
        end

kx(i,j)=(((-1)*vx(i,j))/(12*delta_x))*(vx(x_a,j)-8*vx(x_b,j)+8*vx(x_c
,j)-vx(x_d,j))+(((-1)*vy(i,j))/(12*delta_y))*(vx(i,y_e)-8*vx(i,y_f)+8
*vx(i,y_g)-vx(i,y_h))+((-1)*g/(12*delta_x))*(h(x_a,j)-8*h(x_b,j)+8*h(
x_c,j)-h(x_d,j));

ky(i,j)=(((-1)*vx(i,j))/(12*delta_x))*(vy(x_a,j)-8*vy(x_b,j)+8*vy(x_c
,j)-vy(x_d,j))+(((-1)*vy(i,j))/(12*delta_y))*(vy(i,y_e)-8*vy(i,y_f)+8
*vy(i,y_g)-vy(i,y_h))+((-1)*g/(12*delta_y))*(h(i,y_e)-8*h(i,y_f)+8*h(
i,y_g)-h(i,y_h));

kh(i,j)=(-1/(12*delta_x))*(vx(x_a,j)*h(x_a,j)-8*vx(x_b,j)*h(x_b,j)+8*
vx(x_c,j)*h(x_c,j)-vx(x_d,j)*h(x_d,j))+(-1/(12*delta_y))*(vy(i,y_e)*h
(i,y_e)-8*vy(i,y_f)*h(i,y_f)+8*vy(i,y_g)*h(i,y_g)-vy(i,y_h)*h(i,y_h))
;
        end
    end
end
```

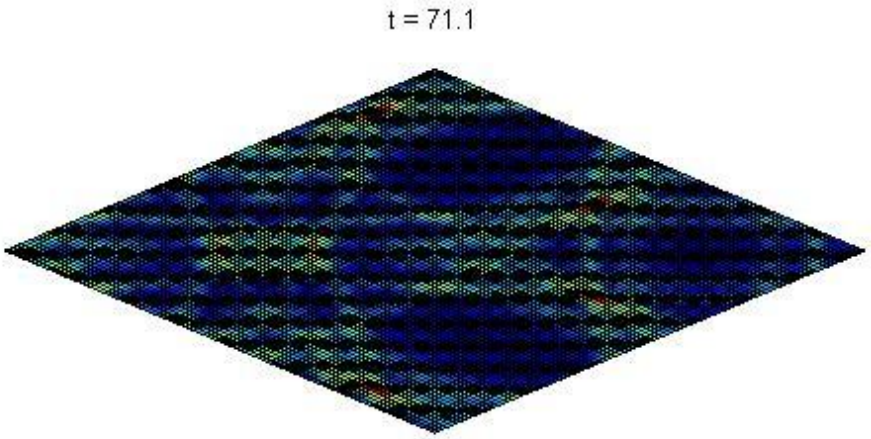
After developing the function to calculate k values for vx, vy and h. Forth-order Runge-Kutta method are applied to calculated next stage vx, vy and h values by solving the systems of Ordinary Differential Equations of $\frac{dv_x}{dt}$ $\frac{dv_y}{dt}$ and $\frac{dh}{dt}$. The function is implemented as below with time loop variable l:

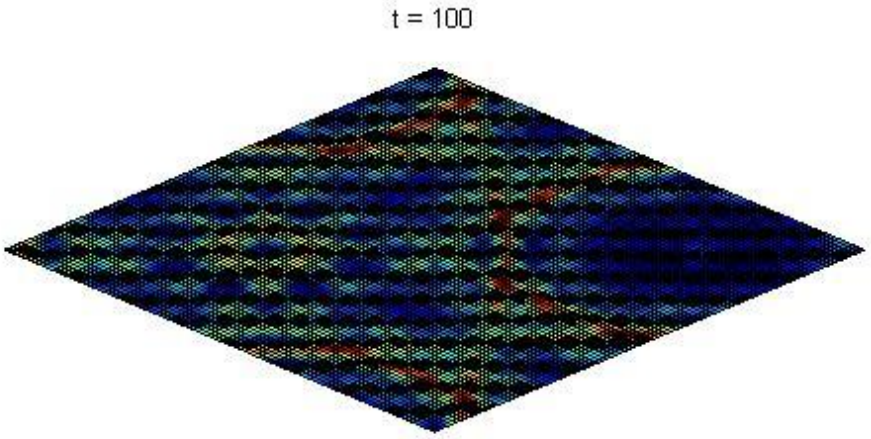
```
for l=1:N_t-1       //time marching loop

