PyAAO Guide

This document will be a useful resource for getting started with running the PyAAO code.

Theory

PyAAO runs the adjoint equations optimization technique. Refer to the several other pdf documents under */docs* for more information. The PhysRev paper is probably the most detailed, but also the most complex, document you can look at. The “Envelope equations\_1\_14\_21.pdf” goes through most of the math used in the code. Feel free to also contact Levon Dovlatyan or Thomas Antonsen for any questions.

The following will reference equations from the PhysRevAccelBeams paper under */docs*.

There are essentially 11 variables we are solving for in the adjoint formation of the envelope equations. In the paper these are Eq. 9, 10, 11, 12, and 7:

Text, letter

Description automatically generatedText

Description automatically generated with low confidence

These represent various moments of the beam. We solve for these variables by numerically solving the moment equations in Eq. 13:

Text, letter

Description automatically generated

If we look in the code, most of this is done under *momentSolver/MomentSolver.py*. In that you will see functions like:

Graphical user interface, text, application

Description automatically generated

Which calculates the initial values for the 11 variables we are going to solve for. We then call the “Run” function to go ahead and numerical solve Eq. 13:

Graphical user interface, text, application, email

Description automatically generated

The adjoint technique depends on having some kind of “Figure of Merit” to solve for. The goal is to perturb different variables in our equations to find the minimal value of the “Figure of Merit”. In our case the “Figure of Merit” (FoM) is an equation that tells us our beam has achieved “roundness” at the end of our lattice. The purpose of the Flat-To-Round FTR experiment is to see if we can turn a flat beam into a round beam. Thus, our FoM is a measure of how “round” our beam is. If we can minimize the FoM, we can maximize the “roundness” of the beam.

There have been various versions of the FoM, unless it is something we need to change, it doesn’t need to be messed with. You can see the FoM in the paper from Eq. 40:

Text, letter

Description automatically generated

As well as another version later in Eq. 46:

A screenshot of a computer

Description automatically generated with low confidence

Text, letter

Description automatically generated

In the code you can look at functions such as:

Graphical user interface, text

Description automatically generated

that go ahead and calculate the FoM we are using.

Given a FoM, we can then solve for the adjoint equations, in the paper these are Eq. 28:

Text, letter

Description automatically generated

You can find the function for this in the MomentSolver.py:

Text

Description automatically generated

Once we have numerically solved our moment equations (Eq. 13) and our adjoint equations (Eq. 28), we then have all the information we need to measure the change that would occur in our FoM if we happen to perturb or change a magnet or other parameter in our lattice. We can then use some optimization technique to perturb all these parameters (we will talk about it later) and minimize our FoM.

The following is a general flow chart of the math that gets calculated:

Graphical user interface, text, application

Description automatically generated

Think of each step as a function in our code that calculates the math.

Optimization

Given the adjoint equations we talked about above, we can use these to optimize our Flat-to-Round (FTR) system.

The parameters we will perturb/adjust in order to achieve our Figure of Merit (FoM) are the following:

Quadrupole Magnet #1 – start position, strength, angle

Quadrupole Magnet #2 – start position, strength, angle

Quadrupole Magnet #3 – start position, strength, angle

Solenoid Magnet – start position, strength

This leaves us with 11 parameters that we will feed into our optimization and have it output the best values for them in order to achieve our FoM. Remember the FoM in this case is achieving a “round” beam within the solenoid magnet. The optimization technique being used is a simple gradient descent.

Let’s go through the *exampleOptRun.py* file as this runs a full optimization.

Add the top of the file you will see a few functions defined to help with the optimization. Functions like *plotEnvelope, setRestrictions,* etc…

Next we see an import of the magnet parameters that grabs magnet profiles defined in *systems/bbc/magnetParameters.py*. If we look in that file you will see us creating the three quadrupole magnets + the solenoid with the appropriate settings. The field profiles for these were given to us by Santiago Bernal and Dave Sutter. If we need to magnet settings, it is best to contact them. The profiles are set based off the following information given by Santiago & Dave:

Chart, line chart

Description automatically generated

Back to the *exampleOptRun.py.* After the first import, we have a second import that grabs optimization parameters from *systems/bbcFTR/optParameters.py* This file generates our initial values for the 11 parameters we are going to perturb/optimize.

