600.112: Introduction to Programming
for Scientists and Engineers

Assignment 6: Heat Transfer∗†

Peter H. Fröhlich  Joanne Selinski
phf@cs.jhu.edu  joanne@cs.jhu.edu

Due Date: 2014/10/14 before 10:00 pm!

Introduction

The sixth assignment for 600.112: Introduction to Programming for Scientists and Engineers is all about heat transfer and how to simulate it.

There are three things to do: First you’ll write a program to solve a simple one-dimensional heat transfer problem for a metal rod. Second you’ll write a program to solve a more complex two-dimensional heat transfer problem for a metal plate. Third you’ll use the program for two-dimensional heat transfer to explore what happens to your PC when the heatsink of your CPU doesn’t make good contact.

There are detailed submission instructions on Piazza which you should follow to the letter! You can lose points if you create more work than necessary for the graders by not following the instructions.

Background

What is heat? It’s probably in bad taste to start with a joke about the miserably humid 100°F Baltimore summer, so let’s be more scientific.

Thermodynamics tells us that heat—on a macroscopic level—is energy: the internal energy of a system is the sum of heat supplied to the system and the amount of work done on it. Statistical mechanics tells us that heat—on a microscopic level—is movement: the atoms of a system move (or vibrate) faster the hotter the system happens to be.

Heat can be transferred in various ways: by conduction (coal heats oven), by convection (flame heats pan), or by radiation (sun heats earth). We’ll only consider heat transfer by conduction for this project, and we’ll also restrict our attention to solids.

Join us in a Gedankenexperiment! Imagine two copper cubes of equal size and mass. One of them has a temperature of 25°C while the other has a temperature of 75°C. Imagine further that these copper cubes are perfectly insulated from the surrounding world and that there is no heat transfer by convection or radiation. What will happen when we bring the two cubes into “perfect” physical contact?

Along the area of contact, the atoms of the 75°C cube will vibrate faster than those of the 25°C cube. Every now and then a 75°C atom will “hit” a 25°C atom.

∗Special thanks to Jimmy Su and William Yu for helping us develop this project.
†Disclaimer: This is not a course in physics or biology or epidemiology or even mathematics. Our exposition of the science behind the projects cuts corners whenever we can do so without lying outright. We are not trying to teach you anything but computer science!
atom and transfer its “kinetic energy” in the process, creating a 75°C atom in the 25°C cube and vice versa. Since this happens all along the area of contact, pretty soon we’ll have an even mixture of 75°C and 25°C atoms wiggling around in these two “layers” of each cube. So on average, across all the atoms along the area of contact, we have a temperature of \( \frac{25°C + 75°C}{2} = 50°C \).

What about the “next two layers” of atoms in each cube? They will in turn “exchange” equal amounts of 75°C and 25°C atoms with the contact layers, and so on. Eventually the average temperature of both cubes will be 50°C, and it will remain 50°C even if we separate them again.

**Mathematics**

The basic mathematical model for one-dimensional heat transfer by conduction is a second-order partial differential equation:

\[
\frac{\partial^2 T}{\partial x^2} = 0
\]

Sometimes we can find analytical solutions to such equations, but often all we can do is approximate the solution numerically. One approach to a numerical approximation is to transform the differential equations into finite difference equations:

\[
\frac{df}{dx} \approx \frac{f(x + \Delta x) - f(x)}{\Delta x}
\]

We discretize the continuous \( x \) domain into \( n \) points \( p_1, p_2, \ldots, p_n \) spaced \( \Delta x \) apart; pick some point \( p_m \) and approximate the first derivatives at \( p_m - \frac{1}{2} \Delta x \) and \( p_m + \frac{1}{2} \Delta x \) as follows:

\[
\frac{\partial T}{\partial x} \bigg|_{m-1/2} \approx \frac{T_m - T_{m-1}}{\Delta x}
\]

\[
\frac{\partial T}{\partial x} \bigg|_{m+1/2} \approx \frac{T_{m+1} - T_m}{\Delta x}
\]

Then we can in turn approximate the second derivative at \( p_m \) like this:

\[
\frac{\partial^2 T}{\partial x^2} \bigg|_m \approx \frac{\frac{\partial T}{\partial x} \bigg|_{m+1/2} - \frac{\partial T}{\partial x} \bigg|_{m-1/2}}{\Delta x}
\]

\[
\approx \frac{T_{m+1} - T_m}{\Delta x} - \frac{T_m - T_{m-1}}{\Delta x} = \frac{T_{m+1} - 2T_m + T_{m-1}}{\Delta x^2}
\]

Our approximation for one-dimensional heat transfer by conduction becomes

\[
\frac{T_{m+1} - 2T_m + T_{m-1}}{\Delta x^2} = 0
\]

which—when solved for the point \( p_m \)—confirms our earlier suspicion that heat transfer is averaging:

\[
T_m = \frac{1}{2} (T_{m+1} + T_{m-1})
\]

The obvious extension to the two-dimensional case is also valid:

\[
T_{m,n} = \frac{1}{4} (T_{m-1,n} + T_{m+1,n} + T_{m,n-1} + T_{m,n+1})
\]

There’s a lot to say about the limitations of finite difference equations, but we won’t worry about that here.

