MATLAB Electric Field Simulation Tutorial

Luca Scharrer

September 2020

This tutorial is intended to teach the reader how to use matlab's PDE toolbox to simulate electric potentials, and the motion of ions in those potentials. The two tools you'll need are matlab with the PDE toolbox installed, and some kind of CAD software. I've been using FreeCAD, but theoretically, and kind of CAD software able to output .stl files should work just as well. I will include a very short tutorial on how to export a .stl file from AutoCAD, but all of the .stl files used in this tutorial will be included in the github, so you don't need to actually go through that part in order to learn to use the matlab simulations, it's more of just a general guide for future design. In order to use the functions I've written, you'll need to download the files from

https://github.com/Dave-Patterson-Group/potential-sims

and unzip them into a folder. You'll need to open that folder up in matlab, making sure that's your working directory. Then, you should be able to use all of the functions I mention below.

1 Exporting a Part as a .stl File

I'm going to assume the reader already knows how to create objects in whatever CAD program they choose. If not, there are plenty of tutorials online for whatever program you like. In this tutorial, I'm going to simulate a parallel plate capacitor inside of a grounded box. So, I create two parallel rectangular prisms in autoCAD:

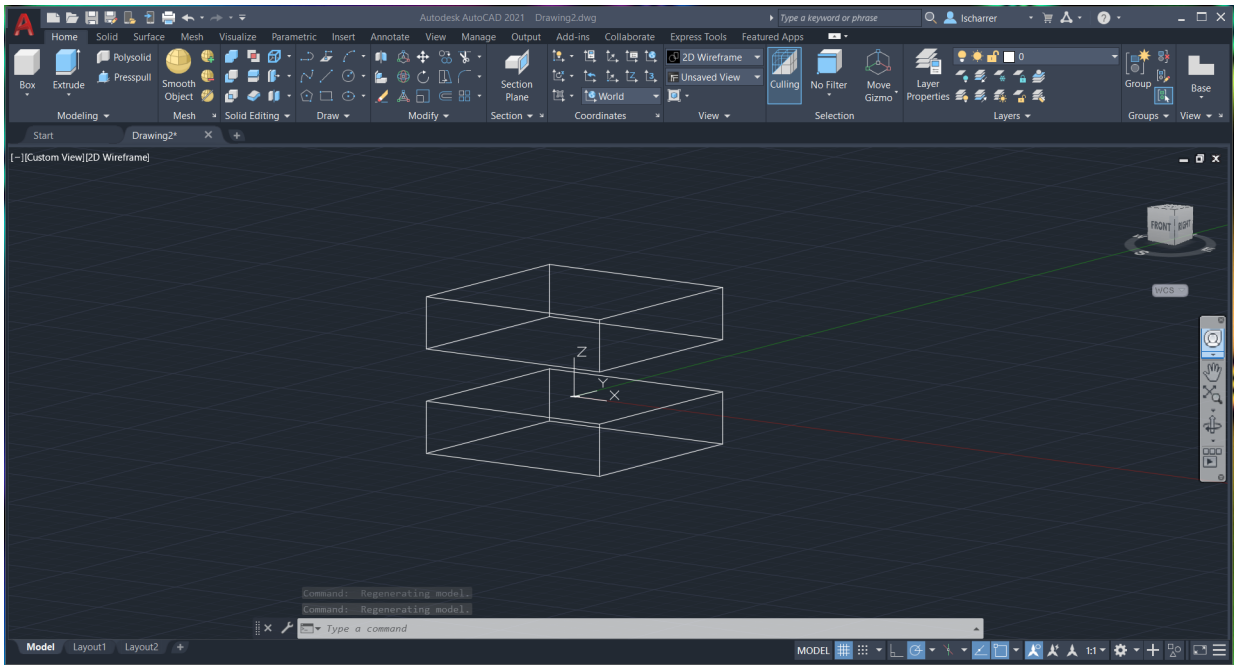


Figure 1: Two rectangular prisms which will be the electrodes of our parallel plate capacitor.

Now, the matlab PDE toolbox only simulates the insides of any solids we give it, so if we want these rectangular prisms to be our electrodes, and we want to see the electric field around them, we need to take a much larger box, and cut these plates out of the box, so that they form empty voids in the larger container box:

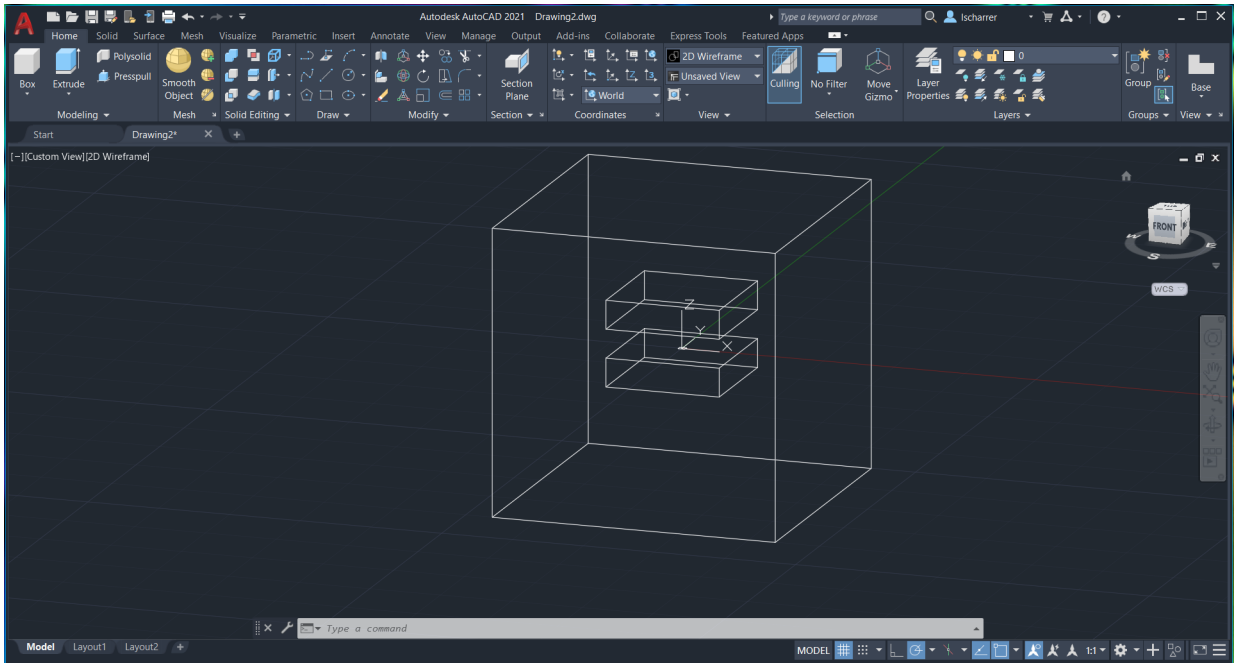
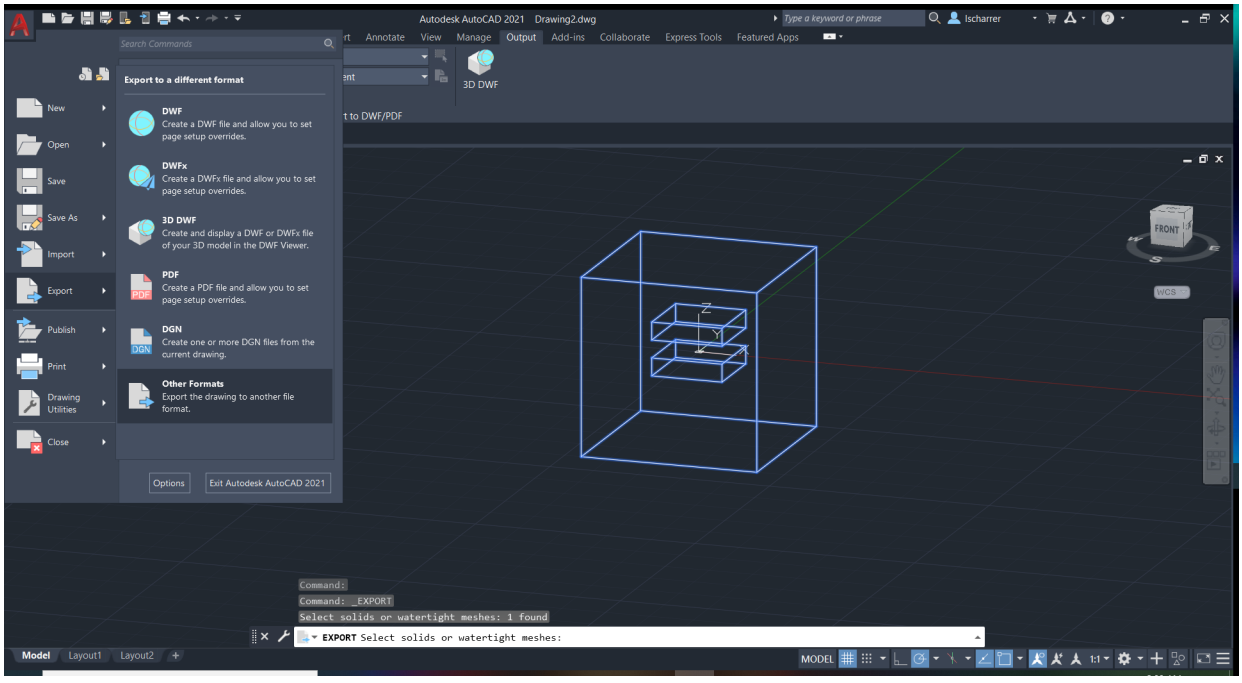
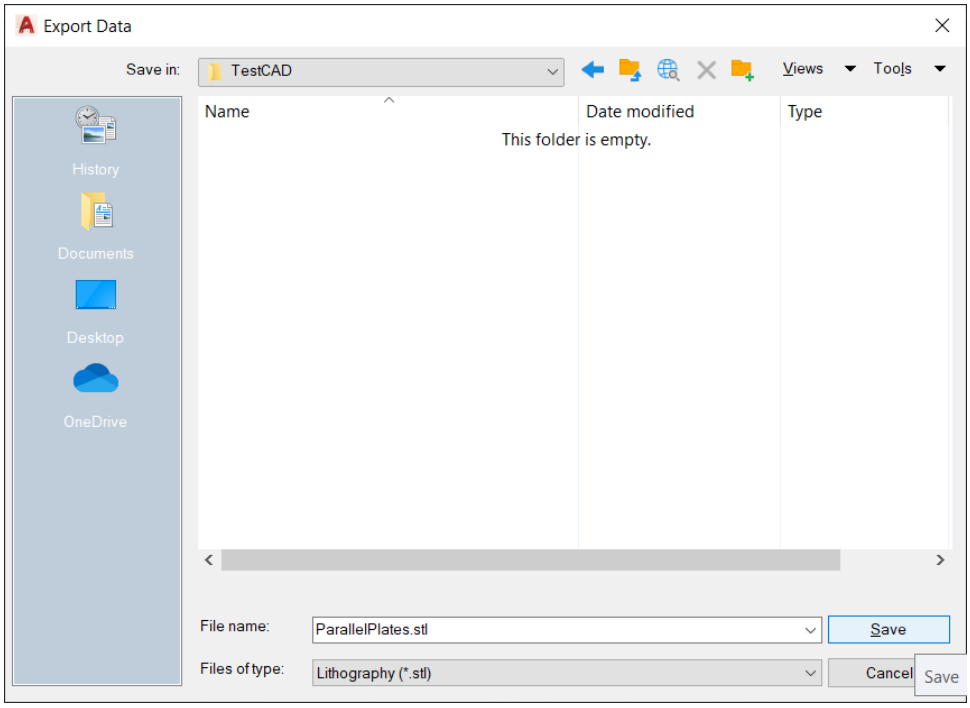


Figure 2: In order to simulate the electric field around these plates, we need to cut out them out from a larger box containing them. That way, we can just apply boundary conditions to these surfaces inside the larger box, and determine the solution to Laplace's equation in the space in between

This process will be general for any kind of design you come up with. If you have some electrode configuration designed in a CAD program, then in order to simulate the electric fields from those electrodes in the space around them, **you must cut them out of a larger shape that contains all of the electrodes.** After you've done this, you'll end up with one final solid, which is usually a large box or cylinder with a bunch of shapes cut out from the inside of it. From there, you just need to export that solid as a .stl from your CAD program. After you have that .stl, you're done with CAD, and the rest of the work will be in matlab.

In AutoCAD, to export a solid as a .stl file, you need to highlight the solid by clicking on it, then go to the main drop down menu (the big red A in the top left corner), click Export, and then click Other Formats:





Then, just save the file as a lithography (.stl) file: Now, we're done with CAD, and can move into matlab. Just make sure you remember where you've saved the .stl file if you're working on something other than what's included in the github.

2 Simulating an Electrostatic Potential in MATLAB

In this section, I'll go over the structure of a simulation script in detail, explaining the purpose of each function. I will be following the pdeTutorial.m script I wrote, so I recommend having that open while you read this tutorial. Also, all of the functions below have extensive official documentation here: https://www.mathworks.com/help/pde I highly recommend utilizing that, but it can sometimes be more general than we need it to be, since we're only solving Laplace's equation here.

There are two main types of objects in the matlab PDE toolbox: models, and results. A model contains all of the information about some problem you want to solve. This information includes (but is not limited to) the geometry of the system, the number of faces, edges, and vertices, the boundary conditions on these, and the computer-generated mesh over which the problem will be solved. A result is acquired after solving a model, and contains the just the mesh, and the values of the function and its gradient at each point in the mesh.

