Numerical Method Programming

Juan Camilo Anzola Gomez, Stiven Yepes Vanegas, Yhilmar Andrés Chaverra Castaño

November 22, 2021

1 Introduction

Numerical methods are a very effective way to solve problems with a high degree of complexity, which analytically would not have a solution. Also, thanks to the exponential growth in computing power in computers, it is possible for us to implement these numerical methods to obtain better results. As a final project of the numerical analysis subject, we have developed some numerical methods for solving different problems in the Python programming language.

2 How to use the program

The first step to use the program is to download the funete code from the repository. In order to run our program, you have to do the following process: Having Python installed, the version used is Python3, along with some additional libraries (Tkinter, numpy, sympy). What we will do is run the following command in the project directory:

Python ./Principal.py principal being the class with the initial interface.

The next thing we will see will be a screen with 3 categories of methods, we will see a list of buttons with the name of the corresponding method, what follows will be to click on the method we want to execute, a sub-window will appear which will have the necessary fields for calculate method and a button to calculate, the values will come out in the same interface.

Second option: the buttons of the main interface will be linked to their respective code on the Replit page, for later execution, each method will have input variables which we will change at will and execute the corresponding operations, the results appearing on the console.

3 Methods developed

3.1 Numerical solution of nonlinear equations

- **Incremental searches**

  program that finds an interval where f(x) has a change of sign using the method of incremental searches.
  Pseudocode:

```
input:
f, continuous function
x0, starting point
h, step
Nmax, maximum number of iterations

output:
a, left end of the interval
b, right end of interval
iter, number of iterations

//function
```

```
Incremental searches(f, x0, h, Nmax):

    // Initialization
    xant = x0
    fant = f (xant)
    xact = xant + h
    fact = f (xact)

    //Cycle
    for i = 1: Nmax
         if fant * fact <0
             break
         end if
         xant = xact
         fantastic = fact
         xact = xant + h
         fact = f (xact)
    end for

    // Delivery of results
    a = xant
    b = xact
    iter = i
```

- **Bisection**

  program that finds the solution to the equation f (x) = 0 in the interval [a, b] using the bisection method.
  Pseudocode:

```
input:
f, continuous function
a, right end of the initial interval
b, end end of end interval
tol, tolerance
Nmax, maximum number of iterations

output:
x, solution
iter, number of iterations
err, error

//function
bisection (f, a, b, tol, Nmax):

    // Initialization
    fa = f (a)
    pm = (a + b) / 2
    fpm = f (pm)
    E = 1000;
    cont = 1

    //Cycle
    while E> tol && cont <Nmax
       if fa * fpm <0
          b = pm;
       else
          a = pm
```

```
      end if
      p0 = pm
      pm = (a + b) / 2
      fpm = f (pm)
      E = abs (pm-p0)
      cont = cont + 1
   end while

   // Delivery of results
   x = pm
   iter = cont
   err = E
```

- **False rule**

  program that finds the solution to the equation f (x) = 0 in the interval [a, b] using the false rule method.
  Pseudocode:

```
Input:
f, continuous function
a, right end of the initial interval
b, end end of end interval
tol, tolerance
Nmax, maximum number of iterations

Output:
x, solution
iter, number of iterations
err, error

//function
false rule (f, a, b, tol, Nmax):

   // Initialization
   fa = f (a)
   fb = f (b)
   pm = (fb * a-fa * b) / (fb-fa)
   fpm = f (pm)
   E = 1000
   cont = 1

   //Cycle
   while E> tol && cont <Nmax
      if fa * fpm <0
         b = pm
      else
         a = pm
      end if
      p0 = pm
      pm = (f (b) * a-f (a) * b) / (f (b) -f (a))
      fpm = f (pm)
      E = abs (pm-p0)
      cont = cont + 1
   end while

   // Delivery of results
   x = pm
```

```
    iter = cont
    err = E
```

- **Fixed point**

  program that finds the solution to the equation f (x) = 0 by solving the analogous problem x = g (x) using the fixed point method.
  Pseudocode:

```
Input:
f, continuous function
g, continuous function
x0, initial approximation
tol, tolerance
Nmax, maximum number of iterations

output:
x, solution
iter, number of iterations
err, error

fixedpoint (g, x0, tol, Nmax):

    // Initialization
    xant = x0
    E = 1000
    cont = 0

    //Cycle
    while E> tol && cont <Nmax
       xact = g (xant)
       E = abs (xact-xant)
       cont = cont + 1
       xant = xact
    end while

    // Delivery of results
    x = xact
    iter = cont
    err = E
```

