600.112: Intro Programming for Scientists and Engineers

Assignment 6: Heat Transfer*†

Peter H. Fröhlich
phf@cs.jhu.edu

Joanne Selinski
joanne@cs.jhu.edu

Due Dates: 3pm Wednesdays 10/21 & 10/28

Introduction

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

There are three required things to do: First you'll write a program to solve a simple one-dimensional heat transfer problem for a metal rod (`rod.py`, 10 points). Second you'll write a program to solve a more complex two-dimensional heat transfer problem for a metal plate (`plate.py`, 22 points). Third you'll write a faster version of the two-dimensional heat transfer problem (`plateFast.py`, 18 points).

We'll continue working with vectors and matrices in this assignment - exploring cell level manipulations and also whole array manipulations. We'll also introduce two new Python libraries! The first is for plotting images: matplotlib. This library has many features and functions, including a set common to MATLAB. The matplotlib webpage contains many examples as well as links to several tutorials if you're interested in learning more. The second library we'll use is the NUMPY library for scientific computations. This will allow us to speed-up the matrix computations for the heat plate problem by moving many computations from PYTHON down closer to the *machine* itself. Make sure that you have installed the necessary packages (PYTHON-MATPLOTLIB and NUMPY) through the synaptic package manager before beginning.

You must submit complete zip files with all *.py and *.txt files needed to run your solutions as detailed below, and on Blackboard before the deadlines. Also, don't forget to include descriptive docstrings and doctests for your functions, and make sure that your programs are all pep8 compliant with regard to style.

Background

What is heat? It's probably in bad taste to start with a joke about the miserably humid $100°F$ Baltimore summer or refer to a popular 2009 tune by Nelly, 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 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

*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!

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 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 $(25°C + 75°C) \div 2 = 50°C$.

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 Δx 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:

$$\left.\frac{\partial T}{\partial x}\right|_{m-1/2} \approx \frac{T_m - T_{m-1}}{\Delta x}$$

$$\left.\frac{\partial T}{\partial x}\right|_{m+1/2} \approx \frac{T_{m+1} - T_m}{\Delta x}$$

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

$$\begin{aligned}
\left.\frac{\partial^2 T}{\partial x^2}\right|_m &\approx \frac{\left.\frac{\partial T}{\partial x}\right|_{m+1/2} - \left.\frac{\partial T}{\partial x}\right|_{m-1/2}}{\Delta x} \\
&\approx \frac{\frac{T_{m+1} - T_m}{\Delta x} - \frac{T_m - T_{m-1}}{\Delta x}}{\Delta x} \\
&\approx \frac{T_{m+1} - T_m - T_m + T_{m-1}}{\Delta x^2} \\
&\approx \frac{T_{m+1} - 2T_m + T_{m-1}}{\Delta x^2}
\end{aligned}$$

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:

$$T_m = \frac{1}{2}\left(T_{m+1} + T_{m-1}\right)$$

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

$$T_{m,n} = \frac{1}{4}\left(T_{m-1,n} + T_{m+1,n} + T_{m,n-1} + T_{m,n+1}\right)$$

There's a lot to say about the limitations of finite difference equations, but we won't worry about that here. Also note that a full understanding of this math background is not necessary for completing the assignment!

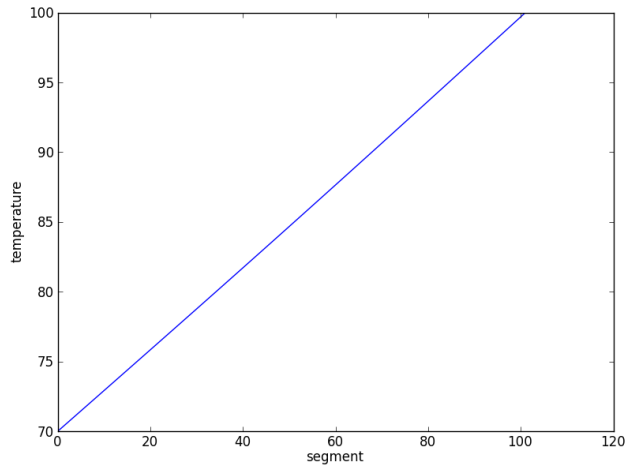
1 Metal Rod [10 points]

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 $70°F$ inside the building to $100°F$ outside the building.

