Golfing the Method of Relaxation in Electrostatics
and
Experimental Verification That Numpy's Vector Operations are Badass

Patrick D. Cook

Department of Physics, Fort Hays State University,
Hays, Kansas 67601, USA

Abstract

In this work, I present an overview of the method of relaxation for two-dimensional electrostatics. This method solves Laplace's equation with arbitrary boundary conditions. An implementation of the method in Python 3 is presented with results. The problem is then "golfed" in which the shortest possible Python 3 program which implements the method is presented, again with results.

Keywords: Electrostatics, Laplace's equation, Code Golf

```
from numpy import*
from matplotlib.pyplot import*
C=30
O=10
o=10
K=roll
P=zeros((C,C))
exec('P=(K(P,1,0)+K(P,-1,0)+K(P,1,1)+K(P,-1,1))/4;P[o:-o,o:o-~O:O]=1,-1;'*9*C)
D=gradient(-P)
imshow(P)
quiver(D[1],-D[0])
show()
```

1 Basics

1.1 Introduction

The equation describing the electric potential, V, in the absence of net electric charge is Laplace's equation [1],

$$\boldsymbol{\nabla}^2 V = 0. \tag{1}$$

Carl Friedrich Gauss, of Gauss's law, proved that any function which is harmonic within some sphere—meaning that it satisfies Eq. 1 in that sphere—has the property that its value at the center of the sphere is the arithmetic mean of its value on the surface of that sphere [2]. Since the electric potential must satisfy Eq. 1 everywhere, assuming there is no net charge anywhere,

$$V(\vec{r}) = \frac{1}{4\pi R^2} \oiint V da, \ \forall \vec{r}. \tag{2}$$

Where the integral is taken over the entire sphere of radius R centered at $\vec{r}$. This generalizes to lower dimensions,

$$\textbf{3D} \quad V(\vec{r}) = \frac{1}{4\pi R^2} \oiint_{Sphere} V da, \ \forall \vec{r} \tag{3}$$

$$\textbf{2D} \quad V(\vec{r}) = \frac{1}{2\pi R} \oint_{Circle} V dl, \ \forall \vec{r} \tag{4}$$

$$\textbf{1D} \quad V(r) = \frac{V(r+R) + V(r-R)}{2}, \ \forall r. \tag{5}$$

This is the mathematical foundation for the method of relaxation. If the boundary conditions are known, the above equations suggest that successive averaging will converge to the solution everywhere in space. Since solutions to Eq. 1 are necessarily unique, this will produce the only solution.

Also of importance is the electric field, which can be found from the electric potential with [1]

$$\vec{E} = -\vec{\boldsymbol{\nabla}} V \tag{6}$$

1.2 Method of Relaxation in Two Dimensions

Suppose, instead of a continuous distribution, V is instead a discrete function. Denoting $V(x_i, y_j)$ as $V_{i,j}$, Eq. 4 becomes [1]

$$V_{i,j} = \frac{V_{i+1,j} + V_{i-1,j} + V_{i,j+1} + V_{i,j-1}}{5} + \frac{V_{i+1,j+1} + V_{i-1,j-1} + V_{i+1,j-1} + V_{i-1,j+1}}{20}. \tag{7}$$

This can be seen as the weighted average of the eight neighboring points around $V_{i,j}$. The factors of $\frac{1}{5}$ and $\frac{1}{20}$ are weightings which are necessary due to the fact that the corners are farther from $V_{i,j}$ than the edges. While this equation is simple, it is only true when V is the solution to Eq. 1. However, it can be used to find this solution. Let V^n denote the n^{th} trial function of V. The initial trial function, V^0, will consist only of the known boundary conditions and be zero everywhere else. Given V^n, the next trial function is given by a modified version of Eq. 7,

$$V^{n+1}_{i,j} = \frac{V^n_{i+1,j} + V^n_{i-1,j} + V^n_{i,j+1} + V^n_{i,j-1}}{5} + \frac{V^n_{i+1,j+1} + V^n_{i-1,j-1} + V^n_{i+1,j-1} + V^n_{i-1,j+1}}{20}. \tag{8}$$

The trial functions will approach the true solution, V, as $n \to \infty$.