- **Newton**

  program that finds the solution to the equation f (x) = 0 using Newton's method.
  Pseudocode:

```
Input:
f, continuous function
f ', continuous function
x0, initial approximation
tol, tolerance
Nmax, maximum number of iterations

output:
x, solution
iter, number of iterations
err, error
```

```
//function
newton (f, df, x0, tol, Nmax):

    // Initialization
    xant = x0
    fant = f (xant)
    E = 1000
    cont = 0

    //Cycle
    while E> tol && cont <Nmax
       xact = xant-fant / (df (xant))
       fact = f (xact)
       E = abs (xact-xant)
       cont = cont + 1
       xant = xact
       fantastic = fact
    end while

    // Delivery of results
    x = xact
    iter = cont
    err = E
```

- **Secant**

  program that finds the solution to the equation f (x) = 0 using the secant method.
  Pseudocode:

```
Input:
f, continuous function
x0, initial approximation
x1, initial approximation
tol, tolerance
Nmax, maximum number of iterations

Output:
x, solution
iter, number of iterations
err, error

secant (f, x0, x1, tol, Nmax):

    // Initialization
    f0 = f (x0)
    f1 = f (x1)
    E = 1000
    cont = 1

    //Cycle
    while E> tol && cont <Nmax
       xact = x1-f1 * (x1-x0) / (f1-f0)
       fact = f (xact)
       E = abs (xact-x1)
       cont = cont + 1
       x0 = x1
       f0 = f1
       x1 = xact
```

```
        f1 = fact
    end while

    // Delivery of results
    x = xact
    iter = cont
    err = E
```

- **Multiple roots**

  program that finds the solution to the equation f (x) = 0 using the multiple roots method.
  Pseudocode:

```
Input:
f, continuous function
f ', continuous function
f '', continuous function
x0, initial approximation
tol, tolerance
Nmax, maximum number of iterations

Output:
x, solution
iter, number of iterations
err, error

multiple roots (f, df, d2f, x0, tol, Nmax):

    // Initialization
    xant = x0
    fant = f (xant)
    E = 1000
    cont = 0

    //Cycle
    while E> tol && cont <Nmax
       xact = xant-fant * df (xant) / ((df (xant)) ^ 2-fant * d2f (xant))
       fact = f (xact)
       E = abs (xact-xant)
       cont = cont + 1
       xant = xact
       fantastic = fact
    end while

    // Delivery of results
    x = xact
    iter = cont
    err = E
```

3.2 Solution of systems of linear equations

- **Gauss**

  program that finds the solution to the system Ax = b using the Gaussian elimination method.
  Pseudocode:

```
Input:
A, invertible matrix
```

```
b, constant vector

Output:
x, solution

//function
gausspl (A, b):

    // Initialization
    n = size (A, 1)
    M = [A b]

    We reduce the system
    for i = 1: n-1
         for j = i + 1: n
             if M (j, i) ~ = 0
                M (j, i: n + 1) = M (j, i: n + 1) - (M (j, i) / M (i, i)) * M (i, i: n + 1)
             end for
         end for
    end for

    // Delivery of results
    x = subst (M); // Backward substitution
```

- **LU factorization**

  Program that finds the solution to the system Ax = b and the LU factorization of A using the LU factorization method with simple Gaussian elimination.
  Pseudocode:

```
Input:
A, invertible matrix
b, constant vector

Output:
x, solution
L, factorization matrix L
U, U matrix of factorization

lu (A, b):

    // Initialization
    n = size (A, 1)
    L = eye (n)
    U = zeros (n)
    M = A

    //Factoring
    for i = 1: n-1
         for j = i + 1: n
             if M (j, i) ~ = 0
                L (j, i) = M (j, i) / M (i, i)
                M (j, i: n) = M (j, i: n) - (M (j, i) / M (i, i)) * M (i, i: n)
             end if
         end for
         U (i, i: n) = M (i, i: n)
         U (i + 1, i + 1: n) = M (i + 1, i + 1: n)
    end for
```

```
    U (n, n) = M (n, n)

    // Delivery of results
    z = subs ([L b])
    x = subs ([U z])
```

- **Doolitle**

  Program that finds the solution to the system Ax = b and the LU factorization of A using the Doolitle method.
  Pseudocode:

```
Input:
A, invertible matrix
b, constant vector

Output:
x, solution
L, factorization matrix L
U, U matrix of factorization

//function
Doolittle (A, b):

    // Initialization
    n = size (A, 1)
    L = eye (n)
    U = eye (n)

    //Factoring
    for i = 1: n-1
         for j = i: n
             U (i, j) = A (i, j) -dot (L (i, 1: i-1), U (1: i-1, j) ')
         end for
         for j = i + 1: n
             L (j, i) = (A (j, i) -dot (L (j, 1: i-1), U (1: i-1, i) ')) / U (i, i)
         end for
    end for
    U (n, n) = A (n, n) -dot (L (n, 1: n-1), U (1: n-1, n) ')

    // Delivery of results
    z = nounprgr ([L b])
    x = subregr ([U z])
```