We need to decide two things to solve this problem in Python: First, how should we represent

Figure 1 Correct temperatures along the rod.



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.[1] If we divide the rod itself into 100 equal-sized segments, the total length of our sequence is 102 elements including the fixed endpoints. So we can represent the rod with a vector of floats. 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 using pyplot from the matplotlib library:

```
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 `mathplotlib.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. 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. So we write a function that "runs through" the current rod and computes a new temperature distribution by averaging the relevant elements of the old temperature distribution:

```
def distribute(old):
    new = old[:]  # makes a copy
    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. Clearly if we only call the distribute function once, only the temperatures at the ends of the rod will 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.

But when do we stop the computation? After distributing 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 "equivalent" as far as our simulation is concerned. Let's write a function that performs this "approximate equality check" given a bound ϵ (epsilon); we'll use the *absolute* error version here (for other applications the *relative* error version may be more appropriate):

1. In the language of partial differential equations, these two temperatures are our *boundary conditions*.

```
def floats_equiv(a, b, epsilon):
    '''
    Return True if two floating point
    values a and b are within some
    epsilon difference, False otherwise.
    '''
    return abs(a - b) <= epsilon
```

So we consider two floats "equivalent" if they don't differ by more than ϵ. **Note that you must add doctests to this function!**

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

```
def lists_equiv(a, b, epsilon):
    assert len(a) == len(b)
    for i in range(len(a)):
        if not floats_equiv(a[i], b[i], epsilon):
            return False
    return True
```

Note how we extended the definition of "equivalent" for two floating point values to a sequence: two sequences are "equivalent" if all their corresponding values are "equivalent." Make sure you add a docstring with doctests to this function also.

