Project E: Surveying Telescopes

Using interference data from a telescope array, it is possible to deduct where the light collectors are located with respect to each other. In this report, optical path delay measurement data was used from the CHARA array in California to achieve just that. My results reveal the relative collector displacements with very high precision, and highlight the small movement over a 3-year period, due to seizmic activiy. The proposed computational solution uses the Python libraries Numpy and Pandas, and solves the problem with a linear complexity in data volume.

Introduction

This problem consists of one main task which is to determine the relative 3D positions of all pairs of telescopes. For a single pair, this can be achieved through selecting the right subset of data and passing it on to an algorithm that determines the relative position of one telescope with respect to the other. Once this can be done efficiently, the rest of the problem could be approached by parsing the main dataset and applying the single pair algorithm to each subset that represents measurements for a particular telescope pair. This forms the core of the solution, and the results can be improved through refining the input data by removing outliers and subsets that are underconstrained.

Since this is a problem in analysing a static dataset, the main concern regarding efficiency is how the running time of the code scales up with increasing volume of data. Efficiency in memory has not been investigated, as the dataset is too small for this to become a problem in a conventional computer. The core numerical routine is a linear regression in four dimensions, which has been outsourced to Numpy, an established and optimised numerical library.

In the next section, I will present the theoretical reasoning behind my approach, and will put the problem into a well-defined mathematical framework. In the Implementation section, I will discuss my choice of tools and libraries, as well as the overall program structure. Finally, I will present my results along with a discussion on the performance of my code.

Analysis

The equation that defines when an interference fringe can be observed is given as:

$$\Delta p = \hat{\mathbf{S}} \cdot (\mathbf{x}_1 - \mathbf{x}_2) + d_1 - d_2 \tag{1}$$

where

- Δp is the optical path difference to the measurement point.
- $\hat{\mathbf{S}}$ is the unit vector direction where the telescopes are pointing.
- $d_{1,2}$ are the internal delays associated with each telescope pair.

The $d_1 - d_2$ term incorporates all delays in the system, including atmospheric delay, POP and OPLE settings, and any unknown constants relating to a specific pair of telescopes.

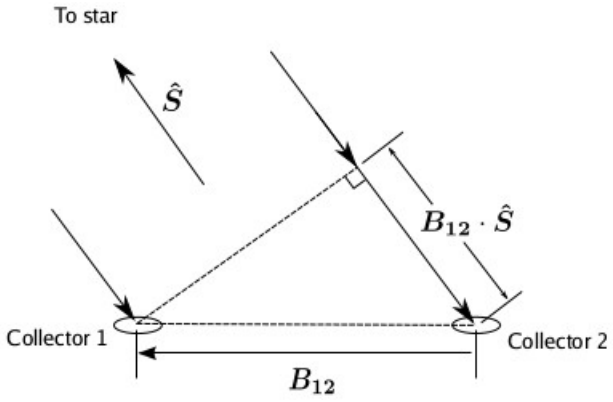

Figure 1: Diagram of a telescope pair illustrating the origin of parameters in equation (1). The $\mathbf{B_{12}}$ term is also called the "baseline" and is equivalent to $(\mathbf{x}_1 - \mathbf{x}_2)$.

A fringe can be observed when Δp is less than a few microns. The starting point for my solution is to let $\Delta p = 0$. This is a good assumption, because the atmospheric fluctuations are on the order of $100\mu m$.

Since the POP setting is only safely assumed to be constant for a single night, and every telescope pair has a different unknown constant optical path offset, the system of equations can further be represented as:

$$\hat{\mathbf{S}} \cdot (\mathbf{x}_1 - \mathbf{x}_2) + A_n = D_2 - D_1 \tag{2}$$

where

- $\mathbf{x_1}$,$\mathbf{x_2}$,D_1,D_2 are given by the datapoints.
- A_n is a constant for every telescope pair in a given night.

The gaussian noise due to atmospheric fluctuations is the main source of spread in data, and is incorporated within the $D_{1,2}$ measurements.

Assuming a coordinate system with y-axis aligned North (x-axis East, z-axis vertically up), equation (3) takes its final form:

$$\begin{pmatrix} \cos\theta_0\cos\phi_0 & \cos\theta_0\sin\phi_0 & \sin\theta_0 & 1 \\ \cos\theta_1\cos\phi_1 & \cos\theta_1\sin\phi_1 & \sin\theta_1 & 1 \\ \vdots & & & \\ \cos\theta_i\cos\phi_i & \cos\theta_i\sin\phi_i & \sin\theta_i & 1 \end{pmatrix} \begin{pmatrix} x_n \\ y_n \\ z_n \\ A_n \end{pmatrix} = \begin{pmatrix} {D_0}' - {D_0}'' \\ {D_1}' - {D_1}'' \\ \vdots \\ {D_i}' - {D_i}'' \end{pmatrix} \quad (3)$$

where

- θ_i,ϕ_i are elevation and azimuth respectively.
- D',D'' are the OPLE delay measurements from telescopes 1 and 2.

