C&EE 103: Final Project Report
Agam Bedi
08/21/2022

Analyzing a Boat Traversing a River

Introduction:

The project discusses the path of a boat that moves from point A to point B along a river, with a strong current flowing through. Our goal for this project is to navigate the boat to ensure that it reaches point B while traveling the smallest distance. Two different methods can be utilized to navigate this pathway. Firstly, the boat goes at a fixed angle from point A to point B, which mimics the current of the river. The angle doesn't change to illustrate the constant current that the boat is facing. The second way that this problem can be analyzed is by constantly changing the angle at which the boat is traveling. This is to imitate the boat constantly looking and going towards point B. Our job is to illustrate these two navigation methods in two different ways, using a linear initial value problem to show the first path, and a non-linear initial value problem to show the second path. Furthermore, different numerical methods can be employed to further analyze these IVPs, to ultimately understand the path of the boat using both methods and the time taken to go from point A to point B.

Theory:

For this project, two initial value problems (IVPs) were used to illustrate the two navigation methods, as explained above. Before delving into these IVPs, some key assumptions must be ironed out. Firstly, we will be modeling the boat as a point, to remove negligible deviations caused by utilizing a full boat as our model. Moreover, it is assumed that the boat will travel at a constant speed, which will be labeled as v_B for the rest of the project. The surface velocity of the current of the river will also be taken into consideration and will be described as the variable v_R, which only changes in the x direction. This is to simplify the process, and not have a multi-variable equation for the current of the river. Finally, it is assumed that the weather has no impact on the path of the boat. The navigation paths of both the methods are illustrated in figures 1 and 2 shown below:

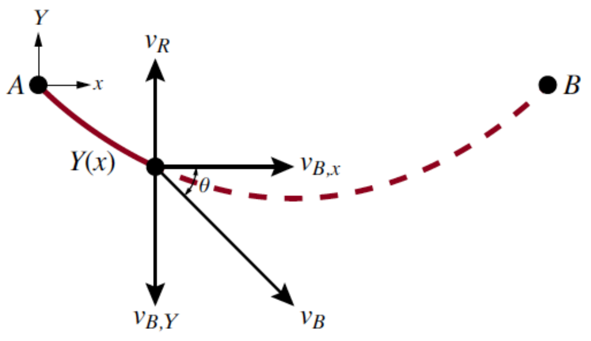


Figure 1: Navigation one (linear)

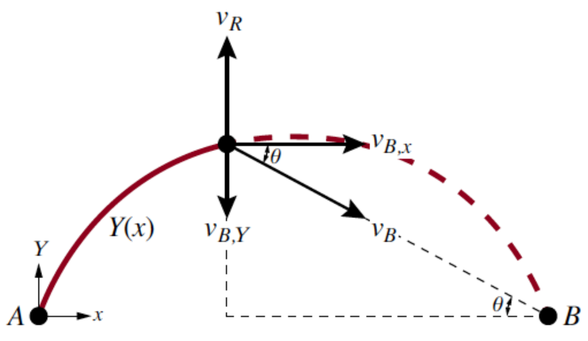


Figure 2: Navigation two (non linear)

The analysis will be done based on these two diagrams. After analyzing these paths, and looking at figure one, in order to define the velocity of the boat, $v_{B,Y}$ at an arbitrary point (x,Y), while taking into account the current and the time, *t*, the following equation can be used:

$v_{B,Y} = dY/dt = v_R(x(t)) - v_B\sin(\theta)$

Which can then further be analyzed to become the first-order, linear ordinary differential equation:

$$Y'(x) = \frac{v_R(x)}{v_B cos(\theta)} - tan(\theta) \qquad \text{Equation. 1}$$

$Y(A) = 0$

Where Y(x) illustrates the path of the boat, point A describes the boat's starting point and θ represents the constant angle illustrated in figure 1.

It is also important to note the time taken for the boat to traverse the river. We can analyze, utilizing our knowledge of physics, that the time taken to travel will be calculated by the following equation:

$$T = \int_A^B (\frac{1}{v_B cos(\theta)} dx = \frac{B}{v_B cos(\theta)} \qquad \text{Equation. 2}$$

For the second navigation method, we can utilize Equation. 1, and see how the pythagorean theorem is applied in Figure 2. Using this information, we can come up with the non linear IVP:

$$Y'(x) = \frac{v_R(x)}{v_B} \frac{\sqrt{(B-x)^2 + Y(x)^2}}{B-x} - \frac{Y(x)}{B-x} \qquad \text{Equation. 3}$$

$Y(A) = 0$

With the same definitions as mentioned above for equation 1, and B being the final destination of the boat. For the time taken to travel, *T*, for navigation method 2, we can use the following integration:

$$T = \int_A^B \frac{\sqrt{(B-x)^2 + Y(x)^2}}{v_B(B-x)} dx \qquad \text{Equation. 4}$$

Numerical Methods:

For my specific project, after constructing a least-squares fitting function on MATLAB to describe the surface velocity of the current of the river as a function of x, the linear IVP (given by equation 1) was solved using Heun's method. This method was used to create data points (x_i, y_i), which can then be graphed. It was important to also create a flexible script to input any polynomial order from 1 to 7.

