ANÁLISIS NUMÉRICO BÁSICO CON PYTHON V4.4

Código de la instrumentación computacional en lenguaje Python

3.1.5 Instrumentación computacional del método de la Bisección

```
def biseccion(f, a, b, e):
    while b-a>=e:
        c=(a+b)/2
        if f(c)==0:
            return c
        else:
            if f(a)*f(c)>0:
                a=c
            else:
                b=c
    return c
```

Instrumentación alternativa usando el tipo simbólico de la librería SymPy de Python

```
from sympy import*
def biseccions(f,v,a,b,e):
    while b-a>=e:
        c=(a+b)/2
        if f.subs(v,c)==0:
            return c
        else:
            if f.subs(v,a)*f.subs(v,c)>0:
                a=c
            else:
                b=c
    return c
```

3.3.6 Instrumentación computacional del método de Newton

```
from sympy import*
def newton(f, v, u, e, m):
    g=v-f/diff(f,v)
    for i in range(m):
        r=float(g.subs(v,u))
        if abs(r-u)<e:
            return r
        u=r
    return None
```

3.5.4 Instrumentación computacional del método de Newton para un sistema de n ecuaciones no-lineales

```
import numpy as np
import sympy as sp
#Resolución de Sistemas no lineales
def snewton(F, V, U):
    n=len(F)
    J=np.zeros([n,n],dtype=sp.Symbol)
    T=list(np.copy(F))

    for i in range(n):                              #Construir J
        for j in range(n):
            J[i][j]=sp.diff(F[i],V[j])
    for i in range(n):                              #Evaluar J
        for j in range(n):
            for k in range(n):
                J[i][j]=J[i][j].subs(V[k],float(U[k]))
    for i in range(n):                              #Evaluar F
        for j in range(n):
            T[i]=T[i].subs(V[j],float(U[j]))
    J=np.array(J,float)
    T=np.array(T,float)
    U=U-np.dot(np.linalg.inv(J),T)                  #Nuevo vector U
    return U
```

3.6.4 Instrumentación computacional del algoritmo del gradiente de máximo descenso

```
#Método del gradiente de máximo descenso
from sympy import*
from sympy.plotting import*
import numpy as np

def obtener_gradiente(f,v):
    n=len(v)
    g=[]
    for i in range(n):
        d=diff(f,v[i])
        g=g+[d]
    return g

def evaluar_gradiente(g,v,u):
    n=len(v)
    c=[]
    for i in range(n):
        t=g[i]
        for j in range(n):
            t=t.subs(v[j],u[j])
        c=c+[float(t)]
    return c

def magnitud_del_gradiente(c):
    norma=sqrt(np.dot(c,c))
    return norma

def gradiente_normalizado(c):
    norma=magnitud_del_gradiente(c)
    t=list(np.array(c)/norma)
    cn=[]
    for i in range(len(c)):
        cn=cn+[float(t[i])]
    return cn

def evaluar_solucion(f,v,u):
    fm=f.subs(v[0],u[0])
    for i in range(1,len(v)):
        fm=fm.subs(v[i],u[i])
    return fm
```

```
def calcular_paso(f,g,v,u):
    c=evaluar_gradiente(g,v,u)
    cn=gradiente_normalizado(c)
    t=Symbol('t')
    xt=[]
    for i in range(len(v)):
        xt=xt+[float(u[i])-t*float(cn[i])]
    fs=f.subs(v[0],xt[0])
    for i in range(1,len(v)):
        fs=fs.subs(v[i],xt[i])
    df=diff(fs,t)
    ddf=diff(df,t)
    s=1
    for i in range(5):
        s=s-float(df.subs(t,s))/float(ddf.subs(t,s))
    return s

def metodo_gradiente(f,v,u,e,m,imp=0):
    u0=u.copy()
    g=obtener_gradiente(f,v)
    for k in range(m):
        c=evaluar_gradiente(g,v,u0)
        norma=magnitud_del_gradiente(c)
        if norma<e:
            fm=evaluar_solucion(f,v,u0)
            return u0,fm
        s=calcular_paso(f,g,v,u0)
        cn=gradiente_normalizado(c)
        uk=[]
        for i in range(len(c)):
            uk=uk+[float(u0[i])-s*float(cn[i])]
        u0=uk.copy()
        if imp>0:
            print('k=',k+1,' s=',s,' vector=',u0)
    return [],None
```

4.2.4 Instrumentación computacional del método de Gauss-Jordan básico

```
#Solución de un sistema lineal: Gauss-Jordan básico
import numpy as np
def gaussjordan1(a,b):
    n=len(b)
    c=np.concatenate([a,b],axis=1)     #matriz aumentada
    for e in range(n):
        t=c[e,e]
        for j in range(e,n+1):
            c[e,j]=c[e,j]/t            #Normalizar fila e
        for i in range(n):
            if i!=e:
                t=c[i,e]
                for j in range(e,n+1):
                    c[i,j]=c[i,j]-t*c[e,j]     #Reducir otras filas
    x=c[:,n]
    return x
```

Instrumentación del algoritmo Gauss-Jordan usando notación implícita de índices

```
#Solución de un sistema lineal: Gauss-Jordan básico
import numpy as np
def gaussjordan1(a,b):
    n=len(b)
    c=np.concatenate([a,b],axis=1)                  #Matriz aumentada
    for e in range(n):
        c[e,e:]=c[e,e:]/c[e,e]                      #Normalizar fila e
        for i in range(n):
            if i!=e:
                c[i,e:]=c[i,e:]-c[i,e]*c[e,e:]      #Reducir otras filas
    x=c[:,n]
    return x
```