1.3 Implementation

Appendix A shows the implementation of the 2D method of relaxation in Python 3. The code solves for the electric potential, V, and electric field, $\vec{E}$, in and around a finite parallel plate capacitor.

Part of the implementation that is worth pointing out is the cyclic spacial boundary conditions. By using Numpy's `roll` function, the boundaries wrapped. For example $V_{-1,j} = V_{N,j}$ and $V_{N+1,j} = V_{0,j}$ where N is the final index in the x direction. This can be both advantageous and detrimental, so it is important to keep in mind.

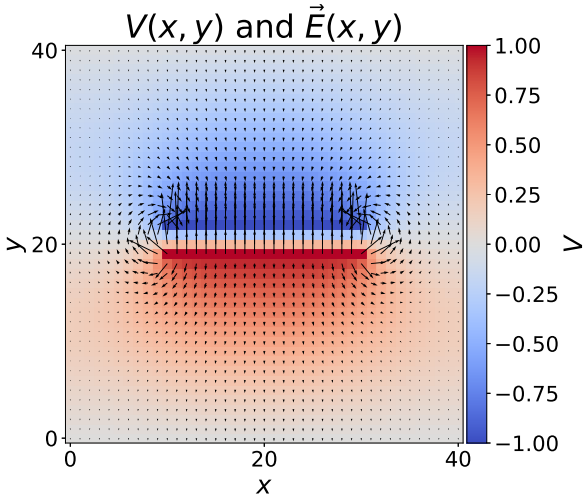

Figure 1: Capacitor with large plate area and small separation distance.

1.4 Results and Discussion

Three different parallel plate capacitors were investigated using this code: one with large area and small separation distance, one with small area and small separation distance, and one with small area and large separation distance. In all cases the top plate was fixed at -1 V while the bottom plate was fixed at $+1$ V. In the model, distances were not assigned to each grid point so the results are reported in arbitrary units. The results are shown below.

Figure 1 shows the first case, large area and small separation distance. This case exhibited the characteristic strong electric field between the plates and weak electric field outside. Fringe fields are present but small.

In the second case, with small plate area and small separation distance, was very different. Figure 2 demonstrates that the field outside the capacitor was no longer weak and that the fridge field dominated overall. There is still a stronger field inside than outside, but the magnitudes comparable.

Finally, the third case: small plate area and large separation distance. As expected, this capacitor behaves like two monopoles, as demonstrated in Fig. 3.

All results are qualitatively consistent with theory. Convergence testing was not performed, neither between iterations nor to analytic solutions, due to lack of interest.

2 Code Golf

I hope you didn't think this whole paper was going to be as boring as the last two pages. Don't worry, this isn't baby's first computa-

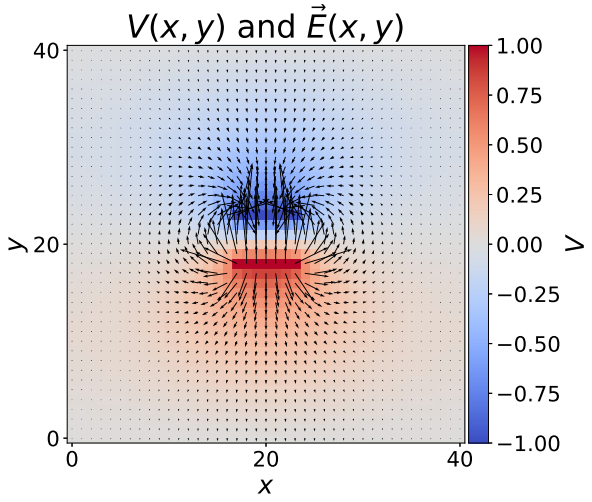

Figure 2: Capacitor with small plate area and small separation distance.

