

Governing Equations in Granular Flow

0.1 Governing Equations in Granular Flow

The governing equations for the mass and momentum balances can be written as:

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial z} = 0; \quad (1)$$

0.1.1 Momentum Balance in the x Direction

$$\rho \left(\frac{\partial u}{\partial t} + u \frac{\partial u}{\partial x} + v \frac{\partial u}{\partial z} \right) - \rho g \sin \theta + \frac{\partial \sigma_{xx}}{\partial x} + \frac{\partial \sigma_{xz}}{\partial z} = 0; \quad (2)$$

0.1.2 Momentum Balance in the z Direction

$$\rho \left(\frac{\partial v}{\partial t} + u \frac{\partial v}{\partial x} + v \frac{\partial v}{\partial z} \right) = \rho g \cos \theta + \frac{\partial \sigma_{xz}}{\partial x} + \frac{\partial \sigma_{zz}}{\partial z}; \quad (3)$$

where σ is the stress tensor of the material and g is the force due to gravity.

0.2 Non-Dimensionalization

Let L be the characteristic length scale of spatial variations along the slope (in the x direction) and H the typical flow thickness. The shallow-water approximation assumes that the ratio $\epsilon = H/L$ is small:

$$\epsilon = \frac{H}{L} \ll 1; \quad (4)$$

The dimensionless variables are then chosen as follows:

$$x = \tilde{x}L, \quad (5)$$

$$z = \tilde{z}H, \quad (6)$$

$$t = \tilde{t} \frac{L}{U}, \quad (7)$$

$$u = \tilde{u}U, \quad (8)$$

$$v = \tilde{v}\epsilon U, \quad (9)$$

$$\sigma_{xx} = \tilde{\sigma}_{xx}\rho g H, \quad (10)$$

$$\sigma_{zz} = \tilde{\sigma}_{zz}\rho g H, \quad (11)$$

$$\sigma_{xz} = \tilde{\sigma}_{xz}\rho g H, \quad (12)$$

where $U = \sqrt{gH}$ is a typical flow velocity along the slope.

Using these dimensionless variables, the governing equations become:

$$\frac{\partial \tilde{u}}{\partial \tilde{x}} + \frac{\partial \tilde{v}}{\partial \tilde{z}} = 0; \quad (13)$$

$$\epsilon \left(\frac{\partial \tilde{u}}{\partial \tilde{t}} + \tilde{u} \frac{\partial \tilde{u}}{\partial \tilde{x}} + \tilde{v} \frac{\partial \tilde{u}}{\partial \tilde{z}} \right) = \sin \theta + \epsilon \frac{\partial \tilde{\sigma}_{xx}}{\partial \tilde{x}} + \frac{\partial \tilde{\sigma}_{xz}}{\partial \tilde{z}}; \quad (14)$$

$$\epsilon^2 \left(\frac{\partial \tilde{v}}{\partial \tilde{t}} + \tilde{u} \frac{\partial \tilde{v}}{\partial \tilde{x}} + \tilde{v} \frac{\partial \tilde{v}}{\partial \tilde{z}} \right) = -\cos \theta + \epsilon \frac{\partial \tilde{\sigma}_{xz}}{\partial \tilde{x}} + \frac{\partial \tilde{\sigma}_{zz}}{\partial \tilde{z}}; \quad (15)$$

For $\epsilon \rightarrow 0$, the momentum equation in the vertical direction reduces to $\frac{\partial \tilde{\sigma}_{zz}}{\partial \tilde{z}} = \cos \theta$. Therefore, in the shallow-water approximation, the vertical normal stress is given by the hydrostatic balance. Integrating this equation with the zero-stress condition at the free surface gives:

$$\sigma_{zz} = -\rho g \cos \theta (h(x, t) - z); \quad (16)$$

0.2.1 Depth-Averaged Mass Conservation Equation

$$\frac{\partial h}{\partial t} = -\frac{\partial h \bar{u}}{\partial x}; \quad (17)$$

where \bar{u} is the average depth velocity and h is the thickness of the granular layer.

0.2.2 Depth-Averaged Momentum Conservation Equation

$$\frac{\partial h \bar{u}}{\partial t} + \alpha \frac{\partial h \bar{u}^2}{\partial x} = gh \cos \theta \left(\tan \theta - \frac{\tau_b}{\rho g h \cos \theta} - k \frac{\partial h}{\partial x} \right); \quad (18)$$

where τ_b is the basal shear stress and θ is the slope at which the granular fluid is flowing down.

0.2.3 Assumptions

These equations are derived under the following assumptions:

1. An incompressible medium.
2. The shallow-layer approximation.
3. An assumption on the shape of the velocity profile through the parameter α .
4. Proportionality between the normal stresses along the x and z directions, $\sigma_{xx} = k \sigma_{zz}$.

0.2.4 Friction Law

$$\tau_b = \mu_b \rho g h \cos \theta; \quad (19)$$

where $\rho g h \cos \theta$ is the basal normal stress and μ_b is an effective coefficient of friction between the flowing layer and the bottom.

$$\mu_b(\bar{u}, h) = \mu_1 + \frac{(\mu_2 - \mu_1)}{(2I_0 h \sqrt{\phi g h \cos \theta + 5d\bar{u}})} (5d\bar{u}); \quad (20)$$

The basal friction is simply written as $\mu_b = \mu(I_b)$, where I_b is the inertial number evaluated at the plane. To express I_b as a function of \bar{u} and h , we assume that the velocity profile is locally at equilibrium.

$$I_b = \frac{5d\bar{u}}{2h\sqrt{g\phi h \cos \theta}}; \quad (21)$$

0.2.5 Simplification of the Momentum Conservation Equation

After substituting the value of τ_b and μ_b , we get:

$$\frac{\partial h\bar{u}}{\partial t} + \alpha \frac{\partial h\bar{u}^2}{\partial x} = gh \cos \theta \left(\tan \theta - \mu_b - k \frac{\partial h}{\partial x} \right); \quad (22)$$

Further simplification:

$$\bar{u} \frac{\partial h}{\partial t} + h \frac{\partial \bar{u}}{\partial t} + \alpha \frac{\partial h\bar{u}^2}{\partial x} = gh \cos \theta \left(\tan \theta - \mu_b - k \frac{\partial h}{\partial x} \right); \quad (23)$$

0.3 PDEPE

$$\frac{\partial h}{\partial t} = -u \frac{\partial h}{\partial x} - h \frac{\partial u}{\partial x}; \quad (24)$$

After substituting the above equation in equation (22):

$$h \frac{\partial u}{\partial t} = -u \left(-u \frac{\partial h}{\partial x} - h \frac{\partial u}{\partial x} \right) - \alpha \frac{\partial h\bar{u}^2}{\partial x} + gh \cos \theta \left(\tan \theta - \mu_b - k \frac{\partial h}{\partial x} \right); \quad (25)$$

After substituting $\mu_b = \frac{\tau_b}{\rho g h \cos \theta}$ in the above equation and simplifying, we get:

$$h \frac{\partial u}{\partial t} = (u^2 \frac{\partial h}{\partial x} + uh \frac{\partial u}{\partial x}) - \alpha(u^2 \frac{\partial h}{\partial x} + 2uh \frac{\partial u}{\partial x}) + gh \cos \theta \left(\tan \theta - \frac{\tau_b}{\rho g h \cos \theta} - k \frac{\partial h}{\partial x} \right); \quad (26)$$

$$h \frac{\partial u}{\partial t} = gh \cos \theta \left(\tan \theta - \frac{\tau_b}{\rho g h \cos \theta} - k \frac{\partial h}{\partial x} \right) + u^2(1 - \alpha) \frac{\partial h}{\partial x} + uh(1 - 2\alpha) \frac{\partial u}{\partial x}; \quad (27)$$

PDEPE variables Case 1:

1. $c = h$;
2. $m = 0$;
3. $f = 0$;
4. $s = gh \cos \theta \left(\tan \theta - \frac{\tau_b}{\rho g h \cos \theta} - k \frac{\partial h}{\partial x} \right) + u^2(1 - \alpha) \frac{\partial h}{\partial x} + uh(1 - 2\alpha) \frac{\partial u}{\partial x}$;

$$\frac{\partial h}{\partial t} = -\frac{\partial h\bar{u}}{\partial x}; \quad (28)$$

$$c = 1; \quad s = -h \frac{\partial u}{\partial x} - u \frac{\partial h}{\partial x}; \quad f = 0; \quad m = 0; \quad (29)$$

0.4 Algorithms

0.4.1 Granular Flow Simulation Algorithm

The following algorithm describes the process of simulating granular flow using the PDE solver in MATLAB:

0.4.2 Function pdex2pde Algorithm

The following algorithm describes the function `pdex2pde` used in the MATLAB simulation to define the PDE system:

0.4.3 Function pdex2ic Algorithm

The following algorithm describes the function `pdex2ic` used in the MATLAB simulation to specify the initial conditions for the PDE system:

0.4.4 Function pdex2bc Algorithm

The following algorithm describes the function `pdex2bc` used in the MATLAB simulation to specify the boundary conditions for the PDE system:

0.5 Method 1: PDEPE Solver

The PDEPE solver in MATLAB is designed to solve partial differential equations (PDEs) of the form:

$$c(x, t, u, \frac{\partial u}{\partial x}) \frac{\partial u}{\partial t} = x^{-m} \frac{\partial}{\partial x} \left(x^m f(x, t, u, \frac{\partial u}{\partial x}) \right) + s(x, t, u, \frac{\partial u}{\partial x})$$

where $u(x, t)$ is the dependent variable, x and t are the independent variables, and m is a constant that specifies the symmetry of the problem. The functions c , f , and s are defined by the specific PDE system being solved.

