```
#### ODE Steppers for y'(t) = f(t, y(t)) #####

##### Secondary Methods ######

function Derivative(x, f, d=1, h=10.0^-8)
    """
    Estimate the derivative of f at x using the center
    finite difference technique with step size h.

    x and f can be vector valued, in which case
    d must be provided. d should be a list of only
    zeros and ones.

    For example, if x and f(x) are vectors of length four
    and we want the derivative with respect to the second
    component in x, then we call Derivative(x, f, [0, 1, 0, 0]).
    """
    (f(x+h*d) - f(x-h*d)) / (2*h)
end

function NewtonsMethod(x, f, error_tol=10.0^-10)
    """
    Estimate the root of f starting from x.

    x and f(x) are vector valued. Newton's method becomes

    \vec{x}_{n+1} = \vec{x}_{n} - inv(Jacobian) * \vec{f(\vec{x}_n)}

    I create the Jacobian by estimating the derivatives
    using a center finite difference method.
    """
    h, n = 10.0^-5, length(x)
    J = zeros(n, n)
    for j=1:n
        d = zeros(n)
        d[j] = 1
        J[:, j] = Derivative(x, f, d)
    end
    count = 0
    while norm(f(x)) > error_tol
        count += 1
        x -= inv(J) * f(x)
        if count > 50
            println("Newton's Method took unusually long so it was terminated prematurely.")
            return x
        end
    end
    x
end

##### End Secondary Methods #####

###### Single Steppers ######

## Step functions that do not include an error estimate ##
# Call these with Stepper(f, method, t0, y0, tf, h)

function EulerStep(f, t, y, h)
    h * f(t, y)
end

function ImprovedEulerStep(f, t, y, h)
    f0 = f(t, y)
    (h / 2.0) * (f0 + f(t + h, y + h * f0))
end
```

```
function BackwardEulerStep(f, t, y, h)
    """ Implicit Backward Euler Step """
    #Use Euler step as starting point for NewtonsMethod.
    NewtonsMethod(f(t, y), f1 -> h*f(t+h, y+f1) - f1)
end

function GL2Step(f, t, y, h)
    """ Implicit Gauss-Legendre Order 2 Method """
    #Use Euler step as starting point for NewtonsMethod.
    NewtonsMethod(f(t, y), f1 -> h*f(t+h/2.0, y+f1/2.0) - f1)
end

function MidpointStep(f, t, y, h)
    h * f(t + h / 2, y + (h / 2) * f(t, y))
end

function CrankNicholsonStep(f, t, y, h)
    """ Second order implicit """
    f1 = f(t, y)
    NewtonsMethod(f1, f2 -> (h/2.0) * (f1 + f(t+h, y+f2)) - f2)
end

function RK4Step(f, t, y, h)
    f1 = f(t, y)
    f2 = f(t + h / 2, y + (h / 2) * f1)
    f3 = f(t + h / 2, y + (h / 2) * f2)
    f4 = f(t + h, y + h * f3)
    (h / 6) * (f1 + f4 + 2 * (f2 + f3))
end

function Stepper(f, method, t0, y0, tf, h)
    """
    The step function `method` returns just the step,
    not an error estimate.
    """
    ts, ys = Float64[], Vector{Float64}[]
    while t0 <= tf
        push!(ts, t0)
        push!(ys, y0)
        y0 += method(f, t0, y0, h)
        t0 += h
    end
    ts, ys
end

## End step functions that do not include an error estimate ##

## Step functions that include an error estimate ##
# Call these with AdaptiveStepper(f, method, t0, y0, tf)

function SimpleErrorStep(f, t, y, h)
    """ Compare implicit Crank Nicholson to Midpoint to get error """
    step = CrankNicholsonStep(f, t, y, h)
    step, norm(step - MidpointStep(f, t, y, h))
end

function RKTrapStep(f, t, y, h)
    """ Implicit Runge-Kutta Trapazoidal Method """
    f1 = h * f(t, y)
    f2 = NewtonsMethod(f1, f2 -> h*f(t+h, y+f1/2.0+f2/2.0) - f2)

    soln = f1/2.0 + f2/2.0
