Aitor Amatriain

Final Project Physics 495

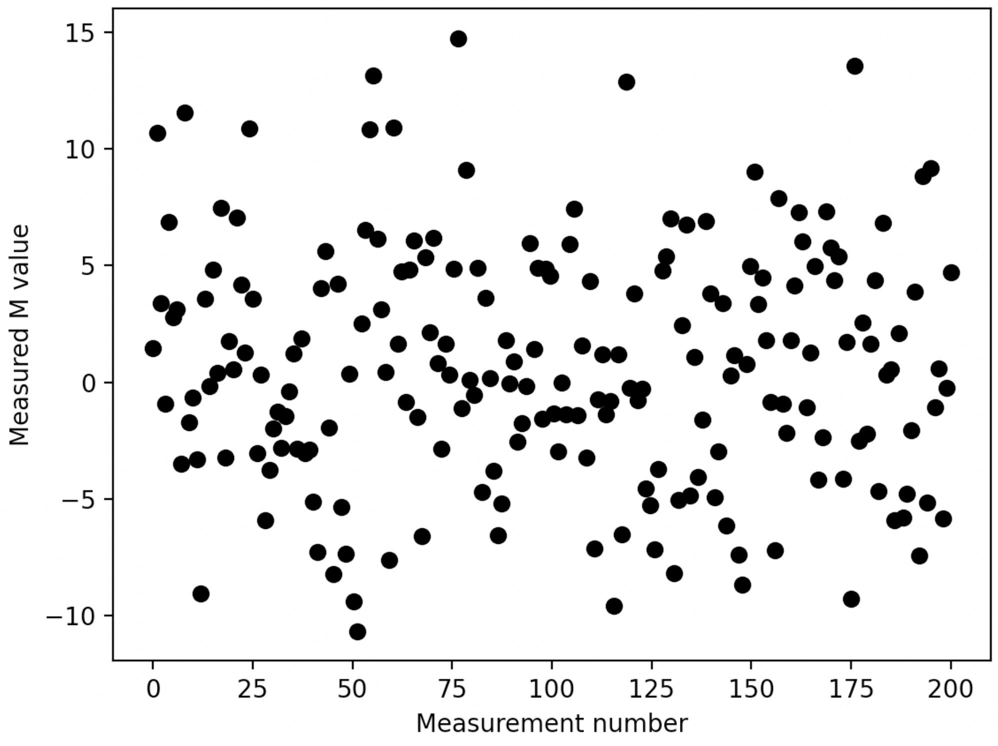
For an introduction to the purpose of the project, read the senior project description file. The first task for this project was to build a program that, given a bunch of simulated data representing the measured qubit states for many consecutive trials as well as their actual state, could return an optimized threshold where, all trials to the left of that threshold would represent qubits that had started with spin down, and all trials to right would represent qubits that started with spin up.

Simulating the trials

To do so, I first had to make a simulator that modeled the typical behavior of qubits under a measurement device. The simulator is fairly simple, with one trial simulating a measurement device that takes measurements of the same qubit every 5 nanoseconds (0.005 in the units used in my code, microseconds) during any given integration time (time of duration for the trial), with each measurement returning a float value representing M (the state of the qubit). The trials can start with a float value M0 of 0.5, representing the qubits that started with spin up, or with a float value of -0.5, representing spin down. The device is supposed to have a lot of noise (which is the reason the measurements can't ever be exact, and a threshold is needed to classify trials), so I coded a noise mean of 0 and a standard deviation of 5 for the noise. Given that the noise is so large, many trials starting with different values of M will nevertheless give similar results. Each measurement taken will therefore have a value of $M + random.gauss(mean, standard\ deviation)$, where random.gauss gives a random value for the noise with the given standard deviation and mean. For the trials starting with spin up (M = 0.5), there is always a probability of $1 - exp(- \mathtt{t}/T)$ for M to switch to spin down (-0.5), where $\mathtt{t}$ (t_between_points in the code) represents the time between measurements (0.005 microseconds, as said before), and T(T1 in the code) represents the expected time for a qubit to change states (10 microseconds). All of this can be seen in code in the function

