# SECAR Optimizers Manual V.3

Sara Ayoub

January 2021

## Contents

# 1 Pre-experiment Checklist

Make sure you are ssh-ed into the `e18514` account (or wherever the codes now are). Here are some things to check prepare before beamtime.

## 1.1 Update Background Images

For all viewers (VD_D1515 to VD_D1879), do the following with no beam on the viewer.

1. Insert the viewer into position along the beam's path. You can turn on the light to make sure it is moving and in place.
2. Turn the light off and open the viewer's "Settings" window in CSS.
3. Set the desired camera settings (gain, exposure time, attentuation). If you don't have beam yet, keep the settings as is (saved settings from last experiment).
4. Give the new background image a name (e.g. D***_bg_date.tiff). Make sure the name has the date and time so it is easily found later if the analysis needs to be reproduced.
5. Set the "Path" to `/mnt/daqtesting/secar_camera/new_captures/lighton_and_background/`.
6. Click "Save" to take the image.
7. Set the "Path" back again to `/mnt/daqtesting/secar_camera/new_captures/`.
8. Go to `/user/e18514/Documents/viewer-image-analysis/src/`.
9. In `settings.py`, find the dictionary `bg_im`, and update the corresponding viewer's image name to the new name.

**Note:** Once beam is on the viewer, if there is a need to adjust these settings to see the beam better, a new background image needs to be taken following the steps above.

## 1.2 Check Codes Run

A quick check needs to be done to make sure all codes run with no issues. Here are the steps to take to check them.

1. Before running any code, load Python 3.7 (see Section 3) by running `module load anaconda/python3.7`.
2. Go to `/user/e18514/Documents/tuneoptimizer/`.
3. Open `beam-optimizer.py`.
4. Under the imports, set `sandbox = 'y'`.
5. In the command line, run `python beam_optimizer.py`. If the code runs with no issues (you should see GP iterations printing), halt the code by hitting Ctrl+C.
6. Set `sandbox = 'n'`.

7. Open `quad-optimizer.py`, and repeat steps 4 - 6 with this code.
8. Finally you can check that the viewer analysis works for any viewer by going to `/user/e18514/Documents/viewer-image-analysis/`
   and running for any raw tiff image for example with D1638:
   `python im_analysis.py /mnt/daqtesting/secar_camera/new_captures/D1638_q1q2half_06-19_00\:12.094799_000.tiff d1638`.
9. If code runs with no error, the last print statement should be the image name.

**Note:** If you get a "SyntaxError" from print statements at any point, it is likely you forgot load the correct Python version after disconnecting.

### 1.3 Data Storage

Since the ML codes outputs several files (txt and png) at every iteration, occasionally you will run out of space if they are being saved to the experimental account. Check the available space in the e18514 experimental account (5 GB capacity):
`df -h .`
and if it near full, move some items to permanent storage (2 TB capacity):
`/mnt/events/e18514/`.

Currently the viewer image analysis is always saving the PNGs to
`/mnt/events/e18514/tuning/images_optimizer/`.

## 2 General Tuning Steps

Make sure you are **ssh**-ed into the `e18514` account. Here you can find a step-by-step summary of how to run each code in
`/user/e18514/Documents/tuneoptimizer/`.

### 2.1 Beam Optimizer: ReA Steerers

1. Load Python 3.7 by running the `module load anaconda/python3.7` command.
2. Open the version of `beam-optimizer.py` that you want to use (different files use different quads).
3. Select the steerers to optimize in `magnet_list`.
4. Set the `viewer` to be used.
5. Set the desired phase space for the steerers in `spaceArray`.
6. In the `while` loop, set the quads and the different quad tunes that the optimizer will use to get the steering distance (there are different files already made for each section).
7. Still in the `while` loop, set the steering distance to be either the steering in x and y combined or steering in one direction (if using only vertical steerers for example). At the as-

signment of `distance`, set `separateXY = True` and take the first index of the returned value ([0]) for x, or the second ([1]) for y. For combined, set it to `separateXY = False` and take the first index.

8. In the Gaussian process section after the GP regression is defined, set the lengthscale prior `kern.lengthscale.set_prior(GPy.priors.Gaussian(10,2))` if at FP1, or down to (5,2) at more sensitive downstream location.
9. Set the noise prior `Gaussian_noise.variance.set_prior` to (1,0.5).
10. Finally, set the `exploration_weight` in the acquisition function initialization to 0.5 (up to 3 if more exploration is needed).
11. Run the code: `python beam-optimizer.py` (or whichever version you are using).

