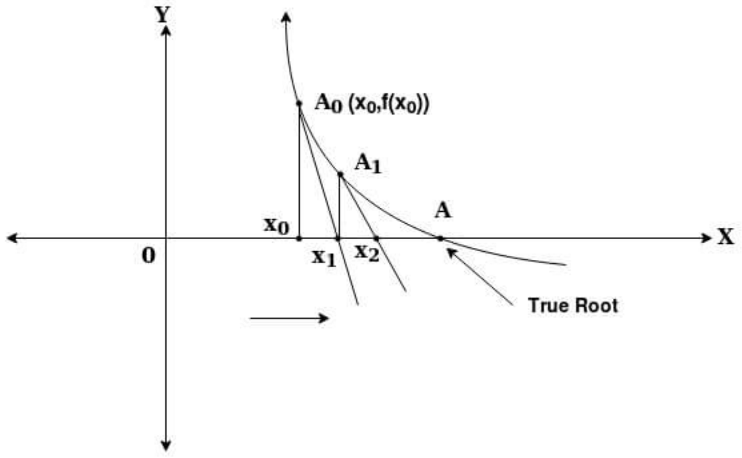
First Obligatory Assignment - Numerical Analysis

Vasileios Papastergios (ID: 3651)

December 22, 2020



Aristotle University of Thessaloniki
School of Informatics

SCHOOL OF INFORMATICS

Course:	Numerical Analysis
Semester:	3rd
Instructor:	Anastasios Tefas

Contents

1 Introduction and Overview

The present document serves as report and code documentation for the first assignment in the course of Numerical Analysis. The author attended the course during their 3$^{\text{rd}}$ semester of studies at the School of Informatics AUTh. The document is handed complementary with the script files developed in GNU Octave as code solutions for the assignment and contains justification as well as further analysis on them.

To start with, the assignment consists of four (4) exercises. The **first** and the **second** exercise deal with the fundamental root-finding algorithms with iterative logic. The bisection method, as well as the Newton-Raphson and the Secant ones are presented twice: once in their conventional, classic form (Exercise 1) and once in a slightly modified and alternative proposal (Exercise 2).

The **third** exercise focuses on linear systems of equations. The tasks related to it are code-oriented and feature some of the basic algorithms surrounding square matrices.

The **fourth** exercise dives into the interesting field of eigenanalysis of matrices. In fact, the exercise focuses on a well-known application of it in computer science. In particular, a rough explanation of the page rank model is given. The tasks aim in getting a deeper insight into this brilliant way of solving a worldwide, everyday-life problem; searching and sorting the web pages in terms of significance throughout the World Wide Web.

This report is partitioned in sections, one for every exercise in this assignment. Each section contains smaller logical units that present the individual tasks of the exercise. At the start of every section, the relative code files are mentioned. The author's recommendation is to read this report in parallel with the respective code, for enhanced comprehension and justification.

Before stepping into the first exercise, it is deemed important to state that throughout this procedure, special emphasis is given on the explanation of reasoning, as well the code documentation. Mathematical relations are introduced whenever necessary, as well as programming reasoning, in a try to explain in a more clear way the author's point of view.

At this point, the introductory information has come to its end. The first exercise analysis comes right on the next page.

2 Exercise One

The first exercise focuses on root-finding algorithms, and especially on the classic iterative methods of finding roots of non-linear equations. Handed code files that are related to the exercise solution are the following: `bisectionMethod.m`, `newtonRaphsonMethod.m`, `secantMethod.m` and `mainExercise1.m`.

2.1 Wording

Let $f(x) = e^{sin^3(x)} + x^6 - 2x^4 - x^3 - 1$ be an one-variable function in the interval [-2, 2]. Make the graph of f in this interval. Afterwards, write appropriate scripts in any programming language, in order to calculate the roots of f in the given interval, using the following methods: (a) the bisection method, (b) the Newton-Raphson method and (c) the secant method. For each one of the roots, compare the number of iterations executed by each method. For the Newton-Raphson method, in particular, examine whether the algorithm converges to the roots quadratically or not. What specialty do specific roots feature that can justify for the non-quadratic convergence of the algorithm? Explain your answer.

2.2 Basic Analysis

According to the exercise requirements, we are going to adopt a graph-oriented approach, in order to gradually accomplish the total task. In other words, visual observations on the figure of the function f(x) in the given interval will lead our reasoning, towards the root-finding process. Figure 1 shows the plot of the function produced by GNU Octave programming language and, in particular, its `fplot` command.

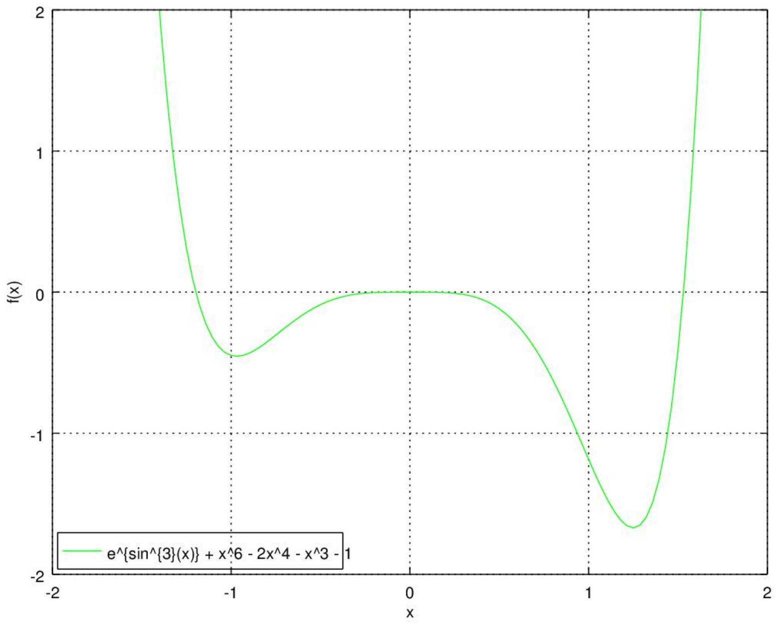


Figure 1: The plot of f(x) in [-2, 2] x [-2, 2]

Only a few glances on the figure make it obvious enough to the naked eye to spot (for sure and at least) two approximate points, where the graph intersects the x-axis. Some floating point number near -1.1 and another one near 1.5 are quite clear roots of our function. Keeping in the back of our mind these two (easily-spotted) roots, our interest now focuses on the behavior of the graph near the axes' origin.

Intersection between the graph and the x-axis is not so clear near the origin. As a matter of fact, the function seems to take values really close to 0 in a wide range of points lying on the x-axis. More accurately, there is a range of points in the form $(x, 0)$, with $x \to 0^+, 0^-$, that are candidate roots of our function. This ascertainment leads us to resort to a more algebraic way of spotting the actual root near the origin. The observation that the described range seems to be symmetrical with respect to the y-axis indicates $x = 0$ as a reasonable guess for our algebraic trial. Indeed, $f(0) = e^0 - 1 = 0$, thus $x = 0$ is also a root of the function within the given interval. Completely respectively, we can algebraically check by hand that there is no other x within the described range of candidate roots that satisfies $f(x) = 0$, since the value of the function decreases, as x fends off the origin in both directions.

Diving a little bit more into calculus, we can compute the first derivative of the function and evaluate its value for $x = 0$. We find out that 0 is, as well, a root of the first derivative of our function ($f'(0) = 0$), which means $O(0, 0)$ is a local extremum point. It is obvious from the graph that $O(0, 0)$ is a local maximum point, thus $f(0) \geq f(x)$ $\forall$ x in an area near the origin and the equality is valid only for $x = 0$. This statement is just a rough verification that the root $x = 0$ found algebraically by hand is correct. More details about the derivative of the function and the possible obstacles imposed to algorithmic methods by the non-singular multiplicity of root $x = 0$ are to be thoroughly analyzed later on, when presenting the Newton-Raphson Method.

For now, the important news is that we have managed to define the number of roots for our function in the given interval, as well as to spot them approximately. Consequently, we are theoretically and practically ready to introduce the root-finding algorithms and feed them with the appropriate initial data, in order to accurately determine the roots within accuracy of five decimal places. The following sections serve this purpose.

2.3 The bisection method

In this subsection we are going to discuss about the root-finding process for our function f(x) within the interval of interest ([-2, 2]), utilizing the classic version of the bisection method. Code files involved: `bisectionMethod.m` and `mainExercise1.m`.

2.3.1 Theoretical Background

According to the method, the root quest starts from an initial interval (a, b) that meets the requirements of Bolzano's Theorem. The Theorem requires that $f(a) * f(b) \leq 0$. This condition, in combination with the fact that our function is continuous in $[-2, 2]$, guarantees the existence of at least one root contained in the initial interval.

In a deeper analysis, let m be the middle of the (a, b) interval. The algorithm indicates m as a potential root. Hence, the value of $f(m)$ is evaluated. In case m is indeed a root, then the algorithm stops its execution, having found the desired root.

Otherwise, the algorithm re-arranges the interval (a, b), thus its guess for the root, as well. One of the limit points of the interval is replaced by m, always taking care of not violating the Bolzano's Theorem condition. That means that if $f(a) * f(m) \leq 0$, then the initial interval is re-arranged to (a, m). Otherwise (if $f(m) * f(b) \leq 0$), the new interval gets the form (m, b). Having re-estimated the interval, the algorithm repeats its execution from the beginning by refining the root guess to the new middle $m^{(1)}$ and the process goes on. The algorithm ceases when either the root is found or the desired accuracy has been accomplished.

Based on the wording of the exercise, calculating the root has to be governed by (at least) five decimal places of accuracy. Translating this requirement into an upper-limit tolerance, we can safely make the algorithm cease, when the error in root calculation is less than:

$$tolerance = \frac{1}{2} * 10^{-d} = \frac{1}{2} * 10^{-5} = 0.5e - 5, \tag{1}$$

where d represents the number of desired decimal places of accuracy.

Another important knowledge we are going to need in defining the stopping criterion for the algorithm, is the bisection method worst-case error formula. According to it:

$$\left| error^{(k)} \right| \leq \frac{b^{(k)} - a^{(k)}}{2}, \tag{2}$$

which in natural language can be explained as following: The upper limit for the bisection method absolute error after k steps of execution, is the half of the re-arranged (in k^{th} step) interval's absolute length.

Combining equations (1) and (2), we can reach a really useful formula, indicating that we can make the algorithm cease looping for the root when:

$$\left|error^{(k)}\right| < tolerance \Leftrightarrow \frac{b^{(k)} - a^{(k)}}{2} < 0.5e-5. \tag{3}$$

In case the criterion is satisfied, the calculated root is the middle point of the interval, as re-arranged after k steps, thus $root = \frac{b^{(k)} + a^{(k)}}{2}$. The root calculation in that case has the desired accuracy, as we have proven.

On that point we have covered the theoretical background for the root finding process, using the bisection method. In the subsection that follows, the theoretical claims are implemented into GNU Octave code.

2.3.2 GNU Octave Code

```
function [root, iterationsExecuted] = bisectionMethod(aFunction,
                                                      leftLimit,
                                                      rightLimit,
                                                      decimalPlacesAccuracy)

    if (leftLimit > rightLimit)
        [leftLimit, rightLimit] = swapNumbers(leftLimit, rightLimit);
    endif

    iterationsExecuted = 0;
    desiredAccuracy = 0.5 * 10^(-decimalPlacesAccuracy);

    if (aFunction(leftLimit)*aFunction(rightLimit) > 0)
        error("bisectionMethod: The given interval does not meet the Bolzano's Theorem
    requirements. The product f(a)*f(b) must be non-positive!");
    elseif (aFunction(leftLimit) == 0)
        root = leftLimit;
        return;
    elseif (aFunction(rightLimit) == 0)
        root = rightLimit;
        return;
    endif

    while ((rightLimit - leftLimit)/2 >= desiredAccuracy)
        iterationsExecuted++;
        middle = ((leftLimit + rightLimit)/2);
        if (aFunction(middle) == 0)
            root = middle;
            return;
        elseif (aFunction(leftLimit)*aFunction(middle) < 0)
            rightLimit = middle;
        else
            leftLimit = middle;
        endif
    endwhile
    root = double((rightLimit + leftLimit)/2);
endfunction
```

Listing 1: GNU Octave code for classic bisection method

2.3.3 Intervals selection

Up to that point, we have browsed through the theoretical background of the bisection method, as well as its code implementation. Utilizing the method, in order to find the actual roots of the given function, comes naturally as the next step of our reasoning. As a matter of fact, the actual task of this last step lies in picking appropriate intervals (a, b) each time when calling the method. The ultimate goal is to feed the function with appropriate initial data, in order for the algorithm to converge to all the 3 of the function roots (converge to one root at a call) in the interval of interest (as shown in 2.2). Throughout the process, obeying the Bolzano's Theorem remains always the priority.

As a more observant reader could have noticed, the bisection method algorithm returns -apart from the calculated root- the number of iterations executed. This number shall not get more spotlight on for now. We are going to discuss in detail about it in the next subsection. Our interest for now is devoted just in finding appropriate a and b numbers in order to make the algorithm converge to the roots, ignoring temporarily the number of iterations.

Back to the task, we have to admit that for the two single-multiplicity roots it is quite easy to spot appropriate intervals, that meet the theorem's requirement, just by observing the graph of the function. Actually, there are plenty of (infinite) intervals that make the algorithm converge to these two roots, when passed as arguments to the function. Indicatively, we can choose the following intervals:

```
[root, iterations] = bisectionMethod(f, -2, -1, 5);
```

Listing 2: Starting from (-2, -1) the bisection method converges to the root r_1=-1.19762

```
[root, iterations] = bisectionMethod(f, 1, 2, 5);
```

Listing 3: Starting from (1, 2) the bisection method converges to the root r_2=1.53013

As we had approximately predicted back in 2.2 section, the two single-multiplicity roots of the function are $r_1 = -1.19762$ and $r_2 = 1.53013$. Things, however, are a little bit more complicated for the third known root, $r_3 = 0$. An important specialty is the fact that r_3 lies between the two already found roots. Nevertheless, the function keeps lying underneath the x-axis in the whole range between r_1 and r_2. In math words, $f(x) \leq 0 \ \forall \ x \in (r_1, r_2)$ and the equality is only valid for $x = r_3 = 0$.

To make matters stiffer, outside the range delimited by the two roots, the function flies off away from the x-axis. That means $f(x) \gg 0 \ \forall \ x \in \{[-2, r_1) \cup (r_2, 2]\}$; thus the Bolzano's Theorem is neither applicable there.

Having claimed all the above, we now have to combine them, in order to conclude to an interval (a, b) that satisfies the Bolzano's Theorem and makes the bisection method algorithm converge to 0, when the interval is fed to it.

Concerning the Bolzano's Theorem alone and utilizing simple reasoning and graph observation, we can come down to the conclusion that the interval has to be of the form (a, b), where $a \in (-|r_2|, \ r_1)$ and $b \in (|r_1|, \ r_2)$. Only intervals of that form do verify $f(a) * f(b) \leq 0$ and $0 \in (a, b)$.

