Aproximaciones

Luis Eduardo Robles Jimenez

0224969

Input

In [ ]:
```
Matrix = [
    [-5, 5, 3],
    [5, 6, 1],
    [3, 1, 7]
]
Independent = [1, 2, 3]
```

Method

In [ ]:
```
x = GaussSeidel(Matrix, Independent, 2000, 0.00000000001)
print(x)
```

Gauss-Seidel

In [ ]:
```
def GaussSeidel(m, it, n, e):
    print(len(m), "x", len(m), " System:\n", sep = "", end = "")
    for i in range(len(m)):
        print("\t[", end = "")
        for j in range(len(m[i])):
            print(m[i][j], end = " ")
        print("= ", it[i], "]", sep = "")
    print()
    x = [0 for _ in range(len(m))]
    for i in range(len(m)):
        d = m[i][i]
        for j in range(len(m[i])):
            m[i][j] /= d
        it[i] /= d

    for i in range(len(m)):
        s = it[i]
        for j in range(len(m[i])):
            if i != j:
                s -= m[i][j]*x[j]
        x[i] = s
    for _ in range(n):
        print("x[", _, "] = ", x, sep = "")
        c = 1
        for i in range(len(m)):
            o = x[i]
            s = it[i]
            for j in range(len(m[i])):
                if i != j:
                    s -= m[i][j]*x[j]
            x[i] = s
            if c and x[i]:
                error = 100*abs((x[i]-o)/x[i])
                if error > e:
                    c = 0
        if c: break;
    return x
```

Aproximación

Luis Eduardo Robles Jimenez

0224969

Input

In [ ]:
```
#expression = "2*x**3 - 4*log(x)"
#expression = "x**(1/3)"
expression = "x**3*log(x)"
```

Method

In [ ]:
```
f = Taylor(expression, 3, 2)
print("\n", f, "\n")
print("f(x) ≈", N(f.subs(x, 2.1)))
```

Taylor

In [ ]:
```
def Taylor(function, order, a = 0):
    d, fS = [parse_expr(expression)], ""
    for i in range(order + 1):
        if i > 0:
            d.append(diff(d[i-1], x))
            fS += " + "
        print("\t", d[i])
        fS += str(N(d[i].subs(x, a))) + "*(x - " + str(a) + ")**" + str(i) + "/(" + str(i) + "!)"
    return collect(expand(parse_expr(fS)), x)
```

Run First

In [ ]:
```
from sympy import *
x = symbols("x")
```

```cpp
#include <bits/stdc++.h>
using namespace std;
using ll = long long;
int main(){
  string a;
  ll r = 0;
  cin >> a;
  int n = a.size() - 1;
  for(int i = 0; i < n + 1; i++){
    a[n - i] -= '0';
    r += a[n - i] * (1 << i);
  }
  cout << r;
  return 0;
}

```

```
#include <bits/stdc++.h>
using namespace std;
string ntoB(){
  int n;
  cin >> n;
  string s = "";
  while(n){
    s = to_string(n % 2) + s;
    n /= 2;
  }
  return s;
}
long long btoD(string s){
  string s;
  cin >> s;
  int n = s.size();
  long long res = 0, m = 1;
  for(int i = n - 1; i >= 0; i--, m <<= 1)
    if(s[i] == '1') res += m;
}
int main(){
  return 0;
}
```

Aproximación de un Polinomio Característico

Luis Eduardo Robles Jiménez

0224969

Input

In [ ]:
```
#matrix = np.array([[3, 1, 5], [3, 3, 1], [4, 6, 4]]) #1, -10, 4, -40
#matrix = np.array([[3, 2, 4], [2, 0, 2], [4, 2, 3]]) #1, -6, -15, -8
matrix = np.array([[1, -1, 4], [3, 2, -1], [2, 1, -1]])
#matrix = np.array([[5, -2, 0], [-2, 3, -1], [0, -1, 1]])
```