## 2.2 Quad Optimizer

1. Open `quad-optimizer.py`.
2. Set the `viewer` to be used.
3. Select the quads/hex/oct to optimize in `magnet_list`.
4. In the `while` loop, set the desired phase space for the steerers in `q_ps`.
5. In the Gaussian process section after the GP regression is defined, set the lengthscale prior `kern.lengthscale.set_prior(GPy.priors.Gaussian(2,1))`.
6. Set the noise prior `Gaussian_noise.variance.set_prior` to (1,0.5) or (3,3) depending on the noise observed.
7. Finally, set the `exploration_weight` in the acquisition function initialization to 0.5 (up to 3 if more exploration is needed).
8. Run the code: `python quad-optimizer.py`.

## 2.3 Dipole Scans

This describes how to run `dipole-nmr-tuner.py` for B1 and B2.

1. Set B1 and B2 to a high current so the beam is on the right side of the viewer. No need to cycle or match (but keep them at close values).
2. Open the file and select the `viewer` where the scan is being done.
3. To change the stepsize, change the variable `dI`.
4. Run: `python dipole-nmr-tuner.py`.

There are several versions of `dipole-hall-tuner.py` and `b*-b*-custom-tune.py`. These are generally ready to run similarly as above as the modifications for viewer/quads have already been made in the past. For all dipoles other than B1/B2, the scan is done by

varying one dipole in large steps, and the other changes in little steps while the first is held constant. These steps can be changed at the top of each file.

To plot the results of the dipole scans: `python fit_dipole_scan.py filename`.

# 3 Python Version and Packages

These codes were developed in Python 3.7 and can be run on any NSCL machine that has that Python version installed. The module can be loaded using Anaconda with the command

```
module load anaconda/python3.7
```

after which any of these Python programs can be called with the `python` command. The programs described have some dependencies on packages that may need to be manually installed (or installed by IT). The built-in modules from the Python standard library are not listed, and the dependencies of each package are assumed to be taken care of during installation. To run the codes, the following packages are needed:

```
GPy==1.9.9
GPyOpt==1.2.6
matplotlib==2.2.4
numpy==1.16.5
pyepics==3.3.3
scipy==1.2.2
skimage==0.17.2
```

These were installed for the `e18514` user for the commissioning, but may need to be installed for other users if the codes are being run through another account. During the commissioning, the experimental account `e18514` was used and the steps explained in this manual assume the same account is being used.

Currently a clone of the codes exists in the `/user/e18514/Documents/` folder and the origin lives on the FRIB Stash git repository where changes are frequently committed.

# 4 Viewer Analysis Code

This code serves to analyze the TIFF files captured by the viewer cameras along the SECAR beamline. It takes as input the desired TIFF file and the viewer location, and returns the location of the beam center, the width of the beam spot, and a visualization of the results. The raw TIFF images are saved in `/mnt/daqtesting/secar_camera/new_captures/` and the background and light images should be saved in the subfolder `lighton_and_background/`.

## 4.1 Code Structure

The source code in the `/user/e18514/Documents/viewer-image-analysis/src/` directory is comprised of six main files:

- `settings.py`
  **This is typically the only file you may need to edit**. Here you can set the flag to show the plot on the screen every time the code runs to True or False - either way the image is saved in the output folder. The background images used for each viewer are defined here. They should be updated when the background is not being well subtracted (or preferably before each experiment). Other parameters relevant to defining the viewer dots are set. *These should not change unless changes in the beamline affected the viewer.* These include: the light image for each viewer, the region of interest (ROI) where the five dots in the viewer are, the center location of the middle viewer dot, and the mm to pixel scale extracted. The center and the scale are detected automatically using the `dot_detection.py` script and manually updated in this file. To use the `dot_detection.py`, a template of what the dots look like is defined in `dot_im` as well as a detection threshold (which you probably don't need to change). Finally a flag is set to detect the dots each time the code runs (True) or only when the `dot_detection.py` is run (False).