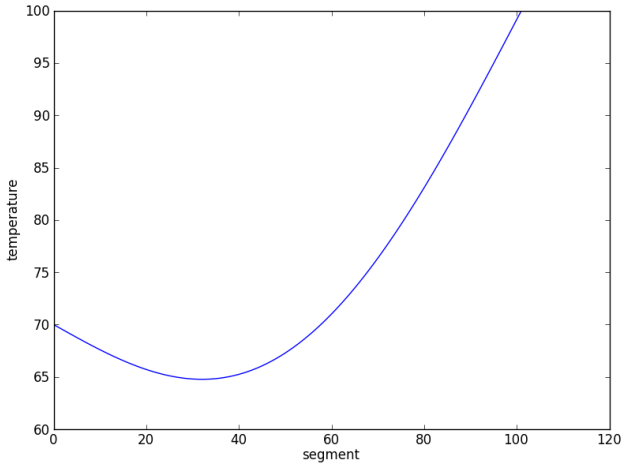
We can now rewrite the main program to perform the appropriate number of averaging steps:

```
def main():
    current = [0.0] * (LENGTH + 2)
    current[0] = 70.0
    current[-1] = 100.0

    after = distribute(current)
    while not lists_equiv(after, current, EPSILON):
        current = after  # this is not a copy, but a relabelling
        after = distribute(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 ϵ should be for this simulation! Play with different values for ϵ (for example $0.1, 0.01, 0.001, \ldots$) and find one that actually produces the correct result from Figure 1. For reference, Figure 2 shows what the *wrong* output for an ϵ that's *too large* looks like.

Figure 2 Incorrect temperatures, ϵ too large.



Challenge: Imagine that the rod touches a pipe with a temperature of $40°F$ at a single segment in the middle of the brick wall. How hard is it to modify our existing program to compute the correct temperature distribution for this new problem? Can you think of a way to reorganize the program that would have made this change easier? Can you think of a way to generalize the program to the point where **any number of boundary conditions** for a one-dimensional heat transfer problem could be handled by the same exact program? You can either discuss these issues in a README file (make sure to include it in your zip), or write and submit a more general version of the solution called `rod2.py`. Just make sure that no matter what, you submit the original `rod.py` solution without enhancements.

2 Metal Plate [22 points]

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 $25°C$ when we apply a single heat source of $100°C$: Eventually the *entire* plate will have a temperature of $100°C$. Figure 4 shows a slightly more interesting scenario: If the plate is surrounded by a cooling agent that keeps the edges at $25°C$, then only a relatively small part of the plate will actually get warm. Figures 5 and 6 illustrate how heat

Figure 3 One heat source, no boundary conditions.

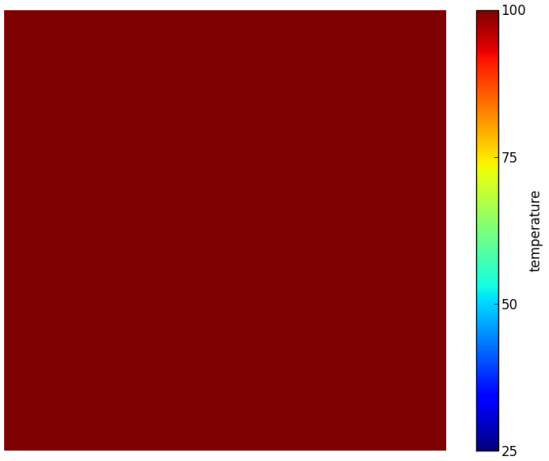
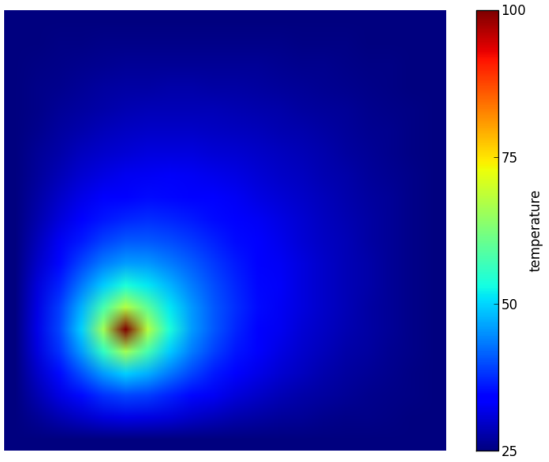


Figure 4 One heat source, all boundaries cooled.



from our heat source spreads through the plate if two boundaries or only one boundary are cooled to $25°C$.

Figure 5 One heat source, two boundaries cooled.

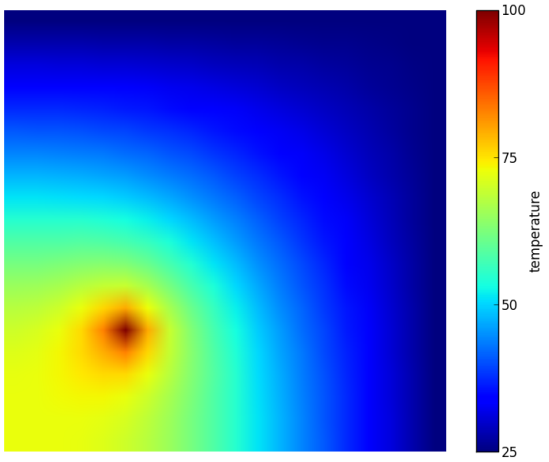


Figure 6 One heat source, one boundary cooled.

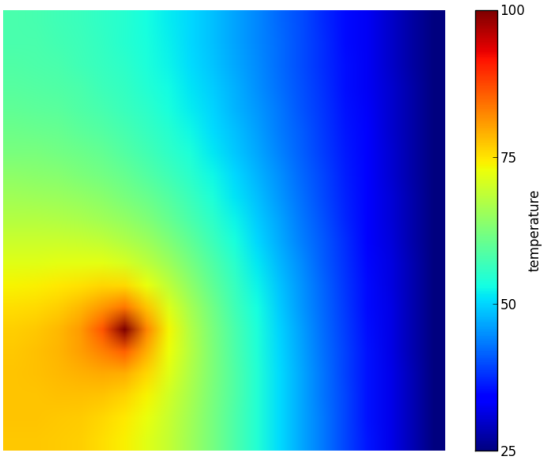


Figure 7 Sample input to the `plate.py` program.