Filtering away the underdefined subsets of data will leave us with mostly having to solve for an overdefined system. The simplest approach from here is to solve a linear regression problem in 4 dimensions, treating (x_n, y_n, z_n, A_n) as the parameters which fit the data. The numpy.linalg.lstsq is a least squares solution routine that provides the necessary functionality.

Implementation

Solving the problem as outlined in the previous section, requires the selection of subsets of data, passing them on to a regression algorithm, and filtering of the results upon given conditions. This is very reminiscent of the functionality provided in a database or a spreadsheet. While multidimensional Python lists and Numpy arrays can be adapted to suit this purpose, it would be much more simple to take advantage of the DataFrame structure provided by the Pandas library.

The code was grouped into three wrapper classes: Loader, LinSolver, and Routines, where the first two represent the core data handling and numerical algorithms, and the last one is a collection of functions that use the existing framework to produce results.

The LinSolver class works with a clean subset of data to yield a solution for a telescope-telescope pair of constant POP offset. Its most important feature is the `LinSolver.solve_linear_system` function that takes as input a matrix of coefficients and a vector of delay differences, and applies the `numpy.linalg.lstsq` algorithm to them. The other helper functions provide a way to extract the matrix and vector coeffiecients from a passed DataFrame object. The class was designed, in principle, to require only a single subset of data as input, and all of the required functionality for manipulation and solution of the resulting system is contained within it. It was not implemented to be fully closed however, due to the open-ended nature of the problem, so the functions are still accessible outside the class.

On creation, the Loader class imports locally saved data from `./data/filename` where filename can stand for either `2012_S1W2.csv` or `2012_all.csv`, depending on the problem being solved. It provides functionality for manipulating the raw data on a more general level, in order to be able to pass it on to the relevant parts of the program that handle the solution procedures. This is to avoid functions passing between them copies of a large dataset when only a fraction of it is required.

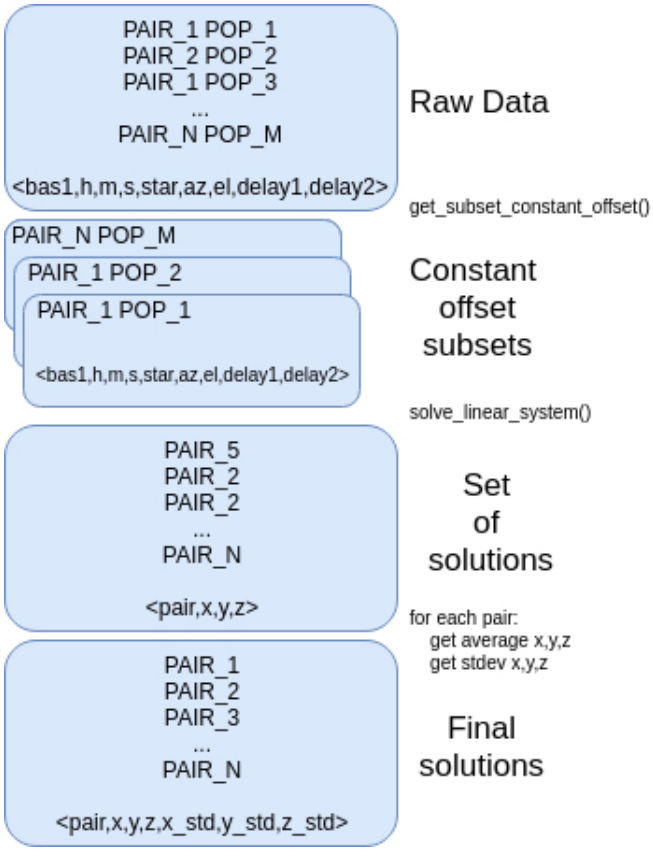

Figure 2: Simplified flowchart of the employed algorithm. The program begins executing from top, starting with the raw dataset and applying a sequence of functions (pseudocode), before yielding the final results. The actual numbers from the last box can be seen in Table (1).

The overall workflow of the program is the following. For the solution of the S1W2 telescope pair (`2012_S1W2.csv`) the raw data is loaded and the az,el,delay1,delay2 subset is picked. Since the data has been adjusted to account for the changing POP setting every night, the offset is constant and the data is ready for regression fitting, yielding a single set of displacement coordinates with high precision.

