afccp_tutorial

September 21, 2023

1 Air Force Cadet Career Problem Walkthrough

This jupyter notebook demonstrates the implementation of the Air Force Cadet Career Problem (afccp) python module developed and maintained by Griffen Laird for his master's thesis, and now implemented by AFPC/DSYA in the process to match Air Force cadets to their career fields. This notebook showcases the "CadetCareerProblem" object, which defines the family of cadet/AFSC matching problem instances.

2 Contact Info

This project has consumed a lot of my time/focus for the past couple of years both academically (at AFIT) and now operationally (at AFPC) and so I have no problem helping you make sense of all of it! I really mean it, if you ever have any problems with this and you're not sure what to do I highly encourage you to reach out to me through whatever means is easiest for you and I can help. Realistically, the people who'd be reading this tutorial are either at AFPC/DSYA or are working on this problem through some academic setting, meaning there aren't too many people so I have no problem pledging my ability to assist!

Personal email: griffenlaird007@gmail.com Work email: daniel.laird.4@us.af.mil Cell #: 8186877221

My preferred means of communication is either via my work email or my cell phone. You'll get a quicker response from me via text message, but my work email is good too. My personal email sometimes gets flooded with random things and I don't always see it all, but I do still check it so don't be afraid to reach me there if you don't want to try the other two options!

3 Getting Started

This module's main purpose is to automate as much of the matching process as possible. If we can focus on the general application of solving these problems, there is no reason that AFPC needs to spend so much time each year cleaning data and figuring out how to assign cadets to their AFSCs! That's where afccp comes in.

3.1 Working Directory

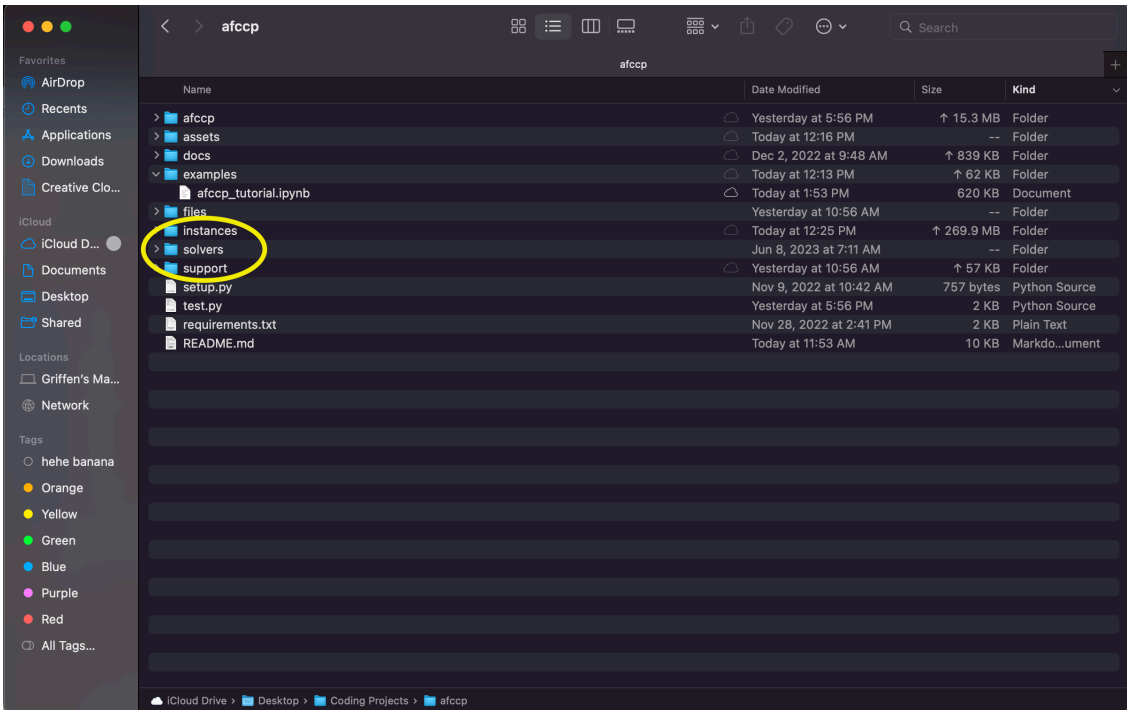
To keep things cleaner, I put all executables, this notebook included, in the "examples" folder of the "afccp" project. When we run one of those files, the working directory is initially that subfolder (examples) and so we would only have access to the files and modules within that sub-folder. This isn't going to work, so we need to change the working directory.

[1]:
```
# Import basic libraries
import numpy as np
import pandas as pd
import os

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

# Get main afccp folder path
index = dir_path.find('afccp')
dir_path = dir_path[:index + 6]

# Update working directory
os.chdir(dir_path)
print('updated working directory:', dir_path)
```

```
initial working directory: /Users/griffenlaird/Desktop/Coding
Projects/afccp/examples/
updated working directory: /Users/griffenlaird/Desktop/Coding Projects/afccp/
```

When I am running afccp code, I use a "test.py" script that is located in the root afccp folder. If instead of cloning in afccp, you decided to install it from github (with setup.py), this is more along the lines of how it'd work. The afccp module would operate like a python package and be located in your site-packages folder and so you'd use it within your working directory. Since you've cloned the repo, you can make your own "executables" folder (like "examples") and keep whatever code you build in there. If you do that, all you need to do is run the code above to set the current working directory to the root afccp (project) folder so you can access all afccp modules.

Here's an example of what my afccp project (and working directory) looks like. The folders circled in yellow will not be included in your project folder at the beginning but will be created when you import "CadetCareerProblem" for the first time.



The code below will show you what packages you have (I commented it out for sake of keeping this pdf output shorter). Notably, the main packages I use are numpy, pandas, matplotlib, pyomo, python-pptx, and some others. One easy way of getting all that you need is to install the packages using requirements.txt

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

The code below will install the packages from the requirements.txt file in the afccp project folder. I commented it out since for me it's just a lot of "Requirement already satisfied" and it also takes up quite a bit of ouput.

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

3.2 Importing the module

Now, you should be able to import "CadetCareerProblem". You can do that like this:

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

```
Importing 'afccp' module...
Pyomo module found.
SDV module found.
Sklearn Manifold module found.
Python PPTX module found.
```

I like to include some print statements just to show that the module is working properly (and it checks some of the less popular packages that I use to see if you have them installed because I still

want this code to work even if you don't have them). As mentioned previously, the first time you import CadetCareerProblem three folders will be created for you: instances, support, and solvers. The "instances" folder will store all of the data pertaining to the instances of CadetCareerProblem that you'll be working with. The "support" folder will contain the files that apply to all instances and are therefore shared. The "solvers" folder is meant to store pyomo executables which may be useful depending on how you work with pyomo.

If for some reason the code failed and gave you some error saying it doesn't recognize "afccp" as a module, you probably need to add it to the path. You can do that like this: (commented out again)

[5]:
```
# import sys

# print('System path before:', sys.path)
# sys.path.append(dir_path)  # Add the working directory to the path (contains␣
 ↪afccp)
# print('\nSystem path after:', sys.path)
```

That really shouldn't be an issue though since you've already changed the working directory to the root folder. If it's really not working correctly and you don't know why, please let me know and I can certainly assist.

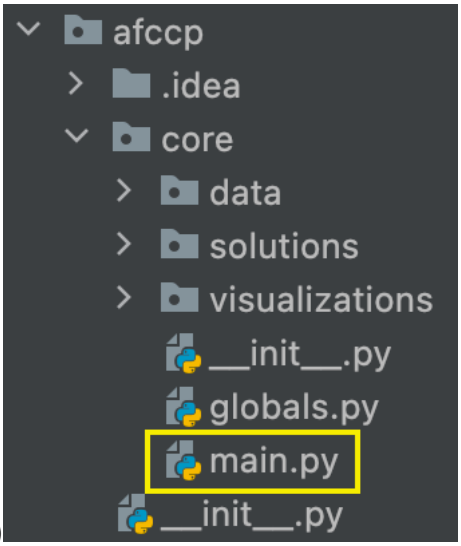
Once you have CadetCareerProblem imported, you should be good to go! The rest of the tutorial pertains to how to use it to its full capacity and how to contribute to it. Whether you're working on this problem academically or operationally (AFPC/DSYA), these tutorials should hopefully help you out!

4 CadetCareerProblem

Now that the CadetCareerProblem class is imported, it's time to talk about what it does. 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.

Before we get into the data, let's talk about how the code is structured. The class "CadetCareer-

Problem" lives in main.py (afccp.core.main)



CadetCareerProblem calls all of the other various functions across the "core" module that handle this problem. I've broken up these processes into three categories: "data", "solutions", and "visualizations". The next three sections will discuss these three different concepts in much more detail. For now, let's see what "main" looks like.

main.py test.py

```
import copy
import time

# afccp modules
import afccp.core.globals
import afccp.core.data.adjustments
import afccp.core.data.generation
import afccp.core.data.preferences
import afccp.core.data.processing
import afccp.core.data.support
import afccp.core.data.values
import afccp.core.solutions.algorithms
import afccp.core.solutions.handling
import afccp.core.visualizations.animation
import afccp.core.visualizations.charts

# Import optimization models if pyomo is installed
if afccp.core.globals.use_pyomo:
    import afccp.core.solutions.optimization
    import afccp.core.solutions.sensitivity

# Import slides script if python-pptx installed
if afccp.core.globals.use_pptx:
    import afccp.core.visualizations.slides
```

One thing I do across all my .py scripts is import each afccp module directly so that you can see which modules are dependent on each other. All the core scripts get imported directly into "main" since this serves as the hub for all the functionality of afccp. In other scripts, the modules imported are only the ones that are required. Another thing I do for context is include the entire module "path" in front of each function when I call it so that you can see where the function is written. For instance, if I wanted to call the function that defines all of the afccp model hyperparameters ("mdl_p"), I can do so like this:

[6]:
```
# Import the "data.support" module (this would be at the top of the script)
import afccp.core.data.support
```

```
# Call the specific function from that script (this example returns a␣
 ↪dictionary of default "hyper-parameters")
mdl_p = afccp.core.data.support.initialize_instance_functional_parameters(N=20)␣
 ↪ # Requires the number of cadets

# There is a lot of information in here used across afccp
print(mdl_p['figsize'])  # This is the default figure size for my matplotlib␣
 ↪plots!
```

```
(19, 10)
```

To actually match cadets to their AFSCs, there is a "preprocessing" submodule next to "core" that deals with the real class years that AFPC/DSYA handles. This is why I do have the designation between afccp and core, even though core is currently the only submodule of afccp (it isn't when run operationally!). The intent of the "preprocessing" submodule is to get the data from its source(s) into the correct format to work with CadetCareerProblem. Throughout my code I try to provide decent "docstrings" that describe what that function or object is doing. For example, here's the one for "CadetCareerProblem" itself:

```
class CadetCareerProblem:
    def __init__(self, data_name="Random", data_version="Default", degree_qual_type="Consistent",
                 num_value_function_breakpoints=None, N=1600, M=32, P=6, generate_only_nrl=False, printing=True):
        """
        Represents the AFSC/Cadet matching problem object.

        Parameters:
            data_name (str): The name of the data set. It can be an existing instance folder name or one of the following:
                             "Random", "Perfect", or "Realistic". Defaults to "Random".
            data_version (str): The version of the data set. It is used to manage different versions of cadets/AFSCs
                                and handle NRL vs All cadet scenarios. Defaults to "Default".
            degree_qual_type (str): The type of degree tier qualifications. It can be "Binary", "Relaxed", or "Tiers".
                                    Defaults to "Consistent".
            num_value_function_breakpoints (int, optional): The number of breakpoints to use in the value functions.
                                                            If None, the value functions will be loaded with the existing
                                                            breakpoints. If an integer is provided, new value functions
                                                            will be created. Defaults to None.
            N (int): Number of cadets to generate. Defaults to 1600.
            M (int): Number of AFSCs to generate. Defaults to 32.
            P (int): Number of AFSC preferences to generate for each cadet. Defaults to 6.
            generate_only_nrl (bool): Whether to generate only NRL AFSCs. Defaults to False.
            printing (bool): Whether to print status updates or not. Defaults to True.

        This class represents the AFSC/Cadet problem object. It can import existing data or generate new instances.
        The data set can be specified by providing the name of the instance folder or using the predefined data types
        ("Random", "Perfect", "Realistic"). The problem instance includes information about cadets, AFSCs, value parameters,
        and solutions.

        Example usage:
            instance = CadetCareerProblem(data_name="Random", N=200, M=5, P=5)
        """
```

I highly encourage you to read through my code at least to some extent to understand what it's doing. I encourage this for two reasons: to understand how afccp works but perhaps more importantly to understand how python works if you're not as strong of a python coder just yet! A really good way to start, at least in my opinion, is to read through the "**init**" function of CadetCareerProblem (what's shown above). This is the function that gets called as soon as you create an instance of this object (ie. you run "instance = CadetCareerProblem()"). Within that **init** function I call many other functions from other modules too that you can read if you want a deeper dive into this process. If you're able to follow the flow of the code along and understand, at least to some extent, what it's doing then you'll really be able to understand my structure for

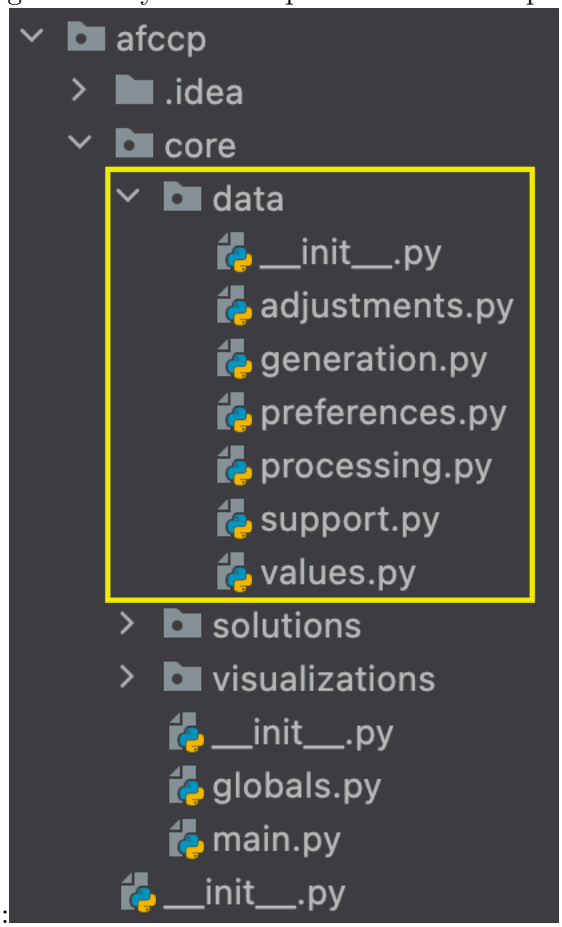
afccp.

5 Data

A "CadetCareerProblem" instance comes with a lot of data, and so the largest section of this tutorial is here to talk about that data. I'll outline where the data is coming from (the various csvs), how the data is captured within the CadetCareerProblem class, and where the code is that manages all of it.

5.1 afccp.core.data

The "data" module of afccp.core contains the scripts and functions that all have something to do with processing the many sets and parameters of the problem. At a high level the modules are setup like this:



The "adjustments.py" script holds the functions that manipulate the parameters. There is also a function that sanity checks many different sets and subsets of parameters here which is extremely important in avoiding preventable errors. For generating simulated data, "generation.py" contains those functions. Because I ended up making quite a few functions that deal with cadet and AFSC preferences, I took those and put them into "preferences.py". For importing and exporting data, as well as handling some of the file information, we have "processing.py". There are a few functions designated to support CadetCareerProblem and its hyper parameters and whatnot so I put those into "support.py". Lastly, the value parameters, which I will discuss later, are all mostly handled by "values.py".

5.2 Generating data

CadetCareerProblem allows for "fake" class years using simulated data generated through various means. You may or may not have real class year data, but we can generate data to play around with here.

[7]:
```
# Create a randomly generated problem instance with 20 cadets and 4 AFSCs
instance = CadetCareerProblem('Random', N=20, M=4, P=4)
```

```
Generating 'Random_1' instance…
Instance 'Random_1' initialized.
```

That one line above initializes a new instance of the cadet-AFSC matching problem (CadetCareerProblem). N is the number of cadets, M is the number of AFSCs, and P is the number of preferences the cadets are allowed to express. Originally, cadets could only express six preferences but today they're able to provide complete preference lists. I recommend always making P equal to M.

The first parameter specified is referred to as the "data_name" of the instance. When generating data, simply write "Random" and the code will determine which instance to generate based on what you currently have in your "instances" folder. Since we haven't generated and exported any data yet, "Random_1" is the instance we create. Note, there won't be a "Random_1" instance folder in your working directory yet since you haven't exported it. I did this purposely so as not to flood your files with instances until you want it and call "instance.export_data()".

The following code is here to manipulate the data I generated to get it into the correct format and ultimately export it back as csvs so I can point to different elements in the code from the data. NOTE: This is purely meant for GENERATED data and I'm doing this so I can export it and then you'll have a generated dataset to import and follow along with as well if you run this code. Don't worry about this just yet.

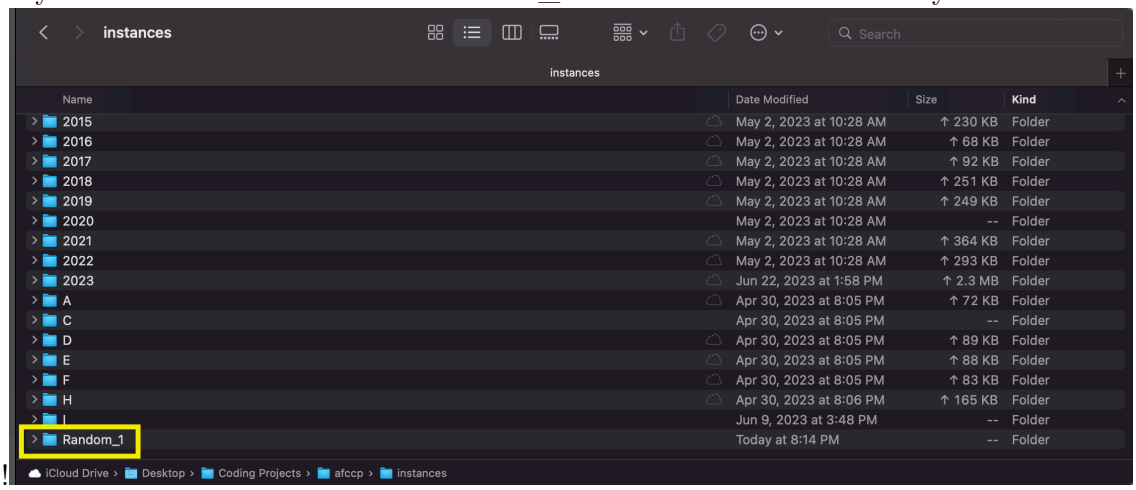
[8]:
```
# Fix "Random" data (only meant for generated data!!)
instance.fix_generated_data()

# Export everything
instance.export_data()
```

```
1 Making 6 cadets ineligible for 'R2' by altering their qualification to 'I2'.
3 Making 4 cadets ineligible for 'R4' by altering their qualification to 'I2'.
Removing ineligible cadets based on any of the three eligibility sources
(c_pref_matrix, a_pref_matrix, qual)…
Edit 1: Cadet 0 not eligible for R4 based on degree qualification matrix but the
AFSC was in the cadet preference list. c_pref_matrix position (0, 3) set to 0.
Edit 2: Cadet 2 not eligible for R2 based on degree qualification matrix but the
AFSC was in the cadet preference list. c_pref_matrix position (2, 1) set to 0.
Edit 3: Cadet 6 not eligible for R2 based on degree qualification matrix but the
AFSC was in the cadet preference list. c_pref_matrix position (6, 1) set to 0.
Edit 4: Cadet 7 not eligible for R4 based on degree qualification matrix but the
AFSC was in the cadet preference list. c_pref_matrix position (7, 3) set to 0.
Edit 5: Cadet 10 not eligible for R4 based on degree qualification matrix but
the AFSC was in the cadet preference list. c_pref_matrix position (10, 3) set to
```

```
0.
Edit 6: Cadet 12 not eligible for R2 based on degree qualification matrix but
the AFSC was in the cadet preference list. c_pref_matrix position (12, 1) set to
0.
Edit 7: Cadet 16 not eligible for R2 based on degree qualification matrix but
the AFSC was in the cadet preference list. c_pref_matrix position (16, 1) set to
0.
Edit 8: Cadet 17 not eligible for R2 based on degree qualification matrix but
the AFSC was in the cadet preference list. c_pref_matrix position (17, 1) set to
0.
Edit 9: Cadet 18 not eligible for R2 based on degree qualification matrix but
the AFSC was in the cadet preference list. c_pref_matrix position (18, 1) set to
0.
Edit 10: Cadet 19 not eligible for R4 based on degree qualification matrix but
the AFSC was in the cadet preference list. c_pref_matrix position (19, 3) set to
0.
10 total adjustments.
Updating cadet preference matrices from the preference dictionaries. ie. 1, 2,
4, 6, 7 -> 1, 2, 3, 4, 5 (preference lists need to omit gaps)
Converting AFSC preferences (a_pref_matrix) into percentiles (afsc_utility on
AFSCs Utility.csv)…
Updating cadet columns (Cadets.csv…c_utilities, c_preferences) from the
preference matrix (c_pref_matrix)…
Sanity checking the instance parameters…
Done, 0 issues found.
Exporting datasets ['Cadets', 'AFSCs', 'Preferences', 'Goal Programming', 'Value
Parameters', 'Solutions']
```

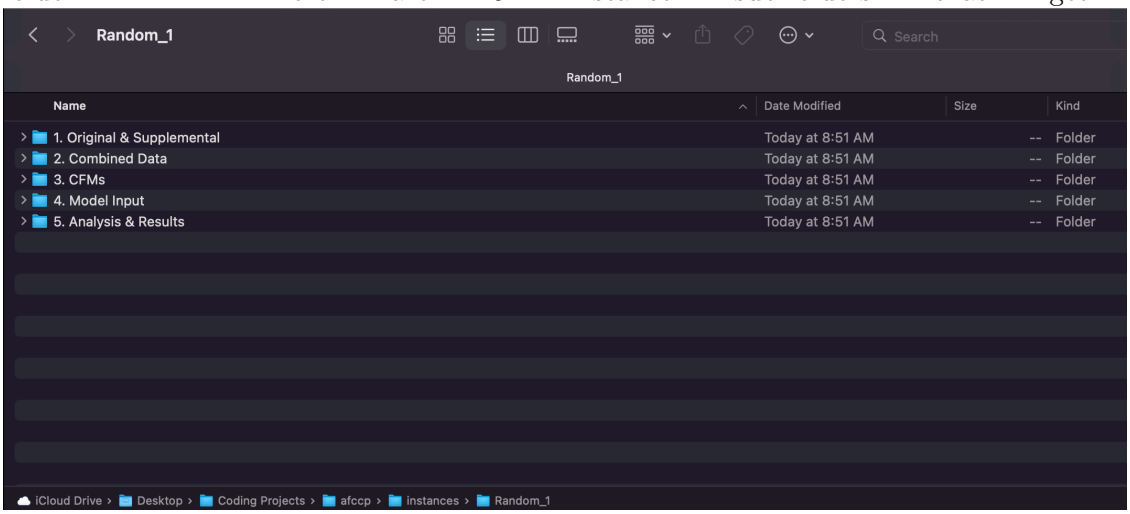
Quick note: the output above doesn't match the Random_1 instance you'll see below because I've run this notebook multiple times and therefore generate different instances that I discard. I'm only referring to the output above, and everything below IS the instance I initially generated. Anyway, just don't worry about it!

Now that we've exported the data (after manipulating it a little), you should have a "Random_1" sub-folder within your "instances" folder!



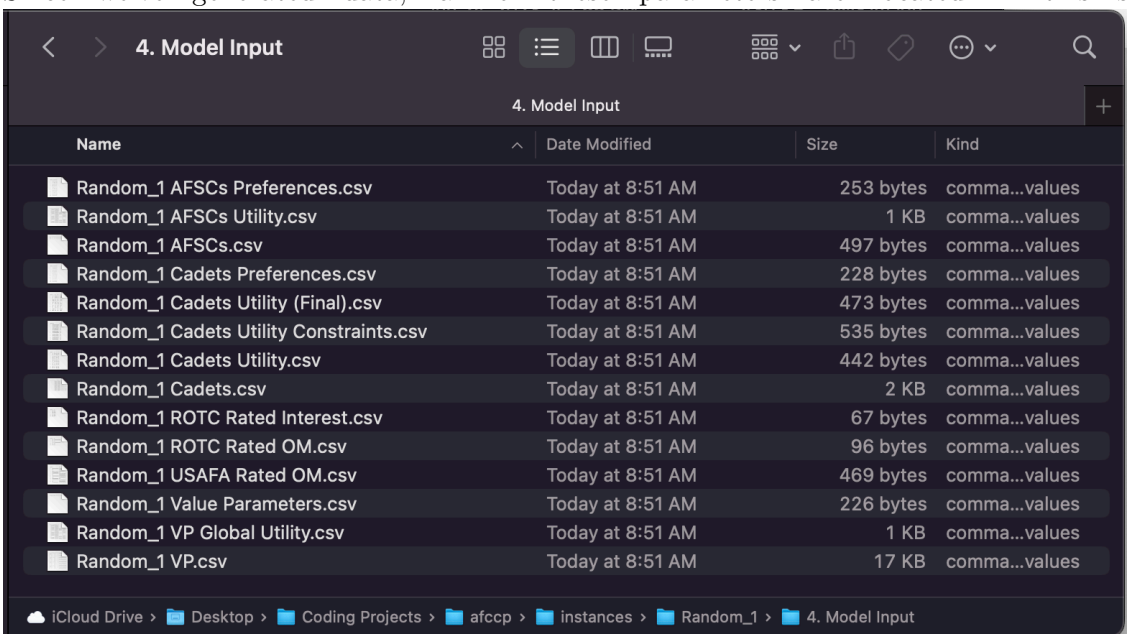
5.3 Instance Folder Structure

You have data now located within the "Random_1" sub-folder. There are 5 instance sub-folders that get created:



The first 3 all relate to the pre-processing "phases" that AFPC/DSYA goes through to get the data into the problem instance format and are irrelevant for this tutorial. For a real class year of cadets/AFSCs, these 3 sub-folders will be filled with real data in order to get it into sub-folder "4. Model Input".

Since we've generated data, all of these parameters are located in this sub-folder:



The files shown above, and which you should also have if you're following along, contain all the information stored in the data dictionaries "parameters" and "value_parameters". In a moment I will discuss what that data looks like and how it's stored in this dictionary structure. First, however, let's re-import the "Random_1" problem instance. Note: you already have that instance in memory since we exported it above. I'm just going to show you what it looks like to import it since this is likely what you'll be doing more of anyway.

5.4 Importing data

To import data, it is very simple: specify the "data_name" of the instance you want to import. Here, we will import "Random_1".

[9]:
```
# Import "Random_1" instance
instance = CadetCareerProblem('Random_1')
```

```
Importing 'Random_1' instance…
Instance 'Random_1' initialized.
```

5.5 "parameters"

We now have a working problem instance. This "instance" object has many different attributes and methods defined that we can now access. The "parameters" of the instance are represented by a dictionary, which is an attribute of the instance object. Various parameters are loaded in as numpy arrays within that dictionary. These are the "fixed" parameters, and contain many different characteristics of this particular dataset. I call them "fixed" parameters because these are the attributes of the problem that the analyst does not have much, if any, control over (the characteristics of the cadets and AFSCs themselves). Let's first discuss the two "primary" datasets: "Cadets.csv" and "AFSCs.csv".

5.5.1 Cadets

"Random_1 Cadets.csv" defines the basic features of the cadets in this problem instance. It looks like this:

Cadet	Assigned	USAFA	Male	Minority	Merit	Real Merit	Util_1	Util_2	Util_3	Util_4	Pref_1	Pref_2	Pref_3	Pref_4	qual_R1	qual_R2	qual_R3	qual_R4
0		0	1	1	0.49154715	0.49154715	1	0.79	0.71	0.07	R3	R1	R4	R2	D1	P1	P2	P1
1		1	1	0	0.11113553	0.11113553	1	0.37	0.27	0.07	R3	R1	R4	R2	P2	P1	P2	P1
2		1	1	0	0.25882888	0.25882888	1	0.96	0.74	0.25	R2	R4	R3	R1	P2	P1	D1	P1
3		0	1	1	0.53686789	0.53686789	1	0.88	0.82	0.73	R1	R3	R4	R2	D1	P1	D1	P1
4		1	1	1	0.91847252	0.91847252	1	0.58	0.53	0.25	R1	R4	R3	R2	D1	P1	P2	P1
5		1	1	1	0.533902	0.533902	1	0.43	0.27	0.2	R3	R2	R1	R4	D1	P1	P2	P1
6		1	1	1	0.00451476	0.00451476	1	0.46	0.17	0.04	R2	R3	R4	R1	D1	P1	D1	P1
7		1	0	1	0.09631769	0.09631769	1	0.91	0.27	0.23	R3	R4	R1	R2	D1	P1	D1	P1
8		0	1	0	0.27906107	0.27906107	1	0.54	0.31	0.29	R4	R2	R3	R1	D1	P1	P2	P1
9		1	1	1	0.89425573	0.89425573	1	0.98	0.26	0.07	R2	R4	R3	R1	D1	P1	D1	P1
10		0	1	1	0.18714076	0.18714076	1	0.53	0.5	0.19	R2	R1	R4	R3	P2	P1	D1	P1
11		1	1	1	0.72979023	0.72979023	1	0.79	0.7	0.5	R3	R2	R1	R4	D1	P1	P2	P1
12		0	0	0	0.68416277	0.68416277	1	0.83	0.19	0	R1	R2	R3		D1	P1	D1	I2
13		0	0	1	0.86596516	0.86596516	1	0.48	0.36	0	R3	R2	R1		P2	P1	D1	I2
14		1	1	1	0.55904708	0.55904708	1	0.49	0.23	0	R3	R1	R2		P2	P1	D1	I2
15		1	0	1	0.47407413	0.47407413	1	0.44	0.22	0	R3	R1	R2	R4	P2	P1	P2	P1
16		1	1	1	0.58194032	0.58194032	1	0.7	0.6	0.21	R2	R4	R3	R1	P2	P1	D1	P1
17		1	0	1	0.00653595	0.00653595	1	0.75	0.64	0.01	R2	R3	R4	R1	D1	P1	P2	P1
18		0	0	0	0.30188118	0.30188118	1	0.64	0.23	0.14	R2	R1	R3	R4	D1	P1	P2	P1
19		1	1	1	0.63428494	0.63428494	0.72	0.35	0.22	0	R1	R2	R3		D1	P1	P2	I2

Note: if you're following along, your data will differ from this image since you're generating your own unique set of cadets and AFSCs! We can gain quite a bit of information from this dataset. I will reiterate that data is represented in this module as numpy arrays within certain dictionaries. I extract these arrays from excel using pandas as the dataframe vehicle. Let's go through some of these arrays.

[10]:
```
# Cadet identifier
print("'cadets':", instance.parameters['cadets'])

# Binary USAFA array
print("'usafa':", instance.parameters['usafa'])

# Binary Male array
print("'male':", instance.parameters['male'])
```

```
# Binary Minority array
print("'minority':", instance.parameters['minority'])
```

```
'cadets': [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19]
'usafa': [0 1 1 0 1 1 1 1 0 1 0 1 0 0 1 1 1 1 0 1]
'male': [1 1 1 1 1 1 1 0 1 1 1 1 0 0 1 0 1 0 0 1]
'minority': [1 0 0 1 1 1 1 1 0 1 1 1 0 1 1 1 1 1 0 1]
```

There are two arrays containing order of merit percentile data. This is because this module used to simply be used for the Non-Rated line. In the NRL process, we re-scaled OM so that it averaged to about 0.50 since the Rated and USSF cadets were not in the mix. This creates two separate OM arrays: the re-sorted OM (merit) and the "real" OM where the cadets ranked among their entire class (merit_all).

[11]:
```
print("'Real' Merit:", instance.parameters['merit_all'])
print("'NRL only' Merit:", instance.parameters['merit'])
```

```
'Real' Merit: [0.49154715 0.11113553 0.25882888 0.53686789 0.91847252 0.533902
 0.00451476 0.09631769 0.27906107 0.89425573 0.18714076 0.72979023
 0.68416277 0.86596516 0.55904708 0.47407413 0.58194032 0.00653595
 0.30188118 0.63428494]
'NRL only' Merit: [0.49154715 0.11113553 0.25882888 0.53686789 0.91847252
0.533902
 0.00451476 0.09631769 0.27906107 0.89425573 0.18714076 0.72979023
 0.68416277 0.86596516 0.55904708 0.47407413 0.58194032 0.00653595
 0.30188118 0.63428494]
```

You won't see a difference above because this data was generated and I didn't really see a need to differentiate it. Additionally, since AFPC/DSYA is now tasked with matching all cadets (not just NRL), there likely won't need to be a designation in the future so we may go back to just one "Merit" column.

The "Assigned" column contains the AFSCs that may be fixed for certain cadets. Perhaps some cadets were rolled over from the previous AFSC and had already been awarded an AFSC. In those cases, we want to count them within our calculations but don't want to change their assigned AFSC. Again, since this is generated data, it does not play much of a role.

[12]:
```
# Array of already awarded AFSCs for each of the cadets
instance.parameters['assigned']  # Empty array!
```

[12]:
```
array([nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan,
       nan, nan, nan, nan, nan, nan, nan])
```

The "Util_1" -> "Util_4" columns indicate the "utilities" that the cadets have placed on their first, second, third, and fourth choice AFSCs. The "Pref_1" -> "Pref_4" columns indicate the ordered list of AFSC choices the cadets provided. Capturing the preference data in this manner (rows are cadets & columns are the choices) is the way we've always "initially" represented it. I will show later in the "Preferences" section that we can convert them into another useful representation of the data where the rows are still cadets but the columns are the AFSCs themselves.

[13]:
```
# Utility cadet columns shown in numpy arrays
print('Cadet Utilities\n', instance.parameters['c_utilities'])

# Preference cadet columns shown in numpy arrays
print('\nCadet Preferences\n', instance.parameters['c_preferences'])
```

```
Cadet Utilities
 [[1.   0.79 0.71 0.07]
 [1.   0.37 0.27 0.07]
 [1.   0.96 0.74 0.25]
 [1.   0.88 0.82 0.73]
 [1.   0.58 0.53 0.25]
 [1.   0.43 0.27 0.2 ]
 [1.   0.46 0.17 0.04]
 [1.   0.91 0.27 0.23]
 [1.   0.54 0.31 0.29]
 [1.   0.98 0.26 0.07]
 [1.   0.53 0.5  0.19]
 [1.   0.79 0.7  0.5 ]
 [1.   0.83 0.19 0.  ]
 [1.   0.48 0.36 0.  ]
 [1.   0.49 0.23 0.  ]
 [1.   0.44 0.22 0.  ]
 [1.   0.7  0.6  0.21]
 [1.   0.75 0.64 0.01]
 [1.   0.64 0.23 0.14]
 [0.72 0.35 0.22 0.  ]]

Cadet Preferences
 [['R3' 'R1' 'R4' 'R2']
 ['R3' 'R1' 'R4' 'R2']
 ['R2' 'R4' 'R3' 'R1']
 ['R1' 'R3' 'R4' 'R2']
 ['R1' 'R4' 'R3' 'R2']
 ['R3' 'R2' 'R1' 'R4']
 ['R2' 'R3' 'R4' 'R1']
 ['R3' 'R4' 'R1' 'R2']
 ['R4' 'R2' 'R3' 'R1']
 ['R2' 'R4' 'R3' 'R1']
 ['R2' 'R1' 'R4' 'R3']
 ['R3' 'R2' 'R1' 'R4']
 ['R1' 'R2' 'R3' '          ']
 ['R3' 'R2' 'R1' '          ']
 ['R3' 'R1' 'R2' '          ']
 ['R3' 'R1' 'R2' 'R4']
 ['R2' 'R4' 'R3' 'R1']
 ['R2' 'R3' 'R4' 'R1']
 ['R2' 'R1' 'R3' 'R4']
```

```
 ['R1' 'R2' 'R3' '          ']]
```

The last cadet has a utility < 1 as their first choice by coincidence since this is generated data. A real class year should have all 1s in their first choice.

The last section of data contains the degree qualifications! Qualifications for AFSCs are currently determined by the Air Force Officer Classification Directory (AFOCD). Each AFSC provides a tiered list of degree groups (tiers 1, 2, 3, etc.) as well as a requirement level for that degree tier ("Mandatory", "Desired", "Permitted). In some cases, the AFSC also has an implied"Ineligible" tier. M, D, P, and I are the letters representing the four kinds of tiers shown in the qualification matrix below. The numbers correspond with the tier itself (1, 2, 3, ...).