0.5.1 Description of PDEPE Variables

For the equations provided:

1. c corresponds to the coefficient of $\frac{\partial u}{\partial t}$.
2. m represents the symmetry of the problem. In your case, $m = 0$ indicates no symmetry.
3. f is the flux term, which generally depends on u and its spatial derivative $\frac{\partial u}{\partial x}$.
4. s is the source term, which includes all other terms of the PDE that are independent of the time derivative.

Algorithm 1: Granular Flow Simulation

[1] **Initialization:** Set global variables:
 $h_{\text{initial}}, v_{x\text{initial}}, \text{bin}_h, \text{bint}, I_{\text{not}}, \mu_s, \mu_m, d, \theta, \phi, g, \alpha, k, h_{\text{initial}0}, v_{x\text{initial}0}, L_x$
Set simulation time step $ts = 10$
Set system properties and parameters:
 $g = 9.81$ (gravity)
 $\theta = 30$ (slope angle)
 $d = 0.001$
 $L_x = 1000 \cdot d$
Rheological model parameters: $I_{\text{not}} = 1$
 $\mu_s = 0.1$
 $\mu_m = 0.75$
 $\phi = 0.58$
 $\alpha = \frac{5}{4}$
 $k = 1$

Initial values: $\text{bin}_h = 100$
 $\text{bin}_t = 100$
 $h_{\text{initial}} = 39.d \cdot \text{ones}(1, \text{bin}_h)$
 $v_{x\text{initial}} = 5 \cdot \text{ones}(1, \text{bin}_h)$
 $h_{\text{initial}}(1) = 39 \cdot d$
 $v_{x\text{initial}}(1) = 5$
 $h_{\text{initial}0} = 39 \cdot d$
 $v_{x\text{initial}0} = 5$

Simulation:

Initialize spatial and time vectors: x and t
Initialize counter count = 1 and time = 0
Set ODE solver options with tolerances and maximum step size
Create output directory if it does not exist

while $time < 500$ **do**

Set symmetry parameter $m = 0$
Solve PDE using `pdepe` with functions `pdex2pde`, `pdex2ic`, `pdex2bc`, and options
if *Exception occurs* **then**

else
Display warning and exit loop
Extract solutions for velocity and height
Update count and time
Update h_{initial} and $v_{x\text{initial}}$ with latest values
Write data to file with updated filename
Plot and save the figure showing height and velocity

Algorithm 2: Function pdex2pde

[1] **Inputs:** x (spatial variable)
 t (time variable)
 u (vector of unknowns [velocity, height])
 $DuDx$ (vector of derivatives [velocity gradient, height gradient])

Global Variables: $I_{\text{not}}, \mu_s, \mu_m, d, \theta, \phi, g, \alpha, k$

Extract Variables:

$$\begin{aligned}v_x &= u(1) \quad (\text{velocity}) \\h &= u(2) \quad (\text{height}) \\ \frac{dv_x}{dy} &= DuDx(1) \quad (\text{velocity gradient}) \\ \frac{dh}{dy} &= DuDx(2) \quad (\text{height gradient})\end{aligned}$$

Compute Intermediate Quantities: $I = \frac{5 \cdot d \cdot v_x}{2 \cdot h \cdot \sqrt{\phi \cdot g \cdot h \cdot \cos(\theta)}}$

$$\mu_b = \mu_s + \frac{\mu_m - \mu_s}{1 + \frac{I}{I_{\text{not}}}}$$

Define PDEPE Coefficients:

For the height equation:

$$c1 = h$$

$$\begin{aligned}s1 &= g \cdot h \cdot \cos(\theta) \cdot (\tan(\theta) - \mu_b) - k \cdot g \cdot h \cdot \cos(\theta) \cdot \frac{dh}{dx} + v_x^2 \cdot (1 - \alpha) \cdot \frac{dh}{dx} + v_x \cdot h \cdot (1 - 2 \cdot \alpha) \cdot \frac{dv_x}{dx} \\F1 &= 0\end{aligned}$$

For the velocity equation:

$$C2 = 1$$

$$S2 = -h \cdot \frac{dv_x}{dx} - v_x \cdot \frac{dh}{dx}$$

$$F2 = 0$$

Outputs:

$$C = [C1; C2]$$

$$S = [S1; S2]$$

$$F = [F1; F2]$$

Algorithm 3: Function pdex2ic

[1] **Inputs:** x (spatial variable)

Global Variables: bin_h, L_x , vx_initial, h_initial, time

Compute Bin Index:

$$s = \frac{x - 0}{\frac{L_x}{\text{bin_h} - 1}} + 1$$

$$s = \text{round}(s) \quad (\text{bin index})$$

Extract Initial Values:

$$f_0 = h_{\text{initial}}(s) \quad (\text{initial height})$$

$$v_0 = vx_{\text{initial}}(s) \quad (\text{initial velocity})$$

Define Initial Condition Vector: $u_0 = [v_0; f_0]$

Output: u_0 (initial condition vector)

Algorithm 4: Function pdex2bc

[1] **Inputs:** x_l (left boundary position) u_l (solution vector at left boundary)
 x_r (right boundary position) u_r (solution vector at right boundary) t (time)
Global Variables: h_initial0, vx_initial0
Define Left Boundary Conditions: $p_l = [u_l(1) - vx_initial0; u_l(2) - h_initial0]$
 $q_l = [0; 0]$
Define Right Boundary Conditions:
 $p_r = [0; 0]$
 $q_r = [1; 1]$
Output: p_l (left boundary conditions)
 q_l (left boundary fluxes)
 p_r (right boundary conditions)
 q_r (right boundary fluxes)

0.5.2 PDEPE Variables in the Given Equations

Given the derived equations:

$$h \frac{\partial u}{\partial t} = gh \cos \theta \left(\tan \theta - \frac{\tau_b}{\rho g h \cos \theta} - k \frac{\partial h}{\partial x} \right) + u^2 (1 - \alpha) \frac{\partial h}{\partial x} + uh(1 - 2\alpha) \frac{\partial u}{\partial x}$$

This can be identified with the PDEPE form where:

$$\begin{aligned} c(x, t, u, \frac{\partial u}{\partial x}) &= h, \\ m &= 0, \\ f(x, t, u, \frac{\partial u}{\partial x}) &= 0, \\ s(x, t, u, \frac{\partial u}{\partial x}) &= gh \cos \theta \left(\tan \theta - \frac{\tau_b}{\rho g h \cos \theta} - k \frac{\partial h}{\partial x} \right) + u^2 (1 - \alpha) \frac{\partial h}{\partial x} + uh(1 - 2\alpha) \frac{\partial u}{\partial x}. \end{aligned}$$

For the second PDE:

$$\frac{\partial h}{\partial t} = -\frac{\partial h \bar{u}}{\partial x}$$

This corresponds to:

$$\begin{aligned} c(x, t, u, \frac{\partial u}{\partial x}) &= 1, \\ m &= 0, \\ f(x, t, u, \frac{\partial u}{\partial x}) &= 0 \\ s(x, t, u, \frac{\partial u}{\partial x}) &= -h \frac{\partial u}{\partial x} - u \frac{\partial h}{\partial x}. \end{aligned}$$

0.5.3 Explanation of p , q , f , s , and c

In the context of PDEPE, the variables are as follows:

- $c(x, t, u, \frac{\partial u}{\partial x})$: The coefficient of the time derivative $\frac{\partial u}{\partial t}$. It controls the scaling of the time evolution.
- m : A symmetry parameter where $m = 0$ denotes Cartesian coordinates, $m = 1$ denotes cylindrical symmetry, and $m = 2$ denotes spherical symmetry.
- $f(x, t, u, \frac{\partial u}{\partial x})$: The flux term, which typically depends on u and its spatial derivative $\frac{\partial u}{\partial x}$. It represents the flow of the quantity u .
- $s(x, t, u, \frac{\partial u}{\partial x})$: The source term, representing any additional sources or sinks in the system, not associated with the time derivative.

In your equations, these variables are explicitly defined as:

$$\text{Equation 1: } c = h, \quad m = 0, \quad f = 0, \quad s = gh \cos \theta \left(\tan \theta - \frac{\tau_b}{\rho g h \cos \theta} - k \frac{\partial h}{\partial x} \right) + u^2 (1 - \alpha) \frac{\partial h}{\partial x} + uh \quad (1)$$

$$\text{Equation 2: } c = 1, \quad m = 0, \quad f = -h\bar{u}, \quad s = 0.$$

0.6 MATLAB Code

```

1 clear all; close all; clc;
2 global h_initial vx_initial bin_h bin_t I_not mu_s mu_m d theta phi g
3 alpha k h_initial0 vx_initial0 L_x
4 ts = 10; % Reduce time step
5
6 %% System variables
7 g = 9.81;
8 theta = 25;
9
10 %% System properties
11 d = 0.001;
12 L_x = 1000*d;
13
14 %% Rheological model parameters
15 I_not = 1;
16 mu_s = 0.1;
17 mu_m = 0.75;
18 phi = 0.58;
19 alpha = 5/4; % Bagnold velocity profile
20 k = 1; % Neglecting normal stress differences
21
22 %% Initial values
23 bin_h = 100;
24 bin_t = 100;
25 h_initial = 39*d*ones(1, bin_h);
26 vx_initial = 5*ones(1, bin_h);
27 h_initial(1) = 39* d;
28 vx_initial(1) = 5;
29 h_initial0 = 39* d;
30 vx_initial0 = 5;
31
32 %% Initialization
y = linspace(0, L_x, bin_h);