```
10
CCCCCCCCCC
.........C
.........C
.........C
.........C
.........C
.........C
..H......C
.........C
.........C
```

For this problem you will write a program that computes the "steady state" temperature distribution of a square 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.txt` 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.[2] 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 $25°C$ whereas a capital "`H`" stands for a "heat source" which has a temperature of $100°C$. 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 pe-

2. 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.

riod "." 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 to $0°C$ 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 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*. We have written some code for matrices before, and if you have it in your notes or download it from the course schedule, you can put it to good use here. Your choices for using it are to cut and paste the necessary functions into this program file, or to put our matrices.py module in the same folder as these programs and import it into your plate.py program. If you do it the second way, make sure that you include matrices.py in your submission zip!

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

```
new_matrix(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

```
numRows(matrix)
numCols(matrix)
```

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

```
>>> new_matrix(2, 2, 0.0)
[[0.0, 0.0], [0.0, 0.0]]
>>> new_matrix(3, 1, False)
[[False], [False], [False]]
>>> numRows([[1], [2]])
2
>>> numCols([[1], [2]])
1
```

You can reuse `floats_equiv(a, b)` from the previous problem, but you'll have to write a new `matrices_equiv(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 (called `current` in the sample listing below) 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 (called `fixed` in the sample listing below) is a matrix of boolean values, `True` and `False`; an entry in this matrix is `True` if it's temperature is specified by a boundary condition and therefore fixed; an entry in this matrix is `False` if the corresponding temperature cell 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:

```
def main():
    current, fixed = read_config("↩
        plate.txt")

    after = distribute(current, fixed)
    while not matrices_equiv(after, ↩
        current):
        current = after
        after = distribute(current, ↩
            fixed)

    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 `distribute` function needs both the temperature distribution `current` and the matrix `fixed` 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 (north, south, east and west); this is more complicated than in the case of the metal rod, and you *probably* don't want to do it in the `distribute` 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 should 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:

```
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 individual elements of your matrices as appropriate. Add any other helpful functions you may think of to simplify the coding in this one. Note that the way we've written and used this function, it returns a *tuple* of both matrices!

Lastly, update main() so that it prompts the user for the name of the input file. That way you can easily run your program with many different plate configurations. We've created a starter file for you to use, `plateStart.py`, which has the skeleton of each method described here, along with some documentation and a few of the harder new statements as well. Don't forget docstrings, doctests and pep8 style for all parts of your finished program!

3 NumPy Heat Transfer [18 points]

For this problem you'll re-write the program for the heat-plate simulation, but this time using the NUMPY library for scientific computations. That is, your new program should produce the *exact* same results as the previous program (as far as possible with floating point numbers), but it should be *faster* because NUMPY allows you to move many computations from PYTHON down closer to the *machine* itself. Please call your program `plateFast.py`, nothing else.

The last step of the program to draw the resulting steady-state heat distribution using `matplotlib` will be exactly the same in this version of the program. The part of the program that sets up the temperature and fixed matrices from an input file will be enhanced to create a third matrix necessary to perform the revised computation as described below. Also, the function to determine if two matrices are equivalent should be rewritten to take advantage of NUMPY's fast operations.

However, the main difference will be in the part of the program that repeatedly *averages* the temperatures around each cell to compute the new temperature of a cell using something like this equation:

$$T_{i,j} = \frac{1}{4}\left(T_{i-1,j} + T_{i+1,j} + T_{i,j-1} + T_{i,j+1}\right)$$

At the edges and corners of the plate where we have only 2 or 3 neighbors instead of 4, the formula is adjusted accordingly. This is the part we'll tackle rewriting first.

