

Computation Methods for Numerical Analysis Using SageMath

MAT251 PROJECT

Lecturer: Dr. D. A. Dikko

University of Ibadan, Ibadan, Oyo State, Nigeria

October, 2023



Table of Contents

- Introduction
- Why Numerical Analysis?
- Group Members
- SageMath: Your Power Tool
 - The Jupyter Notebook
 - SageMath as a Calculator
 - Plotting Graphs in SageMath
 - * 2D Plots
 - * 3D Plots
- Exploring Numerical Methods with SageMath
- The Root Bisection Method
- Linear Iteration
- Newton-Raphson Iteration Method
- Interpolation
- Newton's Forward Difference
- Newton's Backward Difference
- Binomial Series
- Trapezoidal Rule
- Simpson's Rule
- Summary
- References

Introduction

As 200-level students in the Departments of Mathematics and Statistics, University of Ibadan, we are embarking on an exciting journey into the world of computational methods for numerical analysis using SageMath. This project will provide us with valuable insights and hands-on experience in utilizing mathematical software to solve real-world problems.

Why Numerical Analysis?

Numerical analysis is a fundamental branch of mathematics that focuses on developing and implementing computational techniques to solve mathematical problems. It plays a crucial role in a wide range of disciplines, from engineering and physics to economics and computer science. In this project, we will explore how numerical methods can be applied to practical problems and gain a deeper understanding of their significance.

Group Members

1. 231020 Babalola Isa Gbolahan
2. 231007 Adenekan Damilola Omotoyosi
3. 236634 Owolabi Odunayo Joseph
4. 231224 Oladokun Shukuroh Tolani
5. 231010 Agunbiade Olamide Roimot
6. 231235 Raji Mueez Olanrewaju
7. 232089 Mudasiru Samad Olamide
8. 231023 Chukwudi Emmanuel Emeka
9. 231024 Daramola Similoluwa
10. 231068 Shoyemi Opeyemi Samuel
11. 231008 Adeyemi Ridwan Oluwaseyi
12. 231191 Afolabi Nathanael Ademola
13. 231190 Adetoye Inumidun Adekoyejo

SageMath: Your Powerful Tool

SageMath is a powerful open-source mathematics software system that integrates various mathematical tools, packages, and programming languages into a single, cohesive environment. It was developed to provide a unified platform for mathematical research, exploration, and teaching.

SageMath offers a wide array of features, including symbolic and numerical mathematics, data visualization, and support for various programming languages such as Python. Researchers, mathematicians, educators, and students use SageMath for a range of mathematical tasks, from algebra and calculus to number theory and cryptography.

One of SageMath's distinguishing characteristics is its open-source nature, allowing users to access and modify its source code freely. This fosters collaboration, customization, and the development of mathematical tools that cater to specific needs.

SageMath's versatility and the support of a robust community of users and developers make it a valuable resource in the world of mathematics and computational science. Whether you are a seasoned mathematician or a student learning mathematics, SageMath provides a comprehensive and flexible environment to explore and solve mathematical problems.

The Jupyter Notebook

Jupyter Notebook is a web-based computational environment for creating documents interacting with Python code. It is intended to present code in a nice way and is also capable of \LaTeX formatting. Jupyter Notebook is part of the open-source project Jupyter[2], and is already included in Sage.

In order to start Jupyter, we type the following command into the Sage command line:

```
sage: !sage -n jupyter
```

A browser window pops up with the Jupyter web-interface.

SageMath as a Calculator

SageMath offers a wide range of mathematical functions and capabilities, making it a versatile tool for various mathematical calculations and problem-solving tasks. We can perform both basic arithmetic and complex mathematical operations using SageMath as a calculator.

Henceforth, all commands and calculations in Sage are presented as follows:

```
sage: 2+3  
5
```

Here, the prefix sage indicates that we are using the Sage command line. If we are working on Jupyter instead, all we just type in the subsequent code and the kernel will do its job.

Notice that all common operations such as $+$, $-$, $*$, $/$ and parentheses are carried out as usual. To operate with exponents, we use $^$ or $**$, the following are some other basic operations, predefined functions and constants.

General Arithmetics	
Binary Operations	a+b, a-b, a*b, a/b
Exponent	a^b or a**b
Square Root	sqr(a)
n-th Root	a^(1/n)
Integer Arithmetics	
Floor-Division	a // b
Remainder	a % b
Floor-Division & Remainder	divmod(a,b)
Factorial n!	factorial(n)
Binomial Coefficient $\binom{n}{k}$	binomial (n,k)

(a) Basic operations in sage

Predefined Functions	
Exponential, Natural Logarithm	exp, log
Logarithm w.r.t. Base b	log(a,b)
Trig. Functions	sin, cos, tan
Inverse Trig. Functions	arcsin, arccos, arctan
Hyp. Trig. Functions	sinh, cosh, tanh
Inverse Hyp. Trig. Functions	arcsinh, arccosh, arctanh
Absolute Value / Modulus	abs(a)
Special Values / Constants	
Imaginary Unit i	I or i
Plus / Minus Infinity	\pm Infinity or $\pm\infty$
π	pi
Euler's Constant	e
Euler-Mascheroni Constant γ	euler_gamma
Golden Ratio $\phi = (1 + \sqrt{5})/2$	golden_ratio

(b) Predefined functions and constants

Plotting Graphs in SageMath

Plotting graphs in SageMath is a fundamental and powerful feature. We can create various types of graphs, including functions, parametric plots, polar plots, and more. We shall consider simple plot and 3D plot.

