University of Costa Rica
School of Mathematics
MA-501 Numerical Analysis
Homework 1

Juan Ignacio Padilla B.
ID: B55272
Prof. Juan Gabriel Calvo
January 29, 2026

Homework 1

Problem 1.

Given a vector $\boldsymbol{v} \in \mathbb{R}^{n+1}$, the command

```
c=poly(v)
```

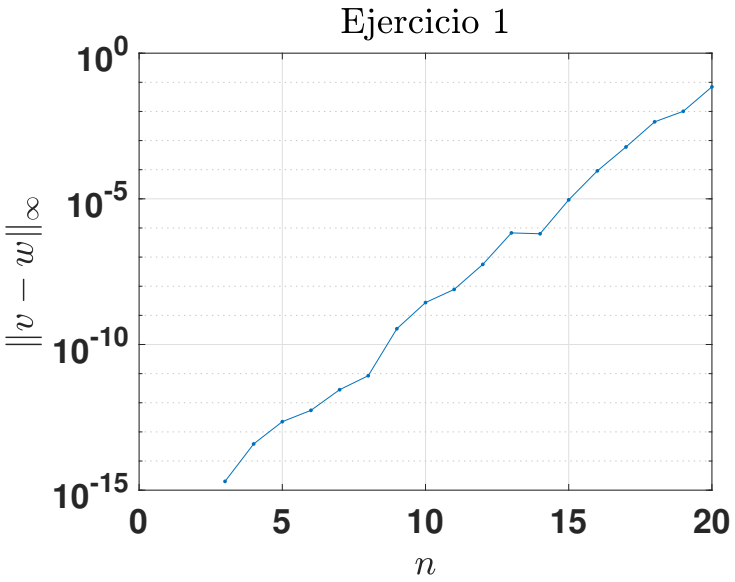
constructs a vector $\boldsymbol{c} \in \mathbb{R}^{n+1}$ whose entries are the coefficients of the monic polynomial

$$p(x) = \prod_{j=1}^{n}(x - v_j),$$

so that its zeros are the entries of the vector $\boldsymbol{v}$. If we solve the equation $p(x) = 0$, its solutions should be precisely the entries of $\boldsymbol{v}$.

a. In `MATLAB`, the vector $\boldsymbol{w}$ corresponding to the roots of p is computed numerically. Then the error $||\boldsymbol{v} - \boldsymbol{w}||_\infty$ is plotted as a function of n, for the vector $\boldsymbol{v} = (1, 2, 3, \ldots, n)^T$. **Explain why the error increases as n grows.**

Solution: For $n \in \{1, 2, \ldots, 20\}$ we have the error plotted by the code provided by the professor



For the first values of n, the error is almost negligible, on the order of 10^{-15}. However, note that as n grows, the error seems to grow linearly, until reaching an error of almost 0.1, for $n = 20$. Since we do not know the nature of the `roots` and `poly` commands, the most logical explanation for this phenomenon is that it involves some rounding defect. We need to analyze our polynomial p more closely.

Observe that for $n = 1$, we have $\boldsymbol{v} = (1)$, and then

$$p(x) = x - 1$$

and so when using the `poly` command we get

```
poly(v) = [1 - 1]
```

However, when $n = 20$, we have

$$p(x) = (x-1)(x-2)\cdots(x-20) = x^{20} - 210x^{19} + \cdots + 20!$$

Where $20! \approx 2.45 \times 10^{18}$. So we have a polynomial whose coefficients vary enormously in order of magnitude. When applying `poly` for $n = 20$, we have

```
poly(v) =
1.0e+19 *
  Columns 1 through 6
    0.0000   -0.0000    0.0000   -0.0000    0.0000   -0.0000
  Columns 7 through 12
    0.0000   -0.0000    0.0000   -0.0000    0.0001   -0.0010
  Columns 13 through 18
    0.0063   -0.0311    0.1207   -0.3600    0.8038   -1.2871
  Columns 19 through 21
    1.3804   -0.8753    0.2433
```