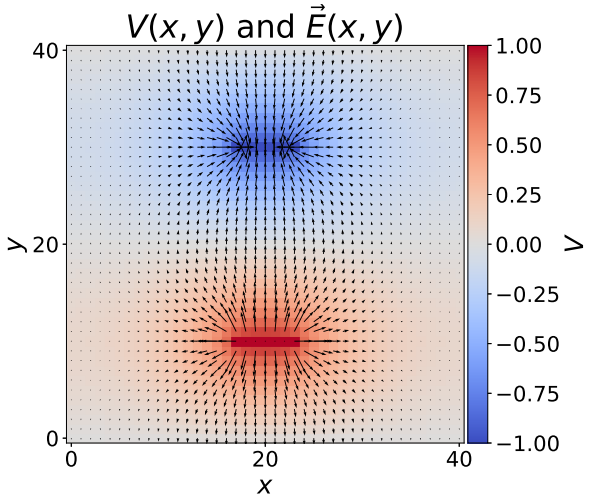

Figure 3: Capacitor with small plate area and large separation distance.

tional physics project. Things are about to get a whole lot more interesting. And obfuscated.

2.1 What in the World is Code Golf?

Code golf refers to the pastime, sport, hobby, or whatever of trying to program a task in as few characters—or bytes—as possible. There are special languages made just for this, called golf languages. These languages are often next-to-impossible to read if you're new to them. For example, here's a program that prints the first 1000 digits of Pi, written in GolfScript:

```
;''
6666,-2%{2+.2/@*\/10.3??2*+}*
`1000<~\;
```

And here's "Hello World!" in Brainfuck:

```
+[-[<<[+[---->]-[<<<]]]>>>-]>-.---.>..>.<<<<-.<+.>>>>>.>.<<.<-.
```

Needless to say, these aren't always useful, and should never be used in read-world applications—but they're great for golf!

Luckily, you don't need one of these fancy-schmancy languages to golf. Every language has golf potential. There are tips and tricks you could learn to golf in your favorite language, as long as it's not Java. Just please don't use those tips and tricks in real programs.

2.2 Golfing in Python

You could write a book on golfing in Python. Don't. In order to prepare you for the discussion to come, I will provide an overview of just a few neat golfing tricks in Python 3.

Here are some obvious golf tricks that are applicable not just in Python, but pretty much any language: remove all unnecessary spaces, use one letter variable names, don't worry about execution speed or efficiency. Okay now into the real stuff.

So starting at the top of a Python file, you'll probably have some `import` statements. Using these are kind of cheating, since you're using potentially hundreds of thousands of lines almost for free. However, I don't think anyone is going to expect you to golf a plotting engine just to plot the solution to your golfed problem, so they're alright in certain cases—unless of course the challenge was to golf a renderer. When importing Python modules, we can save a few bytes by using `from module import*` instead of `import module as m` or `from module import *`, since the space between `import` and `*` isn't necessary, and we don't want to have to type `m.` in front of every function call from that module. Of course, that means default functions may be overwritten, so you'll have to be carefull of that.

In the same realm of shortening function calls is Python's neat ability to assign new names to functions without needing to call them. For instance, say you're using `sqrt` a lot, then consider renaming the function to `s` with:

```
s=sqrt
```

Now, instead of `sqrt(2)`, you can write `s(2)`! This only saves bytes if you're using the function more than once. It also saves more bytes the longer the function name is.

`for` loops are messy, require indentation, and usually use `range()`, all of which take up precious bytes. If you don't care about the incremented value in the `for` loop, you can use this extremely slow, but very compact, trick:

```
for i in range(N):
    f(x)
```

becomes

```
exec("f(x);"*N)
```

Let's look at exactly what's going on here. First, lets look at the argument to `exec()`:

```
"f(x);"*N
```

This will evaluate to `"f(x);f(x);f(x);...;f(x)"` where there are N `f(x)`'s. Now, since this is the argument to `exec()`—which runs whatever string it was given as if it were code—it runs `f(x)` N times, just like a `for` loop! This saves numerous bytes in many circumstances. Of course, we can also perform standard string methods on the argument to `exec` before it runs that code, like `.replace()` or formatting with `"%s"`. These are both common tricks that are also used.

There are a few special case arithmetic operations that are important. For example, `N//M` is the same as `int(N/M)`. This means

that instead of performing floating point division and converting the result to an integer, we can directly perform integer division. Also, the unary operator, ~ becomes useful to us when golfing. All ~ does is reverse the bits of the input. Nevermind what that means, but for integers `~x = -x - 1`. So we can add one and subtract one from a number by pairing ~ with -:

```
-~N # same as N+1
~-N # same as N-1
```

Now, if you're *really* good at counting, you've noticed that those expressions, (`-~N` and `N+1`) have the same number of characters. So why are these useful? It's because ~ has operator precedence over + and - so instead of

```
(N+1)/2
```

we can save parenthesis and use

```
-~N/2
```

Like I said, there's a million more things to discuss about this. However, this is all we'll need for the upcoming discussion. If you're interested in learning more, just type it into Google yourself, don't come ask me.