2D Plots

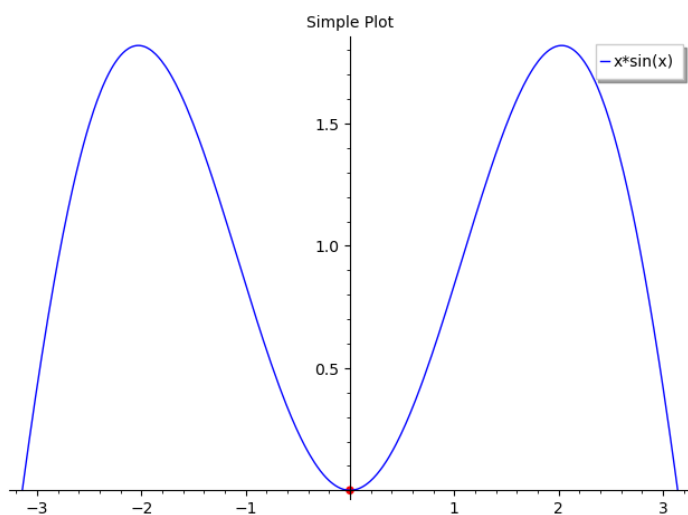
The code to plot 2D(or simple plot) is given in the diagram below:

```

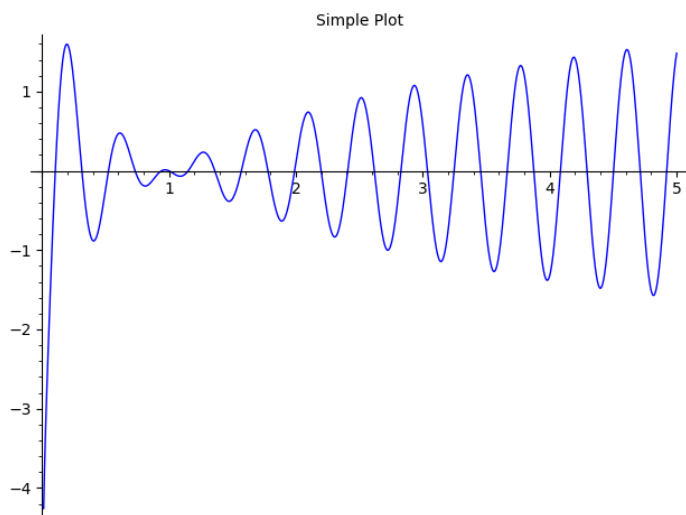
1 # Define the mathematical function or expression you want to plot
2 f(x) =
3
4
5 # Create a plot of the function in the range a to b
6 p = plot(f, (x, x_0, x_1), title='Simple Plot', legend_label= {f(x)}, color='blue')
7
8 #p += point((0, 0), color='red', size=30)
9 # Show the plot
10 p.show()
```

We now draw the graph of the following functions using the above code.

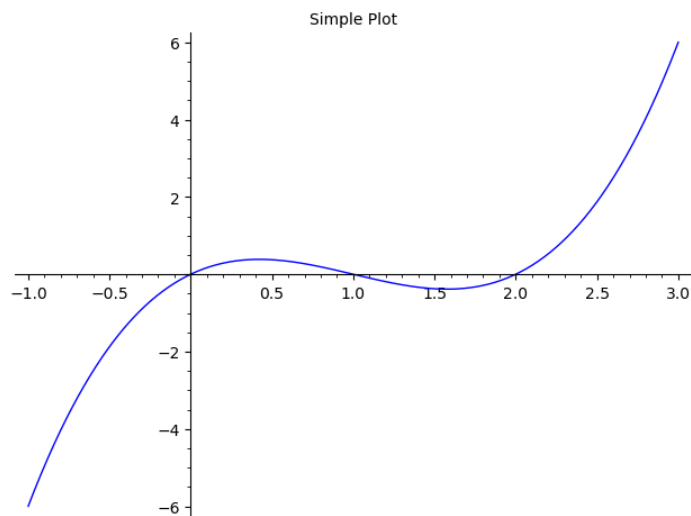
- $y = x \sin(x)$,
- $y = \ln(x) \cos(15x)$,
- $y = x^3 - 3x^2 + 2x$,
- $y = x^2 (x - 1)^2$,
- $y = -x^3 + 2x^2 + 5x - 30$,
- $y = x^5 - 5x^4 + 5x^3 + 5x^2 - 6x - 1$