2.1 Initializing and Importing Geometry

The first thing we need to do is initialize a model object:

```
model = createpde();
```

Now, this model doesn't contain any information about the problem we want to solve. We need to put that information into the model. The first thing to add is the geometry of problem, which comes from the .stl file we created in the first section. This is done using the importGeometry function:

```
importGeometry(model,'STLs/ParallelPlates.stl');
```

Something to keep in mind, the command in this form only works if ParallelPlates.stl is located in a directory called STLs which is inside of your working directory (this should be the case if you're working with the files directly as downloaded from github). If the .stl file is located in a different directory, for the string in the second argument, you need to write out the path of that directory. For example, on my computer running Ubuntu, the path is

```
/home/luca/Desktop/Patterson_Group_Projects/PDEs/STLs/ParallelPlates.stl
```

For a mac, the path should look similar. For a windows computer, the path would look something more like

```
C:\Users\luca\Patterson_Group_Projects\PDEs\STLs\ParallelPlates.stl
```

So, now that we've created the model and imported the geometry, we need to apply the boundary conditions of our problem. In order to assign boundary conditions to a face, we need to know the labels of each face. So, we use the following function to plot an image of the geometry, where we can see the labels assigned to each face, and where the transparency of each face is set to 0.15, so that we can see the inside faces:

```
pdegplot(model,'FaceLabels','on','FaceAlpha',0.15);
```

Generally, I've found the best practice to be to include this function with a proper generation of a figure, and then putting a stop before any of the rest of your code, like in Figure 1. So, you run the function, it plots the geometry and pauses, and then you can write down which faces correspond to which labels in the comments of the code. Then, you stop the function from running completely, put the labels into the boundary conditions functions (which I'll get to in a moment), and comment out the code that generates the figure, since you're done with it.

```
figure(1);
pdegplot(model,'FaceLabels','on','FaceAlpha',0.15);
xlabel('x');
ylabel('y');
zlabel('z');
title('Geometry of system');

%Faces:
%  Outer Boundaries: 1:6
%  Upper Electrode: 13:18
%  Lower Electrode: 7:12

applyBoundaryCondition(model,'dirichlet','face',7:12,'u',100); % Lower Electrode
applyBoundaryCondition(model,'dirichlet','face',13:18,'u',-100); % Upper Electrode
```

Figure 3: Put a stop in the code before continuing beyond generating the figure, so you can write down the labels of the faces.

This is usually the most tedious part of the process, since the plot of the model's geometry matlab generates tends to be quite slow to load. Instead of using your mouse to zoom, pan, and rotate the geometry in the figure, I instead recommend using the xlim, ylim, zlim, and view commands to change your field of view, which you will probably need to do, since the labels can sometimes be hard to read. The first three functions are pretty self-explanatory, the view function just changes the angle you're looking at the figure from. Documentation for all 4 functions exists online.

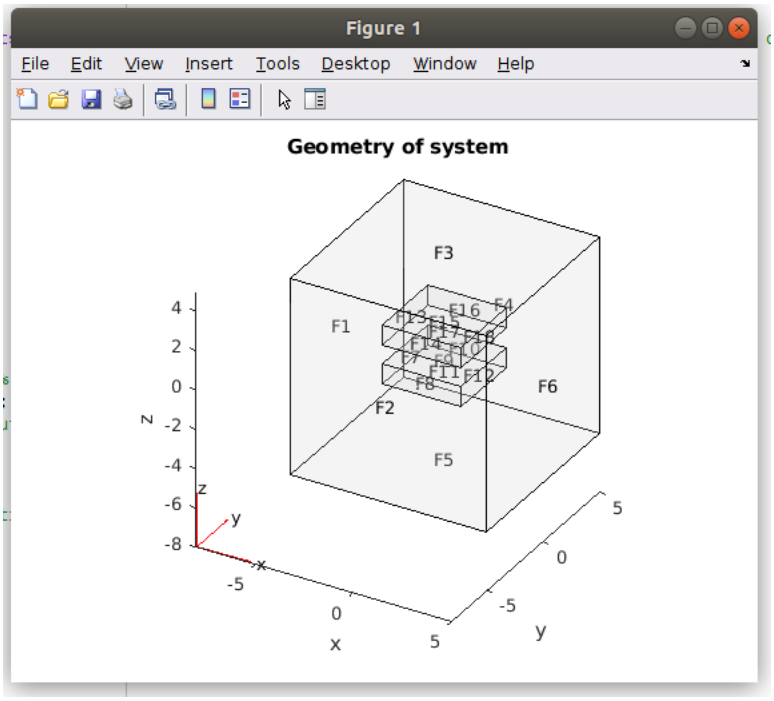


Figure 4: What the generated geometry plot will look like. Each face has a label, though they can sometimes be hard to read without changing your angle or zooming in or out.

2.2 Applying Boundary Conditions

So, after you know which labels correspond to which faces, it's time to apply your boundary conditions. There are two types of boundary conditions you can apply, Dirichlet, or Neumann. Dirichlet conditions mean that the value of the potential is constant at a surface, like on an electrode with an applied constant voltage. If you want to apply a constant voltage of 100 V on faces 1, 3, and 5, then you would write

```
applyBoundaryCondition(model,'dirichlet','face',[1,3,5],'u',100);
```

Neumann conditions mean that the normal derivative of the potential into the surface is constant, i.e., the strength of the electric field lines pointing into/out of the surface. Almost always we'll be dealing with just Dirichlet conditions, but Neumann conditions can be useful on occasion, like for modeling a perfect insulator with zero charge on the surface:

```
applyBoundaryCondition(model,'neumann','face',[1,3,5],'q',0,'g',0);
```

For Dirichlet conditions, 'u' refers to the value of the solution itself on the boundaries. For Neumann conditions, 'q' and 'g' refer to constants in an equation describing the behavior of the derivative at the boundary. More detailed information about these can be found here: https://www.mathworks.com/help/pde/ug/steps-to-specify-a-boundary-conditions-object.html

For our example, we found that the outer boundaries of the box correspond to faces 1 through 6, the lower plate electrode to be faces 7 through 12, and the upper plate electrode to be faces 13 through 18. So, if we want to model a parallel plate capacitor with 100 V on one electrode, and -100 V on the other, inside of a grounded box, we apply the following boundary conditions:

```
applyBoundaryCondition(model,'dirichlet','face',7:12,'u',100);
applyBoundaryCondition(model,'dirichlet','face',13:18,'u',-100);
applyBoundaryCondition(model,'dirichlet','face',1:6,'u',0);
```

Something to keep in mind, you don't necessarily have to apply boundary conditions to every single face. In that case, matlab seems to just treat those boundaries like any other point in the interior space. Often times you can omit boundary conditions on the outer boundaries if you're just interested in the fields in the interior. I tend to keep all of my outer boundaries at ground though, as I've occasionally come across nonphysical results if I omit too many boundary conditions, like results where the entire space is at a constant potential, since a constant still solves Laplace's equation.

2.3 Specifying Coefficients, Generating the Mesh, and Solving the Model

So, now that we've applied our boundary conditions, the next step is to specify exactly what equation we're trying to solve, which is Laplace's equation. This done with the following command:

```
specifyCoefficients(model,'m',0,'d',0,'c',1,'a',0,'f',0);
```

By default, the matlab PDE toolbox is set to solve a very general, complicated differential equation. All this command does is set most of the constants in that equation to zero, so that the resulting equation is Laplace's equation. More information about this can be found here:

https://www.mathworks.com/help/pde/ug/equations-you-can-solve.html

Then, we need to generate a mesh for our model to solve the equation on. If you trust matlab's default parameters (which you often shouldn't), then you write:

```
generateMesh(model);
```

There are three optional parameters that specify the details of this mesh. The first, and most important, is the maximum element size parameter, 'hmax'. The mesh generated is composed of tetrahedrons, and the hmax parameter tells matlab what the maximum side length of one of these tetrahedrons can be, in the units of your model (usually mm). So, by decreasing hmax, the mesh is finer, being composed of smaller tetrahedrons. Generally, this will give you a more accurate result, but will come at the cost of computation time. The smaller you set hmax, the longer it will take the computer to solve the pde. **If you set hmax to be too small, you will freeze and crash matlab and/or your entire computer.** If you're working on fairly complicated, intricate geometries, this will inevitably happen a lot, so **save your work often!**

I recommend starting with a relatively large hmax, so that you start with a rough result that generates instantly. Then, rerun the code with incrementally smaller values of hmax, until decreasing hmax doesn't change the solution anymore, or until the program starts to take so long to solve that decreasing it any further will crash matlab. For really complicated geometries, I've had to let these programs run for multiple hours to get good results. During that time, my computer was essentially frozen, so beware of that.

The other two parameters are hmin and hgrad. hmin is what it sounds like, it governs the minimum side length of the mesh. Generally smaller is better, and altering this parameter won't usually affect the runtime too much. You probably won't need to alter this parameter unless you have some pretty small, intricate parts in your geometry. hgrad governs how quickly a finer portion of the mesh can transition into a more coarse portion as you move throughout the geometry. You probably won't need to alter this one at all, but it's good to at least know it exists if you do want to. More information about these parameters can be found here:

https://www.mathworks.com/help/pde/ug/pde.pdemodel.generatemesh.html

So, if you want to generate a mesh with a maximum element size of 0.5 mm, and a minimum size of 0.1 mm, then you would write

```
generateMesh(model,'hmax',0.5,'hmin',0.1);
```

Now, you've applied all of the conditions of your problem to the model, so the only command left is to solve it:

```
result = solvepde(model);
```

This is the part of the program that will take a long time to run if you're dealing with a very fine mesh. So, instead of continuing with the following commands in the same script, I generally have a script that just returns the result object, so that I can hold the result in that matlab workspace, where I can either input the commands that follow directly into the command line, or write them into another script, and call that script with the result in the workspace. This also allows you to save the result as a .mat file, so that instead of having to run the script again, which could take up to hours, you can simply load it back into the matlab workspace with the load command.

2.4 Extracting and Visualizing the Results

Now that we have our result object, we need to figure out how to extract the relevant information from it. This is mainly done using two functions, depending on whether you want the function itself, or its gradient. If you have a point inside of the geometry with coordinates x,y, and z, then, to find the potential V at that point, you type:

```
V = interpolateSolution(result,x,y,z);
```

which will return the scalar value V. Something to note, if you try to use interpolateSolution with the coordinates of a point that is not contained within the geometry of the model, like outside of the outer boundaries, or inside of one of the plates in our example, then function will return NaN, and will show up as an empty spot on plots. If, instead, you want the gradient at that point, you use:

```
[gradX,gradY,gradZ] = evaluateGradient(result,x,y,z);
```

Something to keep in mind: while the potential V will be returned in units of volts, the gradient will have units V / mm, since the default length unit is mm. So, in order to calculate the electric field from the gradient, you need to multiply by 1000 to convert it to V / m, and by -1, since $E = -\nabla V$:

```
Ex = - 1e3 * gradX;
Ey = - 1e3 * gradY;
Ez = - 1e3 * gradZ;
```

So, we now know how to get the potential and electric field at any point from our simulation. But what if we want to consider a whole bunch of points, so that we can plot the potential or electric field along a line or over a plane? In order to do that, we use the meshgrid function. This function generates 3 3-dimensional arrays, X, Y, and Z, that list the coordinates at a bunch of different points. For example, say I want a grid of coordinates, where $-2 < x < 2$, $-3 < y < 3$, $-4 < z < 4$, and where the x and y coordinates are spaced by 0.1, and the z coordinates are spaced by 1:

```
[X,Y,Z] = meshgrid(-2:0.1:2, -3:0.1:3, -4:1:4);
```

This will return three 61 by 41 by 9 arrays, Then, you can extract the potential at each of these points with the following two commands:

```
V = interpolateSolution(result,X,Y,Z);
V = reshape(V,size(X));
```

The interpolateSolution command returns a 22,509 by 1 array, so the reshape command is required to put it in the same format as the X, Y, and Z arrays. So, for three indices i, j, and k, X(i,j,k), Y(i,j,k), and Z(i,j,k) return the coordinates corresponding to those indices, and V(i,j,k) returns the potential at those coordinates. The two most useful ways of visualizing potentials (or electric fields) are 1-D plots and 2-D surface plots. Say you want to plot the potential along the z-axis of some object, like an ion trap. Then, you would start with generating the grid and interpolating the solution:

```
[X,Y,Z] = meshgrid(0,0,-5:0.1:5);
V = interpolateSolution(result,X,Y,Z);
V = reshape(V,size(Z));
```

Then, all of the arrays are 3-D, 1 by 1 by 101 arrays. In order to use matlab's plot function, they need to be 101 by 1, so we again use the reshape function:

```
z = reshape(Z,[101, 1]);
v = reshape(V,size(z));
```

Then, we can just plot the results:

```
figure;
plot(z,v);
xlabel('z (mm)');
ylabel('Potential (V)');
```

Now, instead of a 1-D plot along an axis, let's say you want to visualize the potential on the x-y plane. The process is fairly similar to before:

```
[X,Y,Z] = meshgrid(-5:0.1:5,-5:0.1:5,0);
V = interpolateSolution(result,X,Y,Z);
V = reshape(V,size(X));
x = reshape(X,[101,101]);
y = reshape(Y,[101,101]);
v = reshape(V,size(x));
```

```
figure;
surf(x,y,v);
xlabel('x (mm)');
ylabel('y (mm)');
zlabel('Potential (V)');
```

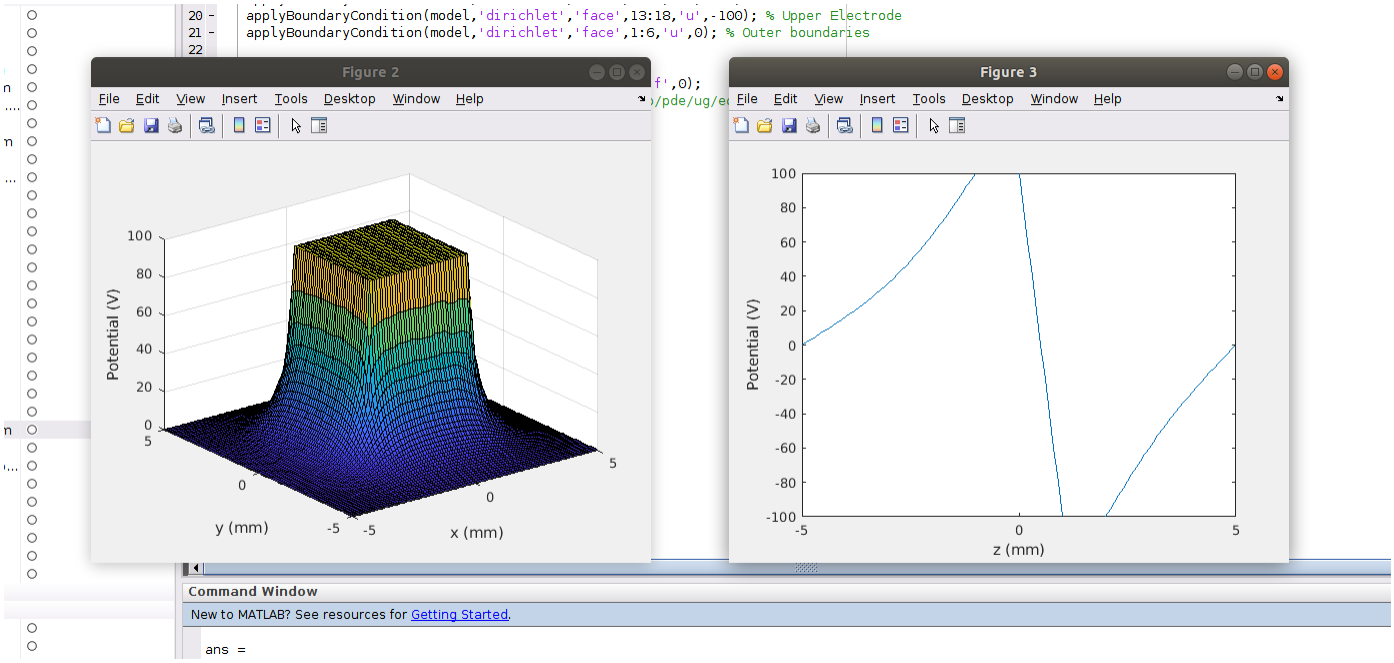


Figure 5: What the two figures generated from the plot and surf functions of the potential should look like. On the left, the x-y plane is right up against the side of the lower plate, so the potential in the center is at 100 V. On the right, notice how at the z-coordinates inside of the two plates, the plot is empty, because the interpolateSolution function returns NaN at these points.

