```
import os
import numpy as np
from math import exp
import copy
```

Para guardar los resultados en un csv

```
results_dir = './Results/' # Ruta
if not os.path.exists(results_dir):
os.makedirs(results_dir)
def save_result(name,result):
np.savetxt(results_dir+str(name)+'.csv', result, delimiter=',', fmt='%s')
```

Para saber qué apartado correr

```
data = 'data' #Nombre del archivo
try:
with open(data, 'r') as data:
for line in data:
if 'apartado_a' in line:
p = line.split()
apartado_a = p[2]
if 'apartado_b' in line:
p = line.split()
apartado_b = p[2]
if 'apartado_c' in line:
p = line.split()
apartado_c = p[2]
if 'apartado_d' in line:
p = line.split()
apartado_d = p[2]
if 'apartado_e' in line:
p = line.split()
apartado_e = p[2]
if 'numerico' in line:
p = line.split()
numerico = p[2]
if 'disp' in line:
p = line.split()
disp = float(p[2])
except:
### Apartados
apartado_a = 'yes'
apartado_b = 'yes'
apartado_c = 'yes'
apartado_d = 'yes'
apartado_e = 'yes'
###
numerico = 'yes'
###
disp = 100
```

```
print('No data configuration file found')
```

Clases

class elemento(object):

```
def __init__(self):
    # Atributos geométricos
    self.Lx = ''
    self.Ly = ''
    self.Lz = ''

    # Atributos térmicos
    self.k = ''
    self.kxy = ''
    self.kx = ''
    self.ky = ''
    self.kz = ''

    self.rho_c = ''

    self.W = ''

def made_of(self,objects):

    self.Lx = max(c.Lx for c in objects)
    self.Ly = max(c.Ly for c in objects)
    self.Lz = sum(c.Lz for c in objects)

    self.V = self.Lx * self.Ly * self.Lz

    self.Ax = self.Ly*self.Lz #Área efectiva de paso en la dirección x

    self.kx = sum((c.kx*c.Ly*c.Lz) for c in objects)/self.Ax
    self.rho_c = sum((c.rho_c*c.Ly*c.Lz*c.Lx) for c in objects)/self.V

def add(self,objects):

    self.kx2 = sum((c.kx*c.Ly*c.Lz) for c in objects)/self.Ax + self.kx
    self.rho_c2 = sum((c.rho_c*c.Ly*c.Lz*c.Lx) for c in objects)/self.V + self.rho_c
```

ENUNCIADO

Considérese una tarjeta electrónica (PCB) de 140×100×1,5 mm3 de FR-4, con un recubrimiento de 50 μm de

cobre por cada lado, que en una de las caras es continuo, y en la otra sólo ocupa el 10% de la superficie, en la

cual van montados tres circuitos integrados (IC), cada uno de 40×20×3 mm3

, disipando 5 W, con kIC=50 W/(m·K)

de conductividad térmica, CIC=20 J/K de capacidad térmica, y distribuidos uniformemente en la PCB (20 mm de

separación entre ellos). Se supondrá que los lados cortos de la PCB tienen contacto térmico perfecto con paredes

permanentemente a 25 ºC, y que los otros dos bordes están térmicamente aislados. Tómese para el FR-4 k=0,5

W/(m·K) en el plano y la mitad a su través. Se pide:

FR4

```
FR4 = elemento()
FR4.Lx = 140e-3 #m
FR4.Ly = 100e-3 #m
FR4.Lz = 1.5e-3 #m
FR4.kx = 0.5 #W/(m·K)
FR4.ky = 0.5 #W/(m·K)
FR4.kz = 0.25 #W/(m·K)
FR4.rho_c = 1850 * 3000 # J/(K·m^3)
```

Cu

```
Cu = elemento()
Cu.Lx = FR4.Lx #m
Cu.Ly = FR4.Ly #m
Cu.Lz = 50e-6 #m
Cu.F = 0.1
Cu.kx = 395.0 #W/(m·K)
Cu.ky = 395.0 #W/(m·K)
Cu.kz = 395.0 #W/(m·K)
Cu.rho_c = 385 * 8260 # J/(K·m^3)

Cu_up = copy.deepcopy(Cu)
Cu_up.kx = 395.0 * 0.1 #W/(m·K)
Cu_up.ky = 395.0 * 0.1 #W/(m·K)
Cu_up.kz = 395.0 * 0.1 #W/(m·K)
Cu_up.rho_c = Cu.rho_c * 0.1 # J/(K·m^3)
```

