

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

```
1  """A module to find the required attenuation."""
2  import decimal as dec
3
4
5  def attlist(self):
6      """Generate a list of all the values in the attenuation sheet."""
7      listatt = []
8      for row in self.dsa.iter_rows(row_offset=2):
9          if row[self.dsas2128].value is not None:
10             # Adds value to list if it actually has a value
11             listatt.append(row[self.dsas2128].value)
12      return listatt
13
14
15  def closest(self, attlist):
16      """Find closest attenuation to target."""
17      closest = min(attlist, key=lambda x: abs(
18          dec.Decimal(x) - dec.Decimal(self.targetatt)))
19      closest = dec.Decimal(closest)
20      return closest
21
22
23  def attenuationsearch(self, attlist, closest):
24      """Search for the most accurate attenuation."""
25      for row in self.dsa.iter_rows():
26          if row[self.dsas2128].value == closest:
27              att2128 = closest.quantize(dec.Decimal(
28                  '.001'), rounding=dec.ROUND_HALF_UP)
29              row28 = row[self.dsastate28].value
30              att2124 = dec.Decimal(row[self.dsas2124].value).quantize(
31                  dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
32              att2132 = dec.Decimal(row[self.dsas2132].value).quantize(
33                  dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
34              phase2124 =
35                  ↪ dec.Decimal(row[self.dsaphase2124].value).quantize(
36                      dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
37              phase2128 =
38                  ↪ dec.Decimal(row[self.dsaphase2128].value).quantize(
39                      dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
40              phase2132 =
41                  ↪ dec.Decimal(row[self.dsaphase2132].value).quantize(
42                      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

```

1  """Literally some magic."""
2  import decimal as dec
3
4
5  def check180(self, set180):
6      """180 degrees of magic."""
7      dec.getcontext().prec = 6
8      if self.targetphase in set180:
9          for row in self.s180.iter_rows():
10             if row[self.phase28].value == self.targetphase:
11                 row1 = int(row[0].value)
12                 att1 = dec.Decimal(row[self.att28].value)
13                 phaselow =
14                 ↪ dec.Decimal(row[self.phase24].value).quantize(
15                     dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
16                 phasehigh =
17                 ↪ dec.Decimal(row[self.phase32].value).quantize(
18                     dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
19                 phasediff = phasehigh - phaselow
20
21             return {'phase1': self.targetphase, 'row1': row1, 'att1': att1,
22                 ↪      'phasediff': phasediff, 'source1': 's180', 'totalphase':
23                 ↪      att1, 'total': self.targetphase}
24     else:
25         closest = min(set180, key=lambda x: abs(

```

```

22         dec.Decimal(x) - dec.Decimal(self.targetphase)))
23     for row in self.s180.iter_rows():
24         if row[self.phase28].value == closest:
25             closestround = closest.quantize(
26                 dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
27             row1 = int(row[0].value)
28             att1 = dec.Decimal(row[self.att28].value).quantize(
29                 dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
30             phaselow =
31                 ↪ dec.Decimal(row[self.phase24].value).quantize(
32                     dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
33             phasehigh =
34                 ↪ dec.Decimal(row[self.phase32].value).quantize(
35                     dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
36             phasediff = phasehigh - phaselow
37
38     return {'phase1': closestround, 'row1': row1, 'att1': att1,
39             ↪ 'phasediff': phasediff, 'source1': 's180', 'totalphase':
40             ↪ att1, 'total': closestround}
41
42 def check90(self, set90):
43     """90 degrees of magic."""
44     dec.getcontext().prec = 6
45     if self.targetphase in set90:
46         for row in self.s90.iter_rows():
47             if row[self.phase28].value == self.targetphase:
48                 row1 = int(row[0].value)
49                 att1 = dec.Decimal(row[self.att28].value).quantize(
50                     dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
51                 phaselow =
52                     ↪ dec.Decimal(row[self.phase24].value).quantize(
53                         dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
54                 phasehigh =
55                     ↪ dec.Decimal(row[self.phase32].value).quantize(
56                         dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
57                 phasediff = phasehigh - phaselow
58
59         return {'phase1': self.targetphase, 'row1': row1, 'att1': att1,
60                 ↪ 'phasediff': phasediff, 'source1': 's90', 'totalphase':
61                 ↪ att1, 'total': self.targetphase}
62
63     else:
64         closest = min(set90, key=lambda x: abs(
65             dec.Decimal(x) - dec.Decimal(self.targetphase)))

```

