Abelardo Riojas
ISC 4232C Lab 1
Prof. Quaife Fall 2020

Lab 1

1. Newton's Method

2.3 Code Deliverable

Create a function `Newtons_Method` that takes as inputs

- A function handle for the function, $f(x)$,
- A function handle for the function's derivative, $f'(x)$,
- The initial point, x_0,
- The maximum number of iterations,
- A tolerance, ϵ,

and returns x_k and the number of iterations it took to find x_k.

a.

```
[x1, iter] = newtons(1000,@func,@funcp,1e-8,2)

function val = func(x)
val = x^2 - 1;
end

function val = funcp(x)
val = 2*x;
end

function [x1, iter] = newtons(maxiter,f,fp,tol,x0)
    iter = 0;
    while iter < maxiter
        x0 %not suppressed for code check
        abs(x0 - 1) %not suppressed for code check
        x1 = x0 - (f(x0)/fp(x0));
        iter = iter + 1;
        if abs(x1 - x0) <= tol
            break;
        else
            x0 = x1;
        end
    end
end
```

You should get these results:

k	x_k	$\lvert x_k - 1 \rvert$
1	2.0000	1.0000
2	1.2500	2.5000×10^{-1}
3	1.0250	2.5000×10^{-2}
4	1.0003	3.0488×10^{-4}
5	1.0000	4.6461×10^{-8}
6	1.0000	1.1102×10^{-15}

b.

Output from command line:

```
x0 =
     2
ans =
     1
x0 =
    1.2500
ans =
    0.2500
x0 =
    1.0250
ans =
    0.0250
x0 =
    1.0003
ans =
   3.0488e-04
x0 =
    1.0000
ans =
   4.6461e-08
x0 =
    1.0000
ans =
   1.1102e-15
x1 =
     1
iter =
     6
```

2. Lagrangian Interpolation

Write a function called `lagrange_interp.m` or `lagrange_interp.py` that takes as an input

- Interpolation points, $\{x_i\}_{i=1}^n$, and function values, $\{y_i\}_{i=1}^n$,
- A target point, x,

and returns the value of the Lagrange interpolation polynomial evaluated at the point x. This function should call a subfunction `lagrange_basis` that takes as an input

- Interpolation points, $\{x_i\}_{i=1}^n$,
- A target point, x,

and returns a vector of Lagrange polynomials, $L_i(x)$, for all $i = 1, \ldots, n$, evaluated at the target point x.

```
function val = lagrange_interp(xi,yi,target)
n = length(xi);
val = 0;
L = lagrange_basis(xi, target);
    for i = 1:n
        val = val + yi(i)*L(i);
    end
end

function L = lagrange_basis(xi, x)
n = length(xi);
L = ones(n,1);
    for i = 1:n
        for j = 1:n
            if i ~= j
            L(i) = L(i) * ((x - xi(j))/(xi(i) - xi(j))) ;
            end
        end
    end
end
```

Given a function $f : [a, b] \to \mathbb{R}$, the quality of the interpolant depends on the location of the interpolation points. To demonstrate this, consider the function

$$f(x) = \frac{1}{1 + 25x^2}, \quad x \in [-1, 1].$$

Use your code `lagrange_interp` to evaluate Lagrange interpolants of $f(x)$ at 500 equispaced target points in $[-1, 1]$. Compare $f(x)$ and $p(x)$ when using the following interpolation points