This confirms the fact that the orders of magnitude of the coefficients of $p(x)$ are very disparate. This will lead us to think that any numerical evaluation that the `roots` command performs is subject to rounding errors, since evaluating $p(x)$ at any number will generate a sum of values with different magnitudes, which will inevitably produce rounding errors. This is even more evident when computing $p(1) \approx 1024$, which is completely inaccurate, since it is obvious that $p(1) = 0$.

b. Fix $n = 20$ and compute `w = roots(poly(1:n))`. Use Newton's method to improve the approximation of each entry of $\boldsymbol{w}$, using w_j as the initial value for $j \in \{1, 2, \ldots, n\}$. Is each new approximation more accurate? Justify your answer.
Solution: When computing `w = roots(poly(1:n))`, we get

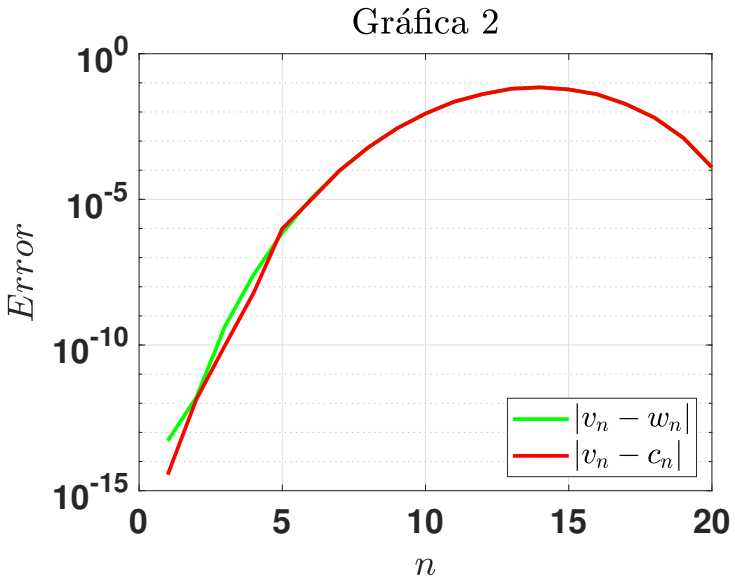
```
w=
0.999999999999949
1.999999999998383
3.000000000444877
3.999999973862455
5.000000705531480
5.999989523351082
7.000096952230211
7.999394310958664
9.002712743189727
```

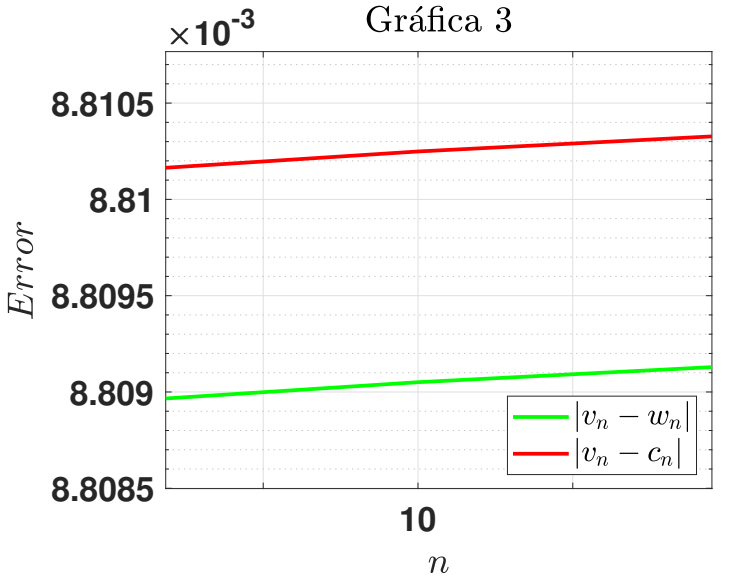
```
9.991190949230132
11.022464271003383
11.958873995343460
13.062663652011070
13.930186454760916
15.059326234074415
15.959717574548915
17.018541647321989
17.993671562737585
19.001295393676987
19.999874055724192
```