```

59     for row in self.s90.iter_rows():
60         if row[self.phase28].value == closest:
61             closestround = closest.quantize(
62                 dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
63             row1 = int(row[0].value)
64             att1 = dec.Decimal(row[self.att28].value).quantize(
65                 dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
66             phaselow =
67                 ↳ dec.Decimal(row[self.phase24].value).quantize(
68                     dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
69             phasehigh =
70                 ↳ dec.Decimal(row[self.phase32].value).quantize(
71                     dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
72             phasediff = phasehigh - phaselow
73
74     return {'phase1': closestround, 'row1': row1, 'att1': att1,
75             ↳ 'phasediff': phasediff, 'source1': 's90', 'totalphase':
76             ↳ att1, 'total': closestround}
77
78 def check45(self, set45):
79     """45 Degrees of magic.
80
81     Parameters
82     -----
83     self, set45, set45
84     Returns
85     -----
86     phase1, row1, att1, phasediff
87     """
88     dec.getcontext().prec = 6
89     if self.targetphase in set45:
90         for row in self.s45.iter_rows():
91             if row[self.phase28].value == self.targetphase:
92                 row1 = int(row[0].value)
93                 att1 = dec.Decimal(row[self.att28].value).quantize(
94                     dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
95                 phaselow =
96                     ↳ dec.Decimal(row[self.phase24].value).quantize(
97                         dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
98                 phasehigh =
99                     ↳ dec.Decimal(row[self.phase32].value).quantize(
100                         dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
101                 phasediff = phasehigh - phaselow

```

```

98         return {'phase1': self.targetphase, 'row1': row1, 'att1': att1,
99                ↪ 'phasediff': phasediff, 'source1': 's45', 'totalphase':
100                ↪ att1, 'total': self.targetphase}
101
102     else:
103         closest = min(set45, key=lambda x: abs(
104             dec.Decimal(x) - dec.Decimal(self.targetphase)))
105         for row in self.s45.iter_rows():
106             if row[self.phase28].value == closest:
107                 closestround = closest.quantize(
108                     dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
109                 row1 = int(row[0].value)
110                 att1 = dec.Decimal(row[self.att28].value).quantize(
111                     dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
112                 phaselow =
113                 ↪ dec.Decimal(row[self.phase24].value).quantize(
114                     dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
115                 phasehigh =
116                 ↪ dec.Decimal(row[self.phase32].value).quantize(
117                     dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
118                 phasediff = phasehigh - phaselow
119
120         return {'phase1': closestround, 'row1': row1, 'att1': att1,
121                ↪ 'phasediff': phasediff, 'source1': 's45', 'totalphase':
122                ↪ att1, 'total': closestround}
123
124     def check225(self, set225):
125         """22.5 degrees of magic.
126
127         Parameters
128         -----
129         self, s225, set225
130         Returns
131         -----
132         phase1, row1, att1, phasediff
133         """
134         dec.getcontext().prec = 6
135         if self.targetphase in set225:
136             for row in self.s225.iter_rows():
137                 if row[self.phase28].value == self.targetphase:
138                     row1 = int(row[0].value)
139                     att1 = dec.Decimal(row[self.att28].value).quantize(
140                         dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
141                     phaselow =
142                     ↪ dec.Decimal(row[self.phase24].value).quantize(

```

```

137         dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
138     phasehigh =
139         ↪ dec.Decimal(row[self.phase32].value).quantize(
140         dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
141     phasediff = phasehigh - phaselow
142
143     return {'phase1': self.targetphase, 'row1': row1, 'att1': att1,
144             ↪ 'phasediff': phasediff, 'source1': 's225', 'totalphase':
145             ↪ att1, 'total': self.targetphase}
146
147 else:
148     closest = min(set225, key=lambda x: abs(
149         dec.Decimal(x) - dec.Decimal(self.targetphase)))
150     for row in self.s225.iter_rows():
151         if row[self.phase28].value == closest:
152             closestround = closest.quantize(
153                 dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
154             row1 = int(row[0].value)
155             att1 = dec.Decimal(row[self.att28].value).quantize(
156                 dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
157             phaselow =
158                 ↪ dec.Decimal(row[self.phase24].value).quantize(
159                 dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
160             phasehigh =
161                 ↪ dec.Decimal(row[self.phase32].value).quantize(
162                 dec.Decimal('.001'), rounding=dec.ROUND_HALF_UP)
163             phasediff = phasehigh - phaselow
164
165     return {'phase1': closestround, 'row1': row1, 'att1': att1,
166             ↪ 'phasediff': phasediff, 'source1': 's225', 'totalphase':
167             ↪ att1, 'total': closestround}
168
169 def checkall(self, set180, set90, set45, set225):
170     """Check to find most accurate solution so far.
171
172     Parameters
173     -----
174     self
175     Returns
176     -----
177     bestsol data dict
178     """
179     dec.getcontext().prec = 6
180     sol180 = check180(self, set180)
181     sol90 = check90(self, set90)

```

