1 Appendix

1.1 Lotka-Volterra Solutions Code

```
T = 20;
N = 1000;
h = T/N;

% Initializing the arrays to store solutions over time.
Z_exp = zeros(2,N); Z_imp = zeros(2,N); Z_symp = zeros(2,N);

% Lotka-Volterra Equations
f = @(x) [x(1)-x(1)*x(2); x(1)*x(2)-2*x(2)];
df = @(x, y) [1-x(2); x(1) - 2];

% Newton-Raphson iteration for implicit methods
F_IE = @(x, y0, h) x - y0 - h*f(x);
DF_IE = @(x, h) eye(2) - h*df(x);
N_IE = @(y0, h) y0+h - (DF_IE(y0+h, h))^-1*F_IE(y0+h,y0,h);

% Initial value (2,4).
ysolv_EE = [2;4]; ysolv_IE = [2;4]; ysolv_symp = [2;4];
Z_exp(:,1) = ysolv_EE; Z_imp(:,1) = ysolv_IE; Z_symp(:,1) = ysolv_symp;


for i = 2: N
   ysolv_EE = ysolv_EE + h*f(ysolv_EE);
   ysolv_IE = N_IE(ysolv_IE, h);

   % Symplectic Euler variant: explicit in u, implicit in v
   Z_symp(1,i) = Z_symp(1,i-1)/(1-h*(1-Z_symp(2,i-1)));
   Z_symp(2,i) = Z_symp(2,i-1)*(1+h*(Z_symp(1,i)-2));

   Z_exp(:,i) = ysolv_EE;
   Z_imp(:,i) = ysolv_IE;

end

% Phase planes of Lotka-Volterra equations
x = 0.2:0.2:max(Z_exp(1,:));
y = 0.2:0.2:max(Z_exp(2,:));
[X,Y] = meshgrid(x,y);
Z = X - 2*real(log(X)) + Y - real(log(Y));

% Plotting the numerical solutions beside each other
ax1 = subplot(1,3,1);
plot(ax1, Z_exp(1,:), Z_exp(2,:),'b')
hold on;
plot(ax1, [2], [4], 'o');
hold on;
contour(ax1, X,Y,Z,'k:');
title(ax1,'Explicit Euler');
xlabel(ax1, 'u');
ylabel(ax1, 'v');

ax2 = subplot(1,3,2);
```

```
53 plot(ax2, Z_imp(1,:), Z_imp(2,:),'b')
54 hold on;
55 plot(ax2, [2], [4], 'o');
56 hold on;
57 contour(ax2, X,Y,Z,'k:');
58 title(ax2,'Implicit Euler');
59 xlabel(ax2, 'u');
60 ylabel(ax2, 'v');
61
62 ax3 = subplot(1,3,3);
63 plot(ax3, Z_symp(1,:), Z_symp(2,:),'b')
64 hold on;
65 plot(ax3, [2], [4], 'o');
66 hold on;
67 contour(ax3, X,Y,Z,'k:');
68 title(ax3,'Symplectic Euler');
69 xlabel(ax3, 'u');
70 ylabel(ax3, 'v');
```

1.2 Convergence Tests on Pendulum Problem Code

```
% Code adapted from A. Paganini, Numerical Solutions of DEs I.

% Pendulum problem initial configuration
l = 5; g = 10; T = 5;
theta = 0;
v_theta = 5;
y0 = [theta; v_theta];

f = @(y) [y(2); -g*(1/l)*sin(y(1))];

% Reference solution with MATLAB function ode45
opts = odeset('AbsTol', 1e-12, 'RelTol', 1e-12);
[~, yref] = ode45(@(t,y) f(y), [0, T], y0, opts);
yref_final = yref(end,:).';

N = 2.^(4:10); hsave = zeros(size(N));
err = zeros(size(N));
figure_num = 0;
beta = 1.00; % we vary this value as preferred

% For each value of gamma, repeat the pendulum problem for different step sizes h
for gamma =0:0.25:1
    for i = 1:length(N)
        h = T/N(i); hsave(i) = h;
        x = theta; v = v_theta;

        for jj = 1:N(i)
            [x, v] = newmark_beta(beta, gamma, x, v, h, ...
