Lab III : Taylor expansion and projectile motion

Example 1 : Taylor expansion
In the previous lab hour, you wrote a function called `factorial`. Using this function now, we are going to write another function that enables us to visualize how fast the Taylor expansion converges to the exponential.

The Taylor expansion of any function around a number, provided that it is convergent, is a series that can be written in the following way

$$f(x-a) = \sum_{i=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n. \tag{1}$$

In this example, we are going to use the simple exponential function, e^x and see how fast the terms in the sum

$$e^x = \sum_{i}^{N} \frac{x^n}{n!} \tag{2}$$

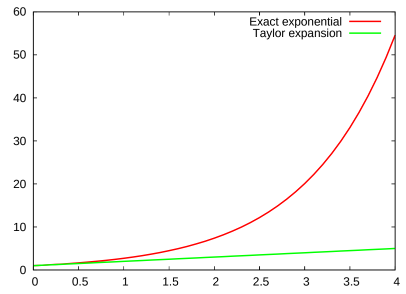
converges as a function of `N`.

```
hande@p439a:~$ cat exponential.m

## Find the polynomial approximation to the exponential to an arbitrary order.
function f=exponential(n,x)

  hold off
  f=ones(size(x));
  plot(x,exp(x),';Exact exponential;',x,f,';Taylor expansion;');
  pause
  hold on

  for order=1:n

    f+=(1./factorial(order))*x.^order;
    plot(x,f,';;');
    err=sum((exp(x)-ex)./exp(x))/length(x)
    pause

  endfor


endfunction
```

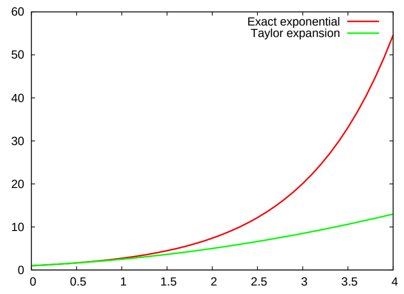
The `for` loop in the above exercise will progressively add to the output variable `ex` a new term from the Taylor expansion. It then plots the improved expansion together with the exact exponential at every step, pausing and waiting for the user to press any key before going on to the next step in the loop. If you now call your `exponential` function from the `Octave` prompt now as in the following

```
octave:1> x=0:0.1:5;
octave:2> ex=exponential(6,x);
err = 0.60690
err = 0.43444
err = 0.28896
err = 0.17754
err = 0.10052
err = 0.052450
```

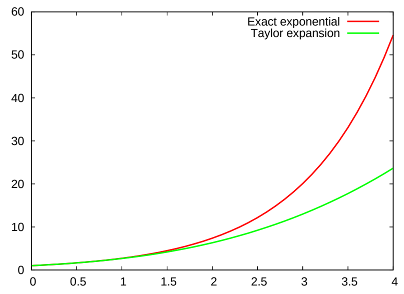
you should see the following progression of the expansion.



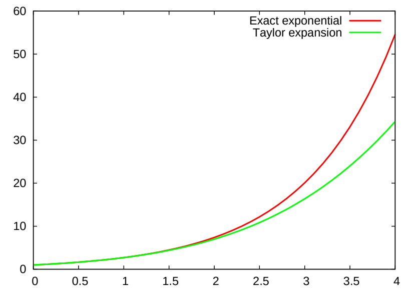
n=1



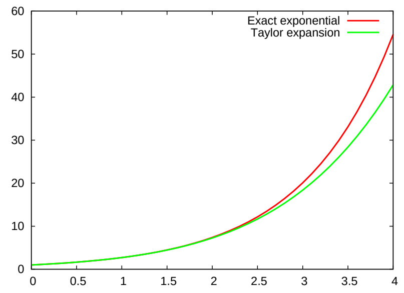
n=2



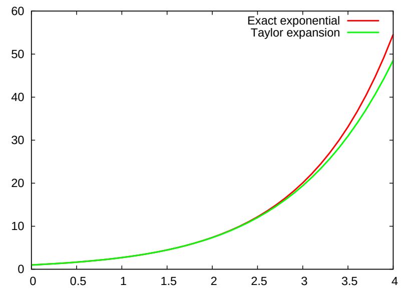
n=3



n=4



n=5



n=6

The function also outputs an error value, which is a measure of the sum of the deviation of the exact function from the expansion at every point divided by the number of points. As expected the error goes down as we increase the order of the expansion.

Because the function e^x is *entire*, meaning that it can be Taylor-expanded at any point in its domain, this expansion will work for any range of `x`. How about $\log(x)$?

Example 2 : Projectile motion in two dimensions, the cannon ball in the absence of air drag
In class, we saw a simple example of motion in one-dimension. Already at this level, the problem demonstrates sufficient analytical complexity that it pays off to resort to numerical methods. In this example, we will increase the comlexity by carrying the problem over to two dimensions and also adding more realistic effects due to air friction. We will develop our problem in stages. Let's start with the simplest case where we ignore air friction.

Consider a cannonball launched at a given angle θ_0. Ignoring air friction for the moment, we may write the equations of motion as follows :

$$\begin{aligned} \frac{d^2x}{dt^2} &= 0 \\ \frac{d^2y}{dt^2} &= -g \end{aligned} \tag{3}$$

which should be familiar from freshman kinematics. The equations above are of second order this time, which we can of course convert to first order easily by taking the analytical derivative but because the later development of the problem will involve dealing with coordinates directly, we'll keep them as second order.

