Bachelor in Mathematics

Experimental Mathematics 4

Winter semester 2020/21

Solving polynomial equations

Authors:
Clara Popescu
Dylan Mota
Kim Da Cruz

Supervisors:
Prof. Dr. Gabor Wiese
Guendalina Palmirotta

Abstract

In the following, we will take a look at some methods to factorise polynomials. More precisely, we will study the Newton-Raphson method for solving polynomial equations over real numbers and for factorisation over finite fields, we will look at Berlekamp's algorithm. Moreover, we will examine Hensel's Lemma in order to solve polynomial equations modulo p^n, where $n \in \mathbb{N}$ and p is a prime number. Our goal is to become familiar with methods used for solving polynomial equations over real numbers and over finite fields.

Contents

1 Introduction

Factorisation of polynomials, or solving polynomial equations sounds easy when we are talking about real polynomials of small degrees. In these cases, there exist some well-known formulas that are easily applicable. Let us start with the easiest case, where the polynomial has degree 2.

1.1 Quadratic equations

First, let us recall that a quadratic equation is an equation that can be rearranged to the point that it is of the form:

$$ax^2 + bx + c = 0, \tag{1}$$

where $x \in \mathbb{R}$ is the unknown variable, a, b, $c \in \mathbb{R}$ are the known coefficients such that $a \neq 0$, otherwise we would have a linear equation.
In order to solve such an equation, one can use a few methods. One way is to first factorise it using the standard algebraic identities:

$$\begin{aligned} ab + ac &= a(b + c) \\ ac + ad + bc + bd &= (a + b)(c + d) \\ a^2 + 2ab + b^2 &= (a + b)^2 \\ a^2 - 2ab + b^2 &= (a - b)^2 \\ a^2 - b^2 &= (a + b)(a - b). \end{aligned}$$

After applying these to the polynomial in the equation, we use the "zero factor property" that states that if $ab = 0$, then $a = 0$ or $b = 0$.
Another method is the process of completing the square. Here, we try to use the algebraic identity

$$a^2 + 2ab + b^2 = (a + b)^2. \tag{2}$$

Consider the quadratic equation (1) from above, in order to use (2), we do the following calculations:

$$\begin{aligned} ax^2 + bx + c = 0 \qquad & \mid : a \\ x^2 + \frac{b}{a}x + \frac{c}{a} = 0 \qquad & \mid - \frac{c}{a} \\ x^2 + \frac{b}{a}x = -\frac{c}{a} \qquad & \mid + \left(\frac{b}{2a}\right)^2 \\ x^2 + \frac{b}{a}x + \left(\frac{b}{2a}\right)^2 = -\frac{c}{a} + \left(\frac{b}{2a}\right)^2 \qquad & \mid \text{use identity (2) on the lefthand side} \\ \left(x + \frac{b}{2a}\right)^2 = \frac{b^2 - 4ac}{4a^2}. & \end{aligned}$$

Finally, we solve the equation by taking the square root and subtracting $\frac{b}{2a}$ on both sides:

$$\begin{aligned}\left(x+\frac{b}{2a}\right)^2 &= \frac{b^2-4ac}{4a^2} && \mid \text{take the square root}\\ x+\frac{b}{2a} &= \pm\sqrt{\frac{b^2-4ac}{4a^2}} && \mid -\frac{b}{2a}\\ x &= \pm\frac{\sqrt{b^2-4ac}}{2a}-\frac{b}{2a}.\end{aligned}$$

Completing the square can be used to derive a general formula for solving quadratic equations, called the quadratic formula. Given the quadratic equation (1), the quadratic formula is given by:

$$x=\frac{-b\pm\sqrt{b^2-4ac}}{2a}=\frac{-b\pm\sqrt{\Delta}}{2a}.$$

where, the expression under the square root is called the discriminant and is denoted by the greek letter Δ. In fact, the discriminant determines the number of real roots the quadratic equation has:

- If $\Delta>0$, then the equation has 2 real roots given by:

$$x=\frac{-b+\sqrt{\Delta}}{2a} \quad \text{and} \quad x=\frac{-b-\sqrt{\Delta}}{2a},$$

- If $\Delta=0$, then the equation has one real root given by:

$$x=\frac{-b}{2a},$$

- If $\Delta<0$, then the equation has no real roots, since we would take the square root of a negative number in the quadratic formula. In this case, we have 2 complex roots.

We can conclude that there are a lot of methods for solving these equations. After having looked at equations of degree 2, we are going to look at methods for solving polynomial equations of degree 3.

1.2 Cubic equations

Let us recall that a cubic equation is an equation that it is of the form:

$$ax^3 + bx^2 + cx + d = 0, \tag{3}$$

where $x \in \mathbb{R}$ is again the unknown variable, a, b, c, $d \in \mathbb{R}$ are the known coefficients such that $a \neq 0$, otherwise we would speak of a quadratic equation.
As for the quadratic equations, there is more than one method for solving cubic equations, one of them being factoring out common factors, or factorisation using the standard algebraic identities:

$$\begin{aligned} a^3 + 3a^2b + 3ab^2 + b^3 &= (a+b)^3 \\ a^3 - 3a^2b + 3ab^2 - b^3 &= (a-b)^3 \\ a^3 - b^3 &= (a-b)(a^2+ab+b^2) \\ a^3 + b^3 &= (a+b)(a^2-ab+b^2). \end{aligned}$$

After applying these to the polynomial of the equation, we use again the "zero factor property" and if necessary, we have to solve a quadratic equation using one of the methods stated in the previous subsection.
In order to look at another method, let us recall the meaning of a depressed cubic. A depressed cubic is a polynomial of degree 3 of the form:

$$t^3 + pt + q,$$

where t is a real variable and p, q are real coefficients. This type of cubic polynomial is much simpler to study, however it's still fundamental since any cubic can be reduced to a depressed cubic by a change of variable. In fact, consider the general cubic equation (3). If we set $x := t - \frac{b}{3a}$ and calculate everything out, then we obtain:

$$at^3 + t\left(\frac{3ac - b^2}{3a}\right) + \frac{2b^3 - 9abc + 27a^2d}{27a^2}.$$

Thus, by dividing by a, we obtain the depressed cubic equation:

$$t^3 + pt + q = 0, \tag{4}$$

where $t = x + \frac{b}{3a}$, $p = \frac{3ac-b^2}{3a^2}$ and $q = \frac{2b^3-9abc+27a^2d}{27a^3}$.
The next method we are going to explain is Cardano's formula. This formula only solves depressed cubic equations, however, as stated above, any cubic polynomial can be reduced to a depressed cubic. Consider the depressed equation (4), then Cardano states that the solutions are given by:

$$t_k = u_k + v_k,$$

with $k \in \{0, 1, 2\}$ and

$$u_k = \xi^k \sqrt[3]{\frac{1}{2}\left(-q + \sqrt{\frac{-\Delta}{27}}\right)}, \qquad v_k = \xi^{-k} \sqrt[3]{\frac{1}{2}\left(-q - \sqrt{\frac{-\Delta}{27}}\right)},$$

where $\Delta = -(4p^3 + 27q^2)$ is the discriminant of the depressed cubic and $\xi = \frac{-1+\sqrt{-3}}{2}$. There are 3 possible cases:

- If $\Delta > 0$, then there are three real solutions.
- If $\Delta = 0$, then there are either two real roots, such that one of them has multiplicity 2 and the other root has multiplicity 1, or there is one real solution of multiplicity 3.
- If $\Delta < 0$, then there is one real solution and 2 complex solutions that are complex conjugates.

For more information, we refer to [3].
From Cardano's formula, we can deduce a general cubic formula. Consider the equation (3) and set:

$$\begin{aligned} \Delta_0 &= b^2 - 3ac \\ \Delta_1 &= 2b^3 - 9abc + 27a^2d \\ C &= \sqrt[3]{\frac{\Delta_1 \pm \sqrt{\Delta_1^2 - 4\Delta_0^3}}{2}}. \end{aligned}$$

Then one of the roots of the equation is given by:

$$x = -\frac{1}{3a}\left(b + C + \frac{\Delta_0}{C}\right).$$

The other two roots can be found, either by changing the choice of the cube root or by multiplying C by a primitive cube root of unity, that is $\frac{-1\pm\sqrt{-3}}{2}$. So the three roots are given by:

$$x_k = -\frac{1}{3a}\left(b + \xi^k C + \frac{\Delta_0}{\xi^k C}\right),$$

where $k \in \{0, 1, 2\}$ and $\xi = \frac{-1+\sqrt{-3}}{2}$.

We have already seen that polynomial equations of degree 3 are more difficult to solve than polynomial equations of degree 2. Next, we are going to study a method for solving polynomial equations of higher degrees. This method does not only work for polynomial equations, but also for non-linear equations.

2 Solving polynomial equations over real numbers using Newton's approximation

In this section, we are going to explain one of the fundamental problems in numerical analysis, which is solving non-linear equations. Consider, for example, the following equation:

$$e^{-x} = x, \quad x \in \mathbb{R}.$$

It can be represented graphically:

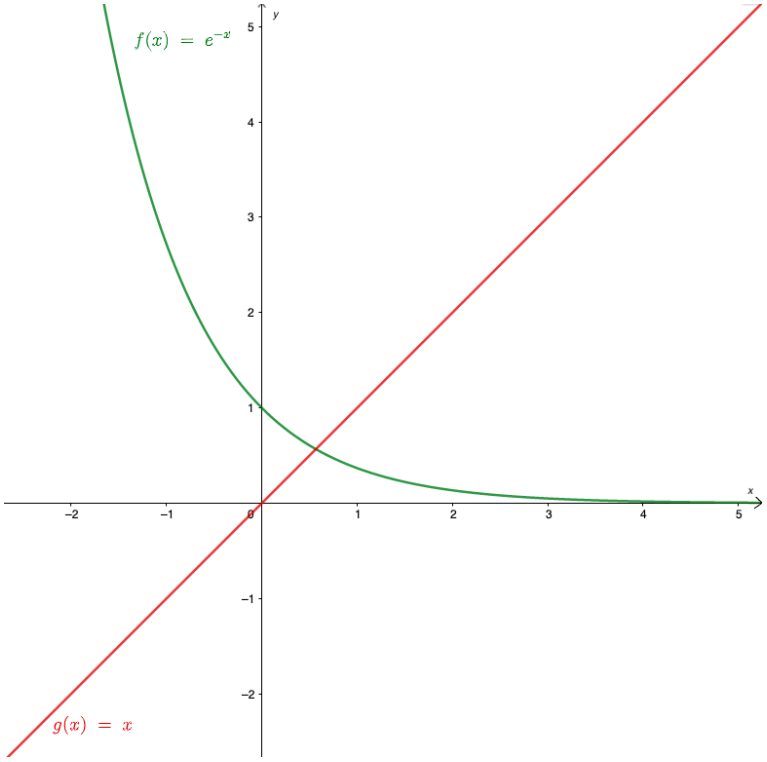


Figure 1: Graphic representation of $e^{-x} = x$, for $x \in \mathbb{R}$.

We can observe that the curves of $y = e^{-x}$ (in green) and $y = x$ (in red) only intersect in one point in the interval $]0, 1[$, thus the equation $e^{-x} = x$ has a unique solution in $]0, 1[$. However, it is nearly impossible to find this solution analytically. To solve equations of this type, where no solution can be found analytically, one often uses iterative methods. That is, we start with an initial "guess" of the solution and correct it using an algorithm that hopefully makes the initial "guess" converge to the "correct solution".

One of these methods is the so-called Newton-Raphson method. The idea behind the Newton-Raphson method is to approximate the root of a function that is continuous and differentiable by tangents. It is an effective algorithm that helps to find numerically a precise approximation for the root of a real-valued function.

2.1 Newton-Raphson method

The Newton-Raphson method is named after Isaac Newton and Joseph Raphson and as already said, it is an algorithm used to find roots of given functions. The algorithm produces successively better approximations of the root by considering the intersections of the tangents of the curve and the x-axis.

2.1.1 Explanation of the method

Consider a real-valued function f and assume that

- f is continuous and differentiable on $\mathbb{R}$, i.e. $f \in C_1(\mathbb{R})$,
- $f'(x_m) \neq 0$, where $x_m \in \mathbb{R}$ is an approximation of a root of f for $m \in \mathbb{N}$,
- x_0 is an initial guess for the root of f.

We want to find an iterative formula that allows us to find an approximation of the root of f by replacing the initial guess x_0 by x_1, which is the x coordinate of the intersection of the tangent of f at x_0 and the x-axis, and continuing this procedure until a sufficiently precise value of the root is reached. In fact, for some x_m, x_{m+1} is obtained by the intersection of the x-axis with the tangent line to the graph of f at x_m.
In the following figure, we explain graphically, the iterative procedure for the function $f(x) = e^{-x} - x$, represented by the black curve and the iterative tangents coloured in blue. In this situation, we impose as initial guess $x_0 = 3$. After some iterations, we can observe in the picture on the right hand side, that the tangent gets closer and closer to the "exact" root of f.

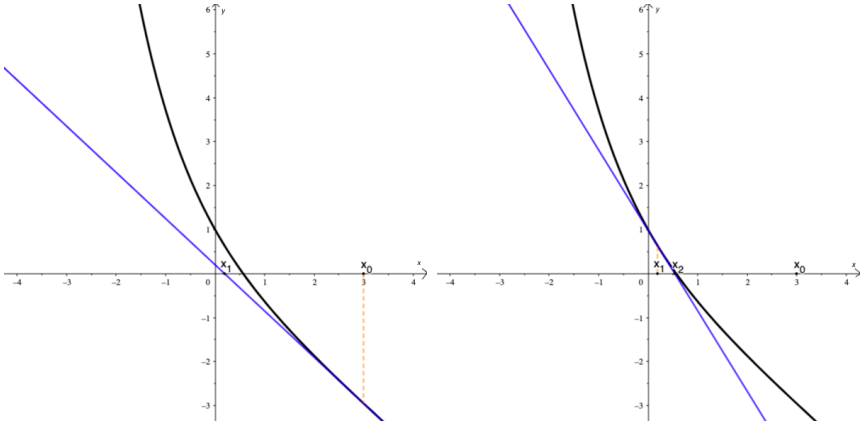


Figure 2: Illustration of the Newton-Raphson method for $f(x) = e^{-x} - x$.