Heun’s method is used to solve Ordinary Differential Equations (ODEs) when you have an initial condition (usually derived from the forward Euler method). It uses the same conceptual backbone as the Euler method. Heun’s method works as a more accurate Euler method, as it uses values from the forward Euler method and utilizes the format of the implicit trapezoidal method, but makes it explicit, and combines them to essentially be a predictor-corrector. It has lower error as it has second order accuracy, in comparison to the first order accuracy of the Euler method.

This was then further interpolated using radial basis functions. What these radial basis functions do is they create a value phi, which is an interpolant, that interpolates the data points we calculated from using the Heun's method. Using this interpolation method, for this problem specifically, it can create four different pathways to describe the boat's travel. The equation to find phi is:
phiVal = sqrt(1+(eps*r)^2); where eps is the shape parameter (1/2h) and r is the distance between the node and x.

The non linear IVP was solved by utilizing the backward Euler method, with the Newton's method with a tolerance of 10^{-12}. This was used to find data points (x_i, y_i), and was subsequently graphed. Backward euler method for non-linear IVPs also utilizes Newton's method in order to find the next point (y_{n+1}). This is because the function depends on both y and x, and therefore, with on unknown, we use Newton's method. This method first finds the root for y_{n+1}, and then uses Newton's iterations, which is an equation used to solve the IVP. As stated earlier, Backward Euler has first order accuracy (an error of O(h)) and is a very stable method.

Finally, we utilized the natural cubic spline method, as well as the trapezoidal rule, to perform numerical integration to find the travel time T and interpolate the paths of the boat using navigation method 2. The natural cubic spline uses the LU-factorization of a tridiagonal matrix, which found the M_i values for the cubic spline. After which the trapezoidal rule was used to find the travel time T. Cubic splines are usually used to generate non-oscillatory interpolations, which are usually smooth. Natural cubic spline is an interpolant that follows a set of rules, and therefore, forms a tridiagonal set of solutions.

Ultimately, both the numerical analysis methods and the integration methods work hand in hand to solve the IVPs, find the travel time T, and interpolate the path of the boat.

Results:

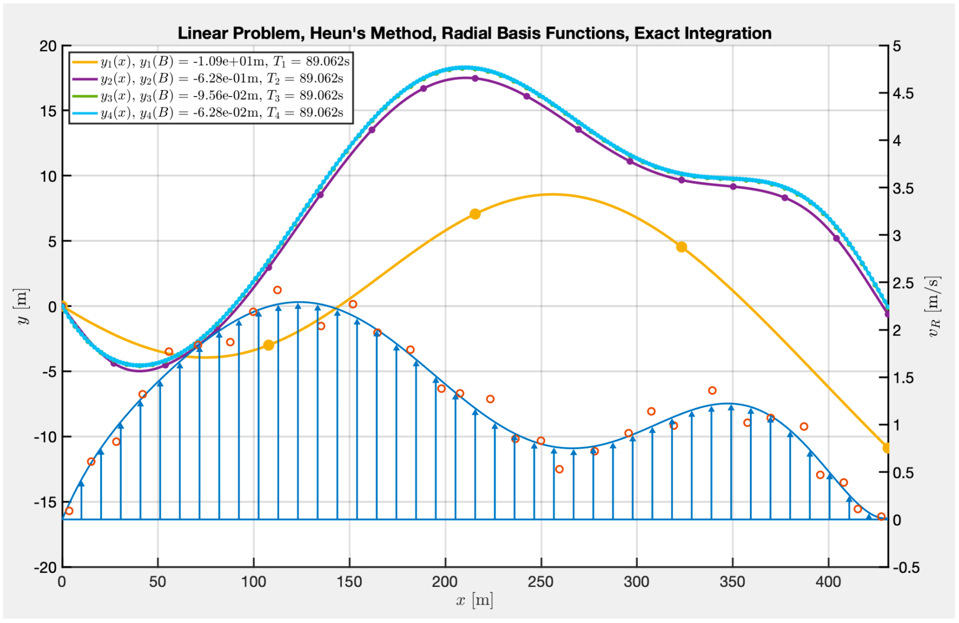


Figure 3: Results of Navigation 1, using a polynomial order = 7

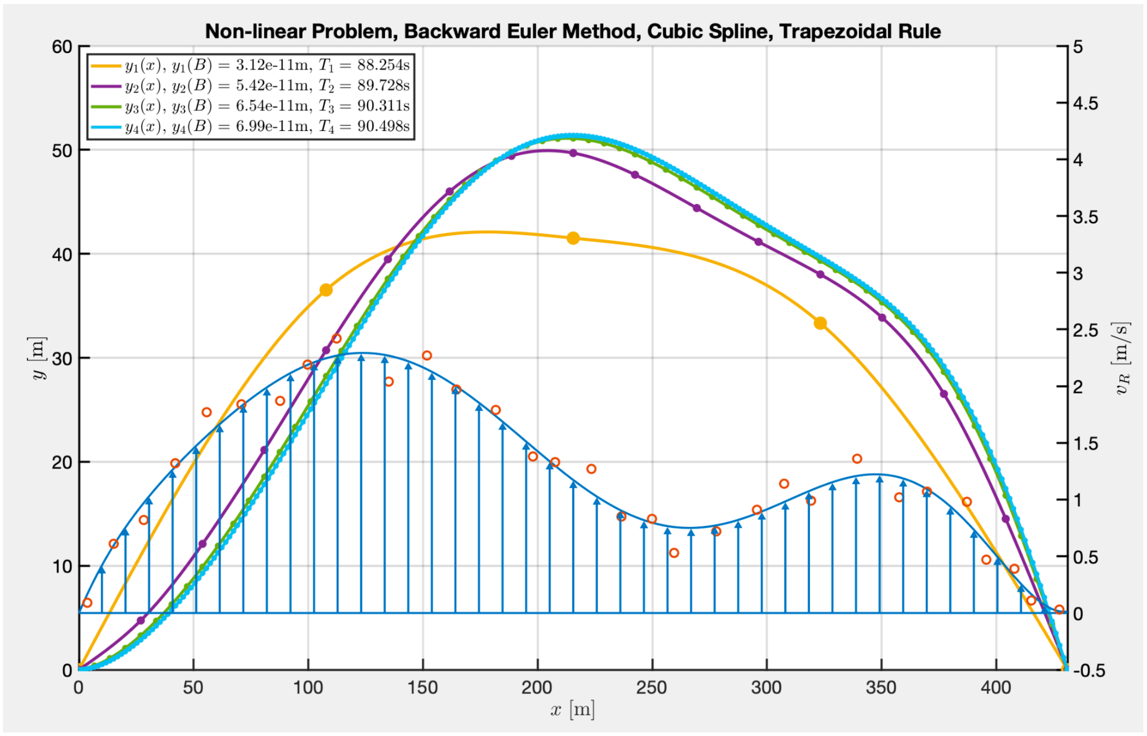


Figure 4: Results of Navigation 2, using a polynomial order = 7

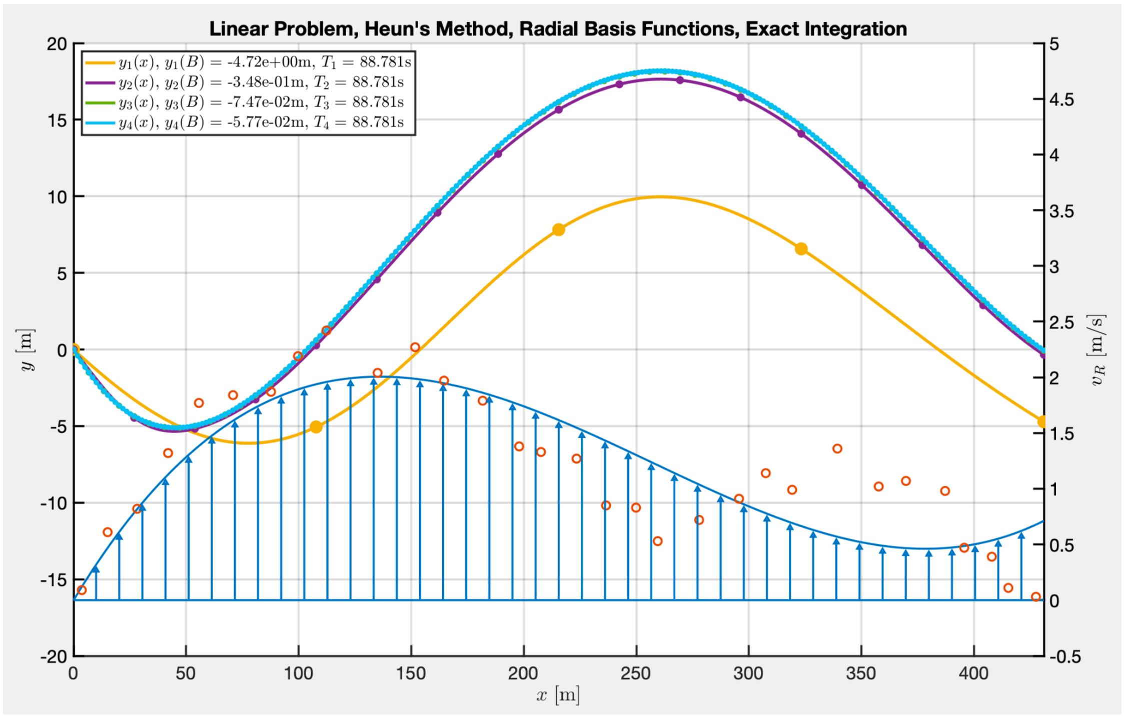


Figure 5: Results of Navigation 1, using a polynomial order = 3

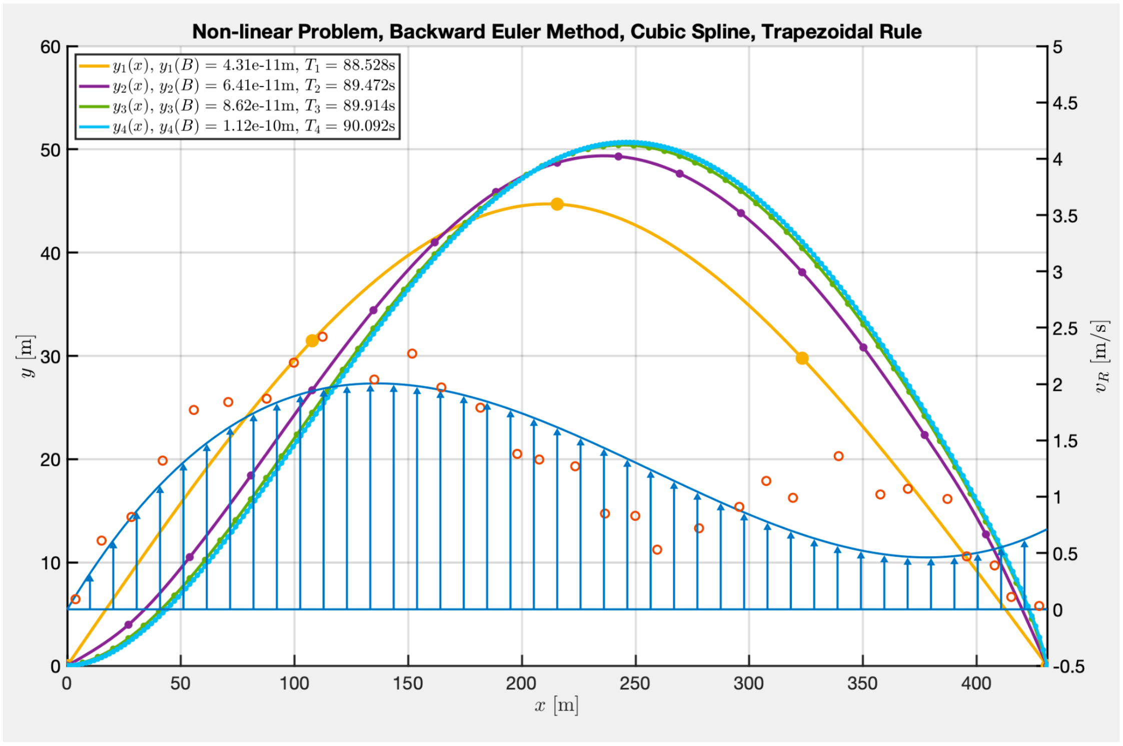


Figure 6: Results of Navigation 2, using a polynomial order = 3

Discussion of Results:

As shown in figures 3-6, there is a stark difference between the graphs and travel time when changing the polynomial order from 7 to 3, as well as when using the different navigation methods.

Looking at the two figures for the linear method (figures 3 and 5), you can see that at a polyOrd of 3, the path is a lot smoother, and the travel time decreases by 0.281 seconds than a polyOrd of 7. Although it is a slight difference, it shows how there is a more optimal polynomial order when looking at solving the linear IVP. Additionally, although not pictured in this report, when increasing the value of theta significantly, the graph starts to not make any sense, as the script illustrates the pathway going in reverse. Something interesting I found about the linear data, and using the Heun's method, is how it is different from the forward and backward Euler method. It seems to do the job of both at the same time, by approximating it from both directions, and utilizing that to come up with its solutions.

For the non-linear solution (shown in figures 4 and 6), similar conclusions can be made. As you lower the polyOrd (down to 3 from 7) the path becomes a lot smoother. However, the travel time varies for each path. With a polyOrd of 7, there is the lowest time taken with 