Table of Contents

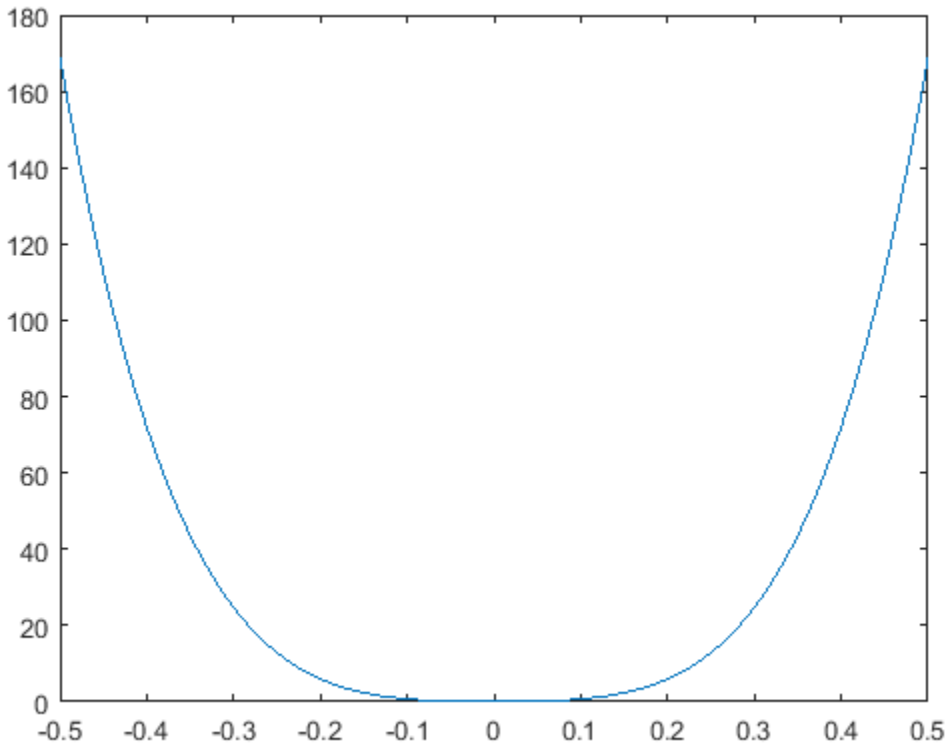
PHY5340 Laboratory 5: Shooting for Eigenvalues and Eigenfunctions

Jeremiah O'Neil, SN6498391

```
format shortG
newV = @(pars) (@(x) pars(1)*x.^2 + pars(2)*x.^3 + pars(3)*x.^4);
tol = 5e-7;  % for bisect; choose lower for shooter
inita = [0, 1e-5];
initb = [0, 1e-5];
```

Case 1: V(x) = 50x^2 + 2500x^4

```
V = newV([50, 0, 2500]);
x = [-0.5:0.01:0.5];
plot(x, V(x))
```



Choose endpoints and bind a function to shoot for E.

```
x0 = -0.5;  xm = 1e-2;  x1 = 0.5;
type('solve_TISE.m')
type('shooter.m')
% bind parameters, making a shooter function shoot(E)
shoot = @(E) shooter( V, E, x0, x1, xm, inita, initb, 0.2*tol );
```

```
function [ x, u ] = solve_TISE( x0, xend, init, V, E, tol )
% use RK45 from Project 2 to integrate the TISE for given V(x) and E
 from x0 to xend
m2_hsqr = 2 / 0.076199682;  % eV^-1 nm^-2
[x, u] = RK45(@(x, u)[u(2), m2_hsqr*(V(x) - E)*u(1)], x0, xend, init,
 tol);
end
```