Method

In [ ]:
```
Leverrier_Faddeev(matrix)
```

In [ ]:
```
Krilov(matrix, np.array([1, 0, 0]))
```

Krilov

In [ ]:
```
def Krilov(A, y = np.ones(0)):
    n = A.shape[0]
    b = np.empty((n, n))
    if y.size == 0: y = np.ones(n)
    b[0] = y
    print("Matrix:\n\n", A, "\n\nUsing vector:\n\n", y, "\n\nVectors calculated:\n")
    for i in range(1, n): b[i] = A @ b[i-1]
    print(b)
    a, s = np.linalg.solve(np.transpose(b), A @ b[n-1]), "λ^" + str(n)
    for i in np.flip(a):
        n -= 1
        s += " + " + str(-i) + "λ^" + str(n)
    return s
```

Leverrier Faddeev

In [ ]:
```
def Leverrier_Faddeev(A):
    print("Matrix:\n\n", A, "\n\n")
    n = A.shape[0]
    b, B, i = np.empty(n+1), np.empty((n+1, n, n)), np.identity(n)
    b[n], B[0] = 1, np.zeros((n, n))
    for k in range(1, n+1):
        B[k] = (A @ B[k-1]) + (b[n-k+1] * i)
        b[n-k] = -np.trace(A @ B[k])/k
    s = ""
    n += 1
    for i in np.flip(b):
        n -= 1
        if len(s): s += " + "
        s += str(i) + "*λ^" + str(n)

    return s
```

Run first

In [ ]:
```
import numpy as np
from sympy import *
x, lmbd = symbols("x"), symbols("lambda")
```

```cpp
#include <bits/stdc++.h>
using namespace std;
using ll = long long;
int main(){
  double a;
  cin >> a;
  ll x = a;
  for(int i = 15; i >= 0; i--)
    cout << (bool)(x & (1 << i));
  if((ll)a != a){
    cout << ".";
    for(int i = 0; i < 10; i++){
      a -= (ll)a;
      a *= 2;
      cout << (ll)a;
      if((ll)a) a--;
    }
  }
  return 0;
}
```

Aproximación de Ecuaciones Diferenciales

Luis Eduardo Robles Jiménez

0224969

Input

In [ ]:
```
#yp, a, b, n, c = 'y - t**2 + 1', 0, 2, 10, 0.5
#yp, a, b, n, c = '-2*t**3 + 12*t**2 - 20*t + 8.5', 0, 4, 8, 1
#yp, a, b, n, c = 'y - t**2 + 1', 0, 2, 10, 0.5
yp, a, b, n, c = '-5*y + 5*t**2 + 2*t', 0, 1, 10, 1/3
```

Method

In [ ]:
```
Euler(yp, a, b, n, c)
```

In [ ]:
```
RunggeKutta(yp, a, b, n, c)
```

Euler

In [ ]:
```
def Euler(fun, a, b, n, c):
    f = parse_expr(fun)
    print("\tf(x) =", f, end = "\n\n")
    h = (b - a)/n
    tT, yV, p = a, c, []
    for i in range(1, n+1):
        print(tT, yV, sep = "\t")
        yV += h*N(f.subs([(t, tT), (y, yV)]))
        tT += h
        p.append(yV)
    return (tT, yV)
```

Rungge Kutta (Cuarto Grado)

In [ ]:
```
def RunggeKutta(fun, a, b, n, c):
    f = parse_expr(fun)
    print("\tf(x) =", f, end = "\n\n")
    h = (b - a)/n
    tT, yV, p = a, c, []
    for i in range(n):
        ku = h*N(f.subs([(t, tT), (y, yV)]))
        kd = h*N(f.subs([(t, tT + h/2), (y, yV + ku/2)]))
        kt = h*N(f.subs([(t, tT + h/2), (y, yV + kd/2)]))
        kc = h*N(f.subs([(t, tT + h), (y, yV + kt)]))
        yV += (ku + 2*kd + 2*kt + kc)/6
        tT += h
        p.append(yV)
        print(tT, yV, sep = "\t")
    return (tT, yV)
```