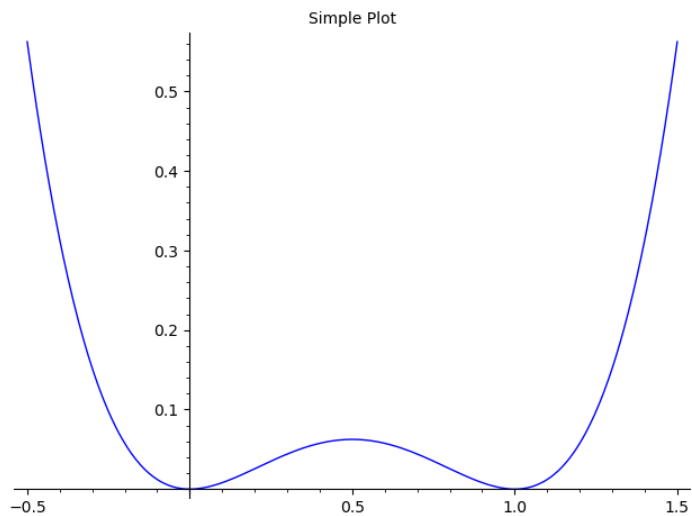
(a) Graph of $y = x \sin(x)$



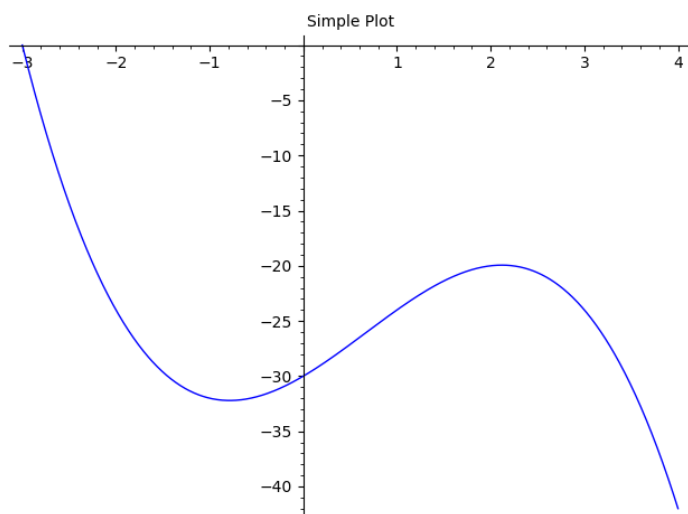
(b) Graph of $y = \ln(x) \cos(15x)$



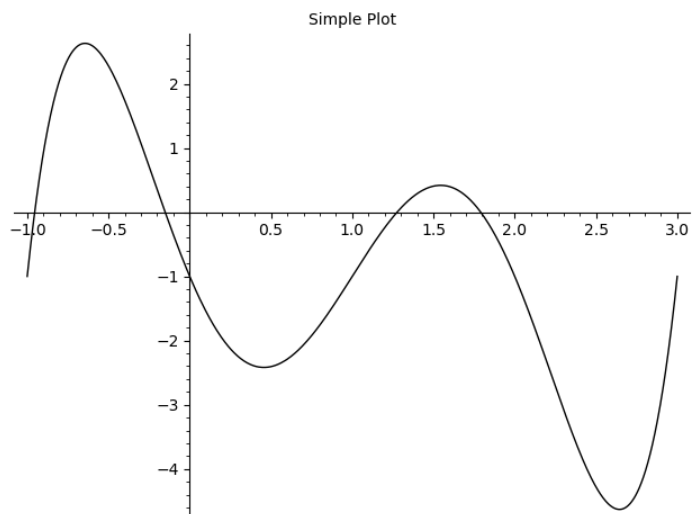
(c) Graph of $y = x^3 - 3x^2 + 2x$



(d) Graph of $y = x^2 (x - 1)^2$



(e) Graph of $y = -x^3 + 2x^2 + 5x - 30$



(f) Graph of $y = x^5 - 5x^4 + 5x^3 + 5x^2 - 6x - 1$

Figure 2: Graphs of functions by SageMath

3D Graphs

SageMath is a powerful open-source mathematics software system that offers a wide range of features, including 3D plotting capabilities. 3D plotting allows you to visualize mathematical functions and surfaces in a three-dimensional space.

It's a valuable tool to gain insights into complex mathematical relationships and data.

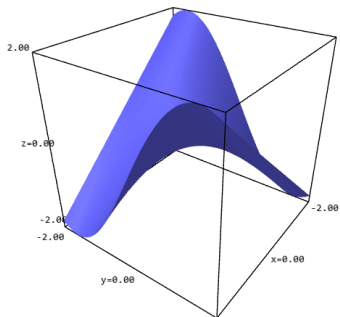
In this subsection, we'll demonstrate how to create a 3D plot of a simple function using SageMath, where we can use this as a starting point to explore and visualize our own 3D mathematical expressions. The SageMath code to plot 3D graph is given by:

```
1 # Define a mathematical function or surface log(x)*cos(15*x)
2 f(x, y) =
3
4
5 # Create a 3D plot of the function
6 plot3d(f, (x, x_0,x_1), (y,y_0,y_1))
7
```

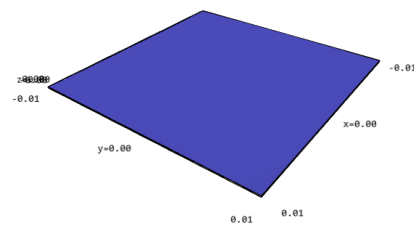
Figure 3: Sage code to plot in 3D

We now draw the graph of the following functions using the above code.

- $f(x, y) = x \sin(y)$,
- $(f(x, y) = x^2 - y^2$,
- $f(x, y) = \sqrt{x^2 - y^2}$,
- $f(x, y) = \sqrt{x^2 + y^2}$,
- $f(x, y) = (1 - x)^2 + 100 (y - x^2)^2$,
- $(f(x, y) = \frac{1}{9}x^2 - \frac{1}{4}y^2$

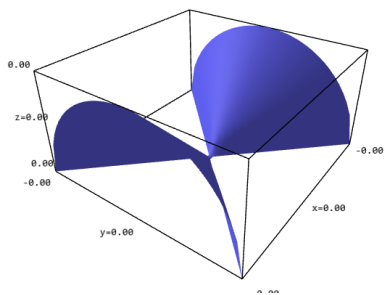


(a) 3D Graph of $f(x, y) = x \sin(y)$

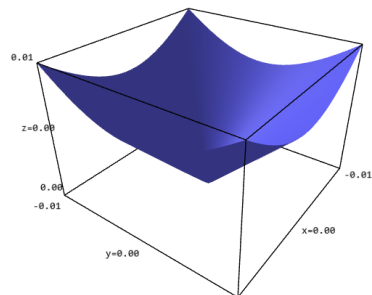


(b) 3D Graph of $(f(x, y) = x^2 - y^2$

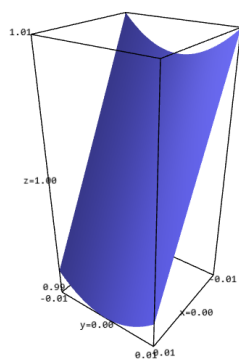
Figure 4: 3D Graphs of functions by SageMath



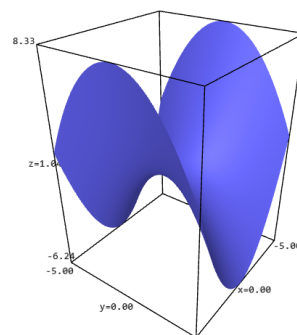
(a) 3D Graph of $f(x, y) = \sqrt{x^2 - y^2}$



(b) 3D Graph of $f(x, y) = \sqrt{x^2 + y^2}$



(a) 3D Graph of $f(x, y) = (1 - x)^2 + 100(y - x^2)^2$



(b) 3D Graph of $f(x, y) = \frac{1}{9}x^2 - \frac{1}{4}y^2$

Figure 6: 3D Graphs of functions by SageMath

Exploring Numerical Methods with SageMath

Having introduced the fundamental features of SageMath, we will delve into its practical application of performing numerical computations in various mathematical topics.