Mathematically, let

$$y - f(x_m) = f'(x_m)(x - x_m), \quad \text{for } x, y \in \mathbb{R},$$

be the tangent of f at x_m and let $(x,\, y)$ be the point of intersection of the tangent of f at x_m and the x-axis. Then, $(x,\, y)$ is on the tangent of f at x_m and on the x-axis if and only if

$$\begin{cases} y - f(x_m) &= f'(x_m)(x - x_m) \\ y &= 0. \end{cases}$$

We get:

$$x = x_m - \frac{f(x_m)}{f'(x_m)}, \quad \text{for } m \in \mathbb{N}.$$

If x_{m+1} is the x coordinate of the intersection of the x-axis with the tangent line to the graph of f at x_m, then we obtain the iterative formula

$$x_{m+1} = x_m - \frac{f(x_m)}{f'(x_m)}, \quad \text{for } m \in \mathbb{N}. \tag{5}$$

2.1.2 Problems of the Newton-Raphson method

When everything goes well, the values of x_m approach a zero of the function f very quickly. In a lot of cases, the number of correct digits doubles with each iteration. However, there are some cases, for which the method does not work. Let us consider the following situations:

- **Horizontal tangent:** Consider for example $f(x) = \cos(x)$ with an initial value $x_0 = 0$. The Newton-Raphson method fails, because, numerically for

the initial value $x_0 = 0$, we get that $f'(x_0) = 0$. Thus, when calculating the value of x_1, we divide by 0 which is mathematically impossible. Hence, x_1 is undefined. Geometrically, if $f'(x_m) = 0$ this means that the tangent line to the graph of f at x_m has a slope equal to 0. Thus, the tangent line is horizontal and does not intersect the x-axis at any point.

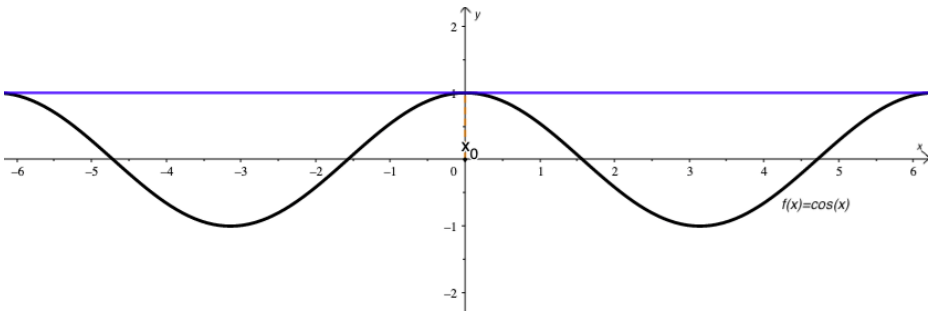


Figure 3: Horizontal tangent in blue.

- **Repeating loop:** Consider the function $f(x) = x^3 - 2x + 2$ with an initial value $x_0 = 0$. The Newton-Raphson method fails, because, numerically the first iteration gives $x_1 = 1$, and the second iteration yields $x_2 = 0$ which is equal to the value of x_0. Thus the sequence will alternate between these two values without converging to the root of f. Geometrically, the tangent at x_0 intersects the x-axis at x_1, and the tangent at x_1 intersects the x-axis at x_2 which is equal to x_0. This induces a repeating loop for the values of x_m.

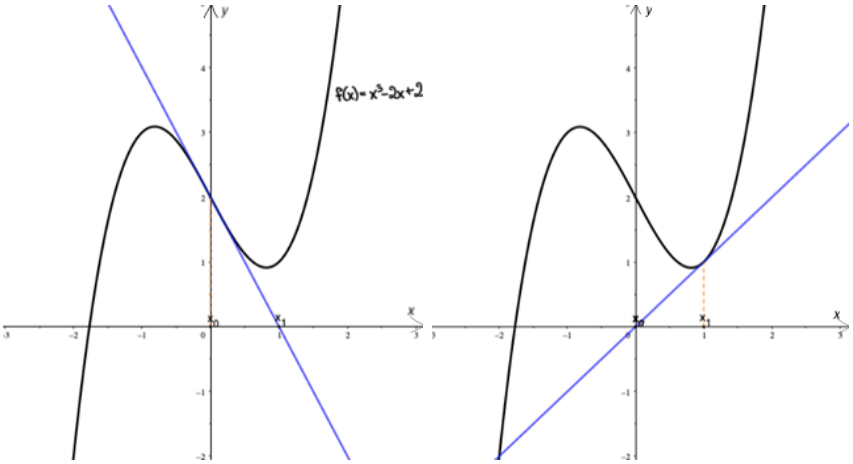


Figure 4: Repeating loop.

- **Diverging sequence:** Consider the function $f(x) = \sqrt[3]{x}$ with an initial value $x_0 = 1$, then the Newton-Raphson method fails. Note that f is continuous and infinitely differentiable, except for $x = 0$, where its derivative is undefined. Numerically, by looking at the iteration formula, we obtain

$$x_{m+1} = x_m - \frac{f(x_m)}{f'(x_m)} = x_m - \frac{\sqrt[3]{x_m}}{\frac{1}{3\sqrt[3]{x_m^2}}} = x_m - 3x_m = -2x_m.$$

  Thus, the distances from the solution double after each iteration, which shows that the values of x_m shift farther away from the zero of the function f. In fact, they diverge to infinity. Geometrically, after each iteration, the intersections of the tangents with the x-axis move farther away from the root of f.

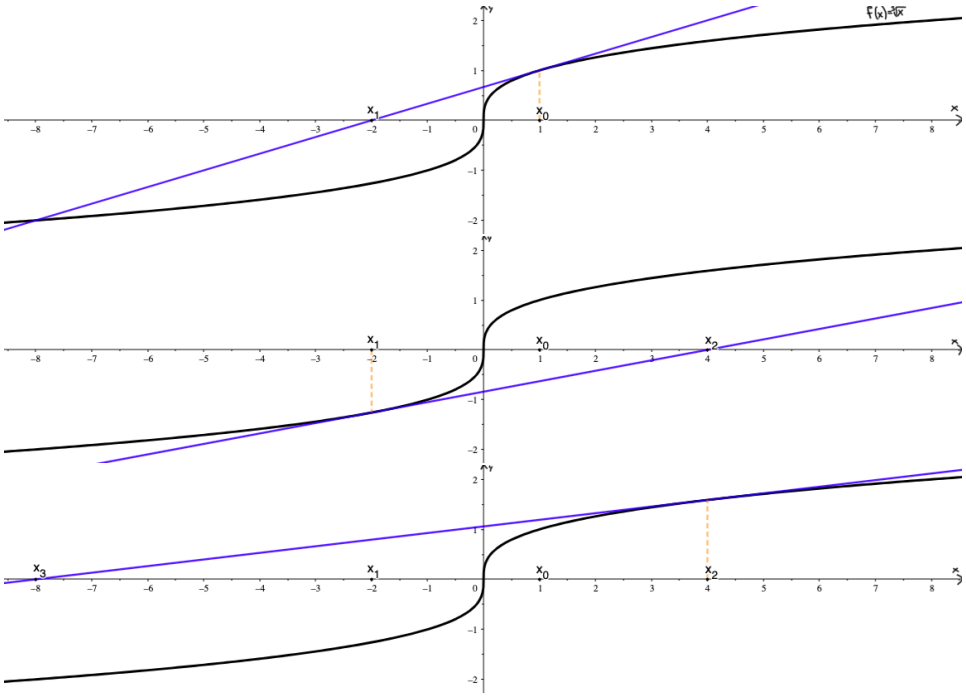


Figure 5: Diverging sequence.

  In conclusion, for every function $f(x) = |x|^\alpha$, where $0 < \alpha < \frac{1}{2}$, the values of x_m diverge to infinity.

2.1.3 Geometric illustration

In order to get a better feeling on how the Newton-Raphson method can be programmed, let us first illustrate it using GeoGebra. Under this link, https://www.

geogebra.org/m/cgdx6tzn, we created a GeoGebra program which allows us to show graphically how the Newton-Raphson method works.

What happens in this program?
The program requires, at the beginning, a real-valued function, a random initial value, two real numbers a, b for the interval $[a,\ b]$, in which we want to find the zero of the function, and the maximum number of iterations. Note that the maximum number of iterations is set to 15 since it is really time consuming writing out every single iteration formula. The program stops when the maximum number of iterations is achieved. Let us illustrate this with an example.

Example 2.1. Choose the real-valued function $f(x) = x^3 + 3$, with initial guess $x_0 = -2$ and $[a,\ b] = [-10,\ 10]$ as the interval. After pressing the button *Start*, the program draws the tangent of the function at x_0 and denotes x_1 the point where this tangent intersects the x-axis. We obtain for the first iteration $x_1 \approx -1,5$. Then, the program is going to draw the tangent to f at x_1 and after some iterations, we obtain the correct approximation -1.44224957030741 with a 15 digits accuracy.

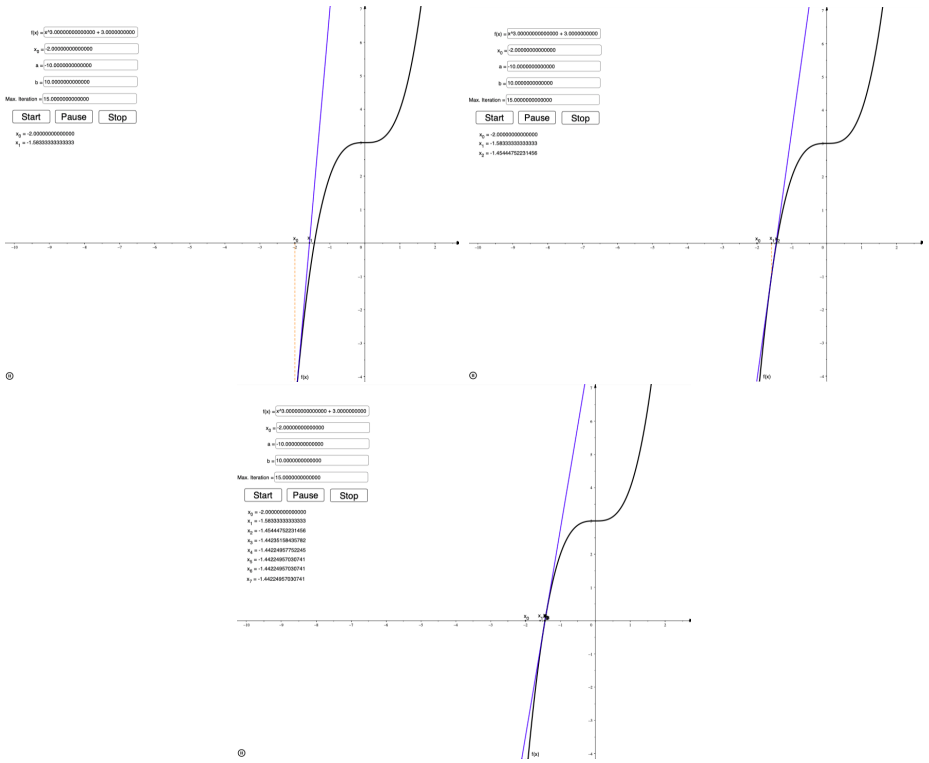


Figure 6: Animation of the Newton-Raphson method for $f(x) = x^3 + 3$.

2.1.4 Algorithmic illustration

We are now ready to implement the Newton-Raphson method in Python. The Python code is the following:

```
import sympy
from sympy.parsing.sympy_parser import parse_expr
from sympy import *

x = symbols("x")
print("\nInput:\n")

f = input("Enter a function: ")
x0 = float(input("Enter an initial value: "))
t = float(input("Enter a desired tolerance of the root: "))
it = int(input("Enter the total number of iterations: "))

expr = parse_expr(f)
firstDeriv = expr.diff(x)

def Func(y):
    return expr.subs(x,y)

def FirstDerivFunc(y):
    return firstDeriv.subs(x,y)

def Newton(f, x0, t, it):
    print("Let f(x) = ", f, ", then f'(x) = ", diff(f), ".\n",sep =
    "")
    x = x0
    fx = Func(x)
    dfx = FirstDerivFunc(x)

    if t < 1e-15:
        print("The desired tolerance is to high for Python. Choose
   in between 1e-1 and 1e-15.")

    elif dfx == 0:
        print("Since f'(x0) = f'(", x, ") = 0.0, we can't use the
   Newton-Raphson method.\n", sep = "")

    else:
        print("The initial value is given by x0 = ", x, ", and
   applying the Newton-Raphson method, we obtain\n", sep = "")
        for n in range(it):
            y = x
            x = x - fx/dfx
            fx = Func(x)
            dfx = FirstDerivFunc(x)
```

```
41
42              if dfx == 0:
43                  print("\nSince f'(x", n + 1, ") = f'(", x, ") =
    0.0, we can't use the Newton-Raphson method.\n", sep = "")
44                  return None
45
46              elif dfx.is_real == False:
47                  print("The derivative evaluated at x", n + 1, " = "
    , x ," is complex. Thus, we can't use the Newton-Raphson method
    .\n", sep = "")
48                  return None
49
50              elif x0 == x:
51                  print("\nSince x", n + 1, " = ", x, " = x0, we see
    that we are stuck in a loop, and thus, we can't use the Newton-
    Raphson method.\n", sep = "")
52                  return None
53
54              elif abs(x - y) > t:
55                  print("x", n + 1, " = ", x, sep = "")
56
57              elif abs(x - y) <= t:
58                  print("\nWe found a solution after ", n, "
    iterations, given by ", y, ".\n", sep = "")
59                  return None
60
61          print("\nWe have surpassed the maximum number of iterations
    .\n")
62
63
64 print("\nOutput:\n")
65
66 Newton(f, x0, t, it)
```

Listing 1: Python code for the Newton-Raphson method.