- `dot_detection.py`
  This file can be run standalone to detect the five dots on each viewer and get the mm to pixel scale as well as the dot location for plotting them on the final image output. This only needs to be done when there are changes in the beamline that affect the dot locations in the camera field of view, or when camera settings have changed (rotation, zoomed in/out, warp, etc). The function `find_dots()` looks in the ROI defined in `settings.py` for a peak in intensity resembling the dot template also defined in the settings, and returns the peak locations. The `get_scale()` function then takes the output and loops over all matches and looks for the peaks that share an axis (since the five dots make a cross). Once those are found, the pixel distance between the correct dots is determined and the scale is found. When the script is called, a plot of the template and the results prints to the screen. To run it, make sure the correct light image is defined in `settings.py`, then run `python dot_detection.py D*`, where D* is the corresponding viewer location. If it did not detect the correct dots, you can change the template or adjust the threshold in `settings.py`. A new template can be created by simply cropping a light image around a dot.

- `im_input.py`
  This file connects the command line arguments with the settings and provides the correct input to the analysis scripts that will do the work and the correct path to save the output. There is generally no need to edit this file.

- `im_reduction.py`
  This file is the essence of the image analysis. The `Image` class is defined here with methods that handle the background subtraction and the integration over the x and y axes. The function `find_median()` that takes an integrated profile in either direction and finds the median (index separating 50% of the data) and the $\pm 1\sigma$ (indices at 15.8 and 84.2%) is defined. This is the function used to find the center location and width in the x-axis and y-axis for all images.

- `im_analysis.py`

This file contains the main function that is run when the program is called. The other modules described above are imported, and the image is analyzed: background is subtracted, the integrated profiles in each axis are found, and then the median, width and peak locations (max intensity, not currently used) are found from the integrated profiles. Plotting these parameters and the known dot locations on the viewer, a figure showing the subtracted image and the integrated profiles is saved (and shown on the screen if specified in the settings). A CSV file is also saved with the following information in a row, all in the unit of pixels: median location in x, median location in y, peak intensity location in x, peak intensity location in y, x distance from the center dot, y distance from the center dot, -1$\sigma$ location in x, +1$\sigma$ location in x, -1$\sigma$ location in y, +1$\sigma$ location in y. **Any desired changes to the plot (such as changing x-axis limits to zoom in) or the saved output need to be made in this file.**

- `stats_profile.py`
  If your heart desires a fancy Monte Carlo simulation to find the mean and standard deviation of the beam profile in the x- or y-axis instead of just the median and width from integration, this file is here to satisfy that. **Caution:** this feature has not been used in a while, so errors might come up. There are three functions: `profile_stats()`, `find_mean()` and `find_std()`. The first one runs the MC 50000 times or whatever number you define and samples a new profile each time from a Gaussian distribution with mean as the given profile and sigma as its error (as calculated in `im_reduction.py` and passed on from `im_analysis.py`). The other two functions are then used to find the weighted mean and standard deviation of each sample, giving a distribution of 50000 means and standard deviation for a given profile. The final returned mean (center of the beam) and standard deviation (half width of the beam) is then just the distribution mean (mean of the means) and the distribution standard deviation (mean of the standard deviations). The corresponding section in `im_analysis.py` can just be uncommented to use this method instead of the median method.

Other directories in `viewer-image-analysis` that are important are the `output/`, and the `tiff_files` which contains the viewer dot image templates used in the `dot_detection.py` file.

## 4.2 Running the Code

To run the analysis on a certain image (only takes one image at a time), type in the following command with the image name and the corresponding viewer location (e.g. D1542)

```
python /path/to/src/im_analysis.py /path/to/images/imagename.tiff D*
```

The raw images are currently being stored in `/mnt/daqtesting/secar_camera/new_captures/`. The output is saved to whatever directory is specified in the `im_analysis.py` file. It outputs a PNG of the visualization along with a CSV file with the beam information as described in the previous section (file `im_analysis.py`).

**Note:** due to the large amount of PNG files being saved when running the optimizer code, the PNG files were being saved in the e18514 account's `events/` folder, while the CSV files were still being saved in the local `output/` folder.

# 5 Bayesian Optimizers

This section presents an overview of the scripts used to optimize the incoming beam angle of the beam using the ReA3 steerers at D1413 and D1431 and to adjust the quad strengths to optimize the ion optics of SECAR, both via a Bayesian optimization using Gaussian processes (GPs). Both optimization codes are described in this section as they share the framework and the Gaussian process modules. The GPy[1] library and the associated GPyOpt[2] tool are utilized in these scripts for the Bayesian optimization and the GP framework, respectively.