88.254 with the y1 path. Yet, this pattern is not always consistent, as paths 2-4 are faster for polyOrd 3 than polyOrd 7. This is an interesting pattern, and illustrates how using the cubic spline to find four different paths for a non-linear IVP can yield varying results for T, which contrasts the linear IVP. Although it is not pictured here, at polyOrd 1, for both IVPs, the lowest T is calculated.

Conclusion:

In conclusion, one interesting thing I found about this problem was how we can utilize MATLAB and all the elements that we have learned throughout this course, to solve a complex problem. Using everything we have learned, we were able to analyze the pathway of a boat in a river in two different ways. Furthermore, we were able to plot this, and extrapolate a lot of data using MATLAB, which shows how powerful of a tool it can be.

The most interesting part of this project was learning how to apply everything we learned and how all the functions and methods work hand in hand. How different numerical analysis methods can all be utilized to approximate the pathway of the boat, and we can use different interpolation techniques to come up with a solution for the travel time. I liked how we had to use different methods for both IVPs, so there was a more thorough understanding of the coursework learned.

Appendix:

Attached after this page is all of the MATLAB code, including the main script and 8 function files.

Contents

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%   Final Project Structure with Code
%   A Boat Traversing a River
%
%   Author: Marcus Rüter
%   Date: 08/11/2022
%   Edited By: Agam Bedi
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Clear Cache

```
clc;
clearvars;
close all;
```