This exploration will include the following techniques and methods:

- 1. Root Bisection Method**
- 2. Linear Iterations**
- 3. Newton-Raphson Iteration Method**
- 4. Interpolation**
- 5. Newton's Forward Difference**
- 6. Newton's Backward Difference**
- 7. Binomial Series**
- 8. Trapezoidal Rule**
- 9. Simpson's Rule**

Throughout this journey, [SageMath](#) will serve as a valuable tool for hands-on experience in these numerical methods, enriching our understanding of these mathematical concepts, and their practical applications.

The Root Bisection Method

The root bisection method is a numerical technique for finding the root of a continuous function within a specified interval. It relies on the intermediate value theorem, which states that if the function values at the interval's endpoints have different signs, there exists at least one root within that interval. This method divides the interval into smaller sub-intervals, narrowing down the location of the root by repeatedly choosing the sub interval with endpoints of opposite signs and calculating the midpoint. It's a simple but relatively slow method, often referred to as the interval halving method, root-finding method, binary search method, or dichotomy method. In essence, it works by iteratively reducing the gap between positive and negative intervals until the root is approximated.

For a continuous function "f" defined on the closed interval $[a, b]$ with $f(a)$ and $f(b)$ having different signs, it follows from the intermediate theorem that $\exists x \in (a, b) \mid f(x) = 0$.

We follow the below procedure to get the solution for the continuous function:

For any continuous function $f(x)$,

- Find two points, say a and b such that $a < b$ and $f(a) \times f(b) < 0$
- Find the midpoint of a and b , say " c ", c is the root of the given function if $f(c) = 0$; else we follow the next step
- Divide the interval $[a, b]$ – If $f(c) \times f(a) < 0$, there exist a root between c and a , else if $f(c) \times f(b) < 0$, there exist a root between c and b
- Repeat above three steps until $f(c) = 0$.

SageMath, a powerful mathematical software system, provides an effective environment for implementing this method. Here's the code to solve the root bisection method.

```
1 # Define the function for which you want to find the root
2 def f(x):
3     return f(x)
4
5 # Bisection method
6 def bisection_method(f, a, b, eps, max_iter):
7     if f(a) * f(b) >= 0:
8         print("Bisection method may not converge as f(a) and f(b) have the same sign.")
9         return None
10
11     for i in range(max_iter):
12         c = (a + b) / 2
13         if abs(f(c)) < eps:
14             return c
15         if f(a) * f(c) < 0:
16             b = c
17         else:
18             a = c
19
20     print("Bisection method did not converge within the maximum number of iterations.")
21     return None
22
23 # Set the initial interval [a, b], tolerance (tol), and maximum number of iterations
24 a = 'lower limit'
25 b = 'upper limit'
26 epsilon = 1e-6
27 max_iterations = 'max no. of iteration'
28
29 # Call the bisection method function
30 root = bisection_method(f, a, b, epsilon, max_iterations)
31
32 #To round up to a n significant values as given
33 result = round(root, n)
34
35 if root is not None:
36     print(f"The approximate root found is {result}")
```

Figure 7: SageMath code for root bisection method

Here's an explanation of each part of the code:

- **Define the Function $f(x)$:** The code begins by defining a function $f(x)$ which represents the function for which we want to find the root.
- **Bisection Method Function `bisection_method`:** This function implements the Bisection Method. It takes the following parameters: `f`: the function for which you want to find the root, `a` and `b`: the initial interval $[a, b]$ where the root is searched, `eps`: the tolerance (how close the approximation needs to be to the real root). `max_iter`: the maximum number of iterations allowed. Then:
 - It first checks if `f(a)` and `f(b)` have the same sign. If they do, the method may not converge, and a warning is displayed.
 - It then enters a loop, repeatedly dividing the interval in half ($c = (a + b)/2$) and checking if the function value at `c` is close enough to zero. If it is, the method returns the approximate root. If not, it updates the interval $[a, b]$ based on the sign of $f(a)$ and $f(c)$.
 - If the method doesn't converge within the maximum number of iterations, it displays a warning.
- **Set Parameters:** we set the initial interval $[a, b]$, tolerance `epsilon`, and the maximum number of iterations `max_iterations`.
- **Call the Bisection Method:** the `bisection_method` function is called with the provided parameters, and it attempts to find the root of the function.
- **Display the Result:** if a root is found (i.e., if `root` is not `None`), it is rounded to three significant figures(which can be adjusted) and displayed as **"The approximate root found is result"**.

Solving Examples with the Code

Example 1

Use the method of bisection to find the possible root of equation $5x^2 + 11x - 17 = 0$. Correct to 3 s.f.

```
1 # Define the function for which you want to find the root
2 def f(x):
3     return 5*(x**2) + 11*x - 17
4
5 # Bisection method
6 def bisection_method(f, a, b, eps, max_iter):
7     if f(a) * f(b) >= 0:
8         print("Bisection method may not converge as f(a) and f(b) have the same sign.")
9         return None
10
11     for i in range(max_iter):
12         c = (a + b) / 2
13         if abs(f(c)) < eps:
14             return c
15         if f(a) * f(c) < 0:
16             b = c
17         else:
18             a = c
19
20     print("Bisection method did not converge within the maximum number of iterations.")
21     return None
22
23 # Set the initial interval [a, b], tolerance (tol), and maximum number of iterations
24 a = 0
25 b = 3
26 epsilon = 1e-6
27 max_iterations = 100
28
29 # Call the bisection method function
30 root = bisection_method(f, a, b, epsilon, max_iterations)
31
32 #To round up to a 3 significant values as given
33 result = round(root, 3)
34
35 if root is not None:
36     print(f"The approximate root found is {result}")
```

Language: Sage

Evaluate

Share

The approximate root found is 1.047

Help | Powered by SageMath

Figure 8: Solution to example 1.