$def\ noise\_simulator(M,\ it)$. Here is what a random trial starting in spin up (M=0.5) and having an integration time of just 1 microsecond will look like with my code (Fig.1):

Figure 1

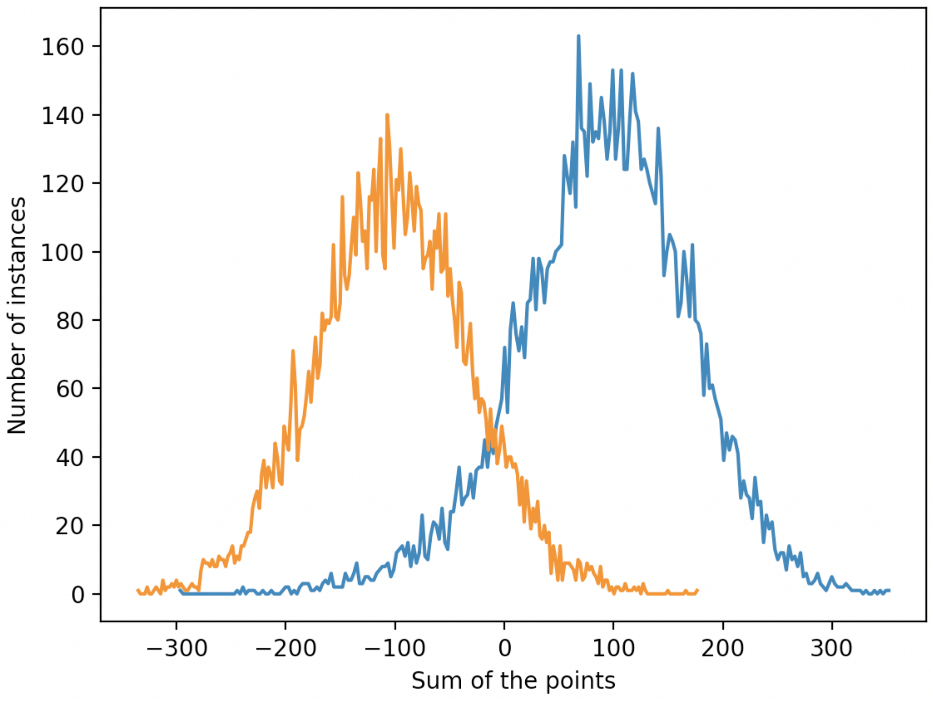


This is one trial, where the x axis defines each measurement of the qubit state taken during the trial (as you can see, 1 trial has 200 measurements of a single qubit when the integration time is 1 microsecond), and the y axis is each measured M value taken. There is not much interesting to see here, but remember that this is only one trial, which represents only one data point out of the many thousands of trials that will be used and compared. It is nevertheless important to understand how each trial was simulated.

Creating the distributions

For each trial, each of the measured M values are then added up, and that summation will represent one data point in the larger distribution. Therefore, if 10,000 trials are taken (each of which has a similar plot as the one seen above, and half of which start with M = 0.5 and half with M = -0.5), there will be 10,000 different data points in the distribution. Here is what a histogram distribution consisting of 10,000 different trials, each with an integration time of 1, would look like:

Figure 2.