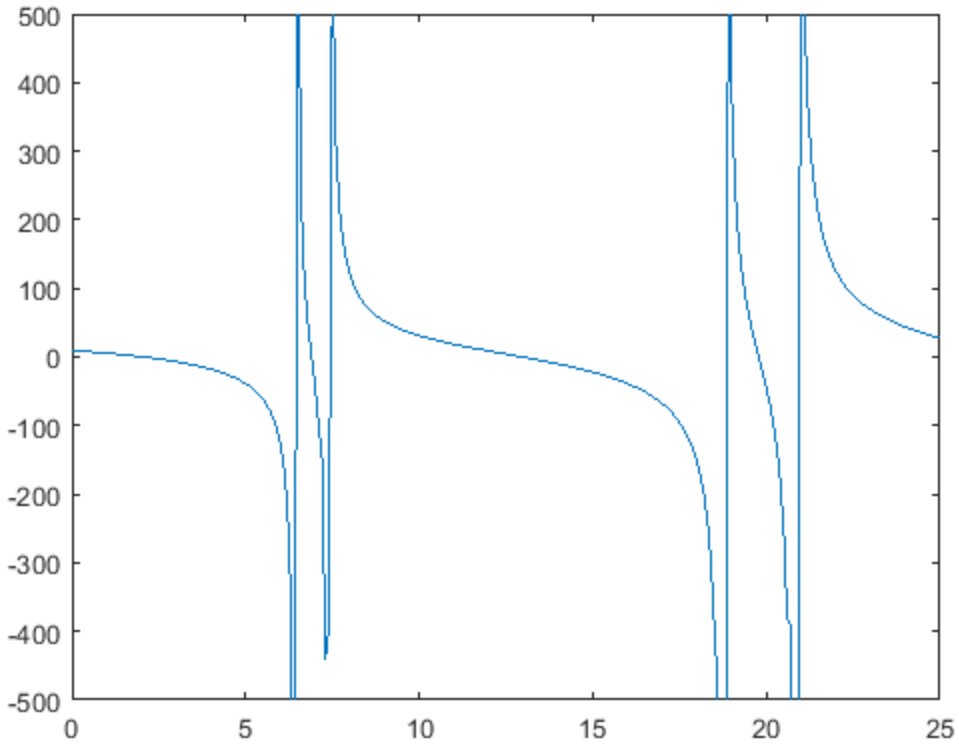
```
function [ miss, xa, xb, psia, psib ] = shooter( V, E, x0a, x0b, xm,
 inita, initb, tol )
% Solve the TISE to xm from x0a on the left and x0b on the right and
 return
% the offset 'miss' from the matching condition and the solutions for
% increasing x
[xa, ua] = solve_TISE( x0a, xm, inita, V, E, tol );
[xb, ub] = solve_TISE( x0b, xm, initb, V, E, tol );
miss = ua(2,end)/ua(1,end) - ub(2,end)/ub(1,end);
psia = ua(1,:);
psib = fliplr(ub(1,:));
xb = fliplr(xb);
end
```

Compute the offset from matching condition 'mismatch' for a grid of energies, starting from the minimum potential below which no solution is possible.

```
E = [0:0.1:25];
% map shoot to E grid
mismatch = arrayfun(shoot, E);
```

Plot the mismatch to locate eigenvalues (ie., where mismatch = 0)

```
plot(E, mismatch)
ylim([-500, 500])
```



It appears that every other zero-crossing corresponds to an asymptote -- where a wavefunction derivative is zero at xm -- so I'll pick out the grid energies about odd-numbered zero crossings in as bounds on my eigenvalues.

```
type('find_zero_crossings.m')
cross_idcs = find_zero_crossings(mismatch);
target_idcs = cross_idcs(1:2:end);  % skip asymptotes
Ebounds = [E(target_idcs); E(target_idcs + 1)]
Ebounds = Ebounds(:,1:3);  % select first three intervals
```

```
function [ indices ] = find_zero_crossings( x )
%return the indices {i} of x where x crosses zero from x(i) to x(i +
 1)
indices = find(diff(sign(x)));
% diff(sign(y)) is nonzero only where the sign changes, ie., at
% zero-crossings
end
```

```
Ebounds =

            2          6.9         12.9         19.7
          2.1            7           13         19.8
```

I'll normalize my eigenfunctions in the standard manner using trapezoid-rule quadrature, and I'll use my function 'find_eig' to find a solution withing energy bounds by the shooting method, through another function 'compute_eigs' which runs find_eig for each interval in Ebounds and packs up the solutions.

```
type('discrete_trap.m')
type('find_eig.m')
type('compute_eigs.m')
[eigE, eigfuncs] = compute_eigs( shoot, Ebounds, tol );
```

```
function [ result ] = discrete_trap( f, x )
% Trapezoid rule quadrature, for discrete f above x, for arbitrary
 spacing.
result = 0.5 * sum(diff(x) .* (f(1:end-1) + f(2:end)));
end
```

```
function [ E, xpsi, est_err, miss, numiter ] = find_eig( shoot, Ea,
 Eb, tol )
% Use bisection rootfinding (from Lab2) to shoot for eigenfunctions,
% corresponding to eigenvalues where the mismatch is zero.
[E, est_err, numiter] = bisect(50, tol, shoot, Ea, Eb);
[miss, xa, xb, psia, psib] = shoot(E);
x = [xa, xb];
psi = [psia, psib * psia(end)/psib(1)];
% Normalize
N = (discrete_trap(psi.^2, x))^-0.5;
xpsi = [x; N * psi];
end
```