Run first

In [ ]:
```
from sympy import *
t, y = symbols("t"), symbols("y")
```

Aproximaciones

Luis Eduardo Robles Jimenez

0224969

Input

In [ ]:
```
#xInput = (1, 4, 6, 5)
#yInput = "ln(x)"
#xInput = (1, 4, 6)
#yInput = "ln(x)"
#xInput = (1.0, 1.3, 1.6, 1.9, 2.2)
#yInput = (0.765197, 0.6200860, 0.4554022, 0.2818186, 0.1103623)
#xInput = (1.0, 1.3, 1.6)
#yInput = (0.7651977, 0.6200860, 0.4554022)
#xInput = (8.1, 8.3, 8.6, 8.7)
#yInput = (16.94410, 17.56492, 18.50515, 18.82091)
#xInput = (1.3, 1.6, 1.9)
#yInput = (0.6200860, 0.4554022, 0.2818186)
#dInput = (-0.5220232, -0.5698959, -0.5811571)
#xInput = (8.3, 8.6)
#yInput = (17.56492, 18.50515)
#dInput = (3.116256, 3.151762)
#xInput = (0, 0.6, 0.9)
#yInput = "ln(x+1)"
#xInput = (8, 9, 11)
#yInput = "log(x, 10)"
#xInput = (8, 9, 11)
#yInput = Cloud(xInput, "log(x, 10)")
#dInput = Cloud(xInput, "1/(x*log(10))")

#HERMITE SEGUNDO EXAMEN
#xInput = (8.3, 8.6)
#yInput = (17.5649, 18.5051)
#dInput = (3.1162, 3.1517)

#NEWTON SEGUNDO EXAMEN
xInput = (8, 9, 11)
yInput = Cloud(xInput, "log(x)")
dInput = Cloud(xInput, "1/x")
#yInput = (-15, 15, -153, 291)

#LAGRANGE SEGUNDO EXAMEN
#xInput = (1, -4, -7)
#yInput = (10, 10, 34)
```

Method

In [ ]:
```
f, e = Lagrange(xInput, yInput), 10
print("\ng(", e, ") ≈ ", N(f.subs(x, e)), sep = "")
```

In [ ]:
```
f, e = Newton(xInput, yInput), 10
print("\nf(", e, ") ≈ ", N(f.subs(x, e)), sep = "")
```

In [ ]:
```
f, e = Hermite(xInput, yInput, dInput), 10
print("\nf(", e, ") ≈ ", N(f.subs(x, e)), sep = "")
```

Lagrange

In [ ]:
```
def Lagrange(xInput, yInput, p = None):
    yInput, n, s = Cloud(xInput, yInput), len(xInput), ""
    print(n, "points:")
    for i in range(n): print("\tf(", xInput[i], ") = ", yInput[i], sep = "")
    for i in range(n):
        p = str(yInput[i])
        for j in range(n):
            if i != j:
                p += "*(x - " + str(xInput[j]) + ")/(" + str(xInput[i]) + " - " + str(xInput[j]) + ")"
        s += (" + " if i else "") + p
    return showPoly(s)
```

Newton's Polynomial

In [ ]:
```
def Newton(xInput, yInput):
    yInput = Cloud(xInput, yInput)
    n = len(xInput)
    print(n, "points:")
    for i in range(n): print("\tf(", xInput[i], ") = ", yInput[i], sep = "")
    m = [[0 for i in range(n)] for j in range(n)]
    for i in range(n): m[i][0] = yInput[i]
    for j in range(1, n):
        for i in range(n - j):
            m[i][j] = (m[i+1][j-1] - m[i][j-1])/(xInput[i+j] - xInput[i])
    r, a = str(m[0][0]), ""
    for i in range(1, n):
        a += "*" + "(x-" + str(xInput[i - 1]) + ")"
        r += " + " + str(m[0][i]) + a
    return showPoly(r)
```

Hermite

In [ ]:
```
def Hermite(xInput, yInput, dInput):
    n = len(xInput)
    print(n, "points:")
    for i in range(n):
        print("\tf(", xInput[i], ") = ", yInput[i], "\tf'(", xInput[i], ") = ", dInput[i], sep = "")
    m = [[0 for i in range(2*n)] for j in range(2*n)]
    for i in range(n):
        m[2*i][0] = m[2*i+1][0] = yInput[i]
        m[2*i][1] = dInput[i]
        if i: m[2*i-1][1] = (m[2*i][0]-m[2*i-1][0])/(xInput[i]-xInput[i-1])
    for j in range(2, 2*n):
        for i in range(2*n-j):
            m[i][j] = (m[i+1][j-1] - m[i][j-1])/(xInput[int((i+j)/2)] - xInput[int(i/2)])
    r, a = str(m[0][0]), ""
    for i in range(1, 2*n):
        a += "*" + "(x-" + str(xInput[int((i - 1)/2)]) + ")"
        r += " + " + str(m[0][i]) + a
    return showPoly(r)
```

