CS111 F21 Homework 2

Michael Glushchenko, 9403890

October 7, 2021

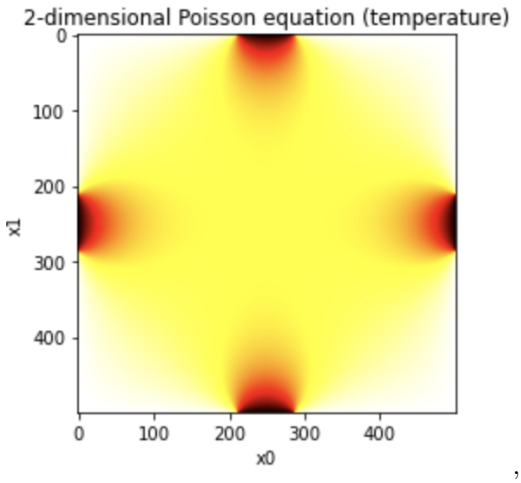
1 Experimenting with *temperature.py*

I used the following Python code to produce the image below:

```
k = 100
A = cs111.make_A(k)

rad_rad = cs111.radiator(k, width = .15, rad_temp = 32., wall_temp = 400.)

b = cs111.make_b(k, rad_rad, rad_rad, rad_rad, rad_rad)
x = scipy.sparse.linalg.spsolve(A,b) # ugly!
```



,

This was obtained by making all 4 walls very hot, while making each wall's radiator both small and cold. It's cool how it produces a circle of warmth in the middle (the center of the circle having the lowest heat in the room). I tried messing around with putting different-sized radiators in different spots, but the colors mix too much with very big radiators ($width > 0.85$). The result of having those two big warm radiators on the sides, and a small cold one on top, was a thing that looked like a face without a nose. It'd be cool to make the walls longer, and put multiple radiators along each wall, the face would look smoother.

2 Inspecting the *make_b* function

2.1 Answer: 10000

There are $k * k$ elements in b, so $100 * 100 =$ **10000** elements when $k = 100$.

```
# Start with a vector of zeros
ndim = k*k
b = np.zeros(shape = ndim)
```

2.2 Answer: 396

Array b will have the most nonzero elements when $top, bottom, left, right \neq 0$, and $-bottom \neq right \neq -top$ and $-bottom \neq left \neq -top$, reasoning in **2.3**.

```
# Fill in the four boundaries as appropriate
b[0        : k       ] += 1 # first 100 elements (0-99)
b[ndim - k : ndim    ] += 2 # last 100 elements (9900-9999)
b[0        : ndim : k] += 4 # every 100 elements from 0 to 9900
b[k-1      : ndim : k] += 8 # every 100 elements from 99 to 9999

print("First 100 elements: \n", b[0        : k       ])
print("Last 100 elements: \n", b[ndim - k : ndim    ])
print("Number of nonzero elements in b: ", np.count_nonzero(b))
print("Number of zero elements in b: ", len(b) - np.count_nonzero(b))
```

```
First 100 elements:
 [5. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 9.]
Last 100 elements:
 [6. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2.
  2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2.
  2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2.
  2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2.
  2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2.
  2. 2. 2. 2. 2. 2. 2. 2. 2. 10.]
Number of nonzero elements in b: 396
Number of zero elements in b: 9604
```

2.3 Reasoning

When conditions in **2.2** hold, the first k elements of b will be nonzero because $top \neq 0$. The last k elements will be nonzero because $bottom \neq 0$. Every k'th element starting at index 0 will be nonzero because $left \neq 0$. And every k'th element starting at index 99 is nonzero because $right \neq 0$.

So, $4 * 100 = 400$ changes are made to elements of b, but $right$ makes a change to 1 element in the first 100 and 1 element in the last 100. So does $left$. Thus we over-counted elements at indexes $0, 99, 9900$, and 9999. So, the max nonzero elements b could have is $400 - 4 =$ **396** (when conditions specified above hold). The other 9604 elements must be 0, as no changes are ever made to them.

3 Permutation Matrix P

$$P = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$

3.1 Permutation v

Expression $P@A$ shuffles rows of matrix A as such:

1. Row 0 of $P@A$ gets row 1 of A (P has a 1 at index 1 of row 0).
2. Row 1 of $P@A$ gets row 3 of A (P has a 1 at index 3 of row 1).
3. Row 2 of $P@A$ gets row 2 of A (P has a 1 at index 2 of row 2).
4. Row 3 of $P@A$ gets row 0 of A (P has a 1 at index 0 of row 3).

Thus we have $\mathbf{v} = \mathbf{[1,3,2,0]}$.

3.2 Permutation w

Similar to the logic in **3.1**, $A@P$ shuffles the columns of A rather than the rows: column 3 of A goes to column 0 of A@P, column 0 of A goes to column 1 of A@P, and so on.
So, our w is $\mathbf{w} = \mathbf{[3,0,2,1]}$.

3.3 Python Test

Using a few lines of Python, we test whether v and w above are correct:

```
P = np.array([[0,1,0,0], [0,0,0,1], [0,0,1,0], [1,0,0,0]])
v = np.array([1,3,2,0])
w = np.array([3,0,2,1])

# Testing v and w on a few randomized 4x4 matrices:
for i in range(100000):
    A = np.random.random((4, 4))
    if (not ((A[v,:]) == P@A).all()):
        print("vector v is wrong: \n", A[v,:], "\n does not equal to \n", P @ A)
    if (not ((A[:,w]) == A@P).all()):
        print("vector w is wrong: \n", A[:,w], "\n does not equal to \n", A @ P)
print("Loop finished successfully")
```

```
Loop finished successfully
```

Indeed, $v = [1, 3, 2, 0]$ and $w = [3, 0, 2, 1]$.

4 Writing $Usolve()$

Here's my code for $Usolve()$:

```
def Usolve(U, y, unit_diag=False):
    # Check the input
    m, n = U.shape
    assert m == n, "matrix U must be square"
    assert np.all(np.triu(U) == U), "matrix U must be upper triangular"
    assert np.all(np.diag(U) != 0), "matrix U must have nonzeroes on the diagonal"

    # Make a copy of the rhs that we will transform into the solution
    assert y.ndim == 1, "right-hand side must be a 1-dimensional vector"
    assert y.shape[0] == n, "right-hand side must be same size as matrix"
    x = y.astype(np.float64).copy()

    # Backward solve
    for col in reversed(range(n)):
        x[col] = (x[col] - (U[col, col+1:] @ x[col+1:])) / U[col, col]

    return x
```

Below, I have $Usolve()$ tested on its own in loop 1, while loop 2 tests $Usolve()$ with $LUsolve()$, which uses $cs111.LUfactor()$, $cs111.Lsolve()$, and my $Usolve()$:

```
# testing Usolve() on its own:
for i in range(10000):
    b = np.random.rand(4)
    U = np.triu(np.random.random((4,4)))

    answer = np.around(scipy.sparse.linalg.spsolve(U,b),3) # ugly!
    our_answer = np.around(Usolve(U, b),3)

    if(not (answer==our_answer).all()):
        print("Something's wrong with loop 1.")
print("Loop 1 finished successfully")

# testing Usolve() with LUsolve():
for i in range(10000):
    b = np.random.rand(4)
    U = np.triu(np.random.random((4,4)))
    our_answer, rel_res = LUsolve(U, b)

    if(rel_res > 0.0001):
        print("Something's wrong with loop 2.")
print("Loop 2 finished successfully")
```

```
Loop 1 finished successfully
Loop 2 finished successfully
```

Both loops finish successfully without error.