And when computing the vector $\boldsymbol{c} = (c_1, c_2, \ldots, c_n)$ where c_j is the approximation of the roots of $p(x)$, using Newton's method, with initial value w_j, we get

```
c=
1.000000000000004
1.999999999998623
2.999999999902548
3.999999993793522
5.000000993489359
5.999990488567676
7.000097488971716
7.999394052477573
9.002712561395008
9.991189751268259
11.022464631556501
11.958873639788711
13.062663808423107
13.930186706646897
15.059321001539574
15.959717676390879
17.018541443827338
17.993671604299941
19.001295472149295
19.999873774976844
```

To determine if our approximation is better or worse than `MATLAB`'s, we create graph 2. The green line corresponds to the approximation `w = roots(poly(1:n))`, while the red one corresponds to the Newton application $\boldsymbol{c}$, taking the w_j as initial values. As can be seen at a glance, between 1 and 5 the quality of the approximation fluctuates slightly, initially our approximation being better, and then the one preset by the program. After $n = 5$, they seem to be similar approximations. However, when zooming in on graph 3, it can be observed that the original approximation is marginally better at $n = 10$ since it is closer to the original vector $\boldsymbol{v}$. This may again be due to rounding errors, since the derivative of $p(x)$ has high-order coefficients, and when evaluated at a small number like 10, it will overlook some values. Furthermore, reasoning from another angle, it is implausible to think that a simple and unoptimized code can produce better approximations than the internal code of `MATLAB`, so it would be expected that the solution `w = roots(poly(1:n))` is closer to $\boldsymbol{v}$.





Problem 2.

Let $c > 0$. Prove that the iteration

$$c_{k+1} = \frac{1}{2}\left(c_k + \frac{c}{c_k}\right)$$

converges quadratically to $\sqrt{c}$ for any $c_0 > 0$.

Solution: First, it is clear that $c_k > 0$ for all k. Note that

$$\begin{aligned} c_{k+1}^2 &= \frac{1}{4}\left(c_k + \frac{c}{c_k}\right)^2 \\ &= \frac{c_k^2}{4} + \frac{c}{2} + \frac{c^2}{4c_k^2} \\ &= \frac{1}{4}\left(c_k - \frac{c}{c_k}\right)^2 + c \geq c \end{aligned}$$

So $c_k \geq \sqrt{c}$ for all $k > 0$. Then we can assume *w.l.o.g.* that $c_0 > \sqrt{c}$ (otherwise we can start the sequence at c_1). Now let us see that

$$\begin{aligned} c_{k+1} - c_k &= \frac{c_k}{2} + \frac{c}{2c_k} - c_k \\ &= \frac{c - c_k^2}{2c_k} \leq 0 \end{aligned}$$

We then have that $\{c_k\}$ is a decreasing sequence, and bounded below. Therefore it converges. Let also $L = \lim_{k\to\infty} c_k$. It must be that

$$\begin{aligned} L = \frac{L}{2} + \frac{c}{2L} &\iff L^2 = \frac{L^2}{2} + \frac{c}{2} \\ &\iff L^2 = c \\ &\iff L = \sqrt{c} \end{aligned}$$

Therefore we have that the sequence converges to $\sqrt{c}$. To see the order of convergence, consider

$$\begin{aligned} \lim_{k\to\infty} \frac{|c_{k+1} - \sqrt{c}|}{(c_k - \sqrt{c})^2} &= \lim_{k\to\infty} \frac{\frac{1}{2}\left(c_k + \frac{c}{c_k}\right) - \sqrt{c}}{(c_k - \sqrt{c})^2} \\ &= \lim_{k\to\infty} \frac{c_k^2 + c - 2c_k\sqrt{c}}{2c_k(c_k^2 - 2\sqrt{c}c_k + c)} \\ &= \lim_{k\to\infty} \frac{1}{2c_k} \\ &= \frac{1}{2\sqrt{c}} \end{aligned}$$

We conclude that the sequence converges with quadratic order.

Problem 3.

(Newton's Method for roots of multiplicity m). Consider $f \in C^{m+2}([a,b])$ [1], with a root $c \in (a,b)$ of multiplicity m. That is

$$f(c) = f'(c) = \cdots = f^{(m-1)}(c) = 0 \ , \ f^{(m)}(c) \neq 0$$

a. Prove that there exists $\delta > 0$ sufficiently small, such that if $c_0 \in (c - \delta, c + \delta)$, then the sequence

$$c_{k+1} = c_k - m\frac{f(c_k)}{f'(c_k)}$$

converges quadratically to c.

