Documentation

tsmoffat

September 2016

1 Introduction

The aim of this document is to explain the code in the program and how to run it etc. It will be in lieu of comments within the code, due to the large amount of repeated code within the program

2 Installation

To install the program, make sure you have Git installed on your system as well as Git LFS (for large files). Get the URL of the project from GitHub, navigate to where you would like the project to be stored and run git clone (URL) in your terminal/command line. This will pull the project from GitHub. Then type git lfs track "*.txt" into your terminal to make sure that the large file system is tracking text files, before running git clone (URL) again to download the text files for this program.

To run the program, install Anaconda for Python 3.5 from `https://www.continuum.io/downloads`. This will make sure that everything is installed correctly. Once this is installed, restart your terminal then type "pip install tabulate" to install the missing package needed for this program.

3 Running

There are two options for this. Either navigate to the directory containing the project in your terminal and type Testing2.py or load the project up in PyCharm and run it from there.

4 Code Explanation

This section is intended to explain what the code in the program does, as quite a lot of it is repeated and it saves repeating comments. The self arguments that every function has refer to global variables and constants.

4.1 AttenuationSearch.py

```python
"""A module to find the required attenuation."""
import decimal as dec


def attlist(self):
    """Generate a list of all the values in the attenuation sheet."""
    listatt = []
    for row in self.dsa.iter_rows(row_offset=2):
        if row[self.dsas2128].value is not None:
            # Adds value to list if it actually has a value
            listatt.append(row[self.dsas2128].value)
    return listatt


def closest(self, attlist):
    """Find closest attenuation to target."""
    closest = min(attlist, key=lambda x: abs(
        dec.Decimal(x) - dec.Decimal(self.targetatt)))
    closest = dec.Decimal(closest)
    return closest


def attenuationsearch(self, attlist, closest):
    """Search for the most accurate attenuation."""
    for row in self.dsa.iter_rows():
        if row[self.dsas2128].value == closest:
            att2128 = closest.quantize(dec.Decimal(
                '.001'), rounding=dec.ROUND_HALF_UP)
            row28 = row[self.dsastate28].value
            att2124 = dec.Decimal(row[self.dsas2124].value).quantize(
                dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
            att2132 = dec.Decimal(row[self.dsas2132].value).quantize(
                dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
            phase2124 = dec.Decimal(row[self.dsaphase2124].value).quantize(
                dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
            phase2128 = dec.Decimal(row[self.dsaphase2128].value).quantize(
                dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
            phase2132 = dec.Decimal(row[self.dsaphase2132].value).quantize(
                dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)

```

```
41        return {'row28': row28, 'att2124': att2124, 'att2128': att2128, 'att2132': att2132, 'phase2124': phase2124, 'phase2128': phase2128, 'phase2132': phase2132}
```

This is AttenuationSearch.py. Its purpose is to search through the DSA sheet in the spreadsheet and find the closest attenuation to the input attenuation. The first function, attlist, just generates a list of all the values present in the DSA sheet, then returns it. It does this by iterating through the column with the S21 values of attenuation, checking if they have a value, and appending them to the list if they do. The row offset in line 8 is to remove anything that is not a number from the search area.

The next function, closest, takes as its argument attlist, which is the list generated in the previous function. It uses a lambda function to find the value with the smallest absolute distance from the target attenuation, which is called through self.targetatt. This is then converted into a decimal value, as this allows for rounding, unlike a float, and then returns the value.

The last function takes as its arguments attlist and closest, and then it iterates through the rows, finding which value is equal to the closest value then gets the values needed for various other parts of the program from the spreadsheet. All these are then returned in a data dictionary, which is a data type in python that allows for values to be referenced using a key.