        [kx1,ky1,kh1]         = f(vx(:,:),vy(:,:),h(:,:));
        [kx2,ky2,kh2]         = f(vx(:,:) +
(delta_t/2)*kx1(:,:),vy(:,:)+(delta_t/2)*ky1(:,:),h(:,:)+(delta_t/2)*
kh1(:,:));
        [kx3,ky3,kh3]         = f(vx(:,:) +
(delta_t/2)*kx2(:,:),vy(:,:)+(delta_t/2)*ky2(:,:),h(:,:)+(delta_t/2)*
kh2(:,:));
        [kx4,ky4,kh4]         = f(vx(:,:) +
delta_t*kx3(:,:),vy(:,:)+delta_t*ky3(:,:),h(:,:)+delta_t*kh3(:,:));
        vx(:,:)  = vx(:,:) + delta_t  *(kx1(:,:)/6 + kx2(:,:)/3 +
kx3(:,:)/3 + kx4(:,:)/6);      //next stage vx is calculated
        vy(:,:)  = vy(:,:) + delta_t  *(ky1(:,:)/6 + ky2(:,:)/3 +
ky3(:,:)/3 + ky4(:,:)/6);    //next stage vy is calculated
        h(:,:)   = h(:,:) + delta_t  *(kh1(:,:)/6 + kh2(:,:)/3 + kh3(:,:)/3
+ kh4(:,:)/6);     //next stage h is calculated
```

```
    % Plot the solution
  set(Solution, 'ZData', h(:,:));
  title(['t = ' num2str(t(l+1))]);
  drawnow;

end
```

Result Plotting from MATLAB analysis at random times:







C++ program Implementation:

The idea of the C++ code implementation is same as the MATLAB code implementation except

some of the function for calculating Matrix Addition and Matrix Multiplication by a constant are written manually since there are tool boxes in the MATLAB program. The initial values and all the delta_x, delta_y and delta_t values are set to be the same in the MATLAB code in order to produce the same result which will be plotted by PARAVIEW. With consideration of paralleling the C++ code in the next stages in this analysis, the arrays (Matrices) are constructed contiguously in the memory in order to avoid changing the code for the next stages. The sample code of the array construction is:

```
// Allocate arrays
double** vx = new double* [N_x];
vx[0]=new double [N_x*N_y];
for( m=1, mm=N_y;m<N_x; m++,mm+=N_y)
{
    vx[m]=&vx[0][mm];
}
```

The function of calculating k values for vx vy and h is very similar to the MATLAB implementation just changed the position of each value in the matrices because arrays in MATLAB starts with position 1 but in C++ it starts with position 0. The function is also designed to return three matrices of k values of vx vy and h at the same time, therefore all the calculation are done in the original array by using pointers and a void function. The sample code of the f function is:

```
void    f(double** vx, double** vy, double** h, double** kx, double** ky, double** kh)
{
    int x_a, x_b,x_c,x_d,y_e,y_f,y_g,y_h;

    for(int i=0; i<N_x; i++)
    {
        x_a=i-2;
        x_b=i-1;
        x_c=i+1;
        x_d=i+2;

        if(x_a==-2)
        {
            x_a=N_x-2;
        }
        else if (x_a==-1)
        {
            x_a=N_x-1;
        }
        else
        {
            x_a=i-2 ;
        }
        if(x_b==-1)
        {
```

```
        x_b=N_x-1;
}
if(x_c==N_x)
{
        x_c=0;
}
if(x_d==N_x+1)
{
        x_d=1;
}
else if(x_d==N_x)
{
        x_d=0;
}
else
{
        x_d=i+2;
}

for (int j=0;j<N_y;j++)
{
        y_e=j-2;
        y_f=j-1;
        y_g=j+1;
        y_h=j+2;
        if(y_e==-2)
        {
                y_e=N_y-2;
        }
        else if (y_e==-1)
        {
                y_e=N_y-1;
        }
        if(y_f==-1)
        {
                y_f=N_y-1;
        }
        if(y_g==N_y)
        {
                y_g=0;
        }
        if(y_h==N_y+1)
        {
                y_h=1;
```

```
            }
            else if(y_h==N_y)
            {
                  y_h=0;
            }
            else
            {
                  y_h=j+2;
            }
```

```
kx[i][j]=(((-1)*vx[i][j])/(12*delta_x))*(vx[x_a][j]-8*vx[x_b][j]+8*vx[x_c][j]-vx[x_d][j])+(((-1)*vy[i][j])/(12*delta_y))*(vx[i][y_e]-8*vx[i][y_f]+8*vx[i][y_g]-vx[i][y_h])+((-1)*g/(12*delta_x))*(h[x_a][j]-8*h[x_b][j]+8*h[x_c][j]-h[x_d][j]);