Remark 2.1. Before running the Python code, it is useful to know that

a) for plugging in the function f,

- use the symbol * for multiplication, e.g. $3\cos(x)$ in Python language is $3*\cos(x)$
- for exponents, use the symbol **, e.g. x^2 will be typed as $x**2$.

b) The desired tolerance is under the form $1e-n$, where the integer $n \in [1, 15]$. Thus, Python only gives 15 significant digits of the desired root.

Explanation of the Python code.
When running the Python code, the first step is to correctly plug in the function f along with an initial value, a desired tolerance of the correct root and the wanted maximal number of iterations. From here on, the program transforms the plugged in function, which is of type *String*, into an expression in order to define the two functions **Func(y)** and **FirstDerivFunc(y)**. These two functions just evaluate the plugged in function f at y, respectively the first derivative of the function at y.
Then, the function **Newton(f, x0, t, it)**, which takes the given function f, the initial value $x0$, the tolerance t, and the maximum number of iterations it as the only variables, first defines the iterative values by x, starting with $x = x0$, and fx to be the value of the function f evaluated at x, respectively dfx to be the first derivative of f evaluated at x. Now consider the following three cases.
The first case is if the tolerance t is to high for Python, it prints out that one has to choose a tolerance in between $1e-1$ and $1e-15$.
The second case checks if the first derivative evaluated at x is equal to zero, since the iteration formula of the Newton-Raphson method only works if it is not equal to zero. Hence, it prints out that the derivative is zero, and the Newton-Raphson method is not applicable.
Lastly, if the derivative is not equal to zero, a for loop is introduced which goes over every n from 0 up to the maximum number of iterations minus one. Then in this loop, it defines y to be equal to x, and replaces x using the iteration formula. It then proceeds to replace fx with the value of the function evaluated at this new x, respectively dfx with the first derivative evaluated at x, and considers the following five cases.
In the first case, it again has to check if the derivative at x is equal to 0. Thus, it prints out again that the Newton-Raphson method is not applicable, and returns *None* in order to end the function at this point. Note that this return is essential, because otherwise it would go back to the for loop.
In the second case, the program checks if the value of the derivative is real or not. Since we only consider real functions and real variables, in this case the program is stopped.
The third case is if the initial value $x0$ is equal to this new x. Hence, it prints out that one is stuck in a loop, and again the Newton-Raphson method is not applicable, and returns *None*.
In the fourth case, it checks if the absolute value of the difference of x and y is greater than the tolerance t. If it is the case, it prints out the value of x, and defines this value to be equal to the $n+1^{th}$ iteration. Note that since the for loop starts at 0, it has to add 1 as $x0$ is already defined in the beginning. After this, it goes back to the for loop.
Lastly, if the absolute value of the difference is smaller or equal to the tolerance t,

i.e. the desired root with the given tolerance is found, it prints out the number of iterations needed for finding this root, along with the value of this root. As before, in order to end the function at this point, it returns *None*.
If the for loop has gone over every n, i.e. it does not have found a solution, it prints out that the maximum number of iterations has been surpassed.

Example 2.2. Choose the same function as in Example 2.1, $f(x) = x^3 + 3$, along with an initial value $x_0 = -2$ and a tolerance of $1e - 10$. Moreover, set the total number of iterations to 100. After running the Python code, we obtain the following:

```
Input:

Enter a function: x**3 + 3
Enter an initial value: -2
Enter a desired tolerance of the root: 1e-10
Enter the total number of iterations: 100

Output:

Let f(x) = x**3 + 3, then f'(x) = 3*x**2.

The initial value is given by x0 = -2.0, and applying the Newton-Raphson method, we obtain

x1 = -1.58333333333333
x2 = -1.45444752231456
x3 = -1.44235158435782
x4 = -1.44224957752245
x5 = -1.44224957030741

We found a solution after 5 iterations, given by -1.44224957030741.
```

Figure 7: Input and output of Example 2.2.

Thus after 5 iterations, we obtain the correct root of the function f.

nbr. of iterations	x_m	nbr. of correct digits
0	-2.00000000000000	0
1	-1.58333333333333	1
2	-1.45444752231456	2
3	-1.44235158435782	4
4	-1.44224957752245	9
5	-1.44224957030741	15

Remark 2.2. We observe from the above table, that after each iteration, the number of correct digits is roughly doubled. Intuitively, we can assume that the Newton-Raphson method has a quadratic convergence.

Example 2.3. Choose the same function as above, $f(x) = x^3 + 3$, along with the same initial value $x_0 = -2$, but this time we want a better tolerance, we want a tolerance of $1e - 20$. Moreover, set the total number of iterations to 100. The tolerance we wish to have is to good for python, thus after running the Python code, we obtain the following:

```
Input:

Enter a function: x**3 + 3
Enter an initial value: -2
Enter a desired tolerance of the root: 1e-20
Enter the total number of iterations: 100

Output:

Let f(x) = x**3 + 3, then f'(x) = 3*x**2.

The desired tolerance is to high for Python. Choose in between 1e-1 and 1e-15.
```

Figure 8: Input and output of Example 2.3.

Example 2.4. Consider the function, $f(x) = x^4 + 3x^2 + 2$, along with an initial value $x_0 = 0$, and a tolerance of $1e-15$. Moreover, set the total number of iterations to 100. After calculating the derivative of f evaluated in x_0 we find that $f'(x_0) = 0$, which means that we have a horizontal tangent. We see that we cannot use Newton-Raphson's method, thus after running the Python code, we obtain the following:

```
Input:

Enter a function: x**4 + 3*x**2 + 2
Enter an initial value: 0
Enter a desired tolerance of the root: 1e-15
Enter the total number of iterations: 100

Output:

Let f(x) = x**4 + 3*x**2 + 2, then f'(x) = 4*x**3 + 6*x.

Since f'(x0) = f'(0.0) = 0.0, we can't use the Newton-Raphson method.
```

Figure 9: Input and output of Example 2.4.

Example 2.5. Let $f(x) = x^3 - 3x + 2$ be a function, and choose an initial value $x_0 = 0$, and a tolerance of $1e - 15$. Moreover, set the total number of iterations to 20. As seen in the section "Problems of the Newton-Raphson method", this is an example of a repeating loop an in this case, the Python code returns the following:

```
Input:

Enter a function: x**3 - 2*x + 2
Enter an initial value: 0
Enter a desired tolerance of the root: 1e-15
Enter the total number of iterations: 20

Output:

Let f(x) = x**3 - 2*x + 2, then f'(x) = 3*x**2 - 2.

The initial value is given by x0 = 0.0, and applying the Newton-Raphson method, we obtain

x1 = 1.00000000000000

Since x2 = 0 = x0, we see that we are stuck in a loop, and thus, we can't use the Newton-Raphson method.
```

Figure 10: Input and output of Example 2.5.

2.2 *Revisited* Newton-Raphson method with faster convergence

Newton-Raphson's method, with quadratic convergence, is useful, because if the initial guess is close to the root of our function, we don't need a lot of iterations, since they will quickly converge to the zero. However, there exist other variants of iteration methods that converge even faster to the root. In this section, we are going to look at a method, that has a speed of convergence of order 3 and where the tangent line is being replaced by a curve, which will be closer to the given function. This curve will be an arc of parabola. More information can be found in the following paper [6].

2.2.1 Explanation of the method

Let

- $I \in \mathbb{R}$ be an interval,
- $f \in C_2(I)$ be a function and denote $a \in \mathbb{R}$ to be such that $f(a) = 0$,
- x_0 be an initial guess for the root of f.

Again, we want to find an iterative formula that allows us to find an approximation of the root of f. In the Newton-Raphson method, we found it by approaching f with the tangent of f at $x_m \in \mathbb{R}$, but for a faster convergence, we approach f by an osculating parabola curve, that has the same first and second derivatives as $f(x)$ in a neighbourhood of $x_m \in \mathbb{R}$ for $m \in \mathbb{N}$. In fact, for some $m \in \mathbb{N}$, x_{m+1} is obtained by the intersection of the x-axis with the parabola curve that has the same first and

second derivatives as $f(x)$ in a neighbourhood of x_m.

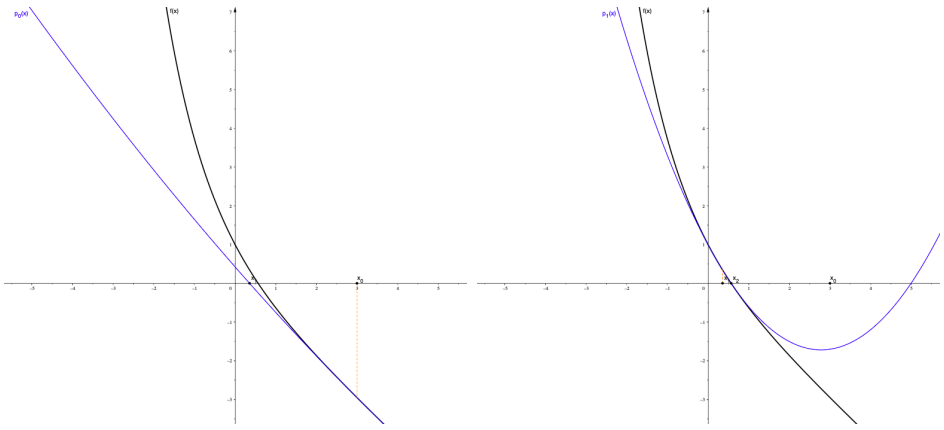


Figure 11: Illustration of the *revisited* Newton-Raphson method for $f(x) = e^{-x} - x$.

First of all let us find explicitly the equation of the parabola.
Let us recall the Taylor series of a real-valued (or complex-valued) function f that is infinitely differentiable at $a \in \mathbb{R}$. It is given by

$$\sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n, \quad x \in \mathbb{R} \text{ (or } \mathbb{C}),$$

where $f^{(n)}$ denotes the n^{th} derivative of f (by convention, $f^{(0)}(x) = f(x)$).
Since f is of class C_2, we obtain

$$f(x_m) + f'(x_m) \cdot (x - x_m) + \frac{1}{2!} \cdot f''(x_m) \cdot (x - x_m)^2.$$

so the approximating quadratic polynomial $p_m(x)$ is:

$$p_m(x) = f(x_m) + f'(x_m) \cdot (x - x_m) + \frac{1}{2} \cdot f''(x_m) \cdot (x - x_m)^2.$$

Set

$$a_m := \frac{1}{2} f''(x_m) \neq 0, \quad b_m := f'(x_m) \neq 0, \text{ and } c_m := f(x_m), \text{ then}$$

the equation of the parabola is

$$p_m(x) = c_m + b_m \cdot (x - x_m) + a_m \cdot (x - x_m)^2.$$

Now, we want to find the intersections of this parabola with the x-axis, which are the solutions of $p_m(x) = 0$. We see that $p_m(x) = 0$ is a quadratic equation for the variable $(x - x_m)$, so let us calculate the discriminant:

$$\Delta_m = b_m^2 - 4a_m c_m.$$

Knowing this, we have two possible cases:

- $\Delta_m < 0$: In this case, $p_m(x) = 0$ has no real solution for $(x - x_m)$. Since there is no real solution for $(x - x_m)$, there is also no real solution for x satisfying $p_m(x) = 0$, so the parabola doesn't intersect the x-axis which implies that no iterative formula can be found. Thus, we can't use this method to find an approximation of the root of f.

- $\Delta_m \geq 0$: In this case, $p_m(x) = 0$ has two real solutions for $(x - x_m)$ given by $(x - x_m) = \frac{-b_m + \sqrt{b_m^2 - 4a_m c_m}}{2a_m}$ and $(x - x_m) = \frac{-b_m - \sqrt{b_m^2 - 4a_m c_m}}{2a_m}$.
  Since we want x_{m+1} to be the intersection of the parabola with the x-axis, we want that $p(x_{m+1}) = 0$, thus

$$x_{m+1} - x_m = \frac{-b_m \pm \sqrt{b_m^2 - 4a_m c_m}}{2a_m} = -\frac{b_m}{2a_m} \pm \frac{\sqrt{b_m^2 - 4a_m c_m}}{2a_m}.$$

  - If $b_m < 0$: $-\frac{b_m}{2a_m}\left(1 \pm \sqrt{\frac{b_m^2 - 4a_m c_m}{b_m^2}}\right)$,
  - If $b_m > 0$: $-\frac{b_m}{2a_m}\left(1 \mp \sqrt{\frac{b_m^2 - 4a_m c_m}{b_m^2}}\right)$.

  Therefore, x_{m+1} is given by:

$$x_{m+1} = x_m - \frac{b_m}{2a_m}\left(1 \pm \sqrt{1 - \frac{4a_m c_m}{b_m^2}}\right).$$

The problem now is that we found 2 values for x_{m+1}, however we only want 1 value because else, the algorithm would not work and would also be very complicated since the number of x_m's would double after each iteration.

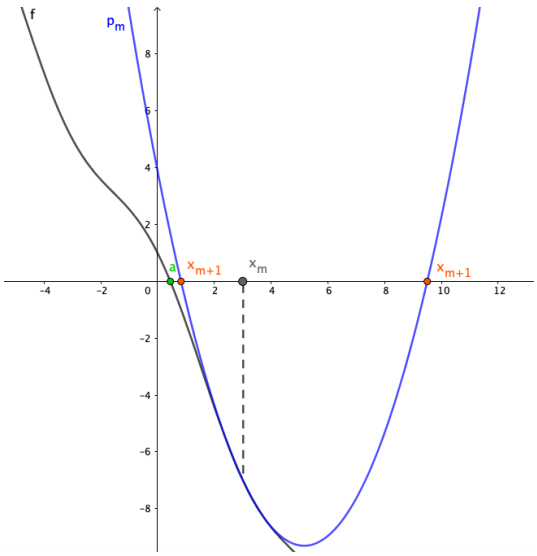


Figure 12: The problem is that the parabola (in blue) intersects the x-axis twice.