4.2 extras.py

```python
"""Literally some magic."""
import decimal as dec


def check180(self, set180):
    """180 degrees of magic."""
    dec.getcontext().prec = 6
    if self.targetphase in set180:
        for row in self.s180.iter_rows():
            if row[self.phase28].value == self.targetphase:
                row1 = int(row[0].value)
                att1 = dec.Decimal(row[self.att28].value)
                phaselow = dec.Decimal(row[self.phase24].value).quantize(
                    dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
                phasehigh = dec.Decimal(row[self.phase32].value).quantize(
                    dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
                phasediff = phasehigh - phaselow

        return {'phase1': self.targetphase, 'row1': row1, 'att1': att1, 'phasediff': phasediff, 'source1': 's180', 'totalphase': att1, 'total': self.targetphase}
    else:
        closest = min(set180, key=lambda x: abs(
```

```
            dec.Decimal(x) - dec.Decimal(self.targetphase)))
        for row in self.s180.iter_rows():
            if row[self.phase28].value == closest:
                closestround = closest.quantize(
                    dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
                row1 = int(row[0].value)
                att1 = dec.Decimal(row[self.att28].value).quantize(
                    dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
                phaselow = dec.Decimal(row[self.phase24].value).quantize(
                    dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
                phasehigh = dec.Decimal(row[self.phase32].value).quantize(
                    dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
                phasediff = phasehigh - phaselow

        return {'phase1': closestround, 'row1': row1, 'att1': att1, 'phasediff': phasediff, 'source1': 's180', 'totalphase': att1, 'total': closestround}


def check90(self, set90):
    """90 degrees of magic."""
    dec.getcontext().prec = 6
    if self.targetphase in set90:
        for row in self.s90.iter_rows():
            if row[self.phase28].value == self.targetphase:
                row1 = int(row[0].value)
                att1 = dec.Decimal(row[self.att28].value).quantize(
                    dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
                phaselow = dec.Decimal(row[self.phase24].value).quantize(
                    dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
                phasehigh = dec.Decimal(row[self.phase32].value).quantize(
                    dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
                phasediff = phasehigh - phaselow

        return {'phase1': self.targetphase, 'row1': row1, 'att1': att1, 'phasediff': phasediff, 'source1': 's90', 'totalphase': att1, 'total': self.targetphase}

    else:
        closest = min(set90, key=lambda x: abs(
            dec.Decimal(x) - dec.Decimal(self.targetphase)))
```

```
        for row in self.s90.iter_rows():
            if row[self.phase28].value == closest:
                closestround = closest.quantize(
                    dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
                row1 = int(row[0].value)
                att1 = dec.Decimal(row[self.att28].value).quantize(
                    dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
                phaselow = dec.Decimal(row[self.phase24].value).quantize(
                    dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
                phasehigh = dec.Decimal(row[self.phase32].value).quantize(
                    dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
                phasediff = phasehigh - phaselow

        return {'phase1': closestround, 'row1': row1, 'att1': att1, 'phasediff': phasediff, 'source1': 's90', 'totalphase': att1, 'total': closestround}


def check45(self, set45):
    """45 Degrees of magic.

    Parameters
    ----------
    self, set45, set45
    Returns
    -------
    phase1, row1, att1, phasediff
    """
    dec.getcontext().prec = 6
    if self.targetphase in set45:
        for row in self.s45.iter_rows():
            if row[self.phase28].value == self.targetphase:
                row1 = int(row[0].value)
                att1 = dec.Decimal(row[self.att28].value).quantize(
                    dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
                phaselow = dec.Decimal(row[self.phase24].value).quantize(
                    dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
                phasehigh = dec.Decimal(row[self.phase32].value).quantize(
                    dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
                phasediff = phasehigh - phaselow