Solution: To prove convergence, note that if we define [2]

$$g(x) = x - m\frac{f(x)}{f'(x)}$$

[1] This hypothesis is not from the official statement, but I was allowed to assume one more degree of smoothness.

[2] Note that a priori we do not know if g is undefined when $f'(c) = 0$, since the zeros of the quotient could "cancel"

We see that the fixed points of g coincide with the zeros of f. If c is a root of f, of multiplicity m, we can write

$$f(x) = (x-c)^m h(x)$$

Where $h(c) \neq 0$. Now

$$\begin{aligned} g(x) &= x - \frac{m(x-c)^m h(x)}{m(x-c)^{m-1}h(x) + (x-c)^m h'(x)} \\ \Rightarrow g(x) &= x - \frac{m(x-c)h(x)}{mh(x) + (x-c)h(x)} \end{aligned}$$

Therefore g is well-defined. Now we study the situation more closely. We have

$$\begin{aligned} & c_{k+1} = c_k - m\frac{f(c_k)}{f'(c_k)} \\ \Rightarrow \quad & c - c_{k+1} = c - c_k + m\frac{f(c_k)}{f'(c_k)} \\ \Rightarrow \quad & (c - c_{k+1})f'(c_k) = f'(c_k)(c-c_k) + mf(c_k). \end{aligned}$$

If we call $G(x) = f'(x)(c-x) + mf(x)$, we have

$$(c - c_{k+1})f'(c_k) = G(c_k). \qquad (\star)$$

Now observe that for $i = 1, \ldots, m$,

$$\begin{aligned} & G'(c_k) = mf'(c_k) + f''(c_k)(c-c_k) - f'(c_k) \\ \Rightarrow \quad & G''(c_k) = mf''(c_k) + f'''(c_k)(c-c_k) - 2f''(c_k) \\ & \vdots \\ \Rightarrow \quad & G^{(i)}(c_k) = mf^{(i)}(c_k) + f^{(i+1)}(c_k)(c-c_k) - if^{(i)}(c_k). \end{aligned}$$

Since c is a root of f of multiplicity m, we have

$$G(c) = G'(c) = \cdots = G^{(m-1)}(c) = 0$$

and also $G^{(m)}(c) = mf^{(m)}(c) - mf^{(m)}(c) = 0$. Now, by Taylor we see that

$$G(x) = \frac{G^{(m+1)}(\xi_1)}{(m+1)!}(x-c)^{m+1}$$

Where ξ_1 is between x and c. We also have

$$f'(x) = \frac{f^{(m)}(\xi_2)}{(m-1)!}(x-c)^{m-1}$$

Where ξ_2 is between x and c. Now, by $(\star)$, we obtain

$$\begin{aligned} (c - c_{k+1})\frac{f^{(m)}(\xi_2)}{(m-1)!}(c_k - c)^{m-1} &= \frac{G^{(m+1)}(\xi_1)}{(m+1)!}(c_k - c)^{m+1} \\ \Rightarrow \qquad c - c_{k+1} &= \frac{G^{(m+1)}(\xi_1)}{m(m+1)f^{(m)}(\xi_2)}(c_k - c)^2 \end{aligned}$$

Since $f^{(m)}(c) \neq 0$, it is possible to choose r such that in $I_r = [c-r, c+r]$, we have $f^{(m)}(x) \neq 0$. Take $M = \max_{x,y \in I_r} \left| \frac{G^{(m+1)}(x)}{(m+1)f^{(m)}(y)} \right|$, and we obtain

$$|c - c_{k+1}| \leq \frac{M}{m}(c - c_k)^2$$

Therefore, if we take $\delta = \min\{1/M, r\}$, it will be satisfied that $c_k \in I_\delta$ (by induction), since it would hold that $|c - c_{k+1}| \leq |c - c_k|$. Finally, in I_δ

$$\begin{aligned} |c - c_{k+1}| &\leq \frac{1}{m}|c_k - c| \\ &\leq \frac{1}{m^2}|c_{k-1} - c| \\ &\vdots \\ &\leq \frac{1}{m^{k+1}}|c_0 - c| \xrightarrow[k\to\infty]{} 0. \end{aligned}$$