```

```

33 t = linspace(0, ts, bin_t);
34
35 count = 1;
36 time = 0;
37
38 % ODE solver options
39 options = odeset('RelTol', 1e-4, 'AbsTol', 1e-6, 'MaxStep', 1e-2); %
    Adjusted tolerances and max step
40
41 output_dir = 'Time';
42 if ~exist(output_dir, 'dir')
43     status = mkdir(output_dir);
44     if status == 0
45         error('Failed to create directory: %s', output_dir);
46     else
47         disp(['Directory created: ', output_dir]);
48     end
49 else
50     disp(['Directory already exists: ', output_dir]);
51 end
52
53 %% Simulation loop
54 while time < 500
55     m = 0; % Symmetry parameter for PDEPE
56
57     % Solve the PDE
58     try
59         sol = pdepe(m, @pdex2pde, @pdex2ic, @pdex2bc, y, t, options);
60     catch ME
61         warning('Time integration has failed. Solution is available at
62             requested time points up to t=%2f.\nError: %s', time, ME.
63             message);
64         break;
65     end
66
67     % Extract solutions
68     velocity = sol(:,:,1);
69     height = sol(:,:,2);
70
71     count = count + 1;
72     time = time + ts;
73     disp(['Current simulation time: ', num2str(time)]);
74
75     %% Properties calculation
76     vx = velocity(end, :); % Update velocity, vx after time step ts
77     h = height(end, :);
78
79     %% Restricting f to be in between 0 to 1
80     h_initial = h;
81     vx_initial = vx;
82
83     % Update data
84     data = [y', vx', h'];
85
86     % Write data to file
87     filename = fullfile(output_dir, sprintf('properties-%05d.txt', count))
88         ; % Adjusted filename format

```

```

86 disp(['Writing data to file: ', filename]);
87 try
88     writematrix(data, filename, 'Delimiter', ' ');
89 catch ME
90     error('Failed to write file: %s\nError: %s', filename, ME.message)
91     ;
92 end
93
94 % Plot and save the figure
95 figure;
96 subplot(2, 1, 1);
97 plot(y, h, 'linewidth', 2);
98 title(sprintf('Height at time = %.2f', time));
99 xlabel('Position y');
100 ylabel('Height h');
101
102 subplot(2, 1, 2);
103 plot(y, vx, 'linewidth', 2);
104 title(sprintf('Velocity at time = %.2f', time));
105 xlabel('Position y');
106 ylabel('Velocity vx');
107 saveas(gcf, fullfile(output_dir, sprintf('plot-%05d.png', count)));
108 close
109 end

```

Boundary condition function of PDEPE

```

1 function [pl ,ql ,pr ,qr] = pdex2bc(xl ,ul ,xr ,ur ,t)
2 global time h_initial0 vx_initial0
3 pl = [ul(1) - vx_initial0; ul(2) - h_initial0];
4 % pl = [ul(1); ul(2)];
5 ql = [0; 0];
6
7 pr = [0; 0];
8 qr = [1; 1];
9 end

```

Initial condition Function of PDEPE

```

1 function u0 = pdex2ic(x)
2 global bin_h L_x vx_initial h_initial time
3
4 s = (x - 0) / ((L_x) / (bin_h - 1)) + 1;
5 s = round(s); % bin index
6
7 f0 = h_initial(s);
8 v0 = vx_initial(s);
9 u0 = [v0; f0];
10 end

```

PDEPE Function

```

1 function [C, F, S] = pdex2pde(x, t, u, DuDx)
2 global I_not mu_s mu_m d theta phi g alpha k
3
4 vx = u(1);
5 h = u(2);

```

```

6 dvx_dy = DuDx(1);
7 dh_dy = DuDx(2);
8
9 I = 5 * d * vx / (2 * h * sqrt(phi * g * h * cosd(theta)));
10
11 mu_b = mu_s + (mu_m - mu_s) / (1 + I_not / I);
12
13 C1 = h ;
14 S1 = g * h * cosd(theta) * (tand(theta) - mu_b) - k * g * h * cosd(
15 theta) * dh_dy+vx^2*(1-alpha)*dh_dy+vx*h*(1-2*alpha)*dvx_dy;
16 F1 = 0;
17
18 C2 = 1;
19 S2 = -h*dvx_dy-vx*dh_dy;
20 F2 = 0;
21
22 C = [C1; C2];
23 S = [S1; S2];
24 F = [F1; F2];
end

```

0.7 steady state code

```

1 clear all; close all; clc;
2 global h_initial V_0 I_not mu_s mu_m d phi g alpha k L_x
3
4 % Parameters
5 g = 9.81;
6
7 % System properties
8 d = 0.001;
9 L_x = 1000* d;
10
11 % Rheological model parameters
12 I_not = 0.434;
13 mu_s = tand(20.16);
14 mu_m = tand(37.65);
15 phi = 0.58;
16 alpha = 5/4; % Bagnold velocity profile
17 k = 1; % Neglecting normal stress differences
18
19 % Discretization parameters
20 bin_h = 100;
21 h_initial = 25 * d * ones(1, bin_h);
22 V_0 = 5*sqrt(g*d) * ones(1, bin_h);
23 m = h_initial .* V_0; % Mass per unit width
24
25 % Time-stepping parameters
26 dx = L_x / bin_h; % Step size in space
27
28 % Theta values to simulate
29 theta_values = [25, 22, 24, 26, 28, 30];
30
31 % Initialize figure

```

```

32 figure;
33 hold on;
34
35 % Color map for different theta values
36 colors = lines(length(theta_values));
37
38 % Loop over theta values
39 results = struct();
40 for t_idx = 1:length(theta_values)
41     theta = theta_values(t_idx); % Current theta in degrees
42
43     % Initialize variables
44     V_new = V_0;
45     h_new = h_initial;
46
47     % Initial calculation for rheological properties
48     I = 5 * d .* V_new ./ (2 .* h_new .* sqrt(phi * g .* h_new * cosd(
49         theta)));
50     mu_b = mu_s + (mu_m - mu_s) ./ (1 + I_not ./ I);
51
52     % ODE solving loop over the spatial domain
53     for i = 1:bin_h
54         if i == 1
55             % Skip update for the first point (initial condition)
56             continue;
57         end
58
59         % Define the ODE for the current spatial step
60         ode = @(x, V) V * cosd(theta) * (tand(theta) - mu_b(i)) ...
61             / (alpha * V^3 - cosd(theta) * m(i));
62
63         % Solve ODE using numerical method (e.g., ode45)
64         [x, V_temp] = ode45(@(x, V) ode(x, V), [0, dx], V_new(i-1));
65
66         % Update velocity
67         V_new(i) = V_temp(end);
68
69         % Update height using continuity equation
70         h_new(i) = (h_initial(i) * V_0(i)) / V_new(i);
71
72         % Recalculate I and mu_b for the next iteration
73         I(i) = 5 * d .* V_new(i) ./ (2 .* h_new(i) .* sqrt(phi * g .* 
74             h_new(i) * cosd(theta)));
75         mu_b(i) = mu_s + (mu_m - mu_s) ./ (1 + I_not ./ I(i));
76     end
77
78     % Store data for current theta
79     results(t_idx).theta = theta;
80     results(t_idx).V = V_new;
81     results(t_idx).h = h_new;
82
83     % Plot results for velocity
84     subplot(2, 1, 1);
85     plot(linspace(0, L_x, bin_h), V_new, 'LineWidth', 2, 'Color', colors(
        t_idx, :));
86     hold on;

```

```

86 % Plot results for height
87 subplot(2, 1, 2);
88 plot(linspace(0, L_x, bin_h), h_new, 'LineWidth', 2, 'Color', colors(
89     t_idx, :));
90 hold on;
91 end
92
93 % Customize velocity profile subplot
94 subplot(2, 1, 1);
95 xlabel('length(m)');
96 ylabel('Velocity V (m/s)');
97 title('Velocity Profile');
98 legend(arrayfun(@(t) sprintf('\theta = %d', t), theta_values, 'UniformOutput', false), ...
99     'Location', 'best');
100 grid on;
101
102 % Customize height profile subplot
103 subplot(2, 1, 2);
104 xlabel('length (m)');
105 ylabel('Height h (m)');
106 title('Height Profile');
107 legend(arrayfun(@(t) sprintf('\theta = %d', t), theta_values, 'UniformOutput', false), ...
108     'Location', 'best');
109 grid on;
110
111 % Save results to a file
112 save('flow_results.mat', 'results');
113 disp('Simulation complete. Results saved to flow_results.mat.');

```

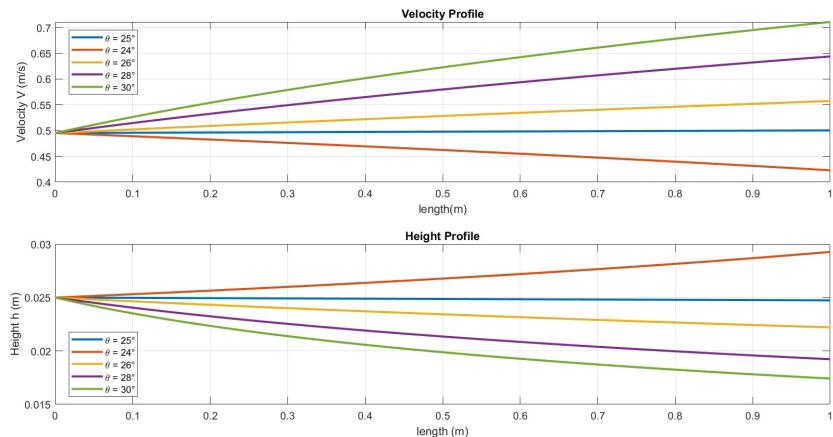


Figure 1: at $v = 5\sqrt{gd}$ and $h=25^*d$ steady state plot

Newton-Raphson Method

The Newton-Raphson method is an iterative technique used to find the roots of a real-valued function. Given a function $f(x)$ and its derivative $f'(x)$, the method uses the following iterative formula:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

where:

- x_n is the current approximation,
- x_{n+1} is the next approximation,
- $f(x_n)$ is the value of the function at x_n ,
- $f'(x_n)$ is the value of the derivative of the function at x_n .

Procedure

1. **Initial Guess**: Start with an initial guess x_0 for the root.
2. **Iteration**: Apply the iterative formula to compute x_1, x_2, \dots until convergence.
3. **Convergence**: The process is repeated until the difference between successive approximations is smaller than a predetermined tolerance level, i.e., $|x_{n+1} - x_n| < \epsilon$, where ϵ is a small positive number.