```

```
        return {'phase1': self.targetphase, 'row1': row1, 'att1': att1, 'phasediff': phasediff, 'source1': 's45', 'totalphase': att1, 'total': self.targetphase}

    else:
        closest = min(set45, key=lambda x: abs(
            dec.Decimal(x) - dec.Decimal(self.targetphase)))
        for row in self.s45.iter_rows():
            if row[self.phase28].value == closest:
                closestround = closest.quantize(
                    dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
                row1 = int(row[0].value)
                att1 = dec.Decimal(row[self.att28].value).quantize(
                    dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
                phaselow = dec.Decimal(row[self.phase24].value).quantize(
                    dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
                phasehigh = dec.Decimal(row[self.phase32].value).quantize(
                    dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
                phasediff = phasehigh - phaselow

        return {'phase1': closestround, 'row1': row1, 'att1': att1, 'phasediff': phasediff, 'source1': 's45', 'totalphase': att1, 'total': closestround}


def check225(self, set225):
    """22.5 degrees of magic.

    Parameters
    ----------
    self, s225, set225
    Returns
    -------
    phase1, row1, att1, phasediff
    """
    dec.getcontext().prec = 6
    if self.targetphase in set225:
        for row in self.s225.iter_rows():
            if row[self.phase28].value == self.targetphase:
                row1 = int(row[0].value)
                att1 = dec.Decimal(row[self.att28].value).quantize(
                    dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
                phaselow = dec.Decimal(row[self.phase24].value).quantize(
```

```
                    dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
                phasehigh = dec.Decimal(row[self.phase32].value).quantize(
                    dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
                phasediff = phasehigh - phaselow

        return {'phase1': self.targetphase, 'row1': row1, 'att1': att1, 'phasediff': phasediff, 'source1': 's225', 'totalphase': att1, 'total': self.targetphase}

    else:
        closest = min(set225, key=lambda x: abs(
            dec.Decimal(x) - dec.Decimal(self.targetphase)))
        for row in self.s225.iter_rows():
            if row[self.phase28].value == closest:
                closestround = closest.quantize(
                    dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
                row1 = int(row[0].value)
                att1 = dec.Decimal(row[self.att28].value).quantize(
                    dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
                phaselow = dec.Decimal(row[self.phase24].value).quantize(
                    dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
                phasehigh = dec.Decimal(row[self.phase32].value).quantize(
                    dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
                phasediff = phasehigh - phaselow

        return {'phase1': closestround, 'row1': row1, 'att1': att1, 'phasediff': phasediff, 'source1': 's225', 'totalphase': att1, 'total': closestround}


def checkall(self, set180, set90, set45, set225):
    """Check to find most accurate solution so far.

    Parameters
    ----------
    self
    Returns
    -------
    bestsol data dict
    """
    dec.getcontext().prec = 6
    sol180 = check180(self, set180)
    sol90 = check90(self, set90)
```

```python
    sol45 = check45(self, set45)
    sol225 = check225(self, set225)
    bestsollist = [sol180['phase1'], sol90[
        'phase1'], sol45['phase1'], sol225['phase1']]

    if sol180['phase1'] == self.targetphase:
        return sol180

    elif sol90['phase1'] == self.targetphase:
        return sol90

    elif sol45['phase1'] == self.targetphase:
        return sol45

    elif sol225['phase1'] == self.targetphase:
        return sol45

    else:
        closest = min(bestsollist, key=lambda x: abs(
            dec.Decimal(x) - dec.Decimal(self.targetphase)))
        if closest == sol180['phase1']:
            return sol180
        elif closest == sol90['phase1']:
            return sol90
        elif closest == sol45['phase1']:
            return sol45
        elif closest == sol225['phase1']:
            return sol225
        else:
            return None


def mostaccurate(self, bestresult, bestresult2, bestresult3, bestresult4, sollist):
    """Find most accurate solution and return."""
    bestsol = min(sollist, key=lambda x: abs(
        dec.Decimal(x) - dec.Decimal(self.targetphase)))
    if bestsol == bestresult['total']:
        return bestresult
    elif bestsol == bestresult2['total']:
        return bestresult2
    elif bestsol == bestresult3['total']:
        return bestresult3
    elif bestsol == bestresult4['total']:
        return bestresult4
    else:
```

```
            return None
```

Extras.py is the multi-use module in this program. The first four functions are used to find if there is a single array answer. These all work in roughly the same way. First, the looks to see if the target phase is actually present at all in the spreadsheet. If it is then it returns that value, along with the corresponding row and attenuation. Otherwise, it uses a lambda function to find the closest value in the set (which is like a list but doesn't allow for duplicate values) and returns the values for that.

The checkall function is what controls the previous four functions. It calls them then decides which value to return, by looking to see if any of the returned values equal the target value, and if not finding the one with the smallest distance.

The final module in this takes the most accurate values from finding one, two, two, three and four phase answers and then returns the best one. the way it is laid out, if a value is found with two and three phases, the program will always prefer the more efficient solution (i.e. the one with the fewest phases involved).

