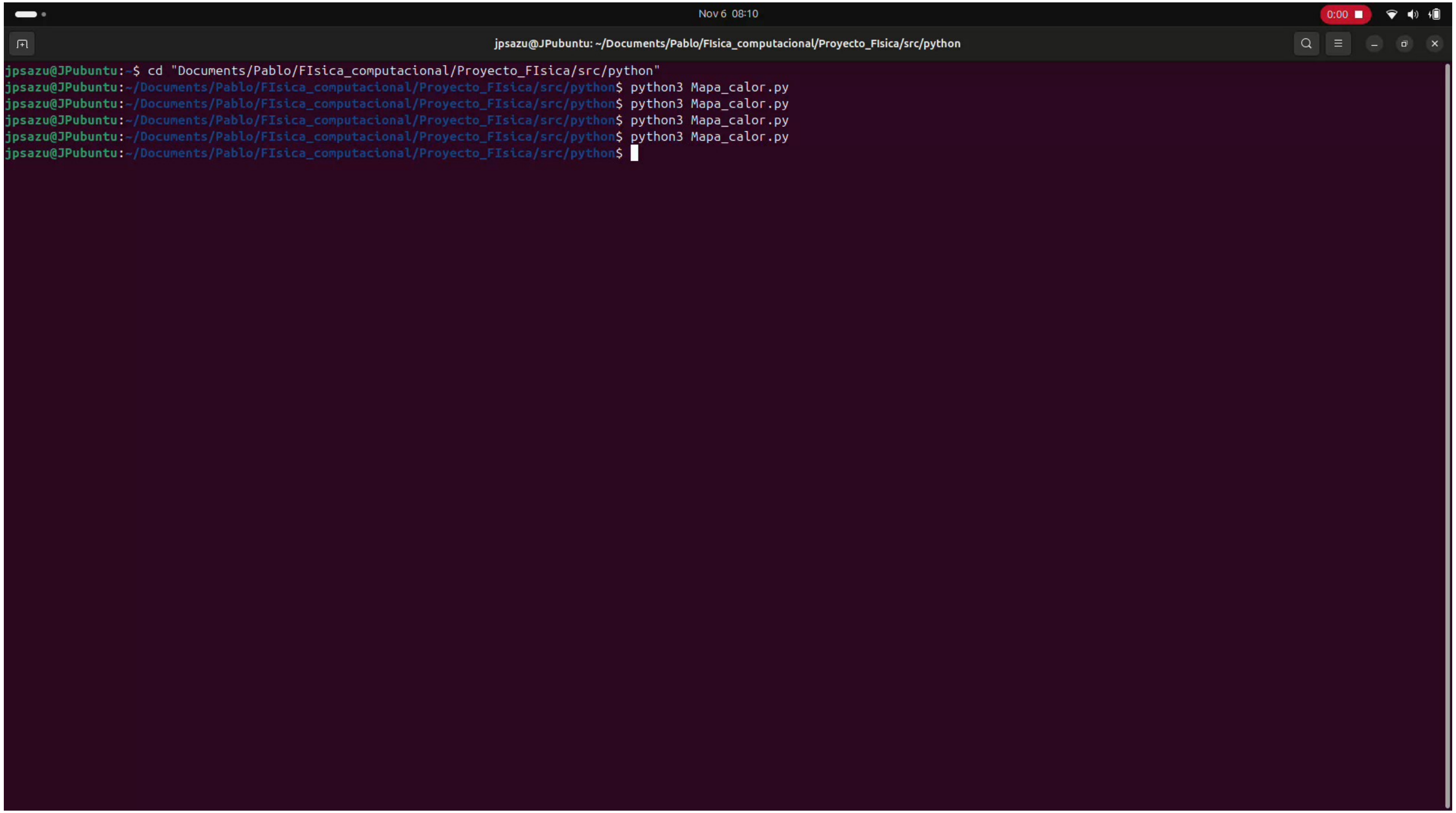
Proyecto Fisica Computacional

ECUACIÓN DE CALOR EN DOS DIMENSIONES

Profesor: Marlon Brenes

Estudintes: Jose Pablo Sanchez, Pavel Chaves, Andres Molina

Video



Nov 6 08:10
0:00
jpsazu@JPubuntu: ~/Documents/Pablo/FIsica_computacional/Proyecto_FIsica/src/python
jpsazu@JPubuntu:~$ cd "Documents/Pablo/FIsica_computacional/Proyecto_FIsica/src/python"
jpsazu@JPubuntu:~/Documents/Pablo/FIsica_computacional/Proyecto_FIsica/src/python$ python3 Mapa_calor.py
jpsazu@JPubuntu:~/Documents/Pablo/FIsica_computacional/Proyecto_FIsica/src/python$ python3 Mapa_calor.py
jpsazu@JPubuntu:~/Documents/Pablo/FIsica_computacional/Proyecto_FIsica/src/python$ python3 Mapa_calor.py
jpsazu@JPubuntu:~/Documents/Pablo/FIsica_computacional/Proyecto_FIsica/src/python$ python3 Mapa_calor.py
jpsazu@JPubuntu:~/Documents/Pablo/FIsica_computacional/Proyecto_FIsica/src/python$

Problema: *ECUACIÓN DE CALOR EN DOS DIMENSIONES*

- El problema que se busca resolver de manera numerica es el siguiente:

Sea $u = u(x, y, t)$ una variable escalar que define la temperatura de una región de dos dimensiones en el plano Cartesiano (x, y) como función del tiempo. La ecuación de calor correspondiente está dada por:

$$\frac{\partial u}{\partial t} = c^2 \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right)$$