While second order differential equations may also be tackled using the Euler method, it's often easier to write them as a combination of first order equations. Eq. 3 may then be cast in the following form

$$\begin{aligned} \frac{dx}{dt} &= v_x \qquad & \frac{dv_x}{dt} &= 0 \\ \frac{dy}{dt} &= v_y \qquad & \frac{dv_y}{dt} &= -g \end{aligned} \tag{4}$$

We may now write Eq. 4 in the finite difference (Euler) form.

$$\begin{aligned} x_{n+1} &= x_n + v_{x,n}\Delta t \\ v_{x,n+1} &= v_{x,n} \\ y_{n+1} &= y_n + v_{y,n}\Delta t \\ v_{y,n+1} &= v_{y,n} - g\Delta t \end{aligned} \tag{5}$$

Unlike the bicycle example, at every step of our loop, we will be incrementing four arrays this time. They each depend on the values of themselves and others in the previous step. The other big difference between this and the one-dimensional example is that here, it's less obvious when we should terminate the loop. Previously, we were given a fixed number of steps and we were thus able to write a `for` loop containing that many steps. In the present example, though, we know physically that it's meaningful to continue the iterations until the y-coordinate of the trajectory becomes less than zero. In this case, it is more appropriate to use a `while` loop since we only know the condition on the system and not the number of steps.

With these in mind, we cast Eq. 5 into `Octave` code, following this outline :

- Declare suitable variables :

Constant	Symbol	Value
Gravitational acceleration	`g`	9.8 m/sec^2
Launch velocity	`v0`	700 m/sec
Launch angle	`theta0`	45^o (see below)
Time step	`dt`	0.1 sec

- Using the constants, initialize the arrays that you will be building up, namely `x, y, vx, vy`.
- Initialize the loop index, `n`.
- Iterate the four variables until the `while` condition is no longer satisfied.
- Plot `y` versus `x`.

```
hande@p439a:~$ cat cannon_nodrag.m

## Motion of the cannonball without air drag
g  = 9.8;          ## m/sec^2
v0 = 700;          ## m/sec
theta0 = 45;       ## degrees
dt=0.1;            ## sec

## Convert degrees to radians
theta0 *= pi/180; ## rad
x=0;  y=eps;
vx=v0*cos(theta0); vy=v0*sin(theta0);
n=1;    ## Initialize the loop index

## Run the loop until the vertical component is   smaller than zero
while ( y(n)>0 );
  x=[x;x(n)+vx(n)*dt];
  y=[y;y(n)+vy(n)*dt];
  vx=[vx;vx(n)];
  vy=[vy;vy(n)-g*dt];
  n++;
endwhile

plot(x/1000,y/1000,';no drag;')
```

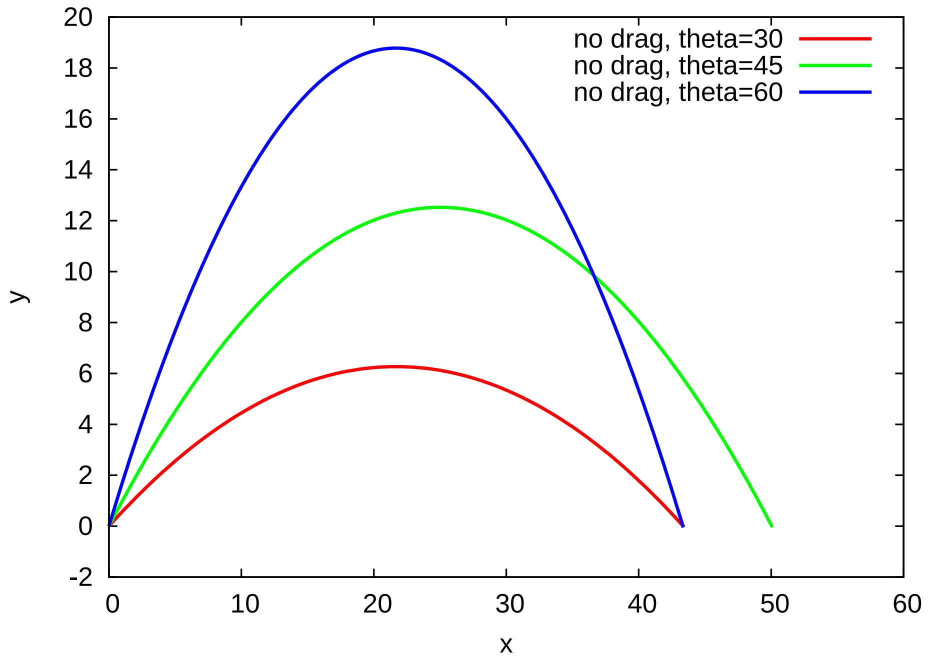
In the piece of code above, there are a few points (highlighted in red) that need to be taken care of.

1. Degreed in `Octave` need to be in radians. You can see this by going to your `Octave` prompt and trying to evaluate `sin(45)`. You'll see that the number that you obtain has nothing to do with `sin(pi/4)`. However, it's still better to let the user enter the angle in degrees, because users will usually have a better intuition into degrees than radians. You can do the conversion internally hidden away from the user.
2. While initializing `y`, we used `y=eps`, where `eps` is a very small number ($\mathcal{O}(10^{-16})$) in `Octave`. Although it's zero for all intents and purposes, it still enables the loop condition to be satisfied. If, on the other hand, we initialize `y` to exactly zero, the loop never starts.

3. Unlike the `for` loop, the `while` loop doesn't increment the loop index itself. We must therefor increment `n` manually. `n++` is a shortcut for `n=n+1`.

4. In the end, we plot the distances in km's.

You can run the above code for a set of different angles and see the difference in the corresponding trajectories. You can verify that the maximum trajectory is obtained for $\theta = 45^o$. You will see that this is no longer the case when drag is involved. You can also superimpose trajectories corresponding to different angles on the same plot using the `hold` command that you've learnt a while ago.


no drag, theta=30
no drag, theta=45
no drag, theta=60
y
x