The same solution is then extended to the main dataset, through the functions in the `Routines` class. The data is first split according to constant offsets, which is the same as splitting by night and telescope pair. There are multiple underconstrained cases which are filtered out before passing the list of datasets to the solution algorithm. At this point, the solutions

can be saved using the functionality provided from `Loader.save_persistent_data` because these results can be used between subsequent runs and in further testing. We then have the relative position of every telescope pair as calculated at different nights of measurement. Outlier values are removed by grouping the solutions into a histogram and keeping only those that lie within 3 sigmas of the most frequent value. This approach was adopted because the outliers are spread over a wide range of values, and they strongly influence the mean. The final result is presented as the average of the telescope-telescope pair solutions for each night, and the standard deviation was calculated by the statistical spread in resulting values.

Results and Discussion

The final results for all pairs of telescopes can be seen in Table (1). The coordinate system ends up being centered on the first telescope in a pair because delay 1 and delay 2 correspond to the delays from the first and second telescopes in a pair in the data given.

Correctness of the results can be guaranteed from double checking with the official data on the relative displacements of the CHARA array [1]. Another way would be using the data for the S1W2 pair as given in our project manual to check that the algorithm produces the correct results for this pair. The latter method however can only be used for checking absolute values of (x,y,z) because the orientation of the baseline in the manual does not agree with the article by Pedretti et al.

These results were calculated after removing outlier solutions from some measurement nights, which were found to deviate significantly from the majority of resulting values. This was done with the `Routines.remove_outliers_1` routine which removes data that lies further than three standard deviations from the most frequent value for the displacement. The overall results could be improved by searching for a subset of data within the problematic measurement days that yields a realistic solution, thus removing any datapoints that cause the error. Another way would be to implement a solution that distinguishes between the POP offset and the fixed path difference between different pairs of telescopes, which would allow adjustment for POP on a night-by-night basis. However, this additional degree of freedom seems undefined in this dataset, so this approach was not adopted.

The results found here are more precise than the official ones for 2009, but the mean values do not agree within error. A possible explanation is seismic activity, as California experiences over 10000 earthquakes per year.

To investigate this, a routine was written that fits a linear function to the solutions for the coordinate displacements of every pair of telescopes over an year. An example fit if shown in Figure (3). The average gradient comes out as $-7.7 \times 10^{-6} \frac{meters}{day}$ and the standard deviation as $4.6 \times 10^{-5} \frac{meters}{day}$. This implies that there is a negligible overall trend of decreasing distance, but the large standard deviation shows that any relative coordinate could increase or decrease on a yearly basis. Over the course of three years, which is the period between the official results and the calculations here, assuming that the coordinate creep due to seismic activity is roughly equal to the standard deviation, the coordinate difference is reckoned to be on the centimeter scale, which is of the same order as the discrepancy between the official results for 2009 and the results for 2012 presented in this report.

Table 1: Displacement data between different pairs of telescopes along with error estimates. The y-axis points North and the x-axis is East. The coordinate system is centered on the first telescope in a pair. The last column shows how many datapoints were used in calculating the standard deviations.

pairs	x/m	y/m	z/m	x_std/m	y_std/m	z_std/m	data_number
S1E2	70.394	269.715	-2.799	0.004	0.003	0.005	24
S1W2	-69.091	199.335	0.467	0.003	0.005	0.006	29
E2W2	-139.487	-70.381	3.268	0.006	0.007	0.019	56
S1E1	125.332	305.934	-5.912	0.005	0.004	0.010	73
E1W2	-194.421	-106.601	6.387	0.012	0.011	0.040	36
S1S2	-5.746	33.581	0.638	0.002	0.003	0.006	17
E2W1	-245.466	-53.398	-7.986	0.008	0.012	0.013	22
W1W2	105.980	-16.984	11.258	0.003	0.005	0.007	34
E1E2	-54.937	-36.219	3.115	0.003	0.002	0.007	28
S1W1	-175.071	216.318	-10.789	0.005	0.007	0.009	59
E1W1	-300.403	-89.615	-4.876	0.011	0.013	0.034	77
S2E1	131.076	272.355	-6.549	0.007	0.007	0.015	15
S2E2	76.140	236.135	-3.437	0.005	0.005	0.008	41
S2W1	-169.327	182.737	-11.427	0.003	0.004	0.006	26
S2W2	-63.345	165.753	-0.171	0.003	0.004	0.006	39

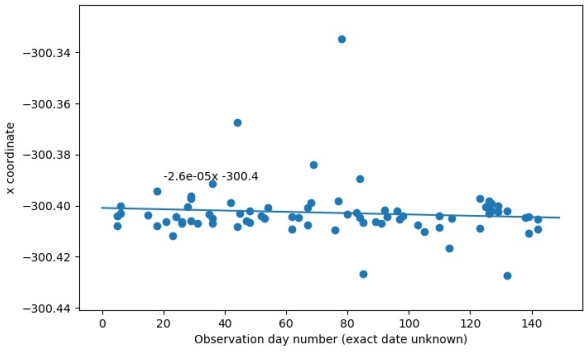

Figure 3: Change in x (East) displacement of E1W1 over 1 year.

