Numerical AS

Santiago Espinosa Valderrama

201810075010

Edwar Samir Posada Murillo - ST0236

- ANÁLISIS NUMÉRICO

Departamento de ingeniería de sistemas
Universidad EAFIT

Medellín, Colombia

Jueves 24 de Noviembre de 2020

1. Introducción

El análisis numérico a sufrido un cambio abismal en las últimas décadas debido a la potenciación en las tecnologías computacionales, es necesario año a año renovar estas tecnologías, pero es imprescindible comprender los cimientos de está rama matemática para poder hacer métodos sustentados en las leyes numéricas. Con idea de entender y manejar estos conceptos básicos es que se cursa la materia de análisis numérico. Para sustentar este conocimiento la mejor manera es llevar a cabo un proyecto que nos de la posibilidad de confirmar este propósito.

2. Objetivos

2.1. Objetivo general

Dadas las características del enfoque numérico hacia la solución de problemas matemáticos, por medio de este proyecto se busca ofrecer una serie de herramientas que nos permitirán brindar soluciones acertadas dentro de los limitantes que cada una de estas herramientas presenta. De igual modo, procuramos facilitar al usuario el análisis de estas soluciones por medio del seguimiento dentro de cada una de las etapas, por las que atraviesan los métodos para llegar a dicha solución.

2.2. Objetivos específicos

Los principales elementos que darán sentido al cumplimiento de nuestro propósito son los siguientes:

- Presentar uno a uno los métodos agrupados dentro de su respectiva sección dando alusión al tipo de problemas que pueden abordar y la mirada desde la cual se abordan dichos problemas.
- Aclarar el funcionamiento y propósito de cada método por medio de una breve introducción que contextualice al usuario con los procesos que se llevan a cabo dentro del método, los parámetros a ingresar y demás aspectos a tener en cuenta.
- Favorecer el análisis de las soluciones que brindan los métodos dispuestos, por medio de un seguimiento paso a paso de los procesos y etapas por las cuales pasa cada uno de ellos, la transformación de los datos, el manejo de los errores y el feedback continuo hacia el usuario.

3. Cuerpo de trabajo

Una de las formas de poner en práctica la adquisición de competencias con respecto a los diferentes conceptos que se trabajaron durante el curso, es la implementación de los métodos numéricos dentro del campo del desarrollo de software que permitan hacer uso de éstos en diferentes contextos sólo con cambiar los parámetros de entrada.

De forma interactiva, didáctica y simplificada, este proyecto se acerca a esta idea. A continuación, se plantean los diferentes métodos implementados dentro del desarrollo de la aplicación web, cuyo tipo he escogido para brindar fácil acceso por parte de los usuarios sin necesidad de comprometer los recursos de hardware de sus máquinas.

3.1. Descripción y algoritmo de los métodos

Para cada una de las secciones, siguiendo el hilo propuesto por Correa (2010), se plantean cada uno de los métodos comprendidos dentro de dicha sección junto con las consideraciones necesarias para argumentar matemática y computacionalmente el uso de éstos.

3.1.1. Métodos numéricos para la solución de ecuaciones de una variable

Dada una función $f(x)$ definida sobre los números reales, la ecuación general de una variable tiene la forma $f(x) = 0$ y solución está dada por el conjunto de valores de x que permitan que la igualdad sea cierta (Correa, 2010).

Para encontrar la solución a una ecuación con la forma como se ha descrito anteriormente, es necesario conocer algunas condiciones iniciales como la continuidad, máximos, mínimos o demás aspectos que pueden ser necesarios dependiendo del método numérico a utilizar, estos aspectos se irán mencionando a medida que se presentan los métodos que los requieren. Los siguientes son los métodos desarrollados dentro del proyecto para permitirnos encontrar los valores necesarios para la solución de una ecuación de una variable o aproximaciones a ellos.

3.1.1.1. Método de búsquedas incrementales

Este método es útil para encontrar un intervalo $[a, b]$ que contenga una raíz.

Algoritmo:

```
read x0, delta, n
fx0 = f(x0)
if fx0 = 0 then
  show “x0 is a root”
else if delta <= 0 or n <= 0 then
  show “delta and n must be > 0”
else
  x1 = x0 + delta
  fx1 = f(x1)
  i = 1
  while fx0 * fx1 > 0 and i < n do
      x0 = x1
```

```
        fx0 = fx1
        x1 = x0 + delta
        fx1 = f(x1)
        i += 1
    EndWhile
    if fx1 = 0 then
        x1 is a root
    else if fx0 * fx1 < 0 then
        show "There is a root in [x0, x1]"
    else
        show "Solution failed for n iterations"
    EndIf
EndIf
```

Métodos cerrados

Un método cerrado, es aquel cuya característica principal es que requiere un intervalo donde se encuentre una raíz. A lo largo de la ejecución, éste reduce el intervalo cada vez más hasta un punto de convergencia. A continuación, se describen dos métodos de esta clasificación, bisección y regla falsa. Correa (2018).