4.3 FourPhase.py

```
"""Test all four phases in one."""
import decimal as dec
import os


def closest_finder(self):
    """Find the closest value to a given value."""
    combined = set(map(str.rstrip, open(
        (os.path.join(os.path.dirname(__file__), '1809045225.txt')))))
    closest = min(combined, key=lambda x: abs(
        dec.Decimal(x) - dec.Decimal(self.targetphase)))
    closest = dec.Decimal(closest)
    print(closest)
    return closest
    del combined


def check(self, set180, set90, set45, set225, closest):
    """Check to find the most accurate four phase solution."""
    dec.getcontext().prec = 6
    for i in set180:
        for j in set90:
            for k in set45:
                for l in set225:
                    total = i + j + k + l
                    if total == closest:
                        for row in self.s180:
                            if row[self.phase28].value == i:
```

```python
                row1 = int(row[0].value)
                att1 = dec.Decimal(row[self.att28].value).quantize(
                    dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
                phaselow1 = dec.Decimal(row[self.phase24].value).quantize(
                    dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
                phasehigh1 = dec.Decimal(row[self.phase32].value).quantize(
                    dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
                phasediff1 = phasehigh1 - phaselow1

        for row in self.s90:
            if row[self.phase28].value == j:
                row2 = int(row[0].value)
                att2 = dec.Decimal(row[self.att28].value).quantize(
                    dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
                phaselow2 = dec.Decimal(row[self.phase24].value).quantize(
                    dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
                phasehigh2 = dec.Decimal(row[self.att32].value).quantize(
                    dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
                phasediff2 = phasehigh2 - phaselow2

        for row in self.s45:
            if row[self.phase28].value == k:
                row3 = int(row[0].value)
                att3 = dec.Decimal(row[self.att28].value).quantize(
                    dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
                phaselow3 = dec.Decimal(row[self.att24].value).quantize(
                    dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
```

```
                phasehigh3 = dec.Decimal(row[self.att32].value).quantize(dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
                phasediff3 = phasehigh3 - phaselow3

        for row in self.s225:
            if row[self.phase28].value == l:
                row4 = int(row[0].value)
                att4 = dec.Decimal(row[self.att28].value).quantize(dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
                phaselow4 = dec.Decimal(row[self.phase24].value).quantize(dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
                phasehigh4 = dec.Decimal(row[self.phase32].value).quantize(dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
                phasediff4 = phasehigh4 - phaselow4

        totalatt = att1 + att2 + att3 + att4
        return {'phase1': i.quantize(dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP), 'row1': row1, 'att1': att1, 'phasediff1': phasediff1, 'source1': 's180', 'phase2': j.quantize(dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP), 'row2': row2, 'att2': att2, 'phasediff2': phasediff2, 'source2': 's90', 'phase3': k.quantize(dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP), 'row3': row3, 'att3': att3, 'phasediff3': phasediff3, 'source3': 's45', 'phase4': l.quantize(dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP), 'row4': row4, 'att4': att4, 'phasediff4': phasediff4, 'source4': 's225', 'total': total.quantize(dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP), 'totalatt': totalatt.quantize(dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)}
```

This is FourPhase.py. The first function, closestfinder, finds the closest value to the target value. The line starting "combined =" gets the path of the directory where the program currently is running and then loads the text file specified, before loading it into a set, which ensures there are no duplicate numbers present. The following line uses a lambda function, which is a function that isn't attached to an identifier, so is only called in that one spot. The purpose of this lambda function is to find the number with the smallest absolute difference from the target. These two numbers are converted into decimal format, which is like a floating point number but it allows for rounding, so the number isn't 64 bits. The closest value is then returned to whatever bit of code called it, and is then used later in the program.

The other function, check, is an enormous bit of code, but in essence what it does is iterates through the four lists of numbers available to it, s180, s90, s45 and s225, which correspond to the four spreadsheets of data. It then adds the four numbers together, one from each sheet, and checks to see if the total matches the closest value from the previous snippet. If the total matches then it it iterates through each sheet in turn, looking for the value of the corresponding iterator (e.g.it looks for the value of i in the 180 sheet). When it finds this value, it assigns the value of the row (the value in the zeroth column) to row(n), the value of the attenuation of that phase to att(n), then it works out the difference in phases between the phase at 32GHz and the phase at 24GHz. This is used in other parts of the program.

