THIRD FOLLOWAGE

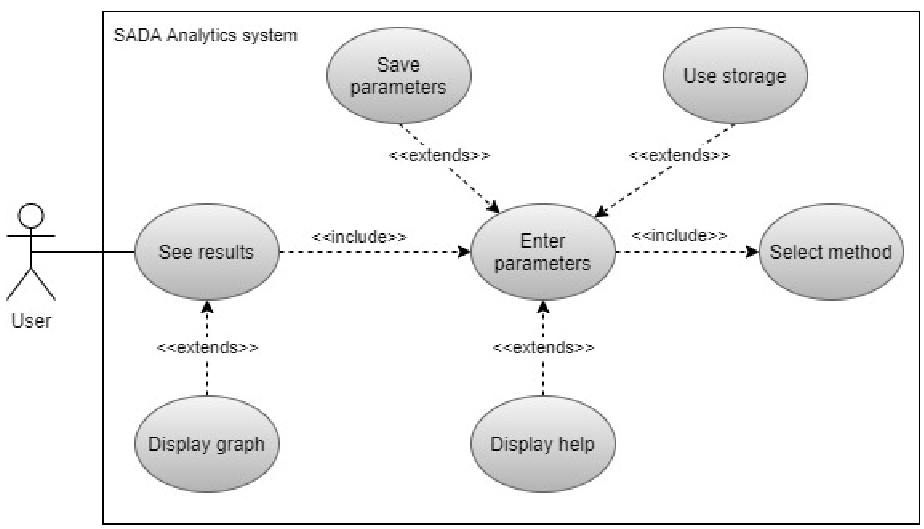
Daniel Felipe Gómez Martínez
César Andrés García Posada
Juan Sebastian Pérez Salazar
Yhoan Alejandro Guzmán García

November 24 2020

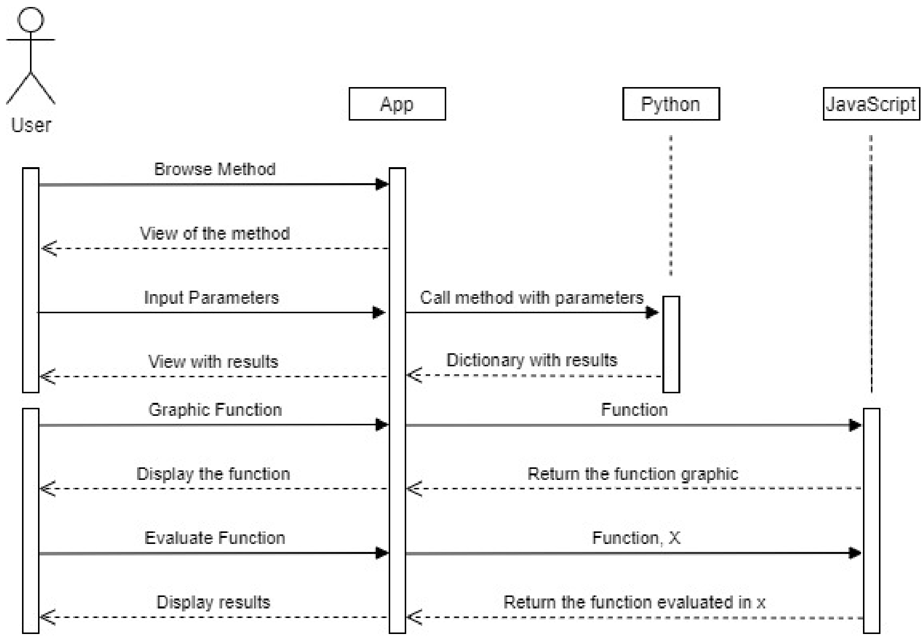
Subject: Numerical Analysis
Professor in charge: Edwar Samir Posada Murillo
Semester: 6th
System name (project): SADA ANALYTICS
repository where we will work:
SADA_ANALYTICS

