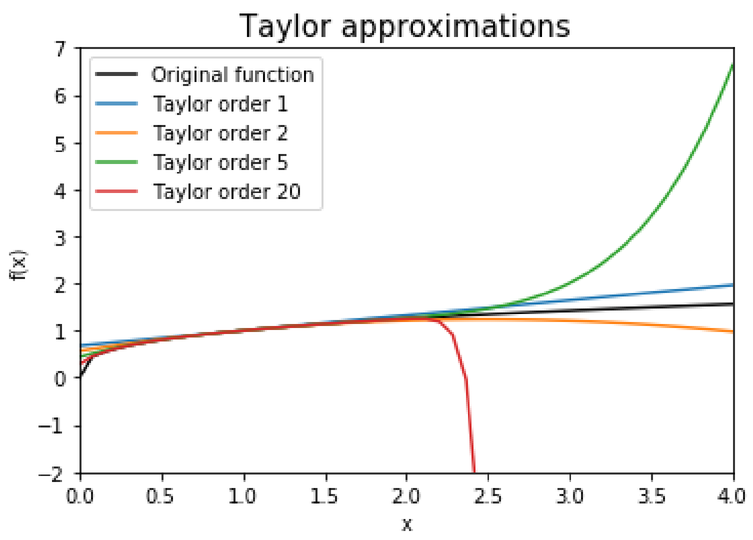
I did this problem set in collaboration with **Germán Sánchez Arce** and **Joan Alegre Cantón**.

Question 1. Function Approximation: Univariate

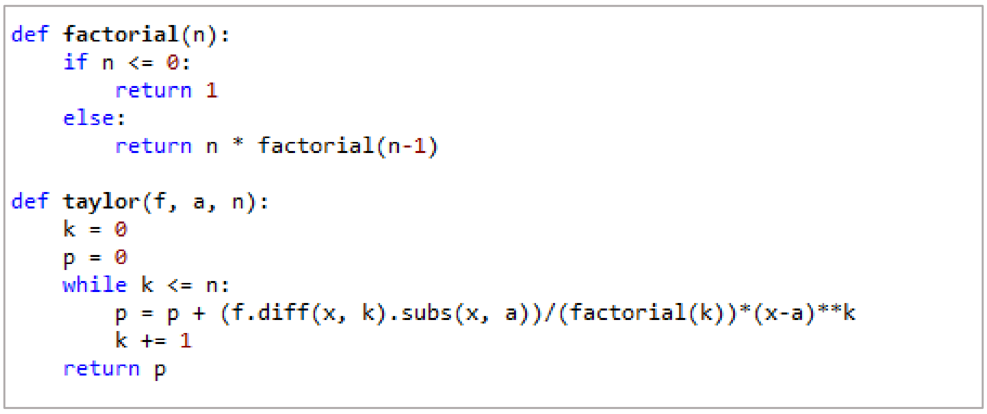
1. Approximate f(x) = $x^{0.321}$ with a Taylor series around x = 1, of order 1, 2, 5 and 20:



The Taylor expansion is a local method that approximates the function at some given point, in this case x = 1. What we can see in the previous graph is how the approximation adjusts better or worse to the real function, depending on the order of the Taylor expansion, in a neighborhood around 1.

Moreover, what we can see is that not for a higher order we end up with a better approximation of the real function. This means that, for using this kind of local approximations we need to take into account some characteristics or making assumptions about the behavior of the real function, in order to minimize the approximation error for points of x far away from the considered one. In this example, we can see that a higher order implies that the approximation error increases sharply. However, since we don't have any singularity at x = 1, all this Taylor expansions are doing a well local approximation of the function.

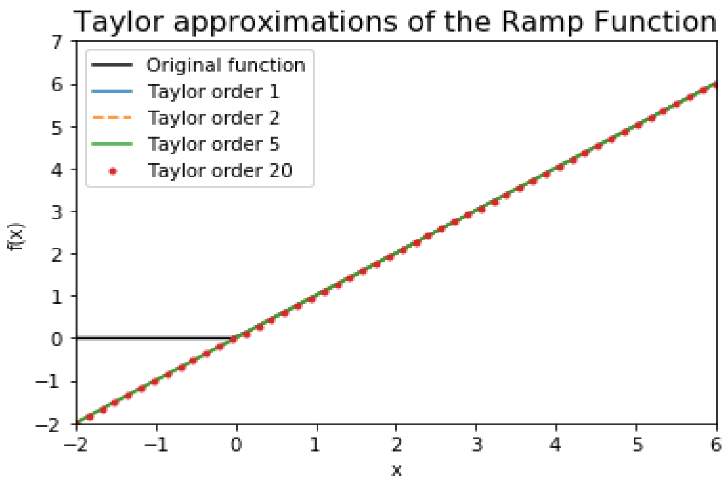
In order to do these approximations, we used the following code, which allows us to obtain a Taylor approximation of grade n, of a given function f, around the point a.



```python
def factorial(n):
    if n <= 0:
        return 1
    else:
        return n * factorial(n-1)

def taylor(f, a, n):
    k = 0
    p = 0
    while k <= n:
        p = p + (f.diff(x, k).subs(x, a))/(factorial(k))*(x-a)**k
        k += 1
    return p
```