IC

```
IC = elemento()
IC.Lx = 20e-3 #m
IC.Ly = 40e-3 #m
IC.Lz = 3e-3 #m
IC.W = 5 #W
IC.kx = 50 #W/(m·K)
```

IC.ky = 50 #W/(m·K)
IC.kz = 50 #W/(m·K)
IC.rho_c = 20 * 1/(IC.Lx*IC.Ly*IC.Lz) #J/(K·m^3)
IC.pitch = 20e-3 #m

PCB

PCB = elemento()
PCB.made_of([FR4,Cu,Cu_up]) #Los IC van aparte porque a veces no están
PCB.add([IC])

Paredes

T_wall = 25+273.15 #K

if apartado_a == 'yes':
print('*** a ***')

```
## a) Considerando que la tarjeta sólo evacua calor por los bordes, determinar la temperatura máxima que se
## alcanzaría si toda la disipación estuviese uniformemente repartida en la PCB y los IC no influyeran.

# Debido a la simetría del problema se puede separar el problema en dos
W_dis = 3*IC.W
Q_wall = W_dis/2
phi = W_dis/(PCB.V)

A_eff = PCB.Ax
k_eff = PCB.kx
L = PCB.Lx/2

## Solución analítica
print('Solución analítica')

# Primera aproximación, toda la potencia concentrada en el centro: T(x) = a + b*x
a1 = T_wall
b1 = Q_wall/(k_eff * A_eff)

T_max = a1 + b1* L

print(' Potencia puntual: T_max = ',round(T_max),'K ó',round(T_max-273.15),'C')

# Segunda aproximación, potencia distribuida: T(x) = a + b*x + c*x^2 donde c = - phi /(2k)
a2 = T_wall
b2 = Q_wall/(k_eff * A_eff)
c2 = -phi /(2*k_eff)

T_max = a2 + b2*L + c2*L**2

print(' Potencia uniforme: T_max = ',round(T_max),'K ó',round(T_max-273.15),'C')

# Discretización de la solución

N = 50
xa = np.linspace(0,L,N+1)
Ta1 = a1 + b1*xa
Ta2 = a2 + b2*xa + c2*xa**2
xa = np.concatenate((xa,xa+L))
Ta1 = np.concatenate((Ta1,np.flip(Ta1)))
Ta2 = np.concatenate((Ta2,np.flip(Ta2)))

# Guardar resultados
save_result('xa',xa)
save_result('Ta1',Ta1)
save_result('Ta2',Ta2)
```

```python
## Solución numérica
print('Solución numérica')

if numerico == 'yes':
    # Discretización

    L = 2*L        # Espacio de simulación
    T = 5000       # Tiempo de simulación
    N = 70         # Número de elementos espaciales
    M = int(1e5)   # Número de elementos temporales (ver criterio)
    Dx = L/N
    Dt = T/M
    xan = np.linspace(0,L,N+1)
    ta = np.linspace(0,T,M+1)

    rho_c = PCB.rho_c
    p = 0
    h = 0

    # Estabilidad
    a=k_eff/(rho_c)             #Diffusivity [m^2/s]
    Fo=a*Dt/(Dx*Dx)             #Fourier's number
    Bi=h*p*Dx/(k_eff*A_eff/Dx)  #Biot's number

    if (1-Fo*(2+Bi)) < 0:
        print('This is unstable increase number of time steps')

    # Potencia puntual

    T = T_wall * np.ones((M+1,N+1)) # T(t,x) inicials

    for t in range(0,len(ta)-1):
        if t%disp == 0: print('Tiempo de simulación:',ta[t], 's \Temperatura máxima:',np.amax(T[t,:]),'K')

        # Condiciones de contorno
        T[t,0]=T_wall
        T[t,N]=T_wall

        for x in range(1,len(xan)-1):
            # Potencia puntual
            if x == int(N/2): phii = W_dis/(Dx*PCB.Ly*PCB.Lz)
            else: phii=0
            # Euler explícito
            T[t+1,x] = T[t,x] + Dt/(rho_c*A_eff)*(k_eff*A_eff*(T[t,x+1]-T[t,x])/Dx**2-k_eff*A_eff*(T[t,x]-T[t,x-1])/Dx*

    Ta1n = np.zeros((len(xan)))
    Ta1n[:]=T[-1,:]
    T_max = max(Ta1n)

    print(' Potencia puntual: T_max = ',round(T_max),'K ó',round(T_max-273.15),'C')

    # Potencia uniforme

    T = T_wall * np.ones((M+1,N+1)) # T(t,x) inicial

    for t in range(0,len(ta)-1):
        if t%disp == 0: print('Tiempo de simulación:',ta[t], 's \Temperatura máxima:',np.amax(T[t,:]),'K')

        # Condiciones de contorno
        T[t,0]=T_wall
        T[t,N]=T_wall

        for x in range(1,len(xan)-1):
            # Euler explícito
            T[t+1,x] = T[t,x] + Dt/(rho_c*A_eff)*(k_eff*A_eff*(T[t,x+1]-T[t,x])/Dx**2-k_eff*A_eff*(T[t,x]-T[t,x-1])/Dx*



    Ta2n = np.zeros((len(xan)))
    Ta2n[:]=T[-1,:]
    T_max = max(Ta2n)
```

```
    print(' Potencia uniforme: T_max = ',round(T_max),'K ó',round(T_max-273.15),'C')

    # Guardar resultados
    save_result('xan',xan)
    save_result('Ta1n',Ta1n)
    save_result('Ta2n',Ta2n)
```