AuxFucnt

In [ ]:
```
def Cloud(xI, yI):
    if isinstance(yI, str):
        a, yI = list(), parse_expr(yI)
        for xVal in xI: a.append(N(yI.subs(x, xVal)))
        yI = tuple(a)
    return yI
def showPoly(s):
    print("\nPolynomial", s, sep = "\n")
    print("\nSimplified", simplify(parse_expr(s)), sep = "\n")
    print("\nBy Powers", r := collect(expand(parse_expr(s)), x), sep = "\n")
    return r
```

Run First

In [ ]:
```
from sympy import *
x = symbols("x")
```

Aproximaciones

Luis Eduardo Robles Jimenez

0224969

Function

In [ ]:
```
#expression = "sqrt(1 + 1/x) - x"
#expression = "x**3+5*x**2+2"
#expression = "2*sin(sqrt(x))-x"
#expression = "2 - x/2 -x**2/4"
expression = "2*x**3 - 11.7*x**2 + 17.7*x - 5"
```

Method

In [ ]:
```
NewtonRaphson(3, 0.001, 50)
```

In [ ]:
```
BinarySearch(0, 3, 0.1, 50)
```

In [ ]:
```
Secant(3, 4, 0.01, 100)
```

In [ ]:
```
FixedP(1, 0.001, 100)
```

Newton Raphson

In [ ]:
```
def NewtonRaphson(p0, e, n):
    f = parse_expr(expression)
    d = diff(f, x)
    print("\tf(x) =", f, "\n\tf'(x) =", d, "\n")
    for i in range(n):
        p = p0 - N(f.subs(x, p0))/N(d.subs(x, p0))
        error = abs(N((p - p0)/p))*100
        print(i + 1, ". ", sep = '', end = '')
        print("P =", p, "\tEr =", error)
        if error < e: return p
        p0 = p
    return p
```

Binary Search

In [ ]:
```
def BinarySearch(a, b, e, n):
    f = parse_expr(expression)
    print("\tf(x) = ", f, "\n\t[", a, ", ", b, "]", "\n", sep = "")
    fp0, p0 = N(f.subs(x, a)), a
    for i in range(n):
        p = a + (b - a)/2
        fp = N(f.subs(x, p))
        error = abs((p - p0)/p)*100
        print(i + 1, ". ", sep = '', end = '')
        print("P = ", p, "\tEr = ", error, " %", sep = '')
        if error < e: return p
        if fp * fp0 > 0: a, fp0 = p, fp
        else: b = p
        p0 = p
    return p
```

Secant

In [ ]:
```
def Secant(pa, pb, e, n):
    f = parse_expr(expression)
    print("\tf(x) =", f, "\n")
    for i in range(n):
        qa, qb = N(f.subs(x, pa)), N(f.subs(x, pb))
        pc = pb - qb*(pa - pb)/(qa - qb)
        error = abs(N((pc - pb)/pc))*100
        print(i + 1, ". ", sep = '', end = '')
        print("P =", pc, "\tEr =", error)
        if error < e: return pc
        pa, pb = pb, pc
    return p
```

Fixed Point

In [ ]:
```
def FixedP(pa, e, n):
    f = parse_expr(expression)
    print("\tf(x) =", f)
    f = parse_expr(expression + " + x")
    print("\tx =", f, "\n")
    for i in range(n):
        pb = N(f.subs(x, pa))
        if not pb: return pa
        error = abs((pb - pa)/pb)*100
        print(i + 1, ". ", sep = '', end = '')
        print("P = ", pb, "\tEr = ", error, sep = '')
        if error < e: return pb
        pa = pb
    return pb
```

Run First

In [ ]:
```
from sympy import *
x = symbols("x")
```