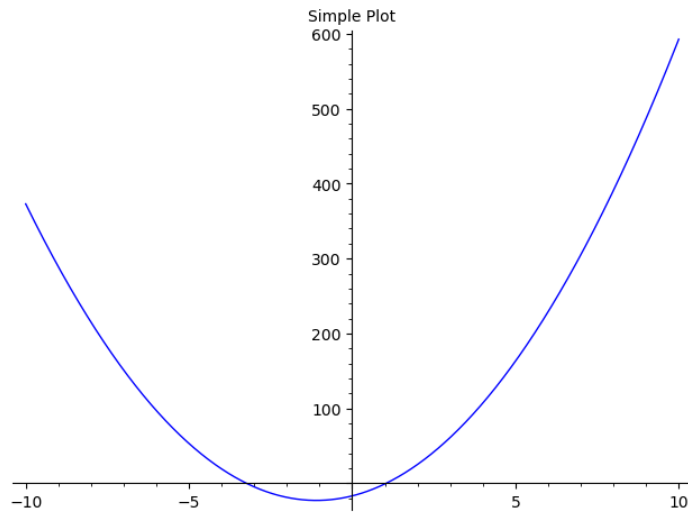


Figure 9: Graph of $5x^2 + 11x - 17 = 0$

Example 2

Locate the smallest positive root of the equation $xe^x = \cos(x)$

```

1 # Define the function for which you want to find the root
2 def f(x):
3     return x*exp(x) - cos(x)
4
5 # Bisection method
6 def bisection_method(f, a, b, eps, max_iter):
7     if f(a) * f(b) >= 0:
8         print("Bisection method may not converge as f(a) and f(b) have the same sign.")
9         return None
10
11     for i in range(max_iter):
12         c = (a + b) / 2
13         if abs(f(c)) < eps:
14             return c
15         if f(a) * f(c) < 0:
16             b = c
17         else:
18             a = c
19
20     print("Bisection method did not converge within the maximum number of iterations.")
21     return None
22
23 # Set the initial interval [a, b], tolerance (tol), and maximum number of iterations
24 a = 0
25 b = 3
26 epsilon = 1e-6
27 max_iterations = 100
28
29 # Call the bisection method function
30 root = bisection_method(f, a, b, epsilon, max_iterations)
31
32 # To round up to a 3 significant values as given
33 result = round(root, 3)
34
35 if root is not None:
36     print(f"The approximate root found is {result}")

```

Evaluate

Language: Sage

Share

The approximate root found is 0.518

Figure 10: Solution to example 2.

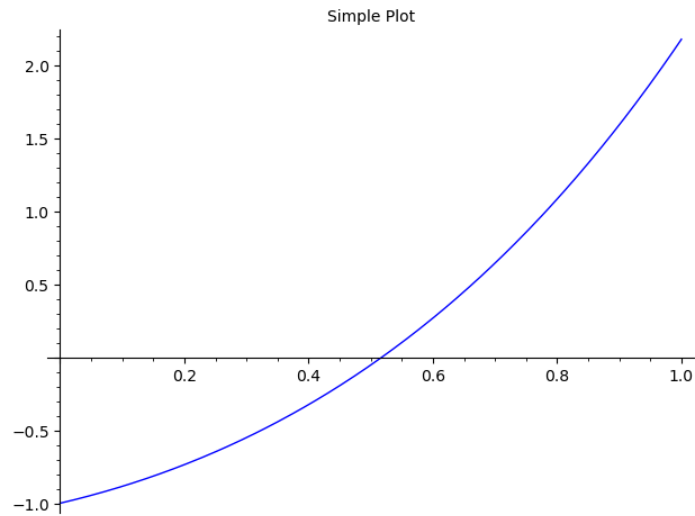


Figure 11: Graph of $xe^x = \cos(x)$

Example 3

Use Bisection Method to find a root of $f(x) = e^{-x} - x$ in the interval $[0.5, 0.8]$.

```

1  # Define the function for which you want to find the root
2  def f(x):
3      return e^(-x)-x
4
5  # Bisection method
6  def bisection_method(f, a, b, eps, max_iter):
7      if f(a) * f(b) >= 0:
8          print("Bisection method may not converge as f(a) and f(b) have the same sign.")
9          return None
10
11     for i in range(max_iter):
12         c = (a + b) / 2
13         if abs(f(c)) < eps:
14             return c
15         if f(a) * f(c) < 0:
16             b = c
17         else:
18             a = c
19
20     print("Bisection method did not converge within the maximum number of iterations.")
21     return None
22
23 # Set the initial interval [a, b], tolerance (tol), and maximum number of iterations
24 a = 0.5
25 b = 0.8
26 epsilon = 1e-6
27 max_iterations = 100
28
29 # Call the bisection method function
30 root = bisection_method(f, a, b, epsilon, max_iterations)
31
32 #To round up to a 3 significant values as given
33 result = round(root, 3)
34
35 if root is not None:
36     print(f"The approximate root found is {result}")

```

Language: Sage

Share

The approximate root found is 0.567

Help | Powered by SageMath

Figure 12: Solution to example 3.