1 Diagrams

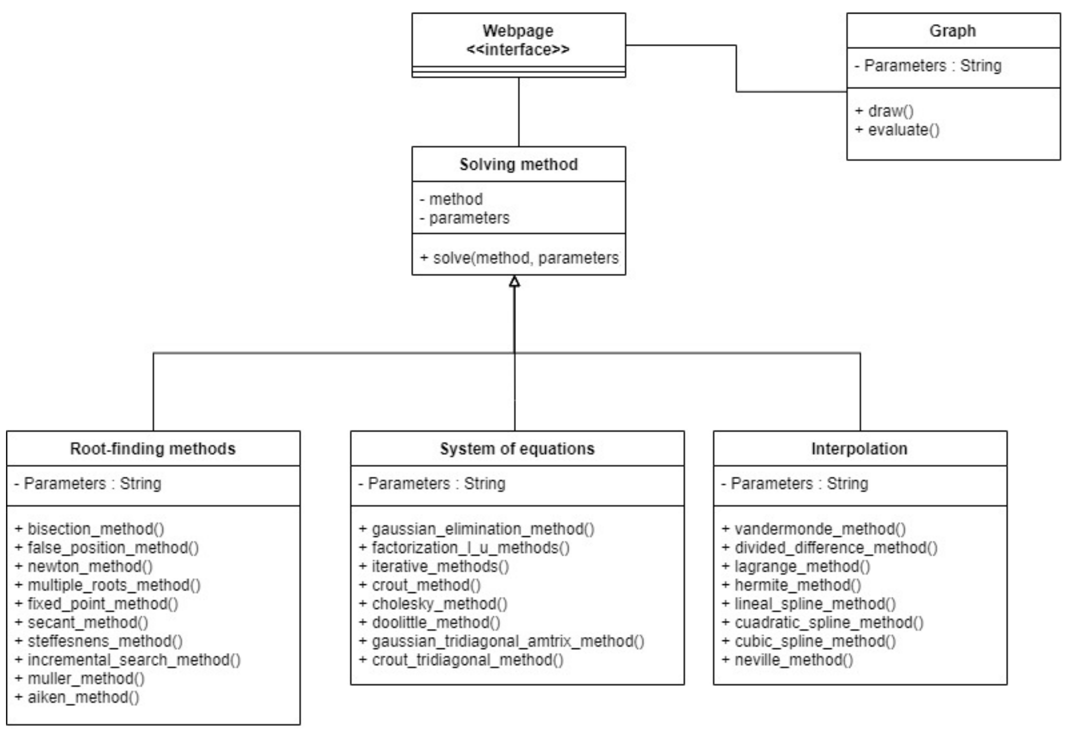
- Use Case diagram



- Sequence diagram



- Class diagram



2 Pseudocodes

- Incremental search

```
Input: function f(x), float x_0, float delta, int iterations
Output: solution vector results

begin incrementalSearch

        if (delta or iterations or x_0 is not a valid numbers) then
                break;
        array results
        float previous_x <- x_0
        float current_x <- previous_x + delta
        float previous_f <- function(previous_x)
        float current_f <- function(current_x)
        int count <- 0
        while (count < iterations) do
                if (current_f * previous_f < 0) then
                        array iteration <- [previous_x,current_x]
                        results[count] <- iteration
                previous_x <- current_x
                current_x <- current_x + delta
                previous_f <- current_f
                current_f <- function(current_x)
                count <- count + 1
        end while
        return results

end incrementalSearch
```

- False position

```
Input: function f(x), float a, float b, float tolerance,
int max_n_iterations
Output: result table results

begin falsePosition

    if(f(x) is not valid function)
        break;
    if(a or b or tolerance or max_n_iterations are not valid numbers)
        break;
    if(tolerance < 0)
        break;
    if(iterations < 1)
```

```
        break;

    array results

    float f_a <- f(a)
    float f_b <- f(b)
    float middle_point <- (a + b)/2
    float f_middle_point <- f(middle_point)
    float error <- MAXIMUM FLOAT VALUE
    int iterations_counter <- 1

    results[iterations_counter] <- [iterations_counter,
    a, middle_point, b, f_middle_point, "N/A"]

    float p_0

    while ((error > tolerance) and (iterations_counter < max_n_iterations))
        iterations_counter <- iterations_counter + 1
        if(f_a * f_b < 0):
                b <- middle_point
        else
                a <- middle_point

        p_0 <- middle_point
        middle_point <- (f(b)*a - f(a)*b)/(f(b) - f(a))
        f_middle_point <- f(middle_point)
        error <- |middle_point - p_0|
        results[iterations_counter] <- [iterations_counter, a,
        middle_point, b, f_middle_point, error]
    end while

    return results

end falsePosition
```

- Fixed point

```
Input: function f(x), function g(x), float initial_x, float tolerance,
int iterations
Output: result table results

begin fixedPoint

    if(f(x) and g(x) are not valid functions)
        break;
    if(initial_x or tolerance or iterations are not valid numbers)
        break;
```

```
    if(tolerance is lees than 0)
        break;
    if(iterations is less than 1)
        break;

    array results

    float current_x
    int iter_count <- 0
    float g_x <- g(initial_x)
    float f_x <- f(initial_x)
    float previous_x <- initial_x
    float error <- MAXIMUM FLOAT VALUE
    results[iter_count] <- [iter_count, initial_x, g_x, f_x, "N/A"]

        while iter_count < iterations and error > tolerance do:
            iter_count <- iter_count + 1
            current_x <- g_x
            g_x <- g(current_x)
            f_x <- f(current_x)
            error <- |previous_x - current_x|
            previous_x <- current_x
            results[iter_count] <- [iter_count, current_x, g_x, f_x, error]
        end while

    return results

end fixedPoint
```

- Bisection

```
Input: function f(x), float a, float b
Output: solution vector results

begin bisection:
    array results
    f_a <- function(a)
    f_b <- function(b)
    if (f_a * f_b >= 0) then
        return 0
    else:
        array aux
        mp <- (a + b)/2
        f_mp <- (function(mp)
                aux <- [a,mp,b]
        results add(aux)
        cont <- 1
```

```
        while (cont <= 2) do
            if (f_a * f_mp < 0) then
                b <- mp
            else:
                a <- mp
            p_0 <- mp
            mp <- (a + b)/2
            f_mp <- function(mp)
            cont <- cont + 1
            aux <- [a,mp,b]
                        results add(aux)
        return results

end Bisection
```

- Newton

```
Input: function f(x), function df(x), float initial_x, float tolerance,
int max_n_iterations
Output: result table results

begin newton

    if(f(x) or df(x) are not valid function)
        break;
    if(initial_x or tolerance or max_n_iterations are not valid number)
        break;
    if(tolerance < 0)
        break;
    if(iterations < 1)
        break;

    array results

    float previous_x <- initial_x
    float previous_f <- f(previous_x)
    float error <- MAXIMUM FLOAT VALUE
    int iterations_counter <- 0

    results[iterations_counter] <- [iterations_counter, previous_x,
    previous_f, "N/A"]

    float current_x, current_f, previous_df

    while ((error > tolerance) and (iterations_counter < max_n_iterations))
        iterations_counter <- iterations_counter + 1
        previous_df <- df(previous_x)
```

```
        if(previous_df == 0):
            break;

        current_x <- previous_x - (previous_f/previous_df)
        current_f <- f(current_x)
        error <- |current_x - previous_x|
        previous_x <- current_x
        previous_f <- current_f
        results[iterations_counter] <- [iterations_counter,
        previous_x, previous_f, error]
    end while

    return results

end newton
```

- Secant

```
Input: function f(x), float x0, float x1, float tolerance, int iterations
Output: result table results

begin secant

    if(f(x) is not a valid function)
        break;
    if(x0 or x1 or tolerance or iterations are not valid numbers)
        break;
    if(tolerance is lees than 0)
        break;
    if(iterations is less than 1)
        break;

    array results

    int iter_count <- 0
    float g_x <- g(initial_x)
    float f_x <- f(initial_x)
    float error <- MAXIMUM FLOAT VALUE
    results[iter_count] <- [iter_count, x0, f(x0), "N/A"]
    iter_count <- iter_count + 1
    results[iter_count] <- [iter_count, x1, f(x1), "N/A"]
    float previous_x <- x1
    float second_previous_x <- x0
    float current_x

        while iter_count < iterations and error > tolerance do:
```

```
            iter_count <- iter_count + 1
            current_x <- previous_x - ((f(previous_x)*
                (previous_x - second_previous_x))/(f(previous_x) -
                f(second_previous_x)))
            error <- |current_x - previous_x|
            results[iter_count] <- [iter_count, current_x, f(current_x), error]
            second_previous_x <- previous_x
            previous_x <- current_x
        end while

    return results

end secant
```