Keeping that in mind, we should now check whether (a, b) makes the algorithm always converge to 0, or whether there is more conditioning. Decisive role in this plays the fact that our function takes negative values in-between the two already found roots. As a result, the selected initial interval for r_3 has to be also symmetrical in relation with the origin. In math words, the interval has to be of the final form (-c, c), where $c \in (|r_1|, \ |r_2|)$. In any other case, the algorithm is going to converge to the root $r_1 \approx -1.19$, rather than to the desired $r_3 = 0$. Finally, we indicatively choose $c = 1.3$ in order for the algorithm to converge to 0:

```
[root, iterations] = bisectionMethod(f, -1.3, 1.3, 5);
```

Listing 4: Starting from (-1.3, 1.3) the bisection method converges to the root r_3=0

2.3.4 Number of iterations executed

The fact that the bisection method successively divides the initial interval in half, let us know from beforehand the number of iterations needed, in order for the algorithm to calculate a root within a given accuracy. Utilizing previous knowledge, we have the following inequality for the number n of iterations needed, in relation with the number of decimal places d of accuracy:

$$\frac{b-a}{2^{n+1}} < \frac{1}{2} * 10^{-d} \quad \Leftrightarrow \quad \frac{b-a}{2^n} < 10^{-d} \quad \Leftrightarrow \quad \log_{10}(b-a) - \log_{10} 2^n < -d \quad \Leftrightarrow \quad n > \frac{\log_{10}(b-a) + d}{\log_{10} 2} \tag{4}$$

Inequality (4) can be very useful in calculating in advance the maximum number of iterations the algorithm is going to execute. The word ”maximum” is a key-word in the previous sentence, since the algorithm can cease earlier, in case the actual root is found. Making simple value replacement for a, b and d, we can evaluate the iterations for the specific intervals we have chosen in 2.3.3. The results are presented in Table 1:

a	b	d	iterations (calculated before)
-2	-1	5	$\approx 16.61 \Rightarrow n_{max} = 17$
1	2	5	$\approx 16.61 \Rightarrow n_{max} = 17$
-1.3	1.3	5	$\approx 17.988 \Rightarrow n_{max} = 18$

Table 1: Upper limit for iterations, calculated algebraically in advance

Bearing in mind the above upper limits for the number of iterations n, we can present the more accurate (yet more naive) approach adopted in the script of subsection 2.3.2. As a matter of fact, an on-the-fly approach is adopted, rather than the in-advance alternative. The first two rows of the table remain identical, since the algorithm exhausts the maximum multitude of iterations. In the third row, however, a big improvement shows up, since the algorithm makes a perfectly targetted first guess. The actual number of iterations for the selected intervals are displayed in Table 2:

a	b	d	iterations (calculated on the fly)
-2	-1	5	$n_{actual} = 17$
1	2	5	$n_{actual} = 17$
-1.3	1.3	5	$n_{actual} = 1$

Table 2: Number of actual iterations calculated on the fly

2.4 The Newton-Raphson method

In this subsection we are going to discuss about the root-finding process for the function $f(x)$ within the interval of interest $[-2, 2]$, utilizing the classic version of the Newton-Raphson method. Code files involved: `newtonRaphsonMethod.m` and `mainExercise1.m`.

2.4.1 Theoretical background

The Newton-Raphson method belongs to the general category of fixed-point iteration algorithms. Starting from an initial guess x_0, the algorithm repeatedly calculates the next x value with the aid of the following recursive formula:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}, \qquad \textbf{for i = 1, 2, ...}$$

Concerning the function given in the current exercise, we can declare the first derivative of the function $f(x)$ as:

$$f'(x) = 3\mathrm{e}^{\sin^3(x)} \cos(x) \sin^2(x) + 6x^5 - 8x^3 - 3x^2$$

Replacing the expressions for $f(x)$ and $f'(x)$ we can form the Newton-Raphson recursive formula for evaluating the root guess in step $i+1$:

$$x_{i+1} = x_i - \frac{\mathrm{e}^{\sin^3(x_i)} + x_i^6 - 2x_i^4 - x_i^3 - 1}{3\mathrm{e}^{\sin^3(x_i)} \cos(x_i) \sin^2(x_i) + 6x_i^5 - 8x_i^3 - 3x_i^2} \tag{1}$$

Concerning the stopping criterion of the algorithm, there is one part in common and one in difference, in relation with the bisection method. The similarity lies in the tolerance involved in the stopping criterion condition. That means that, for the current case, the tolerance is $\frac{1}{2} * 10^{-5} = 0.5e - 5$. Where the Newton-Raphson differs from the bisection method is the error between the calculated and the actual root. In case of the Newton-Raphson method, it is previous knowledge that the desired accuracy has been accomplished when:

$$|x_i - x_{i-1}| < tolerance \quad \Leftrightarrow \quad |x_i - x_{i-1}| < 0.5e - 5 \tag{2}$$

Equations (1) and (2) are the gist of the classic Newton-Raphson method, and we are going to utilize them when implementing the algorithm in GNU Octave programming language. Before stepping into the code, however, it is deemed useful to make a few notes on issues that will be encountered later on in the root-finding process.

Firstly, having calculated the first derivative of the function we can investigate a little bit more about the root $r_3 = 0$ of the given function. Back in the basic analysis of the Exercise (2.2) we had assumed (based on the graph) that r_3 has multiplicity greater than one. Indeed, we can now algebraically verify such an assumption, since $f'(0) = 0$.

As a matter of fact, $r_3 = 0$ has even greater (than double) multiplicity. In order to avoid making the paper "heavy" and calculations-packed, we are not going to present in detail the derivatives of higher order of f. However, we will do present the really interesting fact that $f(r_3) = f'(r_3) = f''(r_3) = f'''(r_3) = 0$. In simple words, not only $r_3 = 0$ is a root of the given function, but also has multiplicity $m = 4$.

Taking this in mind, we are going to make a slight alteration to the Newton-Raphson recursive formula, in order for it to be able to handle roots with higher-than-single multiplicity. The (slightly) changed algorithm is presented by Timothy Sauer in his book "Numerical Analysis - Second Edition" and involves the root multiplicity m:

$$x_{i+1} = x_i - \frac{mf(x_i)}{f'(x_i)}, \qquad for \;\; i = 1, 2, \ldots \;\; and \;\; m \geq 1 \tag{3}$$

Implementing the above theoretical claims in code, we get the following

2.4.2 GNU Octave Code

```
function [root, iterationsExecuted] = newtonRaphsonMethod(aFunction,
                                                          itsDerivative,
                                                          startingPoint,
                                                          decimalPlacesAccuracy,
                                                          rootMultiplicity = 1)

    iterationsExecuted = 0;
    if (aFunction(startingPoint) == 0)
        root = startingPoint;
        return;
    endif

    desiredAccuracy =0.5*10^(-decimalPlacesAccuracy);
    x = startingPoint;

    do
        iterationsExecuted++;
        previous = x;
        x = x - (rootMultiplicity*aFunction(x)/itsDerivative(x));
    until (abs(previous - x) < desiredAccuracy || aFunction(x) == 0)

    root = x;
endfunction
```

Listing 5: GNU Octave code for classic Newton-Raphson method (involving root multiplicity)

On that point, a small notice, concerning the root multiplicity parameter of the script-function, should be made. Stepping out from the current exercise and algebraic function f for a while, we have to admit that it is not the common case to know the multiplicity of a root in advance. As a matter of fact, it is usual that we do not even know the root itself, let alone its multiplicity.

That is the reason why the root multiplicity parameter of the script-function is declared as a default-value parameter, set to $m = 1$ by default. This last feature attaches to the function the capability to be called in a polymorphic variety of ways. That means:

- the script function can be called without the multiplicity parameter (set by default to 1), in case the root multiplicity is not known in advance or is known to be 1

- the script function can be called with the multiplicity parameter stated, in case the root multiplicity is known in advance to be greater than single.

The above note is going to be utilized later on, when discussing about the method's convergence to specific roots. In general, however, and except for the root multiplicity parameter, the algorithm does not feature any bizzare or unexpected (based on the theoretical analysis) pieces of code.

2.4.3 Initial points selection

As it is known from the basic theory of the Newton-Raphson method, the initial point to start iterations from, should be carefully selected. More specifically, the initial point x_0 has to satisfy the inequality $f(x_0) * f''(x_0) > 0$.

Bearing this in mind, we can indicatively choose the following starting points, so as to make the algorithm converge to each one of the two single-multiplicity known roots ($r_1 = -1.19762$ and $r_2 = 1.53013$):

```
[root, iterations] = newtonRaphsonMethod(f, f_derivative, -1.4, 5);
```

Listing 6: Starting from x_0=-1.5 the Newton-Raphson method converges to the root r_1=-1.19762

```
[root, iterations] = newtonRaphsonMethod(f, f_derivative, 1.8, 5);
```

Listing 7: Starting from x_0=2.1 the Newton-Raphson method converges to the root r_2=1.53013

A more thoroughly made investigation is needed, however, when trying to make the algorithm converge to the other root $r_3 = 0$. The specialty lies in its multiplicity, $m = 4$.

More accurately, we are going to call the function twice. The first time we are going to pretend the root multiplicity is not known (ignore the multiplicity parameter). The second time, we are going to let the algorithm know that the wanted root has multiplicity $m = 0$. Choosing $x_0 = -0.7$ as a starting point, the algorithm seems to converge to 0 both times.

```
[root, iterations] = newtonRaphsonMethod(f, f_derivative, -0.7, 5);
```

Listing 8: Starting from x_0=-1.5 the Newton-Raphson method converges to the root r_3=0. Root multiplicity is conceived as unknown in advanced

```
[root, iterations] = newtonRaphsonMethod(f, f_derivative, -0.7, 5, 4);
```

Listing 9: Starting from x_0=-1.5 the Newton-Raphson method converges to the root r_3=0. Root multiplicity is conceived as known in advanced

By first sight, no difference is obvious to the naked eye, since the algorithm converges to 0 in both calls. A more detailed analysis, however, can disclose the rate of convergence of the two calls, revealing an important difference between them. This analysis is presented in the subsection that follows.

2.4.4 Convergence rate

In order to calculate the convergence rate of the algorithm, concerning each one of the three roots, we are going to use the following two basic Theorems.

Theorem1: Let e_i denote the error after step i of an iterative method. The iteration is quadratically convergent if

$$M = \lim_{i \to \infty} \frac{e_{i+1}}{e_1^2} < \infty.$$

Theorem2: Let f be twice continuously differentiable and $f(r) = 0$. If $f'(r) \neq 0$, then Newton Raphson's method is locally and quadratically convergent to r. The error e_i at step i satisfies

$$\lim_{i\to\infty} \frac{e_{i+1}}{e_1^2} = M,$$

where

$$M = \frac{f''(x)}{2 * f'(x)}.$$

Combining the two theorems and adjusting them to the function of the current exercise, we can conclude to the following statement:

In order to prove that the Newton-Raphson algorithm converges ***quadratically****, we just need to show that:*

$$\frac{f''(x)}{2 * f'(x)} < \infty \quad \Leftrightarrow$$

$$\Leftrightarrow \quad \frac{9e^{\sin^3(x)} \cos^2(x) \sin^4(x) - 3e^{\sin^3(x)} \sin^3(x) + 6e^{\sin^3(x)} \cos^2(x) \sin(x) + 30x^4 - 24x^2 - 6x}{2(3e^{\sin^3(x)} \cos(x) \sin^2(x) + 6x^5 - 8x^3 - 3x^2)} < \infty \tag{4}$$

Implementing relation (4) for each one of the single-multiplicity roots we can easily prove that the Newton-Raphson algorithm converges quadratically.

In particular, for $r_1 = -1.19762$ we have:

$$\lim_{i\to\infty} \frac{e_{i+1}}{e_1^2} = \frac{f''(r_1)}{2f'(r_1)} \approx \frac{34.636}{-9.8409} = -3.5196 < \infty \Rightarrow \text{quadratic convergence}$$

Respectively, for $r_2 = 1.53013$ we have:

$$\lim_{i\to\infty} \frac{e_{i+1}}{e_1^2} = \frac{f''(r_2)}{2f'(r_2)} \approx \frac{39.616}{29.945} = 1.3230 < \infty \Rightarrow \text{quadratic convergence}$$

Thus, the algorithm features quadratic convergence, concerning the single-multiplicity roots r_1 and r_2. Nevertheless, the algorithm cannot boast of such a quick performance in case of the third root, $r_3 = 0$. To be completely accurate, the classic version of the method, the one that does not involve the root multiplicity, fails to achieve quadratic convergence. As a matter of fact, this classic version converges linearly for roots with multiplicity greater than singular.

In order to prove that, we need to introduce one more

Theorem: Assume that the $(m+1)$-times continuously differentiable function f on $[a, b]$ has a multiplicity m root at r. Then Newton's Method is locally convergent to r, and the error e_i at step i satisfies

$$\lim_{i\to\infty} \frac{e_{i+1}}{e_1} = S, \tag{5}$$

where $S = \frac{m-1}{m}$.

In our case, we have that $m = 4$ for the root $r_3 = 0$. Consequently, we can easily find out that the classic version of the algorithm converges linearly to the root, since

$$\lim_{i\to\infty} \frac{e_{i+1}}{e_1^2} = \frac{m-1}{m} = \frac{3}{4} < \infty. \tag{6}$$

In contrast with the classic method, the slightly changed (to involve root multiplicity m) method has by far a better performance. According to T. Sauer, the alternated method converges locally and quadratically to the root. T. Sauer skips the proof of such a claim. However, it would be completely unfounded to let this claim pass without any kind of verification, even partial. For that reason, we are going to calculate the fracture $\frac{e_i}{e_{i-1}^2}$ at every step of the algorithm starting from $x_0 = -0.7$. The outcome is presented in Table 3:

i	x_i	$e_i = \|x_i - r\|$	e_1/e_{i-1}^2
0	-0.7	0.7	
1	-0.11377	0.11377	0.23218
2	0.0014203	0.0014203	0.10973
3	0.000001	1.3272e-007	0.065787

Table 3: : Actual numbers of error throughout the slightly-modified method, starting from $x_0 = 0$.

Indeed, the fracture $\frac{e_i}{e_{i-1}^2}$ is a real number in each one of the iterations executed by the algorithm. Consequently, it is verified, even in an experimental manner, that the modified method (involving the root multiplicity) does converge quadratically to the root $r_3 = 0$.