0.8 Depth-Averaged Mass Conservation and Momentum Conservation Equations

$$\frac{\partial h}{\partial t} = -\frac{\partial h \bar{u}}{\partial x}; \quad (30)$$

$$h \frac{\partial \bar{u}}{\partial t} = -\bar{u} \frac{\partial h}{\partial t} - \alpha \frac{\partial h \bar{u}^2}{\partial x} + gh \cos \theta \left(\tan \theta - \mu_b - k \frac{\partial h}{\partial x} \right); \quad (31)$$

0.8.1 Residual and Jacobian Matrix Formation

After discretizing the given equations with respect to x and t , we obtain two nonlinear equations at each grid point. Given $Nx = n$ and $Nt = m$, this results in a total of $2 \times n \times m$ nonlinear equations and $2 \times n \times m$ unknowns. These equations can be solved using an iterative method like the Newton-Raphson method, which requires forming a residual vector and a Jacobian matrix.

Residual Vector Formation

The residual vector \mathbf{R} is formed by evaluating the left-hand sides of the nonlinear equations at each grid point:

$$R_1(i, j) = h_{i,j+1} - h_{i,j-1} + 2\Delta t \left[h_{i,j} \left(\frac{u_{i+1,j} - u_{i-1,j}}{2\Delta x} \right) + u_{i,j} \left(\frac{h_{i+1,j} - h_{i-1,j}}{2\Delta x} \right) \right]$$

$$R_2(i, j) = \left(h_{i,j} \left(\frac{u_{i,j+1} - u_{i,j-1}}{2\Delta t} \right) \right) + \left(u_{i,j} \left(\frac{h_{i,j+1} - h_{i,j-1}}{2\Delta t} \right) \right) + \alpha \left[2h_{i,j}u_{i,j} \left(\frac{u_{i+1,j} - u_{i-1,j}}{2\Delta x} \right) + u_{i,j}^2 \left(\frac{h_{i+1,j} - h_{i-1,j}}{2\Delta x} \right) \right]$$

The residual vector \mathbf{R} is then constructed by stacking all $R_1(i, j)$ and $R_2(i, j)$ values for each $i = 1, \dots, n$ and $j = 1, \dots, m$.

Jacobian Matrix Formation

The Jacobian matrix \mathbf{J} consists of partial derivatives of the residuals with respect to the unknown variables. For each residual $R_1(i, j)$ and $R_2(i, j)$, the elements of the Jacobian matrix are calculated as:

$$J_{(i,j),(k,l)} = \frac{\partial R_1(i, j)}{\partial h_{k,l}}, \quad \frac{\partial R_1(i, j)}{\partial u_{k,l}}, \quad \frac{\partial R_2(i, j)}{\partial h_{k,l}}, \quad \frac{\partial R_2(i, j)}{\partial u_{k,l}}$$

The Jacobian matrix \mathbf{J} has a size of $2nm \times 2nm$, and its elements are populated by the derivatives of the residuals with respect to the unknowns. The differentiation is performed with respect to the variables $h_{i,j}$ and $u_{i,j}$.

Differentiation of the Equations

Differentiating the first equation $R_1(i, j)$ with respect to $h_{k,l}$ and $u_{k,l}$:

$$\frac{\partial R_1(i, j)}{\partial h_{k,l}} = \begin{cases} 1 & \text{if } k = i, l = j + 1 \text{ or } l = j - 1 \\ -\frac{2\Delta t}{2\Delta x} \cdot u_{i,j} & \text{if } k = i + 1 \text{ or } k = i - 1 \\ \frac{2\Delta t}{2\Delta x} \cdot u_{i,j} & \text{if } k = i \text{ and } l = j \\ 0 & \text{otherwise} \end{cases}$$

$$\frac{\partial R_1(i, j)}{\partial u_{k,l}} = \begin{cases} -\frac{2\Delta t}{2\Delta x} \cdot h_{i,j} & \text{if } k = i + 1 \text{ or } k = i - 1 \\ \frac{2\Delta t}{2\Delta x} \cdot h_{i,j} & \text{if } k = i \text{ and } l = j \\ 0 & \text{otherwise} \end{cases}$$

Differentiating the second equation $R_2(i, j)$ with respect to $h_{k,l}$ and $u_{k,l}$ involves applying the product rule due to the terms with both $h_{i,j}$ and $u_{i,j}$:

$$\frac{\partial R_2(i, j)}{\partial h_{k,l}} = (\text{similar process as above, involving each term of } R_2(i, j))$$

$$\frac{\partial R_2(i, j)}{\partial u_{k,l}} = (\text{similar process as above, involving each term of } R_2(i, j))$$

The differentiation is done term by term, considering the dependencies on $h_{i,j}$ and $u_{i,j}$ and applying the chain rule where necessary.

Final System of Equations

Once the residual vector \mathbf{R} and Jacobian matrix \mathbf{J} are formed, the Newton-Raphson update is applied iteratively:

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \mathbf{J}^{-1}\mathbf{R}$$

where \mathbf{x} is the vector of unknowns $h_{i,j}$ and $u_{i,j}$.

0.9 Boundary conditions

0.9.1 Boundary condition at $x=0$

$$v = v_{initial} \quad (32)$$

$$h = h_{initial} \quad (33)$$

0.9.2 Boundary condition at $x=L$

$$\frac{\partial v}{\partial x} = 0; \quad (34)$$

$$\frac{\partial h}{\partial x} = 0; \quad (35)$$

Algorithm 5: Newton-Raphson Update for Boundary Conditions and Residuals Calculation

Input: Matrices h, u ; Integers n, m ; Scalars $\Delta t, \Delta x, \alpha, g, \phi, \theta, \mu_{b1}, \mu_{b2}, I0, d, k, h_{bc}, u_{bc}$

Output: Updated matrices h and u

Initialize:

$size \leftarrow 2 \cdot n \cdot m$

$F \leftarrow \mathbf{0}_{size}$

$J \leftarrow$ Jacobian matrix for boundary conditions and residuals

$delta_X \leftarrow$ Vector of Newton-Raphson updates

$x \leftarrow$ Vector of spatial coordinates for plotting

while Convergence criteria not met **do**

Calculate residuals F

Calculate Jacobian matrix J

$delta_X \leftarrow -J \cdot F$

for $i \leftarrow 1$ to n **do**

for $j \leftarrow 1$ to m **do**

$h(i, j) \leftarrow h(i, j) + delta_X[(i - 1) \cdot m + j]$

$u(i, j) \leftarrow u(i, j) + delta_X[n \cdot m + (i - 1) \cdot m + j]$

Display 'Updated h:'

Display h

Display 'Updated u:'

Display u

Plot h and u :

- Create a figure with two subplots.
- Plot h vs. x in the first subplot.
- Plot u vs. x in the second subplot.
- Add labels, titles, and legends as needed.

Check for convergence criteria.

return h, u

Algorithm 6: Algorithm for Boundary Conditions and Residuals Calculation

Input: Integers n, m ; Matrices h, u ; Scalars $h_{bc}, u_{bc}, d, g_\phi \cos \theta, I0, \mu_{b1}, \mu_{b2}, k, g, \alpha, \theta, \Delta t, \Delta x$

Output: Vector F of residuals

Initialize:

$size \leftarrow 2 \cdot n \cdot m$

$F \leftarrow \mathbf{0}_{size}$

$idx \leftarrow 1$

for $i \leftarrow 1$ **to** n **do**

for $j \leftarrow 1$ **to** m **do**

if $i == 1$ **then**

$h_{i-1} \leftarrow h_{bc}$

$u_{i-1} \leftarrow u_{bc}$

else

$h_{i-1} \leftarrow h(i-1, j)$

$u_{i-1} \leftarrow u(i-1, j)$

if $i == n$ **then**

$h_{i+1} \leftarrow h(i, j)$

$u_{i+1} \leftarrow u(i, j)$

else

$h_{i+1} \leftarrow h(i+1, j)$

$u_{i+1} \leftarrow u(i+1, j)$

if $j == 1$ **then**

$h_{j-1} \leftarrow h_{bc}$

$u_{j-1} \leftarrow u_{bc}$

else

$h_{j-1} \leftarrow h(i, j-1)$

$u_{j-1} \leftarrow u(i, j-1)$

if $j == m$ **then**

$h_{j+1} \leftarrow h(i, j)$

$u_{j+1} \leftarrow u(i, j)$

else

$h_{j+1} \leftarrow h(i, j+1)$

$u_{j+1} \leftarrow u(i, j+1)$

$h_i \leftarrow h(i, j)$

$u_i \leftarrow u(i, j)$

$I_b \leftarrow \frac{5 \cdot d \cdot u_i}{2 \cdot h_i \cdot \sqrt{g_\phi \cos \theta \cdot h_i}}$

$\mu_b \leftarrow \mu_{b1} + \left(\frac{\mu_{b2} - \mu_{b1}}{I0/I_b} \right) + 1$

$F[idx] \leftarrow h_{j+1} - h_{j-1} + 2 \cdot \Delta t \cdot \left(\frac{u_{j+1} - 2 \cdot u_j + u_{j-1}}{\Delta x^2} + \text{RHS term} \right)$

$F[idx + 1] \leftarrow u_{j+1} - u_{j-1} + \Delta t \cdot \left(\frac{g}{\Delta x} \cdot (h_{j+1} - h_{j-1}) - \text{RHS term} \right)$

$idx \leftarrow idx + 2$

return F

Algorithm 7: Jacobian Calculation

Input: Matrices h, u ; Integers n, m ; Scalars $\Delta t, \Delta x, \alpha, g, \phi, \theta, \mu_{b1}, \mu_{b2}, I0, d, k, h_{bc}, u_{bc}$

Output: Jacobian matrix J

Initialize:

$J \leftarrow$ sparse matrix of size $2nm \times 2nm$

$g_\phi \cos \theta \leftarrow g \cdot \phi \cdot \cos(\theta)$

$idx \leftarrow 1$

for $i \leftarrow 1$ **to** n **do**

for $j \leftarrow 1$ **to** m **do**

$h_i \leftarrow h(i, j)$

$u_i \leftarrow u(i, j)$

$I_b \leftarrow \frac{5 \cdot d \cdot u_i}{2 \cdot h_i \cdot \sqrt{g_\phi \cos \theta \cdot h_i}}$

$\mu_b \leftarrow \mu_{b1} + \left(\frac{\mu_{b2} - \mu_{b1}}{I0/I_b} \right) + 1$

$dF1_dh_{ij} \leftarrow 2 \cdot \Delta t \cdot \left(\frac{u_{i+1} - u_{i-1}}{2 \cdot \Delta x} \right)$

$dF1_dh_{ij-1} \leftarrow -1$

$dF1_dh_{ij+1} \leftarrow 1$

$dF1_dh_{i-1j} \leftarrow -2 \cdot \Delta t \cdot \left(\frac{u_i}{2 \cdot \Delta x} \right)$

$dF1_du_{ij} \leftarrow 2 \cdot \Delta t \cdot \left(\frac{h_{i+1} - h_{i-1}}{2 \cdot \Delta x} \right)$

$J(idx, idx) \leftarrow dF1_dh_{ij}$

$J(idx, idx - 1) \leftarrow dF1_dh_{ij-1}$

$J(idx, idx + 1) \leftarrow dF1_dh_{ij+1}$

$J(idx, idx - m) \leftarrow dF1_dh_{i-1j}$

$J(idx, idx + m) \leftarrow dF1_dh_{i+1j}$

$J(idx, idx + n \cdot m) \leftarrow dF1_du_{ij}$

$idx \leftarrow idx + 1$

$dF2_dh_{ij} \leftarrow \frac{u_i \cdot (u_{i+1} - u_{i-1})}{2 \cdot \Delta t}$

$dF2_du_{ij} \leftarrow \frac{h_{j+1} - h_{j-1}}{2 \cdot \Delta t}$

$dF2_du_{i-1j} \leftarrow -\frac{h_{j+1} - h_{j-1}}{2 \cdot \Delta t}$

$dF2_du_{i+1j} \leftarrow -\frac{h_{j+1} - h_{j-1}}{2 \cdot \Delta t}$

$dF2_dh_{j-1} \leftarrow -\frac{u_i^2}{2 \cdot \Delta t}$

$dF2_dh_{j+1} \leftarrow -\frac{u_i^2}{2 \cdot \Delta t}$

$dF2_du_{i-1j} \leftarrow -\frac{h_{j+1} - h_{j-1}}{2 \cdot \Delta x}$

$dF2_du_{i+1j} \leftarrow -\frac{h_{j+1} - h_{j-1}}{2 \cdot \Delta x}$

$J(idx, idx) \leftarrow dF2_dh_{ij}$

$J(idx, idx + n \cdot m) \leftarrow dF2_du_{ij}$

$J(idx, idx - m) \leftarrow dF2_dh_{i-1j}$

$J(idx, idx + m) \leftarrow dF2_dh_{i+1j}$

$J(idx, idx - n \cdot m - m) \leftarrow dF2_du_{i-1j}$

$J(idx, idx - n \cdot m + m) \leftarrow dF2_du_{i+1j}$

$idx \leftarrow idx + 1$

return J

0.10 MATLAB Code

```
1 % Parameters
2 phi = 0.58;
3 alpha = 5/4;
4 g = 9.81;
5 theta = 25;
6 mu_b1 = tand(20.16);
7 mu_b2 = tand(37.65);
8 I0 = 0.434;
9 d = 0.001;
10 k = 1.0;
11 L = 5000 * d;
12 Nx = 50;
13 delta_x = L / (Nx - 1);
14 x = linspace(0, L, Nx);
15 T = 10;
16 Nt = 100;
17 delta_t = T / (Nt - 1);
18 t = linspace(0, T, Nt);
19
20 n = Nx;
21 m = Nt;
22 h = 39 * d * ones(n, m); % Initial values of h
23 u = 5 * ones(n, m); % Initial values of u
24 h_bc = 39 * d;
25 u_bc = 5;
26
27 % Calculate residuals
28 F = calculate_residuals(h, u, n, m, delta_t, delta_x, alpha, g, phi, theta,
   , mu_b1, mu_b2, I0, d, k, h_bc, u_bc);
29
30 % Calculate Jacobian matrix
31 J = calculate_jacobian(h, u, n, m, delta_t, delta_x, alpha, g, phi, theta,
   mu_b1, mu_b2, I0, d, k, h_bc, u_bc);
32
33 % Newton-Raphson update
34 delta_X = -J^(-1) * F;
35
36
37 for i = 1:n-1
38     for j = 1:m
39         h(i+1, j) = h(i, j) + delta_X((i - 1) * m + j);
40         u(i+1, j) = u(i, j) + delta_X(n * m + (i - 1) * m + j);
41     end
42 end
43
44 % Display updated h and u
45 disp('Updated h:');
46
47 disp('Updated u:');
48 disp(u);
49
50 % After the loop for updating h and u:
51
52 % Specify the directory to save the plots
```

```

53 output_dir = 'Time';
54
55 % Create the folder if it doesn't exist
56 if ~exist(output_dir, 'dir')
57     mkdir(output_dir);
58 end
59
60 % Loop through all time steps (from t = 1 to T)
61 for t_idx = 1:m % m is the number of time steps
62     % Extract the appropriate column for the current time step
63     figure;
64
65     % Plot h vs x
66     subplot(2, 1, 1);
67     plot(x, h(:, t_idx));
68     xlabel('x');
69     ylabel('h');
70     title(['h vs x at t = ' num2str(t(t_idx)) ' sec']);
71
72     % Plot u vs x
73     subplot(2, 1, 2);
74     plot(x, u(:, t_idx));
75     xlabel('x');
76     ylabel('u');
77     title(['u vs x at t = ' num2str(t(t_idx)) ' sec']);
78
79     % Save the figure in the specified folder
80     saveas(gcf, fullfile(output_dir, ['h_u_t' num2str(t(t_idx)) '.png']));
81         % Save as PNG
82 end

```

Calculating Residual Matrix Function

```

1 function F = calculate_residuals(h, u, n, delta_t, delta_x, alpha, g,
2     phi, theta, mu_b1, mu_b2, I0, d, k, h_bc, u_bc)
3 F = zeros(2*n*n, 1);
4 g_phi_cos_theta = g * phi * cosd(theta);
5 idx = 1;
6 for i = 1:n
7     for j = 1:n
8         if i-1 == 0
9             h_im1 = h_bc;
10            u_im1 = u_bc;
11        else
12            h_im1 = h(i-1, j);
13            u_im1 = u(i-1, j);
14        end
15        if i+1 == n+1
16            h_ip1 = h(i, j);
17            u_ip1 = u(i, j);
18        else
19            h_ip1 = h(i+1, j);
20            u_ip1 = u(i+1, j);
21        end
22        if j-1 == 0
23            h_jm1 = h_bc;

```

```

24         else
25             h_jm1 = h(i, j-1);
26             u_jm1 = u(i, j-1);
27         end
28         if j+1 == n+1
29             h_jp1 = h(i, j);
30             u_jp1 = u(i, j);
31         else
32             h_jp1 = h(i, j+1);
33             u_jp1 = u(i, j+1);
34         end
35
36         hi = h(i,j);
37         ui = u(i,j);
38         Ib = (5 * d * ui) / (2 * hi * sqrt(g_phi_cos_theta * hi));
39         mu_b = mu_b1 + (mu_b2 - mu_b1) / (I0 / Ib)+1;
40
41         F(idx) = h_jp1 - h_jm1 + 2*delta_t*(hi*(u_ip1 - u_im1)/(2*
42             delta_x)) + 2*delta_t*(ui*(h_ip1 - h_im1)/(2*delta_x));
43         idx = idx + 1;
44
45         F(idx) = hi*(u_jp1 - u_jm1)/(2*delta_t) + ui*(h_jp1 - h_jm1)
46             /(2*delta_t) ...
47                 + alpha*(2*hi*ui*(u_ip1 - u_im1)/(2*delta_x) + ui^2*(
48                     h_ip1 - h_im1)/(2*delta_x)) ...
49                     - g*hi*cosd(theta)*(tand(theta) - mu_b - k*(h_ip1 -
50                         h_im1)/(2*delta_x));
51         idx = idx + 1;
52     end
53 end

```

Calculating Jacobian Matrix Function

```

1 function J = calculate_jacobian(h, u, n, delta_t, delta_x, alpha, g,
2     phi, theta, mu_b1, mu_b2, I0, d, k, h_bc, u_bc)
3 J = zeros(2*n*n, 2*n*n);
4 g_phi_cos_theta = g * phi * cosd(theta);
5 idx = 1;
6 for i = 1:n
7     for j = 1:n
8         if i-1 == 0
9             h_im1 = h_bc;
10            u_im1 = u_bc;
11         else
12             h_im1 = h(i-1, j);
13             u_im1 = u(i-1, j);
14         end
15         if i+1 == n+1
16             h_ip1 = h(i, j);
17             u_ip1 = u(i, j);
18         else
19             h_ip1 = h(i+1, j);
20             u_ip1 = u(i+1, j);
21         end
22         if j-1 == 0
23             h_jm1 = h_bc;