- Multiple roots

```
Input: function f(x), function df(x), function d2f(), float initial_x,
float tolerance, int max_n_iterations
Output: result table results

begin multipleRoots

    if(f(x) or df(x) or d2f(x) are not valid function)
        break;
    if(initial_x or tolerance or max_n_iterations are not valid number)
        break;
    if(tolerance < 0)
        break;
    if(iterations < 1)
        break;

    array results

    float previous_x <- initial_x
    float previous_f <- f(previous_x)
    float error <- MAXIMUM FLOAT VALUE
    int iterations_counter <- 0

    results[iterations_counter] <- [iterations_counter,
        previous_x, previous_f, "N/A"]

    float current_x, current_f, previous_df, previous_d2f

    while ((error > tolerance) and (iterations_counter < max_n_iterations))
        iterations_counter <- iterations_counter + 1
        previous_df <- df(previous_x)
        previous_d2f <- d2f(previous_x)
```

```
        current_x <- previous_x - ((previous_f*previous_df)/
            (previous_df^2 - previous_f*previous_d2f))
        current_f <- f(current_x)
        error <- |current_x - previous_x|
        previous_x <- current_x
        previous_f <- current_f
        results[iterations_counter] <- [iterations_counter, previous_x,
            previous_f, error]
    end while

    return results

end multipleRoots
```

- Simple gaussian method

```
Input: Augmented n x n+1 matrix Augmented_matrix
Output: square nxn matrix A,colum vector b, solution array x with steps

begin simpleGaussianMethod

    auxialiry_matrix <- Augmented_matrix
    for i from 0 to n-1 do
        pivot_number = auxialiry_matrix[0][0]        --> l x l+1 matrix
        if (pivot_number = 0) then
            for j from 0 to l-1 do
                if (auxialiry_matrix[j][0] = 0) then
                    switch auxialiry_amtrix[j][0] and auxialiry_matrix[0][0]
        if (pivot_number = 0) and (i = n-2) then
            break
        fj <- auxialiry_matrix[0]
        column_vector <- columnFrompivotnumber(auxialiry_matrix)
        multiplier <- column_vector/pivot_number
        fi <- auxiliary_matrix[1:]
        fi <- fi - (multiplier*fj)
        if (i = 0) then
            Augmented_matrix[i+1:] <- fi
        else:
            Augmented_matrix <- complitFirstColumnWithZeros(fi)
        auxiliary_matrix <- cutFisrtRowAndFisrtColumn(fi)
        solution_array[i+1] <- Augmented_matrix
    matrix_A <- deleteLastColumn(Augmented_matrix)
    vector_b <- getLastColumn(Augmented_matrix)
    matrix_A,vector_b,solution_array

end simpleGaussianMethod
```

- Partial gaussian method

```
Input: Augmented n x n+1 matrix Augmented_matrix
Output: square nxn matrix A,colum vector b, solution array x with steps

begin partialGaussianMethod

        auxialiry_matrix <- Augmented_matrix
        for i from 0 to n-1 do
                pivot_number <- auxialiry_matrix[0][0]
--> l x l+1 matrix
                pivot_column <- getFirstColumn(auxialiry_matrix)
                pivot_column <- absoluteValueInColumn(pivot_column)
                pos_max_pivot <- getIndexMaxValueFromColumn(pivot_column)
                if (pos_max_pivot != 0) then
                        switchColmn auxialiry_matrix[0][0] and
                            auxialiry_matrix[pos_max_pivot][0]
                        switchColmn Augmented_matrix[0][0] and
                            Augmented_matrix[pos_max_pivot][0]
                if (pivot_number = 0) and (i = n-2) then
                        break
                fj <- auxialiry_matrix[0]
                column_vector <- columnFrompivotnumber(auxialiry_matrix)
                multiplier <- column_vector/pivot_number
        fi <- auxiliary_matrix[1:]
        fi <- fi - (multiplier*fj)
        if (i = 0) then
                Augmented_matrix[i+1:] <- fi
        else:
            Augmented_matrix <- complitFirstColumnWithZeros(fi)
        auxiliary_matrix <- cutFisrtRowAndFisrtColumn(fi)
        solution_array[i+1] <- Augmented_matrix
    matrix_A <- deleteLastColumn(Augmented_matrix)
    vector_b <- getLastColumn(Augmented_matrix)
    matrix_A, vector_b, solution_array

end simpleGaussianMethod
```

- Total gaussian method

```
Input: Augmented n x n+1 matrix Augmented_matrix
Output: square nxn matrix A,colum vector b, solution array x with steps

begin totalGaussianMethod

        auxialiry_matrix <- Augmented_matrix
        for i from 0 to n-1 do
```

```
                sub_matrix <- deleteLastColumn(auxialiry_matrix)
                    --> l x l+1 matrix
                pivot_number <- sub_matrix[0][0]
                pos_max_pivot <- 0
        row <- 0
        for j from 0 to l-1 do
                        pivot_column <- getFirstColumn(sub_matrix[j])
                        pivot_column <- absoluteValueInColumn(pivot_column)
                        temporal_max_pivot <-
                            getMaxValueFromRow(pivot_column)
                    temporal_pos_max_pivot <-
                        getIndexMaxValueFromColumn(pivot_column)
                    if (pivot_number < temporal_max_pivot) then
                        pivot_number <- temporal_max_pivot
                pos_max_pivot <- temporal_pos_max_pivot
                row <- j
        if (row != 0) then
                switchRow auxialiry_matrix[0] and
                    auxialiry_matrix[row]
                switchRow Augmented_matrix[0] and
                    Augmented_matrix[i+row]
                if (pos_max_pivot != 0) then
                        switchColmn auxialiry_matrix[0][0] and
                            auxialiry_matrix[pos_max_pivot][0]
                        switchColmn Augmented_matrix[0][0] and
                            Augmented_matrix[pos_max_pivot][0]
                if (pivot_number = 0) and (i = n-2) then
                        break
                fj <- auxialiry_matrix[0]
                column_vector <- columnFrompivotnumber(auxialiry_matrix)
                multiplier <- column_vector/pivot_number
        fi <- auxiliary_matrix[1:]
        fi <- fi - (multiplier*fj)
        if (i = 0) then
                Augmented_matrix[i+1:] <- fi
        else:
            Augmented_matrix <- complitFirstColumnWithZeros(fi)
        auxiliary_matrix <- cutFisrtRowAndFisrtColumn(fi)
        solution_array[i+1] <- Augmented_matrix
    matrix_A <- deleteLastColumn(Augmented_matrix)
    vector_b <- getLastColumn(Augmented_matrix)
    matrix_A, vector_b, solution_array

end totalGaussianMethod
```