Initialization

```
whichProblem = 1;           % 1 = Lin
                            % 2 = Non-Lin
whichMethod = 1;           % 1 = Heun
                            % 2 = Backward Euler
whichInterpolant = 1;      % 1 = Radial Basis
                            % 2 = Cubic Spline
whichIntegration = 1;      % 1 = Trapezoidal Rule

color = lines(6);           % Default Matlab colors for piecewise plots
B = 431;                     % Point B [m]
delta = 1e-10;                 % delta [m] to create B+delta
vB = 5;                    % Boat velocity [m/s]
x0 = 0;                    % Start x-value [m]
y0 = 0;                    % Initial condition [m]
theta = pi/6;                 % Start value of theta
nSI = [4 16 64 256];                 % Array of numbers of subintervals
nSIMax = nSI(end);          % Maximum number of subintervals
nN = nSI + 1;                % Array of numbers of nodes
h = (B-x0) ./ nSI;            % Array of step sizes
nRun = length(nSI);         % Number of runs
y = zeros(nRun,nN(end));    % Array of y-values at nodes x_i
w = zeros(nRun,nN(end));    % Array of w-values at nodes x_j
T = zeros(nRun, 1);              % Array of travel times
nPP = nN(end)*10+1;         % Number of points for plotting
xPlot = linspace(x0,B,nPP); % x-values for plotting (1D array)
yPlot = zeros(nRun,nPP);    % y-values for plotting for all runs (2D array)
xRiver = [3.65 15.13 28.24 41.98 55.67 70.82 87.71 99.67 112.33 135.05 151.73 164.54 181.71 197.92 207.61 223.42 236.51 249.84 259.48 277.89 295.58 30
vRiver = [0.09 0.61 0.82 1.32 1.77 1.84 1.87 2.19 2.42 2.04 2.27 1.97 1.79 1.38 1.33 1.27 0.85 0.83 0.53 0.72 0.91 1.14 0.99 1.36 1.02 1.07 0.98 0.47
xRiverScaled = xRiver/B;    % Scaled xRiver-values for least squares
nDP = length(xRiver);           % Number of river data points
polyOrd = 7;               % Polynomial order for least squares
p = zeros(polyOrd,nDP);     % Vector of basis functions
M = zeros(polyOrd,polyOrd); % M-matrix
b = zeros(polyOrd,1);       % Right-hand side b-vector

set(0,'DefaultFigureRenderer','painters');  % Create vector graphics
```

Least-squares Data Fitting

```
for i = 1:polyOrd
       p(i,:) = xRiverScaled.^i; % Evaluate vector of basis functions
end
for i = 1:nDP % Loop over all river data points
       M = M + p(:,i)*p(:,i)';
       b = b + vRiver(i)*p(:,i);
end
aLS = M\b;
vRPlot = computeLSRiverVel(B+delta, polyOrd, aLS, xPlot);% Call computeLSRiverVel to generate data points and to plot the
%least-squares function (river velocity)
```

Anonymous Functions:

f(x) for linear IVP

```
fLin = @(x,theta) computeLSRiverVel(B,polyOrd,aLS,x')/(vB*cos(theta))-(tan(theta));
% f(x,y(x)) for non-linear IVP
fNonLin = @(x,y) computeLSRiverVel(B,polyOrd,aLS,x')/vB * (sqrt((B+delta-x)^2 + y^2)/(B+delta-x)) - (y/(B+delta-x));
% Derivative of f(x,y(x)) w.r.t. y for non-linear IVP
dfNonLinDy = @(x,y) computeLSRiverVel(B,polyOrd,aLS,x') .* ((y)/(vB*(B+delta-x)*(sqrt(y^2+(B+delta-x)^2))))-(1/(B+delta-x));%derivative with respect t
% Integrand for numerical integration
f = @(x,y) sqrt((B+delta-x).^2 + (y.^2))./(vB*(B+delta-x));
```

Compute Numerical Results for Linear or Non-linear IVP

```
if whichProblem == 1            % Find (fixed) theta only for linear IVP
    x = x0:h(end):B; % Nodes array
    theta = pi/6;
    y(end,end) = 10;            % Large value at y(B) to start while loop
    yPlot(end,end) = 10;        % Large val. at yPlot(B) to st. while loop

    % Loop until y(B) <= 0.1 or yPlot(B) <= 0.1 (for collocation method)
    while abs(y(end, end)) >= 0.1
        % Decrease theta
        theta = theta - (1/10000);
        [y(end,:),~] = computeHeunLinSol(y0,fLin,theta,x);
    end
end

% Solve linear or non-linear IVP
for i = 1:nRun
    x = x0:h(i):B; % Nodes array

    if whichProblem == 1        % Linear IVP
        problem = "Linear Problem";         % Define problem for plot

        [y(i,1:nN(i)),method] = computeHeunLinSol(y0,fLin,theta,x);
    else                        % Non-linear IVP
        problem = "Non-linear Problem";     % Define problem for plot

        [y(i,1:nN(i)),method] = computeBEulerNonLinSol(y0,fNonLin,dfNonLinDy,x);
    end
end
```

(Numerical) Integration (and Interpolation for Non-linear IVP)

```
for i = 1:nRun
    x = x0:h(i):B; % Nodes array

    if whichProblem == 1                % Linear IVP
        T(i) = B/(vB*cos(theta));
        integration = "Exact Integration";
    else                                % Non-linear IVP
        xEP = x;
        [yEP, interpolant] = evaluateCubicSpline(x,y(i,1:nN(i)),xEP);

        %or
        % Initialize y_EP
        % Determine y_EP based on w-values and phi-functions
        [T(i),integration] = computeTrapezoidalRuleSol(f,xEP,yEP);
   end
end
```