These two methods I've found to be the most useful in visualizing potentials and electric fields. There are various methods for visualizing these in 3-D, but I rarely use them, so instead of writing an extra tutorial for them, I will just link you to this page, which already has a pretty nice tutorial:
https://www.mathworks.com/help/pde/ug/plot-3-d-solutions.html

3 Simulating an RF Pseudopotential

In the previous section, we learned how to use the matlab PDE toolbox to simulate an electrostatic potential due to constant potentials on the surfaces of electrodes. In this section, we now switch our focus to electrodes with an applied RF voltage, and calculate the pseudopotential due to this RF voltage. We'll be following along in the pdeTutorialPseudopot.m script, so have that open. In order to do this, we must calculate the ponderomotive force, described in detail here: https://en.wikipedia.org/wiki/Ponderomotive_force

I highly recommend at least reading the first section of the above wikipedia page, in order to understand what we're doing. First, I will describe exactly how to calculate the pseudopotential mathematically, then I'll show how to do so in matlab. From the wikipedia page, if you apply an RF voltage to some electrode with frequency ω, then the electric field at any point $\vec{r}$ will be $\vec{E}(\vec{r})\cos(\omega t)$, where $\vec{E}(\vec{r})$ is the electric field at $\vec{r}$ when the voltage on the electrode is at its maximum, i.e. what the electrostatic field would be if the electrode were held at a constant voltage equal to the amplitude of the RF voltage. Then, as proven in the wikipedia page, the effective ponderomotive force on a particle with charge e and mass m would be

$$\vec{F}_P = -\frac{e^2}{4m\omega^2}\nabla(|\vec{E}|^2) = -\nabla(\frac{e^2E^2}{4m\omega^2})$$

Then, since $\vec{F} = -\nabla U$, where U is the potential energy, we thus have an effective potential energy:

$$U = \frac{e^2 E^2}{4m\omega^2}$$

The dimension of U here is units of energy, but we want an effective potential with units of volts, so, since a particle with charge e in a potential V has potential energy $U = eV$, we then divide the previous equation by e to obtain an effective potential with units of volts:

$$V(\vec{r}) = \frac{e}{4m\omega^2} E^2(\vec{r})$$

So, in order to calculate the pseudopotential due to some RF electrodes, all we have to do is calculate their electrostatic field when the electrodes are at their amplitude voltage, then the pseudopotential at each point is just the amplitude squared of the electric field, multiplied by a bunch of constants.

Now, let's move into pdeTutorialPseudopot.m in order to calculate the pseudopotential of a cylindrical quadrupole, an essentially ideal version of the radial part of a linear paul trap. I won't go over all of the preliminary steps like before, but I do want to mention one thing. If you're using the CylindricalQuadrupole.stl file, then on the pdegplot() step, you should notice that the 4 cylinders have a lot more than 3 faces each. This happens when you have cylinders in your model that are large relative to the entire model. If there's a small cylinder, it will generally just have 3 faces, but if the cylinder is large, then the curved side will be split up into multiple faces. This doesn't really change anything, except for that fact that in order to put a potential on the cylinder, you have to count up and include all of the face labels, which can sometimes be hard to read, often requiring you to change your perspective and zoom in. This is just something to watch out for that can be a bit of a pain.

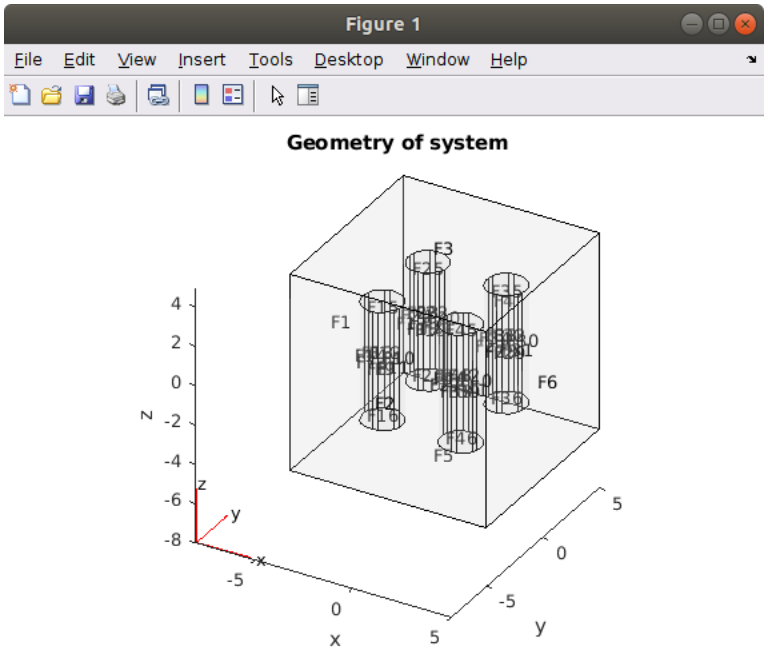


Figure 6: The geometry of the cylindrical quadrupole. Notice how, since the cylinders are relatively large with respect to the entire model, the curved sides have been split up into multiple faces.

Now, in this cylindrical quadrupole model, we're going to keep the outer boundaries and 2 diagonal electrodes at ground, and apply an RF voltage with frequency 2 MHz and amplitude 150 V to the other two. We begin by just following the same process as before, solving the pde, and getting a result object. We can plot the true instantaneous potential in the x-y plane, and we'll see the saddle point at the center, showing that this electrostatic potential would not be able to trap an ion on its own.

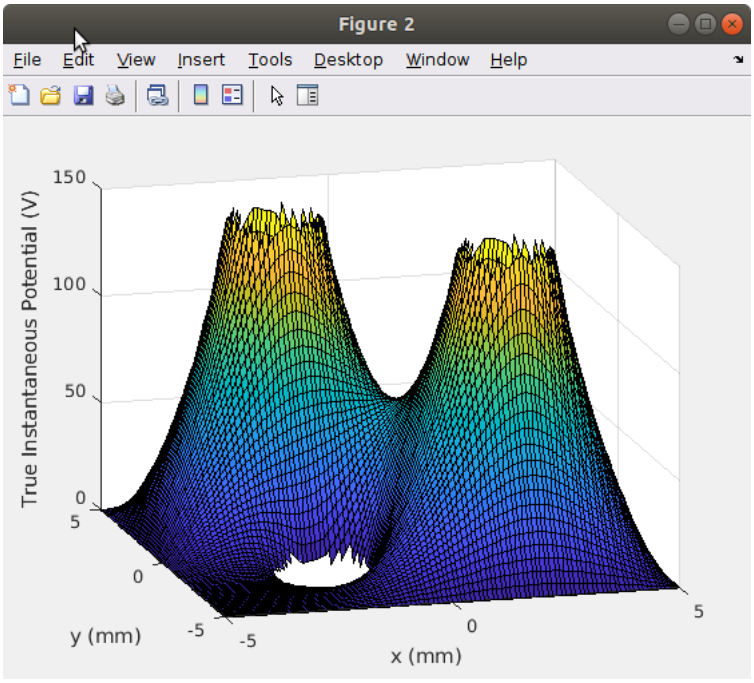


Figure 7: True instantaneous potential of the cylindrical quadrupole along the x-y plane.

After this, we then call the calcPseudoPot function:

```
Vpseudo = calcPseudoPot(result,X,Y,Z);
```

This is a function I wrote, it is not included by default in the PDE toolbox, so you'll need to make sure you have the calcPseudoPot.m file in the same directory as pdeTutorialPseudopot.m, and any other scripts that call it. If we open up calcPseudoPot.m, the first thing we'll see is the calculation of the amplitude squared of the E field:

```
function pseudoPot = calcPseudoPot(result,X,Y,Z,frequency)
[gradX,gradY,gradZ] = evaluateGradient(result,X,Y,Z);
gradX = reshape(gradX,size(X));
gradY = reshape(gradY,size(Y));
gradZ = reshape(gradZ,size(Z));
Ex = -gradX;
Ey = -gradY;
Ez = -gradZ;
Esquared = (Ex .* Ex) + (Ey .* Ey) + (Ez .* Ez);
```

In the top line, you'll see there's an extra argument we didn't include previously. This is because it defaults to the value we want of 2 MHz, as we'll see in the following code. Next, we input the values of the constants. Here, I assume the charge and mass of a Sr^{+1} ion.

```
if nargin < 5
    frequency = 2e6; % Default value of 2 MHz
end
charge = 1.602e-19; % Coulombs
mass = 1.46127438e-25; % kg
angFreq = frequency * 2 * pi;
conversionFactor = 1e6; %mm^2 / m^2

coeff = conversionFactor * charge / (4 * mass * angFreq^2);
```

Something important to keep in mind: the units of length in all of the .stl files I've used is mm. So, when we calculate the electric field from the gradient, we get it in units of V/mm, meaning E^2 has units V^2/mm^2. However, all of the constants used in the formula for the pseudopotential I've written in SI units, so I need to multiply by a conversion

factor of $10^6 mm^2/m^2$ in order to get the pseudopotential in volts. So, we combine all of these constants into a single coefficient according to the above derivation, and finally we return:

```
pseudoPot = coeff * Esquared;
```

Thus, we have calculated the pseudopotential, and have it in the same form as the true instantaneous potential. We can then plot it on the same x-y plane with the surf function, and we see that instead of a saddle point, we have an approximately harmonic local minimum at the center of the quadrupole, showing that the RF pseudopotential can confine ions radially. Notice how at their lowest points, the walls of the pseudopotential between the electrodes only go up to about 4-5 V. This means that, despite a 150 V amplitude RF potential on the electrodes, the quadrupole can only radially confine ions with up to about 4-5 eV of energy.

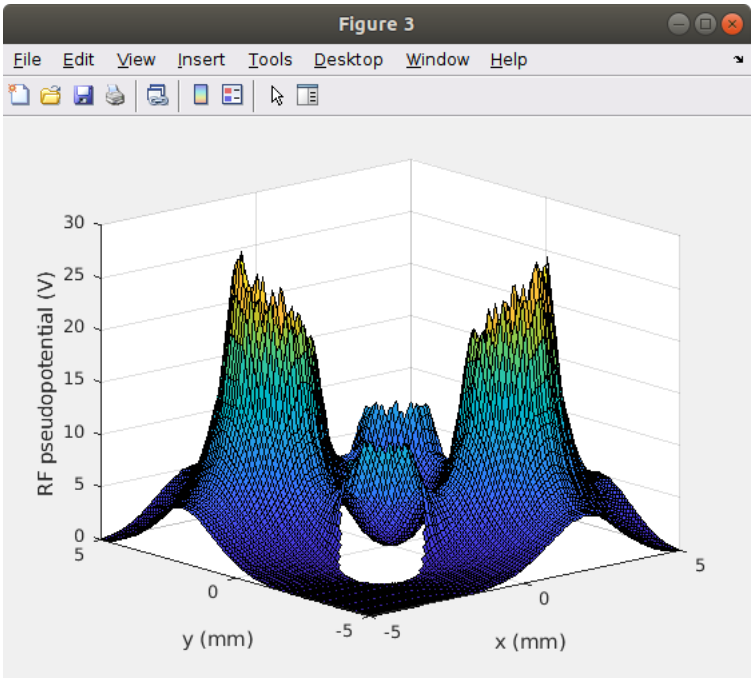


Figure 8: The pseudopotential due to the RF electrodes in the x-y plane, showing that the RF field provides radial confinement in an approximately harmonic pseudopotential.

4 Adding and Scaling Potentials

So, we now know how to calculate individual potentials and pseudopotentials. However, when you're working with models that have different potentials on different electrodes, it's often useful to be able to quickly modify the potential on one electrode while holding the others constant. Fortunately, because of the linearity of Laplace's equation, we can do this without having to re-solve the PDE object.

Consider the parallel plate capacitor example from section 2. In this example, we put 100 V on the bottom plate, and -100 V on the top plate. If we instead wanted to see the results of +/- 200 V, instead of re-running the whole program with the new boundary conditions, we can just take our initial potential, and multiply the value at every point by 2:

```
V100 = interpolateSolution(result,X,Y,Z);
V200 = V100 * 2;
```

What if the new boundary conditions you want aren't just a scalar multiple of a result you already have? Let's say you want to apply Vb volts to the bottom plate, and Vt to the top plate, and you want to see the results for a variety of different values of both Vb and Vt. Instead of running scripts to solve the model for each combination of voltages, instead, you can write a script that will just solve two models: 1 V on the bottom plate with 0 V on the top, and 0 V on the bottom plate with 1 V on the top (Note: I believe you will need to calculate each result one at a time, as it seems like matlab won't let you run a script that generates and solves two different models at once). Say the former result is called result10, and the latter is result01. Then, for any Vb and Vt, you can quickly determine the total potential Vtotal:

```
V10 = interpolateSolution(result10,X,Y,Z);
V01 = interpolateSolution(result01,X,Y,Z);
Vtotal = (Vb * V10) + (Vt * V01);
```

This method is extremely useful, particularly for simulating ion trajectories in the Paul trap, or in any configuration where you want to be able to change the voltage on the electrodes. Be thankful that potentials are governed by a linear PDE! If Laplace's equation weren't linear, we wouldn't be able to pull this trick.