- **Crout**

  Program that finds the solution to the system Ax = b and the LU factorization of A using the Crout method.
  Pseudocode:

```
Input:
A, invertible matrix
b, constant vector

Output:
x, solution
L, factorization matrix L
U, U matrix of factorization
```

```
//function
Crout (A, b):

    // Initialization
    n = size (A, 1)
    L = eye (n)
    U = eye (n)

    //Factoring
    for i = 1: n-1
         for j = i: n
             L (j, i) = A (j, i) -dot (L (j, 1: i-1), U (1: i-1, i) ')
         end for
         for j = i + 1: n
             U (i, j) = (A (i, j) -dot (L (i, 1: i-1), U (1: i-1, j) ')) / L (i, i)
         end for
    end for
    L (n, n) = A (n, n) -dot (L (n, 1: n-1), U (1: n-1, n) ')

    // Delivery of results
    z = nounprgr ([L b])
    x = subregr ([U z])
```

- **Cholesky**

  Program that finds the solution to the system Ax = b and the LU factorization of A using the Cholesky method.
  Pseudocode:

```
Input:
A, invertible matrix
b, constant vector

Output:
x, solution
L, factorization matrix L
U, U matrix of factorization

//function
Cholesky (A, b):

    // Initialization
    n = size (A, 1)
    L = eye (n)
    U = eye (n)

    //Factoring
    for i = 1: n-1
         L (i, i) = sqrt (A (i, i) -dot (L (i, 1: i-1), U (1: i-1, i) '))
         U (i, i) = L (i, i)
         for j = i + 1: n
             L (j, i) = (A (j, i) -dot (L (j, 1: i-1), U (1: i-1, i) ')) / U (i, i)
         end for
         for j = i + 1: n
             U (i, j) = (A (i, j) -dot (L (i, 1: i-1), U (1: i-1, j) ')) / L (i, i)
         end for
    end for
```

```
    L (n, n) = sqrt (A (n, n) -dot (L (n, 1: n-1), U (1: n-1, n) '))
    U (n, n) = L (n, n)

    // Delivery of results
    z = nounprgr ([L b])
    x = subregr ([U z])
```

- **Jacobi**

  Program that finds the solution to the system Ax = b using the Jacobi method.
  Pseudocode:

```
Input:
A, invertible matrix
b, constant vector
x0, initial approximation
tol, tolerance
Nmax, maximum number of iterations

Output:
x, solution
iter, number of iterations
err, error

jacobi (A, b, x0, tol, Nmax):

    // Initialization
    D = diag (diag (A))
    L = -tril (A) + D
    U = -triu (A) + D
    T = inv (D) * (L + U)
    C = inv (D) * b
    xant = x0
    E = 1000
    cont = 0

    //Cycle
    while E> tol && cont <Nmax
         xact = T * xant + C
         E = norm (xant-xact)
         xant = xact
         cont = cont + 1
    end while

    // Delivery of results
    x = xact
    iter = cont
    err = E
```

- **Gauss-Seidel** Program that finds the solution to the system Ax = b using the Gauss-Seidel method.
  Pseudocode:

```
Input:
A, invertible matrix
b, constant vector
x0, initial approximation
```

```
tol, tolerance
Nmax, maximum number of iterations

Output:
x, solution
iter, number of iterations
err, error

gseidel (A, b, x0, tol, Nmax):

    // Initialization
    D = diag (diag (A))
    L = -tril (A) + D
    U = -triu (A) + D
    T = inv (D-L) * U
    C = inv (D-L) * b
    xant = x0
    E = 1000
    cont = 0

    //Cycle
    while E> tol && cont <Nmax
         xact = T * xant + C
         E = norm (xant-xact)
         xant = xact
         cont = cont + 1
    end while

    // Delivery of results
    x = xact
    iter = cont
    err = E
```

3.3 Interpolation

- **Vandermonde**

  Program that finds the interpolating polynomial of the given data using the Vandermonde method.
  Pseudocode:

```
Input:
X, abscissa
And, ordered

Output:
Coef, coefficients of the polynomial

vandermonde (X, Y):

    // Initialization
    n = length (X)
    A = zeros (n)

    //Cycle
    for i = 1: n
         A (:, i) = (X. ^ (N-i)) '
    end for
```

```
// Delivery of results
Coef = A \ Y '
```