- LU with simple gaussian

```
Input: Augmented n x n+1 matrix Augmented_matrix
Output: square nxn matrix A,colum vector b, solution array x with steps

begin LuSimpleMethod

        auxialiry_matrix <- Augmented_matrix
        for i from 0 to n-1 do
                pivot_number = auxialiry_matrix[0][0]
                        --> l x l+1 matrix
                if (pivot_number = 0) then
                        for j from 0 to l-1 do
                                if (auxialiry_matrix[j][0] = 0) then
                                        switch auxialiry_amtrix[j][0] and
                                                auxialiry_matrix[0][0]
                if (pivot_number = 0) and (i = n-2) then
                        break
                fj <- auxialiry_matrix[0]
                column_vector <- columnFrompivotnumber(auxialiry_matrix)
                multiplier <- column_vector/pivot_number
        fi <- auxiliary_matrix[1:]
        fi <- fi - (multiplier*fj)
        if (i = 0) then
                Augmented_matrix[i+1:] <- fi
        else:
            Augmented_matrix <- complitFirstColumnWithZeros(fi)
        auxiliary_matrix <- cutFisrtRowAndFisrtColumn(fi)
        solution_array[i+1] <- Augmented_matrix
        solution_array_l[i+1] <- triangular_boton(Augmented_matrix)
        solution_array_u[i+1] <- triangular_top(Augmented_matrix)
    matrix_A <- deleteLastColumn(Augmented_matrix)
    vector_b <- getLastColumn(Augmented_matrix)
    matrix_A,vector_b,solution_array,solution_array_u,solution_array_l

end LuSimpleMethod
```

- Lu with partial pivoting

```
Input: matrix A, array b
Output: array x, matrix L, matrix U, matrix P

begin partial_lu

        int n <- A.size()
        matrix u <- zeros_matrix(n,n)
        matrix l <- identity_matrix(n)
        matrix p <- identity_matrix(n)
```

```
        for (int k from 0 until n)
                A, p <- search_bigger_and_swap(A, n, k, p)

                for (int i form k + 1 until n)
                        float mult<- A[i][k] / A[k][k]
                        l[i][k] <- mult

                        for (int j from k until n)
                                A[i][j] <- A[i][j] - mult * A[k][j]
                        end for
                end for

                for (int i from 0 until n)
                        u[k][i] <- A[k][i]
                end for
        end for

        matrix pb <- matmul(p, b)
                // matmul is a function that calculate the product
                between matrix
        array z <- solution(l, pb)
                // solution is a function that solve systems of equations
        array x <- solution(u, z)

        return x
end partial_lu

begin search_bigger_and_swap(matrix Ab, int n, int i, matrix p)
        int row = i

    for (int j from i + 1 until n)
        if (absolute_value(Ab[row][i]) < absolute_value(Ab[j][i]))
            row <- j
                end if
    end for

    array temp <- Ab[i]
    array aux <- p[i]
    Ab[i] <- Ab[row]
    p[i] <- p[row]
    Ab[row] <- temp
    p[row] <- aux

    return Ab, p

end partial_lu
```

- Doolittle method

```
matrix_function.soltion: Method in matrix_function that
        do a progressive or backward substitutionv to find an array

Input: matrix A, array b, int size
Output: solution vector results

begin doolittle

L = identityMatrix(size)
U = identityMatrix(size)

int count <- 0
while (count < size) do
        int count2 <- count
        while (count2 < size) do
                float sum <- 0
                int count3 <- 0
                while (count3 < count) do
                        sum <- sum + (L[count][count3]*U[count3][count2])
                end while
                U[count][count2] <- A[count][count2]-sum
        while (count2 < size) do
                if (count == count2):
                        L[conut][count] -<- 1
                else:
                        sum <- 0
                        while (count3 < count) do
                                sum <- sum + (L[count2][count3]*
                                    U[count3][count])
                        end while
                        L[count2][count] <- ((A[count2][count]-sum)/
                            U[count][count]
z = array(matrix_function.soltion(L,b))
x = matrix_function.soltion(U,z)

array sol
int count <- 0
while (count < size(x)) do
        sol[count] <- x[i]
return sol

end doolittle
```

- Crout

```
Input: matrix A, matrix b
Output: array x, matrix L, matrix U

int n <- A.size()
matrix L <- identity_matrix(n)
matrix U <- identity_matrix(n)

for (int i from 0 until n)
        for (int k from i until n)
                float sum <- 0
                for (int j from 0 until i)
                        sum <- sum + (L[k][j] * U[j][i])
                end for

                L[k][i] <- A[k][i] - sum
        end for

        for (int k from i until n)
                if (i = k)
                        U[i][i] <- 1
                else
                        float sum <- 0
                        for (int j from 0 until i)
                                sum <- sum + (L[i][j] * U[j][k])
                        end for

                        U[i][k] <- ((A[i][k] - sum)/L[i][i])
                end if
        end for
end for

array z <- solution(L, b) // solution is a function
        that solve systems of equations
array x <- solution(U, z)

return x, L, U
```

- Cholesky

```
Input: Matrix A, vector B
Output: Steps of making the cholesky factorization
    and the answer to the system

if(determinant(A) is 0)
        return error detrminant equals 0
```

```
  n = lenght A
  L,U = matrix of nxn filled of ceros
  for k=0:n
    sum1=0;
    for p=1:k
        sum1=sum1+L(k,p)*U(p,k);
    end for
    L(k,k)*U(k,k)=A(k,k) - sum1;

    for i=k+1:n
        sum2= 0;
        for p=1:k
            suma2= suma2+ L(i,p)*U(p,k);
        end for
        L(i,k) = (A(i,k)-suma2)/U(k,k);
    end for

    for j=k+1:n
        sum3= 0;
        for p = 1:k-1
            sum3=sum3+ L(k,p)*U(p,j);
        end for
        U(k,j) = (A(k,j)- sum3)/L(k,k);
    end for
  end for
  return L,U
end
```