This part of the program takes the most time: We repeatedly "run across" the rows and columns of the plate to produce new values for all computed cells, and all of the computation is carried out "cell-by-cell." So only the actual arithmetic is performed directly by the machine (fast on average), most of the remaining code (loops, access of matrix elements) is performed by PYTHON (slow on average). Therefore this is the part where we can improve performance the most if we are able to express our solution in terms of *matrix operations* provided by NUMPY: each of those operations is fast (on average) and deals with *all* cells of the matrix on the machine level without involving PYTHON again.

We need to rethink the way we perform averaging to get away from the idea of doing it "cell-by-

cell", a relatively slow sequential approach. Instead of dealing with a single cell and it's neighbors, we want to deal with all cells at the same time, a faster parallel processing approach. Obviously we have to still worry about the edges and corners of the plate where the averaging computation needs to proceed differently than in the center, but let's focus on the center first.

If you consider a cell (i, j) for which we want to compute a new temperature, we first need to sum up the temperatures of the cells around it: $(i-1, j), (i+1, j), (i, j-1), (i, j+1)$. The key insight is to realize that instead of looking at the left $(i, j-1)$ neighbor, we could simply "shift the matrix" one column to the right and use the cell (i, j) *of the shifted matrix*. Of course the same is true for the other neighbors: Each of which can equally well be found at position (i, j) in some appropriately shifted matrix. As soon as we "line up" the elements correctly, we can use NUMPY's elementwise whole matrix addition operation to express the averaging step as follows:

$$\boxed{\mathrm{N}} = \frac{1}{4}\left(\uparrow \boxed{\mathrm{O}} + \downarrow \boxed{\mathrm{O}} + \leftarrow \boxed{\mathrm{O}} + \rightarrow \boxed{\mathrm{O}}\right)$$

Here $\boxed{\mathrm{N}}$ is the "new" matrix of (averaged) temperatures while $\boxed{\mathrm{O}}$ is the "old" matrix of temperatures. Obviously this is not yet correct for edges and corners: First we should only add 2 or 3 cells there instead of 4, second we should divide by 2 or 3 instead of 4 to get the correct average.

Let's focus on how we would shift the matrix first; luckily NUMPY provides a function `roll` that works as demonstrated in Figure 8: We can "shift" a matrix by a certain number of elements along a certain axis; for example, `numpy.roll(a, 1, 0)` shifts all rows *down* by one. In general, for a multi-dimensional array, the roll function takes an array as the first parameter, the shift amount as the second parameter, and the dimension or axis along which to shift as the third parameter. In the first example, note that the row "shifted out" at the bottom re-enters the matrix at the top.

That last aspect is what makes `roll` not quite suitable for us: The only way to "ignore" a neighbor when averaging (as we need to along the edges) is to fill in a row or column of zeros instead! The functions you need to shift the temperature matrix for averaging will have to do that, but we can still use `roll` to do the expensive part of moving matrix elements around. Once we roll, we need to zero out the new row or column, and we can use NUMPY's convenient array slicing operations for that. Here is one of the resulting functions - it's up to you to write the other three.

Figure 8 The `numpy.roll` function in action.

```
>>> a
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
>>> numpy.roll(a, 1, 0)
array([[7, 8, 9],
       [1, 2, 3],
       [4, 5, 6]])
>>> numpy.roll(a, -1, 0)
array([[4, 5, 6],
       [7, 8, 9],
       [1, 2, 3]])
>>> numpy.roll(a, 1, 1)
array([[3, 1, 2],
       [6, 4, 5],
       [9, 7, 8]])
>>> numpy.roll(a, -1, 1)
array([[2, 3, 1],
       [5, 6, 4],
       [8, 9, 7]])
```

```
def up(matrix):
    """Move matrix up one row."""
    new = N.roll(matrix, -1, 0)
    new[-1, :] = 0
    return new

def down(matrix):
    """Move matrix down one row."""
    pass

def left(matrix):
    """Move matrix left one column."""
    pass

def right(matrix):
    """Move matrix right one column.
    """
    pass
```