```

```

23         u_jm1 = u_bc;
24     else
25         h_jm1 = h(i, j-1);
26         u_jm1 = u(i, j-1);
27     end
28     if j+1 == n+1
29         h_jp1 = h(i, j);
30         u_jp1 = u(i, j);
31     else
32         h_jp1 = h(i, j+1);
33         u_jp1 = u(i, j+1);
34     end
35
36     hi = h(i,j);
37     ui = u(i,j);
38     Ib = (5 * d * ui) / (2 * hi * sqrt(g_phi_cos_theta * hi));
39     mu_b = mu_b1 + ((mu_b2 - mu_b1) / (I0/Ib))+1;
40
41 % Calculate partial derivatives for F1
42 dF1_dhij = 2*delta_t*(u_ip1 - u_im1)/(2*delta_x);
43 dF1_dhijm1 = -1;
44 dF1_dhijp1 = 1;
45 dF1_dhim1j = -2*delta_t*(ui/(2*delta_x));
46 dF1_dhip1j = 2*delta_t*(ui/(2*delta_x));
47 dF1_duij = 2*delta_t*(h_ip1 - h_im1)/(2*delta_x);
48 dF1_duim1j = -2*delta_t*(hi/(2*delta_x));
49 dF1_duip1j = 2*delta_t*(hi/(2*delta_x));
50
51 % Fill in Jacobian for F1
52 J(idx, (i-1)*n + j) = dF1_dhij;
53 if j-1 > 0
54     J(idx, (i-1)*n + j-1) = dF1_dhijm1;
55 end
56 if j+1 <= n
57     J(idx, (i-1)*n + j+1) = dF1_dhijp1;
58 end
59 if i-1 > 0
60     J(idx, (i-2)*n + j) = dF1_dhim1j;
61 end
62 if i+1 <= n
63     J(idx, i*n + j) = dF1_dhip1j;
64 end
65 J(idx, n*n + (i-1)*n + j) = dF1_duij;
66 if i-1 > 0
67     J(idx, n*n + (i-2)*n + j) = dF1_duim1j;
68 end
69 if i+1 <= n
70     J(idx, n*n + i*n + j) = dF1_duip1j;
71 end
72 idx = idx + 1;
73
74 % Calculate partial derivatives for F2
75 dF2_dhij = (u_jp1 - u_jm1)/(2*delta_t) + alpha*(2*u(i,j)*(
76     u_ip1 - u_im1)/(2*delta_x) + u(i,j)^2/(2*delta_x)) - g*cosd
77     (theta)*(-mu_b - k*(h_ip1 - h_im1)/(2*delta_x));
    dF2_dhijm1 = -u(i,j)/(2*delta_t);
    dF2_dhijp1 = u(i,j)/(2*delta_t);

```

```

78      dF2_dhim1j = -alpha*(2*u(i,j)*(u_ip1 - u_im1)/(2*delta_x) + u(
79          i,j)^2/(2*delta_x));
80      dF2_dhip1j = alpha*(2*u(i,j)*(u_ip1 - u_im1)/(2*delta_x) + u(i
81          ,j)^2/(2*delta_x));
82      dF2_duij = hi*(h_jp1 - h_jm1)/(2*delta_t) + 2*alpha*(hi*(u_ip1
83          - u_im1)/(2*delta_x));
84      dF2_duijm1 = -hi/(2*delta_t);
85      dF2_duijp1 = hi/(2*delta_t);
86      dF2_duim1j = -alpha*2*hi*u(i,j)/(2*delta_x);
87      dF2_duip1j = alpha*2*hi*u(i,j)/(2*delta_x);

88      % Fill in Jacobian for F2
89      J(idx, (i-1)*n + j) = dF2_dhij;
90      if j-1 > 0
91          J(idx, (i-1)*n + j-1) = dF2_dhijm1;
92      end
93      if j+1 <= n
94          J(idx, (i-1)*n + j+1) = dF2_dhijp1;
95      end
96      if i-1 > 0
97          J(idx, (i-2)*n + j) = dF2_dhim1j;
98      end
99      if i+1 <= n
100         J(idx, i*n + j) = dF2_dhip1j;
101     end
102     J(idx, n*n + (i-1)*n + j) = dF2_duij;
103     if j-1 > 0
104         J(idx, n*n + (i-1)*n + j-1) = dF2_duijm1;
105     end
106     if j+1 <= n
107         J(idx, n*n + (i-1)*n + j+1) = dF2_duijp1;
108     end
109     if i-1 > 0
110         J(idx, n*n + (i-2)*n + j) = dF2_duim1j;
111     end
112     if i+1 <= n
113         J(idx, n*n + i*n + j) = dF2_duip1j;
114     end
115     idx = idx + 1;
116 end
end