2. Approximation of the ramp function with a Taylor series:

Using the function we have shown in the previous exercise, we get the following plot where we are representing the Taylor approximations of the ramp function.



In this case, the Taylor approximation works well for the points of x larger than 1. However, in this special case we can see how the approximation error becomes larger for values of x smaller than zero, where the function presents a kink.

Moreover, what we can see is that increasing the order of the Taylor expansion does not report additional information about the function, so it is sufficient doing a Taylor approximation of order 1.

Given that, our conclusion is that, Taylor expansion works well for local approximations at points where the function presents a well-behavior (without kinks or singularities). Moreover, we need to take into account some characteristics of the real function (or certain assumptions) in order to decide the order of the Taylor expansion that we'll use.

3. Approximation of three functions with evenly spaced interpolation nodes with monomials, Chebyshev interpolation nodes with monomials, and Chebyshev interpolation nodes with Chebyshev polynomial:

- Evenly spaced interpolation nodes with monomials and errors:

  For this part of the exercise, what we have done is a simple polynomial interpolation with monomials of grade 3, 5 and 10. For doing that, we use the predefined function polyfit (imported from numpy), where we only need to specify: the points x where it will interpolate the function, the function we are interested in (called y) and the grade for the interpolation.

  This function polyfit give us the coefficients for the polynomial, and with these coefficients, the predefined function polyval give us the value of the monomial at the x points we have previously defined (and that are equally spaced).

  The code for the three interpolations is the following:

```
pol3 = np.polyfit(x, y, 3)
val3 = np.polyval(pol3, x)

pol5 = np.polyfit(x, y, 5)
val5 = np.polyval(pol5, x)

pol10 = np.polyfit(x, y, 10)
val10 = np.polyval(pol10, x)
```

  For plotting the approximation error, we only did the absolute value of the difference between the original function and the values of the approximations (those given by the polyval function):

```
error1 = abs(y-val3)
error3 = abs(y-val5)
error5 = abs(y-val10)
```

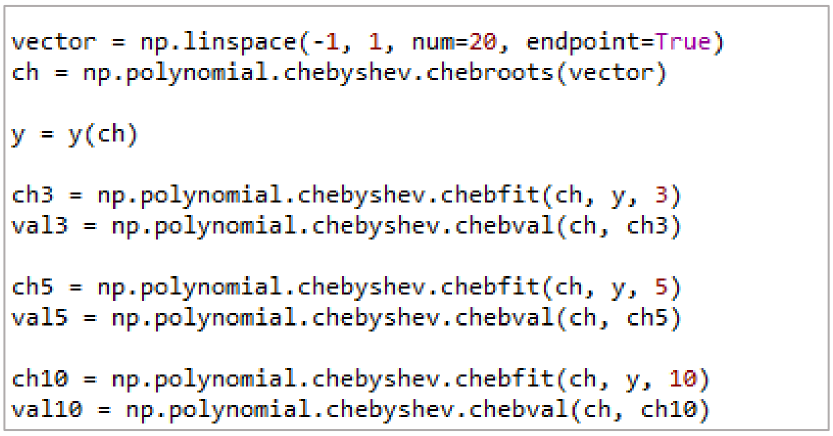
- Chebyshev interpolation nodes and monomials:

  For this part, the only that changes with respect to the previous part of the exercise, is that now the x points where we interpolate the function are not evenly spaced. We use the Chebyshev nodes, which are given by the following predefined function:

```
vector = np.linspace(-1, 1, num=20, endpoint=True)

ch = np.polynomial.chebyshev.chebroots(vector)
```

- Chebyshev interpolation nodes and Chebyshev polynomial:

  Now, we not only use the Chebyshev nodes that we have used in the previous part, but also the Chebyshev polynomial, instead of normal polynomials of grade 2, 5 and 10. For doing that, we use the following code:

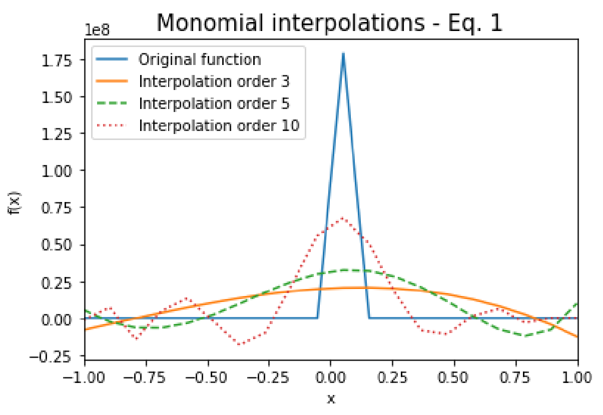


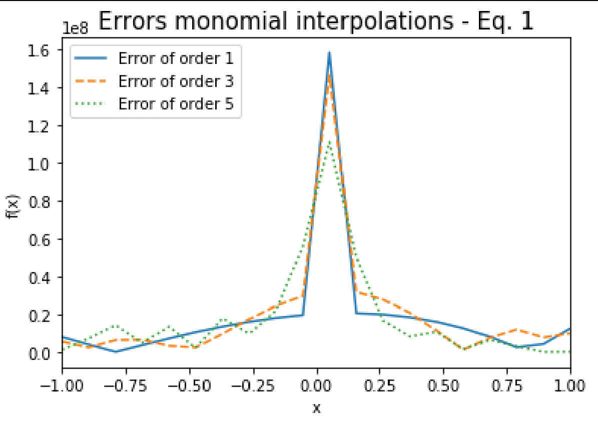
```python
vector = np.linspace(-1, 1, num=20, endpoint=True)
ch = np.polynomial.chebyshev.chebroots(vector)

y = y(ch)

ch3 = np.polynomial.chebyshev.chebfit(ch, y, 3)
val3 = np.polynomial.chebyshev.chebval(ch, ch3)

ch5 = np.polynomial.chebyshev.chebfit(ch, y, 5)
val5 = np.polynomial.chebyshev.chebval(ch, ch5)

ch10 = np.polynomial.chebyshev.chebfit(ch, y, 10)
val10 = np.polynomial.chebyshev.chebval(ch, ch10)
```

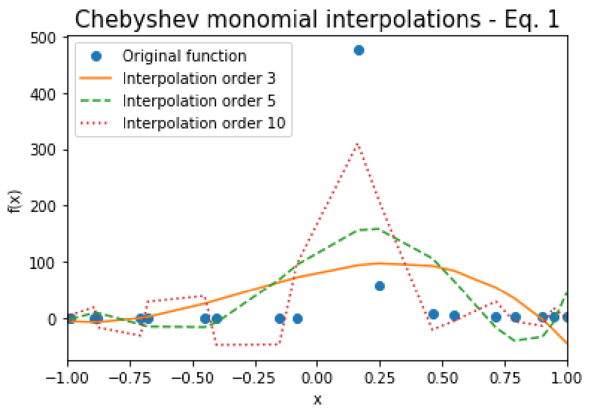
Here, as in the case of the function polyfit (or polyval), chebfit is a predefined function of Python that give us the coefficients of the Chebyshev polynomial (and chebval give us the value of the Chebyshev polynomial with these coefficients at the x points corresponding with the Chebyshev nodes).

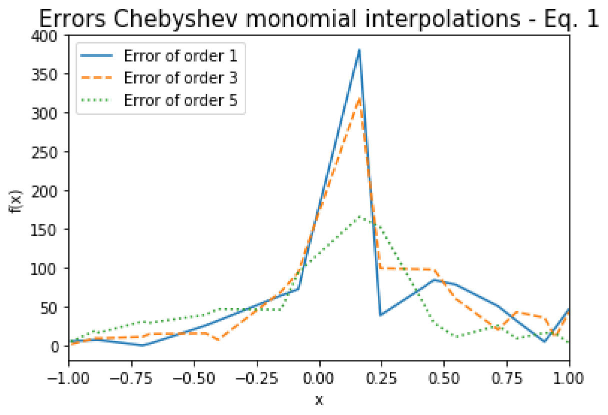
In what follows, there are plotted the representations of all this kind of approximations and their corresponding errors.