- **Divided differences**

  Program that finds the interpolating polynomial of the given data using the Divided differences method.
  Pseudocode:

```
Input:
X, abscissa
And, ordered

Output:
Coef, coefficients of Newton's polynomial

differences (X, Y):

    // Initialization
    n = length (X)
    A = zeros (n)

    //Cycle
    D (:, 1) = Y '
    for i = 2: n
         aux0 = D (i-1: n, i-1)
         aux = diff (aux0)
         aux2 = X (i: n) -X (1: n-i + 1)
         D (i: n, i) = aux / aux2 '
    end for

    // Delivery of results
    Coef = diag (D) '
```

- **Lagrange**

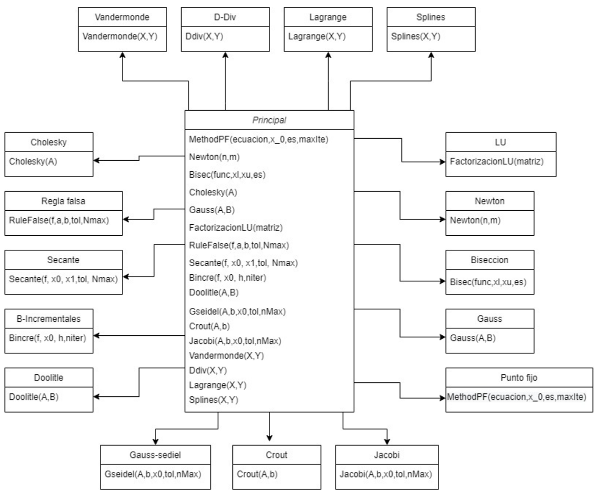
  Program that finds the interpolating polynomial of the given data using the Lagrange method.
  Pseudocode:

```
Input:
X, abscissa
And, ordered

Output:
L, Lagrange polynomials
Coef, coefficients of the interpolation polynomial

lagrange (X, Y):

    // Initialization
    n = length (X)
    L = zeros (n)

    //Cycle
    for i = 1: n
         aux0 = setdiff (X, X (i))
         aux = [1 -aux0 (1)]
```

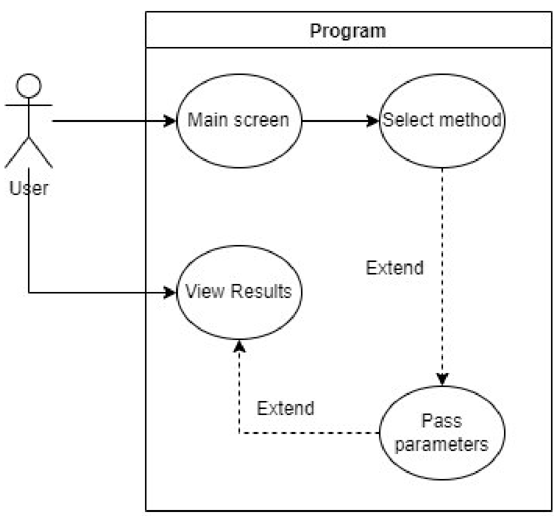
```
        for j = 2: n-1
            aux = conv (aux, [1 -aux0 (j)])
        end for
        L (i,:) = aux / polyval (aux, X (i))
    end for

    // Delivery of results
    Coef = Y * L
```

4 Diagrams

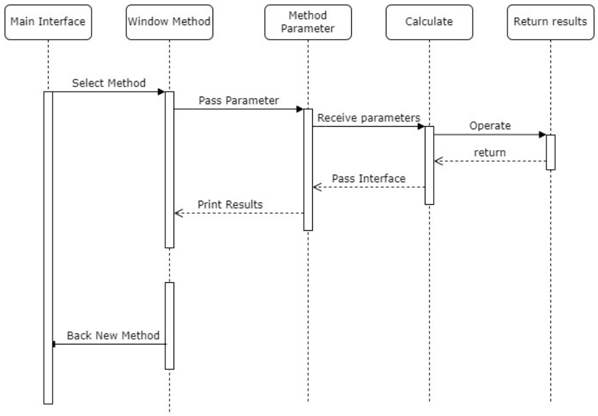
4.1 class diagram



4.2 use case diagram



4.3 sequence diagram



5 Conclusions

To carry out this project we have chosen the Python programming language, since it has different benefits compared to other programming languages. the first one is the simplicity in some aspects of the code, since this is a language interpreted with dynamic types. the second is the ease of working with mathematical functions using specialized libraries. and the third reason is the large amount of documentation that exists about the different numerical methods implemented in Python. the limitations or problems we had were related to the graphical interface, since there was no experience in the team to work with it and an accelerated learning curve was required to carry out the project.