2.4.5 Number of iterations

A rather imminent and more obvious to the naked eye impact of the convergence rate of the algorithm is the number of iterations executed. The number of iterations executed, each time the function is called, depicts the convergence rate that rules the call. In other words, the difference between the quadratically convergent calls and the linear convergent one is quite impressive. The following table presents the plain truth the number of iterations can reveal:

wanted root	x_0	root multiplicity (argument)	iterations
-1.19762	-1.4	not passed (1 by default)	5
1.53013	1.8	not passed (1 by default)	6
0	-0.7	not passed (1 by default)	32
0	-0.7	4	3

Table 4: : Number of actual iterations calculated on the fly

2.5 The Secant Method

In this subsection we are going to discuss about the root-finding process for the function $f(x)$ within the interval of interest $[-2, 2]$, utilizing the classic version of the secant method. Code files involved: `secantMethod.m` and `mainExercise1.m`.

2.5.1 Theoretical Background

The secant method is, actually, an alternative version of the Newton-Raphson method, in which the derivative of the function is replaced by an approximation of it. The method is really useful in case the derivative of the function is either unknown or expensive (or even impossible) to claculate. The method needs to be fed with two initial guesses x_0, x_1. The recursive formula for the next x value is:

$$x_{i+1} = x_i - \frac{f(x_i)(x_i - x_{i-1})}{f(x_i) - f(x_{i-1})}, \quad \text{for i=1, 2, 3, ...} \tag{1}$$

Except for this formula-related change, the rest of the gist remains intact, relatively to the Newton-Raphson method. As a result, it is rather easy to transfer the secant method into code.

2.5.2 GNU Octave Code

```
function [root, iterationsExecuted] = secantMethod(aFunction,
                                                   startingPoint1,
                                                   startingPoint2,
                                                   decimalPlacesAccuracy)

    iterationsExecuted = 0;
    if (aFunction(startingPoint1) == 0)
        root = startingPoint1;
        return;

    elseif (aFunction(startingPoint2) == 0)
        root = startingPoint2;
        return;
    endif

    desiredAccuracy =10^(-decimalPlacesAccuracy);

    x = startingPoint1;
    previous1 = startingPoint2;

    do
        iterationsExecuted++;
        previous2 = previous1;
        previous1 = x;
        x = previous1 - (aFunction(previous1)*(previous1 - previous2))/(aFunction(previous1)
    -aFunction(previous2));
    until (abs(previous1 - x) < desiredAccuracy || aFunction(x) == 0)

    root = x;
endfunction
```

Listing 10: GNU Octave code for classic secant method

2.5.3 Initial guesses' selection

As we have already demonstrated, the secant method needs two initial guesses, in order to calculate an approximation of the derivative and then start the iterations. The following listings present the initial points selected indicatively by the author, in order to make the algorithm converge to each one of the wanted roots.

```
[root, iterations] = secantMethod(f, -2, -1, 5);
```

Listing 11: Starting from x_0=-2 and x_1=-1 the secant method converges to the root r_1=-1.19762

```
[root, iterations] = secantMethod(f, 1.3, 1.6, 5);
```

Listing 12: Starting from x_0=1.3 and x_1=1.6 the secant method converges to the root r_2=1.53013

```
[root, iterations] = secantMethod(f, -0.8, -0.7, 5);
```

Listing 13: Starting from x_0=-0.8 and x_1=-0.7 the secant method converges to the root r_3=0

2.5.4 Number of iterations

As an alternated version of the Newton-Raphson algorithm, the secant method attempts to achieve the convergence rate its super-category features. Nevertheless, the fact that the derivative is not calculated directly, but replaced by an approximation of it, leads to a loss of performance, in terms of convergence rate. Table 5 presents the number of iterations executed by the script-function, until each one of the wanted roots is found:

wanted root	x_0	x_1	iterations
-1.19762	-2	-1	13
1.53013	1.3	1.6	7
0	-0.8	-0.7	44

Table 5: Number of iterations calculated on the fly

2.6 Comparison between number of iterations

Up to this point, we have covered in detail all three root-finding, iterative methods: bisection, Newton-Raphson and secant. The logic trail we followed was based on an experimental approach of the root finding process, utilizing one of the above methods each time. Throughout that process, we placed a little bit more emphasis on the number of iterations executed by each algorithm, measuring it on an actual function f, given by the exercise.

As a matter of fact, via this experimental and code-oriented trail of reasoning, we managed to come down to conclusions that verify (in an empirical manner) plenty of the principles that are theoretically known for the three methods. In specific, the fact that the function passed to the methods was the same, as well as the wanted roots were identical among the three methods, guaranteed the formation of an ideal environment for us to investigate practically the convergence rate of each one of them.

As a matter of fact, it is more than obvious that the bisection method converges with the slowest rate to all the three roots. Judging after the fact, this is rather expected, since the bisection method is known for its linear convergence rate. This linearity is depicted in the number of iterations.

One step further, the secant method seems to have, in general, a better performance, compared to the bisection method. The algorithm converges to the two single-multiplicity roots quicker than its previous rival. However, the performance degrades a lot in case of the third root, because of its greater-than-singular multiplicity. As a result, the experimental approach totally verifies the already known fact that the secant method boasts of superlinear convergence.

Stepping up one more level in performance, the Newton-Raphson method presents the minimum number of iterations among the algorithms, proving in action its quadratic convergence. To be absolutely precise, the classic version of the algorithm (the one that does not involve the root multiplicity) finds it hard with multiple roots. The slightly modified version that includes multiplicity is really helpful when the multiplicity of the wanted root is known in advance, letting Newton-Raphson algorithm converge quadratically in all cases of root multiplicity.

3 Exercise Two

The second exercise focuses on root-finding algorithms, as well. This time, however, special emphasis is placed on methods that come from the classic algorithms, merged with several modifications. Handed code files that are related to the exercise solution are: `bisectionMethodModified.m`, `newtonRaphsonMethodModified.m`, `secantMethodModified.m` and `mainExercise2.m`.

3.1 Wording

Implement:

(a) a modified version of the Newton-Raphson method where:

$$x_{n+1} = x_n - \frac{f(x_n}{f'(x_n)} - \frac{1}{2}\frac{f(x_n)^2 f''(x_n)}{f'(x_n)^3}$$

(b) a modified version of the bisection method, where the root guess is not the middle of the current interval, but a randomly picked point within this interval.

(c) a modified version of the secant method that gets three (3) initial guesses x_n, x_{n+1}, x_{n+2} and calculates the next guess for the root based on the formula:

$$x_{n+3} = x_{n+2} - \frac{r(r-q)(x_{n+2} - x_{n+1}) + (1-r)s(x_{n+2} - x_n)}{(q-1)(r-1)(s-1)}$$

where $q = \frac{f(x_n)}{f(x_{n+1})}$, $r = \frac{f(x_{n+2})}{f(x_{n+1})}$ and $s = \frac{f(x_{n+2})}{f(x_n)}$. The new point x_{n+3} is introduced in the formula replacing the oldest point amongst the three, thus x_n. The algorithm keeps executing until the convergence is achieved within the accepted tolerance of error. In essence, this method finds a parabola that passes through the three points and subsequently spots the intersection point between the parabola and the x-axis.

1. Find all the roots of the function $f(x) = 94\cos^3 x - 24\cos x + 177\sin^2 x - 108\sin^4 x - 72\cos^3 x \sin^2 x - 65$ within the interval $[0, 3]$ with accuracy of five decimal places. Use the above modified methods, choosing appropriate initialization each time.

2. Run algorithm (B) ten (10) times and investigate whether it converges always to same number of iterations or not.

3. Compare in an experimental way the classic with the modified versions of the algorithms in terms of how fast do they converge.

3.2 Basic Analysis

The theoretical background for this second exercise is almost identical with the analysis made for the first exercise. In fact, we could claim that the exercise extends the already presented algorithms, focusing on modified versions of them.

Apart from these slight modifications, the difference is spotted in the function $f(x)$ given by the exercise. In the current case, we have that $f(x) = 94\cos^3 x - 24\cos x + 177\sin^2 x - 108\sin^4 x - 72\cos^3 x \sin^2 x - 65$, given within the interval $[0, 3]$. Taking into consideration the new input function, we need to demonstrate its roots and its derivatives.

Concerning the roots, we are going to adopt a graph-oriented approach, exactly as we did for the first exercise. Observing the figure of f in the given interval, we can find out that the function intersects the x-axis three times. Each one of these intersections represents a root for the function. Apart from the geometric way of root-finding, we can disclose them in an algebraic way, as well.

The detailed way of finding in paper the roots is not in the context of this assignment, so we are going to take for granted that the three roots of f in the given interval are $x_1 = 0.84107, x_2 = 1.04719, x_3 = 2.30052$.

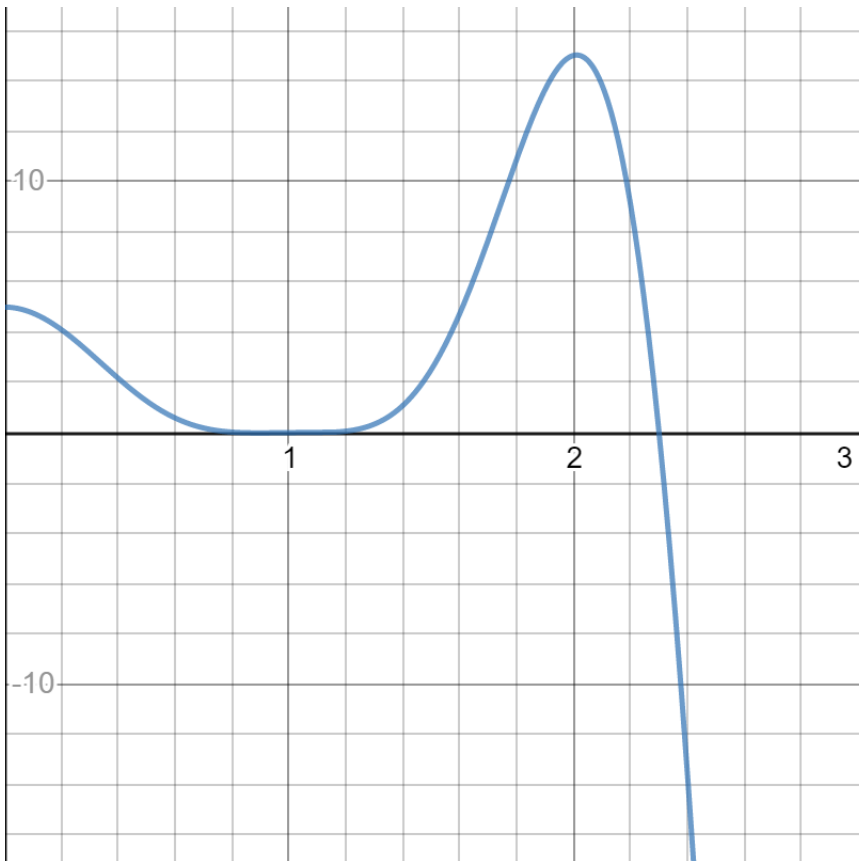


Figure 2: The plot of $f(x)$ in [0, 3]

In addition, for the given f(x) we can calculate its first and second derivative, since we are going to utilize them inside the methods' formulas. So, executing algebraic calculations, we can claim that:

$$f'(x) = 216\cos^2(x)\sin^3(x) - 432\cos(x)\sin^3(x) - 144\cos^4(x)\sin(x)$$
$$-282\cos^2(x)\sin(x) + 354\cos(x)\sin(x) + 24\sin(x)$$

and

$$f''(x) = (432 - 432\cos(x))\sin^4(x) + \left(1224\cos^3(x) - 1296\cos^2(x) + 564\cos(x) - 354\right)\sin^2(x) - 144\cos^5(x)$$
$$-282\cos^3(x) + 354\cos^2(x) + 24\cos(x)$$

In the subsections that follow, we are going to utilize the above, in order to present the modified root-finding algorithms. Special emphasis will be given on the code implementation in GNU Octave.

3.3 The modified bisection method

In this subsection we are going to discuss about the root-finding process for our function f(x) within the interval of interest ([0, 3]), utilizing a modified version of the bisection method. Code files involved: `bisectionMethodModified.m` and `mainExercise2.m`.

3.3.1 Theoretical Background

According to the requirements of the exercise, the modified bisection method algorithm, shares in common with the classic version the vast majority of specs. More detailed, the prerequisite (Bolzano's Theorem) and the basic idea (splitting the initial interval and guessing for the root) remain intact, in comparison with the bisection method we have already presented.

The slight modification lies in the root guess the algorithm executes in each step of its execution. More specifically, whilst the classic version was placing a rather "safe" guess on the middle of the interval, the alternated method chooses a random number within it.

This last ascertainment, leads to a small modification in the maximum error in calculation in each step of the algorithm. As we have already presented, the absolute error for a root guess made by the classic algorithm was $\frac{b-a}{2}$, which in natural language can be described as the half of the interval's length. On the contrary, in case of the modified method, the maximum error in the root guess is twice as big.

Such a claim can be justified by a simple worst-case analysis: Let $[a, b]$ be the interval of interest at a specific step of the algorithm execution. Let's suppose that the actual root of the function contained in this interval is $x_{actual} = a$. Unable to have this information in advance, the algorithm makes a random guess, going for a randomly selected number in the interval. It is apparent, thus, that the worst-case scenario is for the algorithm to pick b as the root guess.

As a result, the maximum absolute error in root calculation for the algorithm is identical with the interval's length. In math words this can be written as: $|error| = (b - a)$, given that $[a, b]$ is the interval that the algorithm investigates in its specific step.

Except for the root guess and the maximum error, the modified version of the bisection method has nothing to differ from its classic predecessor. Bearing this in mind, we can easily implement the algorithm into code.

3.3.2 GNU Octave code

```
function [root, iterationsExecuted] = bisectionMethodModified(aFunction,
                                                              leftLimit,
                                                              rightLimit,
                                                              decimalPlacesAccuracy)

    if (leftLimit > rightLimit)
        [leftLimit, rightLimit] = swapNumbers(leftLimit, rightLimit);
    endif

    iterationsExecuted = 0;
    desiredAccuracy = 0.5 * 10^(-decimalPlacesAccuracy);

    if (aFunction(leftLimit)*aFunction(rightLimit) > 0)
        error("bisectionMethodModified: The given interval does not meet the Bolzano's
    Theorem requirements. The product f(a)*f(b) must be non-positive!");
    elseif (aFunction(leftLimit) == 0)
        root = leftLimit;
        return;
    elseif (aFunction(rightLimit) == 0)
        root = rightLimit;
        return;
    endif

    while ((rightLimit - leftLimit) >= desiredAccuracy)
        iterationsExecuted++;
        middle = leftLimit + (rightLimit - leftLimit)*rand();
        if (aFunction(middle) == 0)
            root = middle;
            return;
        elseif (aFunction(leftLimit)*aFunction(middle) < 0)
            rightLimit = middle;
        else
            leftLimit = middle;
        endif
    endwhile
    root = leftLimit + (rightLimit - leftLimit)*rand();
endfunction
```

Listing 14: GNU Octave code for modified bisection method

3.3.3 Intervals selection

Up to that point, we have browsed through the theoretical background of the modified bisection method, as well as its code implementation. Utilizing the method, in order to find the actual roots of the given function, comes naturally as the next step of our reasoning.