3.1.1.2. Método de la bisección

Este es un método que consiste en la reducción del intervalo que contiene la raíz en dos subintervalos de igual distancia, luego de ello se conserva el intervalo que continúa manteniendo las características iniciales. De este modo se construye una sucesión de puntos medios que termina por converger al valor de la raíz.

Si ya conocemos el teorema de valor intermedio, podemos determinar que éste método está fundamentado allí, ya que el método parte del supuesto que el teorema se cumple y genera una sucesión de valores xm los cuales se reducen cada vez a la mitad dada por la media.

3.1.1.3. Método de la regla falsa

Este es una variación del método anterior, por lo tanto su fundamento matemático es casi el mismo. Con la diferencia de que el valor de punto medio xm se halla al encontrar la intersección entre la recta secante que cruza los dos extremos del intervalo y el eje x.

Dado lo anterior, hemos generado una pequeña modificación al algoritmo de bisección y regla falsa en busca de evitar repeticiones en el código.

Algoritmo biseccion y regla falsa:

```
read xi, xs, tolerance, n, reg_falsa
if tolerance < 0 then
  show “tolerance must be >= 0”
else if  n < 1 then
  show “n must be >= 1"
else
  fxi = f(xi)
  fxs = f(xs)
  if fxi = 0 then
      show "xi is a root"
  else if fxs = 0 then
      show "xs is a root"
  else if fxi * fxs < 0 then
      if reg_falsa then
          xm = xi - f(xi)*(xi, xs) / (f(xs) - f(xi))
      else
          xm = (xi, xs) / 2
      EndIf
      fxm = f(xm)
      count = 1
      error = tolerance + 1
      while error > tolerance and fxm != 0 and count < n do
          if fxi * fxm < 0 then
              xs = xm
              fxs = fxm
          else
              xi = xm
              fxi = fxm
              x_aux = xm
          EndIf
          if reg_falsa then
              xm = xi - f(xi)*(xi, xs) / (f(xs) - f(xi))
          else
              xm = (xi, xs) / 2
          EndIf
          fxm = f(xm)
          error = error (xm,  x_aux)
          count = count + 1
      EndWhile
```

```
      if fxm = 0 then
          show "xm is a root"
      else if error < tolerance then
           show "xm is an approximation to a root with tolerance =
tolerance"
      else
          show "Failure in n iterations"
      EndIf
  else
      show "The interval is inappropriate"
  EndIf
EndIf
```

Métodos abiertos

Éstos métodos, a diferencia de los anteriores calculan aproximaciones cada vez más cercanas a la raíz sin necesidad de verificar si se genera o no un intervalo que contiene a la raíz. De igual forma producen una sucesión de valores que **no siempre** convergen a la raíz pero cuando ocurre, en general lo hacen más rápido (Correa, 2010). Seguidamente, presentamos los métodos implementados dentro del proyecto cuya característica principal es ésta: punto fijo, Newton, secante y raíces múltiples.

3.1.1.4. Método de punto fijo

Calcula una aproximación a una raíz por medio de encontrar una intersección entre la función y = x y la ecuación x = g(x) donde g(x) es una ecuación equivalente a f(x) = 0 que resulta de aplicar despejes y sustituciones.

Algoritmo:

```
read x0, tolerance, n
if tolerance < 0 then
     show "tolerance must be >= 0"
else if n < 1 then
     show "n must be >= 1"
else
  fx = f(x0)
  count = 0
  error = tolerance + 1
  while fx != 0 and error > tolerance and count < n do
      xn = g(x0)
      fx = f(xn)
      error = error(x0,xn)
      x0 = xn
```

```
      count = count 1
  EndWhile
  if fx == 0 then
      show "x0 is a root"
  else if error < tolerance then
       show "x0 is an approximation to a root with tolerance =
tolerance"
  else
      show "failure in n iterations"
  EndIf
Endif
```

3.1.1.5. Método de Newton

Puede verse como una variante del método de punto fijo. Por ende tratará de encontrar una aproximación a la raíz mediante una sucesión de valores dada una expresión de la forma x = g(x). Sin embargo, aquí sabremos que g(x) = x - f(x) / f'(x).
Visto de otro modo, cada x se genera por medio de la intersección entre el eje x y la tangente a la curva y = f(x) en el punto dado al momento de la iteración.

Algoritmo:

```
read x0, tolerance, n
if tolerance < 0 then
    show "tolerance must be >= 0"
else if n < 1 then
    show "n must be >= 1"
else
   fx = f(x0)
   df = df_dx(x0, 1)
   count = 0
   error = tolerance + 1
   while fx != 0 and df != 0 and error > tolerance and count<n do
       xn = x0 - (fx / df)
       fx = f(xn)
       df = df_dx(xn, 1)
       error = error(x0,xn)
       x0 = xn
       count = count + 1
   EndWhile
   if fx = 0 then
```

```
        show "x0 is a root"
    else if error < tolerance then
          show "x0 is an approximation to a root with tolerance =
tolerance"
    else if df = 0 then
        show "x0 is a possible multiple root"
    else
        show "failure in n iterations"
    End
EndIf
```

3.1.1.6. Método de la secante

Es una variante del método de Newton y transitivamente del método del punto fijo. Por lo tanto, al igual que los anteriores busca una aproximación a la raíz de forma iterativa por medio de la sucesión de valores que se espera que converjan a un valor. Es ideal cuando la derivada de la función f(x) es demasiado compleja, entonces el método de la secante hallará una aproximación a dicha derivada por medio de límites.