4.3.4 Instrumentación computacional de método de Gauss básico

```
#Solución de un sistema lineal: Gauss básico
import numpy as np
def gauss1(a,b):
    n=len(b)
    c=np.concatenate([a,b],axis=1)        #matriz aumentada
    for e in range(n):
        t=c[e,e]
        for j in range(e,n+1):            #Normalizar fila  e
            c[e,j]=c[e,j]/t
        for i in range(e+1,n):            #Reducir filas debajo
            t=c[i,e]
            for j in range(e,n+1):
                c[i,j]=c[i,j]-t*c[e,j]
    x=np.zeros([n,1])                     #Celdas para el vector X
    x[n-1]=c[n-1,n]
    for i in range(n-2,-1,-1):            #Sistema triangular
        s=0
        for j in range(i+1,n):
            s=s+c[i,j]*x[j]
        x[i]=c[i,n]-s
    return x
```

Instrumentación del método de Gauss básico usando notación implícita de índices

```
#Solución de un sistema lineal: Gauss básico
import numpy as np
def gauss1(a,b):
    n=len(b)
    c=np.concatenate([a,b],axis=1)            #Matriz aumentada
    for e in range(n):
        c[e,e:]=c[e,e:]/c[e,e]                #Normalizar fila  e
        for i in range(e+1,n):
            c[i,e:]=c[i,e:]-c[i,e]*c[e,e:]    #Reducir filas debajo
    x=np.zeros([n])
    x[n-1]=c[n-1,n]
    for i in range(n-2,-1,-1):                #Sistema triangular
        x[i]=c[i,n]-np.dot(x[i+1:n],c[i,i+1:n])
    return x
```

4.3.7 Instrumentación computacional del método de Gauss con pivoteo

```
#Solución de un sistema lineal: Gauss con pivoteo
import numpy as np
def gauss(a,b):
    n=len(b)
    c=np.concatenate([a,b],axis=1)            #Matriz aumentada

    for e in range(n):
        p=e
        for i in range(e+1,n):                #Pivoteo
            if abs(c[i,e])>abs(c[p,e]):
                p=i

        for j in range(e,n+1):               #Intercambio de filas
            t=c[e,j]
            c[e,j]=c[p,j]
            c[p,j]=t

        t=c[e,e]
        if abs(t)<1e-20:                      #Sistema singular
            return []

        c[e,e:]=c[e,e:]/c[e,e]                #Normalizar fila  e
        for i in range(e+1,n):
            c[i,e:]=c[i,e:]-c[i,e]*c[e,e:]   #Reducir filas debajo

    x=zeros([n])
    x[n-1]=c[n-1,n]
    for i in range(n-2,-1,-1):                 #Sistema triangular
        x[i]=c[i,n]-np.dot(x[i+1:n],c[i,i+1:n])
    return x
```

4.6.2 Instrumentación computacional del Método de Thomas

```
def tridiagonal(a, b, c, d):
    n=len(d)
    w=[b[0]]
    g=[d[0]/w[0]]
    for i in range(1,n):
        w=w+[b[i]-a[i]*c[i-1]/w[i-1]]
        g=g+[(d[i]-a[i]*g[i-1])/w[i]]
    x=[]
    for i in range(n):
        x=x+[0]
    x[n-1]=g[n-1]
    for i in range(n-2,-1,-1):
        t=x[i+1]
        x[i]=g[i]-c[i]*t/w[i]
    return x
```

5.1.2 Manejo computacional de la fórmula de Jacobi

```
def jacobi(a,b,x):
    n=len(x)
    t=x.copy()
    for i in range(n):
        s=0
        for j in range(n):
            if i!=j:
                s=s+a[i,j]*t[j]
        x[i]=(b[i]-s)/a[i,i]
    return x
```

5.1.4 Instrumentación computacional del método de Jacobi

```
from jacobi import*
import numpy as np
def jacobim(a,b,x,e,m):
    n=len(x)
    t=x.copy()
    for k in range(m):
        x=jacobi(a,b,x)
        d=np.linalg.norm(array(x)-array(t),inf)
        if d<e:
            return [x,k]
        else:
            t=x.copy()
    return [[],m]
```

5.2.2 Manejo computacional de la fórmula de Gauss-Seidel

```
def gaussseidel(a,b,x):
    n=len(x)
    for i in range(n):
        s=0
        for j in range(n):
            if i!=j:
                s=s+a[i,j]*x[j]
        x[i]=(b[i]-s)/a[i,i]
    return x
```

5.2.3 Instrumentación computacional del método de Gauss-Seidel

```
from gaussseidel import*
import numpy as np
def gaussseidelm(a,b,x,e,m):
    n=len(x)
    t=x.copy()
    for k in range(m):
        x=gaussseidel(a,b,x)
        d=np.linalg.norm(array(x)-array(t),inf)
        if d<e:
            return [x,k]
        else:
            t=x.copy()
    return [[],m]
```