The time complexity of this implementation is linear with respect to data volume, and the code produces a result in a few seconds. The two computationally heavy methods are `Routines.get_subset_constant_offset`, which splits the data into subsets of constant POP + telescope pair offset, and `LinSolver.solve_linear_system` which finds the best fit to the data.

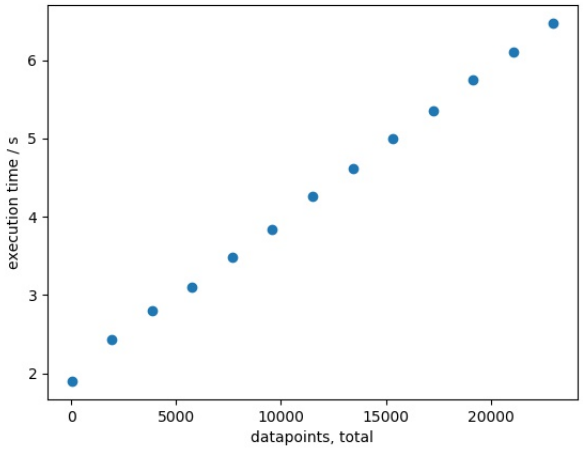

Figure 4: Runtime of the code for different number of datapoints. The time complexity is linear in data volume with gradient that can vary depending on level of optimisation.

For a linear regression problem, the time complexity is known as $O(np^2)$ where n is the number of datapoints (equations in the linear system), and p is the number of regression parameters, which are fixed at $p = 4$. The results from timed execution can be seen in Figure (4).

Conclusions

The data provided is enough for determining the relative telescope locations with a high degree of precision. The results have been found to vary slightly between different years of measurement, which is likely attributed to high seismic activity.

The proposed computational solution scales linearly with the data volume, which is not unexpected. The code can be optimised by reducing the number of Python level function calls, variables, and loops, and outsourcing as much as possible to the efficient Numpy and Pandas libraries. Pandas can be replaced by Numpy containers, which should be more lightweight, but seeing that results are produced in reasonable time, this will make the code unnecessarily complicated. Another way to boost performance would be to incorporate Cython, but it is not installed on University computers. Overall, due to the relatively small dataset, improving the efficiency of this implementation can do little in the way of extracting better results.

References

[1] E. Pedretti, J. D. Monnier, T. Ten Brummelaar, and N. D. Thureau. Imaging with the CHARA interferometer. *NEW ASTRONOMY REVIEWS*, 53(11-12):353–362, NOV-DEC 2009.

Appendix A Code

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import time

class Loader:
    def __init__(self, filename = "2012_S1W2.csv"):
        """ This class loads raw data and slices/modifies it in ways so as to be passed to the
            rest of the program.  """
        self.dataframe = pd.DataFrame.from_csv("./data/" + filename)  # Change if data not
        ↪  locally available
        self.day_index_dataframe()  # Add the day index on loading

    def get_linsolver_coef_raw_dataframe(self, dataframe):
        """ Picks subset of data to be passed to LinSolver. """
        dataframe_result = pd.DataFrame()
        dataframe_result["az"] = dataframe["az"]
        dataframe_result["el"] = dataframe["el"]
        dataframe_result["delay1"] = dataframe["delay1"]
        dataframe_result["delay2"] = dataframe["delay2"]
        dataframe_result.reset_index(drop=True, inplace=True)

        return dataframe_result

    def day_index_dataframe(self):
        """ Adds another column that indicates which day of measurements
            the data relates to.

            Performance of this function is not very important, because it
            is only used once for utility purposes.
        """

        #days_array = np.zeros((self.dataframe.size, 1), dtype=int)
        days_array = [0]*self.dataframe["h"].size

        current_day = 0

        for i in range(0, self.dataframe["h"].size - 1):
            if self.dataframe["h"].iloc[i+1] < self.dataframe["h"].iloc[i]:
                days_array[i] = current_day
                current_day += 1
            else:
                days_array[i] = current_day

        days_array[-1] = days_array[-2]

        self.dataframe["day_number"] = days_array


    @staticmethod
    def get_telescope_pop_pairs(dataframe):
        """ INPUT: A dataframe indexed by telescope-pop identifiers
            OUTPUT: A list of the unique telescope-pop identifiers

            Output from this function is used for grouping the data
            into clusters that have the same constant offset in delay
```

```
            measurements.