```

176 sol45 = check45(self, set45)
177 sol225 = check225(self, set225)
178 bestsollist = [sol180['phase1'], sol90[
179     'phase1'], sol45['phase1'], sol225['phase1']]
180
181 if sol180['phase1'] == self.targetphase:
182     return sol180
183
184 elif sol90['phase1'] == self.targetphase:
185     return sol90
186
187 elif sol45['phase1'] == self.targetphase:
188     return sol45
189
190 elif sol225['phase1'] == self.targetphase:
191     return sol225
192
193 else:
194     closest = min(bestsollist, key=lambda x: abs(
195         dec.Decimal(x) - dec.Decimal(self.targetphase)))
196     if closest == sol180['phase1']:
197         return sol180
198     elif closest == sol90['phase1']:
199         return sol90
200     elif closest == sol45['phase1']:
201         return sol45
202     elif closest == sol225['phase1']:
203         return sol225
204     else:
205         return None
206
207
208 def mostaccurate(self, bestresult, bestresult2, bestresult3,
209     ↪ bestresult4, sollist):
210     """Find most accurate solution and return."""
211     bestsol = min(sollist, key=lambda x: abs(
212         dec.Decimal(x) - dec.Decimal(self.targetphase)))
213     if bestsol == bestresult['total']:
214         return bestresult
215     elif bestsol == bestresult2['total']:
216         return bestresult2
217     elif bestsol == bestresult3['total']:
218         return bestresult3
219     elif bestsol == bestresult4['total']:
220         return bestresult4
221     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

```
1  """Test all four phases in one."""
2  import decimal as dec
3  import os
4
5
6  def closest_finder(self):
7      """Find the closest value to a given value."""
8      combined = set(map(str.rstrip, open(
9          (os.path.join(os.path.dirname(__file__), '1809045225.txt')))))
10     closest = min(combined, key=lambda x: abs(
11         dec.Decimal(x) - dec.Decimal(self.targetphase)))
12     closest = dec.Decimal(closest)
13     print(closest)
14     return closest
15     del combined
16
17
18 def check(self, set180, set90, set45, set225, closest):
19     """Check to find the most accurate four phase solution."""
20     dec.getcontext().prec = 6
21     for i in set180:
22         for j in set90:
23             for k in set45:
24                 for l in set225:
25                     total = i + j + k + l
26                     if total == closest:
27                         for row in self.s180:
28                             if row[self.phase28].value == i:
```

```

29         row1 = int(row[0].value)
30         att1 =
31             ↳ dec.Decimal(row[self.att28].value).quantize(
32                 dec.Decimal('.001'),
33                 ↳ rounding=dec.ROUND_HALF_UP)
34         phaselow1 =
35             ↳ dec.Decimal(row[self.phase24].value).quantize(
36                 dec.Decimal('.001'),
37                 ↳ rounding=dec.ROUND_HALF_UP)
38         phasehigh1 =
39             ↳ dec.Decimal(row[self.phase32].value).quantize(
40                 dec.Decimal('.001'),
41                 ↳ rounding=dec.ROUND_HALF_UP)
42         phasediff1 = phasehigh1 - phaselow1
43
44     for row in self.s90:
45         if row[self.phase28].value == j:
46             row2 = int(row[0].value)
47             att2 =
48                 ↳ dec.Decimal(row[self.att28].value).quantize(
49                     dec.Decimal('.001'),
50                     ↳ rounding=dec.ROUND_HALF_UP)
51             phaselow2 =
52                 ↳ dec.Decimal(row[self.phase24].value).quantize(
53                     dec.Decimal('.001'),
54                     ↳ rounding=dec.ROUND_HALF_UP)
55             phasehigh2 =
56                 ↳ dec.Decimal(row[self.att32].value).quantize(
57                     dec.Decimal('.001'),
58                     ↳ rounding=dec.ROUND_HALF_UP)
59             phasediff2 = phasehigh2 - phaselow2
60
61     for row in self.s45:
62         if row[self.phase28].value == k:
63             row3 = int(row[0].value)
64             att3 =
65                 ↳ dec.Decimal(row[self.att28].value).quantize(
66                     dec.Decimal('.001'),
67                     ↳ rounding=dec.ROUND_HALF_UP)
68             phaselow3 =
69                 ↳ dec.Decimal(row[self.att24].value).quantize(
70                     dec.Decimal('.001'),
71                     ↳ rounding=dec.ROUND_HALF_UP)