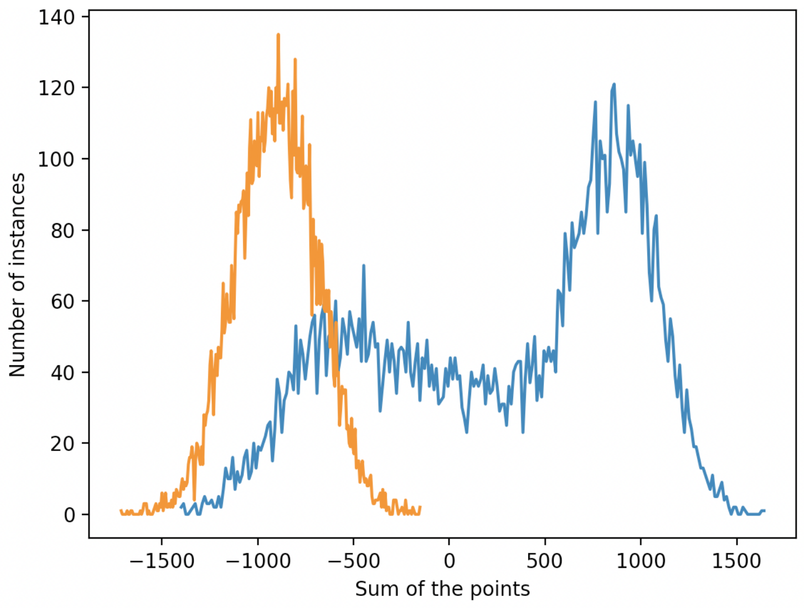
The x axis is the sum of the points of each separate trial (all the points seen in the scatter plot of Fig.1), and the number of instances means the number of times each sum value (within a small range) has shown up. This distribution can be obtained by calling

$def\ create\_distribution(it,\ size)$, with an it (integration time) of 1 and size 10,000. This will return two lists, the first one (in fig.2 it is the blue curve) being the trials that started with M = 0.5 and the second being the trials that started with M = -0.5 (orange curve). For more information on the $create\_distribution$ function, look at the function library at the end of this document. This function does all the steps outlined so far; it simulates the noise, adds trails up, and finalizes the distribution. To plot this graph, simply send a distribution to

$def\ plot\_distributions(distribution1,\ distribution2)$, where $distribution1$ is the first list returned by $create\_distribution$ starting in the upper state and $distribution2$ is the second one starting in the lower state.

Now we need to find the best threshold in order to classify each of the points. In figure 2 you can visibly guess where that point could be, around x = 0, but for other integration times the choice is much less clear. For example, here is a distribution when the integration time is 9 microseconds (attained by calling $create\_distributions(9,\ 10000)$ and then plotting both lists returned with $plot\_distributions$) :

Figure 3



As you can see, the choice about where the best threshold would be is less clear now, since many of the trials that started in the upper state decayed into the lower state due to the larger integration time. I then devised a couple different approaches for classifying the points, which I will now explain.

Finding the best threshold for a given integration time using linear search

The first potential approach is simple, involving a linear search through all the histogram x values until the best threshold is found. In order to make the distribution graphs seen above, two histograms (one for upper, one for lower) are created, each of which is made out of 250 bins. These bins all have different ranges, and all the data points that fall in each of the bins adds one to its respective instance count (y value). This method then simply tries to put the threshold in the starting x value of each of the 500 bins (250 lower + 250 upper), stepping through each of them and finding the bin where the error is smallest. Error is calculated as $incorrect\ upper\ +\ incorrect\ lower\ /\ total$. The numerator here corresponds to the total number of trials that were incorrectly classified due to them being at the wrong side of the threshold for both the trials that started on the lower state and those that started in the upper state. The denominator is the total number of trials. This error is measured for any given threshold using the function $def\ calculate\_error(threshold,\ d1,\ d2)$. Once the histogram bin with the smallest error is found, the corresponding x value is returned. This can all be done in the function $def\ linear\_bins\_threshold(dist1,\ dist2)$, which takes in the upper state distribution and the lower state distribution returned by $create\_distribution$. The fidelity (1 - error) slightly varies each time the program is run (since the distributions are randomly produced and simulated) and for each integration time, but mostly stays around 0.9 / 90% accuracy. For example, randomly running $linear\_bins\_threshold$ with integration time 1 gave a threshold of -2.29 and a fidelity of 0.91, and running it with integration time 10 gave a threshold of -626.73 and a fidelity of 0.89. Again, check out the function library at the end of this document for more info on how to run each function.

