AFCCP NRL Module Walkthrough

December 2, 2022

1 Air Force Cadet Career Problem (AFCCP) Non-Rated Line (NRL) Model Walkthrough

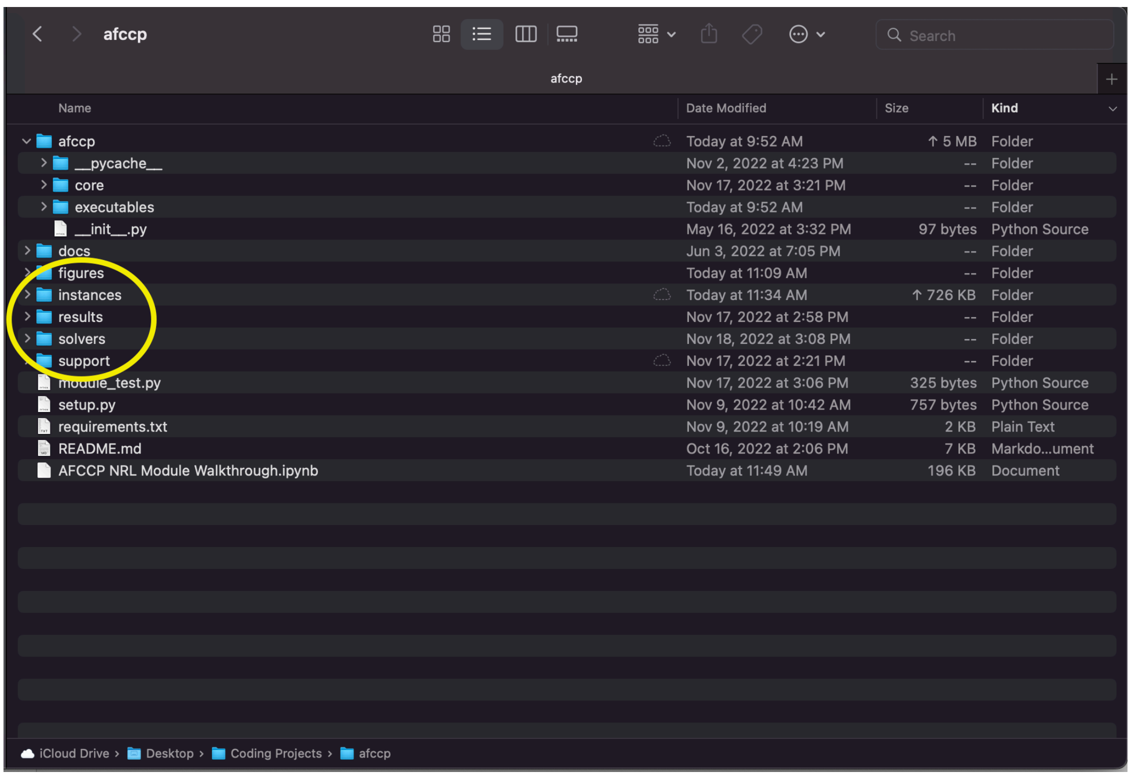
1.1 Part One (Data Wrangling)

1.2 Part Two (Code Walkthrough)

Whatever script or .ipynb notebook you're running code from should be in the same root "afccp" directory. For example, the picture below is the directory that this notebook "AFCCP NRL Module Walkthrough.ipynb" is currectly in. The folders that I've circled in yellow may or may not be there for you, but they get created as soon as you import "afccp" into your code.

[1]:
```
from IPython.display import Image
Image(filename='pic1.png')
```

[1]:



Let's go ahead and import the module. I'm going to assume that you have afccp either installed or it's cloned into wherever you're looking at. You should also have all of the following packages installed as well.

[2]:
```
import sys
!{sys.executable} -m pip list
```

```
Package              Version
-------------------- -----------
appnope              0.1.3
argon2-cffi          21.3.0
argon2-cffi-bindings 21.2.0
asttokens            2.0.5
attrs                21.4.0
backcall             0.2.0
beautifulsoup4       4.11.1
bleach               5.0.0
Brotli               1.0.9
certifi              2021.10.8
cffi                 1.15.0
charset-normalizer   2.0.12
click                8.1.3
copulas              0.6.1
cryptography         38.0.1
ctgan                0.5.1
cycler               0.11.0
dash                 2.6.1
dash-core-components 2.0.0
dash-html-components 2.0.0
dash-table           5.0.0
debugpy              1.6.0
decorator            5.1.1
deepecho             0.3.0.post1
defusedxml           0.7.1
entrypoints          0.4
et-xmlfile           1.1.0
executing            0.8.3
Faker                9.9.1
fastjsonschema       2.15.3
Flask                2.2.2
Flask-Compress       1.12
fonttools            4.31.2
graphviz             0.19.1
idna                 3.3
importlib-metadata   4.12.0
importlib-resources  5.7.1
ipykernel            6.12.1
ipython              8.2.0
```

ipython-genutils	0.2.0
ipywidgets	7.7.0
itsdangerous	2.1.2
jedi	0.18.1
Jinja2	3.1.2
joblib	1.1.0
jsonschema	4.5.1
jupyter	1.0.0
jupyter-client	7.2.2
jupyter-console	6.4.3
jupyter-core	4.9.2
jupyterlab-pygments	0.2.2
jupyterlab-widgets	1.1.0
kiwisolver	1.4.2
llvmlite	0.38.0
lxml	4.9.0
MarkupSafe	2.1.1
matplotlib	3.5.1
matplotlib-inline	0.1.3
mistune	0.8.4
nbclient	0.6.3
nbconvert	6.5.0
nbformat	5.4.0
nest-asyncio	1.5.5
notebook	6.4.11
numba	0.55.1
numpy	1.21.5
openpyxl	3.0.9
packaging	21.3
pandas	1.4.2
pandoc	2.2
pandocfilters	1.5.0
parso	0.8.3
pdfminer	20191125
pdfminer.six	20220524
pexpect	4.8.0
pickleshare	0.7.5
Pillow	9.1.0
pip	21.2.4
plotly	5.10.0
plumbum	1.8.0
ply	3.11
prometheus-client	0.14.1
prompt-toolkit	3.0.29
psutil	5.9.0
ptyprocess	0.7.0
pure-eval	0.2.2
pycparser	2.21

```
pycryptodome        3.15.0
Pygments            2.11.2
Pyomo               6.4.0
pyparsing           3.0.7
pyrsistent          0.18.1
python-dateutil     2.8.2
python-pptx         0.6.21
pyts                0.12.0
pytz                2022.1
PyYAML              5.4.1
pyzmq               22.3.0
qtconsole           5.3.0
QtPy                2.1.0
rdt                 0.6.4
requests            2.27.1
scikit-learn        1.0.2
scipy               1.7.3
sdmetrics           0.4.1
sdv                 0.14.0
Send2Trash          1.8.0
setuptools          58.0.4
six                 1.16.0
sklearn             0.0
soupsieve           2.3.2.post1
stack-data          0.2.0
tenacity            8.0.1
terminado           0.15.0
text-unidecode      1.3
threadpoolctl       3.1.0
tinycss2            1.1.1
torch               1.11.0
torchvision         0.12.0
tornado             6.1
tqdm                4.64.0
traitlets           5.1.1
typing_extensions   4.1.1
urllib3             1.26.9
wcwidth             0.2.5
webencodings        0.5.1
Werkzeug            2.2.2
wheel               0.37.1
widgetsnbextension  3.6.0
xlrd                2.0.1
XlsxWriter          3.0.3
zipp                3.8.0
```

If you don't have all of the above packages, simply run the following code to install them from the requirements.txt file: (I commented it out for sake of the pdf length!)

```
[3]: # # Install a pip package in the current Jupyter kernel
     # !{sys.executable} -m pip install -r requirements.txt
```

1.2.1 CadetCareerProblem Overview

Now that we have the required packages, let's import the "CadetCareerProblem" class from the "afccp" module. This is the main class object that we'll be dealing with. It represents the class of all cadet-AFSC matching problems (various cadet class years). Please note the two different meanings of the word "class" in the previous sentence! Each "instance" of CadetCareerProblem is a distinct academic class year (2019, 2020, 2021, etc.) with various cadet/AFSC parameters. Let's load in this class from the afccp module. If this is the first time you run this line, several folders will be created in your working directory (instances, figures, results, etc.)

```
[4]: from afccp.core.problem_class import CadetCareerProblem
```

```
Importing 'afccp' module…
Data folders found.
Pyomo module found.
SDV module found.
Sklearn Manifold module found.
```

The "instances" folder is where you will keep all of your problem instance files labeled according to their "data_name". The data names are the names of the class years or generated data. Things like 2019, 2020, 2021, or A, B, C, or Random_1, Random_2, Realistic_1, etc. for generated data. All that is needed in order to create a problem instance is the two sheets: Cadets_Fixed and AFSCs_Fixed

Here we will load in the two excel sheets (cadets/afscs fixed) for the class of 2023.

```
[5]: import pandas as pd
     import numpy as np
     import os

     # Obtain working directory
     dir_path = os.getcwd() + '/'
     print('Working directory:', dir_path)

     filepath = dir_path + "instances/2023b.xlsx"
     cadets_fixed = pd.read_excel(filepath, sheet_name="Cadets Fixed")
     cadets_fixed
```

```
Working directory: /Users/griffenlaird/Desktop/Coding Projects/afccp/
```

```
[5]:       Cadet  Assigned  Male  Minority       Race         Ethnicity  USAFA  \
     0         0       NaN     1         0  CAUCASIAN           UNKNOWN      0
     1         1       NaN     1         0  CAUCASIAN           UNKNOWN      0
     2         2       NaN     1         1      OTHER  MEXICAN AMERICAN      0
     3         3       NaN     0         0  CAUCASIAN              NONE      0
     4         4       NaN     1         0  CAUCASIAN              NONE      0
     …       …       …    …     …         …                …     …
```

```
1529   2566       NaN     0         0  CAUCASIAN             NaN      1
1530   2567       NaN     0         0  CAUCASIAN             NaN      1
1531   2568       NaN     1         0  CAUCASIAN             NaN      1
1532   3019       NaN     1         1    UNKNOWN             NaN      0
1533   3020       NaN     1         1    UNKNOWN             NaN      0

        CIP1  CIP2  percentile  …  qual_62EXA  qual_62EXB  qual_62EXC  \
0     520801  None    0.269828  …           I           I           I
1     260202  None    0.931034  …           I           I           I
2     522101  None    0.429310  …           I           I           I
3     150801  None    0.885345  …           I           I           I
4     143501  None    0.964655  …           I           I           I
…        …     …           …   …         …          …           …
1529  141001  None    0.970297  …           I           I           I
1530  141001  None    0.863861  …           I           I           I
1531  141001  None    0.599010  …           I           I           I
1532     Unk  None    0.500000  …           I           I           I
1533     Unk  None    0.500000  …           I           I           M

      qual_62EXE  qual_62EXG  qual_62EXH  qual_62EXI qual_63A qual_64P  \
0              I           I           I           I        D        D
1              I           I           I           I        I        D
2              I           I           I           I        D        D
3              I           I           I           I        I        D
4              I           M           I           M        M        D
…            …           …           …           …       …        …
1529           M           M           I           I        M        D
1530           M           M           I           I        M        D
1531           M           M           I           I        M        D
1532           I           I           I           I        I        P
1533           I           I           I           I        I        P

     qual_65F
0           D
1           P
2           D
3           P
4           D
…        …
1529        D
1530        D
1531        D
1532        P
1533        P

[1534 rows x 55 columns]
```

[6]:
```
afscs_fixed = pd.read_excel(filepath, sheet_name="AFSCs Fixed")
afscs_fixed
```

[6]:
```
     AFSC  USAFA Target  ROTC Target  PGL Target  Estimated  Desired  Min  \
0     13H             2            6           8         12       12   10
1     13M             3           16          19         28       27   19
2     13N            26           79         105        161      180  169
3     14F             2            5           7          9        8    7
4     14N            71          124         195        210      195  195
5     15A            14           21          35         64       70   60
6     15W             9           16          25         34       34   25
7     17X            52          129         181        193      185  181
8     21A            15           69          84         92       92   84
9     21M            17           12          29         29       29   29
10    21R            12           49          61         68       67   61
11    31P             9           20          29         35       35   29
12  32EXA             1            2           3          5        5    3
13  32EXC             2            5           7         10       10    7
14  32EXE             1            2           3          3        3    3
15  32EXF             1            2           3          5        5    3
16  32EXG            10           32          42         60       60   42
17  32EXJ             1            2           3          3        3    3
18    35P             2           16          18         22       22   18
19    38F            14           70          84         92       92   84
20    61C             0            1           1          3        3    1
21    61D             4            9          13         13       13   13
22  62EXA             3           12          15         15       15   15
23  62EXB             4            8          12         12       12   12
24  62EXC             5           15          20         24       24   24
25  62EXE            15           48          63         51       60   51
26  62EXG             5           29          34         48       48   34
27  62EXH             3            9          12         24       24   12
28  62EXI             2            0           2          2        2    2
29    63A            15           54          69         95       95   69
30    64P             8           42          50         75       75   50
31    65F             8           26          34         37       37   34

    Max  Eligible Cadets  USAFA Cadets  Mandatory Cadets  Desired Cadets  \
0    14               28             0                24               4
1    27             1534           400                 0             165
2   210             1534           400               563               0
3     8              190            56                92              12
4   210             1534           400               957             357
5    72              578           126                98             102
6    50              137            27                23               0
7   193             1534           400               527              93
8    92             1534           400                 0             740
```

9	38	1534	400	0	756
10	67	1534	400	0	341
11	35	1534	400	0	133
12	6	6	0	6	0
13	10	58	14	58	0
14	6	45	10	45	0
15	5	119	2	119	0
16	60	311	40	247	64
17	5	5	0	5	0
18	22	1534	400	25	145
19	92	1534	400	80	223
20	3	78	14	62	9
21	14	25	2	25	0
22	30	54	14	54	0
23	24	54	14	54	0
24	40	33	0	33	0
25	126	54	10	54	0
26	48	366	53	366	0
27	24	119	2	119	0
28	4	30	10	30	0
29	95	1035	400	472	356
30	75	1534	400	0	1151
31	37	1534	400	0	630

	Permitted Cadets	1st Choice Cadets	2nd Choice Cadets	3rd Choice Cadets	\
0	0	15	1	1	
1	1369	39	68	87	
2	971	29	23	25	
3	86	34	50	28	
4	220	345	260	184	
5	378	36	49	39	
6	114	25	12	11	
7	914	153	91	84	
8	794	30	63	75	
9	778	6	13	27	
10	1193	83	150	178	
11	1401	73	56	57	
12	0	5	0	0	
13	0	29	21	1	
14	0	2	8	7	
15	0	16	15	16	
16	0	28	38	27	
17	0	3	0	2	
18	1364	82	107	97	
19	1231	75	74	108	
20	7	30	11	11	
21	0	11	5	3	

22	0	25	15	2
23	0	13	14	5
24	0	12	6	11
25	0	26	12	5
26	0	32	76	72
27	0	50	19	13
28	0	3	7	6
29	207	101	123	171
30	383	78	101	102
31	904	38	33	62

	4th Choice Cadets	5th Choice Cadets	6th Choice Cadets
0	1	0	1
1	104	86	83
2	25	27	24
3	13	20	15
4	134	120	111
5	27	28	39
6	5	9	9
7	65	81	45
8	65	80	95
9	33	35	25
10	214	211	135
11	47	54	65
12	0	0	0
13	2	1	1
14	3	5	2
15	16	10	6
16	36	15	20
17	0	0	0
18	125	98	98
19	140	138	112
20	6	9	3
21	1	2	2
22	3	3	1
23	0	3	0
24	6	0	0
25	1	1	0
26	39	23	15
27	9	6	0
28	1	3	2
29	144	128	88
30	130	135	93
31	87	78	86

Assuming you have these two excel sheets in a workbook called "2023b.xlsx" in the "instances" subfolder, then this next line should work! We're going to import these files to create a problem instance for "2023b".

[7]:
```
instance = CadetCareerProblem("2023b")  # That's all you have to do!
```

```
Importing 2023b problem instance…
Imported.
```

1.2.2 Parameters

Structure Demo The code will grab this data and load it into a "unique" data structure that I've created for this model. Rather than have a bunch of variables corresponding to the parameters of this problem, I created a dictionary called "parameters" which contains all of the necessary "fixed" parameters to the problem. The fixed parameters are the ones that you as an analyst can't really change. They're inherent characteristics about the cadets and the AFSCs. Most are loaded into the dictionary as numpy arrays of various sizes. These elements are mutable too which is very convenient. Let's take a look at some of them.

[8]:
```
# Here is a list of all the "keys" to this dictionary
print(instance.parameters.keys())
```

```
dict_keys(['afsc_vector', 'P', 'quota', 'N', 'M', 'qual', 'quota_max',
'quota_min', 'utility', 'quota_e', 'quota_d', 'pgl', 'ID', 'assigned',
'ineligible', 'eligible', 'mandatory', 'desired', 'permitted', 'usafa',
'usafa_proportion', 'male', 'male_proportion', 'minority',
'minority_proportion', 'cip1', 'cip2', 'merit', 'merit_all', 'race',
'ethnicity', 'usafa_quota', 'rotc_quota', 'afsc_utility', 'c_pref_matrix',
'a_pref_matrix', 'I', 'J', 'J^E', 'J^P', 'I^E', 'num_eligible', 'I^P', 'I^D',
'sum_merit', 'J^Fixed'])
```

[9]:
```
# This is just shorthand so I don't have to type "instance.parameters" everytime
p = instance.parameters

# Numbers of Cadets, AFSCs, and AFSC preferences, respectively
for param in ['N', 'M', 'P']:
    print(param + ':', p[param])
```

```
N: 1534
M: 32
P: 6
```

[10]:
```
# Utility matrix  (Each row is a cadet, each column an AFSC)
print(p['utility'])
```

```
[[0.   0.   0.   … 0.75 1.   0.  ]
 [0.   0.75 0.   … 0.   0.   0.  ]
 [0.   0.   0.   … 0.33 0.   0.  ]
 …
 [0.   0.   0.   … 0.65 0.   0.  ]
 [0.   0.   0.   … 0.   0.   0.  ]
 [0.   0.   0.   … 0.   0.   0.  ]]
```

[11]:
```
# Qualification matrix
print(p['qual'])
```

```
[['I' 'P' 'P' … 'D' 'D' 'D']
 ['I' 'P' 'P' … 'I' 'D' 'P']
 ['I' 'P' 'P' … 'D' 'D' 'D']
 …
 ['I' 'P' 'M' … 'M' 'D' 'D']
 ['I' 'P' 'P' … 'I' 'P' 'P']
 ['I' 'P' 'P' … 'I' 'P' 'P']]
```

[12]:
```
# Array of AFSCs
print(p['afsc_vector'])
```

```
['13H' '13M' '13N' '14F' '14N' '15A' '15W' '17X' '21A' '21M' '21R' '31P'
 '32EXA' '32EXC' '32EXE' '32EXF' '32EXG' '32EXJ' '35P' '38F' '61C' '61D'
 '62EXA' '62EXB' '62EXC' '62EXE' '62EXG' '62EXH' '62EXI' '63A' '64P' '65F'
 '*']
```

The * indicates the "unmatched AFSC" which is useful for the stable marriage stuff

[13]:
```
# Sets of cadets and AFSCs
for param in ['I', 'J']:
    print(param, p[param])
```

```
I [   0    1    2 … 1531 1532 1533]
J [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
 24 25 26 27 28 29 30 31]
```

These are purely the indices of the cadets and the AFSCs, nothing else. This is not the "ID" of the cadet.