This guarantees pointwise convergence. Furthermore, we ensure that $M < \infty$, since the denominator $f^{(m)}(y)$ will not vanish in I_δ. Therefore:

$$\lim_{k\to\infty} \frac{|c - c_{k+1}|}{(c - c_k)^2} \leq \frac{M}{m}$$

we have quadratic order convergence.

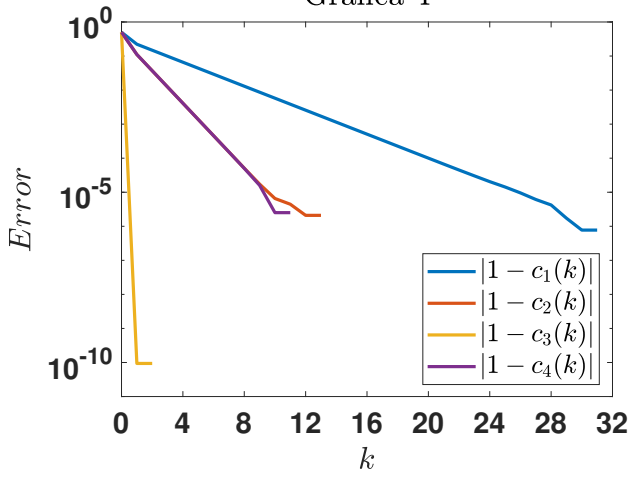
b. Implement a function in `MATLAB` that includes as inputs the function f, the initial value c_0, an integer m representing the order of the zero, and a tolerance ε. Consider $|c_k - c_{k+1}|$ as the condition to stop the iteration.

Solution: The function `NewtonMulti` was implemented, attached in the files.

c. Verify the behavior of the method for the function $g(x) = x^3 - 3x^2 + 3x - 1$, with $x_0 = 1/2$, $\varepsilon = 10^{-8}$ and $m \in \{1, 2, 3, 4\}$. Plot the error as a function of k. Comment on your results.

Solution: The function `NewtonMulti` was applied, with the required parameters, and graph 4 was obtained.

Gráfica 4



Where the vector $c_m(k)$ represents the k-th iteration of the solution to the equation $f(x) = 0$, using the function `NewtonMulti` with multiplicity m. It can be observed that the worst speed was obtained with $m = 1$, since it needed about 30 iterations to adjust to the required tolerance. Meanwhile, the cases $m = 2, 4$ seem to share similar speeds, and an error similar to that of $m = 1$. However, note that the case $m = 3$ presented a high speed, adjusting to the tolerance in only 3 iterations, and also achieving a precision many orders greater than the others. This is clearly because

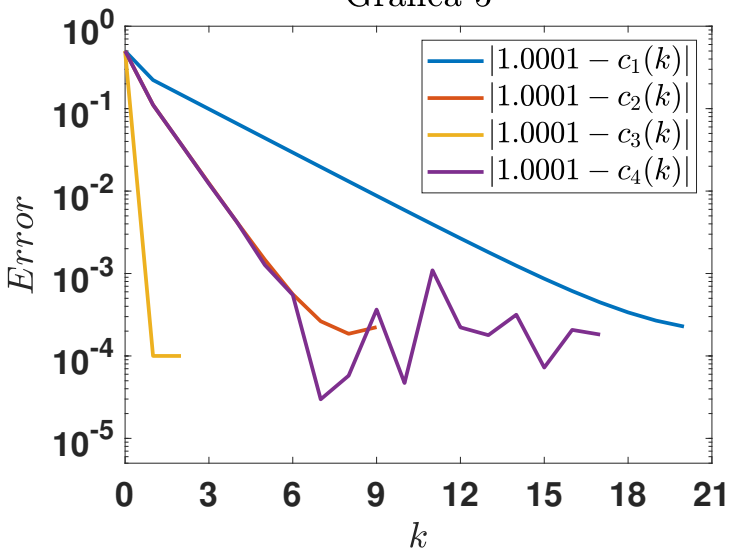
$$f(x) = (x-1)^3$$

has a zero of multiplicity 3, and as shown in part a), it will converge with quadratic speed to the root, while the other methods will do so linearly.

d. Repeat the previous part for $h(x) = (x-1)(x-1.0001)(x-0.9999)$. Numerically, what multiplicity does the largest zero have?