I've found that the best way to use this is to generate result objects for every electrode that isn't constantly ground, where each result has 1 V on that electrode, and 0 V on every other electrode. Then, you can easily generate arbitrary configurations by scaling and combining the fields of all of the different electrodes.

We'll cover this again in the section on simulating ion trajectories, but you can also use this method to simulate time-dependent potentials. There is a built-in way to do time-dependent solutions for general PDEs in the matlab PDE toolbox, however I've found this solution to be much easier to implement, so I haven't bothered with that. I don't think it would work for every possible PDE, but again, since Laplace's equation is linear, it works fine for us. Let's consider the cylindrical quadrupole from section 3: put 0 V on two of the diagonal electrodes, and an RF voltage with amplitude 150 V and frequency 2 MHz on the other two. Then, you can find the potential at time t by generating a result (here called resultRF) with 1 V on two of the electrodes and 0 V on the other two, and by multiplying this result's potential by the the amplitude and a cosine function:

```
V1 = interpolateSolution(resultRF,X,Y,Z);
amplitude = 150;
angularFrequency = 2 * pi * 2e6;
V = V1 * amplitude * cos(angularFrequency * t);
```

Of course you can also add an initial phase into the cosine function. This method will be useful for simulating trajectories in the Paul trap, where we'll calculate the overall instantaneous potential by summing contributions from both the DC electrodes and the RF rods.

Finally, to some extent, you can also use this adding and scaling potentials method to combine RF pseudopotentials with constant real potentials, to get an overall "effective potential", on which you can do things like determine secular frequencies or see how harmonic a trap is. However, there is a caveat to this: you can scale a true potential by any amount you want, and you can add a true potential to a pseudopotential, but you **cannot** scale pseudopotentials in the same way as potentials. This is because the pseudopotential is calculated from a nonlinear equation for the ponderomotive force. Multiplying a pseudopotential by a scalar is not equivalent to multiplying the amplitude on the electrodes by that scalar. So, whenever you want to try changing the amplitude or frequency of an RF pseudopotential, you must re-calculate it from the beginning (Note: there might be a more complicated way to do this scaling for varying the amplitude and frequency, but I didn't do this often enough to try to figure that out).

As an example, let's go back to the cylindrical quadrupole. Let's say we want to know the overall effective potential if we put the same RF voltage from before on two of the electrodes, but instead of having the other two at ground, we hold one of them at a constant 5 V, and another at a constant 10 V. So, you'd generate 3 results: resultRF, which is the same as before, except with 150 V on each electrode instead of 1 V (since we can't scale pseudopotentials), resultDC1 with 1 V on one of the two previously ground electrodes with 0 V on everything else, and resultDC2 with 1 V on the other of the two previously ground electrodes with 0 V on everything else. Then, you can calculate the total effective potential Veff:

```
V1 = interpolateSolution(result1,X,Y,Z);
V2 = interpolateSolution(result2,X,Y,Z);
Vpseudo = calcPseudoPot(resultRF,X,Y,Z);
Veff = (5 * V1) + (10 * V2) + Vpseudo;
```

5 An Example: The Linear Paul Trap

Now we know how to calculate potentials and pseudopotentials, and how to quickly scale and add them together. We're almost ready to start using these simulations to extract real, useful information. In order to do this, however, we need to have an example system to work on. So, we're going to go through the previously described processes with the current (as of August 2020) design of the Paul trap in the ion trapping experiment. I will show you how to use a couple of the scripts I've written that will do everything above, then I'll show you a few examples of visualizing the potentials.

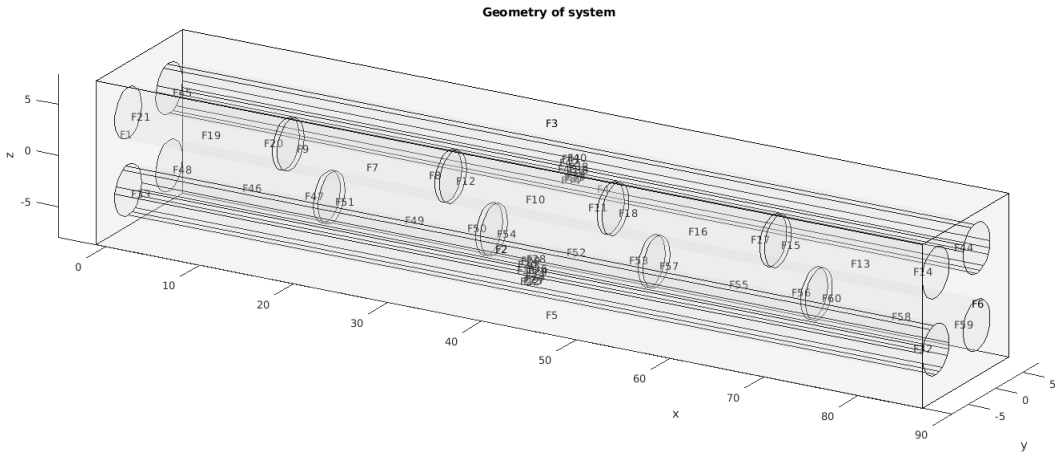


Figure 9: The geometry of the Paul trap. This is what the pdegplot() function returns in the paulFields.m script. Notice how the RF electrode cylinders are broken up into many faces because of their size, while the individual DC electrodes are not.

There are two scripts that you should have open in your working directory: paulFields.m and paulTutorial.m. A quick explanation of the two functions: paulFields(i,v) performs the actual calculations of the fields by placing v volts on some set of electrodes and 0 V on everything else. Which set of electrodes that is depends on the input integer i, which is between 1 and 6. Each set of diagonal electrodes is assumed to have the same voltage, so the RF rods will have the same voltage, the upper and lower electrodes on the far right will be the same, the two in the center will be the same, etc. Take a look at the commented sections inside paulFields() to see which values of i correspond to which electrodes. paulTutorial() calls paulFields() to get result objects, and then uses them in a variety of example visualizations of the potentials of the Paul trap.

All you need to do is run paulTutorial(). It will likely take a few minutes to finish running, then will generate a few different figures. Figure 1 (in matlab, not figure 1 in this paper) shows the instantaneous true potential from the RF electrodes at their amplitude voltage on a radial plane.

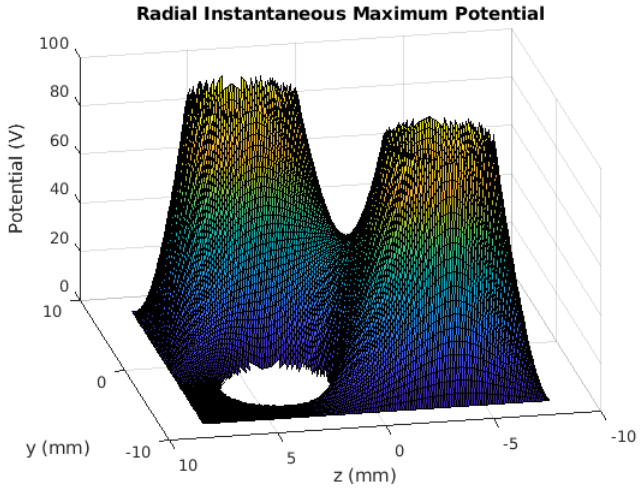


Figure 10: Radial instantaneous potential due to the RF electrodes. Notice the characteristic saddle point in the center.

Figure 2 shows the pseudopotential due to the RF electrodes on the same plane.

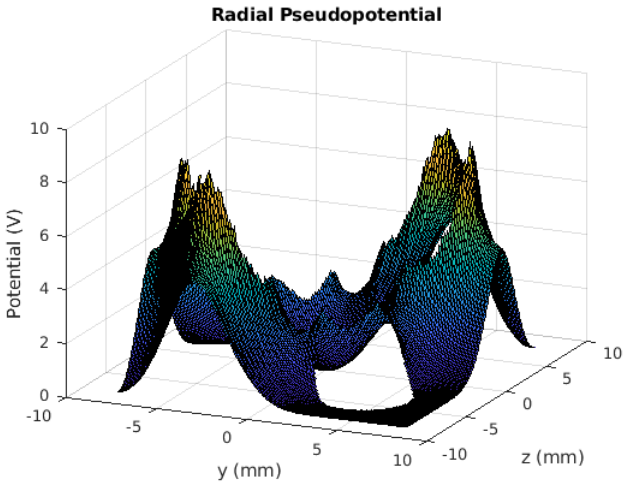


Figure 11: Radial RF pseudopotential. Notice how, unlike the true potential, the center is a stable equilibrium point, and looks fairly harmonic.

Figure 3 shows the potential due to voltages on the DC electrodes along the axis of the trap.

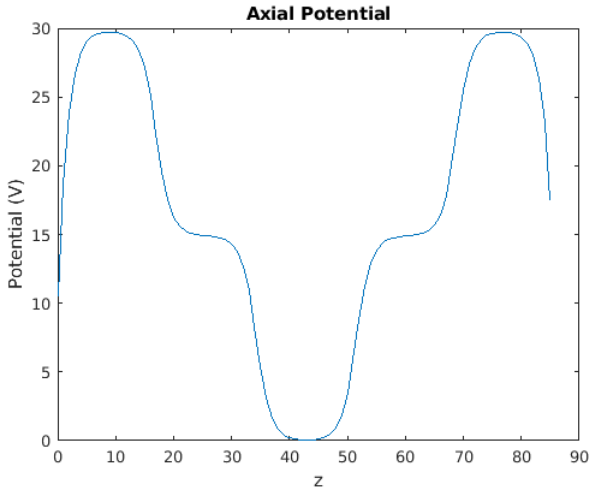


Figure 12: Potential along the axis of the trap due to the DC electrodes. Here, the two furthest endcaps are placed at 60 V, the two closer endcaps at 30 V, and the center electrodes at 0V. Notice how, along the trap axis, the potential is only half the value on the closest endcap pair. This is because the RF electrodes are assumed to be at 0 V in these simulations, so, being halfway in between, the potential along the axis is half the potential on th electrodes.

I recommend running the program a few times, varying the voltages and RF frequency to see how the potentials change. The places to vary these parameters are indicated in the script with comments.

6 Calculating Secular Frequencies and Harmonicity

Now that we can calculate potentials and pseudopotentials, and have a way of quickly scaling and adding them together, we can finally start to get some useful information out of these simulations. For this section, you'll need to have open the paulSecularFreqsTutorial.m and harmonicity.m scripts.

We'll start with two questions that we'd like to be able to answer about our ion traps. Assume we put some arbitrary voltages on the electrodes. What will the trap's secular frequencies be? And how closely do the trap's potentials and pseudopotentials approximate to being harmonic? We'll learn how to answer these questions for the linear Paul trap from the previous section, but the methods should generalize to any kind of ion trap. We'll work with 60 V on all of the DC electrodes except for the center, which will be our trapping region, at 0 V. The RF electrodes

will have an amplitude of 100 V and a frequency of 2 MHz. In order to calculate the axial and radial secular frequencies, we'll need to look at 1D slices of the overall effective potential (DC potentials + RF pseudopotential). If we plot the axial frequency, we get

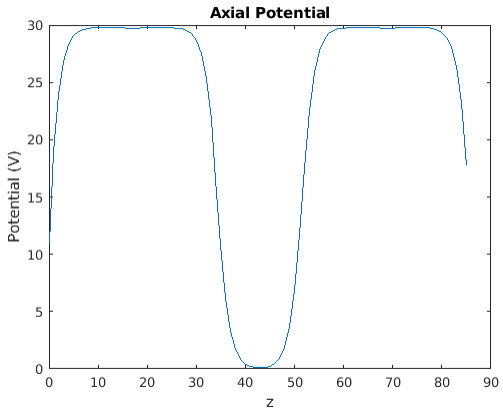


Figure 13: The full axial effective potential along the length of the entire trap.

Plotting a 1-D slice of the radial effective potential, we have

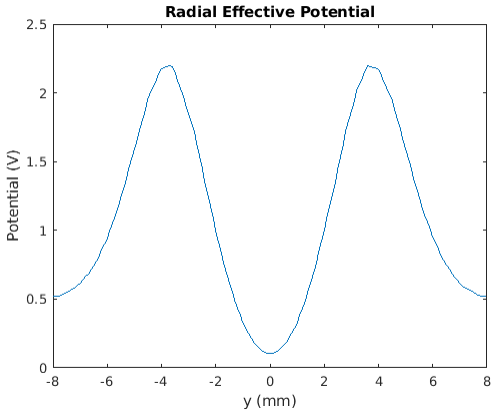


Figure 14: The full radial effective potential in a slice along the y-axis.

In order to calculate the secular frequencies and quantify the "harmonicity", we're going to use a function called harmonicity(). It will apply a best-fit parabola to the potential we input into it, and return the secular frequency of that parabola (as well as the sum of the squared residuals, but we'll get to that later). However, if we put either of the two potentials plotted above, the function's choice for a best-fit parabola is going to be way off, since neither of these potentials are very parabolic at all. So, we need to restrict our domain to just the trapping region, i.e. the part of the potential that looks at least harmonic-ish. For the axial potential, this appears to be between about x = 30 and 60 mm, and for the radial potential, between about y = -4 and 4 mm. You don't need to have it exactly symmetric about the trapping region, as long as it looks at least somewhat like a parabola, with the trap center being the absolute minimum of the graph, harmonicity() should be able to handle it. Let's plot the axial and radial potentials again, now restricted to their new domains.

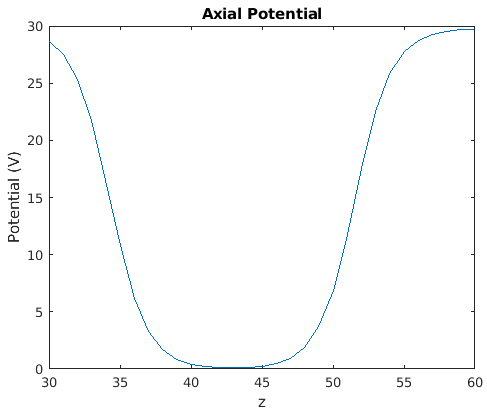


Figure 15: The axial effective potential confined to a smaller domain around the trap center.