4.4 lookuptablegenerator.py

```
"""To generate a look-up table."""
import decimal as dec
import extras as ex
import TwoPhase as twp
import ThreePhase as thp
import FourPhase as fp
import tabulate
import os
import csv


def tablegen(self):
    """Generate look up table for phase."""
    list180 = []
    for row in self.s180.iter_rows(row_offset=2):
        if row[self.phase28].value is not None:
            list180.append(dec.Decimal(row[self.phase28].value))
    set180 = set(list180)

    list90 = []
    for row in self.s90.iter_rows(row_offset=2):
        if row[self.phase28].value is not None:
            list90.append(dec.Decimal(row[self.phase28].value))
    set90 = set(list90)
```

```python
        list45 = []
        for row in self.s45.iter_rows(row_offset=2):
            if row[self.phase28].value is not None:
                list45.append(dec.Decimal(row[self.phase28].value))
        set45 = set(list45)

        list225 = []
        for row in self.s225.iter_rows(row_offset=2):
            if row[self.phase28].value is not None:
                list225.append(dec.Decimal(row[self.phase28].value))
        set225 = set(list225)
        headers = ["Target value", "180 Used", "90 Used", "45 Used",
                   "22.5 Used", "180 Setting", "90 Setting", "45 Setting",
                   ↪  "22.5 Setting"]
        table = []
        for i in range(1, 65):
            self.targetphase = (-360 / 64) * i
            bestresult = ex.checkall(self, set180, set90, set45, set225)
            bestresult2 = twp.checkall(self, set180, set90, set45, set225)
            bestresult3 = thp.checkall(self, set180, set90, set45, set225)
            bestresult4 = fp.check(self, set180, set90, set45, set225)
            sollist = [bestresult['total'], bestresult2['total'],
                       bestresult3['total'], bestresult4['total']]
            bestphase = ex.mostaccurate(
                self, bestresult, bestresult2, bestresult3, bestresult4,
                ↪  sollist)
            s180present = 0
            s90present = 0
            s45present = 0
            s225present = 0
            print(bestphase)
            if 'source1' in bestphase:
                if bestphase['source1'] == 's180':
                    s180present = 1
                elif bestphase['source1'] == 's90':
                    s90present = 1
                elif bestphase['source1'] == 's45':
                    s45present = 1
                elif bestphase['source1'] == 's225':
                    s225present = 1

            if 'source2' in bestphase:
                if bestphase['source2'] == 's90':
                    s90present = 2
                elif bestphase['source2'] == 's45':
                    s45present = 2
```

```
        elif bestphase['source2'] == 's225':
            s225present = 2

    if 'source3' in bestphase:
        if bestphase['source3'] == 's45':
            s45present = 3
        elif bestphase['source3'] == 's225':
            s225present = 3

    if 'source4' in bestphase:
        if bestphase['source4'] == 's225':
            s225present = 4
    print(s180present, s90present, s45present, s225present)
    if s180present == 1:
        s180setting = '{0:02b}'.format(bestphase['row1'])
    else:
        s180setting = '{0:02b}'.format(2)

    if s90present == 1:
        s90setting = '{0:06b}'.format(bestphase['row1'])
    elif s90present == 2:
        s90setting = '{0:06b}'.format(bestphase['row2'])
        s90present = 1
    else:
        s90setting = '{0:06b}'.format(32)

    if s45present == 1:
        s45setting = '{0:09b}'.format(bestphase['row1'])
    elif s45present == 2:
        s45setting = '{0:09b}'.format(bestphase['row2'])
        s45present = 1
    elif s45present == 3:
        s45setting = '{0:09b}'.format(bestphase['row3'])
        s45present = 1
    else:
        s45setting = '{0:09b}'.format(256)

    if s225present == 1:
        s225setting = '{0:09b}'.format(bestphase['row1'])
    elif s225present == 2:
        s225setting = '{0:09b}'.format(bestphase['row2'])
        s225present = 1
    elif s225present == 3:
        s225setting = '{0:09b}'.format(bestphase['row3'])
        s225present = 1
    elif s225present == 4:
```

```python
            s225setting = '{0:09b}'.format(bestphase['row4'])
            s225present = 1
        else:
            s225setting = '{0:09b}'.format(256)

        endlist = [self.targetphase, s180present, s90present, s45present,
                   s225present, s180setting, s90setting, s45setting, s225setting]
        table.append(endlist)

    print(tabulate.tabulate(table, headers, tablefmt="fancy_grid"))
    with open(os.path.join(
            os.path.dirname(__file__), 'phasetable.csv'), 'w') as csvfile:
        writer = csv.writer(csvfile)
        [writer.writerow(r) for r in table]
    return


def atttablegen(self):
    """Generate a table for attenuation."""
    listatt = []
    for row in self.dsa.iter_rows(row_offset=2):
        if row[self.dsas2128].value is not None:
            listatt.append(dec.Decimal(row[self.dsas2128].value))
    setatt = set(listatt)
    headers = ["Target", "Attenuation", "Setting"]
    table = []
    for i in range(1, 65):
        self.targetatt = dec.Decimal((-24 / 64) * i)
        if self.targetatt in setatt:
            for row in self.dsa.iter_rows():
                if row[self.dsas2128].value == self.targetatt:
                    row28 = row[self.dsastate28].value
                    att2128 = self.targetatt.quantize(
                        dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
                    att2124 = dec.Decimal(row[self.dsas2124].value).quantize(
                        dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
                    att2132 = dec.Decimal(row[self.dsas2132].value).quantize(
```

```
153                         dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
154
155         else:
156             closest = min(listatt, key=lambda x: abs(
157                 dec.Decimal(x) - dec.Decimal(self.targetatt)))
158             closest = dec.Decimal(closest)
159             for row in self.dsa.iter_rows():
160                 if row[self.dsas2128].value == closest:
161                     att2128 = closest.quantize(dec.Decimal(
162                         '.001'), rounding=dec.ROUND_HALF_UP)
163                     row28 = row[self.dsastate28].value
164                     att2124 = dec.Decimal(row[self.dsas2124].value).quantize(
165                         dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
166                     att2132 = dec.Decimal(row[self.dsas2132].value).quantize(
167                         dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
168
169         DSAsetting = '{0:012b}'.format(row28)
170         table.append([self.targetatt, att2128, DSAsetting])
171
172     print(tabulate.tabulate(table, headers, tablefmt="fancy_grid"))
173     with open(os.path.join(
174             os.path.dirname(__file__), 'atttable.csv'), 'w') as csvfile:
175         writer = csv.writer(csvfile)
176         [writer.writerow(r) for r in table]
```