Finding the best threshold for a given integration time using dual annealing search

The second approach is similar to the first, but instead of making the histogram and running through the bins, it uses *dual (simulated) annealing* on <u>all</u> of the distribution points (not just the histogram) to find the best value for the threshold. Simulated annealing is an optimization technique that, when told to minimize the $calculate_error$function, tries different threshold values until a good one is found using a mathematical method. I would recommend to search up this method if you are interested in knowing more about how it works. I used scipy's dual annealing algorithm, which uses a modified version of simulated annealing, to minimize my $calculate\_error$ function for the two given distributions. Again, the fidelity slightly varies each time the program is run, but a random run using $def\ dual\_annealing\_threshold(dist1,\ dist2)$ of integration time 1 returned a threshold of -0.91 and a fidelity of 0.90, while a random run of integration time 10 returned a threshold of -598.09 and a fidelity of 0.89. Check out the function library for more info on the$dual_annealing_threshold$ function. In addition, also made a version $dual\_annealing\_threshold\_cpp$ that runs in C++, making it about 3 times faster for this function. Look at C++ section of this document where I will explain how to compile the function.

Finding the best threshold for a given integration time using Machine Learning

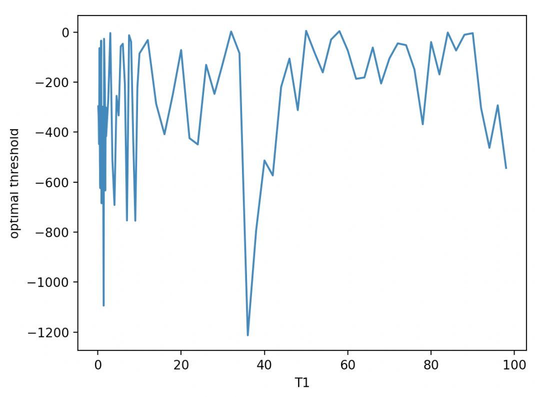
Finally, the last method I made to classify the points was to train a machine learning model that could learn to classify each trial point. Unlike the previous two approaches, this does not calculate a hard threshold where everything on the left side is classified as lower and everything on the right side is classified as upper. This method uses a probabilistic approach called *random forest,* which generates a large amount of probabilistic decision trees and averages out each of their decisions. This model then individually classifies each of the points as having started either at the upper state or at the lower state, without making any threshold whatsoever. If you are interested to learn more about this model, I recommend you search up the basics of how random forest works, since I will not delve any deeper here. Another useful feature of this method is that it can learn to classify over however many different integration times you want all at once. This is why I created a different function called $def\ create\_multiple\_distributions(low,\ high,\ size)$, which this time returns two 2d lists of distributions starting in the lower and upper state, for a given range of integration times (more info in function library). You can then train the model to quickly classify points for any integration time. For more information on how to prepare the data, train the model, and use it to make predictions, check out the function library under $def\ train\_random\_forest(distribution,\ spins)$. This random forest model is currently not as good at making predictions as the threshold counterparts, but a big reason for this is because the model has not been tweaked very much or been given very large amounts of data in my tests. For example, a random run using integration time of 1 returned an accuracy score (fidelity) of 0.8575, while a run using integration time of 10 returned an accuracy score of 0.8356. Other machine learning models, such as gradient boosting, could potentially be even better.

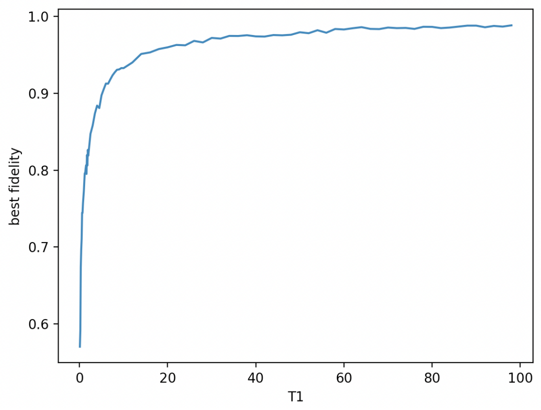
Finding the best integration AND threshold for given data