This means that we have to chose between the two solutions. Since we want to approach the root of f, we choose the solution for x_{m+1} that is closer to the zero. We want this to be true for every x_m, thus, the solution we choose for x_{m+1} must also be the one that is closer to x_m. Looking at $|x_{m+1} - x_m|$, we see that the solution with the negative sign is closer to the root of f, therefore

$$x_{m+1} = x_m - \frac{b_m}{2a_m}\left(1 - \sqrt{1 - \frac{4a_m c_m}{b_m^2}}\right), \tag{6}$$

which is the iterative formula for the *revisited* Newton-Raphson method.

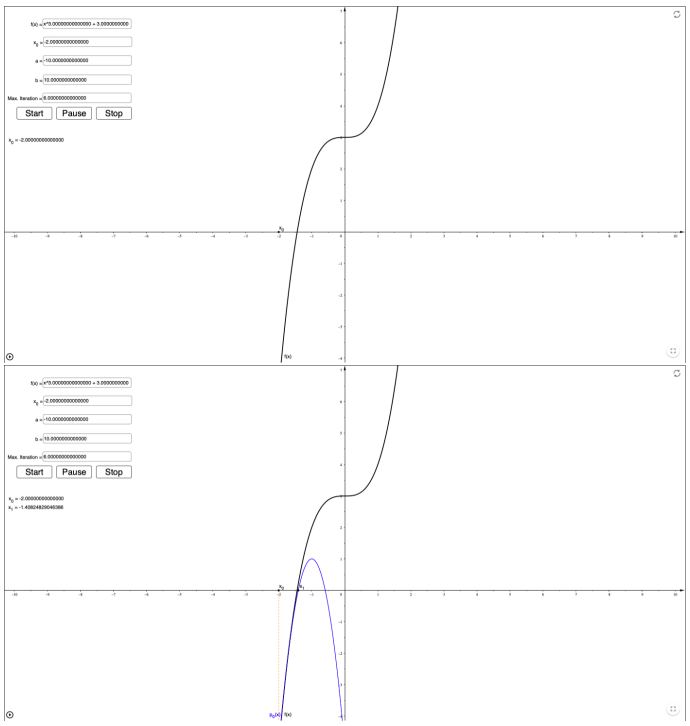
2.2.2 Geometric illustration

In order to get a better feeling on how the *revisited* Newton-Raphson method can be programmed, let us first illustrate it using GeoGebra. Under this link, https://www.geogebra.org/m/f4edtdfg, we created a GeoGebra program which allows us to show graphically how this Newton-Raphson method works.

What happens in this program?
The program requires, at the beginning, a real-valued function, a random initial value, two integers a, b for the interval $[a, b]$, in which we want to find the zero of the function; and the maximum number of iterations. Note that the maximum number of iterations is set to 6 since it is really time consuming writing out every

single iteration formula. The program stops when the maximum number of iterations is achieved. Let us illustrate it with an example.

Example 2.6. Choose the real-valued function $f(x) = x^3 + 3$, with initial guess $x_0 = -2$ and $[a, b] = [-10, 10]$ as the interval. After pressing the button *Start*, the program draws the arc of parabola p_0 that is similar to the function at x_0. This parabola has 2 points of intersection with the x-axis and the program denotes x_1 the one that is closer to x_0. We obtain for the first iteration, $x_1 \approx -1,4$. Then, the program is going to draw the arc of parabola p_1 and denotes x_2 its point of intersection with the x-axis that is closer to x_1 and after some iterations, we obtain the correct approximation -1.44224957030741 with a tolerance of 15 digits.



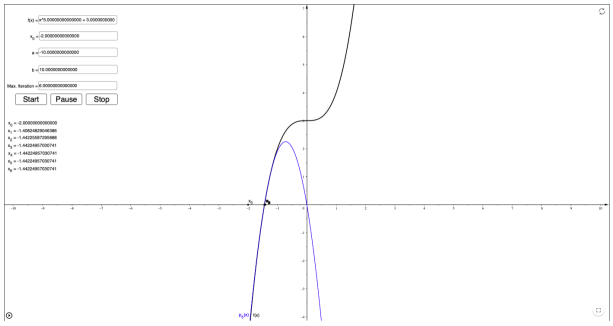


Figure 13: Animation of the *revisited* Newton-Raphson method for $f(x) = x^3 + 3$.

2.2.3 Algorithmic illustration

We are now ready to implement the *revisited* Newton-Raphson method in Python. The Python code is the following:

```
import sympy
import math
from sympy.parsing.sympy_parser import parse_expr
from sympy import *

x = symbols("x")
print("\nInput:\n")

f = input("Enter a function: ")
x0 = float(input("Enter an initial value: "))
t = float(input("Enter a desired tolerance of the root: "))
it = int(input("Enter the total number of iterations: "))

expr = parse_expr(f)
firstDeriv = expr.diff(x)
secondDeriv = firstDeriv.diff(x)

def Func(y):
    return expr.subs(x,y)

def FirstDerivFunc(y):
    return firstDeriv.subs(x,y)

def SecondDerivFunc(y):
    return secondDeriv.subs(x,y)

def NewtonParabola(f, x0, t, it):
    print("Let f(x) = ", f, ", then f'(x) = ", diff(f), ", and f''(
    x) = ", diff(diff(f)), ".\n",sep = "")
```

```
    x = x0
    fx = Func(x)
    dfx = FirstDerivFunc(x)
    ddfx = SecondDerivFunc(x)

    if t < 1e-15:
        print("The desired tolerance is to high for Python. Choose in between 1e-1 and 1e-15.")

    elif 4*0.5*(ddfx)*(fx) > (dfx)**2 or 0.5*ddfx == 0 or dfx == 0:
        print("The initial value does not have the required condition. Hence, we can't use the 'revisited' Newton-Raphson method.")

    else:
        print("The initial value is given by x0 = ", x, ", and applying the 'revisited' Newton-Raphson method, we obtain\n", sep = "")
        for n in range(it):
            y = x
            x = x - dfx/(2*0.5*ddfx)*(1 - math.sqrt(1 - (4*0.5*(ddfx)*(fx))/(dfx**2)))
            fx = Func(x)
            dfx = FirstDerivFunc(x)
            ddfx = SecondDerivFunc(x)

            if 4*0.5*(ddfx)*(fx) > (dfx)**2 or 0.5*ddfx == 0 or dfx == 0:
                print("We conclude that the sequence diverges.")
                return None

            elif abs(x - y) > t:
                print("x_", n + 1, " = ", x, sep = "")

            elif abs(x - y) <= t:
                print("\nWe found a solution after ", n, " iterations, given by ", y, ".\n", sep = "")
                return None

        print("We have surpassed the maximum number of iterations.")


print("\nOutput:\n")

NewtonParabola(f, x0, t, it)
```

Listing 2: Python code for the *revisited* Newton-Raphson method.

Explanation of the Python code.
When running the Python code, the first step is to correctly plug in the function f along with an initial value, a desired tolerance of the correct root and the wanted maximal number of iterations. From here on, the program transforms the plugged in function, which is of type *String*, into an expression in order to define the three functions **Func(y)**, **FirstDerivFunc(y)**, and **SecondDerivFunc(y)**. These three functions just evaluate the plugged in function f at y, respectively the first and second derivative of the function at y.
Then, the function **NewtonParabola(f, x0, t, it)**, which takes the given function f, the initial value $x0$, the tolerance t, and the maximum number of iterations it as the only variables, first defines the iterative values by x, starting with $x = x0$, and fx to be the value of the function f evaluated at x, respectively dfx and $ddfx$ to be the first and second derivative of f evaluated at x. Now consider the following three cases.
The first case is if the tolerance t is to high for Python, it prints out that one has to choose a tolerance in between $1e-1$ and $1e-15$.
The second case checks whether $4 \cdot \frac{1}{2} \cdot ddfx \cdot fx > (dfx)^2$, $\frac{1}{2} \cdot ddfx = 0$, or $dfx = 0$, because if one of the three conditions is verified, the iteration formula for the *revisited* Newton-Raphson method fails. Hence, it prints out that the initial value does not fulfill the conditions for the iteration formula, and it is not applicable in this case.
Lastly, if all the three conditions are not verified, a for loop is introduced which goes over every n from 0 up to the maximum number of iterations minus one. Then in this loop, it defines y to be equal to x, and replaces x using the iteration formula, which does not fail since all conditions were checked. It then proceeds to replace fx with the value of the function evaluated at this new x, respectively dfx and $ddfx$ with the first and second derivative evaluated at x, and considers the following three cases.
In the first case, it again has to check if one of the three conditions occurs. If it is the case, it prints out that the sequence diverges, and returns *None* in order to end the function at this point. Note that this return is essential, because otherwise it would go back to the for loop.
The second case is if the absolute value of the difference of x and y is greater than the tolerance t. If it is the case, it prints out the value of x, and defines this value to be equal to the $n + 1^{th}$ iteration. Note that since the for loop starts at 0, it has to add 1 as $x0$ is already defined in the beginning. After this, it goes back to the for loop.
Lastly, if the absolute value of the difference is smaller or equal to the tolerance t, i.e. the desired root with the given tolerance is found, it prints out the number of iterations needed for finding this root, along with the value of this root. As before, in order to end the function at this point, it returns *None*.

If the for loop has gone over every n, i.e. it does not have found a solution, it prints out that the maximum number of iterations has been surpassed.

Example 2.7. Choose the same function as in Example 2.2, $f(x) = x^3 + 3$, along with the same initial value $x_0 = -2$ and the same tolerance of $1e - 10$. Moreover, set the total number of iterations to 100. After running the Python code, we obtain the following:

```
Input:

Enter a function: x**3 + 3
Enter an initial value: -2
Enter a desired tolerance of the root: 1e-10
Enter the total number of iterations: 100

Output:

Let f(x) = x**3 + 3, then f'(x) = 3*x**2, and f''(x) = 6*x.

The initial value is given by x0 = -2.0, and applying the 'revisited' Newton-Raphson method, we obtain

x_1 = -1.40824829046386
x_2 = -1.44225587295888
x_3 = -1.44224957030741

We found a solution after 3 iterations, given by -1.44224957030741.
```

Figure 14: Input and output of Example 2.7.

Thus after 3 iterations, we obtain the correct root of the function f.

nbr. of iterations	x_m	nbr. of correct digits
0	-2,00000000000000	0
1	-1,40824829046386	2
2	-1,44225587295888	5
3	-1,44224957030741	15

Remark 2.3. We observe from the above table, that after each iteration, the number of correct digits is roughly tripled. Intuitively, we can assume that the *revisited* Newton-Raphson method has a cubic convergence.

Example 2.8. Choose the same function as above, $f(x) = x^3 + 3$, along with the same initial value $x_0 = -2$, but this time we want a better tolerance, we want a tolerance of $1e - 20$. Moreover, set the total number of iterations to 100. The tolerance we wish to have is to good for Python, thus after running the Python code, we obtain the following:

```
Input:

Enter a function: x**3 + 3
Enter an initial value: -2
Enter a desired tolerance of the root: 1e-20
Enter the total number of iterations: 100

Output:

Let f(x) = x**3 + 3, then f'(x) = 3*x**2, and f''(x) = 6*x.

The desired tolerance is to high for Python. Choose in between 1e-1 and 1e-15.
```

Figure 15: Input and output of Example 2.8.

Example 2.9. Consider the function, $f(x) = x^4 + 3x^2 + 2$, along with an initial value $x_0 = 0$, and a tolerance of $1e-7$. Moreover, set the total number of iterations to 10. After calculating the derivative of f evaluated in x_0 we find that $f'(x_0) = 0$, which implies that we can't use the iteration formula, since we divide by $f'(x_0)$. We see that we cannot use Newton-Raphson's method, thus after running the Python code, we obtain the following:

```
Input:

Enter a function: x**4 + 3*x**2 + 2
Enter an initial value: 0
Enter a desired tolerance of the root: 1e-7
Enter the total number of iterations: 10

Output:

Let f(x) = x**4 + 3*x**2 + 2, then f'(x) = 4*x**3 + 6*x, and f''(x) = 12*x**2 + 6.

The initial value does not have the required condition. Hence, we can't use the 'revisited' Newton-Raphson method.
```

Figure 16: Input and output of Example 2.9.

Example 2.10. Let $f(x) = x^4 + 3x^2 + 2$ be a function, and choose an initial value $x_0 = 4$, and a tolerance of $1e-10$. Moreover, set the total number of iterations to 20. After calculating $f(x_0)$, $f'(x_0)$ and $f''(x_0)$, we see that $4 \cdot \frac{1}{2} \cdot f''(x_0) \cdot f(x_0) > f'(x_0)^2$, so that we cannot use the iteration formula since the value in the square root is negative. In this case, the Python code returns the following:

```
Input:

Enter a function: x**4 + 3*x**2 + 2
Enter an initial value: 4
Enter a desired tolerance of the root: 1e-10
Enter the total number of iterations: 100

Output:

Let f(x) = x**4 + 3*x**2 + 2, then f'(x) = 4*x**3 + 6*x, and f''(x) = 12*x**2 + 6.

The initial value does not have the required condition. Hence, we can't use the 'revisited' Newton-Raphson method.
```

Figure 17: Input and output of Example 2.10.