Algoritmo:

```
read x0, x1, tolerance, n
fx0 = f(x0)
if tolerance < 0 then
     show "tolerance must be >= 0"
else if n < 1 then
     show "n must be >= 1"
else if fx0 = 0 then
    print (f"{x0} is a root")
else
    fx1 = f(x1)
    den = fx1 - fx0
    count = 0
    error = tolerance + 1
    while fx1!=0 and error > tolerance and den != 0 and count<n do
        x2 = x1 - (fx1 * (x1 - x0)) / den
        error = error(x1, x2)
        x0 = x1
        fx0 = fx1
        x1 = x2
        fx1 = f(x1)
        den = fx1 - fx0
```

```
        count = count + 1
    EndWhile
    if fx1 = 0 then
        show "x1 is a root"
    else if error < tolerance then
         show "x1 is an approximation to a root with tolerance =
tolerance"
    else if den = 0 then
        show "x1 is a possible multiple root"
    else
        show "failure in n iterations"
    EndIf
EndIf
```

3.1.1.7. Método de raíces múltiples

Este es un método útil para hallar una aproximación a una raíz, cuando los métodos anteriores pierden velocidad de convergencia debido a una aproximación de la derivada de la función f(x) al valor 0, principalmente cuando la raíz está posicionada sobre un máximo o un mínimo, es decir, existe una raíz múltiple. En ese caso, Ralston y Rabinowitz (1978), como lo menciona Correa (2010) han propuesto una modificación a la forma como se calcula x = g(x).

Algoritmo:

```
read self, x0, tolerance, n
if tolerance < 0 then
    show "tolerance must be >= 0"
else if n < 1 then
    show "n must be >= 1"
else
    fx = f(x0)
    df1 = df_dx(x0, 1)
    df2 = df2_dx(x0, 2)
    count = 0
    error = tolerance + 1
    while fx != 0 and error > tolerance and count < n do
        xn = x0 - (fx*df1) / ((df1^2) - fx*df2)
        fx = f(xn)
        df1 = df_dx(xn, 1)
        df2 = df2_dx(xn, 2)
        error = error(x0,xn)
```

```
        x0 = xn
        count = count + 1
    EndWhile
    if fx = 0 then
        show "x0 is a root"
    else if error < tolerance then
          show "x0 is an approximation to a root with tolerance =
tolerance"
    else
        show "failure in n iterations"
    EndIf
EndIf
```

3.1.2. Métodos de sistemas de ecuaciones

Como lo describe Correa (2010), un sistema de ecuaciones es un conjunto de m ecuaciones por n incógnitas, donde la solución es encontrar los valores de esas incógnitas de modo que se satisfagan todas las ecuaciones.

En nuestro proyecto asumimos que los sistemas a solucionar cuentan con n ecuaciones por n incógnitas y están definidos sobre los números reales.

3.1.2.1. Eliminación gaussiana simple

Como primer método para la solución de sistemas de ecuaciones, busca encontrar los valores de las n incógnitas de un sistema de ecuaciones, satisfaciendo la ecuación Ax = b. Para esto, el método se basa en dos fases fundamentales, la transformación de la matriz convirtiéndola en una matriz triangular superior U y el despeje de las variables por medio de una sustitución regresiva.

Algoritmos:

```
read A,b
if length(b) != length(A) then
    show "A must be a nxn Matrix and b a n vector"
else

    U,B = Elimination(A, b)
    x = regresiveSustitution(U,B)
    show x
EndIf

function Elimination(A,b)
    for k = 1 to n do
        if A[k][k] = 0 do
```

```
            return "A must not have any zero in its diagonal"
        EndIf
    EndFor
    Ab = concat(A,b)
    for k = 1 to n - 1 do
        for i = k+1 to n do
            multiplier = Ab[i][k] / Ab[k][k]
            for j = k to n+1  do
                Ab[i][j] = Ab[i][j] - multiplier * Ab[k][j]
            EndFor
        EndFor
    EndFor
    return U,B
EndFunction

function regresiveSustitution(Ab,n)
    x[n] = Ab[n][n+1] / Ab[n][n]
    for i = n-1 to 1 step -1 do
        sum = 0
        for p = i + 1 to n do
            sum = sum + Ab[i][p] * X[p]
        EndFor
        X[i] = (Ab[i][n+1] - sum) / Ab[i][i]
    EndFor
    return x
EndFunction

function gaussianEliminationWithPivoting(A, B, n)
  for k = 1 to n do
      if A[k][k] = 0 then
          return "A must not have any zero in its diagonal"
      EndIf
  EndFor
  Ab = concat(A,b)
  for k = 1 to n - 1 do
       Ab = pivoting(Ab, n , k)
      for i = k+1 to n do
          multiplier = Ab[i][k] / Ab[k][k]
          for j = k to n+1  do
              Ab[i][j] = Ab[i][j] - multiplier * Ab[k][j]
```

```
            EndFor
        EndFor
    EndFor
    return U,B
EndFunction
```