5.7 Instrumentación del método de Gauss-Seidel con el radio espectral

```
from numpy import*
def gs(A,B,E):
    n=len(B)
    D=diag(diag(A))
    LD=tril(A)
    U=triu(A)-D
    C=dot(linalg.inv(LD),B)
    T=-dot(linalg.inv(LD),U)
    [val,vec]=linalg.eig(T)
    ro=max(abs(val))
    if ro>=1:                    #No converge
        return [[],0,ro]
    X0=ones([n,1],float)         #Vector inicial
    i=1
    while True:
        X1=C+dot(T,X0)
        if linalg.norm(X1-X0,inf)<E:
            return[X1,i,ro]
        i=i+1
        X0=X1.copy()
```

6.2.3 Instrumentación computacional del método de Lagrange

```
from sympy import*
def lagrange(x,y,u=None):
    n=len(x)
    t=Symbol('t')
    p=0
    for i in range(n):
        L=1
        for j in range(n):
            if j!=i:
                L=L*(t-x[j])/(x[i]-x[j])
        p=p+y[i]*L
        p=expand(p)
    if u==None:
        return p
    elif type(u)==list:
        v=[]
        for i in range(len(u)):
            v=v+[p.subs(t,u[i])]
        return v
    else:
        return p.subs(t,u)
```

6.11.3 Instrumentación computacional del trazador cúbico natural

```
import numpy as np
from sympy import*
def trazador_natural(x,y,z=[]):
    n=len(x)
    h=np.zeros([n-1])
    A=np.zeros([n-2,n-2]);B=np.zeros([n-2]);S=np.zeros([n])
    a=np.zeros([n-1]);b=np.zeros([n-1]);c=np.zeros([n-1]);d=np.zeros([n-1])
    if n<3:
        T=[]
        return
    for i in range(n-1):
        h[i]=x[i+1]-x[i]
    A[0,0]=2*(h[0]+h[1])                                    #Armar el sistema
    A[0,1]=h[1]
    B[0]=6*((y[2]-y[1])/h[1]-(y[1]-y[0])/h[0])
    for i in range(1,n-3):
        A[i,i-1]=h[i]
        A[i,i]=2*(h[i]+h[i+1])
        A[i,i+1]=h[i+1]
        B[i]=6*((y[i+2]-y[i+1])/h[i+1]-(y[i+1]-y[i])/h[i])
    A[n-3,n-4]=h[n-3]
    A[n-3,n-3]=2*(h[n-3]+h[n-2])
    B[n-3]=6*((y[n-1]-y[n-2])/h[n-2]-(y[n-2]-y[n-3])/h[n-3])
    r=np.linalg.solve(A,B)                                  #Resolver el sistema
    for i in range(1,n-1):
      S[i]=r[i-1]
    S[0]=0
    S[n-1]=0
    for i in range(n-1):
      a[i]=(S[i+1]-S[i])/(6*h[i])
      b[i]=S[i]/2
      c[i]=(y[i+1]-y[i])/h[i]-(2*h[i]*S[i]+h[i]*S[i+1])/6
      d[i]=y[i]
    try:
       if len(z)==0:                                        #Detecta si es un vector
          pass
    except TypeError:
       z=[z]                                                #Vector con un número
    if len(z)==0:                                           #Construir el trazador
      t=Symbol('t')
      T=[]
      for i in range(n-1):
          p=expand(a[i]*(t-x[i])**3+b[i]*(t-x[i])**2+c[i]*(t-x[i])+d[i])
          T=T+[p]
      return T
    else:                                                   #Evaluar el trazador
      m=len(z)
```

```
q=np.zeros([m])
for k in range(m):
    t=z[k]
    for i in range(n-1):
        if t>=x[i] and t<=x[i+1]:
            q[k]=a[i]*(t-x[i])**3+b[i]*(t-x[i])**2+c[i]*(t-x[i])+d[i]
if m>2:
    k=m-1
    i=n-2
    q[k]=a[i]*(t-x[i])**3+b[i]*(t-x[i])**2+c[i]*(t-x[i])+d[i]
if len(q)==1:
    return q[0]                                    #Retorna un valor
else:
    return q                                       #Retorna un vector
```

6.11.6 Instrumentación computacional del trazador cúbico sujeto

```
import numpy as np
from sympy import*
def trazador_sujeto(x,y,u,v,z=[]):
    n=len(x)
    h=np.zeros([n-1])
    A=np.zeros([n,n]);B=np.zeros([n]);S=np.zeros([n-1])
    a=np.zeros([n-1]);b=np.zeros([n-1]);c=np.zeros([n-1]);d=np.zeros([n-1])
    if n<3:
        T=[]
        return
    for i in range(n-1):
        h[i]=x[i+1]-x[i]
    A[0,0]=-h[0]/3
    A[0,1]=-h[0]/6
    B[0]=u-(y[1]-y[0])/h[0]
    for i in range(1,n-1):
        A[i,i-1]=h[i-1]
        A[i,i]=2*(h[i-1]+h[i])
        A[i,i+1]=h[i]
        B[i]=6*((y[i+1]-y[i])/h[i]-(y[i]-y[i-1])/h[i-1])
    A[n-1,n-2]=h[n-2]/6
    A[n-1,n-1]=h[n-2]/3
    B[n-1]=v-(y[n-1]-y[n-2])/h[n-2]
    S=np.linalg.solve(A,B)
    for i in range(n-1):
      a[i]=(S[i+1]-S[i])/(6*h[i])
      b[i]=S[i]/2
      c[i]=(y[i+1]-y[i])/h[i]-(2*h[i]*S[i]+h[i]*S[i+1])/6
      d[i]=y[i]
    try:
       if  len(z)==0:
          pass
    except TypeError:
       z=[z]
    if  len(z)==0:
      t=Symbol('t')
      T=[]
      for i in range(n-1):
          p=expand(a[i]*(t-x[i])**3+b[i]*(t-x[i])**2+c[i]*(t-x[i])+d[i])
          T=T+[p]
      return T
    else:
      m=len(z)
```

```
q=np.zeros([m])
for k in range(m):
    t=z[k]
    for i in range(n-1):
        if t>=x[i] and t<=x[i+1]:
            q[k]=a[i]*(t-x[i])**3+b[i]*(t-x[i])**2+c[i]*(t-x[i])+d[i]
if m>2:
    k=m-1
    i=n-2
    q[k]=a[i]*(t-x[i])**3+b[i]*(t-x[i])**2+c[i]*(t-x[i])+d[i]
if len(q)==1:
    return q[0]
else:
    return q
```