As a matter of fact, the actual task of this last step lies in picking appropriate intervals (a, b) each time when calling the method. The ultimate goal is to feed the function with appropriate initial data, in order for the algorithm to converge to all the 3 of the function roots (converge to one root at a call) in the interval of interest. Throughout the process, obeying the Bolzano's Theorem remains always the priority.

Adopting a trial-and-error approach, we can indicatively present the following intervals that are checked to make the modified algorithm converge to all the three roots of the function in $[0, 3]$. These intervals are depicted in the following listings:

```
[root, iterations] = bisectionMethodModified(f, 0, 1, 5);
```

Listing 15: Starting from (0, 1) the modified bisection method converges to the root r_1=0.84107

```
[root, iterations] = bisectionMethodModified(f, 1, 2, 5);
```

Listing 16: Starting from (1, 2) the modified bisection method converges to the root r_2=1.04719

```
[root, iterations] = bisectionMethodModified(f, 2, 3, 5);
```

Listing 17: Starting from (1, 2) the modified bisection method converges to the root r_3=2.30052

3.3.4 Number of iterations

In contrast with the classic version of the bisection method, the modified version of it does not converge to a specific root within the same number of steps, even if the initial interval is identical at call time. This claim can be naively approached by the fact that the modified version picks a random number within the interval as a root guess, thus, it is not certain that the interval split will be the same on every call.

As a matter of fact, the interval split is always different, and that affects in the first place the number of iterations executed by the algorithm. This ascertainment can also be reached by an experimental way of reasoning. The following tables host the number of iterations executed by the algorithm, when the modified function is called ten (10) times. All the calls are identical, which means that the initial interval is always the same, as presented in the previous subsection. The numbers speak from themselves, depicting the variation in the number of iterations executed, while the input (function f and initial interval) remains fixed:

The above table was filled out by running a simple script that calls 10 times the method with the same input and keeps the number of iterations. The process (run the algorithm 10 times) was repeated three times, one for each root.

The numbers presented are completely indicative and have been putted down in a single execution of the simple script. It is obvious that we are going to have a different series of iterations every time we follow the same process. The important thing, however, is not spotting the specific numbers, but showing in an experimental way that this number varies.

3.4 The modified Newton-Raphson method

In this subsection we are going to discuss about the root-finding process for our function f(x) within the interval of interest ([0, 3]), utilizing a modified version of the Newton-Raphson method. Code files involved: `newtonRaphsonMethodModified.m` and `mainExercise2.m`.

wanted root	a	b	iterations	wanted root	a	b	iterations	wanted root	a	b	iterations
0.84107	0	1	23	1.04719	1	2	23	2.30052	2	3	19
0.84107	0	1	29	1.04719	1	2	17	2.30052	2	3	33
0.84107	0	1	19	1.04719	1	2	19	2.30052	2	3	26
0.84107	0	1	24	1.04719	1	2	23	2.30052	2	3	28
0.84107	0	1	27	1.04719	1	2	23	2.30052	2	3	20
0.84107	0	1	28	1.04719	1	2	27	2.30052	2	3	22
0.84107	0	1	22	1.04719	1	2	22	2.30052	2	3	32
0.84107	0	1	26	1.04719	1	2	24	2.30052	2	3	17
0.84107	0	1	22	1.04719	1	2	27	2.30052	2	3	27
0.84107	0	1	20	1.04719	1	2	20	2.30052	2	3	32

Table 6: : Number of iterations calculated on the fly for the modified bisection method

3.4.1 Theoretical background

Taking into account the requirements of the exercise, we can easily observe that the modified version of the Newton-Raphson method shares the same theoretical background with its classic predecessor. As a matter of fact, the only difference lies in the formula used to calculate the next guess for the root, thus the next x value, in order to keep the iteration going.

As a result, there is almost nothing else to say on a theoretical base, than to remind the formula for the next x value, as it is given in the wording of the exercise:

$$x_{n+1} = x_n - \frac{f(x_n}{f'(x_n)} - \frac{1}{2}\frac{f(x_n)^2 f''(x_n)}{f'(x_n)^3}$$

Utilizing the above equation, and combining it with the already known principles of the classic Newton-Raphson method, make it quite easy for us to transfer the algorithm into code, written in GNU Octave.

3.4.2 GNU Octave code

```
function [root, iterationsExecuted] = newtonRaphsonMethodModified(aFunction,
                                                                  itsDerivative,
                                                                  itsSecondDerivative,
                                                                  startingPoint,
                                                                  decimalPlacesAccuracy)

    iterationsExecuted = 0;
    if (aFunction(startingPoint) == 0)
        root = startingPoint;
        return;
    endif

    desiredAccuracy =0.5*10^(-decimalPlacesAccuracy);
    x = startingPoint;

    do
        iterationsExecuted++;
        previous = x;
        x = x - aFunction(x)/itsDerivative(x) - 0.5*(aFunction(x)^2*itsSecondDerivative(x))
    /(itsDerivative(x)^3);
    until (abs(previous - x) < desiredAccuracy)

    root = x;
endfunction
```

Listing 18: GNU Octave code for the modified Newton-Raphson method

3.4.3 Initial point selection

Aiming to disclose the three roots of the function in the given interval, we need to utilize the modified algorithm inside a "drive" script. The most interesting part of this procedure is finding appropriate initial point for the algorithm to start iterations from and converge to the roots. The following listings show some indicative selections for x_0, that are checked to make the algorithm converge to the wanted roots:

```
[root, iterations] = newtonRaphsonMethodModified(f, f_derivative, f_second_derivative, 0.5,
    5);
```

Listing 19: Starting from x_0=0.5 the modified bisection method converges to the root r_1=0.84107

```
[root, iterations] = newtonRaphsonMethodModified(f, f_derivative, f_second_derivative, 1, 5,
     2);
```

Listing 20: Starting from x_0=1 the modified bisection method converges to the root r_2=1.04719

```
[root, iterations] = newtonRaphsonMethodModified(f, f_derivative, f_second_derivative, 2.5,
    5);
```

Listing 21: Starting from x_0=2.5 the modified bisection method converges to the root r_3=2.30052

3.4.4 Number of iterations

As we have already proven, the classic Newton-Raphson method features, in general, quadratic convergence. Based on an experimental approach, the modified method seems to be rather as quick as its classic predecessor. The following table presents the number of iterations executed by the modified Newton-Raphson method, for each one of the wanted roots.

wanted root	x_0	iterations
0.84107	0.5	6
1.04719	1	14
2.30052	2.5	4

Table 7: : Number iterations executed by the Newton-Raphson method

3.5 The modified secant method

In this subsection we are going to discuss about the root-finding process for our function f(x) within the interval of interest ([0, 3]), utilizing a modified version of the secant method. Code files involved: `secantMethodModified.m` and `mainExercise2.m`.

3.5.1 Theoretical Background

Taking into consideration the explanations provided in the wording of the exercise, the modified version of the secant method just differs from the classic one in the formula used to find the next root guess. In essence, the function requires three initial guesses. Based on them, the algorithm finds a parabola that passes through the three points and subsequently spots the intersection point between the parabola and the x-axis. In math words, the formula for the next root guess is the following:

$$x_{n+3} = x_{n+2} - \frac{r(r-q)(x_{n+2} - x_{n+1}) + (1-r)s(x_{n+2} - x_n)}{(q-1)(r-1)(s-1)}$$

where $q = \frac{f(x_n)}{f(x_{n+1})}$, $r = \frac{f(x_{n+2})}{f(x_{n+1})}$ and $s = \frac{f(x_{n+2})}{f(x_n)}$.

3.5.2 GNU Octave code

```
function [root, iterationsExecuted] = secantMethodModified(aFunction,
                                                           startingPoint1,
                                                           startingPoint2,
                                                           startingPoint3,
                                                           decimalPlacesAccuracy)
    iterationsExecuted = 0;
    if (aFunction(startingPoint1) == 0)
        root = startingPoint1;
        return;
    elseif (aFunction(startingPoint2) == 0)
        root = startingPoint2;
        return;
    endif

    desiredAccuracy =10^(-decimalPlacesAccuracy);
    x_n1 = startingPoint1;
    x_n2 = startingPoint2;
    x = startingPoint3;

    do
        iterationsExecuted++;
        x_n = x_n1;
        x_n1 = x_n2;
        x_n2 = x;
        q = calculateFraction(aFunction, x_n, x_n1);
        r = calculateFraction(aFunction, x_n2, x_n1);
        s = calculateFraction(aFunction, x_n2, x_n);
        x = x_n2 - (r*(r-q)*(x_n2 - x_n1) + (1-r)*s*(x_n2 - x_n))/((q-1)*(r-1)*(s-1));
    until (abs(x_n2 - x) < desiredAccuracy || aFunction(x) == 0)

    root = x;
endfunction
```

Listing 22: GNU Octave code for the modified secant method

3.5.3 Initial guesses' selection

Finding the roots of the function within the given interval, utilizing the modified method, is the next matter we are going to discuss. The issue can be re-phrased to finding appropriate initializations for our function to pass as arguments at call time. The following listings show the indicatively chosen initializations that make the algorithm converge to each one of the three wanted roots.

```
[root, iterations] = secantMethodModified(f, 0, 0.5, 0.75, 5);
```

Listing 23: Starting from x_0=0, x_1=0.5 and x_2=0.75, the modified secant method converges to the root r_1=0.84107

```
[root, iterations] = secantMethodModified(f, 0.9, 1.1, 1.2, 5);
```

Listing 24: Starting from x_0=0.9, x_1=1.1 and x_2=1.2, the modified secant method converges to the root r_2=1.04719

```
[root, iterations] = secantMethodModified(f, 1.5, 1.8, 2.3, 5);
```

Listing 25: Starting from x_0=1.5, x_1=1.8 and x_2=2.3, the modified secant method converges to the root r_3=2.30052

3.5.4 Number of iterations

The following table shows the number of iterations executed by the algorithm for each one of the wanted roots:

wanted root	x_0	x_1	x_2	iterations
0.84107	0	0.5	0.75	7
1.04719	0.9	1.1	1.2	22
2.30052	1.5	1.8	2.3	4

Table 8: : Number iterations executed by the modified secant method

3.6 Head to head: Classic vs Modified

Up to that point, we have covered in detail two different versions of the three iterative root-finding algorithms. More specifically, we have presented theoretically and implemented in code, both the classic and a modified version of the bisection, the Newton-Raphson and the secant method. A reasonable question that comes out easily is which version of each algorithm is, eventually, better, concerning how quick they converge.

In order to find out, we are going to adopt a testing-oriented approach. First of all, we need to create a uniform testing environment, in order to get safer conclusions. For that reason we are going to pick the function f of the current exercise, thus $f(x) = 94\cos^3 x - 24\cos x + 177\sin^2 x - 108\sin^4 x - 72\cos^3 x\sin^2 x - 65$. That is the function that all the algorithms will be called to deal with.

Apart from the function uniformity, another step to enhance the reliability of the testing environment is to pursue identical, or, at least, almost identical initializations for the respective couples of algorithms. That means the classic and the modified version of the bisection method will be starting from the same initial interval. Completely respectively, the initializations for the two versions of the Newton-Raphson method and the two versions of the secant method will be chosen to match as much as possible.

Utilizing this testing environment, we are going to examine and compare by two the versions of each algorithm, in terms of the number of iterations as well as the time elapsed during their execution. Throughout the process, the environment parameters will remain intact. That means the function f and the appropriate initializations will be uniform for each compared couple.

3.6.1 Round One: The bisection method

In order to compare the two versions of the bisection method, we call each one of them $n = 1000$ times, using exactly the same initializations. After the n calls of each version, we calculate the mean number of iterations, as well as the mean time elapsed for each one of the n executions of the algorithm. The script in GNU Octave that goes through this procedure is contained in `compareBisectionMethodVersions.m` and `bisectionComparisonDriver.m` code files. The following table shows indicative results after running the above script.

			Classic bisection		**Modified bisection**	
root	a	b	iterations	time elapsed	iterations	time elapsed
0.84107	0	1	17	2.05 millis	25.75	3.25 millis
1.04719	1	2	17	1.98 millis	20.98	2.60 millis
2.30052	2	3	17	2.05 millis	25.35	3.25 millis

Table 9: : Comparison between the classic and the modified bisection method

Trying to make a few notices on the results, we could claim that the classic version of the bisection algorithm features better performance on both fields: number of iterations and time of execution. Such a conclusion, of course, cannot at all be generalized for the two versions, concerning every function or initial interval. However, for the specific function $f(x)$ and the specific initial intervals, the classic version of the algorithm seems to be the ultimate winner.

3.6.2 Round Two: The Newton-Raphson method

Exactly the same procedure is followed for the Newton-Raphson method. The two versions are set to start from the same initial guess and executed $n = 1000$ times each. The procedure is implemented in `compareNewtonRaphsonMethodVersions.m` and `newtonRaphsonComparisonDriver.m` code files. The average iterations and time of execution are presented in the following table.

		Classic Newton-Raphson		**Modified Newton-Raphson**	
root	x_0	iterations	time elapsed	iterations	time elapsed
0.84107	0.5	8	1.12 millis	6	1.69 millis
1.04719	1	20	2.73 millis	14	3.78 millis
2.30052	2.5	5	0.70 millis	4	1.13 millis

Table 10: : Comparison between the classic and the modified Newton-Raphson method

In contrast with the bisection method, the Newton-Raphson seems to perform under better convergence rate in its modified version, rather than the classic one. As a matter of fact, for all three roots of the function, the modified version converges in less iterations, than the classic alternative.

Nevertheless, it is really interesting to notice something that may seem rather bizarre in the first sight. Although the number of iterations of the modified version is, in all cases, minimum, it is not the quickest to find the root in terms of absolute execution time! In other words, a single iteration is faster in the classic version, than in the modified one, leading in this interesting difference.

A claim that can probably serve as justification for this observation is the fact that the formula for the next guess is of higher computational expense, concerning the modified version than the classic one. In simple words, the algebraic calculations, as well as the function calls are significantly more inside the formula of the modified version. Thus, it takes more time for the system to operate them and calculate the next x-value, in order to repeat iteration step.

In brief, the modified Newton-Raphson method converges quicker to the wanted root. However, the computational load is measurably higher, making the modified version slower (in absolute execution time) than its classic predecessor.

3.6.3 Round Three: The secant method

Having presented twice and in detail our approach, it is needless to say that we are going to follow the same reasoning for the last couple of algorithms, the classic and the modified secant method. However, we need to point out that, in this method, the number of initial guesses passed as arguments differs amongst the two versions. As a matter of fact, the classic method needs two initial guesses, while the modified one needs one more, in order to calculate a parabola.

Taking into account this specialty, we are going to preserve our uniform testing environment to the greatest possible extent. That means we are going to keep the two initial guesses identical for the two versions, and simply add one more initial guess, when dealing with the modified version.