[14]:
```
instance.parameters['qual']
```

[14]:
```
array([['D1', 'P1', 'P2', 'P1'],
       ['P2', 'P1', 'P2', 'P1'],
       ['P2', 'P1', 'D1', 'P1'],
       ['D1', 'P1', 'D1', 'P1'],
       ['D1', 'P1', 'P2', 'P1'],
       ['D1', 'P1', 'P2', 'P1'],
       ['D1', 'P1', 'D1', 'P1'],
       ['D1', 'P1', 'D1', 'P1'],
       ['D1', 'P1', 'P2', 'P1'],
       ['D1', 'P1', 'D1', 'P1'],
       ['P2', 'P1', 'D1', 'P1'],
       ['D1', 'P1', 'P2', 'P1'],
       ['D1', 'P1', 'D1', 'I2'],
       ['P2', 'P1', 'D1', 'I2'],
       ['P2', 'P1', 'D1', 'I2'],
       ['P2', 'P1', 'P2', 'P1'],
       ['P2', 'P1', 'D1', 'P1'],
       ['D1', 'P1', 'P2', 'P1'],
       ['D1', 'P1', 'P2', 'P1'],
       ['D1', 'P1', 'P2', 'I2']], dtype='<U2')
```

5.5.2 AFSCs

Like cadets, AFSCs are also defined in a separate csv file (Random_1 AFSCs.csv) which looks like this:

AFSC	Accessions G	USAFA Targe	ROTC Target	PGL Target	Estimated	Desired	Min	Max	Deg Tier 1	Deg Tier 2	Deg Tier 3	Deg Tier 4
R1	NRL	1	5	6	6	6	6	7	D > 0.54	P < 0.4599		
R2	NRL	1	4	5	7	7	5	8	P = 1			
R3	USSF	1	3	4	4	4	4	4	D > 0.12	P < 0.88		
R4	Rated	0	2	2	2	2	2	3	P = 1	I = 0		

Here we have 4 AFSCs, and each has its own set of unique characteristics.

[15]:
```
# Array of AFSC names
print(instance.parameters['afscs'])
```

```
['R1' 'R2' 'R3' 'R4' '*']
```

One thing to note is the extra AFSC "*". This represents the"unmatched" AFSC since we can have

partial solutions where not all cadets go matched (think Rated) or in certain algorithms we may simply leave cadets unmatched. By allowing this extra AFSC at the end we can still evaluate these kinds of solutions. As a result, we do have certain parameters where we add a column at the end for this unmatched AFSC. One example is the cadet utility matrix below. For context, this matrix represents the same information captured in "c_utilities" only this time the columns are sorted by AFSC order, not the preference order.

[16]:
```
# Utility matrix (cadet submitted)
print(instance.parameters['utility'])
```

```
[[0.79 0.07 1.   0.71 0.  ]
 [0.37 0.07 1.   0.27 0.  ]
 [0.25 1.   0.74 0.96 0.  ]
 [1.   0.73 0.88 0.82 0.  ]
 [1.   0.25 0.53 0.58 0.  ]
 [0.27 0.43 1.   0.2  0.  ]
 [0.04 1.   0.46 0.17 0.  ]
 [0.27 0.23 1.   0.91 0.  ]
 [0.29 0.54 0.31 1.   0.  ]
 [0.07 1.   0.26 0.98 0.  ]
 [0.53 1.   0.19 0.5  0.  ]
 [0.7  0.79 1.   0.5  0.  ]
 [1.   0.83 0.19 0.99 0.  ]
 [0.36 0.48 1.   0.22 0.  ]
 [0.49 0.23 1.   0.7  0.  ]
 [0.44 0.22 1.   0.   0.  ]
 [0.21 1.   0.6  0.7  0.  ]
 [0.01 1.   0.75 0.64 0.  ]
 [0.64 1.   0.23 0.14 0.  ]
 [0.72 0.35 0.22 1.   0.  ]]
```

Note the extra column of zeros at index 4. There are 4 AFSCs (0, 1, 2, 3) but we make an extra for the unmatched AFSC (always at the end!).

When we generate random data, and if we have at least 4 AFSCs, I make sure I generate at least one AFSC from each of the three "accessions groups": Rated, USSF, NRL. You can track which AFSCs are in which group here:

[17]:
```
# "Accessions Groups", and their associated AFSCs, represented in this instance:
instance.parameters['afscs_acc_grp']
```

[17]:
```
{'Rated': array(['R4'], dtype=object),
 'USSF': array(['R3'], dtype=object),
 'NRL': array(['R1', 'R2'], dtype=object)}
```

[18]:
```
# Indices of AFSCs in each accessions group
for grp in instance.parameters['afscs_acc_grp']:
    param = "J^" + grp
    print(param, instance.parameters[param])
```

```
J^Rated [3]
J^USSF [2]
J^NRL [0 1]
```

The next 7 columns all refer to the quantities of cadets assigned to the AFSCs. The USAFA and ROTC "targets" are taken from the Production Guidance Letter (PGL) produced by A1PT. These outline how many new lieutenants need to be produced from both sources of commissioning.

[19]:
```
# Number of USAFA cadets needed for each AFSC
print('USAFA:', instance.parameters['usafa_quota'])

# Number of ROTC cadets needed for each AFSC
print('ROTC', instance.parameters['rotc_quota'])
```

```
USAFA: [1. 1. 1. 0.]
ROTC [5. 4. 3. 2.]
```

These numbers are largely ignored since the real goal is meeting the combination of the two targets. If we were strict on meeting these quotas for both sources of commissioning it would be very challenging and result in a worse outcome for everyone. Therefore, the main PGL target we shoot for is aptly named "PGL Target"

[20]:
```
# Real quota of cadets needed for each AFSC
instance.parameters['pgl']
```

[20]:
```
array([6., 5., 4., 2.])
```

The "Estimated" and "Desired" numbers of cadets are both used purely in the Value-Focused Thinking (VFT) model. The VFT model, as it stands, is non-linear and non-convex since there is a variable divided by another variable in the objective function. For example, the "average merit" calculation for a particular AFSC j is:

$$\frac{\sum_{i \in \mathcal{I}} merit_i \cdot x_{ij}}{\sum_{i \in \mathcal{I}} x_{ij}}$$

Because it is non-linear, I created an "approximate" model where I approximate the number of cadets using some estimated number -> hence, the "Estimated" parameter! The "Desired" parameter is fed into the quota value function which I will discuss later on.

[21]:
```
# Estimated number of cadets for each AFSC (Used in objective function as␣
↪denominator for certain objectives)
print("Estimated:", instance.parameters['quota_e'])

# Desired number of cadets for each AFSC (same as above because it's fake data)
print("Desired:", instance.parameters['quota_d'])
```

```
Estimated: [6. 7. 4. 2.]
Desired: [6. 7. 4. 2.]
```

Because the PGL target only provides one data point, I still need to have a range on the number of cadets that can be assigned. This is where the minimum and maximum quantities are used (lower and upper bounds on the number of cadets to be assigned).

[22]:
```
# Minimum number of cadets that can be assigned to each AFSC
print('Minimum:', instance.parameters['quota_min'])

# Maximum number of cadets that can be assigned to each AFSC
print('Maximum:', instance.parameters['quota_max'])
```

```
Minimum: [6. 5. 4. 2.]
Maximum: [7. 8. 4. 3.]
```

The "Deg Tier" columns contain the data on the target proportions of degrees from each tier requested for the AFSCs. This information is gathered into the value parameters that will be discussed later on.

[23]:
```
instance.parameters['Deg Tiers']
```

[23]:
```
array([['D > 0.54', 'P < 0.4599', '', ''],
       ['P = 1', '', '', ''],
       ['D > 0.12', 'P < 0.88', '', ''],
       ['P = 1', 'I = 0', '', '']], dtype='<U10')
```

5.5.3 Parameter "Additions"

From this initial set of data, we can derive more parameters and sets to use in the various models.

[24]:
```
# Numbers of Cadets, AFSCs, and AFSC preferences
for param in ['N', 'M', 'P']:
    print(param + ':', instance.parameters[param])
```

```
N: 20
M: 4
P: 4
```

[25]:
```
# Sets of cadets and AFSCs (indices)
for param in ['I', 'J']:
    print(param, instance.parameters[param])
```

```
I [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19]
J [0 1 2 3]
```

My sets of cadets and AFSCs ($\mathcal{I}$ and $\mathcal{J}$) are numpy arrays of indices since this allows for more flexibility than just using AFSC/cadet names since we can access other arrays using those indices through the power of numpy.

[26]:
```
# Set of cadets that are eligible for AFSC at index 3 (R4)
print('cadet indices', instance.parameters['I^E'][3])
```

```
cadet indices [ 0  1  2  3  4  5  6  7  8  9 10 11 15 16 17 18]
```

[27]:
```
# Set of AFSCs that the cadet at index 14 is eligible for
print('AFSC indices', instance.parameters['J^E'][14])
```

```
AFSC indices [0 1 2]
```

[28]:
```
# Names of the AFSCs that the cadet at index 14 is eligible for
print("AFSC Names:", instance.parameters['afscs'][instance.
 ↪parameters['J^E'][14]])
```

```
AFSC Names: ['R1' 'R2' 'R3']
```

[29]:
```
# USAFA cadets
print("USAFA cadets", instance.parameters['usafa_cadets'])

# ROTC cadets
print("ROTC cadets", instance.parameters['rotc_cadets'])
```

```
USAFA cadets [ 1  2  4  5  6  7  9 11 14 15 16 17 19]
ROTC cadets [ 0  3  8 10 12 13 18]
```

I have a dictionary I^D which contains the cadets with certain demographics that are also eligible for the various AFSCs. Many of these sets and subets are discussed in my thesis too which are potentially better defined there: https://apps.dtic.mil/sti/pdfs/AD1172355.pdf

[30]:
```
# The keys to the "I^D" dictionary are the objectives for the AFSCs that deal␣
 ↪with demographics of the cadets
print(instance.parameters['I^D'].keys())
```

```
dict_keys(['USAFA Proportion', 'Mandatory', 'Desired', 'Permitted', 'Tier 1',
'Tier 2', 'Tier 3', 'Tier 4', 'Male', 'Minority'])
```

The objectives above are a subset of the possible AFSC objectives $\mathcal{K}$ and are used with the value parameters that I will discuss in a later section!

[31]:
```
# Cadets with Tier 2 degrees that are eligible for the AFSC at index 3 (R4)
print("Cadets:", instance.parameters['I^D']['Tier 2'][3])
```

```
Cadets: [12 13 14 19]
```

[32]:
```
# USAFA Cadets with Tier 2 degrees that are eligible for the AFSC at index 3␣
 ↪(R4)
usafa_cadets_with_tier_2_degrees_afsc_r4 = \
    np.intersect1d(instance.parameters['I^D']['Tier 2'][3], instance.
 ↪parameters['usafa_cadets'])
print("Intersection:", usafa_cadets_with_tier_2_degrees_afsc_r4)
```

```
Intersection: [14 19]
```

[33]:
```
# The OM of those cadets
print('Merit', instance.
 ↪parameters['merit'][usafa_cadets_with_tier_2_degrees_afsc_r4])
```

```
Merit [0.55904708 0.63428494]
```

5.5.4 Preferences

Cadet preferences, as well as AFSC preferences now, are provided as numpy arrays of shape (NxM) with an extra column for cadet preferences for the unmatched AFSC (like the utility matrix shown earlier). Let's discuss cadet preferences first. There are multiple csv files containing information on cadet and AFSC preferences.

Cadet Preferences For cadets, there is the "utility" matrix I depicted earlier which is contained in "Random_1 Cadets Utility.csv":

Cadet	R1	R2	R3	R4
0	0.79	0.07	1	0.71
1	0.37	0.07	1	0.27
2	0.25	1	0.74	0.96
3	1	0.73	0.88	0.82
4	1	0.25	0.53	0.58
5	0.27	0.43	1	0.2
6	0.04	1	0.46	0.17
7	0.27	0.23	1	0.91
8	0.29	0.54	0.31	1
9	0.07	1	0.26	0.98
10	0.53	1	0.19	0.5
11	0.7	0.79	1	0.5
12	1	0.83	0.19	0.99
13	0.36	0.48	1	0.22
14	0.49	0.23	1	0.7
15	0.44	0.22	1	0
16	0.21	1	0.6	0.7
17	0.01	1	0.75	0.64
18	0.64	1	0.23	0.14
19	0.72	0.35	0.22	1

NOTE: Your numbers will be different if you're following along with this tutorial in your own jupyter notebook! (Data was randomly generated!). As you can see, the data is the same as was printed from "instance.parameters['utility']" above. The picture will not match your output, however, if you regenerate this data. If I want to access Cadet 4's utility for the AFSC at index 2 (R3), I can do so like this:

[34]:
```
instance.parameters['utility'][4, 2]
```

[34]:
```
0.53
```

Remember, python index starts at 0! This "utility" matrix is meant to represent the cadet's reported utility for the AFSC they receive. Here's a little history on this real problem:

Up until FY24, cadets were allowed to express 6 preferences for NRL AFSCs and assign utility

values to each. This was the extent of their input to the process, and the optimization model just used those utility values. Ties are allowed and are regularly provided by cadets by expressing multiple 100% utilities for their top however many choices. Often 0s were also expressed signaling the cadets lack of desire for a given AFSC (even within their top 6).

For the FY24 class, cadets rank ordered all 47 AFSCs (I know- yikes!) and were allowed to express utility values on their top 10 choices (same rules as before). This essentially creates two separate matrices: 'c_pref_matrix' and 'utility'. "Random_1 Cadets Preferences.csv" contains this preference matrix:

[35]:
```
instance.parameters['c_pref_matrix']
```

[35]:
```
array([[2, 4, 1, 3],
       [2, 4, 1, 3],
       [4, 1, 3, 2],
       [1, 4, 2, 3],
       [1, 4, 3, 2],
       [3, 2, 1, 4],
       [4, 1, 2, 3],
       [3, 4, 1, 2],
       [4, 2, 3, 1],
       [4, 1, 3, 2],
       [2, 1, 4, 3],
       [3, 2, 1, 4],
       [1, 2, 3, 0],
       [3, 2, 1, 0],
       [2, 3, 1, 0],
       [2, 3, 1, 4],
       [4, 1, 3, 2],
       [4, 1, 2, 3],
       [2, 1, 3, 4],
       [1, 2, 3, 0]])
```

NOTE: this matrix takes on the same format as before where the rows are the cadets and the columns are the AFSCs (NOT the "choice" of the cadets; the matrix cell values are the rank that the cadet put on that AFSC column). Cadet 0, for example, ranks AFSC "R3" first followed by "R1" then "R4" then "R2". The "0"s represent an AFSC that is not on a cadet's preference list. I will touch on the concept of "eligibility" later, but this effectively means that this cadet cannot be matched to this particular AFSC. Once we have the "c_pref_matrix", we can then convert it to an ordered list of AFSCs for each cadet. I have that piece as a dictionary where the keys are the cadets and the values lists of AFSC indices in order of the cadet's ranking of them:

[36]:
```
instance.parameters['cadet_preferences']
```

[36]:
```
{0: array([2, 0, 3, 1]),
 1: array([2, 0, 3, 1]),
 2: array([1, 3, 2, 0]),
 3: array([0, 2, 3, 1]),
 4: array([0, 3, 2, 1]),
```

```
 5: array([2, 1, 0, 3]),
 6: array([1, 2, 3, 0]),
 7: array([2, 3, 0, 1]),
 8: array([3, 1, 2, 0]),
 9: array([1, 3, 2, 0]),
 10: array([1, 0, 3, 2]),
 11: array([2, 1, 0, 3]),
 12: array([0, 1, 2]),
 13: array([2, 1, 0]),
 14: array([2, 0, 1]),
 15: array([2, 0, 1, 3]),
 16: array([1, 3, 2, 0]),
 17: array([1, 2, 3, 0]),
 18: array([1, 0, 2, 3]),
 19: array([0, 1, 2])}
```

Cadet 0's preferences in order with AFSC names:

[37]:
```
cadet_0_afsc_indices = instance.parameters['cadet_preferences'][0]

# Ordered list of AFSC names for cadet 0
instance.parameters['afscs'][cadet_0_afsc_indices]
```

[37]:
```
array(['R3', 'R1', 'R4', 'R2'], dtype=object)
```

So, for FY24 since I had a "utility" matrix only containing information on the cadet's top 10 choices and also a "c_pref_matrix" with the ordinal rankings the cadets gave, I had to reconcile the two to represent the most accurate amount of information. I couldn't just use one or the other since both had valuable information. I combine them by converting ordinal rankings (1, 2, 3, 4, 5) to a continuous 1 -> 0 scale (1, 0.8, 0.6, 0.4, 0.2). I then average this matrix with the cadet-provided utility matrix to create a new matrix I conveniently name "cadet_utility". This is located in "Random_1 Cadets Utility (Final).csv".

[38]:
```
instance.parameters['cadet_utility']
```

[38]:
```
array([[0.77  , 0.16  , 1.    , 0.605 ],
       [0.56  , 0.16  , 1.    , 0.385 ],
       [0.25  , 1.    , 0.62  , 0.855 ],
       [1.    , 0.49  , 0.815 , 0.66  ],
       [1.    , 0.25  , 0.515 , 0.665 ],
       [0.385 , 0.59  , 1.    , 0.225 ],
       [0.145 , 1.    , 0.605 , 0.335 ],
       [0.385 , 0.24  , 1.    , 0.83  ],
       [0.27  , 0.645 , 0.405 , 1.    ],
       [0.16  , 1.    , 0.38  , 0.865 ],
       [0.64  , 1.    , 0.22  , 0.5   ],
       [0.6   , 0.77  , 1.    , 0.375 ],
       [1.    , 0.7483, 0.2617, 0.    ],
```

```
[0.3467, 0.5733, 1.    , 0.    ],
[0.5783, 0.2817, 1.    , 0.    ],
[0.595 , 0.36  , 1.    , 0.125 ],
[0.23  , 1.    , 0.55  , 0.725 ],
[0.13  , 1.    , 0.75  , 0.57  ],
[0.695 , 1.    , 0.365 , 0.195 ],
[0.86  , 0.5083, 0.2767, 0.    ]])
```

Based on the method of collecting cadet preference data for FY24 I didn't want to assume cadets were really indifferent between their top 2 choices if they provided a utility of "1" for both. They still chose to sort them somehow, and so by taking the average of the two matrices I ensure a strictly decreasing utility function for each cadet.

AFSC Preferences In FY24, AFSCs have preferences on cadets now too! This, I do believe, is here to stay so it's worth discussing here. For the Non-Rated Line AFSCs, we actually met with all of the Career Field Managers (CFMs) to discuss what was important to them in their officers. They provided their input and "1-N" lists were created for each of their respective AFSCs. For Rated, each Source of Commissioning (SOC) acted as the CFM and their specific order of merit (OM) lists were used as the 1-Ns. I will touch on the Rated OM data momentarily. The Space Force did a similar thing for their AFSCs. This is all captured in the "Random_1 AFSCs Preferences.csv" file in a very similar manner as the "Cadets Preferences" version. Now, each column contains an AFSC's ranking for each cadet:

[39]:
```
instance.parameters['a_pref_matrix']
```

[39]:
```
array([[ 6, 18, 12,  7],
       [20, 20, 16, 15],
       [19,  9,  9,  8],
       [ 3,  7,  3,  3],
       [ 1,  6, 10,  2],
       [ 9, 12, 11, 10],
       [16, 14, 14, 16],
       [13, 19,  7,  9],
       [11, 16, 19,  6],
       [ 7,  1,  4,  1],
       [18, 10, 15, 11],
       [ 4,  3,  6,  5],
       [ 2,  4,  8,  0],
       [10,  5,  1,  0],
       [12, 15,  2,  0],
       [14, 17, 13, 13],
       [15,  2,  5,  4],
       [17, 13, 18, 12],
       [ 8,  8, 20, 14],
       [ 5, 11, 17,  0]])
```

Again, 0s represent cadets that are not on the AFSC's list. In the exact same manner I mentioned previously with converting cadet ordinal rankings (1, 2, 3, 4, 5) to a continuous scale (1, 0.8, 0.6,

0.4, 0.2), we do that with AFSCs to get the "afsc_utility" matrix located in "Random_1 AFSCs Utility.csv":

[40]:
```
instance.parameters['afsc_utility']
```

[40]:
```
array([[0.75  , 0.15  , 0.45  , 0.625 ],
       [0.05  , 0.05  , 0.25  , 0.125 ],
       [0.1   , 0.6   , 0.6   , 0.5625],
       [0.9   , 0.7   , 0.9   , 0.875 ],
       [1.    , 0.75  , 0.55  , 0.9375],
       [0.6   , 0.45  , 0.5   , 0.4375],
       [0.25  , 0.35  , 0.35  , 0.0625],
       [0.4   , 0.1   , 0.7   , 0.5   ],
       [0.5   , 0.25  , 0.1   , 0.6875],
       [0.7   , 1.    , 0.85  , 1.    ],
       [0.15  , 0.55  , 0.3   , 0.375 ],
       [0.85  , 0.9   , 0.75  , 0.75  ],
       [0.95  , 0.85  , 0.65  , 0.    ],
       [0.55  , 0.8   , 1.    , 0.    ],
       [0.45  , 0.3   , 0.95  , 0.    ],
       [0.35  , 0.2   , 0.4   , 0.25  ],
       [0.3   , 0.95  , 0.8   , 0.8125],
       [0.2   , 0.4   , 0.15  , 0.3125],
       [0.65  , 0.65  , 0.05  , 0.1875],
       [0.8   , 0.5   , 0.2   , 0.    ]])
```

Also in the same manner as cadets, we have a separate dictionary of ordered cadets for each AFSC:

[41]:
```
instance.parameters['afsc_preferences']
```

[41]:
```
{0: array([ 4, 12,  3, 11, 19,  0,  9, 18,  5, 13,  8, 14,  7, 15, 16,  6, 17,
        10,  2,  1]),
 1: array([ 9, 16, 11, 12, 13,  4,  3, 18,  2, 10, 19,  5, 17,  6, 14,  8, 15,
         0,  7,  1]),
 2: array([13, 14,  3,  9, 16, 11,  7, 12,  2,  4,  5,  0, 15,  6, 10,  1, 19,
        17,  8, 18]),
 3: array([ 9,  4,  3, 16, 11,  8,  0,  2,  7,  5, 10, 17, 15, 18,  1,  6])}
```

In the above, the AFSC at index 0 (R1) has cadet 4 ranked #1 and cadet 1 ranked #20.

Just to confirm that these lists are the "sorted indices" of "a_pref_matrix", you can look at both the utilities and rankings of the cadets using "afsc_preferences" as the indices to sort on. We'll use AFSC 'R1' as an example:

[42]:
```
j = 0  # Index of "R1"
sorted_indices = instance.parameters['afsc_preferences'][j]
print('Ordered cadets:', sorted_indices)
print('Rankings on these cadets:', instance.
 ↪parameters['a_pref_matrix'][sorted_indices, j])
```

```
print('Utilities on these cadets:', instance.
  ↪parameters['afsc_utility'][sorted_indices, j])
```

```
Ordered cadets: [ 4 12  3 11 19  0  9 18  5 13  8 14  7 15 16  6 17 10  2  1]
Rankings on these cadets: [ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18
19 20]
Utilities on these cadets: [1.   0.95 0.9  0.85 0.8  0.75 0.7  0.65 0.6  0.55
0.5  0.45 0.4  0.35
 0.3  0.25 0.2  0.15 0.1  0.05]
```

There you have it as far as cadet and AFSC preferences go! I will soon discuss why eligibility is important as it directly relates to who is or is not on each of the lists. First, let's discuss the rated OM situation

Rated OM Currently, each SOC provides their own rated OM lists that we need to combine. The lists for ROTC and USAFA are located in "Random_1 ROTC Rated OM.csv" and "Random_1 USAFA Rated OM.csv", respectively.

ROTC	
Cadet	R4
0	0.6
3	1
8	0.8
10	0.4
18	0.2

USAFA	
Cadet	R4
1	0.18181818
2	0.63636364
4	0.90909091
5	0.45454545
6	0.09090909
7	0.54545455
9	1
11	0.72727273
15	0.27272727
16	0.81818182
17	0.36363636

NOTE: In this problem instance we have only one rated AFSC: "R4". One of the reasons I like using cadet indices as identifiers (rather than some other ID or name) is for this example right here. I can extract the indices of the rated cadets from both SOCs directly from these matrices:

[43]:
```
instance.parameters['Rated Cadets']
```

[43]:
```
{'rotc': array([ 0,  3,  8, 10, 18]),
 'usafa': array([ 1,  2,  4,  5,  6,  7,  9, 11, 15, 16, 17])}
```

Once I have that, I can look at whatever other features I want to see from these cadets. For example:

[44]:
```
# General Order of Merit
indices = instance.parameters['Rated Cadets']['rotc']
print('GOM', instance.parameters['merit'][indices])
```

```
# Preference of the rated ROTC cadets for the one rated AFSC (R4)
print('Cadet preference on R4', instance.parameters['c_pref_matrix'][indices,␣
↪3])
```

```
GOM [0.49154715 0.53686789 0.27906107 0.18714076 0.30188118]
Cadet preference on R4 [3 3 1 3 4]
```

Since both of these lists for R4 are relative to each SOC, we can combine them like we do with general OM. We convert to "percentiles" relative to the SOCs and then zipper them together. I have a method that does this: "construct_rated_preferences_from_om_by_soc". It takes these two matrices and then zippers them together where the final "product" is an updated "a_pref_matrix" and "afsc_preferences".

Let's demonstrate the "zippering" (again, this may not be accurate if you're re-running all of my code since R4 may not be a rated AFSC in your instance!):

[45]:
```
j = 3 # R4

# Alternating USAFA/ROTC cadets based on proportions of both SOCs in R4's list
sorted_indices = instance.parameters['afsc_preferences'][j]
print('Binary USAFA array:', instance.parameters['usafa'][sorted_indices])
print('Ordered Cadet List:', sorted_indices)
```

```
Binary USAFA array: [1 1 0 1 1 0 0 1 1 1 0 1 1 0 1 1]
Ordered Cadet List: [ 9  4  3 16 11  8  0  2  7  5 10 17 15 18  1  6]
```

5.5.5 Eligibility/Qualifications

One thing I've alluded to in several areas before is the concept of eligibility. Certain cadet/AFSC pairings cannot happen. Rated eligibility is determined by medical qualifications and volunteerism. Space Force eligibility is determined by degree qualifications and volunteerism. NRL AFSC eligibility is a combination of degree qualifications and the new CFM rankings via the AFSC preferences. The intent for them was to open the door to more people potentially being eligible for certain career fields based on factors beyond academic degrees.

The degree "qual" matrix outlines the tier of degree that each cadet has but it also signals which cadets are eligible or not for each of the AFSCs. For rated, everyone is a "P" unless you're not eligible (see below for "R4").

[46]:
```
# Degree Qualification matrix (AFOCD)
print(instance.parameters['qual'])
```

```
[['D1' 'P1' 'P2' 'P1']
 ['P2' 'P1' 'P2' 'P1']
 ['P2' 'P1' 'D1' 'P1']
 ['D1' 'P1' 'D1' 'P1']
 ['D1' 'P1' 'P2' 'P1']
 ['D1' 'P1' 'P2' 'P1']
 ['D1' 'P1' 'D1' 'P1']
 ['D1' 'P1' 'D1' 'P1']
 ['D1' 'P1' 'P2' 'P1']
```

```
 ['D1' 'P1' 'D1' 'P1']
 ['P2' 'P1' 'D1' 'P1']
 ['D1' 'P1' 'P2' 'P1']
 ['D1' 'P1' 'D1' 'I2']
 ['P2' 'P1' 'D1' 'I2']
 ['P2' 'P1' 'D1' 'I2']
 ['P2' 'P1' 'P2' 'P1']
 ['P2' 'P1' 'D1' 'P1']
 ['D1' 'P1' 'P2' 'P1']
 ['D1' 'P1' 'P2' 'P1']
 ['D1' 'P1' 'P2' 'I2']]
```

[47]:
```
# Embedded eligibility matrix
print(instance.parameters['eligible'])
```

```
[[1 1 1 1]
 [1 1 1 1]
 [1 1 1 1]
 [1 1 1 1]
 [1 1 1 1]
 [1 1 1 1]
 [1 1 1 1]
 [1 1 1 1]
 [1 1 1 1]
 [1 1 1 1]
 [1 1 1 1]
 [1 1 1 1]
 [1 1 1 0]
 [1 1 1 0]
 [1 1 1 0]
 [1 1 1 1]
 [1 1 1 1]
 [1 1 1 1]
 [1 1 1 1]
 [1 1 1 0]]
```

In the age of wanting to try matching algorithms, preferences on both sides (cadets and AFSCs) must agree. This means that if you're on one AFSC's preference list, that AFSC must be on your preference list too. Essentially, this creates three separate eligibility sources (cadet preferences, afsc preferences, and the qual matrix). All three of these need to match up. This is why I have one method of CadetCareerProblem that "ensures" this is true: instance.remove_ineligible_choices(). This is a fairly aggressive approach since all it does is check if you're ineligible according to one source, and if you are it removes you from all other sources to force ineligibility. If you're doing this for real data, make sure you know what you're doing! Here's the code I have in the "instance.fix_generated_data()" method when I remove these choices to show what you need to do afterwards to get the data looking right:

[48]:
```
# Removes ineligible cadets from all 3 matrices: degree qualifications, cadet␣
 ↪preferences, AFSC preferences
instance.remove_ineligible_choices()

# Take the preferences dictionaries and update the matrices from them (using␣
 ↪cadet/AFSC indices)
instance.update_preference_matrices()  # 1, 2, 4, 6, 7 -> 1, 2, 3, 4, 5␣
 ↪(preference lists need to omit gaps)

# Convert AFSC preferences to percentiles (0 to 1)
instance.convert_afsc_preferences_to_percentiles()  # 1, 2, 3, 4, 5 -> 1, 0.8,␣
 ↪0.6, 0.4, 0.2

# The "cadet columns" are located in Cadets.csv and contain the utilities/
 ↪preferences in order of preference
instance.update_cadet_columns_from_matrices()  # We haven't touched␣
 ↪"c_preferences" and "c_utilities" until now
```

```
Removing ineligible cadets based on any of the three eligibility sources
(c_pref_matrix, a_pref_matrix, qual)…
0 total adjustments.
Updating cadet preference matrices from the preference dictionaries. ie. 1, 2,
4, 6, 7 -> 1, 2, 3, 4, 5 (preference lists need to omit gaps)
Converting AFSC preferences (a_pref_matrix) into percentiles (afsc_utility on
AFSCs Utility.csv)…
Updating cadet columns (Cadets.csv…c_utilities, c_preferences) from the
preference matrix (c_pref_matrix)…
```

No changes are made because I've already done this earlier!

5.5.6 Note on parameters

To drive home the idea that my "parameters" dictionary is an attribute of the problem instance I've been writing "instance.parameters" up until this point. In most of my functions within afccp, however, I convert "instance.parameters" to "p" for sake of typing less which I highly encourage you to do when writing your own functions.

[49]:
```
# Shorthand example
p = instance.parameters
p['afscs']
```

[49]:
```
array(['R1', 'R2', 'R3', 'R4', '*'], dtype=object)
```

[50]:
```
# "Accessions group" for each of the Four AFSCs
p['acc_grp']
```

[50]:
```
array(['NRL', 'NRL', 'USSF', 'Rated'], dtype=object)
```

There are plenty of other parameters attached to this dictionary too that you can explore. I need

to write a better documentation file on them but for now you can view them in the keys of the parameter dictionary.

[51]:
```
# Parameter keys
p.keys()
```

[51]:
```
dict_keys(['Qual Type', 'afscs', 'acc_grp', 'usafa_quota', 'rotc_quota', 'pgl',
'quota_e', 'quota_d', 'quota_min', 'quota_max', 'M', 'Deg Tiers', 'cadets',
'assigned', 'usafa', 'male', 'minority', 'merit', 'merit_all', 'N',
'ineligible', 'eligible', 'tier 1', 'tier 2', 'tier 3', 'tier 4', 'mandatory',
'desired', 'permitted', 'exception', 't_count', 't_proportion', 't_leq',
't_geq', 't_eq', 't_mandatory', 't_desired', 't_permitted', 'qual', 'P',
'num_util', 'c_preferences', 'c_utilities', 'utility', 'cadet_utility',
'c_pref_matrix', 'afsc_utility', 'a_pref_matrix', 'usafa_cadets',
'rr_interest_matrix', 'rr_om_matrix', 'rr_om_cadets', 'ur_om_matrix',
'ur_om_cadets', 'I', 'J', 'J^E', 'I^E', 'num_eligible', 'I^Choice', 'Choice
Count', 'I^D', 'usafa_proportion', 'Deg Tier Values', 'male_proportion',
'minority_proportion', 'sum_merit', 'rotc_cadets', 'J^Fixed', 'J^Reserved',
'cadet_preferences', 'num_cadet_choices', 'afsc_preferences', 'afscs_acc_grp',
'J^Rated', 'J^USSF', 'J^NRL', 'ussf_usafa_pgl', 'ussf_rotc_pgl', 'J^USAF',
'Rated Cadets', 'Rated Cadet Index Dict', 'Rated Choices', 'Num Rated Choices',
'J^P', 'I^P', 'I^USAFA', 'I^ROTC', 'I^Male', 'I^Female'])
```

5.6 "value_parameters"

The value parameters contain all the data on the weights, values, and constraints that the analyst controls on the problem. Eventually, however, I hope that the other stakeholders will control their pieces of this puzzle (ie. their weights/values for the aspects of this problem that affect them). The more data we have on what everyone cares about the more we know what to look for, and the more we can provide sensitivity analysis on to show why the solution is the way it is.

5.6.1 Defaults

The first thing I want to talk about here is the "default value parameters". I have a method in CadetCareerProblem to generate value parameters which is made exclusively for simulated data. Since that is the instance we are working with, we have already done that (it happens in "instance.fix_generated_data()"). This set of value parameters exists in our "vp_dict" but we haven't actually activated it yet. My code works with the idea that you could have different sets of things you care about, and when you solve the model with one set of value parameters you could solve it again with a different set. Let's see what this looks like:

[52]:
```
# List of sets of value parameters
print(instance.vp_dict.keys())

# Current "activated" value parameters (we haven't told CadetCareerProblem to␣
 ↪activate any yet!)
print('VP dictionary:', instance.value_parameters, 'VP Name:', instance.vp_name)
```

```
dict_keys(['VP'])
VP dictionary: None VP Name: None
```

We need to activate this set of value parameters, which we can do like this:

[53]:
```
instance.set_value_parameters()  # Defaults to grabbing the first set in the list

print('VP Name:', instance.vp_name)
```

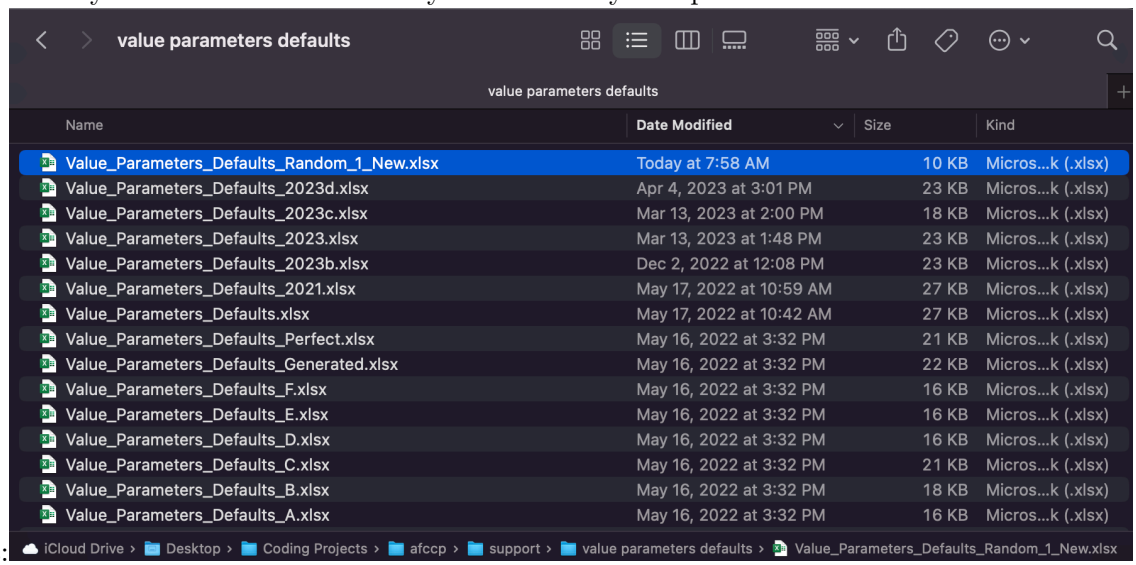
```
VP Name: VP
```

One of the things we can do is export the current set of value parameters as "defaults" back to excel. That is what I do here:

[54]:
```
instance.export_value_parameters_as_defaults()
```

```
Exporting value parameters as defaults to excel…
```

This set of value parameters nows exists in it's "default" form in your afccp/support/value parameters defaults/ folder! I add a "New" to the end of the name purely as a way to ensure you don't unintentionally overwrite your previous set of default value parameters:



Simply go in and manually change the name to "Value_Parameters_Defaults_Random_1.xlsx" (again, the "New" thing is because I don't want you to accidentally overwrite the real one if applicable!). Once you've done that, we can import this set of value parameters as "defaults" rather than generating random ones like we did at the beginning for this simulated dataset!