6.11.11 Instrumentación computacional del trazador cúbico paramétrico cerrado

```
import numpy as np
from sympy import*
def trazador_cerrado(z,u,s=[]):
    n=len(z)
    h=np.zeros([n-1])
    A=np.zeros([n-1,n-1]);B=np.zeros([n-1]);S=np.zeros([n])
    a=np.zeros([n-1]);b=np.zeros([n-1]);c=np.zeros([n-1]);d=np.zeros([n-1])
    if n<3:
        T=[]
        return
    for i in range(n-1):
        h[i]=z[i+1]-z[i]
    A[0,0]=-1/3*(h[0]+h[n-2])              #Construir el sistema de ecuaciones
    A[0,1]=-1/6*h[0]
    A[0,n-2]=-1/6*h[n-2]
    B[0]=-(u[1]-u[0])/h[0]+(u[n-1]-u[n-2])/h[n-2]
    for i in range(1,n-2):
        A[i,i-1]=h[i-1]
        A[i,i]=2*(h[i-1]+h[i])
        A[i,i+1]=h[i]
        B[i]=6*((u[i+1]-u[i])/h[i]-(u[i]-u[i-1])/h[i-1])
    A[n-2,0]=h[n-2]
    A[n-2,n-3]=h[n-3]
    A[n-2,n-2]=2*(h[n-3]+h[n-2])
    B[n-2]=6*((u[n-1]-u[n-2])/h[n-2]-(u[n-2]-u[n-3])/h[n-3])
    r=np.linalg.solve(A,B)                 #Resolver el sistema
    for i in range(n-1):
        S[i]=r[i]
    S[n-1]=r[0]
    for i in range(n-1):                   #Coeficientes de los polinomios
        a[i]=(S[i+1]-S[i])/(6*h[i])
        b[i]=S[i]/2
        c[i]=(u[i+1]-u[i])/h[i]-(2*h[i]*S[i]+h[i]*S[i+1])/6
        d[i]=u[i]
    try:
        if len(s)==0:                      #Detecta si es un vector
            pass
    except TypeError:
        s=[s]
    if len(s)==0:                          #Construir el trazador
        t=Symbol('t')
        T=[]
        for i in range(n-1):
            p=expand(a[i]*(t-z[i])**3+b[i]*(t-z[i])**2+c[i]*(t-z[i])+d[i])
            T=T+[p]
        return T                           #Retorna los polinomios
    else:                                  #Evaluar el trazador
        m=len(s)
```

```
q=np.zeros([m])
for k in range(m):
   t=s[k]
   for i in range(n-1):
      if t>=z[i] and t<=z[i+1]:
        q[k]=a[i]*(t-z[i])**3+b[i]*(t-z[i])**2+c[i]*(t-z[i])+d[i]
if m>2:
  k=m-1
  i=n-2
  q[k]=a[i]*(t-z[i])**3+b[i]*(t-z[i])**2+c[i]*(t-z[i])+d[i]
if len(q)==1:
  return q[0]                                  #Retorna un valor
else:
  return q                                     #Retorna un vector
```

7.1.3 Instrumentación computacional de la fórmula de los trapecios

```
def trapecios(f,a,b,m):
    h=(b-a)/m
    s=0
    for i in range(1,m):
        s=s+f(a+i*h)
    r=h/2*(f(a)+2*s+f(b))
    return r
```

7.1.6 Instrumentación computacional de la fórmula de Simpson

```
def simpson(f, a, b, m):
    h=(b-a)/m
    s=0
    x=a
    for i in range (1,m):
        s=s+2*(i%2+1)*f(x+i*h)          #Coeficientes 4, 2, 4, 2, ...
    s=h/3*(f(a)+s+f(b))
    return s
```

7.1.12 Cálculo de la longitud del arco usando el Trazador Cúbico paramétrico

```
# Cálculo de la longitud de un arco con
# ecuaciones paramétricas con variable v, parámetro s
from sympy import*
def arco(Tx,Ty,v,s):
    n=len(Tx)
    r=0
    for i in range(n):
        x=Tx[i]
        dx=diff(x,v)
        y=Ty[i]
        dy=diff(y,v)
        r=r+integrate(sqrt(dx**2+dy**2),(v,s[i],s[i+1]))
    return float(r)
```

```
# Fórmula de Simpson con f simbólica, variable v, y m franjas
def simpsons(f, v, a, b, m):
    h=(b-a)/m
    s=0
    for i in range (1,m):
        s=s+2*(i%2+1)*f.subs(v,a+i*h)
    s=h/3*(f.subs(v,a)+s+f.subs(v,b))
    return s
```

```
# Cálculo de la longitud de un arco con ecuaciones
# paramétricas Tx, Ty, con variable v, parámetro s, y m franjas
from simpsons import*
from sympy import*
def arco(Tx,Ty,v,s,m):
    n=len(Tx)
    r=0
    for i in range(n):
        x=Tx[i]
        dx=diff(x,v)
        y=Ty[i]
        dy=diff(y,v)
        f=sqrt(dx**2+dy**2)
        r=r+simpsons(f,v,s[i],s[i+1],m)
    return r
```