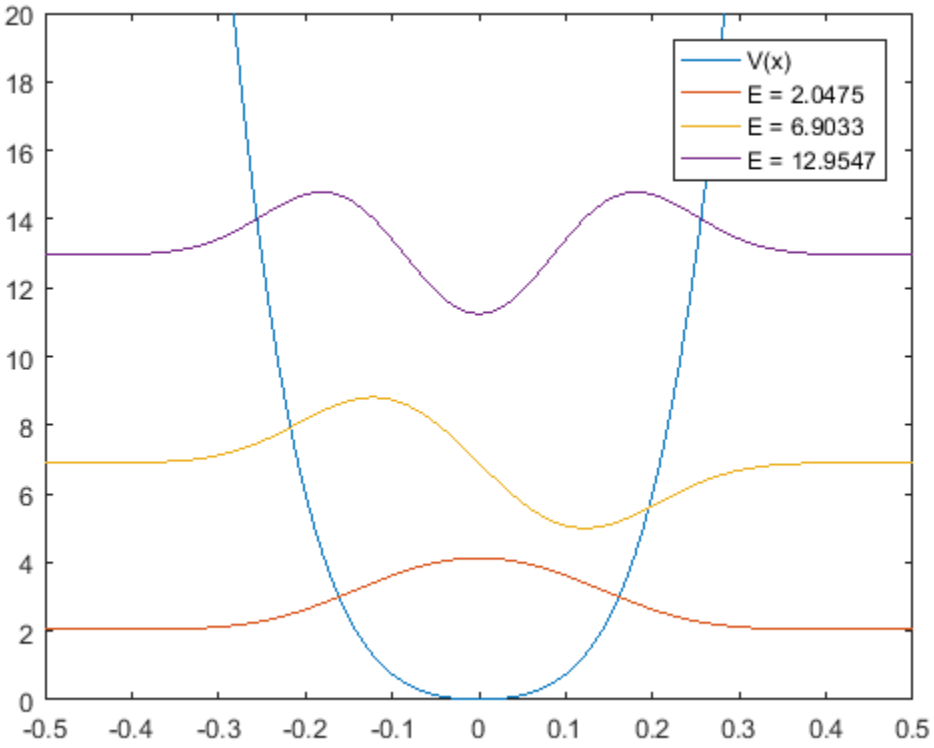
```
function [ eigE, eigfuncs, miss, est_err, numiter ] =
 compute_eigs( shoot, Ebounds, tol )
% find the eigenvalues between specified bounds using the find_eig
 function
num = size(Ebounds, 2);
eigE = zeros(1, num);
eigfuncs = cell(1, num);
miss = zeros(1, num);
est_err = zeros(1, num); numiter = zeros(1, num);
for i = 1:num
    [eigE(i), eigfuncs{i}, est_err(i), miss(i), numiter(i)] = ...
        find_eig( shoot, Ebounds(1,i), Ebounds(2,i), tol );
end
end
```

Plot the results!

```
type('plot_eigs.m')
plot_eigs( V, eigE, eigfuncs );
ylim([0, 20]);
```

```
function [] = plot_eigs( V, eigE, eigfuncs )
labels = cell(1, 4);
```

```
x = [-1:0.01:1];
plot(x, V(x)); hold on;
labels{1} = ['V(x)'];
for i = 1:3
    plot(eigfuncs{i}(1,:), eigE(i) + eigfuncs{i}(2,:));
    labels{i+1} = ['E = ' num2str(eigE(i))];
end
legend(labels); hold off;
end
```



To ensure the accuracy of my results I'll test the sensitivity of the computed eigenvalues to my choice of endpoints by randomly varying the endpoints and observing if the difference with respect to my solution above is above tolerance.