- Jacobi

```
Input: matrix l, matrix d, matrix u, array b,
        array x0, float tol, int n_max
Output: int iter, array x, float E

begin jacobiMethod
        matrix T = dot_product(inverse_matrix(d),(l + u))
        matrix C = refact_matrix(inverse_matrix(
                time_matrix(inverse_matrix(d),b)),(b.size,1))
        float E <- infinity_value()
        array xant <- xo.transpose()
        int cont <- 0

        array values, array normalized_eigenvectors <- eigen_values(T)
        float spectral_radius <- maximum_value(absolute_value(values))

        if (spectral_radius > 1)
```

```
            return ERROR
        end if

        while ((E > tol) and (cont < nmax))
            matrix xact <- dot_product(T, xant) + C
            E <- norm2(xant - xact)
            xant <- xact
            cont <- cont + 1
        end while
        cont, E, xant.transpose()[0]
end
```

- Gauss-Seidel

```
Input: matrix l, matrix d, matrix u, array b,
    array x0, float tol, int n_max
Output: int iter, array x, float E

begin gaussSeidel

    matrix T <- dot_product(inverse_matrix(d - l), u)
    matrix C <- dot_product(inverse_matrix(d - l), b.transpose())
    float E <- infinity_value()
    array xant <- xo.transpose()
    int cont <- 0

    array values, array normalized_eigenvectors <- eigen_values(T)
    float spectral_radius <- maximum_value(absolute_value(values))

    if (spectral_radius > 1)
        return ERROR
    end if

    while ((E > tol) and (cont < nmax))
        matrix xact <- dot_product(T, xant) + C
        E <- norm2(xant - xact)
        xant <- xact
        cont <- cont + 1
    end while

    return cont, E, xant.transpose()[0]

end gaussSeidel
```

- SOR

```
Input: matrix l, matrix d, matrix u, array b,
```

```
        array x0, float tol, int n_max, int w
Output: matrix xact, array_steps E

begin sorMethod

        matrix T <- dot_product(inverse_matrix(d-dot_product(w,l)),
                (dot_product((1-w),d))+dot_product(w,u))
        matrix C <- dot_product((inverse_matrix(d-(w*l))*w),b.transpose())
        matrix C <- dot_product(inverse_matrix(d - l), b.transpose())
        float E <- infinity_value()
        array xant <- xo.transpose()
        int cont <- 0

        array values, array normalized_eigenvectors <- eigen_values(T)
        float spectral_radius <- maximum_value(absolute_value(values))

        while ((E > tol) and (cont < nmax))
                matrix xact <- dot_product(T, xant) + C
                E <- norm2(xant - xact)
                xant <- xact
                cont <- cont + 1
                array_steps[cont] <- xant
        end while

        xact, array_steps

end sor
```

- Vandermonde

```
Input: array X, array Y
Output: array Coef

begin vandermonde

    int n <- X.size()
    matrix A <- zeros_matrix(n, n)

    for (int i from 0 until n)
        for (int j from 0 until n)
            A[j][i] <- X[j]^(n - (i + 1))
        end for
    end for

    array coef <- solution(A, Y.transposed)
        // solution is a function that solve systems of equations
```

```
    return coef

end vandermonde
```

- Divided difference method:

```
Inpunt: vector x, vector y
Output: matrix D, vector coefficient , string polynomial

begin  dividedDifferenceMethod

    n = size(x)
    D = matrix_zeros(n,n)

    D[:,0] = y.transpose()

    for i to n:
        aux0 = D[i-1:n,i-1]
        aux1 = adjacent_difference(aux0)
        aux2 = vector_subtraction(x[i:n],x[0:n-1-i+1])
        D[i:n,i] = vector_division(aux1,aux2.transpose())
    end

    coefficient = diagonal(D)

    polynomial = coefficient[0]
    m = '(x' + (-x[0]) + ')'
    for i to n:
        polynomial += coefficient[i] + m
        m += '(x' + -x[i] + ')'
    end
    D,coefficient ,polynomial

end dividedDifferenceMethod
```

- Lagrange

```
dictionary results is a dictionarity that have every Li
        lagrnage coefficient and have the final polynomial
sm is a import of sympy from python to to represent
        variables in a polynomial
value.expand(): this function do a expands for the math
        expression (is a function from sympy)

Input: matrix data, int n
Output: dictionary results
```

```
begin lagrange

int count <- 0
array Arrx
array Arry
while (count < 2*n) do:
        if (count < n):
                Arrx.add(data[count])
        else:
                Arry.add(data[count])
end while
sizeX <- size(Arrx)
sizeY <- size(Arry)
dict result
if (sizeX != sizeY):
        x <- sm.symbols('x')
        array polynomial
        array arrayL
        count <- 0
        while (count < sizeX) do:
                int pos <- count
                float value <- Arrx[count]
                float numerator <- 1
                float denominator <- 1
                int count2 <- 1
                while (count2 < sizeX) do:
                        if count != count2:
                                numerator <- numerator*(x-Arrx[count2])
                                denominator <- denominator*(value-Arrx[count2])
                        end if
                end while
                floar aux <- numerator/denominator
                aux <- aux.expand()
                result[count] <- aux
                coefficient <- numerator*Arry[count]/denominator
                coefficient <- coefficient.expand()
                polynomial.add(coefficient)
        end while
        float sumPol <- 0
        count <- 0
        while (count < size(polynomial)):
                sumPol <- sumPol + polynomial[count]
        end while
        result["polynomial"] <- sumPol
return result
```

```
end lagrange
```