```

```

56         phasehigh3 =
57             ↪ dec.Decimal(row[self.att32].value).quantize(
58                 dec.Decimal('.001'),
59                 ↪ rounding=dec.ROUND_HALF_UP)
60         phasediff3 = phasehigh3 - phaselow3
61
62     for row in self.s225:
63         if row[self.phase28].value == 1:
64             row4 = int(row[0].value)
65             att4 =
66                 ↪ dec.Decimal(row[self.att28].value).quantize(
67                     dec.Decimal('.001'),
68                     ↪ rounding=dec.ROUND_HALF_UP)
69             phaselow4 =
70                 ↪ dec.Decimal(row[self.phase24].value).quantize(
71                     dec.Decimal('.001'),
72                     ↪ rounding=dec.ROUND_HALF_UP)
73             phasehigh4 =
74                 ↪ dec.Decimal(row[self.phase32].value).quantize(
75                     dec.Decimal('.001'),
76                     ↪ rounding=dec.ROUND_HALF_UP)
77             phasediff4 = phasehigh4 - phaselow4
78
79     totalatt = att1 + att2 + att3 + att4
80     return {'phase1':
81         ↪ i.quantize(dec.Decimal('.001'),
82         ↪ rounding=dec.ROUND_HALF_UP), 'row1': row1,
83         ↪ 'att1': att1, 'phasediff1': phasediff1,
84         ↪ 'source1': 's180', 'phase2':
85         ↪ j.quantize(dec.Decimal('.001'),
86         ↪ rounding=dec.ROUND_HALF_UP), 'row2': row2,
87         ↪ 'att2': att2, 'phasediff2': phasediff2,
88         ↪ 'source2': 's90', 'phase3':
89         ↪ k.quantize(dec.Decimal('.001'),
90         ↪ rounding=dec.ROUND_HALF_UP), 'row3': row3,
91         ↪ 'att3': att3, 'phasediff3': phasediff3,
92         ↪ 'source3': 's45', 'phase4':
93         ↪ l.quantize(dec.Decimal('.001'),
94         ↪ rounding=dec.ROUND_HALF_UP), 'row4': row4,
95         ↪ 'att4': att4, 'phasediff4': phasediff4,
96         ↪ 'source4': 's225', 'total':
97         ↪ total.quantize(dec.Decimal('.001'),
98         ↪ rounding=dec.ROUND_HALF_UP), 'totalatt':
99         ↪ totalatt.quantize(dec.Decimal('.001'),
100         ↪ 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 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

```

1  """To generate a look-up table."""
2  import decimal as dec
3  import extras as ex
4  import TwoPhase as twp
5  import ThreePhase as thp
6  import FourPhase as fp
7  import tabulate
8  import os
9  import csv
10
11
12  def tablegen(self):
13      """Generate look up table for phase."""
14      list180 = []
15      for row in self.s180.iter_rows(row_offset=2):
16          if row[self.phase28].value is not None:
17              list180.append(dec.Decimal(row[self.phase28].value))
18      set180 = set(list180)
19
20      list90 = []
21      for row in self.s90.iter_rows(row_offset=2):
22          if row[self.phase28].value is not None:
23              list90.append(dec.Decimal(row[self.phase28].value))
24      set90 = set(list90)
25

```