            Not enhanced for performance because only serves a utility
            purpose.
        """
        ids = []

        for name in dataframe.index:
            if name not in ids:
                ids.append(name)

        return ids

    def get_telescope_pairs(self, dataframe):
        """This function extracts the unique telescope pairs
        """

        pairs = []

        for identifier in dataframe.index:
            pair = identifier[2:-6]
            if pair not in pairs:
                pairs.append(pair)

        return pairs


    def save_persistent_data(self, dataframe):
        """
        This function is used once to save central
        results that persist between subsequent tests.
        """

        dataframe.to_csv(path_or_buf = "results.csv", index = False)

    def load_persistent_data(self):
        """
        This supporting function loads locally saved
        processed data to speed up subsequent processing.
        """

        return pd.read_csv("results.csv")

class LinSolver:

    def __init__(self):
        """ This class handles the solution of a large set of linear equations, and
                and provides functionality for error analysis.  """
        pass

    def parse_matrix_coef_from_data(self, dataframe):
        """ Calculates the coefficients matrix of linear equations from slice of original data.
        ↪  """
        coefs_list = []  # List so that it can be dynamically extended. TODO implement more
        ↪  efficient data structure

        """
        The following procedure could be implemented in a more efficient
```

```
        way, without using Python lists. I went with this approach because
        lists can be dynamically extended as opposed to arrays, which is
        convenient.

        I implement the formula:

            cos(el)cos(az)*x + cos(el)sin(az)*y + sin(el)*z + A = d2 - d1

        as derived from the geometry of the problem.

        A is an unknown constant offset in the delays, as discussed in the
        manual. I fit for it as well, giving it a constant coefficient of 1
        for every data point.
        """

        # The following lambda function returns [coef_x, coef_y, coef_z, 1]
        add_coefs_to_local_array = lambda series : coefs_list.append(
            [np.cos(np.radians(series["el"])) * np.sin(np.radians(series["az"])),
             np.cos(np.radians(series["el"])) * np.cos(np.radians(series["az"])),
             np.sin(np.radians(series["el"])),
             1]
            )

        dataframe.apply(add_coefs_to_local_array, axis=1)

        return np.array(coefs_list)

    def parse_delays_vector_from_data(self, dataframe):
        """ Creates a vector of delay differences to be passed on
        to the solution routine, from the original data.  """

        result_series = dataframe["delay2"] - dataframe["delay1"]
        result_series.reset_index(drop=True, inplace=True)

        return np.array(result_series)

    def solve_linear_system(self, A, b, rcond=None):
        """ This function applies the numpy lstsq routine to the
            parsed data and returns a solution that minimises the
            square error """

        return np.linalg.lstsq(A, b, rcond)

    #TODO Extract uncertainties in positions from residual


class Routines:
    """
    Wrapper class for the numerical routines and results.
    """

    def get_subset_constant_offset(self, dataframe):
        """
        Routine for grouping the raw data into subsets of
        constant delay offset

        Data has to be grouped once into subsets of same
        telescope-telescope and POP configuration pairs,
```

```
    and then these subsets are split into different
    data for different days. This ensures that the
    regression analysis is fitting data that corres-
    ponds to costant offsets as POP delay varies on day.
    """
    pairs = Loader.get_telescope_pop_pairs(dataframe)

    subsubsets = []
    for pair in pairs:
        subset = dataframe.loc[[pair]]
        unique_days = []
        for day in subset["day_number"]:
            if day not in unique_days:
                unique_days.append(day)
                subsubsets.append(subset.loc[subset["day_number"] == day])

    return subsubsets

def filter_underconstrained(self, subsets): #  O(n) in subset number

    results = []
    for s in subsets:
        if s.index.size > 4:
            results.append(s)

    return results


def solution_routine(self, subsets):
    l = Loader("2012_all.csv")
    ls = LinSolver()

    xs = []
    ys = []
    zs = []
    const = []
    res = []
    pairs = []
    data_number = []
    day_number = []

    subsets = self.filter_underconstrained(subsets)  # O(n) in subset number

    for s in subsets:
        pairs.append((s.index)[0][2:-6])
        A = ls.parse_matrix_coef_from_data(s)
        b = ls.parse_delays_vector_from_data(s)
        solution = ls.solve_linear_system(A,b)
        xs.append(solution[0][0])
        ys.append(solution[0][1])
        zs.append(solution[0][2])
        const.append(solution[0][3])
        res.append(solution[1])
        data_number.append(s.index.size)
        day_number.append(s["day_number"].iloc[0])

    results = pd.DataFrame({"day_number": day_number, "pair": pairs,"x": xs,"y": ys,"z":
    ↪  zs,"offset": const,"residue": res, "data_number": data_number})
```

```
    results = results[["pair","x","y","z","offset","residue","data_number","day_number"]]
    results["baseline"] = np.sqrt(np.power(results["x"],2) + np.power(results["y"],2) +
    ↪  np.power(results["z"],2))

    return results

def get_coordinates(self, dataframe):
    """
    INPUT: Dataframe of solutions for different
    constant delays offsets, but a single telescope
    pair