[55]:
```
v = instance.import_default_value_parameters()  # I add the "v = " because there's a lot of output otherwise
```

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

There are two reasons I'm having you import the value parameters as defaults above: to show you how can initialize a set of value parameters for an instance from excel (rather than generating

random ones), and also to show that I have a nifty function that checks if a new set of value parameters is really "new". What I mean by that is if you acquire a new set of value parameters by importing defaults, ideally it should only be added as a new one if it really is a unique set of parameters. You already had a set of value parameters "VP", and now you've just imported a new one so you should have "VP2". However, you'll still only see "VP" in your list because the two were identical:

[56]:
```
# Only one set of value parameters found for this instance
instance.vp_dict.keys()
```

[56]:
```
dict_keys(['VP'])
```

Now that we have our value parameters imported, I want to take a moment to describe the various components. I'm going to do this by using the "Value_Parameters_Defaults" excel file as a reference. Remember, these are the "defaults" that get imported for a particular problem instance that then turn into the actual set of value parameters used. I will do my best to explain the differences and where they're both located. Let's start by importing each dataframe from that file to convey these ideas:

[57]:
```
import openpyxl

# File path
filepath = dir_path + "support/value parameters defaults/
 ↪Value_Parameters_Defaults_Random_1.xlsx"

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

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

5.6.2 "Overall" value parameters

The first dataframe I'll show contains the information for the "big toggles" on the value parameters:

[58]:
```
dfs["Overall Weights"]
```

[58]:

	Cadets Weight	AFSCs Weight	Cadets Min Value	AFSCs Min Value	Cadet Weight Function	AFSC Weight Function	USAFA-Constrained AFSCs	Cadets Top 3 Constraint	USSF OM
0	0.430326	0.569674	0	0	Curve_1	Curve_1	NaN	NaN	False

Once initialized for the "Random_1" problem instance, these highest level settings are stored in "Random_1 Value Parameters.csv" in the "Model Input" folder. This dataframe controls the overall settings for each set of value parameters. It's also what tells afccp the names of the different sets of value parameters as well. Right now, we only have one set ("VP").

[59]:
```
# Shorthand
p = instance.parameters
vp = instance.value_parameters

print("Current value parameter set name:", instance.vp_name)

# Overall weights on Cadets/AFSCs
print('\nCadets Overall Weight:', vp['cadets_overall_weight'])
print('AFSCs Overall Weight:', vp['afscs_overall_weight'])

# If we want to constrain the overall values on Cadets/AFSCs (we won't, but␣
 ↪it's here)
print('\nCadets Overall Minimum Value:', vp['cadets_overall_value_min'])
print('AFSCs Overall Minimum Value:', vp['afscs_overall_value_min'])
```

```
Current value parameter set name: VP

Cadets Overall Weight: 0.4303264995913427
AFSCs Overall Weight: 0.5696735004086573

Cadets Overall Minimum Value: 0
AFSCs Overall Minimum Value: 0
```

For the "individual" weight on each cadet relative to all other cadets (and vice versa for AFSCs), we use weight functions. For cadets, their weights are based on their order of merit. In my random set of data, the cadet weight function initialized is "Curve_1".

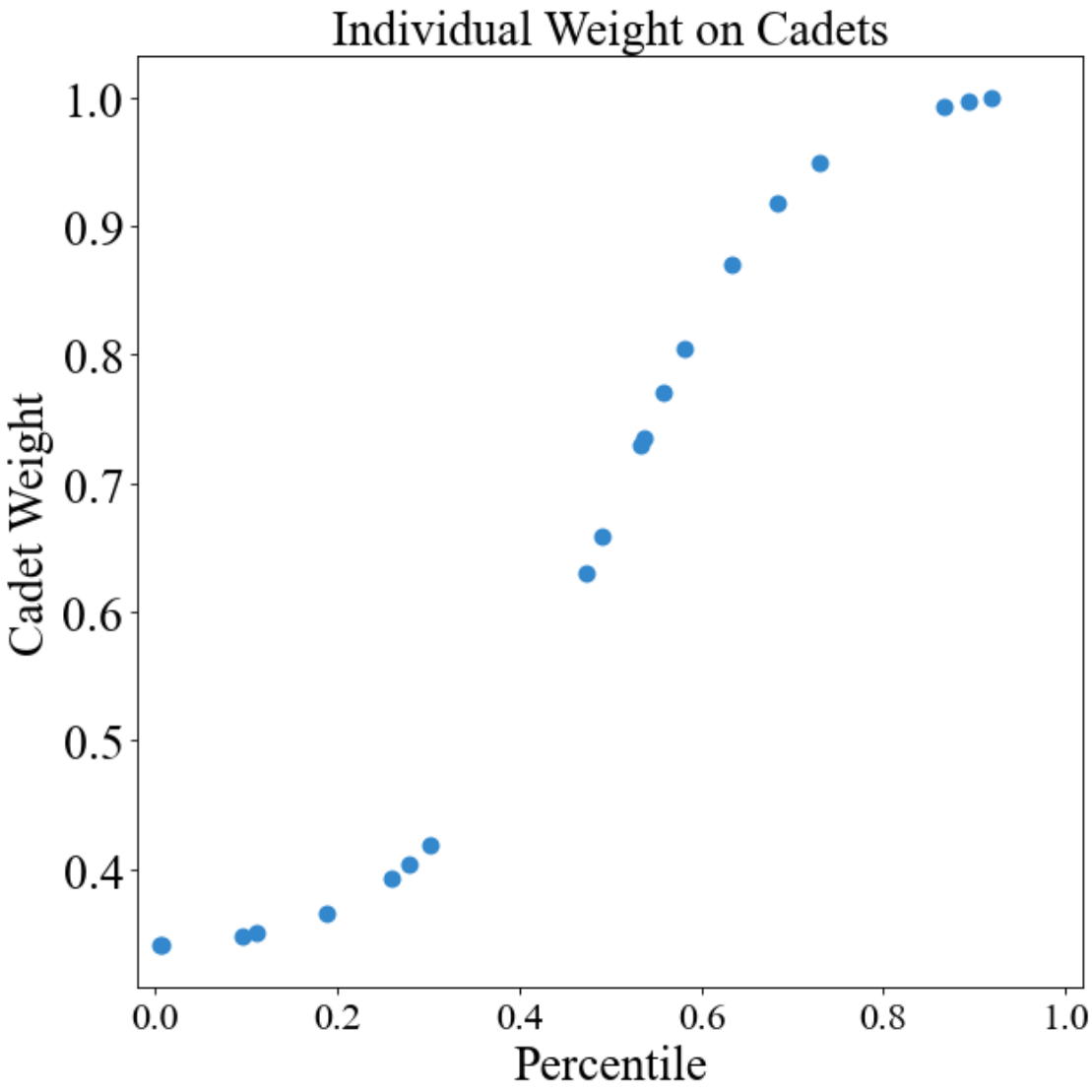
[60]:
```
# Cadet weight function
vp['cadet_weight_function']
```

[60]:
```
'Curve_1'
```

Here, cadet weight is a "sigmoid" function of their order of merit. I can illustrate the weight function by plotting cadet weight versus their OM:

[61]:
```
chart = instance.display_weight_function({"square_figsize": (8, 8), "dpi": 80})
```

```
Creating cadet weight chart…
```



Now, the percentiles for a real class will be uniformly distributed between 0 and 1. This is a fake class of 20 cadets and so they were randomly selected between 0 and 1 which is why the graph looks a little weird. The y-axis shows the "swing weights" for the cadets. Swing weights simply mean that they've been scaled so the biggest value is 1 and all other weights are relative to that one. "Local" weights, by contrast, sum to 1 collectively. I've printed out the differences below and you can see how I calculate them:

[62]:
```
print("Merit", np.around(p['merit'], 3))
print("\n'Local' Weight", np.around(vp['cadet_weight'], 3), "Local Weight Sum:
  ↪", np.around(np.sum(vp['cadet_weight']), 3))
print("\n'Swing' (Scaled) Weight", np.around(vp['cadet_weight'] / np.
  ↪max(vp['cadet_weight']), 3))
```

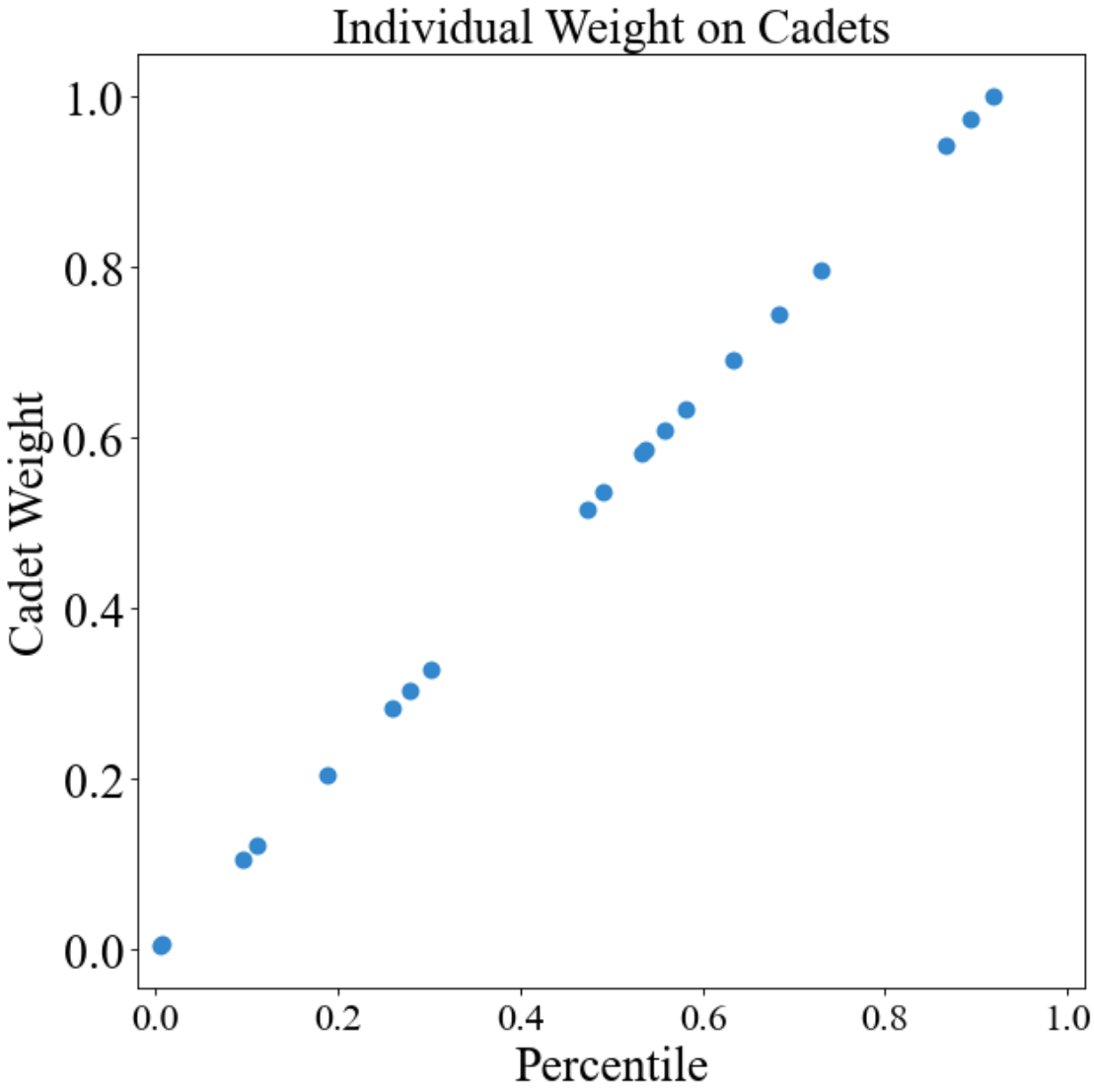
```
Merit [0.492 0.111 0.259 0.537 0.918 0.534 0.005 0.096 0.279 0.894 0.187 0.73
```

```
 0.684 0.866 0.559 0.474 0.582 0.007 0.302 0.634]

'Local' Weight [0.051 0.027 0.03  0.056 0.077 0.056 0.026 0.027 0.031 0.077
0.028 0.073
 0.071 0.076 0.059 0.048 0.062 0.026 0.032 0.067] Local Weight Sum: 1.0

'Swing' (Scaled) Weight [0.659 0.35  0.392 0.735 1.    0.73  0.341 0.348 0.403
0.997 0.365 0.949
 0.918 0.993 0.77  0.63  0.804 0.342 0.418 0.871]
```

We can also change the weight function through afccp if we want to.

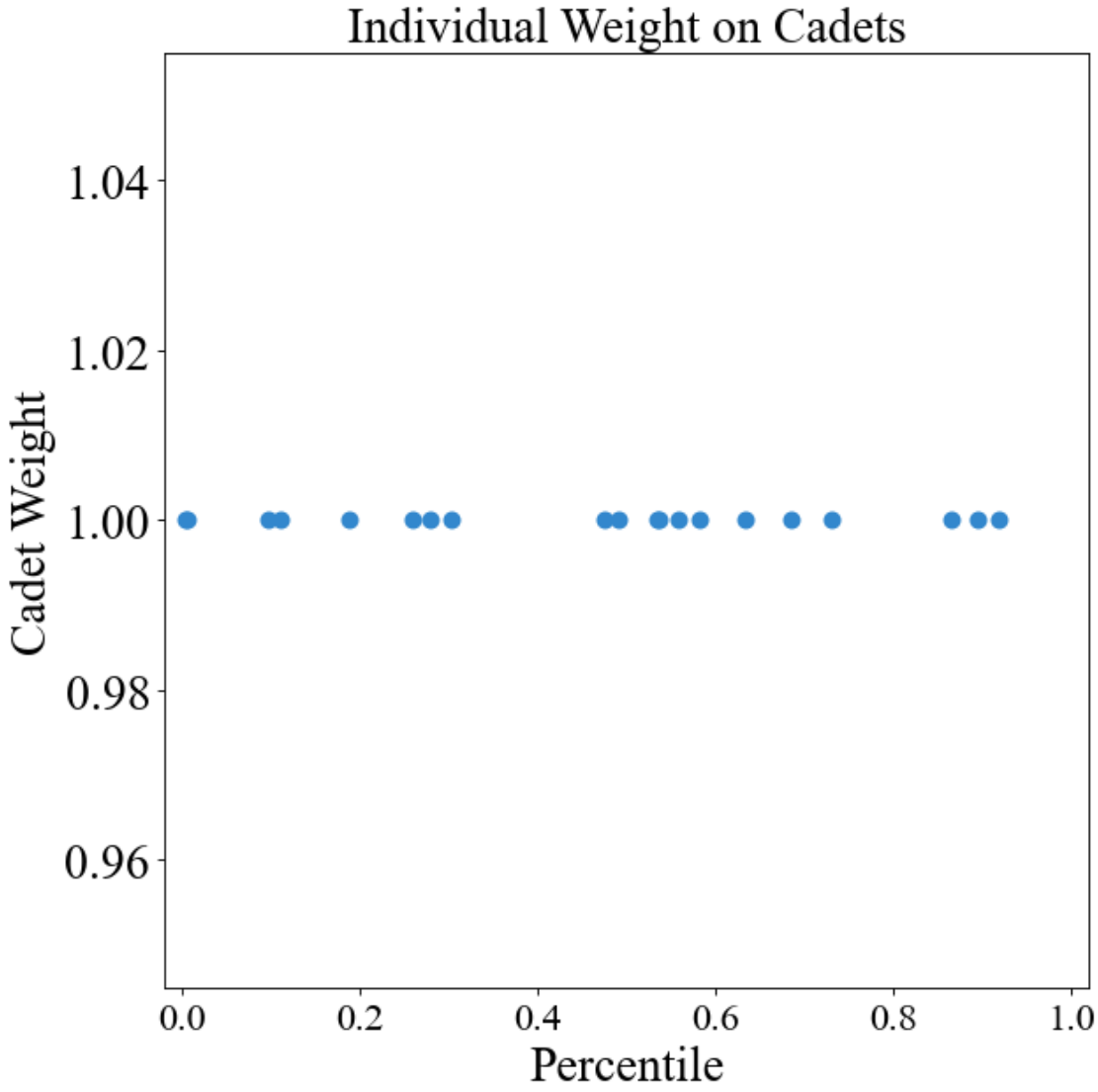
[63]:
```
# Linear function of OM (not very "forgiving" to low OM cadets)
instance.change_weight_function(cadets=True, function="Direct")
chart = instance.display_weight_function({"square_figsize": (8, 8), "dpi": 80})
```

```
Creating cadet weight chart…
```



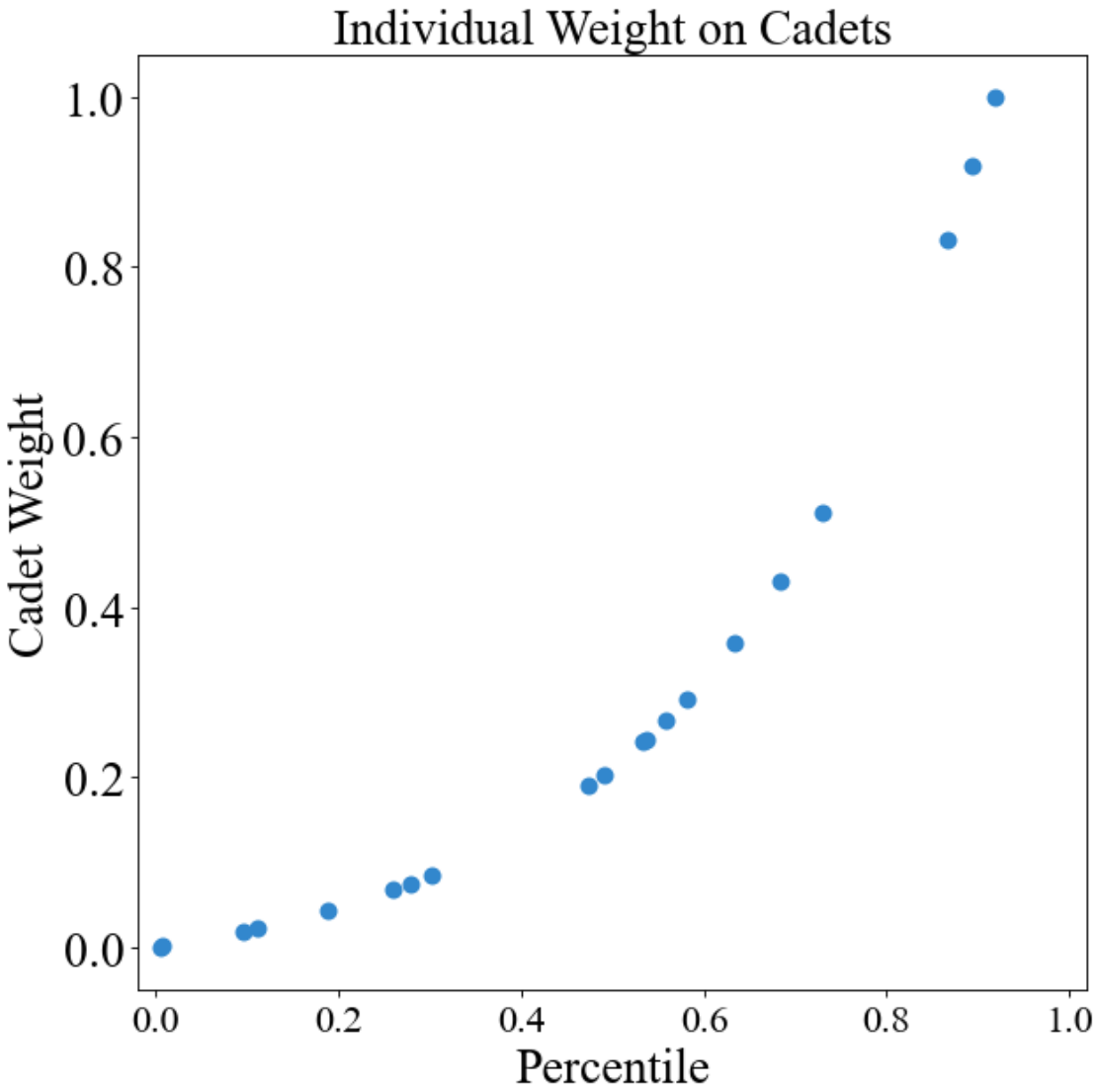
[64]:
```
# Cadets are equal no matter what their OM is
instance.change_weight_function(cadets=True, function="Equal")
chart = instance.display_weight_function({"square_figsize": (8, 8), "dpi": 80})
```

```
Creating cadet weight chart…
```



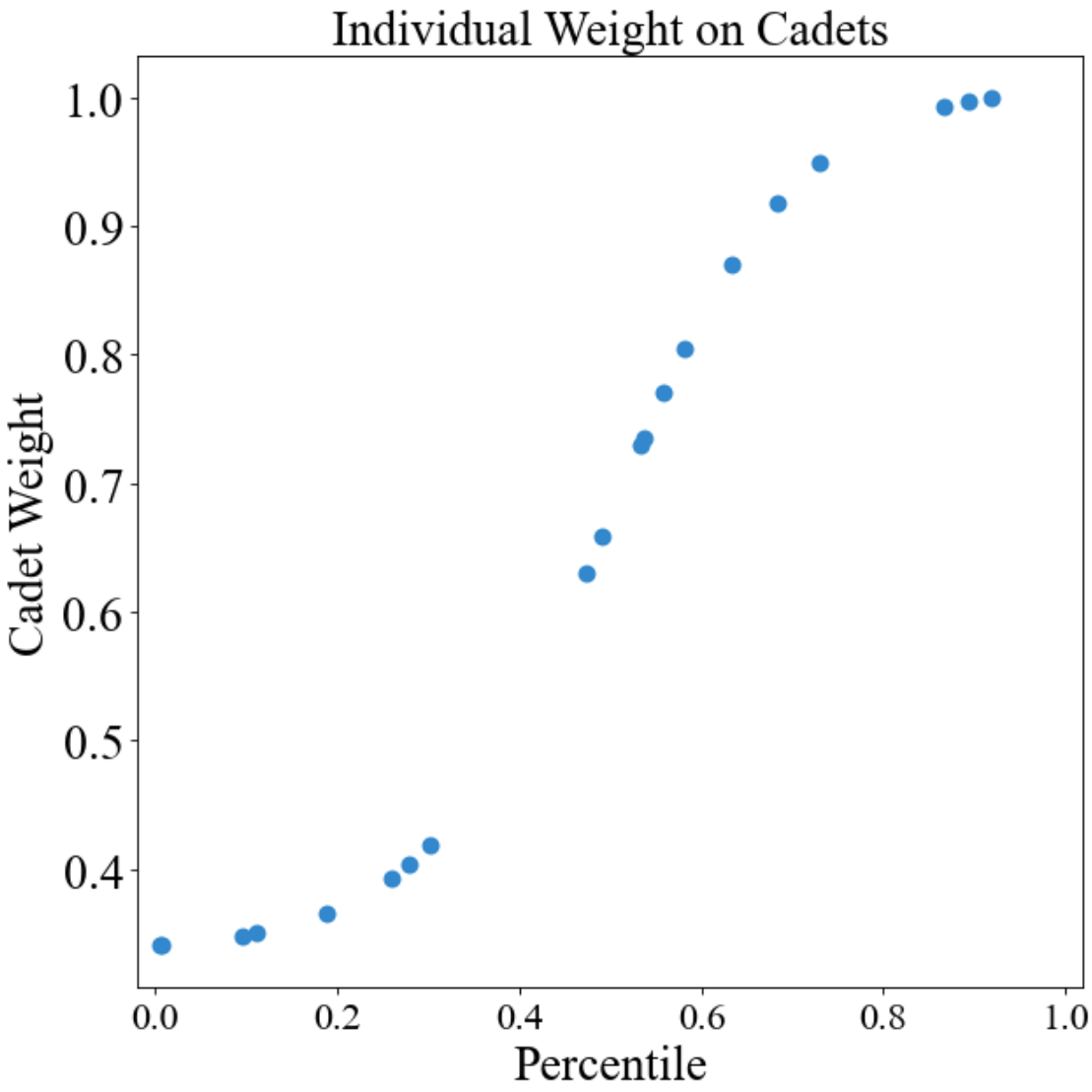
[65]:
```
# Exponential curve (not recommended since it puts heavy emphasis on top␣
↪performers)
instance.change_weight_function(cadets=True, function="Exponential")
chart = instance.display_weight_function({"square_figsize": (8, 8), "dpi": 80})
```

```
Creating cadet weight chart…
```



[66]:
```
# Sigmoid curve of OM (more forgiving in terms of differences between highest
  ↪and lowest rank)
instance.change_weight_function(cadets=True, function="Curve_1")
chart = instance.display_weight_function({"square_figsize": (8, 8), "dpi": 80})
```

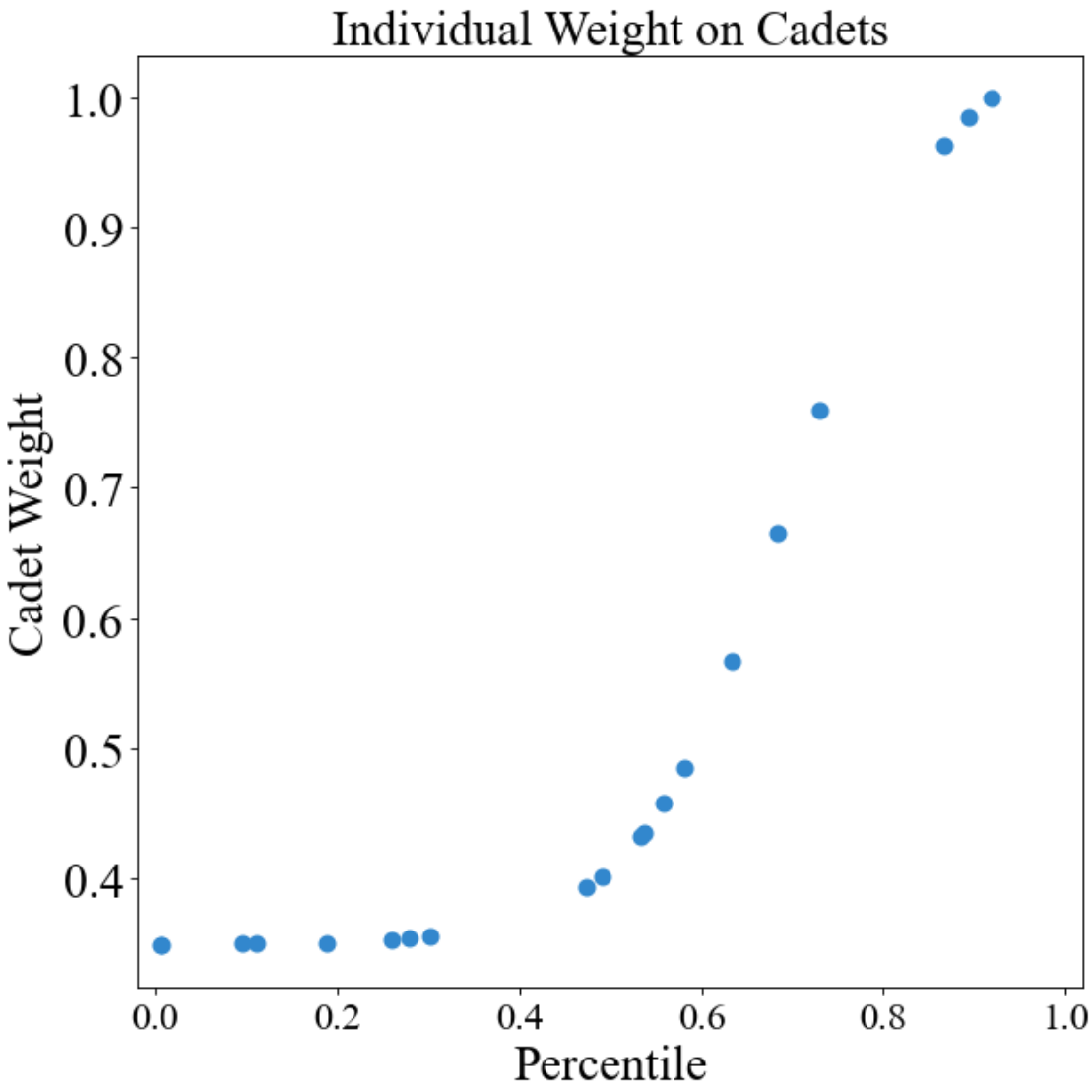
```
Creating cadet weight chart…
```



These curves are what I'd use since the top cadet is a little more than twice as "important" as the lowest cadet. On the other linear/exponential curves, the difference is quite drastic (100% to 0%)

[67]:
```
# Sigmoid curve of OM (very similar to previous one)
instance.change_weight_function(cadets=True, function="Curve_2")
chart = instance.display_weight_function({"square_figsize": (8, 8), "dpi": 80})
```

```
Creating cadet weight chart…
```



[68]:
```
# Change back to "Curve_1" weight function
instance.change_weight_function(cadets=True, function="Curve_1")
instance.value_parameters['cadet_weight_function']
```

[68]: 'Curve_1'

AFSC weights may be determined as a function of their size. Ideally, in the future it'd be some function of their size, difficulty to fill, manpower needs, and maybe more. I want a better method for determining those weights on the AFSCs.

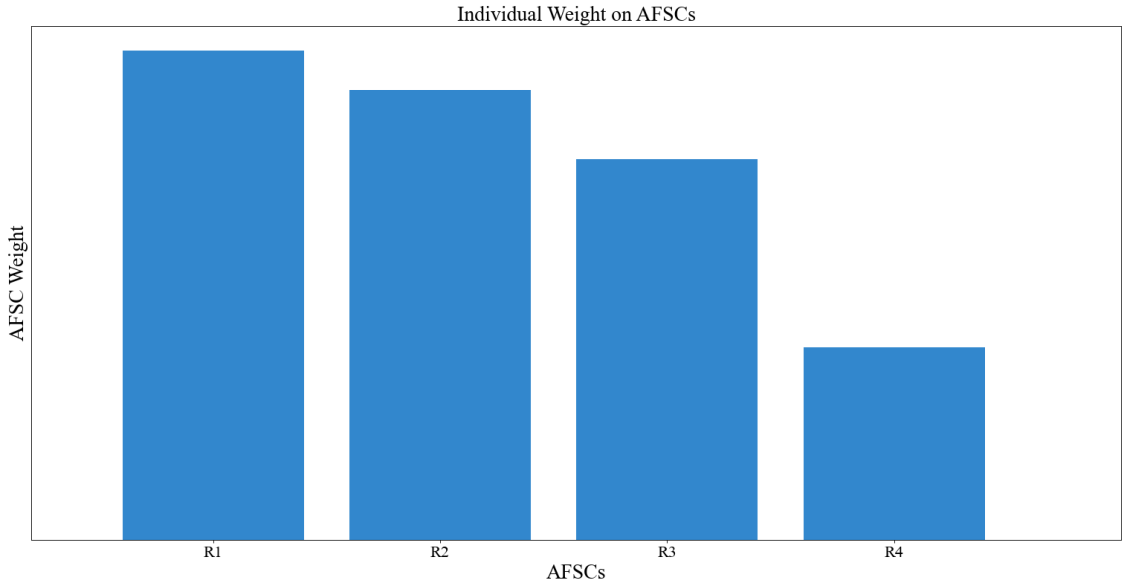
[69]:
```
# In my generated data, AFSCs are weighted similarly to the cadets (Curve_1)
vp['afsc_weight_function']
```

[69]: 'Curve_1'

Here is the AFSC weight chart. The AFSC weight chart is a bar chart since we can show relative importance pretty well with those kinds of charts.

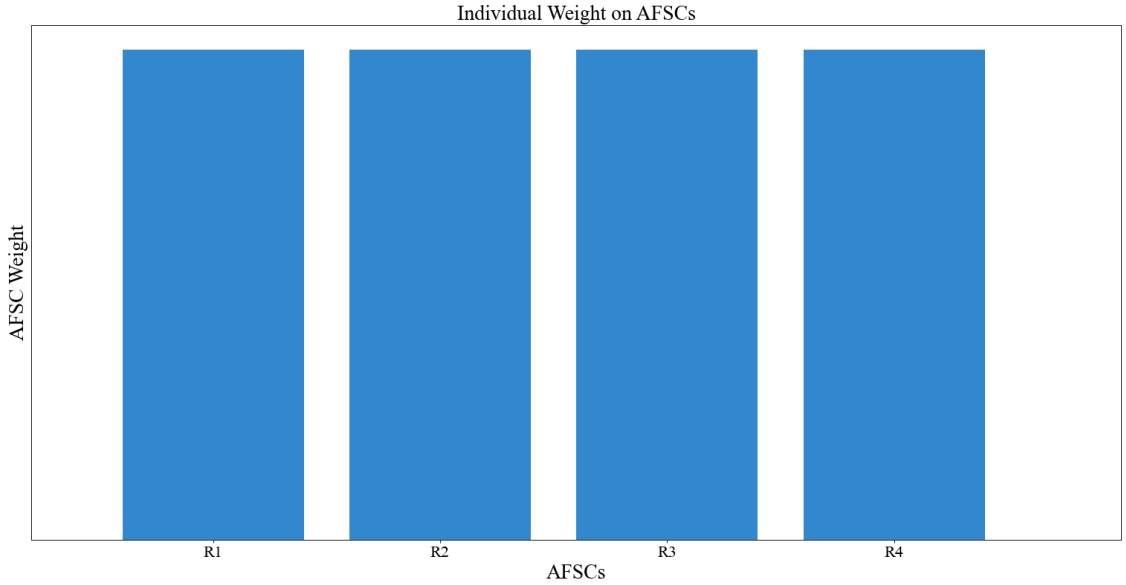
[70]:
```
chart = instance.display_weight_function({"dpi": 80, "cadets_graph": False,
                                          "skip_afscs": False})
```

```
Creating AFSC weight chart…
```



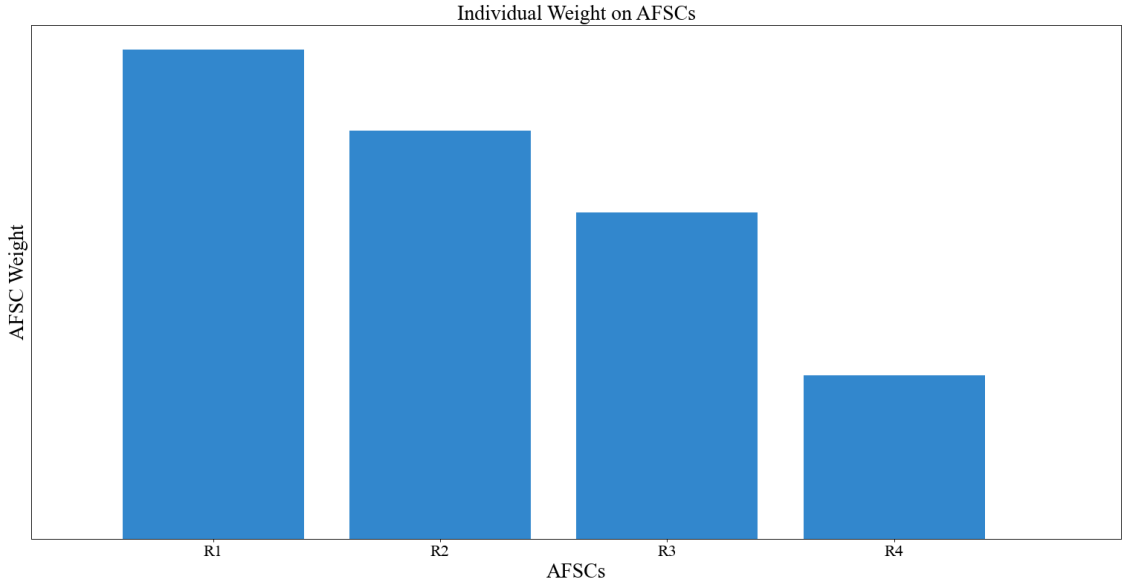
[71]:
```
# All AFSCs are equal
instance.change_weight_function(cadets=False, function="Equal")
chart = instance.display_weight_function({"dpi": 80, "cadets_graph": False,␣
 ↪"skip_afscs": False})
```

```
Creating AFSC weight chart…
```



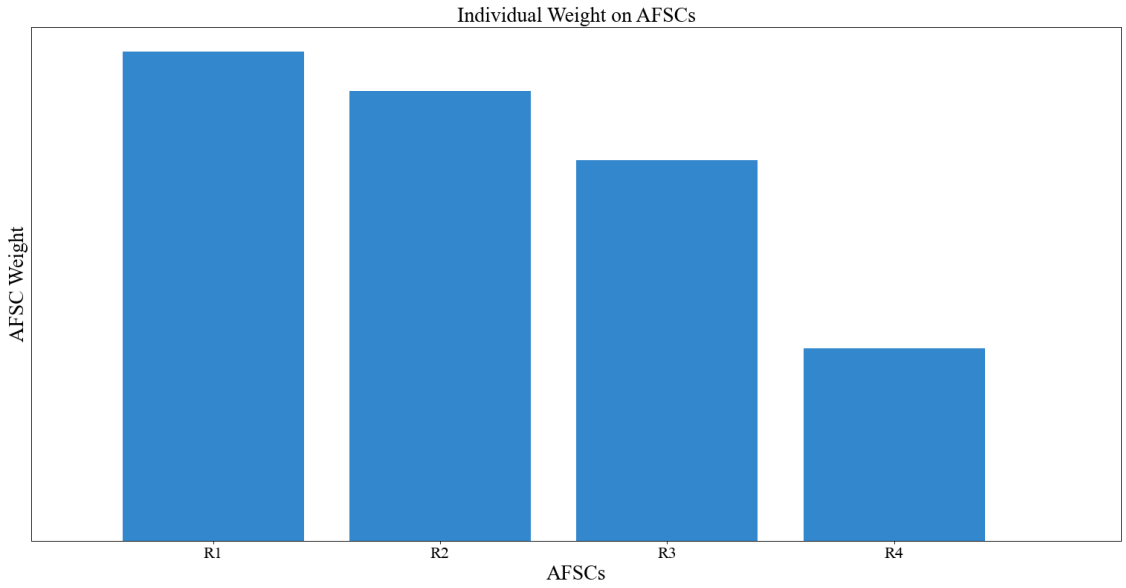
[72]:
```
# Weight based purely on size
instance.change_weight_function(cadets=False, function="Size")
chart = instance.display_weight_function({"dpi": 80, "cadets_graph": False,␣
↪"skip_afscs": False})
```