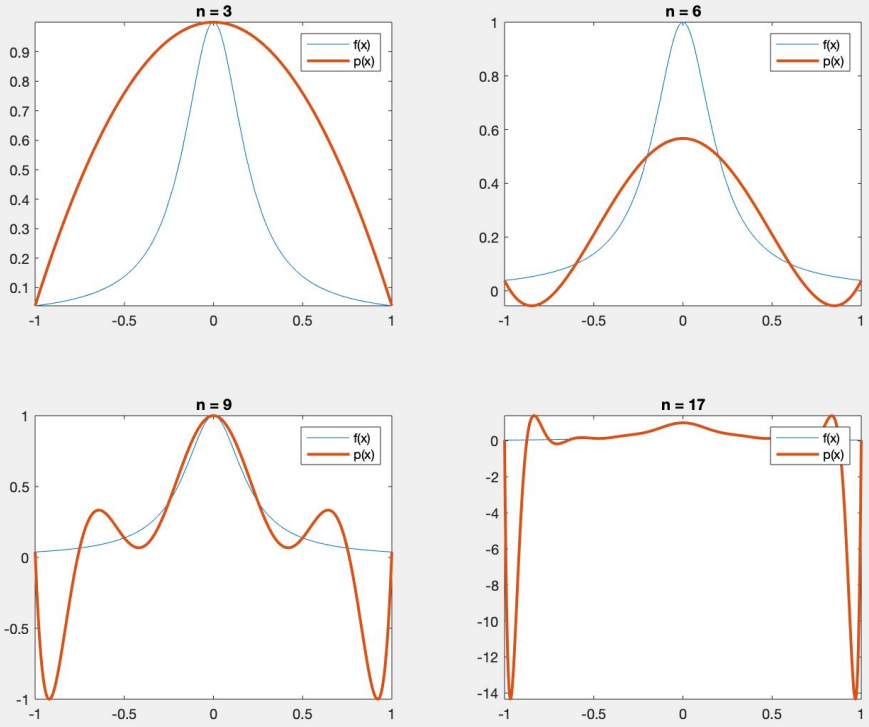
- n equispaced points in $[-1, 1]$,
- The n Chebyshev points

$$x_i = \cos\left(\frac{2i - 1}{2n}\pi\right), \quad i = 1, \ldots, n.$$

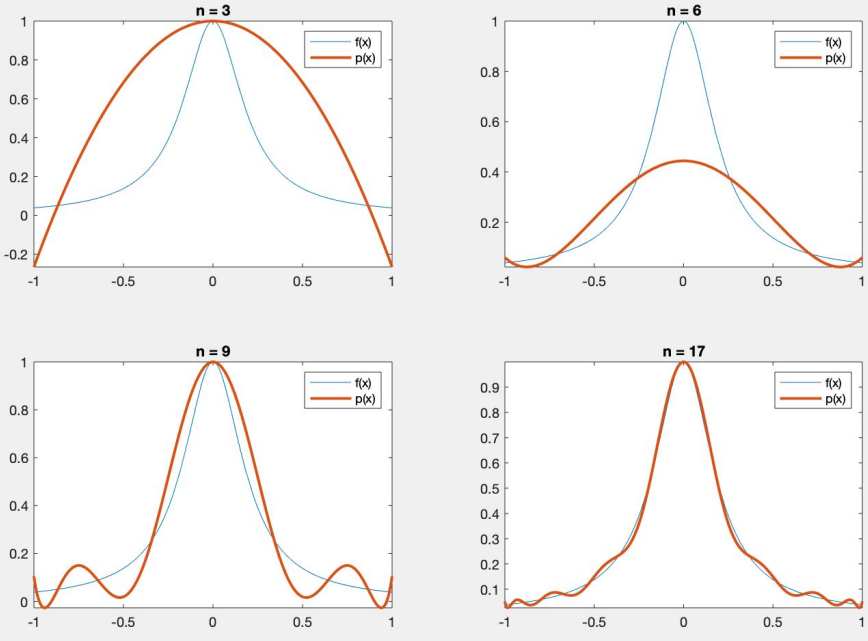
For both choices of interpolation points, use $n = 3$, 5, 9, and 17 interpolation points. For each of these 8 sets of interpolation points, plot $p(x)$ and $f(x)$ on the same plot. Also plot $f(x) - p(x)$ for all 8 sets of interpolation points. Use subplots to save space. **Compare these choices of interpolation points and discuss your results.**