1 **Metal Rod (20%)**

Imagine a metal rod of negligible thickness sticking through a building’s brick wall. The inside of the building is cooled to 70°F while the outside of the building is sweltering at 100°F. To keep things simple we’ll assume that the wall itself has no influence whatsoever on the temperature of the rod.

For this problem you will write a program that computes the temperature distribution along the rod inside the wall. Please call your program `rod.py`
and nothing else. Figure 1 shows what the output of your program should look like: The temperature along the rod increases linearly from $70^\circ F$ inside the building to $100^\circ F$ outside the building.

We need to decide two things to solve this problem in Python: First, how should we represent the temperature distribution of the rod? Second, how should we perform the “repeated averaging” of temperatures along the rod?

The temperature at any given point in the rod is a floating point number. Therefore the distribution of temperatures along the rod is a sequence of floating point numbers. The first and last temperatures in this sequence are special because they represent the fixed temperatures inside and outside the building.

If we divide the rod itself into 100 equal-sized segments, the total length of our sequence is 102 elements. The initial temperature of “unknown” interior segments could be set to anything, so let’s set those to 0.0 to begin with. Here is a first version of the program that simply sets up the initial temperature distribution along the rod and plots it:

```python
import matplotlib.pyplot as P
LENGTH = 100

def main():
current = [0.0] * (LENGTH + 2)
current[0] = 70.0
current[-1] = 100.0
P.plot(current)
P.xlabel("segment")
P.ylabel("temperature")
P.show()
main()
```

Note the new form of import we’ve used here. Since it would be rather tedious to keep writing `matplotlib.pyplot` over and over again we abbreviate the imported module to a single capital letter. For a larger program this convention of using a single capital letter may be confusing, but for a rather short program we’ll be okay.

Finding the temperature of a segment inside the rod requires that we average the temperatures of its two neighboring segments. It’s important that we average the two neighbors from the old temperature distribution into a new temperature distribution. So we write a function that “runs down” the current rod and computes a new temperature distribution by averaging the relevant elements of the old temperature distribution:

```python
def average(old):
    new = old[:]
    for i in range(1, LENGTH + 1):
        new[i] = (old[i - 1] + old[i + ↩ 1]) / 2
    return new
```

Note that we only recompute the temperatures inside the rod, the boundary conditions never change. If we keep averaging, this process will “grow” successive approximations to the final temperature distribution from those endpoints toward the center of the rod.

1. In the language of partial differential equations, these two temperatures are our boundary conditions.
2. Otherwise we would average the values inconsistently: one would be from the next step in the simulation while the other would be from the previous step.
But when do we stop the computation? After averaging the temperatures across the rod again and again, we will eventually reach a “steady state” in which none of the temperatures change significantly anymore. So ideally we’d simply check if the temperature distribution before averaging is equal to the temperature distribution after averaging. However, since our temperatures are floats we cannot compare for “exact” equality. Instead we have to figure out if two floating point values are “close enough” so we can consider them “equal” as far as our simulation is concerned. Let’s write a function that performs this “approximate equality check” given a bound $\epsilon$ as discussed in lecture; we’ll use the absolute error version here (for other applications the relative error version may be more appropriate):

```python
def floats_close_enough(a, b):
    return abs(a - b) <= EPSILON
```

So we consider two floats “equal” if they don’t differ by more than $\epsilon$.

Now we can write a function that compares two temperature distributions in the same way and returns `True` if they are “close enough” for our purposes:

```python
def lists_close_enough(a, b):
    assert len(a) == len(b)
    for i in range(len(a)):
        if not floats_close_enough(a[i], b[i]):
            return False
    return True
```

Note how we extended the definition of “close enough” for two floating point values to a sequence: two sequences are “close enough” if all their values are “close enough.”

