CAZ PRACTIC

Se da $f(x) = e^{2x} - 1.6x^2 - 2 = 0, pentru\ x \in [0.2, 2.2]$, cu eroare er=1e-4,

A. Estimati apriori a erorii (estimarea numarului de pasi necesar) pentru metoda bisectiei

B. Rezolvati ecuatia prin metoda bisectiei , a aproximarilor succesive, si a tangentei (Newton- Raphson).

Implementare de cod pentru cele 3 metode.

A. Estimare apriori a erorii

$$\frac{b-a}{2^n} < er => \frac{b-a}{er} < 2^n => n = floor\left(\frac{ln\frac{(b-a)}{er}}{ln2}\right) + 1$$

Rezulta $n = floor\left(\frac{ln(2*10^4)}{ln\,2}\right) + 1 = 15$

B. Pentru metoda bisectiei se da codul sursa.
Pentru metoda aproximarilor succesive trebuie sa scriu f(x)=0 sub forma fi(x)=x. Alternative ar fi

$$x = \frac{ln(1.6x^2+2)}{2}, x = \sqrt{\frac{e^{2x}-2}{1.6}}, x = e^{2x} - 1.6x^2 + x - 2$$

Cod c pentru metoda bisectiei.

```
#include <stdio.h>
#include <math.h>

int bis(float,float,float,float *);
float f(float);

int main(void)
{
  float rad;
  float ls=0.2, ld=2.2;
  float er=1e-4 ;

  if(bis(ls,ld,er,&rad)) {
    printf("\nRadacina este = %f",rad);
    printf("\nVerificare: f(%f)=%f",rad, f(rad));
  }
  else printf("\nEroare");
  while(1);
  return 1;
}

int bis(float ls, float ld, float er,
float *pRad)
{
   float xm;

   if(f(ls)*f(ld)>0) {return(0);}
   if(f(ls)==0) {*pRad=ls; return(1);}
   if(f(ld)==0) {*pRad=ld; return(1);}

   xm=(ls+ld)/2;
   while(fabs(ls-ld)>er && f(xm)!=0)
   {
      xm=(ls+ld)/2;
      if(f(xm)*f(ls)<0) ld=xm;
      else ls=xm;
      }
   *pRad=xm;
   return(1);
}

float f(float x) {
    return exp (2*x)-1.6*x*x-2;
}
```

2.1.1.1. Algoritmul 2.1. Şirul lui Şturm

```
void Sturm
(
intreg grad, // gradul polinomului
real P1[], // vectorul coeficienţilor polinomului
real P2[], // vectorul coeficienţilor derivatei
// polinomului;
real Rez[], // vectorul coeficienţilor restului
real C1, // pentru calculul coeficienţilor
// restului
real C2 // coeficienţii pentru calculul
// coeficienţilor restului
)
{ // corpul funcţiei
pentru i=grad-1 pana la 0 calculează
P2[i]=(i+1)P1[i+1];
daca grad=0 Rez(0)=P1(0)-P1(1);
// Restul are gradul grad-2
altfel
{
C1 = P1[grad]/P2[grad-1];
C2 = (P1[grad-1]-C1*P2[grad-2])/P2[grad-1];
pentru i= (grad-2 pana la 1) calculează
Rez(i)=P1(i)-C1*P2[i-1]-C2*P2[i];
Rez(0)=P1[0]=P1-C2*P2[0];
}
tipăreşte coeficienţii restului cu semn schimbat;
}
```

2.2.1.1. Algoritmul 2.1. Bisecţia pentru polinoame

```
// Funcţia întoarce:
// True (1) - in caz de succes (s-a găsit
rădăcina);
// False (0) - in caz de eşec.
intreg BisPol
(
intreg n, // gradul polinomului
real A[], // coeficienţii polinomului
real ls, // limita stânga a intervalului
real ld, // limita dreapta a intervalului
real er, // eroarea de aproximare.
real *pRad // adresa rădăcinii (pointer)
){
// variabile locale
real xm; // jumătatea intervalului.
// instrucţiunile ce modelează algoritmul metodei
daca (valPol(n,A,ls) * valPol(n,A,ld) > 0) return
False;
daca (valPol(n,A,ls) == 0) {*pRad = ls; return
True;}
daca (valPol(n,A,ld) == 0) {*pRad = ld; return
True;}
xm=(ls+ld)/2; // condiţia de intrare in ciclul
'while'
cat timp((fabs(ld-ls) > er) ŞI valPol(n,A,xm) !=
0)
{
xm=(ls+ld)/2;
daca (valPol(n,A,xm) * valPol(n,A,ls) <0) ld = xm;
altfel ls = xm;//se stabilesc noile limite de
// interval
}
*pRad = xm; // stabilesc valoarea rădăcinii drept
mijlocul
// ultimului interval ce satisface condiţiile
// buclei 'while()' anterioare.
return True; // succes: am găsit rădăcina .
}
```