3.1.2.2. Eliminación gaussiana con pivoteo parcial

Con una ligera variación de la eliminación gaussiana simple, su sentido es el mismo, buscar un valor para las n incógnitas del sistemas de ecuaciones. Sin embargo, con el fin de reducir el error de propagación, la estrategia es que en cada etapa k de la eliminación se busca que el pivote (el elemento de la diagonal) sea el mayor en valor absoluto de los elementos de la columna.

Algoritmo:

```
function partialPivoting(Ab, n, k)
    major = |Ab[k][k]|
    majorRow = k
    for s = k+1 to n do
        if |Ab[s][k]| > major then
            majorRow = s
        EndIf
    EndFor
    if major = 0 then
        return "No unique solution"
    else
        if majorRow != k then
            Ab = changeRows(Ab, majorRow, k)
        EndIf
        return Ab
    EndIf
EndFunction
```

3.1.2.3. Eliminación gaussiana con pivoteo total

Es una variación de la eliminación gaussiana un poco más estricta. Su propósito es buscar que los mayores elementos de toda la matriz A queden en la diagonal. Ésto, para reducir considerablemente el error de propagación.

Algoritmo:

```
function partialPivoting(Ab, n, k)
    major = |Ab[k][k]|
```

```
    majorRow = k
    majorColumn = k
    for r = k to n do
        for s = k to n do
            if |Ab[r][s]| > major then
                majorRow = r
                majorColumn = s
            EndIf
        EndFor
    EndFor
    if major = 0 then
        return "No unique solution"
    else
        if majorRow != k then
            Ab = changeRows(Ab, majorRow, k)
        EndIf
        if majorColumn != k then
            Ab = changeColumns(Ab, majorRow, k)
            marks = changeMarks(marks, majorColumn, k)
        EndIf
        return Ab,marks
    EndIf
EndFunction
```

Factorización LU

La factorización LU buscan reemplazar la matriz A por dos matrices triangulares cuyo producto es igual a ella. La primera matriz es L triangular inferior y la segunda es U triangular superior.

En ese sentido el sistema Ax = b para a convertirse en dos sistemas:

Lz = b

Ux = z

Para dicho propósito, se presentan los 3 métodos directos propuestos en el libro guía del curso:

Cholesky, cuyo punto de partida es Lii = Uii

Crout, cuyo punto de partida es Uii = 1

Doolittle, cuyo punto de partida es Lii = 1

Macroalgoritmo para cada uno de los siguientes casos de factorización:

```
function MatrixFactorization(A,b,n)
    L,U = FactorizationLU(A,n)
```

```
    z = ProgressiveSustitution(L,b)
    x = RegressiveSustitution(U,z)
    return x
EndFunction
```

3.1.2.4. Factorización LU Cholesky

Es uno de los puntos de partida para la factorización de matrices, teniendo como fin encontrar los valores de Lij y Uij tales que el producto de las matrices calculadas sea igual a la matriz A. En este punto de partida, la diagonal Lii será igual a la diagonal Uii.

Algoritmo:

```
function cholesky(A, b)
    n = length(A)
    L = matrix(n, n)
    U = matrix(n, n)
    phases = vector(n)
    for k = 1 to n do
        thread = Thread(diagonal_operation_async(k))
        thread.start
        thread.join

        if L[k][k] = 0 then
            return "zero division"
        EndIf

        threads = vector(n)
        for i = k + 1 to n do
            thread = Thread(row_operation_async(k, i))
            threads.add(thread)
            thread.start
        EndFor

        for i = 1 to n do
            threads[i].join
        EndFor

        threads.clear
        for j = k + 1 to n do
            thread = Thread(column_operation_async(k, j))
```

```
            threads.add(thread)
            thread.start
        EndFor
        for i = 1 to n do
            threads[i].join
        EndFor

        if k < n - 1 then
            phase = vector(L, U)
            phases.add(phase)
        EndIf
    EndFor
    x = solve_x(L, U, b)
    return U, L, x, phase
EndFunction

function diagonal_operation_async(k)
    incr = 0
    for p = 1 to k do
        incr = incr + L[k][p] * U[p][k]
    EndFor
    L[k][k] = sqrt(A[k][k] - incr)
    U[k][k] = L[k][k]
EndFunction

function row_operation_async(k, i)
    incr = 0
    for r = 1 to k do
        incr = incr + L[i][r] * U[r][k]
    L[i][k] = (A[i][k] - incr) / L[k][k]
EndFunction

function column_operation_async(k, j):
    incr = 0
    for s = 1 to k do
        incr = incr + L[k][s] * U[s][j]
    U[k][j] = (A[k][j] - incr) / L[k][k]
EndFunction
```

3.1.2.5. Factorización LU Crout

Es uno de los puntos de partida para la factorización de matrices, teniendo como fin encontrar los valores de Lij y Uij tales que el producto de las matrices calculadas sea igual a la matriz A. En este punto de partida, la diagonal Uii será igual a 1 y se busca hallar tanto la matriz L completa como lo que resta de la matriz U.