[14]:
```
# Here is the list of cadet ID's!
print(p["ID"])
```

```
[   0    1    2 … 2568 3019 3020]
```

[15]:
```
# Set of cadets that are eligible for the AFSC at index 23 (which happens to be␣
↪62EXB)
print('cadet indices:', p['I^E'][23], '\n')
print('AFSC at index 23:', p["afsc_vector"][23])
```

```
cadet indices: [   7   14   22   65   69   89  128  152  161  166  168  309  350
391
  412  420  472  475  509  534  603  612  709  731  754  756  763  773
  789  792  815  857  873  877  933  992 1044 1049 1072 1079 1365 1366
 1367 1368 1369 1370 1371 1372 1373 1374 1375 1376 1377 1378]

AFSC at index 23: 62EXB
```

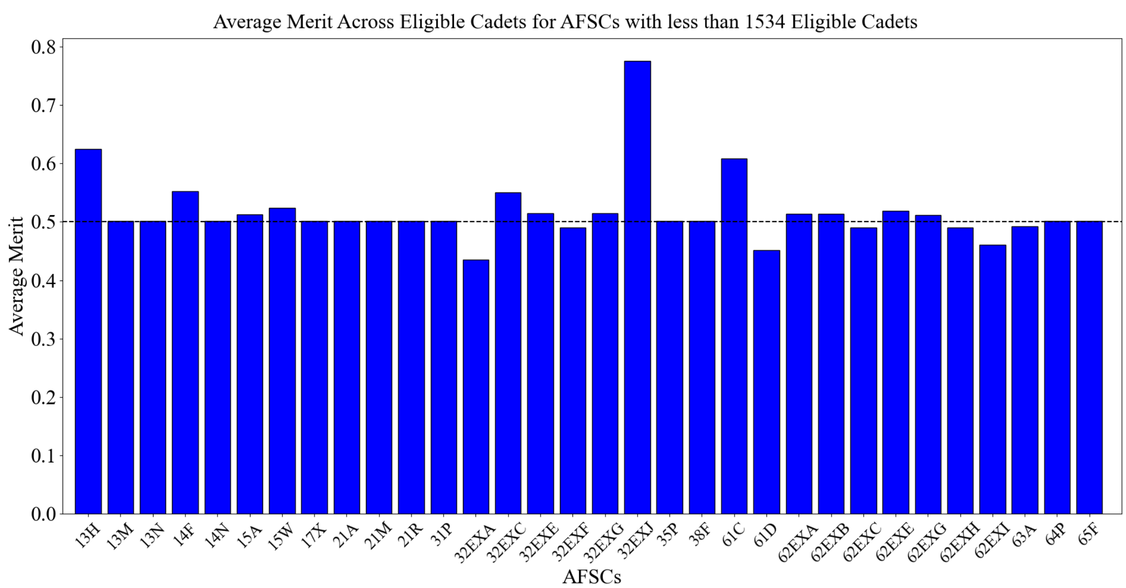
```
[16]: # Set of USAFA cadets that are eligible for the AFSC at index 23
print('USAFA cadet indices', p['I^D']['USAFA Proportion'][23])

# ['I^D'][objective][AFSC index] is how I do it there^^^^
pass
```

```
USAFA cadet indices [1365 1366 1367 1368 1369 1370 1371 1372 1373 1374 1375 1376
1377 1378]
```

```
[17]: # Set of AFSCs for which the cadet at index 0 is eligible
print('AFSC indices', p['J^E'][0])
print('AFSC names', p['afsc_vector'][p['J^E'][0]])
```

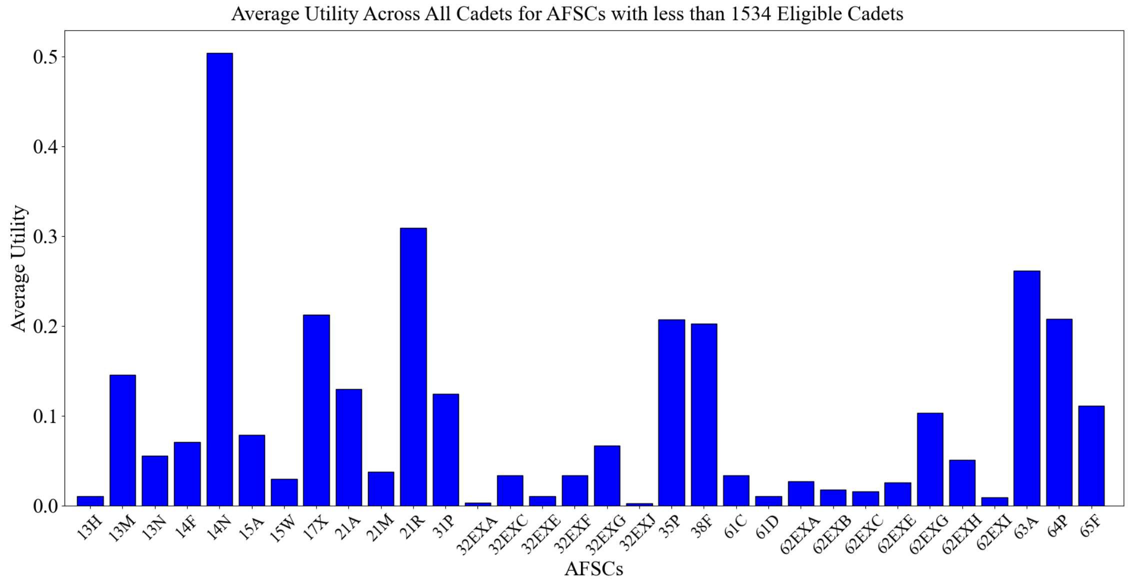
```
AFSC indices [ 1  2  4  7  8  9 10 11 18 19 29 30 31]
AFSC names ['13M' '13N' '14N' '17X' '21A' '21M' '21R' '31P' '35P' '38F' '63A'
'64P'
 '65F']
```

```
[18]: # Now let's show the data using one of our methods
chart = instance.display_data_graph({"graph":'Average Merit', "save": False,
 ↪"bar_color": "blue"})
```

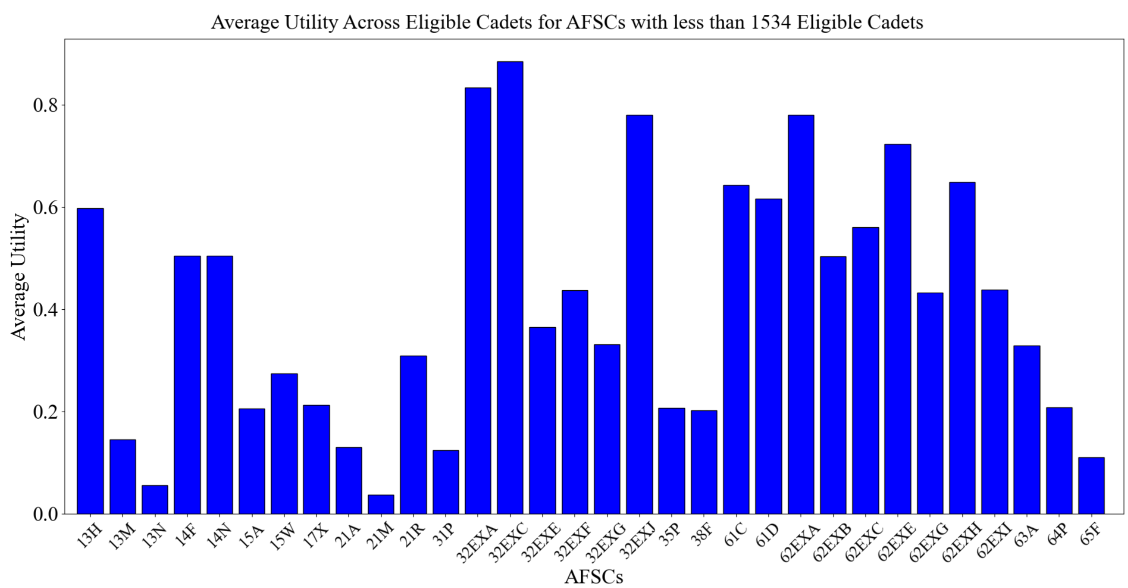


There are many other parameters to this method that control the size of the different fonts, colors, titles, and so on.

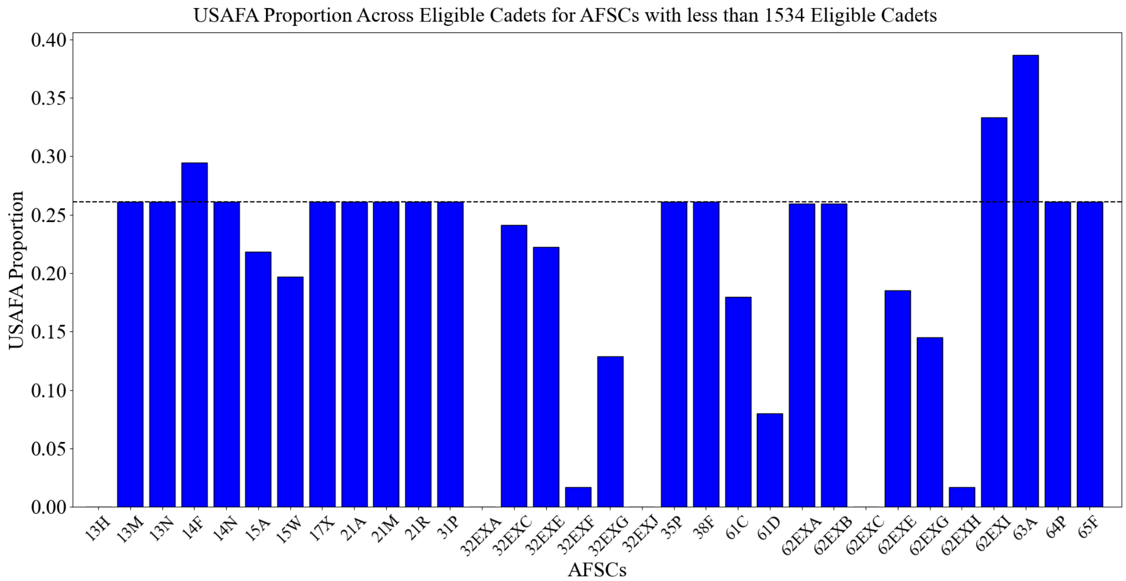
```
[19]: # We can look at the average utility placed on each of the AFSCs
chart = instance.display_data_graph({"graph": "Average Utility", "eligibility":
 ↪False})
```



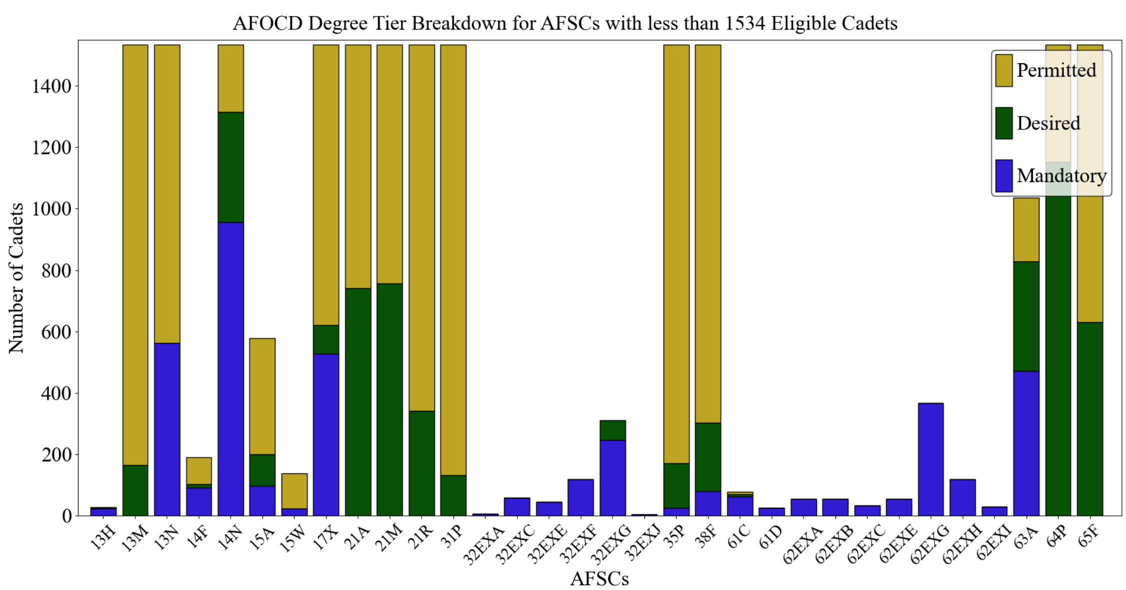
[20]:
```
# We can also look at the average utility placed on each of the AFSCs of the␣
↪set of eligible cadets for each AFSC
chart = instance.display_data_graph({"graph": "Average Utility", "eligibility":␣
↪True})

# This is probably more fair since most cadets aren't going to place␣
↪preferences on AFSCs they're not eligible for
pass
```



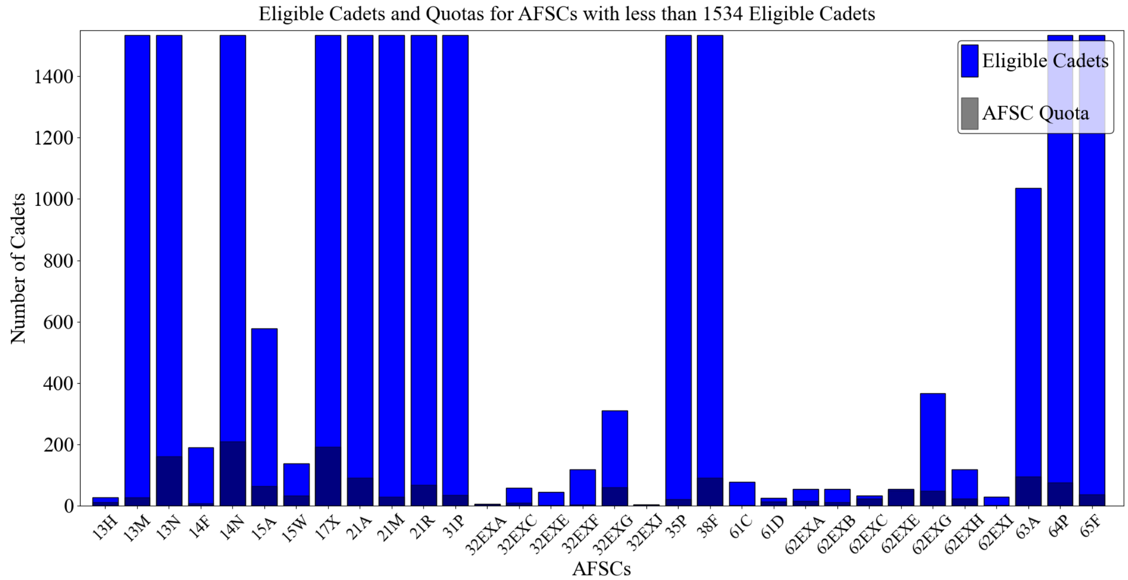
[21]:
```
chart = instance.display_data_graph({"graph": 'USAFA Proportion'})
```



[22]:
```
chart = instance.display_data_graph({"graph": 'AFOCD Data'})
```

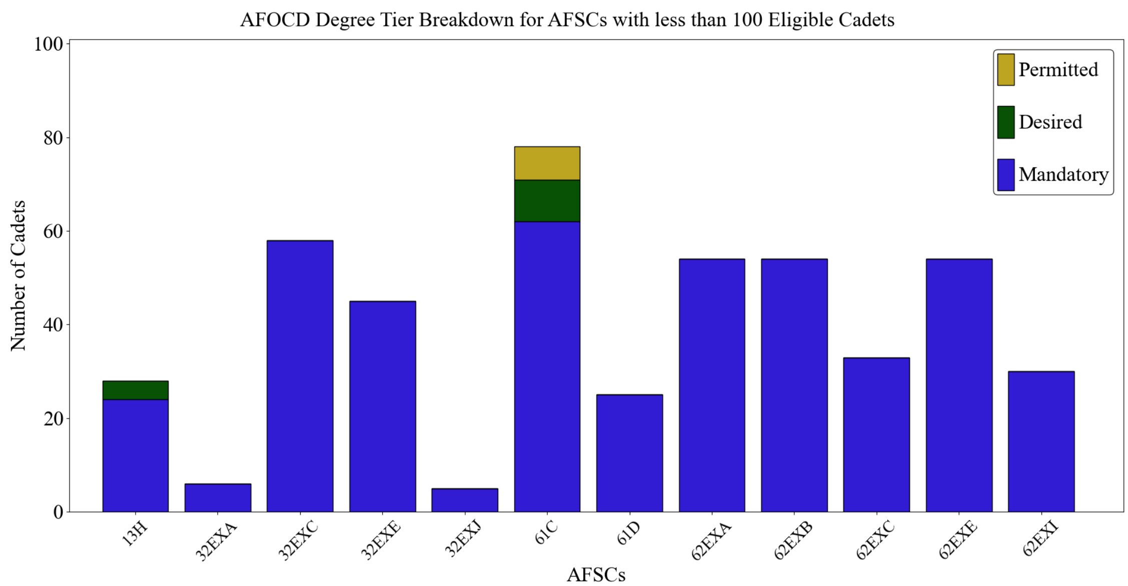


[23]:
```
chart = instance.display_data_graph({"graph": 'Eligible Quota'})
```

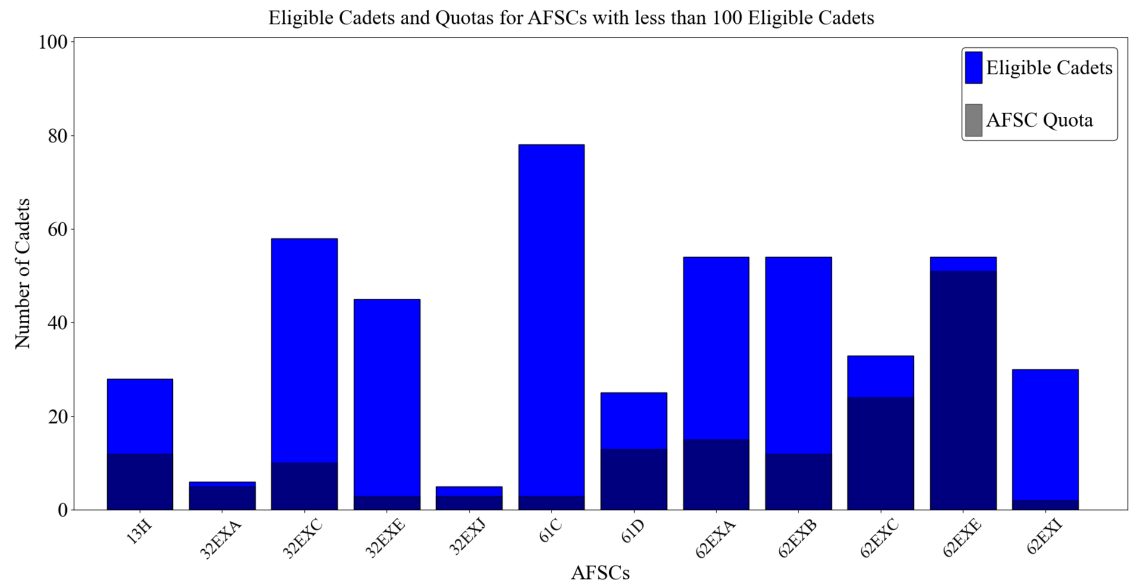


One thing you'll notice is that the title always says "for AFSCs with less than 1534 Eligible Cadets". This is because we can also "zoom in" on the smaller AFSCs if need be using the "eligibility_limit" parameter. For example

[24]:
```
chart = instance.display_data_graph({"graph": 'AFOCD Data', "eligibility_limit":
↪ 100, "skip_afscs": False})
```



```
[25]: chart = instance.display_data_graph({"graph": 'Eligible Quota',␣
        ↪"eligibility_limit": 100})
```



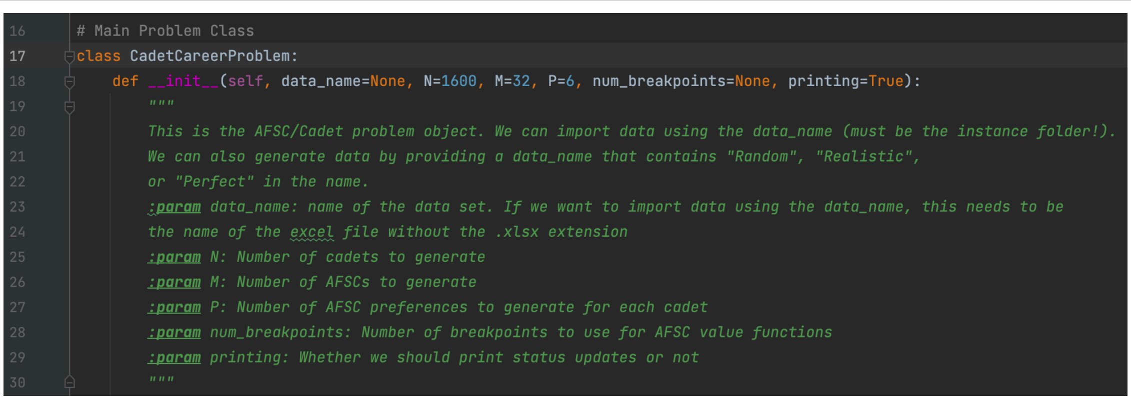
62EXE was a problem, so we had to reduce the target for that AFSC!