if apartado_b == 'yes':
print('*** b ***')

```
## b) Considerando que la tarjeta sólo evacua calor por los bordes, determinar la temperatura máxima que se
## alcanzaría con un modelo unidimensional en el que los IC llegaran hasta los bordes aislados, en el límite
## kIC→∞, y con la kIC dada.

### Para los dos problemas

W_dis = 3*IC.W
Q_wall = W_dis/2

A_eff = PCB.Ax
V_eff = A_eff * IC.Lx
phi = IC.W/V_eff
L = PCB.Lx/2

# Con ejes en el borde de cada una de las 4 zonas
L1 = L - IC.Lx- IC.pitch- IC.Lx/2
L2 = IC.Lx
L3 = IC.pitch
L4 = IC.Lx/2

## Solución analítica
print('Solución analítica')

N = 50
x1 = np.linspace(0,L1,N+1)
x2 = np.linspace(0,L2,N+1)
x3 = np.linspace(0,L3,N+1)
x4 = np.linspace(0,L4,N+1)

xb = np.concatenate((x1,x2+L1,x3+L1+L2,x4+L1+L2+L3))

## Con kIC→∞

k_eff1 = PCB.kx
#k_eff2→∞

# Zona 1: T(x) = a1 + b1*x
k_eff = k_eff1
a1 = T_wall
b1 = Q_wall/(k_eff * A_eff)
T1 = a1 + b1*x1
# Zona 2: T(x) = a2 + b2*x + c2*x^2 donde c2 = phi /(2k)
a2 = a1 + b1*L1
b2 = 0
c2 = 0
T2 = a2 + b2*x2 + c2*x2**2
# Zona 3: T(x) = a3 + b3*x
k_eff = k_eff1
a3 = a2 + b2*L2 + c2*L2**2
b3 = (IC.W/2)/(k_eff * A_eff)
T3 = a3 + b3*x3
# Zona 4: T(x) = a4 + b4*x + c4*x^2 donde c4 = -phi /(2k)
a4 =  a3 + b3*L3
b4 = 0
c4 = 0
T4 = a4 + b4*x4 + c4*x4**2

Tb1 = np.concatenate((T1, T2, T3, T4))
T_max = max(Tb1)

print(' Con kIC→∞: T_max = ',round(T_max),'K ó',round(T_max-273.15),'C')
```

```python
## Con kIC dada

k_eff1 = PCB.kx
k_eff2 = PCB.kx2

# Zona 1: T(x) = a1 + b1*x
k_eff = k_eff1
a1 = T_wall
b1 = Q_wall/(k_eff * A_eff)
T1 = a1 + b1*x1
# Zona 2: T(x) = a2 + b2*x + c2*x^2 donde c2 = -phi /(2k)
k_eff = k_eff2
a2 = a1 + b1*L1
b2 = Q_wall/(k_eff * A_eff)
c2 = - phi /(2*k_eff)
T2 = a2 + b2*x2 + c2*x2**2
# Zona 3: T(x) = a3 + b3*x
k_eff = k_eff1
a3 = a2 + b2*L2 + c2*L2**2
b3 = (IC.W/2)/(k_eff * A_eff)
T3 = a3 + b3*x3
# Zona 4: T(x) = a4 + b4*x + c4*x^2 donde c4 = -phi /(2k)
k_eff = k_eff2
a4 =  a3 + b3*L3
b4 = (IC.W/2)/(k_eff * A_eff)
c4 = - phi/2 /(2*k_eff)
T4 = a4 + b4*x4 + c4*x4**2

Tb2 = np.concatenate((T1, T2, T3, T4))
T_max = max(Tb2)

print(' Con la kIC dada: T_max = ',round(T_max),'K ó',round(T_max-273.15),'C')

# Doblar la solución

xb = np.concatenate((xb,np.flip(2*L-xb)))
Tb1 = np.concatenate((Tb1,np.flip(Tb1)))
Tb2 = np.concatenate((Tb2,np.flip(Tb2)))

# Guardar resultados
save_result('xb',xb)
save_result('Tb1',Tb1)
save_result('Tb2',Tb2)

## Solución numérica
print('Solución numérica')

# Discretización

if numerico == 'yes':

    L1 = PCB.Lx/2 - IC.Lx- IC.pitch- IC.Lx/2
    L2 = IC.Lx + L1
    L3 = IC.pitch + L2
    L4 = IC.Lx/2 + L3
    L5 = IC.Lx/2 + L4
    L6 = IC.pitch + L5
    L7 = IC.Lx + L6
    L8 = PCB.Lx

    L = PCB.Lx     # Espacio de simulación
    T = 6000       # Tiempo de simulación
    N = 70         # Número de elementos espaciales
    M = int(1e5)   # Número de elementos temporales (ver criterio)
    Dx = L/N
    Dt = T/M
    xbn = np.linspace(0,L,N+1)
    tb  = np.linspace(0,T,M+1)

    p = 0
    h = 0

    # Estabilidad
```

```