This module is supposed to be rarely used. Its purpose is to generate a table of 64 values of either attenuation or phase, and then output them in an easily readable way using the tabulate module.

Starting with the tablegen function. This generates sets for the different phases. It then creates a list of the headers used by tabulate. Then, it starts the main part of the function, where it iterates from 1 to 64 and stores this value in i, before calculating the appropriate multiple of $\frac{-360}{64}$ to be used in this run through the cycle. It then runs in much the same way as the main program (seen later) to get the values of the best results for that attenuation. Once this has been completed it goes through a very long series of if/else if statements to find the settings to output to the user. The settings themselves are calculated by taking the row number and formatting it into binary, with various lengths of leading bits. This starts on line 83. The relevant values are added to the table, which is a list of lists. Once the program has run through 64 times, it prints the table out with nice formatting, and then outputs all the values to a CSV file for easier browsing once the terminal window has been closed. atttablegen is very similar, it just looks for the attenuation as opposed to the phase.

4.5 minattvar.py

```python
"""Module to find minimum attenuation variation across frequency."""
import decimal as dec
from heapq import nsmallest
import AttenuationSearch as ats


def minattvar(self):
    """Search for minimum amplitude variation across frequency."""
    listatt = []
    totallist = []
    attvar = []
    # Iterates through all the rows in the spreadsheet from the third row
    # onwards
    for row in self.dsa.iter_rows(row_offset=2):
        if row[self.dsas2128].value is not None:
            # Adds the value to a list if it actually has a value, and isn't
            # blank
            listatt.append(row[self.dsas2128].value)
    closest_values = nsmallest(int(self.k), listatt, key=lambda x: abs(
        dec.Decimal(x) - dec.Decimal(self.targetatt)))  # Finds the k closest numbers to a given value from the list
    # Converts the values in closest_values to a decimal from a float. This
    # makes it easier to work with as it adds in rounding
    closest = [dec.Decimal(i) for i in closest_values]
    print(closest)
    for i in closest:
        # Calls the AttenuationSearch module to search for the various values.
        # It does this for each item in the closest list
        resultsdict = ats.attenuationsearch(self, listatt, i)
        variation = resultsdict['att2132'] - resultsdict['att2124']
        # Adds a new value to the dictionary, with the key variation
        resultsdict['variation'] = variation
        totallist.append(resultsdict)
        attvar.append(variation)
    print(totallist)
    print(attvar)
    # Finds the smallest number in the list
    minattdiff = min(abs(i) for i in attvar)
    for m in totallist:
        if abs(m['variation']) == minattdiff:
```

```
            # If the value in the dictionary/key combo m['variation'] matches
            # the minimum value, it gets returned and printed
            return m
```

This module searches for the attenuation value with the minimum variation across frequency. It does this by finding the n closest values, n being an integer specified by the user, and calling the attenuation search module then using the phase value from calling attenuation search. The attenuation for 24GHz is subtracted from the attenuation for 32GHz, and then it finds the smallest absolute difference and returns the dictionary with that value in.

4.6 mininsertloss.py

```python
"""Find the minimum insertion loss, i.e., the minimum attenuation for a given phase."""
import decimal as dec
import os
from heapq import nsmallest
import FourPhase as fp


def mininsertloss(self, set180, set90, set45, set225):
    """Find minimum insertion loss for a phase."""
    dec.getcontext().prec = 6
    combined = set(map(dec.Decimal, open((os.path.join(
        os.path.dirname(__file__), '1809045225.txt')))))
    closest_values = nsmallest(int(self.k), combined, key=lambda x: abs(
        x - dec.Decimal(self.targetphase)))
    closest = [dec.Decimal(i) for i in closest_values]
    total_values = []
    insertlosslist = []
    del combined
    for item in closest:
        resultsdict = fp.check(self, set180, set90, set45, set225, item)
        insertlosslist.append(resultsdict['totalatt'])
        total_values.append(resultsdict)
    print(total_values)
    mininsertloss = min(abs(i) for i in insertlosslist)
    for m in total_values:
        print(m['totalatt'])
        if abs(m['totalatt']) == mininsertloss:
            return m
```

This module works in much the same way as the previous module, but instead of using AttenuationSearch, it uses FourPhase, and finds the value for the total attenuation from all four phases, then takes the minimum value out of the n values found.