- Lineal spline

```
matrix_function.mix_matrix: Method in matrix_function that
        mix to b array and a array to find a A matrix
matrix_function.soltion: Method in matrix_function that do
        a progressive or backward substitutionv to find an array
total_gaussian_method.totalGaussianMethod: Method in
        total_gaussian_method that found the solution of matrix
        Ax = B by total gaussian method.
matrix_function.sort: Method in matrix_function that organize
        in case of changing rows

Input: array x, array y
Output: array coefficient

begin splineLineal

sizeX <- size(x)
sizeY <- size(y)

if (sizeX != sizeY):
        break;
int m <- 2*(sizeX -1)
A = identityMatrix(m)

int count <- 0
while (count < m) do:
        A[count][count] <- 0
end while

int counter <- 0
int counterRow <- 0
array b
int count <- 0
while (count < sizeX) do:
        array vec_x <- x[count]
        A[count][counter] <- vec_x
        A[count][counter+1] <- 1
        counter <- counter + 2
        if (count == 0):
                counter <- 0
        b.add(y[count])
        counterRow <- counterRow + 1
        count <- count + 1
end while
```

```
counter <- 0
count <- 0
while (count < m) do:
        A[counterRow][counter] <- x[count]
        A[counterRow][counter+1] <- 1
        A[counterRow][counter+2] <- -(x[count])
        A[counterRow][counter] <- -1
        counter <- counter + 2
        counterRow <- counterRow + 1
        b.add(0)
end while
array arr
count <- 0
while (count < m) do:
        array arr2
        int countAux <- 0
        while (countAux < m) do:
                arr2.add(A[count][countAux])
        arr.add(arr2)
        end while
end while
a, b, matrix <- matrix_function.mix_matrix(A,b)
a,b,dic,movement <- total_gaussian_method.totalGaussianMethod(matrix)
x = matrix_function.soltion(a,b)
x = matrix_function.sort(x,movement)
array aux
count <- 0
while (count < size(x)):
        aux.add(x[count])
end while
array coefficient
array plotter
count <- 0
while (count < size(aux)-1) do:
        array aux2
        aux2.add(aux[count])
        aux2.add(aux[count+1])
        count <- count + 2
        coefficient.add(aux2)
end while
count <- 0
return coefficient

end splineLineal
```

- Cuadratic spline

```
Input: vector x, vector y
Output: coefitiens , Matrix A

begin cuadraticSpline

    if x or y has duplicates:
        return error
    end if

    if lenght of x is not equals to lenght of y:
        return error
    end if

    set n = lenght of x
    set m = (n - 1) * 3
    set A = matrix[m][m]
    set B = vector[m]
    set A[0][0] = x[0] ^2
    set A[0][1] = x[0]
    set A[0][2] = 1
    set B[0] = y[0]
    #interpolation conditions
    For i = 0,..., n    1
        set A[i+1][3*(i+1)-3] = math.pow(x[i+1], 2)
        set A[i+1][3*(i+1)-2] = x[i+1]
        set A[i+1][3*(i+1)-1] = 1
        set B[i+1] = y[i+1]
    end for
    #continuity conditions
    For i = 1,..., n    1
        set  A[n-1+i][3*i-3] = math.pow(x[i], 2)
        set  A[n-1+i][3*i-2] = x[i]
        set  A[n-1+i][3*i-1] = 1
        set  A[n-1+i][3*i] = -math.pow(x[i], 2)
        set  A[n-1+i][3*i+1] = -x[i]
        set  A[n-1+i][3*i+2] = -1
        set  B[n-1+i] = 0
    end for
    #softness condition
    for i = 1,..., n    1
        set A[2*n-3+i][3*i-3] = 2 * x[i]
        set   A[2*n-3+i][3*i-2] = 1
        set  A[2*n-3+i][3*i-1] = 0
        set  A[2*n-3+i][3*i] = -2 * x[i]
        set  A[2*n-3+i][3*i+1] = -1
        set  A[2*n-3+i][3*i+2] = 0
```

```
        set  B[2*n-3+i] = 0
    end for
    set  A[m-1][0] = 2
    set  B[m-1] = 0
    x = solveSystem(A, B)
    return x, A

end cuadraticSpline
```

- Cubic spline

```
Input: vector x, vector y
Output: coefitiens , Matrix A

begin cubicSpline

    if x or y has duplicates:
        return error
    end if

    if lenght of x is not equals to lenght of y:
        return error
    end if

    set n = lenght of x
    set m = (n - 1) * 4
    set A = matrix[m][m]
    set B = vector[m]
    set A[0][0] = x[0] ^3
    set A[0][1] = x[0] ^2
    set A[0][2] = x[0]
    set A[0][3] = 1
    set B[0] = y[0]
        #interpolation conditions
    For i = 0,..., n    1
        set A[i+1][4*(i+1)-4] = math.pow(x[i+1], 3)
        set A[i+1][4*(i+1)-3] = math.pow(x[i+1], 2)
        set A[i+1][4*(i+1)-2] = x[i+1]
        set A[i+1][4*(i+1)-1] = 1
        set B[i+1] = y[i+1]
    #continuity conditions
    for i = 1,..., n    1
        set A[n-1+i][4*i-4] = math.pow(x[i], 3)
        set A[n-1+i][4*i-3] = math.pow(x[i], 2)
        set A[n-1+i][4*i-2] = x[i]
        set A[n-1+i][4*i-1] = 1
        set A[n-1+i][4*i] = -math.pow(x[i], 3)
```

```
        set A[n-1+i][4*i+1] = -math.pow(x[i], 2)
        set A[n-1+i][4*i+2] = -x[i]
        set A[n-1+i][4*i+3] = -1
        set B[n-1+i] = 0
    #softness condition
    for i = 1,..., n    1
        set A[2*n-3+i][4*i-4] = 3 * math.pow(x[i], 2)
        set A[2*n-3+i][4*i-3] = 2 * x[i]
        set set A[2*n-3+i][4*i-2] = 1
        set A[2*n-3+i][4*i-1] = 0
        set A[2*n-3+i][4*i] = -3 * math.pow(x[i], 2)
        set A[2*n-3+i][4*i+1] = -2 * x[i]
        set A[2*n-3+i][4*i+2] = -1
        set A[2*n-3+i][4*i+3] = 0
        set B[2*n-3+i] = 0
    #concavity conditions
    for i = 1,..., n    1
        set A[3*n-5+i][4*i-4] = 6 * x[i]
        set A[3*n-5+i][4*i-3] = 2
        set A[3*n-5+i][4*i-2] = 0
        set A[3*n-5+i][4*i-1] = 0
        set A[3*n-5+i][4*i]   = -6 * x[i]
        set A[3*n-5+i][4*i+1] = -2
        set A[3*n-5+i][4*i+2] = 0
        set A[3*n-5+i][4*i+3] = 0
        set B[n+5+i]= 0
    #boundary conditions
    set A[m-2][0] = 6 * x[0]
    set A[m-2][1] = 2
    set A[m-1][m-4] = 6 * x[n-1]
    set A[m-1][m-3] = 2
    x = solveSystem(A, B)
    return x, A

end cubicSpline
```