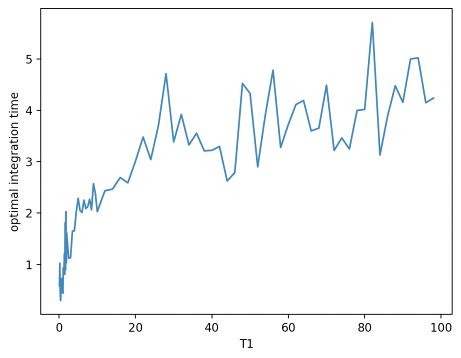
Next I made a method that, using dual annealing, can find the best integration time and threshold values to maximize fidelity for a given data set of trials. This function $DA\_threshold\_it(T1,\ threshold\_guess)$ creates 8000 simulations of a large trial time first (not distributions), and then runs dual annealing to minimize a new error function $advanced\_calculate\_error$. Note: T1 is the time (in microseconds) that a qubit will usually take to decay, which you must input. This new error function adds up the simulations itself to the chosen integration time limit to create a distribution (more efficient than making a new simulation and distribution each time). It then calculates the error of the chosen threshold and returns it. Once the dual annealing is completed, the best integration time and threshold is found. The function can be slow (about 800 seconds for the basic one), but runs much faster (about 10x+) on its C++ version $DA\_threshold\_it\_cpp(T1,\ threshold\_guess)$. Because it runs so much faster on this version, I also allowed it to run many more trials of dual annealing, making it more accurate than the base version, so I highly recommend you use the C++ version for this.

Finding the best integration and threshold for given data for various different T1 values.

Finally, for testing and analysis purposes, I also made a function that finds the best integration time and threshold, and does so for a bunch of different T1 values (From 0.1 to 2 in increasing steps of 0.1, 2 to 10 in steps of 0.5, and 10 to 100 in steps of 2). This function is called $find\_T1\_to\_optimal\_thresh\_and\_it()$, and essentially simply runs $DA\_threshold\_it$ over and over again. Here's an example of what you get when you graph the outputs of this function.







Because of these repetitive tasks (this basic version of the function can take about 8 hours or so), I created a function $parallel\_find\_t1\_to\_optimal\_thresh\_and\_it$ that makes use of multiprocessing, so it can use the power of all the CPU available on your device, which speeds it up by a factor of about 3 on my 8 core device. On a larger machine, it will run even faster. I also made a version that makes use of both multiprocessing and C++, giving the best performance (recommended function).

Running the C++ versions

A few of the functions (outlined above and in the function library) have C++ versions that make them run much faster than their full python counterparts. These functions always have _cpp at the end of their names, making them easy to notice. I used ctypes in order to call C++ functions in a separate file (*faster.cpp)* straight from python, so you don't have to worry about looking at the c++ code, the actual function should be called in python like the others. However, in order to run these, you must first compile the C++ code in the *faster.cpp* file by running the following command on your terminal:

```
 g++ -fPIC -shared -o fasterTest.so faster.cpp
```

You should now be able to run the C++ versions of these functions. All the C++ code is in the *faster.cpp* file, and does essentially the same logic as the ones fully in python.

Function Library (Also in the README):

The following functions are not an exhaustive list of all the functions made in the project, but only the most important ones / the ones that should be called. Some of the functions that aren't meant to be called (such as $noise\_simulator$) are explained in the project document above. These are the only functions that are meant to be called by the user, do not call the others.

$def\ create\_distribution(it,\ size)$

This function makes a distribution of size $size$ within a given integration time it, simulating the noise, adding trails up, and finalizing the distribution.The function returns two lists (each of size $size/2$) , the first of which represents the distributions that started in the upper state (M = 0.5) and the second of which represents the distributions that started in the lower state (M = -0.5).
Example:
`upper_dist, lower_dist = create_distribution(4, 10000)`
`upper_dist` is the upper state list, `lower_dist` is the lower state list. Integration time is 4.

