Recomendaciones de eficiencia

Este semestre hemos estado usando `Python` principalmente porque es fácil de manejar, sin embargo, esa facilidad tiene precio: `Python` es lento. En verdad, no es tan lento, pero ocurre que si programamos `Python` como quien programa `C++` o `FORTRAN`, estamos siendo muy ineficientes. `Python` programado de forma "Pythonica" puede llegar a ser bastante rápido. En general, los ejercicios que hicimos en los laboratorios eran pequeños, y podían ser ineficientes sin sufrir mucho, pero probablemente la experiencia en el Proyecto 1 sea distinta. Este documento les dará algunas recomendaciones para que su código corra más rápido; no es obligatorio seguirlas, pero puede hacerles la vida más fácil. De hecho, si no entienden bien que es lo que hace cada recomendación, quizás sea mejor que se queden con lo que tienen, que entienden bien.

Recomendación 1: No al loop! Loops `for` en `Python` no es una buena idea: es muy lento, y hay maneras más "Pythonicas" de hacerlo. Digamos que, por ejemplo, queremos calcular la derivada x de una variable bidimensional u

$$\frac{\partial u}{\partial x} \approx \frac{u_{i+1,j} - u_{i-1,j}}{2\Delta x} \tag{1}$$

El clásico loop que deben hacer para resolver esto sería

```
dudx = numpy.zeros((ny,nx))
for j in range(1,ny-1):
    for i in range(1,nx-1):
        dudx[j,i] = (u[j,i+1]-u[j,i-1])/(2*dx)
```

lo que es correcto, pero ineficiente si usan `Python` (no así si usamos `C++` o `FORTRAN`). *Nota:* Fíjense que estamos guardando la data en el orden `u[j,i]`, a pesar que normalmente usamos la notación $u_{i,j}$, pero no hay nada de malo en hacerlo al revés.

Una forma más "Pythonica" de hacer esto es operar sobre los arreglos completos (miren la Figura 1): la matriz `u[1:-1,1:-1]` contiene todos los puntos "interiores" de la malla, `u[1:-1,0:-2]` los puntos interiores desfasados uno hacia atrás en el eje x y `u[1:-1,2:]` los puntos interiores desfasados un puesto hacia adelante en el eje x. Por lo tanto, al restar `u[1:-1,2:]-u[1:-1,0:-2]`, estamos restando "simultáneamente" puntos que están en `i+1` menos puntos que están en `i-1`, para todos los `i` en `1:-1`. De esta forma, el doble loop anterior se puede reemplazar por:

```
dudx = numpy.zeros((ny,nx))
dudx[1:-1,1:-1] = (u[1:-1,2:]-u[1:-1,:-2])/(2*dx)
```

lo que es **mucho** más rápido!

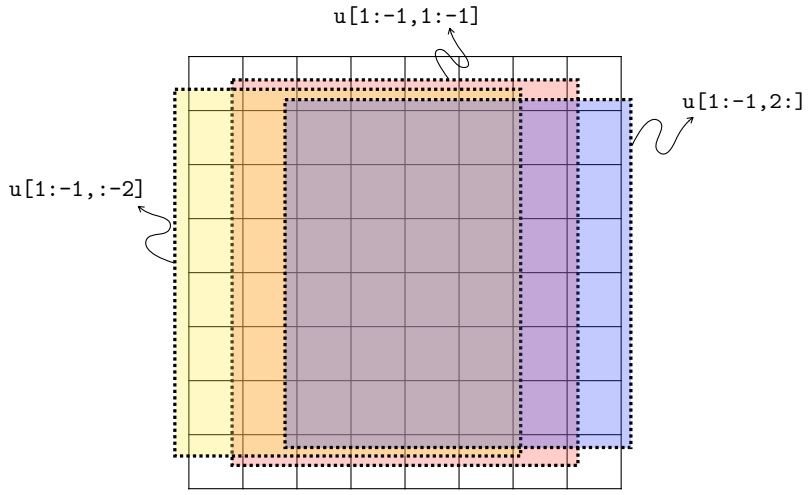

Figura 1: Malla

Recomendación 2: la matriz es constante Uno puede pasar harto tiempo calculando la matriz de coeficientes para la ecuación de Poisson para la presión. Ahora, hay que darse cuenta que es una matriz de coeficientes que dependen de la posición de la malla, la cual no cambia en el tiempo, por lo que no es necesario hacerlo en cada paso de tiempo: háganla una sola vez, y la reutilizan.

Recomendación 3: la matriz está llena de ceros! Las matrices llenas de ceros se conocen como matrices ralas (o *sparse*), y son muy comunes en métodos numéricos. De hecho, las encuentran en diferencias finitas, volúmenes finitos y elementos finitos, y hay mucho trabajo en hacer de las operaciones sobre matrices ralas lo más eficiente posible.

Lo primero que podemos decir al ver la matriz de Poisson es que está llena de ceros ¿para qué guardar todos los ceros? Eso es una buena pregunta, y la respuesta es simple: no es necesario. El gran problema es que al resolver el sistema lineal con `scipy.linalg.solve`, `Python` va a recorrer la matriz completa, y operará casi siempre sobre ceros, haciendo un montón de cálculos inútiles.

Afortunadamente, nos podemos apoyar en gente que ha visto este tipo de problemas antes, y que ha inventado formatos más eficientes para operar sobre matrices ralas. Un formato muy usado es el *compressed storage row* (CSR), y las funciones de `scipy` reconocen y operan sobre matrices CSR. Para usar esta capacidad deben importar unas funciones

```
from scipy import sparse
from scipy.sparse.linalg import spsolve
```

donde `spsolve` no es más que un `solve` especial para matrices ralas. Una vez que tienen la matriz generada en el formato "poco eficiente", cambien el formato de la siguiente forma

```
A_sparse = sparse.csr_matrix(A_dense)
```

y en vez de usar `solve` para resolver el sistema lineal, usen `spsolve`:

```
x = spsolve(A_sparse, b)
```

Verán que estos pequeños cambios harán que su código sea muchísimo más rápido!

Recomendación 5: gráficos visibles En el texto, les recomendamos visualizar sus resultados con la funcin `matplotlib.pyplot.quiver(X,Y,u,v)`, lo cual posiciona vectores con componentes `u` y `v` en las posiciones `X` e `Y`. Sin embargo, si graficamos todos los puntos la figura estar llena de flechas y no se entender nada. Un opcin es usar el siguiente comando:

```
pyplot.quiver(X[::20,::2],Y[::20,::2],
              u[::20,::2],v[::20,::2], headaxislength=5)
```

que plotea cada 20 líneas en el eje y (`::20`), y línea por medio en el eje x (`::2`). Éste es el comando que usé en los gráficos elaborados en el documento.