2.3 Numpy Arrays and Vector Operations

Numpy is the most important thing to ever come into my life, and the same could be true for you. Learning Numpy is daunting, especially because knowing where to start is hard, but in the end it's more than worth it. I'm not saying this in terms of golfing, I mean in scientific computing, Numpy is a boon.

However, it's also great when golfing. I'm going to cover just two of the many things that make it great at golfing here, since I'll be using them in the next section.

Array indexing and slicing in Numpy is done MATLAB-style, like this:

```
array[start:stop:step]
```

The above code will spit out the array elements, starting at `start`, ending at `stop-1`, and skipping every `step` values. For example, suppose we have the array:

```
array([0, 1, 2, 3, 4, 5, 6, 7])
```

and we index it as `array[1:5:2]`, then we will get:

```
array([1, 3])
```

because we started at the 1^{st} position, and picked every 2^{nd} element, until we got to the $(5-1)^{\text{th}}$ position. Importantly, this scales to higher dimensional arrays. If we have a 2D array, we can index each axis like this, separating them with commas, like `array[1:5:2,1:5:2]`.

Finally, Numpy's `roll` function is powerful. All it does is shift the positions of all the lements in an array by a specified amount, rolling elements at the end back to the beginning. It allows us to write operations which involve relative positions in arrays, like derivatives, in one line. For example, the second-order central finite difference for the second derivative is:

$$\frac{d^2f}{dx^2}(x_i) = f(x_{i+1}) - 2f(x_i) + f(x_{i-1}) \quad (9)$$

In normal Python, this would usually be implemented as:

```
for i in range(1,len(x)-1):
    d2f_dx2[i] = f[i-1]-2*f[i]+f[i+1]
```

But with Numpy this can be done without the `for` loop, instead using `roll`:

```
d2f_dx2 = roll(f,-1) + 2*f + roll(f,1)
```

Again, this applies in higher dimensions, it just becomes necessary to specify the axis of the roll.

2.4 Golfing the Method of Relaxation and Its Implementation

Here are the requirements:

- The code must solve for both V and $\vec{E}$ everywhere around a parallel plate capacitor using the method of relaxation
- The code must be able to solve any configuration of plate area and separation distance, as specified by the user (directly modifying the source is okay)
- Assume all user-specified numbers are two digits in base 10
- Both V and $\vec{E}$ must be graphically displayed
- The number of iterations must be enough so that the results qualitatively converge

There are some tricks that can be applied to the method of relaxation itself before we start strinking our code. This is where the real golfing is done, in cleverly finagling the problem to reduce its size.

One of the easiest reductions here is that we can postpone our setting of the boundary conditions until we are done with the first iteration, since we're going to have to reset them at the end of every iteration anyways. Of course, this means the first iteration performs no useful computations, but we're not going for efficiency here.

Of course, we don't really need a rectangular domain, so I've opted for a square domain instead to save an entire line of variable assignment.

The orientation of the capacitors isn't critical, so they're vertical instead of horizontal now.

We also don't need to specify the potential on each parallel plate, since the results will scale. So we'll just hardcode them to be $+1$ V and -1 V.

Again, we're not concerned with fast convergence anymore, so we can actually reduce Eq. 7 to only average the edges:

$$V_{i,j} = \frac{V_{i+1,j} + V_{i-1,j} + V_{i,j+1} + V_{i,j-1}}{4}. \quad (10)$$

However, this feels like cheating, so I've golfed both versions, a faster-convergence method which uses Eq. 7 and a slow-convergence method which uses the above equation.

We don't want to waste bytes on making our graphs look pretty, so I've opted for whatever `matplotlib`'s defaults are.

Finally, the number of iterations can just be hardcoded to be $9N$, where N is the edge length of the domain. This seems to be enough, and it doesn't require a two-digit number.