Solution: Once again, the function `NewtonMulti` was applied to the function $h(x)$, with multiplicities $m = 1, 2, 3, 4$, and the error obtained was plotted in graph 5, with respect to the solution $c = 1.0001$.

Gráfica 5



It should be noted that fluctuations occurred in the convergence of the method; this could be due to rounding defects, since the three roots of the polynomial are close, and their basins of attraction could overlap when evaluating numerically. Doing more numerical evaluations, it was estimated that $h'(1) \approx -9.9990 \times 10^{-9} \approx 0$, which is a harmful value for Newton's method, since from a geometric point of view, the tangent lines at a point with derivative close to 0 tend to intersect the x axis very far from where we are working. In fact, for this part, it was necessary to relax the tolerance to a value of 10^{-5} since otherwise, infinite oscillations occurred (this can be verified by running the attached code for this exercise).

On the other hand, once again for the case $m = 3$, there is a much higher convergence speed than the other two, which suggests that numerically 1.0001 **appears to be a triple root**, which we know is incorrect. This fact can again be attributed to rounding conflicts, and to the positioning of the basins of attraction of the roots of $h(x)$.

Problem 4.

(False Position Method) Consider a continuous function f defined on the initial interval $I_0 = [a_0, b_0]$, where $f(a_0)f(b_0) < 0$. At iteration k, given the interval $I_k = [a_k, b_k]$, compute

$$c_k = \frac{a_k f(b_k) - b_k f(a_k)}{f(b_k) - f(a_k)}$$

Then evaluate $f(c_k)$. If $f(a_k)f(c_k) < 0$, take $I_{k+1} = [a_k, c_k]$; otherwise, take $I_{k+1} = [c_k, b_k]$.

a) Explain the geometric meaning of the method, as well as its connection with the Secant and Bisection methods.

Solution: We will work only with the first iteration. We will prove that

$$c = \frac{af(b) - bf(a)}{f(b) - f(a)}$$

Is precisely the x-coordinate of the intersection of the secant line passing through $(a, f(a)), (b, f(b))$, and the x-axis. Consider this secant line; it is easy to see that its equation is given by

$$y = \frac{f(a) - f(b)}{a - b}x + \frac{af(b) - bf(a)}{a - b}$$

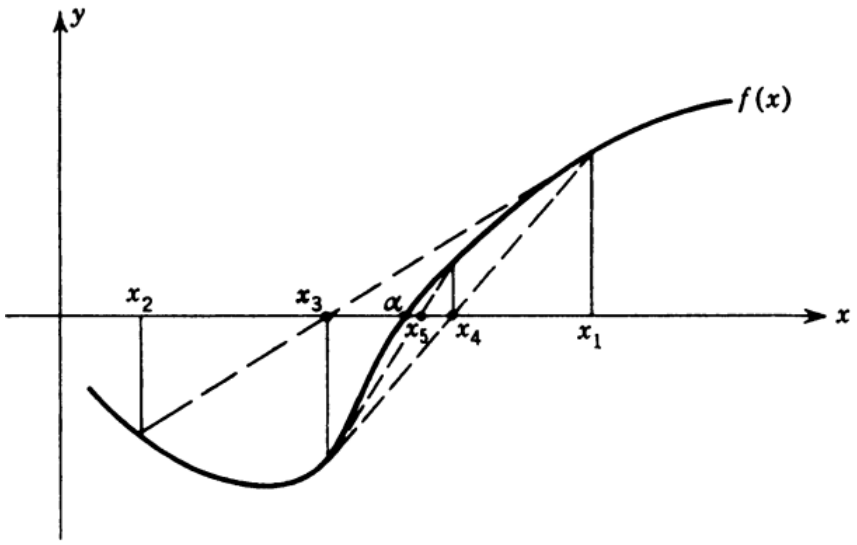
And solving the equation $y = 0$, we get

$$x = \frac{-af(b) + bf(a)}{a - b}\frac{a - b}{f(a) - f(b)} = \frac{af(b) - bf(a)}{f(b) - f(a)} = c$$