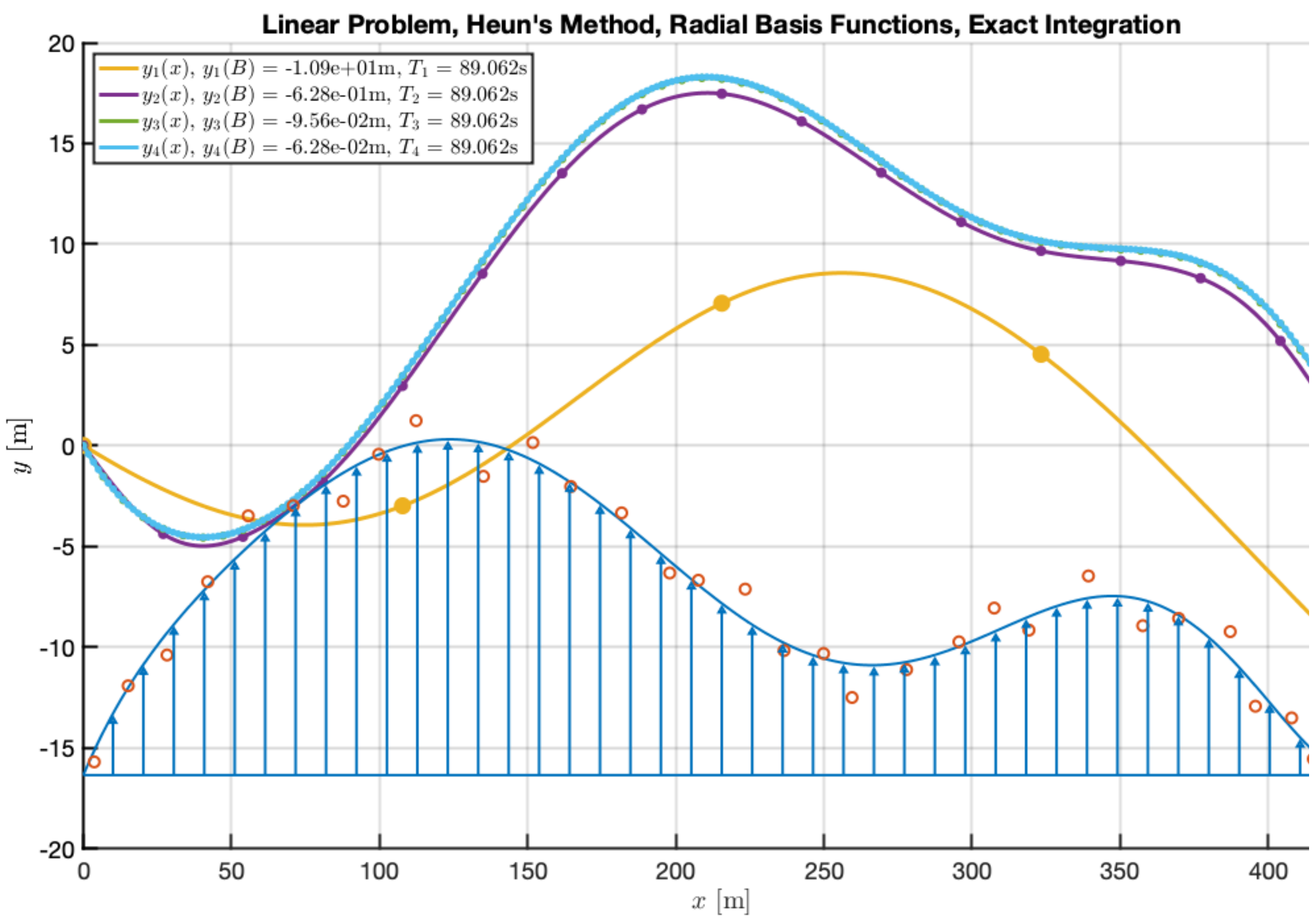
Interpolation for Plotting (only for non-collocation methods)

```
for i = 1:nRun
    x = x0:h(i):B; % Nodes array

    [yPlot(i,:),interpolant] = evaluateInterpolant(whichInterpolant,x,y(i,1:nN(i)),xPlot);
end
```