    OUTPUT: A single solution for the telescope pair
    """

    #Exclude underdefined solutions (with less than 4 datapoints)
    dataframe = dataframe.drop(dataframe.loc[dataframe["data_number"] < 4].index)

    pair = "underconstrained"
    x = 0
    y = 0
    z = 0

    x_std = 0
    y_std = 0
    z_std = 0

    data_number = 0

    if dataframe.index.size == 0:
        pass
    else:
        pair = dataframe["pair"].iloc[0]
        x = np.average(dataframe["x"])
        y = np.average(dataframe["y"])
        z = np.average(dataframe["z"])
        x_std = np.std(dataframe["x"])
        y_std = np.std(dataframe["y"])
        z_std = np.std(dataframe["z"])
        data_number = dataframe.index.size

    return [pair,x,y,z,x_std,y_std,z_std,data_number]

def remove_outliers(self, dataframe):
    """
    INPUT: Dataframe of pair,x,y,z values

    OUTPUT: Dataframe of pair,x,y,z values with
    outliers in x,y,z removed
    """

    current = dataframe["x"].iloc[0]
    flagged_indices = []
    for i in range(0, dataframe["x"].size):
        if current - dataframe["x"].iloc[i] > 1:
            flagged_indices.append(dataframe.index[i])
```

```
    dataframe.drop(labels=flagged_indices, inplace=True)

    current = dataframe["y"].iloc[0]
    flagged_indices = []
    for i in range(0, dataframe["y"].size):
        if current - dataframe["y"].iloc[i] > 1:
            flagged_indices.append(dataframe.index[i])

    dataframe.drop(labels=flagged_indices, inplace=True)

    current = dataframe["z"].iloc[0]
    flagged_indices = []
    for i in range(0, dataframe["z"].size):
        if current - dataframe["z"].iloc[i] > 1:
            flagged_indices.append(dataframe.index[i])

    dataframe.drop(labels=flagged_indices, inplace=True)

    return dataframe


def remove_outliers_coord(self, dataframe, coordinate_name):
    """
    INPUT: dataframe of pair, x, y, z values; "x", "y", or "z"
    OUTPUT: dataframe with outliers in this particular coordinate
    removed

    If the dataframe is empty, return it.
    """

    if(dataframe.index.size == 0):
        print("returned empty dataframe..")
        return dataframe

    else:
        bins = np.arange(-330,330)

        hist, hist_edges = np.histogram(dataframe[coordinate_name], bins=bins)
        value_lower = hist_edges[np.argmax(hist)]
        subset = dataframe.loc[dataframe[coordinate_name] > value_lower]
        subset = subset.loc[subset[coordinate_name] < (value_lower + 1)]

        average = np.average(subset[coordinate_name])
        stdev = np.std(subset[coordinate_name])

        subset = subset.loc[subset[coordinate_name] > (average - 3 * stdev)]
        subset = subset.loc[subset[coordinate_name] < (average + 3 * stdev)]

        return subset



def remove_outliers_1(self, dataframe):
    """
    INPUT: Dataframe of pair, x, y, z values
    OUTPUT: Dataframe of pair,x,y,z, values and
    outliers in x,y,z removed.
```

```
    This is a different implementation of remove_outliers.

    Here, a histogram is used to detect in what range the
    answer lies. In order to avoid crossover between bins
    in cases where the value is close to the bin edge, the
    algorithm computes the average value of the bin and
    runs through the entire dataset, thresholding the values
    to only those that lie within +- 1 of this value, which
    is much higher than the sigma of the value distribution.
    """

    for coord in ["x", "y", "z"]:
        dataframe = self.remove_outliers_coord(dataframe, coord)

    return dataframe


def solution_routine_s1w2(self, dataframe):
    l = Loader()
    ls = LinSolver()

    xs = []
    ys = []
    zs = []
    res = []
    day_number = []
    data_number = []

    for day in range(dataframe["day_number"].iloc[dataframe.index.size - 1]):
        s = dataframe.loc[dataframe["day_number"] == day]
        A = ls.parse_matrix_coef_from_data(s)
        b = ls.parse_delays_vector_from_data(s)
        solution = ls.solve_linear_system(A,b)
        xs.append(solution[0][0])
        ys.append(solution[0][1])
        zs.append(solution[0][2])
        res.append(solution[1])
        day_number.append(day)
        data_number.append(s.index.size)

    results = pd.DataFrame({"x": xs,"y": ys,"z": zs, "residue": res, "data_number":
    ↪  data_number, "day_number": day_number})

    results = results[["x","y","z","day_number", "residue","data_number"]]
    results["baseline"] = np.sqrt(np.power(results["x"],2) + np.power(results["y"],2) +
    ↪  np.power(results["z"],2))

    return results

def get_seizmic_avg_std(self):
    l = Loader("2012_all.csv")
    ls = LinSolver()
    r = Routines()

    pairs = l.get_telescope_pairs(l.dataframe)

    subsets = r.get_subset_constant_offset()
    results = r.solution_routine(subsets)
```

