EAFIT UNIVERSITY
IT AND SYSTEMS DEPARTMENT
CHOICE OF THE PROJECT
PSEUDOCODES

Course: Numerical analysis.
Teacher: Edwar Samir Posada Murillo.
Semester: 2020-2.
Project name: Numerical views.
Project Repository:Link Repo1Link Repo2

Members:

Mariana Ramírez Duque (marami21@eafit.edu.co)

Nicolás Roldán Ramírez (nroldanr@eafit.edu.co) Mateo

Sánchez Toro (msanchezt@eafit.edu.co) Maria Cristina

Castrillon (Mcastri6@eafit.edu.co)

Report description:We will write the pseudocode of each method. The code can be found on the repository link cited above

1 Methods:

Algorithm 1: Incremental search

input : f: function to find root of, X0:
first root approximation,
Delta: delta(x),
N: maximum number of iterations,

if *(y=0)* **then**
  x0 is root;
else
  $x1 \leftarrow [x0$
  $delta]$ $y1 \leftarrow [f$
  $(x1)]$ $counter$
  $\leftarrow [1]$
while $y1 * y0 > 0$ *and counter* $< n$ *do***do**
  $x0 \leftarrow [x1]$
  $y0 \leftarrow [y1]$
  $x1 \leftarrow [x0 + delta]$
  $y1 \leftarrow [f(x1)]$
  $counter \leftarrow counter + 1$
if *(y1==0)* **then**
  x1 is root;
else
  if $(y1 * y0 < 0)$ **then**
    [x0,x1] define an interval;
  else
    print("didn't find interval in N iterations")

Algorithm 2: Bisection

input : xi: lower limit of the interval, xs:
the upper range,
f: function to find,
N: maximum number of iterations, tol:
tolerance,

$yi \leftarrow f(xi)$
$ys \leftarrow f(xs)$
if *(yi==0)* **then**
  print(”xi is root”)
else
  if *(ys==0)* **then**
    print(”xs is root”)
else
  if $(yi * ys > 0)$ **then**
    print(”Inappropriate range”)
else
  $xm \leftarrow (xi + xs)/2$
  $ym \leftarrow f(xm)$
  $error \leftarrow tol + 1$
  $counter \leftarrow [1]$
while $ymf = 0$ *and* $(error > tol)$*and*$(contador < n$ *do* **do**
$(yi * ym > 0)$ $xi \leftarrow xm, yi \leftarrow ym$
else
  $xs \leftarrow xm, ys \leftarrow ym$
  $xaux \leftarrow xm$
  $xm \leftarrow xi + xs/2$
  $ym \leftarrow f(xm)$
  $error \leftarrow xm - xaux$
  $counter \leftarrow +1$
if $(ym == 0)$ **then**
  print(”xm is root”)
else
  if $(error < tol)$ **then**
    print(”Xm is root with an error equal to error”)
else
  print(”The method failed”)

Algorithm 3: False Rule

```
input : a: lower limit of the interval, b:
        the upper range,
        f: function to find,
        N: maximum number of iterations, tol:
        tolerance,
ya ← f(a)
yb ← f(b)
if (ya==0) then
    print(”a is root”)
else
    if (yb==0) then
        print(”b is root”)
else
    if (ya * yb > 0) then
        print(”Inappropriate range”)
else
    p ← a − f(a) * (b − a)/f(b) − f(a)
    yp ← f (p)
    error ← tol +1
    counter ← [1]
while ypf = 0 and (error > tol)and(contador < n do do
(ya * yp > 0)  a ← p, ya ← yp
else
    b ← p, yb ← yp
    paux ← p
    p ← a − f(a) * (b − a)/f(b) − f(a)
    yp ← f(p)
    error ← p − paux
    counter ← +1
if (yp == 0) then
    print(”p is root”)
else
    if (error < tol) then
        print(”p is root with an error equal to error”)
else
    print(”The method failed”)
```