    soln, norm(soln - f1)
end
```

```
function DopriStep(f, t, y, h)
    """ Dormand-Prince Method """
    f1 = h * f(t, y)
    f2 = h * f(t + h/5.0, y + f1/5.0)
    f3 = h * f(t + 3*h/10.0, y + 3*f1/40.0 + 9*f2/40.0)
    f4 = h * f(t + 4*h/5.0, y + 44*f1/45.0 - 56*f2/15.0 + 32*f3/9.0)
    f5 = h * f(t + 8*h/9.0, y + 19372*f1/6561.0 - 25360*f2/2187.0 + 64448*f3/6561.0 - 212*f4/729.0)
    f6 = h * f(t + h, y + 9017*f1/3168.0 - 355*f2/33.0 + 46732*f3/5247.0 + 49*f4/176.0 -
    5103*f5/18656.0)
    soln = 35*f1/384.0 + 500*f3/1113.0 + 125*f4/192.0 - 2187*f5/6784.0 + 11*f6/84.0
    f7 = h * f(t + h, y + soln)
    alt = 5179*f1/57600.0 + 7571*f3/16695.0 + 393*f4/640.0 - 92097*f5/339200.0 + 187*f6/2100.0 +
    f7/40.0

    soln, norm(soln - alt)
end

function CarpStep(f, t, y, h)
    """ Cash-Karp Method """
    f1 = h * f(t, y)
    f2 = h * f(t + h/5.0, y + f1/5.0)
    f3 = h * f(t + 3*h/10, y + 3*f1/40.0 + 9*f2/40.0)
    f4 = h * f(t + 3*h/5.0, y + 3*f1/10.0 - 9*f2/10.0 + 6*f3/5.0)
    f5 = h * f(t + h, y - 11*f1/54.0 + 5*f2/2.0 - 70*f3/27.0 + 35*f4/27.0)
    f6 = h * f(t + 7*h/8.0, y + 1631*f1/55296 + 175*f2/512 + 575*f3/13824 + 44275*f4/110592.0 +
    253*f5/4096.0)
    soln = 37*f1/378.0 + 250*f3/621.0 + 125*f4/594.0 + 512*f6/1771.0
    alt = 2825*f1/27648.0 + 18575*f3/48384.0 + 13525*f4/55296.0 + 277*f5/14336.0 + f6/4.0

    soln, norm(soln - alt)
end

function RKFStep(f, t, y, h)
    """ Runge-Kutta-Fehlberg Method """
    f1 = h * f(t, y)
    f2 = h * f(t + h/4.0, y + f1/4.0)
    f3 = h * f(t + 3*h/8.0, y + 3*f1/32.0 + 9*f2/32.0)
    f4 = h * f(t + 12*h/13.0, y + 1932*f1/2197.0 - 7200*f2/2197.0 + 7296*f3/2197.0)
    f5 = h * f(t + h, y + 439*f1/216.0 - 8*f2 + 3680*f3/513.0 - 845*f4/4104.0)
    f6 = h * f(t + h/2.0, y - 8*f1/27.0 + 2*f2 - 3544*f3/2565.0 + 1859*f4/4104.0 - 11*f5/40.0)
    soln = 16*f1/135.0 + 6656*f3/12825.0 + 28561*f4/56430.0 - 9*f5/50.0 + 2*f6/55.0
    alt = 25*f1/216.0 + 1408*f3/2565.0 + 2197*f4/4104.0 - f5/5.0

    soln, norm(soln - alt)
end

function AdaptiveStepper(f, method, t0, y0, tf, error_tol=10.0^-7)
    """
    The step function `method` must return the tuple,
    (step, error estimate).
    """
    ts, ys = Float64[t0], Vector{Float64}[y0]
    h, totalerror = 0.001, 0
    while t0 < tf
        if h < 10.0^-8 && h < tf - t0
            println("Step size effectively zero at t = ", t0)
            h = 0.001
        else
            if h > tf - t0
                h = tf - t0
            end
            m, error_est = method(f, t0, y0, h)
            if error_est > error_tol
                h *= .75
```

```
            else
                t0 += h
                y0 += m
                push!(ts, t0)
                push!(ys, y0)
                if error_est < error_tol / 10.0
                    h *= 1.2
                end
                totalerror += error_est
            end
        end
    end
    println("Estimated upper bound on total error of ", method, ": ", totalerror)
    ts, ys
end

## End step functions that include an error estimate ##

##### End Single Steppers #####

##### Multi Steppers ######
#Call these with MultiStepper(f, method, t0, y0, tf, h)

function AdamsBash2(f, t, y, h)
    """ Adams-Bashforth Order 2 explicit method """
    y0n = Array[y, y + RK4Step(f, t, y, h)]
    function g(t, yn)
        #yn = Array[y0, y1]
        y0, y1 = yn
        y2 = y1 + h * (1.5 * f(t + h, y1) - 0.5 * f(t, y0))