7.3.2 Instrumentación computacional de la cuadratura de Gauss

```
from math import*
def cgauss(f,a,b):
    t0=-(b-a)/2*1/sqrt(3)+(b+a)/2
    t1= (b-a)/2*1/sqrt(3)+(b+a)/2
    s = (b-a)/2*(f(t0) + f(t1))
    return s
```

7.3.3 Instrumentación extendida de la cuadratura de Gauss

```
from cgauss import*
def cgaussm(f,a,b,m):
    h=(b-a)/m
    s=0
    x=a
    for i in range(m):
        a=x+i*h
        b=x+(i+1)*h
        s=s+cgauss(f,a,b)
    return s
```

7.6.1 Instrumentación computacional de la fórmula de Simpson en dos direcciones

```
from sympy import*
from simpson import*
def simpson2(f,ax,bx,ay,by,mx,my):
    x=Symbol('x')
    dy=(by-ay)/my
    v=ay
    r=[]
    for i in range (0,my+1):
        def g(x): return f(x,v)
        u=simpson(g,ax,bx,mx)
        r=r+[u]
        v=v+dy
    s=0
    for i in range(1,my):
        s=s+2*(2-(i+1)%2)*r[i]
    s=dy/3*(r[0]+s+r[my])
    return s
```

7.8 Cálculo aproximado del área de figuras cerradas en el plano

```
from sympy import*
def green(Tx,Ty,s):
# Cálculo del área de una región cerrada
# con el teorema de Green
    t=Symbol('t')
    n=len(Tx)
    r=0
    for i in range(n):
        x=Tx[i]
        dx=diff(x,t)
        y=Ty[i]
        dy=diff(y,t)
        r=r+integrate(x*dy-y*dx,(t,s[i],s[i+1]))
    return 0.5*r
```

9.1.1 Método de la serie de Taylor

Instrumentación computacional del método de la Serie de Taylor

```
import numpy as np
def taylor3(f,df,x,y,h,m):
    u=np.zeros([m,2])                #Iniciar la matriz u con mx2 ceros
    for i in range(m):
        y=y+h*f(x,y)+h**2/2*df(x,y)
        x=x+h
        u[i,0]=x                     #La primera columna almacena x
        u[i,1]=y                     #La segunda columna almacena y
    return u
```

9.1.2 Obtención de derivadas de funciones implícitas

```
import sympy as sp
x,y=sp.symbols('x,y')
def derive(f,nd):
    t=f
    for j in range(1,nd+1):
        d=sp.diff(f.subs(y,y(x)),x)
        f=d.subs(sp.Derivative(y(x),x),t).subs(y(x),y)
    return f
```

9.1.3 Instrumentación de un método general para resolver una E.D.O con la serie de Taylor

```
from derive import *
import numpy as np
import sympy as sp
x,y=sp.symbols('x,y')
def taylorg(f,a,b,h,m,k):
    u=np.zeros([m,2])
    D=[ ]
    for j in range(1,k+1):
        D=D+[derive(f,j)]        #Vector de derivadas simbólicas

    for i in range(m):
        g=f.subs(x,a).subs(y,b)
        t=b+h*g
        for j in range(1,k+1):
            z=D[j-1].subs(x,a).subs(y,b)
            t=float(t+h**(j+1)/sp.factorial(j+1)*z) #Evaluar Taylor
        b=t
        a=a+h
        u[i,0]=a
        u[i,1]=b
    return u
```

Instrumentación computacional de la fórmula de Euler

```
import numpy as np
def euler(f,x,y,h,m):
    u=np.zeros([m,2])
    for i in range(m):
        y=y+h*f(x,y)
        x=x+h
        u[i,0]=x
        u[i,1]=y
    return u
```

Instrumentación computacional de la fórmula de Heun

```
import numpy as np
def heun(f,x,y,h,m):
    u=np.zeros([m,2],dtype=float)
    for i in range(m):
        yn=y+h*f(x,y)
        y=y+h/2*(f(x,y)+f(x+h,yn))
        x=x+h
        u[i,0]=x
        u[i,1]=y
    return u
```

Instrumentación computacional de la fórmula de Runge-Kutta de segundo orden

```
import numpy as np
def rk2(f,x,y,h,m):
    u=np.zeros([m,2],float)
    for i in range(m):
        k1=h*f(x,y)
        k2=h*f(x+h,y+k1)
        y=y+0.5*(k1+k2)
        x=x+h
        u[i,0]=x
        u[i,1]=y
    return u
```

Instrumentación computacional de la fórmula de Runge-Kutta de cuarto orden

```
import numpy as np
def rk4(f,x,y,h,m):
    u=np.zeros([m,2],dtype=float)
    for i in range(m):
        k1=h*f(x,y)
        k2=h*f(x+h/2,y+k1/2)
        k3=h*f(x+h/2,y+k2/2)
        k4=h*f(x+h,y+k3)
        y=y+1/6*(k1+2*k2+2*k3+k4)
        x=x+h
        u[i,0]=x
        u[i,1]=y
    return u
```