4.7 minphaseatt.py

```python
"""A module to find the attenuation with minimum phase change."""
import decimal as dec
from heapq import nsmallest
import AttenuationSearch as ats


def minampvar(self):
    """Search for minimum variation in phase across attenuation frequency."""
    listatt = []
    totallist = []
    ampvar = []
    for row in self.dsa.iter_rows(row_offset=2):
        if row[self.dsas2128].value is not None:
            listatt.append(row[self.dsas2128].value)
    closest_values = nsmallest(int(self.k), listatt, key=lambda x: abs(
        dec.Decimal(x) - dec.Decimal(self.targetatt)))
    closest = [dec.Decimal(i) for i in closest_values]
    print(closest)
    for i in closest:
        resultsdict = ats.attenuationsearch(self, listatt, i)
        # Finds total variation in phase
        variation = resultsdict['phase2132'] - resultsdict['phase2124']
        resultsdict['variation'] = variation
        totallist.append(resultsdict)
        ampvar.append(variation)

    print(totallist)
    print(ampvar)
    minampdiff = min(abs(i) for i in ampvar)
    for m in totallist:
        if abs(m['variation']) == minampdiff:
            return m  # Returns the dictionary with the minimum phase variation in it
```

Another module that works in much the same way as the previous two. In this case it takes the differences in phase across frequency from the DSA sheet and finds the smallest difference in phase between 24 and 32 GHz.

4.8 minphasevariation

```python
"""Module to find setting with minimum variation across phases."""
import decimal as dec
import os
from heapq import nsmallest
import FourPhase as fp


def minvariation(self, set180, set90, set45, set225):
    """Find the closest value with the least variation across frequency."""
    dec.getcontext().prec = 6
    combined = set(map(dec.Decimal, open(
        (os.path.join(os.path.dirname(__file__), '1809045225.txt'))))) # Gets the path to the directory that the program is running in
    closest_values = nsmallest(int(self.k), combined, key=lambda x: abs(
        x - dec.Decimal(self.targetphase)))
    closest = [dec.Decimal(i) for i in closest_values]
    print(closest)
    total_values = []
    phasedifflist = []
    del combined
    for item in closest:
        resultsdict = fp.check(self, set180, set90, set45, set225, item)
        totalphasediff = resultsdict['phasediff1'] + resultsdict[
            'phasediff2'] + resultsdict['phasediff3'] + resultsdict['phasediff4']
        resultsdict['totalphasediff'] = totalphasediff
        phasedifflist.append(totalphasediff)
        total_values.append(resultsdict)
    print(total_values)
    minphasediff = min(phasedifflist)
    for m in total_values:
        print(m['totalphasediff'])
        if m['totalphasediff'] == minphasediff:
            return m
```

This module is the last of the four very similar modules, except in this case it finds the minimum phase variation across frequency by using FourPhase.py and getting the phases for 24 and 32 GHz before returning the option with the minimum phase variation.