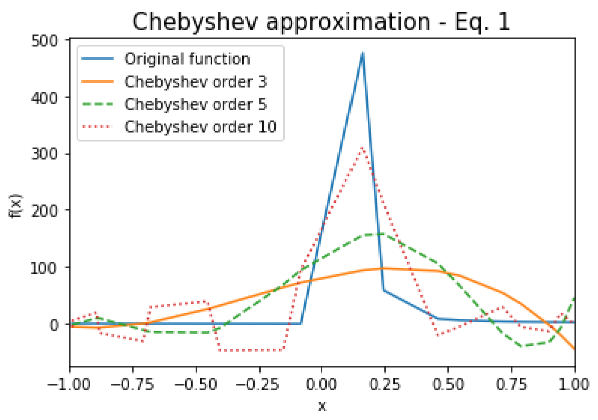
First function: $e^{1/x}$

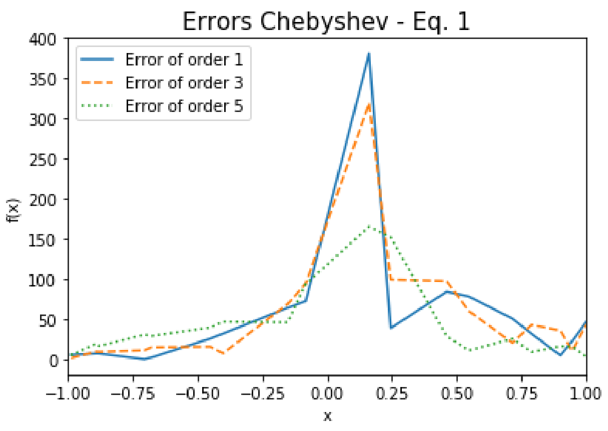




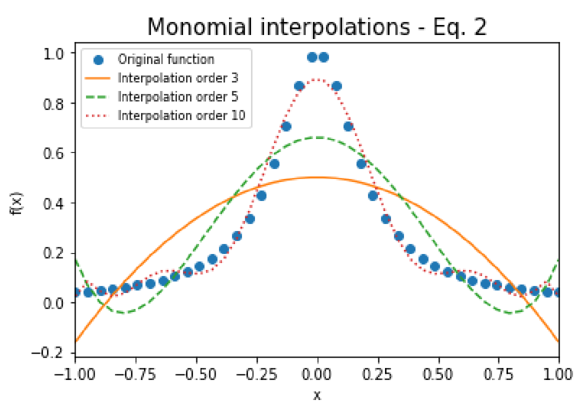


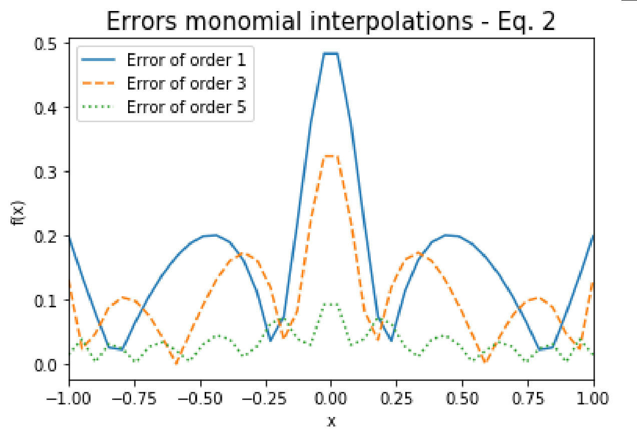


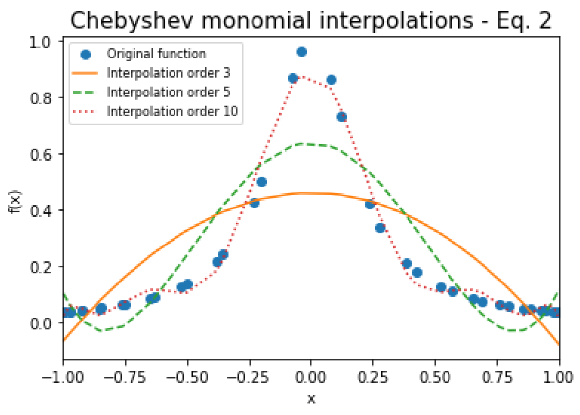


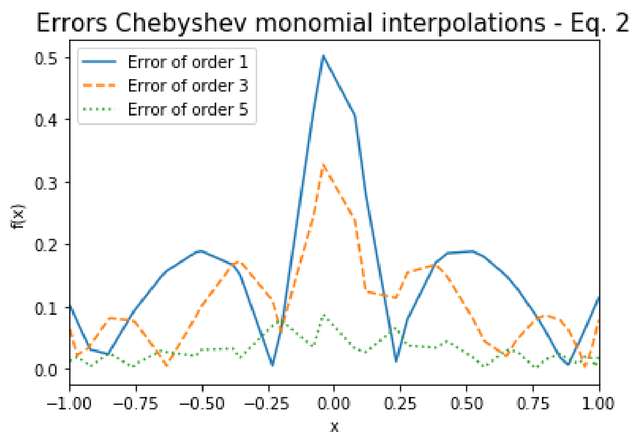


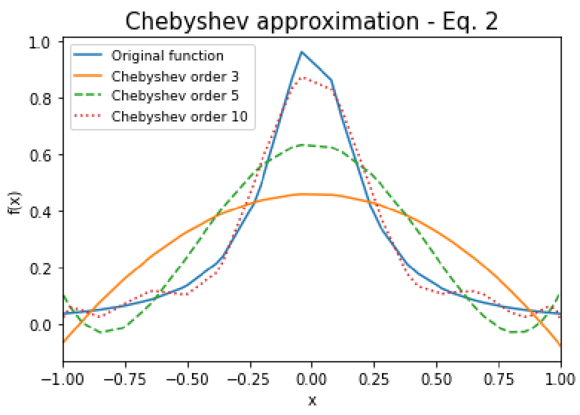
Second function: Runge function

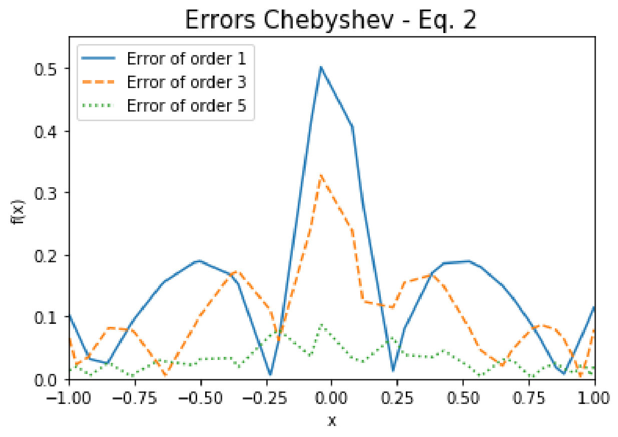




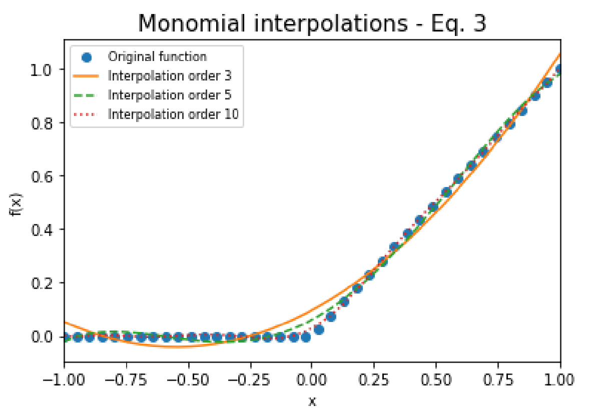


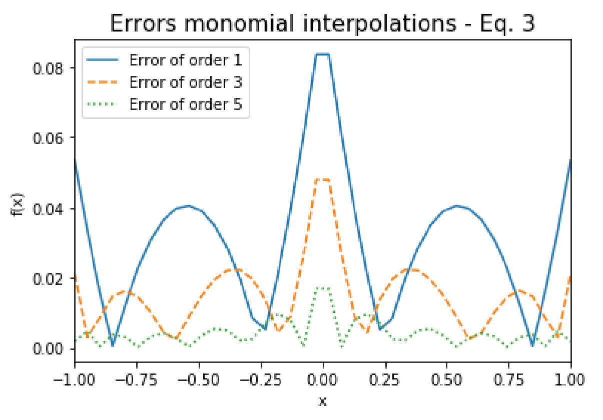


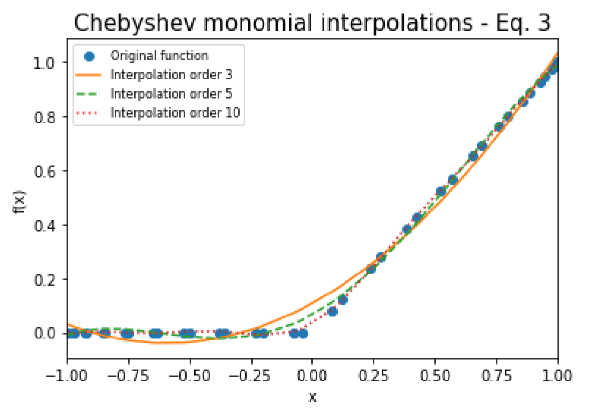


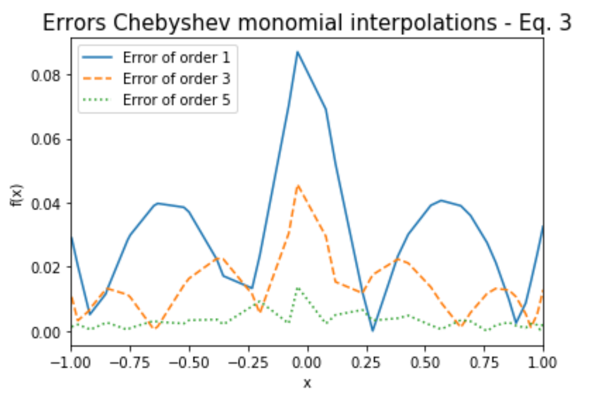