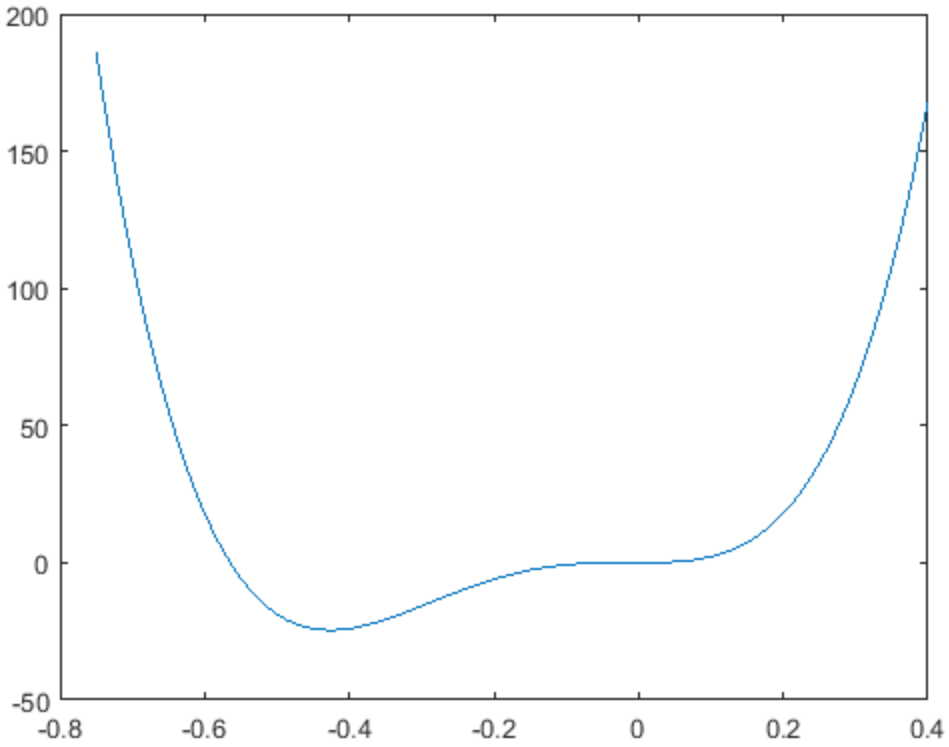
```
type('test_endpoint_sensitivity.m')
% shooter with parameters partially bound
partial_shooter = @(E, x0, x1) shooter( V, E, x0, x1, xm, inita,
 initb, 0.2*tol );
[xd, delta_eigE] = test_endpoint_sensitivity( 5, partial_shooter, x0,
 x1, eigE, Ebounds, tol)
% Some choices of endpoints lead to significant differences, but these
 are
% outliers; most results agree with mine exactly! (Note, bisection of
% nearly identical functions on the same interval to the same
 tolerance
% should always find the same root though the functions are slightly
```

```
% different.)


function [ xd, delta_eigE ] = test_endpoint_sensitivity( trials,
 partial_shooter, x0, x1, eigE, Ebounds, tol)
d = 1.1 + 0.05 * randn(2, trials);
x0d = x0 * d(1,:);
x1d = x1 * d(2,:);
xd = [x0d; x1d];
delta_eigE = zeros(3, trials);
for i = [1:trials]
    shoot = @(E) partial_shooter( E, x0d(i), x1d(i) );
    eigEd = compute_eigs( shoot, Ebounds, tol );
    delta_eigE(:,i) = eigEd - eigE;
end
end

xd =

    -0.59522    -0.55498    -0.53191    -0.56003    -0.55751
       0.523     0.51197     0.53517     0.57355     0.54067


delta_eigE =

           0    0.015448           0           0           0
           0           0           0           0           0
           0           0           0           0           0
```