ky[i][j]=(((-1)*vx[i][j])/(12*delta_x))*(vy[x_a][j]-8*vy[x_b][j]+8*vy[x_c][j]-vy[x_d][j])+(((-1)*vy[i][j])/(12*delta_y))*(vy[i][y_e]-8*vy[i][y_f]+8*vy[i][y_g]-vy[i][y_h])+((-1)*g/(12*delta_y))*(h[i][y_e]-8*h[i][y_f]+8*h[i][y_g]-h[i][y_h]);

kh[i][j]=(-1/(12*delta_x))*(vx[x_a][j]*h[x_a][j]-8*vx[x_b][j]*h[x_b][j]+8*vx[x_c][j]*h[x_c][j]-vx[x_d][j]*h[x_d][j])+(-1/(12*delta_y))*(vy[i][y_e]*h[i][y_e]-8*vy[i][y_f]*h[i][y_f]+8*vy[i][y_g]*h[i][y_g]-vy[i][y_h]*h[i][y_h]);
```

```
            }
      }


      return;
}
```

To perform the forth-order Runge-Kutta method, the matrix addition and matrix time by a number functions are designed. Because all of the matrices in our analysis is the same size of N_x and N_y. The functions of matrix addition and multiplication are implemented by double-for-loops and by using pointers. The code examples are:

```
void    matrixmult(double** k, double** m,    double c)
{
      for( int i=0;i<N_x;i++)
      {
            for (int j=0; j<N_y;j++)
            {
                  k[i][j]=m[i][j]*c;
            }
      }
```