```

0.11 Plots at different value of h and v at theta equal to 25

0.11.1 newton rapson plot

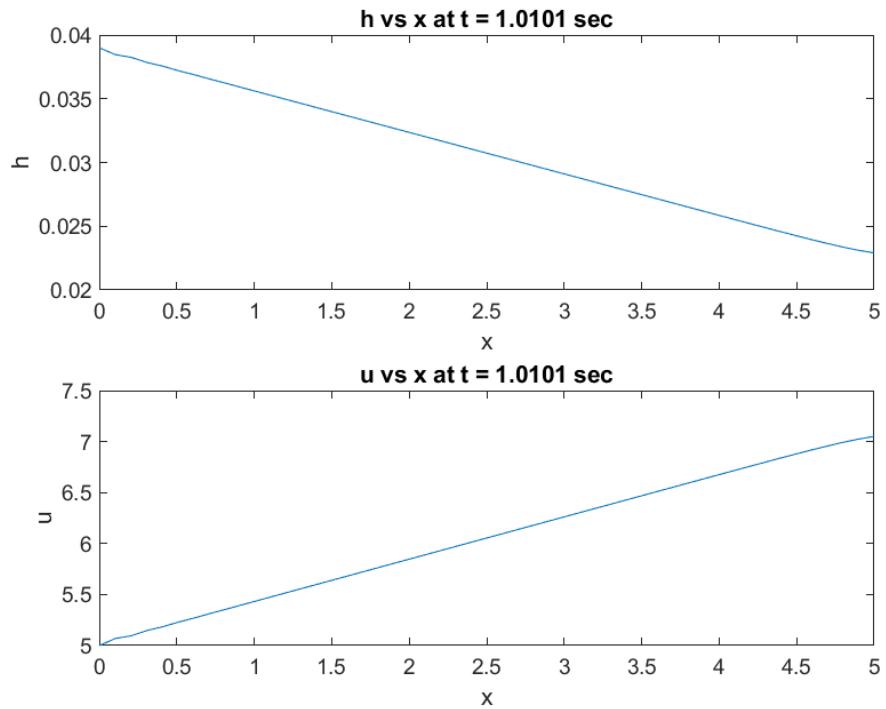


Figure 2: at $v=5$ and $h=39*d$

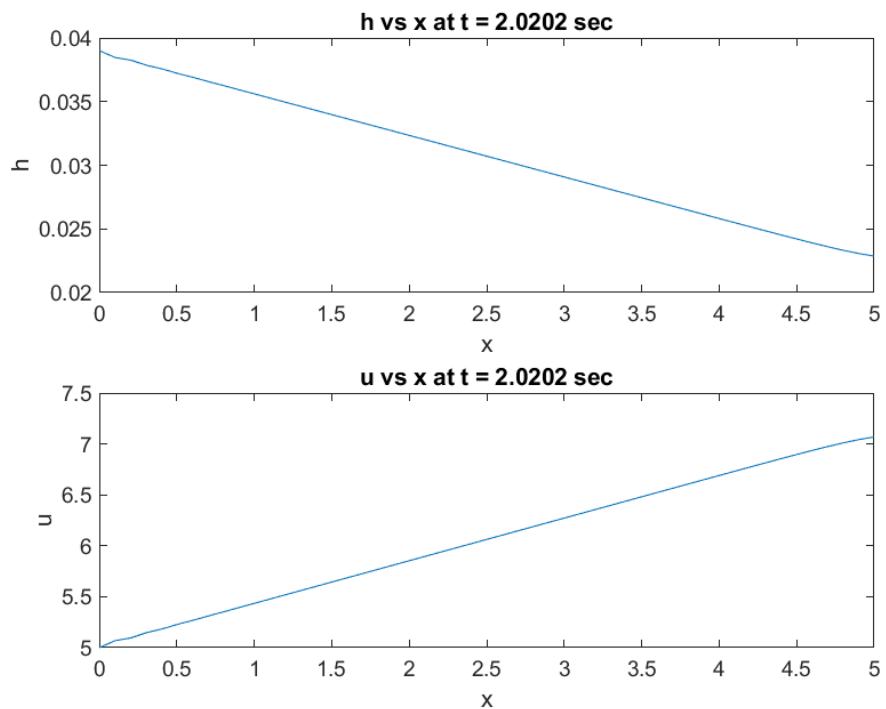


Figure 3: at $v=5$ and $h=39*d$

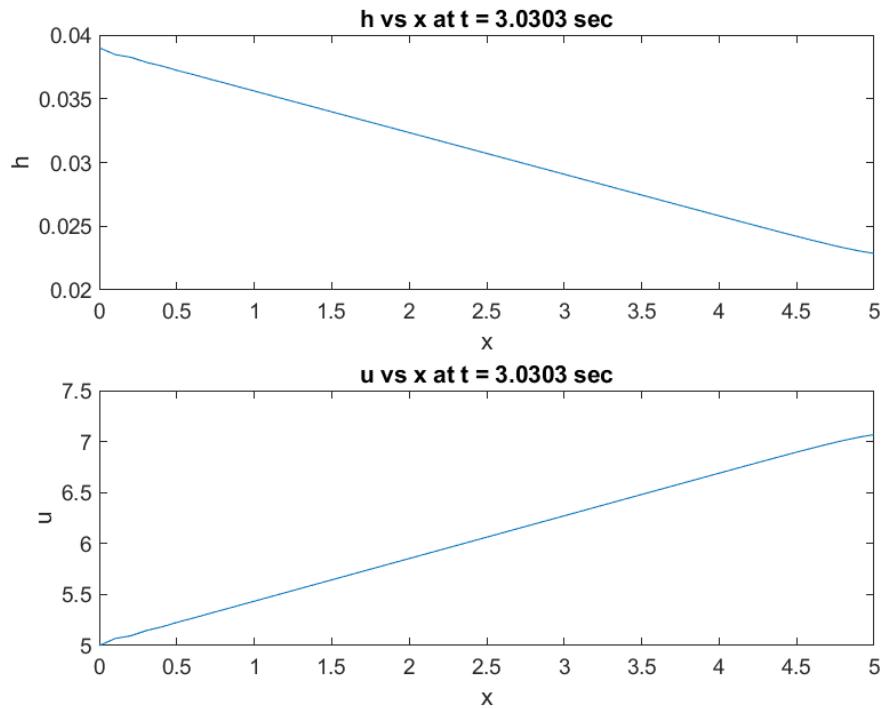


Figure 4: at $v=5$ and $g=39*d$

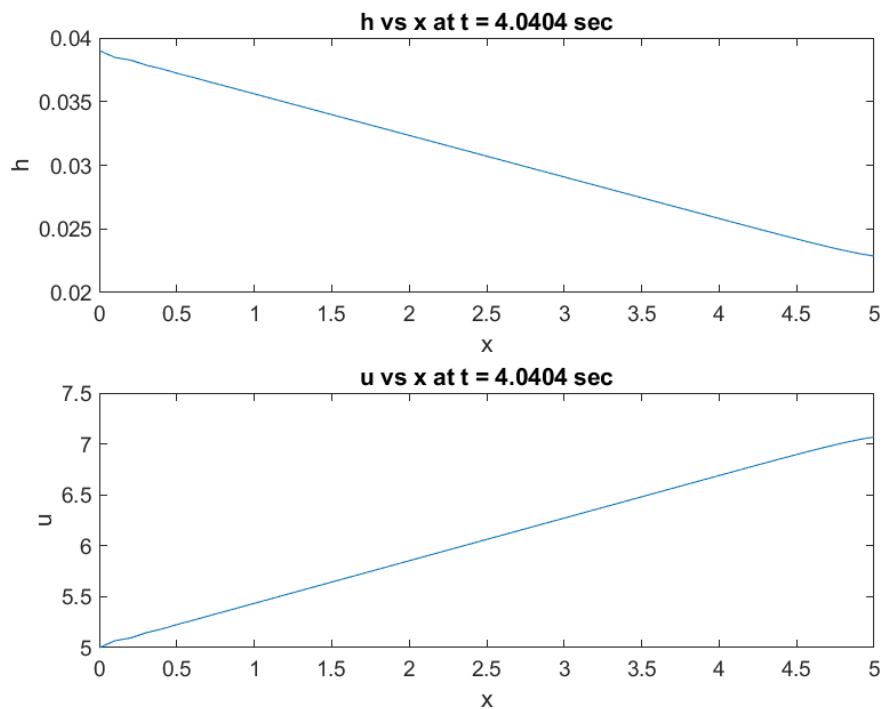


Figure 5: at $v=5$ and $h=39*d$

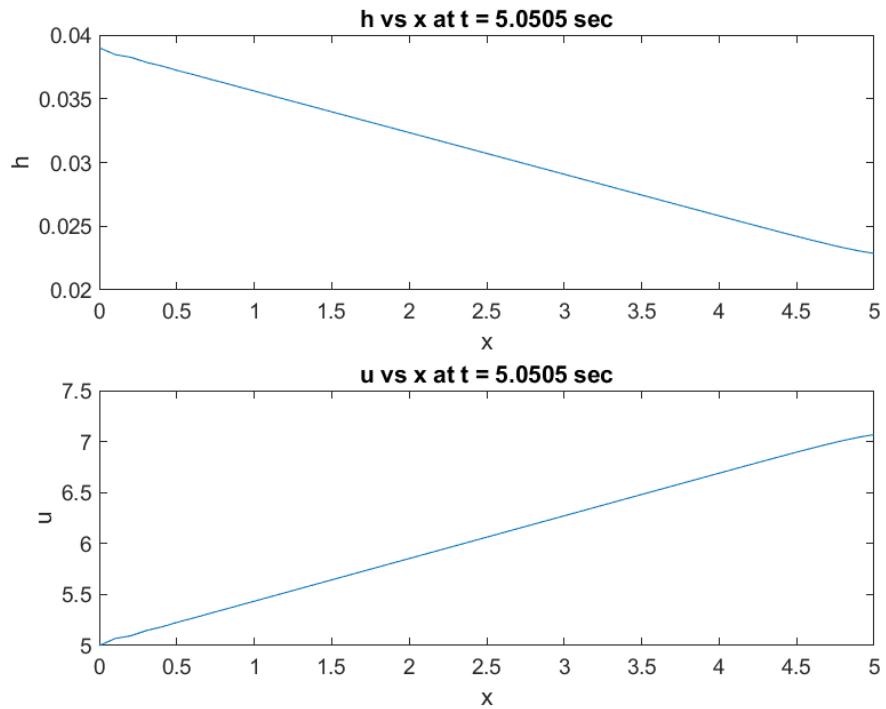


Figure 6: at v=5 and h=39*d

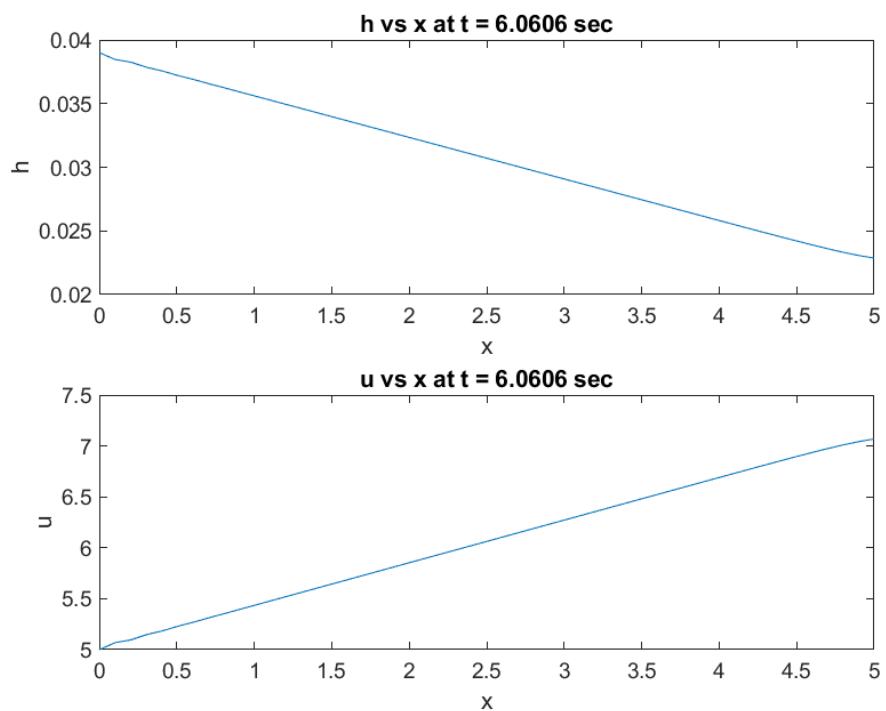