Case 2: V(x) = 50x^2 + 1500x^3 + 2500x^4

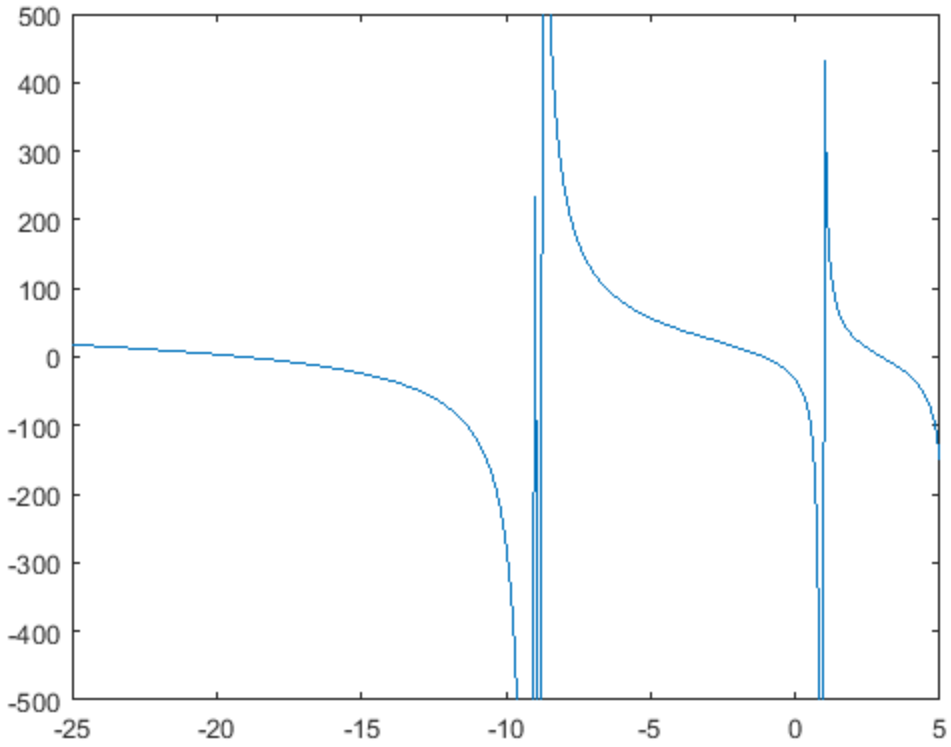
Exactly as above, with a new V(x) and x0, xm, x1.

```
V = newV([50, 1500, 2500]);
x = [-0.75:0.01:0.4];
plot(x, V(x))
```



I'll choose xm near the center of the lowest part of the well, which I expect to align to the centre of the lowest energy eigenfunctions, and x1 and x0 out in the excluded region to roughly the same potential height.

```
x0 = -0.9;  xm = -0.4;  x1 = 0.5;
shoot = @(E) shooter( V, E, x0, x1, xm, inita, initb, 0.2*tol );

E = [-25:0.1:5];
mismatch = arrayfun(shoot, E);

plot(E, mismatch)
ylim([-500, 500])
```

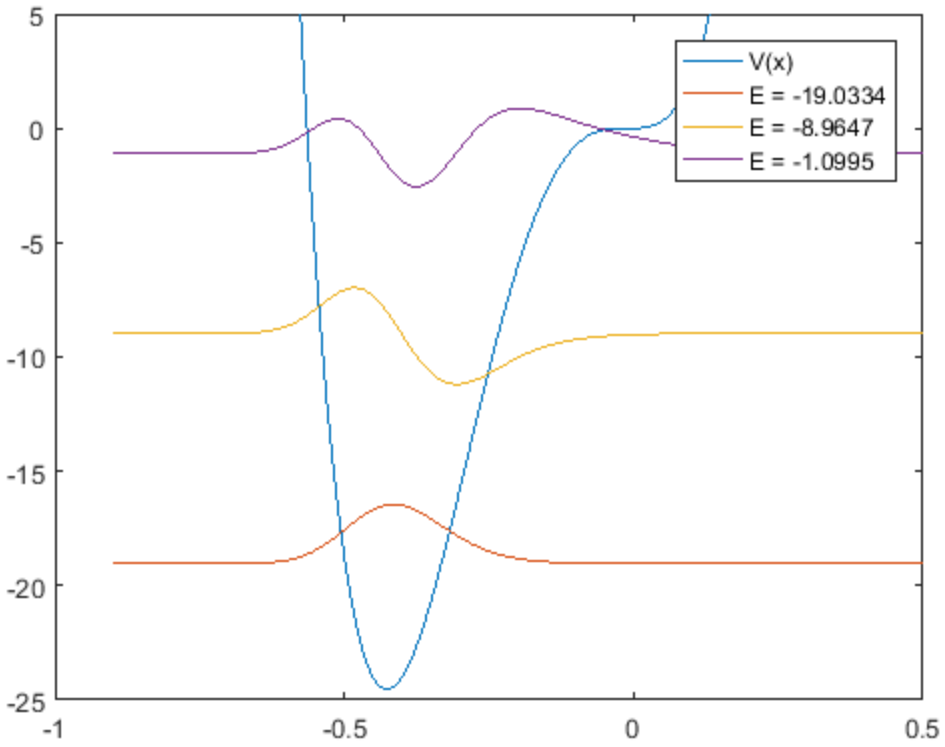



```
cross_idcs = find_zero_crossings(mismatch);
target_idcs = cross_idcs(1:2:end);
Ebounds = [E(target_idcs); E(target_idcs + 1)]
Ebounds = Ebounds(:,1:3);
```