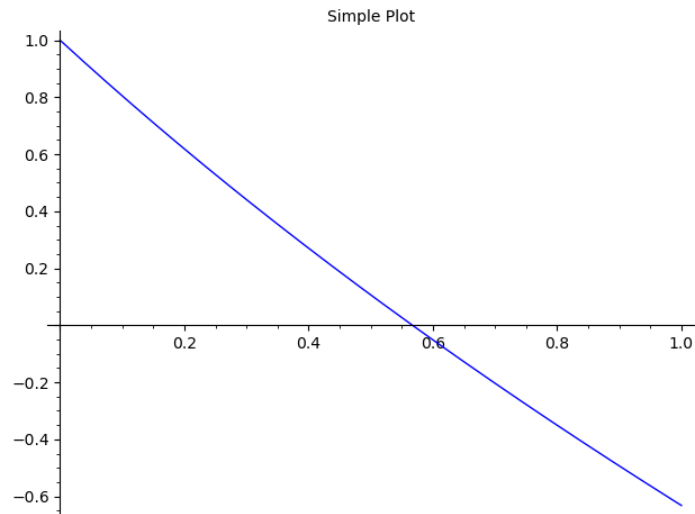


Figure 13: Graph of $f(x) = e^{-x} - x$

Example 4

Use the method of bisection to find the possible root of equation $x^3 - 6x^2 + 11x - 6$.

```

1  # Define the function for which you want to find the root
2  def f(x):
3      return x^3 - 6*(x^2) + 11*x - 6
4
5  # Bisection method
6  def bisection_method(f, a, b, eps, max_iter):
7      if f(a) * f(b) >= 0:
8          print("Bisection method may not converge as f(a) and f(b) have the same sign.")
9          return None
10
11     for i in range(max_iter):
12         c = (a + b) / 2
13         if abs(f(c)) < eps:
14             return c
15         if f(a) * f(c) < 0:
16             b = c
17         else:
18             a = c
19
20     print("Bisection method did not converge within the maximum number of iterations.")
21     return None
22
23 # Set the initial interval [a, b], tolerance (tol), and maximum number of iterations
24 a = 0
25 b = 6
26 epsilon = 1e-6
27 max_iterations = 100
28
29 # Call the bisection method function
30 root = bisection_method(f, a, b, epsilon, max_iterations)
31
32 #To round up to a 3 significant values as given
33 result = round(root, 3)
34
35 if root is not None:
36     print(f"The approximate root found is {result}")
37

```

Evaluate

Language: Sage

Share

The approximate root found is 3.0

Figure 14: Solution to example 3.

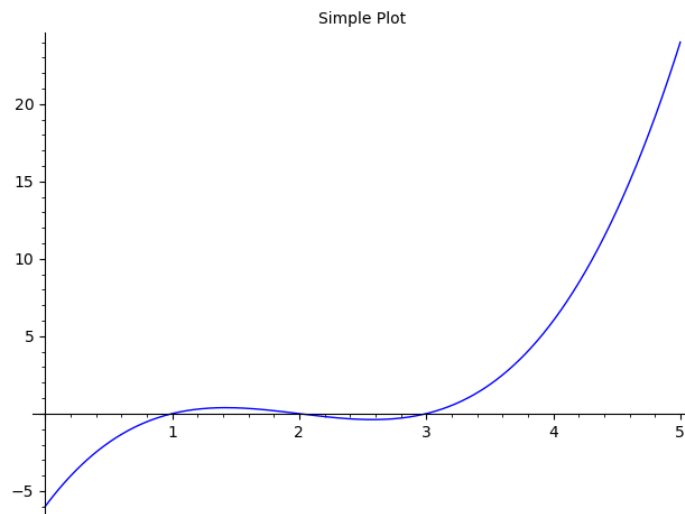


Figure 15: Graph of $x^3 - 6x^2 + 11x - 6$

Remark

Using SageMath for the root bisection method provides a convenient and efficient way to find solutions to equations and is particularly useful for solving equations with one variable. It's an essential tool for numerical analysis and problem-solving in various fields of science and engineering.

Linear Iteration

The linear iteration method is a numerical technique used to find the roots of a given equation $f(x) = 0$ by transforming it into the form $x = g(x)$. The basic idea is to start with an initial guess x_0 and then repeatedly apply the function $g(x)$ to get a sequence of values (x_1, x_2, \dots) that ideally converges to a solution of $f(x) = 0$.

The iteration formula is given by

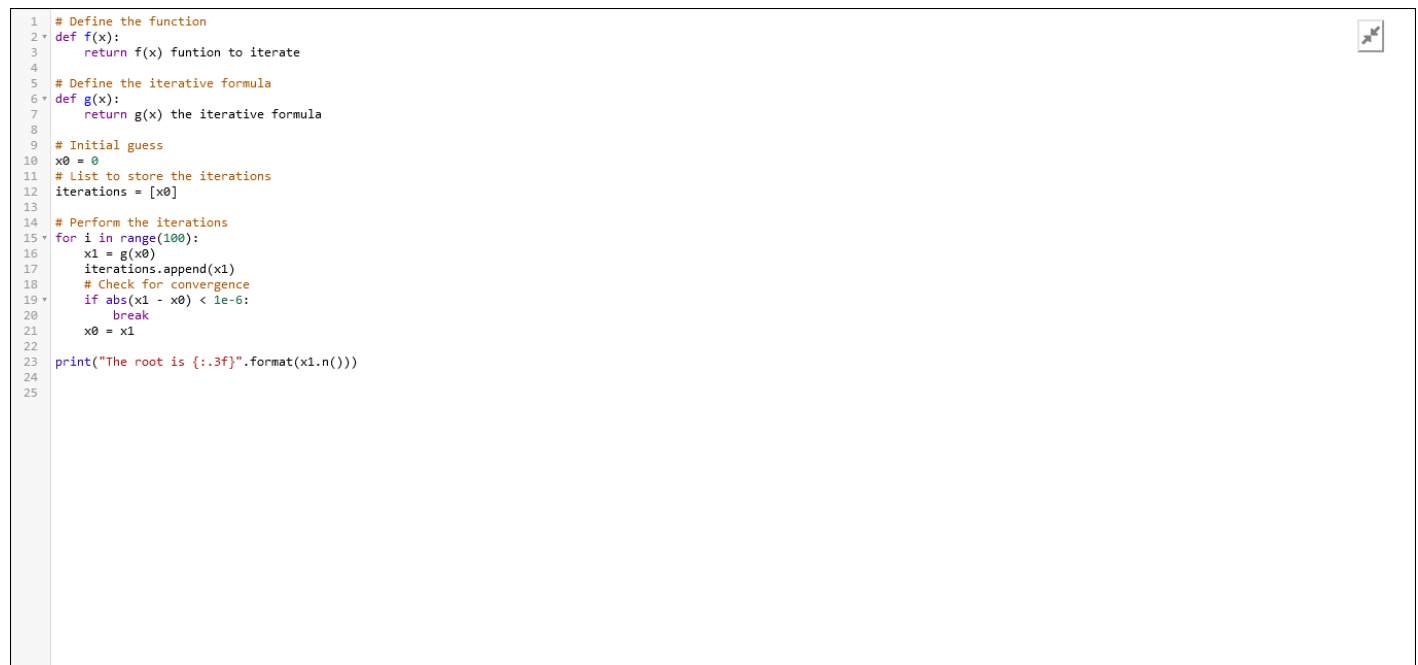
$$x_{n+1} = g(x_n),$$

where:

- x_n is the current approximation to the root,
- $x_n + 1$ is the next approximation obtained by applying the function $g(x)$ to x_n .