```
    return    ;
}
void matrixadd(double** k,double** h, double** m)
{
    for( int i=0;i<N_x;i++)
    {
        for (int j=0; j<N_y;j++)
        {
            k[i][j]=h[i][j]+m[i][j];
        }
    }
    return;
}
void final_matrixadd(double** a, double** b, double** c, double** d, double** e, double f)
{
    for( int i=0; i<N_x;i++)
    {
        for (int j=0; j<N_y;j++)
        {
            a[i][j]=f*(b[i][j]/6+c[i][j]/3+d[i][j]/3+e[i][j]/6);
        }
    }
    return;
}
```

Inside the Main function, the forth-order runge-kutta method is used after discretization of functions of dvx/dt, dvy/dt and dh/dt. The calculation are inside a time marching loop from t=0 to t=100 with a certain size of delta_t to calculate the next stage vx, vy and h values. The code is designed as:

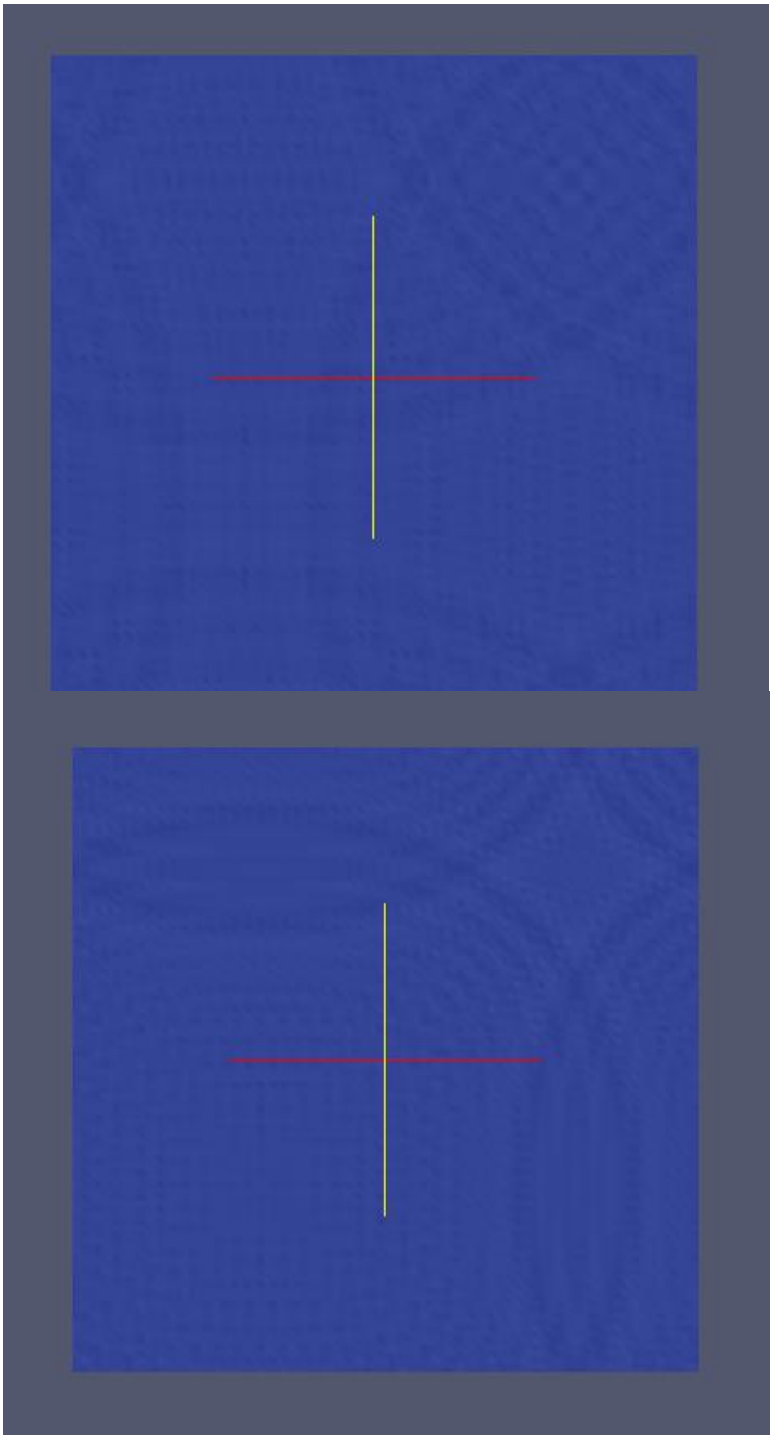
```
for(int l=0; l<N_t-1; l++)
    {
        t += delta_t;;
        f(vx,vy,h,kx1,ky1,kh1);
        matrixmult(kxt,kx1,delta_t/2);
        matrixmult(kyt,ky1,delta_t/2);
        matrixmult(kht,kh1,delta_t/2);
        matrixadd(vxt,vx,kxt);
        matrixadd(vyt,vy,kyt);
        matrixadd(ht,h,kht);
        f(vxt,vyt,ht,kx2,ky2,kh2);
        matrixmult(kxt,kx2,delta_t/2);
        matrixmult(kyt,ky2,delta_t/2);
        matrixmult(kht,kh2,delta_t/2);
```

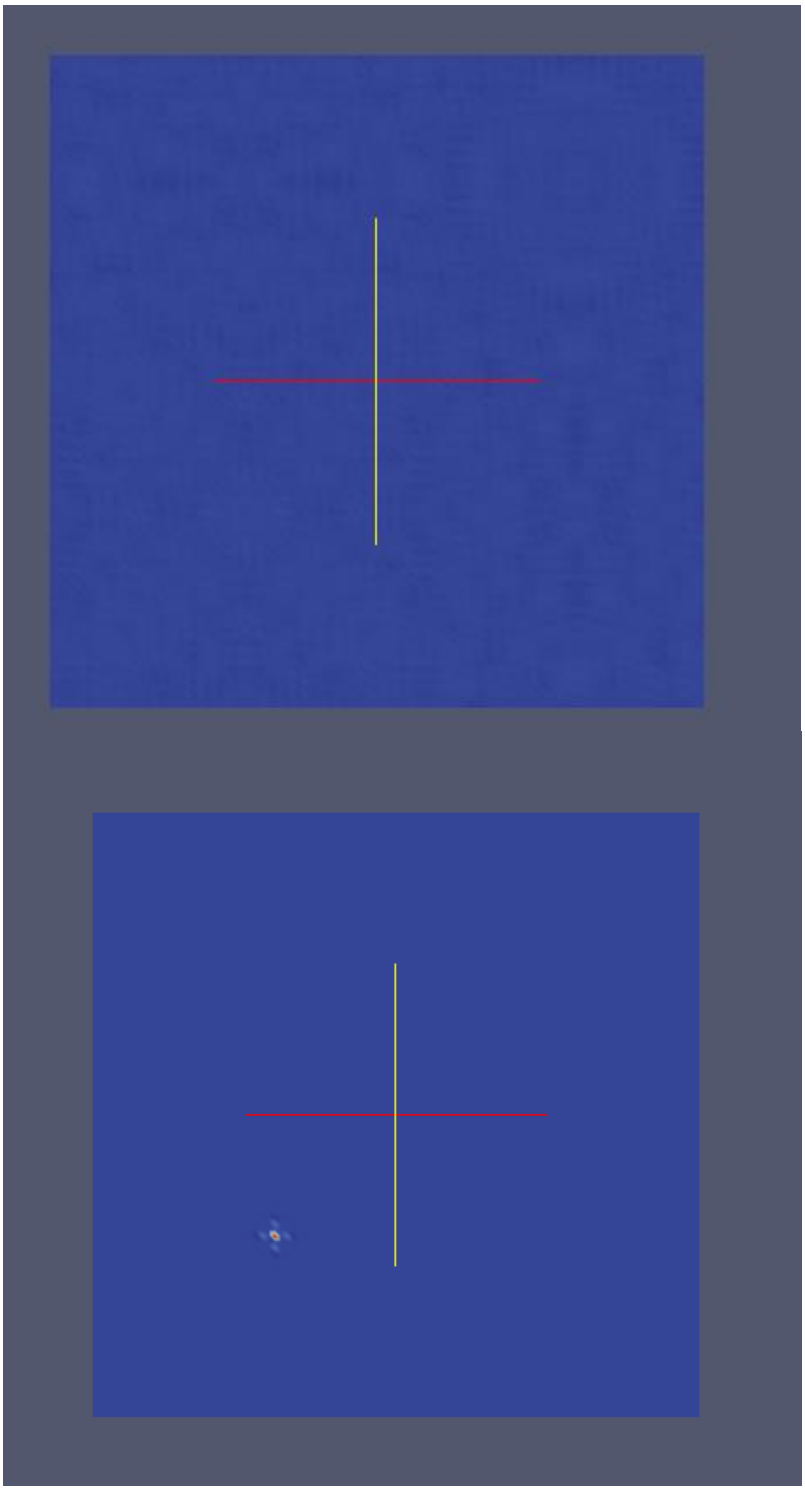
```
            matrixadd(vxt,vx,kxt);
            matrixadd(vyt,vy,kyt);
            matrixadd(ht,h,kht);
            f(vxt,vyt,ht,kx3,ky3,kh3);
            matrixmult(kxt,kx3,delta_t);
            matrixmult(kyt,ky3,delta_t);
            matrixmult(kht,kh3,delta_t);
            matrixadd(vxt,vx,kxt);
            matrixadd(vyt,vy,kyt);
            matrixadd(ht,h,kht);
            f(vxt,vyt,ht,kx4,ky4,kh4);
            final_matrixadd(kxt,kx1,kx2,kx3,kx4,delta_t);
            final_matrixadd(kyt,ky1,ky2,ky3,ky4,delta_t);
            final_matrixadd(kht,kh1,kh2,kh3,kh4,delta_t);
            matrixadd(vx,vx,kxt);
            matrixadd(vy,vy,kyt);
            matrixadd(h,h,kht);