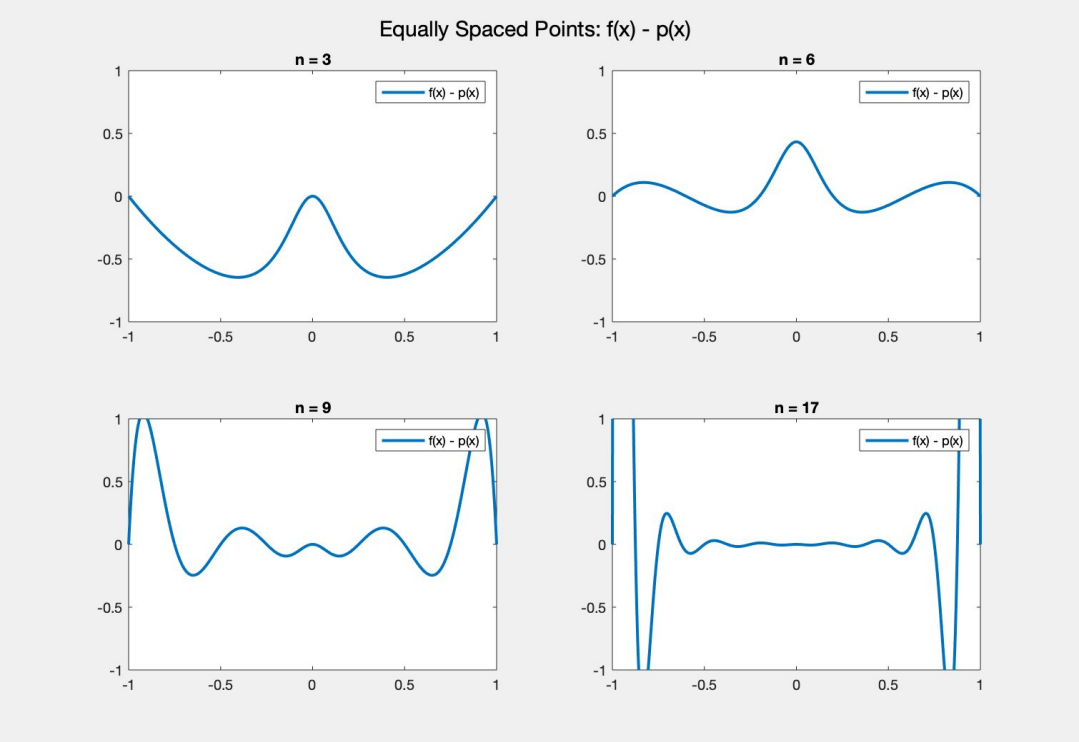
```
N = [3 6 9 17];
for i = 1:length(N)
    n = N(i);
    interpX = linspace(-1,1,n); %EQUALLY SPACED
%     interpX = zeros(n,1);
%     for c = 1:n
%         interpX(c) = cos(((2*c-1)/(2*n))*pi); %CHEBYSHEV
%     end
    interpY = lf(interpX);
    targetX = linspace(-1,1,500);
    targetY = zeros(1,500);
    for j = 1:500
        targetY(j) = lagrange_interp(interpX,interpY,targetX(j));
    end
    sgtitle('Equally Spaced Points')
    subplot(2,2,i)
    %fxminuspx = lf(targetX) - targetY; % for f(x) - p(x)
    %plot(targetX,fxminuspx,'LineWidth', 2 % ^^^^
    fplot(@(x) 1/(1+25*x^2), [-1 1])
    hold on
    plot(targetX,targetY,'LineWidth', 2)
    subptitle = sprintf('n = %d',n);
    title(subptitle)
    legend('f(x)','p(x)')
 end

function val = lf(x)
val = 1./(1 + (25.*x.^2));
end
```