Instrumentación computacional de la fórmula de Runge-Kutta de segundo orden para resolver sistemas de E. D. O. de primer orden

```
#Runge-Kutta de segundo orden para n EDO-condiciones en el inicio
import sympy as sp
import numpy as np
def rk2n(F,V,U,h,m):
    nF=len(F)
    nV=len(V)
    K1=np.zeros([nF],dtype=sp.Symbol)
    K2=np.zeros([nF],dtype=sp.Symbol)
    rs=np.zeros([m,nV],dtype=float)
    T=list(np.copy(U))

    for p in range(m):
        for i in range(nF):
            K1[i]=F[i]
            K2[i]=F[i]
        for i in range(nF):
            for j in range(nV):
                K1[i]=K1[i].subs(V[j],float(T[j]))
            K1[i]=h*K1[i]
        for i in range(nF):
            K2[i]=K2[i].subs(V[0],float(T[0])+h)
            for j in range(1,nV):
                K2[i]=K2[i].subs(V[j],float(T[j])+K1[j-1])
            K2[i]=h*K2[i]

        T[0]=T[0]+h
        rs[p,0]=T[0]
        for i in range(nF):
            T[i+1]=T[i+1]+0.5*(K1[i]+K2[i])
            rs[p,i+1]=T[i+1]
    return rs
```

Instrumentación computacional de la fórmula de Runge-Kutta de cuarto orden para resolver sistemas de E. D. O. de primer orden

```
#Runge Kutta de cuarto orden para n EDO's
import sympy as sp
import numpy as np
def rk4n(F,V,U,h,m):
    nF=len(F)
    nV=len(V)
    K1=np.zeros([nF],dtype=sp.Symbol)
    K2=np.zeros([nF],dtype=sp.Symbol)
    K3=np.zeros([nF],dtype=sp.Symbol)
    K4=np.zeros([nF],dtype=sp.Symbol)
    rs=np.zeros([m,nV],dtype=float)
    T=list(np.copy(U))

    for p in range(m):
        for i in range(nF):
            K1[i]=F[i]
            K2[i]=F[i]
            K3[i]=F[i]
            K4[i]=F[i]
        for i in range(nF):
            for j in range(nV):
                K1[i]=K1[i].subs(V[j],float(T[j]))
            K1[i]=h*K1[i]
        for i in range(nF):
            K2[i]=K2[i].subs(V[0],float(T[0])+h/2)
            for j in range(1,nV):
                K2[i]=K2[i].subs(V[j],float(T[j])+K1[j-1]/2)
            K2[i]=h*K2[i]
        for i in range(nF):
            K3[i]=K3[i].subs(V[0],float(T[0])+h/2)
            for j in range(1,nV):
                K3[i]=K3[i].subs(V[j],float(T[j])+K2[j-1]/2)
            K3[i]=h*K3[i]
        for i in range(nF):
            K4[i]=K4[i].subs(V[0],float(T[0])+h)
            for j in range(1,nV):
                K4[i]=K4[i].subs(V[j],float(T[j])+K3[j-1])
            K4[i]=h*K4[i]
        T[0]=T[0]+h
        rs[p,0]=T[0]
        for i in range(nF):
            T[i+1]=T[i+1]+1/6*(K1[i]+2*K2[i]+2*K3[i]+K4[i])
            rs[p,i+1]=T[i+1]
    return rs
```

Instrumentación computacional del método de diferencias finitas para una EDO

```
from tridiagonal import *
import numpy as np
def edodif(P,Q,R,S,x0,y0,xn,yn,n):
    h=(xn-x0)/n
    a=[];b=[];c=[];d=[]
    u=np.zeros([n-1,2],float)
    for i in range(0,n-1):
        x=x0+h*i
        a=a+[P(x,h)]                #diagonales del sistema tridiagonal
        b=b+[Q(x,h)]
        c=c+[R(x,h)]
        d=d+[S(x,h)]                #constantes del sistema tridiagonal
        u[i,0]=x
    d[0]=d[0]-a[0]*y0               #corrección para la primera ecuación
    d[n-2]=d[n-2]-c[n-2]*yn         #corrección para la última ecuación
    u[:,1]=tridiagonal(a,b,c,d)
    return u
```

Instrumentación computacional del método de diferencias finitas con derivadas en los bordes

```
from tridiagonal import *
import numpy as np
def edodifdi(P,Q,R,S,x0,dy0,xn,yn,n):
    h=(xn-x0)/n
    a=[];b=[];c=[];d=[]
    u=np.zeros([n,2],float)
    for i in range(0,n):
        x=x0+h*i
        a=a+[P(x,h)]
        b=b+[Q(x,h)]
        c=c+[R(x,h)]
        d=d+[S(x,h)]
        u[i,0]=x
    x=h
    c[0]=P(x,h)+R(x,h)
    d[0]=S(x,h)+P(x,h)*2*h*dy0
    d[n-1]=d[n-1]-c[n-1]*yn
    u[:,1]=tridiagonal(a,b,c,d)
    return u
```