Algoritmo:

```
function crout( A, b)
   n = length(A)
   L = matrix(n, n)
   U = matrix(n, n)
   phases = vector(n)

   for k = 1 to n do
       threads = vector(n)
       for i = k to n do
           thread = Thread(row_operation_async(k, i))
           threads.add(thread)
           thread.start
       EndFor

       for i = 1 to n do
           threads[i].join
       EndFor

       if L[k][k] = 0 then
           return "zero division"
       EndIf

       threads.clear
       for j = k to n do
           thread = Thread(column_operation_async(k, j))
           threads.add(thread)
           thread.start

       for i = 1 to n do
           threads[i].join
       EndFor

       if k < n - 1 then
           phase = vector(L,U)
```

```
            phases.add(phase)
        EndIf
    EndFor
    x = solve_x(L, U, b)
    return L,U,x,phases
EndFunction

function row_operation_async(k, i)
    incr = 0
    for p = 1 to k do
        incr = incr + L[i][p] * U[p][k]
    EndFor
    L[i][k] = A[i][k] - incr
EndFunction

function column_operation_async(k, j)
    incr = 0
    for p = 1 to k do
        incr = incr + L[k][p] * U[p][j]
    EndFor
    U[k][j] = (A[k][j] - incr) / L[k][k]
EndFunction
```

3.1.2.6. Factorización LU Doolittle

Es uno de los puntos de partida para la factorización de matrices, teniendo como fin encontrar los valores de Lij y Uij tales que el producto de las matrices calculadas sea igual a la matriz A. En este punto de partida, la diagonal Lii será igual a 1 y se busca hallar tanto la matriz U completa como lo que resta de la matriz L.

Algoritmo:

```
function doolittle(A, b)
    n = length(A)
    L = matrix(n, n)
    U = matrix(n, n)
    phases = vector(n)
    for k = 1 to n do
        threads = vector(n)
        for j = k to n do
            thread = Thread(column_operation_async(k, j))
            threads.add(thread)
```

```
            thread.start
        EndFor

        for i = 1 to n do
           threads[i].join
        EndFor

        if U[k][k] = 0 then
            return "Zero division"
        EndIf

        threads.clear
        for i = k to n do
            thread = Thread(row_operation_async(k, i))
            threads.add(thread)
            thread.start
        EndFor

        for i = 1 to n do
           threads[i].join
        EndFor

        if k < n - 1 then
           phase = vector(L,U)
           phases.add(phase)
        EndIf
    EndFor
    x = solve_x(L, U, b)
    return L,U,x,phases
EndFunction

function column_operation_async(k, j)
    incr = 0
    for p = 1 to k do
        incr = incr + L[k][p] * U[p][j]
    EndFor
    U[k][j] = (A[k][j] - incr)
EndFunction

function row_operation_async(k, i)
```

```
   incr = 0

   for r = 1 to k do
       incr = incr + L[i][r] * U[r][k]
   EndFor
   L[i][k] = (A[i][k] - incr) / U[k][k]
EndFunction
```

Métodos iterativos

Son una generalización del método de punto fijo presentado en el numeral 3.1.1.4. Se comportan de forma tal que, a partir de un vector x0 (de valores iniciales), en cada iteración se halla una aproximación que se espera sea más cercana al valor real que la anterior.

Para hallar la ecuación x = G(x) se despejan todas las x del vector cuyos coeficientes se espera que sean los más grandes posibles, evitando así divisiones por cero.

3.1.2.7. Jacobi

Aquí, los valores del vector x de cada iteración se calculan con los mismos de la iteración inmediatamente anterior.

Algoritmo:

```
jacobi A, b, iterations, tol, l, x, is_rel_error
if tol < 0 then
   return 'Tolerance must be greater or equals than 0'
else if iterations <= 0 then
   return 'Iterations must be greater than 0'
else
   m = length(x)
   n = 1
   error = tol + 1
   result = vector(m)
   while error>tol and n < iterations do
       xnew = vector(m)
       threads = vector(m)
       for i = 1 to m do
            thread = Thread(solve_new_x_async(A, b, i, x, xnew, l,
m))
           threads[i] = thread
           thread.start
       EndFor
       for t=1 to m do
           threads[t].join
```

```
        EndFor

        error = solve_error(typeError, xold, x)
        xnew.add(error)
        result.add(xnew)
        x = xnew
        n+=1
    EndWhile

    if error < tol then
        return result
    else
        return 'Solution failed for n={n} iterations'
    EndIf
EndIf

function solve_new_x_async( A, b, i, xi, xn, lamb, n)
    j = 0
    den = 1
    sum = b[i]
    while den != 0 and j < n do
        if j != i then
            sum = sum A[i][j] * xi[j]
        else
            den = A[i][j]
        j += 1
        EndIf
    EndWhile
    if den != 0 then
        xn[i] = (lamb*(sum/den) + (1-lamb)*xi[i])
    else
        return "Zero division"
    EndIf
EndFunction
```

3.1.2.8. Gauss-Seidel

En función de encontrar una aproximación más rápido, gauss-seidel utilizará para calcular cada iteración, los valores del vector x más nuevos posibles.