a=PCB.kx/(PCB.rho_c)           #Diffusivity [m^2/s]
Fo=a*Dt/(Dx*Dx)                #Fourier's number
Bi=h*p*Dx/(k_eff*A_eff/Dx)     #Biot's number

if (1-Fo*(2+Bi)) < 0:
    print('This is unstable increase number of time steps')

# Propiedades
def k_fun(pos):
    x = xbn[pos]
    k1 = PCB.kx
    k2 = 1e2
    from numpy import heaviside as H
    val = k1 + (H(x-L1,dis)-H(x-L2,dis)+H(x-L3,dis)-H(x-L5,dis)+H(x-L6,dis)-H(x-L7,dis))*(k2-k1)
    return val

def rho_c_fun(pos):
    x = xbn[pos]
    rho_c1 = PCB.rho_c
    rho_c2 = PCB.rho_c2
    from numpy import heaviside as H
    val = rho_c1 + (H(x-L1,dis)-H(x-L2,dis)+H(x-L3,dis)-H(x-L5,dis)+H(x-L6,dis)-H(x-L7,dis))*(rho_c2-rho_c1)
    return val

def phii_fun(pos):
    x = xbn[pos]
    from numpy import heaviside as H
    val = (H(x-L1,dis)-H(x-L2,dis)+H(x-L3,dis)-H(x-L5,dis)+H(x-L6,dis)-H(x-L7,dis))*(phi)
    return val

k=np.zeros((len(xbn)))
rho_c=np.zeros((len(xbn)))
phii=np.zeros((len(xbn)))
for x in range(0,len(xbn)):
        k[x] = k_fun(x)
        rho_c[x] = rho_c_fun(x)
        phii[x] = phii_fun(x)

## Con kIC→∞

T = T_wall * np.ones((M+1,N+1)) # T(t,x) inicial
for t in range(0,len(tb)-1):
    if t%disp == 0: print('Tiempo de simulación:',tb[t], 's \Temperatura máxima:',np.amax(T[t,:]),'K')

    # Condiciones de contorno
    T[t,0]=T_wall
    T[t,N]=T_wall

    for x in range(1,len(xbn)-1):
        # Propiedades
        kp = (k[x+1]+k[x])/2
        kn = (k[x]+k[x-1])/2
        # Euler explícito
        T[t+1,x] = T[t,x] + Dt/(rho_c[x]*A_eff)*(kp*A_eff*(T[t,x+1]-T[t,x])/Dx**2-kn*A_eff*(T[t,x]-T[t,x-1])/Dx**2


Tb1n = np.zeros((len(xbn)))
Tb1n[:]=T[-1,:]
T_max = max(Tb1n)

print('Con kIC→∞: T_max = ',round(T_max),'K ó',round(T_max-273.15),'C')

## Con kIC dada

def k_fun(pos):
    x = xbn[pos]
    k1 = PCB.kx
    k2 = PCB.kx2
    from numpy import heaviside as H
    val = k1 + (H(x-L1,dis)-H(x-L2,dis)+H(x-L3,dis)-H(x-L5,dis)+H(x-L6,dis)-H(x-L7,dis))*(k2-k1)
    return val

k=np.zeros((len(xbn)))
rho_c=np.zeros((len(xbn)))
```

```
    phii=np.zeros((len(xbn)))
    for x in range(0,len(xbn)):
            k[x] = k_fun(x)
            rho_c[x] = rho_c_fun(x)
            phii[x] = phii_fun(x)

    T = T_wall * np.ones((M+1,N+1)) # T(t,x) inicial
    for t in range(0,len(tb)-1):
        if t%disp == 0: print('Tiempo de simulación:',tb[t], 's \Temperatura máxima:',np.amax(T[t,:]),'K')

        # Condiciones de contorno
        T[t,0]=T_wall
        T[t,N]=T_wall

        for x in range(1,len(xbn)-1):
            # Propiedades
            kp = (k[x+1]+k[x])/2
            kn = (k[x]+k[x-1])/2
            # Euler explícito
            T[t+1,x] = T[t,x] + Dt/(rho_c[x]*A_eff)*(kp*A_eff*(T[t,x+1]-T[t,x])/Dx**2-kn*A_eff*(T[t,x]-T[t,x-1])/Dx**2


    Tb2n = np.zeros((len(xbn)))
    Tb2n[:]=T[-1,:]
    T_max = max(Tb2n)

    print('Con la kIC dada: T_max = ',round(T_max),'K ó',round(T_max-273.15),'C')

    # Guardar resultados
    save_result('xbn',xbn)
    save_result('Tb1n',Tb1n)
    save_result('Tb2n',Tb2n)
```