```
Creating AFSC weight chart…
```



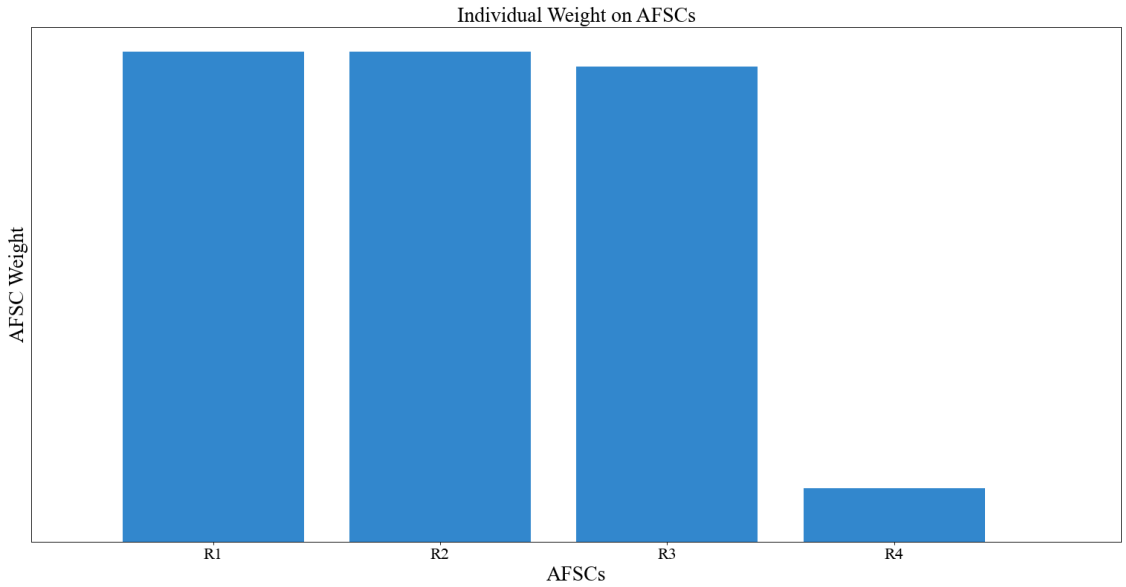
[73]:
```
# Slightly different function of size (function from generated data)
instance.change_weight_function(cadets=False, function="Curve_1")
chart = instance.display_weight_function({"dpi": 80, "cadets_graph": False,␣
 ↪"skip_afscs": False})
```

```
Creating AFSC weight chart…
```



[74]:
```
# Another function of size
instance.change_weight_function(cadets=False, function="Curve_2")
chart = instance.display_weight_function({"dpi": 80, "cadets_graph": False,␣
 ↪"skip_afscs": False})
```

```
Creating AFSC weight chart…
```



[75]:
```
# Change back to "Curve_1" weight function
instance.change_weight_function(cadets=False, function="Curve_1")
instance.value_parameters['afsc_weight_function']
```

[75]:
```
'Curve_1'
```

If we pass the function "Custom" for AFSC weights, we will pull from the predefined weights in the "AFSC Weights" excel sheet

[76]:
```
dfs["AFSC Weights"]
```

[76]:

	AFSC	AFSC Swing Weight	AFSC Min Value
0	R1	100.00	0
1	R2	91.89	0
2	R3	77.83	0
3	R4	39.34	0

Right now they are from the "Curve_1" function. If we want to constrain the AFSC values, we can do that using the "AFSC Min Value" column

[77]:
```
print("AFSC 'local' weight:", vp['afsc_weight'])  # Sum to 1!
print("AFSC minimum values:", vp['afsc_value_min'])
```

```
AFSC 'local' weight: [0.32355563 0.29733011 0.25181378 0.12730047]
AFSC minimum values: [0. 0. 0. 0.]
```

I'm going to print out the "overall weights" dataset again for reference since there are a lot of charts above and I don't want you to have to keep scrolling up!

[78]:
```
dfs["Overall Weights"]
```

[78]:
```
   Cadets Weight  AFSCs Weight  Cadets Min Value  AFSCs Min Value  \
0       0.430326      0.569674                 0                0

  Cadet Weight Function AFSC Weight Function  USAFA-Constrained AFSCs  \
0               Curve_1              Curve_1                      NaN

   Cadets Top 3 Constraint  USSF OM
0                      NaN    False
```

The "USAFA-Constrained AFSCs" component is here for when there was a restriction on the number of USAFA cadets that could go into the 4 Support AFSCs: 35P (Public Affairs), 38F (Force Support), 64P (Contracting), and 65F (Finance). There's a lot of backstory here but effective Mar '23 that restriction is no longer active. The option to constrain it is still here in case we ever want to run any kind of experiment with it. In a real class, we'd write "35P, 38F, 64P, 65F" there.

[79]:
```
vp['USAFA-Constrained AFSCs']  # Blank, but in the context of "Random_1" we␣
↪could write "R1, R2" for example
```

[79]:
```
''
```

Ignore the "Cadets Top 3 Constraint". The intent of this was for you to be able to constrain the top 10% of the class so they get one of their top 3 preferences. I believe the best way to do this is manually after you've run the model without that kind of constraint and then go back, filter on the people who aren't getting one of their top 3 from the top 10%, and then adjust accordingly and put those "utility values" in as constraints. This is something I discuss later on, but is meant only for AFPC/DSYA analysts to have to worry about for a real class year.

[80]:
```
vp['USSF OM']  # Whether or not we want to constrain average OM for the USSF to␣
↪be around 0.5.
```

[80]:
```
False
```

There are a few toggles when it comes to the USSF OM constraint that I will discuss in the "Solutions" section.

5.6.3 Model Controls ("mdl_p" side-bar)

I've mentioned the attribute "mdl_p" earlier on in this tutorial but haven't gone too much into detail on it. Essentially, this is my dictionary of all the various toggles and components used across afccp. Everything from genetic algorithm hyperparameters to the colors of various components of the visualizations. There's a lot there. I've actually been using them for the charts above everytime I pass in a dictionary as a parameter for the method I'm calling. If you recall the "instance.display_weight_function()" method was taking a dictionary including things like {"dpi": 80, "cadets_graph": False, "skip_afscs": False}. These control specific components used in some place within afccp. In that context, they're controls used in the weight function chart.

[81]:
```
# DPI (Dots per inch) of my charts
instance.mdl_p['dpi']
```

[81]: 80

I alluded to this towards the beginning of the tutorial, but essentially within "afccp.core.data.support" there is a function that initializes the many "hyperparameters" of afccp. "Hyperparameters" traditionally refer to the parameters that control the learning process of some algorithms and are probably not the best term to use for this since that's really only applicable to the genetic algorithm. "Controls" is probably a better word, since I've generalized this dictionary to control for a lot of different elements. When I say a lot, I mean it!

[82]:
```
# Number of keys in the "mdl_p" dictionary
print("Number of afccp 'controls':", len(instance.mdl_p.keys()))
```

```
Number of afccp 'controls': 217
```

If you scroll through "main.py" and look at the keyword arguments used you'll notice "p_dict={}" is quite common. What this does is allow you to change the default settings that are initialized for mdl_p. Using "mdl_p" as I do allows me to come up with a needed "control" for some function buried deep within afccp and not worry about passing it through the many layers of functions to get to where it needs to be. The instance object contains mdl_p as an attribute and so we just define it in the parameter initialization function of support.py and bam- we have it wherever we need it. It's also now something I can have a default setting for and potentially change using "p_dict". Here's an example:

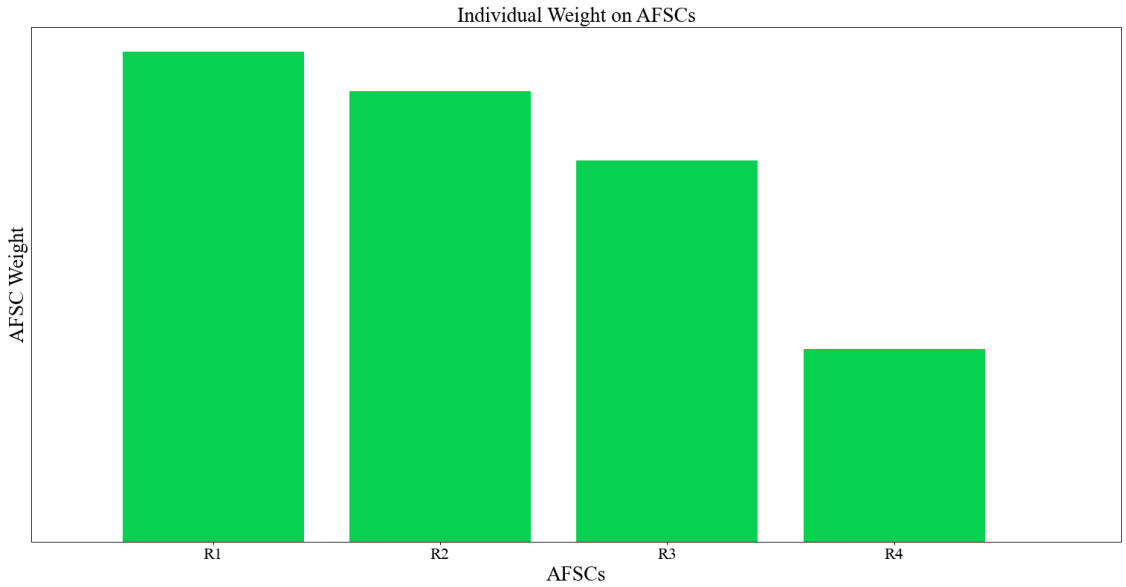
[83]:
```
# Default bar color- HEX codes are useful ways of selecting precise colors
print('Default bar color:', instance.mdl_p['bar_color'])  # (google "color␣
 ↪picker")
```

```
Default bar color: #3287cd
```

The color above is the light-ish shade of blue you've seen for the charts above. Let's produce the AFSC chart again after changing the 'bar_color' parameter.

[84]:
```
# Slightly different function of size
chart = instance.display_weight_function({"dpi": 80, "cadets_graph": False,␣
 ↪"skip_afscs": False,
                                          'bar_color': "#08d152"})  # Shade of␣
 ↪green
```

```
Creating AFSC weight chart…
```



As a reminder, in order to make use of "p_dict" as a means of passing a new value for one of the controls inside "mdl_p", you simply call the desired method and pass a dictionary ie. {"bar_color": "blue"} containing the keys that you want to change as the only argument.

5.6.4 AFSC Objectives

Before I dive deep into the components of the AFSC objectives, it's probably worthwhile to talk about what the objectives themselves are. Here they are printed out for you:

[85]:
```
vp['objectives']
```

[85]:
```
array(['Norm Score', 'Merit', 'USAFA Proportion', 'Combined Quota',
       'USAFA Quota', 'ROTC Quota', 'Utility', 'Male', 'Minority',
       'Tier 1', 'Tier 2', 'Tier 3', 'Tier 4'], dtype='<U16')
```

The "Norm Score" objective refers to the newly defined career field preference lists. Basically, career fields get to rank cadets in order of preference similar to how the cadets rank their AFSC choices. To evaluate how well we meet the needs of the AFSC according to their preferences, I came up with a normalized score idea. Imagine you have a set of ten cadets, ranked 1 to 10. If you are picking 3 cadets from that list, the best cadets you could select are the ones ranked 1, 2, 3. The worst are the cadets ranked 8, 9, 10. The former is a score of 1 and the latter is a score of 0. Whatever you ultimately pick is likely going to be somewhere in between, which is where the norm score comes in. Here is that example:

[86]:
```
import random

# Cadet rankings
num_cadets = 10  # 10 cadets in the above example
rankings = np.arange(num_cadets) + 1
```

```python
print('Rankings:', rankings)

# Picking "n" cadets
n = 3  # picking 3 for this example

# Selecting n cadets
print('\nBest Cadets:', rankings[:n])

# "Score" is the sum of these numbers
best_score = np.sum(rankings[:n])
print('Best Cadets Score:', best_score)

# Selecting n cadets
print('\nWorst Cadets:', rankings[num_cadets-n:])

# "Score" is the sum of these numbers
worst_score = np.sum(rankings[num_cadets-n:])
print('Worst Cadets Score:', worst_score)

# Pick a random set of n cadets
selected_cadets = random.sample(list(rankings), n)
print('\nRandomly selected cadets:', selected_cadets)

selected_score = np.sum(selected_cadets)
print('Random cadets score:', selected_score)

# "Norm Score" normalizes that "selected_score" on a 1 to 0 scale using the␣
 ↪best/worst scores
norm_score = 1 - (selected_score - best_score) / (worst_score - best_score)
print('\nNorm Score:', round(norm_score, 4))

# Everything described above is what is used in afccp
from afccp.core.solutions.handling import calculate_afsc_norm_score_general

# This function takes the rankings and selected rankings as arguments
norm_score_2 = calculate_afsc_norm_score_general(rankings, selected_cadets)
print('Norm Score (from afccp):', round(norm_score_2, 4))
```

```
Rankings: [ 1  2  3  4  5  6  7  8  9 10]

Best Cadets: [1 2 3]
Best Cadets Score: 6

Worst Cadets: [ 8  9 10]
Worst Cadets Score: 27

Randomly selected cadets: [7, 1, 8]
Random cadets score: 16
```

```
Norm Score: 0.5238
Norm Score (from afccp): 0.5238
```

The "Merit" objective is one that was used to fairly distribute "quality" cadets across the AFSCs. The idea is that no single "large" AFSC should be composed of entirely high or low performers. I've never liked this objective because it puts too much emphasis on defining quality for a career field purely on graduating order of merit. I believe that the career field preferences provide a much better way of defining quality cadets that is specific to each career field. It is no longer a zero-sum game, and it is theoretically possible (though highly improbable) that the rankings given could perfectly line up with the needs of the Air Force such that every single AFSC receives their top performers. Again, this won't ever happen, but we are now deviating from order of merit as the one-size-fits-all metric of quality.

[87]:
```
# Average order of merit of the class
print('Average OM:', np.mean(p['merit'])) # Should be about 0.5 for a real␣
 ↪class (random data will not)

# Proportion of USAFA cadets of the class
print('USAFA Proportion:', np.mean(p['usafa'])) # Closer to 1/3 for a real class
```

```
Average OM: 0.4574862875146219
USAFA Proportion: 0.65
```

In a very similar way that we want to keep average OM around 0.5 (or whatever the actual average is) for each of the large AFSCs, we also don't want any single large AFSC to be composed of entirely USAFA or ROTC cadets. We take the actual proportion of the cadets as the baseline and then shoot to be within +- 15% of that number. That is another AFSC objective that may or may not actually be that important. The idea now is that it should be left up to the career field manager to determine.

[88]:
```
# USAFA quota (for each AFSC)
print('USAFA Quota:', p['usafa_quota'])

# ROTC quota (for each AFSC)
print('ROTC Quota:', p['rotc_quota'])
```

```
USAFA Quota: [1. 1. 1. 0.]
ROTC Quota: [5. 4. 3. 2.]
```

If you recall the USAFA/ROTC quotas from earlier on, these numbers are fed into their appropriate objectives. Meeting the individual USAFA and ROTC quotas are two objectives that are separate from the USAFA proportion objective. They're doing similar things, but one is trying to balance the proportion of cadets assigned to be around some baseline while the others are simply trying to meet a quota and that's it. These objectives really only come into play now with the rated AFSCs since we need to keep the slots specific to each source of commissioning.

[89]:
```
# The "desired" number of cadets for a given AFSC
print('PGL Target (Combined SOC quotas):', p['pgl'])
print('\nDesired number:', p['quota_d'])
```

```
PGL Target (Combined SOC quotas): [6. 5. 4. 2.]

Desired number: [6. 7. 4. 2.]
```

The quota objective that we absolutely do care about is the "Combined Quota" objective which is used right now to meet the PGL. It currently provides the minimum number of cadets to classify and so as long as we meet each of the minimums then we are good! In the future, there's a lot more we should do with this objective to really hone in on the importance of assigning more or fewer cadets to a given AFSC (cross-collaboration with AFMAA/A1XD in the works).

[90]:
```
# Proportion of male cadets of the class
print('Male Proportion:', np.mean(p['male']))

# Proportion of minority cadets of the class
print('Minority Proportion:', np.mean(p['male']))
```

```
Male Proportion: 0.7
Minority Proportion: 0.7
```

The "Male" and "Minority" objectives do the exact same thing as the "USAFA Proportion" objective by balancing the proportions of these demographics around some baseline. Both of these objectives are "dormant" however and not actually used in any way because if the Air Force really wanted us to do this it would have to come down from the top as a written directive. We highly recommend against this as it would only hurt the cadets (from a preference perspective).

[91]:
```
# The AFOCD Tier objectives (from the "AFSCs.csv" I showed earlier)
p['Deg Tiers']
```

[91]:
```
array([['D > 0.54', 'P < 0.4599', '', ''],
       ['P = 1', '', '', ''],
       ['D > 0.12', 'P < 0.88', '', ''],
       ['P = 1', 'I = 0', '', '']], dtype='<U10')
```

There are generally up to four Air Force Officer Classification Directory (AFOCD) degree tiers per career field. Each degree tier has a target proportion and requirement level associated with it: Mandatory, Desired, or Permitted (M, D, P). Above, the columns correspond to degree tiers (1, 2, 3, 4) and the rows are the AFSCs. It just so happens that in this example we have 4 AFSCs and so it's worth clarifying! The format above provides a few pieces of information: the requirement level, target proportion, and the type of inequality specified (>, <, or =). So, for a degree tier format of "D > 0.54", we know the requirement is "Desired" and the AFSC wants at LEAST 54% of their accessions to have degrees in that tier. If you recall, the information on what tier everyone is placed in for each AFSC based on their degree is located in the "qual" matrix. There is a function in afccp.core.support called "cip_to_qual_tiers" that creates the qual matrix based on the cadets' degrees. This is an important function for the AFPC/DSYA analyst to maintain, and is irrelevant for random data since it's all fake anyway!

[92]:
```
# Qual matrix! This conveys requirement level (M, D, P), tier (1, 2, 3, 4) and␣
 ↪even eligibility ("I" is ineligible)
p['qual']  # Rows are cadets, columns are AFSCs!
```

[92]:
```
array([['D1', 'P1', 'P2', 'P1'],
       ['P2', 'P1', 'P2', 'P1'],
       ['P2', 'P1', 'D1', 'P1'],
       ['D1', 'P1', 'D1', 'P1'],
       ['D1', 'P1', 'P2', 'P1'],
       ['D1', 'P1', 'P2', 'P1'],
       ['D1', 'P1', 'D1', 'P1'],
       ['D1', 'P1', 'D1', 'P1'],
       ['D1', 'P1', 'P2', 'P1'],
       ['D1', 'P1', 'D1', 'P1'],
       ['P2', 'P1', 'D1', 'P1'],
       ['D1', 'P1', 'P2', 'P1'],
       ['D1', 'P1', 'D1', 'I2'],
       ['P2', 'P1', 'D1', 'I2'],
       ['P2', 'P1', 'D1', 'I2'],
       ['P2', 'P1', 'P2', 'P1'],
       ['P2', 'P1', 'D1', 'P1'],
       ['D1', 'P1', 'P2', 'P1'],
       ['D1', 'P1', 'P2', 'P1'],
       ['D1', 'P1', 'P2', 'I2']], dtype='<U2')
```

As you can imagine, the objectives for "Tier 1" -> "Tier 4" are there to meet the respective degree tier proportions!

Lastly, the "Utility" objective is simply to maximize cadet utility (happiness) and is measured by the average cadet utility of the cadets assigned. I have this in there so AFSCs can prioritize the preferences of their incoming cadets as well.

5.6.5 AFSC Objective Components

This section describes the various pieces of the AFSC objectives: their weights, targets, and constraints. The value functions are another component of the objectives but we'll cover them in their own section since there's a lot going on there! The first component of the AFSC objectives we'll discuss are the weights.

[93]:
```
dfs["AFSC Objective Weights"]
```

[93]:

	AFSC	Norm Score	Merit	USAFA Proportion	Combined Quota	USAFA Quota	\
0	R1	37.498000	21.675000	25.693001	100.000000	0	
1	R2	46.490000	10.415999	23.918000	100.000000	0	
2	R3	53.933000	5.622999	27.626999	100.000000	0	
3	R4	66.674001	80.508998	38.415000	70.876999	0	

	ROTC Quota	Utility	Male	Minority	Tier 1	Tier 2	Tier 3	\
0	0	0	18.103999	6.690001	80.814998	46.179999	0	
1	0	0	12.816000	16.089000	60.694001	0.000000	0	
2	0	0	20.343999	10.479999	76.497999	43.712999	0	
3	0	0	24.825998	35.272999	100.000000	0.000000	0	

```
   Tier 4
0       0
1       0
2       0
3       0
```

Here are the objective weights for each AFSC for each objective. Like the AFSC "individual" weights, these are swing weights that will be scaled for each AFSC so that they sum to 1. Many objectives are weighted at 0 which effectively removes them from consideration for a given AFSC. As mentioned previously, for a real class year the "Male" and "Minority" objectives will be zeros too.

[94]:
```
dfs["AFSC Objective Targets"]
```

[94]:
```
  AFSC  Norm Score     Merit  USAFA Proportion  Combined Quota  USAFA Quota  \
0   R1           1  0.457486              0.65               6            1
1   R2           1  0.457486              0.65               7            1
2   R3           1  0.457486              0.65               4            1
3   R4           1  0.457486              0.65               2            0

   ROTC Quota  Utility  Male  Minority  Tier 1  Tier 2  Tier 3  Tier 4
0           5        0   0.7      0.75    0.54  0.4599       0       0
1           4        0   0.7      0.75    1.00  0.0000       0       0
2           3        0   0.7      0.75    0.12  0.8800       0       0
3           2        0   0.7      0.75    1.00  0.0000       0       0
```

This dataframe displays the target measure for each of the objectives. In a perfect world, we'd meet every AFSC objective by hitting these values for each of them.

[95]:
```
dfs["AFSC Objective Min Value"]
```

[95]:
```
  AFSC            Norm Score                 Merit      USAFA Proportion  \
0   R1
1   R2
2   R3
3   R4

  Combined Quota USAFA Quota ROTC Quota              Utility  \
0           6, 7        1, 7       5, 7
1           5, 8        1, 8       4, 8
2           4, 4        1, 4       3, 4
3           2, 3        0, 3       2, 3

                  Male              Minority   Tier 1               Tier 2  \
0                                             0.54, 5           0, 0.4599
1                                              1.0, 5
2                                             0.12, 5             0, 0.88
3                                              1.0, 5              0.0, 5
```

```
                Tier 3                Tier 4
0
1
2
3
```

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 "Random_1 AFSCs.csv". The AFOCD Tier objective constraint ranges come from the "Deg Tiers" columns of "Random_1 AFSCs.csv" as well. Since this is random data, nothing else is constrained to begin with.

[96]:
```
dfs["Constraint Type"]
```

[96]:

	AFSC	Norm Score	Merit	USAFA Proportion	Combined Quota	USAFA Quota	\
0	R1	0	0	0	2	0	
1	R2	0	0	0	2	0	
2	R3	0	0	0	2	0	
3	R4	0	0	0	2	0	

	ROTC Quota	Utility	Male	Minority	Tier 1	Tier 2	Tier 3	Tier 4
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0
3	0	0	0	0	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. A "1" 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 "2", therefore, is an "exact" constraint. The only place where we could legimately use a "1" instead of a "2" is for the AFOCD constraints.

Example: Let's say 14N wants 70% of their cadets to have tier 1 degrees. Let's also say the PGL is 190 and we assign 220 cadets. A "1" constraint is a less restrictive constraint, and would ensure that 133 cadets (190 * 0.70) have "Tier 1" degrees. Alternatively, a "2" constraint ensures the actual proportion gets constrained, so 154 cadets (220 * 0.70) will have "Tier 1" degrees. Sometimes it is really hard to meet the AFOCD for some AFSCs, and so a "1" constraint is necessary to ensure we meet the target based on the PGL, not the actual number of cadets. Most of the time, however, we use "2" as the constraint type.

Once these default value parameters have been imported/initialized for Random_1, they will be imported from Random_1 VP.csv in "Model Input".

AFSC	Objective	Objective Weight	Objective Target	AFSC Weight	Min Value	Min Objective Value	Constraint Type	Function Breakpoint Measures (a)	Function Breakpoint Values (f^hat)	Value Functions
R1	Norm Score	37.498	1	100	0		0	0.0,0.04167,0.08333,0.125,0.16667	0.0,0.00551,0.01185,0.01912,0.027	Min Increasing|0.3
R1	Merit	21.675	0.457486288	100	0		0	0.0,0.01906,0.03812,0.05719,0.076	0.0,0.13447,0.25151,0.35337,0.442	Min Increasing|-0.3
R1	USAFA Proportion	25.693	0.65	100	0		0	0.0,0.08333,0.16667,0.25,0.33333,(	0.0,0.00043,0.00218,0.00916,0.037	Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
R1	Combined Quota	100	6	100	0	6, 7	2	0.0,0.5,1.0,1.5,2.0,2.5,3.0,3.5,4.0,4.	0.0,0.00738,0.01768,0.03206,0.052	Quota_Normal|0.2, 0.25, 0.2
R1	USAFA Quota	0	1	100	0	1, 7	0			Min Increasing|0.3
R1	ROTC Quota	0	5	100	0	5, 7	0			Min Increasing|0.3
R1	Utility	0	0	100	0		0			
R1	Male	18.104	0.7	100	0		0	0.0,0.09167,0.18333,0.275,0.36667	0.0,0.00023,0.00127,0.00607,0.028	Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
R1	Minority	6.69	0.75	100	0		0	0.0,0.1,0.2,0.3,0.4,0.5,0.6,0.625,0.6	0.0,0.00012,0.00074,0.00402,0.021	Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
R1	Tier 1	80.815	0.54	100	0	0.54, 5	0	0.0,0.0225,0.045,0.0675,0.09,0.112	0.0,0.00551,0.01185,0.01912,0.027	Min Increasing|0.3
R1	Tier 2	46.18	0.4599	100	0	0, 0.4599	0	0.0,0.4599,0.4824,0.50491,0.52741	1.0,1.0,0.84644,0.71598,0.60516,0.!	Min Decreasing|0.3
R1	Tier 3	0	0	100	0		0			
R1	Tier 4	0	0	100	0		0			
R2	Norm Score	46.49	1	91.895	0		0	0.0,0.04167,0.08333,0.125,0.16667	0.0,0.00551,0.01185,0.01912,0.027	Min Increasing|0.3
R2	Merit	10.416	0.457486288	91.895	0		0	0.0,0.01906,0.03812,0.05719,0.076	0.0,0.13447,0.25151,0.35337,0.442	Min Increasing|-0.3
R2	USAFA Proportion	23.918	0.65	91.895	0		0	0.0,0.08333,0.16667,0.25,0.33333,(	0.0,0.00043,0.00218,0.00916,0.037	Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6
R2	Combined Quota	100	7	91.895	0	5, 8	2	0.0,0.58333,1.16667,1.75,2.33333,:	0.0,0.00738,0.01768,0.03206,0.052	Quota_Normal|0.2, 0.25, 0.2

As you can see, this file structure has a row for every AFSC and objective pair. The dataframes shown above are flattened into columns and exist here. Again, these are the actual value parameters used for the problem instance, since the "Defaults" could have been more generalized (they aren't here since we exported our randomly generated set as defaults, therefore having it be the same thing as what you're seeing here). For a real class you could simply take the previous years defaults, tweak them a bit if needed, and import those as a starting point for a new class year. The objective targets would be updated to reflect the information of the problem instance you're looking at (the USAFA proportion objective target would be the proportion of the USAFA cadets of the instance you're solving, for example).

The "AFSC Weight" and "Min Value" columns above pertain to the AFSC itself, not the AFSC-objective pair like the others (which is why "AFSC Weight" is all 100s for R1). The three columns to the right pertain to the value functions used for each AFSC and objective which I will discuss in more detail in the following section.

5.6.6 Value Functions

[97]:
```
dfs["Value Functions"]
```

[97]:
```
  AFSC          Norm Score                Merit  \
0   R1  Min Increasing|0.3  Min Increasing|-0.3
1   R2  Min Increasing|0.3  Min Increasing|-0.3
2   R3  Min Increasing|0.3  Min Increasing|-0.3
3   R4  Min Increasing|0.3  Min Increasing|-0.3

                                  USAFA Proportion                Combined Quota  \
0  Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6  Quota_Normal|0.2, 0.25, 0.2
1  Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6  Quota_Normal|0.2, 0.25, 0.2
2  Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6  Quota_Normal|0.2, 0.25, 0.2
3  Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6  Quota_Normal|0.2, 0.25, 0.2

          USAFA Quota          ROTC Quota  \
0  Min Increasing|0.3  Min Increasing|0.3
1  Min Increasing|0.3  Min Increasing|0.3
2  Min Increasing|0.3  Min Increasing|0.3
3  Min Increasing|0.3  Min Increasing|0.3

                                              Utility  \
0                                                   …
1                                                   …
```

```
2                                                   …
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

                                          Minority              Tier 1  \
0  Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6  Min Increasing|0.3
1  Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6  Min Increasing|0.3
2  Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6  Min Increasing|0.3
3  Balance|0.15, 0.15, 0.1, 0.08, 0.08, 0.1, 0.6  Min Increasing|0.3

                                             Tier 2  \
0                                 Min Decreasing|0.3
1                                                 …
2                                 Min Decreasing|0.3
3                                 Min Increasing|0.3

                                             Tier 3  \
0                                                 …
1                                                 …
2                                                 …
3                                                 …

                                             Tier 4
0                                                 …
1                                                 …
2                                                 …
3                                                 …
```

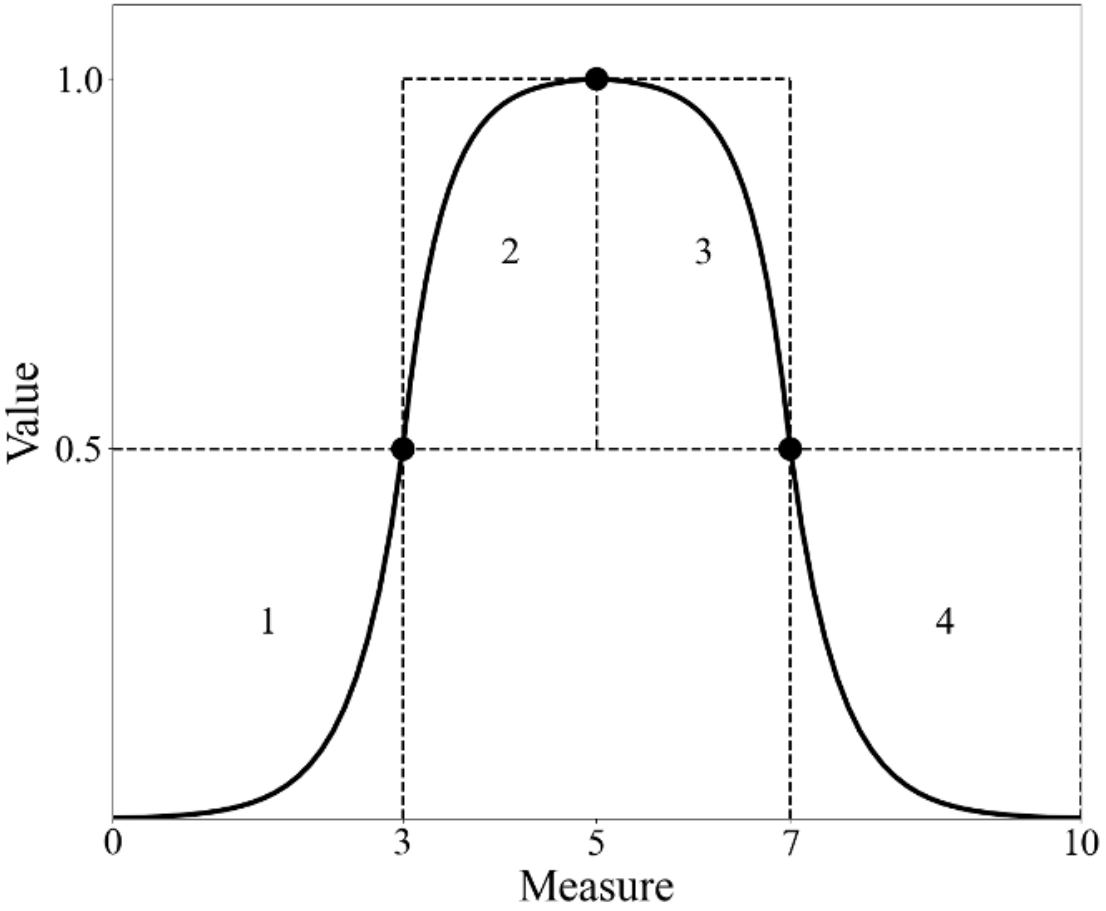
Here we have the value functions for each of the AFSC objectives. These definitely require some explaining. I've created my own terminology so that they can be generalized and constructed into actual value functions for each of the objectives. I have an excel file that outlines how these functions are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

[98]:
```
# I need to import this script
import afccp.core.data.values
```

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".



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"

[99]:
```
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.data.values.
 ↪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}
```

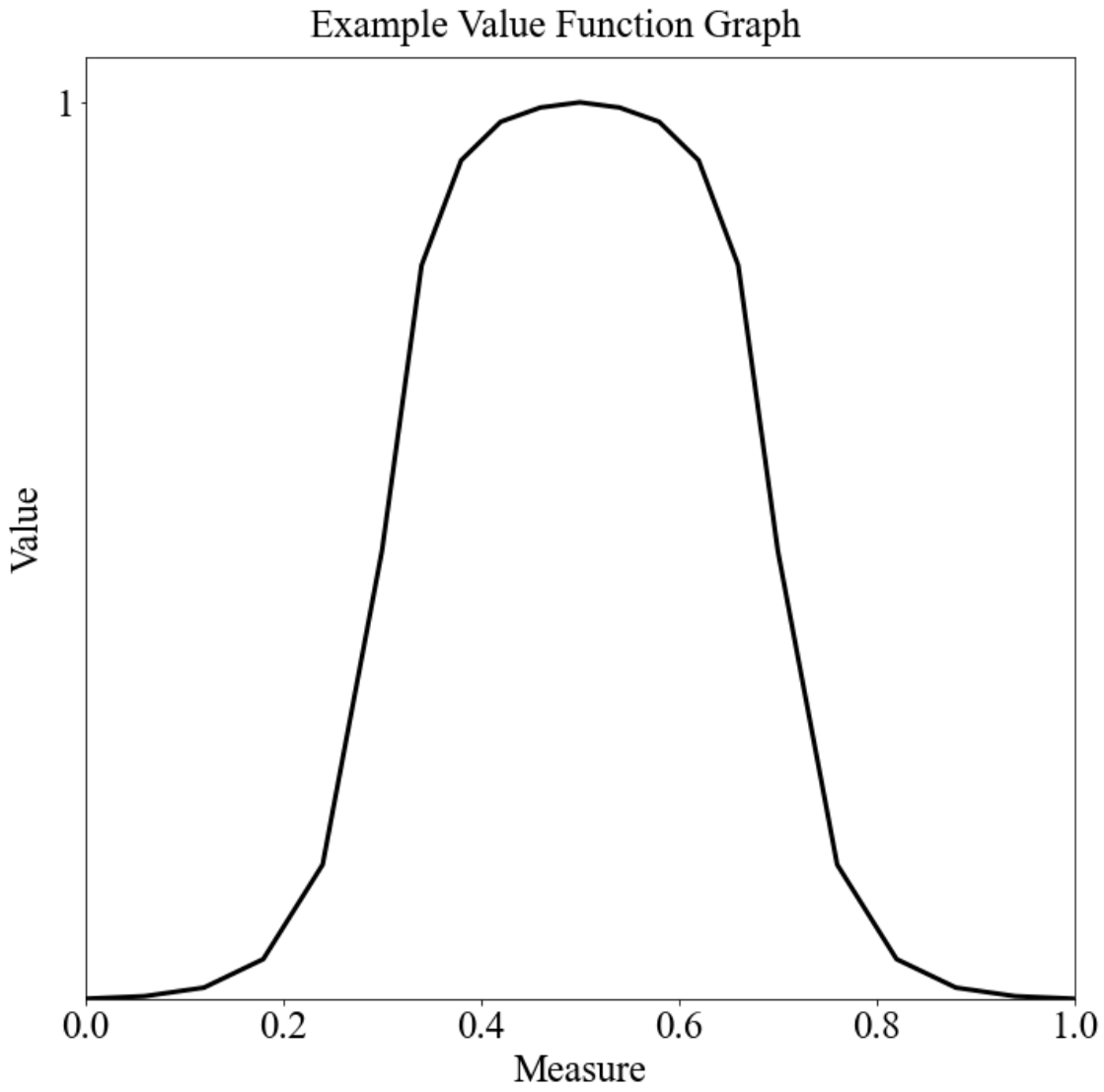
Now we have our segment dictionary! We know what the coordinates for the "main" breakpoints are, so we can now generate the rest of the breakpoints to make the function linear. Let's calculate the x and y coordinates of our function's breakpoints.

[100]:
```
x, y = afccp.core.data.values.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!