Comparison of the two Newton-Raphson methods.
Comparing the results obtained in Example 2.2 and Example 2.7, we see that the *revisited* Newton-Raphson method, converges faster to the desired root of the function f as the usual Newton-Raphson method.

nbr. of iterations	nbr. of correct digits using Newton-Raphson	nbr. of correct digits using accelerated version of Newton-Raphson
0	0	0
1	1	2
2	2	5
3	4	15
4	9	
5	15	

3 Factorising polynomials over finite fields using Berlekamp's algorithm

In the section above, we explained the Newton-Raphson method, one of many methods that helps solving polynomial equations over real numbers. Now, it might be interesting to look at a method to solve polynomial equations over finite fields. The algorithm we are going to explain is the so called *Berlekamp algorithm*.
Berlekamp's algorithm is named after the mathematician Elwyn Berlekamp, who developed it in 1967, and it is an algorithm used to factorise polynomials with coefficients over finite fields. For years, Berlekamp's algorithm was the dominant method for solving polynomial equations over finite fields, until 1981, when the Cantor-Zassenhaus algorithm was created.
Berlekamp's algorithm consists mainly of matrix reduction and polynomial greatest common divisor (short gcd) computations. The input is a polynomial P of degree $n \in \mathbb{N}$ with coefficients in a given finite field and the output is a polynomial b with coefficients in the same finite field, that divides P. Applying the algorithm recursively to these subsequent divisors b, we find the decomposition of P into irreducible polynomials.

3.1 General statement

Let

- $\mathbb{F}_p[x]$ be a finite field of size $p \in \mathbb{P}$ (where $\mathbb{P}$ is the set of all prime numbers), e.g. $\mathbb{F}_3[x]$ is the set of polynomials with coefficients in $\mathbb{F}_3 = \{0, 1, 2\}$ (we can imagine it as $\mathbb{F}_3 \equiv \mathbb{N} \mod 3$),
- $P(x) \in \mathbb{F}_p[x]$ be a polynomial of degree $n \in \mathbb{N}$ with $\mathbb{F}_p$-coefficients in x.

The **goal** is to factorise $P(x)$ in $\mathbb{F}_p[x]$, i.e. to find irreducible factors $p_1(x), ..., p_k(x) \in \mathbb{F}_p[x]$, where $k \in \mathbb{N}$ is unknown, such that

$$P(x) = p_1(x) \cdot ... \cdot p_k(x) = \prod_{i=1}^{k} p_i(x).$$

Note that if $k = 1$, then $P(x) = p(x)$ and we conclude that $P(x)$ is already irreducible.
In addition, we assume that $P(x)$ is square-free, i.e. that all the p_i's are distinct. One of the reasons why we assume that $P(x)$ is square-free is that by taking the gcd of $P(x)$ and the derivative of $P(x)$ with respect to x, denoted by $P'(x) = \frac{d}{dx}P(x)$ we always have that $\gcd(P(x),\ P'(x)) = 1$. Note that this is the case because every

multiple factor of a polynomial P over a field introduces a nontrivial common factor of P and P'. If $P(x)$ was not square-free, then we would have found a divisor of $P(x)$ by calculating $\gcd(P(x),\ P'(x))$.
The important question now is, how to find the irreducible factors $p_i(x)$ of $P(x)$ $\forall i \in \{1, ..., k\}$. In order to do this, consider a polynomial

$$G(x) := \prod_{a \in \mathbb{F}_p} (b(x) - a) \quad \in \mathbb{F}_p[x],$$

that is divisible by $P(x)$ and that is easier to factorise, where $b(x) \in \mathbb{F}_p[x]$. This polynomial is divisible by $P(x)$ if

$$P(x) = \prod_{i=1}^{k} p_i(x) = \prod_{a \in \mathbb{F}_p} \gcd(P(x),\ b(x) - a). \tag{7}$$

Also, since $\mathbb{F}_p$ is a finite field, we can replace in the identity $x^p - x = \prod_{a \in \mathbb{F}_p}(x - a)$, x by our $b(x)$ and we obtain:

$$b(x)^p - b(x) = \prod_{a \in \mathbb{F}_p} (b(x) - a) = G(x).$$

One may ask how the polynomials $b(x)$ are found.
In fact, we want that $P(x)$ divides $G(x)$ and we know that $G(x) = b(x)^p - b(x)$, thus we need to find $b(x)$ such that

$$b(x)^p \equiv b(x) \mod P(x).$$

Note that these polynomials $b(x)$ form a subalgebra of the factor ring

$$\mathcal{R} = \frac{\mathbb{F}_p[x]}{\langle P(x) \rangle}$$

called *Berlekamp subalgebra*. Since $P(x)$ divides the polynomial $G(x)$, we conclude by (7) that if the $\gcd(P(x),\ b(x) - a)$'s are irreducible, the $p_i's$ must be given by $p_i(x) = \gcd(P(x),\ b(x) - a)$ for $a \in \mathbb{F}_p$, and $b(x)$ in the Berlekamp subalgebra. If some $\gcd(P(x),\ b(x) - a)$ are not irreducible, we use the algorithm on these polynomials, until every polynomial is irreducible.
To find the p_i's, we have to find all the polynomials $b(x)$ from the Berlekamp subalgebra, i.e. all polynomials $b(x)$ satisfying $b(x)^p \equiv b(x) \mod P(x)$. In order to do this, we use the following observation.

Consider a $n \times n$ matrix $M = (m_{i,j})_{0 \leq i,j \leq n-1}$, where the coefficients $m_{i,j}$ are given by the following congruence:

$$x^{ip} \equiv \sum_{j=0}^{n-1} m_{i,j} x^j \mod P(x). \tag{8}$$

We can now show that each eigenvector $b = \begin{pmatrix} b_0 \\ \vdots \\ b_{n-1} \end{pmatrix}$ of M with eigenvalue 1 provides a polynomial $b(x)$ given by

$$b(x) := \sum_{l=0}^{n-1} b_l x^l, \tag{9}$$

that satisfies $b(x)^p \equiv b(x) \mod P(x)$. Indeed,

$$\begin{aligned}
b(x)^p \overset{(9)}{=} \left(\sum_{l=0}^{n-1} b_l x^l \right)^p \overset{\mathbb{F}_p}{=} \sum_{l=0}^{n-1} b_l^p x^{lp} \quad & \overset{\mathbb{F}_p}{=} \quad \sum_{l=0}^{n-1} b_l x^{lp} \\
& \overset{(8)}{\equiv} \quad \sum_{l=0}^{n-1} b_l \sum_{j=0}^{n-1} m_{l,j} x^j \mod P(x) \\
& \overset{\text{linearity}}{=} \quad \sum_{j=0}^{n-1} \left(\sum_{l=0}^{n-1} m_{l,j} b_l \right) x^j \mod P(x) \\
& \overset{\text{eigenvectors}}{=} \quad \sum_j b_j x^j \mod P(x) \\
& = \quad b(x) \mod P(x).
\end{aligned}$$

Finding all the eigenvectors of M for the eigenvalue 1 is equivalent to finding the null space (i.e. the Kernel) of the matrix $(M - 1 \cdot I_n)$, where I_n is the $n \times n$ identity matrix. Thus to find the polynomials $b(x)$ of the Berlekamp subalgebra, it suffices to find the basis vectors of the null space of $(M - I_n)$.
Finally, we obtain the factors $p_i(x) \forall i \in \{1, ..., k\}$ by calculating for each $a \in \mathbb{F}_p$ and for every vector $b^{(j)}$ in the null space of $(M - I_n)$:

$$\gcd(P(x), b^{(j)}(x) - a) \ \forall j,$$

where $b^{(j)}(x)$ is the polynomial corresponding to the vector $b^{(j)}$ and with the conditions that

- $b^{(j)} = (1, 0, ..., 0)$ can be rejected and

- we continue until we have determined $k := n - r = deg(P(x)) - rang(M - I_n)$ factors of $P(x)$ (i.e. until we have found k different $\gcd(P(x), b^{(j)}(x) - a) \neq 1$).

Hence, we can summarize Berlekamp's algorithm in the following 4 steps.

Step 1: Verify that the polynomial is square-free, i.e check if $\gcd(P(x),\ P'(x)) = 1$.

Step 2: Calculate the matrix $M = (m_{i,j})_{n \times n}$, by computing

$$x^{ip} \equiv \sum_{j=0}^{n-1} m_{i,j} x^j \mod P(x). \quad \forall i = 0, ..., n-1.$$

Step 3: Find the basis of the null space of $(M - I_n)$.

Step 4: Calculate for each $b^{(j)} \neq (1, 0, ..., 0)$ in the basis of the null space of $(M - I_n)$ and for every $a \in \mathbb{F}_p$:
$\gcd(P(x), b^{(j)} - a)$ until $n - r = deg(P(x)) - rang(M - I_n)$ factors of $P(x)$ have been found.

Remark 3.1. In the case where the leading coefficient of the polynomial we wish to factorise is not 1, we need to multiply the obtained factorisation by the leading coefficient. For an application, look at Example 3.5.

3.2 Algorithmic illustration

We are now ready to implement Berlekamp's algorithm in Sage. The Sage code is the following:

```
def ModP(P, a, p):
    R.<x> = PolynomialRing(GF(p), 'x')
    S.<x> = R.quotient(P)

    return x^a


def Berlekamp(P, p):
    R.<x> = PolynomialRing(GF(p), 'x')
    P = R(P.list())
    n = P.degree()
    c = P.lc()
    dP = derivative(P)
    listOfCoeff = []
    print("Let P(x) = ", P, ". Then, P'(x) = ", dP, ".\n", sep = ""
   )

    gcdPDP = P.gcd(dP)
```

```
18      if gcdPDP != 1:
19          print("As gcd(P(x), P'(x)) = ", gcdPDP, ", the polynomial
    is not square free. Thus, we can't use Berlekamp's algorithm.",
    sep = "")
20
21      elif gcdPDP == 1:
22          print("As gcd(P(x), P'(x)) = ", gcdPDP, ", the polynomial
    is square free, and we can use Berlekamp's algorithm. Let us now
     compute x^(", p, "*i) mod P(x) for 0 <= i <= ", n - 1, ".\n",
    sep = "")
23
24          for i in range(n):
25              print("x^", i*p, " = ", ModP(P, i*p, p), " mod P(x).",
    sep = "")
26              listOfCoeff.append(ModP(P, i*p, p).list())
27
28          M = matrix(GF(p), listOfCoeff)
29
30          print("\nThe matrix M of order ", n, "*", n , " is given by
     \n\n", M, ",\n", sep = "")
31
32          MI = M - matrix.identity(n)
33
34          print("and the matrix M-I is \n\n", MI, ".\n", sep = "")
35
36          k = n - MI.rank()
37
38          if k == 1:
39              print("The rank of M-I is ", MI.rank(), ", therefore P
    has k = ", n, " - ", MI.rank(), " = ", k, " distinct monic
    irreducible factor. This means that the polynomial is already
    irreducible.", sep = "")
40
41          else:
42              print("The rank of M-I is ", MI.rank(), ", therefore P
    has k = ", n, " - ", MI.rank(), " = ", k, " distinct monic
    irreducible factors.\n", sep = "")
43
44              nullSpace = MI.left_kernel(basis = 'pivot')
45
46              print("The following ", nullSpace.dimension(), "
    vectors form the basis of the nullspace of M-I\n", sep = "")
47
48              for i in range(nullSpace.dimension()):
49                  print("Vector ", i+1, ": ", nullSpace.basis()[i], "
    .", sep = "")
50
51              print("\nThe polynomials corresponding to the basis
    vectors are\n")
```

```
52
53          for i in range(nullSpace.dimension()):
54              print("b", i+1, "(x) = ", R(list(nullSpace.basis()[
    i])), ".", sep = "")
55
56          print("\nThen, successively calculate\n")
57
58          for i in range(1, nullSpace.dimension()):
59              listOfMonicPolys = []
60
61              for j in range(p):
62                  print("gcd(P(x), b", i+1, "(x) - ", j, ") = ",
    P.gcd(R(list(nullSpace.basis()[i])) - j), ".", sep = "")
63
64                  if P.gcd(R(list(nullSpace.basis()[i])) - j) !=
    1:
65                      listOfMonicPolys.append(P.gcd(R(list(
    nullSpace.basis()[i])) - j))
66
67              if len(listOfMonicPolys) != k:
68                  print("\nSince P has ", k, " distinct monic
    irreducible factors, but from the above one obtains\n", sep = ""
    )
69
70                  print("P(x) = (", ")(".join(str(e) for e in
    listOfMonicPolys), ").\n", sep = "")
71
72                  if i + 2 > nullSpace.dimension():
73                      for t in range(p):
74                          for e in listOfMonicPolys:
75                              if e.degree() > 1 and e(x = t) ==
    0:
76                                  listOfMonicPolys.remove(e)
77                                  print("Since ", e, " is not
    irreducible in F", p, "[x], we use Berlekamp's algorithm to
    factorise this polynomial.\n", sep = "")
78
79                                  listOfMonicPolys = Berlekamp2(e
    , p) + listOfMonicPolys
80                                  print("We find that, ", e, " =
    (", ")(".join(str(e1) for e1 in Berlekamp2(e, p)), ").\n", sep =
     "")
81
82                                  if len(listOfMonicPolys) == k:
83                                      print("Thus, the final
    factorisation of P is\n")
84                                      if c == 1:
85                                          print("P(x) = (", ")(".
    join(str(e1) for e1 in listOfMonicPolys), ").", sep = "")
```

```
                                                    break
                                                else:
                                                    print("P(x) = ", c, "("
    , ")(".join(str(e1) for e1 in listOfMonicPolys), ").", sep = "")
                                                    break
                        else:
                            print("Thus, repeat the procedure for b", i
    +2, "(x).\n",sep = "")
                            continue

                    else:
                        print("\nSince P has ", k, " distinct monic
    irreducible factors, our desired factorisation of P is\n", sep =
     "")
                        if c == 1:
                            print("P(x) = (", ")(".join(str(e) for e in
     listOfMonicPolys), ").", sep = "")
                            break
                        else:
                            print("P(x) = ", c, "(", ")(".join(str(e)
    for e in listOfMonicPolys), ").", sep = "")
                            break
```

Listing 3: Sage code for Berlekamp's algorithm.