Ebounds =

```
   -19.1          -9        -1.1         2.9
     -19        -8.9          -1           3
```

```
[eigE, eigfuncs] = compute_eigs( shoot, Ebounds, tol );

plot_eigs( V, eigE, eigfuncs );
ylim([-25, 5]);
```




```
partial_shooter = @(E, x0, x1) shooter( V, E, x0, x1, xm, inita,
 initb, 0.2*tol );
[xd, delta_eigE] = test_endpoint_sensitivity( 5, partial_shooter, x0,
 x1, eigE, Ebounds, tol)
% all good!


xd =

      -1.0267      -0.99541       -1.0086       -1.0242      -0.96282
      0.56997       0.56428       0.52533       0.53357       0.55442


delta_eigE =

     0     0     0     0     0
     0     0     0     0     0
     0     0     0     0     0
```

Appendix: RK45 and bisect

```
type('RK45.m')
type('bisect.m')
```

```
function [xx,yy,n_failures]=RK45(f,x0,x_end,y0,tol)
%Adaptive step-size 4/5 order Runge Kutta solver, with tolerance tol.
 The
%code below is incomplete and will not run. It has not implemented the
%adaptive step size feature. You must
%   1) Make sure you understand every line of the code.
%   2) Add in the appropriate commands to make the adaptive step size.
%%
c = [0   1/4    3/8   12/13 1 1/2];
b5 =[16/135 0   6656/12825  28561/56430     -9/50   2/55];
b4 =[25/216 0   1408/2565   2197/4104   -1/5 0];
A=zeros(6);
A(2,1)=1/4;
A(3,1:2)=[3/32 9/32];
A(4,1:3)=[1932/2197   -7200/2197  7296/2197];
A(5,1:4)=[439/216     -8      3680/513    -845/4104];
A(6,1:5)=[-8/27   2    -3544/2565     1859/4104   -11/40];

%ensure that y0 is a column vector
y0=y0(:);

xx=x0; %initialize
yy=y0; %initialize

%Find the direction of propagation
if x_end>=x0
    xdir=1;
else
    xdir=-1;
end

%guess an initial step size
h=min([abs(x_end-x0)/10,0.1]); %h is always positive


x=x0;y=y0;
done=false;
n_failures=0;
dimy=numel(y0);

%Main Loop
while ~done
    failures=false;     %no step-size failures yet
    minh=16*eps(x);     %minimum acceptable value of h
    if  h<minh
        h = minh;
    end
    %Make sure to hit last step exactly
    if abs(x_end-x)<=h
        h=abs(x_end-x);
        done=true;
    end

    %Loop for advancing one step
```

```
    while true
        fn=zeros(6,dimy);

        %Please study these lines carefully. How do they work?
        for i=1:6
            fn(i,:)=f(x+xdir*h*c(i),y + xdir*h* (A(i,:)*fn).'  ).';
        end
        y5=y+xdir*h*(b5*fn).';
        y4=y+xdir*h*(b4*fn).';

        err_rel=max(abs((y4 - y5) ./ y5));

        if err_rel<tol
            %accept step and update step size. Do not increase step
 size by
            %more than a factor of ten nor reduce it by more than a
 factor
            %of 2.
            x=x+xdir*h;
            y=y5;
            xx=[xx,x];
            yy=[yy,y];
            h=h*min([10, 0.9*(tol/err_rel)^(0.2)]); %you must fill
 this in (may be more than one command)
            break
        else
            %reject step. Reduce h (by at most a factor of 2) and try
 again
            if ~failures
                h=h*max([0.2, 0.9*(tol/err_rel)^(0.2)]); %you must
 fill this in (may be more than one command)
                failures=true;
                n_failures=n_failures+1;
            else
                h=h*0.5;
            end
        end

    end
end



function [root, est_err, numiter] = bisect(maxiter, tol, func, a, b)
% Based on C1_1.m
assert(func(a)*func(b) < 0);
est_err = 1; numiter = 0;
while est_err > tol
    numiter = numiter + 1;
    mid = (a + b) / 2;
    if func(mid)*func(a) < 0
        b = mid;
    elseif func(mid)*func(b) < 0
        a = mid;
```

```
    end
    est_err = abs((a - b) / mid);
    if numiter > maxiter, break, end
end
root = mid;
end
```

Published with MATLAB® R2016a