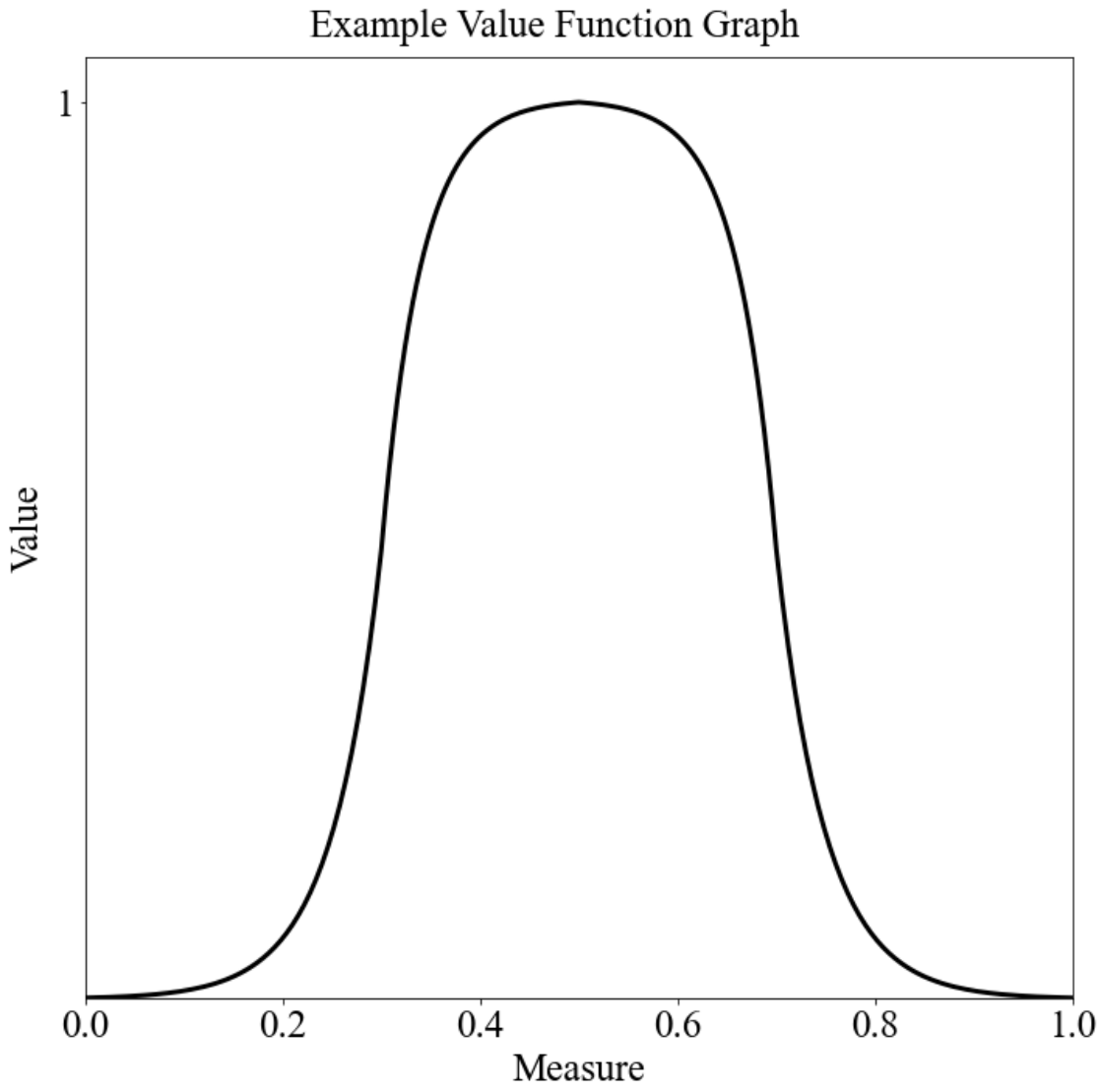
[101]:
```
# "Balance" type of value function!
from afccp.core.visualizations.charts import ValueFunctionChart
chart = ValueFunctionChart(x, y)
```



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!

[102]:
```
# 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.data.values.
  ↪create_segment_dict_from_string(vf_string, target=target, actual=actual)
x, y = afccp.core.data.values.value_function_builder(segment_dict=segment_dict,␣
  ↪num_breakpoints=num_breakpoints)
chart = ValueFunctionChart(x, y)
```

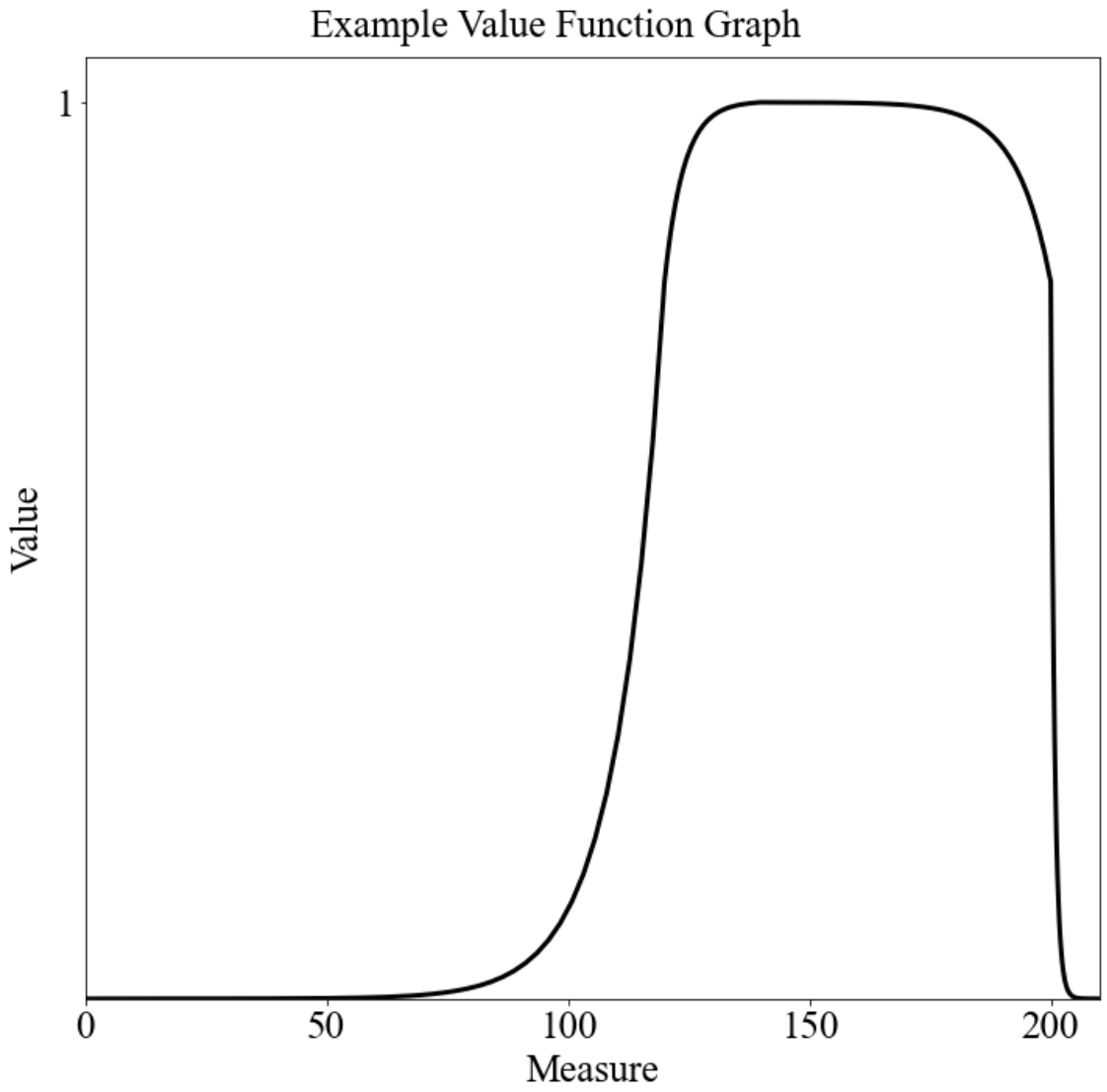


That is the "Balance" value function type. This is intended for the objectives that seek to "balance" certain characteristics of the cadets (USAFA/Male/Minority proportions and sometimes Merit as well). I did end up changing the Merit value function to be a "Min Increasing" because I decided against 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:

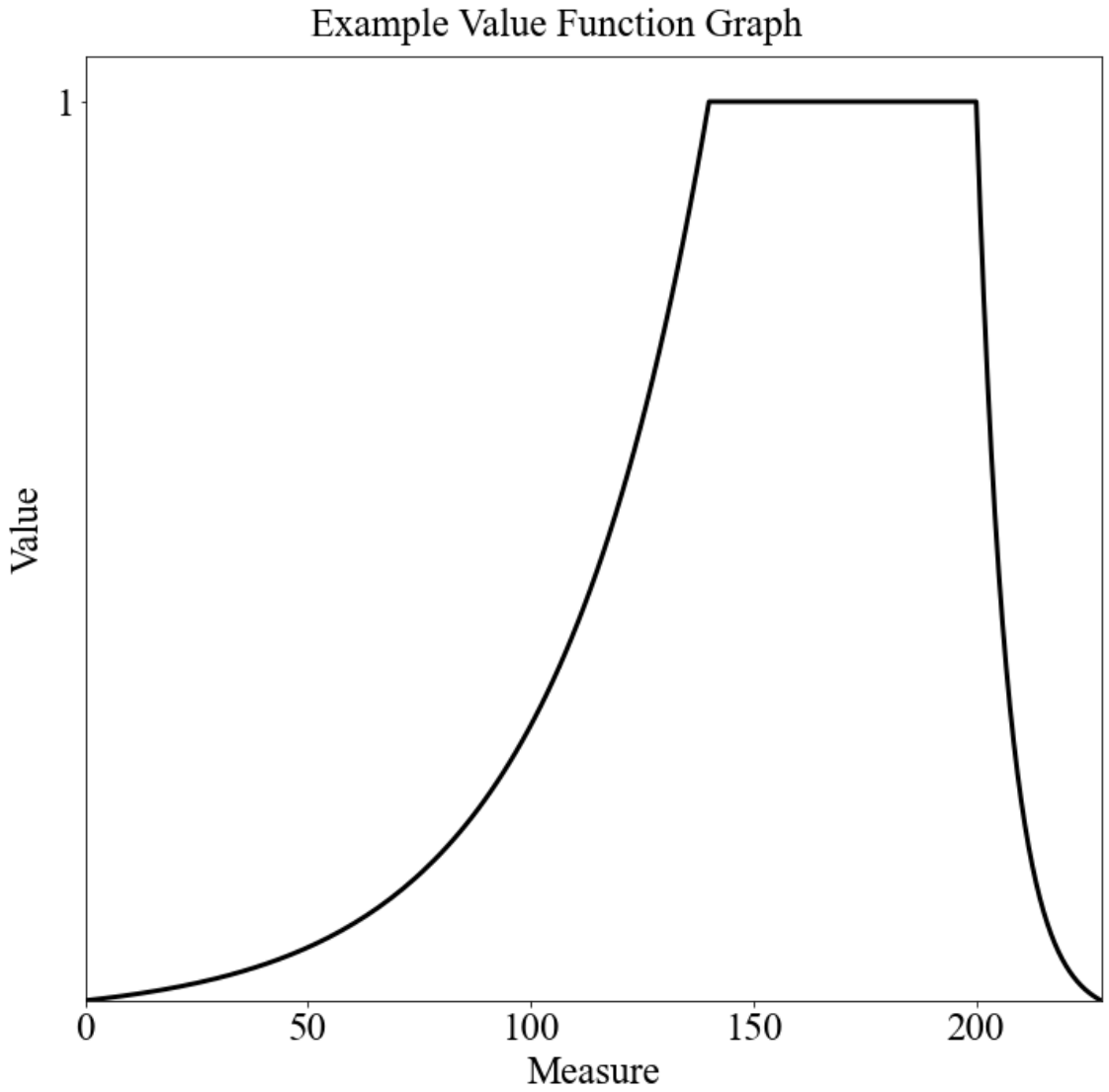
[103]:
```
vf_string = "Quota_Direct|0.1, 1, 0.6, 0.1, 0.8, 0.8"
minimum = 120  # Lower Bound
maximum = 200  # Upper Bound
target = 140  # Desired number of cadets within the range
num_breakpoints = 200  # How many breakpoints to use

# Don't change this
segment_dict = afccp.core.data.values.
 ↪create_segment_dict_from_string(vf_string, target=target,
                                                                  ␣
 ↪minimum=minimum, maximum=maximum)
x, y = afccp.core.data.values.value_function_builder(segment_dict=segment_dict,␣
 ↪num_breakpoints=num_breakpoints)
chart = ValueFunctionChart(x, y)
```



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 "Random_1 AFSCs.csv" data!

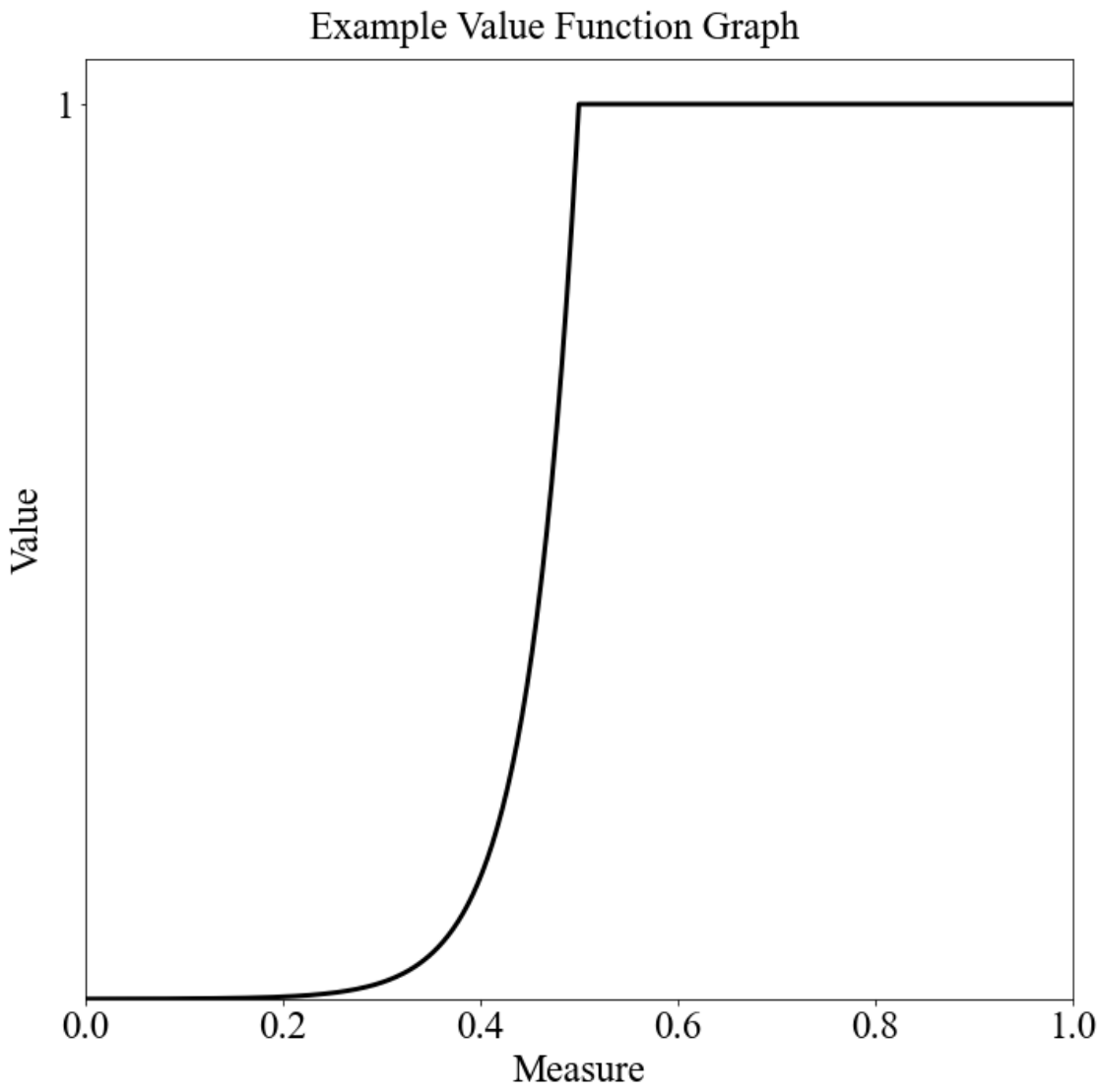
Another value function we can choose for the quota objective is the "Quota_Normal" function type. This is intended for AFSCs that either don't care about the number of cadets (as long as they fall within a certain range) or didn't specify. For example, the PGL says 120 and after speaking with them we 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:

[104]:
```
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.data.values.
 ↪create_segment__dict_from_string(vf_string, target=target,
                                                                  ␣
 ↪minimum=minimum, maximum=maximum)
x, y = afccp.core.data.values.value_function_builder(segment_dict=segment_dict,␣
 ↪num_breakpoints=num_breakpoints)
chart = ValueFunctionChart(x, y)
```

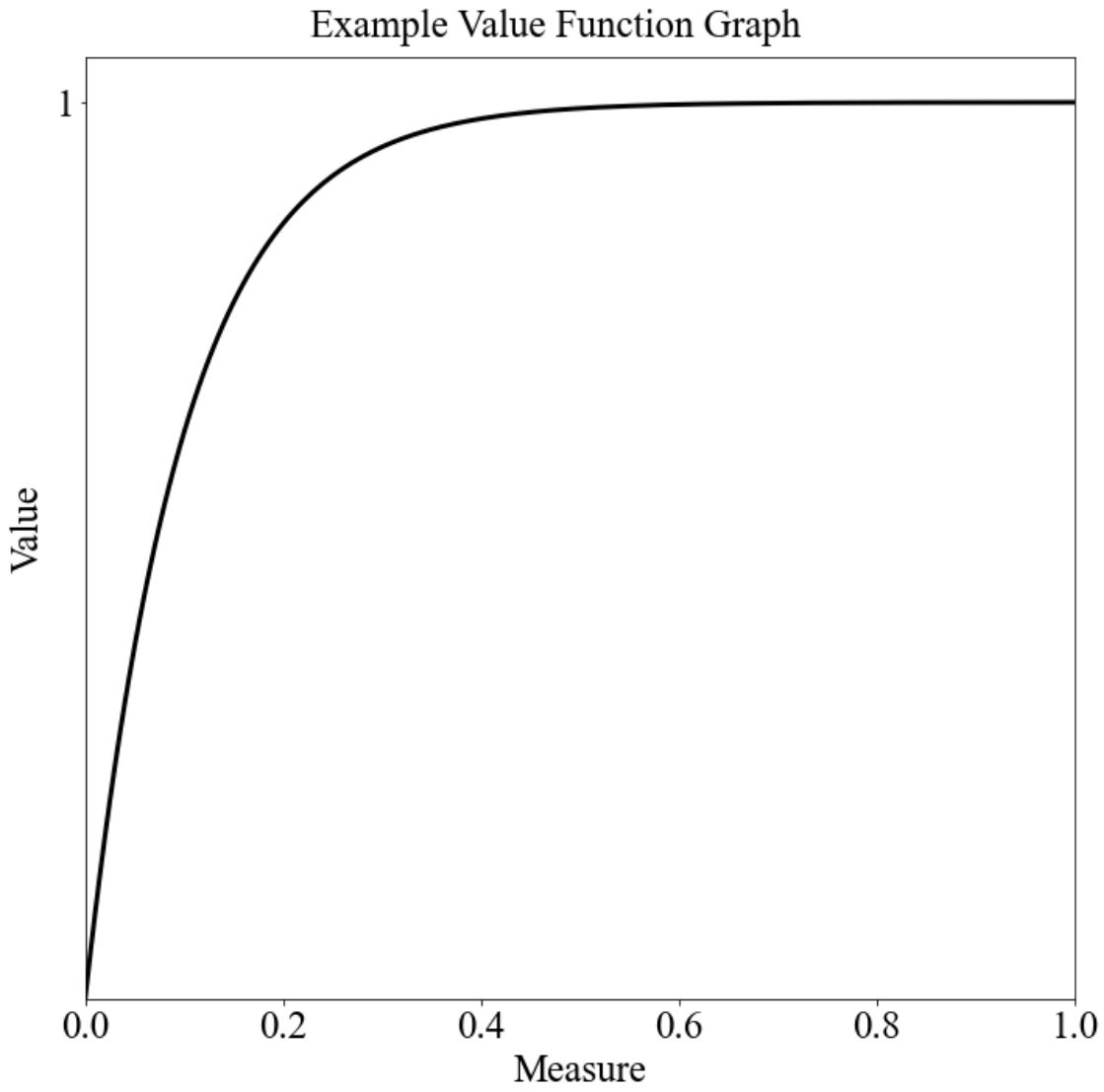


The last two kinds of value functions I'll discuss are the "Min Increasing" and "Min Decreasing" types. They are very simple and only have one segment which is a simple exponentional curve to get to the target measure (in the x space). The only parameter is ρ. The vf_string then looks like: "Min Increasing|ρ" 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:

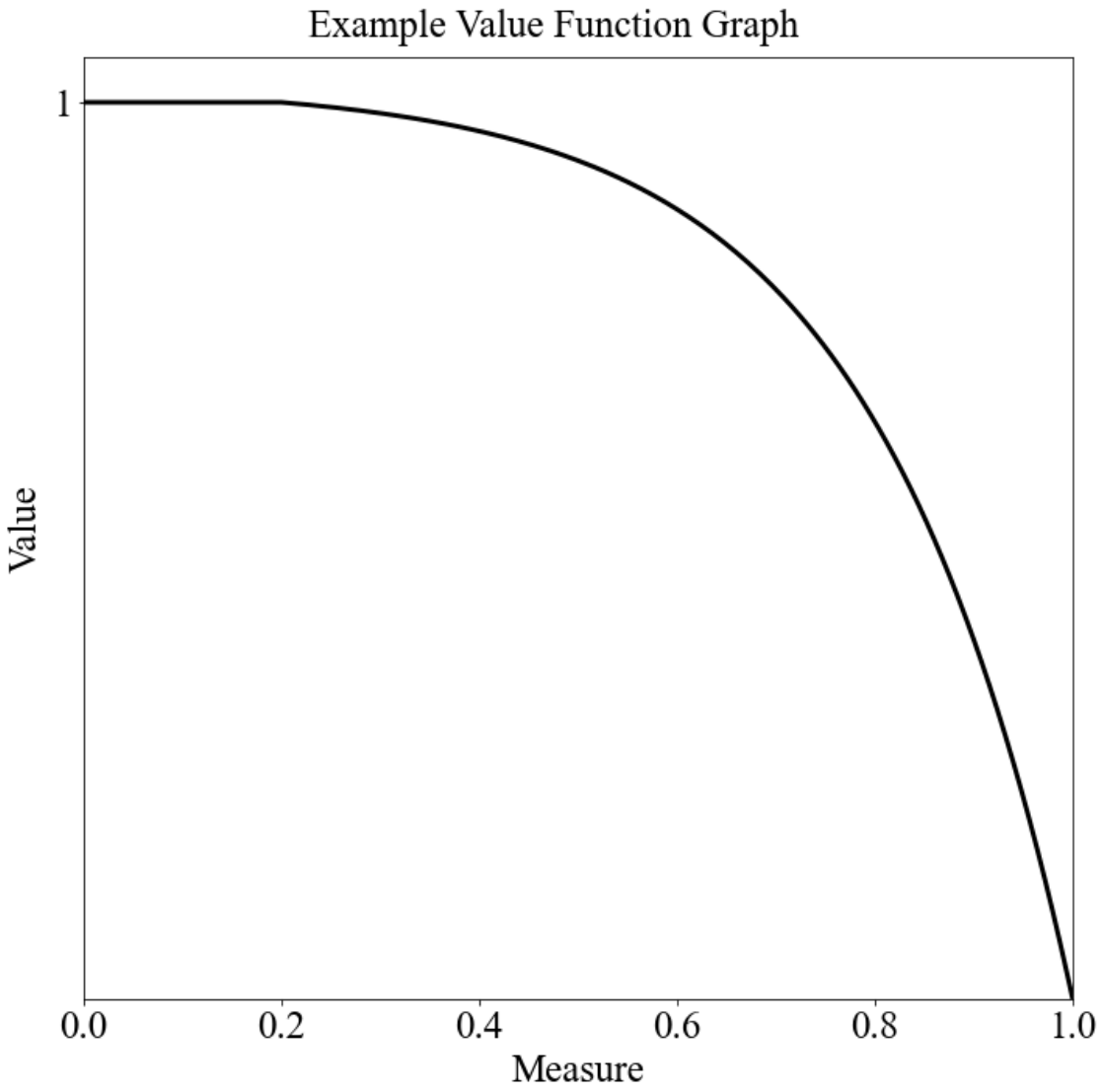
[105]:
```
vf_string = "Min Increasing|0.1"
target = 0.5
num_breakpoints = 200  # How many breakpoints to use

# Don't change this
segment_dict = afccp.core.data.values.
 ↪create_segment_dict_from_string(vf_string, target=target)
x, y = afccp.core.data.values.value_function_builder(segment_dict=segment_dict,␣
 ↪num_breakpoints=num_breakpoints)
chart = ValueFunctionChart(x, y)
```



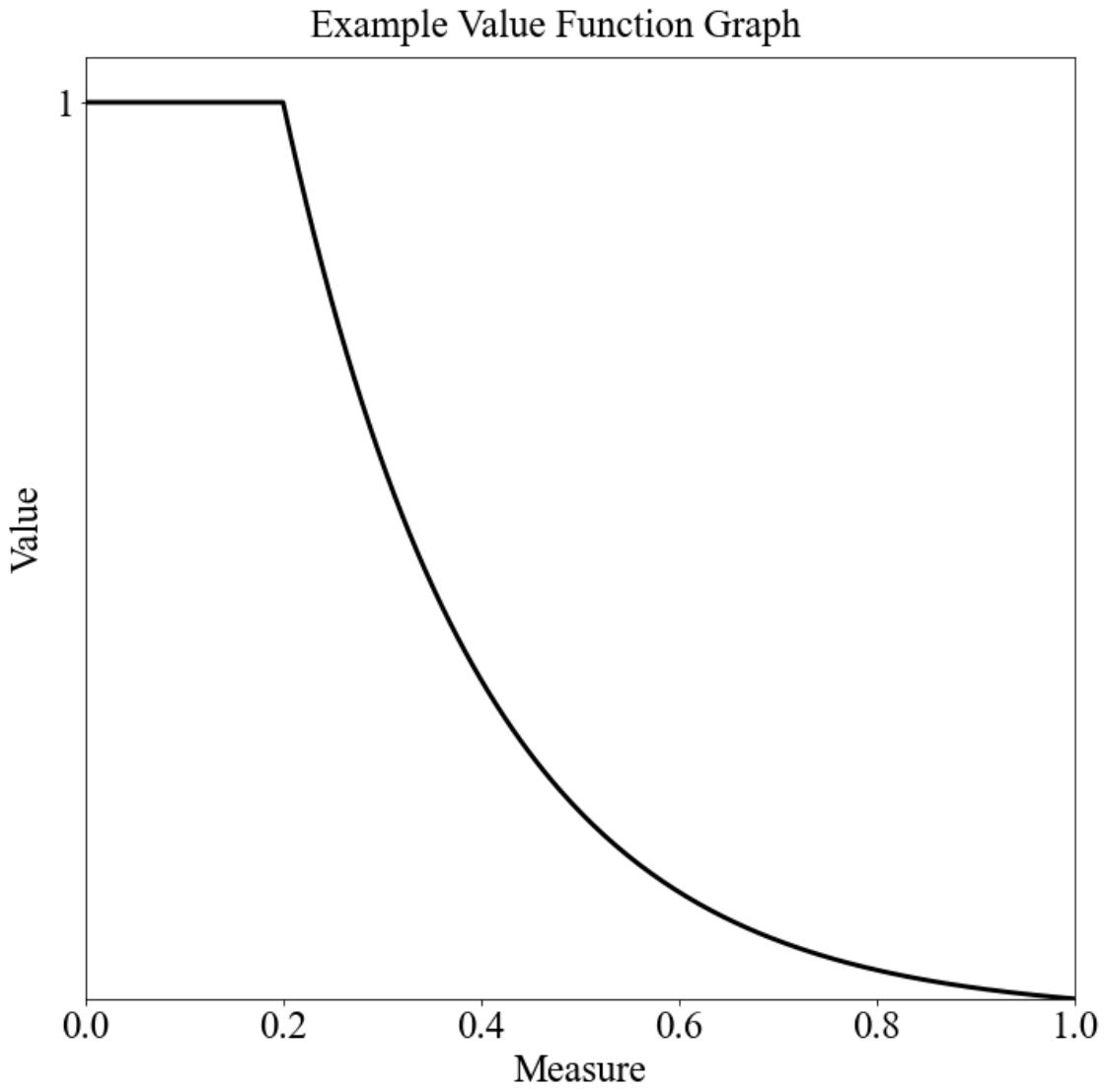
[106]:
```
vf_string = "Min Increasing|-0.1"
target = 1
num_breakpoints = 200  # How many breakpoints to use

# Don't change this
segment_dict = afccp.core.data.values.
 ↪create_segment_dict_from_string(vf_string, target=target)
x, y = afccp.core.data.values.value_function_builder(segment_dict=segment_dict,␣
 ↪num_breakpoints=num_breakpoints)
chart = ValueFunctionChart(x, y)
```



[107]:
```
vf_string = "Min Decreasing|-1"
target = 0.2
num_breakpoints = 200  # How many breakpoints to use

# Don't change this
segment_dict = afccp.core.data.values.
 ↪create_segment_dict_from_string(vf_string, target=target)
x, y = afccp.core.data.values.value_function_builder(segment_dict=segment_dict,␣
 ↪num_breakpoints=num_breakpoints)
chart = ValueFunctionChart(x, y)
```



[108]:
```
vf_string = "Min Decreasing|1"
target = 0.2
num_breakpoints = 200  # How many breakpoints to use

# Don't change this
segment_dict = afccp.core.data.values.
 ↪create_segment_dict_from_string(vf_string, target=target)
x, y = afccp.core.data.values.value_function_builder(segment_dict=segment_dict,␣
 ↪num_breakpoints=num_breakpoints)
chart = ValueFunctionChart(x, y)
```

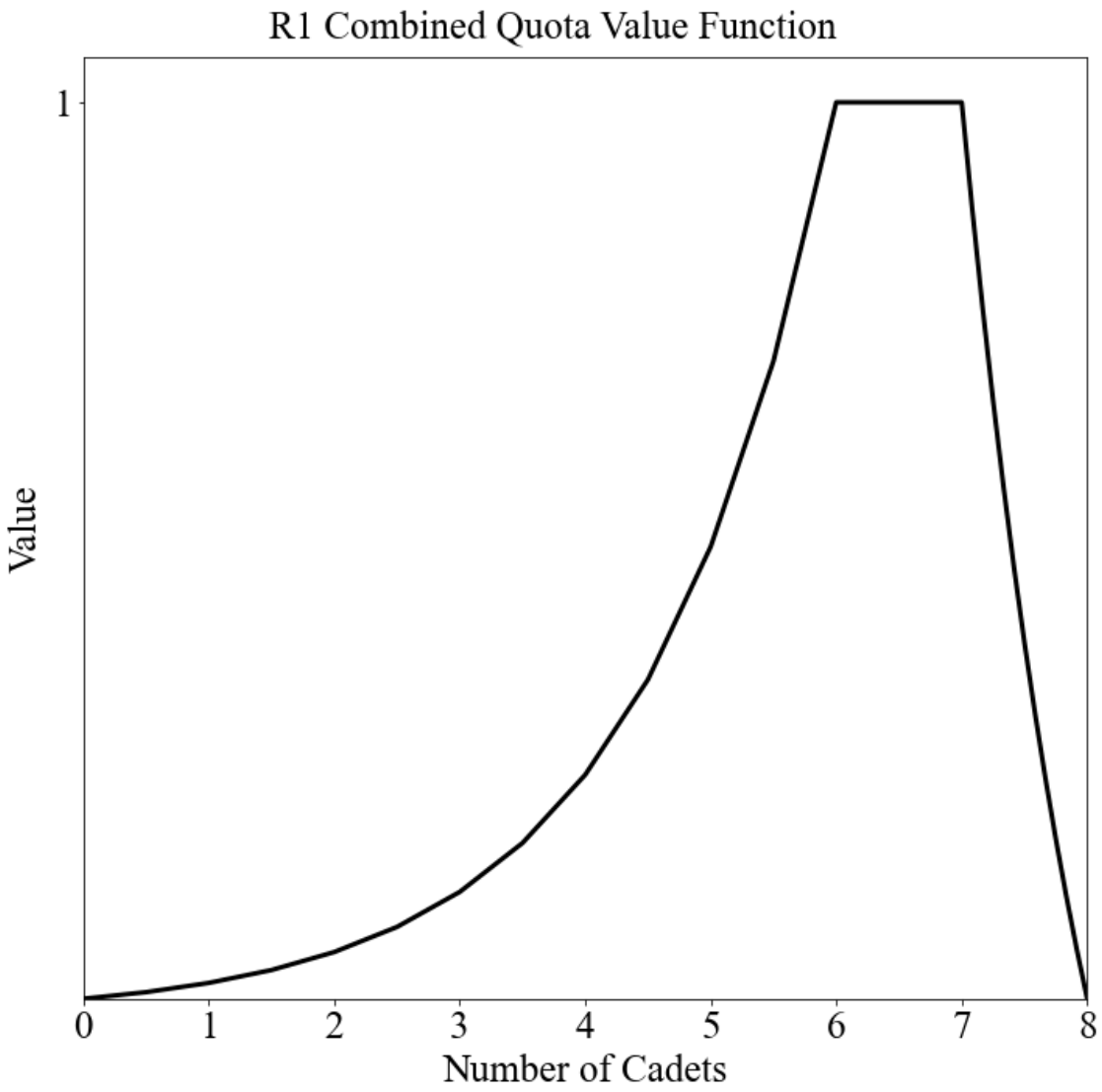


And there you have it! This is how I code up and construct my many value functions for each of the objectives for each of the AFSCs. Please reach out if you have any questions as I know this is a confusing section.