```

26 list45 = []
27 for row in self.s45.iter_rows(row_offset=2):
28     if row[self.phase28].value is not None:
29         list45.append(dec.Decimal(row[self.phase28].value))
30 set45 = set(list45)
31
32 list225 = []
33 for row in self.s225.iter_rows(row_offset=2):
34     if row[self.phase28].value is not None:
35         list225.append(dec.Decimal(row[self.phase28].value))
36 set225 = set(list225)
37 headers = ["Target value", "180 Used", "90 Used", "45 Used",
38            "22.5 Used", "180 Setting", "90 Setting", "45 Setting",
39            ↪ "22.5 Setting"]
40 table = []
41 for i in range(1, 65):
42     self.targetphase = (-360 / 64) * i
43     bestresult = ex.checkall(self, set180, set90, set45, set225)
44     bestresult2 = twp.checkall(self, set180, set90, set45, set225)
45     bestresult3 = thp.checkall(self, set180, set90, set45, set225)
46     bestresult4 = fp.check(self, set180, set90, set45, set225)
47     sollist = [bestresult['total'], bestresult2['total'],
48               bestresult3['total'], bestresult4['total']]
49     bestphase = ex.mostaccurate(
50         self, bestresult, bestresult2, bestresult3, bestresult4,
51         ↪ sollist)
52     s180present = 0
53     s90present = 0
54     s45present = 0
55     s225present = 0
56     print(bestphase)
57     if 'source1' in bestphase:
58         if bestphase['source1'] == 's180':
59             s180present = 1
60         elif bestphase['source1'] == 's90':
61             s90present = 1
62         elif bestphase['source1'] == 's45':
63             s45present = 1
64         elif bestphase['source1'] == 's225':
65             s225present = 1
66
67     if 'source2' in bestphase:
68         if bestphase['source2'] == 's90':
69             s90present = 2
70         elif bestphase['source2'] == 's45':
71             s45present = 2

```

```

70         elif bestphase['source2'] == 's225':
71             s225present = 2
72
73     if 'source3' in bestphase:
74         if bestphase['source3'] == 's45':
75             s45present = 3
76         elif bestphase['source3'] == 's225':
77             s225present = 3
78
79     if 'source4' in bestphase:
80         if bestphase['source4'] == 's225':
81             s225present = 4
82     print(s180present, s90present, s45present, s225present)
83     if s180present == 1:
84         s180setting = '{0:02b}'.format(bestphase['row1'])
85     else:
86         s180setting = '{0:02b}'.format(2)
87
88     if s90present == 1:
89         s90setting = '{0:06b}'.format(bestphase['row1'])
90     elif s90present == 2:
91         s90setting = '{0:06b}'.format(bestphase['row2'])
92         s90present = 1
93     else:
94         s90setting = '{0:06b}'.format(32)
95
96     if s45present == 1:
97         s45setting = '{0:09b}'.format(bestphase['row1'])
98     elif s45present == 2:
99         s45setting = '{0:09b}'.format(bestphase['row2'])
100         s45present = 1
101     elif s45present == 3:
102         s45setting = '{0:09b}'.format(bestphase['row3'])
103         s45present = 1
104     else:
105         s45setting = '{0:09b}'.format(256)
106
107     if s225present == 1:
108         s225setting = '{0:09b}'.format(bestphase['row1'])
109     elif s225present == 2:
110         s225setting = '{0:09b}'.format(bestphase['row2'])
111         s225present = 1
112     elif s225present == 3:
113         s225setting = '{0:09b}'.format(bestphase['row3'])
114         s225present = 1
115     elif s225present == 4:

```