- Aitken

```
Input: function f(x), float x_0, x_1, float tolerance, int iterations
Output: solution vector results

begin aitkent:
    array results
    bisectionResult <- bisection(function,x_0,x_1)
    infinite <- MAXIMUM FLOAT VALUE
    if (bisectionResult != 0) then
        count <- 1
```

```
        error <- infinite
        xAitken0 <- 0
        while (count <= iterations and error > tolerance and
                error != 0 and bisectionResult != 0) do
            x1 <- bisectionResult[0][1]
            x2 <- bisectionResult[1][1]
            x3 <- bisectionResult[2][1]
            xAitken <- (x1 * x3 - (x2 ** 2)) / (x3 - 2 * x2 + x1)
            f_xAitken <- function(xAitken)
            error <- |xAitken0 - xAitken|
            if (error == 0) then
                error <- infinite
            xAitken0 <- xAitken
                        array aux = [count,xAitken,f_xAitken,error]
            results[count] <- aux
            x_0 <- bisectionResult[1][0]
            x_1 <- bisectionResult[1][2]
            bisectionResult <- bisection(function,x_0,x_1)
            count <- count + 1
    for key in results do:
        if (results[key][3] == infinite) then
            results[key][3] <- 0
    return results

end aitken
```

- Steffensen

```
Input: function f(x), float initial_x, float tolerance, int iterations
Output: result table results

if(f(x) is not a valid function)
    break;
if(initial_x or tolerance or iterations are not valid numbers)
    break;
if(tolerance is lees than 0)
    break;
if(iterations is less than 1)
    break;

array results

int iter_count <- 0
float xi_plus_f_xi <- initial_x + f(initial_x)
float f_xi_plus_f_xi <- f(xi_plus_f_xi)
float error <- MAXIMUM FLOAT VALUE
results[iter_count] <- [iter_count, initial_x,
```

```
f(initial_x), xi_plus_f_xi, f(xi_plus_f_xi), "N/A"]
float previous_x <- initial_x

    while iter_count < iterations and error > tolerance do:
        iter_count <- iter_count + 1
        current_x <- previous_x - ((f(previous_x)^2)/
        (f(previous_x + f(previous_x)) - f(previous_x)))
        xi_plus_f_xi <- current_x + f(current_x)
        f_xi_plus_f_xi <- f(xi_plus_f_xi)
        error <- |previous_x - current_x|
        results[iter_count] <- [iter_count, current_x,
        f(current_x), xi_plus_f_xi, f_xi_plus_f_xi, error]
        previous_x <- current_x
    end while

return results
```

- Muller

```
Input: function f(x), float x_0, float x_1, float x_2, int iterations
Output: solution vector results

begin muller

if (tolerance or iterations or x_0 or x_1 or x_2 is not a valid numbers):
    break;
if (x_1 is equals to x_2):
        break;
array results
if ((function(x_0) > 0 and function(x_1) < 0) or
            (function(x_0) < 0 and function(x_1) > 0)) then
    int count <- 0
    float error <- |x_1 - x_2|
    while ((error > tolerance) and (count < iterations)) do
        float h_0 <- x_1 - x_0
        float h_1 <- x_2 - x_1
        float f_x0 <- function(x_0)
        float f_x1 <- function(x_1)
        float f_x2 <- function(x_2)
        float delta_0 <- (f_x1 - f_x0) / h_0
        float delta_1 <- (f_x2 - f_x1) / h_1
        float a <- (delta_1 - delta_0) / (h_1 - h_0)
        float b <- (a * h_1) + delta_1
        float c <- f_x2
                float aux <- (b ** 2) - (4 * a * c)
                if (aux < 0) then
                        break;
```

```
        float raiz <- raiz((b ** 2) - (4 * a * c))
        if (b < 0) then
            denominador = b - raiz
        else:
            denominador = b + raiz
        x_3 <- x_2 + ((-2*c)/denominador)
        x_0 <- x_1
        x_1 <- x_2
        x_2 <- x_3
        error <- |x_1 - x_2|
                array iteration <- [count,x_2,f_x2,error]
        results[count] <- iteration
        count <- count + 1
        end while
return results

end muller
```

- Gaussian elimination for tridiagonal matrices

```
a: diagonal above main diagonal
b: principal diagonal
a: diagonal down the main diagonal
b: constant vector

        Input: vector a,vetor b,vector c, vector d
begin gaussianTridiagonalMatrixMethod:
        n = size(d)
        matrix = matrix_zeros(n,n)
        for i to (n-1):
        m = a[i]/b[i]
        matrix[i+1][i+1] = b[i+1] = b[i+1] - (m*c[i])
        matrix[i][i+1] = c[i]
        d[i+1] = d[i+1] - (m*d[i])
    end
    matrix
end
```

- Gaussian elimination with stepped pivoting

```
numpy as np is python numpy library to converts to array a matrix
matrix_function.soltion:
Method in matrix_function that do a progressive or backward substitutionv
to find an array

Input: matrix matrix
```

```
Output: solution array x

begin steppedMethod

dict dictionary
auxiliary_matrix <- np.array(matriz)
dictionary[0] <- matrix
int count - 0
array temporal_array
while (count < matrix.shape[0]-1) do:
        float pivot_number <- auxiliary_matrix
        if (count == 0):
                for row in auxiliary_matrix:
                        pivot_column <- np.abs(row[:-1])
                        temporal_maxpivot <- np.max(pivot_column)
                        temporal_array.add(temporal_maxpivot)
                end for
        end if
        sub_matrix <- auxiliary_matrix.T[0]
        division_colum = np.abs(sub_matrix)/temporal_array[count:]
        posmax_pivot <- np.where(division_colum ==
                        np.max(division_colum))[0][0]
        if (posmax_pivot != 0):
                pivot_number <- auxiliary_matrix[posmax_pivot][0]
        temporal_matrix <- np.array(auxiliary_matrix[0])
        auxiliary_matrix[0] <- np.array(auxiliary_matrix[posmax_pivot])
        auxiliary_matrix[posmax_pivot] <- temporal_matrix
        temporal_matrix <- np.array(matrix[i])
        matrix[i] <- np.array(matrix[i+posmax_pivot])
        matrix[i+posmax_pivot] <- temporal_matrix
        end if
        if (pivot_number==0 and i == matrix.shape[0]-2):
                break;
        end if
        fj <- auxiliary_matrix[0]
    column_vector <- np.reshape(auxiliary_matrix.T[0][1:],
    (auxiliary_matrix.T[0][1:].shape[0], 1))
    multiplier <- column_vector/pivot_number
    fi <- auxiliary_matrix[1:]
    fi <- fi - (multiplier*fj)
        if(count == 0):
                matrix[i+1:] <- fi
        else:
                axiliary_fi <- fi
                while (axiliary_fi.shape[1]+1 <matrix[i+1:].shape[1]):
                        axiliary_fi <-  np.insert(axiliary_fi,
```

```
                        0, np.zeros(1), axis=1)
                matrix[i+1:] <- np.insert(axiliary_fi , 0,
                            np.zeros(1), axis=1)
        auxiliary_matrix <- fi.T[1:].T
    dictionary[count+1] <- np.array(matrix)
end while
a <- np.delete(matrix, matrix.shape[1]-1, axis=1)
b <- matrix.T[matrix.shape[1]-1]
return matrix_function.soltion(a,b)

end steppedMethod
```