Algorithm 4: Multiple roots

input : f: function to find root of, fprime: derivative of f, f2prime: second derivative of f, tol: error tolerance,
N: maximum number of iterations,
X0: first root approximation

output: root approximation x

x ← $x0$
fun ← (x)
funprime ← $fprime(x)$
fun2prime ← $f2prime(x)$
error ← $infinity$

for i ← 0 **to** N **do**
  if *error* ≤ *tol(i,i) = 0* **then**
    break;
  error ← xx ← $x0$
  fun ← $f(x)$
  funprime ← $fprime(x)$
  fun2prime ← $f2prime(x)$
  error ← $infinity$
  error ← $abs(error - x)$

Algorithm 5: Gaussian elimination

input : nxn matrix A, column vector b

output: solution vector x

if *(A is not square) or (size of A and size of b are not computable)* **then**
  break ;
if *det(A) = 0* **then**
  break ;
A ← $[A\ b]$
for i ← 1 **to** $n - 1$ **do**
  if *A(i,i) = 0* **then**
    fund l such that $\bar{A}$(l,i) = 0 ;
    swith A(i) and A(l)
  for $j \leftarrow i + 1$ **to** n **do**
    multiplier $M_{j,i} \leftarrow \frac{\bar{A}(j,i)}{\bar{A}(i,i)}$
    $\bar{A}_j \leftarrow \bar{A}_J - M_{j,i} * \bar{A}_i$
x ← susreg($(\bar{A})$)

Algorithm 6: Gaussian elimination with partial pivot

input : square n x n matrix A, n vector b

output: solution vector x

if *det(A) == 0* **then**

  break ;

for $i \leftarrow 1$ **to** $n - 1$ **do**

  champion $\leftarrow i$

  for $j \leftarrow champion + 1$ **to** $n - 1$ **do**

    if *abs(A(j,i)) == abs(A(champion,i))* **then**

      champion $\leftarrow i$

  Swap row in A(champion) with A(i)

  Swap value in b(champion) with b(i)

  for $j \leftarrow i + 1$ **to** $n - 1$ **do**

    $multiplicand \leftarrow \frac{A(j,i)}{A(i,i)}$

    $A_j \leftarrow A_J - M_{j,i} * A_i$

$x \leftarrow$ susreg(A)

Algorithm 7: Gaussian elimination with total pivot

input : square n x n matrix A, n vector b

output: solution vector x

if *det(A) == 0* **then**

  break ;

posStamp $\leftarrow [from 0 to n - 1]$**for** $i \leftarrow 1$ **to** $n - 1$ **do**

  row $\leftarrow i$

  col $\leftarrow i$

  for $j \leftarrow row$ **to** $n - 1$ **do**

    for $k \leftarrow col$ **to** $n - 1$ **do**

      if *abs(A(i,j)) == abs(A(row,col))* **then**

        row $\leftarrow i$ col $\leftarrow j$

  if *col != i* **then**

    Swap column A(,col) with A(,i) Swap value in posStamp(col) with posStamp(i)

  if *col != i* **then**

    Swap row A(row) with A(i) Swap value in b(row) with b(i)

  for $j \leftarrow i + 1$ **to** $n - 1$ **do**

    $multiplicand \leftarrow \frac{A(j,i)}{A(i,i)}$

    $A_j \leftarrow A_J - M_{j,i} * A_i$

$x \leftarrow$ susreg(A)

Sort x using posStamp

Algorithm 8: Fixed Point

input : f(x), g(x), x0, tol, N

output: The approximate root

```
Define variables
int i = 0; double
x = 0;
Start While (AbsoluteValue (f (x))¿ = tol i ¡= N) x =
g(x);
i ++;
End While
Start if (AbsoluteValue (f (x)) ¡1E-8)
print ”Root:” + x;
else
print ”Not possible to obtain root”
End if
```

Algorithm 9: Secant

input : iter, xi, f(xi),E

output: The approximate root

```
I = f (Xo)
If I = 0 Then Show:
”Xo is Root” Y1 = f
(X1)
Counter = 0
Error = Tolerance + 1
While
X2 = X1 - ((Y1 * (X1 Xo)) / Den)
Error = Abs ((X2 - X1) / X2) Xo
= X1
I = Y1 X1
= X2
Y1 = f (X1)
Counter = Counter + 1 End
while
If Y1 = 0 Then
Show: ”X1 is Root”
Otherwise If Error ¡Tolerance Then
Show: “ X1 ’is an approximate root with a tolerance‘ Tolerance ” Otherwise If
Den = 0 Then
Show: ’There is possibly a multiple root’
```