            if(l%50==0)
            {

                  writeData(l, h, x_min, y_min, delta_x, delta_y,N_x, N_y);

            }



      }
void writeData(int k, double** h, double x_min, double y_min, double delta_x, double delta_y,
int N_x, int N_y)
{
      double x, y;
      char fileName[128];
      fstream file;
      sprintf(fileName, "waterequation_%05d.csv", k);
      file.open(fileName, ios::out);
      for(i=0; i<N_x; i++)
      {
            x = x_min + i*delta_x;
            for(j=0; j<N_y; j++)
            {
                  y = y_min + j*delta_y;
                  file << x << ",\t" << y << ",\t" << h[i][j] << endl;
            }
```

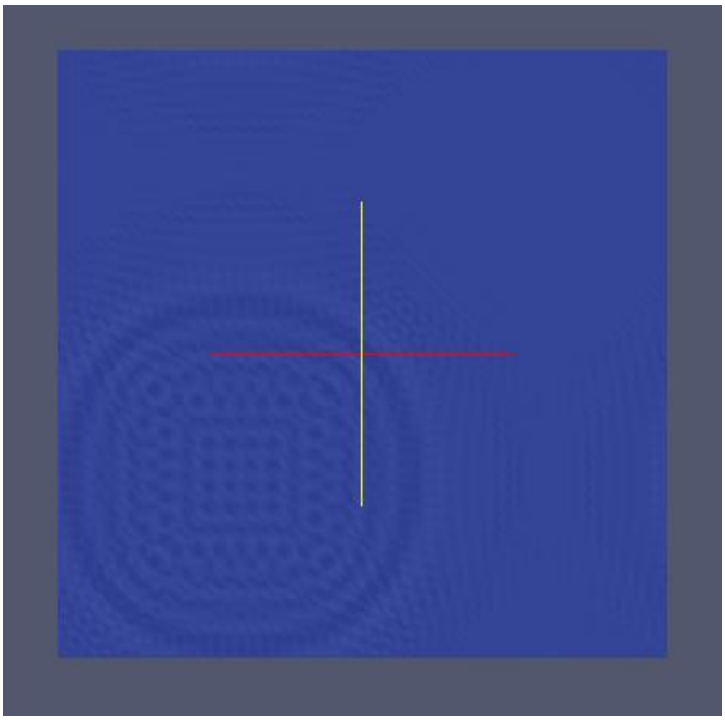
```
        }
        file.close();
        return;
}
```

Where l is the time marching loop variable and writeData function is to perform data file generating at different stages which will be used in PARAVIEW plotting.

Sample Plots by PARAVIEW based on C++ and parallelized C++:







OpenMP Program Implementation:

```
double wtime = omp_get_wtime();
int N_Threads = omp_get_max_threads();
```

We started off by initializing a time clock using the omp_get_time() method. The clock starts at execution and is used to measure the time it takes for the parallel execution to complete with a variety of different workloads and worker threads. Using Omp_get_max_threads(); the number of threads set at runtime are called on and depending on the implementation of OMP, workloads at various for loops throughout the program are divided amongst workers. Said are stored in an int variable N_Threads.