if apartado_c == 'yes':
print('*** c ***')
c) Considerando que se transmite calor por radiación, con una emisividad media de 0,7 por el lado de los
componentes, y de 0,5 por la cara opuesta, con una caja electrónica que se puede suponer negra y a 45 ºC,
determinar la temperatura máxima linealizando las pérdidas radiativas y con disipación uniforme.

```
eps1 = 0.7
p1 = PCB.Ly
eps2 = 0.5
p2 = PCB.Ly
T_inf = 45+273.15 #K
sigma = 5.67e-8  #W/m^2·K^4
T_media = T_inf

phi = 3*IC.W/PCB.V
A_eff = PCB.Ax
k_eff = PCB.kx
L = PCB.Lx

## Solución analítica
print('Solución analítica')

# T(x) = c1 * exp(a**0.5*x) + c2 * exp(-a**0.5*x) + T_chi

a = 4*(p1*eps1+p2*eps2)*sigma*T_media**3 / (k_eff*A_eff)
eta = a**0.5
T_chi = T_inf + phi*A_eff/(4*(p1*eps1+p2*eps2)*sigma*T_media**3)
T_gorro0 = T_wall - T_chi

c2 = T_gorro0 * (1-exp(eta*L))/(exp(-eta*L)-exp(eta*L))
c1 = T_gorro0 - c2

# Discretización de la solución

N = 50
xc = np.linspace(0,L,N+1)
Tc = c1 * np.exp(eta*xc) + c2 * np.exp(-eta*xc) + T_chi
```

```
T_max = c1 * np.exp(eta*L/2) + c2 * np.exp(-eta*L/2) + T_chi

print(' T_max = ',round(T_max),'K ó',round(T_max-273.15),'C')

# Guardar resultados
save_result('xc',xc)
save_result('Tc',Tc)

## Solución numérica
print('Solución numérica')

# Discretización

if numerico == 'yes':

    L1 = PCB.Lx/2 - IC.Lx- IC.pitch- IC.Lx/2
    L2 = IC.Lx + L1
    L3 = IC.pitch + L2
    L4 = IC.Lx/2 + L3
    L5 = IC.Lx/2 + L4
    L6 = IC.pitch + L5
    L7 = IC.Lx + L6
    L8 = PCB.Lx

    L = PCB.Lx     # Espacio de simulación
    T = 5000       # Tiempo de simulación
    N = 70         # Número de elementos espaciales
    M = int(1e5)   # Número de elementos temporales (ver criterio)
    Dx = L/N
    Dt = T/M
    xcn = np.linspace(0,L,N+1)
    tc  = np.linspace(0,T,M+1)

    p = 0
    h = 0

    # Estabilidad
    a=PCB.kx/(PCB.rho_c)            #Diffusivity [m^2/s]
    Fo=a*Dt/(Dx*Dx)                 #Fourier's number
    Bi=h*p*Dx/(k_eff*A_eff/Dx)      #Biot's number

    if (1-Fo*(2+Bi)) < 0:
        print('This is unstable increase number of time steps')

    # Propiedades
    k_eff = PCB.kx
    rho_c = PCB.rho_c
    phi = 3*IC.W/PCB.V

    T = T_wall * np.ones((M+1,N+1)) # T(t,x) inicial
    for t in range(0,len(tc)-1):
        if t%disp == 0: print('Tiempo de simulación:',tc[t], 's \Temperatura máxima:',np.amax(T[t,:]),'K')

        # Condiciones de contorno
        T[t,0]=T_wall
        T[t,N]=T_wall

        for x in range(1,len(xcn)-1):
            # Euler explícito
            T[t+1,x] = T[t,x] + Dt/(rho_c*A_eff)*(k_eff*A_eff*(T[t,x+1]-T[t,x])/Dx**2-k_eff*A_eff*(T[t,x]-T[t,x-1])/Dx*


    Tcn = np.zeros((len(xcn)))
    Tcn[:]=T[-1,:]
    T_max = max(Tcn)

    print(' T_max = ',round(T_max),'K ó',round(T_max-273.15),'C')

    # Guardar resultados
    save_result('xcn',xcn)
    save_result('Tcn',Tcn)
```