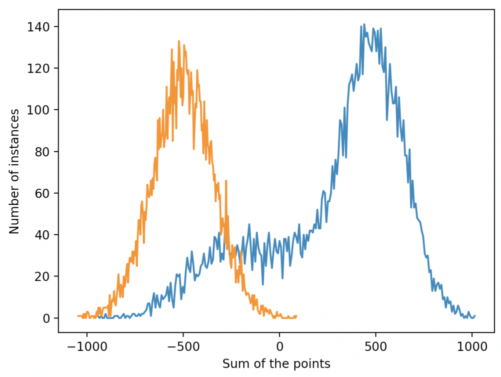
$def\ create\_multiple\_distributions(low,\ high,\ size)$

This function makes distributions of size $size$ within a given low to $high$ integration time range, simulating the noise, adding trails up, and finalizing the distribution.The function returns two lists

(each of size $size/2$) , the first of which represents the distributions that started in the upper state (M = 0.5) and the second of which represents the distributions that started in the lower state (M = -0.5). If the range low, $high$ is 5 to 6, then two lists will be returned, each of which were calculated with only integration time 5. If the range was 1 to 6, the function would return 2 lists still, but each of the lists would have integration times 1, 2, 3, 4, and 5, and the list would also be 5 times larger than the one calculated with the range of 5 to 6. This added functionality of multiple distributions for different integration times is added in case the user wants to train a model that understands multiple different integration times all at once.
Example:
```
upper_dist, lower_dist = create_multiple_distributions(4, 9, 10000)
```
`upper_dist` the upper state list, `lower_dist` is the lower state list. Integration time is 4, 5, 6, 7, and 8 for this range, and the size of each is 5,000.

$def\ plot\_distributions(distribution1,\ distribution2)$

Plots two different distributions into a histogram. For most purposes, $distribution1$ will represent the first list returned by $create\_distributions$ (trials starting in upper state), and $distribution2$ will represent the second list (lower state), for only one integration time.
Example:
```
upper_dist, lower_dist = create_distribution(5, 10000)
plot_distributions(upper_dist, lower_dist)
```
Output:



$def\ linear\_bins\_threshold(dist1,\ dist2)$

Returns the best threshold value using a linear step through histogram bins. First creates a histogram from distributions dist1 and dist2. For most purposes, $dist1$ will represent the first list returned by $create\_distributions$ (trials starting in upper state), and $dist2$ will represent the second list (lower state), for only one integration time. Function then steps through each of the histogram bins and finds the bin where the error is smallest. Error is calculated as

$incorrect\ upper\ +\ incorrect\ lower\ /\ total$. The numerator here corresponds to the total number of trials that were incorrectly classified due to them being at the wrong side of the threshold for both the trials that started on the lower state and those that started in the upper state. The denominator is the total number of trials. This error is measured for any given threshold using the function $def\ calculate\_error(threshold,\ d1,\ d2)$. Once the histogram bin with the smallest error is found, the corresponding x value is returned.
Example:

```
upper_dist, lower_dist = create_distribution(5, 10000)
threshold = linear_bins_threshold(upper_dist, lower_dist)
```

Output of `print(threshold)`:

```
-201.48506047125466
```

Note: The fidelity (`1 - calculate_error(threshold, upper_dist, lower_dist)`) of this threshold for the above integration time and distributions is 0.9181

$def\ dual\_annealing\_threshold(dist1,\ dist2)$:

Returns the best threshold value using a dual annealing approach. For most purposes, $dist1$ will represent the first list returned by $create\_distributions$ (trials starting in upper state), and $dist2$ will represent the second list (lower state), for only <u>one</u> integration time. Dual annealing is imported from $scipy.optimize\ import\ dual\_annealing$, which attempts to minimize the $def\ calculate\_error(threshold,\ d1,\ d2)$ function, with $threshold$ as the variable to be changed and the two distributions returned by $create\_distributions$ as constants. The function returns an object that describes the result obtained, including the minimum error found by dual annealing as well as the threshold value that obtained that error.
Example:

```
upper_dist, lower_dist = create_distribution(5, 10000)
threshold = dual_annealing_threshold(upper_dist, lower_dist)
```

Output of `print(threshold)`:

```
fun: 0.0801
 message: ['Maximum number of iteration reached']
    nfev: 2021
    nhev: 0
     nit: 1000
    njev: 10
  status: 0
 success: True
```

```
        x: array([-203.35351449])
```