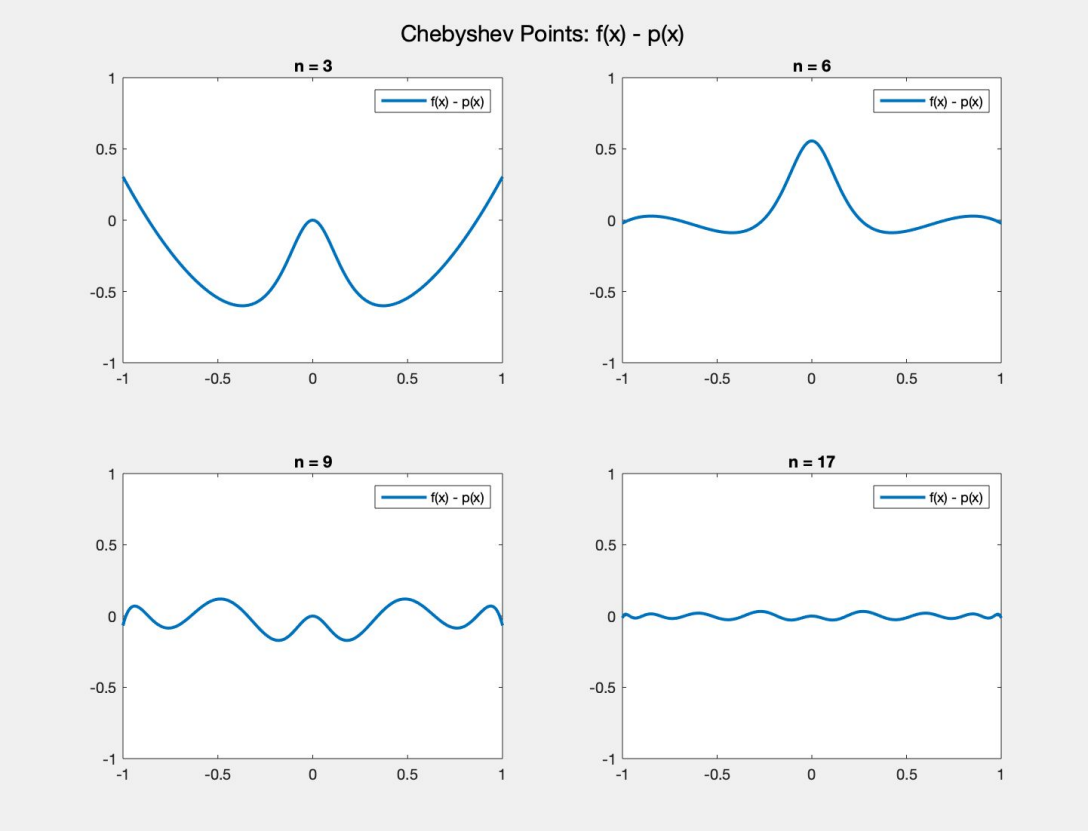
Equally Spaced Points



Chebyshev Points







Discussion: The Chebyshev points avoid Runge's Phenomenon as the difference between f(x) and p(x) at the edges of the interpolation stays small as n gets larger.

3. Forward Euler

Write a function called `Forward_Euler` that takes as inputs

- An initial condition y_0 and time 0,
- A final time T and time step Δt,
- The function, $f(y,t)$,

and returns a vector of approximate solutions, $Y = Y^n$, and a vector of time steps, $t = n\Delta t$, for $n = 1, \ldots, N$. Note that N depends on the final time T.

```
function [T,Y,err] = Forward_Euler(y0,t0,T,dt,fyt)
T = linspace(t0,T,1/dt);
n = length(T);
Y = zeros(1,n);
Y(1) = y0;
yn = y0;
for i = 2:n
    Y(i) = yn + dt*fyt(yn,T(i));
    yn = Y(i);
end
    err = abs(Y(end) - exp((-T(end)^2)/2));
end
```