Source Code Walkthrough Now that I've shown what you can do with the parameters that you've loaded in, I'm going to show you how they get there in the first place. The line "instance = CadetCareerProblem('2023b')" does a lot of things up front. The class object is defined in the "problem_class.py" script which is located in the afccp/core/ directory. I did my best to try to show where everything is located using the full python module names (ie. afccp.core.problem_class = afccp/core/problem_class.py) All throughout the code you will see functions defined using the full location to try to help you understand where everything is.

If we open up problem_class.py, we will see the class defined. Let's look at its "**init**" function, which is what gets executed when we define a problem instance.

```
[26]: Image(filename='pic2.png')
```

[26]:



```
# Main Problem Class
class CadetCareerProblem:
    def __init__(self, data_name=None, N=1600, M=32, P=6, num_breakpoints=None, printing=True):
        """
        This is the AFSC/Cadet problem object. We can import data using the data_name (must be the instance folder!).
        We can also generate data by providing a data_name that contains "Random", "Realistic",
        or "Perfect" in the name.
        :param data_name: name of the data set. If we want to import data using the data_name, this needs to be
        the name of the excel file without the .xlsx extension
        :param N: Number of cadets to generate
        :param M: Number of AFSCs to generate
        :param P: Number of AFSC preferences to generate for each cadet
        :param num_breakpoints: Number of breakpoints to use for AFSC value functions
        :param printing: Whether we should print status updates or not
        """
```

Essentially, when we create a problem instance we can do one of two things: import an already existing instance, or generate a new one. The "data_name" function parameter determines which problem instance to import, or alternatively generate. All of these parameters are defined in the screenshot above. Let's press on through this initialization class method.

[27]:
```
Image(filename='pic3.png')
```

[27]:

```
32        # Get list of generated data name problem instances
33        self.generated_data_names = {'Random': [], 'Perfect': [], 'Realistic': []}
34        for file_name in glob.iglob(afccp.core.globals.paths['instances'] + '*.xlsx', recursive=True):
35
36            # Start of filename
37            start_index = file_name.find(
38                afccp.core.globals.paths['instances']) + len(afccp.core.globals.paths['instances']) + 1
39            end_index = len(file_name) - 5  # Remove ".xlsx"
40            full_name = file_name[start_index:end_index]  # Name of the file (not the path)
41            sections = full_name.split(' ')  # Split the filename by each ' ' (space)
42            d_name = sections[0]  # Second part is the "data_name"
43
44            # Loop through each of the kinds of generated data (random, perfect, realistic)
45            for variant in self.generated_data_names:
46
47                # Ex. "Random" in "Random_1" and "Random_1" isn't already in the list
48                if variant in d_name and d_name not in self.generated_data_names[variant]:
49                    self.generated_data_names[variant].append(d_name)  # add this data name to the list!
```

This block of code is here to gather lists of already existing problem instances that have been generated. There are three categories: "Random", "Realistic", and "Perfect". Random data is just that, fake cadets and AFSCs that have been generated randomly. Realistic data is a set of cadets that have been generated through CTGAN using realistic data distributions. Perfect data is my own thought experiment about generating cadets that perfectly align themselves with what the AFSCs want. The objective value of the best solution should theoretically be 1.

Anyway, we loop through each file in the "instances" folder and check to see if it is a Random, Perfect, or Realistic dataset. We then add the file to the appropriate list. This will be used in the next block of code. Line 34 is an example of how I've included the module path in the name of a python object. "afccp.core.globals.paths" is a global variable that is defined in the globals.py script. It is a dictionary of folder paths within the directory. I did this so that you would see where the variable or function is located for reference.

[28]:
```
Image(filename='pic4.png')
```

[28]:

```
51         # Get correct data attributes
52         if data_name is None:
53 
54             # If we didn't specify a data_name, we're just going to generate a random set of cadets
55             self.data_name = "Random_" + str(len(self.generated_data_names["Random"]) + 1)
56             self.data_variant = "Random"
57             generate = True
58         else:
59 
60             # We specified a data_name
61             self.data_name = data_name
62             generate = False
63 
64             # Loop through "Random", "Realistic", "Perfect"
65             for data_variant in self.generated_data_names:
66 
67                 # If we passed one of those three names generally, we will generate a new instance of that kind
68                 if data_variant == self.data_name:
69                     self.data_name = data_variant + "_" + str(len(self.generated_data_names[data_variant]) + 1)
70                     generate = True
71                     break
72 
73                 # If we specified a specific version ("Random_4" for example), then we'll load it in
74                 elif data_variant in self.data_name and '_' in self.data_name:
75                     generate = False
76                     break
```

This block of code does two things: determine the "name" of the dataset and whether or not we're generating new data. Lines 52-57 are what happens by default if we don't pass a name of a problem instance. We generate random data. The reason why we put all of the generated data names into their appropriate lists was to determine how many of each kind of generated datasets we have. If we already have 4 "Random" data files, then this new one will be "Random_5", for example.

The "else" statement handles the situation where we did pass a data name. By default, we assume it's an imported dataset. We then loop through each kind of generated data and determine if this dataset fits one of those categories. If it does, we then determine whether we're importing a pre-existing generated dataset, or we're creating a new one.

[29]:
```
Image(filename='pic5.png')
```

[29]:

```python
78          # Figure out the data variant
79          self.data_variant = None
80          if len(self.data_name) == 1:  # A, B, C, D, etc.
81              self.data_variant = "Scrubbed"
82
83          else:
84
85              # Random, Realistic, or Perfect
86              for data_variant in self.generated_data_names:
87                  if data_variant in self.data_name:
88                      self.data_variant = data_variant
89
90              # If we still haven't found the right data variant, we know it's a real class year
91              if self.data_variant is None:
92                  self.data_variant = "Year"  # 2018, 2019, 2020, etc.
93
94          # Get correct filepath  (for importing/exporting to)
95          self.filepath = afccp.core.globals.paths['instances'] + self.data_name + '.xlsx'
96
97          # Create a "results" folder for this problem instance
98          if not os.path.exists("results/" + self.data_name):
99              os.makedirs("results/" + self.data_name)
100
101         # Create multiple "figures" folders for this problem instance
102         if not os.path.exists("figures/" + self.data_name):
103             os.makedirs("figures/" + self.data_name + "/value parameters")
104             os.makedirs("figures/" + self.data_name + "/results")
105             os.makedirs("figures/" + self.data_name + "/slides")
106             os.makedirs("figures/" + self.data_name + "/data")
```

Lines 78-92 are here to determine the “variant” of the data. The five options are “Scrubbed”, “Year”, “Random”, “Realistic”, and “Perfect”. This helps indicate differences in what we’d expect the data to look like for various areas throughout the code. Lines 98-106 create new subfolders corresponding to this particular problem instance if they haven’t already been created. Lines 108-127 simply initialize many different attributes of the CadetCareerProblem class and therefore I didn’t include a screenshot here.

[30]:
```python
Image(filename='pic6.png')
```

[30]:

```python
        # Check if we're generating data, and the data variant
        if self.data_variant == 'Random' and generate:

            if printing:
                print('Generating ' + self.data_name + ' problem instance...')
            parameters = afccp.core.handling.simulation_functions.simulate_model_fixed_parameters(N=N, P=P, M=M)
            self.parameters = afccp.core.handling.data_handling.model_fixed_parameters_set_additions(parameters)

        elif self.data_variant == 'Perfect' and generate:

            if printing:
                print('Generating ' + self.data_name + ' problem instance...')
            parameters, self.solution = \
                afccp.core.handling.simulation_functions.perfect_example_generator(N=N, P=P, M=M)
            self.parameters = afccp.core.handling.data_handling.model_fixed_parameters_set_additions(parameters)

        elif self.data_variant == 'Realistic' and generate:

            if printing:
                print('Generating ' + self.data_name + ' problem instance...')

            if use_sdv:
                data = afccp.core.handling.simulation_functions.simulate_realistic_fixed_data(N=N)
                cadets_fixed, afscs_fixed = \
                    afccp.core.handling.simulation_functions.convert_realistic_data_parameters(data)
                parameters = afccp.core.handling.data_handling.model_fixed_parameters_from_data_frame(
                    cadets_fixed, afscs_fixed)
            else:
                parameters = afccp.core.handling.simulation_functions.simulate_model_fixed_parameters(N=N, P=P, M=M)

            self.parameters = afccp.core.handling.data_handling.model_fixed_parameters_set_additions(parameters)
```

This block of code is where we actually generate the data. Again, I've included the full module name of the functions to show where the functions are defined. If you want to look further into the code that generates the parameters, you're welcome to do so!

[31]:
```python
Image(filename='pic7.png')
```

[31]:
```python
        else:

            if printing:
                print('Importing ' + self.data_name + ' problem instance...')

            # If the path exists, import the data. If not, raise an error
            if os.path.exists(self.filepath):
                self.info_df, self.parameters, self.vp_dict, self.solution_dict, self.metrics_dict, self.gp_df, \
                self.similarity_matrix = afccp.core.comprehensive_functions.import_aggregate_instance_file(
                    self.filepath, num_breakpoints=num_breakpoints)
            else:
                raise ValueError("Instance '" + self.data_name + "' not found at path '" +
                                 afccp.core.globals.paths['instances'] + "'")

        # Initialize more "functional" parameters
        self.plt_p, self.mdl_p = \
            afccp.core.handling.ccp_helping_functions.initialize_instance_functional_parameters(self.parameters["N"])

        if printing:
            if generate:
                print('Generated.')
            else:
                print('Imported.')
```

If we're not generating data, then we're importing it. Lines 160-172 contain the "else" statement that imports the data from excel. I included some error handling to ensure that there is a file with the "data_name" you specified in the instances folder. Lines 174-176 grab some more class parameters that are initialized in the "ccp_helping_functions.py" script. The attribute "plt_p" is a dictionary of parameters controlling the various plots/graphs that you can create (things like color, size, legend, etc.). Alternatively, "mdl_p" is a dictionary of parameters controlling how you solve the various models to find solutions (things like solve time, solver used, genetic algorithm hyper parameters, etc.)

Now that you know a little bit about how the code is all set up, feel free to explore whatever other functions you want to see how they all work. I will continue with the other components needed to get this code/model to work properly!

1.2.3 Value Parameters

Data Explanation The next thing we need to do is generate our "value parameters". These are the things that the analyst (but eventually the other decision makers hopefully) can control. These are all the various objectives, weights, constraints, and value functions. I have excel sheets that contain the defaults for generating these different components. These "default value parameter" excel sheets are located in the support folder. This excel file should be where you make all of your initial adjustments to the value parameters for a new class year. Let's import the various dataframes to see what's going on here.

[32]:
```
import openpyxl

# File path
filepath = dir_path + "support/Value_Parameters_Defaults_2023b.xlsx"

# Load workbook and get sheet names
wb = openpyxl.load_workbook(filepath)
sheet_names = wb.sheetnames

# Load in dataframes
dfs = {}
for sheet_name in sheet_names:
    dfs[sheet_name] = pd.read_excel(filepath, sheet_name=sheet_name)
```

The above code just loads in each of the excel sheets into a pandas dataframe.

[33]:
```
dfs["Overall Weights"]
```

[33]:
```
   Cadets Overall  AFSCs Overall  AFSCs Min Value  Cadets Min Value  \
0             0.7            0.3                0                 0

  Cadet Weight Function AFSC Weight Function USAFA-Constrained AFSCs  \
0               Curve_1               Custom      35P, 38F, 64P, 65F
```

	Similarity Constraint	Cadets Top 3 Constraint
0	NaN	NaN

The "Overall Weights" df contains the overall weights on Cadets/AFSCs, the minimum overall values on Cadets/AFSCs, the weight functions for the individual weights on Cadets/AFSCs, and the USAFA-Constrained AFSCs. To see a list of all the weight functions for both cadets and AFSCs, look at their respective functions on the "value_parameter_handling.py" script in the "afccp/core/handling" directory. A "custom" AFSC weight function means that we will use the AFSC weights that are explicitly defined in the next dataframe. The USAFA-Constrained AFSCs are the ones that the SecAF has stated must have a limit on the USAFA cadets accessed. As it stands the constraint is as follows: No more than 5% of the total USAFA class may be collectively assigned to these four AFSCs. Don't worry about the similarity constraint, that may come into play later on.

[34]:
```
dfs["AFSC Weights"]
```

[34]:

	AFSC	AFSC Swing Weight	AFSC Min Value
0	13H	6.42	0
1	13M	6.44	0
2	13N	100.00	0
3	14F	6.42	0
4	14N	83.33	0
5	15A	20.00	0
6	15W	6.46	0
7	17X	83.32	0
8	21A	21.82	0
9	21M	30.00	0
10	21R	8.19	0
11	31P	6.48	0
12	32EXA	6.41	0
13	32EXC	6.42	0
14	32EXE	6.41	0
15	32EXF	6.41	0
16	32EXG	6.67	0
17	32EXJ	6.41	0
18	35P	6.43	0
19	38F	21.82	0
20	61C	6.41	0
21	61D	6.42	0
22	62EXA	6.43	0
23	62EXB	6.42	0
24	62EXC	6.44	0
25	62EXE	8.58	0
26	62EXG	6.52	0
27	62EXH	6.42	0
28	62EXI	6.41	0
29	63A	30.00	0

30	64P	20.00	0
31	65F	10.00	0

If we selected "Custom" for the AFSC weight function in the Overall Weights df, then we will use these weights for each of the AFSCs. This was to ensure 13N got some special attention. Don't worry about the min values for the AFSCs, we never actually constrain them.

[35]:
```
dfs["AFSC Objective Weights"]
```

[35]:

	AFSC	Norm Score	Merit	USAFA Proportion	Combined Quota	\
0	13H	4.999999	30.000001	0.000000	100	
1	13M	4.999999	30.000000	0.000000	100	
2	13N	5.000000	90.000000	20.000000	100	
3	14F	4.999999	29.999998	0.000000	100	
4	14N	5.000002	30.000000	20.000000	100	
5	15A	4.999999	30.000000	0.000000	100	
6	15W	5.000000	30.000002	0.000000	100	
7	17X	5.000000	30.000000	20.000000	100	
8	21A	5.000001	30.000001	19.999999	100	
9	21M	4.999999	80.000000	0.000000	100	
10	21R	5.000000	30.000002	20.000001	100	
11	31P	4.999999	30.000000	0.000000	100	
12	32EXA	5.000000	29.999999	0.000000	100	
13	32EXC	5.000002	30.000000	0.000000	100	
14	32EXE	5.000001	30.000000	0.000000	100	
15	32EXF	5.000002	30.000000	0.000000	100	
16	32EXG	5.000001	30.000000	20.000001	100	
17	32EXJ	5.000000	29.999999	0.000000	100	
18	35P	4.999999	29.999998	0.000000	100	
19	38F	5.000001	29.999999	0.000000	100	
20	61C	5.000000	29.999999	0.000000	100	
21	61D	4.999999	30.000000	0.000000	100	
22	62EXA	5.000000	29.999999	0.000000	100	
23	62EXB	5.000002	30.000000	0.000000	100	
24	62EXC	5.000000	29.999999	0.000000	100	
25	62EXE	5.000001	29.999999	5.000001	100	
26	62EXG	5.000000	29.999999	0.000000	100	
27	62EXH	5.000000	29.999999	0.000000	100	
28	62EXI	5.000000	29.999999	0.000000	100	
29	63A	5.000000	29.999999	20.000001	100	
30	64P	5.000000	29.999999	0.000000	100	
31	65F	4.999999	30.000000	0.000000	100	

	USAFA Quota	ROTC Quota	Mandatory	Desired	Permitted	Utility	Male	\
0	0	0	89.999999	50.000000	0.000000	4.999999	0.0	
1	0	0	0.000000	70.000000	30.000000	4.999999	0.0	
2	0	0	90.000000	0.000000	30.000000	30.000000	0.0	
3	0	0	89.999998	49.999998	29.999998	4.999999	0.0	

4	0	0	90.000000	50.000000	30.000000	5.000002	0.0
5	0	0	90.000002	50.000002	30.000000	40.000001	0.0
6	0	0	90.000002	0.000000	30.000002	5.000000	0.0
7	0	0	90.000000	70.000000	30.000000	20.000000	0.0
8	0	0	0.000000	50.000000	30.000001	40.000002	0.0
9	0	0	0.000000	70.000002	30.000002	20.000000	0.0
10	0	0	0.000000	70.000001	30.000002	5.000000	0.0
11	0	0	0.000000	70.000000	30.000000	4.999999	0.0
12	0	0	90.000000	0.000000	0.000000	5.000000	0.0
13	100	0	90.000000	0.000000	0.000000	5.000002	0.0
14	0	0	90.000001	0.000000	0.000000	5.000001	70.0
15	100	0	90.000000	0.000000	0.000000	5.000002	0.0
16	0	0	70.000000	70.000000	0.000000	5.000001	0.0
17	0	0	90.000000	0.000000	0.000000	5.000000	0.0
18	0	0	89.999998	49.999998	29.999998	4.999999	0.0
19	0	0	90.000001	59.999999	29.999999	40.000001	0.0
20	0	100	89.999999	0.000000	29.999999	5.000000	0.0
21	0	0	89.999999	0.000000	0.000000	15.000000	0.0
22	0	0	90.000000	0.000000	0.000000	5.000000	0.0
23	100	0	90.000000	0.000000	0.000000	5.000002	0.0
24	0	0	90.000000	0.000000	0.000000	5.000000	0.0
25	0	0	90.000000	0.000000	0.000000	5.000001	0.0
26	0	0	90.000000	0.000000	0.000000	5.000000	0.0
27	0	0	90.000000	0.000000	0.000000	5.000000	0.0
28	0	0	90.000000	0.000000	0.000000	5.000000	0.0
29	0	0	89.999999	69.999998	39.999999	5.000000	0.0
30	0	0	0.000000	60.000001	29.999999	5.000000	0.0
31	0	0	0.000000	70.000000	30.000000	4.999999	0.0

	Minority
0	0
1	0
2	0
3	0
4	0
5	0
6	0
7	0
8	0
9	0
10	0
11	0
12	0
13	0
14	0
15	0
16	0

```
17           0
18           0
19           0
20           0
21           0
22           0
23           0
24           0
25           0
26           0
27           0
28           0
29           0
30           0
31           0
```

Here are the objective weights for each AFSC for each objective. Like the previous df, these are swing weights that will be scaled for each AFSC so that they sum to 1. The "Norm Score" objective is the new one that incorporates AFSC preference lists and could essentially replace all the others (except quota). USAFA proportion right now only applies to the large AFSCs, and the USAFA/ROTC quota objectives were just for some of the smaller AFSCs to ensure that each of them got some USAFA cadets or ROTC cadets depending on what they requested. The AFOCD objectives (Mandatory, Desired, Permitted) must be checked against the new AFOCD! These are current as of April 2022. If an AFSC changes their tier, make sure to reflect that here. Some of them are zero'd out because that AFSC doesn't have that particular requirement level. Utility is cadet preference! Don't worry about Male/Minority objectives (32EXE needed one female-> that's why it had a non-zero weight)

[36]:
```
dfs["AFSC Objective Targets"]
```

[36]:
```
     AFSC  Norm Score     Merit  USAFA Proportion  Combined Quota  \
0     13H           1  0.501835          0.325426              12
1     13M           1  0.501835          0.325426              27
2     13N           1  0.501835          0.260756             180
3     14F           1  0.501835          0.325426               8
4     14N           1  0.501835          0.260756             195
5     15A           1  0.501835          0.325426              70
6     15W           1  0.501835          0.325426              34
7     17X           1  0.501835          0.260756             185
8     21A           1  0.501835          0.260756              92
9     21M           1  0.501835          0.325426              29
10    21R           1  0.501835          0.260756              67
11    31P           1  0.501835          0.325426              35
12  32EXA           1  0.501835          0.000000               5
13  32EXC           1  0.501835          0.325426              10
14  32EXE           1  0.501835          1.000000               3
15  32EXF           1  0.501835          0.000000               5
16  32EXG           1  0.501835          0.260756              60
```

17	32EXJ	1	0.501835	0.325426	3
18	35P	1	0.501835	0.000000	22
19	38F	1	0.501835	0.281924	92
20	61C	1	0.501835	0.325426	3
21	61D	1	0.501835	0.325426	13
22	62EXA	1	0.501835	0.325426	15
23	62EXB	1	0.501835	0.325426	12
24	62EXC	1	0.501835	0.325426	24
25	62EXE	1	0.501835	0.260756	60
26	62EXG	1	0.501835	0.325426	48
27	62EXH	1	0.501835	0.325426	24
28	62EXI	1	0.501835	1.000000	2
29	63A	1	0.501835	0.260756	95
30	64P	1	0.501835	0.281924	75
31	65F	1	0.501835	0.000000	37

	USAFA Quota	ROTC Quota	Mandatory	Desired	Permitted	Utility	Male \
0	0	0	0.90	0.10	0.00	1	0.720000
1	0	0	0.00	0.80	0.20	1	0.720000
2	0	0	0.10	0.00	0.90	1	0.720000
3	0	0	0.70	0.15	0.15	1	0.720000
4	0	0	0.75	0.20	0.05	1	0.720000
5	0	0	0.65	0.25	0.10	1	0.720000
6	0	0	0.60	0.00	0.40	1	0.720000
7	0	0	0.70	0.20	0.10	1	0.720000
8	0	0	0.00	0.75	0.25	1	0.720000
9	0	0	0.00	0.80	0.20	1	0.720000
10	0	0	0.00	0.80	0.20	1	0.720000
11	0	0	0.00	0.80	0.20	1	0.720000
12	0	0	1.00	0.00	0.00	1	0.720000
13	2	0	1.00	0.00	0.00	1	0.720000
14	0	0	1.00	0.00	0.00	1	0.656454
15	1	0	1.00	0.00	0.00	1	0.720000
16	0	0	0.80	0.20	0.00	1	0.720000
17	0	0	1.00	0.00	0.00	1	0.720000
18	0	0	0.70	0.20	0.10	1	0.720000
19	0	0	0.20	0.65	0.15	1	0.720000
20	0	1	0.60	0.30	0.10	1	0.720000
21	0	0	1.00	0.00	0.00	1	0.720000
22	0	0	1.00	0.00	0.00	1	0.720000
23	4	0	1.00	0.00	0.00	1	0.720000
24	0	0	1.00	0.00	0.00	1	0.720000
25	0	0	1.00	0.00	0.00	1	0.720000
26	0	0	1.00	0.00	0.00	1	0.720000
27	0	0	1.00	0.00	0.00	1	0.720000
28	0	0	1.00	0.00	0.00	1	0.720000
29	0	0	0.20	0.70	0.10	1	0.720000

```
30            0            0       0.00      0.70        0.30          1  0.720000
31            0            0       0.00      0.70        0.30          1  0.720000

    Minority
0       0.27
1       0.27
2       0.27
3       0.27
4       0.27
5       0.27
6       0.27
7       0.27
8       0.27
9       0.27
10      0.27
11      0.27
12      0.27
13      0.27
14      0.27
15      0.27
16      0.27
17      0.27
18      0.27
19      0.27
20      0.27
21      0.27
22      0.27
23      0.27
24      0.27
25      0.27
26      0.27
27      0.27
28      0.27
29      0.27
30      0.27
31      0.27
```

This df displays the target measure for each of the objectives. Ideally, this is what the objective measures should be. This doesn't really matter a whole lot except for the AFOCD objectives. Again, make sure this matches what the actual AFOCD says! These are the target proportions for each tier for each AFSC. Where there is a 0, there should also be a 0 in the objective weights slot.