```

116         s225setting = '{0:09b}'.format(bestphase['row4'])
117         s225present = 1
118     else:
119         s225setting = '{0:09b}'.format(256)
120
121     endlist = [self.targetphase, s180present, s90present,
122               ↪ s45present,
123               ↪ s225present, s180setting, s90setting, s45setting,
124               ↪ s225setting]
125     table.append(endlist)
126
127     print(tabulate.tabulate(table, headers, tablefmt="fancy_grid"))
128     with open(os.path.join(
129         os.path.dirname(__file__), 'phasetable.csv'), 'w') as
130         ↪ csvfile:
131         writer = csv.writer(csvfile)
132         [writer.writerow(r) for r in table]
133     return
134
135 def atttablegen(self):
136     """Generate a table for attenuation."""
137     listatt = []
138     for row in self.dsa.iter_rows(row_offset=2):
139         if row[self.dsas2128].value is not None:
140             listatt.append(dec.Decimal(row[self.dsas2128].value))
141     setatt = set(listatt)
142     headers = ["Target", "Attenuation", "Setting"]
143     table = []
144     for i in range(1, 65):
145         self.targetatt = dec.Decimal((-24 / 64) * i)
146         if self.targetatt in setatt:
147             for row in self.dsa.iter_rows():
148                 if row[self.dsas2128].value == self.targetatt:
149                     row28 = row[self.dsastate28].value
150                     att2128 = self.targetatt.quantize(
151                         dec.Decimal('.001'),
152                         ↪ rounding=dec.ROUND_HALF_UP)
153                     att2124 =
154                         ↪ dec.Decimal(row[self.dsas2124].value).quantize(
155                             dec.Decimal('.001'),
156                             ↪ rounding=dec.ROUND_HALF_UP)
157                     att2132 =
158                         ↪ 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         DSAssetting = '{0:012b}'.format(row28)
170         table.append([self.targetatt, att2128, DSAssetting])
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

```
1  """Module to find minimum attenuation variation across frequency."""
2  import decimal as dec
3  from heapq import nsmallest
4  import AttenuationSearch as ats
5
6
7  def minattvar(self):
8      """Search for minimum amplitude variation across frequency."""
9      listatt = []
10     totallist = []
11     attvar = []
12     # Iterates through all the rows in the spreadsheet from the third
13     ↪ row
14     # onwards
15     for row in self.dsa.iter_rows(row_offset=2):
16         if row[self.dsas2128].value is not None:
17             # Adds the value to a list if it actually has a value, and
18             ↪ isn't
19             # blank
20             listatt.append(row[self.dsas2128].value)
21     closest_values = nsmallest(int(self.k), listatt, key=lambda x: abs(
22     dec.Decimal(x) - dec.Decimal(self.targetatt))) # Finds the k
23     ↪ closest numbers to a given value from the list
24     # Converts the values in closest_values to a decimal from a float.
25     ↪ This
26     # makes it easier to work with as it adds in rounding
27     closest = [dec.Decimal(i) for i in closest_values]
28     print(closest)
29     for i in closest:
30         # Calls the AttenuationSearch module to search for the various
31         ↪ values.
32         # It does this for each item in the closest list
33         resultsdict = ats.attenuationsearch(self, listatt, i)
34         variation = resultsdict['att2132'] - resultsdict['att2124']
35         # Adds a new value to the dictionary, with the key variation
36         resultsdict['variation'] = variation
37         totallist.append(resultsdict)
38         attvar.append(variation)
39     print(totallist)
40     print(attvar)
41     # Finds the smallest number in the list
42     minattdiff = min(abs(i) for i in attvar)
43     for m in totallist:
44         if abs(m['variation']) == minattdiff:
```

```

40         # If the value in the dictionary/key combo m['variation']
        ↪ matches
41         # the minimum value, it gets returned and printed
42         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

```

1  """Find the minimum insertion loss, i.e., the minimum attenuation for a
    ↪ given phase."""
2  import decimal as dec
3  import os
4  from heapq import nsmallest
5  import FourPhase as fp
6
7
8  def mininsertloss(self, set180, set90, set45, set225):
9      """Find minimum insertion loss for a phase."""
10     dec.getcontext().prec = 6
11     combined = set(map(dec.Decimal, open((os.path.join(
12         os.path.dirname(__file__), '1809045225.txt')))))
13     closest_values = nsmallest(int(self.k), combined, key=lambda x:
    ↪ abs(
14         x - dec.Decimal(self.targetphase)))
15     closest = [dec.Decimal(i) for i in closest_values]
16     total_values = []
17     insertlosslist = []
18     del combined
19     for item in closest:
20         resultsdict = fp.check(self, set180, set90, set45, set225,
    ↪ item)
21         insertlosslist.append(resultsdict['totalatt'])
22         total_values.append(resultsdict)
23     print(total_values)
24     mininsertloss = min(abs(i) for i in insertlosslist)
25     for m in total_values:
26         print(m['totalatt'])
27         if abs(m['totalatt']) == mininsertloss:
28             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

```
1  """A module to find the attenuation with minimum phase change."""
2  import decimal as dec
3  from heapq import nsmallest
4  import AttenuationSearch as ats
5
6
7  def minampvar(self):
8      """Search for minimum variation in phase across attenuation
9      ↪ frequency."""
10     listatt = []
11     totallist = []
12     ampvar = []
13     for row in self.dsa.iter_rows(row_offset=2):
14         if row[self.dsas2128].value is not None:
15             listatt.append(row[self.dsas2128].value)
16     closest_values = nsmallest(int(self.k), listatt, key=lambda x: abs(
17         dec.Decimal(x) - dec.Decimal(self.targetatt)))
18     closest = [dec.Decimal(i) for i in closest_values]
19     print(closest)
20     for i in closest:
21         resultsdict = ats.attenuationsearch(self, listatt, i)
22         # Finds total variation in phase
23         variation = resultsdict['phase2132'] - resultsdict['phase2124']
24         resultsdict['variation'] = variation
25         totallist.append(resultsdict)
26         ampvar.append(variation)
27
28     print(totallist)
29     print(ampvar)
30     minampdiff = min(abs(i) for i in ampvar)
31     for m in totallist:
32         if abs(m['variation']) == minampdiff:
33             return m # Returns the dictionary with the minimum phase
34             ↪ 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