```
#pragma omp parallel default(shared) private(m, mm)
for( m=1, mm=N_y;m<N_x; m++,mm+=N_y)
{
vx[m]=&vx[0][mm];
vy[m]=&vy[0][mm];
h[m]=&h[0][mm];
}
```

The above small excerpt from our loop is where all our arrays pointers are set into position to point towards the beginning of out greater 2d arrays aka where we set our initial conditions for allocated arrays prior to this. Using OpenMP's parallel default (shared) function meant that memory allocated to all worker threads was to be shared amongst them. That meant that different threads could access the same memory and manipulate various variables to

achieve a common result. In this case, our target variables were m and mm, both of which represent the row and column vector values of our 2D arrays. Because no dependency exists between each at runtime, we can parallel these processes in between one loop.

```
#pragma omp parallel default(shared) private(i, j, l, x, y)
Set initial condition and array values to zero
for(i=0; i<N_x; i++){
        for(j=0; j<N_y;j++){
        vx[i][j]=0;
vy[i][j]=0;
        }
}
```

Its here where we set the initial conditions for all our main 2D arrays, vx, vy, h, as well as the arrays required to store calculated values post Runge Kutta Methods. Again, due to lack of dependency, we decided to parallelize the process with shared memory and based on variables l,j,l,x,y.

```
#pragma omp parallel default (shared) private(l)
for(l=0; l<N_t-1; l++)    {
f(vx,vy,h,kx1,ky1,kh1);
matrixmult(kxt,kx1,delta_t/2);
matrixmult(kyt,ky1,delta_t/2);
matrixmult(kht,kh1,delta_t/2);
matrixadd(vxt,vx,kxt);
matrixadd(vyt,vy,kyt);
matrixadd(ht,h,kht);
}
```

In the time marching loop, depending on the number of active threads and size of your number of time steps, the threads parallelize the whole time marching loop which consists of a series of functions designed to divide the tasks of calculating a system of ODEs, adding them up and finally, applying the fourth order Runge Kutta method on them. Every iteration where processes are repeated four times each, the final values are then sent to a file to be written onto. Following a completion of all iterations, a number of Csv files are created by the writeData function.

```
#pragma omp single
if(l%10==0){
file<<t<<"¥n";
writeData(file,h);
file<<"¥n";
}
```

The writeData function is not parallelized but rather is done on a dedicated single workthread. Writing data is a sequential long task that takes time, and after testing, we found, was not a parallel task.