[37]:
```
dfs["AFSC Objective Min Value"]
```

[37]:
```
     AFSC  Norm Score    Merit USAFA Proportion Combined Quota USAFA Quota  \
0     13H           0  0.35, 5                0         10, 14           0
1     13M           0  0.35, 5                0         19, 28           0
2     13N           0  0.39, 5         0.15, 0.4       162, 210           0
```

3	14F	0	0.35, 5	0	7, 9	0
4	14N	0	0.35, 5	0.15, 0.4	195, 210	0
5	15A	0	0.35, 5	0	60, 72	0
6	15W	0	0.35, 5	0	25, 50	0
7	17X	0	0.35, 5	0.15, 0.4	181, 193	0
8	21A	0	0.35, 5	0.15, 0.4	84, 92	0
9	21M	0	0.35, 5	0	29, 38	0
10	21R	0	0.35, 5	0.15, 0.4	61, 68	0
11	31P	0	0.35, 5	0	29, 35	0
12	32EXA	0	0.35, 5	0	3, 6	0
13	32EXC	0	0.35, 5	0	7, 10	2, 10
14	32EXE	0	0.35, 5	0	3, 6	0
15	32EXF	0	0.35, 5	0	3, 5	1, 5
16	32EXG	0	0.35, 5	0.15, 0.4	42, 60	0
17	32EXJ	0	0.35, 5	0	3, 5	0
18	35P	0	0.35, 5	0	18, 22	0
19	38F	0	0.35, 5	0	84, 92	0
20	61C	0	0.35, 5	0	1, 3	0
21	61D	0	0.35, 5	0	13, 14	0
22	62EXA	0	0.35, 5	0	15, 30	0
23	62EXB	0	0.35, 5	0	12, 24	4, 24
24	62EXC	0	0.35, 5	0	24, 40	0
25	62EXE	0	0.35, 5	0	51, 126	0
26	62EXG	0	0.35, 5	0	34, 48	0
27	62EXH	0	0.35, 5	0	12, 24	0
28	62EXI	0	0.35, 5	0	2, 4	0
29	63A	0	0.35, 5	0.15, 0.4	69, 95	0
30	64P	0	0.35, 5	0	50, 75	0
31	65F	0	0.35, 5	0	34, 37	0

	ROTC Quota	Mandatory	Desired	Permitted	Utility	Male	Minority
0	0	0.9, 5	0.1, 5	0	0	0	0
1	0	0	0.789, 5	0	0	0	0
2	0	0.1, 5	0	0	0.4945, 5	0	0
3	0	0.7, 5	0, 0.15	0	0	0	0
4	0	0.7475, 5	0.2, 5	0	0	0	0
5	0	0.65, 5	0, 0.25	0, 0.1	0	0	0
6	0	0.588, 5	0	0	0	0	0
7	0	0.69, 5	0, 0.2	0	0	0	0
8	0	0	0.75, 5	0	0	0	0
9	0	0	0.8, 5	0	0	0	0
10	0	0	0.8, 5	0	0	0	0
11	0	0	0.685, 5	0	0	0	0
12	0	1, 5	0	0	0	0	0
13	0	1, 5	0	0	0	0	0
14	0	1, 5	0	0	0	0, 0.8	0
15	0	1, 5	0	0	0	0	0

16	0	0.8, 5	0, 0.2	0	0	0	0
17	0	1, 5	0	0	0	0	0
18	0	0.68, 5	0.2, 5	0	0	0	0
19	0	0.1775, 5	0.65, 5	0	0	0	0
20	1, 1	0.6, 5	0.3, 5	0	0	0	0
21	0	1, 5	0	0	0	0	0
22	0	1, 5	0	0	0	0	0
23	0	1, 5	0	0	0	0	0
24	0	1, 5	0	0	0	0	0
25	0	0.94, 5	0	0	0	0	0
26	0	1, 5	0	0	0	0	0
27	0	1, 5	0	0	0	0	0
28	0	1, 5	0	0	0	0	0
29	0	0.2, 5	0.6805, 5	0	0	0	0
30	0	0	0.7, 5	0	0	0	0
31	0	0	0.7, 5	0	0	0	0

These are the constraints for each objective for each AFSC. Most are determined automatically based on the "fixed" data. For example, the Combined Quota constraint is determined by the "Min, Max" values in "AFSCs_Fixed". The USAFA Proportion objective constraint should match the weight (if there is a zero in one, there is a zero in the other). I constrained Utility for 13N and I ensured 32EXE had one female. Here is another place where AFOCD matters! Make sure the constraints match the targets! This is the place where you should list all the potential constraints that you'd have (this doesn't toggle them on/off however, that's the next dataframe).

[38]:
```python
dfs["Constraint Type"]
```

[38]:

	AFSC	Norm Score	Merit	USAFA Proportion	Combined Quota	USAFA Quota	\
0	13H	0	4	0	4	0	
1	13M	0	4	0	4	0	
2	13N	0	4	4	4	0	
3	14F	0	4	0	4	0	
4	14N	0	4	4	4	0	
5	15A	0	4	0	4	0	
6	15W	0	4	0	4	0	
7	17X	0	4	4	4	0	
8	21A	0	4	4	4	0	
9	21M	0	4	0	4	0	
10	21R	0	4	4	4	0	
11	31P	0	4	0	4	0	
12	32EXA	0	4	0	4	0	
13	32EXC	0	4	0	4	4	
14	32EXE	0	4	0	4	0	
15	32EXF	0	4	0	4	4	
16	32EXG	0	4	4	4	0	
17	32EXJ	0	4	0	4	0	
18	35P	0	4	0	4	0	
19	38F	0	4	0	4	0	

20	61C	0	4	0	4	0
21	61D	0	4	0	4	0
22	62EXA	0	4	0	4	0
23	62EXB	0	4	0	4	4
24	62EXC	0	4	0	4	0
25	62EXE	0	4	0	4	0
26	62EXG	0	4	0	4	0
27	62EXH	0	4	0	4	0
28	62EXI	0	4	0	4	0
29	63A	0	4	4	4	0
30	64P	0	4	0	4	0
31	65F	0	4	0	4	0

	ROTC Quota	Mandatory	Desired	Permitted	Utility	Male	Minority
0	0	4	0	0	0	0	0
1	0	0	3	0	0	0	0
2	0	4	0	0	4	0	0
3	0	4	0	0	0	0	0
4	0	4	3	0	0	0	0
5	0	4	4	4	0	0	0
6	0	4	0	0	0	0	0
7	0	4	0	0	0	0	0
8	0	0	0	0	0	0	0
9	0	0	3	0	0	0	0
10	0	0	3	0	0	0	0
11	0	0	3	0	0	0	0
12	0	4	0	0	0	0	0
13	0	4	0	0	0	0	0
14	0	4	0	0	0	4	0
15	0	4	0	0	0	0	0
16	0	4	0	0	0	0	0
17	0	4	0	0	0	0	0
18	0	4	0	0	0	0	0
19	0	4	0	0	0	0	0
20	4	4	0	0	0	0	0
21	0	4	0	0	0	0	0
22	0	4	0	0	0	0	0
23	0	4	0	0	0	0	0
24	0	4	0	0	0	0	0
25	0	4	0	0	0	0	0
26	0	4	0	0	0	0	0
27	0	4	0	0	0	0	0
28	0	4	0	0	0	0	0
29	0	4	4	0	0	0	0
30	0	0	3	0	0	0	0
31	0	0	3	0	0	0	0

Here is where you actually turn different constraints on or off. If there is a 0, the constraint is turned off. 1s and 2s related to "value" constraints instead of "measure" constraints and I ultimately decided against using them for a number of reasons. The only ones you should be playing with are 3s and 4s. A "3" is an "approximate" constraint. This means that the denominator is the PGL target for an AFSC, not the actual number of cadets assigned. If this is confusing, please reference my thesis or my slides that talk about the difference between the Approximate Model and the Exact Model. The "4", therefore, is an "exact" constraint. The only place where we could legimately use a "3" instead of a "4" is for the AFOCD constraints.

Example: Let's say 14N wants 70% of their cadets to have mandatory-tiered degrees. Let's say the PGL is 190 and we assign 220 cadets. A "3" constraint is a less restrictive constraint, and would ensure that 133 cadets (190 * 0.70) have "M" degrees. Alternatively, a "4" constraint ensures the actual proportion gets constrained, so 154 cadets (220 * 0.70) will have "M" degrees. Sometimes it is really hard to meet the AFOCD for some AFSCs, and so a "3" constraint is necessary to ensure we meet the target based on the PGL, not the actual number of cadets.

[39]:
```
dfs["Value Functions"]
```

[39]:
```
     AFSC          Norm Score                Merit  \
0     13H  Min Increasing|0.3  Min Increasing|-0.3
1     13M  Min Increasing|0.3  Min Increasing|-0.3
2     13N  Min Increasing|0.3  Min Increasing|-0.3
3     14F  Min Increasing|0.3  Min Increasing|-0.3
4     14N  Min Increasing|0.3  Min Increasing|-0.3
5     15A  Min Increasing|0.3  Min Increasing|-0.3
6     15W  Min Increasing|0.3  Min Increasing|-0.3
7     17X  Min Increasing|0.3  Min Increasing|-0.3
8     21A  Min Increasing|0.3  Min Increasing|-0.3
9     21M  Min Increasing|0.3  Min Increasing|-0.3
10    21R  Min Increasing|0.3  Min Increasing|-0.3
11    31P  Min Increasing|0.3  Min Increasing|-0.3
12  32EXA  Min Increasing|0.3  Min Increasing|-0.3
13  32EXC  Min Increasing|0.3  Min Increasing|-0.3
14  32EXE  Min Increasing|0.3  Min Increasing|-0.3
15  32EXF  Min Increasing|0.3  Min Increasing|-0.3
16  32EXG  Min Increasing|0.3  Min Increasing|-0.3
17  32EXJ  Min Increasing|0.3  Min Increasing|-0.3
18    35P  Min Increasing|0.3  Min Increasing|-0.3
19    38F  Min Increasing|0.3  Min Increasing|-0.3
20    61C  Min Increasing|0.3  Min Increasing|-0.3
21    61D  Min Increasing|0.3  Min Increasing|-0.3
22  62EXA  Min Increasing|0.3  Min Increasing|-0.3
23  62EXB  Min Increasing|0.3  Min Increasing|-0.3
24  62EXC  Min Increasing|0.3  Min Increasing|-0.3
25  62EXE  Min Increasing|0.3  Min Increasing|-0.3
26  62EXG  Min Increasing|0.3  Min Increasing|-0.3
27  62EXH  Min Increasing|0.3  Min Increasing|-0.3
28  62EXI  Min Increasing|0.3  Min Increasing|-0.3
29    63A  Min Increasing|0.3  Min Increasing|-0.3
```

```
30    64P  Min Increasing|0.3  Min Increasing|-0.3
31    65F  Min Increasing|0.3  Min Increasing|-0.3

                                    USAFA Proportion  \
0   Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
1   Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
2   Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
3   Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
4   Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
5   Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
6   Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
7   Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
8   Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
9   Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
10  Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
11  Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
12                             Min Decreasing|0.3
13  Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
14                             Min Increasing|0.3
15                             Min Decreasing|0.3
16  Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
17  Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
18                             Min Decreasing|0.3
19                             Min Decreasing|0.3
20  Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
21  Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
22  Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
23  Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
24  Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
25  Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
26  Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
27  Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
28                             Min Increasing|0.3
29  Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
30                             Min Decreasing|0.3
31                             Min Decreasing|0.3

                                    Combined Quota         USAFA Quota  \
0   Quota_Direct|0.07, 0.8, 1.2, 0.8, 0.9, 0.88  Min Increasing|0.1
1                   Quota_Normal|0.2, 0.25, 0.2  Min Increasing|0.1
2     Quota_Direct|0.07, 1.5, 1, 0.2, 0.7, 0.99  Min Increasing|0.1
3    Quota_Direct|0.07, 0.5, 0.4, 0.2, 0.9, 0.8  Min Increasing|0.1
4   Quota_Direct|0.07, 0.5, 0.8, 0.2, 0.9, 0.75  Min Increasing|0.1
5      Quota_Direct|0.07, 0.4, 0.5, 1, 0.6, 0.8  Min Increasing|0.1
6   Quota_Direct|0.07, 0.8, 0.4, 0.3, 0.9, 0.95  Min Increasing|0.1
7    Quota_Direct|0.07, 0.5, 0.8, 0.2, 0.9, 0.9  Min Increasing|0.1
8    Quota_Direct|0.07, 0.5, 0.8, 0.2, 0.9, 0.9  Min Increasing|0.1
```

```
9                    Quota_Normal|0.2, 0.25, 0.2  Min Increasing|0.1
10   Quota_Direct|0.07, 0.5, 0.8, 0.2, 0.9, 0.9  Min Increasing|0.1
11   Quota_Direct|0.07, 0.5, 0.8, 0.2, 0.9, 0.9  Min Increasing|0.1
12  Quota_Direct|0.07, 0.8, 0.4, 0.3, 0.9, 0.95  Min Increasing|0.1
13                   Quota_Normal|0.2, 0.25, 0.2  Min Increasing|0.1
14                   Quota_Normal|0.2, 0.25, 0.2  Min Increasing|0.1
15                   Quota_Normal|0.2, 0.25, 0.2  Min Increasing|0.1
16                   Quota_Normal|0.2, 0.25, 0.2  Min Increasing|0.1
17                   Quota_Normal|0.2, 0.25, 0.2  Min Increasing|0.1
18   Quota_Direct|0.07, 0.5, 0.8, 0.2, 0.9, 0.9  Min Increasing|0.1
19   Quota_Direct|0.07, 0.5, 0.8, 0.2, 0.9, 0.9  Min Increasing|0.1
20                   Quota_Normal|0.2, 0.25, 0.2  Min Increasing|0.1
21                   Quota_Normal|0.2, 0.25, 0.2  Min Increasing|0.1
22  Quota_Direct|0.07, 0.8, 0.4, 0.3, 0.9, 0.95  Min Increasing|0.1
23  Quota_Direct|0.07, 0.8, 0.4, 0.3, 0.9, 0.95  Min Increasing|0.1
24  Quota_Direct|0.07, 0.8, 0.4, 0.3, 0.9, 0.95  Min Increasing|0.1
25  Quota_Direct|0.07, 0.8, 0.4, 0.3, 0.9, 0.95  Min Increasing|0.1
26                   Quota_Normal|0.2, 0.25, 0.2  Min Increasing|0.1
27                   Quota_Normal|0.2, 0.25, 0.2  Min Increasing|0.1
28                   Quota_Normal|0.2, 0.25, 0.2  Min Increasing|0.1
29   Quota_Direct|0.07, 0.8, 0.4, 0.5, 0.8, 0.5  Min Increasing|0.1
30                   Quota_Normal|0.2, 0.25, 0.2  Min Increasing|0.1
31   Quota_Direct|0.07, 0.5, 0.8, 0.2, 0.9, 0.9  Min Increasing|0.1

            ROTC Quota           Mandatory             Desired  \
0   Min Increasing|0.1  Min Increasing|0.1  Min Increasing|0.2
1   Min Increasing|0.1  Min Increasing|0.1  Min Increasing|0.2
2   Min Increasing|0.1  Min Increasing|0.1  Min Increasing|0.2
3   Min Increasing|0.1  Min Increasing|0.1  Min Decreasing|0.2
4   Min Increasing|0.1  Min Increasing|0.1  Min Increasing|0.2
5   Min Increasing|0.1  Min Increasing|0.1  Min Decreasing|0.2
6   Min Increasing|0.1  Min Increasing|0.1  Min Increasing|0.2
7   Min Increasing|0.1  Min Increasing|0.1  Min Decreasing|0.2
8   Min Increasing|0.1  Min Increasing|0.1  Min Increasing|0.2
9   Min Increasing|0.1  Min Increasing|0.1  Min Increasing|0.2
10  Min Increasing|0.1  Min Increasing|0.1  Min Increasing|0.2
11  Min Increasing|0.1  Min Increasing|0.1  Min Increasing|0.2
12  Min Increasing|0.1  Min Increasing|0.1  Min Increasing|0.2
13  Min Increasing|0.1  Min Increasing|0.1  Min Increasing|0.2
14  Min Increasing|0.1  Min Increasing|0.1  Min Increasing|0.2
15  Min Increasing|0.1  Min Increasing|0.1  Min Increasing|0.2
16  Min Increasing|0.1  Min Increasing|0.1  Min Decreasing|0.2
17  Min Increasing|0.1  Min Increasing|0.1  Min Increasing|0.2
18  Min Increasing|0.1  Min Increasing|0.1  Min Increasing|0.2
19  Min Increasing|0.1  Min Increasing|0.1  Min Increasing|0.2
20  Min Increasing|0.1  Min Increasing|0.1  Min Increasing|0.2
21  Min Increasing|0.1  Min Increasing|0.1  Min Increasing|0.2
```

```
22  Min Increasing|0.1  Min Increasing|0.1  Min Increasing|0.2
23  Min Increasing|0.1  Min Increasing|0.1  Min Increasing|0.2
24  Min Increasing|0.1  Min Increasing|0.1  Min Increasing|0.2
25  Min Increasing|0.1  Min Increasing|0.1  Min Increasing|0.2
26  Min Increasing|0.1  Min Increasing|0.1  Min Increasing|0.2
27  Min Increasing|0.1  Min Increasing|0.1  Min Increasing|0.2
28  Min Increasing|0.1  Min Increasing|0.1  Min Increasing|0.2
29  Min Increasing|0.1  Min Increasing|0.1  Min Increasing|0.2
30  Min Increasing|0.1  Min Increasing|0.1  Min Increasing|0.2
31  Min Increasing|0.1  Min Increasing|0.1  Min Increasing|0.2

             Permitted             Utility  \
0   Min Decreasing|0.3  Min Increasing|0.3
1   Min Decreasing|0.3  Min Increasing|0.3
2   Min Decreasing|0.3  Min Increasing|0.3
3   Min Decreasing|0.3  Min Increasing|0.3
4   Min Decreasing|0.3  Min Increasing|0.3
5   Min Decreasing|0.3  Min Increasing|0.3
6   Min Decreasing|0.3  Min Increasing|0.3
7   Min Decreasing|0.3  Min Increasing|0.3
8   Min Decreasing|0.3  Min Increasing|0.3
9   Min Decreasing|0.3  Min Increasing|0.3
10  Min Decreasing|0.3  Min Increasing|0.3
11  Min Decreasing|0.3  Min Increasing|0.3
12  Min Decreasing|0.3  Min Increasing|0.3
13  Min Decreasing|0.3  Min Increasing|0.3
14  Min Decreasing|0.3  Min Increasing|0.3
15  Min Decreasing|0.3  Min Increasing|0.3
16  Min Decreasing|0.3  Min Increasing|0.3
17  Min Decreasing|0.3  Min Increasing|0.3
18  Min Decreasing|0.3  Min Increasing|0.3
19  Min Decreasing|0.3  Min Increasing|0.3
20  Min Decreasing|0.3  Min Increasing|0.3
21  Min Decreasing|0.3  Min Increasing|0.3
22  Min Decreasing|0.3  Min Increasing|0.3
23  Min Decreasing|0.3  Min Increasing|0.3
24  Min Decreasing|0.3  Min Increasing|0.3
25  Min Decreasing|0.3  Min Increasing|0.3
26  Min Decreasing|0.3  Min Increasing|0.3
27  Min Decreasing|0.3  Min Increasing|0.3
28  Min Decreasing|0.3  Min Increasing|0.3
29  Min Decreasing|0.3  Min Increasing|0.3
30  Min Decreasing|0.3  Min Increasing|0.3
31  Min Decreasing|0.3  Min Increasing|0.3

                                           Male  \
0   Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
```

```
1   Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
2   Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
3   Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
4   Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
5   Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
6   Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
7   Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
8   Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
9   Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
10  Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
11  Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
12  Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
13  Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
14  Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
15  Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
16  Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
17  Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
18  Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
19  Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
20  Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
21  Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
22  Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
23  Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
24  Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
25  Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
26  Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
27  Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
28  Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
29  Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
30  Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
31  Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6

                                         Minority
0   Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
1   Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
2   Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
3   Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
4   Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
5   Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
6   Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
7   Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
8   Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
9   Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
10  Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
11  Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
12  Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
13  Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
```

```
14   Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
15   Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
16   Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
17   Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
18   Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
19   Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
20   Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
21   Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
22   Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
23   Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
24   Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
25   Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
26   Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
27   Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
28   Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
29   Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
30   Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
31   Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
```

Here we have the value functions for each of the AFSC objectives. These definitely require some explaining. I've created my own terminology so that they can be generalized and constructed into actual value functions for each of the objectives. I have an excel file that outlines how these functions are created and what they look like (Value_Function_Builds.xlsx), but I will also detail them here.