```
        gradients = []
        for pair in pairs:
            results_pair = results.loc[results["pair"] == pair]
            results_pair = r.remove_outliers_1(results_pair)
            for coord in ["x","y","z"]:
                coefs = np.polyfit(results_pair["day_number"], results_pair[coord], 1)
                gradients.append(coefs[0])

        results_pair = results.loc[results["pair"] == "E1W2"]
        results_pair = r.remove_outliers_1(results_pair)

        return np.average(gradients), np.std(gradients)


def main():

# Paste some test code here, or try writing your own!

if __name__ == "__main__":
    main()
```

Appendix B Tests

```
    """ Test: timing execution
    data_numbers = []
    times = []
    for i in np.linspace(10, 23000, 13):

        i = int(i)
        t1 = time.time()

        l = Loader("2012_all.csv")
        l.dataframe = l.dataframe.iloc[:i]

        ls = LinSolver()
        r = Routines()
        pairs = l.get_telescope_pairs(l.dataframe)

        subsets = r.get_subset_constant_offset(l.dataframe)
        results = r.solution_routine(subsets)

        pairs_df = []
        xs = []
        ys = []
        zs = []
        xs_std = []
        ys_std = []
        zs_std = []
        data_number = []

        for pair in pairs:
            dataframe = results.loc[results["pair"] == pair]  # Select data
            dataframe = r.remove_outliers_1(dataframe)
            coords = r.get_coordinates(dataframe)
            pairs_df.append(coords[0])
            xs.append(coords[1])
```

```
            ys.append(coords[2])
            zs.append(coords[3])
            xs_std.append(coords[4])
            ys_std.append(coords[5])
            zs_std.append(coords[6])
            data_number.append(coords[7])

        result = pd.DataFrame({"pairs":pairs_df, "x":xs, "y":ys, "z":zs, "x_std": xs_std,
↪   "y_std": ys_std, "z_std": zs_std, "data_number":data_number})
        result = result[["pairs","x","y","z","x_std","y_std","z_std","data_number"]]
        result.round(decimals=2)

        t2 = time.time()

        data_numbers.append(i)
        times.append(t2-t1)

    plt.scatter(data_numbers, times)
    plt.xlabel("datapoints, total")
    plt.ylabel("execution time / s")

    plt.show()

    """




    """ Final results
    l = Loader("2012_all.csv")
    ls = LinSolver()
    r = Routines()

    pairs = l.get_telescope_pairs(l.dataframe)

    subsets = r.get_subset_constant_offset(l.dataframe)
    results = r.solution_routine(subsets)


    pairs_df = []
    xs = []
    ys = []
    zs = []
    xs_std = []
    ys_std = []
    zs_std = []
    data_number = []

    for pair in pairs:
        dataframe = results.loc[results["pair"] == pair]  # Select data
        dataframe = r.remove_outliers_1(dataframe)
        coords = r.get_coordinates(dataframe)
        pairs_df.append(coords[0])
        xs.append(coords[1])
        ys.append(coords[2])
        zs.append(coords[3])
        xs_std.append(coords[4])
```

```
        ys_std.append(coords[5])
        zs_std.append(coords[6])
        data_number.append(coords[7])

    result = pd.DataFrame({"pairs":pairs_df, "x":xs, "y":ys, "z":zs, "x_std": xs_std, "y_std":
↪   ys_std, "z_std": zs_std, "data_number":data_number})
    result = result[["pairs","x","y","z","x_std","y_std","z_std","data_number"]]
    result.round(decimals=2)
    result.to_csv(path_or_buf = "final_results.csv", index=False)

    print(result)
    """









    """ S1W2 residual buildup
    l = Loader()
    ls = LinSolver()

    i_vals = []
    residuals = []
    for i in range(10,700):
        dataframe = l.get_linsolver_coef_raw_dataframe(l.dataframe)[:i]
        A = ls.parse_matrix_coef_from_data(dataframe)
        b = ls.parse_delays_vector_from_data(dataframe)
        i_vals.append(i)
        residuals.append(ls.solve_linear_system(A,b)[1][0])

    plt.plot(i_vals, residuals)
    plt.title("Residual vs data points used data[:x]")
    plt.xlabel("number of data points")
    plt.ylabel("residual")
    plt.show()
    """







    """ Show different POP settings can be fitted with same A
    l = Loader("2012_all.csv")
    ls = LinSolver()

    l.day_index_dataframe()
    subset = l.dataframe.loc["D/S1W2P15B21"]