Figure 7: at v=5 and h=39*d

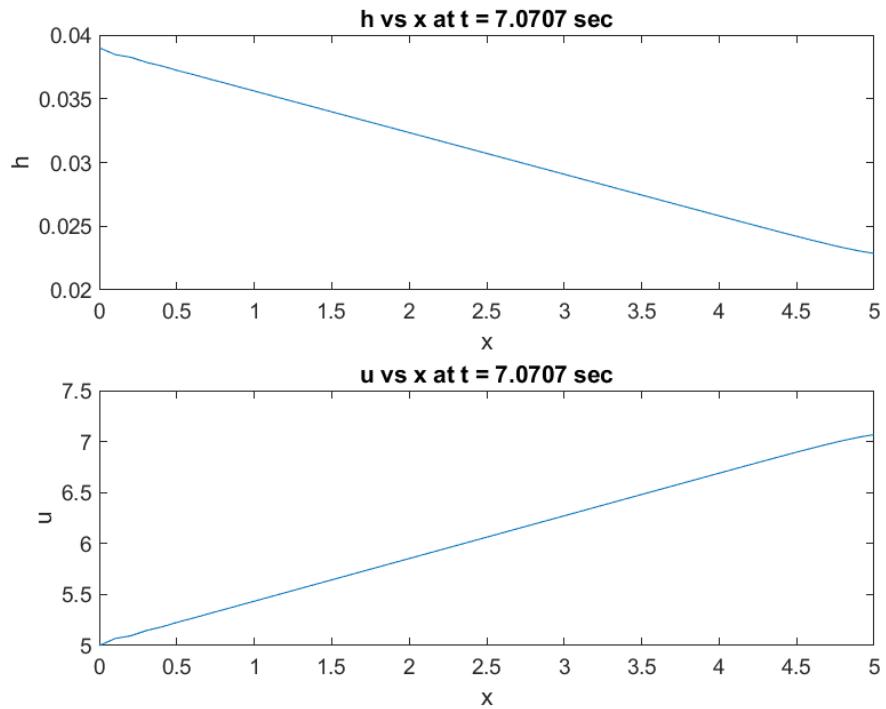


Figure 8: at $v=5$ and $h=39*d$

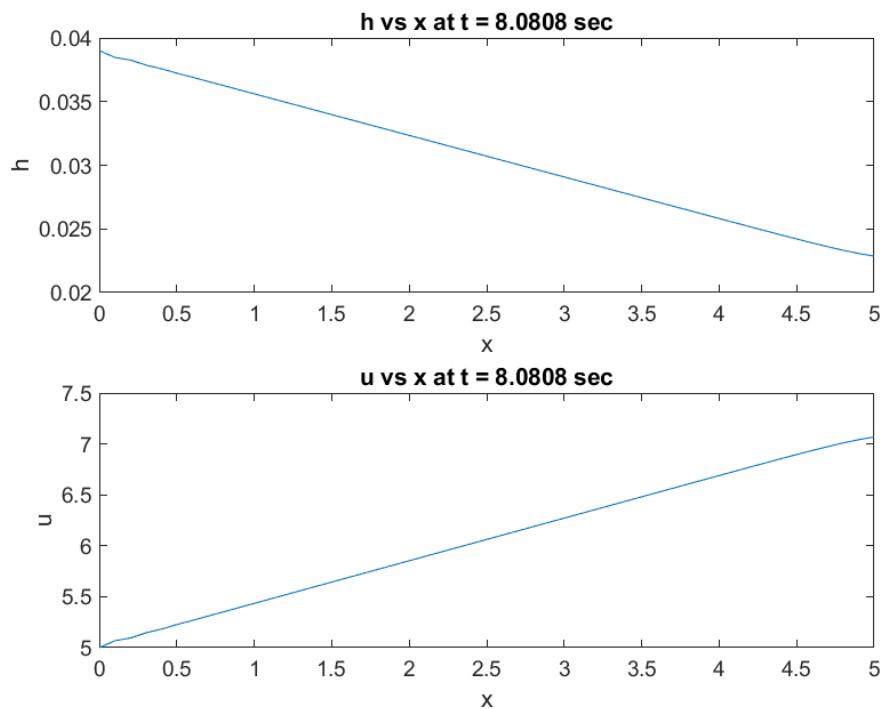


Figure 9: at $v=5$ and $h=39*d$

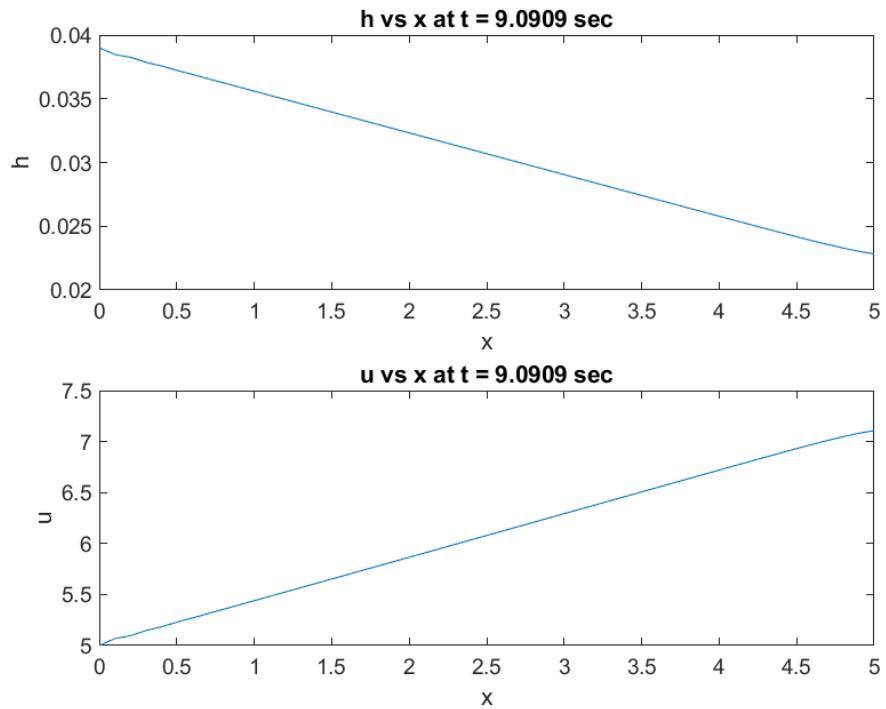


Figure 10: at v=5 and h=39*d

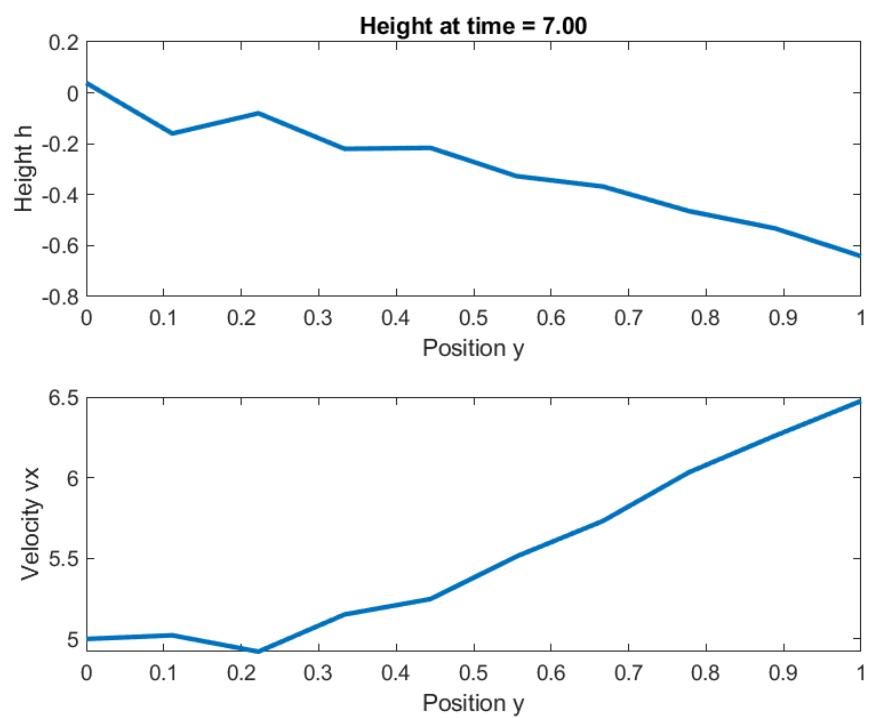


Figure 11: at v=5 and h=39*d PDEPE plots

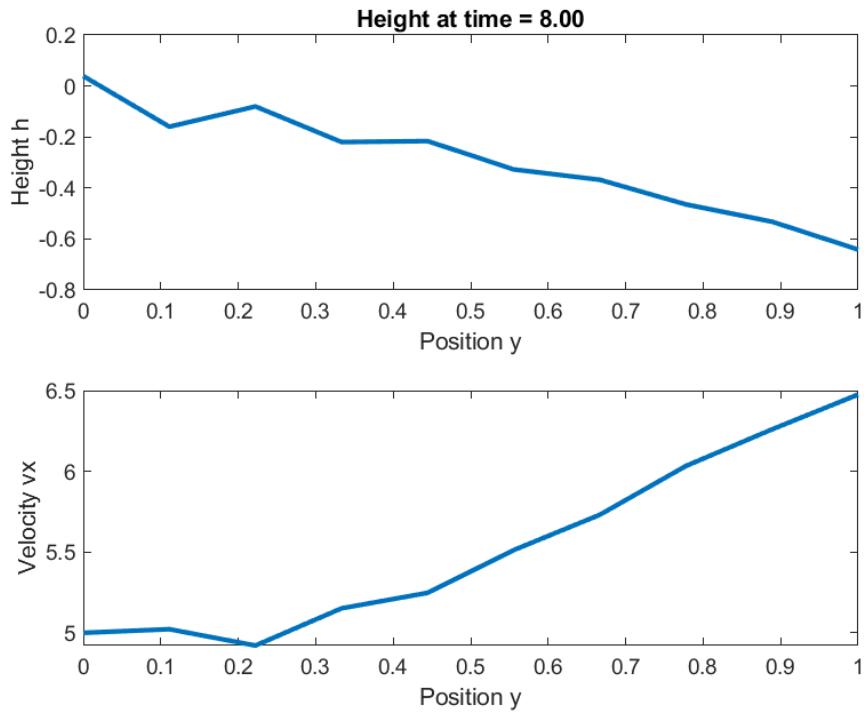


Figure 12: at $v=5$ and $h=39*d$ PDEPE plots

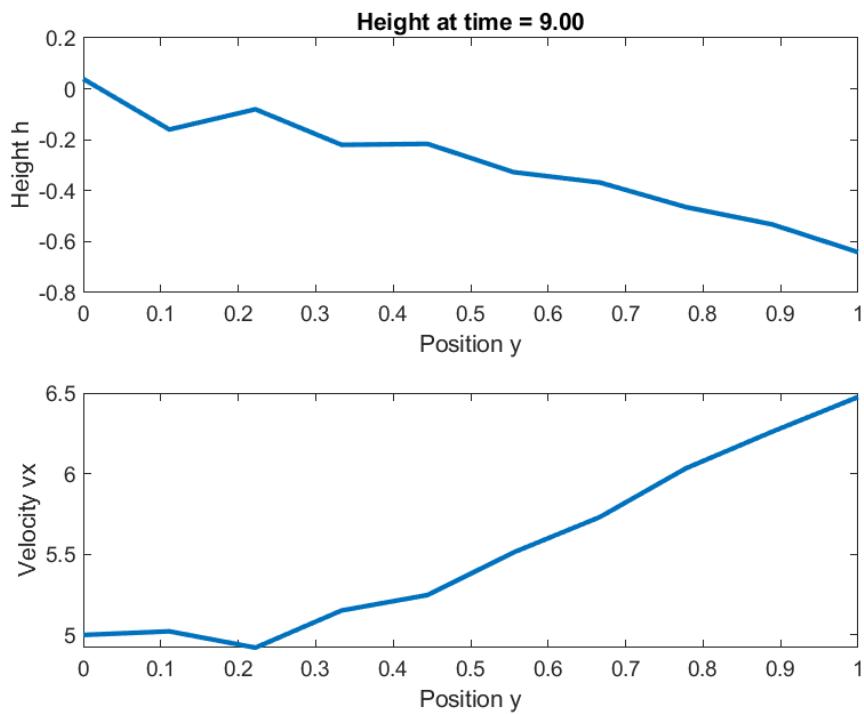


Figure 13: at $v=5$ and $h=39*d$ PDEPE plots