Sea el rectángulo $0 < x < L_x, \qquad 0 < y < L_y$

Metodologia: FTCS

- Realizando el cambio $c^2 = \alpha$ y discretizando $u = u(x, y, t)$ tenemos:

$$x_i = i\,\Delta x, \quad i = 0, \ldots, N_x, \qquad y_j = j\,\Delta y, \quad j = 0, \ldots, N_y, \qquad t^n = n\,\Delta t, \qquad n = 0, 1, 2, \ldots$$

De tal modo que: $u^n_{i,j} \approx u(x_i, y_j, t^n).$

Usando diferencias finitas:

$$\frac{\partial u}{\partial t}(x_i, y_j, t^n) \approx \frac{u^{n+1}_{i,j} - u^n_{i,j}}{\Delta t} \quad \frac{\partial^2 u}{\partial x^2}(x_i, y_j, t^n) \approx \frac{u^n_{i+1,j} - 2u^n_{i,j} + u^n_{i-1,j}}{\Delta x^2} \quad \frac{\partial^2 u}{\partial y^2}(x_i, y_j, t^n) \approx \frac{u^n_{i,j+1} - 2u^n_{i,j} + u^n_{i,j-1}}{\Delta y^2}.$$

Metodologia

- Sustituyendo en la ecuacion de calor y despejando se tiene:

$$\boxed{u_{i,j}^{n+1} = u_{i,j}^{n} + \lambda_x \left(u_{i+1,j}^{n} - 2u_{i,j}^{n} + u_{i-1,j}^{n}\right) + \lambda_y \left(u_{i,j+1}^{n} - 2u_{i,j}^{n} + u_{i,j-1}^{n}\right)}$$

- con $\lambda_x = \dfrac{\alpha \Delta t}{\Delta x^2}, \qquad \lambda_y = \dfrac{\alpha \Delta t}{\Delta y^2}.$ $\qquad i = 1, \ldots, N_x - 1, \qquad j = 1, \ldots, N_y - 1.$

Usamos condiciones de frontera de Newman => en los bordes: $\dfrac{\partial u}{\partial n} = 0$
Es decir: $u_{\text{frontera}} = u_{\text{nodo vecino interior}}$