The above tactics may impose some shadows of uniformity in our testing environment. Nevertheless, within the context of this assignment, we can consider this approach reliable enough for us, in order to deduce simple, rough and experimentally-driven conclusions. The following table depicts the results of running $n = 1000$ times both the versions.

				Classic secant		**Modified secant**	
root	x_0	x_1	x_2	iterations	time elapsed	iterations	time elapsed
0.84107	0	0.5	0.75	10	1.98 millis	7	2.79 millis
1.04719	0.9	1.1	1.2	24	4.63 millis	22	8.49 millis
2.30052	1.5	1.8	2.3	36	6.88 millis	4	1.59 millis

Table 11: : Comparison between the classic and the modified bisection method

The results are rather expected, since the secant method is actually a sub-category of the Newton-Raphson method. In this case, as well, the modified version of the algorithm converges quicker in terms of iterations executed, However, the classic method takes less absolute time to execute, in 2 out of the 3 roots. The computational load of the modified version can justify for such an ascertainment, as we have already analyzed.

In conclusion, there is no clear winner between the classic and the modified "team" of algorithms. Except for the bisection method field (where the modified version has by far a better performance), in the other two fields the result is rather balanced. The computational expense of the modified versions compensates for the surplus number of iterations executed by the classic ones, and vice versa. In general, we could claim that the version selection should be in close relation with the computational capacities of the system, as well as, the context of the root-finding process.

4 Exercise Three

The third exercise focuses on systems of linear equations. The tasks are programming-oriented and cover the basic operations on square matrices. Handed code files that are related to the exercise solution are: `gauss.m`, `cholesky.m`, `gaussSeidel.m` and the files contained in the same directories with them.

4.1 Wording

Suppose that we have to solve the following linear system:

$$\mathbf{Ax} = \mathbf{b}$$

1. Write in any programming language a function that takes as input the matrix $\mathbf{A}$ and the vector $\mathbf{b}$ of the above system. The function should return the vector $\mathbf{x}$ of the unknowns (e.g. `[x] = gauss(A, b)`). The solution of the system has to follow the methodology $\mathbf{PA} = \mathbf{LU}$ of solving systems of linear equations.
2. Write in any programming language a function that takes as input a symmetric and positively-definite matrix $\mathbf{A}$. The function should return a lower-triangular matrix $\mathbf{L}$ that constitutes the Cholesky decomposition of the matrix $\mathbf{A}$.
3. Write in any programming language a function that implements the Gauss-Seidel method. Utilize the method, in order to solve within accuracy of 4 decimal places (using the infinite norm in the solution error) the $n \times n$ sparse system $\mathbf{Ax} = \mathbf{b}$, for $n = 10$ and $n = 1000$, with $A(i,i) = 5$, $A(i+1,i) = A(i,i+1) = -2$ and $\mathbf{b} = [3, 1, 1, \dots, 1, 1, 3]^T$.

$$\begin{bmatrix} 5 & -2 & & & \\ -2 & 5 & -2 & & \\ & \ddots & \ddots & \ddots & \\ & & -2 & 5 & -2 \\ & & & -2 & 5 \end{bmatrix} \begin{bmatrix} x_1 \\ \\ \vdots \\ \\ x_n \end{bmatrix} = \begin{bmatrix} 3 \\ 1 \\ \vdots \\ 1 \\ 3 \end{bmatrix}$$

Note: You cannot use built in functions (e.g., chol(), lu(), left division operand, etc.).

4.2 The PA = LU factorization

The $\mathbf{PA} = \mathbf{LU}$ factorization is a smart, improved way of executing Gaussian elimination, in order to solve a linear system of equations. In essence, the gist of the factorization is based on the "naive", conventional Gaussian elimination, extended with mechanisms that are developed to deal with the problems emerging in it.

In specific, the $\mathbf{PA} = \mathbf{LU}$ factorization for a non-singular matrix can successfully avoid the problems of encountering a zero pivot, as well as swamping. The main idea is that before starting elimination with a pivot, the algorithm checks whether it is needed to execute row exchanges. The reason behind these exchanges is to use the greater (in terms of absolute value) candidate pivot and, thus, avoid the problems described.

In matrices form, the row exchanges are represented as a Permutation Matrix $\mathbf{P}$, that left-multiplies $\mathbf{A}$ and executes the row exchanges. The matrix $\mathbf{P}$ is, in fact, an identity matrix with row exchanges applied on it.

Concerning the right hand side of the factorization, the matrices $\mathbf{L}$ and $\mathbf{U}$ are named after their form. $\mathbf{L}$ is a lower triangular matrix and $\mathbf{U}$ is an upper triangular one. When $\mathbf{L}$ and $\mathbf{U}$ are multiplied in that order, they produce the initial matrix $\mathbf{A}$, with appropriate row exchanges for assisting Gaussian elimination. The factorization is a typical example of a long computing tradition in science and engineering; splitting a complicated task into smaller, simpler and easier to solve parts. In total, writing the coefficient matrix $\mathbf{A}$ as a product of $\mathbf{L}$ and $\mathbf{U}$ comes through the well-known Gaussian elimination step.

Once $\mathbf{L}$ and $\mathbf{U}$ are known, the problem $\mathbf{Ax} = \mathbf{b}$ can be written as $\mathbf{LUx} = \mathbf{b}$. The trick on that point is to define a new "auxiliary" vector $\mathbf{c} = \mathbf{Ux}$. Then, back substitution is a two-step procedure:

1. Solve Lc = b for c.
2. Solve Ux = c for x.

Both steps are straightforward since L and U are triangular matrices.

4.2.1 GNU Octave code

Remaining loyal to the principles of top-down programming, we are going to partition the factorization into smaller, easier to handle segments. First of all, the basic function has pretty simple syntax, utilizing other, smaller functions:

```
1  function vectorX = gauss(aMatrix,
2                           aRightHandSideVector)
3
4    if (rows(aMatrix) != columns(aMatrix))
5      error("gauss: The input matrix is not square! I can't put up with this!");
6    endif
7
8    [matrixP, matrixL, matrixU] = PA_LU(aMatrix);
9
10   vectorB = matrixP * aRightHandSideVector;
11   vectorC = forwardSubstitution(matrixL, vectorB);
12   vectorX = backSubstitution(matrixU, vectorC);
13
14 endfunction
```

Listing 26: GNU Octave code for Gaussian elimination, using **PA** = **LU** factorization

The most important function call is the one in line 8 of the `gauss` function. `PA_LU` is a function that takes as input the coefficient matrix **A** and returns the matrices **P**, **L** and **U**, performing **PA** = **LU** factorization.

```
function [matrixP, matrixL, matrixU] = PA_LU(aMatrix)
  rowsOfMatrix = rows(aMatrix);
  columnsOfMatrix = columns(aMatrix);

  matrixP = eye(size(aMatrix));
  matrixL = eye(size(aMatrix));

  for parsingIndex = 1:rowsOfMatrix
    pivotRow = findPivotInColumn(parsingIndex, aMatrix);
    if (parsingIndex != pivotRow)
      aMatrix = exchangeRows(parsingIndex, pivotRow, aMatrix, parsingIndex);
      matrixL = exchangeRows(parsingIndex, pivotRow, matrixL, 1, parsingIndex-1);
      matrixP = exchangeRows(parsingIndex, pivotRow, matrixP);
    endif

    for rowToEliminate = parsingIndex+1:rowsOfMatrix
      multiplier = aMatrix(rowToEliminate, parsingIndex) / aMatrix(parsingIndex,
    parsingIndex);
      aMatrix(rowToEliminate, parsingIndex+1:columnsOfMatrix) -= multiplier*aMatrix(
    parsingIndex, parsingIndex+1:columnsOfMatrix);
      matrixL(rowToEliminate, parsingIndex) = multiplier;
    endfor
  endfor

  matrixU = aMatrix;
endfunction
```

Listing 27: GNU Octave code for the **PA** = **LU** factorization

For enhanced code readability, the `PA_LU` function utilizes, as well, a couple of auxiliary sub-functions. First of all, the function `findPivotInColumn` gets as input an index and a matrix and returns the pivot index. In essence, the function scans all the items starting from the main diagonal (contained) and spots the greater element in the desired column, in terms of absolute value. The index of the pivot found is returned, indicating the row it lies in.

```
function pivotRowIndex = findPivotInColumn(aColumn, aMatrix)
  pivotRowIndex = aColumn;
  for otherRow = aColumn+1:rows(aMatrix)
    if abs(aMatrix(otherRow, aColumn)) > abs(aMatrix(pivotRowIndex, aColumn))
      pivotRowIndex = otherRow;
    endif
  endfor
endfunction
```

Listing 28: GNU Octave function for finding pivot in a certain column of **A**

Another simple but important function is `exchangeRows`, which performs row exchanges on a matrix. The function is declared with default start and end values for the column range, and can be called in a polymorphic variety of ways (with or without starting and ending column). During the factorization, row exchanges play a crucial role and are operated on all matrices, **P**, **A** and **L**.

```
function finalMatrix = exchangeRows(rowIndexA,
                                    rowIndexB,
                                    aMatrix,
                                    startColumn = 1,
                                    endColumn = columns(aMatrix))

  columnRange = startColumn:endColumn;

  temp = aMatrix(rowIndexA, columnRange);
  aMatrix(rowIndexA, columnRange) = aMatrix(rowIndexB, columnRange);
  aMatrix(rowIndexB, columnRange) = temp;

  finalMatrix = aMatrix;

endfunction
```

Listing 29: GNU Octave code operating row exchange on a matrix

An important note that has to be made on that point is the fact that the row exchanges are performed by `PA_LU` in a greedy (programming) way. More specifically, at each step of its execution, the algorithm completely ignores the elements that, up to that point, have ”locked” their placement in the don't-care, below the main diagonal of **A** area.

As a result, as long as an element is definitely placed in this don't-care area, the algorithm involves it in no operations at all, any longer. In simple words, the elements that (up to that point) is definitely sure they are going to remain below the main diagonal till the end of the factorization, are not multiplied, eliminated or exchanged any more.

Such a strategy may seem odd or even incorrect, concerning the way we are used to perform Gaussian elimination by hand and in paper. In other words, the algorithm skips the - satisfying for the human eye - step of placing zeros below the main diagonal. However, the actual fact is that the algorithm will never need to return to that elements, neither in the phase of back substitution. As a result, multiplying or exchanging those elements would be a time and computational waste, which would result in higher time complexity and time of execution.

For all these purposes, the algorithm makes carefully selected multiplications and row exchanges on specific column range, in order to save unnecessary operations. Taking it a step further, the same can be claimed for the matrix **L**, which is lower triangular, thus has its don't-care area is located above its main diagonal.

The above conditioning is really important to have in mind, when examining the intermediate results of the `PA_LU` execution. More specifically, the method calculates the vector **x** of unknowns. However, if for some reason we want to take a look on the matrices **U** or **L** that are formed, we have to bear in mind that their don't-care area will not be filled with zeros, but with ”garbage”. Nevertheless, their important area (below or above the main diagonal respectively) will be filled with the correct numbers, ensuring the validity of the finally calculated vector **x** of unknowns.

Returning back to the code, the `PA_LU` function has nothing special left. The heavy work is done and matrices **P**, **L** and **U** are returned to the `gauss` function. The next two steps are straight forward and involve solving twice a system that contains a triangular matrix (once lower and once upper triangular). The functions `forwardSubstitution` and `backwardSubstitution` are simple functions for these two steps.

```
function x = forwardSubstitution(L, b)

  [n,n] = size(L);
  for k = 1:n
    j = 1:k-1;
    x(k) = (b(k) - L(k,j)*b(j))/L(k,k);
  end
  x = x';
endfunction
```

Listing 30: GNU Octave code for solving $\mathbf{Lc} = \mathbf{b}$ for c, $\mathbf{L}$ is lower triangular

```
function x = backSubstitution(U, x)
  [n,n] = size(U);

  for k = n:-1:1
    j = k+1:n;
    x(k) = (x(k) - U(k,j)*x(j))/U(k,k);
  end
endfunction
```

Listing 31: GNU Octave code for solving $\mathbf{Ux} = \mathbf{b}$ for x, $\mathbf{U}$ is upper triangular

In total, the solution of the last system is, as well, the final solution of the initial problem. As a result, the vector **x** of the unknowns is returned

4.3 Cholesky decomposition

The Cholesky decomposition is a useful algorithm that can be applied on symmetric, positive-definite matrices. The main idea of the algorithm is to write the matrix **A** of interest as a product $\mathbf{A} = \mathbf{R^T R}$, where **R** is an upper triangular matrix.

To start with, the matrix **A** is partitioned as:

$$A = \left[\begin{array}{c:c} a & b^T \\ \hdashline b & C \end{array}\right]$$

where **b** is an $(n-1)$-vector and C is an $(n-1)\times(n-1)$ sub-matrix. The next step is to set $u = b/\sqrt{a}$, $A_1 = C - uu^T$ and define invertible matrix S:

$$S = \left[\begin{array}{c:c} \sqrt{a} & u^T \\ \hdashline 0 & \\ & \\ \vdots & I \\ 0 & \end{array}\right]$$

Utilizing the matrix S yields

$$S^T \left[\begin{array}{c:ccc} 1 & 0 & \dots & 0 \\ \hdashline 0 & & & \\ & & & \\ \vdots & & A_1 & \\ 0 & & & \end{array}\right] S = \left[\begin{array}{c:ccc} \sqrt{a} & 0 & \dots & 0 \\ \hdashline & & & \\ u & & I & \\ & & & \end{array}\right] \left[\begin{array}{c:ccc} 1 & 0 & \dots & 0 \\ \hdashline 0 & & & \\ & & & \\ \vdots & & A_1 & \\ 0 & & & \end{array}\right] \left[\begin{array}{c:c} \sqrt{a} & u^T \\ \hdashline 0 & \\ & \\ \vdots & I \\ 0 & \end{array}\right] = \left[\begin{array}{c:c} a & b^T \\ \hdashline & \\ b & uu^T \quad + \quad A_1 \\ & \\ 0 & \end{array}\right] = A$$

So we can now easily end up to the final, wanted matrix R, which is defined as:

$$R = \left[\begin{array}{c:c} \sqrt{a} & u^T \\ \hdashline 0 & \\ & \\ \vdots & V \\ 0 & \end{array}\right]$$

where $A_1 = V^T V$ and V is upper triangular. We can easily check that $R^T R = A$. Bearing the above procedure in mind, we can transform the Cholesky decomposition into code.

4.3.1 GNU Octave code

```
function [RTranspose] = cholesky (aMatrix)

  n = rows(aMatrix);

  for k = 1:n
    if aMatrix(k, k) < 0
      return;
    endif

    R(k, k) = sqrt(aMatrix(k, k));
    UTranspose = (1 / R(k, k))*aMatrix(k, k+1:n);
    R(k, k+1:n) = UTranspose;
    aMatrix(k+1:n, k+1:n) = aMatrix(k+1:n, k+1:n) - (UTranspose)'*UTranspose;
  endfor
  RTranspose = (R)';
endfunction
```

Listing 32: GNU Octave code for the Cholesky decomposition

4.4 The Gauss-Seidel method

The Gauss-Seidel is a well-known method for solving linear systems of equations. In contrast with the Gaussian elimination, the Gauss-Seidel method is iterative and tries to calculate an approximation of the vector $\mathbf{x}$ of unknowns.