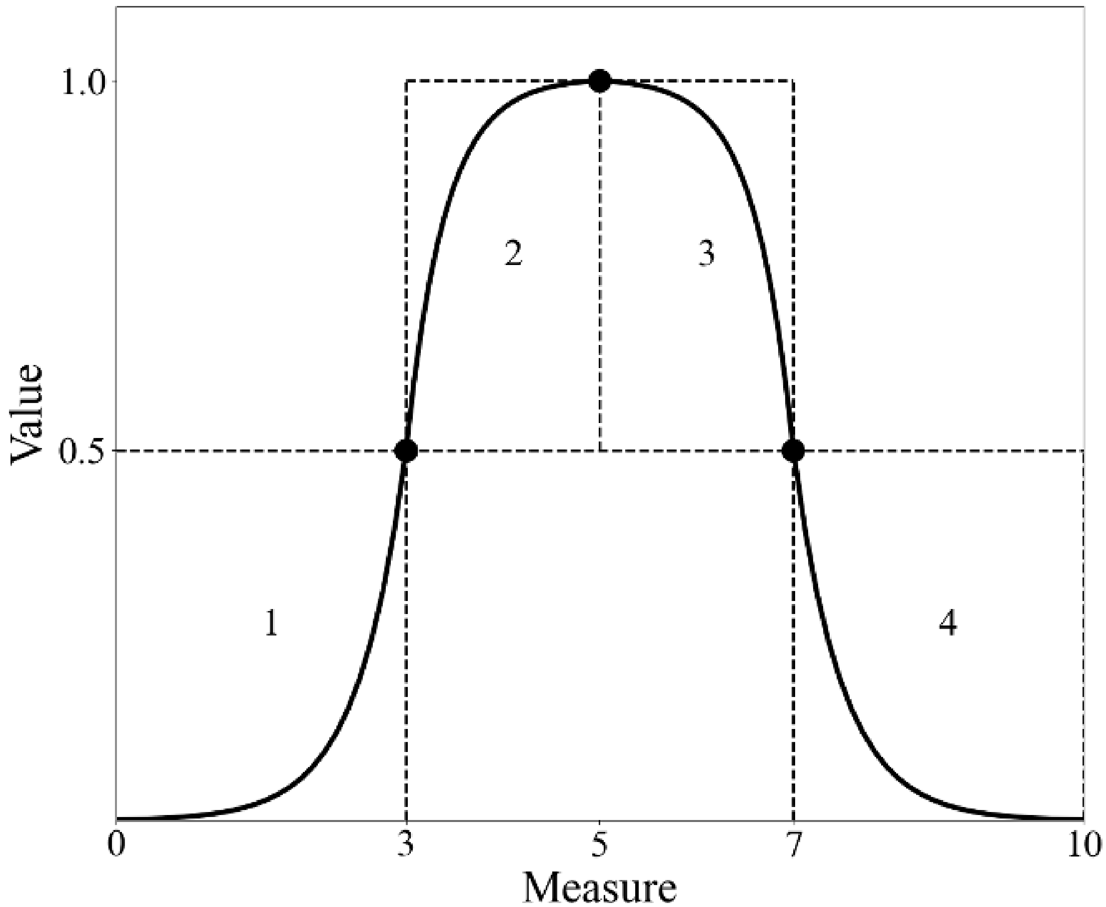
[40]:
```
# I need to import this script
import afccp.core.handling.value_parameter_handling
```

Before you read this next section on the value functions, please look at my slides in "VFT_Model_Slides.pptx" (located in the docs folder). Navigate to the "Creating Value Functions" section (starts on slide 130), and just click through them. This is how I construct the value functions, and this should help your understanding of the different piece-wise "segments" used.

The purpose of the "vf_string" (Value Function string) is to construct the "segment_dict" (Segment Dictionary) which provides the coordinates for the main piece-wise value function segment breakpoints. As illustrated below, there are four "segments" of exponential functions that are pieced together using "breakpoints". There are therefore 5 breakpoints. For this example, they are at the coordinates (0, 0), (3, 0.5), (5, 1), (7, 0.5), and (10, 0). This would compose the "segment_dict"

[41]:
```
Image(filename='pic8.png')
```

[41]:



Let's illustrate the "Balance" value function. It takes several inputs pertaining to the "margins" and the ρ parameters. Here is what it looks like:

vf_string = "Balance|left_base_margin, right_base_margin, ρ_1, ρ_2, ρ_3, ρ_4, margin_y"

Honestly, you really don't need to worry about what these all mean. The only thing you should focus on is the ρ ("rho") parameters. These control how steep each of the exponential segments are. Let's see an example. We'll first generate the "segment_dict" based on the "vf_string"

[42]:
```
vf_string = "Balance|0.2, 0.2, 0.1, 0.08, 0.08, 0.1, 0.5"
target = 0.5
actual = 0.5
segment_dict = afccp.core.handling.value_parameter_handling.
 ↪create_segment_dict_from_string(
    vf_string, target=target, actual=actual)
for segment in segment_dict:
    print(str(segment) + ":", segment_dict[segment])
```

```
1: {'x1': 0, 'y1': 0, 'x2': 0.3, 'y2': 0.5, 'rho': -0.1}
2: {'x1': 0.3, 'y1': 0.5, 'x2': 0.5, 'y2': 1, 'rho': 0.08}
```

```
3: {'x1': 0.5, 'y1': 1, 'x2': 0.7, 'y2': 0.5, 'rho': 0.08}
4: {'x1': 0.7, 'y1': 0.5, 'x2': 1, 'y2': 0, 'rho': -0.1}
```

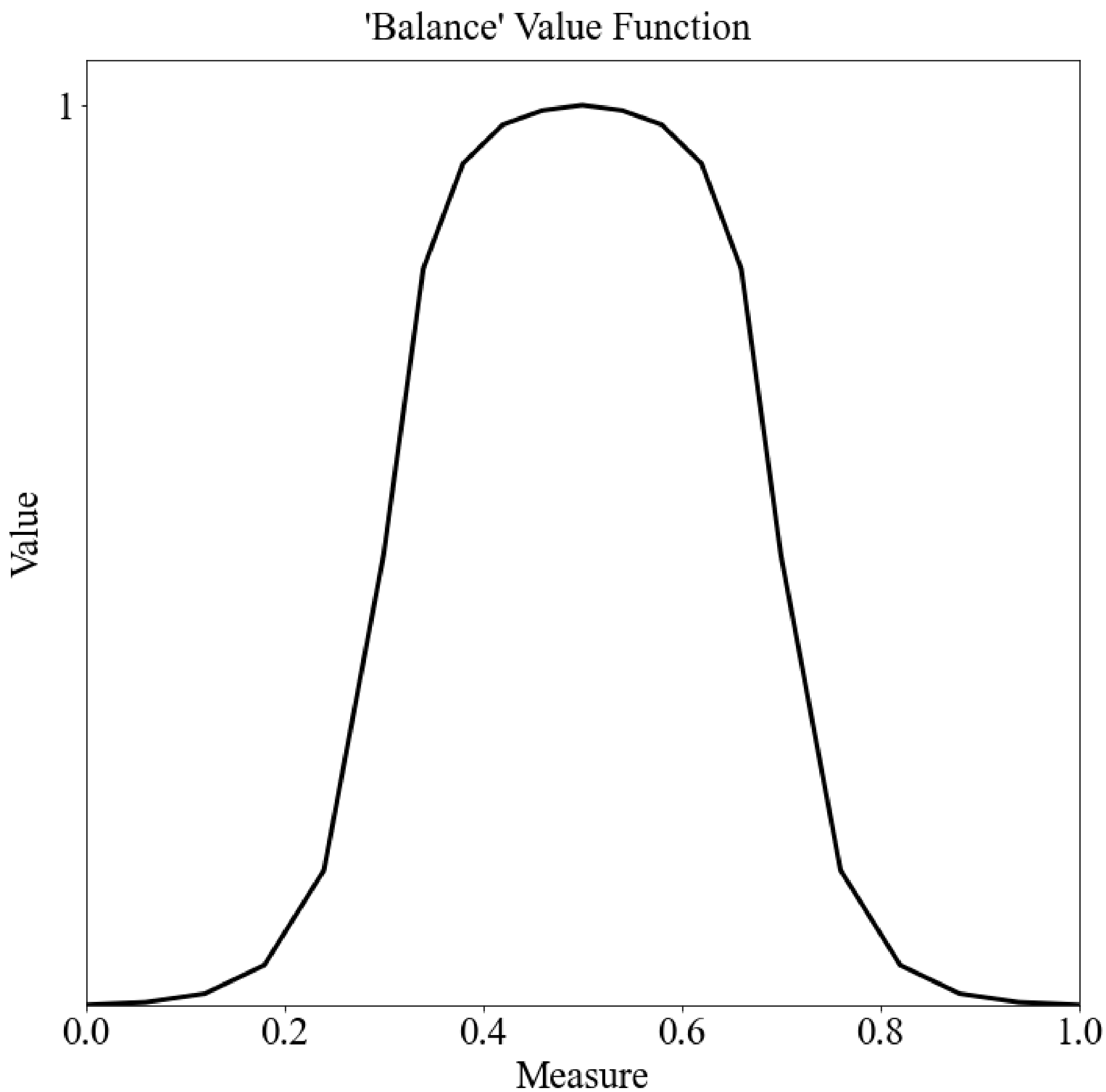
Now we have our segment dictionary! We know what the coordinates for the main breakpoints are, so we can now generate all of the breakpoints to the full function. Let's calculate the x and y coordinates of our function's breakpoints.

[43]:
```
x, y = afccp.core.handling.value_parameter_handling.
 ↪value_function_builder(segment_dict=segment_dict,
                                                                                        ␣
 ↪num_breakpoints=20)
print("x:", x, "\n\n", "y:", y)
```

```
x: [0.   0.06 0.12 0.18 0.24 0.3  0.34 0.38 0.42 0.46 0.5  0.54 0.58 0.62
 0.66 0.7  0.76 0.82 0.88 0.94 1.  ]

 y: [0.      0.00288 0.01245 0.04423 0.14973 0.5     0.8182  0.93527 0.97833
 0.99417 1.      0.99417 0.97833 0.93527 0.8182  0.5     0.14973 0.04423
 0.01245 0.00288 0.     ]
```

Now we plot our value function!

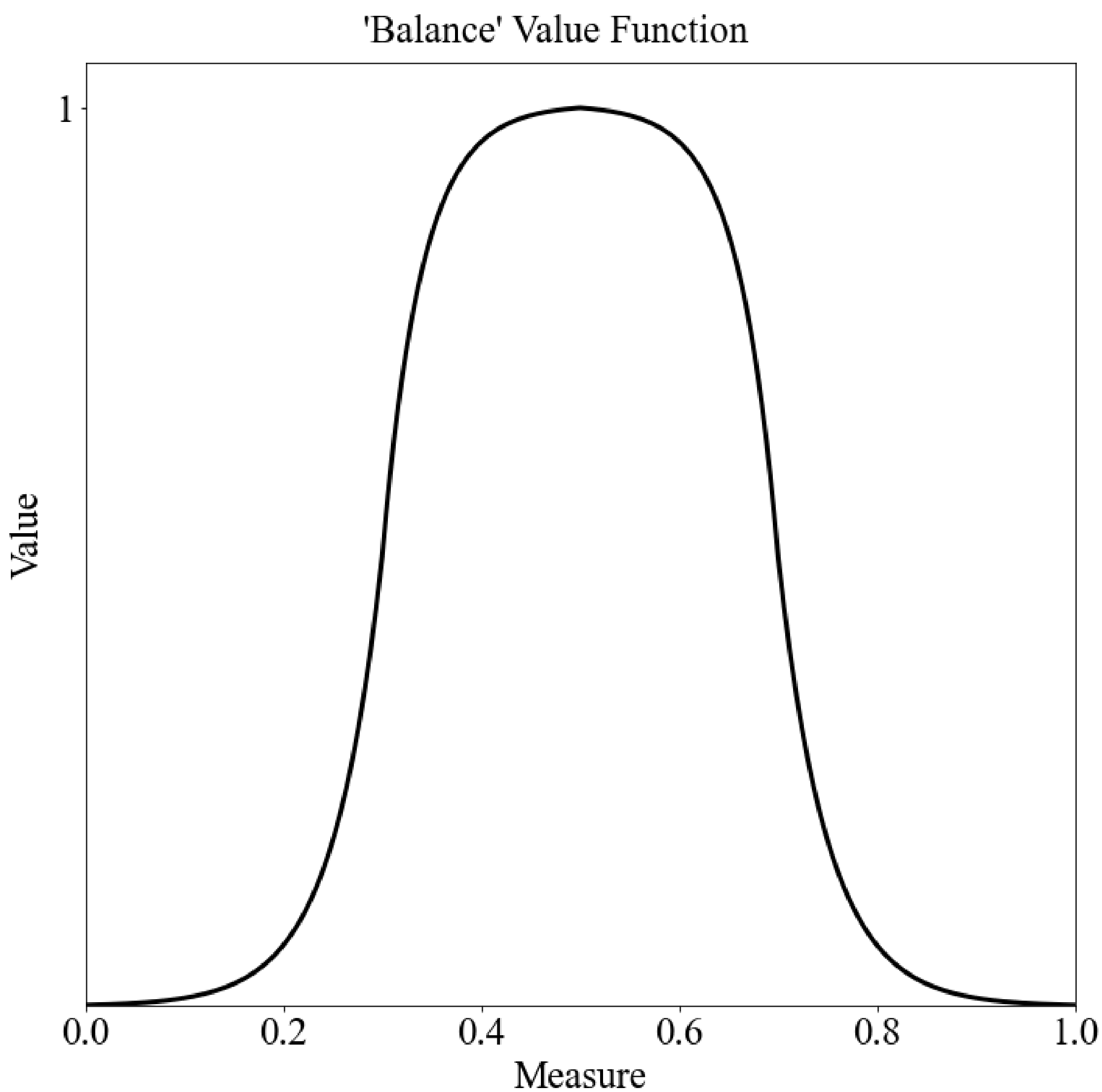
[44]:
```
import afccp.core.visualizations.instance_graphs
chart = afccp.core.visualizations.instance_graphs.value_function_graph(x, y,␣
 ↪title="'Balance' Value Function")
```



And there we have it. This is the value function we've constructed from that initial "vf_string". Play around with the different parameters and see what happens here!

[45]:
```
# Change this
vf_string = "Balance|0.2, 0.2, 0.1, 0.08, 0.08, 0.1, 0.5"
target = 0.5  # This is what we're after
actual = 0.5  # This is essentially what we could realistically expect (based␣
 ↪on set of eligible cadets)
num_breakpoints = 200  # How many breakpoints to use
# (the more breakpoints used, the more the function appears non-linear)

# Don't change this
```

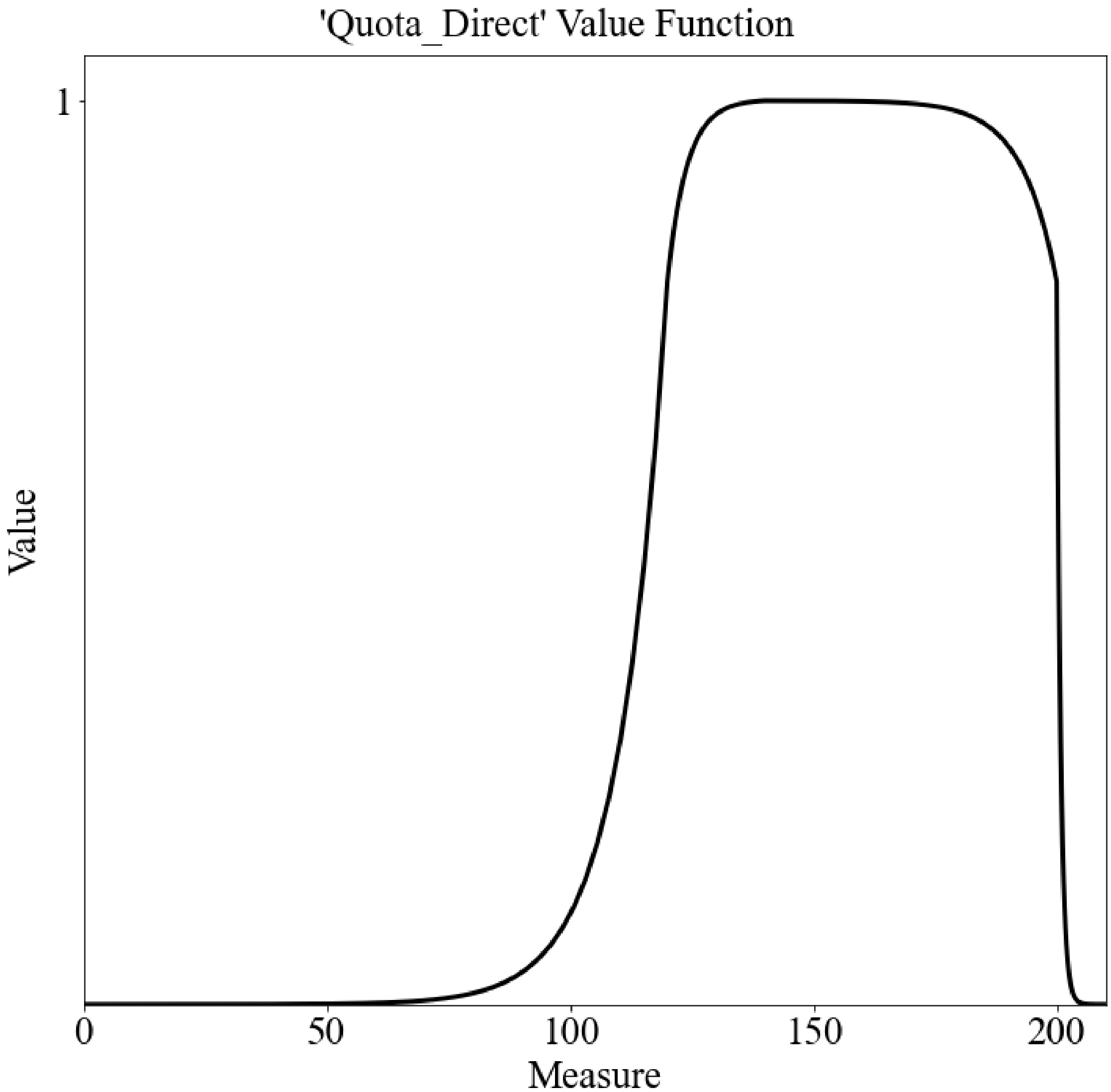
```
segment_dict = afccp.core.handling.value_parameter_handling.
 ↪create_segment_dict_from_string(
    vf_string, target=target, actual=actual)
x, y = afccp.core.handling.value_parameter_handling.
 ↪value_function_builder(segment_dict=segment_dict,
 
 ↪num_breakpoints=num_breakpoints)
chart = afccp.core.visualizations.instance_graphs.value_function_graph(x, y,
 ↪title= "'Balance' Value Function")
```



That is the “Balance” value function type. This is intended for the objectives that seek to “balance” certain characteristics of the cadets (USAFA/Male/Minority proportions and sometimes Merit as well). I did end up changing the Merit value function to be a “Min Increasing” because I decided against penalizing the objective for exceeding 0.5. At this point, I will note that these value functions don't necessarily have to have 4 segments. I do have value function types that use 3, 2, or even 1 segment. Let's discuss the quota value functions.