    A = ls.parse_matrix_coef_from_data(l.get_linsolver_coef_raw_dataframe(subset))
    b = ls.parse_delays_vector_from_data(subset)
    print(ls.solve_linear_system(A,b))

    subset = l.dataframe.loc["D/S1W2P45B13"]

    A = ls.parse_matrix_coef_from_data(l.get_linsolver_coef_raw_dataframe(subset))
```

```
b = ls.parse_delays_vector_from_data(subset)
print(ls.solve_linear_system(A,b))
"""




""" Show different POP settings for S1W2. There seems to be an outlier.
l = Loader("2012_all.csv")
ls = LinSolver()

l.day_index_dataframe()
subset = l.dataframe.loc["D/S1W2P15B21"]

A = ls.parse_matrix_coef_from_data(l.get_linsolver_coef_raw_dataframe(subset))
b = ls.parse_delays_vector_from_data(subset)
print(ls.solve_linear_system(A,b))

subset = l.dataframe.loc["D/S1W2P45B13"]

A = ls.parse_matrix_coef_from_data(l.get_linsolver_coef_raw_dataframe(subset))
b = ls.parse_delays_vector_from_data(subset)
print(ls.solve_linear_system(A,b))

subset = l.dataframe.loc["D/S1W2P15B32"]

A = ls.parse_matrix_coef_from_data(l.get_linsolver_coef_raw_dataframe(subset))
b = ls.parse_delays_vector_from_data(subset)
print(ls.solve_linear_system(A,b))
print(subset)
"""




"""Solve for specific days
l = Loader("2012_all.csv")
ls = LinSolver()

l.day_index_dataframe()
df = l.dataframe
subset_all = df.loc["D/W1W2P25B31"]

subset_day34 = subset_all.loc[subset_all["day_number"] == 34]
subset_day144 = subset_all.loc[subset_all["day_number"] == 144]
subset_day143 = subset_all.loc[subset_all["day_number"] == 143]

print(subset_all.size)
print(subset_day34.size)
print(subset_day144.size)
print(subset_day143.size)

A = ls.parse_matrix_coef_from_data(l.get_linsolver_coef_raw_dataframe(subset_all))
```

```
b = ls.parse_delays_vector_from_data(subset_all)
print(ls.solve_linear_system(A,b))

A = ls.parse_matrix_coef_from_data(l.get_linsolver_coef_raw_dataframe(subset_day34))
b = ls.parse_delays_vector_from_data(subset_day34)
print(ls.solve_linear_system(A,b))

A = ls.parse_matrix_coef_from_data(l.get_linsolver_coef_raw_dataframe(subset_day144))
b = ls.parse_delays_vector_from_data(subset_day144)
print(ls.solve_linear_system(A,b))

A = ls.parse_matrix_coef_from_data(l.get_linsolver_coef_raw_dataframe(subset_day143))
b = ls.parse_delays_vector_from_data(subset_day143)
print(ls.solve_linear_system(A,b))
"""
```

```
""" Filtered solutions
l = Loader("2012_all.csv")
ls = LinSolver()
r = Routines()

pairs = l.get_telescope_pairs(l.dataframe)

subsets = r.get_subset_constant_offset(l.dataframe)
results = r.solution_routine(subsets)

for pair in pairs:
    dataframe = results.loc[results["pair"] == pair]  # Select data
    dataframe = r.remove_outliers_1(dataframe)
    print(r.get_coordinates(dataframe))
"""
```

```
""" Seismic linear fit
l = Loader("2012_all.csv")
ls = LinSolver()
r = Routines()

pairs = l.get_telescope_pairs(l.dataframe)

subsets = r.get_subset_constant_offset(l.dataframe)
results = r.solution_routine(subsets)


for pair in pairs:
    results_pair = results.loc[results["pair"] == pair]
    results_pair = r.remove_outliers_1(results_pair)
    for coord in ["x","y","z"]:
```

```
        coefs = np.polyfit(results_pair["day_number"], results_pair[coord], 1)
        print(coefs, pair, coord)

results_pair = results.loc[results["pair"] == "E1W2"]
results_pair = r.remove_outliers_1(results_pair)

plt.scatter(results_pair["day_number"], results_pair["z"])
plt.show()
"""




""" Saving persistent data
r = Routines()
l = Loader("2012_all.csv")

pairs = l.get_telescope_pairs(l.dataframe)

# Save data first, then load to get faster performance on further tests
# This allows for a computationally heavy operation to be skipped

#subsets = r.get_subset_constant_offset(l.dataframe)
#results = r.solution_routine(subsets)

results = l.load_persistent_data()
#l.save_persistent_data(results)  # Saves time

for pair in pairs:
    pair_dataframe = results.loc[results["pair"] == pair]
    pair_dataframe = r.remove_outliers_1(pair_dataframe)
    print(r.get_coordinates(pair_dataframe))
"""
```