if apartado_d == 'yes':
print('*** d ***')
d) Resolver el caso anterior pero sin linealizar y con la disipación no uniforme.

```
eps1 = 0.7
p1 = PCB.Ly
eps2 = 0.5
p2 = PCB.Ly
T_inf = 45+273.15 #K
sigma = 5.67e-8  #W/m^2·K^4

phi = IC.W/(IC.Lx*PCB.Ly*PCB.Lz)
A_eff = PCB.Ax
k_eff = PCB.kx

## Solución numérica
print('Solución numérica')

# Discretización

if numerico == 'yes':

    L1 = PCB.Lx/2 - IC.Lx- IC.pitch- IC.Lx/2
    L2 = IC.Lx + L1
    L3 = IC.pitch + L2
    L4 = IC.Lx/2 + L3
    L5 = IC.Lx/2 + L4
    L6 = IC.pitch + L5
    L7 = IC.Lx + L6
    L8 = PCB.Lx

    L = PCB.Lx     # Espacio de simulación
    T = 5000       # Tiempo de simulación
    N = 70         # Número de elementos espaciales
    M = int(1e5)   # Número de elementos temporales (ver criterio)
    Dx = L/N
    Dt = T/M
    xdn = np.linspace(0,L,N+1)
    td  = np.linspace(0,T,M+1)

    p = 0
    h = 0

    # Estabilidad
    a=PCB.kx/(PCB.rho_c)            #Diffusivity [m^2/s]
    Fo=a*Dt/(Dx*Dx)                 #Fourier's number
    Bi=h*p*Dx/(k_eff*A_eff/Dx)      #Biot's number

    if (1-Fo*(2+Bi)) < 0:
        print('This is unstable increase number of time steps')

    # Propiedades
    def k_fun(pos):
        x = xdn[pos]
        k1 = PCB.kx
        k2 = PCB.kx2
        dis = 1
        from numpy import heaviside as H
        val = k1 + (H(x-L1,dis)-H(x-L2,dis)+H(x-L3,dis)-H(x-L5,dis)+H(x-L6,dis)-H(x-L7,dis))*(k2-k1)
        return val

    def rho_c_fun(pos):
        x = xdn[pos]
        rho_c1 = PCB.rho_c
        rho_c2 = PCB.rho_c2
        dis = 1
        from numpy import heaviside as H
        val = rho_c1 + (H(x-L1,dis)-H(x-L2,dis)+H(x-L3,dis)-H(x-L5,dis)+H(x-L6,dis)-H(x-L7,dis))*(rho_c2-rho_c1)
        return val

    def phii_fun(pos):
        x = xdn[pos]
```

```
        dis = 1
        from numpy import heaviside as H
        val = (H(x-L1,dis)-H(x-L2,dis)+H(x-L3,dis)-H(x-L5,dis)+H(x-L6,dis)-H(x-L7,dis))*(phi)
        return val

    k=np.zeros((len(xdn)))
    rho_c=np.zeros((len(xdn)))
    phii=np.zeros((len(xdn)))
    for x in range(0,len(xdn)):
            k[x] = k_fun(x)
            rho_c[x] = rho_c_fun(x)
            phii[x] = phii_fun(x)

    T = T_wall * np.ones((M+1,N+1)) # T(t,x) inicial

    for t in range(0,len(td)-1):
        if t%disp == 0: print('Tiempo de simulación:',td[t], 's \Temperatura máxima:',np.amax(T[t,:]),'K')

        # Condiciones de contorno
        T[t,0]=T_wall
        T[t,N]=T_wall

        for x in range(1,len(xdn)-1):
            # Propiedades
            kp = (k[x+1]+k[x])/2
            kn = (k[x]+k[x-1])/2
            # Euler explícito
            T[t+1,x] = T[t,x] + Dt/(rho_c[x]*A_eff)*(kp*A_eff*(T[t,x+1]-T[t,x])/Dx**2-kn*A_eff*(T[t,x]-T[t,x-1])/Dx**2


    Tdn = np.zeros((len(xdn)))
    Tdn[:]=T[-1,:]
    T_max = max(Tdn)

    print(' T_max = ',round(T_max),'K ó',round(T_max-273.15),'C')

    # Guardar resultados
    save_result('xdn',xdn)
    save_result('Tdn',Tdn)
```