Looking at Appendix C, we see the golfed version with slow convergence. The trick of not using the corners makes the line that is iterated so short that it's not worth doing any string replacements. Therefore, it's not too hard to decode this line. Numpy's `roll` function has been renamed `r`. The line with **`exec`** uses the trick discussed earlier to replace a **`for`** loop. However, inside that line is a particularly nasty Numpy trick:

```
V[s:-s,s:s-~d:d]=1,-1
```

All this line is doing is setting the boundary conditions of the capacitors. But how? Well, let's look at the array indexing. The first part, `s:-s`, just says to take every element on the y axis from s to $-s$. The second part, `s:s-~d:d`, says to take every d^{th} element on the x axis, starting at s and ending at $s - d$ (recall `s-~d=s+d+1`). If you math that out, you'll see that this will always give only 2 elements on the x axis, the element at s and $s + d$. These two elements get assigned to `1,-1` corresponding to the positive and negative plates of the capacitor. That's pretty much the only cool trick in that script.

The golfed version with faster convergence, shown in Appendix B, looks a lot worse, but in reality it's just a bunch of string replacement and formatting. The line where all the iterations happen, after all the replacements and formats is just:

```
V=(r(V,1,(0))+r(V,-1,(0)) \
+r(V,1,(1))+r(V,-1,(1)))/5 \
+(r(V,1,(0,1))+r(V,-1,(0,1))+ \
r(V,1,((0,)*~-N+(1,))) \
+r(V,-1,((0,)*~-N+(1,))))/20; \
V[s:-s,s:s-~d:d]=1,-1;
```

(Of course without all the line continuation characters, `'\'`, and linebreaks that I've added for readibility here.) Again, we see the same trick with array indexing for assigning the capacitor voltages, and the same trick with renaming `roll`. However, the last two `roll`s are worth looking at. This function call,

```
r(V,1,((0,)*~-N+(1,)))
```

is equivalent to an anti-diagonal roll, which in the un-golfed code was done as two consecutive rolls. All this line says is to roll 1 over the 0^{th} axis `~-N` times, then roll once over the first axis. This is the same as rolling -1 over the zeroth axis, then rolling 1 over the first axis, but requires one less call to `roll`.

2.5 Golf Results

Figures 4 and 5 show the results from both golfed codes. They're not as pretty as the un-golfed code, and they took way longer to run, but hey, the scripts are less than 300 bytes.

2.6 Author's Note

That's all I've got, hope you enjoyed. You might be wondering what's different about the code included on the title page. The answer is nothing. All I did was rename some

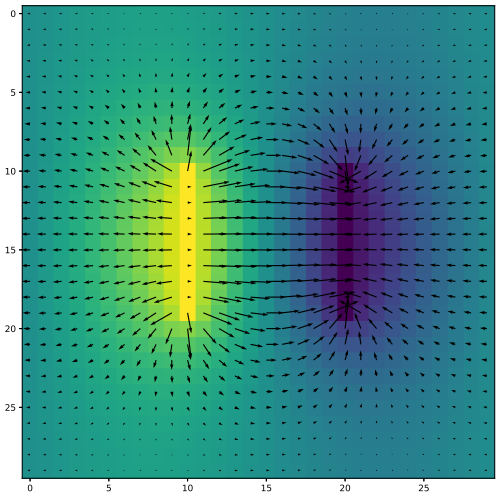

Figure 4: Output from the golfed method of relaxation using the method with faster convergence.

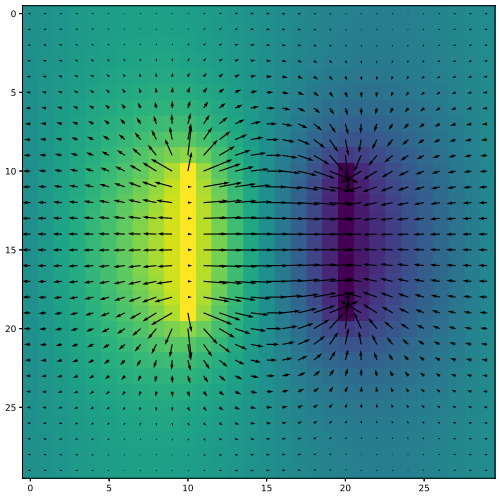

Figure 5: Output from the golfed method of relaxation using the method with slower convergence.

variables so that the first column spelled out my last name and initials: Cook P. D.