4.9 Testing2.py

```python
"""Module to work on more ideas for MPAC tuning."""
import decimal as dec
import openpyxl as xl
import extras as ex
import TwoPhase as tp
import ThreePhase as thp
import FourPhase as fp
import AttenuationSearch as ats
import lookuptablegenerator as lutg
import os
import minphasevariation as mpv
import mininsertloss as mil
import minphaseatt as mpa
import minattvar as mav


class Main:
    """The main controlling class."""

    def __init__(self):
        """Initialise class."""
        print("Initialising")
        self.phase24 = 1  # Sets up all the important values in trhe program, mostly referring to positions in the spreadsheet
        self.phase28 = 2
        self.phase32 = 3
        self.att24 = 4
        self.att28 = 5
        self.att32 = 6
        self.dsastate24 = 0
        self.dsas2124 = 2
        self.dsaphase2124 = 4
        self.dsastate28 = 6
        self.dsas2128 = 8
        self.dsaphase2128 = 10
        self.dsastate32 = 12
        self.dsas2132 = 14
        self.dsaphase2132 = 16
        self.targetphase = 0
        self.targetatt = 0
        self.workbook = xl.load_workbook(os.path.join(
            os.path.dirname(__file__), 'source.xlsx'))
        self.s180 = self.workbook.get_sheet_by_name('180')
        self.s90 = self.workbook.get_sheet_by_name('90')
```

```
        self.s45 = self.workbook.get_sheet_by_name('45')
        self.s225 = self.workbook.get_sheet_by_name('22.5')
        self.dsa = self.workbook.get_sheet_by_name('DSA')
        self.k = 0

    def main(self):
        """Control all the other modules in the program."""
        dec.getcontext().prec = 6
        consent = input(
            "Would you like to calculate a specific (V)alue or generate
             ↪  a lookup (T)able?")
        if consent == 'V':
            # Selects each module individually, there may be more added
             ↪  in in
            # the future
            print("Choose what you would like to find")
            print("E - the most efficient value")
            print("D - the value with least variation across
             ↪  frequency")
            print("I - the value with minimum insertion loss")
            print("A - the attenuation value with minimum variation
             ↪  across frequency")
            print(
                "P - the attenuation value with minimum phase
                 ↪  difference across frequency")
            minvarchoice = input()

            list180 = []
            for row in self.s180.iter_rows(row_offset=2):
                if row[self.phase28].value is not None:

                 ↪  list180.append(dec.Decimal(row[self.phase28].value))
            set180 = set(list180)

            list90 = []
            for row in self.s90.iter_rows(row_offset=2):
                if row[self.phase28].value is not None:
                    list90.append(dec.Decimal(row[self.phase28].value))
            set90 = set(list90)

            list45 = []
            for row in self.s45.iter_rows(row_offset=2):
                if row[self.phase28].value is not None:
                    list45.append(dec.Decimal(row[self.phase28].value))
            set45 = set(list45)
```

```

        list225 = []
        for row in self.s225.iter_rows(row_offset=2):
            if row[self.phase28].value is not None:
                list225.append(dec.Decimal(row[self.phase28].value))
        set225 = set(list225)
        if minvarchoice == "D":
            self.targetphase = input(
                "Please enter the desired phase shift for 28GHz")
            self.k = input("How many values would you like to search?")
            minphase = mpv.minvariation(self, set180, set90, set45, set225)
            print(minphase)
        elif minvarchoice == "E":
            self.targetphase = input(
                "Please enter the desired phase shift for 28GHz")
            self.targetatt = input(
                "Please enter the desired attenuation for 28GHz")
            bestresult = ex.checkall(self, set180, set90, set45, set225)
            print(bestresult)
            bestresult2 = tp.checkall(self, set180, set90, set45, set225)
            print(bestresult2)
            bestresult3 = thp.checkall(self, set180, set90, set45, set225)
            print(bestresult3)
            closest = fp.closest_finder(self)
            bestresult4 = fp.check(
                self, set180, set90, set45, set225, closest)
            print(bestresult4)
            sollist = [bestresult['total'], bestresult2['total'],
                       bestresult3['total'], bestresult4['total']]
            bestphase = ex.mostaccurate(
                self, bestresult, bestresult2, bestresult3, bestresult4, sollist)
            attlist = ats.attlist(self)
            closest = ats.closest(self, attlist)
            bestatt = ats.attenuationsearch(self, attlist, closest)
            print(bestatt)
        elif minvarchoice == "I":
            self.k = input("How many values would you like to search?")
```

```
                mininsert = mil.mininsertloss(
                    self, set180, set90, set45, set225)
                print(mininsert)
            elif minvarchoice == "P":
                self.targetatt = input(
                    "Please enter the desired attenuation for 28GHz")
                self.k = input("How many values would you like to search?")
                minamp = mpa.minampvar(self)
                print(minamp)
            elif minvarchoice = "A":
                self.targetatt = input(
                    "Please enter the desired attenuation for 28GHz")
                self.k = input("How many values would you like to search?")
                minatt = mav.minattvar(self)
                print(minatt)
            else:
                print("Not a valid option")
        elif consent == 'T':
            option = input("Generate a table for (A)ttenuation or (P)hase?")
            if option == "A":
                lutg.atttablegen(self)
            elif option == "P":
                lutg.tablegen(self)
            else:
                print("That is not an option")

TESTBENCH = Main()
TESTBENCH.main()
```

This is the main module of the program. The first part, __init__ is where all the global variables are created, most of which are values referring to various columns in the spreadsheets of data. This is to make remembering values easier. This is also where the sheets themselves are loaded into memory, using openpyxl. The snippet

```
os.path.join(os.path.dirname(__file__), source.xlsx)
```

is a way of getting the absolute path of the file without hard coding it, due to the program not running if the file paths are relative.