Now that we're done discussing the types of value functions, we can take a look at the actual value functions used on "Random_1". We're plotting the breakpoints (x/y coordinates) for a specific AFSC objective value function. Here is an example:

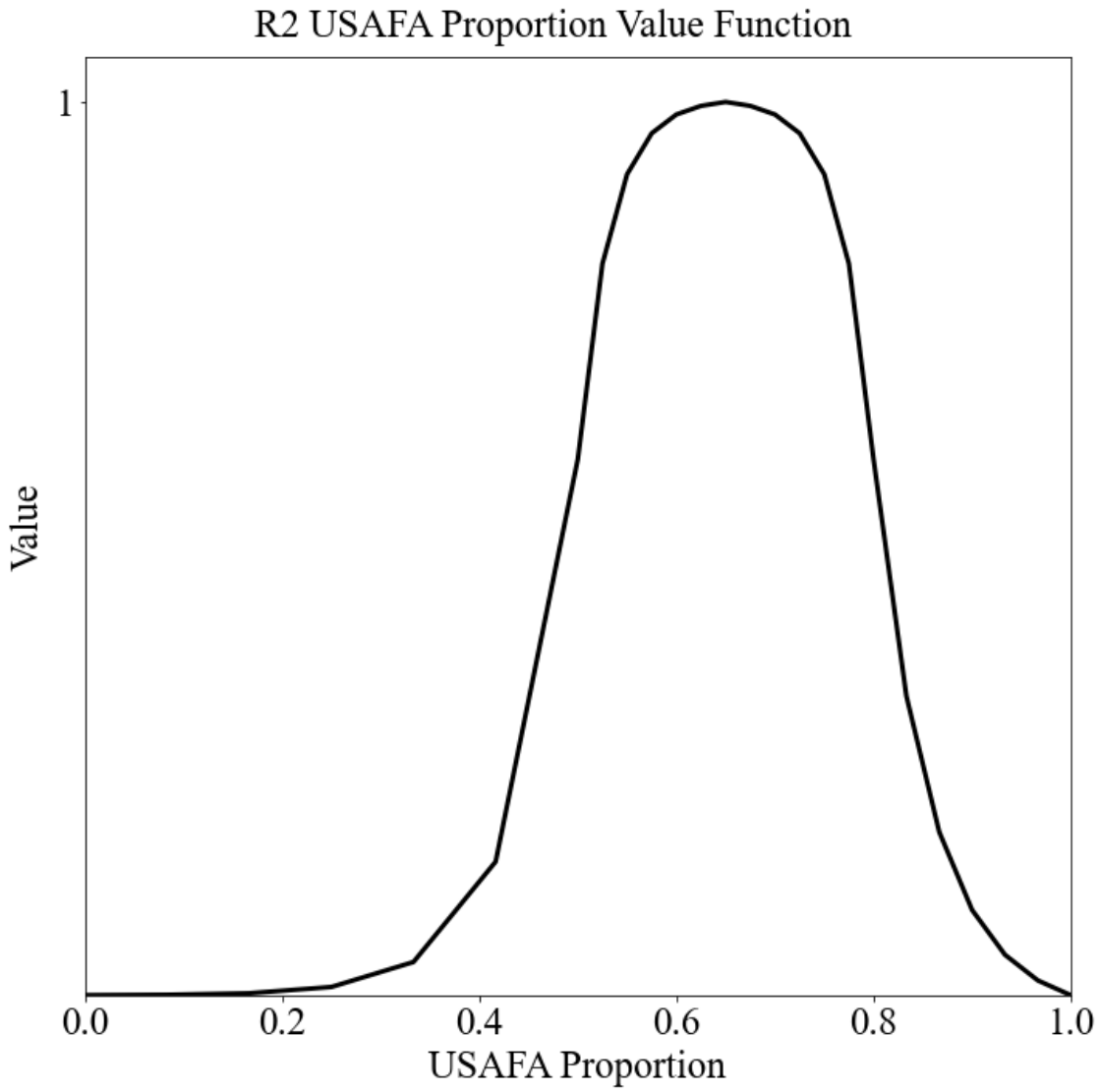
[109]:
```
# Plot the value function (this also saves it to the "Value Functions"␣
↪sub-folder by default FYI)
c = instance.show_value_function({'afsc': "R1", 'objective': 'Combined Quota'})
```

```
Creating value function chart for objective Combined Quota for AFSC R1
```



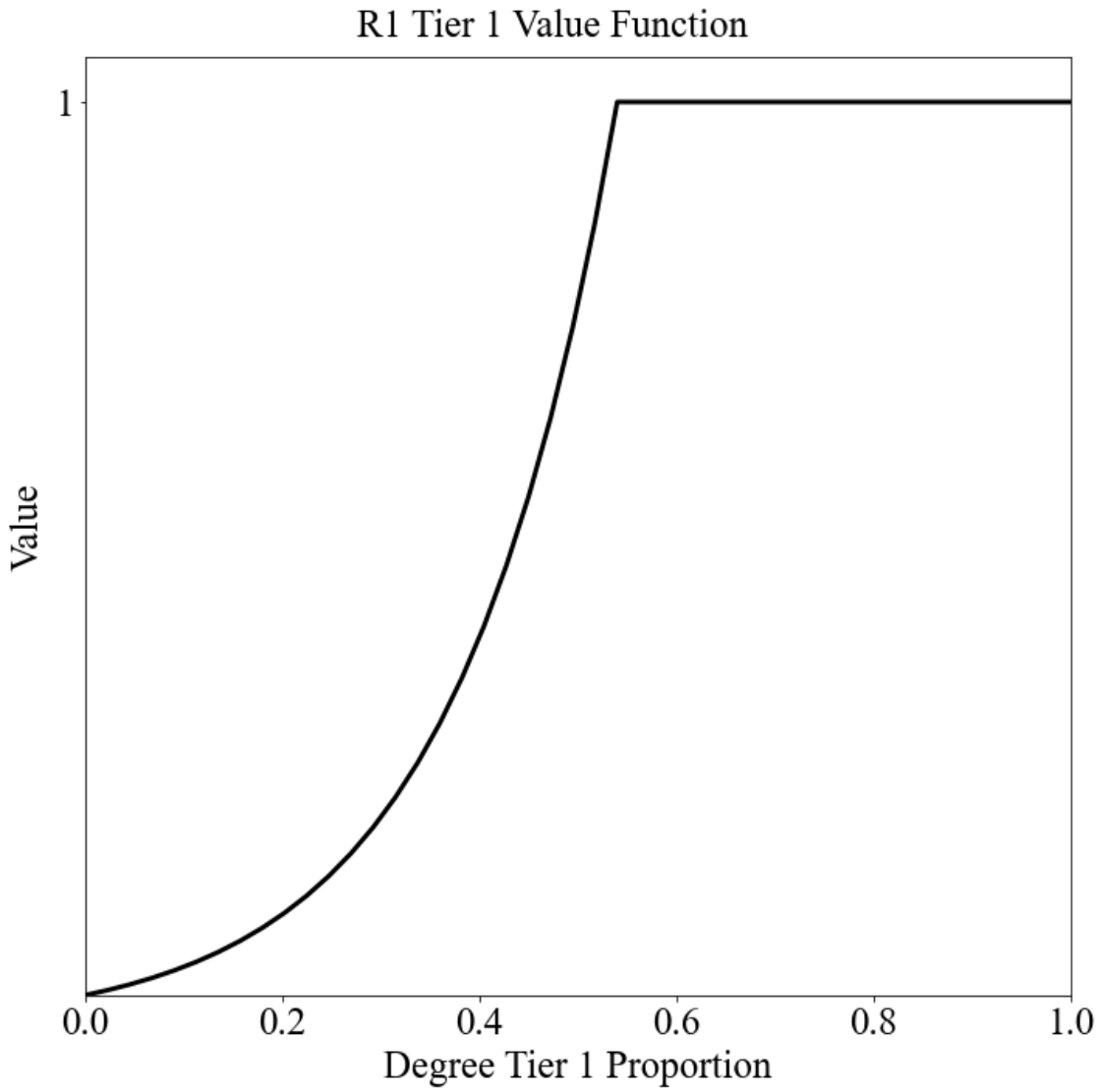
[110]:
```
# Plot the value function (this also saves it to the "Value Functions"␣
↪sub-folder by default FYI)
c = instance.show_value_function({'afsc': "R2", 'objective': 'USAFA␣
↪Proportion'})
```

```
Creating value function chart for objective USAFA Proportion for AFSC R2
```



[111]:
```
# Plot the value function (this also saves it to the "Value Functions"␣
↪sub-folder by default FYI)
c = instance.show_value_function({'afsc': "R1", 'objective': 'Tier 1'})
```

```
Creating value function chart for objective Tier 1 for AFSC R1
```



5.6.7 Global Utility

There are a few other components of the "value parameters" that I haven't mentioned yet. The "global_utility" matrix is based on the cadets' preferences as well as the AFSCs' preferences. The two matrices "cadet_utility" and "afsc_utility" are merged according to the overall weights on the cadets/AFSCs.

[112]:
```
# Cadet Utility Matrix
p['cadet_utility']
```

[112]:
```
array([[0.77  , 0.16  , 1.    , 0.605 ],
       [0.56  , 0.16  , 1.    , 0.385 ],
       [0.25  , 1.    , 0.62  , 0.855 ],
```

```
       [1.    , 0.49  , 0.815 , 0.66  ],
       [1.    , 0.25  , 0.515 , 0.665 ],
       [0.385 , 0.59  , 1.    , 0.225 ],
       [0.145 , 1.    , 0.605 , 0.335 ],
       [0.385 , 0.24  , 1.    , 0.83  ],
       [0.27  , 0.645 , 0.405 , 1.    ],
       [0.16  , 1.    , 0.38  , 0.865 ],
       [0.64  , 1.    , 0.22  , 0.5   ],
       [0.6   , 0.77  , 1.    , 0.375 ],
       [1.    , 0.7483, 0.2617, 0.    ],
       [0.3467, 0.5733, 1.    , 0.    ],
       [0.5783, 0.2817, 1.    , 0.    ],
       [0.595 , 0.36  , 1.    , 0.125 ],
       [0.23  , 1.    , 0.55  , 0.725 ],
       [0.13  , 1.    , 0.75  , 0.57  ],
       [0.695 , 1.    , 0.365 , 0.195 ],
       [0.86  , 0.5083, 0.2767, 0.    ]])
```

[113]:
```
# AFSC Utility Matrix
p['afsc_utility']
```

[113]:
```
array([[0.75  , 0.15  , 0.45  , 0.625 ],
       [0.05  , 0.05  , 0.25  , 0.125 ],
       [0.1   , 0.6   , 0.6   , 0.5625],
       [0.9   , 0.7   , 0.9   , 0.875 ],
       [1.    , 0.75  , 0.55  , 0.9375],
       [0.6   , 0.45  , 0.5   , 0.4375],
       [0.25  , 0.35  , 0.35  , 0.0625],
       [0.4   , 0.1   , 0.7   , 0.5   ],
       [0.5   , 0.25  , 0.1   , 0.6875],
       [0.7   , 1.    , 0.85  , 1.    ],
       [0.15  , 0.55  , 0.3   , 0.375 ],
       [0.85  , 0.9   , 0.75  , 0.75  ],
       [0.95  , 0.85  , 0.65  , 0.    ],
       [0.55  , 0.8   , 1.    , 0.    ],
       [0.45  , 0.3   , 0.95  , 0.    ],
       [0.35  , 0.2   , 0.4   , 0.25  ],
       [0.3   , 0.95  , 0.8   , 0.8125],
       [0.2   , 0.4   , 0.15  , 0.3125],
       [0.65  , 0.65  , 0.05  , 0.1875],
       [0.8   , 0.5   , 0.2   , 0.    ]])
```

[114]:
```
# Overall weights on cadets/AFSCs
print("weight on cadets:", round(vp['cadets_overall_weight'], 2))
print("weight on AFSCs:", round(vp['afscs_overall_weight'], 2))
```

```
weight on cadets: 0.43
weight on AFSCs: 0.57
```

[115]:
```
# Global Utility Matrix (Each cell is the weighted sum of cadet/AFSC utility)
vp['global_utility']  # Extra column is for the unmatched cadets!
```

[115]:
```
array([[0.75860653, 0.15430326, 0.68667957, 0.61639347, 0.        ],
       [0.26946651, 0.09733591, 0.57274487, 0.23688489, 0.        ],
       [0.16454897, 0.7721306 , 0.60860653, 0.6883705 , 0.        ],
       [0.94303265, 0.60963144, 0.86342225, 0.7824798 , 0.        ],
       [1.        , 0.53483675, 0.53493857, 0.82023603, 0.        ],
       [0.5074798 , 0.51024571, 0.71516325, 0.34605562, 0.        ],
       [0.20481572, 0.62971222, 0.45973326, 0.17976397, 0.        ],
       [0.3935451 , 0.16024571, 0.82909795, 0.64200774, 0.        ],
       [0.40102491, 0.41997897, 0.23124958, 0.82197703, 0.        ],
       [0.46762369, 1.        , 0.64774655, 0.94190592, 0.        ],
       [0.36085998, 0.74364692, 0.26557388, 0.42879081, 0.        ],
       [0.74241838, 0.84405756, 0.85758162, 0.58862756, 0.        ],
       [0.97151632, 0.80623579, 0.48290422, 0.        , 0.        ],
       [0.46251462, 0.70244498, 1.        , 0.        , 0.        ],
       [0.50521089, 0.29212503, 0.97151632, 0.        , 0.        ],
       [0.45542999, 0.26885224, 0.6581959 , 0.19620919, 0.        ],
       [0.26987715, 0.97151632, 0.69241838, 0.77484643, 0.        ],
       [0.16987715, 0.6581959 , 0.4081959 , 0.42330907, 0.        ],
       [0.66936469, 0.80061427, 0.18555285, 0.19072745, 0.        ],
       [0.82581959, 0.50357171, 0.23300604, 0.        , 0.        ]])
```

[116]:
```
print("Cadet 0's utility on AFSC 0:", p['cadet_utility'][0, 0])
print("AFSC 0's utility on cadet 0:", p['afsc_utility'][0, 0])
print("Global Utility[0, 0]:", round(vp['cadets_overall_weight'], 2), "*",␣
 ↪p['cadet_utility'][0, 0], "+",
    round(vp['afscs_overall_weight'], 2), "*", p['afsc_utility'][0, 0], "=",␣
 ↪vp['global_utility'][0, 0])
```

```
Cadet 0's utility on AFSC 0: 0.77
AFSC 0's utility on cadet 0: 0.75
Global Utility[0, 0]: 0.43 * 0.77 + 0.57 * 0.75 = 0.7586065299918269
```

The global utility matrix is unique to each set of value parameters (since you can toggle the weights on cadets/AFSCs and get a different matrix). This matrix lives at "Random_1 VP Global Utility.csv".

5.6.8 Cadet Utility Constraints (Meant just for AFPC/DSYA through an operational lens)

Another component of the value parameters is the cadet utility constraints. These constraints ensure certain cadets receive some minimum utility value, and the implication here usually applies to the top 10% of cadets as my method of preventing as much of the "adjudication" piece on the backend. Built into the classification process is adjudication where the sources of commissioning get to review the results prior to release and make adjustments as needed. One thing they always want to make sure is that their top cadets are getting something they want, and if they aren't they'll kick it back to AFPC to fix. I can use the cadet utility constraints to ensure that the top

10% of cadets receive one of their top 3 choices, or there is a good reason why they aren't getting one (if they only have very competitive AFSCs as their top 3 choices and don't rank high enough for them then that may cause problems).

These constraints live in the file "Random_1 Cadets Utility Constraints.csv"

Cadet	Merit	VP
0	0.4915472	0
1	0.1111355	0
2	0.2588289	0
3	0.5368679	0
4	0.9184725	0
5	0.533902	0
6	0.0045148	0
7	0.0963177	0
8	0.2790611	0
9	0.8942557	0
10	0.1871408	0
11	0.7297902	0
12	0.6841628	0
13	0.8659652	0
14	0.5590471	0
15	0.4740741	0
16	0.5819403	0
17	0.006536	0
18	0.3018812	0
19	0.6342849	0

As you can see, I include the cadet indices (as I've said before, it's because I like referring to cadets by their indices in the numpy arrays since that allows me to do a lot of things) as well as the cadets' order of merit. This information gives more context to why certain cadets have constrained minimum values versus other cadets! The "VP" column is the actual constrained minimum utilities for all of the cadets. If you have a second set of value parameters that you're using, there would be another column called "VP2". As you can see, the default is to keep all of the cadets unconstrained. I've thought a lot about how this should work and wrestled with the idea of just making a function to go through and constrain the top 10% to get a utility value greater than or equal to their third choice but ultimately decided against it. The reason is that it's more complicated then that and I firmly believe the AFPC/DSYA analyst needs to be the one to do it manually.

When I ran this for FY23 and FY24, I tuned the model parameters to be what I needed them to

be based on everyone's wants and desires and then solved it initially without any cadet constraints. I can then look at the solution at the top 10/20% of the class and if the cadet is receiving a top 3 preference anyway (vast majority do), then I constrain their utilities to be whatever their third choice utility was. I then filter on the people who aren't getting a top 3 preference. If the reason is just because the optimal solution involved this cadet not getting a top 3 preference, and they really should have received one based on preferences (they didn't have 3 very hard choices to meet), then I also enforce the utility constraint. If there's a clear reason why they're getting their fourth choice (and I mean a VERY justifiable reason), then I constrain them to their fourth choice utility. For top 10%, I don't think this happened at all but did occur for top 20%.

[117]:
```
# Cadet utility constriants
vp['cadet_value_min']  # Defaults to 0! For an example problem (not real class␣
 ↪year), don't mess with this
```

[117]:
```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0.])
```

5.6.9 Goal-Programming Parameters

One final input file that I haven't mentioned is "Random_1 Goal Programming.csv". These are the inputs that another AFIT researcher, former Lt. Rebecca Reynolds (now Capt. Rebecca Eisemann), used for her goal-programming model. My intent for "afccp" has always been to provide a way for researchers in this field to contribute to this "academic" problem and try new things to inspire innovation for AFPC/DSYA. For her goal programming model, her inputs are structured in this way:

Constraint	Raw Penalty	Raw Reward	Normalized P	Normalized R	Run Penalty	Run Reward	Penalty Weig	Reward Weight
T	Unk	Unk	0.05673575	0.00695243	1	1	100	0
F	Unk	Unk	0.04582189	0.00802036	1	1	100	0
M	Unk	Unk	0.12274925	0.00903901	1	1	90	0
D_under	Unk	Unk	0.17549835	0.00706695	1	1	30	0
D_over	Unk	Unk	1	1	1	1	30	0
P	Unk	Unk	0.04595048	0.06188925	1	1	25	0
U_under	Unk	Unk	0.66972477	0.02567568	1	1	50	0
U_over	Unk	Unk	0.17312253	0.0814951	1	1	50	0
R_under	Unk	Unk	0.57480315	0.02828283	1	1	50	0
R_over	Unk	Unk	0.74654849	0.03291835	1	1	50	0
W	Unk	Unk	0.09134515	0.0138976	1	1	25	0
S	Unk	Unk	0	7.43E-05	0	1	0	100

In order to get the penalty/reward terms here we need to run the model on the specific class year to tune the parameters to get the actual parameters needed to run the full goal programming model. It's a little nuanced and you can certainly view her thesis here: https://scholar.afit.edu/etd/5449/. I will briefly cover her model a little more later on.

5.7 Data Methods

Now that I've described in detail the data that afccp uses, let's talk about the data-manipulation methods available for the CadetCareerProblem class. For a "Random" generated problem set, it doesn't really matter what you do to the data which is why I created a convenient "instance.fix_generated_data()" function that gets your generated data looking right for you to run models/algorithms to solve. Many of the components of that function are also relevant for a real class year, but not all!

[118]:
```
# Takes the CIP codes and translates them to the qualification matrix (only␣
 ↪works for real data)
instance.calculate_qualification_matrix()
```

```
Adjusting qualification matrix…
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Input In [118], in <cell line: 2>()
      1 # Takes the CIP codes and translates them to the qualification matrix␣
 ↪(only works for real data)
----> 2 instance.calculate_qualification_matrix()

File ~/Desktop/Coding Projects/afccp/afccp/core/main.py:409, in␣
 ↪CadetCareerProblem.calculate_qualification_matrix(self, printing)
    406         qual_matrix = afccp.core.data.support.cip_to_qual_tiers(
    407             parameters["afscs"][:parameters["M"]], parameters['cip1'])
    408 else:
--> 409     raise ValueError("Error. Need to update the degree tier␣
 ↪qualification matrix to include tiers "
    410                      "('M1' instead of 'M' for example) but don't have␣
 ↪CIP codes. Please incorporate this.")
    412 # Load data back into parameters
    413 parameters["qual"] = qual_matrix

ValueError: Error. Need to update the degree tier qualification matrix to␣
 ↪include tiers ('M1' instead of 'M' for example) but don't have CIP codes.␣
 ↪Please incorporate this.
```

As you can see, the code above produces an error since we don't have any CIP codes in "Random_1 Cadets.csv". Going further, even if we did put them into our dataset it still wouldn't work since the AFSCs themselves are also meaningless. This function only works with real AFSC names and with CIP codes in our cadets data. As an aside, I've tried to incorporate a lot of error handling throughout afccp but it's certainly not flawless and still a work in progress so if you encounter something that doesn't make since please let me know and I will a) help you fix it and also b) include some more error handling measures so we catch that kind of error in a manner that makes more sense to the analyst using afccp.

5.7.1 Operational Data Processing (Meant just for AFPC/DSYA)

The intent of this section is primarily for the "operational lens" of this problem, since we have the easy method I've already described that will fix the fake data for you anyway: instance.fix_generated_data().

Ok, your main job for processing a given class year of cadets is to construct the "Cadets.csv", "AFSCs.csv" and "AFSCs Preferences.csv", and the default value parameters excel file located in the support sub-folder. Additionally, to create the AFSC preferences for Rated career fields, you'll also need the "ROTC Rated OM.csv" and "USAFA Rated OM.csv" files. "AFSCs Preferences.csv"

should contain the necessary rankings for Non-Rated AFSCs, with something as a placeholder for rated career field rankings. The two OM datasets should have the percentiles for their respective source of commissioning cadets that are rated eligible (they volunteered). If a cadet volunteered for Rated, and is eligible for at least one of the rated AFSCs, they should be in this dataset. If the cadet is not eligible for a specific rated AFSC, they have a "0" in that position. The important thing here is that the cadets can be sorted in descending order with highest ranking at the top and lowest at the bottom using those "percentile" values (1, 0.99, 0.98, …, 0.01, 0, 0, etc.). This is what the code does to get the ordered list of cadets (AFSC preferences).

[119]:
```
# Takes the two Rated OM datasets and re-calculates the AFSC rankings for Rated␣
 ↪AFSCs for both SOCs
instance.construct_rated_preferences_from_om_by_soc()
```

```
Integrating rated preferences from OM matrices for each SOC…
```

[120]:
```
instance.parameters['a_pref_matrix']  # Nothing will change here since we␣
 ↪already had this merged correctly
```

[120]:
```
array([[ 6, 18, 12,  9],
       [20, 20, 16, 15],
       [19,  9,  9,  7],
       [ 3,  7,  3,  2],
       [ 1,  6, 10,  3],
       [ 9, 12, 11, 10],
       [16, 14, 14, 16],
       [13, 19,  7,  8],
       [11, 16, 19,  6],
       [ 7,  1,  4,  1],
       [18, 10, 15, 11],
       [ 4,  3,  6,  5],
       [ 2,  4,  8,  0],
       [10,  5,  1,  0],
       [12, 15,  2,  0],
       [14, 17, 13, 13],
       [15,  2,  5,  4],
       [17, 13, 18, 12],
       [ 8,  8, 20, 14],
       [ 5, 11, 17,  0]])
```

Once this method is run, we should now have an "a_pref_matrix" that is almost 100% accurate (the only drift should come from cadet preferences). Once you have the AFSC preferences file that is accurate, you can then update the qualification matrix using these preferences. The default qual matrix allows all cadets to be eligible for rated AFSCs and for the Space Force in general. It also restricts eligbility for Non-Rated AFSCs to be based on the cadets' degrees (CIP codes). We need to update it by restricting Rated/USSF eligibility down to volunteerism, and relaxing the AFOCD a bit for Non-Rated AFSCs (creating "exceptions" designated by "E"). This accomplished through this method:

[121]:
```
# Update qualification matrix from AFSC preferences (treating CFM lists as␣
↪"gospel" except for Rated/USSF)
instance.update_qualification_matrix_from_afsc_preferences()
```

```
3 Making 4 cadets ineligible for 'R4' by altering their qualification to 'I2'.
```

The message above isn't actually doing anything since those cadets were already ineligible since we've run the "instance.fix_generated_data()" function earlier.

Now, we should have a qualification matrix that agrees with the AFSC preference matrix in terms of eligibility. If it didn't, it would warn you and tell you where the discrepancies are so you can correct them or ignore them depending on what's going on. Once these two matrices are rectified, it's time to remove AFSCs from the cadets' preferences if they're truly ineligible for some career field. Be careful with this one as you need to make sure that the AFSC preferences and the qual matrix reflects accurate eligibility as this will remove cadets' choices!

It checks all three "sources of truth" in terms of eligibility (cadet/AFSC preferences and degree qual matrix) and if one of them says that the cadet is ineligible for a given AFSC, this method forces all three to reflect ineligibility. It is a rather "nuclear" approach, so again, be careful!

[122]:
```
# Removes ineligible cadets from all 3 matrices: degree qualifications, cadet␣
↪preferences, AFSC preferences
instance.remove_ineligible_choices()  # Nothing changed since we've already␣
↪done this!
```

```
Removing ineligible cadets based on any of the three eligibility sources
(c_pref_matrix, a_pref_matrix, qual)…
0 total adjustments.
```

This method also re-runs the parameter additions function which will correct the preference lists themselves (the dictionaries for each cadet/AFSC "key" that has the sorted AFSCs/cadets list as the "value"). Just always remember the difference between "a_pref_matrix" and "afsc_preferences" (similarly, "c_pref_matrix" and "cadet_preferences"). One is a 2-dimensional numpy array and the other is a dictionary.

From here, we want to "fill the gaps" in the matrix to create the final 1-N lists. We use the preference dictionaries ("afsc_preferences" and "cadet_preferences") to construct their corresponding matrices from scratch.

[123]:
```
# Take the preferences dictionaries and update the matrices from them (using␣
↪cadet/AFSC indices)
instance.update_preference_matrices()  # 1, 2, 4, 6, 7 -> 1, 2, 3, 4, 5␣
↪(preference lists need to omit gaps)
```

```
Updating cadet preference matrices from the preference dictionaries. ie. 1, 2,
4, 6, 7 -> 1, 2, 3, 4, 5 (preference lists need to omit gaps)
```

Once we have the final AFSC rankings, we can construct the afsc_utility matrix (that lives in "AFSCs Utility.csv") that converts the 1-N preference list into linear percentiles.

[124]:
```
# Convert AFSC preferences to percentiles (0 to 1)
instance.convert_afsc_preferences_to_percentiles()  # 1, 2, 3, 4, 5 -> 1, 0.8,␣
↪0.6, 0.4, 0.2
```

```
Converting AFSC preferences (a_pref_matrix) into percentiles (afsc_utility on
AFSCs Utility.csv)…
```

Until now, we've only been manipulating the cadet/AFSC preference matrices (information contained in "Cadets Preferences.csv" and "AFSC Preferences.csv"). We haven't adjusted the information from "Cadets.csv", since preferences are also contained there.

[125]:
```
# The "cadet columns" are located in Cadets.csv and contain the utilities/
↪preferences in order of preference
instance.update_cadet_columns_from_matrices()  # We haven't touched␣
↪"c_preferences" and "c_utilities" until now
```

```
Updating cadet columns (Cadets.csv…c_utilities, c_preferences) from the
preference matrix (c_pref_matrix)…
```

One last thing you'll want to do is update the utility matrices for cadets from the cadets data. These are the "utility" and "cadet_utility" matrices that live in "Cadets Utility.csv" and "Cadets Utility (Final).csv", respectfully.

[126]:
```
instance.update_cadet_utility_matrices_from_cadets_data()
```

```
Updating cadet utility matrices ('utility' and 'cadet_utility') from the
'c_utilities' matrix
```

The methods I've just described all contribute to the main parameters ("parameters") of the problem. The only other thing I've alluded to is getting the default value parameters setup in the "support/value parameters defaults/" sub-folder. We created this file earlier with the "Random_1" instance by exporting the randomly generated set back to excel as defaults, so it's there for reference with the name "Value_Parameters_Defaults_Random_1.xlsx". The best way to do this for a real class year is to simply copy the previous year's default file and make adjustments as needed. I did the heavy lifting for incorporating Rated/USSF with 2024 and so the 2025 file should be pretty similar, for example.

Once you create that default file, you'll just need to import your value parameters as defaults and it will initialize them for your given class year from that file. This should be one of the very first things you do because my code relies on the value parameters extensively.

[127]:
```
# Execute that function to get your set of value parameters
v = instance.import_default_value_parameters()  # "v =" prevents a lot of␣
↪output (v is meaningless)
```

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

For a real class year, if you followed those steps I described above then you should have all your files good to go and can start thinking about getting solutions! This was by far the biggest section since the data is definitely the most critical component of this process to get right. If you understand

the data, and have processed it all correctly, then solving the model and getting a solution is as easy as hitting "go" with whatever algorithm/model you're using.

5.8 Summary

This section described, in detail, the data used as part of afccp. There is a lot of it, and depending on your role in this project you'll probably need to know what most of it is and where it's stored. To recap, all of the instance files that contain the data are located in the "Model Input" folder of a particular problem instance. They are pulled into afccp using pandas and extracted into numpy arrays that live in the dictionaries "parameters" and "value_parameters". These dictionaries are attributes of the CadetCareerProblem (instance) object. Many new arrays, sets, and subsets are created through various afccp functions based on this data ("I^E", for example, is the indexed set of all eligible cadets for each AFSC).

Below is the recommended way of processing data with afccp after you've already run through all the pre-processsing steps I've described previously (fixing the data).

[128]:
```
# Import the "Random_1" instance
instance = CadetCareerProblem('Random_1')

# "Activate" a particular set of value parameters (since you can have multiple)
instance.set_value_parameters("VP")  # There could be "VP", "VP2", "VP3", etc.

# In case you change some parameter that the value parameters depend on
instance.update_value_parameters()  # AFSC quotas are a good example here

# Always make sure your data is good to go!
instance.parameter_sanity_check()

# Do stuff here
pass

# Export back to excel. If you just want the solutions (when you get them), use␣
 ↪optional argument below
instance.export_data()  # datasets="Solutions" for just exporting solutions
```

```
Importing 'Random_1' instance…
Instance 'Random_1' initialized.
Sanity checking the instance parameters…
Done, 0 issues found.
Exporting datasets ['Cadets', 'AFSCs', 'Preferences', 'Goal Programming', 'Value
Parameters', 'Solutions']
```

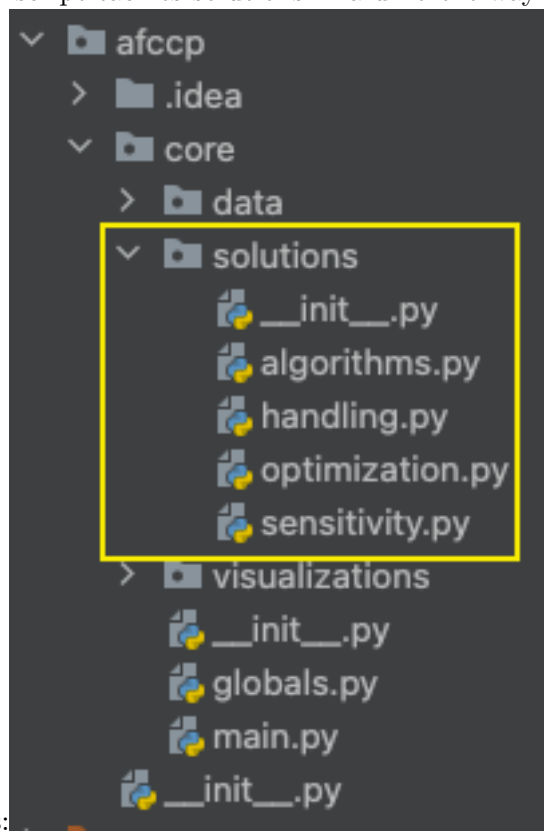
6 Solutions

As you've seen, afccp requires a substantial amount of data in order to work. All the various parameters, value parameters, hyper-parameters, and other "controls" are necessary to determine the best solution assignment of cadets to AFSCs. This is the whole point of afccp: obtain the right

solution! This section of the tutorial contains the information needed for you to work with afccp to calculate different solutions through various algorithms and models.

6.1 afccp.core.solutions

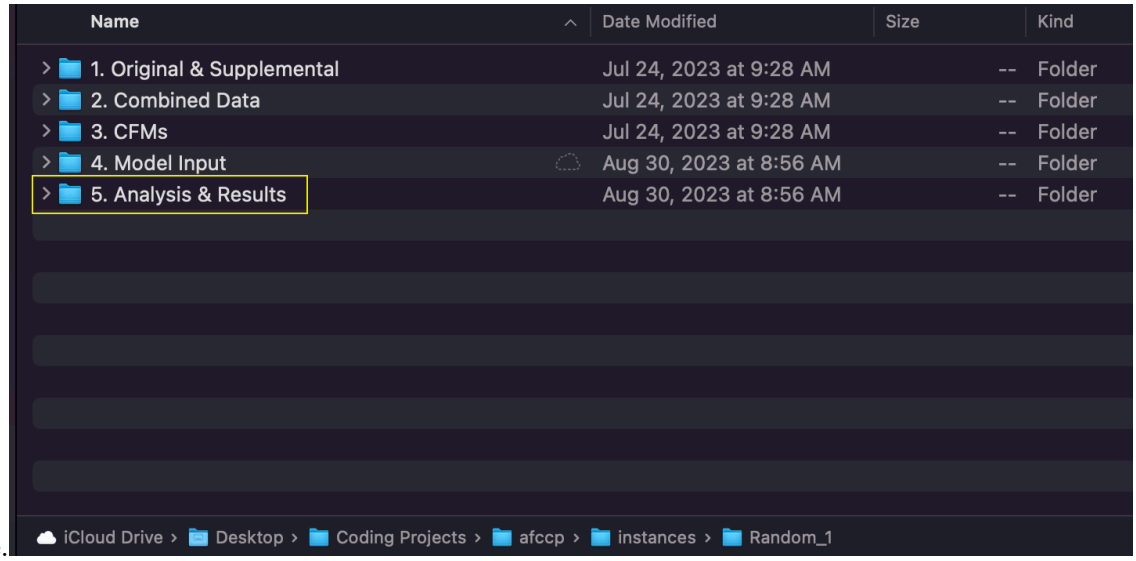
The "solutions" module of afccp.core contains the scripts and functions that handle the determination of a solution assignment of cadets to AFSCs. Each script tackles solutions in a different way. At a high-level, the solutions module is setup as follows:



The algorithms.py script contains the matching algorithms, SOC algorithms, and meta-heuristics that solve this problem. This is meant to contain all the non-optimization solvers for the problem. Alternatively, "optimization.py" contains the optimization models formulated using the python optimization library "pyomo". Pyomo is a great library that seemlessly links native python language with optimization components. The sensitivity.py script incorporates the optimization models to allow for a more algorithmic approach to solving them. We can iterate through different parameters and solve the models accordingly to provide interesting data perspectives to help inform why the solution is the way it is. Lastly, "handling.py" contains the many functions that all involve handling and processing the different solutions that are generated. We need to be able to evaluate these solutions and come up with key metrics that are used to determine which one to implement.

6.2 Structure

Before we dive into the different methods of generating a solution, it's important to discuss what exactly a solution is, or rather how a solution is represented, in the context of afccp. In the same way that the parameters (data) can all be extracted from csvs and pulled into python dictionaries, the solutions for a given instance are also pulled from csvs into a python dictonary. This is where the instance sub-folder "5. Analysis & Results" plays a

role. 

When working with the CadetCareerProblem object, several sub-folders of this folder will be created that hold many of the visualizations you may be working with. One of the different controls (mdl_p) is the dictionary key "save" which determines whether or not we should save charts that are created back to your folders. It is defaulted to True which means that there will be some charts saved into some of these sub-folders already, assuming that you've been following along with your own code.

As mentioned previously, we use dictionaries to hold solutions and solution elements. There are two instance attributes that contain this information: "solution" and "solutions". The first, "solution", is a dictionary containing all the components and metrics of the current activated solution. In the same way that we activate a particular set of value parameters, we also activate a particular solution to analyze for the problem. The second attribute "solutions" is a dictionary of solution dictionaries. The keys are the names of each solution and the values are that particular solution's component dictionary. Confusing, yes, but it will hopefully be made clear soon.

[129]:
```
# Showing that there currently are no solutions available to this "Random_1" instance
print("Current activated solution (instance.solution):", instance.solution)
print("Current set of solutions (instance.solutions):", instance.solutions)
```

```
Current activated solution (instance.solution): None
Current set of solutions (instance.solutions): None
```

We have not yet created any solutions, so none will appear in the "Analysis & Results" folder either. Let's start by creating a random solution and then exporting it back to excel as an example.

[130]:
```
# Generate random solution
instance.generate_random_solution()

# Export all data back to excel
instance.export_data()
```

```
Generating random solution…
New Solution Evaluated.
Measured exact VFT objective value: 0.6844.
```

```
Global Utility Score: 0.6153. 0 / 0 AFSCs fixed. 0 / 0 AFSCs reserved. 0 / 0
cadets correctly pulled from alternate list.
Blocking pairs: 12. Unmatched cadets: 0. Ineligible cadets: 0.
Exporting datasets ['Cadets', 'AFSCs', 'Preferences', 'Goal Programming', 'Value
Parameters', 'Solutions']
```

If you now look at your Analysis & Results folder, you should have a csv file titled "Random_1 Solutions.csv". This dataset contains a column for the cadet indices (0 through 19 in our case) accompanied by the solution columns themselves. There could be as many solution columns as desired stored within this dataset. Right now, we have one solution "Random" containing an assortment of randomly selected AFSCs for each cadet.

Cadet	Random
0	R2
1	R2
2	R1
3	R3
4	R3
5	R2
6	R1
7	R4
8	R4
9	R3
10	R3
11	R1
12	R2
13	R3
14	R1
15	R4
16	R2
17	R1
18	R2
19	R2

A couple of notes: This is a picture of my randomly generated solution and won't match your output (it's random, after all)! Additionally, don't confuse the solution name "Random" with the instance name "Random_1". We're dealing with a randomly generated problem instance of cadets/AFSCs and now we've just introduced our first method of generating a solution: "Random"! These are two different random components. Let's now "start over" and re-import the Random_1 problem instance so we can see how this solution is initially stored in the CadetCareerProblem object.

[131]:
```
# Re-import "Random_1"
instance = CadetCareerProblem("Random_1")

# "Activate" a particular set of value parameters (since you can have multiple)
instance.set_value_parameters("VP")  # There could be "VP", "VP2", "VP3", etc.
```

```
# In case you change some parameter that the value parameters depend on
instance.update_value_parameters()  # AFSC quotas are a good example here

# Always make sure your data is good to go!
instance.parameter_sanity_check()
```

```
Importing 'Random_1' instance…
Instance 'Random_1' initialized.
Sanity checking the instance parameters…
Done, 0 issues found.
```

To hopefully highlight the good practices of working with afccp, I've included the other methods I recommend to have on here after you import a dataset. This instance now has solution data stored in the Analysis & Results folder, which is contained in the instance attribute "solutions". We have not "activated" a solution, however.