        Array[y1, y2]
    end
    g, y0n
end

function AdamsBash3(f, t, y, h)
    """ Adams-Bashforth Order 3 explicit method """
    y1 = y + RK4Step(f, t, y, h)
    y2 = y1 + RK4Step(f, t+h, y1, h)
    y0n = Array[y, y1, y2]
    function g(t, yn)
        #yn = Array[y0, y1, y2]
        y0, y1, y2 = yn
        y3 = y2 + h * (23.0 / 12.0 * f(t+2*h, y2) - 4.0 / 3.0 * f(t+h, y1) + 5.0 / 12.0 * f(t, y0))
        Array[y1, y2, y3]
    end
    g, y0n
end

function AdamsBash4(f, t, y, h)
    """ Adams-Bashforth Order 4 explicit method """
    y1 = y + RK4Step(f, t, y, h)
    y2 = y1 + RK4Step(f, t+h, y1, h)
    y3 = y2 + RK4Step(f, t+2*h, y2, h)
    y0n = Array[y, y1, y2, y3]
    function g(t, yn)
        #yn = Array[y0, y1, y2, y3]
        y0, y1, y2, y3 = yn
        y4 = y3+h * (55.0 / 24.0 * f(t+3*h, y3) - 59.0 / 24.0 * f(t+2*h, y2) + 37.0 / 24.0 * f(t+h, y1)
        - 3.0 / 8.0 * f(t, y0))
        Array[y1, y2, y3, y4]
    end
    g, y0n
end
```

```
function AdamsMoult2(f, t, y, h)
    """ Adams-Moulton Order 2 implicit method """
    y1 = y + RK4Step(f, t, y, h)
    y0n = Array[y, y1]
    function g(t, yn)
        y0, y1 = yn
        f0, f1 = f(t, y0), f(t+h, y1)
        y2 = NewtonsMethod(y1, y2 -> y1+h*(5*f(t+2*h, y2)/12.0 + 2*f1/3.0 - f0/12.0) - y2)
        Array[y1, y2]
    end
    g, y0n
end

function AdamsMoult3(f, t, y, h)
    """ Adams-Moulton Order 3 implicit method """
    y1 = y + RK4Step(f, t, y, h)
    y2 = y1 + RK4Step(f, t+h, y1, h)
    y0n = Array[y, y1, y2]
    function g(t, yn)
        y0, y1, y2 = yn
        f0, f1, f2 = f(t, y0), f(t+h, y1), f(t+2*h, y2)
        y3 = NewtonsMethod(y2, y3 -> y2+h*(3*f(t+3*h, y3)/8.0 + 19*f2/24.0 - 5*f1/24.0 + f0/24.0)-y3)
        Array[y1, y2, y3]
    end
    g, y0n
end

function AdamsMoult4(f, t, y, h)
    """ Adams-Moulton Order 4 implicit method """
    y1 = y + RK4Step(f, t, y, h)
    y2 = y1 + RK4Step(f, t+h, y1, h)
    y3 = y2 + RK4Step(f, t+2*h, y2, h)
    y0n = Array[y, y1, y2, y3]
    function g(t, yn)
        y0, y1, y2, y3 = yn
        f0, f1, f2, f3 = f(t, y0), f(t+h, y1), f(t+2*h, y2), f(t+3*h, y3)
        y4 = NewtonsMethod(y3, y4 -> y3+h*(251*f(t+4*h, y4)/720.0+646*f3/720.0 - 264*f2/720.0 +
        106*f1/720.0 - 19*f0/720.0)-y4)
        Array[y1, y2, y3, y4]
    end
    g, y0n
end

function MultiStepper(f, method, t0, y0, tf, h)
    """
    method returns g(t, yn), y0n with y0n is
    an array  of previous values of y.
    """
    g, y0n = method(f, t0, y0, h)
    #y0n = Array[y0, y1, ..., yn]
    ts, ys = Float64[], Vector{Float64}[]
    while t0 <= tf + h
        push!(ts, t0)
        push!(ys, y0n[1])
        y0n = g(t0, y0n)
        t0 += h
    end
    ts, ys
end

##### End Multi Steppers #####


#### Test ####
function func(t, y)
```

```
    y1, ydot = y
    [ydot, -t*y1]
end

t, tmax, h = 0.0, 20, 0.01
y0 = [-1.0, 0.0]

#xs, ys = MultiStepper(func, AdamsMoult4, t, copy(y0), tmax, h)
xs, ys = Stepper(func, GL2Step, t, copy(y0), tmax, h)
#xs, ys = AdaptiveStepper(func, RKTrapStep, t, copy(y0), tmax, h)
figure(0)
plot(xs, [ys[i][1] for i=1:length(ys)], "o-", label="Position")
plot(xs, [ys[i][2] for i=1:length(ys)], "x--", label="Velocity")
legend()


xs, ys = AdaptiveStepper(func, RKFStep, t, copy(y0), tmax)
figure(1)
plot(xs, [ys[i][1] for i=1:length(ys)], "o-", label="Position")
plot(xs, [ys[i][2] for i=1:length(ys)], "x--", label="Velocity")
legend()
```