10.2.3 Instrumentación computacional del método explícito de diferencias finitas para una E.D.P. de tipo parabólico

```
# Método explícito de diferencias finitas para una EDP parabólica
# U(i,j+1)=(P)U(i-1,j) + (Q)U(i,j) + (R)U(i+1,j)

def edpdif(P,Q,R,U,m):
    u=[U[0]]
    for i in range(1,m):
        u=u+[P*U[i-1]+Q*U[i]+R*U[i+1]]      # Cálculo de puntos
    u=u+[U[m]]
    return u

import pylab as pl
m=10                            # Número de puntos en x
n=100                           # Número de niveles en t
Ta=60; Tb=40                    # Condiciones en los bordes
To=25                           # Condición en el inicio
dx=0.1; dt=0.01                 # incrementos
L=1                             # longitud
k=4                             # dato especificado
U=[Ta]                          # Asignación inicial
for i in range(1,m):
   U=U+[To]
U=U+[Tb]
lamb=dt/(k*dx**2)               #Parámetro lambda
P=lamb
Q=1-2*lamb
R=lamb
pl.title('Curvas de distribución térmica');
pl.xlabel('X (distancia)');
pl.ylabel('U (temperatura)')
x=[0]
for i in range(1,m+1):
    x=x+[i*dx]                  # Coordenadas para el gráfico
pl.plot(x,U,'or')               # Distribución inicial
for j in range(n):
    U=edpdif(P,Q,R,U,m)
    if j%10==0:
        pl.plot(x,U,'-r');      # curvas cada 10 niveles de t
        pl.plot(x,U,'.r')       # puntos
pl.grid(True)
pl.show()
```

10.2.5 Instrumentación computacional del método implícito para una E.D.P. de tipo parabólico

```
# Método implícito de diferencias finitas para una EDP parabólica
# (P)U(i-1,j) + (Q)U(i,j) + (R)U(i+1,j) = -U(i,j-1)
from tridiagonal import*
def edpdifpi(P, Q, R, U, m):
# Método de Diferencias Finitas Implícito
    a=[];b=[];c=[];d=[]
    for i in range(m-2):
        a=a+[P]
        b=b+[Q]
        c=c+[R]
        d=d+[-U[i+1]]
    d[0]=d[0]-a[0]*U[0]
    d[m-3]=d[m-3]-c[m-3]*U[m-1]
    u=tridiagonal(a,b,c,d)
    U=[U[0]]+u+[U[m-1]]
    return U

import pylab as pl
m=11                          # Número ecuaciones: m-1
n=100                         # Número de niveles en t
Ta=60; Tb=40                  # Condiciones en los bordes
To=25                         # Condición en el inicio
dx=0.1; dt=0.01               # incrementos
L=1                           # longitud
k=4                           # dato especificado
U=[Ta]                        # Asignación inicial
for i in range(1,m-1):
   U=U+[To]
U=U+[Tb]
lamb=dt/(k*dx**2)
P=lamb
Q=-1-2*lamb
R=lamb
pl.title('Curvas de distribución térmica');
pl.xlabel('X (distancia)');
pl.ylabel('U (temperatura)')
x=[]
for i in range(m):
    x=x+[i*dx]                # Coordenadas para el gráfico
pl.plot(x,U,'or')             # Distribución inicial
for j in range(n):
    U=edpdifpi(P,Q,R,U,m)
    if j%10==0:
        pl.plot(x,U,'-r');    # curvas cada 5 niveles de t
        pl.plot(x,U,'.r')
pl.grid(True)
pl.show()
```

10.2.8 Instrumentación computacional para una E.D.P. con derivadas en los bordes

```
# Solución de una EDP con una derivada en un borde
from tridiagonal import*
def edpdifpid(P,Q,R,U,der0,dx,m):
# Método de Diferencias Finitas Implícito
    a=[];b=[];c=[];d=[]
    for i in range(m-1):
        a=a+[P]
        b=b+[Q]
        c=c+[R]
        d=d+[-U[i+1]]
    c[0]=P+R;
    d[0]=d[0]+2*dx*P*der0
    d[m-2]=d[m-2]-c[m-2]*U[m-1]
    u=tridiagonal(a,b,c,d)
    U=u+[U[m-1]]
    return U

import pylab as pl
m=11                          # Número ecuaciones: m-1
n=50                          # Número de niveles en t
der0=-5                       # Derivada en el borde izquierdo
Tb=60                         # Condiciones en los bordes
To=40                         # Condición en el inicio
dx=0.1                        # incrementos
dt=0.1
L=1                           # longitud
k=4                           # dato especificado
U=[]                          # Asignación inicial
for i in range(m-1):
   U=U+[To]
U=U+[Tb]
lamb=dt/(k*dx**2)
P=lamb
Q=-1-2*lamb
R=lamb
pl.title('Curvas de distribución térmica');
pl.xlabel('X (distancia)');
pl.ylabel('U (temperatura)')
x=[]
for i in range(m):
    x=x+[i*dx]                      # Coordenadas para el gráfico
pl.plot(x,U,'or')                         # Distribución inicial
for j in range(n):
    U=edpdifpid(P,Q,R,U,der0,dx,m)
    pl.plot(x,U,'-r');              # curvas cada 5 niveles de t
    pl.plot(x,U,'.r')
pl.grid(True)
pl.show()
```