The basic requirement for the Gauss-Seidel method to converge towards the actual solution is the input matrix to be strictly diagonally dominant. Otherwise, there is no secure answer on whether the algorithm converges to or diverges from the solution. This does not mean the algorithm is going to always diverge in the absence of the requirement, it is just an important condition that can ensure convergence of the algorithm.

The main idea is that we start from an initial guess for the vector of unknowns. The method solves the i-th equation for the i-th unknown and iterates, just like as in a Fixed-Point iteration. During the process, the algorithm utilizes the most recent guess for each one of the unknowns, whenever needed.

Focusing on the problem the wording of the exercise describes, we are called to solve the sparse system

$$\begin{bmatrix} 5 & -2 & & & \\ -2 & 5 & -2 & & \\ & \ddots & \ddots & \ddots & \\ & & -2 & 5 & -2 \\ & & & -2 & 5 \end{bmatrix} \begin{bmatrix} x_1 \\ \\ \vdots \\ \\ x_n \end{bmatrix} = \begin{bmatrix} 3 \\ 1 \\ \vdots \\ 1 \\ 3 \end{bmatrix}$$

First of all, we can easily prove that the matrix is strictly diagonally dominant, since $|5| > |-2| + |-2|$ for rows 2 up to $n-1$. Completely respectively, $|5| > |-2|$ for rows 1 and n. Consequently, in all n rows of the matrix the absolute value of the main diagonal element is greater that the sum of the absolute values of all the other row elements.

Next, the exercise suggests that we can try solving the system, utilizing the Gauss-Seidel method, for $n = 10$ and $n = 10000$. In order to do so, we are going to introduce the code implementation of the algorithm in GNU Octave.

4.4.1 GNU Octave Code

```
function x = gaussSeidelMethod(aMatrix,
                               aRightHandSideVector,
                               xInitial,
                               decimalPlacesAccuracy)

  desiredAccuracy = 0.5*10^(-decimalPlacesAccuracy);

  if rows(aMatrix) != columns(aMatrix)
    error("gaussSiedelMethod: The input matrix is not square! I cannot put up with it!");
  endif
  if rows(aMatrix) != columns(xInitial)
    error("gaussSiedelMethod: Incorrect number of initial guesses!");
  endif

  x = xInitial;
  do
    xPrevious = x;
    unknown = 0;
    for unknownCounter = x
      unknown++;
      x(unknown) = aRightHandSideVector(unknown);
      otherUnknown = 0;
      for otherUnknownCounter = x
        otherUnknown++;
        if unknown == otherUnknown
          continue;
        endif
        x(unknown) -= aMatrix(unknown, otherUnknown)*x(otherUnknown);
      endfor
      x(unknown) /= aMatrix(unknown, unknown);
    endfor
  until (infiniteNormOf(subtractVectors(x, xPrevious)) < desiredAccuracy)

endfunction
```

Listing 33: GNU Octave code for the Gauss-Seidel method

We can now use the `gaussSeidelMethod` function in order to calculate the solution of the given sparse $n \times n$ system, for for $n = 10$ and $n = 10000$. The procedure is executed in detail in `1gaussSeidelMain.m` code file, that is handed.

Generally speaking, the algorithm succeeds in finding the solution of the system in both cases. It is rather obvious that $n = 10000$ requires much more time and computational load, in relation to the $n = 10$ case. As a result, it is inevitable that the execution time of the script is significantly greater in the first case.

A small note that can be made on that point is the fact that the Gauss-Seidel method, as an iterative algorithm, features a measurable number of loops, concerning the code implementation. This ascertainment, combined with the fact that GNU Octave is not the fastest language to proceed, when it comes to loop speed of execution, can justify for this significant difference in execution time between the two cases.

Nevertheless, there is no doubt imposed about the convergence of the algorithm. In all cases, the algorithm finally finishes its execution after a finite number of iterations, indicating as solution the $n \times 1$ vector

$$x = \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \\ 1 \end{bmatrix}$$

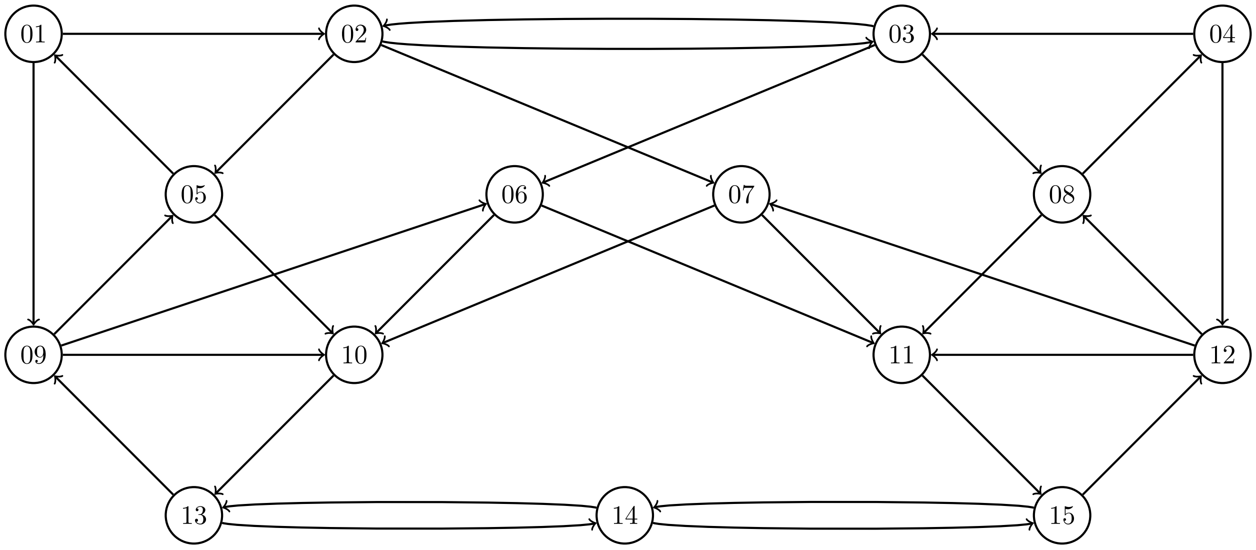
which is indeed the actual solution of the system.

5 Exercise Four

The fourth exercise focuses on the eigenvalues and eigenvectors of a matrix. In particular, special emphasis is given on an application of the eigenanalysis on the computer science field, the well-known page rank. The exercise is settled around a simple page rank model, featuring really interesting questions. Handed code files that are related to the exercise solution are: `powerMethod.m`, `mainPowerMethod.m`, `mainPowerMethodModified.m` and the files included in the same directories with them.

5.1 Wording

All search engines on the web, such as `http://www.google.com/`, are characterized for the astonishing results they return in queries asked by users. We are going to give a rough explanation on how pages are ranked utilizing linking information between sites on the World Wide Web (WWW). Google.com attributes a positive real number, the so-called *page rank*, to every page that is returned to a query. The page rank of a site is calculated with one of the greatest, as it is called, eigenanalysis algorithms, the so-called power method. In the following graph, every node represents a page on the web and an edge from i to j represents a link from page i to page j.



Suppose that $\mathbf{A} \in \mathbb{R}^{n\times n}$, $n = 15$. The adjacency matrix of the graph is the following.

$$\mathbf{A} = \begin{bmatrix}
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0
\end{bmatrix}$$

For the matrix $\mathbf{A}$ it is known that $\mathbf{A}_{(i,j)} = 1$, when there is linking from node i to node j. Otherwise, $\mathbf{A}_{(i,j)} = 0$.

The developers of google.com imagined a user within that network that is in page i with probability p_i. Afterwards, the user either moves to a random page (with specific probability q that is usually equal to 0.15), or selects randomly one of the page links contained in page i, with probability $1-q$. Overall, the probability that describes the fact that the user moves from page i to page $j$is given by the relation:

$$p_{\{i \to j\}} = \frac{q}{n} + (i - q) * \frac{\mathbf{A}_{(i,j)}}{n_i}$$

where n_i is the sum of the i^{th} row of $\mathbf{A}$ (in essence the number of links contained in page i). The probability the user at any time to be in page j is the sum of $p_{\{i \to j\}}$ on the i^{th} row of $\mathbf{A}$.

$$p_j = \sum_{i=1}^{n} \left(\frac{qp_i}{n} + (1-q)\frac{p_i}{n_i}\mathbf{A}_{(i,j)} \right) \tag{1}$$

Equation (1) can be also written in the form of matrices as:

$$\mathbf{p} = \mathbf{Gp} \tag{2}$$

where $\mathbf{p}$ is the vector that contains the probabilities of a user to be located on the n web pages and $\mathbf{G}$ is the matrix whose cell $\mathbf{G}_{(i,j)} = \frac{q}{n} + \frac{\mathbf{A}_{(j,i)}(1-q)}{n_j}$. From now on, we are going to call matrix $\mathbf{G}$ as *Google Matrix*. Each one of the columns of the Google Matrix sums to 1, thus the Google Matrix is stochastic and has maximum eigenvalue equal to 1.The eigenvector that corresponds to this eigenvalue 1 is the total of the non-variable probabilities, which, actually, is the rank of the page.

Based on this example, the page rank of each one of the pages for $q = 0.15$ are given below:

$$\mathbf{p} = [0.0268238, 0.0298611, 0.0298612, 0.0268241, 0.0395877, 0.0395878, \ldots$$
$$\ldots, 0.039588, 0.0395881, 0.074565, 0.106319, 0.10632, 0.0745657, 0.125089, \ldots, \ldots, , 0.11633, 0.12509]^T$$

The above eigenvector has been normalized, so that its sum to be equal to 1 (as probability laws imply). The page rank is higher for pages 13 and 15, followed by pages 14, 10 and 11. Note that the page rank does not only relies on the in-degree (number of pages that point to the current page), but is calculated in a more sophisticated way. Even though nodes 10 and 11 have the greatest in-degree, the fact that they point to pages 13 and 15 transfers their own significance on them. That is the idea behind the phenomenon of google-bombing, the tactics of enhancing the significance of a page, by persuading other pages of high traffic to point on it. Note the connection between terms ”rank” and ”significance” of a page. Page rank is, up to that moment, the best way to represent the significance of a page, even though no one knows for sure what significance of a page stands for. As a result, someone may find a better model to represent page significance.

Requested tasks

1. Prove in detail that matrix $\mathbf{G}$ is stochastic
2. Construct matrix $\mathbf{G}$ for the specific example and check that the eigenvector of the maximum eigenvalue is the one given above (using the power method).
3. Add 4 connections of your own choice and remove one of the already existent on the graph to construct a new matrix $\mathbf{G}$. The goal should be to enhance the significance of a page of your own choice.
4. Change the *transition probability* q to (a) $q = 0.02$ and (b) $q = 0.6$ on the new graph. Which is the aim of the transition probability?

5. Suppose that on the initial graph, page 11 wanted to enhance its rank in relation with its competitor, page 10. Suppose, also, that it wanted to accomplish it by persuading pages 8 and 12 to point to it in a better way. Let's suppose that we model such a case by replacing $\mathbf{A}_{(8,11)}$ and $\mathbf{A}_{(12,11)}$ with 3 (instead of 1) on the adjacency table. Does this strategy pay off?

6. investigate the impact of a possible deletion of page 10 from the graph. Which pages see their ranks to raise and which of them to decrease?

5.2 The stochastic Google Matrix G

To start with, we are going to prove in detail the fact that the Google Matrix is *stochastic.* Utilizing previous theoretical background, we know that a stochastic matrix is a square matrix, whose entries are non-negative real numbers (more often than usual representing probability). In addition, each column of the stochastic matrix needs to sum to 1, as the probability laws impose.

Based on the wording of the exercise, we have that each one of the entries of **G** is given by the formula:

$$\mathbf{G}_{(i,j)} = \frac{q}{n} + \frac{\mathbf{A}_{(j,i)}(1-q)}{n_j}$$

where $\mathbf{A}_{(i,j)}$ is the respective entry of the adjacency matrix, which can be either 1 or 0. Bearing this is mind, we can easily find out that **G** is indeed square (**A** is square as well) and has non-negative entries, since each one of its entries is the sum of non-negative quantities.

Having said the above, we now need to just show that the sum of each one of the columns of **G** is equal to 1. Before heading to the sum, we would like to demonstrate one important relation that will prove to be really helpful throughout the proof. According to the wording of the exercise *"n_i is the sum of the i^{th} row of* **A** *(in essence the number of links contained in page i)".* Transforming the above statement in natural language, into math words, we can claim that:

$$n_k = \sum_{i=1}^{n} \mathbf{A}_{(k,i)} \tag{1}$$

Keeping that in mind, we can now examine the sum of a random column, let say j_r, of matrix **G**. If we manage to prove that entries of j_r sum to 1, we have proven that **G** is stochastic. So, we have that:

$$\begin{aligned}\sum_{i=1}^{n} \mathbf{G}_{(i,j_r)} &= \sum_{i=1}^{n} \left(\frac{q}{n} + \frac{\mathbf{A}_{(j_r,i)}(1-q)}{n_{j_r}} \right) = \\ &\sum_{i=1}^{n} \frac{q}{n} + \sum_{i=1}^{n} \frac{(1-q)}{n_{j_r}} * \mathbf{A}_{(j_r,i)} = \\ &\frac{nq}{n} + \frac{1-q}{n_{j_r}} \sum_{i=1}^{n} \mathbf{A}_{(j_r,i)} = \\ &q + \frac{1-q}{n_{j_r}} \sum_{i=1}^{n} \mathbf{A}_{(j_r,i)} =\end{aligned}$$

Replacing n_{j_r}, according to equation (1) we have that:

$$q + \frac{1-q}{n_{j_r}} \sum_{i=1}^{n} \mathbf{A}_{(j_r,i)} = q + \frac{1-q}{\sum_{i=1}^{n} \mathbf{A}_{(j_r,i)}} \sum_{i=1}^{n} \mathbf{A}_{(j_r,i)} = q + 1 - q = 1$$

Consequently, we have proven that the Google Matrix is stochastic, since $\sum_{i=1}^{n} \mathbf{G}_{(i,j_r)} = 1$ for a random (thus for all) column of it.

5.3 Validity Check: The power method algorithm

Based on the wording of the exercise, the described model represents the significance of the pages on the web. The way each page is ranked is presented as one of the greatest (in computational and data volume) eigenanalysis, which is made by Google, utilizing the so-called power method.

The basic idea behind the power method is that right-multiplying repeatedly a matrix with a random vector, results in moving the vector very close to the dominant eigenvector of **A**. That means executing the above steps a finite number of times we can get really close to the eigenvalue λ_{max} of the matrix that has the greatest magnitude of all the others. That is the eigenvalue that is associated with the dominant eigenvector, the one that the power method calculates.