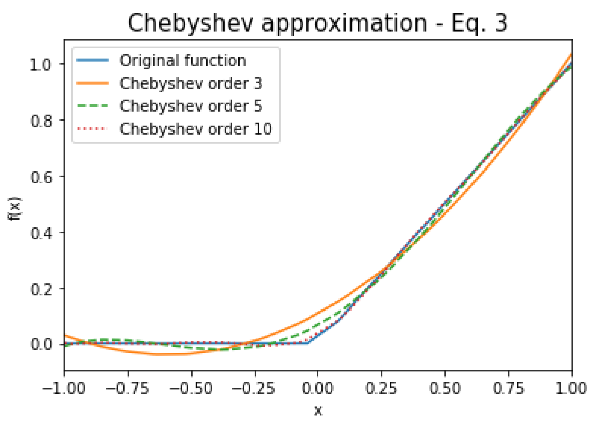
Third function: ramp function

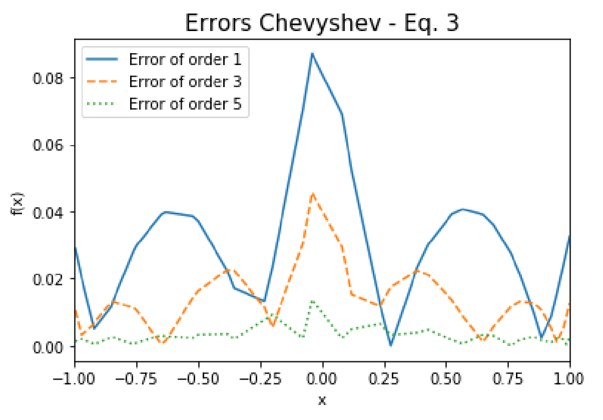












The first approximation is done with evenly-spaced interpolation nodes and monomials of order 3, 5 and 10. The conclusion of this kind of approximation is that, when we increase the order of the polynomial, less informative it is, this is because monomial bases are not orthogonal. Moreover, what we can see is that in the extreme points, the approximation error is bigger than in the next two kinds of approximations.

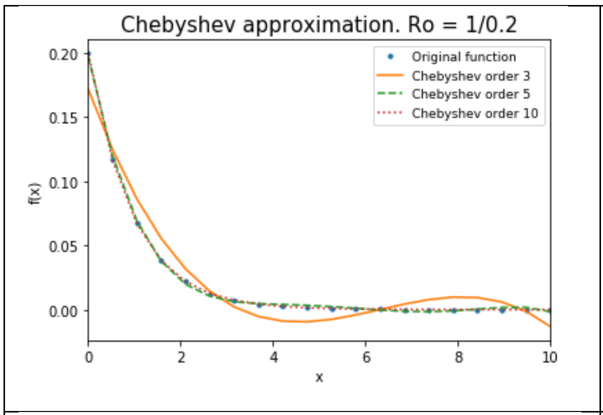
The second approximation is done also with monomials of order 3, 5 and 10, but with the Chebyshev nodes, that is, not evenly-spaced interpolation nodes. The main result we obtain with this approximation is that the errors at the extreme points are smaller than in the previous procedure. This is because the Chebyshev roots are closer to one another in the extreme points and more spaced at the center ones. In fact, what we can see is that polynomials exhibit equioscillant errors if we interpolate using the Chebyshev nodes.