[132]:
```
# Showing the names of the solutions we currently have
print("Names of solutions:", instance.solutions.keys())

# Contents of the "Random" solution
print("'Random' solution components:", instance.solutions['Random'].keys())

# Current solution (haven't selected it yet)
print("Current activated solution components:", instance.solution)
```

```
Names of solutions: dict_keys(['Random'])
'Random' solution components: dict_keys(['j_array', 'name', 'afsc_array'])
Current activated solution components: None
```

You might be wondering why I don't just automatically select "Random" as the current solution. When you're doing this for the real thing, you'll likely be dealing with multiple solutions and I want you to be sure you know which solution you're dealing with. That's why I force the analyst to select a specific solution.

[133]:
```
# Select the "Random" solution (just like the value parameters)
instance.set_solution("Random")  # By default, the first solution is selected␣
 ↪if no name is provided

# Current solution
print("\nCurrent activated solution components (first 5):", list(instance.
 ↪solution.keys())[:5])

# Number of solution components (after evaluating the solution)
print("Number of solution components:", len(instance.solution.keys()))
```

```
Solution Evaluated: Random.
Measured exact VFT objective value: 0.6331.
Global Utility Score: 0.5301. 0 / 0 AFSCs fixed. 0 / 0 AFSCs reserved. 0 / 0
cadets correctly pulled from alternate list.
```

```
Blocking pairs: 15. Unmatched cadets: 0. Ineligible cadets: 0.

Current activated solution components (first 5): ['j_array', 'name',
'afsc_array', 'x', 'objective_measure']
Number of solution components: 75
```

When you activate a solution, or generate one for that matter, afccp also evaluates it according to the parameters of the instance. This is what's happening for the first few lines of output above (before my explicit print statements). The functions that do this live in afccp.core.solutions.handling, which I highly encourage you to explore especially when you're thinking about tracking your own metrics (I already have a lot of them calculated there!). These metrics then get placed into the solution dictionary, and I've printed the first 5 keys of that dictionary above. I also highlight the reason I'm only printing the first five by including the total number of solution dictionary keys right below that!

[134]:
```
# Name of the solution
print("Name of solution ('name'):", instance.solution['name'])

# Array of AFSC names that each cadet is assigned to
print("\nAFSC names that each cadet is assigned ('afsc_array'):", instance.
 ↪solution['afsc_array'])

# Array of AFSC indices that each cadet is assigned to
print("\nAFSC indices that each cadet is assigned ('j_array'):", instance.
 ↪solution['j_array'])

# Binary X-matrix where the rows are cadets and columns are AFSCs
print("\nX-Matrix ('x'):\n", instance.solution['x'])
```

```
Name of solution ('name'): Random

AFSC names that each cadet is assigned ('afsc_array'): ['R2' 'R2' 'R1' 'R3' 'R3'
'R2' 'R1' 'R4' 'R4' 'R3' 'R3' 'R1' 'R2' 'R3'
 'R1' 'R4' 'R2' 'R1' 'R2' 'R2']

AFSC indices that each cadet is assigned ('j_array'): [1 1 0 2 2 1 0 3 3 2 2 0 1
2 0 3 1 0 1 1]

X-Matrix ('x'):
 [[0 1 0 0]
 [0 1 0 0]
 [1 0 0 0]
 [0 0 1 0]
 [0 0 1 0]
 [0 1 0 0]
 [1 0 0 0]
 [0 0 0 1]
 [0 0 0 1]
```

```
[0 0 1 0]
[0 0 1 0]
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[1 0 0 0]
[0 0 0 1]
[0 1 0 0]
[1 0 0 0]
[0 1 0 0]
[0 1 0 0]]
```

As you can probably tell "afsc_array", "j_array", and "x" all contain the same information just presented in three different ways. The "afsc_array" is what is ultimately stored in the "Solutions.csv" file because of its ease of interpretability. The "x" matrix is what optimization models output and it is how the solution is evaluated according to the VFT objective hierarchy (which I will describe later). By far my preferred method of representing a solution, which is used almost everywhere across afccp (outside of the two cases I just described), is "j_array". Utilizing the power of numpy indexing allows us to quickly and efficiently extract necessary information from "j_array".

[135]:
```
# Array of cadets assigned to each AFSC stored in a dictionary
{j: np.where(instance.solution['j_array'] == j)[0] for j in instance.
 ↪parameters['J']}
```

[135]:
```
{0: array([ 2,  6, 11, 14, 17]),
 1: array([ 0,  1,  5, 12, 16, 18, 19]),
 2: array([ 3,  4,  9, 10, 13]),
 3: array([ 7,  8, 15])}
```

I use numpy.where extensively across afccp, and the above line of code is an example of why I like using indices. By the way, the above information is already incorporated in the solution dictionary (see below).

[136]:
```
# Array of cadets assigned to each AFSC stored in the "cadets_assigned"␣
 ↪component
instance.solution['cadets_assigned']
```

[136]:
```
{0: array([ 2,  6, 11, 14, 17]),
 1: array([ 0,  1,  5, 12, 16, 18, 19]),
 2: array([ 3,  4,  9, 10, 13]),
 3: array([ 7,  8, 15])}
```

For more information on what specifically the code currently tracks, I again invite you to look at the functions in the "handling.py" script. Specifically, the "evaluate_solution" function at the top of the script is the main function that obtains this information.

The naming convention I use when determining solution is by incorporating both the solution method and iteration. The only method I've discussed thus far is "Random", which is why the current solution is called "Random". I've only generated one random solution too. By default, the

first solution you generate through whatever method will simply be named using that method. If I were to generate a second solution, it would be called "Random_2" (see below).

[137]:
```
# Generate another random solution
instance.generate_random_solution()

# Current activated solution name
print("\nCurrent activated solution name:", instance.solution['name'])

# Current solutions available
print("Curretn solutions available:", instance.solutions.keys())
```

```
Generating random solution…
New Solution Evaluated.
Measured exact VFT objective value: 0.6354.
Global Utility Score: 0.5299. 0 / 0 AFSCs fixed. 0 / 0 AFSCs reserved. 0 / 0
cadets correctly pulled from alternate list.
Blocking pairs: 14. Unmatched cadets: 0. Ineligible cadets: 0.

Current activated solution name: Random_2
Curretn solutions available: dict_keys(['Random', 'Random_2'])
```

In the same way I check if a set of value parameters is unique, I also make sure a new solution is unique too. There is a very low probability that two randomly generated solutions are the same, but if that did happen we wouldn't see a "Random_2" solution above. Instead, the code would simply kick back the "Random" solution as the current solution since it is equivalent to the new one. This does play a critical role with the deterministic models/algorithms you will see soon as I don't want many copies of the same solution being depicted as unique.

6.3 Matching Algorithms

Now that we've discussed how a solution is represented in the data and stored within the CadetCareerProblem object, we can talk about the methods of generating solutions (ones that are hopefully a bit better than throwing darts at the board with the "Random" method). The first series of solution techniques I'll describe are matching algorithms. The first of which is the classic "Hospital/Residents" (HR) deferred acceptance algorithm (DAA).

6.3.1 Classic HR

We formulate the AFSC/Cadet matching problem as a Hospital/Residents problem where each entity posesses an ordinal preference list for members of the other. It's a simple algorithm, and the function itself is located at "afccp.core.solutions.algorithms.classic_hr".

[138]:
```
# Run the "Classic HR" algorithm on the "Random_1" instance
s = instance.classic_hr({'ma_printing': True})
```

```
Modeling this as an H/R problem and solving with DAA…

Iteration 1
Proposals: {'R1': 4, 'R2': 7, 'R3': 8, 'R4': 1}
```

```
Matched {'R1': 4.0, 'R2': 7.0, 'R3': 4.0, 'R4': 1.0}
Rejected {'R1': 0.0, 'R2': 0.0, 'R3': 4.0, 'R4': 0.0}

Iteration 2
Proposals: {'R1': 7, 'R2': 8, 'R3': 4, 'R4': 1}
Matched {'R1': 7.0, 'R2': 8.0, 'R3': 4.0, 'R4': 1.0}
Rejected {'R1': 0.0, 'R2': 0.0, 'R3': 4.0, 'R4': 0.0}
New Solution Evaluated.
Measured exact VFT objective value: 0.8692.
Global Utility Score: 0.7895. 0 / 0 AFSCs fixed. 0 / 0 AFSCs reserved. 0 / 0
cadets correctly pulled from alternate list.
Blocking pairs: 0. Unmatched cadets: 0. Ineligible cadets: 0.
```

The "ma_printing" parameter controls whether or not the algorithm should print iteration-specific information (it's defaulted to False). As you can see, with 20 cadets and 4 AFSCs this is a pretty short algorithm. With my particular randomly generated cadets, only four cadets were rejected in iteration 1 (all by R3). In the second iteration, three of these cadets propose to R1 and one proposes to R2. All four are accepted and the algorithm concludes.

[139]:
```
# Current solution (HR)
print("Solution Name:", instance.solution['name'])
print("Solution Array:", instance.solution['afsc_array'])
```

```
Solution Name: HR
Solution Array: ['R1' 'R1' 'R2' 'R1' 'R1' 'R2' 'R2' 'R3' 'R4' 'R2' 'R2' 'R3'
'R1' 'R3'
 'R3' 'R1' 'R2' 'R2' 'R2' 'R1']
```

6.3.2 SOC Rated Algorithm

One very important solution technique we need to address is the specific rated algorithm we employ to pre-process cadets into rated career fields in the spirit of the DAA. Although the DAA in its true form does not work for all career fields across the Air & Space Force (due to degree requirements and popularity issues), it does work really well if we use it on the rated career fields in a unique way. Pilot, and the other three rated career fields to a lesser extent, is a very "political" career field and one that must be handled with great care. Any kind of matches are fair game in an optimization model, and people don't always get matched to their desired career fields in order of merit unless we explicitly enforce it. One huge benefit of the DAA is that it's simple and defendable. It is fair to all cadets by directly adhering to the career field rankings. Because we want a defendable solution (especially when it comes to pilot), we apply the DAA to rated cadets from both SOCs to constrain AFSCs for cadets based on what they would have received as a result of the "complete" DAA.

As discussed, this algorithm strips out all non-rated AFSC choices and purely looks at the rated choices, ordered amongst themselves. We then run the DAA, matching cadets to their rated AFSCs from each SOC honoring their specific rated PGL targets. If the DAA matches a cadet to a rated AFSC that was also their true first choice then that cadet is definitvely matched to that AFSC. If the cadet is matched to a rated AFSC but that AFSC was not their first choice, we reserve that slot for them but allow them to compete in the optimization model for their more desired AFSCs.

[140]:
```
# Run the SOC rated algorithm for ROTC
s = instance.soc_rated_matching_algorithm({"soc": "rotc", "ma_printing": True})␣
 ↪ # "s =" prevents lots of output
```

```
Solving the rated matching algorithm for ROTC cadets…

Iteration 1
Proposals: {'R4': 5}
Matched {'R4': 2}
Rejected {'R4': 3}
```

As you can see from the output, we're evaluating three new solutions. I use the solutions dictionary to store my results from the rated algorithm for both SOCs. One solution contains the information on cadets with reserved slots for rated AFSCs, one solution contains the information on cadets that were definitvely matched to a rated AFSC, and the last one simply combines the two. You can see these solutions below for ROTC

[141]:
```
for s_name in instance.solutions:

    if "Rated" in s_name:
        print("Solution name: '" + s_name + "'")
        print("Solution:", instance.solutions[s_name]['afsc_array'])
```

```
Solution name: 'Rated ROTC HR (Reserves)'
Solution: ['*' '*' '*' 'R4' '*' '*' '*' '*' '*' '*' '*' '*' '*' '*' '*' '*' '*'
'*'
 '*' '*']
Solution name: 'Rated ROTC HR (Matches)'
Solution: ['*' '*' '*' '*' '*' '*' '*' '*' 'R4' '*' '*' '*' '*' '*' '*' '*' '*'
'*'
 '*' '*']
Solution name: 'Rated ROTC HR'
Solution: ['*' '*' '*' 'R4' '*' '*' '*' '*' 'R4' '*' '*' '*' '*' '*' '*' '*' '*'
'*'
 '*' '*']
```

In my simple "Random_1" data example, we only have one rated AFSC: R4. There are 2 ROTC slots for this AFSC. One was a case where the cadet had it as their first choice and was matched to it, while the other didn't and therefore a slot is reserved for them in case they don't get a more preferred AFSC. Also in my data example, USAFA doesn't have any slots for R4 and so that algorithm is effectively useless (but we should still run it to have the components).

[142]:
```
print("USAFA Targets:", {p['afscs'][j]: p['usafa_quota'][j] for j in p['J']}) ␣
 ↪# None for R4!
print("ROTC Targets:", {p['afscs'][j]: p['rotc_quota'][j] for j in p['J']})  #␣
 ↪2 for R4!
print("")  # just adding a space

# Run the USAFA rated algorithm
```

```
s = instance.soc_rated_matching_algorithm({"soc": "usafa", "ma_printing":␣
↪True})  # "s =" prevents lots of output
```

```
USAFA Targets: {'R1': 1.0, 'R2': 1.0, 'R3': 1.0, 'R4': 0.0}
ROTC Targets: {'R1': 5.0, 'R2': 4.0, 'R3': 3.0, 'R4': 2.0}

Solving the rated matching algorithm for USAFA cadets…

Iteration 1
Proposals: {'R4': 11}
Matched {'R4': 0}
Rejected {'R4': 11}
```

We've now run these two algorithms (USAFA/ROTC) and can incorporate their information into our instance parameters. This will be used later on within the optimization models, but worth discussing here. In order to incorporate these solutions into the parameters, we need to call a function!

[143]:
```
# Integrate the Rated algorithm solutions into "instance.parameters"
instance.incorporate_rated_algorithm_results()
```

```
Incorporating rated algorithm results…
Rated SOC Algorithm Results [USAFA Matched Cadets: 0. USAFA Reserved Cadets: 0.
ROTC Matched Cadets: 1. ROTC Reserved Cadets: 1.]
```

This creates two new dictionaries: "J^Fixed" and "J^Reserved". They are used to constrain the available AFSCs for each cadet as a result of the SOC-specific rated DAA described above.

[144]:
```
# Cadet 8 must be matched to the AFSC at index 3 (R4)
print("'J^Fixed':", instance.parameters['J^Fixed'])
```

```
'J^Fixed': {8: 3}
```

"J^Fixed" is a dictionary where the keys are the cadets with "fixed" AFSCs and the values are the AFSC indices that they are fixed to. In a similar, but slightly different way, "J^Reserved" is a dictionary where the keys are the cadets with "reserved" AFSCs. For each key here, however, the value is an ordered list of the cadet's preferences up to and including the rated AFSC they're reserved for. This ensures that the cadet receives one of these AFSCs, since the rated AFSC is reserved for them.

[145]:
```
# Cadet 3 must receive either AFSC 0 (R1), 2 (R3), or 3 (R4) -> R4 is reserved␣
↪for them
print("'J^Reserved':", instance.parameters['J^Reserved'])
```

```
'J^Reserved': {3: array([0, 2, 3])}
```

6.3.3 ROTC Rated Board Original Algorithm

This one is not too important anymore, but it is included in the code since it helped my case to convince ROTC that this new rated algorithm was a good idea. I'm not going to discuss it too much, but it uses the ROTC rated OM dataset and the rotc rated interest dataset: Random_1

ROTC Rated Interest.csv. Their board then commenced in different phases and if you're really I suggest you read the function itself (afccp.core.solutions.algorithms.rotc_rated_board_original) and then reach out with questions if you're still unclear!

[146]:
```
# Run the algorithm
s = instance.rotc_rated_board_original()  # "s =" prevents lots of output
```

```
Running status quo ROTC rated algorithm…

Phase 1 High OM & High Interest

Phase 2 High OM & Med Interest
R4 Phase Matched: 2   --->   Total Matched: 2 / 2.0
New Solution Evaluated.
Measured exact VFT objective value: 0.2811.
Global Utility Score: 0.0719. 1 / 1 AFSCs fixed. 0 / 1 AFSCs reserved. 0 / 0
cadets correctly pulled from alternate list.
Blocking pairs: 19. Unmatched cadets: 18. Ineligible cadets: 0.
```

Now that we've incorporated the rated algorithm results within our parameters, you'll notice the solution evaluation output reflects that we should have one cadet with a fixed AFSC and one cadet with a reserved AFSC. In the original rated board algorithm we just ran, one of those constraints was adequately met and the other wasn't. I'll discuss what the alternate list means later on.

6.4 Meta-heuristics

This section describes the meta-heuristics we have to solve the problem. Meta-heuristics refer to solution techniques that use unconventional algorithms to solve optimization models. They are typically meant for really complicated optimization models that are too computationally expensive to solve with a conventional "global" solver. Right now, the only meta-heuristic I have in the code is the genetic algorithm (GA). Genetic algorithms are great methods of finding initial solutions that are then "evolved" into better and better solutions. This algorithm is currently used in conjunction with the VFT pyomo model (specifically, the "Approximate Model" I discuss in my thesis) to get closer to the optimal solution since I don't currently have a method of finding the global optimal solution to the VFT "Exact Model". If these terms are confusing, please reference section 3.4.1 of my master's thesis.

6.4.1 VFT Genetic Algorithm

For more information on this algorithm specifically, you can find it at "afccp.core.solutions.algorithms.vft_genetic_algorithm". One big note here is that I haven't found a really good way of reconciling the many constraints of the optimization model(s) with the GA. When we were just solving the NRL process, before combining Rated and USSF for FY2024 (so just for 2023 really), the only constraints I had placed were the AFSC objective constraints and some cadet utility constraints (top 10%). I will describe how these constraints are handled later on using the "con_fail_dict" (constraint fail dictionary). This is the reason why there are initially a lot of 0s in the initial population of solutions below (constraint violations result in a fitness score of 0). Since this is a small & simple example, however, we quickly determine feasible solutions right out the gate.

[147]:
```
# Solve the GA using the VFT objective function as the fitness function
s = instance.vft_genetic_algorithm({"initialize": False, "ga_max_time": 20})
```

```
Running Genetic Algorithm with no initial solutions (not advised!)…
Initial Fitness Scores [0.7465512  0.74607654 0.         0.         0.
0.
 0.         0.         0.         0.         0.         0.        ]
10% complete. Best solution value: 0.8046
Average evaluation time for 12 solutions: 0.0138 seconds.
Average generation time: 0.0147 seconds.
20% complete. Best solution value: 0.8363
Average evaluation time for 12 solutions: 0.014 seconds.
Average generation time: 0.0149 seconds.
30% complete. Best solution value: 0.8507
Average evaluation time for 12 solutions: 0.0141 seconds.
Average generation time: 0.015 seconds.
40% complete. Best solution value: 0.8593
Average evaluation time for 12 solutions: 0.0142 seconds.
Average generation time: 0.015 seconds.
50% complete. Best solution value: 0.8593
Average evaluation time for 12 solutions: 0.0142 seconds.
Average generation time: 0.015 seconds.
60% complete. Best solution value: 0.8593
Average evaluation time for 12 solutions: 0.0142 seconds.
Average generation time: 0.015 seconds.
70% complete. Best solution value: 0.8593
Average evaluation time for 12 solutions: 0.0142 seconds.
Average generation time: 0.015 seconds.
80% complete. Best solution value: 0.8593
Average evaluation time for 12 solutions: 0.0142 seconds.
Average generation time: 0.015 seconds.
90% complete. Best solution value: 0.8593
Average evaluation time for 12 solutions: 0.0142 seconds.
Average generation time: 0.015 seconds.
100% complete. Best solution value: 0.8593
Average evaluation time for 12 solutions: 0.0142 seconds.
Average generation time: 0.015 seconds.
End time reached in 20.01 seconds.
New Solution Evaluated.
Measured exact VFT objective value: 0.8593.
Global Utility Score: 0.7203. 1 / 1 AFSCs fixed. 1 / 1 AFSCs reserved. 0 / 0
cadets correctly pulled from alternate list.
Blocking pairs: 9. Unmatched cadets: 0. Ineligible cadets: 0.
```

6.4.2 Genetic Matching Algorithm

Ian Macdonald had the idea to create a GA in order to iteratively determine the optimal capacities for each AFSC to find a solution that is as stable as possible. This algorithm works if we have a

designated range on the number of cadets for each AFSC. Essentially, we need a sizeable "surplus" of cadets above the PGL targets. For sake of time, I won't go into too much detail here but the algorithm is shown below.

[148]:
```
# Because this is a small/easy problem, we easily find a stable solution! (GMA␣
↪isn't necessary)
s = instance.genetic_matching_algorithm({'gma_printing': True})  # Turns on␣
↪print statements for the GMA
```

```
Generation 0 Fitness [0. 0. 2. 3.]
Final capacities: [7. 7. 4. 3.]
Modeling this as an H/R problem and solving with DAA…
New Solution Evaluated.
Measured exact VFT objective value: 0.8742.
Global Utility Score: 0.767. 1 / 1 AFSCs fixed. 1 / 1 AFSCs reserved. 0 / 0
cadets correctly pulled from alternate list.
Blocking pairs: 0. Unmatched cadets: 0. Ineligible cadets: 0.
```

6.5 Optimization Techniques

This section is here to describe the various optimization model components we have to solve the CadetCareerProblem. At the time I'm writing this tutorial, the VFT methodology doesn't work on the current combined classification problem (Rated, USSF, NRL). The current best model, that was also implemented for the class of 2024, is the "GUO" model. This is the global utility optimization model that OLEA has advertised, and it's represented in the code as the "assignment problem model". We'll discuss that soon.

6.5.1 VFT Model

For more information on the VFT model, please review my thesis paper discussed earlier in this tutorial. It describes in detail the difference between the "Approximate" model and the "Exact" model. Here, I will solve this random instance using both.

[149]:
```
# Solve with the "Approximate" model  (These three controls I've added are also␣
↪the defaults!)
s = instance.solve_vft_pyomo_model({"approximate": True, "pyomo_max_time": 10,␣
↪"solver_name": "cbc"})
```

```
Building VFT Model…
Solving Approximate VFT Model instance with solver cbc…
Model solved in 10.51 seconds.
New Solution Evaluated.
Measured exact VFT objective value: 0.8376.
Global Utility Score: 0.7588. 1 / 1 AFSCs fixed. 1 / 1 AFSCs reserved. 0 / 0
cadets correctly pulled from alternate list.
Blocking pairs: 4. Unmatched cadets: 0. Ineligible cadets: 0.
```

[150]:
```
# Solve with the "Exact" model (On a small problem, this actually works quite␣
↪well!)
```

```
s = instance.solve_vft_pyomo_model({"approximate": False, "pyomo_max_time": 20,␣
 ↪"solver_name": "ipopt"})
```

```
Building VFT Model…
Solving Exact VFT Model instance with solver ipopt…
Model solved in 0.6 seconds.
New Solution Evaluated.
Measured exact VFT objective value: 0.8465.
Global Utility Score: 0.7559. 1 / 1 AFSCs fixed. 1 / 1 AFSCs reserved. 0 / 0
cadets correctly pulled from alternate list.
Blocking pairs: 5. Unmatched cadets: 0. Ineligible cadets: 0.
```

[151]:
```
instance.solutions.keys()
```

[151]:
```
dict_keys(['Random', 'Random_2', 'HR', 'Rated ROTC HR (Reserves)', 'Rated ROTC
HR (Matches)', 'Rated ROTC HR', 'Rated USAFA HR (Reserves)', 'Rated USAFA HR
(Matches)', 'Rated USAFA HR', 'ROTCRatedBoard', 'VFT_Genetic', 'HR_2', 'A-VFT',
'E-VFT'])
```

We just solved the VFT model twice (once with the "Approximate Model" and another with the "Exact Model"). As you can probably tell, both solutions are represented from two different methods: "A-VFT" and "E-VFT". I'll let you guess which one is which. Since it is a very small problem, they both do exactly what they're supposed to: the Exact model finds the optimal solution to the "real" (Exact) VFT objective function (beating the Approximate model) and they both produce integer solutions (something that isn't true when the problem gets bigger). Additionally, these models are deterministic, meaning they will produce the same solution every time. This is not true for the GA solutions! The next section describes my VFT methodology using the Approximate model.

6.5.2 VFT Main Methodology

I had this weird "con_fail_dict" (constraint fail dictionary) feature of the model to allow some AFSC objective constraints to be broken by AT MOST the amount they were broken as a product of the non-integer pyomo output of the VFT Approximate Model. This will be made more clear in the third bullet below.

My current pyomo VFT model does not produce integer solutions using cbc. I really don't understand why this is and it's certainly the motivation for future AFIT research using that model. My methodology that I came up with (only for NRL cadets/AFSCs at the time) is as follows:

1. [Solve the VFT Model] Solve the VFT Approximate Model for 10 seconds using the cbc solver. The time limit of 10 seconds allows us to find a pretty good solution (not optimal, but close-ish enough) to the Exact Model. As a reminder, the Exact Model is the real objective function using the true number of assigned cadets, whereas the Approximate Model approximates that number (specifically, it uses the "Estimated" number of cadets from AFSCs.csv). The optimal solution to the Approximate Model, therefore, is not the optimal solution to the Exact Model. One other reason to cut the Approximate Model short is that we were never going to find the optimal solution to the "real" problem anyway!

2. [Round the X variables] Ok, so as a result of stopping it at 10 seconds we don't get integer

values for all the variables we need. I therefore have to round them to make sure cadets receive one and only one AFSC. By doing this, I am oftentimes breaking constraints. This was generally ok, however, since constraints were primarily meant as guidelines to ensure quality distributions across AFSCs. The class of 2023 and 2024 were very different problems in many ways, and the fact that we had ~150 extra cadets above the PGL in 2023 allowed for this to be less of an issue. Since we're 4 short in 2024, rounding these variables could have produced other problems.

3. [Create "con_fail_dict"] As mentioned previously, this constraint fail dictionary was used to determine the extent to which we broke the AFSC objective constraints from whatever solution(s) pyomo produced. This dictionary is then used in the GA to allow more flexibility when evaluating the solutions (or "chromosomes") fitness values. Fitness is based directly on the true (Exact, not Approximate) VFT objective function, and constraint violations result in a fitness score of 0. I played around with providing a tolerance but ultimately decided to aggressively restrict constraint violations. This is why con_fail_dict is so important, since it allows the initial solutions to be feasible.
4. [Initialize GA Population] In my "real" solution methodology, I solve the VFT model several different times with slightly different settings on the overall weights for cadets/AFSCs. This essentially creates my initial population of solutions that I use to then create the "con_fail_dict" from. These solutions are then fed into the GA.
5. [Run GA]

6.5.3 Assignment Problem Model- "Global Utility Optimization" (GUO)

The GUO model refers to solving the generalized assignment problem formulation that seeks the optimal assignment of a set of individuals, $\mathcal{I}$, to a set of jobs, $\mathcal{J}$. That is, we find the solution assignment with minimum cost such that every individual receives one and only one job and every job does not exceed its maximum capacity. The formulation of this problem is shown below:

$$\text{minimize} \quad \sum_{i \in \mathcal{I}} \sum_{j \in J} c_{ij} x_{ij}$$

$$\text{subject to} \quad \sum_{j \in \mathcal{J}} x_{ij} = 1 \quad \forall i \in \mathcal{I}$$

$$\sum_{i \in \mathcal{I}} x_{ij} \leq b_j \quad \forall j \in \mathcal{J}$$

$$x_{ij} \in \{0,\ 1\} \quad \forall i \in \mathcal{I},\ j \in \mathcal{J}$$

Again, this is the "generalized" formulation of this academic problem. Like the "original model" (the model AFPC used up until the class of 2023), there are many additional constraints that can be added to this model. The addition of these constraints is one key difference between "GUO" in our context and the generalized framework. Another key difference is the utility matrix itself. Unlike the original AFPC model, this "new" utility matrix is much more direct with its representation of quality for the career fields. This is the merged cadet/AFSC "global utility" matrix I've referenced in the Data section of this tutorial. Quick note: rather than a "min cost" function, it's a "max utility" function. This is a small, but necessary, clarification!

[152]:
```
# Solve the GUO model!
s = instance.solve_guo_pyomo_model()
```

```
Building assignment problem (GUO) model…
Done. Solving model…
Solving GUO Model instance with solver cbc…
Model solved in 0.08 seconds.
New Solution Evaluated.
Measured exact VFT objective value: 0.8553.
Global Utility Score: 0.7718. 1 / 1 AFSCs fixed. 1 / 1 AFSCs reserved. 0 / 0
cadets correctly pulled from alternate list.
Blocking pairs: 4. Unmatched cadets: 0. Ineligible cadets: 0.
```

As you can see, it solves very fast on a small problem! On a larger problem, too, it still solves pretty quickly. Although the value functions and weights from the VFT model aren't used in the GUO model, the constraints certainly are! The value parameters, therefore, are still very important in this model as they control the constraints applied to the problem. This model can be found in the assignment_model_build() function at afccp.core.solutions.optimization. If you look at that function, you can see an additional function "common_optimization_handling" that does exactly what it sounds like: handles the features of the pyomo model that are common across all of my optimization models (VFT, original, GUO). Mostly, this function handles the definition of the "x" variable and the many potential constraints associated with it. This includes the fixed/reserved slot constraints I discussed in the "SOC Rated Algorithm" section above.

[153]:
```
# Run the USAFA/ROTC rated algorithm
s = instance.soc_rated_matching_algorithm({"soc": "usafa"})  # "s =" prevents␣
 ↪lots of output
s = instance.soc_rated_matching_algorithm({"soc": "rotc"})  # "s =" prevents␣
 ↪lots of output

# Integrate the Rated algorithm solutions into "instance.parameters"
instance.incorporate_rated_algorithm_results()

# Solve the GUO model with the rated algorithm results!
s = instance.solve_guo_pyomo_model()
```

```
Solving the rated matching algorithm for USAFA cadets…
Solving the rated matching algorithm for ROTC cadets…
Incorporating rated algorithm results…
Rated SOC Algorithm Results [USAFA Matched Cadets: 0. USAFA Reserved Cadets: 0.
```

```
ROTC Matched Cadets: 1. ROTC Reserved Cadets: 1.]
Building assignment problem (GUO) model…
Done. Solving model…
Solving GUO Model instance with solver cbc…
Model solved in 0.05 seconds.
New Solution Evaluated.
Measured exact VFT objective value: 0.8553.
Global Utility Score: 0.7718. 1 / 1 AFSCs fixed. 1 / 1 AFSCs reserved. 0 / 0
cadets correctly pulled from alternate list.
Blocking pairs: 4. Unmatched cadets: 0. Ineligible cadets: 0.
```

We've now incorporated the rated algorithm results within the GUO model. One thing I haven't mentioned yet is how we give even more special attention to pilots with an "alternate list" idea. The fixed/reserved slot idea works really well to "protect" those individuals that would get rated slots based purely on their rated OM, by giving them a chance to get their more preferred AFSC. If they don't get that more preferred AFSC, they have their reserved rated AFSC to fall back on. What this current methodology does not address, however, is who backfills those reserved slots. Since pilot slots are so political, I want to make sure that the individuals backfilling a vacant reserved slot are the ones next up on the list. That is to say, the optimization model does not get to decide based on global utility. We adhere to the pilot OM list exclusively! To enact this principle, we simply need to pass another parameter when we incorporate the rated algorithm results.

[154]:
```
# R4 is the only rated AFSC in my case, and so we have to pretend that is on␣
↪the same level as pilot
instance.incorporate_rated_algorithm_results({'special_afscs_with_alternates':␣
↪['R4']})
```

```
Incorporating rated algorithm results…
AFSC 'R4' alternate list: []
0 alternates; 14 extra reserved.
Rated SOC Algorithm Results [USAFA Matched Cadets: 0. USAFA Reserved Cadets: 0.
ROTC Matched Cadets: 1. ROTC Reserved Cadets: 1.]
```

The alternate list is based on the individuals "next in line" who also have this AFSC as their first choice. This makes a lot of sense for pilots, but not necessarily for my random data example. There's only one person in the data that has "R4" as their first choice (the same individual who is matched to it from ROTC above) and so there is no alternate list for this example.

If we try to solve the model, we will catch an error telling us it's not possible because we don't have any cadets on R4's alternate list!

[155]:
```
# Solve the GUO model with the rated algorithm results (including alternate␣
↪list concept)!
s = instance.solve_guo_pyomo_model()
```

```
Building assignment problem (GUO) model…
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Input In [155], in <cell line: 2>()
```

```
      1 # Solve the GUO model with the rated algorithm results (including␣
 ↪alternate list concept)!
----> 2 s = instance.solve_guo_pyomo_model()

File ~/Desktop/Coding Projects/afccp/afccp/core/main.py:1167, in␣
 ↪CadetCareerProblem.solve_guo_pyomo_model(self, p_dict, printing)
   1164 self.reset_functional_parameters(p_dict)
   1166 # Build the model and then solve it
-> 1167 model =␣
 ↪afccp.core.solutions.optimization.assignment_model_build(self, printing=print ng)
   1168 solution = afccp.core.solutions.optimization.solve_pyomo_model(self,␣
 ↪model, "GUO", printing=printing)
   1170 # Determine what to do with the solution

File ~/Desktop/Coding Projects/afccp/afccp/core/solutions/optimization.py:103,␣
 ↪in assignment_model_build(instance, printing)
    100 m = ConcreteModel()
    102 # ___________________________________VARIABLE␣
 ↪DEFINITION___________________________________
--> 103 m = common_optimization_handling(m, p, vp, mdl_p)  # Define x along wit␣
 ↪additional functional constraints
    105 # ___________________________________OBJECTIVE␣
 ↪FUNCTION_____________________________________
    106 def objective_function(m):

File ~/Desktop/Coding Projects/afccp/afccp/core/solutions/optimization.py:975,␣
 ↪in common_optimization_handling(m, p, vp, mdl_p)
    973 for j in p['J^Special']:
    974     if len(p['I^Alternate'][j]) == 0:
--> 975         raise ValueError("Error. AFSC '" + p['afscs'][j] + "' is a␣
 ↪'special' AFSC with an alternate list "
    976                                                          "specified␣
 ↪but there are no cadets in that list.")
    978 # [1, 2, 3, 4, …, num_reserved] for each special AFSC
    979 a_counts = {j: np.arange(p['num_reserved'][j]).astype(int) + 1 for j in␣
 ↪p['J^Special']}

ValueError: Error. AFSC 'R4' is a 'special' AFSC with an alternate list␣
 ↪specified but there are no cadets in that list.
```

[156]:
```
# Re-import module and data
from afccp.core.main import CadetCareerProblem
instance = CadetCareerProblem("Random_1")
instance.set_value_parameters()
```

```
Importing 'Random_1' instance…
Instance 'Random_1' initialized.
```

6.5.4 Assignment Problem Model- "Original" AFPC model

To be true to this problem's history, I have coded up the original AFPC model formulation. However, this isn't entirely accurate since it relies on my value parameters for constraints and can be solved in an almost identical manner to GUO. The only difference here is the objective function! We create the original utility matrix that AFPC used up until 2023 for AFSC NRL classification and solve the model with the same features as GUO. If you look at the optimization.py script at the assignment_model_build() function, you will notice that I simply have an "if" statement to differentiate the two models. Here it is:

[157]:
```
s = instance.solve_original_pyomo_model()  # Not recommended in anyway, just␣
↪here as an artifact!
```

```
Building original assignment problem model…
Done. Solving model…
Solving Original Model instance with solver cbc…
Model solved in 0.05 seconds.
New Solution Evaluated.
Measured exact VFT objective value: 0.8054.
Global Utility Score: 0.6825. 0 / 0 AFSCs fixed. 0 / 0 AFSCs reserved. 0 / 0
cadets correctly pulled from alternate list.
Blocking pairs: 9. Unmatched cadets: 0. Ineligible cadets: 0.
```

6.5.5 Goal Programming Model

This section is here to talk about former Lt Rebecca Reynold's optimization model. I have it coded up to follow her methodology to the best of my ability and it lives in the code as well. Again, this is here primarily for historical purposes to capture academic contributions to this problem! You are free to look at all of the ins and outs of this model and see how the data is represented and utilized.

Quick note, part of the process of running her goal programming model involves translating my parameters & value parameters into her own specific parameters that she uses. Additionally, when she and I took on the thesis, we handled AFOCD objectives through the lens of "Mandatory", "Desired", and "Permitted" rather than "Tier 1 -> Tier 4". Her model uses the requirement levels (M, D, P) rather than the tiers in the same way DSY used that through c/2023. Little nuanced thing but since making this tutorial I have updated the random value parameter generation function to still generate these objectives (Mandatory, Desired, Permitted) so this translation function will work. Moral of the story: I just need to regenerate a set of value parameters to include these objectives which will allow me to translate my parameters to hers, and ultimately solve the model.

[158]:
```
# Generate random set of value parameters (will contain M, D, P objectives)
instance.generate_random_value_parameters()

# We now have two sets of value parameters!
print(instance.vp_dict.keys())
```

```
dict_keys(['VP', 'VP2'])
```

[159]:
```
# Solve the goal-programming model!
s = instance.solve_gp_pyomo_model()

# Re-activate the first set of value parameters
instance.set_value_parameters("VP")  # VP_2 evaluated the model by default (we
 ↪want VP)
```

```
Translating VFT model parameters to Goal Programming Model parameters…
Building GP Model…
Model built.
Solving GP Model instance with solver cbc…
Model solved in 0.05 seconds.
Model solved.
New Solution Evaluated.
Measured exact VFT objective value: 0.8666.
Global Utility Score: 0.7442. 0 / 0 AFSCs fixed. 0 / 0 AFSCs reserved. 0 / 0
cadets correctly pulled from alternate list.
Blocking pairs: 4. Unmatched cadets: 0. Ineligible cadets: 0.
Solution Evaluated: GP.
Measured exact VFT objective value: 0.8814.
Global Utility Score: 0.7917. 0 / 0 AFSCs fixed. 0 / 0 AFSCs reserved. 0 / 0
cadets correctly pulled from alternate list.
Blocking pairs: 4. Unmatched cadets: 0. Ineligible cadets: 0.
```

Another note, the "GP" model is not bound by any of the same constraints as the other models. This includes the value parameter constraints and the common optimization handling constraints. This is how here model was formulated and so it doesn't adhere to any of our "rules". That's why the objective function is a bit higher than GUO! This model remains here in case any future researchers want to look at her work and expand upon it in some capacity to work with the current state of the problem.