10.3.2 Instrumentación computacional para una E.D.P. de tipo elíptico

```
# Programa para resolver una EDP Elíptica
# con condiciones constantes en los bordes
from numpy import*
Ta=60;Tb=60;Tc=50;Td=70    #Bordes izquierdo, derecho, abajo, arriba
n=10                       #Puntos interiores en dirección hor. (X)
m=10                       #Puntos interiores en dirección vert.(Y)
miter=100                  #Máximo de iteraciones
e=0.001                    #Error de truncamiento relativo 0.1%
u=zeros([n+2,m+2],float)
for i in range(n+2):
    u[i,0]=Tc
    u[i,m+1]=Td
for j in range(m+2):
    u[0,j]=Ta
    u[n+1,j]=Tb
p=0.25*(Ta+Tb+Tc+Td)            # valor inicial interior promedio
for i in range(1,n-1):
    for j in range(1,m-1):
        u[i,j]=p
k=0                              # conteo de iteraciones
converge=False                   # señal de convergencia
while k<miter and not converge:
      k=k+1
      t=u.copy()
      for i in range(1,n+1):
          for j in range(1,m+1):
              u[i,j]=0.25*(u[i-1,j]+u[i+1,j]+u[i,j+1]+u[i,j-1])
      if linalg.norm((u-t),inf)/linalg.norm(u,inf)<e:
          converge=True
if converge:
    for i in range(n+2):               # Malla con la solución final
        print([float('%5.2f' % (u[i,j])) for j in range(m+2)])

    print('Conteo de iteraciones: ',k)    # Conteo de iteraciones

    import pylab as pl
    from mpl_toolkits.mplot3d import Axes3D      # Gráfico 3D
    fig=pl.figure()
    ax=Axes3D(fig)
    x=pl.arange(0,1.2,0.1)
    y=pl.arange(0,1.2,0.1)
    X,Y=pl.meshgrid(x,y)
    ax.plot_surface(X,Y,u,rstride=1,cstride=1,cmap='hot')
    pl.show()
else:
    print('No converge')
```

10.4.2 Instrumentación computacional para una E.D.P. de tipo hiperbólico

```
# Método de Diferencias Finitas explícito: EDP Hiperbólica

from numpy import*
import pylab as pl
m=11                                  # Número de puntos en x
n=10                                  # Número de niveles en t
c=2                                   # dato especificado
L=1                                   # longitud
dx=L/(m-1)                            # incremento
dt=sqrt(dx**2/c**2)                    # para cumplir la condición
U0=zeros([m])                         # Extremos fijos

x=0
for i in range(1,m-1):                 # Nivel inicial
    x=x+dx
    if x<L/2:
        U0[i]=-0.5*x                  # Expresión para el desplazamiento
    else:
        U0[i]= 0.5*(x-1)

U1=[U0[0]]                            # Primer nivel
for i in range (1,m-1):
   U1=U1 + [0.5*(U0[i-1]+U0[i+1])]
U1=U1+[U0[m-1]]

for j in range(1,n+1):                # Siguientes niveles
    Uj=[U1[0]]
    for i in range(1,m-1):
         Uj=Uj + [U1[i+1]+U1[i-1]-U0[i]]
    Uj=Uj + [U1[m-1]]
    U0=U1                             # Actualizar niveles anteriores
    U1=Uj

    # Mostrar la solución en cada nivel
    print('%4d'%j,[float('%5.2f' % (Uj[j])) for j in range(m)])

# Mostrar el gráfico de la solución en el último nivel
x=[]
for i in range(m):
    x=x+[i*dx]                        # Coordenadas para el gráfico

pl.grid(True)
pl.plot(x,Uj,'or')                    # Graficar puntos y cuerda
pl.plot(x,Uj,'-r')
pl.show()
```

10.4.3 Instrumentación computacional con animación para una E.D.P. de tipo Hiperbólico

```
# Método de Diferencias Finitas explícito: EDP Hiperbólica
# Programa para animación de la figura
global m,c,L,dx,dt
m=51                                # Número de puntos en x
c=2                                 # dato especificado
L=1                                 # longitud
dx=L/(m-1)                          # incremento
dt=(dx**2/c**2)**0.5                # para cumplir la condición

import numpy as np
import matplotlib.pyplot as plt
from matplotlib import animation

def posicion_inicial():
    global U0,U1
    U0=np.zeros([m])                      # Extremos fijos
    x=0
    for i in range(1,m-1):                # Nivel inicial
        x=x+dx
        if x<L/2:
            U0[i]=-0.5*x             # Expresión para el desplazamiento
        else:
            U0[i]= 0.5*(x-1)
    U1=[U0[0]]                           # Primer nivel
    for i in range (1,m-1):
       U1=U1 + [0.5*(U0[i-1]+U0[i+1])]
    U1=U1+[U0[m-1]]

# Definir la figura, ejes, y elemento que se va a animar
fig = plt.figure()
ax = plt.axes(xlim=(0,1), ylim=(-1,1))
linea, = ax.plot([], [], lw=2)

# Iniciar la función de animación
def inicio():
    linea.set_data([], [])
    return linea,

# Función con los datos de animación. Es llamada secuencialmente
def animar(j):
    global U0,U1
    Uj=[U1[0]]
    for i in range(1,m-1):
        Uj=Uj + [U1[i+1]+U1[i-1]-U0[i]]  # Posición actual de la cuerda
    Uj=Uj + [U1[m-1]]
    U0=U1                                # Actualizar niveles anteriores
```

```
    U1=Uj
    x=np.arange(0,1+dx,dx)
    y=Uj                                    # Coordenadas para el gráfico
    linea.set_data(x, y)
    return linea,

posicion_inicial()

# Animación. blit=True para redibujar solo las partes que han cambiado

anim = animation.FuncAnimation(fig, animar, init_func=inicio,
frames=100, interval=20, blit=True)
plt.show()
```