                                  @update_acceleration, @d_acceleration);
         end

        err(i) = norm([x; v] - yref_final);
    end

    % Plot the graph of convergence
    loglog(hsave, err, '*-', 'linewidth', 2)
    ylim([10^-5 10^1]);
    xlabel('log(h)');
    ylabel('log(err)');
    title('\beta = 1.00');
    hold on;
    % Find (asymptotic) slope of the graph plotted
    fit = polyfit(log(hsave(end-3:end)), log(err(end-3:end)), 1);
    fprintf('The convergence order of the Newmark-Beta method with \beta: %f ...
             and \gamma: %f is %1.2f\n', beta, gamma, fit(1));
end
hold off;
legend('0.00','0.25','0.50','0.75','1.00');
title(legend,'\gamma values');
legend('Location','northwest');
```

```
% Define the acceleration function and its derivate (for Newton-Raphson iteration)
function new_acceleration = update_acceleration(x)
    l = 5; g = 10;
    new_acceleration = -g*(1/l)*sin(x);
end

function derivative_acceleration = d_acceleration(x)
   l = 5; g = 10;
   derivative_acceleration = -g*(1/l)*cos(x);
end
```

```
function [x_new, v_new] = newmark_beta(beta, gamma, x_i, v_i, h, ...
                                       update_acceleration, d_acceleration)
    a_i = update_acceleration(x_i);
    y_i = [v_i; x_i];

    % Newton-Raphson iteration
    f = @(z, v_i, a_i, h) [(1-gamma)*a_i + gamma*update_acceleration(z); ...
                   v_i + 0.5*h*((1-2*beta)*a_i + 2*beta*update_acceleration(z))];
    Df = @(z, h) [zeros(size(z)) h*gamma*z; zeros(size(z)) 0.5*beta*h*h*z];
    F = @(y_j, y_i, h) y_j - y_i - h*f(y_j(2), v_i, a_i, h);
    DF = @(y_j, h) eye(size(y_j, 1)) - Df(d_acceleration(y_j(2)), h);
    N = @(y_i, h) y_i+h - (DF(y_i+h, h))^-1*F(y_i+h,y_i,h);

    % New solutions
    y_j = N(y_i, h);
    x_new = y_j(2);
    v_new = y_j(1);

end
```

1.3 Lennard-Jones Potential Graph Code

```
% Plots the Lennard-Jones potential curve

r = 0:0.01:5;
inv_r6 = r.^-6;
inv_r12 = inv_r6.^2;
V = 4*(inv_r12-inv_r6);
plot(r(1,:),V(1,:),'b');
xlim([0 3]);
ylim([-1.5 2]);
hold on;
plot(r,zeros(size(r,2)),'k');
set(gca,'xtick',[],'ytick',[0]);
ylabel('V');
xlabel('r','fontweight','bold');
```

1.4 Molecular Dynamics: Implicit Methods Code

```
tic;
[num_particles, epsilon, sigma, r_cutoff, mass, density, kB, temperature, ...
                      h, N_e, N_f, N_s, N_n, beta, gamma] = initialize_params();

length_cube = find_cube_length(num_particles, density);
% Adding space to the top given how the lattice is created in initialize_cube.m
diff = length_cube/((2*num_particles)^(1/3));
coordinates = initialize_cube(num_particles, length_cube-diff);
velocities = initialize_velocities(num_particles, mass, kB, temperature);

% Calculate initial neighbours_list
[neighbours_list, num_neighbours_list] = find_neighbours(num_particles, ...
                                         coordinates, length_cube, r_cutoff);
[forces, ~, jacobian_matrix] = find_forces(num_particles, epsilon, sigma, ...
                                         coordinates, length_cube,
                                         neighbours_list, num_neighbours_list);
energy = zeros(N_f/N_s, 3);
for i = 1:(N_e+N_f)
    % Update coordinates and velocities
    acceleration = forces/mass;
    seed_coordinates = coordinates+h;
    F = seed_coordinates - coordinates - h*velocities - ...