Plotting a 1-D slice of the radial effective potential, we have

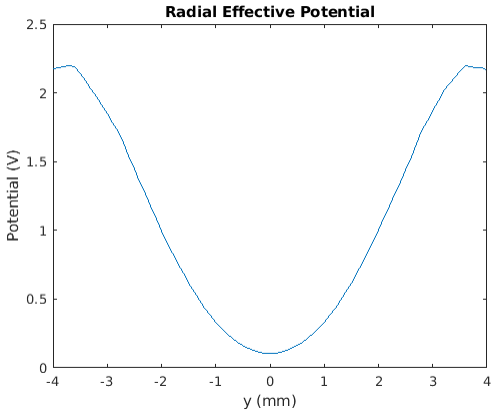


Figure 16: The radial effective potential confined to a smaller domain around the trap center.

The paulSecularFreqsTutorial.m script will already have the confined versions of the the two potentials. For each potential, it will plot it as we have above, then call the harmonicity() function. This function takes three arguments: the domain and range of the potential (X and V, for example), and a cut-off voltage (actually it also takes a fourth parameter, a string that will prefix the word "potential" on the plot it generates, but you don't need to worry about that since it's an optional parameter). This cut-off voltage input determines how much of the potential the function will try to fit to: if you input a cut-off voltage of 1 V, then the function will essentially cut off any part of the potential which is more than 1 V above the lowest point, and only try to fit to the portion of the potential which is within the 1 V maximum of the lowest point. This allows you to easily specify what kind of energy scales you want to deal with. If you're considering ions with up to 1 eV of energy, then they can go 1 V up the potential. If the ions only have 0.1 eV, then you can set the cut-off to 0.1 V, so that the function doesn't try to fit to the unused portion of the potential. This is important because, as we'll see, potentials which stray higher and further away from the minimum tend to be less harmonic, so fitting to only the bottom portion of the potential will give a much more accurate frequency for small oscillations than fitting to the entire potential.

So, given these three inputs, harmonicity() will start by rescaling the potential so that its minimum lies at 0 V, and then removing any of the points which are above the cut-off voltage. Then, it will calculate a best-fit harmonic potential, and determine the frequency of oscillation of a strontium ion inside of that harmonic potential. It will also calculate the sum of the squared residuals (SSR) by subtracting the true potential from the approximate harmonic potential at each point, squaring that value, and summing the values over all of the points. This gives a relative measure of the "harmonicity", how close the true potential is to its best-fit harmonic potential. Something to note, the SSR depends on the number of points in the domain X, so it can't be used as an absolute measure of harmonicity. It can only be used to compare the harmonicities of two different potentials defined on the same domain. For example, if you start with a potential caused by certain voltages on the electrodes, and you want to see whether increasing the voltages on the electrodes will make the potential more or less harmonic, then you can apply the harmonicity

function to both the original potential and the new potential defined on the same domain. If the SSR of the new potential increases, then it has become less harmonic than the original potential, and if the SSR decreases, then it's more harmonic.

Along with returning the calculated approximate frequency and SSR, the harmonicity() function will also generate a plot of both the original potential and the best-fit harmonic potential, allowing you to qualitatively judge how harmonic the potential is. The plot will also list the frequency and SSR, so you don't necessarily need to have the harmonicity function return the values if you just want to see them. Let's take a look at the results of applying harmonicity() to the axial potential, with three different cut-off voltages: 10 V, 1 V, and 0.1 V:

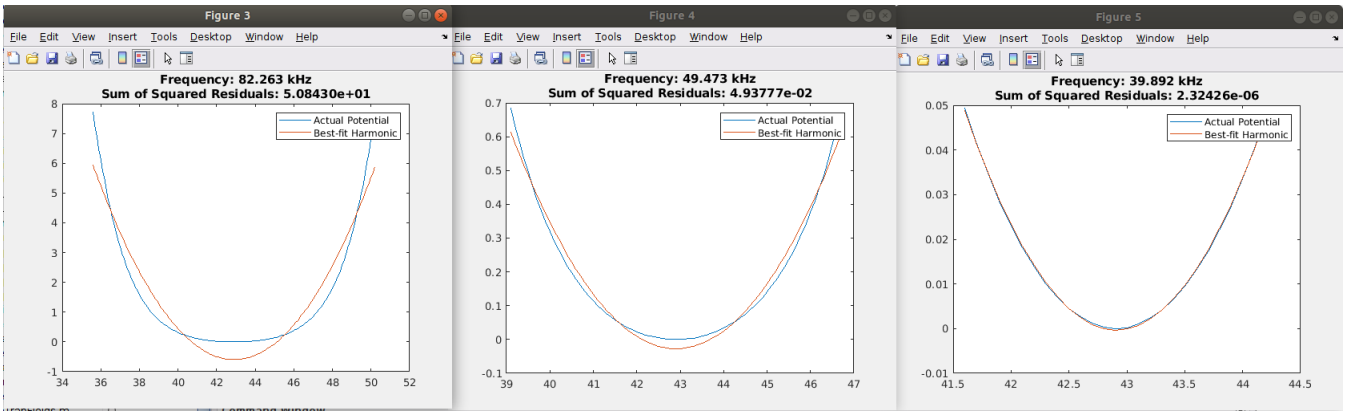


Figure 17: 3 figures generated by calling the harmonicity() function on the axial potential with 3 different cut-off voltages, 10 V, 1 V, and 0.1 V, from left to right. For higher energy scales, the potential is highly anharmonic, but for small amplitude oscillations about the center, the potential looks more and more harmonic.

If we consider the oscillation of ions with 10 eV of energy (cut-off voltage = 10V), then the axial potential looks highly anharmonic, and the best-fit harmonic is a pretty bad approximation. For 1 eV oscillations, the potential starts to look more harmonic, but is still noticeably different from its best-fit harmonic potential. For 0.1 eV oscillations, the potential is almost indistinguishable from the best-fit harmonic, so we can probably trust the calculated frequency of about 40 kHz.

Let's apply this to the radial frequency as well:

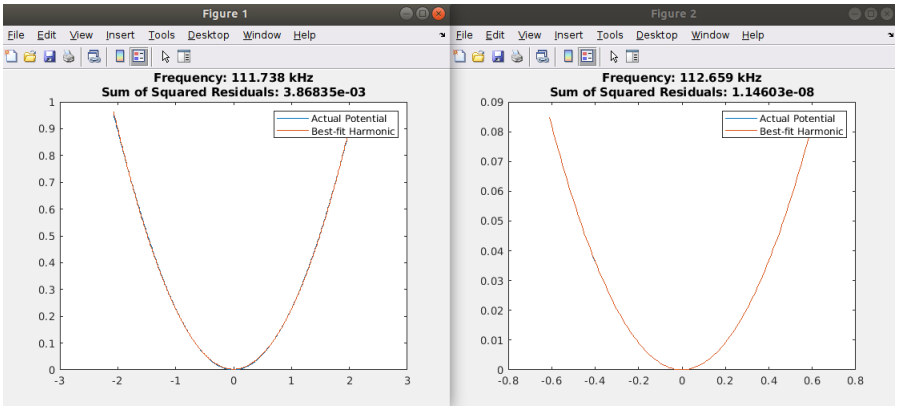


Figure 18: harmonicity() applied to the radial pseudopotential for cut-off voltages of 1 V on the left and 0.1 V on the right.

The radial pseudopotential appears noticeably more harmonic than the axial potential, especially for small oscillations, where the difference between the pseudopotential and best-fit harmonic is essentially nonexistent, giving us a radial secular frequency of about 113 kHz.

So, by applying the harmonicity() function to the potentials in the axial and radial directions of our linear paul trap, we've calculated the axial and radial secular frequencies to be approximately 40 kHz and 113 kHz, respectively. I recommend varying the voltages and RF frequency in the paulSecularFreqsTutorial.m script to see how these potentials and secular frequencies change.

paulSecularFreqsTutorial.m is a good example of both calculating the secular frequencies and adding and scaling different potentials. However, the latter isn't strictly necessary for this application, and actually causes it to run slower than it needs to. So, I've also included another function, paulCalcSecularFreqs.m, which is less pedagogical and more useful for actually calculating secular frequencies in the Paul trap. I've also included whaleCalcSecularFreqs.m, which does the exact same thing but for the whale trap design. I recommend playing around with both, you can call them directly in the command line with the voltages on each electrode and RF amplitude and frequency as inputs, which you can see if you open up both functions.

7 Calculating Ion Trajectories

This section, like the previous one, will be focused less on explaining in detail how the code works (since that would take a long time) and more on how to use the code. There are a few different versions of the code that calculates ion trajectories, corresponding to different models, but each of them works essentially the same way: taking previously generated result objects, using them to calculate potentials (real, time-dependent ones, not pseudopotentials), and numerically integrating the path of an ion through those potentials. We'll begin with two similar versions, which calculate the trajectories in the Paul trap and whale trap. Then, we'll move onto versions of the code which calculate trajectories in the TOF mass spectrometer, with and without the gate, and a version that simulates the entire loading sequence, starting in the TOF, and ending up trapped in the Paul trap. Finally, we'll finish with a brief explanation of how to use a template trajectory code, so that you can simulate trajectories in whatever fields you want.

7.1 Paul and Whale Trap Trajectories

7.1.1 Paul Trap

Beginning with the Paul trap, the only files you need to have open in matlab to follow along are compilePaulFields.m and trajectoryPaul.m.

trajectoryPaul() is the function that will actually calculate the entire trajectory. It takes the following inputs and outputs:

```
[trajectory,simTimes,amps] = trajectoryPaul(results,y0,T,m)
```

results is a 6 x 1 array of result objects, which is generated by the compilePaulFields() function. Each result object is calculated using the method described in the section on adding and scaling potentials, where one electrode (or in this case, two, since we're assuming the top and bottom electrodes diagonal to each other are connected) is set to 1 V, and every other electrode is set to 0 V, so that we can easily scale the voltages however we want. The order of the result objects goes from the electrodes on the far left, to those on the far right, then with the RF electrodes last:

```
results = [resultFarLeft, resultLeft, resultMiddle, resultRight, resultFarRight, resultRF];
```

You'll notice every version of the trajectory code requires a results array like this. So, before you can run anything else, you need to generate this array of results. Type the following command into the command line:

```
results = compilePaulFields();
```

This should take about 5-10 minutes to generate, so give it some time, and don't be surprised if your computer freeezes for a few minutes. By running this in the command line and not in a script, it stores the results object in the matlab workspace window. This allows us to keep reusing this same object for different runs of the trajectoryPaul() function, instead of having to regenerate it each run from compilePaulFields(), drastically shortening the runtime.

Now that we have our results array, let's take a look at the rest of the inputs and outputs of trajectoryPaul(). y0 is 1 by 6 array specifiying the initial position and velocity of our ion, in units of mm and mm/s (since that's the unit system of our model). If for example we want our ion to start at the position x = 1 mm, y = 2 mm, z = 3 mm, moving in the y direction with an initial velocity of 4 mm/s, then we would write y0 as:

```
y0 = [1, 2, 3, 0, 4, 0]
```

Next, T and m are both just numbers. T is the amount of time you want to run the simulation for, in seconds. m is the mass of the ion in AMU. The mass of a strontium ion is 88 AMU, so unless you're trying to simulate something mass-specific, you'll generally want to use that value. Then, onto the outputs. All 3 outputs, trajectory, simTimes, and amps, are arrays with longest dimension N, where N is the number of timesteps the trajectory is simulated along, depending on T. The timestep h is by default set to h = 1 ns, so generally N = 1 + T / h. simTimes is

a 1 by N array, where for an index i, simTimes(i) is the time corresponding to that index. trajectory is a 6 by N array, in the same format as y0. So, trajectory(i,1) returns the x-position (in mm) of the particle at the time simTimes(i), trajectory(i,2) is the y-position, etc. trajectory(:,1) would return a 1 by N array of the x-position over the entire time period, which you can then plot against simTimes to see the position as a function of time. amps is a length(results) by N array, meaning if results has 6 components (like it does for the Paul trap), then amps is 6 by N. amps describes what the voltage on each electrode is at each point in time. So, for an index i corresponding to a time t = simTimes(i), the voltage on the first electrode in the results array (in this case, the far left DC electrode) at time t is amps(i,1). The voltage on the second electrode (left DC) at t is amps(i,2), and so on for all of the electrodes in results.