The iteration is repeated until a certain convergence criterion is met. Common criteria include reaching a specified number of iterations, achieving a desired level of accuracy, or when $|x_{n+1} - x_n|$ is less than a predetermined threshold.

We shall employ this numerical approach in SageMath to solve linear iteration problems, the code to do this is provided below and explanations follow immediately.

A screenshot of a SageMath code editor showing a script for linear iteration. The code is as follows:

```
1 # Define the function
2 def f(x):
3     return f(x) funtion to iterate
4
5 # Define the iterative formula
6 def g(x):
7     return g(x) the iterative formula
8
9 # Initial guess
10 x0 = 0
11 # List to store the iterations
12 iterations = [x0]
13
14 # Perform the iterations
15 for i in range(100):
16     x1 = g(x0)
17     iterations.append(x1)
18     # Check for convergence
19     if abs(x1 - x0) < 1e-6:
20         break
21     x0 = x1
22
23 print("The root is {:.3f}".format(x1.n()))
24
25
```

Figure 16: SageMath code to perform linear iteration

The provided code demonstrates the use of the linear iteration method to find the root of a equation $f(x)$. Let's break down the code:

- **f(x)**: Defines the function whose root has to be found.
- **g(x)**: Defines the iterative formula which is gotten from $f(x)$.
- **x0**: Sets the initial guess for the root.
- **Iteration Loop**:
 - **for i in range(100)**: Performs up to 100 iterations.

- **x1 = g(x0)**: Calculates the next iteration using the iterative formula.
- **iterations.append(x1)**: Appends the current iteration value to the list.
- **if abs(x1 - x0) < 1e-6**: Checks for convergence. If the difference between consecutive iterations is less than 1×10^{-6} , the loop breaks which stops the iteration and therefore implies that convergence has been reached.
- **print("The root is {:.3f}".format(x1.n()))**: Prints the root approximation with three decimal places. The result is printed using a formatted string. The `x1.n()` ensures that the numerical approximation of `x1` is produced as the root of the equation.

The code uses a loop to perform fixed-point iterations, updating the value of `x` until convergence is achieved or the maximum number of iterations is reached. The result is then printed, indicating the approximate root of the given equation.

Solving examples with the code

We shall now use the written SageMath code to solve iteration problems.

Example 1

Form an iterative formula to solve the equation $x^2 - 3x + 1 = 0$, and use it to find the root if $x_0 = 0$.

```

1 # Define the function
2 def f(x):
3     return x^2 - 3*x + 1
4
5 # Define the iterative formula
6 def g(x):
7     return (x^2 + 1)*(1/3)
8
9 # Initial guess
10 x0 = 0
11 # List to store the iterations
12 iterations = [x0]
13
14 # Perform the iterations
15 for i in range(100):
16     x1 = g(x0)
17     iterations.append(x1)
18     # Check for convergence
19     if abs(x1 - x0) < 1e-6:
20         break
21     x0 = x1
22
23 print("The root is {:.3f}".format(x1.n()))
24
25

```

Evaluate Language: Sage Share

The root is 0.382

[Help](#) | Powered by [SageMath](#)

Figure 17: Solution to example 1

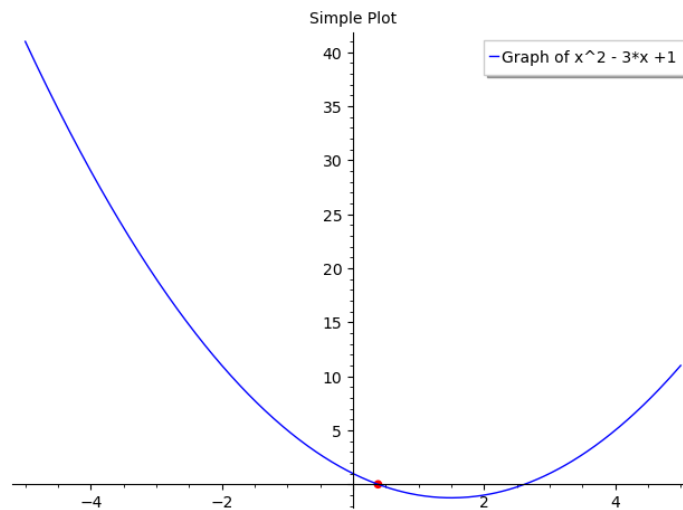


Figure 18: Graph of example 1

Example 2

Form an iterative formula to solve the equation $x^3 - 5x + 1 = 0$, and use it to find the root that lies between 0 and 1 correct to 3 s.f.

```

1  # Define the function
2  def f(x):
3      return x^3 - 5*x + 1
4
5  # Define the iterative formula
6  def g(x):
7      return (x^3 + 1)^(1/5)
8
9  # Initial guess
10 x0 = 0
11 # List to store the iterations
12 iterations = [x0]
13
14 # Perform the iterations
15 for i in range(100):
16     x1 = g(x0)
17     iterations.append(x1)
18     # Check for convergence
19     if abs(x1 - x0) < 1e-6:
20         break
21     x0 = x1
22
23 print("The root is {:.3f}".format(x1.n()))
24
25