Explanation of the Sage code.
The function **ModP**(P, a, p), which takes a polynomial P, a positive integer a, and a prime number p as the only 3 variables, first creates a polynomial ring over a finite field of size p, and denotes it by R. In fact, $R = GF(p)[x] = F_p[x]$. It then uses this polynomial ring R to create the quotient ring $R/\langle P\rangle$, which was also called the *Berlekamp subalgebra*. Lastly, this function returns the result of x^a mod P in R. More specifically, this function will be helpful to compute x^{ip} mod P.

The main function **Berlekamp**(P, p), which takes a polynomial P, and a prime number p as the only 2 variables, again creates a polynomial ring over a finite field of size p, and denotes it by R. It then has to redefine the polynomial P in this polynomial ring, so that $P \in R$. Note that $P.list()$ returns a list of the coefficients of P, and $R(P.list())$ generates the corresponding polynomial to those coefficients in R. It then defines n to be the degree of P, c to be the leading term of P, dP to be the derivative of P, $listOfCoeff$ to be an empty list, which will be helpful to construct the matrix M, and $gcdPDP$ to be the gcd of P and dP.
Now, if $gcdPDP$ is not equal to 1, it prints out $gcdPDP$ along with a message saying that Berlekamp's algorithm is not applicable.
On the other hand, if $gcdPDP$ is equal to 1, P is square-free, and it computes x^{ip} mod P for i starting from 0 up to $n-1$, using the defined function **ModP**(P, ip, p). Simultaneously, it adds a list of the coefficients of each x^{ip} mod P to *listOfCoeff*.

Thus, it defines M to be the matrix over the finite field of size p, the matrix MI to be the difference of M and the $n \times n$ identity matrix, and k to be the difference of n and the rank of MI. Now, consider two cases.
If k is equal to one, it prints out that P is already irreducible.
On the other hand, if k is not one, it prints out that P has k distinct monic irreducible factors. It then defines *nullSpace* to be the null space of MI, and extracts its basis vectors by using *nullSpace.basis[i]*, where i ranges from 0 up to the dimension of *nullSpace* minus one. This function just returns the i-th basis vector of *nullSpace*. The corresponding polynomials to those basis vectors are given by *R(list(nullSpace.basis()[i]))*, where again i ranges from 0 up to the dimension of *nullSpace* minus one, and they are denoted by $b_{i+1}(x)$. Note that each basis vector has to be transformed into a list by using the function *list()* in order to obtain the corresponding polynomial to those coefficients.
Now, the first for loop ranges from 1 up to the dimension of *nullSpace* minus one, and then defines *listOfMonicPolys* to be an empty list, which will be helpful to memorize all monic irreducible factors. Then, the second for loop ranges from 0 up to $p-1$, and in this loop, it computes the gcd of P and $b_{i+1}(x) - j$. Note that since i starts at 1, $b_1(x)$ will not be considered as it is always rejected. Furthermore, if the gcd of P and $b_{i+1}(x) - j$ is not equal to 1, it will be added to *listOfMonicPolys*.
Again, consider the following two cases.
If the length of *listOfMonicPolys* is not equal to k, it prints out the obtained factorisation using *listOfMonicPolys*, which is not the wanted one, since there aren't k monic irreducible factors. Now it has to consider two cases.
If there aren't no more $b_{i+1}(x)$s to check, a first for loop which goes through every t from 0 up to $p-1$, and a second for loop which goes over every element e in *listOfMonicPolys* are introduced, in order to check which element e is not irreducible. Hence, it must check if the degree of e is greater than 1, and if there is a t for which e is zero when evaluating at t. If the two conditions are verified, the element e is removed from *listOfMonicPolys*, since only irreducible factors are memorized in this list. Then, it replaces *listOfMonicPolys* by the sum of **Berlekamp2(e, p)** and *listOfMonicPolys*. Note that this is a modified function of the current function, and just returns a list of all monic irreducible factors of e. Now if all k monic irreducible factors are found, it prints out the obtained factorisation using *listOfMonicPolys*, and by using *break*, it jumps out of both for loops, and reaches the end of the defined function. Note that it distinguishes between the cases whether c is equal to 1 or not for a better visualization.
If there are any $b_{i+1}(x)$s left to check, it has to jump back to the first loop by using *continue*.
On the other side, if the length of *listOfMonicPolys* is indeed equal to k, it prints out the obtained factorisation using *listOfMonicPolys*, and by using *break*, it jumps

out of the first for loop, and reaches the end of the defined function. Again, it distinguishes between the cases whether c is equal to 1 or not for a better visualization.

Example 3.1. If we consider the polynomial $P(x) = x^2 + 2x + 1$ over the finite field $\mathbb{F}_3$, then we cannot use Berlekamp's algorithm since it is not square free. In this case, the Sage program returns the following:

```
Berlekamp(x^2 + 2*x + 1, 3)
    Let P(x) = x^2 + 2*x + 1. Then, P'(x) = 2*x + 2.

    As gcd(P(x), P'(x)) = x + 1, the polynomial is not square free. Thus, we can't use Berlekamp's algorithm.
```

Figure 18: Output of Example 3.1.

Example 3.2. Consider the polynomial $P(x) = x^8 + 3x^5 + 2x + 1$ over the finite field $\mathbb{F}_5$. After calculating the rank of the matrix $M - I_8$ we find that the polynomial is already irreducible. In this case, the Sage program returns the following:

```
Berlekamp(x^8 + 3*x^5 + 2*x + 1, 5)
    Let P(x) = x^8 + 3*x^5 + 2*x + 1. Then, P'(x) = 3*x^7 + 2.

    As gcd(P(x), P'(x)) = 1, the polynomial is square free, and we can use Berlekamp's algorithm. Let us now compute x^(5*i) mod P(x) for 0 <= i <= 7.

    x^0 = 1 mod P(x).
    x^5 = x^5 mod P(x).
    x^10 = 2*x^7 + 3*x^3 + 4*x^2 mod P(x).
    x^15 = 4*x^7 + 3*x^6 + 2*x^5 + 3*x^4 + 2*x^2 + 2 mod P(x).
    x^20 = x^7 + 2*x^6 + x^5 + 3*x^3 + x^2 + 2*x + 4 mod P(x).
    x^25 = 3*x^7 + x^6 + x^5 + x^3 + 4*x + 3 mod P(x).
    x^30 = 2*x^7 + x^6 + 3*x^5 + 2*x^3 + 2*x^2 + 3*x + 2 mod P(x).
    x^35 = 3*x^7 + x^6 + x^5 + x^4 + 3*x^3 + 4*x^2 + 3*x + 1 mod P(x).

    The matrix M of order 8*8 is given by

    [1 0 0 0 0 0 0 0]
    [0 0 0 0 0 1 0 0]
    [0 0 4 3 0 0 0 2]
    [2 0 2 0 3 2 3 4]
    [4 2 1 3 0 1 2 1]
    [3 4 0 1 0 1 1 3]
    [2 3 2 2 0 3 1 2]
    [1 3 4 3 1 1 1 3],

    and the matrix M-I is

    [0 0 0 0 0 0 0 0]
    [0 4 0 0 0 1 0 0]
    [0 0 3 3 0 0 0 2]
    [2 0 2 4 3 2 3 4]
    [4 2 1 3 4 1 2 1]
    [3 4 0 1 0 0 1 3]
    [2 3 2 2 0 3 0 2]
    [1 3 4 3 1 1 1 2].

    The rank of M-I is 7, therefore P has k = 8 - 7 = 1 distinct monic irreducible factor. This means that the polynomial is already irreducible.
```

Figure 19: Output of Example 3.2.

Example 3.3. Consider the polynomial $P(x) = x^3 + x^2 + x + 1$ over the finite field $\mathbb{F}_5$. Let us factorise it using Berlekamp's algorithm. After running the Sage code, we obtain the following:

```
Berlekamp(x^3 + x^2 + x + 1, 5)
    Let P(x) = x^3 + x^2 + x + 1. Then, P'(x) = 3*x^2 + 2*x + 1.

    As gcd(P(x), P'(x)) = 1, the polynomial is square free, and we can use Berlekamp's algorithm. Let us now compute x^(5*i) mod P(x) for 0 <= i <= 2.

    x^0 = 1 mod P(x).
    x^5 = x mod P(x).
    x^10 = x^2 mod P(x).

    The matrix M of order 3*3 is given by

    [1 0 0]
    [0 1 0]
    [0 0 1],

    and the matrix M-I is

    [0 0 0]
    [0 0 0]
    [0 0 0].

    The rank of M-I is 0, therefore P has k = 3 - 0 = 3 distinct monic irreducible factors.

    The following 3 vectors form the basis of the nullspace of M-I

    Vector 1: (1, 0, 0).
    Vector 2: (0, 1, 0).
    Vector 3: (0, 0, 1).

    The polynomials corresponding to the basis vectors are

    b1(x) = 1.
    b2(x) = x.
    b3(x) = x^2.

    Then, successively calculate

    gcd(P(x), b2(x) - 0) = 1.
    gcd(P(x), b2(x) - 1) = 1.
    gcd(P(x), b2(x) - 2) = x + 3.
    gcd(P(x), b2(x) - 3) = x + 2.
    gcd(P(x), b2(x) - 4) = x + 1.

    Since P has 3 distinct monic irreducible factors, our desired factorisation of P is

    P(x) = (x + 3)(x + 2)(x + 1).
```

Figure 20: Output of Example 3.3.

Example 3.4. Let $P(x) = x^5 + x^3 + x^2 + 1$ be a polynomial over the finite field $\mathbb{F}_5$. We will factorise it using Berlekamp's algorithm, however after having considered all the $\gcd(P(x), b^{(j)}(x) - a)$, where the $b^{(j)}(x)$'s are the polynomials corresponding to the basis vectors and $a \in \mathbb{F}_7$, we still haven't found the correct number of irreducible factors. That is why we have to factorise the polynomials, which we obtained, that are not irreducible. The Sage algorithm returns the following:

```
Berlekamp(x^5 + x^3 + x^2 + 1, 5)
    Let P(x) = x^5 + x^3 + x^2 + 1. Then, P'(x) = 3*x^2 + 2*x.

    As gcd(P(x), P'(x)) = 1, the polynomial is square free, and we can use Berlekamp's algorithm. Let us now compute x^(5*i) mod P(x) for 0 <= i <= 4.

    x^0 = 1 mod P(x).
    x^5 = 4*x^3 + 4*x^2 + 4 mod P(x).
    x^10 = 4*x^3 + 4*x + 4 mod P(x).
    x^15 = x^3 mod P(x).
    x^20 = x^4 + x^3 + x^2 + x + 1 mod P(x).

    The matrix M of order 5*5 is given by

    [1 0 0 0 0]
    [4 0 4 4 0]
    [4 4 0 4 0]
    [0 0 0 1 0]
    [1 1 1 1 1],

    and the matrix M-I is

    [0 0 0 0 0]
    [4 4 4 4 0]
    [4 4 4 4 0]
    [0 0 0 0 0]
    [1 1 1 1 0].

    The rank of M-I is 1, therefore P has k = 5 - 1 = 4 distinct monic irreducible factors.

    The following 4 vectors form the basis of the nullspace of M-I

    Vector 1: (1, 0, 0, 0, 0).
    Vector 2: (0, 4, 1, 0, 0).
    Vector 3: (0, 0, 0, 1, 0).
    Vector 4: (0, 1, 0, 0, 1).

    The polynomials corresponding to the basis vectors are

    b1(x) = 1.
    b2(x) = x^2 + 4*x.
    b3(x) = x^3.
    b4(x) = x^4 + x.

    Then, successively calculate

    gcd(P(x), b2(x) - 0) = 1.
    gcd(P(x), b2(x) - 1) = x + 2.
    gcd(P(x), b2(x) - 2) = x^2 + 4*x + 3.
    gcd(P(x), b2(x) - 3) = 1.
    gcd(P(x), b2(x) - 4) = x^2 + 4*x + 1.

    Since P has 4 distinct monic irreducible factors, but from the above one obtains

    P(x) = (x + 2)(x^2 + 4*x + 3)(x^2 + 4*x + 1).

    Thus, repeat the procedure for b3(x).

    gcd(P(x), b3(x) - 0) = 1.
    gcd(P(x), b3(x) - 1) = 1.
    gcd(P(x), b3(x) - 2) = x + 2.
    gcd(P(x), b3(x) - 3) = x + 3.
    gcd(P(x), b3(x) - 4) = x^3 + 1.

    Since P has 4 distinct monic irreducible factors, but from the above one obtains

    P(x) = (x + 2)(x + 3)(x^3 + 1).

    Thus, repeat the procedure for b4(x).

    gcd(P(x), b4(x) - 0) = x^3 + 1.
    gcd(P(x), b4(x) - 1) = 1.
    gcd(P(x), b4(x) - 2) = 1.
    gcd(P(x), b4(x) - 3) = x + 3.
    gcd(P(x), b4(x) - 4) = x + 2.

    Since P has 4 distinct monic irreducible factors, but from the above one obtains

    P(x) = (x^3 + 1)(x + 3)(x + 2).

    Since x^3 + 1 is not irreducible in F5[x], we use Berlekamp's algorithm to factorise this polynomial.

    We find that, x^3 + 1 = (x + 1)(x^2 + 4*x + 1). Thus, the final factorisation of P is

    P(x) = (x + 1)(x^2 + 4*x + 1)(x + 3)(x + 2).
```

Figure 21: Output of Example 3.4.