So, let's say you want to consider the trajectory of an ion initially at rest close to, but not at, the center of the Paul trap. Then, for our model, the trap center is roughly located at the position (42.9,0,0), with the trap axis running along the x-axis, so we'll place our ion at (42,0.5,1). We'll run the simulation for 10 microseconds, and assume a strontium ion with mass 88. Then we type the following into the command line:

```
[trajectory,simTimes,amps] = trajectoryPaul(results,[42 0.5 1 0 0 0],10e-6,88);
```

The function will print out the current simulation time every 5 microseconds, then will return the trajectory, simTimes, and amps objects into the matlab workspace, which you can then plot and manipulate however you choose. However, it will also generate two graphs: a 3-D graph of the ion trajectory inside of the trap, like in the figure below:

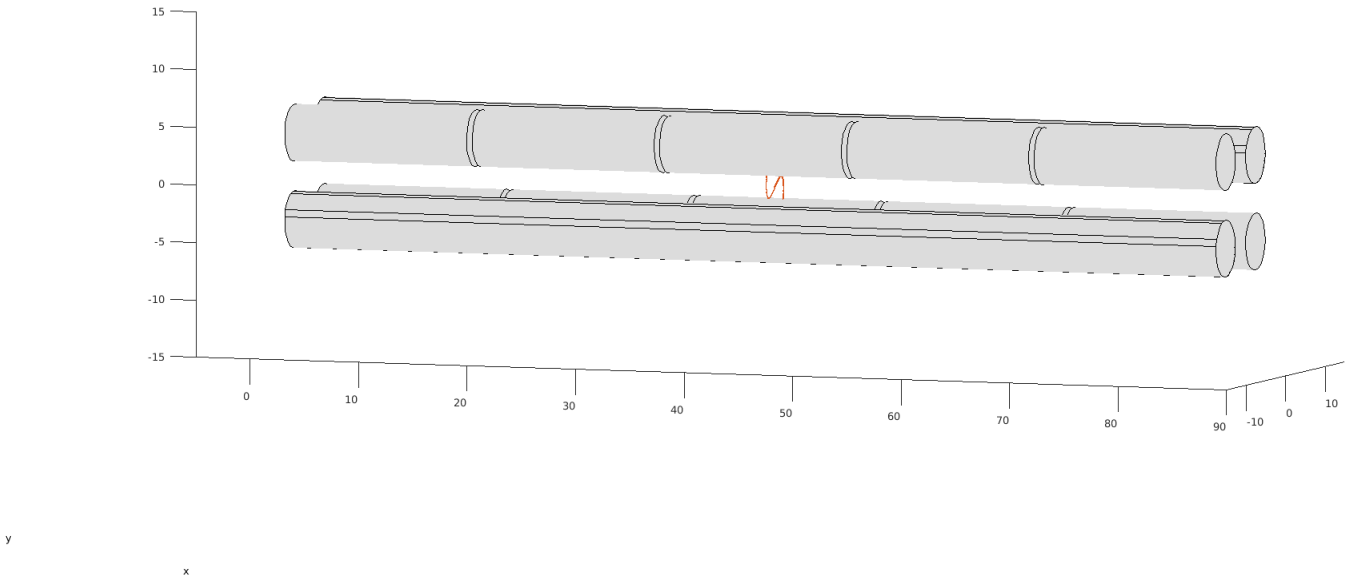


Figure 19: The trajectory of a strontium ion in the paul trap. In the actual figure you'll be able to zoom in and rotate around in 3D to get a better view.

and a plot of the amplitudes on the electrodes as a function of time:

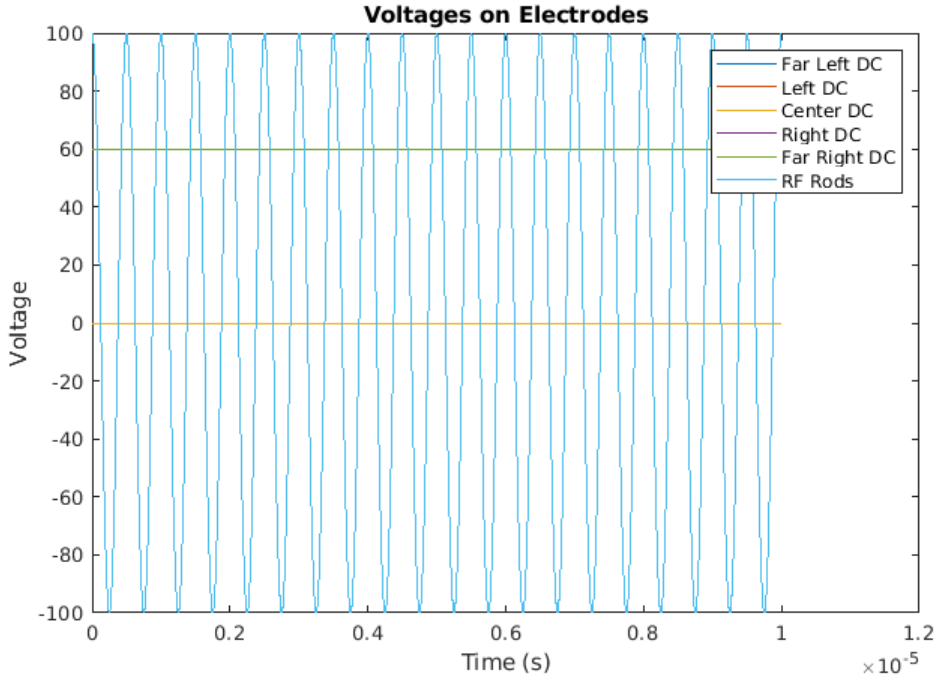


Figure 20: A plot of the voltages on each electrode as a function of time. The right, left, and far left DC voltages are hidden behind the far right DC voltage, since they're all 60 V.

You'll notice we've input nothing into the function about the voltages on the electrodes, or the RF frequency. All of these parameters are tunable, but it would be too messy to make them all inputs. In order to tune these, we need to open up the trajectoryPaul.m script. All of the code that actually numerically integrates the trajectories is located in evolveIon() (or, as we'll see later, evolveIonZDependent()), rkStep(), and ydot(). I'm not going to explain in detail how these work (or this tutorial would be way too long), but I highly recommend opening them up and looking around to see what makes them tick. In brief, evolveIon() creates all of the requisite arrays and loops through all of the times in simTimes, calling rkStep() on each loop. rkStep() takes the various parameters describing the ion and electric fields, and calls ydot() 4 times to perform Runge-Kutta 4 numerical integration. ydot() actually calculates the change in the ion's position due to the electric fields at each point in time. What I'm going to cover here is what you need to input into evolveIon() to simulate whatever you want.

trajectoryPaul.m (and all of the other pre-written trajectory functions) follows this structure: First, it declares what the voltages on each electrode should be as functions of time, then plugs that (and all of the inputs) into evolveIon() to calculate trajectory, simTimes, and amps, and finally plots the trajectory in the 3D model and amps as a function of time. The first part of this sequence is contained in the first few lines:

```
RFfreq = 2e6; % Hz
angFreq = 2 * pi * RFfreq;

syms t;
symAmps(t) = [60, 60, 0, 60, 60, 100*cos(angFreq*t)]; % <== Change Voltages here
```

The first line is where you can input different values of the frequency on the RF rod electrodes. The fourth line is where you can specify the voltage on each electrode as a function of time. It creates a symbolic variable t, then generates symAmps(t), which is a symbolic array of functions, in the same order as results and amps. So, for this configuration, we have constant voltages of 60 V on the far left, left, right, and far right electrodes, a constant voltage of 0 V on the center electrodes, and an AC voltage on the RF rods, with an amplitude of 100 V, and an angular frequency as specified above.

The voltages specified in symAmps(t) don't have to take these values, however; they can be arbitrary functions of time. If you want to say, suddenly switch off the voltages on the left and right DC electrodes, say, 5 microseconds after the beginning of the simulations, then you can use the heaviside/step function, heaviside(t):

```
symAmps(t) = [60, 60*(1-heaviside(t-5e-6)), 0, 60*(1-heaviside(t-5e-6)), 60, 100*cos(angFreq*t)];
```

Alternatively, we can linearly increase the voltage on the center electrode with time, starting at 0 V, and ending at 60 V when we reach our inputted end time T:

```
symAmps(t) = [60, 60, 60*(t/T), 60, 60, 100*cos(angFreq*t)];
```

For most cases, we'll just be dealing with either constant electrodes, switching on and off with the heaviside function, or RF electrodes with sinusoidal voltages, but there's really no limit to how complicated you can make the voltages as functions of time. I highly recommend playing around with symAmps(t) to get a feel for the capabilities, and to get an intuitive understanding of the ion trap.

After we have our symAmps(t) array, we then plug everything into evolveIon():

```
[trajectory,simTimes,amps] = evolveIon(y0,T,m,results,symAmps);
```

Then, the rest of the program just generates the two plots described previously:

```
figure(1);
showModel = createpde();
importGeometry(showModel,'STLs/PaulShow.stl');
pdegplot(showModel);
hold on
plot3(trajectory(:,1),trajectory(:,2),trajectory(:,3));
xlim([-5 90]);
ylim([-15 15]);
zlim([-15 15]);
view(20,5)

figure(2);
plot(simTimes,amps(:,1));
hold on;
plot(simTimes,amps(:,2));
plot(simTimes,amps(:,3));
plot(simTimes,amps(:,4));
plot(simTimes,amps(:,5));
plot(simTimes,amps(:,6));
xlabel('Time (s)');
ylabel('Voltage');
title('Voltages on Electrodes');
legend('Far Left DC','Left DC','Center DC','Right DC','Far Right DC','RF Rods');
```

One thing to notice about the first figure is that the plot is generated using a different .stl file than the paulFields() function. PaulShow.stl is exactly the inverse of the Paul.stl file, where the interior of the geometry is the actual electrodes, instead of being a box with the electrodes cut out of it. This is just so that it's easier to see the trajectories, since we don't have to look through the side of a box. It also makes the plot generate faster, since it doesn't need to be transparent. This allows you to rotate, pan, and zoom with your mouse instead of having to use the view() function, which is much more convenient.

7.1.2 Whale Trap

Fortunately the code for the whale trap is almost exactly the same as for the Paul trap, so I won't go over everything in detail, I'll only point out the differences. First, the whale trap has far fewer parts than the Paul trap, just the top and bottom DC electrodes (not assumed connected), and the top and bottom RF electrodes (assumed connected). So, the results array only has 3 components: resultDCTop, resultDCBottom, and resultRF, in that order. You generate it by typing the following into the command window:

```
results = compileWhaleFields();
```

Then, for the whale trap design, the trap center is at (0,0,0), with the axis of the trap along the z-axis, so, if we want to calculate the trajectory of a strontium ion initially at rest at (0.5,1,1.5) for 10 microseconds, then you type into the command line:

```
[trajectory,simTimes,amps] = trajectoryWhale(results,[0.5 1 1.5 0 0 0],10e-6,88);
```

This will return the same kind of trajectory and simTimes objects as before, and will generate a plot like the one below (as well as another fairly boring voltage plot):

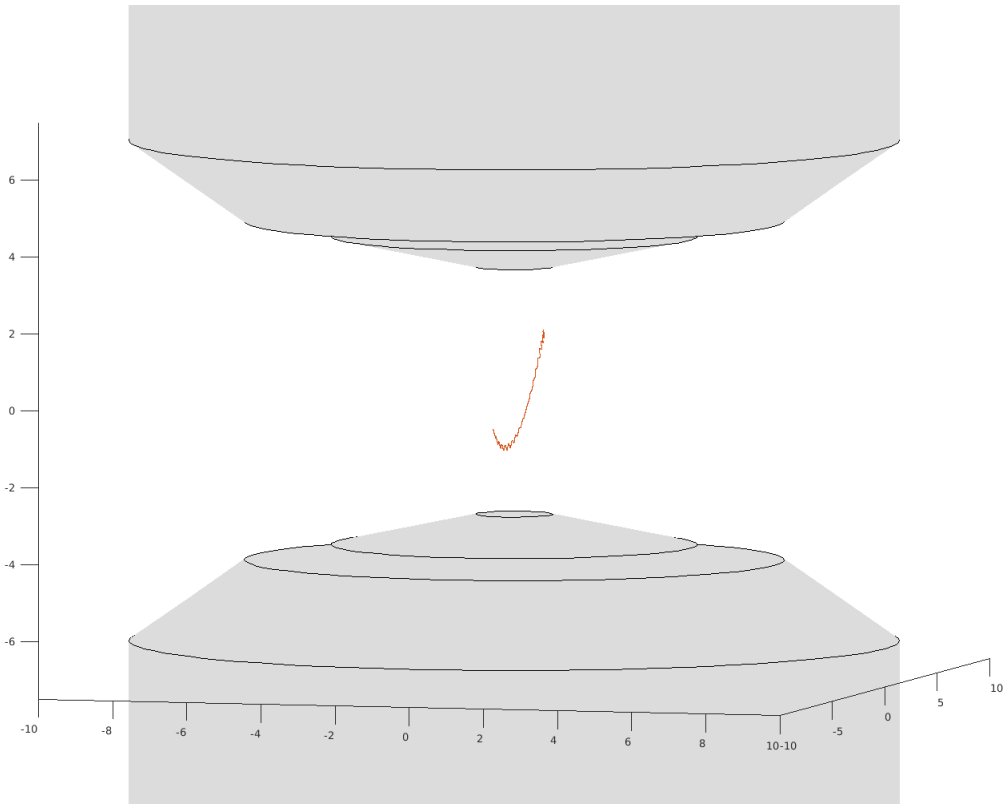


Figure 21: Trajectory of a strontium ion in the whale trap.

If you open up trajectoryWhale.m, you should see that it's almost exactly the same as trajectoryPaul.m, just specified to the whale trap. In order to change the parameters, you should see RFfreq in the same spot as before, also defaulted to 2 MHz. You should also find symAmps in the same spot, except now it only has 3 entries, the voltage on the top DC electrode, that on the bottom electrode, and on the RF rod, in that order (same as the results array). I recommend playing around with these parameters to get an intuitive feel for the whale trap. You should find that, for too high of a DC voltage, somewhere around 5-10 V, you no longer have a stable equilibrium point in the center. You should also feel free to try to add switching with heaviside(t), like in the Paul trap example, to make the DC potential time-dependent.

7.1.3 Paul Trap: Luca's Loading Design

I designed a variation of the Paul trap for loading from the TOF. I've included it here so that you can see how it works. The code is again essentially identical to the Paul trap, so I won't go over it. The only changes were the addition of two DC electrodes on the far left, cut out from the RF rods. These are coupled to the other far left electrodes, so there isn't even any change to the amps parameter, only to the plots. The model is also extended out to the left side, to allow you to simulate ions coming in from along the axis. In order to generate the results array, in the command line, run:

```
results = compilePaulLoadingFields();
```

This, like the other compileFields programs, will likely take a while to run, probably 10-15 minutes, since the model is larger. By default, the amps and switchtimes are set to showcase the loading procedure. I recommend starting out with the following initial conditions: y0 = [-40 0 0.5 1.4e7 0 0]. This corresponds to an ion coming in from the right with a little less than 90 eV of kinetic energy, about what we'd expect coming out of the TOF. To see this trajectory, run:

```
[trajectory,simTimes,amps] = trajectoryPaulLoading(results,[-40 0 0.5 1.4e7 0 0],45e-6,88);
```

Feel free to play around with the initial conditions and other parameters in symAmps(t).

7.2 TOF and Full Loading Trajectories

All versions of the TOF are aligned along the z-axis, unlike the previously mentioned Paul trap, which is aligned along the x-axis. The plate of the acceleration electrode is centered at (0,0,0), so I usually place ions with an initial position of (x,y,0.5), half a mm in front of the acceleration electrode. They'll then be accelerated along the positive z-direction. In this section I'll tell you how to use the code for three versions of the TOF: first, just the TOF with an approximation of the channeltron detector at the end of the drift tube, then one with my design for a mass gate between the end of the drift tube and the detector, and finally a version without the detector at all, where after passing through the gate, the ions are loaded into the Paul trap.