```
1  """Module to find setting with minimum variation across phases."""
2  import decimal as dec
3  import os
4  from heapq import nsmallest
5  import FourPhase as fp
6
7
8  def minvariation(self, set180, set90, set45, set225):
9      """Find the closest value with the least variation across
10         ↪ frequency."""
11      dec.getcontext().prec = 6
12      combined = set(map(dec.Decimal, open(
13          (os.path.join(os.path.dirname(__file__), '1809045225.txt')))))
14          ↪ # Gets the path to the directory that the program is
15          ↪ running in
16      closest_values = nsmallest(int(self.k), combined, key=lambda x:
17          ↪ abs(
18              x - dec.Decimal(self.targetphase)))
19      closest = [dec.Decimal(i) for i in closest_values]
20      print(closest)
21      total_values = []
22      phasedifflist = []
23      del combined
24      for item in closest:
25          resultsdict = fp.check(self, set180, set90, set45, set225,
26              ↪ item)
27          totalphasediff = resultsdict['phasediff1'] + resultsdict[
28              'phasediff2'] + resultsdict['phasediff3'] +
29              ↪ resultsdict['phasediff4']
30          resultsdict['totalphasediff'] = totalphasediff
31          phasedifflist.append(totalphasediff)
32          total_values.append(resultsdict)
33      print(total_values)
34      minphasediff = min(phasedifflist)
35      for m in total_values:
36          print(m['totalphasediff'])
37          if m['totalphasediff'] == minphasediff:
38              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

```
1  """Module to work on more ideas for MPAC tuning."""
2  import decimal as dec
3  import openpyxl as xl
4  import extras as ex
5  import TwoPhase as tp
6  import ThreePhase as thp
7  import FourPhase as fp
8  import AttenuationSearch as ats
9  import lookuptablegenerator as lutg
10 import os
11 import minphasevariation as mpv
12 import mininsertloss as mil
13 import minphaseatt as mpa
14 import minattvar as mav
15
16
17 class Main:
18     """The main controlling class."""
19
20     def __init__(self):
21         """Initialise class."""
22         print("Initialising")
23         self.phase24 = 1 # Sets up all the important values in trhe
24         ↪ program, mostly referring to positions in the spreadsheet
25         self.phase28 = 2
26         self.phase32 = 3
27         self.att24 = 4
28         self.att28 = 5
29         self.att32 = 6
30         self.dsastate24 = 0
31         self.dsas2124 = 2
32         self.dsaphase2124 = 4
33         self.dsastate28 = 6
34         self.dsas2128 = 8
35         self.dsaphase2128 = 10
36         self.dsastate32 = 12
37         self.dsas2132 = 14
38         self.dsaphase2132 = 16
39         self.targetphase = 0
40         self.targetatt = 0
41         self.workbook = xl.load_workbook(os.path.join(
42             os.path.dirname(__file__), 'source.xlsx'))
43         self.s180 = self.workbook.get_sheet_by_name('180')
44         self.s90 = self.workbook.get_sheet_by_name('90')
```

```

44     self.s45 = self.workbook.get_sheet_by_name('45')
45     self.s225 = self.workbook.get_sheet_by_name('22.5')
46     self.dsa = self.workbook.get_sheet_by_name('DSA')
47     self.k = 0
48
49     def main(self):
50         """Control all the other modules in the program."""
51         dec.getcontext().prec = 6
52         consent = input(
53             "Would you like to calculate a specific (V)alue or generate
54             ↪ a lookup (T)able?")
55         if consent == 'V':
56             # Selects each module individually, there may be more added
57             ↪ in in
58             # the future
59             print("Choose what you would like to find")
60             print("E - the most efficient value")
61             print("D - the value with least variation across
62             ↪ frequency")
63             print("I - the value with minimum insertion loss")
64             print("A - the attenuation value with minimum variation
65             ↪ across frequency")
66             print(
67                 "P - the attenuation value with minimum phase
68                 ↪ difference across frequency")
69             minvarchoice = input()
70
71             list180 = []
72             for row in self.s180.iter_rows(row_offset=2):
73                 if row[self.phase28].value is not None:
74
75                     ↪ list180.append(dec.Decimal(row[self.phase28].value))
76             set180 = set(list180)
77
78             list90 = []
79             for row in self.s90.iter_rows(row_offset=2):
80                 if row[self.phase28].value is not None:
81                     list90.append(dec.Decimal(row[self.phase28].value))
82             set90 = set(list90)
83
84             list45 = []
85             for row in self.s45.iter_rows(row_offset=2):
86                 if row[self.phase28].value is not None:
87                     list45.append(dec.Decimal(row[self.phase28].value))
88             set45 = set(list45)