Which is the desired value. The second part of the algorithm consists of checking and selecting in which of the two intervals $[a, c]$ or $[c, b]$ our root is located. Then the algorithm is repeated. This ensures that the root we are looking for is always within our working interval, something that does not happen in the Secant method.

The relationship between this method and the secant and bisection methods is quite direct. The method simply consists first of applying exactly the same iteration as the secant method, however, before starting the second iteration, an extra check is performed, which consists of the technique used in the bisection method of $f(a)f(c) < 0$ or $f(c)f(b) < 0$, to narrow our working interval, which will (possibly) improve the stability of the method. The following image taken from [1] represents the first iterations of this method.

Figure 6. False Position Method



We see that precisely, x_3 is the point of intersection of the secant line. Then, instead of drawing the secant line between x_2 and x_3, it is drawn between x_3 and x_1, since it is the interval that satisfies the bisection hypotheses. Thus, x_4 and x_5 are successively calculated, which seem to approach the root α.

b) Write a function in `MATLAB` that implements the false position method. It should have as inputs the function f, the initial interval $I_0 = [a_0, b_0]$, and the tolerance ε. Consider $|c_k - c_{k+1}|$ as the condition to stop the iteration.

Solution: The function `FalsePosition` was implemented, attached in the files.

c) Verify the behavior of your code with the function $f(x) = 1/x - \sin x + 1$, with $a_0 = -1.3, b_0 = -0.5$. Numerically, what is the order of convergence?

Solution: When applying the function `FalsePosition` to the function f, with a tolerance of 10^{-8}, an approximate value of $c = -0.629446484073333$ is obtained. When numerically computing the quotient

$$L = \left| \frac{c - c_k}{c - c_{k-1}} \right|$$

for the number of iterations obtained ($k = 22$), and taking $c = c_{22}$, we have

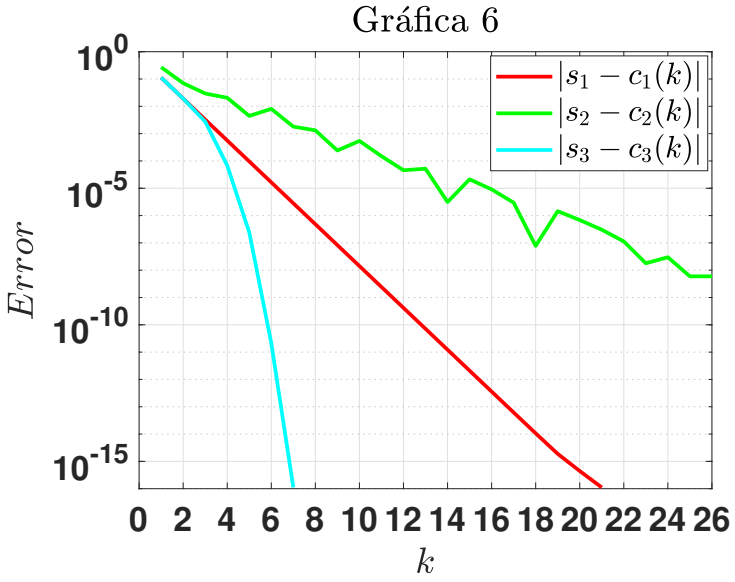
$$L = 0.250000000$$

Which indicates that, at least numerically, we have a linear order of convergence.

d) Compare the rate of convergence for the function from the previous part when using the Bisection and Secant methods. Does the False Position method have any advantage over the others?

Solution: The False Position, Secant, and Bisection methods were applied to the function $f(x) = 1/x - \sin x + 1$, and the results were compared graphically as a function of iterations [3]. The results are summarized in graph 6.

[3]For the Secant method, a slightly lower tolerance was used, to have enough iterations to compare.