Plot Results

```
figure(1);                                  % Open Figure 1

% Title for all other methods
titleText = ...
        sprintf('%s, %s, %s, %s',problem,method,interpolant,integration);

title(titleText);                           % Create title
xlabel('$x$ [m]','interpreter','latex');    % x-label
xlim([x0 B]);                               % x-limits
set(gcf,'Position',[30 350 1250 750]);      % Plot window size and position
set(gca,'LineWidth',2,'FontSize',18);       % Axes line width and font size
grid on;                                    % Turn on grid
hold on;                                    % Set hold to on

colororder({'k','k'});                      % Color order for two y-axes
yyaxis right;                               % Define right y-axis
ylabel('$v_R$ [m/s]','interpreter','latex');% y-label for right y-axis
```

```
ylim([-0.5 5]);                              % y-limits for right y-axis

% Plot LS fitting function plus horizontal line (optional) and data points
plot(xPlot,vRPlot,'-','LineWidth',2,'Color',color(1,:));
plot([x0 B],[0 0],'-','LineWidth',2,'Color',color(1,:));
plot(xRiver,vRiver,'o','MarkerSize',8,'Linewidth',2,'Color',color(2,:));

% Plot arrows to indicate direction of current (optional)
arrowsX = x0+10:(B-20)/40:B-10;              % x-positions of arrows
arrowsY = zeros(length(arrowsX),1);          % y-positions of arrows
vR = computeLSRiverVel(B,polyOrd,aLS,arrowsX);  % River velocity at x-pos.
quiver(arrowsX',vR'-0.05,arrowsY,vR'-0.05,'-^','LineWidth',2,...
    'AutoScale','off','ShowArrowHead','off','Alignment','head',...
    'MarkerSize',4,'Color',color(1,:));      % Plot arrows

yyaxis left;                                 % Define left y-axis
ylabel('$y$ [m]','interpreter','latex');     % y-label for left y-axis
% y-limits for left y-axis based on maximum values of numerical results
ylim([floor(min(yPlot(2,:))/20)*20 ceil(max(yPlot(2,:))/20)*20]);

for i = 1:nRun                               % Loop over all runs
    % Plot numerical solution y(x)
    p(i) = plot(xPlot, yPlot(i,:),'-','LineWidth',3,'Color',color(i+2,:));

    % Add data points only if needed
    plot(x0:h(i):B, y(i,1:nN(i)),'o','LineWidth',3,'Color',...
        color(i+2,:),'MarkerSize',10/i,MarkerFaceColor=color(i+2,:));
end

% Create plot legend
legend([p(1) p(2) p(3) p(4)],...
    sprintf('$y_1(x)$, $y_1(B) = $ %3.2em, $T_1 = $ %5.3fs',...
    yPlot(1,end),T(1)),...
    sprintf('$y_2(x)$, $y_2(B) = $ %3.2em, $T_2 = $ %5.3fs',...
    yPlot(2,end),T(2)),...
    sprintf('$y_3(x)$, $y_3(B) = $ %3.2em, $T_3 = $ %5.3fs',...
    yPlot(3,end),T(3)),...
    sprintf('$y_4(x)$, $y_4(B) = $ %3.2em, $T_4 = $ %5.3fs',...
    yPlot(4,end),T(4)),...
    'Location','NW','interpreter','latex');
```



Function computeLSRiverVel

```
function [vRAtX] = computeLSRiverVel(B,polyOrd,a,x)
vRAtX = zeros(1, length(x));
for i = 1:polyOrd
    vRAtX = vRAtX + a(i)*(x/B).^i;%Construct polynomial, sum of that, coefficients of monomial multiplied by
    %the polynomial order of x's.
end
```

```
Not enough input arguments.

Error in computeLSRiverVel (line 3)
vRAtX = zeros(1, length(x));
```

Published with MATLAB® R2022a

Function computeHeunLinSol

```
function [y,method] = computeHeunLinSol(y0,f,theta,x)

method = 'Heun''s Method';           % Define method for plot
h = x(2) - x(1);                     % Step size h
nSI = (x(end) - x(1))/h;            % Number of subintervals
y = zeros(nSI, 1);                       % y-array (approximate solution)
y(1) = y0;                           % Initial condition
% Loop over all subintervals
for n = 1:nSI
    y(n+1) = y(n) + (h/2)*(f(x(n),theta)+f(x(n+1), theta));
end
end
```

```
Not enough input arguments.

Error in computeHeunLinSol (line 5)
h = x(2) - x(1);                     % Step size h
```

Published with MATLAB® R2022a

Function evaluateInterpolant

```
function [yEP,interpolant] = evaluateInterpolant(whichInterpolant,x,y,xEP)
switch whichInterpolant                                    % Numerical methods
        case 1
            [yEP,interpolant] = evaluateRBFInterpolant(x,y,xEP);
        case 2
            [yEP,interpolant] = evaluateCubicSpline(x,y,xEP);
        otherwise
            error('Unknown numerical method!');
end
end
```

```
Not enough input arguments.

Error in evaluateInterpolant (line 3)
switch whichInterpolant                                    % Numerical methods
```

Published with MATLAB® R2022a

Function evaluatePhiJatX

```
function phiVal = evaluatePhiJatX(nodeJ,x,eps)

% Initialization
% Define radius r(x - x_j)
r = x - nodeJ;
phiVal = sqrt(1+(eps*r)^2);
end
```

```
Not enough input arguments.

Error in evaluatePhiJatX (line 6)
r = x - nodeJ;
```

Function evaluateRBFInterpolant

```
function [yEP,interpolant] = evaluateRBFInterpolant(x,y,xEP)
% Initialization
h = x(2)-x(1);
nSI = (x(end)-x(1))/h;
nN = nSI+1;
A = zeros(nN);
eps = 1/(2*h);                      % Shape parameter (depending on h)
yEP = zeros(length(xEP), 1);
interpolant = 'Radial Basis Functions';

for i = 1:size(A, 1)                                  % Loop over all rows of A
    for j = 1:size(A, 2)                              % Loop over all columns of A
        % Determine A(i,j) by evaluating phi_j(x_i)
        phiVal = evaluatePhiJatX(x(j),x(i),eps);
        A(i,j) = phiVal;
    end
end
% Solve linear system for weights w
w = A\y';
for i = 1:length(xEP)                     %phi = sum(        % Loop over all evaluation points
    for j = 1:nN                          % Loop over all rows of w
        % Construct solution y(x_EP) at all evaluation points x_EP
        phiVal = evaluatePhiJatX(x(j),xEP(i),eps);
        yEP(i) = yEP(i) + w(j)*phiVal;
    end
end
end
```

```
Not enough input arguments.

Error in evaluateRBFInterpolant (line 4)
h = x(2)-x(1);
```