```

```

83
84 list225 = []
85 for row in self.s225.iter_rows(row_offset=2):
86     if row[self.phase28].value is not None:
87         ↪ list225.append(dec.Decimal(row[self.phase28].value))
88 set225 = set(list225)
89 if minvarchoice == "D":
90     self.targetphase = input(
91         "Please enter the desired phase shift for 28GHz")
92     self.k = input("How many values would you like to
93         ↪ search?")
94     minphase = mpv.minvariation(self, set180, set90, set45,
95         ↪ set225)
96     print(minphase)
97 elif minvarchoice == "E":
98     self.targetphase = input(
99         "Please enter the desired phase shift for 28GHz")
100     self.targetatt = input(
101         "Please enter the desired attenuation for 28GHz")
102     bestresult = ex.checkall(self, set180, set90, set45,
103         ↪ set225)
104     print(bestresult)
105     bestresult2 = tp.checkall(self, set180, set90, set45,
106         ↪ set225)
107     print(bestresult2)
108     bestresult3 = thp.checkall(self, set180, set90, set45,
109         ↪ set225)
110     print(bestresult3)
111     closest = fp.closest_finder(self)
112     bestresult4 = fp.check(
113         self, set180, set90, set45, set225, closest)
114     print(bestresult4)
115     sollist = [bestresult['total'], bestresult2['total'],
116         bestresult3['total'], bestresult4['total']]
117     bestphase = ex.mostaccurate(
118         self, bestresult, bestresult2, bestresult3,
119         ↪ bestresult4, sollist)
120     attlist = ats.attlist(self)
121     closest = ats.closest(self, attlist)
122     bestatt = ats.attenuationsearch(self, attlist, closest)
123     print(bestatt)
124 elif minvarchoice == "I":
125     self.k = input("How many values would you like to
126         ↪ search?")

```

```

120         mininsert = mil.mininsertloss(
121             self, set180, set90, set45, set225)
122         print(mininsert)
123     elif minvarchoice == "P":
124         self.targetatt = input(
125             "Please enter the desired attenuation for 28GHz")
126         self.k = input("How many values would you like to
            ↳ search?")
127         minamp = mpa.minampvar(self)
128         print(minamp)
129     elif minvarchoice == "A":
130         self.targetatt = input(
131             "Please enter the desired attenuation for 28GHz")
132         self.k = input("How many values would you like to
            ↳ search?")
133         minatt = mav.minattvar(self)
134         print(minatt)
135     else:
136         print("Not a valid option")
137     elif consent == 'T':
138         option = input("Generate a table for (A)ttenuation or
            ↳ (P)hase?")
139         if option == "A":
140             lutg.atttablegen(self)
141         elif option == "P":
142             lutg.tablegen(self)
143         else:
144             print("That is not an option")
145
146 TESTBENCH = Main()
147 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.