        0.5*h^2*((1-2*beta)*acceleration);
    [forces, ~, jacobian_matrix] = find_forces(num_particles, epsilon, sigma,...
                                          seed_coordinates, length_cube,
                                          neighbours_list, num_neighbours_list);
    acceleration = forces/mass;
    F = F - 0.5*h^2*(2*beta*acceleration);
    DF = eye(3*num_particles) - beta*h^2*jacobian_matrix;
    delta_coordinates = DF\(-F);
    coordinates = coordinates + delta_coordinates;

    velocities = velocities + h*(1-gamma)*acceleration;
    [forces, potential_energy, jacobian_matrix] = ...
                                find_forces(num_particles, epsilon, sigma, ...
                                        coordinates, length_cube, ...
                                        neighbours_list, num_neighbours_list);
    acceleration = forces/mass;
    velocities = velocities + h*gamma*acceleration;

    if (i <= N_e)
       % Equilibration stage
       scaling_factor = (3*kB*num_particles*temperature)/ ...
                        (mass*sum(sum(velocities.^2,2)));
       velocities = velocities * sqrt(scaling_factor) ;
    elseif (mod(i, N_s) == 0)
       % Take sample
       energy_index = (i-N_e)/N_s;
       energy(energy_index, 1) = potential_energy/num_particles;
       energy(energy_index, 2) = 0.5*(mass*sum(sum(velocities.^2,2)))...
                                 /num_particles;
    end

    if (mod(i, N_n) == 0)
```

```
        % Refresh neighbours list
        [neighbours_list, num_neighbours_list] = ...
              find_neighbours(num_particles, coordinates, length_cube, r_cutoff);
    end
end

% Adjust potential energy
r_cutoff_6 = r_cutoff^6;
sigma_6 = sigma^6;
shifting_potential_term = 4*(sigma_6/r_cutoff_6)*((sigma_6/r_cutoff_6) - 1);
energy(:,1) = energy(:,1) - shifting_potential_term;
energy(:,3) = energy(:,1) + energy(:,2);

steps = linspace(1,N_f/N_s,N_f/N_s);
figure;
plot(steps, energy(:,1), '-^', steps, energy(:,2), '-v', steps, energy(:,3), '-o');
xlabel('Iterations'); ylabel('Energy (\epsilon)');
legend('Potential Energy', 'Kinetic Energy', 'Total Energy','Location','east');
toc;
```

```
function [num_particles, epsilon, sigma, r_cutoff, mass, density, kB, ...
          temperature, h, N_e, N_f, N_s, N_n, beta, gamma] = initialize_params()

    num_particles = 864;
    epsilon = 1;
    sigma = 1;
    r_cutoff = 2.5;
    mass = 48;
    density = 38.74457642;
    kB = 1;
    temperature = 1;
    h = 0.032;
    N_e = 100;
    N_f = 500;
    N_s = 10;
    N_n = 15;
    beta = 0.00;  % [0,0.5]
    gamma = 0.50; % [0,1]
end
```

```
% Creates an FCC crystal
function coordinates = initialize_cube(num_particles, length_cube) % FCC crystal
    one_side = (2*num_particles)^(1/3);
    coordinates = zeros(3*num_particles,1);
    i = 1;
    particle_spacing = length_cube / (one_side - 1);
    for x = 0:particle_spacing:length_cube
        for y = 0:particle_spacing:length_cube
            if (mod(x+y,2*particle_spacing) == 0)
               for z = 0:2*particle_spacing:length_cube
                  coordinates(i:i+2) = [x,y,z]';
                  i = i + 3;
               end
            else
                for z = particle_spacing:2*particle_spacing:length_cube
                  coordinates(i:i+2) = [x,y,z]';
```

```
                    i = i + 3;
                end
            end
        end
    end
end
```

```
function length_cube = find_cube_length(num_particles, density)
    length_cube = (48 * num_particles / density)^(1/3);
end
```

```
function velocities = initialize_velocities(num_particles, mass, kB, temperature)
    % Seeding to maintian reproducibility
    seed = 7;
    rng(seed);

    % Generate velocities using Maxwell distribution, and remove linear momentum
    velocities_3d = normrnd(0, (kB*temperature/mass)^0.5, num_particles, 3);
    velocities_3d = velocities_3d - sum(velocities_3d)/num_particles;
    %Arrange as a column vector
    velocities = [velocities_3d(:,1)'; velocities_3d(:,2')'; velocities_3d(:,3)'];
    velocities = velocities(:);
end
```

```
function [neighbours_list,num_neighbours_list] = ...