## 5.1 Code Structure

Of the Python scripts in the `/user/e18514/Documents/tuneoptimizer/` directory, four files are relevant for the Bayesian optimizations:

- `setup.py`
  This module defines all the variables and functions that are common to the optimizations. This includes all the SECAR magnets' location information, and several PVs needed to set/read magnet currents, read probes, etc. All functions needed for the optimizers (outside of the Gaussian process framework) are defined here, including functions to cycle magnets, read and set magnet currents, save viewer images, get beam spot center locations in images, get steering distance, and set steerer magnet currents. If any changes are needed to be made to these functions, this is the file to edit.

- `gaussianprocess.py`
  This is modules contains all custom functions and definitions relating to the Gaussian process. It is imported in the beam and quad optimizer files defined below. The function `x_grid_fun()` creates a grid of points that samples the phase space defined by the domains of each magnet being optimized. This is later used to evaluate the acquisition function over the phase space. The `returnObservations()` function provides a sandbox function to test the algorithm or any new changes in the code without beam. The `evolution()`, `GP_analysis()`, and `plot1D()/plot2D()` are functions used for visualizing the GP results given a text file containing the input data and the function values. Note that in order for the visuals to be correct, the domain limits, the exploration weight, and the priors on the lengthscale and the noise have to be changed in these functions to the exact values that were used when the results in the text file were generated.

- `beam-optimizer.py`
  This is the main file for optimizing the ReA steerers. To initialize the code before running, several things need to be set: the steerers to be optimized (`magnet_list`), the `viewer` location (string, e.g. 'D1542'), the domain for each magnet (`spaceArray`), the acquisition function exploration weight (if applicable), and the model hyperparameters. Additionally, the steering distance needs to be set to be either the steering in x and y combined or steering in one direction (if using only vertical steerers for example). At the assignment of `distance`, set `separateXY = True` and take the first index of the returned value ([0]) for x, or the second ([1]) for y. For combined, set it to `separateXY = False` (no indexing).

[1] http://github.com/SheffieldML/GPy
[2] http://github.com/SheffieldML/GPyOpt

The lengthscale prior and the noise prior are each defined as a Gaussian with a mean $\mu$ and variance $\sigma$. The initial priors have to set in the regression model, for example: `m.kern.lengthscale.set_prior(GPy.priors.Gaussian(10,2))`, and `m.Gaussian_noise.variance.set_prior(GPy.priors.Gaussian(1,0.5))`.

If using the LCB acquisition function, the `exploration_weight` is passed when instantiating the function `acq` and usually set between 0.5 and 3 Amps.

This file looks at steering in Q1 and Q2 (changes tunes as defined in the file). This can be changed at the beginning of the main while loop. Most combinations of quads and viewers have already been made in several version of this code found in the same folder, each named after the location or the quads used (for example `beam-optimizer_with_Q6-Q9.py`). Aside from viewer/quad changes, all these codes work identically.

To test the algorithm without using beamline elements, a sandbox function can be used to generate random data by setting `sandbox='y'` at the top of the file.

- `quad-optimizer.py`
  This is the main file that optimizes the beam spot width using the quadrupoles. It is very similar in structure to `beam-optimizer.py` as it uses the same algorithm. The main differences are the inputs (quad currents vs steerer currents) and the function being sampled (function of beam spot width vs steering). To initialize, the `viewer`, the magnets being optimized `magnet_list` (string list of magnets, see definitions in `setup.py`, dict `mag_loc`), and the phase space of each quad `q_ps` (can set each individually, or take initial current - 2 A for all for example) need to be set. The lengthscale and noise prior and the exploration weight need to be set as well similarly to how it's done in `beam-optimizer.py`.

  To adjust the objective function, change the definition of `f_gp` in this file.

## 5.2 Running the Codes

Once all the settings and variable are initialized, both optimizers can be run similarly through the command line as such:

```
python /path/to/Documents/tuneoptimizer/beam-optimizer.py
```

The main output for `beam-optimizer.py` is the file `correctorValues_Distance_timestamp.txt` that contains the current values of the steerer magnets (first two to four columns depending on how many steerers are being optimized), and the value of the steering distance in the last column.

The `quad-optimizer.py` outputs the file `q*Values_Widths_timestamp.txt` with the first several columns containing the currents of the magnets being optimized, followed by five columns containing: the x width at that tune, the x distance from the initial position, the y width, the objective function value, and the image name of the first tune taken at each iteration (to be able to find the images corresponding to each row later on).

All outputs are in pixels. Additionally, both codes output a text file containing the hyperparameter values at each iteration, `opt_params_timestamp.txt`. This file is helpful to plots results later as the hyperparameters need to be know to visualize the optimization.

# 6 Dipole Scan Codes

# 7 Version Control