Example 3.5. Consider the polynomial $P(x) = 2x^4+5x^3+3x+1$ over the finite field $\mathbb{F}_7$. We will factorise it using Berlekamp's algorithm, however we have to multiply the final factorisation by the leading coefficient since it is not equal to 1 but 2. After running the Sage code, we obtain the following:

```
Berlekamp(2*x^4 + 5*x^3 + 3*x + 1, 7)
    Let P(x) = 2*x^4 + 5*x^3 + 3*x + 1. Then, P'(x) = x^3 + x^2 + 3.

    As gcd(P(x), P'(x)) = 1, the polynomial is square free, and we can use Berlekamp's algorithm. Let us now compute x^(7*i) mod P(x) for 0 <= i <= 3.

    x^0 = 1 mod P(x).
    x^7 = x^3 + 5*x^2 + 2*x + 2 mod P(x).
    x^14 = 3*x^3 + 2*x + 5 mod P(x).
    x^21 = 4*x^3 + 4*x^2 + x + 4 mod P(x).

    The matrix M of order 4*4 is given by

    [1 0 0 0]
    [2 2 5 1]
    [5 2 0 3]
    [4 1 4 4],

    and the matrix M-I is

    [0 0 0 0]
    [2 1 5 1]
    [5 2 6 3]
    [4 1 4 3].

    The rank of M-I is 2, therefore P has k = 4 - 2 = 2 distinct monic irreducible factors.

    The following 2 vectors form the basis of the nullspace of M-I

    Vector 1: (1, 0, 0, 0).
    Vector 2: (0, 3, 5, 1).

    The polynomials corresponding to the basis vectors are

    b1(x) = 1.
    b2(x) = x^3 + 5*x^2 + 3*x.

    Then, successively calculate

    gcd(P(x), b2(x) - 0) = x^2 + 5*x + 3.
    gcd(P(x), b2(x) - 1) = 1.
    gcd(P(x), b2(x) - 2) = 1.
    gcd(P(x), b2(x) - 3) = 1.
    gcd(P(x), b2(x) - 4) = x^2 + x + 6.
    gcd(P(x), b2(x) - 5) = 1.
    gcd(P(x), b2(x) - 6) = 1.

    Since P has 2 distinct monic irreducible factors, our desired factorisation of P is

    P(x) = 2(x^2 + 5*x + 3)(x^2 + x + 6).
```

Figure 22: Output of Example 3.5.

4 Solving polynomial congruence equations using Hensel's Lemma

After having studied a method to factorise polynomials over finite fields, it's interesting to look at a method that solves polynomial equations modulo p^n, where p is a prime number and n is a natural number greater than 1. The so called Hensel Lemma allows us to find such solutions.
Also known as Hensel's lifting Lemma and named after Kurt Hensel, this is a result that uses Berlekamp's algorithm and a variant of Newton's approximation to solve equations modulo p^n. In fact, we start from a solution modulo p, that we obtain using Berlekamp's algorithm and "improve" it to solutions modulo powers of p by using a variant of Newton's approximation.

4.1 General statement

Let

- $P(x)$ be a polynomial with integer coefficients,
- p be a prime number,

- n be a positive integer greater than 1,
- a be a solution of $P(x) \equiv 0 \mod p^n$ such that $P'(a) \not\equiv 0 \mod p$ and
- b be a solution of $P(x) \equiv 0 \mod p^{n+1}$ such that $b \equiv a \mod p^n$.

Since $b \equiv a \mod p^n$, there exists an integer t such that

$$b = a + tp^n.$$

The goal now is to determine t. First, we use the Taylor series expansion on P around a:

$$P(x) = P(a) + \frac{P'(a)}{1!}(x-a) + \frac{P''(a)}{2!}(x-a)^2 + \frac{P'''(a)}{3!}(x-a)^3 + ...$$

For $x = a + tp^n$ we obtain the following:

$$P(a+tp^n) = P(a) + P'(a)tp^n + \frac{P''(a)}{2}(tp^n)^2 + \frac{P'''(a)}{6}(tp^n)^3 + ...$$

So that, the above equation mod p^{n+1} is:

$$P(a+tp^n) \equiv P(a) + P'(a)tp^n + \frac{P''(a)}{2}(tp^n)^2 + \frac{P'''(a)}{6}(tp^n)^3 + ... \mod p^{n+1}.$$

For $m \geq 2$, we have that $(tp^n)^m \equiv 0 \mod p^{n+1}$. This implies that:

$$P(a+tp^n) \equiv P(a) + P'(a)tp^n \mod p^{n+1}.$$

We also know that $P(b) \equiv 0 \mod p^{n+1}$ and $b = a + tp^n$, thus:

$$P(a) + P'(a)tp^n \equiv 0 \mod p^{n+1}.$$

This implies that:

$$P'(a)tp^n \equiv -P(a) \mod p^{n+1}.$$

Since p^n divides $P'(a)tp^n$ and $-P(a)$ (because $P(a) \equiv 0 \mod p^n$), we have that:

$$P'(a)t \equiv -\frac{P(a)}{p^n} \mod p.$$

Also, $P'(a) \not\equiv 0 \mod p$ which means that p doesn't divide $P'(a)$, and p is prime, thus $\gcd(P'(a), p) = 1$ which implies that $P'(a)$ and p are coprime. We conclude that $P'(a)^{-1}$ exists and thus we obtain:

$$t \equiv -\frac{P(a)}{P'(a)p^n} \mod p.$$

In addition, this t is unique.
From this, we can derive Hensel's Lemma that states the following:

Lemma 4.1. Let p be a prime number and let $P(x)$ be a polynomial with integer (or p-adic integer) coefficients. Let also, n and a_n be integers such that a_n is a solution of $P(x) \equiv 0 \mod p^n$ satisfying $P'(a_n) \not\equiv 0 \mod p$.
Then, for m being an integer, there exists an integer a_{n+m} such that

$$P(a_{n+m}) \equiv 0 \mod p^{n+m}$$

and

$$a_{n+m} \equiv a_n \mod p^n.$$

Furthermore, this a_{n+m} is unique modulo p^{k+m} and can be computed explicitly by

$$a_{n+m} \equiv a_n - \frac{P(a_n)}{P'(a_n)} \mod p^{n+m}.$$

This means that, if a_1 satisfies $P(a_1) \equiv 0 \mod p$ and $P'(a_1) \not\equiv 0 \mod p$, then we can obtain a solution a_{n+1} of $P(x) \equiv 0 \mod p^{n+1}$ by the recursive formula $a_{n+1} \equiv a_n - \frac{P(a_n)}{P'(a_n)} \mod p^{n+1}$.

A proof of this Lemma can be found in [12].

4.2 Algorithmic illustration

We are now ready to implement Hensel's Lemma in Sage. The Sage code is the following:

```
def Hensel(P, p, n):
    R.<x> = PolynomialRing(GF(p), 'x')
    RP = P
    P = R(P.list())
    dP = derivative(P)

    if Berlekamp(P, p) == 0:
        print("The polynomial is not square free, thus we can't use
    Hensel's Lemma.")

    elif Berlekamp(P, p) == 1:
        print("The polynomial is already irreducible, thus we can't
    use Hensel's Lemma.")

    elif Berlekamp(P, p) == []:
        print("The polynomial has no solution in F", p, "[x], thus
   we can't use Hensel's Lemma.", sep = "")

    else:
        print("Let P(x) = ", RP, ".\n", sep = "")

```

```
19          print("We want to obtain all solutions of P(x) = 0 mod ", p
    , "^", n, " using Hensel's Lemma.\n", sep = "")
20
21          print("In order to apply Hensel's Lemma, we first find all
    solutions to P(x) = 0 mod ", p, ".\n", sep = "")
22
23          roots = Berlekamp(P, p)
24          print("By Berlekamp's algorithm, we get that ", ", ".join(
    str(r) for r in roots), " is a solution of P(x) = 0 mod ", p, "
    .\n", sep = "")
25
26          print("Let a1 = ", ", ".join(str(r) for r in roots),". Then
    , we need to verify that P'(a1) mod ", p, " is not equal to 0.\n
    ", sep = "")
27
28          print("The derivative of P is given by P'(x) = ", dP, ".
    Thus,\n", sep = "")
29
30          for r in roots:
31              print("P'(", r, ") = ", dP(x = r), " mod ", p, ".", sep
     = "")
32
33              if dP(x = r) == 0:
34                  roots.remove(r)
35
36          if len(roots) == 0:
37              print("We cannot apply Hensel's Lemma, since all the
    above derivatives are equal to 0.")
38
39          elif len(roots) == 1:
40              print("\nSo, Hensel's Lemma can be applied to ", " and
    ".join(str(r) for r in roots), ".\n",sep = "")
41
42              a1 = roots[0]
43              dPx = 1/dP(x = a1)
44              print("First set a1 = ", a1, ". Then, we calculate [P'(
    ", a1, ")]^(-1) = [", dP(x = a1),"]^(-1) = ", dPx, " mod ", p, "
    .\n", sep = "")
45
46              a = [a1]
47              print("Thus by Hensel's Lemma, we obtain\n")
48
49              k = 1
50              for j in range(2, n + 1):
51                  R.<x> = PolynomialRing(Zmod(p^j), 'x')
52                  P = R(RP.list())
53                  print("a", j, " = a", k, " - P(a", k, ")*[P'(a", k,
     ")]^(-1) mod ", p, "^", j, sep = "")
```

```
54                print("a", j, " = ", a[0], " - P(", a[0], ")*[P'(", a[0], ")]^(-1) mod ", p, "^", j, sep = "")
55                print("a", j, " = ", a[0], " - ", P(x = Integer(a[0])), "*", dPx, " mod ", p, "^", j, sep = "")
56                print("a", j, " = ", ( Integer(a[0]) - P(x = Integer(a[0])) * Integer(dPx) ) % p^j, " mod ", p, "^", j, "\n", sep = "")
57                a[0] = ( Integer(a[0]) - P(x = Integer(a[0])) * Integer(dPx) ) % p^j
58                k += 1
59
60            print("So x = ", a[0], " is a solution to P(x) = 0 mod ", p, "^", n, ".", sep = "")
61
62        elif len(roots) != 1:
63            print("\nSo, Hensel's Lemma can be applied to ", ", ".join(str(r) for r in roots), ".\n",sep = "")
64
65            a1 = roots[0]
66            dPx = 1/dP(x = a1)
67            print("First set a1 = ", a1, ". Then, we calculate [P'(", a1, ")]^(-1) = [", dP(x = a1),"]^(-1) = ", dPx, " mod ", p, ".\n", sep = "")
68
69            i = 0
70            while i < len(roots):
71                a = [roots[i]]
72                print("Thus by Hensel's Lemma, we obtain\n")
73
74                k = 1
75                for j in range(2, n + 1):
76                    R.<x> = PolynomialRing(Zmod(p^j), 'x')
77                    P = R(RP.list())
78                    print("a", j, " = a", k, " - P(a", k, ")*[P'(a", k, ")]^(-1) mod ", p, "^", j, sep = "")
79                    print("a", j, " = ", a[0], " - P(", a[0], ")*[P'(", a[0], ")]^(-1) mod ", p, "^", j, sep = "")
80                    print("a", j, " = ", a[0], " - ", P(x = Integer(a[0])), "*", dPx, " mod ", p, "^", j, sep = "")
81                    print("a", j, " = ", ( Integer(a[0]) - P(x = Integer(a[0])) * Integer(dPx) ) % p^j, " mod ", p, "^", j, "\n", sep = "")
82                    a[0] = ( Integer(a[0]) - P(x = Integer(a[0])) * Integer(dPx) ) % p^j
83                    dPx = 1/dP(x = a[0])
84                    k += 1
85
86                print("So x = ", a[0], " is a solution to P(x) = 0 mod ", p, "^", n, ".", sep = "")
```

```
87
88                  i += 1
89                  if i == len(roots):
90                       break
91                  else:
92                      a1 = roots[i]
93                      dPx = 1/dP(x = a1)
94                      print("\nSetting a1 = ", a1, ". Then, we
    calculate [P'(", a1, ")]^(-1) = [", dP(x = a1),"]^(-1) = ", dPx,
     " mod ", p, ".\n", sep = "")
```

Listing 4: Sage code for Hensel's Lemma.

Explanation of the Sage code.

Remark 4.1. For the explanation of the function **Berlekamp**(P, p), we refer Subsection 3.2 with the following modifications. Note that those modifications are necessary in order to use its result in the main function. The only difference is that all the *print()*'s are removed, and where the function is out of arguments, a *return* is added which returns a value that will be used in the main function.
For instance, if *gcdPDP* is not equal to one, it returns 0, i.e. that is the case where the polynomial is not square free.
If k is equal to one, it returns 1, i.e. the case where the polynomial is already irreducible.
Lastly, if the length of *listOfMonicPolys* is equal to k, i.e. if all k monic irreducible factors are found, it first defines an empty list *listOfRoots*, which will be helpful to save all roots of the k monic factors. Then, the for loop goes over all the elements of *listOfMonicPolys*, and checks if the element e has a root in $F_p[x]$. Note that *e.roots()* returns a list of a pair (a, b), where a is the root of e in $F_p[x]$, and b is the multiplicity of a. Also, if e has no roots in $F_p[x]$, it returns an empty list, that is why it checks if *e.roots()* is nonempty. Then, in this case, it adds the root a to *listOfRoots*. Note that *e.roots()[0]* returns the pair (a, b), and *e.roots()[0][0]* returns the root a. At the end, it returns *listOfRoots*, which contains all the roots of P.

Let us now move on to the explanation of the function **Hensel**(P, p, n) which takes P, p and n as the only 3 variables, where P is a polynomial, p is a prime number, and n is a natural number greater or equal than 2. Then as always, it creates a polynomial ring over a finite field of size p, it redefines the polynomial P in this polynomial ring, and defines dP to be the derivative of P. Now, consider the following four cases.
The first case, is if the function **Berlekamp**(P, p) returns 0, then it prints out that the polynomial is not square free, and *Hensel's Lemma* is not applicable.
The second case, if **Berlekamp**(P, p) returns 1, it prints out that the polynomial is already irreducible, and again *Hensel's Lemma* is not applicable.