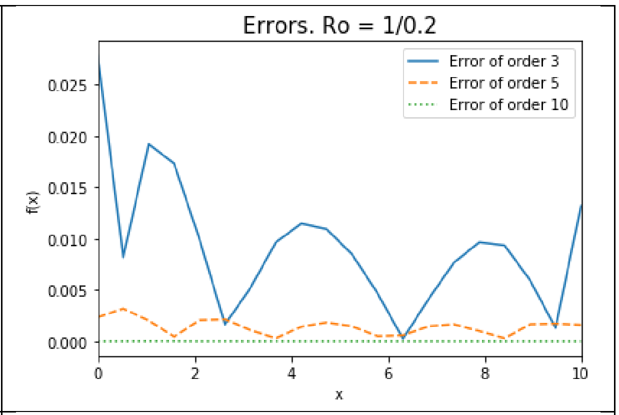
In the third approximation we use, as in the previous one, the Chebyshev roots, but now those points are interpolated with the Chebyshev polynomial instead of the monomials procedure, which is supposed to be a smoother approximation. However, when we have kinks or any singularity of the function, as we have in the first or third functions for instance, Chebyshev polynomials does not approximate better those points.

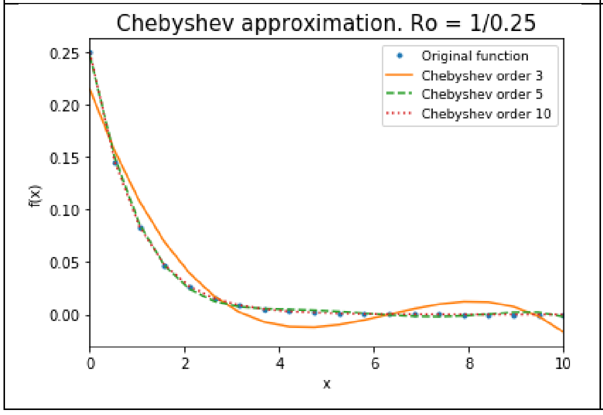
In the last two cases, we observe that the approximation errors are quite similar and smaller than the error generated with the first one. This means that the Chebyshev nodes are important in order to have a better global approximation of the function, but if we have to decide what we would use, we'll end up with the last one (because it has some nice properties like the orthogonality of the basis), always taking into account that, since it is a smooth approximation, all the kinks and singularities the function could have will not be well-approximated.

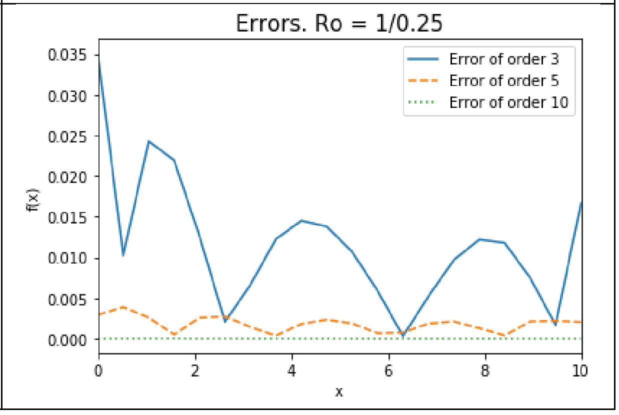
4. Approximate a probability function for two different values of a parameter.

The code used in this exercise is the same as the one used in the las part of previous exercise. The results we obtain are the following:









Question 2. Function Approximation: Multivariate

1. Show that σ is the elasticity of substitution:

f(k, h) = $\left[(1-\alpha)k^{\sigma-1/\sigma} + \alpha h^{\sigma-1/\sigma}\right]^{\sigma/(\sigma-1)}$

First, we need to calculate the marginal productivity of labor (MPL) and the marginal productivity of capital (MPK):

- MPH = $\frac{df(k,h)}{dh} = \frac{\sigma}{\sigma-1}\left[(1-\alpha)k^{\sigma-1/\sigma} + \alpha h^{\sigma-1/\sigma}\right]^{\sigma/(\sigma-1)} \frac{(1-\alpha)(\sigma-1)}{\sigma} k^{-1/\sigma}$

- MPK = $\frac{df(k,h)}{dk} = \frac{\sigma}{\sigma-1}\left[(1-\alpha)k^{\sigma-\frac{1}{\sigma}} + \alpha h^{\sigma-\frac{1}{\sigma}}\right]^{\sigma/(\sigma-1)} \frac{\alpha(\sigma-1)}{\sigma} h^{-\frac{1}{\sigma}}$

Dividing both marginal productivities we get:

$$\frac{df(k,h)}{dk}\Big/\frac{df(k,h)}{dh} = \frac{(1-\alpha)h^{1/\sigma}}{\sigma\, k^{1/\sigma}} = \frac{(1-\alpha)}{\sigma}\left(\frac{h}{k}\right)^{1/\sigma} = \frac{MPK}{MPH}$$

Now, taking logarithms:

$$\log\left(\frac{MPK}{MPH}\right) = \log\left(\frac{1-\alpha}{\sigma}\right) + \frac{1}{\sigma}\log\left(\frac{h}{k}\right) \qquad (1)$$