Published with MATLAB® R2022a

Function computeBEulerNonLinSol

```
function [y,method] = computeBEulerNonLinSol(y0,f,dfdy,x)

method = 'Backward Euler Method';   % Define method for plot
h = x(2) - x(1);                    % Step size h
nSI = (x(end) - x(1))/h;           % Number of subintervals
y = zeros(nSI+1, 1);                      % y-array (approximate solution)
y(1) = y0;                         % Initial condition
eps = 1e-12;                        % Tolerance
maxIt = 150;                        % Maximum number of iterations
whichMethod = 1;                   % 1 - Newton's method
                                    % 2 - Predictor-corrector method

% Loop over all subintervals
for n = 1:nSI
    j = 1;                          % Iteration index
    ynp1 = zeros(1,maxIt);          % y_n+1^(j) initialization

    % Initial guess/predictor based on the Forward Euler method
    ynp1(j) = y(n)+h*f(x(n),y(n));

    if whichMethod == 1
        % Compute initial residual res = -F
        res = -(ynp1(j) - y(n) - h*f(x(n+1),ynp1(j)));

        % Loop until |res| <= eps or j >= maxIt
        while abs(res) > eps && j < maxIt
            % Compute first-order derivative of F = -res
            dFdy = 1 - h*dfdy(x(n+1),ynp1(j));

            % Compute delta from F' delta = res
            delta = res/dFdy;

            % Update ynp1, j, and residual
            ynp1(j+1) = ynp1(j) + delta;
            j = j + 1;
            res = -(ynp1(j) - y(n) - h*f(x(n+1),ynp1(j)));
        end
    y(n+1) = ynp1(j);
    end
end
```

```
Not enough input arguments.

Error in computeBEulerNonLinSol (line 5)
h = x(2) - x(1);                    % Step size h
```

Published with MATLAB® R2022a

Function evaluateCubicSpline

```
function [yEP,interpolant] = evaluateCubicSpline(x,y,xEP)

% Initialization
interpolant = "Cubic Spline";
h = x(2) - x(1);
nSI = (x(end)-x(1))/h;
n = nSI+1;
M = zeros(n,1);                     % Vector of M_i-values
yEP = zeros(1, length(xEP));
f = zeros(n,1);                     % Right-hand side vector f
g = zeros(n-2,1);                   % Right-hand side vector g
a = h/6*ones(n-2,1);                % Vector a of lower diagonal elements
b = 2*h/3*ones(n-2,1);              % Vector b of diagonal elements
c = h/6*ones(n-3,1);                % Vector c of upper diagonal elements
alpha = zeros(n-2,1);               % Vector alpha
beta = zeros(n-2,1);                % Vector beta
nLU = n-2;                          % Define nLU for n in LU-factorization

for i = 2:n-1                       % Loop over all rows of f from 2 to n-1
    f(i) = ((y(i+1)-y(i))/(x(i+1)-x(i))) - ((y(i)-y(i-1))/(x(i)-x(i-1))); % Calculate right-hand side vector f
end

f = f(2:n-1);       %Shrink f-vector
beta(1) = b(1);     %Define beta(1)
%LU Factorization:
for i = 2:nLU                                % Loop over remaining rows
    alpha(i) = a(i)/beta(i-1);                         % Compute alpha_i
    beta(i) = b(i)-(alpha(i)*c(i-1));     % Compute beta_i
end

% Employ forward substitution to solve Lg=f:
% Define g_1-value
g(1) = f(1);

for i = 2:nLU                               % Loop over remaining rows
    g(i) = f(i)-(alpha(i)*g(i-1));        % Compute g_i
end

% Employ back substitution to solve UM=g:
M(n-1) = g(nLU)/beta(nLU);                       % Define M_n-value

for i = (nLU-1):-1:1                          % Loop over remaining rows
    M(i+1) = (g(i)-(c(i)*M(i+2)))/(beta(i));     % Compute x_i
end

% Determine natural cubic spline:
for i=2:n                                % Loop over all end points of subinter.
    % Find indices of eval. points x_EP that are inside subinterval i-1
    indices = find(xEP >= x(i-1) & xEP <= x(i));

    % Evaluate natural cubic spline at x_EP inside subinterval i-1
    yEP(indices) = ((x(i)-xEP(indices)).^3 * M(i-1) + (xEP(indices)-x(i-1)).^3*M(i))/(6*(x(i)-x(i-1)))+(((x(i)-xEP(indices))...
        *y(i-1))+((xEP(indices)-x(i-1))*y(i)))/(x(i)-x(i-1)) - (1/6)*(x(i)-x(i-1))*((x(i)-xEP(indices))*M(i-1)+(xEP(indices)-x(i-1))*M(i));

end

end
```

```
Not enough input arguments.

Error in evaluateCubicSpline (line 6)
h = x(2) - x(1);
```

Published with MATLAB® R2022a

Function computeTrapezoidalRuleSol

```
function [TnSI,integration] = computeTrapezoidalRuleSol(f,x,y)
% Initialization
h = x(2)-x(1);
integrand = f(x,y);
integration = "Trapezoidal Rule";
% Calculate T_nSI according to the trapezoidal rule
TnSI = h/2 * (integrand(1) + 2*sum(integrand(2:end-1)) + integrand(end));
end
```

```
Not enough input arguments.

Error in computeTrapezoidalRuleSol (line 4)
h = x(2)-x(1);
```

Published with MATLAB® R2022a