          find_neighbours(num_particles, coordinates, length_cube, r_cutoff)
    neighbours_list = zeros(num_particles, num_particles -1);
    num_neighbours_list = zeros(num_particles, 1);
    r_cutoff_2 = r_cutoff*r_cutoff;
    for i = 1:3:3*num_particles
        num_neighbours_i = 0;
        i_index = (i+2)/3;
        for j = i+3:3:3*num_particles
            j_index = (j+2)/3;
            diff_r = coordinates(i:i+2) - coordinates(j:j+2);
            diff_r = diff_r - length_cube*round(diff_r/length_cube);
            diff_r_2 = sum(diff_r.^2);
            if diff_r_2 < r_cutoff_2
                num_neighbours_i = num_neighbours_i + 1;
                neighbours_list(i_index, num_neighbours_i) = j;
                num_neighbours_j = num_neighbours_list(j_index);
                num_neighbours_j = num_neighbours_j + 1;
                neighbours_list(j_index, num_neighbours_j) = i;
                num_neighbours_list(j_index) = num_neighbours_j;
            end
        end
        num_neighbours_list(i_index) = num_neighbours_i;
    end
    % Re-adjust size of neighbours_list to save memory
    neighbours_list = neighbours_list(:,1:max(num_neighbours_list));
end
```

```
function [forces, potential_energy, jacobian_matrix] = ...
          find_forces(num_particles, epsilon, sigma, ...
          coordinates, length_cube, neighbours_list, num_neighbours_list)
    potential_energy = 0;
    sigma_6 = sigma^6;
    forces = zeros(3*num_particles, 1);
    jacobian_matrix = zeros(3*num_particles);

    for i = 1:3:3*num_particles
        i_index = (i+2)/3;
        for neighbour = 1:num_neighbours_list(i_index)
            j = neighbours_list(i_index, neighbour);
            if (j > i)
                diff_r = coordinates(i:i+2) - coordinates(j:j+2);
                diff_r = diff_r - length_cube*round(diff_r/length_cube);
                dist_r_2 = sum(diff_r.^2);
                dist_r_6 = dist_r_2^3; dist_r_8 = dist_r_6 * dist_r_2;
                dist_r_10 = dist_r_8 * dist_r_2;

                force_factor = (sigma_6/dist_r_8)*(sigma_6/dist_r_6 - 0.5);
                forces(i:i+2) = forces(i:i+2) + force_factor*diff_r;
                forces(j:j+2) = forces(j:j+2) - force_factor*diff_r;

                jacobian_factor = (sigma_6/dist_r_10)* ...
                                  ((-14*sigma_6/dist_r_6) + 4);
                a = diff_r * diff_r';
                jacobian_force_block = force_factor*eye(3) + ...
                                       jacobian_factor*(diff_r * diff_r');

                jacobian_matrix(i:i+2,i:i+2) = ...
                            jacobian_matrix(i:i+2,i:i+2) + jacobian_force_block;
                jacobian_matrix(j:j+2,j:j+2) = ...
                            jacobian_matrix(j:j+2,j:j+2) + jacobian_force_block;
                jacobian_matrix(i:i+2,j:j+2) = ...
                            jacobian_matrix(i:i+2,j:j+2) - jacobian_force_block;
                jacobian_matrix(j:j+2,i:i+2) = ...
                            jacobian_matrix(j:j+2,i:i+2) - jacobian_force_block;

                potential_energy = potential_energy + (sigma_6/dist_r_6)*...
                                                      ((sigma_6/dist_r_6) - 1);
            end
        end
    end
    forces = 48*epsilon*forces;
    jacobian_matrix = 48*epsilon*jacobian_matrix;
    potential_energy = 4*epsilon*potential_energy;
end
```