In the above output, *fun: 0.0801* means that the optimized error found was 0.0801, or a fidelity of 0.9199. *x: array([-203.35351449])* means that the threshold value that returned this error score was -203.35.

def dual_annealing_threshold_cpp(dist1, dist2):

The same as *dual_annealing_threshold*, but runs in C++. This makes it more than 3x faster. Remember to run *g++ -fPIC -shared -o fasterTest.so faster.cpp* on your terminal before running this, else it will not work.

def train_random_forest(distribution, spins)

This function trains a random forest machine learning model that can learn to classify trial points as either starting in the upper state or the lower state. Before running this function, you should first call the *create_multiple_distributions* function. If you want multiple different integration times to be trained go ahead, if not, still call that function but do so with a range of consecutive numbers (ex: *create_multiple_distributions*(1, 2, 10000) will create a distribution for integration time 1). Do NOT call the *create_distribution* function. Next, prepare the data by calling the *prepare_data* function with the two lists returned by *create_multiple_distributions* as well as whatever *low*, *high* integration times you made the distributions with. This will return 4 different lists; the training data, training labels, testing data, and testing labels. If you want to save this data (good thing to do to save time) call the *save_data* function with these 4 lists as parameters, and this function will save the lists into text files. Use the *read_data* functions to recover these lists from the text files in the future. You are now ready to run the *train_random_forest* function, with the training data as the *distribution* parameter and the training labels as the *spins* parameter. The output of this function will be a trained model, so use *model.predict*() to predict the labels (the starting qubit state, 0 or 1) of the testing data returned by *prepare_data* and then compare the predicted labels with the testing labels. For clarification look at the example below, which trains only on an integration time of 5.
Example:

```
upper_dist, lower_dist = create_multiple_distributions(5, 6, 100000)
training_data, training_labels, testing_data, testing_labels =
prepare_data(upper_dist, lower_dist, 5, 6)
save_data(training_data, training_labels, testing_data,
testing_labels)
model = train_random_forest(training_data, training_labels)
rf_predictions = model.predict(testing_data)
score = accuracy_score(rf_predictions, testing_labels)
```

Output of `print(score): 0.8699`

This output means that for an integration time of 5, the model correctly classified the trial correctly 87% of the time.

def DA_threshold_it(T1, threshold_guess):

Finds the best integration time and best threshold for the given T1. Does this by using the dual annealing algorithm with two varying parameters, calling advanced_calculate_error() and optimizing it. Takes in as parameters a best guess for the threshold as well as T1 (typical time for qubit to decay from upper to lower state). Returns the optimization result represented as a OptimizeResult object, Important attributes are: x the solution array (x[0] js threshold, x[1] is integration time), fun the value of the function at the solution (the fidelity, and message which describes the cause of the termination. Search up OptimizeResult for a description of other attributes. Takes in T1 (The typical time that it takes for a qubit to go from upper to lower state) and your best guess for a threshold (any number, 0 is fine). Takes about 800 seconds to run, so recommend you use the C++ version if possible.
Example:
```
upper_simulations, lower_simulations = noise_simulations(8000, 15,
10)
result = DA_threshold_it(1, 5)
print(result)
```
Output on next page

Output of `print(result)`:
```
fun: 0.11500000208616257
 message: ['Maximum number of iteration reached']
    nfev: 4031
    nhev: 0
     nit: 1000
    njev: 10
  status: 0
 success: True
       x: array([-8.153239  ,  0.40447468])
```

def DA_threshold_it_cpp(T1, threshold_guess)

The same as *DA_threshold_it*, but runs in C++. This makes it more than 3x faster. It is also significantly more accurate, since its increased speed allows it to run more iterations without taking too long. Remember to run *g++ -fPIC -shared -o fasterTest.so faster.cpp* on your terminal before running this, else it will not work.