7.2.1 TOF Detector

The code in this version is essentially the same as all previous versions of the trajectory code. First, generate the results array:

```
results = compileTOFDetectorFields();
```

This results array has three components: results = [resultAccel, resultEinzel, resultDetector]. The first is the potential due to the acceleration electrode, the second is from the center "B" electrode in the Einzel lens, for focusing the ion beam, and the third is the channeltron detector at the end of the drift tube. To calculate the trajectory of a strontium ion (mass 88) at (0,0,0.5) initially at rest, for 35 microseconds, run:

```
[trajectory,simTimes,amps] = trajectoryTOFDetector(results,[0 0 0.5 0 0 0],35e-6,88);
```

If you open up the trajectoryTOFDetector.m script, you'll see it has the exact same structure as all other versions, with the following symAmps declaration:

```
accelTime = 3.4e-6;

syms t;
symAmps(t) = [90*heaviside(t-accelTime), 24.3, -2000]; % <== Change Voltages here
```

Here, we can see that the Einzel lens voltage is constant at 24.3 V (which should focus the beam at around the end of the drift tube, you can see this by running the trajectory multiple times with different initial x and y positions), and the detector voltage is at a constant -2000 V. Before accelTime, the acceleration voltage is at 0, and afterwards is at 90 V. You can set the accelTime to 0 if you just want all of the ions to be instantaneously accelerated, but setting a nonzero accelTime allows you to simulate spatial refocusing, where initially at rest ions are accelerated more than ions with an initial velocity, so that all of the ions of a certain mass meet up at some point in space at the exact same time, no matter the initial velocity. Here, an accelTime of 3.4 microseconds allows us to spatially refocus strontium ions so that they all reach the detector at the exact same time. If we consider ions initially at rest in blue, versus ions with an initial z velocity of 1,375 m/s in red (which we believe to be a reasonable approximate maximum speed for the ablated ions), without any spatial refocusing, we get the following:

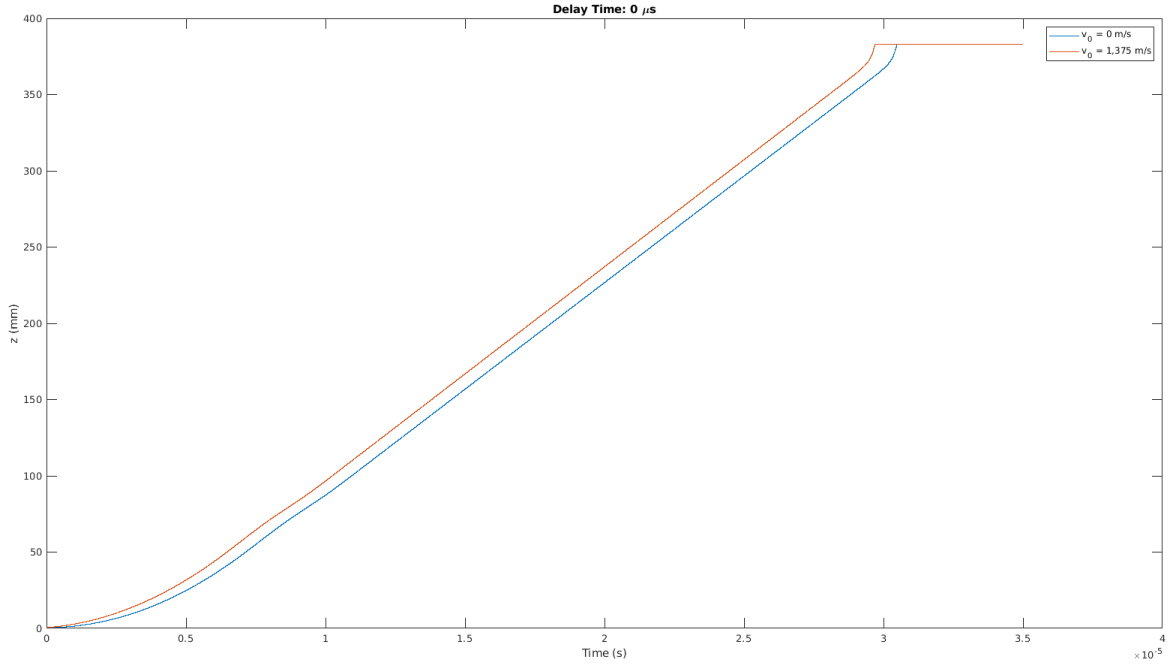


Figure 22: Ions with different initially velocities without any spatial refocusing.

Here we can see that the initially fast ions reach the detector before the initially at rest ions, since they all start out at the same initial point in the acceleration potential. If we then consider a 5 microsecond delay, we get the following:

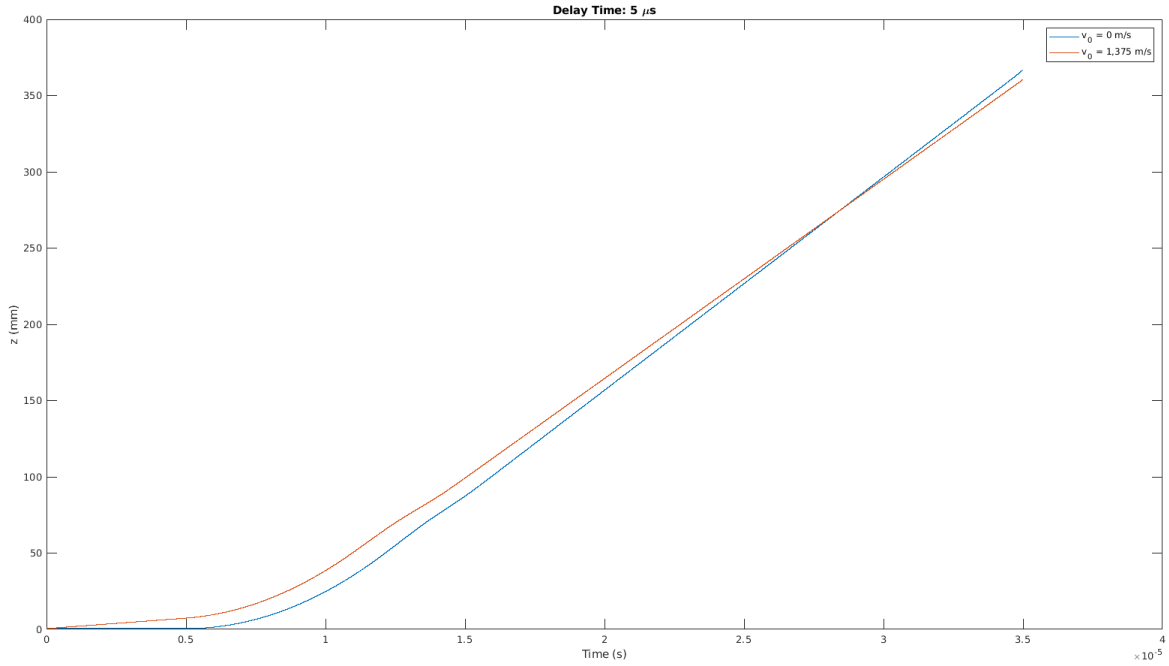


Figure 23: Ions with different initially velocities with 5 microsecond delay in turning on the acceleration electrode.

Here we can see that we've set the delay to be too long: the initially at rest ions are accelerated much more than the initially high-speed ions, so they catch up and pass the initially moving ions before either of them reach the detector at the end of the drift tube. After some trial and error, we see that a delay of 3.4 microseconds is about right for spatially refocusing the ions at the detector:

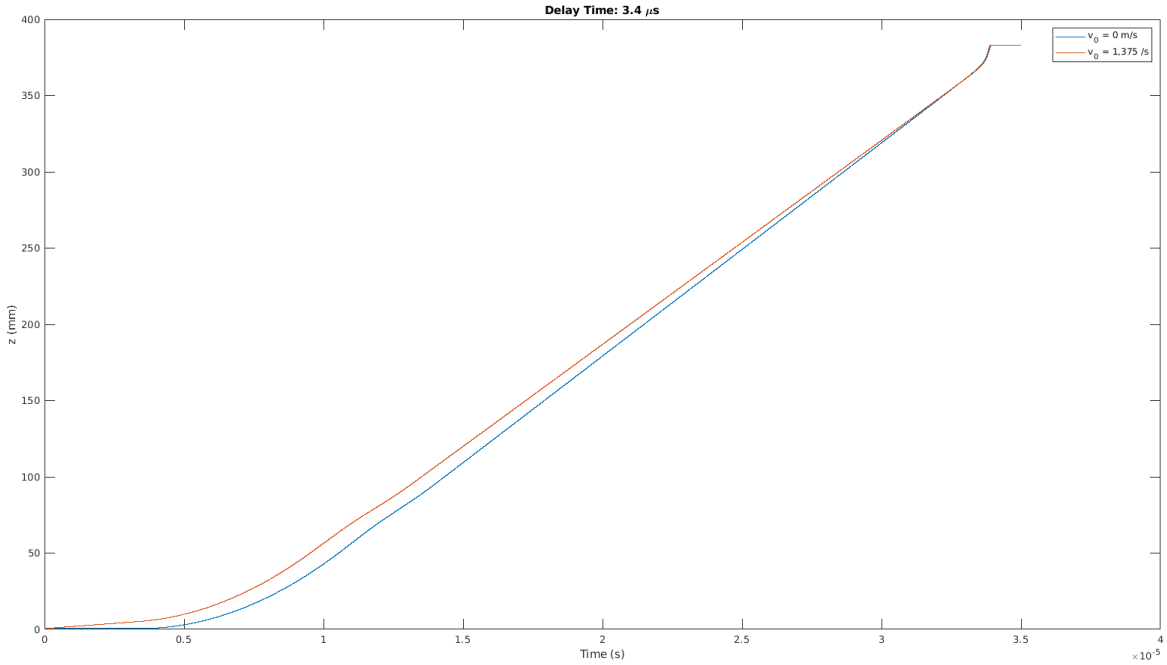


Figure 24: Ions with different initially velocities with 3.4 microsecond delay in turning on the acceleration electrode, ideal for spatially refocusing the ions at the detector.

You can see these results for yourself by playing around with the accelTime, then plotting the z trajectory (trajectory(:,3)) for ions with different initial z velocities as functions of simTimes.

You can also see how changing the voltage on the Einzel lens changes the focal point of the ions with different initial x and y positions. For example, here are the trajectories of ions with different initial positions with 0 V on the Einzel lens:

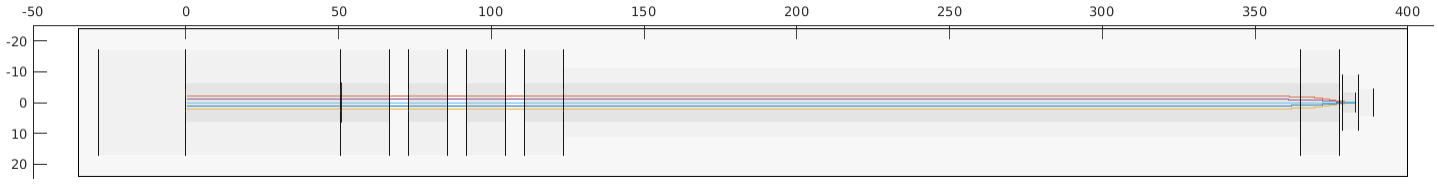


Figure 25: Trajectories of ions with 0 V on the Einzel lens. There is no focusing at all.

Then, if we instead place 40 V on the Einzel lens, we can see that the ions are focused to a point inside of the drift tube:

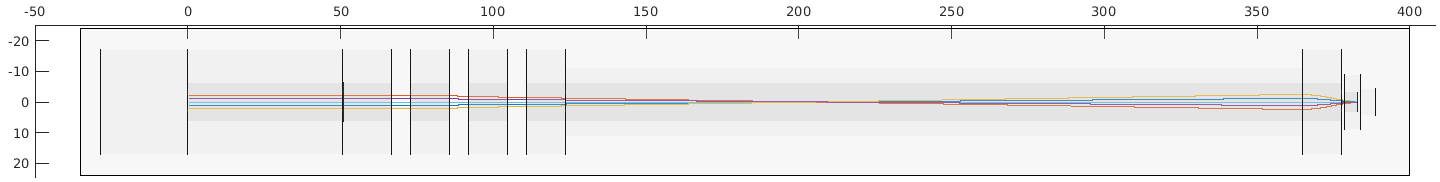


Figure 26: Trajectories of ions with 40 V on the Einzel lens. The ions are focused to a point inside of the drift tube.

Before we move onto the other two versions of the TOF, there are two more small details I want to make sure I've covered, in case you go poking around in how the fields are compiled for the TOF. You can skip this part for now if you just want to learn how to use the code. First, I mentioned before how the Paul trap and Whale trap trajectory functions plot the trajectories in a different model with a different .stl file than that which is used to actually calculate the fields, to make it easier to see the trajectories. This is true for the TOF designs as well, to an even more extreme degree. In the above figures, you can clearly see each of the individual electrodes, all inside a big box containing everything. This is just for show, however. Since we only care about the electric fields inside of the TOF, not in the space outside, the model we use to calculate the potential cuts out all of the outside space, and looks more like this:

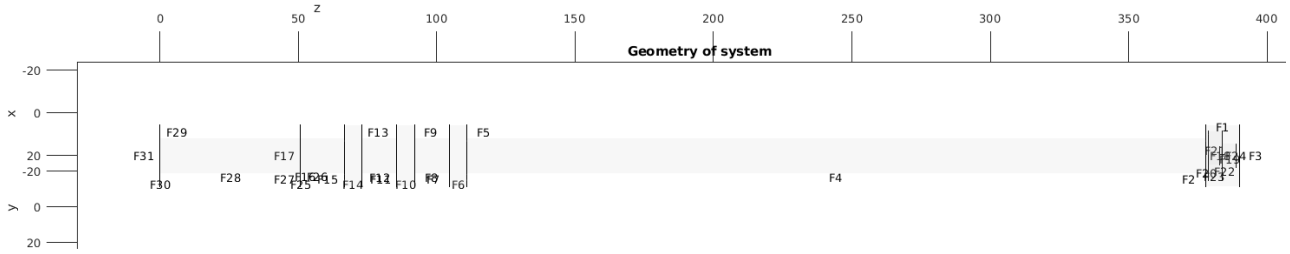


Figure 27: The actual model used to calculate the TOF fields.