Since we are looking for the marginal rate of substitution of capital and labor, we need to take derivative in (1) with respect to $\log(\frac{k}{h})$ (or w.r.t. $\log(\frac{h}{k})$ and take the inverse). The result we obtain is:

$$\varepsilon_{kh} = \sigma$$

2. Obtain the labor share of an economy with CES production function.

We know that labor share is given by:

$$s = \frac{MPH\ h}{f(k,\ h)} \qquad (2)$$

Therefore:

$$\text{MPH h} = \alpha h\left[(1-\alpha)k^{\sigma-1/\sigma} + \alpha h^{\sigma-1/\sigma}\right]^{\sigma/(\sigma-1)} k^{-1/\sigma}$$

Call A to $(1-\alpha)k^{\sigma-1/\sigma} + \alpha h^{\sigma-1/\sigma}$. Following (2), we end up to the following result:

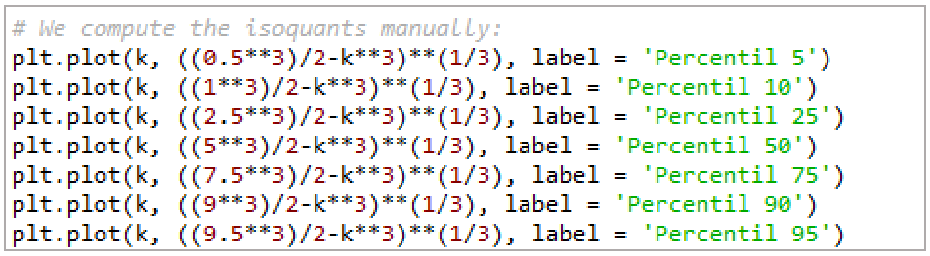
$$s = \frac{1}{A}\alpha h^{\sigma-1/\sigma}$$

3. Sorry, I couldn't do this part. I was stucked for a long time with this exercise, and some friends showed me their codes but there comes a point where I don't understand what's happening and therefore I can't "translate" their code in one that works well, but in my way of doing things.

   I know that this is (in my opinion) the most important part of this problem set. I have all the intuition behind this exercise, but my problem is the programming part. I prefer to wait and spend more days trying to understand how people have done it, even though this implies that I'll have a lower grade in this ps, rather than copy and paste a code that I really don't understand.

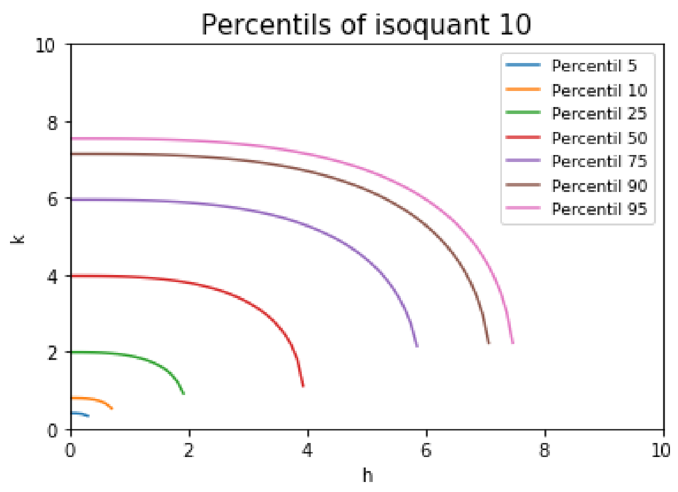
4. Plot the exact isoquants.

   The code used in this part of the exercise is not a very professional code. What we have done was basically compute the isoquants manually and plot them:



```python
# We compute the isoquants manually:
plt.plot(k, ((0.5**3)/2-k**3)**(1/3), label = 'Percentil 5')
plt.plot(k, ((1**3)/2-k**3)**(1/3), label = 'Percentil 10')
plt.plot(k, ((2.5**3)/2-k**3)**(1/3), label = 'Percentil 25')
plt.plot(k, ((5**3)/2-k**3)**(1/3), label = 'Percentil 50')
plt.plot(k, ((7.5**3)/2-k**3)**(1/3), label = 'Percentil 75')
plt.plot(k, ((9**3)/2-k**3)**(1/3), label = 'Percentil 90')
plt.plot(k, ((9.5**3)/2-k**3)**(1/3), label = 'Percentil 95')
```

   And we obtain the following graph:



   These isoquants correspond to the different percentiles of output described in the legend. As we can observe, those percentiles with a higher number correspond with those that require a higher amount of input. Then, there is a positive relationship between input and output.