Algoritmo:

```
read A, b, iter, tol, l, xi, typeError
if tol < 0 then
   return "Tolerance must be greater or equals than 0"
else if iterations <= 0 then
   return 'Iterations must be greater than 0'
else if l < 0 or l > 1 then
   return “l must be between 0 and 1”
else
   m = length (x)
   xold = vector()
   error = solve_error(typeError, xold, x)
   n=1
   while (error > tol) and (n < iterations) do
       xold = vector()
       for i -> 0 to m step 1 do
           add x[i] to xold
           x[i] = solve_new_x(i,x,l,A,b)
       EndFor
       error = solve_error(typeError, xold, x)
       n+=1
   EndWhile
   if error < tol then
        return 'The solution was successful with a tolerance= tol
and n iterations'
   else do
       return 'Solution failed for n={n} iterations'
   EndIf
EndIf

function solve_new_x(i, arrX, l, A, b)
   n = length(A)
   den = 1
   incr = b[i]
   j = 0
   while j < n and den != 0 do
       if i = j then
           den = A[i][j]
       else
           incr = incr + (-1) * A[i][j] * arrX[j]
       EndIf
```

```
        j = j + 1
    EndWhile
    if den = 0 then
        return "zero division"
    else
        xn  = incr / den
        xn = l * xn + (1 - l) * arrX[i]
        return xn
    EndIf
EndFunction
```

3.1.3. Métodos de interpolación

Los métodos de interpolación permiten modelar el comportamiento de un fenómeno matemático por medio de un conjunto de puntos adquiridos con anterioridad.
Para el proyecto hemos desarrollado tres formas de adquirir el polinomio interpolante.

3.1.3.1. Lagrange

Es uno de los métodos propuestos para hallar un polinomio interpolante de grado n que pase por n + 1 puntos dados. Su forma general es p(x) = L(x0)f(x0) + L(x1)f(x1)... + L(xn)f(xn).

Algoritmo:

```
read points, x
n = length (points)
res = 0
den = 1
i = 0
while i < n and den != 0 do
    num = 1
    den = 1
    for j -> 0 to n step 1 do
        if j != i:
            num = num * (x - points[j][0])
            den = den * (points[i][0] - points[j][0])
        EndIf
    EndFor
    if den != 0 then
        Lx = num / den
        LxFx = Lx * points[i][1]
```

```
            res = res + LxFx
        EndIf
    if i = n then
        show "The method was successfully executed"
        show res
    else do
        show "There is a zero division"
    EndIf
```

3.1.3.2. Newton con diferencias divididas

Al igual que el método de Lagrange, Newton busca encontrar un polinomio interpolante de grado n que pase por n + 1 puntos descritos. Tiene la forma p(x) = B0 + B1(x - x0) + B2(x - x1)(x - x0)... + Bn(x - x0)...(x - xn-1). Las diferencias divididas permitirán tener una aproximación cercana a los valores de Bi de forma ágil evitando cálculos complejos.

Algoritmo:

```
function newton(points, x)
    n = length (points)
    res = 0
    x_b = array()
    b = array()
    result = 0
    i = 0
    while i < n and result != "error, zero division" do
        add point[i] to x_b
        result = self.solve_b(x_b)
        if result != "error, zero division" then
            prod = 1 #productoria
            for j -> 0 to i step 1 do
                prod = prod * (x-points[j][0])
            EndFor
            tx = b * prod #término x del polinomio
            res = res + tx
        EndIf
    EndWhile
    if i = n then
        show "The method was successfully executed"
        show res
    else do
```

```
        show "There is a zero division"
    EndIf
EndFunction

function solve_b(x_b):
    if length (x_b) = 1:
        return  x_b[0][1] #fx
    else:
        den = x_b[0][0] - x_b[-1][0]
        if den != 0 then
            b1 = solve_b (from x0 to xn-1)
            b2 = solve_b (from x1 to xn)
            b = (b1 - b2) / den
            return b
        else do
            return "error, zero division"
        EndIf
    EndIf
EndFunction
```

3.1.3.3. Trazadores

Con el método de splines o trazadores, se busca hallar un polinomio interpolante dado por una función polinómica que se define por tramos donde cada uno de ellos es un polinomio.