Here, the relevant surfaces are all the insides of electrodes, so it's a little harder to visualize what's what when you're figuring out what surfaces correspond to which face labels to apply boundary conditions. So, if you're making your own modifications to these TOF designs, keep this in mind.

The other detail I wanted to cover concerns a specific part of this design, between the insulating plastic and the Einzel lens. In the actual TOF, the plan is to place a grounded piece of mesh here, to try to make the electric field as much like a parallel plate capacitor as possible. In the simulations, there isn't really a good way of properly simulating a fine mesh, so instead I've placed a very thin (0.01 mm) solid surface in the way. It's actually so thin that it's pretty difficult to see unless you're very zoomed in. This means that the ions are actually running into a solid surface, which *should* cause them to stop in place, like they do anytime they run into the wall of the model. I won't go into too much detail of my code, but essentially how it determines whether the ion has run into a wall is by counting the number of times the evaluateGradient function returns NaN for the electric field in a row, as it would outside of the model bounds. If the ion hits a wall, it returns NaN, and increases a counter called totalCollisions. It then just assumes 0 E field, steps forward in time, and tries again. If it keeps getting NaN 5 times in a row, then it just freezes the ion in place for the rest of the simulation. There are 2 reasons this threshold is 5 times, instead of just freezing the first time it hits a NaN. First, since numerical simulations aren't always perfect, every once in a while, the ion might run into a lone point that should have a perfectly fine potential, but for some reason or another, unphysically throws NaN. This gives the simulation some leeway in this case, making sure the ion only freezes if it's actually, definitely in a wall, and not just hitting one of these anomalous points. The second reason is that it allows us to simulate this very fine mesh: since the ions are moving at high speed and the "mesh" surface is extremely thin, the ions won't spend more than 5 timesteps inside of it, so they can still experience grounding of the mesh while being able to pass right through it. I just wanted to make sure I explained this in case someone did some digging and came across it, and got confused as to why the ions are able to pass through a solid surface.

7.2.2 TOF Gate Detector

This version of the code includes the mass gate between the end of the drift tube and the detector. It also utilizes one more useful trick for calculating fields in models which are particularly long in some dimension. In order to just

run the code, first, generate the results array:

```
results = compileTOFGateDetectorFields();
```

Here, the results array has five components: results = [resultAccel, resultEinzel, resultDetector, resultGateTop, resultGateBottom]. The first three are the same as the previous model, and the last two correspond to the voltages on the two parallel plate electrodes in the gate. Then, we calculate the trajectory same as always:

```
[trajectory,simTimes,amps] = trajectoryTOFGateDetector(results,[0 0 0.5 0 0 0],40e-6,88);
```

If you open up trajectoryTOFGateDetector.m, you'll notice it's similar, but not exactly the same as the previous functions. Here's the relevant parts of it:

```
accelTime = 3.4e-6;
gateOffTime = 34.325e-6;
gateOnTime = 34.85e-6;

syms t;
symAmpsAccel(t) = [90*heaviside(t-accelTime), 24.3, nan, nan, nan];
symAmpsGateDet(t) = [nan, nan, -2000, 20*(1 - heaviside(t-gateOffTime) + heaviside(t-gateOnTime)),
                     -20*(1 - heaviside(t-gateOffTime) + heaviside(t-gateOnTime))];
symAmpsCell = {symAmpsAccel,symAmpsGateDet};
zOffsetArray = [0 365.5];
zSwitchArray = [367];
[trajectory,simTimes,amps] = evolveIonZDependent(y0,T,m,results,symAmpsCell,zOffsetArray,zSwitchArray);
```

Instead of just one version of symAmps, we have two, symAmpsAccel and symAmpsGateDet, placed inside a cell called symAmpsCell. We also have two extra arrays, zOffsetArray and zSwitchArray, and all three are plugged into a different version of the evolveIon() function, evolveIonZDependent().

The reason for this difference is because the first two result components and the last three result components are actually generated from **different models**. You can see this if you go into the compileTOFGateDetectorFields.m script, the acceleration electrode and Einzel lens fields are generated from the file TOFaccel.stl, while the detector and gate fields are generated from TOFGateDetectorOffset365pt5.stl. The former .stl file contains everything from the acceleration electrode up until the end of the drift tube, and the latter file contains everything from the end of the drift tube up until the detector, including the gate. So, when we send an ion down the TOF, from when it's in the acceleration region to until it reaches the end of the drift tube, we need to apply the fields from the acceleration electrode and Einzel lens to it, contained in the first two components of results. After it passes through the end of the drift tube, we need to apply the fields from the detector and two gate electrodes, in the last three components of results. So, our symAmps array needs to depend on the ion's z position: before it passes a certain threshold, it uses symAmpsAccel, and after it passes that threshold, it uses symAmpsGateDet.

As you can probably guess from the name, the position of the TOFGateDetectorOffset365pt5.stl file is actually not where it should be relative to the end of the former file; it's offset by 365.5 mm, so that instead of running from the z locations z = 365.5 to z = 405.5, it runs from z = 0 to z = 44. When we want to calculate the electric field of an ion as it passes through this region, we thus need to subtract 365.5 from its z position, then calculate the electric field from this model, and then add it back to the z position for when we log the trajectory. This information, the offset of each model, is contained in the array zOffsetArray. The first file, corresponding to symAmpsAccel, has no offset, and the second file, corresponding to symAmpsGateDet, has an offset of 365.5 mm, so zOffsetArray = [0, 365.5]. At some point, we need to switch from the first symAmp to the second. This information is contained in zSwitchArray, which, for our models, should be zSwitchArray = [367], so that when z < 367 mm, it uses symAmpsAccel, and when z > 367 mm, it uses symAmpsGateDet.

So, to recap, I'll describe a more general case of simulating a trajectory through multiple files down the z-axis. Let's say we have 4 files total, and 8 components in results:

```
results = [resultFile1-1, resultFile2-1, resultFile2-2, resultFile3-1, resultFile3-2, resultFile4-1]
```

Then, we need a version of symAmps for each file, each the length of the total results array, but with NaNs in every spot except the results corresponding to that file.[1]

[1] I know, this probably isn't the most elegant way to do this, but this is the solution I came up with in the limited time I have left working in the Patterson group.

```
syms t;
symAmpsFile1(t) = [Voltage1-1(t), nan, nan, nan, nan, nan];
symAmpsFile2(t) = [nan, Voltage2-1(t), Voltage2-2(t), nan, nan, nan];
symAmpsFile3(t) = [nan, nan, nan, Voltage3-1(t), Voltage3-2(t), nan];
symAmpsFile4(t) = [nan, nan, nan, nan, nan, Voltage4-1(t)];
```

We then have to compile all of these into a cell, in order:

```
symAmpsCell = {symAmpsFile1,symAmpsFile2,symAmpsFile3,symAmpsFile4};
```

Then, assuming the first file has 0 offset, we need to set our zOffsetArray as follows:

```
zOffsetArray = [0, zOffsetFile2, zOffsetFile3, zOffsetFile4];
```

Finally, we need to then specify the z-coordinates that the potential switches from the first to the second model, the second to the third, and the third to the fourth.

```
zSwitchArray = [zSwitchFile1to2, zSwitchFile2to3, zSwitchFile3to4];
```

Then, we can finally just call the modified z-dependent evolve function:

```
[trajectory,simTimes,amps] = evolveIonZDependent(y0,T,m,results,symAmpsCell,zOffsetArray,zSwitchArray);
```

You'll need to follow this method any time you want to simulate trajectories where the model has been split up into multiple files. But why are we doing this in the first place, instead of just calculating everything from one big .stl file, like in every other version of the code? There are two main reasons for splitting up the model into two files and offsetting one of them. The first is because the first file, TOFaccel.stl, contains mostly larger parts, so we can set a larger value for hmax and still get a good result, while the second file, TOFGateDetectorOffset365pt5.stl, contains a lot more finer parts, so we need a smaller value of hmax to get an accurate result. By spliting the files in two, we can have different values of hmax for each section, allowing us to more efficiently generate the results by only having a very fine mesh exactly where we need it.

The second reason for this splitting and offsetting is because of a quirk of the matlab PDE toolbox which I don't really understand. For some reason, whenever you have a model which is too far away from the origin (0,0,0), no matter how fine you set the mesh, the calculated potential will have huge chunks of NaNs all over the place, where there clearly shouldn't be. If you instead bring the exact same model closer to the origin, then all of those NaNs disappear, and you get a nice, smooth solution. This is why the part of the model including the gate and detector needed to be offset back to the origin, when I tried calculating the potential with it at z = 365.5, the solution was full of NaNs. **So, whenever you're dealing with models which are very long, with electrodes very far away from the origin, you must split them up and offset them as we have here.** However, for most cases where your model doesn't extend too far out from the origin, you don't need to worry about splitting it up, and you can just use the previous versions of the trajectory code.

I recommend playing around with the timing and voltages on the gate. You can do this just like in the previous versions, by setting the components of the symAmps arrays to whatever arbitrary functions of time you want. Make sure you're altering the correct arrays though, don't try to set the acceleration voltage in symAmpsGateDet, for example. If you put nonzero values for results that aren't in their corresponding symAmps array, it will count the ion as being outside of the model, interpreting it as having collided with a wall, and freezing in place.

The default values on the gate should allow mass 88 ions (strontium) to pass through the gate and into the detector, while any other masses should be deflected out of the way, like below:

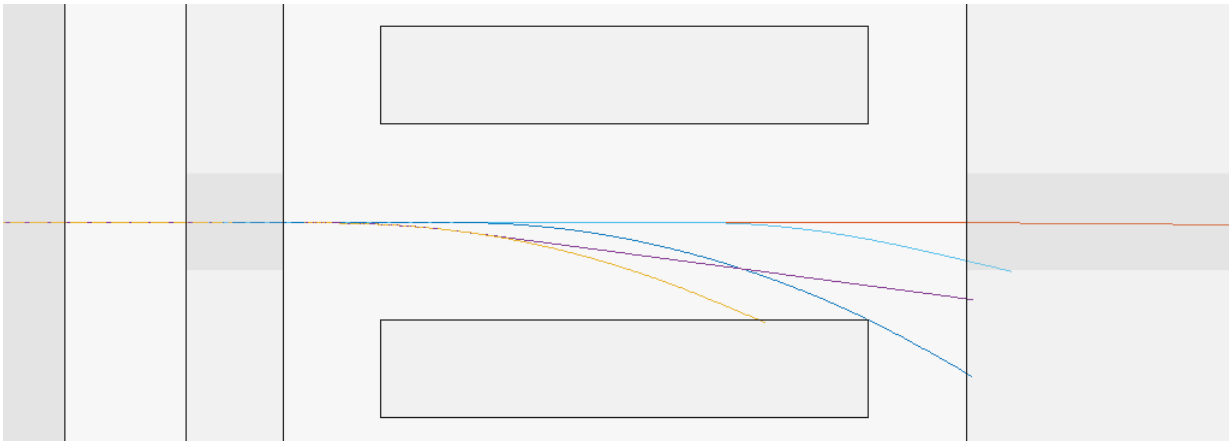


Figure 28: Ions of various masses passing through the gate. The red trajectory, corresponding to an 88 amu ion, passes through unaltered, while the yellow, purple, light blue, and dark blue trajectories, corresponding to 86, 87, 89, and 90 amu respectively, are all deflected out of the way.

7.2.3 Full Loading Trajectory

Finally, we reach the last example of trajectory code here, which simulates the entire trajectory of an ion through the loading process, starting in the TOF, passing through the gate, and being loaded into the Paul trap. I won't be introducing any new concepts here, this code essentially just combines everything we've learned so far. To generate the results array, run the following:

```
results = compileFullLoadingFields();
```

Then to calculate a trajectory, run:

```
[trajectory,simTimes] = trajectoryFullLoading(results,[0 0 0.5 0 0 0],60e-6,88);
```

For the default values of all of the voltages and switch times, this should demonstrate loading a strontium ion into the trap, while excluding ions of any other mass. Try it out for yourself, vary the mass, initial velocity, and initial x and y positions.

If you want to vary the parameters, just open up trajectoryFullLoading.m. It's structured essentially the same as the previous code, but now, the model is split up among three different files: from the acceleration electrode to the end of the drift tube, from the end of the drift tube to the end of the gate, and from the end of the gate to the Paul trap. The model of the gate, just like in the previous example, is offset by 365.5 mm from the origin, and we set the switch point to be 367. The Paul model's origin is offset by 446 mm, but it extends 45 mm into the negative z direction, so its switch point is at 401.

7.3 Trajectory Template

If you want to simulate the trajectory in your own custom model, you'll first need to write your own compileFields() function. I can't provide you with a template of this, since it will depend a lot on the details of your model, but here's what is should do: For every electrode in the model with a potentially nonzero voltage, it needs to generate a result object where that electrode has a voltage of 1 V, and every other electrode is at 0 V. Then, it needs to return an array of these result objects. So, if you have N electrodes with nonzero voltages, then your compileFields() function should return the following:

```
results = [resultElectrode1, ..., resultElectrodeN];
```

Then, all you need to do is make a couple of minor changes to the function trajectoryTemplate.m. First, in the while loop, you need to specify the voltages on each of the electrodes:

```
syms t;
symAmps(t) = [voltageElectrode1(t), voltageElectrode2(t), . . . voltageElectrodeN(t)];
```

Each voltage can be whatever arbitrary function of time you want.

Then, you just need to change the name of the .stl file at the bottom of the script being called in importGeometry() from 'TemplateShow.stl' to whatever the name of the .stl file your model uses. This can be the original file you calculated the fields from, or it can be an altered version that you created which is a bit better for visualization, like I've done with my previous code. Finally, at the bottom, you can use various functions to set whatever default viewing parameters you want. You should then be free to calculate trajectories in whatever models you want.

I believe this covers just about everything I've learned using these PDE simulations. I hope you find them useful, and if you ever have any questions, please feel free to email me at lscharrer@ucsb.edu. Good luck!