"Quota_Direct" is intended for AFSCs that have a range on the number of cadets that are to be assigned, but also know around where they'd like to fall within that range. There are 6 parameters, the ρ (rho) parameters for each of the four segments, and the y values for the two breakpoints on either side of the "peak". The vf_string is then: "Quota_Direct|ρ_1, ρ_2, ρ_3, ρ_4, y_1, y_2". The additional AFSC specific parameters are the upper/lower bounds on the number of cadets as well as the actual target number of cadets within that range. Here is an example:

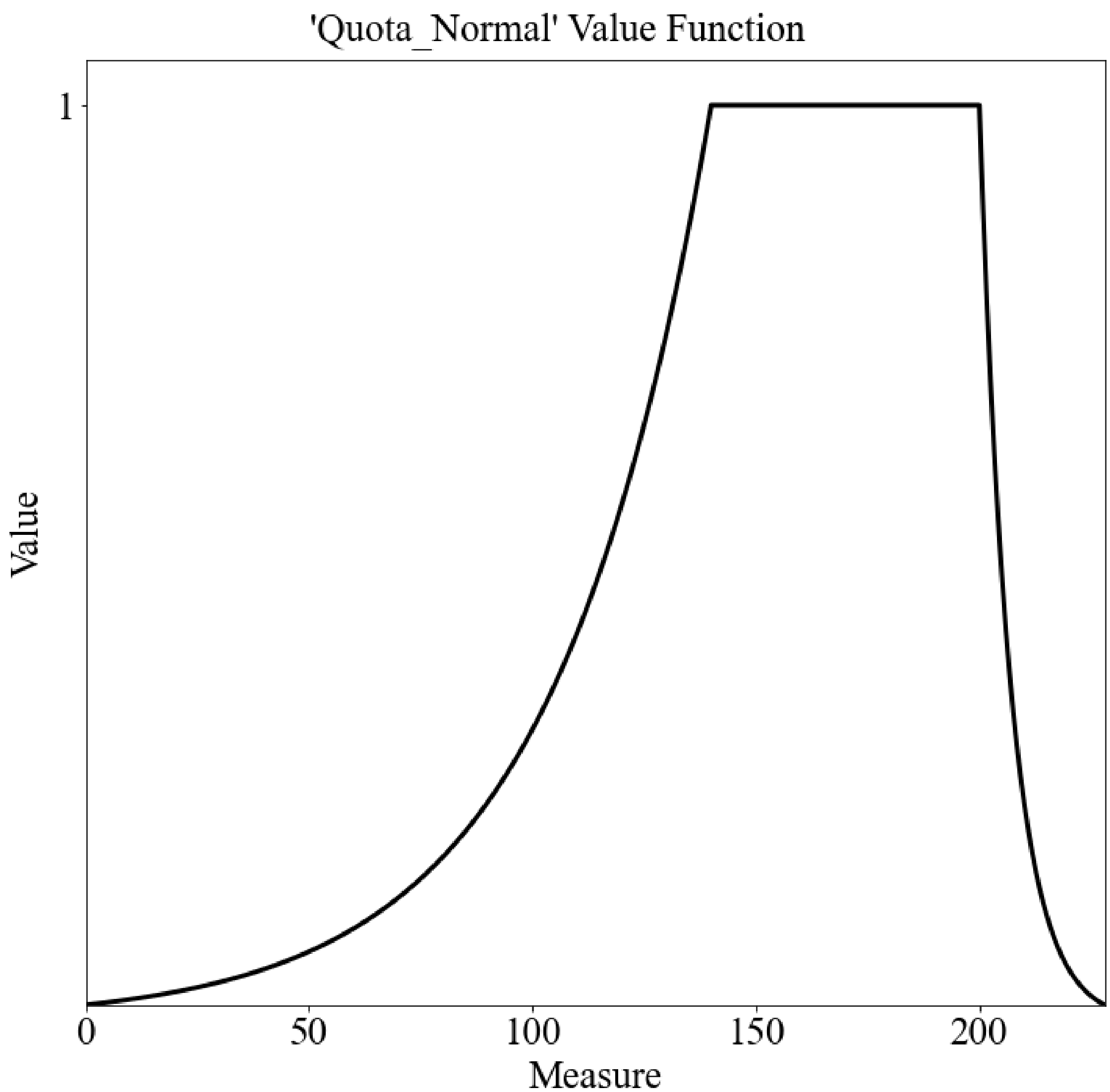
[46]:
```
vf_string = "Quota_Direct|0.1, 1, 0.6, 0.1, 0.8, 0.8"
minimum = 120  # Lower Bound
maximum = 200  # Upper Bound
target = 140  # Desired number of cadets within the range
num_breakpoints = 200  # How many breakpoints to use

# Don't change this
segment_dict = afccp.core.handling.value_parameter_handling.
 ↪create_segment_dict_from_string(
    vf_string, target=target, minimum=minimum, maximum=maximum)
x, y = afccp.core.handling.value_parameter_handling.
 ↪value_function_builder(segment_dict=segment_dict,
                                                                                ␣
 ↪num_breakpoints=num_breakpoints)
chart = afccp.core.visualizations.instance_graphs.value_function_graph(x, y,␣
 ↪title="'Quota_Direct' Value Function")
```



Here you can see that although the range of 120 to 200 is specified, there is a direction of preference within that range (the AFSC wants around 140 cadets, but is fairly accepting of values around that range). I will note that the target, minimum, and maximum parameters are taken from the AFSCs_Fixed data!

Another value function we can choose for the quota objective is the "Quota_Normal" function type. This is intended for AFSCs that either don't care about the number of cadets (as long as they fall within a certain range) or didn't specify. For example, the PGL says 120 and after speaking with them we determine the upper bound is 200 and they say they have no preference between 120 and 200 and everything in between. There are 2 segments for this function, connected by a horizontal line at y = 1 for the range on the cadets. The function parameters are ρ_1, ρ_2, and "domain_max" which is the max number of cadets that could have a nonzero value (arbitrary scalar just to get a curve on the right side of the function). Here is the vf_string: "Quota_Normal|d_max, ρ_1, ρ_2". Here is an example:

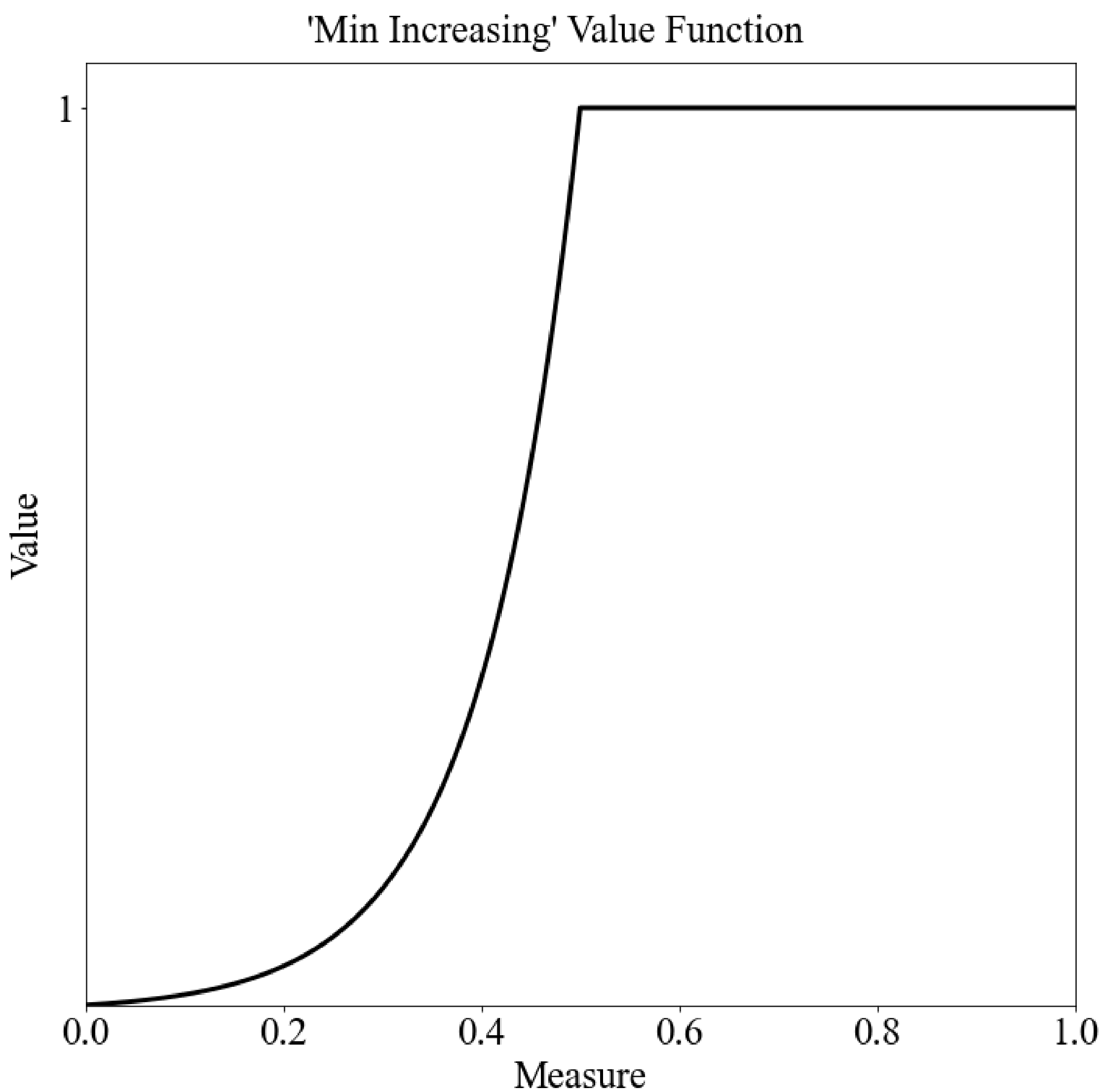
```
[47]: vf_string = "Quota_Normal|0.2, 0.25, 0.05"
minimum = 120  # Lower Bound
maximum = 200  # Upper Bound
target = 140  # (Doesn't matter here)
num_breakpoints = 200  # How many breakpoints to use

# Don't change this
segment_dict = afccp.core.handling.value_parameter_handling.
 ↪create_segment_dict_from_string(
    vf_string, target=target, minimum=minimum, maximum=maximum)
x, y = afccp.core.handling.value_parameter_handling.
 ↪value_function_builder(segment_dict=segment_dict,
                                                                                    ␣
 ↪num_breakpoints=num_breakpoints)
chart = afccp.core.visualizations.instance_graphs.value_function_graph(x, y,␣
 ↪title="'Quota_Normal' Value Function")
```



The last two kinds of value functions I'll discuss are the "Min Increasing" and "Min Decreasing" types. They are very simple and only have one segment which is a simple exponentional curve to get to the target measure (in the x space). The only parameter is ρ. The vf_string then looks like: "Min Increasing|ρ" or "Min Decreasing|ρ". They are called "Min" functions because it's essentially the same thing as taking the minimum value between some exponential curve and 1. Here are some examples:

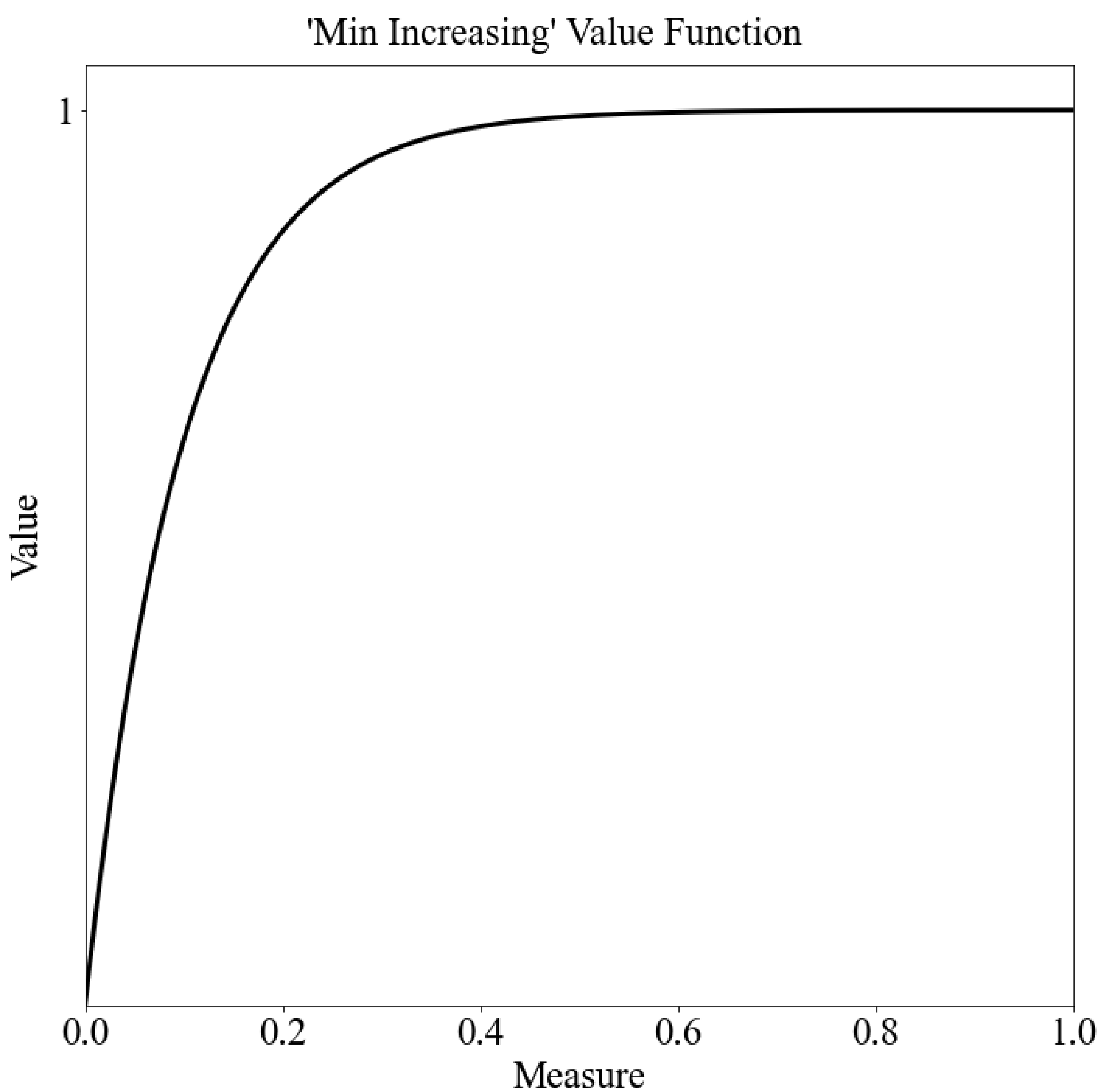
[48]:
```
vf_string = "Min Increasing|0.1"
target = 0.5
num_breakpoints = 200  # How many breakpoints to use

# Don't change this
```

```
segment_dict = afccp.core.handling.value_parameter_handling.
 ↪create_segment_dict_from_string(
    vf_string, target=target, minimum=minimum, maximum=maximum)
x, y = afccp.core.handling.value_parameter_handling.
 ↪value_function_builder(segment_dict=segment_dict,
 ↪num_breakpoints=num_breakpoints)
chart = afccp.core.visualizations.instance_graphs.value_function_graph(x, y,
 ↪title="'Min Increasing' Value Function")
```



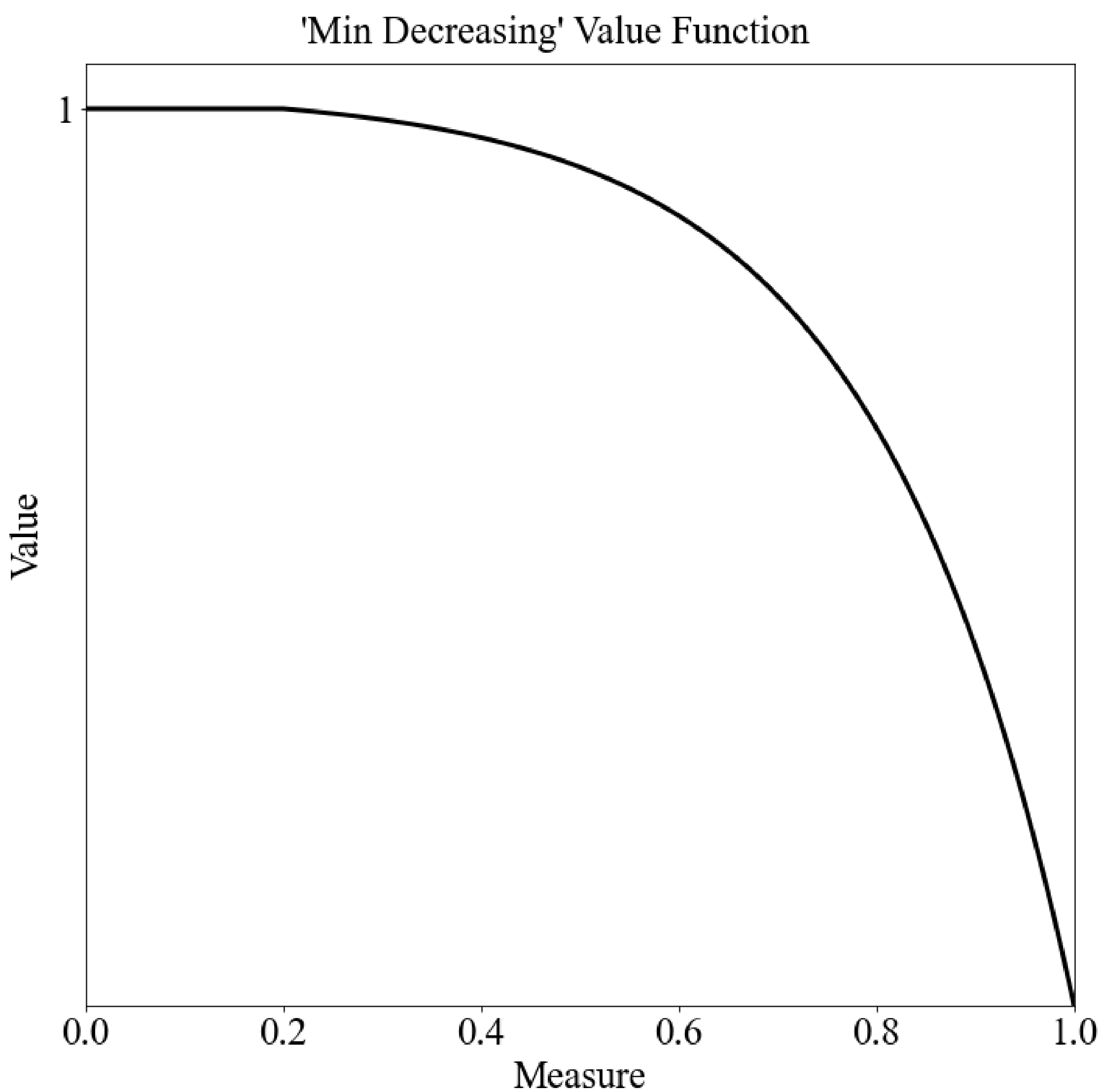
[49]:
```
vf_string = "Min Increasing|-0.1"
target = 1
```

```
num_breakpoints = 200  # How many breakpoints to use

# Don't change this
segment_dict = afccp.core.handling.value_parameter_handling.
↪create_segment_dict_from_string(
    vf_string, target=target, minimum=minimum, maximum=maximum)
x, y = afccp.core.handling.value_parameter_handling.
↪value_function_builder(segment_dict=segment_dict,
                                                                          ␣
↪num_breakpoints=num_breakpoints)
chart = afccp.core.visualizations.instance_graphs.value_function_graph(x, y,␣
↪title="'Min Increasing' Value Function")
```