Algoritmo:

```
read points
n = length (points)
if n <= 2 then
    show 'Points must be greater than 2'
else do
    A = list()
    b = list()
    eq1 = n-1 #primeras ecuaciones
    k = 0
    while k < eq1 do
        for i -> 0 to 2 step 1 do
            ec = array() #ecuacion
            for j -> 0 to eq1 step 1 do
                if j = k then
```

```
                    if i = 0 then
                            x = [points[k][0]**3, points[k][0]**2,
points[k][0], 1]
                    else:
                                            x  =  [points[k+1][0]**3,
points[k+1][0]**2, points[k+1][0], 1]
                    ec = ec + x
                    EndIf
                else:
                    x = [0, 0, 0, 0]
                    ec = ec + x
                EndIf
            EndFor
            add ec to A, add points[k+i][1] to b
        EndFor
        k = k + 1
    EndWhile
    eq2 = n-2 #ecuaciones de las derivadas
    k = 0
    while k < eq2 do
        der1 = array() #ecuacion de primera derivada
        der2 = array() #ecuacion de segunda derivada
        for j -> 0 to eq1 step 1 do
            if j = k then
                x = [3*points[k+1][0]**2, 2*points[k+1][0], 1, 0]
                der1 = der1 + x
                x = [6*points[k+1][0], 2, 0, 0]
                der2 = der2 + x
            else if j = k+1 then
                 x = [-3*points[k+1][0]**2, -2*points[k+1][0], -1,
0]
                der1 = der1 + x
                x = [-6*points[k+1][0], -2, 0, 0]
                der2 = der2 + x
            else do
                x = [0, 0, 0, 0]
                der1 = der1 + x
                der2 = der2 + x
            EndIf
        EndFor
```

```
            add der1 to A, add der2 to A, add 0 to b, add 0 to b
            k = k + 1
        EndWhile
        #condiciones artificiales
        x = [6*points[0][0], 2, 0, 0]
        for j -> 0 to eq1-1 do
            x = x + [0, 0, 0, 0]
        EndFor
        add x to A, add 0 to b
        x = array()
        for j -> 0 to eq1-1 do
            x = x + [0, 0, 0, 0]
        EndFor
        x = x + [6*points[n-1][0], 2, 0, 0]
        add x to A, add 0 to b
        x = calculate_x(A,b)
        show "The method was executed successfully"
        show A,b,x
    EndIf
```

3.1.4. Valor agregado

A continuación, se presentan los métodos adicionales a la práctica presentados como valor agregado.

3.1.4.1. Converter

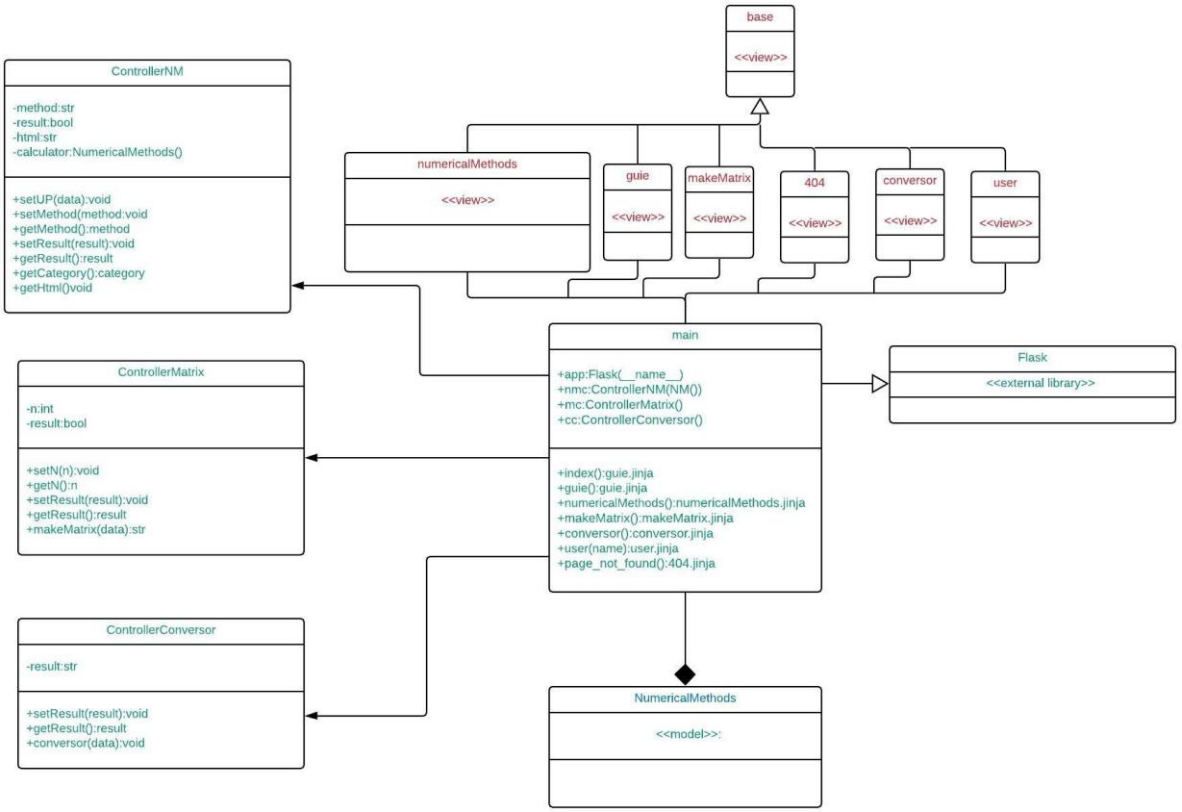
Un programa que permite transformar números en cualquier base entre 2 y 10 a otro número en el mismo rango de bases.