We can now rewrite the main program to perform the appropriate number of averaging steps:

```python
def main():
    current = [0.0] * (LENGTH + 2)
    current[0] = 70.0
    current[-1] = 100.0
    next = average(current)
    while not lists_close_enough(next, current):
        current = next
        next = average(current)
    P.plot(current)
    P.xlabel("segment")
    P.ylabel("temperature")
    P.show()
```

Starting from the initial temperature distribution, we keep averaging until two consecutive distributions are close enough. Once we reach that “steady state” we plot the final temperature distribution. Done!

**Except for one thing:** We never actually said what $\epsilon$ should be for this simulation! Play with different values for $\epsilon$ (for example 0.1, 0.01, 0.001, . . . ) and find one that actually produces the correct result from Figure 1. For reference, Figure 2 shows what the wrong output for an $\epsilon$ that’s too large looks like.

## 2 Metal Plate (50%)

Imagine a metal plate of negligible thickness that we apply heat sources or cooling agents to. Figure 3 shows what happens to a plate that started out at
$25^\circ C$ when we apply a single heat source of $100^\circ C$: Eventually the entire plate will have a temperature of $100^\circ C$. Figure 4 shows a slightly more interesting scenario: If the plate is surrounded by a cooling agent that keeps the edges at $25^\circ C$, then only a relatively small part of the plate will actually get warm. Figures 5 and 6 illustrate how heat from our heat source spreads through the plate if two boundaries or only one boundary are cooled to $25^\circ C$.

For this problem you will write a program that computes the “steady state” temperature distribution of a metal plate to which a number of heat sources and cooling agents are being applied. Please call your program `plate.py` and nothing else.

The input for your program will be a text file `plate.dat` of the form illustrated in Figure 7. The first line of the text file will give the size of the plate, so in the case of Figure 7 we are dealing with a 10-by-10 plate.\(^3\) This is followed by lines describing the boundary conditions: For a 10-by-10 plate there must be 10 lines of 10 characters each. A capital “C” stands for a “cooling agent” which has a temperature of $25^\circ C$ whereas a capital “H” stands for a “heat source” which has a temperature of $100^\circ C$. Cells marked as heat sources or cooling agents are “fixed” once and for all, just like the boundary conditions for the metal rod were. A period “.” simply means that the temperature of that cell is not fixed but computed as part of the simulation; it’s probably easiest if you set those cells it to $0^\circ C$ initially.

Conceptually the program you need to write for this problem is very similar to the one we wrote for the metal rod before: Once again you’ll have to set

\(^3\) The size is given as the number of “cells” on each axis; each “cell” is comparable to a “segment” of the metal rod from the previous problem.
We have written some code for matrices before, and if you have it in your notes, you can put it to good use here. You’ll definitely need a function to create a new matrix of a given size as it would be tedious to repeat the code for that every time you need a new matrix. A function like

\[
\text{new\_matrix}(\text{rows, columns, value})
\]

that returns a new matrix with the given number of rows and columns—and with each cell initialized to the given value—seems about right. When you pass a matrix to another function, that function will also have to be able to find out how many rows and columns the matrix has. So two more functions like

\[
\text{rows}(\text{matrix})
\]
\[
\text{columns}(\text{matrix})
\]

that return the number of rows or columns of the given matrix are probably a good idea as well. Consider these examples:

\[
\begin{align*}
\text{>>> new\_matrix}(2, 2, 0.0) \\
\quad &\begin{bmatrix} 0.0, 0.0 \end{bmatrix}, \begin{bmatrix} 0.0, 0.0 \end{bmatrix} \\
\text{>>> new\_matrix}(3, 1, \text{False}) \\
\quad &\begin{bmatrix} \text{False}, \text{False}, \text{False} \end{bmatrix} \\
\text{>>> rows}([\begin{bmatrix} 1 \end{bmatrix}], [\begin{bmatrix} 2 \end{bmatrix}]) \\
\quad 2 \\
\text{>>> columns}([\begin{bmatrix} 1 \end{bmatrix}], [\begin{bmatrix} 2 \end{bmatrix}]) \\
\quad 1
\end{align*}
\]

You can reuse \text{floats\_close\_enough}(a, b) from the previous problem, but you’ll have to write a new \text{matrices\_close\_enough}(a, b) function for this one.

One matrix for the temperature distribution is not enough however. You’ll also have to somehow keep track of which positions in your matrix are fixed by boundary conditions and which positions are free in the sense that you need to compute their temperature. We suggest that you use two matrices. The first one is a matrix of floating point values and contains the temperatures in each “cell” of the metal plate we’re simulating; this includes the temperatures for boundary conditions as well. The second one is a matrix

up a representation for the temperature distribution, once again you’ll have to average that distribution repeatedly until you reach a “steady state,” and once again you’ll plot the final temperature distribution. Nevertheless, the program for metal plates is significantly more complicated, so thinking things through ahead of time is a very good idea.