- Crout for tridiagonal matrices

```
Input: Matrix A, vector B
Output: Steps of making the
crout factorization for tridiagonal matrices and the answer to the system

Set L11 = a11; u12 = a12/L11; z1 = a1,n+1/L11
Set n = lenght of matrix A
For i = 2,..., n - 1
    Set Li,i-1 = ai,i-1
    Set Lii = aii - Li,i-1*ui-1,i
    Set ui,i+1 = ai,i+1/Lii
    Set zi = (ai,n+1 - Li,i-1*zi-1)/Lii
end for
    Set Ln,n-1 = an,n-1
    Set Ln,n = an,n - Ln,n-1*un-1,n
    Set zn = (an,n+1 - Ln,n-1*zn-1)/Ln,n
Set z = SolveSystem(L,b)
Set x = SolveSystem(U,z)
return x
end
```

- Neville's method

```
Input: vector x, vector y, x_inter (value to interpolate)
Output: (float) y interpolated, Q: Coefficients matrix

    if x or y has duplicates:
        return error
    end if

    if lenght of x is not equals to lenght of y:
        return error
    end if
```

```
    set n = lenght of x
    set Q = Matrix[n][n−1] filled with zeros
    set Q = Q with y vector concatenated as the last column
    For i = 1,..., n
        For j = 1,..., i + 1
            set Q[i,j] = ((x_inter−x[i−j])∗Q[i,j−1]−(x_inter−x[i])∗
                               Q[i−1,j−1])/(x[i]−x[i−j])
        end For
    end For
    y_int = Q[n−1,n−1]
    return y_int,Q
```

- Hermite interpolation

```
Comment:
dictionary data is a dictionarity (json format) that have
every Hi Hermite coefficient and have the final polynomial
sm is a import of sympy from python to to represent variables
in a polynomial

parse_expr(): this function that change a string to sympy expression
(is a function from sympy)
lagrange.lagrange: its a methos in the lagrange.py file that give us
the lagrange coefficient

diff(x) it's a method from sympy to found a derivative of a specific
function

json.dumps it's a method to do a json from a dict


Input: array arrayX array arrayY, array arrayz, int size
Output: json data

begin hermite
x <− sm.symbols('x')
array arrayAux
array arrayDerivate
array arraySquare
array H
array H2
int counter <− 0
dictionarity dic <− lagrange.lagrange(x,y,size)
while (counter < size(dic)−1) do:
        arrayAux.add(parse_expr(dic[counter))
        arraySquare.add(parse_expr(dic[counter)∗parse_expr(dic[counter))
        counter <− counter + 1
```

```
counter <- 0
while (counter < size(arrayAux)) do:
        arrayDerivate.add(arrayAux[counter].diff(x))
        couunter <- counter + 1
counter <- 0
while (counter < size) do:
        value <- arrayDerivate[counter].subs(x,arrayX)
        aux <- (((x-arrayX[counter])*value*(-2))+1)
        aux <- aux * arraySquare[counter]
        H.add(aux)
        value <- (x-arrayX[i])*arraySquare[i]
    H2.add(value)
        counter <- counter + 1
polynomial <- 0
counter <- 0
while (counter < size(H)) do:
        results[counter] <- arrayY[counter]*H[counter]
        polynomial <- polynomial+(arrayY[counter]*H[counter])
        counter <- counter + 1
results["polynomial"] <- polynomial
data <- json.dumps(results)
return data
end hermite
```

3 Project Conclusions

For the development of the web application it was decided to use the Laravel framework, this due to the ease that PHP in conjunction with the HTML tag language provides for web development.

It is important to highlight that all the method algorithms were developed in python using different libraries, for example the numpy library that allows operations with matrices easily and efficiently. Another library used was sympy, this library provided us with operations between polynomials and mathematical expressions also easily and efficiently. Finally the math.js and function-plot libraries provided tools for graphing and evaluating functions dynamically. Regarding the architecture of the application, a View - Controller architecture was considered. It from the Controller component is where the call to each of the python methods mentioned above is made.

It is important to note that there were limitations when deploying the application due to the fact that each library used had to be installed in the container of the server to be used, in the same way there were limitations when installing python, as that was necessary to be able to make the call to the algorithms. We think as a team that the languages and libraries used don't disappoint overall, PHP with the Laravel framework let us develop our architecture pretty easily because it does have classes unlike JavaScript based frameworks. Regarding python as a backend for the methods, we feel that it was a good choice, it provided us with robust libraries to ease the development, the times to solve the methods and the memory consumption are reasonable and python as a whole is a easy language to program in, however we are aware that other languages like mathlab or compiled languages as C++ are faster. We did not encounter any problems with the root finding methods, but we did find that the matrices methods could start taking a while to run when the input matrices were big.

In the About section of the application you can find more information regarding the libraries used. In general, the development of the project was one of much learning in mathematical skills, programming skills and soft skills. This development had challenges that we as a team had to face, and we did it pretty well, we distributed the tasks in a way that none of the team members had work overload, we learned a lot about the coordination needed to allow each team member to contribute without conflicts in our code base.