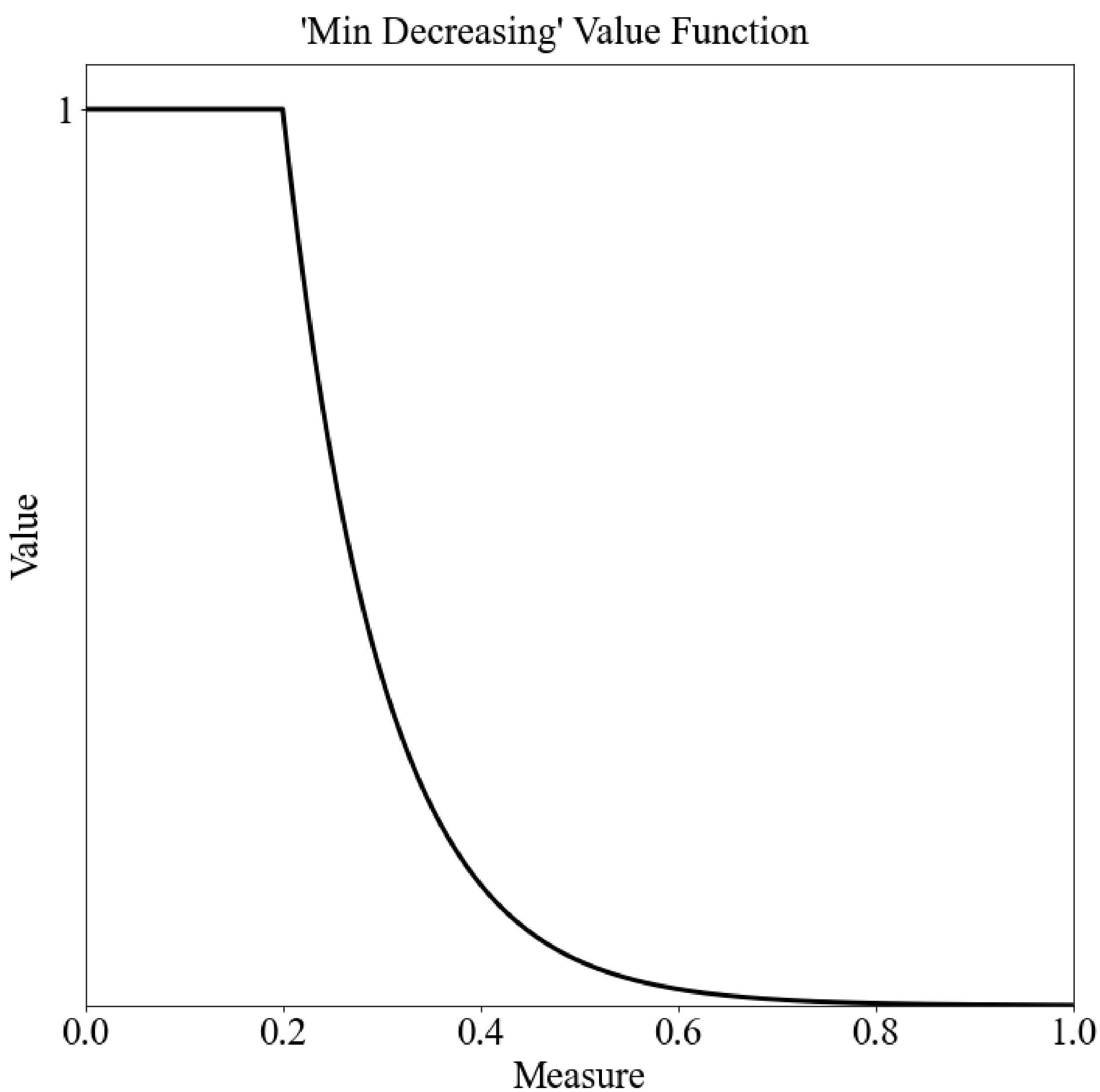


[50]:

```
vf_string = "Min Decreasing|-0.2"
target = 0.2
num_breakpoints = 200  # How many breakpoints to use

# Don't change this
segment_dict = afccp.core.handling.value_parameter_handling.
 ↪create_segment_dict_from_string(
    vf_string, target=target, minimum=minimum, maximum=maximum)
x, y = afccp.core.handling.value_parameter_handling.
 ↪value_function_builder(segment_dict=segment_dict,
                                                                                    ␣
 ↪num_breakpoints=num_breakpoints)
chart = afccp.core.visualizations.instance_graphs.value_function_graph(x, y,␣
 ↪title="'Min Decreasing' Value Function")
```



[51]:
```python
vf_string = "Min Decreasing|0.1"
target = 0.2
num_breakpoints = 200  # How many breakpoints to use

# Don't change this
segment_dict = afccp.core.handling.value_parameter_handling.
 ↪create_segment_dict_from_string(
    vf_string, target=target, minimum=minimum, maximum=maximum)
x, y = afccp.core.handling.value_parameter_handling.
 ↪value_function_builder(segment_dict=segment_dict,
                                                                                          ␣
 ↪num_breakpoints=num_breakpoints)
chart = afccp.core.visualizations.instance_graphs.value_function_graph(x, y,␣
 ↪title="'Min Decreasing' Value Function")
```



And there you have it! This is how I code up and construct my many value functions for each of the objectives for each of the AFSCs. Please reach out if you have any questions as I know this is a confusing section.

Structure Demo Before I demo the value parameters, I need to talk about cadet/AFSC preference lists. This is the new OLEA initiative, and it's why you may have noticed the "Norm Score" objective in the default value parameters above. We need to create preference lists in order for the "Norm Score" objective to be included in this instance when we import default value parameters. Initially, I assume you only have the Cadets Fixed and AFSCs Fixed excel sheets. You will likely also have preferences for the cadets and the AFSCs. Regardless of if you do or don't, I want you to generate four new excel sheets using the code below. If you have the "real" preference lists, you can replace the ones you generate with those afterwards.

[52]:
```
# Create cadet preference lists using the cadet utility matrix
instance.convert_utilities_to_preferences()

# Create AFSC preference lists using merit, AFOCD, and utility characteristics
instance.generate_fake_afsc_preferences()

# Create the AFSC utility matrix by normalizing the fake AFSC preference lists
instance.convert_afsc_preferences_to_percentiles()
```

Once you do that, if you were to export the instance to excel now you would have four new sheets: "Cadets Utility", "AFSCs Utility", "Cadets Preferences", "AFSCs Preferences". If you already have preference lists for cadets/AFSCs generated by OLEA, simply replace those two excel sheets with those. You could then create a new AFSCs Utility matrix by running "instance.convert_afsc_preferences_to_percentiles()". You're now ready to import default value parameters!

Just as "parameters" is a dictionary of many different fixed cadet/AFSC parameters, "value_parameters" is a dictionary of all of the different weight and value parameters. Let's generate our value parameters for the "2023b" instance using the generalized defaults from the excel sheet.

[53]:
```
instance.import_default_value_parameters()
pass  # Prevents printing out the entire dictionary
```

```
Importing default value parameters…
Imported.
```

The above code looks for "Value_Parameters_Defaults_2023b.xlsx" in the "support" folder. If it doesn't find it, it'll grab the generic "Value_Parameters_Defaults.xlsx" file since that one has all of the AFSCs in there. We now have our value parameters loaded in for this class year! Let's see what they look like.

[54]:
```
instance.value_parameters.keys()
```

[54]:

```
dict_keys(['cadets_overall_weight', 'afscs_overall_weight',
'cadet_weight_function', 'afsc_weight_function', 'cadets_overall_value_min',
'afscs_overall_value_min', 'afsc_value_min', 'cadet_value_min',
'objective_weight', 'afsc_weight', 'M', 'objective_target', 'objectives', 'O',
'objective_value_min', 'constraint_type', 'USAFA-Constrained AFSCs', 'Similarity
Constraint', 'Cadets Top 3 Constraint', 'num_breakpoints', 'cadet_weight', 'a',
'f^hat', 'value_functions', 'K', 'K^A', 'K^C', 'J^USAFA', 'K^D', 'J^A', 'I^C',
'J^C', 'r', 'L', 'vp_weight'])
```

We have many different kinds of parameters loaded in this dictionary now. Just by looking at the contents of the dictionary, you can probably guess what most of them are.

[55]:

```python
# Shorthand (just like p = instance.parameters)
vp = instance.value_parameters

# Set of potential objectives
print(vp['objectives'])
```

```
['Norm Score' 'Merit' 'USAFA Proportion' 'Combined Quota' 'USAFA Quota'
 'ROTC Quota' 'Mandatory' 'Desired' 'Permitted' 'Utility' 'Male'
 'Minority']
```

[56]:

```python
# Set of objectives that AFSC at index 21 cares about (61D)
j = 21
afsc = p["afsc_vector"][j]
indices = vp['K^A'][j]
print("AFSC:", afsc)
print('objective indices:', indices)
print('objectives:', vp["objectives"][indices])
```

```
AFSC: 61D
objective indices: [0 1 3 6 9]
objectives: ['Norm Score' 'Merit' 'Combined Quota' 'Mandatory' 'Utility']
```

[57]:

```python
# Set of objectives that AFSC 61D is constraining
indices = vp['K^C'][j]
print('objective indices:', indices)
print('objectives:', vp["objectives"][indices])
```

```
objective indices: [1 3 6]
objectives: ['Merit' 'Combined Quota' 'Mandatory']
```

[58]:

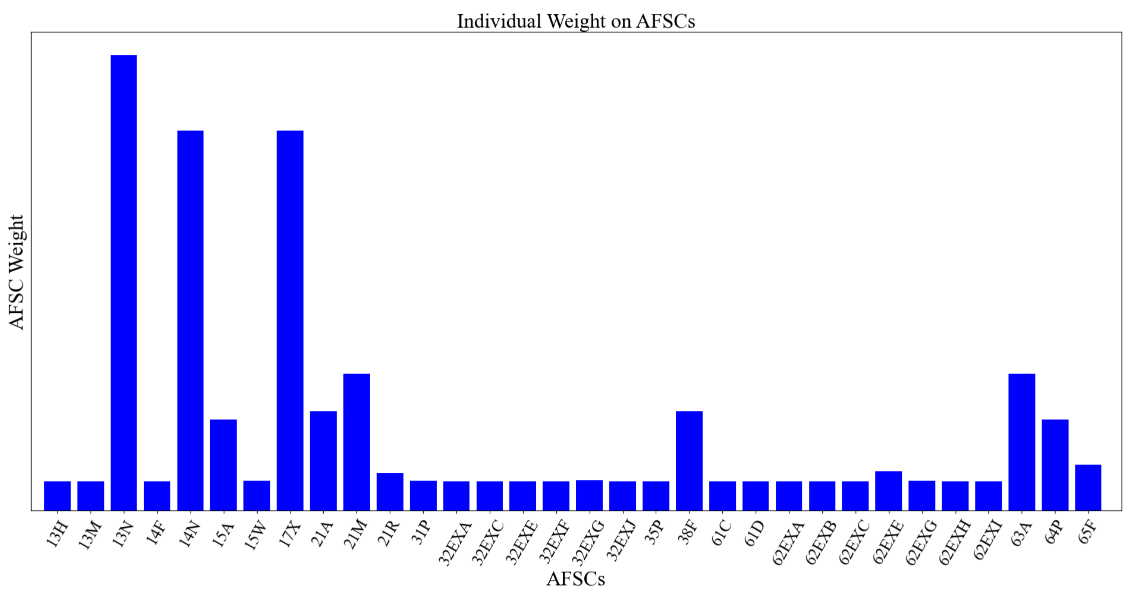
```python
# AFSC individual weight
print('AFSC weight function:', vp['afsc_weight_function'])  # We hand-picked␣
 ↪the weights
print('AFSC "local" weights:', vp['afsc_weight'])  # sum to 1
```

```
AFSC weight function: Custom
AFSC "local" weights: [0.01134456 0.0113799  0.17670654 0.01134456 0.14724956
0.03534131
```

```
0.01141524 0.14723189 0.03855737 0.05301196 0.01447227 0.01145058
0.01132689 0.01134456 0.01132689 0.01132689 0.01178633 0.01132689
0.01136223 0.03855737 0.01132689 0.01134456 0.01136223 0.01134456
0.0113799  0.01516142 0.01152127 0.01134456 0.01132689 0.05301196
0.03534131 0.01767065]
```

[59]:
```python
# Let's visualize the weights on the AFSCs
chart = instance.display_weight_function({"cadets_graph":False, "skip_afscs":
 ↪False, "afsc_rotation":60})
```

```
Creating AFSC weight chart…
```

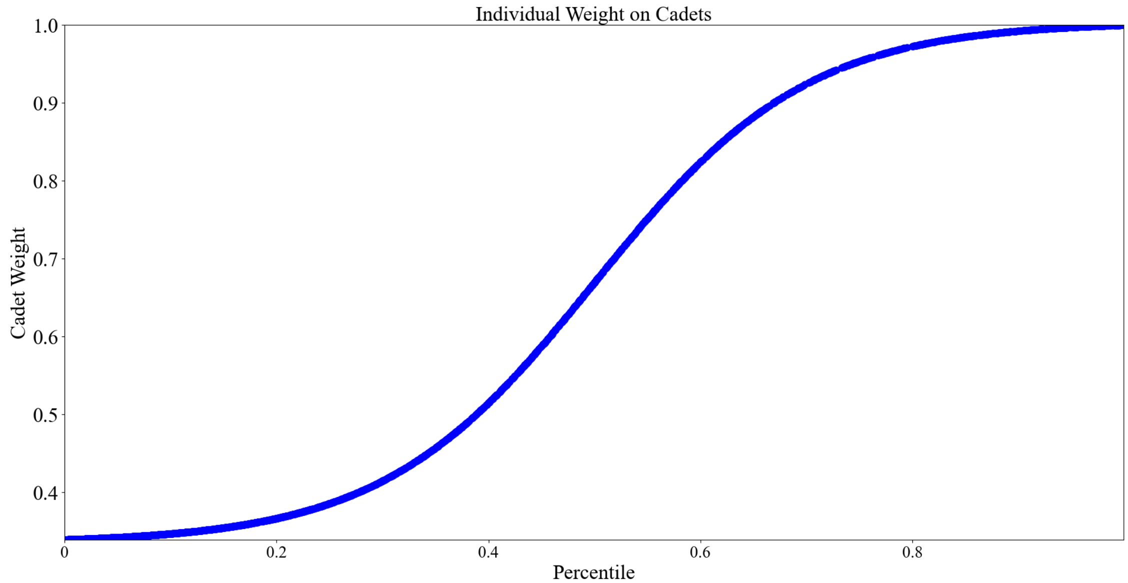


[60]:
```python
# Cadet individual weight
print('Cadet weight function:', vp['cadet_weight_function'])  # Linear function␣
 ↪of merit here!
print('Cadet local weights:', vp['cadet_weight'])  # sum to 1
```

```
Cadet weight function: Curve_1
Cadet local weights: [0.00038665 0.00105211 0.0004545  … 0.00080125 0.00071034
0.00071034]
```

[61]:
```python
# Let's visualize the weights on the cadets
chart = instance.display_weight_function({"cadets_graph":True})
```

```
Creating cadet weight chart…
```




```
[62]: # The "overall weights" are pretty straightforward (and mutable as well)
print('Overall Weight on Cadets:', vp['cadets_overall_weight'])
print('Overall Weight on AFSCs:', vp['afscs_overall_weight'])
```

```
Overall Weight on Cadets: 0.7
Overall Weight on AFSCs: 0.3
```

```
[63]: # AFSC objective weight (Just printing the shape of it- it'll be MxO where M is␣
↪number of AFSCs and
# O is number of objectives!)
print(np.shape(vp['objective_weight']))
```

```
(32, 12)
```

```
[64]: # Let's see the objectives for 61D again
j = 21
afsc = p["afsc_vector"][j]
indices = vp['K^A'][j]
print('C22 objectives:', vp["objectives"][indices])
```

```
C22 objectives: ['Norm Score' 'Merit' 'Combined Quota' 'Mandatory' 'Utility']
```

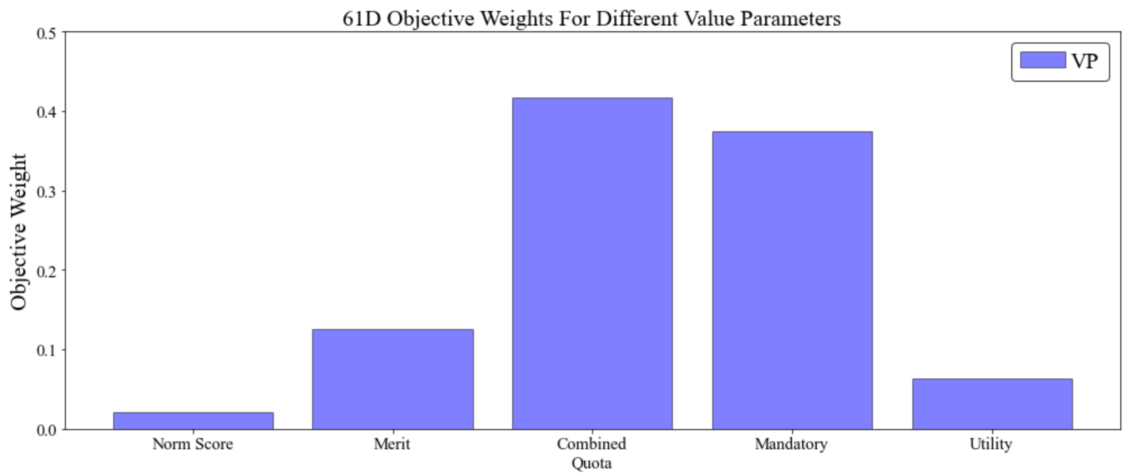
```
[65]: # The weights on this AFSC's objectives:
print(vp['objective_weight'][j, :])
```

```
[0.02083333 0.125      0.         0.41666667 0.         0.
 0.375      0.         0.         0.0625     0.         0.        ]
```

[66]:
```
# Weights are loaded into a large matrix, but not all objective weights are␣
↪needed (many are zero)
print(vp['objective_weight'][j, indices])  # these are the only ones considered!
```

```
[0.02083333 0.125      0.41666667 0.375      0.0625    ]
```

[67]:
```
# Objective weights for AFSC "C22"
chart = instance.display_afsc_objective_weights_chart(afsc="61D")
```

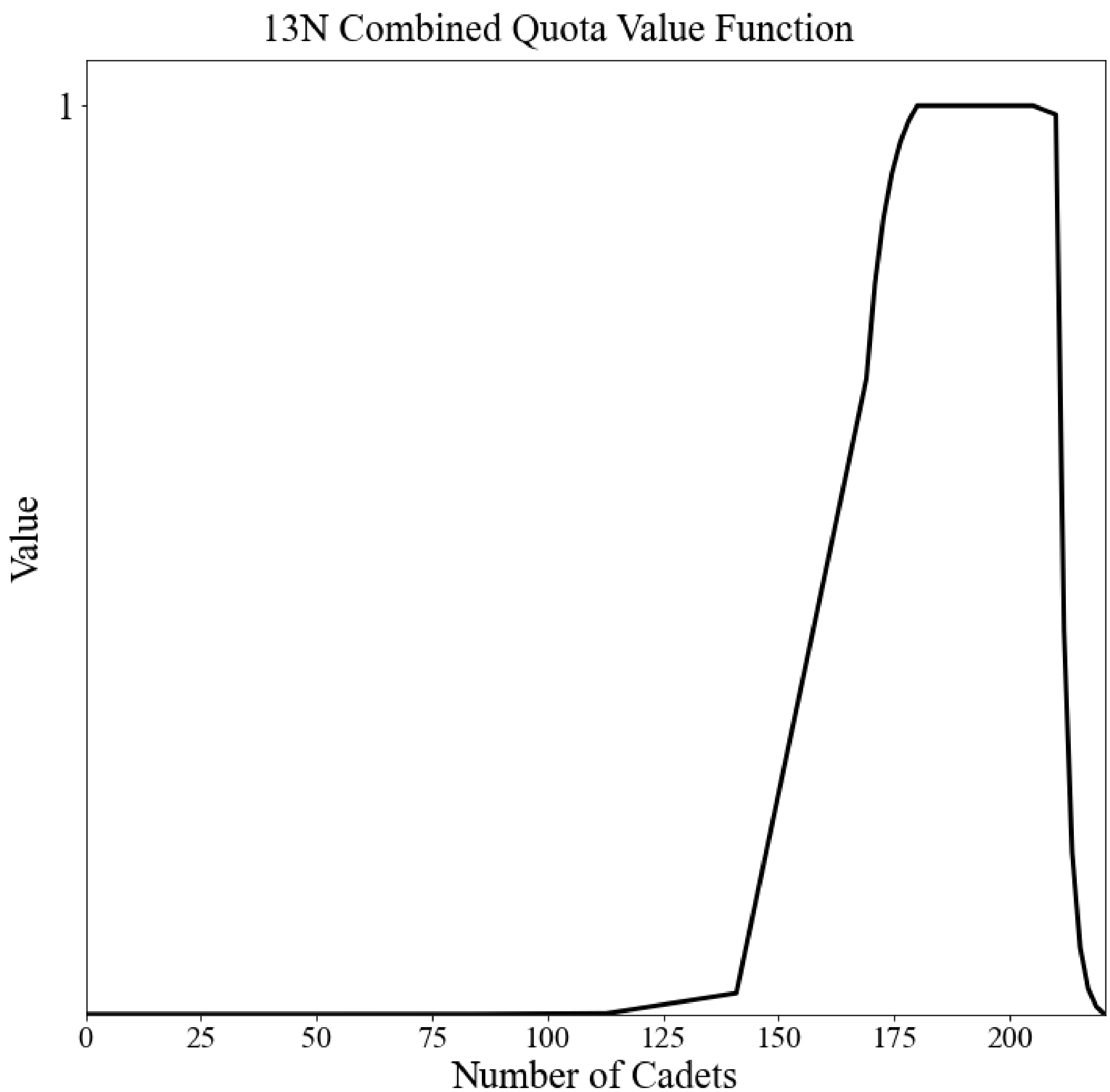


The "For Different Value Parameters" part is in reference to the fact that this function is usually meant to compare different sets of value parameters with each other. Just like before with AFSC weight, we could also change objective weights directly for any AFSC very easily.