Apply forward Euler to the IVP

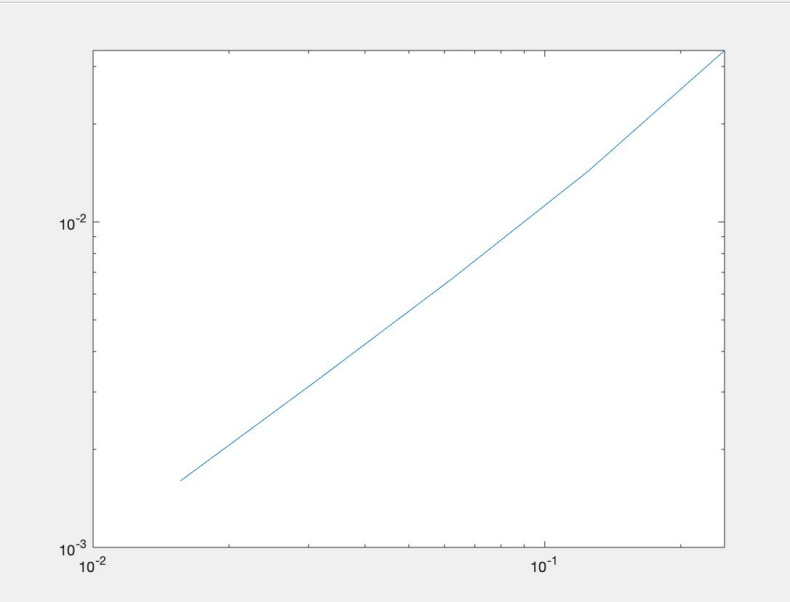
$$y'(t) = -ty(t), \quad y(0) = 1,$$

with a final time $T = 1$. This IVP has the exact solution

$$y(t) = e^{\frac{-t^2}{2}}.$$

Test your code for $\Delta t = \frac{1}{4}, \frac{1}{8}, \frac{1}{16}, \frac{1}{32}, \frac{1}{64}$. For each Δt, tabulate the solution and error at the final time step, $|Y^n - y(t_n)|$, where $t_n = 1$. Plot the errors for each Δt when $t = 1$ on a `loglog` plot. What is the numerical rate of convergence?

```
dts = [1/4 1/8 1/16 1/32 1/64];
errs = zeros(1,length(dts));
for i = 1:length(dts)
    [~,~,err] = Forward_Euler(1,0,1,dts(i),@ode);
    errs(i) = err;
end
loglog(dts,errs);

function val = ode(yn,tn)
val = -tn * yn;
end
```



The rate of convergence appears to be O(t), as the timestep decreases by half, the error also decreases by half. To prove this, I looked at the ratio of the error between err(n) and err(n+1) (n = 1, 2, 3, 4) and found that all of them had a ratio of 2, meaning that all the errors decreased by a factor of 2 as the timesteps were halved.

4. Backward Euler

Create a function `Backward_Euler` that takes as inputs

- An initial condition y_0 and time 0,
- A final time T and time step Δt,
- The function $f(y,t)$,
- The partial derivative with respect to y of $f(y,t)$, $\frac{\partial f(y,t)}{\partial y}$,

and returns a vector of approximate solutions, $Y = Y^n$, and a vector of time steps, $t = n\Delta t$. This function should call a subfunction `Backward_Euler_Step` that takes as an input