References

[1] J. D. Jackson, *Classical Electrodynamics*. Wiley, 1999.

[2] E. W. Weisstein, "Gauss's harmonic function theorem." http://mathworld.wolfram.com/GausssHarmonicFunctionTheorem.html. Accessed: 2019.

A 2D Method of Relaxation Implemented in Python 3

```
import numpy as np                                          # for arrays
import matplotlib.pyplot as plt                             # for plotting
from mpl_toolkits.axes_grid1 import make_axes_locatable # for colorbar on plots

ticks = 3 # number of ticks on x and y axis of plot

N          = 21   # size of domain in y direction
M          = 21   # size of domain in x direction
iterations = 1000 # number of relaxation iterations
distance   = 5    # gridpoints between parallel plates
width      = 5    # distance of each plate endpoint to the spacial boundary
V0         = 1    # absolute potential of each plate

# initalize the domain to zeros
V  = np.zeros((N,M))

# set the potential of both capacitors, one V0 and the other -V0
V[int(N/2-distance/2), width:M-width] = V0
V[int(N/2+distance/2), width:M-width] = -V0

# begin iterating
for _ in range(iterations):
    # Eq. 8
    V = 0.2*(np.roll(V, 1, axis = 0) + np.roll(V, -1, axis = 0) \
        + np.roll(V, 1, axis = 1) + np.roll(V, -1, axis = 1))   \
        + 0.05*(np.roll(V, 1, axis = (0,1)) + np.roll(V, -1, axis = (0,1)) \
        + np.roll(np.roll(V, 1, axis = 1), -1, axis = 0) \
        + np.roll(np.roll(V, -1,axis = 1), 1, axis = 0))

    # reset the boundary conditions (the parallel plates)
    V[int(N/2-distance/2), width:M-width] = V0
    V[int(N/2+distance/2), width:M-width] = -V0

# calculate the electric field with Eq. 6
E = -1*np.array(np.gradient(V))

# plot the results
fig,ax=plt.subplots(1,1,figsize=(8,8)) # make the figure
```

```
im = ax.imshow(V, cmap = plt.get_cmap("coolwarm"), rasterized = True) # plot V

ax.quiver(E[1],E[0]) # plot E

ax.invert_yaxis() # make the y axis point up

ax.set_xlabel("$x$",size=24) # set the x label
ax.set_ylabel("$y$",size=24) # set the y label

ax.set_xticks(np.linspace(0,M-1,ticks)) # set the x ticks
ax.set_yticks(np.linspace(0,N-1,ticks)) # set the y ticks

ax.tick_params(axis='both', which='major', labelsize=20) # change the font size

# make the axis for the colorbar
divider = make_axes_locatable(ax)
cax = divider.append_axes('right', size='5%', pad=0.05)
cbar=fig.colorbar(im, cax=cax)

# add the ticks and the label to the colorbar
cax.tick_params(labelsize=20)
cax.set_ylabel("$V$",size=24)

# set the figure title
ax.set_title("$V(x,y)$",size=30)

# save to file and show on screen
plt.tight_layout()
plt.savefig("V.eps",dpi=100)
plt.show()
```

B Golfed 2D Method of Relaxation with Faster Convergence

```
from numpy import*
from matplotlib.pyplot import*
N=30
d=10
s=10
e=exec
r=roll
V=zeros((N,N))
e(("e(('V=(%s0`0`1`1)))/5+(%s0,1`0,1'+'`(0,)*~-N+(1,)'*2+')))/20').replace('`',','))+%s')%(('r(V,1,(','r(V,-1,(')*4));V[s:-s,s:s-~d:d]=1,-1;"*9*N)
imshow(V)
E=gradient(-V)
quiver(E[1],-E[0])
show()
```

C Golfed 2D Method of Relaxation with Slower Convergence

```
from numpy import*
from matplotlib.pyplot import*
N=30
d=10
s=10
r=roll
V=zeros((N,N))
exec('V=(r(V,1,0)+r(V,-1,0)+r(V,1,1)+r(V,-1,1))/4;V[s:-s,s:s-~d:d]=1,-1;'*9*N)
E=gradient(-V)
imshow(V)
quiver(E[1],-E[0])
show()
```