A small note on this is the fact that the vectors calculated in each step of the power method algorithm tend to get out of hand after a while, featuring extremely high norms that is really difficult to deal with. In order to cope with this issue, the power method normalizes the vector calculated at each step, before looping again for a new calculated eigenvector.

Back to our problem, we are going to utilize the power method algorithm in order to check in practice, with the aid of a machine, a theoretically known claim: that **G** has maximum eigenvalue $\lambda_{max} = 1$. The following subsection explains the coding behind this procedure.

5.3.1 GNU Octave Code

Having presented the basic theoretical background of the power method algorithm, implementing it into code comes naturally as the next step of our reasoning. The following listing shows its implementation in GNU Octave programming language.

```
1  function [eigenvalue, eigenvector] = powerMethod (aMatrix,
2                                                   initialVector = rand(columns(aMatrix), 1),
3                                                   decimalPlacesAccuracy = 6)
4    [n, n] = size(aMatrix);
5    desiredAccuracy = 0.5 * 10 ^(-decimalPlacesAccuracy);
6  
7    if rows(aMatrix) != columns(aMatrix)
8      error("powerMethod: The input matrix is not square! I cannot put up with it!");
9    endif
10 
11   while initialVector == zeros(columns(aMatrix), 1)
12     initialVector = rand(n, 1);
13   endwhile
14 
15   u = initialVector;
16   lambda = inf;
17 
18   do
19     lambdaPrevious = lambda;
20     w = aMatrix * u;
21     lambda = normOf(w);
22     u = (1/lambda) * w;
23   until abs(lambda - lambdaPrevious) < desiredAccuracy
24 
25   eigenvalue = lambda;
26   eigenvector = u;
27 
28 endfunction
```

Listing 34: GNU Octave code for the power method

Apart from the initializations and the easily-understandable defensive programming at the beginning, the gist of the algorithm lies within lines 18 to 23. These lines represent the steps of the power method. The matrix is right-multiplied by the random vector, the norm of which is stored as the calculated eigenvalue. Afterwards, the eigenvector is normalized, in order to have norm equal to 1 and, as a consequence, be easily handled. The algorithm keeps looping for the eigenvalue and the eigenvector, until the desired accuracy is accomplished.

Using the above function, we can now write a mains script, in order to check that the maximum eigenvalue of $\mathbf{G}$ is $\lambda_{max} = 1$.

```
A = [0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0;
     0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0;
     0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0;
     0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0;
     1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0;
     0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0;
     0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0;
     0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0;
     0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0;
     0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0;
     0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1;
     0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0;
     0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0;
     0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 1;
     0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0];

n = rows(A);
q = 0.15;

% calculates the n_i for every row of the matrix A. The results will be used during
  construction of G
for row = 1:n
  sumOfRow(row) = sum(A(row,:));
endfor

% constructs Google Matrix G
for i = 1:n
  for j = 1:n
    G(i, j) = (q/n) + (A(j, i)*(1-q))/sumOfRow(j);
  endfor
endfor

% utilizes the power method function to calculate the maximum eigenvalue and the associated
  eigenvector
[eigenvalue, eigenvector] = powerMethod(G);

% normalizes the eigenvector with its sum, in order to represent probabilities that sum to 1
eigenvector /= sum(eigenvector);

% prints result in an easy-on-the eye format, nothing special
printf("The maximum eigenvalue of the matrix is %.3f. \n\n", eigenvalue);

for page = 1:length(eigenvector)
  printf("The probability that the user is now in page %2d is %f\n", page, eigenvector(page)
    );
endfor
```

Listing 35: GNU Octave main script the power method

Running the script we do find out that our theoretical claims were correct. The terminal informs us in detail that:

```
>> mainPowerMethod
The maximum eigenvalue of the matrix is 1.000.

The probability that the user is now in page  1 is 0.026825
The probability that the user is now in page  2 is 0.029861
The probability that the user is now in page  3 is 0.029861
The probability that the user is now in page  4 is 0.026825
The probability that the user is now in page  5 is 0.039587
The probability that the user is now in page  6 is 0.039587
The probability that the user is now in page  7 is 0.039587
The probability that the user is now in page  8 is 0.039587
The probability that the user is now in page  9 is 0.074564
The probability that the user is now in page 10 is 0.106320
The probability that the user is now in page 11 is 0.106320
The probability that the user is now in page 12 is 0.074564
The probability that the user is now in page 13 is 0.125091
The probability that the user is now in page 14 is 0.116328
The probability that the user is now in page 15 is 0.125091
```

That is exactly what we were expecting, based on the wording of the exercise. In total, sorting the pages on a descending order, we get the following result, as if a search engine would respond to a relative query:

```
Page 15 (0.125092)
Page 13 (0.125092)
Page 14 (0.116328)
Page 11 (0.106320)
Page 10 (0.106320)
Page 12 (0.074564)
Page  9 (0.074564)
Page  8 (0.039587)
Page  7 (0.039587)
Page  6 (0.039587)
Page  5 (0.039587)
Page  3 (0.029861)
Page  2 (0.029861)
Page  4 (0.026825)
Page  1 (0.026825)
```

5.4 Manipulating the adjacency matrix

Having disclosed the legitimate page rank for each one of the web pages, we are now asked to manipulate the adjacency matrix, in order to alternate the significance standing. Within the following lines we are going to analyze in detail the reasoning behind any specific manipulation movement, measuring in practice the impact on the page rankings.

To start with, we notice that node (web page) 10 has a high in-degree (5 links from other pages towards it). In natural language, this could possibly mean that page 10 is a high-status or prestigious website, which means that there are a lot of reference to it on other websites.

This ascertainment, combined with the observation that page 10 points to page 13, make it obvious that lots of significance is transferred from node 10 to node 13. Similarly, node (page) 15 gains a lot of significance, since a possible high-status node, node 11, points towards it. All these combined, make pages 13 and 15 leading the standing of pages.

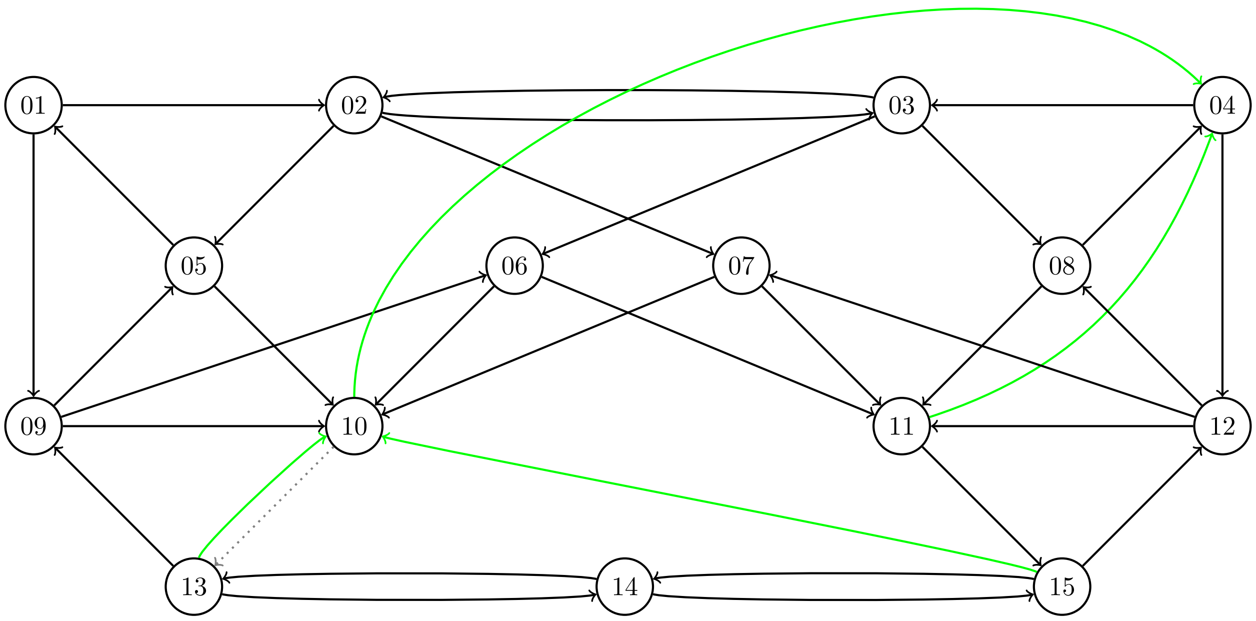
Focusing back to our goal, we want to manipulate the adjacency matrix, by removing one edge and adding four others, so as to make a specific page gain in significance. In order to make results clearer, we are going to chose one of the two last-significant pages, page number 4, and try to enhance its page rank. Taking into account the previous claims on high-status pages, it seems obvious that a good idea is to manipulate pages 10 and 11 (in relation with our goal-page 4), so as to achieve our aim.

Starting from the edge removal, we chose to cut off the linking between node 10 towards node 13. Based on our analysis, we expect this to have a negative impact on the significance of page 13, since it has lost one high-status feeder. Thinking in reverse reasoning, we are going to add an edge starting from node 10 to node 4. Completely respectively, we expect that such a movement will attach to node 4 a (rather unexpected) alliance with high benefits, concerning page rank.

We have three more edges in our quiver to add to the graph, aiming to enhance the significance of node 4. Continuing on the same way of thinking, we can make the other prestigious feeder, node 11, point towards the aim-page. Our second, freshly-introduced edge, thus, goes to 11$\rightarrow$ 4 linking.

Except for direct support to node 4, we can also manipulate the matrix in a favorable to node 4 way *indirectly*. In other words, we can support the two main feeders of node 4 (10 and 11), enhancing their in-degrees. Consequently, our last two additional edges go for the 15 $\rightarrow$ 10 and 13 $\rightarrow$ 10 linking.

After the manipulation, the graph of the network is shown in the following figure. The removed edge is presented as a faded, dotted arrow, while the added edges are drawn in green color.



Accordingly, the new adjacency matrix $\mathbf{A_{manipulated}}$ is the following:

$$\mathbf{A_{manipulated}} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \end{bmatrix}$$

We are now ready to run the script again and let the power method re-calculate the probability eigenvector. Indeed, we have accomplished our goal. Page 4 is has moved form the bottom to the very start of the page rankings, since its probability has been dramatically increased! Exactly as we had expected, page 13 has driven the opposite way towards the last places, concerning page significance.

```
Before Manipulation                   After Manipulation

 Page 15 (0.125092)                   Page  4 (0.174505)
 Page 13 (0.125092)                   Page 11 (0.116076)
 Page 14 (0.116328)                   Page 12 (0.103005)
 Page 11 (0.106320)                   Page 10 (0.102079)
 Page 10 (0.106320)                   Page  3 (0.097592)
 Page 12 (0.074564)                   Page  8 (0.066836)
 Page  9 (0.074564)                   Page 15 (0.066494)
 Page  8 (0.039587)                   Page  7 (0.052612)
 Page  7 (0.039587)                   Page  2 (0.047392)
 Page  6 (0.039587)                   Page  6 (0.044622)
 Page  5 (0.039587)                   Page 14 (0.033703)
 Page  3 (0.029861)                   Page  5 (0.030399)
 Page  2 (0.029861)                   Page  9 (0.024603)
 Page  4 (0.026825)                   Page  1 (0.022919)
 Page  1 (0.026825)                   Page 13 (0.017162)
```

5.5 Manipulating the teleport probability factor

Another important parameter in the page rank calculation is the so-called teleport probability q, with $0 < q < 1$. Utilizing previous theoretical background, we know that this factor represents the probability a web surfer to visit a random web page of the network, irrespective of the links and references contained in the currently located website.

The teleport probability factor is really crucial in calculating the rankings of the pages on the web. Up to that point, in all our previous approaches, we have taken for granted that $q = 0.15$. In other words, it is 15% likely that a surfer chooses to move to a random page, not affected by the links contained in the page they are now browsing.

In this subsection, we are going to alternate the teleport probability for the (above-presented) manipulated matrix.

5.5.1 Decreasing the teleport probability factor

Suppose that for some reason we want to refine our model that represents the page significance. In order to do so, we are going to decrease the teleport probability q to $q_{decreased} = 0.02$. In natural language, this means that a possible surfer on the network is 2% likely to move to a random page (not affected by the links of the current page) and 98 % likely that the new page to browse through will be one of the references on the current site.

Even from the natural language description, we can easily assume that this significant decrease in the teleport probability, is going to have a strong effect on the page ranks.

More accurately, it could be devastating for nodes (web pages) with low in-degree. The reason behind is the fact that this kind of pages (e.g. 1, 13) bet the vast majority of their probabilities to be chosen in the random (not affected by links) selection. They cannot boast of many other sites that point towards them, so random choice is their only hope to get a good rank. Limiting by that much the q factor, limits, as well, their hopes for a decent ranking.

On the contrary, a possible decrease in the q factor could have an enormous uplifting effect on sites that have high in-degree. This kind of sites (e.g. 4, 11) are really pleased to see the probability of link-driven choices to rise dramatically (from 85% to 98%). There are many sites that point towards them, so it is more than obvious that their actual traffic is expected to increase.

We already have the appropriate mathematical and programming tools to check the validity of our claims. We can just run the script one more time, defining that time $q = q_{decreased} = 0.02$. The comparative results are shown below:

```
          q = 0.15                              q = 0.02

     Page  4 (0.174505)                    Page  4 (0.192561)
     Page 11 (0.116076)                    Page 11 (0.130297)
     Page 12 (0.103005)                    Page 12 (0.119178)
     Page 10 (0.102079)                    Page  3 (0.109426)
     Page  3 (0.097592)                    Page 10 (0.091977)
     Page  8 (0.066836)                    Page  8 (0.076011)
     Page 15 (0.066494)                    Page 15 (0.071905)
     Page  7 (0.052612)                    Page  7 (0.054002)
     Page  2 (0.047392)                    Page  2 (0.042052)
     Page  6 (0.044622)                    Page  6 (0.039999)
     Page 14 (0.033703)                    Page 14 (0.027455)
     Page  5 (0.030399)                    Page  5 (0.017990)
     Page  9 (0.024603)                    Page  1 (0.010149)
     Page  1 (0.022919)                    Page  9 (0.008939)
     Page 13 (0.017162)                    Page 13 (0.008060)
```

Indeed, the machine verifies in practice our theoretical claims. In one phrase, decreasing the q factor deepened the gap between the pages on top and at the bottom of the ranking. The already high-ranked pages saw an increase in their page rank, while the low-ranked ones lost even more of their significance.

5.5.2 Increasing the teleport probability factor

Now suppose that for some reason we want to refine our model by increasing the teleport probability q to $q_{increased} = 0.6$. In natural language, this means that a possible surfer on the network is 60% likely to move to a random page (not affected by the links of the current page) and 40 % likely that the new page to browse through will be one of the references on the current site.