```
#pragma omp for schedule(static)
for(i=0; i<N_x; i++){
x_a=i-2;
x_b=i-1;
x_c=i+1;
x_d=i+2;
        }
```

Within the f function, we tried to parallelize it as a whole but ran into a snag with synchronization of data from various threads. That's because of the number of dependencies during the runge kutta calculations (4^{th} order) required subsequent calculations of matrixes to be present. That said, we were able to successfully apply OpenMP to the multiple in built functions to increase efficiency.

Scaling Run Analysis of Paralyzed Programs:

Whilst performing Scaling runs on Merri, we discovered that OpenMP parallelizing the main time marching loop the values for H in the outputted files was NULL or Nan. As a result we decided to parallelize the in built functions of Matrix add and Matrix multiply so that at each time step when they were called, we could parallelize them. This resulted in an average running time of around 4-5 seconds depending on the number of threads.

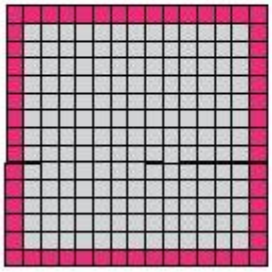
MPI Program Implementation:

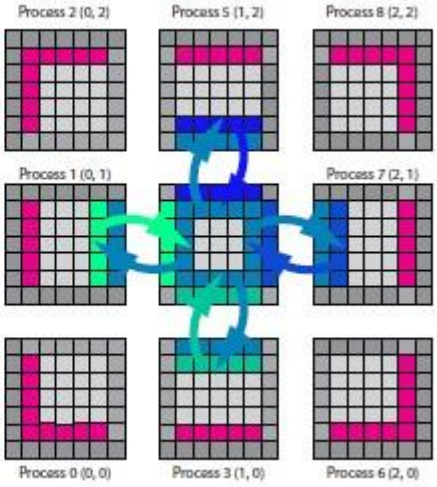
General introduction about MPI:

MPI stands for ***Message Passing Interface***. It could be considered as solving a problem by breaking down the entire problem into N smaller sub-problems with their own allocated memory. The N was determined by the number of processes or for example, CPU cores are used to solve the problem. So if the number of used CPUs is large, more sub-problems with their unique memory will be generated by the program.

Solving Shallow Water Equations by MPI:

The system of shallow water equations is found to be a 2 dimensional problem. The example could be divided as the example graph below:





The data communication is completed through each process with adjacent rankings. The method to solve the system of shallow water equations are done by ***forth-order central difference method***. So **two circles of ghost points** will be added outside the original data since X and Y direction of arrays will need **4 more points** to achieve the forth-order central difference method. The boundary condition of the equations is observed to be ***periodic***. So the ranking of the process will always have four neighbors in left, right, top and bottom directions. For example, the (N, N) process will have neighbors (N-1, N), (0, N), (N, N-1) and (N, 0). The periodic boundary condition makes each process at the boundary receive the information from the process located at the other side of the boundary. Each process will compute concurrently after sending and receiving data from its neighbor processes which means all the sub-problems are solved at the same time. The computing procedure after data sending and receiving are same as the C++ codes for the non-parallel machines. In our approach, every process are chosen to be as a matrix of size 24x24 blocks, therefore with more processors applied in the computation, delta_x and delta_y will have smaller value then the calculation will have a better accuracy. The blocking communication is used in this computing program, this means that the routine or data passing process will wait till the data are received before executing the next steps. This might be less efficient than non-blocking communication but offered safer environment for data transferring.

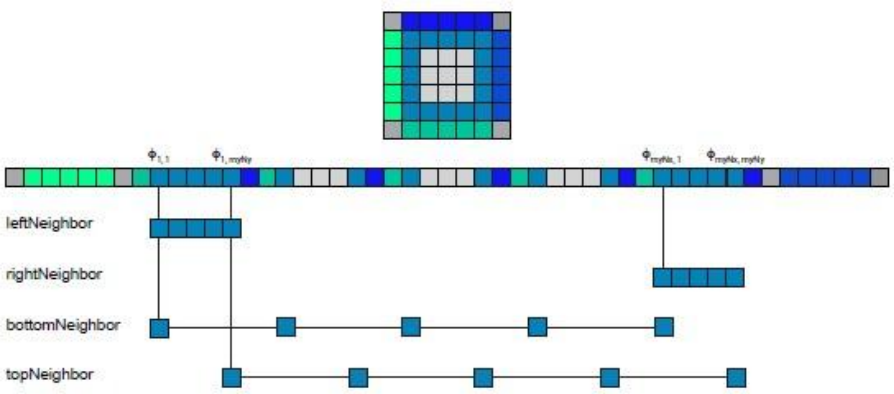
For communication of each process about the right and left layers of data, the data sending and receiving could be executed easily. The matrix was stored contiguously in terms of memory addresses, the left and right layers of data are found to be adjacent to each other along the memory address and the length of the layer is myN_y, so the data send and receive code could be implemented as an example below:

In the time marching loop:

```
MPI_Sendrecv(&(vx[2][2]), myN_y, MPI_DOUBLE,leftNeighbor, 0, &(vx[myN_x+2][2]),
myN_y, MPI_DOUBLE, rightNeighbor, 0, Comm2D, &status);
MPI_Sendrecv(&(vx[3][2]), myN_y, MPI_DOUBLE,leftNeighbor, 0, &(vx[myN_x+3][2]),
myN_y, MPI_DOUBLE, rightNeighbor, 0, Comm2D, &status);
MPI_Sendrecv(&(vx[myN_x+1][2]),myN_y, MPI_DOUBLE, rightNeighbor, 0, &(vx[1][2]),
myN_y, MPI_DOUBLE, leftNeighbor, 0, Comm2D, &status);
```