Let’s talk about the representation of the temperature distribution first. Obviously you cannot use a one-dimensional representation anymore, instead you’ll have to use a matrix—i.e. a list of nested lists—that you can index with a row and a column.
of boolean values, True and False; an entry in this matrix is True if the corresponding position is a boundary condition and therefore fixed; an entry in this matrix is False if the corresponding position is one you have to compute.

With this in mind, here is a possible main program that illustrates how closely related the “big picture” of the metal rod and the metal plate are:

```python
def main():
    current, bounds = read_config("plate←.dat")

    next = average(current, bounds)
    while not matrices_close_enough(next←, current):
        current = next
        next = average(current, bounds)

    P.imshow(current, origin="upper")
    P.axis("off")
    P.tight_layout()
    P.colorbar(ticks=range(0, 126, 25)).←
    set_label("temperature")
    P.show()
```

We’re plotting the results differently now because we have to display two-dimensional data using `imshow`; the `axis` and `tight_layout` functions simply reduce the “wasted space” around the image, and the `colorbar` function creates the bar on the right side that explains which color corresponds to which temperature.

The new `average` function needs both the temperature distribution `current` and the matrix `bounds` telling it which cells are “not to be touched” as it were. So when you’re about to compute the average for a given cell, you check first if that cell is a boundary condition; if so, you skip it; otherwise you actually compute its new average. Computing the average for a given cell requires that you get the temperatures of the 2–4 neighboring cells; this is more complicated than in the case of the metal rod, and you probably don’t want to do it in the `average` function itself; instead you should write a helper function `neighbors` that returns a list with 2–4 temperatures depending on the position you pass in (think about “cells” at the “edge” of the plate).

What’s left is the `read_config` function which has to read the data file and create both of the matrices. After opening the file you to read a single line first to find out how big you need to make your matrices and to know how many more lines to expect:

```python
data = open(name)
line = data.readline()
size = int(line)
...
```

After that step you just read line-by-line as we’ve always done it; for each line, you have to go through the characters on that line and initialize that particular row of your matrices as appropriate. And done!

## 3 Burning Processors (30%)

Deep inside your computer is the central processing unit (CPU), the part that actually makes things happen. From a heat transfer perspective, the CPU is a plate with a heat source in the center. Yes, the “silicon circuitry” of the CPU actually gets hot while the thing is solving your heat transfer problems. Since we don’t want CPUs to catch on fire all the time, we put “heatsinks” on top of them: pieces of metal that are supposed to conduct heat away from the CPU, thereby cooling it. For this to work, however, the heatsink has to make good thermal contact with the CPU itself; if it doesn’t you can still end up with an unexpected BBQ inside your laptop.

Figure 8 shows what can happen for different levels of thermal contact: In the top-left simulation the heatsink had 100% contact, so everything except for the center is cooled to a comfortable 25°C. In the top-right simulation the heatsink had only 10% contact and as you can see the area where thermal contact is particularly bad is getting warmer than it should. In the bottom simulations the heatsink had only 1% contact and things are spinning out of con-
Your program should generate a plate of size 32-by-32 with a big heat source of size 16-by-16 in the center; please put these values at the beginning of your program in the following way so that they can be easily changed to generate plates of different sizes:

```python
SIZE = 32
CORE = 16
CONTACT = 0.1
```

The CONTACT value describes what percentage $p$ of the heatsink should make good thermal contact with the CPU. If you’re creating a plate of size $s$ with a CPU core of size $c$ then there should be $\lfloor (s^2 - c^2) \times p \rfloor$ random points outside the core at which the temperature is fixed to $25^\circ C$ by a boundary condition.

Overall, your program should first generate a matrix of the appropriate size, then fill in the H characters for the heat source correctly, then fill in the C characters for the contact points of the heat sink at random, and finally write the matrix to the output file in the correct format for plate.py to read.

After you have your heatsink.py program, run some simulations for various contact levels and various sizes of the CPU core at the center. Describe your observations in your README file. Are you getting similar results to the ones in Figure 8? If not why not? And why do you think is our simulation so very inaccurate? In the real world a heatsink has to make much better contact than here to keep a CPU from frying...

4. We told you before that we’re not being particularly accurate here. In the real world, a heatsink that makes only 10% contact would certainly set your CPU on fire. (At least if the temperature sensor protecting the CPU is broken.)