If you look in the file you will see two helper functions that the top called *getParamArray* and *getParamObj*:

Graphical user interface, text, application

Description automatically generated

In the code it is useful to store this 11 parameter information as a dictionary object. However, when calculating things in the adjoint optimizations, it can be useful to keep all 11 parameters as just an array since we do lots of matrix/vector multiplication. Hence we have these two helper functions that convert between the two formats.

Further down in the file you see the defining of the 11 parameters, and constructing the parameter object:

Text

Description automatically generated with medium confidence

This is the information we are loading into our optimization as the starting initial values for our parameters.

Back to the exampleOptRun.py

After the import of the magnet/lattice parameters and the import of the 11 optimization parameters, we look further down in the code.

We grab some information like beam energy, beam current, lattice length (zInterval) and create our MomentSolver class object.

We update our lattice based on the imported 11 parameter values since that will be our initial conditions:

Text

Description automatically generated

Next we run the moment equations and run the adjoint equations. We go ahead and plot the results to see what our initial solution looks like. What does our beam look like as it passes through these magnets with the initial 11 parameter values we used:

Text

Description automatically generated

The plot will look something like this:

Chart, line chart

Description automatically generated

The red lines represent the quadrupole magnet fields, the green line is the solenoid fields. The blue and orange lines are the size of the beam in the X and Y directions. We can see the beam starts flat, because at z = 0 our X beam size is big (0.5e-5m), but our Y beam size is very close to 0.0. The goal in the optimization is to get the beam size in X and Y to be equal, to get the blue and orange lines to be equal, INSIDE the solenoid. So the optimization perturbs our 11 parameters related to the magnets to try and get the beam to be round within the solenoid.

Next we grab our initial FoM value and define some initial arrays:

Text

Description automatically generated

Afterwards we take our first step in our gradient descent:

Graphical user interface, text, application

Description automatically generated

If we look up gradient descent algorithm online, you will see something like this:



In the code example above, the red box highlights the equivalent of that in our code. The other lines in the code is just updating matrices, and then running the moment equations. That is once we take a gradient step, our 11 parameters have been changed. We then update our lattice and magnet parameters with the new 11 parameter values and rerun the moment equations to see what our FoM looks like. If our FoM decreased after the gradient step (with our new 11 paramter values), then our beam became a little bit more “round”. We repeat and continue takings steps until our FoM can no longer decrease.

After the initial gradient step, there is some logic in there that repeats the steps in order to find a good value for gamma:

Text

Description automatically generated

Once this is done, we enter the main loop of the optimization. As long as the FoM after we take a gradient step is less than the previous step we took, keep going and takings steps:

Text

Description automatically generated with medium confidence

Eventually we take enough steps where our FoM stops decreasing, or even increases! When this happens we enter the next logic in the code, which is to recompute the adjoint equations with the last set of good values for the 11 parameters:

Text, letter

Description automatically generated

And then we go back to taking gradient steps. Thus, the general process is we take gradient steps as long as our FoM is decreasing. Once the FoM stops decreasing, we recompute our adjoint equations and repeat. If our FoM is not decreasing and recomputing the adjoint equations doesn’t help, there is some logic to try and go back and update the value for “gamma” in the gradient descent:

Text

Description automatically generated

This optimization will then keep running until some stop conditions are met:

A picture containing text

Description automatically generated

If the FoM is smaller than 1e-14, we can stop. Or if we hit 1000+ steps/iterations we can stop. Or, the most commonly used condition, if we just hit ctrl-c to stop the optimization.

After stopping the optimization, we plot the moment solution with the last best set of values for the 11 parameters:

Text

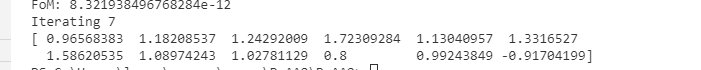
Description automatically generated

Chart, line chart

Description automatically generated

Note the beam size in X & Y is much closer to being equal in the solenoid, telling us the beam is much more “round”.

When we stop the optimization, we can grab the last set of best values for the 11 paramtere:



And we can go back to our initial import of magnet parameters, in the *optParameters.py* and replace our initial values with these no values. That way the next optimization will start from these already pretty good 11 parameter values we just got running the optimization.

Optimization Restrictions