Secant
Newton

Algorithm 10: Newton

---

input : f(x), g(x), x0, tol, N

output: The approximate root

if fx == 0:
return the root
end if
if dfun == 0:
return Error, derivative cannot be 0. end
if
counter = 0
error = tolerance + 1
while error¿ tol and fx! = 0 and dfx! = 0 and counter ¡iterations: xn = xi
- (fx / dfx)
fx = fun.evaluate2 (xn) dfx
= dfun.evaluate2 (xn) end
while
if type$_e$*rror* == 0 :
error = — xn-xi — else:
error = — (xn-xi) / xn —
end if
xi = xn
counter +1 if
fx == 0:
return xi is root
elif error ¡tol:
return xn is an approximation to a root with a tolerance ” elif dfx
== 0:
return xn is a possible multiple root else:
return The method failed in n iterations
end if

Extra method Aikten Extra

method Steffensen

Algorithm 11: Extra method Aikten

input : function f, float tolerance, integer maxIterations

output: solution vector x

```
x ← f(1)
x1 ← f(2)
x2 ← f(3)
aikten1 ←1 aikten2    0
for i ← 1 to maxIterations do
    if abs(aikten1 - aikten0) ¡= tolerance then
        break ;
    aikten1 ← aikten2 aikten2 ← aiktkenEcuation(x, x1, x2)
    x ← x1 x1 ← x2 x2 ← f(i + 3)
```

Algorithm 12: Extra method Steffensen

input : function f, float tolerance, integer maxIterations, float approximation

output: root of f

```
x0 ← approximation
x1 ← f(x0)
x2 ← f(x1)
x3 ← aiktkenEcuation(x0, x1,x2)
for i ← 1 to maxIterations do
    if abs(x0 - x3) ¡= tolerance then
        break ;
    x0 ← x1
    x1 ← x2
    x2 ← f(i + 3)
    x3 ← aiktkenEcuation(x0, x1, x2)
x ← x3
```

Algorithm 13: Extra method Muller

input : function f, root approximation 1 x0,root approximation 2 x1 , root approximation 3 x2, tolerance tol, Maximum number of iterations N

output: root approximation

$h1 = x1 - x0$
$h2 = x2 - x1$
$t1 = (f(x1) - f(x0))/h1$
$t2 = (f(x2) - f(x1))/h2$
$d = (t2 - t1)/(h2 + h1)$
for $i \leftarrow 3$ **to** N **do**
  $b = t2 + h2 * d$
  $D = (b^2 - 4 * f(x2) * d)^{1/2}$
  if $abs(b - D) < abs(b + D)$ **then**
    $E = b + D$;
  else
    $E = b - D$
  $h = (-2 * f(x2))/E$
  $e = x$
  $x = x2 + h$
  $e = abs(e - x)$
  if $abs(e) \leq tol)$ **then**
    break;
  $x0 = x1$
  $x1 = x2$
  $x2 = x$
  $h1 = x1 - x0$
  $h2 = x2 - x1$
  $t1 = (f(x1) - f(x0))/h1$
  $t2 = (f(x2) - f(x1))/h2$
  $d = (t2 - t1)/(h2 + h1)$

Vandermonde

Algorithm 14: Vandermonde

input : x, y

output: Polynomial coefficients, Vandermonde polynom

Repeat
Repeat
A(j,i)=X(j)**(n-i)
until($j < n$) until(i
$< n$)
print: (Reverse A)*y

Newton Divided difference

Algorithm 15: Newton Divided difference

input : vector x0, x1, ...xn, vector values f(x0); a point toevaluate

output: Divided differences F0,0 ...Fn,n

Step 1

for *i=0,...,n* **do**

  Set Fi,0= f(Xi)