2.2.1.2. Algoritmul 2.2. Bisecţia pentru ecuaţii transcendente

```
// functia intoarce:
// True (1) - in caz de succes (s-a găsit
rădăcina);
// False (0) - in caz de eşec.
intreg Bis_Functie
(
Adr_functie, // adresa funcţiei din ecuaţia f(x)=0
real ls, // limita stânga a intervalului de calcul
real ld, // limita dreapta a intervalului de
calcul
real er, // eroarea de calcul (precizia impusa)
real *pRad // adresa variabilei rădăcina
){
```

```
// declaraţiile variabilelor locale:
real xm; // mijlocul intervalului .
// instructiunile functiei
daca(*Adr_functie)(ls) * (*Adr_functie)(ld)>0)
return False;
daca (*Adr_functie)(ls) == 0) {*pRad = ls; return
True;}
daca (*Adr_functie)(ld) == 0) {*pRad = ld; return
True;}
xm=(ls+ld)/2; // conditia de intrare in ciclul
'while'
cat timp((fabs(ld-ls) > er) SI (Adr_functie (xm)
!= 0) )
{
xm=(ls+ld)/2;
daca ((*Adr_functie)(xm) * (*Adr_functie)(ls)<0)
ld = xm;
altfel ls = xm; // se stabilesc noile limite de
interval
}
*pRad = xm; // stabilesc valoarea radacinii drept
// mijlocul ultimului interval ce satisface
// conditiile buclei 'while()' anterioare.
return True; // succes: am gasit radacina .
}
```

Obs.: Notaţiile '*Adr_functie*' şi '*(*Adr_functie)*' sunt simbolice. Vedeţi *Anexa B-* "*Noţiuni de C necesare desfăşurării lucrărilor de laborator*".

2.2.2.1. Algoritmul 2.3. Aproximaţii succesive - ecuatii transcedente

```
// funcţia întoarce: True (1) – succes, s-a găsit
rădăcina;
// False (0) – in caz de eşec.
intreg Aprox_Succesive
(
Adr_functie, // adresa functiei fi(x)(pointer la
functie)
real ls, // limita stânga a intervalului de calcul
real ld, // limita dreapta a intervalului de
calcul
real x0, // punctul de start - aproximaţia
iniţiala
// necesara algoritmului
real er, // eroarea de calcul
real h, // pasul de derivare intre doua abscise
real *pRad // adresa variabilei radacina (pointer)
){
// declaraţiile variabilelor locale:
real pc; // punctul curent de derivare.
real xn, xn_1; // aproximatiile curentă şi
respectiv
// anterioara ale valorii radacinii.
real der; // variabila in care se retine valoarea
// derivatei funcţiei fi(x).

// instrucţiunile funcţiei
pc = ls; // pornesc de la valoarea 'ls' pentru
// verificarea derivatei functiei fi(x) pe
// intervalul dat.
// verifica daca functia fi(x) are derivata
subunitara pe
// intreg intervalul considerat.
repeta
{
der = ((*Adr_functie)(pc+h) - (*Adr_functie)(pc))
/ h;
daca (fabs(der) >=1) return False;
pc=pc+h; // trec la urmatoarea abscisa, cu un pas
cat
// mai mic.
} cat timp(pc<=ld);
xn = x0;
repeta
{
xn_1 = xn;
xn = (*Adr_functie)(xn_1);
} cat timp(fabs(xn-xn_1) >= er);
*pRad = xn; // valoarea radacinii este valoarea
lui 'xn'
// cu care se încheie ciclul imediat anterior
return True; // marchez succesul în găsirea
rădăcinii
}
```

2.2.3.1. Algoritmul 2.4. Newton-Raphson pentru ecuaţii transcendente

```
// functia intoarce: True (1) – succes, s-a gasit
radacina;
// False (0) – in caz de esec.
intreg NewtonRaphson
(
Adr_functie, // referinta la functia f(x), din
f(x)=0
real x0, // valoarea initiala necesara
algoritmului
real er, // eroare de aproximare radacinii
real h, // valoarea pasului de derivare
int nMax // numarul maxim de iteratii impus pentru
// aflarea solutiei cu eroarea 'er' .
real *pRad // adresa variabilei rădăcină
)
{ // declaratiile variabilelor locale:
real xn, xn_1; // valoarea aproximată curent si
anterior
// a radacinii
real der; // valoarea derivatei

// instructiunile metodei
xn=x0;
repetă
{
xn_1=xn;
der=((*Adr_functie)(xn_1+h)-
(*Adr_functie)(xn_1))/h;
dacă (der == 0) return False;
xn = xn_1 - (*Adr_functie)(xn_1)/der;
nMax = nMax-1; // sau: nMax--;
} cat timp((fabs(xn-xn_1)>=er) SI (nMax>0));
dacă (nMax= =0) return False;
*pRad=xn;
return True; // succes.
}
```

2.2.4.1 Algoritmul 2.5. Newton-Raphson pentru polinoame

```
// funcţia întoarce: True (1) – success, s-a găsit
rădăcina
// False (0) – in caz de eşec
intreg Newton_Raphson_P
(
intreg n, // gradul polinomului
intreg nMax, // numărul maxim impus de iteraţii
real A[], // vectorul coeficienţilor polinomului
real x0, // valoarea de start
real er, // eroarea de calcul
real *pRad // rădăcina ecuaţiei
)
{ // declaraţiile variabilelor locale
real xn, xn_1; // valoarea curentă respectiv
anterioara a
// rădăcinii

xn = x0;

repetă
{
xn_1=xn;
dacă (Derpol(n,A,xn_1)==0) return False;
xn=xn_1-Valpol(n,A,xn_1)/Derpol(n,A,xn_1);
nMax=nMax-1; // sau: nmax--;
} cât timp((fabs(xn-xn_1)>=er) ŞI (nMax>0) );
dacă (nMax == 0)
return False; // numarul de pasi este insuficient
*pRrad = xn;
return True;
}
```