Se nececita establecer los parametros acorde con la condicion de estabilidad:

$$\lambda_x + \lambda_y \leq \frac{1}{2}, \quad \Rightarrow \quad \Delta t \leq \frac{1}{2\alpha \left(\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2}\right)}.$$

Si $\Delta x = \Delta y = h$, entonces: $\Delta t \leq \dfrac{h^2}{4\alpha}.$

Recursos

- Python 3.10 o superior
- Libreria numpy
- Libreria matplotlib.pyplot
- Libreria matplotlib.animation
- Compilador C++
- GNU plot
- OpenMP
- Git y GitHub
- Mkdocs _ GitHub pages

Algoritmo General

- Dado que el valor de $u_{i,j}$ en el tiempo $n+1$ solo depende de valores de u en el tiempo n, se usan dos matrices, una para u^{n+1} donde se calculan los nuevos valores, y una par u^{n}, con los valores anteriores. Las referencias a estas matrices se intercambian en cada iteracion, sin requerir nueva memoria cada vez.
- La funcion ***update*** calcula los nuevos valores internos de de u segun la ecuacion dada:
- $$\boxed{u_{i,j}^{n+1} = u_{i,j}^{n} + \lambda_x \left(u_{i+1,j}^{n} - 2u_{i,j}^{n} + u_{i-1,j}^{n}\right) + \lambda_y \left(u_{i,j+1}^{n} - 2u_{i,j}^{n} + u_{i,j-1}^{n}\right)}$$
-
- Los los bordes (frontera) toman su valor de sus vecinos internos

Python

- En cada iteracion, la funcion ***update*** calcula los nuevos valores de de u y genera y retorna una imagen de “calor” a partir de dichos valores
- En el programa principal (***main***), la funcion ***FuncAnimation*** de Python se encarga de invocar iterativamente la funcion ***update*** e ir mostrando el video con esas imagenes en secuencia

```
animation.FuncAnimation(fig, update, frames=Nt, interval=50, blit=False,
fargs=(u,Nx,Ny,cento_medios,cx,cy,dx,dy,alpha,dt,img,ax,fig,Nt))
```

C++ Version Serial

- Se usan dos arreglos de numeros para almacenar la matrices U^n y U^{n+1}
- En cada iteracion, la funcion ***recalcular_matriz*** calcula los nuevos valores de de U como corresponde
- En este caso, a diferencia de la version Pynthon, el ciclo principal que invoca la funcion de recalculo se hace manualmente en el ***main***.
- Al final se almacena en un archivo los datos de la ultima version de la matriz
- Una vez terminando el programa C++, se inoca la libreria ***GNUplot*** y se le pasa el script (*fig.plt*) para generar la imagen final a partir del archivo de datos

C++ Version Paralela

- Es semejante a la version serial, pero usa paralelismo de memoria compartida con ***OpenMP***.
- Dentro de la funcion ***recalcular_matriz***, usando “*#pragma omp for*” se paraleliza el recalculo de bloques de filas de la matriz.
- No se dan condiciones de carrera, pues los nuevos valores de la matriz solo dependen de los valores previos.
- Al final se almacena en un archivo los datos de la ultima version de la matriz
- Una vez terminando el programa C++, se inoca la libreria ***GNUplot*** y se le pasa el script (*fig.plt*) para generar la imagen final a partir del archivo de datos

Resultados

- ✔ Se usó FTCS para resolver numericamente la ecuacion de calor
- ✔ Se implementó una version en Python y se generó un video de la difusion del calor en una placa bidimensional
- ✔ Se desarrolló una version en C++ y se utilizó GNUplot para mostrar el mapa de calor final
- ✔ Se paralelizó la vesion C++ usando memoria compartida aplicando OpenMP
- ✔ Se usó Git y GitHub para el repositorio de versionamiento
- ✔ Se aplicó MkDocs para gnerar el sitio web de documentacion del proyecto