Using these functions, we can shift our matrix appropriately and add the shifted matrices together to compute the sums we need in each new cell: for cells in the center of the matrix, the four shifted matrices will have the correct neighbors at position (i, j); for cells on the edges of the matrix, three of the four shifted matrices will have the correct neighbors while one will have a zero; for cells in the corners of the matrix, two of the four shifted matrices will have the correct neighbors while two

Figure 9 The `numpy.where` function in action.

```
>>> a
array([[False,  True,  True],
       [ True, False,  True],
       [ True,  True, False]], dtype=bool)
>>> b
array([[ 4.,  4.,  4.],
       [ 4.,  4.,  4.],
       [ 4.,  4.,  4.]])
>>> c
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
>>> where(a, b, c)
array([[ 0.,  4.,  4.],
       [ 4.,  0.,  4.],
       [ 4.,  4.,  0.]])
```

will have zeros. However, to compute the averages we still have to divide by the appropriate number of neighbors. Luckily NUMPY allows us to perform element-wise division of a matrix, so if we set up a corresponding divisors matrix of the same size as the temperature matrics, with values 2 in the corners, 3 on the edges, and 4 everywhere else, we can divide the matrix of sums by the divisors matrix to compute the proper averages. Building this matrix is pretty straightfoward, using the functions we've already built:

```
def make_divisor(size):
    base = N.ones((size, size))
    div = up(base) + down(base) + left↩
        (base) + right(base)
    return div
```

Note that you don't want to use the old cell-by-cell method of initializing a matrix so that you have the most speed-up possible.

There's one more thing to take care of though: Our approach of averaging the *entire* matrix also computes results for positions that should *not* be changed because they are boundary conditions! In other words, once we are done computing the averages, we want to write only those into the result matrix that were not already fixed in the initial problem. This is where NUMPY's `where` function comes in handy, see Figure 9: We can select between the elements of two matrices based on a third `bool` matrix; if, for example, all elements of `cond` are `True`, then `numpy.where(cond, a, b)` is just going to be the matrix `a`; if they are all false, it's going to be the matrix `b` instead.

In the solution to part two you should have already used such a matrix to keep track of where the boundary conditions are, but again you had to process it one element at a time; now we can use NUMPY to distinguish between boundary conditions and computed cells in one go. So provided we have three matrices, one with the current temperatures (old), one with the boundary conditions marked by `True` (clamped), and one with the appropriate divisors (divs), our averaging step now reads as follows:

```
def average(old, clamped, divs):
    sums = up(old) + down(old) + left(↩
        old) + right(old)
    averages = sums / divs
    new = N.where(clamped, old, ↩
        averages)
    return new
```

That was simple, wasn't it? No more neighbors to find, no more values to add up and average, it's all taken care of by NUMPY. The rest of the program is mostly concerned with input and output just like before. However, you must use the NUMPY zeros function to create the initial matrices. A NUMPY matrix is not the same data type as a nested list - so all our fast operations and revised functions will only work if we consistently use the same data type throughout the program. Lastly, we will also compare matrices using NUMPY of course. This can be done very compactly with matrix subtraction and the min and max functions, as shown in the listing below.

```
import matplotlib.pyplot as P
import numpy as N

EPSILON = 0.0001
HOT = 100.0
COLD = 25.0

...

def matrices_close_enough(a, b):
    d = a - b
    return max(abs(d.min()), abs(d.max↩
        ())) <= EPSILON
```

Let's not forget the reason we did all this: so our program would run faster! You'll need to use a pretty large input file in order to see a noticeable difference between the original plate.py solution and the new plateFast.py solution. For an extra challenge, write a program to randomly create arbitrarily large input files that simulate CPU cooling.

Submission Summary: Remember to include

all the following files in your final submission zip: program files rod.py, plate.py, plateFast.py and input file plate.txt for testing the plate programs. Lastly, include our lecture matrix module matrices.py if you import it to use in any of your programs, as well as any other modules you create and import to avoid cutting and pasting of functions into multiple programs. Also remember to make all your .py files pep8 compliant, and include docstrings and doctests for all functions (except where random data is used since the outcome can't be predicted).