We already discussed value functions fairly in-depth previously, but now I'll just demonstrate how to work with them through the code.

[68]:
```
# Pick an AFSC and objective
afsc, objective = "13N", "Combined Quota"

# Get the indices
j, k = np.where(p["afsc_vector"] == afsc)[0][0], np.where(vp["objectives"] ==␣
↪objective)[0][0]

# Now let's plot this AFSC objective's value function
chart = instance.show_value_function({"afsc":afsc, "objective":objective})
```

```
Creating value function chart for objective Combined Quota for AFSC 13N…
```



The parameters used to create this value function are:

```
[69]: print("Number of Breakpoints:", vp["r"][j][k])
print("Set of Breakpoints\n", vp["L"][j][k])
print("Breakpoint x-coordinates\n", vp["a"][j][k])
print("Breakpoint y-coordinates\n", vp["f^hat"][j][k])
```

```
Number of Breakpoints: 24
Set of Breakpoints
 [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23]
Breakpoint x-coordinates
 [  0.         56.33333  84.5      112.66667 140.83333 169.       170.83333
 172.66667 174.5      176.33333 178.16667 180.       185.       190.
 195.       200.       205.       210.       211.75     213.5      215.25
```

```
 217.        218.75     220.5     ]
Breakpoint y-coordinates
 [0.0000e+00 0.0000e+00 3.0000e-05 7.8000e-04 2.3330e-02 7.0000e-01
 8.0414e-01 8.7605e-01 9.2570e-01 9.5998e-01 9.8366e-01 1.0000e+00
 1.0000e+00 1.0000e+00 1.0000e+00 1.0000e+00 1.0000e+00 9.9000e-01
 4.2301e-01 1.7867e-01 7.3370e-02 2.7990e-02 8.4300e-03 0.0000e+00]
```

Later on, I will discuss how these parameters are used in the optimization model. For now, I just wanted you to see them.

Because it is very easy to change our value parameters around, I wanted a method of controlling different "sets" of value parameters. By changing your "set" of weight and value parameters that you're using, you could effectively change the entire problem. Therefore, CadetCareerProblem has an attribute called "vp_dict" that is a dictionary of different weight and value parameters.

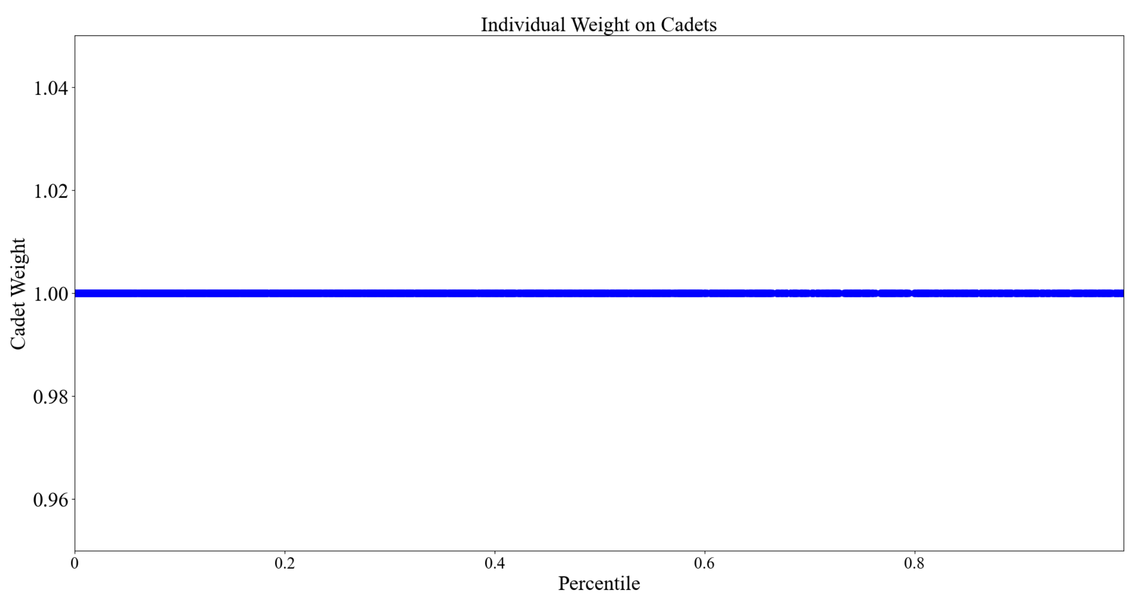
[70]:
```python
print(instance.vp_dict.keys())
```

```
dict_keys(['VP'])
```

Right now, we only have the one set of value parameters that we imported. If we change something and then save it, we'll have another set of value parameters.

[71]:
```python
# Change the overall weights on cadets/AFSCs
instance.value_parameters["cadets_overall_weight"] = 0.1
instance.value_parameters["afscs_overall_weight"] = 0.9

# Change the cadet weight function
instance.change_weight_function(cadets=True, function="Equal")

# Plot the new weight function
chart = instance.display_weight_function({"cadets_graph": True, "save": False})
```

```
Creating cadet weight chart…
```



[72]:
```python
# Save this set of value parameters as a new one!
instance.save_new_value_parameters_to_dict()

# We now have 2 sets of value parameters
print(instance.vp_dict.keys())

# Prints out the name of the current activated set of value parameters
print("Name of activated value parameters:", instance.vp_name)
```

```
dict_keys(['VP', 'VP_2'])
Name of activated value parameters: VP_2
```

I will note that I have a function that checks to see if this "new" set of value parameters is actually new. It compares every element of the new set of value parameters with each of the other ones in the dictionary. If it finds a matching set, then it will not save a copy. If you repeatedly run the above code, you won't keep adding new ones! Similarly by importing default value parameters again, we won't generate a new set.

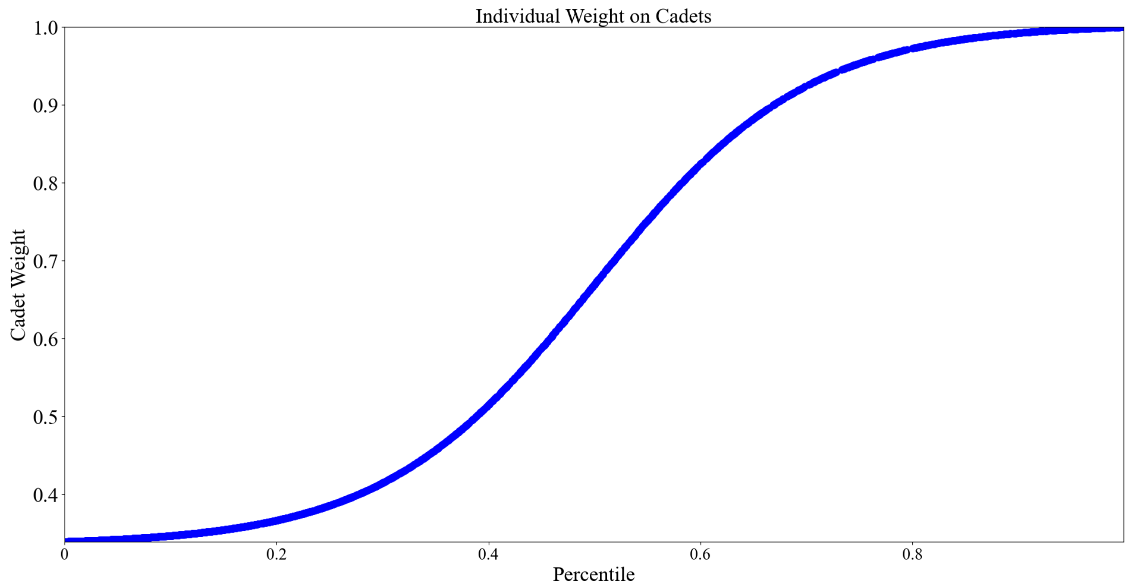
[73]:
```python
# Import default value parameters again
instance.import_default_value_parameters()

# We still have 2 sets of value parameters
print(instance.vp_dict.keys())

# This time though, we revert back to our first set!
print("Name of activated value parameters:", instance.vp_name)
```

```
Importing default value parameters…
Imported.
dict_keys(['VP', 'VP_2'])
Name of activated value parameters: VP
```

And now we're back to the original set we had

[74]:
```python
# Plot the new weight function
chart = instance.display_weight_function({"cadets_graph": True, "save": False})
```

```
Creating cadet weight chart…
```



[75]:
```
# Current weight on cadets
print(instance.value_parameters["cadets_overall_weight"])
print("Name of activated value parameters:", instance.vp_name)
```

```
0.7
Name of activated value parameters: VP
```

You can switch between sets of value parameters by doing this:

[76]:
```
instance.set_instance_value_parameters("VP_2")

# Current weight on cadets
print(instance.value_parameters["cadets_overall_weight"])
print("Name of activated value parameters:", instance.vp_name)

instance.set_instance_value_parameters("VP")
print()

# Current weight on cadets
print(instance.value_parameters["cadets_overall_weight"])
print("Name of activated value parameters:", instance.vp_name)
```

```
0.1
Name of activated value parameters: VP_2

0.7
Name of activated value parameters: VP
```

[77]:
```python
# Remove "VP_2" from the dictionary
instance.vp_dict.pop("VP_2")

# We're back to just "VP"
instance.vp_dict.keys()
```

[77]:
```
dict_keys(['VP'])
```

*IMPORTANT! Every time you import a new problem instance, you have to select a set of value parameters to use. Either generate new ones, or "set" them like above.

1.2.4 Solving the Problem

Now let's talk about how to actually generate new solutions. I'll first discuss non-VFT solution methods and then I will showcase the VFT methods.

Generating Solutions The simplest way to find a solution is to generate a random one!

[78]:
```python
instance.generate_random_solution()
```

```
Generating random solution…
Measured exact solution vector objective value: 0.2626. Unmatched cadets: 0
```

[78]:
```
array([ 9,  7,  8, …,  4, 19,  7])
```

Whenever you aquire a solution, we always measure it according to the VFT weight and value parameters (the Exact model by the way). That's where we get our objective value from. Additionally, just like the "vp_dict" is a dictionary of different sets of value parameters, we also have a "solution_dict" which (you guessed it) is a dictionary of different solutions.

[79]:
```python
# We just have the one random solution!
print(instance.solution_dict.keys())
```

```
dict_keys(['Random'])
```

[80]:
```python
# Solve a stable marriage algorithm version that I created (not very useful)
instance.stable_matching()

# Solve a greedy algorithm that I created (also not very useful)
instance.greedy_method()

# We now have a few different solutions here!
print(instance.solution_dict.keys())
```

```
Solving stable marriage model…
Measured exact solution vector objective value: 0.6341. Unmatched cadets: 0
Solving Greedy Model…
Measured exact solution vector objective value: 0.2546. Unmatched cadets: 0
dict_keys(['Random', 'Stable', 'Greedy'])
```

[81]:
```
# The original IP that AFPC has been using before the VFT model
instance.solve_original_pyomo_model({"max_time": 60, "provide_executable":␣
↪False})
```

```
Building original model…
Done. Solving original model…
Measured exact solution vector objective value: 0.7622. Unmatched cadets: 0
```

The “provide_executable” parameter just tells pyomo where to search for a solver. Don’t worry about it, it’s something I used to do when I’d run this on my macbook.

Another method I have to solve this problem is with a goal-programming (GP) model created by Lt. Rebecca Eisemann when she was at AFIT. Both of us were working on this problem for our theses, and so I coded up her model as best as I could to bring with me to AFPC. It follows her formulation pretty well I’d say, and if you’re interested you can read her thesis online by searching her name at the time (Rebecca Reynolds). Let’s first look at the parameters that her model uses:

[82]:
```
# Import her parameter data-frame just to look at it
filepath = dir_path + "support/gp_parameters.xlsx"
gp_df = pd.read_excel(filepath)
gp_df
```

[82]:

	Constraint	Normalized Penalty	Normalized Reward	Run Penalty	Run Reward
0	T	0.056736	0.006952	1	1
1	F	0.045822	0.008020	1	1
2	M	0.122749	0.009039	1	1
3	D_under	0.175498	0.007067	1	1
4	D_over	1.000000	1.000000	1	1
5	P	0.045950	0.061889	1	1
6	U_under	0.669725	0.025676	1	1
7	U_over	0.173123	0.081495	1	1
8	R_under	0.574803	0.028283	1	1
9	R_over	0.746548	0.032918	1	1
10	W	0.091345	0.013898	1	1
11	S	0.000000	0.000074	0	1

\

	Penalty Weight	Reward Weight
0	100	0
1	100	0
2	90	0
3	30	0
4	30	0
5	25	0
6	50	0
7	50	0
8	50	0
9	50	0
10	25	0
11	0	100

Before we solve the "GP" model, we have to convert my parameters and value_parameters to her "gp_parameters". I have two main methods of doing that: using the parameters above that were calculated for a previous class year or calculating new parameters for the current class year. In order to get these Normalized Penalties and Rewards that the GP model uses, you actually have to solve it each time for each constraint. That's 23 different iterations, and it takes time. So to make things easier, you can just approximate it with past years info. To be exact, however, you do need to solve it for this year's data.

[83]:
```
# Translate the parameters according to our "generalized" dataset above
instance.vft_to_gp_parameters({"use_gp_df": True, "get_new_rewards_penalties":␣
↪False})

# Print out the keys to the dictionary
print(instance.gp_parameters.keys())
```

```
Translating VFT model parameters to Goal Programming Model parameters…
Translated.
dict_keys(['A', 'C', 'con', 'A^', 'C^', 'param', 'utility', 'Big_M', 'u_limit',
'merit', 'mu^', 'lam^'])
```

Just like parameters and value_parameters, gp_parameters is also a dictionary of various items! Many of the above items in that dictionary are also dictionaries themselves. For example, in her formulation she has a set called $\mathcal{A}^M$ which is the set of all AFSCs with Mandatory AFOCD constraints. Since there are many other subsets of AFSCs that pertain to the various constraints ($\mathcal{A}^T$, $\mathcal{A}^F$, $\mathcal{A}^P$, etc.), I created a dictionary of AFSC subsets, where each key is a 1-dimensional numpy array of AFSC indices.

[84]:
```
# "A^M" for example
print(instance.gp_parameters["A^"]["M"])
```

```
[ 0  2  3  4  5  6  7 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
 29]
```

[85]:
```
# Solve the model!
instance.solve_gp_pyomo_model({"max_time": 60, "con_term": None,␣
↪"provide_executable": False})
```

```
Building GP Model…
Model built.
Solving GP Model…
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Input In [85], in <cell line: 2>()
      1 # Solve the model!
----> 2␣
  ↪instance.solve_gp_pyomo_model({"max_time": 60, "con_term": None, "provide_executable": Fals

File ~/Desktop/Coding Projects/afccp/afccp/core/problem_class.py:1154, in␣
  ↪CadetCareerProblem.solve_gp_pyomo_model(self, p_dict, printing)
```

```
   1152 # Solve the model
   1153 r_model = afccp.core.solutions.pyomo_models.gp_model_build(self,␣
 ↪printing=printing)
-> 1154 gp_var =␣
 ↪afccp.core.solutions.pyomo_models.gp_model_solve(self, r_model, printing=printing)
   1156 if self.mdl_p["con_term"] is None:
   1157     solution, self.x = gp_var

File ~/Desktop/Coding Projects/afccp/afccp/core/solutions/pyomo_models.py:844,␣
 ↪in gp_model_solve(instance, model, printing)
    840     return model.objective()
    841 else:
    842 
    843     # Get solution
--> 844     N, M = len(gp['C']), len(gp['A'])
    845     solution = np.zeros(N)
    846     X = np.zeros((N, M))

NameError: name 'gp' is not defined
```

VFT Model

1.2.5 Other

Exporting to Excel

Visualizations

Sensitivity Analysis First of all, I would recommend reading through my thesis as it gives a nice description on how the model all works. You don't have to, but it might help. I will suggest however that if you want to know more about how the value functions work in the optimization model, read section 3.4 in my thesis. Section 3.4.3, introduces the additional parameters, sets, and variables needed to make the model work that I ommit from my model presentation slides. Here is a screenshot of those formulation additions:

```
[ ]: Image(filename='pic9.png')
```

Constraints (20a) and (20b) force the objective measures and values to fall on some point along one of the line segments of the piece-wise value functions. Constraints (20c) through (20i) enforce the value of a particular objective measure to take on the value of the point between two breakpoints by using its slope. The λ variable determines the percentage in the x-space (the *measure* axis) that $measure_{jk}$ has left to travel between two breakpoints. For example, if $measure_{jk}$ is 30% of the way between breakpoints 2 and 3, then λ_{jk2} would be 0.7, and y_{jk2} would be 1. Since $\sum_{l \in \mathcal{L}_{jk}} \lambda_{jkl} = 1$, λ_{jk3} would be 0.3, but have no effect on $value_{jk}$ since y_{jk3} would be 0. Therefore this variable can take on continuous values between 0 and 1, inclusive.

The λ variable is also used to determine the percentage between two breakpoints in the y-space (the value axis) that this measure corresponds to. Because the segment is linear, λ can be calculated using *measure* as defined in the VFT model formulation in Section 3.3.3, as well as with the parameter a, alongside the other constraints. Once the distance along the x-space (the measure axis) between two breakpoints is used to locate a particular objective measure with respect to its value function, this translates to the distance in the y-space, and ultimately the value of that objective.

To enforce the condition that “at most two adjacent λ_{jkl} can be positive,” as described in [43], constraints (20c) through (20e) are defined. These constraints limit the variable λ to only be positive when the line segment on either side of its corresponding breakpoint is activated. Constraint (20f) forces only one line segment to be activated, so the sum of all y_{jkl} variables (for each AFSC objective) must be 1. The sum of all the λ_{jkl} variables for a particular AFSC objective value function must also be 1, since the largest percentage along one line segment that $measure_{jk}$ could be is 1. This is specified in constraint (20g). Constraints (20h) and (20i) define the variable domains.

[ ]:

```

```