In the third case, where **Berlekamp(P, p)** returns an empty list, it prints out that the polynomial has no solutions in the polynomial ring, and again *Hensel's Lemma* is not applicable. Note that from Remark 4.1, *listOfRoots* can be empty if all k factors do not have a root in $F_p[x]$.
Lastly, if none of the first three cases are true, it defines *roots* to be the list containing all the roots of $P(x) \equiv 0 \mod p$. Then, in order to verify that the derivative evaluated at each root is not equal to zero, it uses a for loop which goes over all the elements in *roots*. It first prints out the derivative at the element, and then checks if the derivative at the element is equal to zero, and if it is the case, it removes this element out of *roots*, since *Hensel's Lemma* only considers those, where the derivative is different from zero. Again, consider the following three cases.
If the length of *roots* is equal to zero, i.e. if all element are removed from the list *roots*, it prints out that *Hensel's Lemma* is not applicable, as all the derivatives are equal to 0.
On the other hand, if the length of *roots* is equal to one, i.e. there is only one root that satisfies *Hensel's Lemma*, it first defines *a1* to be equal to this root by using *roots[0]*, *dPx* to be the inverse of the derivative evaluated at *a1*, and a to be a list containing *a1*. Then, by defining k to be equal to 1, in the for loop, where j ranges from 2 up to n, it creates a new polynomial ring over the field $\mathbb{Z}/p^n\mathbb{Z}$. Then, it has to redefine the polynomial P over this new polynomial ring in order to ensure that all calculations are considered in this polynomial ring. It then prints out the calculations for $aj \mod p^j$, and replaces the first element in a by this new calculated value, and k is augmented by one. Note that in order to do these calculations in this polynomial ring, it has to transform every single value into an integer by using the function *Integer()*. Then, if j is still in the range, it repeats those calculations. If it is out of range, it prints out the obtained result, which is the solution to $P(x) \equiv 0 \mod p^n$.
Lastly, if the length of *roots* is not equal to one, i.e. there is more than one root satisfying *Hensel's Lemma*, it again defines *a1* to be the first root in *roots*, and *dPx* to be the inverse of the derivative evaluated at *a1*. By defining i to be equal to zero, a while loop is being introduced, which will be executed if i is less than the length of *roots*. Then, in this while loop, it defines a to be the i-th element of *roots*, and does again the same as explained above. After this, i is augmented be one, and checks if i is equal to the length of *roots*. If it is the case, it uses *break* to jump out of the while loop, and it reaches the end of the function. Note that this condition tells us whether there are any elements left in *roots* or not. If i is not equal to the length of *roots*, it defines *a1* to be the i-th element of *roots*, *dPx* to be the inverse of the derivative evaluated at *a1*, goes back to the while loop, and repeats the procedure, until i is equal to *len(roots)*.

Example 4.1. Consider the polynomial $P(x) = x^2+2x+1$ and let us solve $P(x) \equiv 0 \mod 3^7$, then we cannot use Berlekamp's algorithm since it is not square free and consequently, we cannot use Hensel's algorithm. In this case, the Sage program returns the following:

```
Hensel(x^2 + 2*x + 1, 3, 7)
    The polynomial is not square free, thus we can't use Hensel's Lemma.
```

Figure 23: Output of Example 4.1.

Example 4.2. Consider the polynomial $P(x) = x^8 + 3x^5 + 2x + 1$. We wish to find the solutions of $P(x) \equiv 0 \mod 5^5$, but we find that the polynomial is already irreducible and thus, the equations has no solution. In this case, the Sage program returns the following:

```
Hensel(x^8 + 3*x^5 + 2*x + 1, 5, 5)
    The polynomial is already irreducible, thus we can't use Hensel's Lemma.
```

Figure 24: Output of Example 4.2.

Example 4.3. Consider the equation $P(x) \equiv 0 \mod 7^3$, where $P(x) = 2x^4+5x^3+3x+1$. We will solve it using Hensel's Lemma, however there are no solutions of $P(x) \equiv 0 \mod 7$, so that the equation we wish to solve also has no solution, according to Hensel's Lemma. After running the Sage code, we obtain the following:

```
Hensel(2*x^4 + 5*x^3 + 3*x + 1, 7, 3)
    The polynomial has no solution in F7[x], thus we can't use Hensel's Lemma.
```

Figure 25: Output of Example 4.3.

Example 4.4. Consider the polynomial equation modulo 11^3 given by: $P(x) \equiv 0 \mod 11^3$, where $P(x) = x^{12} + 2x^8 + x^7 + 3x^2 + 1$. Let us solve it using Hensel's Lemma. After running the Sage code, we obtain the following:

```
Hensel(x^12 + 2*x^8 + x^7 + 3*x^2 + 1, 11, 3)
   Let P(x) = x^12 + 2*x^8 + x^7 + 3*x^2 + 1.

   We want to obtain all solutions of P(x) = 0 mod 11^3 using Hensel's Lemma.

   In order to apply Hensel's Lemma, we first find all solutions to P(x) = 0 mod 11.

   By Berlekamp's algorithm, we get that 4 is a solution of P(x) = 0 mod 11.

   Let a1 = 4. Then, we need to verify that P'(a1) mod 11 is not equal to 0.

   The derivative of P is given by P'(x) = x^11 + 5*x^7 + 7*x^6 + 6*x. Thus,

   P'(4) = 4 mod 11.

   So, Hensel's Lemma can be applied to 4.

   First set a1 = 4. Then, we calculate [P'(4)]^(-1) = [4]^(-1) = 3 mod 11.

   Thus by Hensel's Lemma, we obtain

   a2 = a1 - P(a1)*[P'(a1)]^(-1) mod 11^2
   a2 = 4 - P(4)*[P'(4)]^(-1) mod 11^2
   a2 = 4 - 88*3 mod 11^2
   a2 = 103 mod 11^2

   a3 = a2 - P(a2)*[P'(a2)]^(-1) mod 11^3
   a3 = 103 - P(103)*[P'(103)]^(-1) mod 11^3
   a3 = 103 - 0*3 mod 11^3
   a3 = 103 mod 11^3

   So x = 103 is a solution to P(x) = 0 mod 11^3.
```

Figure 26: Output of Example 4.4.

Example 4.5. Let $P(x) = x^3 + x^2 + x + 1$ be a polynomial and let us solve the equation $P(x) \equiv 0 \mod 5^6$ us ing Hensel's Lemma. The Sage algorithm returns the following:

```
Hensel(x^3 + x^2 + x + 1, 5, 6)
   Let P(x) = x^3 + x^2 + x + 1.

   We want to obtain all solutions of P(x) = 0 mod 5^6 using Hensel's Lemma.

   In order to apply Hensel's Lemma, we first find all solutions to P(x) = 0 mod 5.

   By Berlekamp's algorithm, we get that 2, 3, 4 is a solution of P(x) = 0 mod 5.

   Let a1 = 2, 3, 4. Then, we need to verify that P'(a1) mod 5 is not equal to 0.

   The derivative of P is given by P'(x) = 3*x^2 + 2*x + 1. Thus,

   P'(2) = 2 mod 5.
   P'(3) = 4 mod 5.
   P'(4) = 2 mod 5.

   So, Hensel's Lemma can be applied to 2, 3, 4.

   First set a1 = 2. Then, we calculate [P'(2)]^(-1) = [2]^(-1) = 3 mod 5.

   Thus by Hensel's Lemma, we obtain

   a2 = a1 - P(a1)*[P'(a1)]^(-1) mod 5^2
   a2 = 2 - P(2)*[P'(2)]^(-1) mod 5^2
   a2 = 2 - 15*3 mod 5^2
   a2 = 7 mod 5^2

   a3 = a2 - P(a2)*[P'(a2)]^(-1) mod 5^3
   a3 = 7 - P(7)*[P'(7)]^(-1) mod 5^3
   a3 = 7 - 25*3 mod 5^3
   a3 = 57 mod 5^3

   a4 = a3 - P(a3)*[P'(a3)]^(-1) mod 5^4
   a4 = 57 - P(57)*[P'(57)]^(-1) mod 5^4
   a4 = 57 - 375*3 mod 5^4
   a4 = 182 mod 5^4
```

```
a5 = a4 - P(a4)*[P'(a4)]^(-1) mod 5^5
a5 = 182 - P(182)*[P'(182)]^(-1) mod 5^5
a5 = 182 - 2500*3 mod 5^5
a5 = 2057 mod 5^5

a6 = a5 - P(a5)*[P'(a5)]^(-1) mod 5^6
a6 = 2057 - P(2057)*[P'(2057)]^(-1) mod 5^6
a6 = 2057 - 6250*3 mod 5^6
a6 = 14557 mod 5^6

So x = 14557 is a solution to P(x) = 0 mod 5^6.

Setting a1 = 3. Then, we calculate [P'(3)]^(-1) = [4]^(-1) = 4 mod 5.

Thus by Hensel's Lemma, we obtain

a2 = a1 - P(a1)*[P'(a1)]^(-1) mod 5^2
a2 = 3 - P(3)*[P'(3)]^(-1) mod 5^2
a2 = 3 - 15*4 mod 5^2
a2 = 18 mod 5^2

a3 = a2 - P(a2)*[P'(a2)]^(-1) mod 5^3
a3 = 18 - P(18)*[P'(18)]^(-1) mod 5^3
a3 = 18 - 50*4 mod 5^3
a3 = 68 mod 5^3

a4 = a3 - P(a3)*[P'(a3)]^(-1) mod 5^4
a4 = 68 - P(68)*[P'(68)]^(-1) mod 5^4
a4 = 68 - 375*4 mod 5^4
a4 = 443 mod 5^4

a5 = a4 - P(a4)*[P'(a4)]^(-1) mod 5^5
a5 = 443 - P(443)*[P'(443)]^(-1) mod 5^5
a5 = 443 - 625*4 mod 5^5
a5 = 1068 mod 5^5

a6 = a5 - P(a5)*[P'(a5)]^(-1) mod 5^6
a6 = 1068 - P(1068)*[P'(1068)]^(-1) mod 5^6
a6 = 1068 - 0*4 mod 5^6
a6 = 1068 mod 5^6

So x = 1068 is a solution to P(x) = 0 mod 5^6.

Setting a1 = 4. Then, we calculate [P'(4)]^(-1) = [2]^(-1) = 3 mod 5.

Thus by Hensel's Lemma, we obtain

a2 = a1 - P(a1)*[P'(a1)]^(-1) mod 5^2
a2 = 4 - P(4)*[P'(4)]^(-1) mod 5^2
a2 = 4 - 10*3 mod 5^2
a2 = 24 mod 5^2

a3 = a2 - P(a2)*[P'(a2)]^(-1) mod 5^3
a3 = 24 - P(24)*[P'(24)]^(-1) mod 5^3
a3 = 24 - 50*3 mod 5^3
a3 = 124 mod 5^3

a4 = a3 - P(a3)*[P'(a3)]^(-1) mod 5^4
a4 = 124 - P(124)*[P'(124)]^(-1) mod 5^4
a4 = 124 - 250*3 mod 5^4
a4 = 624 mod 5^4

a5 = a4 - P(a4)*[P'(a4)]^(-1) mod 5^5
a5 = 624 - P(624)*[P'(624)]^(-1) mod 5^5
a5 = 624 - 1250*3 mod 5^5
a5 = 3124 mod 5^5

a6 = a5 - P(a5)*[P'(a5)]^(-1) mod 5^6
a6 = 3124 - P(3124)*[P'(3124)]^(-1) mod 5^6
a6 = 3124 - 6250*3 mod 5^6
a6 = 15624 mod 5^6

So x = 15624 is a solution to P(x) = 0 mod 5^6.
```

Figure 27: Output of Example 4.5.

5 Conclusion

The main purpose of this research project, which is the study of methods used for solving polynomial equations over real numbers and over finite fields, has given us the opportunity to analyse and explore several results in greater detail.
In our journey through polynomial equations, we did not only learn new methods and algorithms, such as Newton-Raphson's method, Berlekamp's algorithm and Hensel's Lemma, but we also had the chance to illustrate them practically. This allowed us to verify the learned properties in our animations and implement them in our programs. Moreover, the realisation of these methods and algorithms in Python and Sage have given us the opportunity to better understand the theory studied beforehand. Since, we had to program codes that give the factorisation of given polynomials in given

fields, we had to write down step by step how it's done. By decomposing the theory and programming each execution step, we had to make sure we fully understood the concept behind it.
Thus, programming not only helped us understand the algorithms, but was also a more practical way to understand the theory, and how it can be applied to solve polynomial equations.
To conclude, we can say that the option "Experimental Mathematics" not only allows students to study mathematical methods and algorithms, but also discover the research of mathematics and explore subjects in greater detail, as using programming languages Python and Sage to implement the theory learned.

Acknowledgement. The three authors would like to thank Prof. Dr. Gabor Wiese and Guendalina Palmirotta for having supervised this project in the Experimental Mathematics Lab of University of Luxembourg.

References

[1] Quadratic equation. *Wikipedia,* December 2020. Page Version ID: 43588.

[2] Cubic equation. *Wikipedia,* December 2020. Page Version ID: 180787.

[3] Méthode de Cardan. *Wikipedia,* October 2020. Page Version ID: 153526.

[4] Algebra Identities: Standard Identities Of Algebra, Definition & Examples. https://www.embibe.com/exams/algebra-identities/

[5] Jean-Luc Marichal, *Analyse numérique*, Université du Luxembourg, Version 2013-2014.

[6] Guendalina Palmirotta, *On the Geometry of Householder's and Halley's Methods*, Student project, University of Luxembourg, Winter semester 2015-2016.

[7] Newton's method. *Wikipedia,* December 2020. Page Version ID: 22145.

[8] Sajid Hanif, Muhammad Imran, *Factorization Algorithms for Polynomials over Finite Fields*, Linnaeus University, School of Computer Science, Physics and Mathematics, 2011-05-03. https://www.diva-portal.org/smash/get/diva2:414578/FULLTEXT01.pdf

[9] Berlekamp-Algorithmus. *Wikipedia,* June 2018. Page Version ID: 1875740.

[10] Berlekamp's algorithm. *Wikipedia,* December 2020. Page Version ID: 6057100.

[11] Hensel's lemma. *Wikipedia,* December 2020. Page Version ID: 1633368.

[12] Hensel's Lemma. http://mathonline.wikidot.com/hensel-s-lemma

[13] Lecture 7: Polynomial congruences to prime power moduli. https://people.maths.bris.ac.uk/~mazag/nt/lecture7.pdf