if apartado_e == 'yes':
print('*** e ***')
e) Resolver el problema térmico bidimensional estacionario y comparar el perfil central de temperaturas con
el del caso anterior

```
eps1 = 0.7
eps2 = 0.5
T_inf = 45+273.15 #K
sigma = 5.67e-8  #W/m^2·K^4

z_eff = PCB.Lz
phi = IC.W/(IC.Lx*IC.Ly*z_eff)


## Solución numérica
print('Solución numérica')

# Discretización

if numerico == 'yes':

    L1x = round(PCB.Lx/2 - IC.Lx- IC.pitch- IC.Lx/2,8)
    L2x = IC.Lx + L1x
    L3x = IC.pitch + L2x
    L4x = IC.Lx/2 + L3x
    L5x = IC.Lx/2 + L4x
    L6x = IC.pitch + L5x
    L7x = IC.Lx + L6x
    L8x = PCB.Lx
```

```python
L1y = round((PCB.Ly - IC.Ly)/2,8)
L2y = IC.Ly + L1y
L3y = PCB.Ly

Lx = PCB.Lx     # Espacio de simulación
Ly = PCB.Ly     # Espacio de simulación
T = 3250        # Tiempo de simulación
Nx = 70         # Número de elementos espaciales
Ny = 40         # Número de elementos espaciales
M = int(1e5)    # Número de elementos temporales (ver criterio)
Dx = Lx/Nx
Dy = Ly/Ny
Dt = T/M
xen = np.linspace(0,Lx,Nx+1)
yen = np.linspace(0,Ly,Ny+1)
te  = np.linspace(0,T,M+1)

p = 0
h = 0

# Estabilidad
a=PCB.kx/(PCB.rho_c)            #Diffusivity [m^2/s]
Fo=a*Dt/(Dx*Dx)                 #Fourier's number
Bi=h*p*Dx/(PCB.kx*PCB.Ax/Dx)      #Biot's number

if (1-Fo*(2+Bi)) < 0:
    print('This is unstable increase number of time steps')

# Propiedades
def k_fun(posx,posy):
    x = xen[posx]
    y = yen[posy]
    k1 = PCB.kx #En el plano son iguales
    k2 = PCB.kx2
    dis = 1
    from numpy import heaviside as H
    if y>=0 and y<L1y:
        val = k1
    elif y>=L1y and y<=L2y:
        val = k1 + (H(x-L1x,dis)-H(x-L2x,dis)+H(x-L3x,dis)-H(x-L5x,dis)+H(x-L6x,dis)-H(x-L7x,dis))*(k2-k1) # Los IC
    elif y>L2y and y<=L3y:
        val = k1
    return val

def rho_c_fun(posx,posy):
    x = xen[posx]
    y = yen[posy]
    rho_c1 = PCB.rho_c
    rho_c2 = PCB.rho_c2
    dis = 1
    from numpy import heaviside as H
    if y>=0 and y<L1y:
        val = rho_c1
    elif y>=L1y and y<=L2y:
        val = rho_c1 + (H(x-L1x,dis)-H(x-L2x,dis)+H(x-L3x,dis)-H(x-L5x,dis)+H(x-L6x,dis)-H(x-L7x,dis))*(rho_c2-rho_