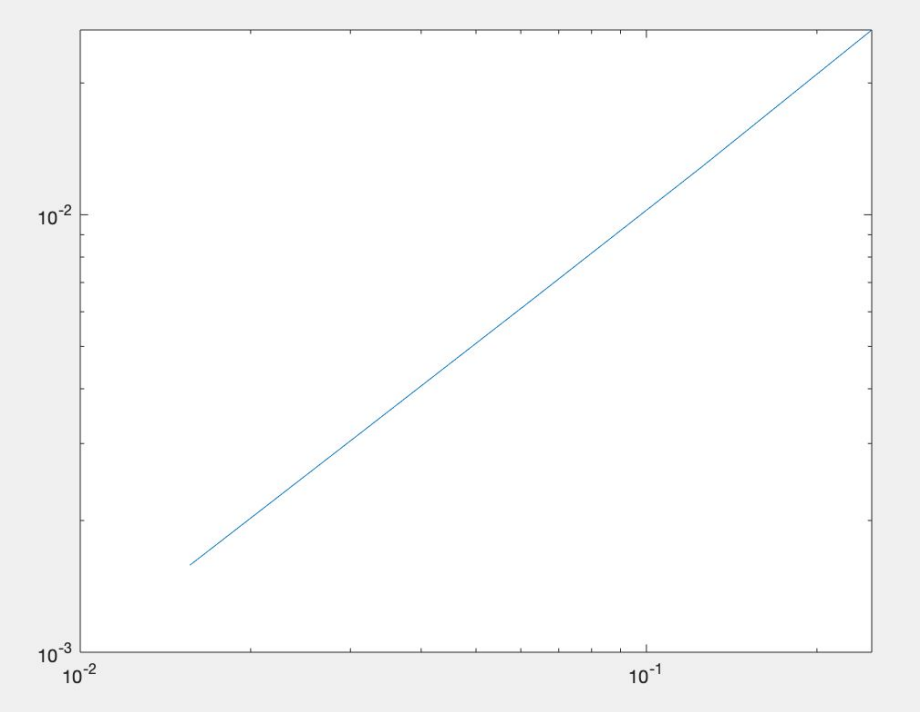
- The current solution, Y^n,
- The current time, t^n,
- The time step, Δt,
- The function $f(y,t)$,
- The partial derivative with respect to y of $f(y,t)$, $\frac{\partial f(y,t)}{\partial y}$,

and returns Y^{n+1}. This function will call your `Newtons_Method` function.

```
function [T,Y,err] = Backward_Euler(y0,t0,T,dt,fyt,Fdy)
T = linspace(t0,T,1/dt);
n = length(T);
Y = zeros(1,n);
Y(1) = y0;
for i = 2:n
    Y(i) = Backward_Euler_Step(Y(i-1),T(i),dt,fyt,Fdy);
End
      err = abs(Y(end) - exp((-T(end)^2)/2));
end

function Ynext = Backward_Euler_Step(Yn,tn,dt,fyt,Fdy)
maxiter = 1000;
tol = 1e-6;
G = @(y) y-Yn-dt*fyt(y,tn);
Gdy = @(y) 1-dt*Fdy(y,tn)
Ynext = newtons(maxiter,G,Gdy,tol,Yn);
end
```

Repeat section 4.3 with your **Backward_Euler** function. Even though you are using Newton's method to solve a linear problem, this will make your code more robust since it can be applied to non-linear IVPs.

```
dts = [1/4 1/8 1/16 1/32 1/64];
errs = zeros(1,length(dts));
for i = 1:length(dts)
    [~,~,err] = Backward_Euler(1,0,1,dts(i),@ode,@partial);
    errs(i) = err;
end
loglog(dts,errs);

function val = ode(yn,tn)
val = -tn * yn;
end

function val = partial(yn,tn)
val = 0;
end
```



The rate of convergence appears to be O(t), as the timestep decreases by half, the error also decreases by half. To prove this, I looked at the ratio of the error between err(n) and err(n+1) (n = 1, 2, 3, 4) and found that all of them had a ratio of 2, meaning that all the errors decreased by a factor of 2 as the timesteps were halved.

```
for i = 1:4
errs(i)/errs(i+1)
end
```

If errs(i+1) = .5 errs(i) and the timestep at i+1 is half of the timestep at i, then the ratio of errs(i)/errs(i+1) is 2 (errs(i) cancel leaving 1/.5) if the convergence is O(t).

Output from the command line:

```
ans =
    2.0595
ans =
    2.0186
ans =
    2.0070
ans =
    2.0032
```