Even from the natural language description, we can easily assume that this significant increase in the teleport probability, is going to have a completely reverse effect, concerning what we have examined up to that point.

More accurately, such an alternation would be bad news for nodes with high in-degree. This type of pages rely a lot on references to them contained in many other sites. As a result, their main advantage loses a significant part of its power (from 85% to 40%). Examples of such pages are pages 4 and 11.

On the other hand, a possible increase in the q factor could have a positive effect on sites that have low in-degree. This kind of sites (e.g. 13, 1) are really pleased to see the probability of non-link-driven choices to rise dramatically (from 15% to 60%). Their lack of supporters (sites that point to them) is counteracted by this strong uplift in the random case calculation.

Exactly as we did before, we are going to execute again the main script and utilize the power method, in order to calculate the new page rank standings, this time for $q = q_{increased} = 0.6$. The comparative results are shown below:

```
          q = 0.15                              q = 0.6

     Page  4 (0.174505)                    Page  4 (0.110184)
     Page 11 (0.116076)                    Page 10 (0.101382)
     Page 12 (0.103005)                    Page 11 (0.089428)
     Page 10 (0.102079)                    Page 12 (0.070482)
     Page  3 (0.097592)                    Page  3 (0.069977)
     Page  8 (0.066836)                    Page 15 (0.063336)
     Page 15 (0.066494)                    Page  2 (0.059548)
     Page  7 (0.052612)                    Page  8 (0.058728)
     Page  2 (0.047392)                    Page  7 (0.057337)
     Page  6 (0.044622)                    Page  6 (0.056834)
```

```
Page 14 (0.033703)                      Page  9 (0.056278)
Page  5 (0.030399)                      Page  5 (0.055443)
Page  9 (0.024603)                      Page 14 (0.054505)
Page  1 (0.022919)                      Page  1 (0.051089)
Page 13 (0.017162)                      Page 13 (0.045450)
```

The program does validate our theoretical expectations. In one phrase, increasing the q factor bridged the gap between the pages on top and the ones at the bottom of the ranking. The already high-ranked pages saw a decrease in their page rank, while the low-ranked ones gained in significance, narrowing the gap between them.

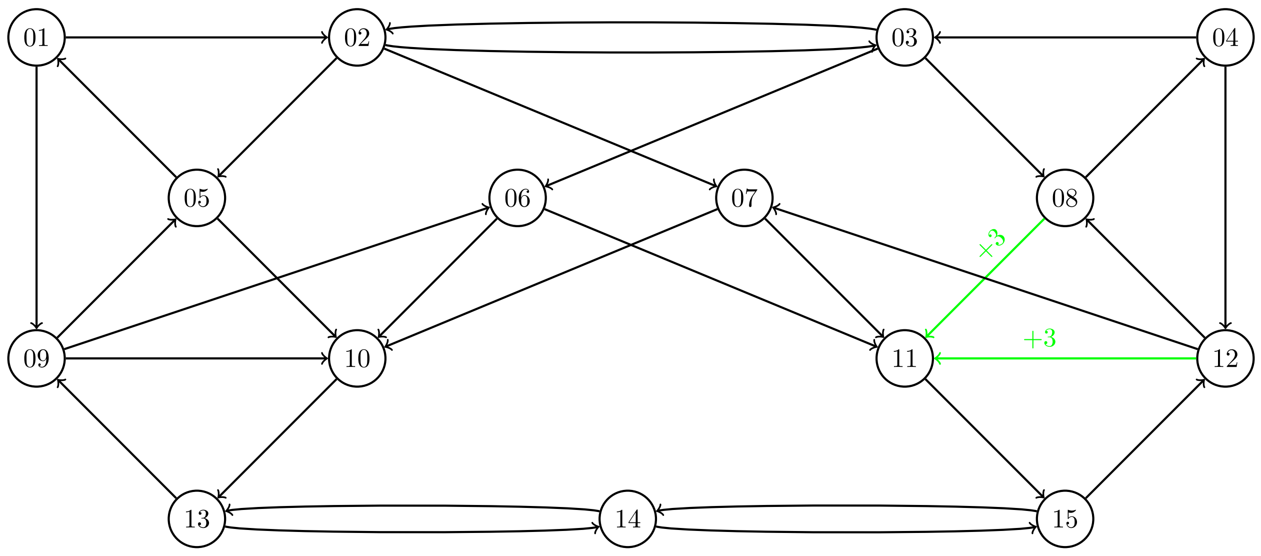
Overall, the teleport probability factor acts as a balancing intermediate mechanism between the high and the low in-degree nodes. We could claim that the factor plays the difficult role of modeling the way surfers browse through the web. Their behaviour is in close relation with the criteria and manner they resort to, when deciding to move from a page to another. Selection of an appropriate teleport probability that corresponds to the users' way of surfing can attribute enormous success to a web search engine. Otherwise, the respond to a relative query can diverge a lot from the expected (by the surfer) results, failing to model the actual significance of the pages.

5.6 Adding weights on the graph

Another really interesting question is what happens if we transform our directed graph to a ***weighed*** directed graph. In simple words, in this subsection we are going to investigate the effect of adding weighed edges on the page rankings.

The graph we are going to work on is the initial, legitimate graph that shows connection between the 15 pages of the exercise example. Based on the wording, the main goal is to enhance the page rank of page 11, in relation with the page rank of its competitor, page 10.

The exercise suggests that page 11 persuades pages 8 and 12 to present their references on page 11 in a more favorable (to page 11) way. In graph terminology, this indicates that we should change the entries $\mathbf{A}_{(8,11)}$ and $\mathbf{A}_{(12,11)}$ of the initial adjacency matrix from 1 to 3, meaning that from now on pages 8 and 12 point to page 11 in a more fetching than usual manner. The new graph is presented in the following figure. The weighed edges are drawn in green color.



Before running the script, we can make a few notes on what are the expected results of this strategy. One issue we have to take into consideration is the fact that by adding weights on edges starting from nodes 8 and 12, this subsequently increases the sum of rows 8 and 12 of the adjacency matrix. This sum is directly involved in the formula that calculates the page rank of a page. In specific, the sum n_i of row i of the adjacency matrix is involved as denominator in the formula that calculates the probability a surfer to be located in page i:

$$p_{\{i \to j\}} = \frac{q}{n} + (i - q) * \frac{\mathbf{A}_{(i,j)}}{n_i}$$

This ascertainment makes us suppose that a part of the uplifting effect the weighed edge imposes is counteracted by that division by a higher than original sum. Complementary, it is absolutely sure that all the other links starting from nodes 8 and 12 will have less positive effect on the pages they point to, due to this increase in the row sum.

A more sophisticated and scientifically correct approach on the issue would be to write the relations that express the page rank of page 11 before and after the addition of the weighed edges. Reaching a mathematical inequality between the $p_{11(prev)}$ and $p_{11(after)}$ would provide us enough scientific base, so as to know for sure whether this strategy is indeed helpful for the page rank of page 11 or not.

However, such mathematical handling diverges from the context of this assignment and, consequently, will not be analyzed within the lines of this paper. Thus, we are going to adopt a more naive and experimental, yet more imminent and obvious approach. This approach is already known from our previous handling, and involves utilizing the power method, in order to calculate the new probabilities, after adding the weights on the graph. The results are the following:

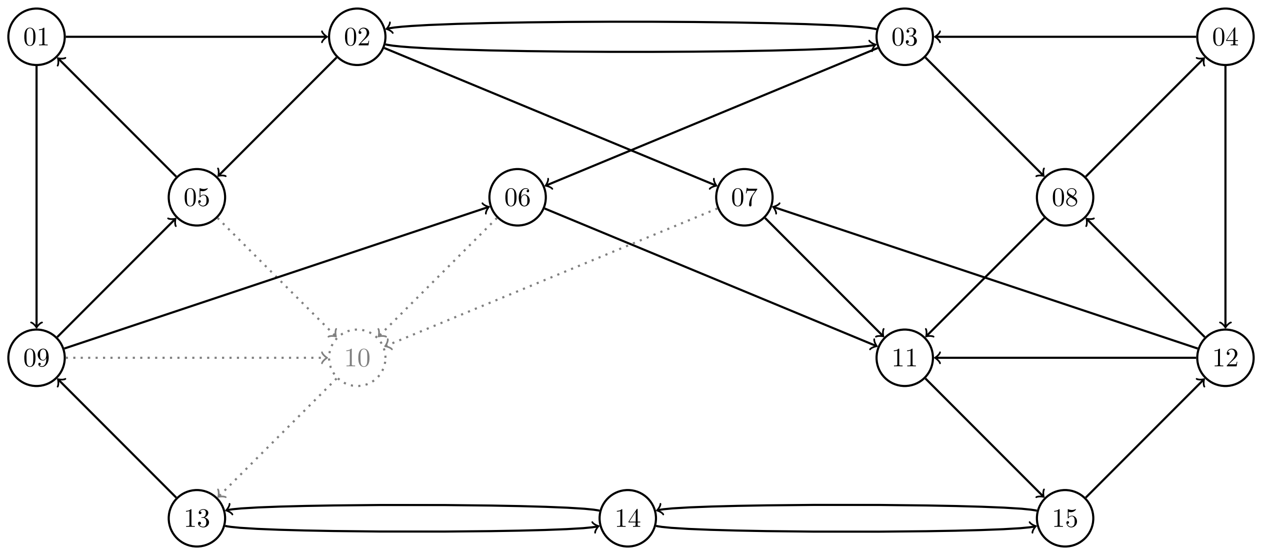
```
   before weights                   after weights

 Page 15 (0.125092)               Page 15 (0.141463)
 Page 13 (0.125092)               Page 11 (0.124008)
 Page 14 (0.116328)               Page 13 (0.123515)
 Page 11 (0.106320)               Page 14 (0.122616)
 Page 10 (0.106320)               Page 10 (0.102893)
 Page 12 (0.074564)               Page 12 (0.077099)
 Page  9 (0.074564)               Page  9 (0.073778)
 Page  8 (0.039587)               Page  5 (0.038942)
 Page  7 (0.039587)               Page  6 (0.037991)
 Page  6 (0.039587)               Page  7 (0.031145)
 Page  5 (0.039587)               Page  8 (0.030195)
 Page  3 (0.029861)               Page  2 (0.028372)
 Page  2 (0.029861)               Page  1 (0.026551)
 Page  4 (0.026825)               Page  3 (0.025016)
 Page  1 (0.026825)               Page  4 (0.016416)
```

Based on them, we can experimentally claim that adding weights has an uplifting effect on page 11. As we can see, the page has moved from the fourth to the second place, while its competitor (page 10) remained stable in position number five. In total, it seems that the uplifting effect of the weights overcompensated for the division with a higher denominator. The strategy, as a result, is deemed as successful, since the goal was achieved.

Nevertheless, we could not say the same for the nodes 4 and 7. These are the nodes that, although they remained intact, the addition of the weights affected them negatively. The reason is that the connections $8 \to 4$ and $12 \to 7$ lost part of their power, due to division with a greater denominator. It seems that nodes 4 and 7 were victims of negative side effects.

5.7 Removing a page from the network

The last issue we are going to examine is the entire removal of a node-page. According to the wording of the exercise, we are called to describe the impact of removing node 10 completely from the graph. Concerning the adjacency matrix, this means eliminating row and column 10 from the initial matrix $\mathbf{A}$. For a better visual comprehension of the situation, the new graph is depicted in the figure that follows. The erased node and edges are drawn in faded color.



Being curious enough, we can run the script one last time, in order to let the power method calculate the new page ranks. The results are rather interesting:

```
 before removal              after removal

Page 15 (0.125092)          Page 15 (0.186480)
Page 13 (0.125092)          Page 11 (0.170963)
Page 14 (0.116328)          Page 14 (0.107462)
Page 11 (0.106320)          Page 12 (0.103598)
Page 10 (0.106320)          Page  7 (0.051659)
Page 12 (0.074564)          Page  8 (0.050249)
Page  9 (0.074564)          Page  9 (0.048223)
Page  8 (0.039587)          Page  1 (0.047095)
Page  7 (0.039587)          Page  5 (0.042801)
Page  6 (0.039587)          Page  6 (0.041391)
Page  5 (0.039587)          Page 13 (0.041162)
Page  3 (0.029861)          Page  2 (0.040911)
Page  2 (0.029861)          Page  3 (0.035936)
Page  4 (0.026825)          Page  4 (0.032070)
Page  1 (0.026825)
```

To start with, we observe that page 15 not only remains on top of the rankings, but it has also increased its page rank. This is rather expected, if we take into consideration the fact that page 15 gains a lot of significance by the prestigious page 11. The last has also enhanced its rank, since its basic competitor, page 10, has been eliminated from the network. As a result, the removal of page 10 has a significant uplifting effect on pages 15 and 11, that stand at the top of the ranking.

In close relation with them, page 12 finds spaces to rise, in terms of page rank. Directly supported by 15 and indirectly by 11, it is more than expected to gain in significance.

The exact opposite way has driven node 13, moving from the top to the bottom of the standing. The main reason is the fact that page 13 has lost its main strong supporter, node 10. The removed node was playing a crucial role in feeding node 13 with significance; a role that it cannot play in its absence.

Talking about impact, the node that seems to be affected the least from the removal is node 14. With a slightly reduced probability, the node remains in the very first places. Such an outcome can be justified by the fact that node 14 was (indirectly) supported by both the main feeders (10 and 11). As a result, the elimination of page 10 did not have a larger negative impact on it, due to the connection it has with the crucial nodes 11 and 15.

Apart from the above, the rest of the pages on the graph are affected less by the removal. In general, we can split the remaining nodes in two groups.

The first group contains pages that have gained in significance, as a side effect of the removal, as well as the rise of other pages that point to them. A representative example of this group is page 7. The page has an increased rank, as an side effect of the rise of page 12, which contains link to page 7. Other pages in this group are pages 8, 2 and 1.

The second group contains the pages left and can be characterized for their loss of significance in an indirect way. Pages 3, 9 and 4 belong to this category.

In total, the removal of page 10 provoked re-arrangements in the graph and, as a consequence, in the final rankings. The down left part of the graph was, in general, negatively affected by the removal, since an important supporter was eliminated. As a matter of fact, this resulted in a loss of significance, concerning the nearby nodes.

On the contrary, the removal of page 10 had an uplifting effect on the down right part of the graph. Page 11 found spare space to rise, lifting upwards simultaneously many nearby nodes.

Finally, the nodes on the upper part of the graph feature medium-low significance. They were slightly and indirectly affected by the removal and its consequences. Nevertheless, the effect was not enough for them to step out from their medium-low category of significance.

References

[1] Timothy Sauer. *Numerical Analysis, Second Edition*. Pearson Education, 2012.

[2] Alfio Quarteroni, Fausto Saleri. *Scientific Computing with MATLAB and Octave, Second edition*. Springer-Verlag, Berlin, Heidelberg, 2006.

[3] Anastasios Tefas. *Lecture Notes on the course of Numerical Analysis*. School of Informatics AUTh, 2020.