    elif y>L2y and y<=L3y:
        val = rho_c1
    return val

def phii_fun(posx,posy):
    x = xen[posx]
    y = yen[posy]
    dis = 1
    from numpy import heaviside as H
    if y>=0 and y<L1y:
        val = 0
    elif y>=L1y and y<=L2y:
        val = (H(x-L1x,dis)-H(x-L2x,dis)+H(x-L3x,dis)-H(x-L5x,dis)+H(x-L6x,dis)-H(x-L7x,dis))*phi # Los IC en x
    elif y>L2y and y<=L3y:
        val = 0
    return val

k=np.zeros((len(xen),len(yen)))
rho_c=np.zeros((len(xen),len(yen)))
```

```python
phii=np.zeros((len(xen),len(yen)))
for y in range(0,len(yen)):
    for x in range(0,len(xen)):
        k[x,y] = k_fun(x,y)
        rho_c[x,y] = rho_c_fun(x,y)
        phii[x,y] = phii_fun(x,y)

save_result('k',k)
save_result('rho_c',rho_c)
save_result('phii',phii)

save_result('xen',xen)
save_result('yen',yen)

T = T_wall * np.ones((M+1,Nx+1,Ny+1)) # T(t,x,y) inicial

for t in range(0,M):
    if t%disp == 0: print('Tiempo de simulación:',te[t],'s \Temperatura máxima:',np.amax(T[t,:,:]),'K')

    # Condiciones de contorno en la pared
    T[t,0,:]=T_wall
    T[t,Nx,:]=T_wall

    for y in range(1,len(yen)-1):
        for x in range(1,len(xen)-1):
            # Propiedades
            kpx = (k[x+1,y]+k[x,y])/2
            knx = (k[x,y]+k[x-1,y])/2
            kpy = (k[x,y+1]+k[x,y])/2
            kny = (k[x,y]+k[x,y-1])/2
            # Euler explícito
            T[t+1,x,y] = T[t,x,y] + Dt/(rho_c[x,y]*z_eff)*(\
            +kpx*z_eff*(T[t,x+1,y]-T[t,x,y])/Dx**2\
            -knx*z_eff*(T[t,x,y]-T[t,x-1,y])/Dx**2\
            +kpy*z_eff*(T[t,x,y+1]-T[t,x,y])/Dy**2\
            -kny*z_eff*(T[t,x,y]-T[t,x,y-1])/Dy**2\
            +phii[x,y]*z_eff \
            -( ((eps1+eps2)**0.25*sigma**0.25*T[t,x,y])**4 - ((eps1+eps2)**0.25*sigma**0.25*T_inf)**4))

    # Condiciones de adiabaticidad
    T[t+1,:,0]=T[t+1,:,1]
    T[t+1,:,Ny]=T[t+1,:,Ny-1]

Ten = np.zeros((len(xen),len(yen)))
Ten[:,:]=T[-1,:,:]
T_max = np.amax(Ten)

print(' T_max = ',round(T_max),'K ó',round(T_max-273.15),'C')

# Guardar resultados
save_result('Ten',Ten)
```