```

Evaluate

Language: Sage ▼

Share

The root is 0.202

[Help](#) | Powered by [SageMath](#)

Figure 19: Solution to example 2

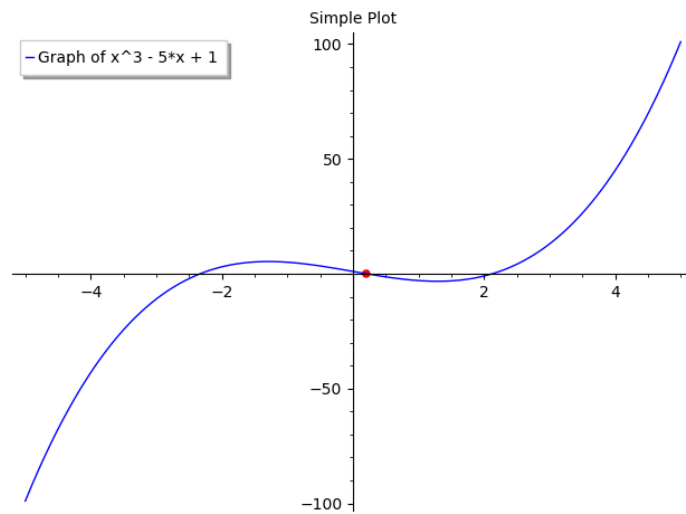


Figure 20: Graph of example 2

Example 3

Using linear iteration method, evaluate the root of $\ln(x) + 3x - 10.8074$.

```

1 # Define the function
2 def f(x):
3     return log(x) + 3*x - 10.8074
4
5 # Define the iterative formula
6 def g(x):
7     return (10.8074 - log(x))**(1/3)
8
9 # Initial guess
10 x0 = 3
11 # List to store the iterations
12 iterations = [x0]
13
14 # Perform the iterations
15 for i in range(100):
16     x1 = g(x0)
17     iterations.append(x1)
18     # Check for convergence
19     if abs(x1 - x0) < 1e-6:
20         break
21     x0 = x1
22
23 print("The root is {:.3f}".format(x1.n()))
24
25

```

Evaluate

Language: Sage

Share

The root is 3.213

Help | Powered by SageMath

Figure 21: Solution to example 3

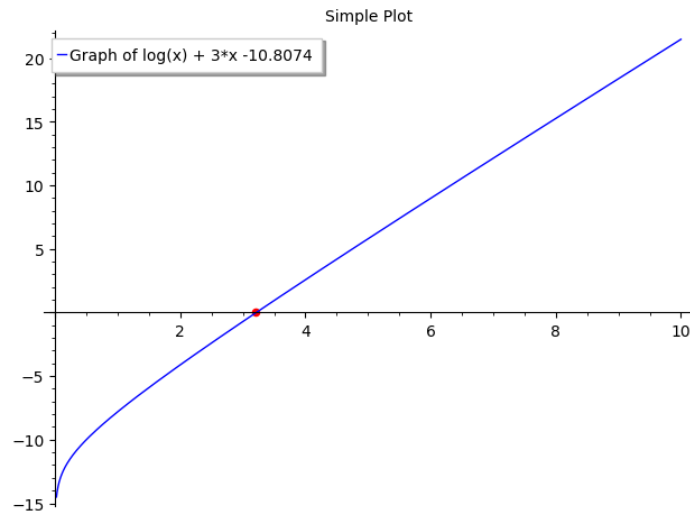


Figure 22: Graph of example 3

Example 4

Find the smallest negative root in magnitude of the equation $3x^4 + x^3 + 12x + 4 = 0$, using the method of linear iteration.

```

1  # Define the function
2  def f(x):
3      return 3*(x^4) + x^3 + 12*x + 4
4
5  # Define the iterative formula
6  def g(x):
7      return (3*(x^4) + x^3 + 4)*(-1/12)
8
9  # Initial guess
10 x0 = -1
11 # List to store the iterations
12 iterations = [x0]
13
14 # Perform the iterations
15 for i in range(100):
16     x1 = g(x0)
17     iterations.append(x1)
18     # Check for convergence
19     if abs(x1 - x0) < 1e-6:
20         break
21     x0 = x1
22
23 print("The root is {:.3f}".format(x1.n()))
24
25

```

Evaluate

Language: Sage

Share

The root is -0.333

[Help](#) | Powered by [SageMath](#)

Figure 23: Solution to example 4

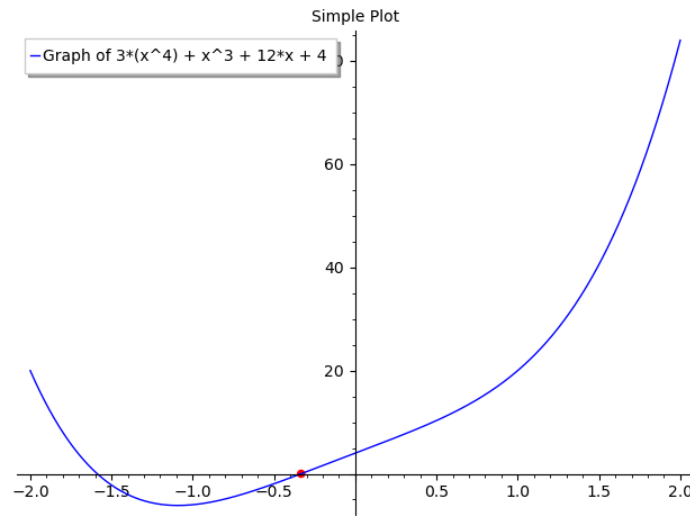


Figure 24: Graph of example 4

Remark

In conclusion, our exploration of the Linear Iteration Method using SageMath has provided valuable insights into the method's application. By combining theoretical discussions with practical examples, we have not only solved mathematical problems but also gained a deeper understanding of the principles underlying iterative methods.

The journey through this project has equipped us with valuable skills applicable across various domains, laying the foundation for further exploration in numerical analysis and computational mathematics.

Thank you!