```
MPI_Sendrecv(&(vx[myN_x][2]),myN_y, MPI_DOUBLE, rightNeighbor, 0, &(vx[0][2]),
myN_y, MPI_DOUBLE, leftNeighbor, 0, Comm2D, &status);
MPI_Sendrecv(&(vy[2][2]), myN_y, MPI_DOUBLE,leftNeighbor, 0, &(vy[myN_x+2][2]),
myN_y, MPI_DOUBLE, rightNeighbor, 0, Comm2D, &status);
MPI_Sendrecv(&(vy[3][2]), myN_y, MPI_DOUBLE,leftNeighbor, 0, &(vy[myN_x+3][2]),
myN_y, MPI_DOUBLE, rightNeighbor, 0, Comm2D, &status);
MPI_Sendrecv(&(vy[myN_x+1][2]),myN_y, MPI_DOUBLE, rightNeighbor, 0, &(vy[1][2]),
myN_y, MPI_DOUBLE, leftNeighbor, 0, Comm2D, &status);
MPI_Sendrecv(&(vy[myN_x][2]),myN_y, MPI_DOUBLE, rightNeighbor, 0, &(vy[0][2]),
myN_y, MPI_DOUBLE, leftNeighbor, 0, Comm2D, &status);
MPI_Sendrecv(&(h[2][2]), myN_y, MPI_DOUBLE,leftNeighbor, 0, &(h[myN_x+2][2]),
myN_y, MPI_DOUBLE, rightNeighbor, 0, Comm2D, &status);
MPI_Sendrecv(&(h[3][2]), myN_y, MPI_DOUBLE,leftNeighbor, 0, &(h[myN_x+3][2]),
myN_y, MPI_DOUBLE, rightNeighbor, 0, Comm2D, &status);
MPI_Sendrecv(&(h[myN_x+1][2]),myN_y, MPI_DOUBLE, rightNeighbor, 0, &(h[1][2]),
myN_y, MPI_DOUBLE, leftNeighbor, 0, Comm2D, &status);
MPI_Sendrecv(&(h[myN_x][2]),myN_y, MPI_DOUBLE, rightNeighbor, 0, &(h[0][2]),
myN_y, MPI_DOUBLE, leftNeighbor, 0, Comm2D, &status);
```

Because the two dimensional array (matrix) is stored contiguously in terms of memory address, the top or bottom layer element's position is re-calculated as they are myN_y+4 blocks apart from the previous block in X direction. So declare another data type to select all the required top and bottom layer of data for the block-communication among each process. This graph is a example of the structure of each matrix:



The example code is implemented as below:

Declare MPI data type first:

```
MPI_Type_vector(myN_x, numElementsPerBlock, myN_y+4, MPI_DOUBLE, &strideType);
MPI_Type_commit(&strideType);
```

In the time marching loop, call the following function:

```
MPI_Sendrecv(&(vx[2][2]), 1, strideType,bottomNeighbor, 0, &(vx[2][myN_y+2]), 1, strideType,
topNeighbor, 0, Comm2D, &status);
```

```
MPI_Sendrecv(&(vx[2][3]), 1, strideType,bottomNeighbor, 0, &(vx[2][myN_y+3]), 1, strideType,
topNeighbor, 0, Comm2D, &status);
MPI_Sendrecv(&(vx[2][myN_y+1]), 1, strideType,topNeighbor, 0, &(vx[2][1]), 1, strideType,
bottomNeighbor, 0, Comm2D, &status);
MPI_Sendrecv(&(vx[2][myN_y]), 1, strideType,topNeighbor, 0, &(vx[2][0]), 1, strideType,
bottomNeighbor, 0, Comm2D, &status);
MPI_Sendrecv(&(vy[2][2]), 1, strideType,bottomNeighbor, 0, &(vx[2][myN_y+2]), 1, strideType,
topNeighbor, 0, Comm2D, &status);
MPI_Sendrecv(&(vy[2][3]), 1, strideType,bottomNeighbor, 0, &(vx[2][myN_y+3]), 1, strideType,
topNeighbor, 0, Comm2D, &status);
MPI_Sendrecv(&(vy[2][myN_y+1]), 1, strideType,topNeighbor, 0, &(vx[2][1]), 1, strideType,
bottomNeighbor, 0, Comm2D, &status);
MPI_Sendrecv(&(vy[2][myN_y]), 1, strideType,topNeighbor, 0, &(vx[2][0]), 1, strideType,
bottomNeighbor, 0, Comm2D, &status);
MPI_Sendrecv(&(h[2][2]), 1, strideType,bottomNeighbor, 0, &(vx[2][myN_y+2]), 1, strideType,
topNeighbor, 0, Comm2D, &status);
MPI_Sendrecv(&(h[2][3]), 1, strideType,bottomNeighbor, 0, &(vx[2][myN_y+3]), 1, strideType,
topNeighbor, 0, Comm2D, &status);
MPI_Sendrecv(&(h[2][myN_y+1]), 1, strideType,topNeighbor, 0, &(vx[2][1]), 1, strideType,
bottomNeighbor, 0, Comm2D, &status);
MPI_Sendrecv(&(h[2][myN_y]), 1, strideType,topNeighbor, 0, &(vx[2][0]), 1, strideType,
bottomNeighbor, 0, Comm2D, &status);
```

Conclusion:

The shallow water equation system is calculated by discretize the system of non-linear PDEs into system of ODEs by applying the finite difference method. Then the next stage values of the three variables are calculated by applying the forth-order Rungge Kutta method. After calculating and plotting the result by MATLAB and C++, the result could be demonstrated at different time steps. Parallelization of the algorithm will decrease the time consumption for solving this problem.