$def\ find\_T1\_to\_optimal\_thresh\_and\_it()$:

Note: Takes several hours, returns arrays that can be used for plotting. This function runs DA_threshold_it several times for a multitude of different integration times. It then returns the thresholds, integration times, and fidelities that were found for each subsequent T1. To see how they fared, plot any of those 3 first arrays returned with the array of T1's (the last, fourth array returned). This way you can see how each of those variables changed as the T1 variable changed. Remember, the 1st value of the T1 array accounts for the same trial of the first value of the threshold array (or it/fidelity ones), the 2nd value of T1 array accounts for the same trial of the 2nd value of the other arrays, and so on. It saves all these values to some text files on your computer, so you don't have to repeat after running the function for hours.

$def\ parallel\_find\_T1\_to\_optimal\_thresh\_and\_it()$:

The same as $find\_T1\_to\_optimal\_thresh\_and\_it$ but makes use of multiprocessing

$def\ find\_T1\_to\_optimal\_thresh\_and\_it\_cpp()$:

The same as $find\_T1\_to\_optimal\_thresh\_and\_it$ but makes use of multiprocessing and C++, making it by far the fastest and best choice. Remember to run *g++ -fPIC -shared -o fasterTest.so faster.cpp* on your terminal before running this, else it will not work.

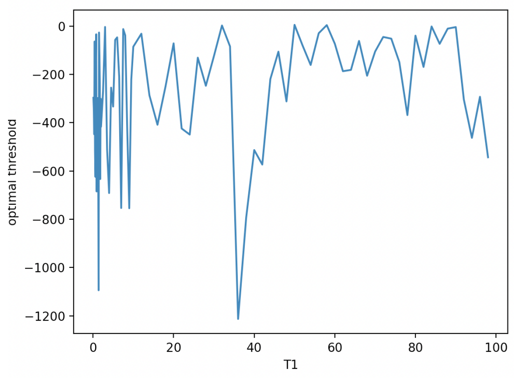
$def\ read\_T1\_to\_optimal\_thresh\_and\_it\_data()$:

Returns the saved arrays from either of the the $T1\_to\_optimal\_thresh\_and\_it\_cpp$ functions by grabbing them from text files they were saved in. Use this whenever you want to get a previous result of $T1\_to\_optimal\_thresh\_and\_it\_cpp$ without having to run it all over again
Example:

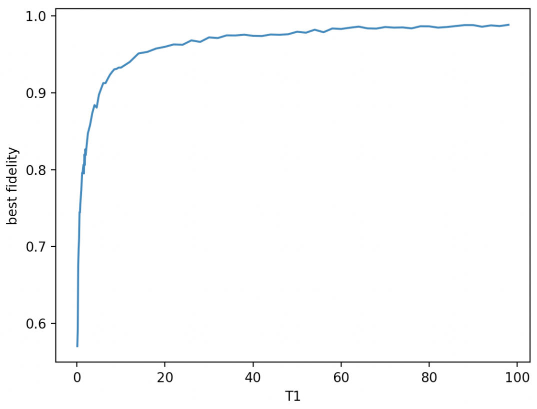
```
thresholds, int_times, fidelities, t1s =
read_T1_to_optimal_thresh_and_it_data()
```

def plot_T1_to_optimal_thresh_and_it(thresholds, i_times, fidelity_list, t1s):

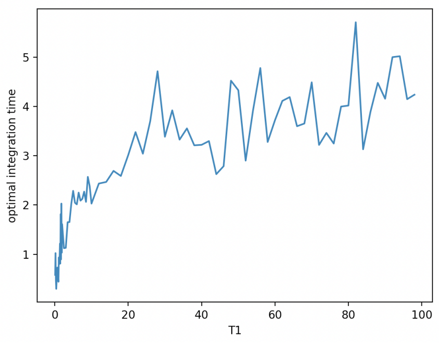
Takes in the result returned by any of the $T1\_to\_optimal\_thresh\_and\_it$ functions (the 4 arrays returned by those functions) and creates 3 plots of threshold, integration times, and fidelity with respect to T1.



optimal threshold
0
−200
−400
−600
−800
−1000
−1200
0
20
40
60
80
100
T1



best fidelity
1.0
0.9
0.8
0.7
0.6
0
20
40
60
80
100
T1



optimal integration time
5
4
3
2
1
0
20
40
60
80
100
T1