Step 2

for *i=1,...,n* **do**

  for *j=1,2,....i* **do**

    Set Fi,j= ((Fi,j-1) -(Fi-1,j-1))/(Xi-Xi-j)

  End ;

Output (F0,0...,Fi,i,...Fn,n)

STOP

Algorithm 16: Lagrange

input : vector x0, x1, ...xn, vector values f(x0); a point toevaluate

output: P Lagrange polynomial P(x) evaluated at z

Step 1

Initialize variables. Set P z equal zero. Set n to the number of pairs of points (x, y).
Set L to be the all ones vector of length n

Step 2

for *i=1,2,....n* **do**

  Step 3

  for *j=1,2,....i* **do**

    Step 4

    if *i = j then Li=(z-Xj)/(Xi-Xj)*Li* **then**

      break ;

    Step 5

Pz= (*Li* ∗ *yi* +*Pz*))

Step 6 Output Pz. STOP

Lagrange
Neville

Algorithm 17: Neville

input : Numbers x0, x1, ...xn, values f(x0),f(x1)...f(xn)

output: the table Q with p(x)=Qn,n

Step 1

for *i=1,2,....n* **do**

  for *j=1,2,....i* **do**

Set Q(ij)=((x-xi-j)Qi,j-1 - (x-xi)Qi-1,j-1)/(xi-xi-j)
Step 2
Output(Q);
STOP

Simple LU

Algorithm 18: Simple LU

input : nxn matrix A, column vector b

output: solution vector x

if *(A is not square) or (size of A and size of b are not computable)* **then**

  break ;

if *det(A) = 0* **then**

  break ;

$A \leftarrow LU$
$z \leftarrow progresiveSustitution([L\ b])$
$x \leftarrow RegressiveSustitution([U\ z])$

Algorithm 19: Partial Pivoting LU

input : nxn matrix A, column vector b

output: solution vector x

if *(A is not square) or (size of A and size of b are not computable)* **then**
  break ;
if *det(A) = 0* **then**
  break ;
$PA \leftarrow LU$
$z \leftarrow progresiveSustitution([L\ P * b])$
$x \leftarrow RegressiveSustitution([U\ z])$

Algorithm 20: SOR

input : nxn matrix A, column vector b, initial aproximation X0, weighing factor w, tolerance, maximum interations Nmax

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

$A \leftarrow D - L - U$
$T \leftarrow (D - w * L)^{-1} * ((1 - w) * D + w * U)$
$C \leftarrow w * (D - w * L)^{-1} * b$
$xant \leftarrow x0$
$count \leftarrow 0$
$error = 100$
while *error > tolerance and counter < nmax* **do**
  $xact = T * xant + C$
  $error = norm(xant - xact)$
  $xant = xact;$
  $cont = cont + 1;$
$x = xact$
$iter = cont$
$err = error$

Algorithm 21: Gauss-Seidel

input : nxn matrix A, column vector b, initial aproximation X0, weighing factor w, tolerance, maximum interations Nmax

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

$A \leftarrow D - L - U$
$T \leftarrow (D - L)^{-1} * U)$
$C \leftarrow (D - L)^{-1} * b)$
$xant \leftarrow x0$
$count \leftarrow 0$
$error = 100$
while $error > tolerance$ *and* $counter < nmax$ **do**
  $xact = T * xant + C$
  $error = norm(xant - xact)$
  $xant = xact;$
  $cont = cont + 1;$
$x = xact$
$iter = cont$
$err = error$

Algorithm 22: Jacobi

input : nxn matrix A, column vector b, initial aproximation X0, weighing factor w, tolerance, maximum interations Nmax

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

$A \leftarrow D - L - U$
$T \leftarrow (D)^{-1} * (L + U)$
$C \leftarrow (D)^{-1} * b$
$xant \leftarrow x0$
$count \leftarrow 0$
$error = 100$
while $error > tolerance$ *and* $counter < nmax$ **do**
  $xact = T * xant + C$
  $error = norm(xant - xact)$
  $xant = xact;$
  $cont = cont + 1;$
$x = xact$
$iter = cont$
$err = error$