3.2. Arquitectura de Software

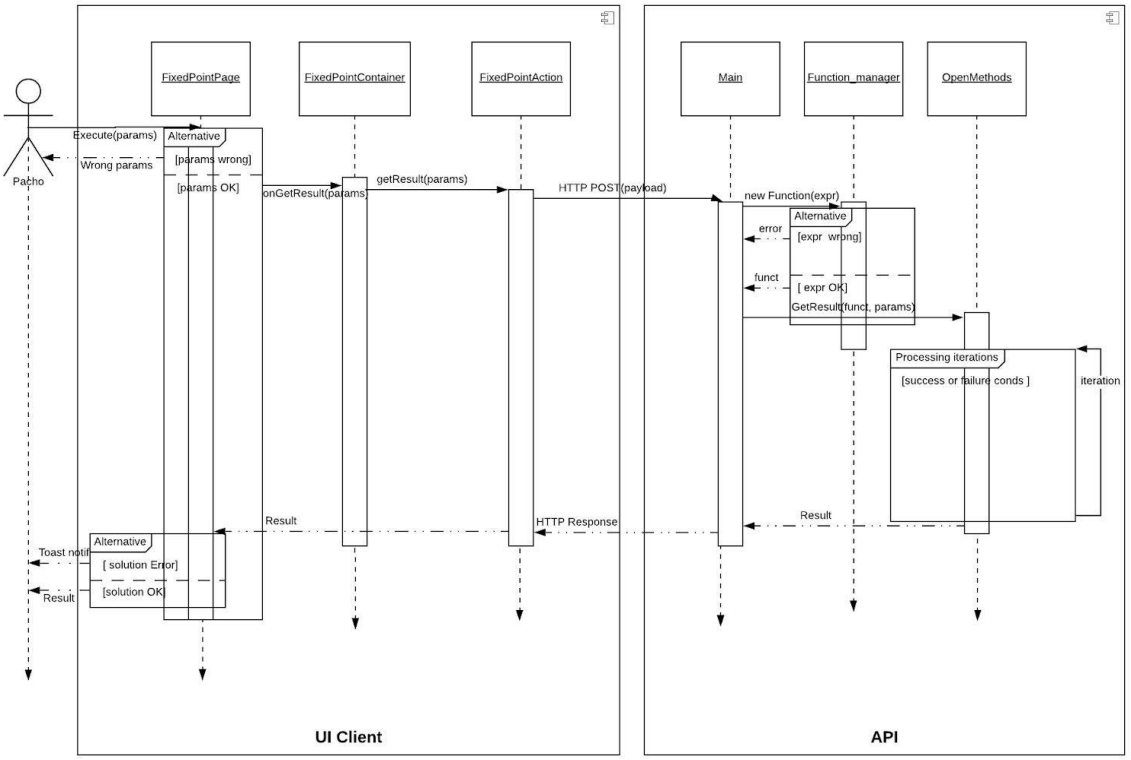
Para la implementación del proyecto, se decidió optar por un entorno web. Esto en busca de no comprometer el rendimiento de los métodos a la espera de los recursos de hardware que el usuario pueda tener.

A continuación, se presentan los diagramas correspondientes a la documentación de una arquitectura cliente-servidor basada en el MVC (Modelo-Vista-Controlador.

3.2.1 Diagrama de clases



3.2.2 Diagrama de secuencias



3.3. Gestión del proyecto

El proyecto se gestionó desde GitHub.com

4. Conclusiones

- Una de las principales habilidades que debemos adquirir al momento de hacer un proyecto de software es la comunicación, tanto con los miembros del equipo como con el tutor encargado. Ésta se verá reflejada en otros aspectos como la coordinación, la asignación de tareas, el cumplimiento de tiempos, la precisión entre lo que se espera y lo que se entrega, además aspectos. En este caso el proyecto fue realizado por una sola persona y se notó la importancia que adquiere un equipo con el cual trabajar.

- El uso de herramientas cuya curva de aprendizaje es alta, nos permite agilizar el proceso de especialización y mejora de los componentes que vamos desarrollando a lo largo del ciclo de implementación.

- Escoger un lenguaje de programación como Python nos ha permitido simplificar el desarrollo de componentes que hubieran podido ser mucho más complejos en otros lenguajes con menor soporte. El enriquecimiento que ofrece este lenguaje con la gran capacidad que tiene el framework Flask, nos permite a los desarrolladores llegar a grandes producciones.

5. Referencias

- Correa F. J. (2010) *Métodos numéricos* (primera edición), Medellín, Colombia. Fondo editorial Universidad EAFIT.