The green curve corresponds to the Bisection method, the red one to the false position method, and the cyan one to the secant method. In the legend, the value s is the final value of each method, i.e., the last value it recorded before stopping. We then have that the bisection method slowly approaches its final value, with fluctuations. The other methods approach with relative smoothness; however, it is evident that the secant method is the fastest of the 3 in this case. Recall that numerically, the order of convergence of the false position was 1, while the theoretical order of convergence of the secant is ϕ (the golden number). In this case, the False Position method is not optimal (however it is better than bisection). In general, according to [1], False Position, although usually slower than the Secant, tends to be more stable, as it always converges.

Problem 5.

(Cooling of a Body) Newton's Law of Cooling states that the rate of change of temperature at time t of an object is proportional to the difference between the ambient temperature T_a and the temperature T of the object. That is

$$\frac{dT}{dt} = k(T_a - T)$$

where k is the proportionality constant.

a) Assume that the ambient temperature is constant. Verify that the temperature of the object at time t is given by

$$T(t) = T_a + (T_0 - T_a)e^{-kt}$$

where $T_0 = T(0)$ is the initial temperature of the object.

Solution: We have the ordinary differential equation

$$dT = k(T_a - T)dt$$

Which we solve by separation of variables.

$$\begin{aligned} kt + C &= \int \frac{dT}{(T_a - T)} \\ \Rightarrow \quad kt + C &= -\log(T_a - T) \\ \Rightarrow \quad -kt + C &= \log(T_a - T) \\ \Rightarrow \quad Ce^{-kt} &= T_a - T \\ \Rightarrow \quad T(t) &= T_a + Ce^{-kt} \end{aligned}$$

But we know that $T(0) = T_0$, so

$$\begin{aligned} \Rightarrow \quad T_0 &= T_a + C \\ \Rightarrow \quad C &= T_0 - T_a \\ \therefore \quad T(t) &= T_a + (T_0 - T_a)e^{-kt} \end{aligned}$$

b) A forensic doctor wants to estimate the value of k in order to determine the time of death of corpses. To do this, they take different measurements, obtaining the following data:

t_i (hours)	$T_i(^\circ C)$
0.0	37.00
0.2	36.72
0.4	36.41
0.6	36.12
0.8	35.90

Furthermore, the ambient temperature that day is $21^\circ C$. To estimate k, the doctor considers minimizing the functional

$$f(k) = \sum_i (T(t_i) - T_i)^2.$$

Approximate the critical point of f and verify that f attains a minimum.

Solution: To calculate the critical points of f, we can explicitly compute its derivative,

$$f(k) = \sum_i (T_a + (T_0 - T_a)e^{-kt} - T_i)^2$$

$$\Rightarrow \quad f'(k) = -2(T_0 - T_a) \sum_i t_i e^{-kt_i}(T_a + (T_0 - T_a)e^{-kt_i} - T_i)$$

In `MATLAB` we numerically compute the values $f'(0) \approx -54.4$, and $f'(1) \approx 209.4174$. Then by the intermediate value theorem, we have a critical point in $(0, 1)$. Then the secant method was applied, with a tolerance of 10^{-8}, to calculate the critical point, which was approximated to $k \approx 0.091283148291059$. To verify that it is a minimum, $f''(k) \approx 544$ was numerically computed. Therefore we are in the presence of a minimum, and we can say that the approximation constant approaches our calculated value.

c) On a hot day in San Pedro at $31^\circ C$ a person died. Arriving late at the scene, it is determined that the body temperature is $34^\circ C$. How long ago did the person die? (take $34^\circ C$ as the average temperature of the human body, and k as the estimated value from the previous part).

Solution: We only need to solve the equation

$$T(t) = T_a + (T_0 - T_a)e^{-kt} - 34 = 0$$

Which is done numerically, using once again the secant method. Since all parameters are known, we just need to solve for t numerically, from which we obtain $t \approx 7.593375048260015$. That is, the person died approximately 7 hours and 35 minutes ago.

References

[1] A.Ralston, P. Rabinowitz *A First Course in Numerical Analysis.* Second Edition. Dover Publications. New York. 1978