6.6 Sensitivity

There are some sensitivity analysis functions within afccp, and this is really the next step of what to do with this project. I don't have many yet, but the goal is to continue to develop these capabilities in the future. (I'm skipping this section for now with the exception of the one function below).

[160]:
```
# Iteratively solve model activating constraints one at a time to check
 ↪feasibility
instance.solve_for_constraints()
```

```
Initializing Assignment Model Constraint Algorithm…
Done. Solving model with no constraints active…
Done. New solution objective value: 0.7719
Running through 4 total constraint iterations…

------[1] AFSC R2 Objective Combined Quota------------
Constraint 1 Active Constraints: 1 Validated: 1
Result: SOLVED [Z = 0.7718]
```

```
Active Objective Measure Constraints: 1
Total Failed Constraints: 0
Current Objective Measure: 8.0 Range: 5, 8
---------- Objective Measure Fails:0-------------------

------[2] AFSC R1 Objective Combined Quota------------
Constraint 2 Active Constraints: 2 Validated: 2
Result: SKIPPED [Measure: 6.0],  Range: (6.0, 7.0)
Active Objective Measure Constraints: 2
Total Failed Constraints: 0
Current Objective Measure: 6.0 Range: 6, 7
---------- Objective Measure Fails:0-------------------

------[3] AFSC R3 Objective Combined Quota------------
Constraint 3 Active Constraints: 3 Validated: 3
Result: SKIPPED [Measure: 4.0],  Range: (4.0, 4.0)
Active Objective Measure Constraints: 3
Total Failed Constraints: 0
Current Objective Measure: 4.0 Range: 4, 4
---------- Objective Measure Fails:0-------------------

------[4] AFSC R4 Objective Combined Quota------------
Constraint 4 Active Constraints: 4 Validated: 4
Result: SKIPPED [Measure: 2.0],  Range: (2.0, 3.0)
Active Objective Measure Constraints: 4
Total Failed Constraints: 0
Current Objective Measure: 2.0 Range: 2, 3
---------- Objective Measure Fails:0-------------------
```

6.7 Summary

The "Solutions" section first discussed how solutions are represented in csv format as well as in the code. Just like the parameters, we have a csv dataframe containing the content which is then pulled into a solutions dictionary containing the various metrics of one particular solution given all the many parameters to the problem. The section then discusses the various solution algorithms/models that may be applied to the problem. Below is my recommended approach to the problem. I start by importing the data and applying the SOC rated algorithms first, and then exporting it back.

[161]:
```python
# Import the "Random_1" instance
instance = CadetCareerProblem('Random_1')

# "Activate" a particular set of value parameters (since you can have multiple)
instance.set_value_parameters("VP")  # There could be "VP", "VP2", "VP3", etc.

# In case you change some parameter that the value parameters depend on
instance.update_value_parameters()  # AFSC quotas are a good example here
```

```
# Always make sure your data is good to go!
instance.parameter_sanity_check()

# Run the SOC algorithms!
instance.soc_rated_matching_algorithm({"soc": "rotc", "ma_printing": True})
instance.soc_rated_matching_algorithm({"soc": "usafa", "ma_printing": True})

# Export data back to csvs
instance.export_data()
```

```
Importing 'Random_1' instance…
Instance 'Random_1' initialized.
Sanity checking the instance parameters…
Done, 0 issues found.
Solving the rated matching algorithm for ROTC cadets…

Iteration 1
Proposals: {'R4': 5}
Matched {'R4': 2}
Rejected {'R4': 3}
Solving the rated matching algorithm for USAFA cadets…

Iteration 1
Proposals: {'R4': 11}
Matched {'R4': 0}
Rejected {'R4': 11}
Exporting datasets ['Cadets', 'AFSCs', 'Preferences', 'Goal Programming', 'Value
Parameters', 'Solutions']
```

The reason I have the above code split up from below is because you only need to run the rated SOC algorithms themselves once, and then from that point on they exist in your "Solutions.csv" file and you can just incorporate them into the parameters everytime! If you're using an alternate list with a "special" rated AFSC, you can add the commented out dictionary back into the function.

[162]:
```
# Import the "Random_1" instance
instance = CadetCareerProblem('Random_1')

# "Activate" a particular set of value parameters (since you can have multiple)
instance.set_value_parameters("VP")  # There could be "VP", "VP2", "VP3", etc.

# In case you change some parameter that the value parameters depend on
instance.update_value_parameters()  # AFSC quotas are a good example here

# From now on, this becomes one of your "default" functions to apply!
instance.incorporate_rated_algorithm_results()  #␣
 ↪{'special_afscs_with_alternates': ['R4']}

# Always make sure your data is good to go!
```

```
instance.parameter_sanity_check()

# Run GUO!
instance.solve_guo_pyomo_model()

# Export data back to csvs
instance.export_data()
```

```
Importing 'Random_1' instance…
Instance 'Random_1' initialized.
Incorporating rated algorithm results…
Rated SOC Algorithm Results [USAFA Matched Cadets: 0. USAFA Reserved Cadets: 0.
ROTC Matched Cadets: 1. ROTC Reserved Cadets: 1.]
Sanity checking the instance parameters…
Done, 0 issues found.
Building assignment problem (GUO) model…
Done. Solving model…
Solving GUO Model instance with solver cbc…
Model solved in 0.05 seconds.
New Solution Evaluated.
Measured exact VFT objective value: 0.8553.
Global Utility Score: 0.7718. 1 / 1 AFSCs fixed. 1 / 1 AFSCs reserved. 0 / 0
cadets correctly pulled from alternate list.
Blocking pairs: 4. Unmatched cadets: 0. Ineligible cadets: 0.
Exporting datasets ['Cadets', 'AFSCs', 'Preferences', 'Goal Programming', 'Value
Parameters', 'Solutions']
```

7 Visualizations

I've got a lot of visualizations, but not enough time right now to hash out this section. Here's one function that creates some of the charts we use. For some reason the numbers are a little off with this random instance but for a real one they all line up.

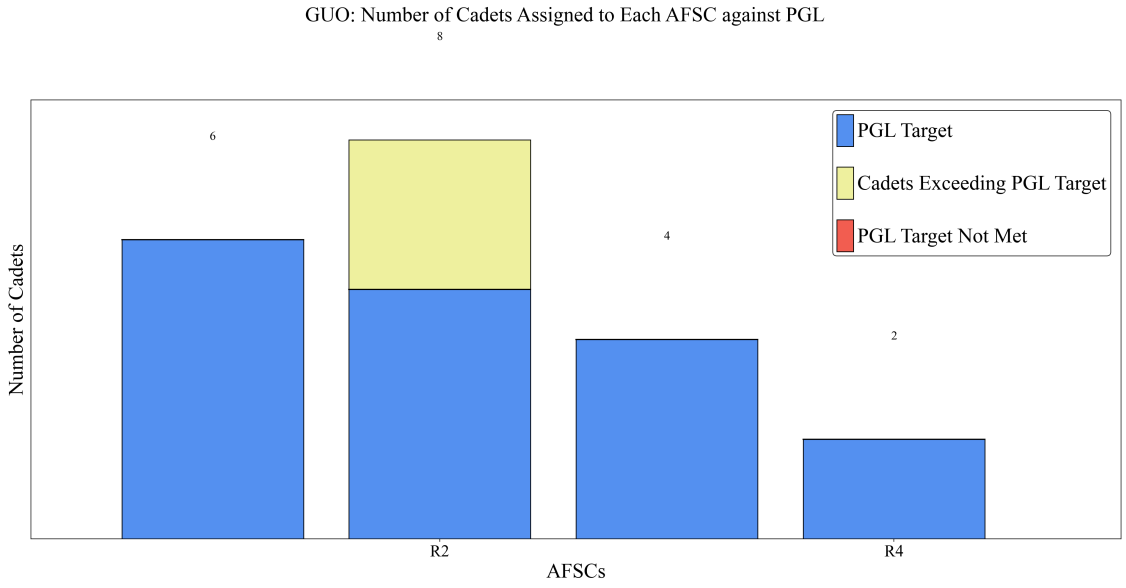
[163]:
```
instance.set_solution('GUO')  # Activate GUO solution
instance.display_all_results_graphs()
```

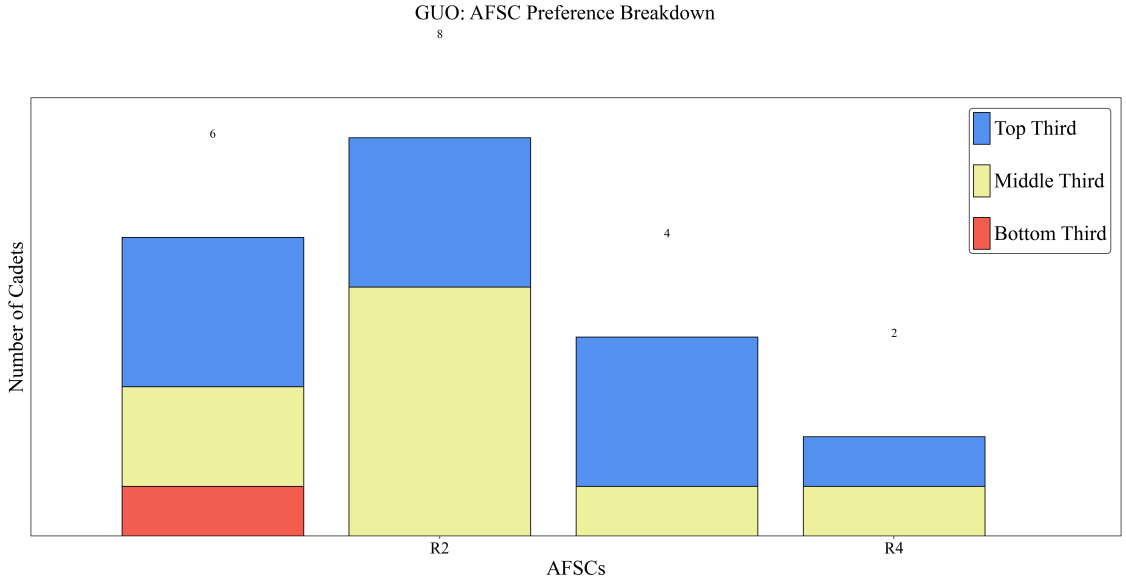
```
New Solution Evaluated.
Measured exact VFT objective value: 0.8553.
Global Utility Score: 0.7718. 1 / 1 AFSCs fixed. 1 / 1 AFSCs reserved. 0 / 0
cadets correctly pulled from alternate list.
Blocking pairs: 4. Unmatched cadets: 0. Ineligible cadets: 0.
Saving all solution results charts to the corresponding folder…
<Objective 'Combined Quota' version 'quantity_bar'>
Saved Random_1 (Default) GUO Combined Quota Measure [quantity_bar] (Results).png
Chart to instances/Random_1/5. Analysis & Results/GUO/.
<Objective 'Norm Score' version 'quantity_bar_proportion'>
Saved Random_1 (Default) GUO Norm Score Measure [quantity_bar_proportion]
(Results).png Chart to instances/Random_1/5. Analysis & Results/GUO/.
```

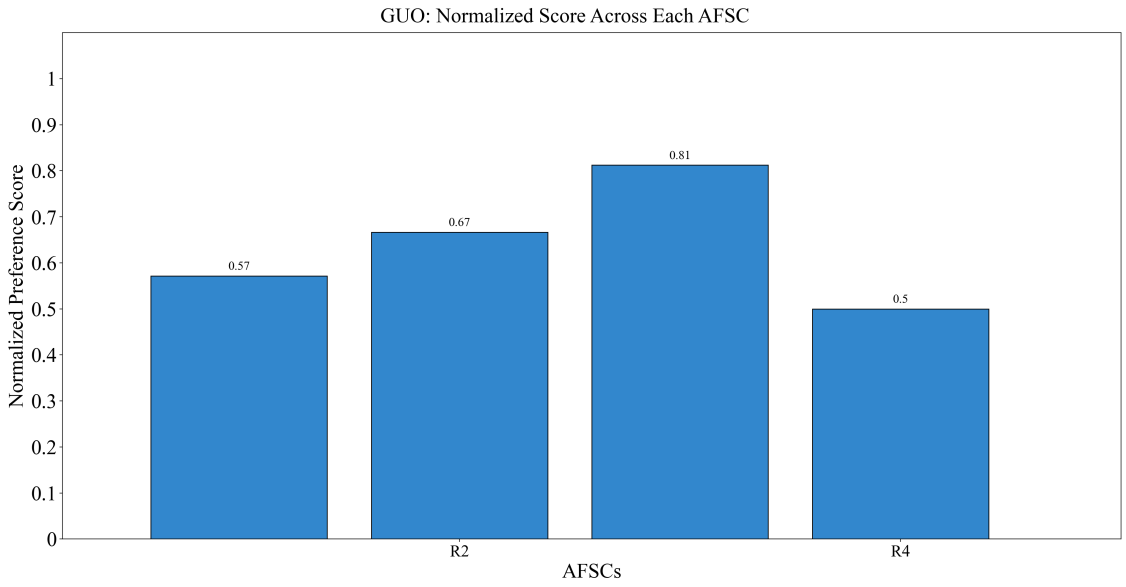
```
<Objective 'Norm Score' version 'bar'>
Saved Random_1 (Default) GUO Norm Score Measure [bar] (Results).png Chart to
instances/Random_1/5. Analysis & Results/GUO/.
<Objective 'Utility' version 'quantity_bar_proportion'>
Saved Random_1 (Default) GUO Utility Measure [quantity_bar_proportion]
(Results).png Chart to instances/Random_1/5. Analysis & Results/GUO/.
<Objective 'Utility' version 'quantity_bar_choice'>
Saved Random_1 (Default) GUO Utility Measure [quantity_bar_choice] (Results).png
Chart to instances/Random_1/5. Analysis & Results/GUO/.
<Objective 'Merit' version 'bar'>
Saved Random_1 (Default) GUO Merit Measure [bar] (Results).png Chart to
instances/Random_1/5. Analysis & Results/GUO/.
<Objective 'USAFA Proportion' version 'quantity_bar_proportion'>
Saved Random_1 (Default) GUO USAFA Proportion Measure [quantity_bar_proportion]
(Results).png Chart to instances/Random_1/5. Analysis & Results/GUO/.
<Objective 'USAFA Proportion' version 'preference_chart'>
Saved Random_1 (Default) GUO USAFA Proportion Measure [preference_chart]
(Results).png Chart to instances/Random_1/5. Analysis & Results/GUO/.
<Objective 'Male' version 'preference_chart'>
Saved Random_1 (Default) GUO Male Measure [preference_chart] (Results).png Chart
to instances/Random_1/5. Analysis & Results/GUO/.
<Objective 'Extra' version 'Race Chart'>
<Objective 'Extra' version 'Race Chart_proportion'>
<Objective 'Extra' version 'Ethnicity Chart'>
<Objective 'Extra' version 'Ethnicity Chart_proportion'>
<Objective 'Extra' version 'Gender Chart'>
<Objective 'Extra' version 'Gender Chart_proportion'>
<Objective 'Extra' version 'SOC Chart'>
<Objective 'Extra' version 'SOC Chart_proportion'>
<Other Charts 'Accessions Group' version 'Race Chart'>
<Other Charts 'Accessions Group' version 'Gender Chart'>
<Other Charts 'Accessions Group' version 'SOC Chart'>
<Other Charts 'Accessions Group' version 'Ethnicity Chart'>
```

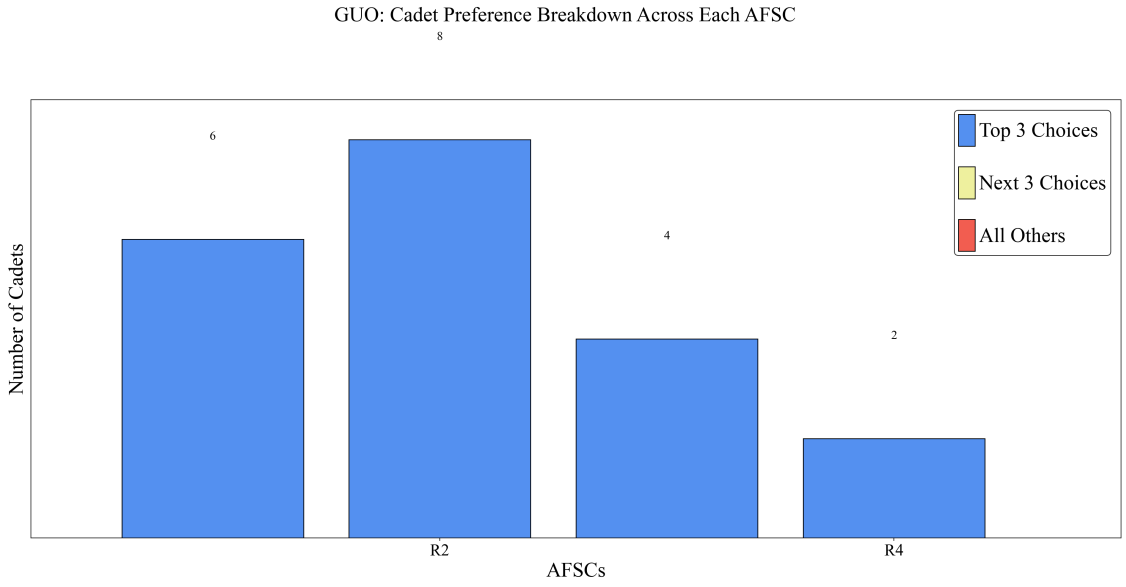
[163]:
```
[<Figure size 3800x2000 with 1 Axes>,
 <Figure size 3800x2000 with 1 Axes>,
 <Figure size 3800x2000 with 1 Axes>,
 <Figure size 3800x2000 with 1 Axes>,
 <Figure size 3800x2000 with 1 Axes>,
 <Figure size 3800x2000 with 1 Axes>,
 <Figure size 3800x2000 with 1 Axes>,
 <Figure size 3800x2000 with 1 Axes>,
 <Figure size 3800x2000 with 1 Axes>,
 None,
 None,
 None,
 None,
```

```
None,
None,
None,
None,
None,
None,
None,
None]
```

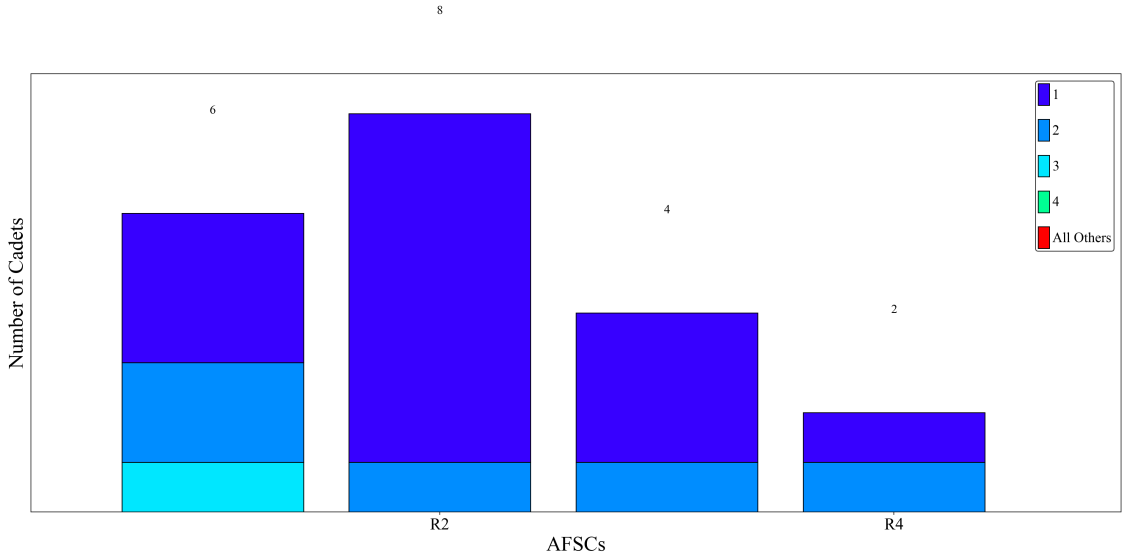




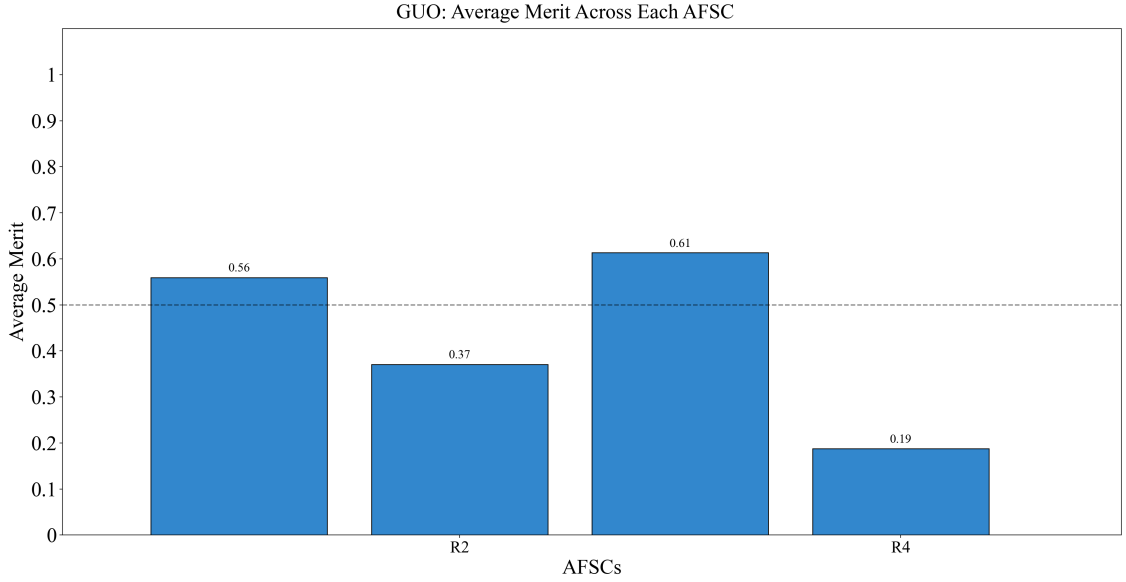

GUO: Normalized Score Across Each AFSC
Normalized Preference Score
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0.57
0.67
0.81
0.5
R2
R4
AFSCs


GUO: Cadet Preference Breakdown Across Each AFSC
8
6
4
2
Number of Cadets
Top 3 Choices
Next 3 Choices
All Others
R2
R4
AFSCs

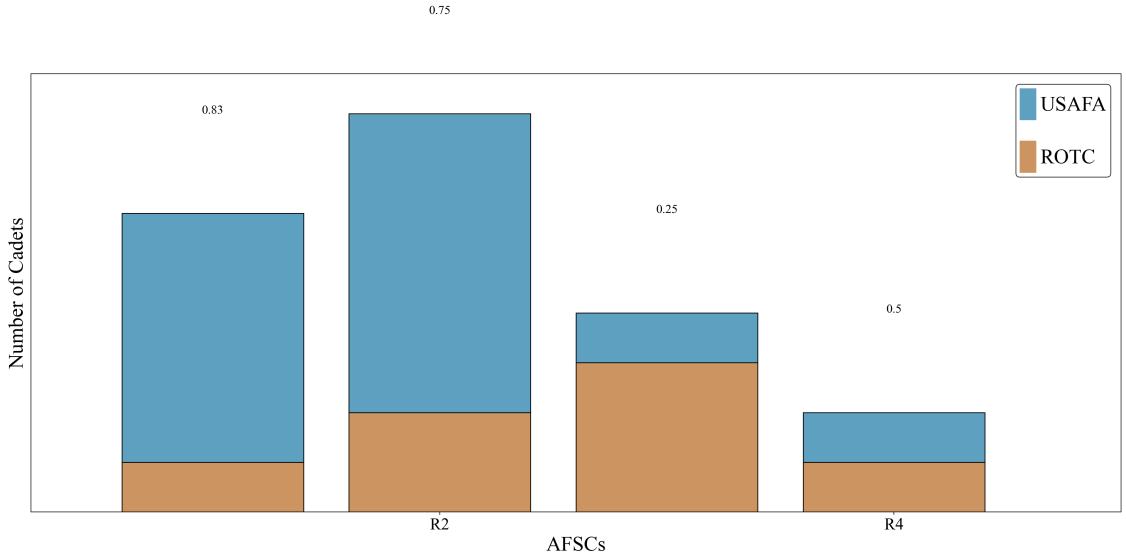
GUO: Cadet Preference Breakdown Across Each AFSC



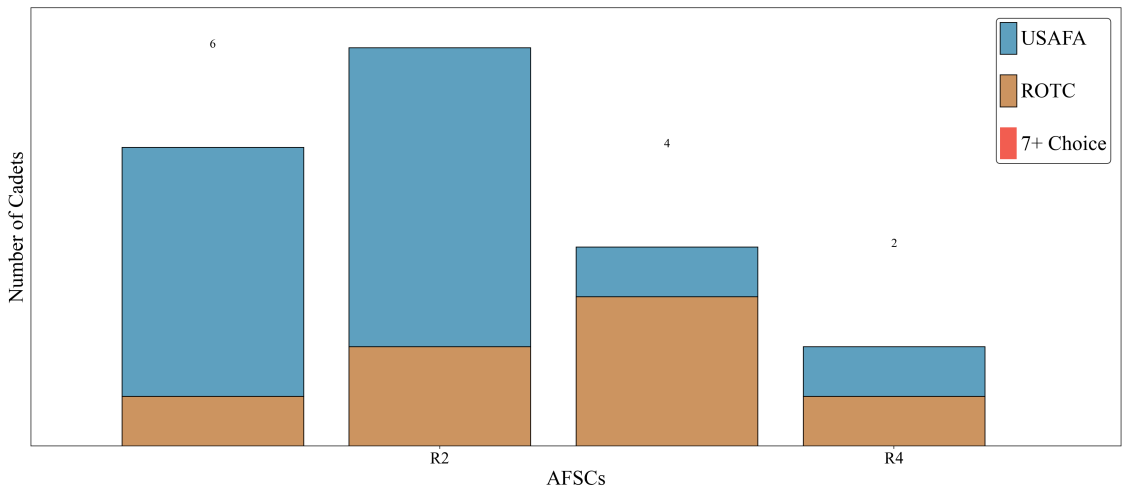
GUO: Average Merit Across Each AFSC



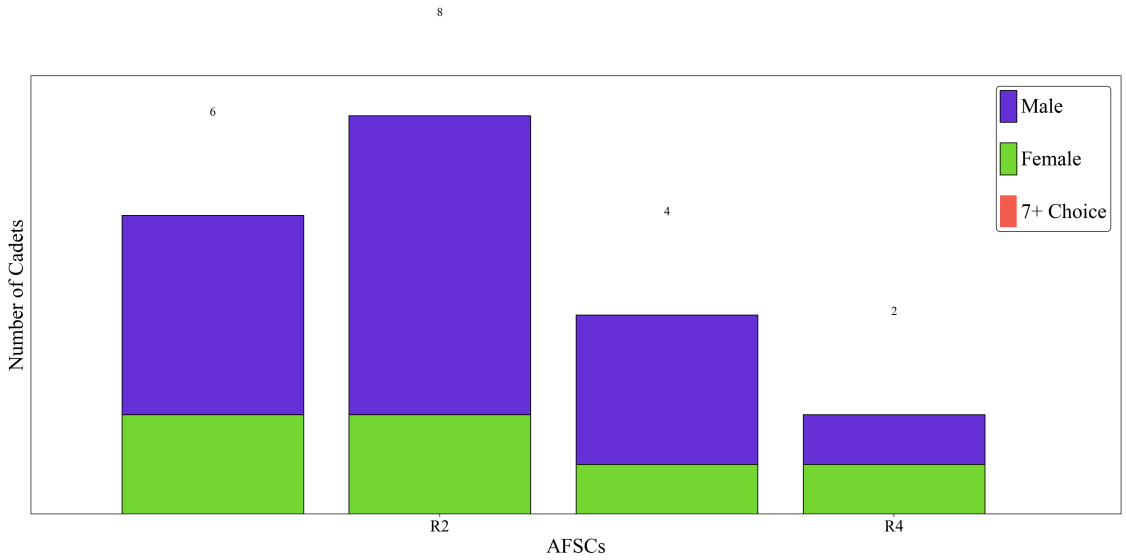
GUO: Source of Commissioning Breakdown Across Each AFSC

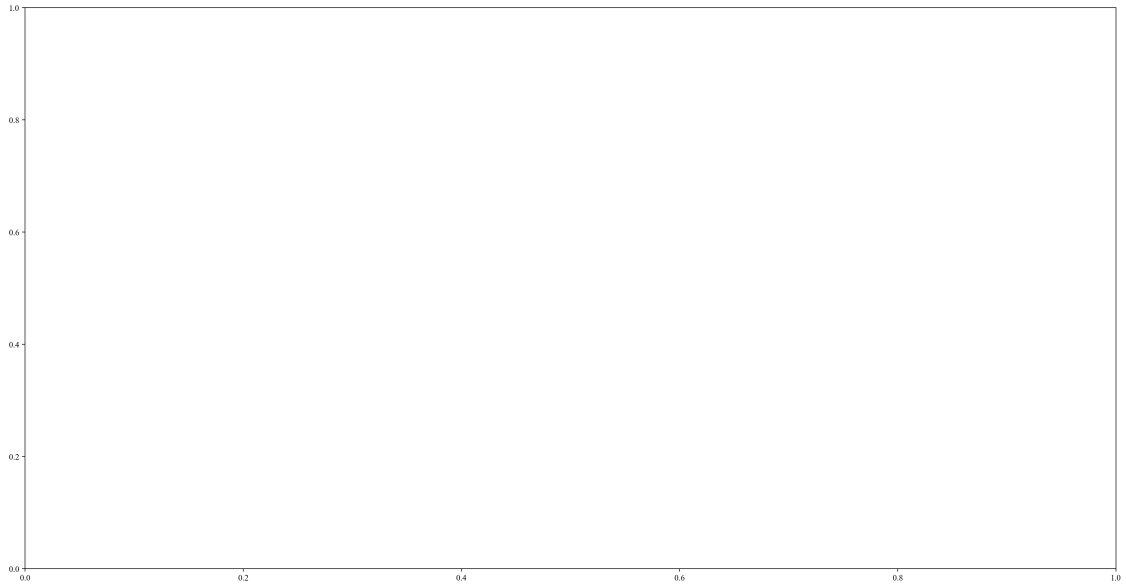


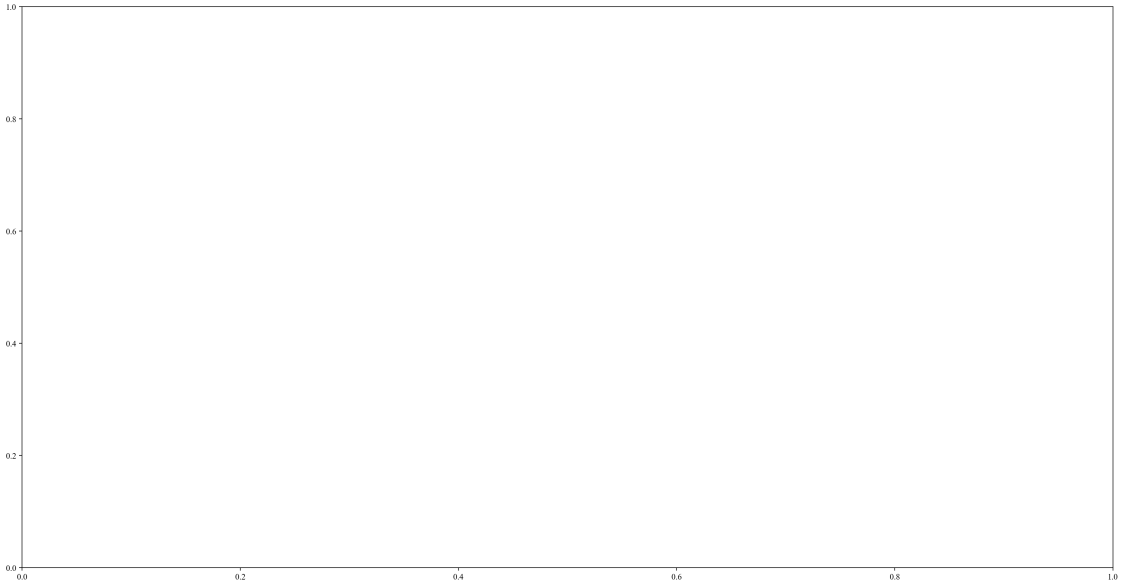
GUO: USAFA/ROTC 7+ Choice Across Each AFSC

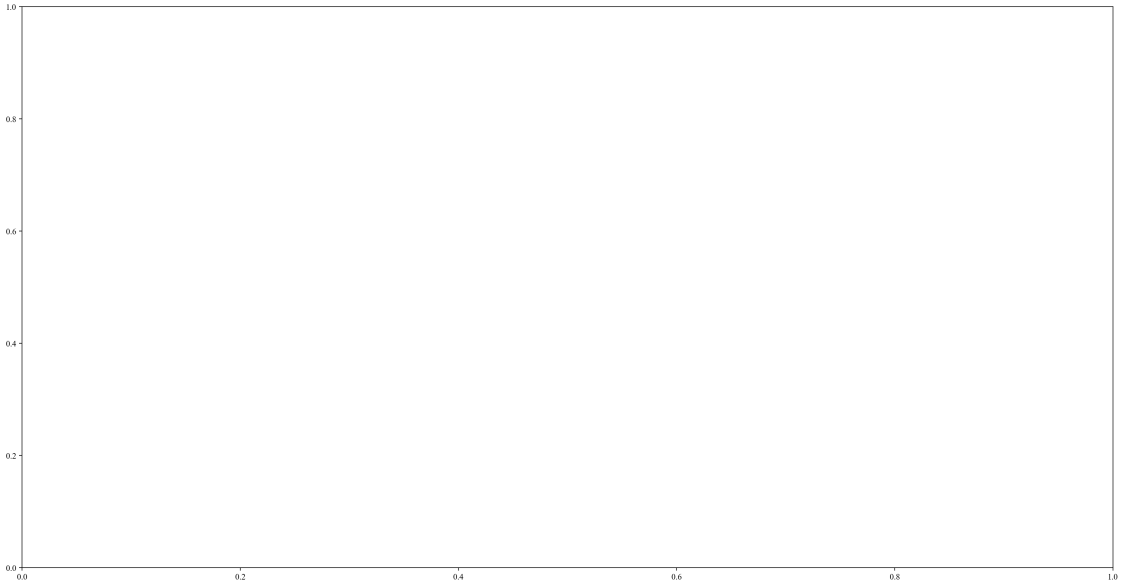


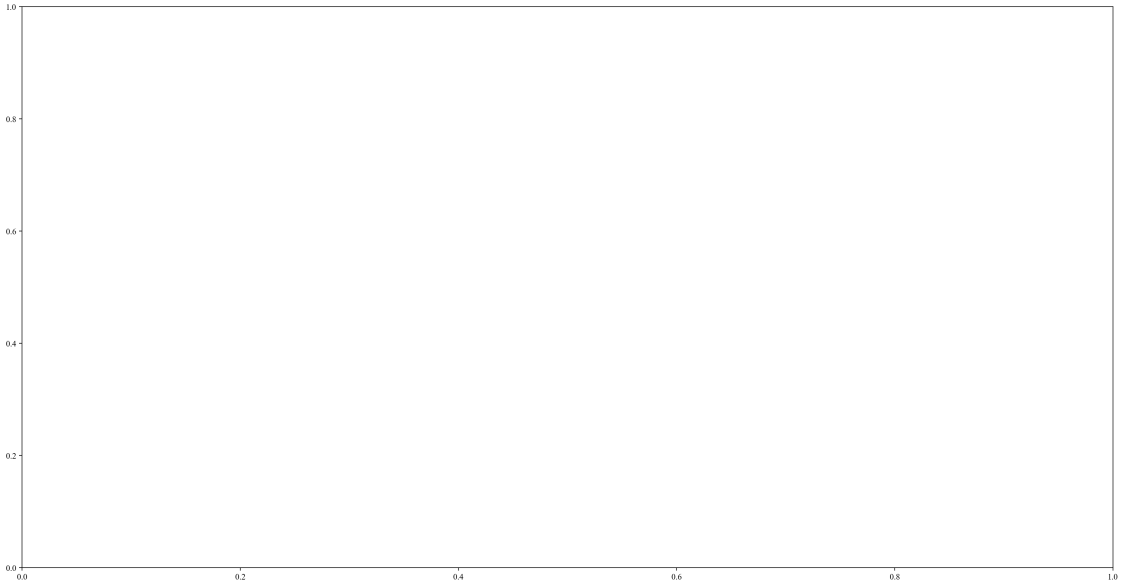
GUO: Male/Female 7+ Choice Across Each AFSC

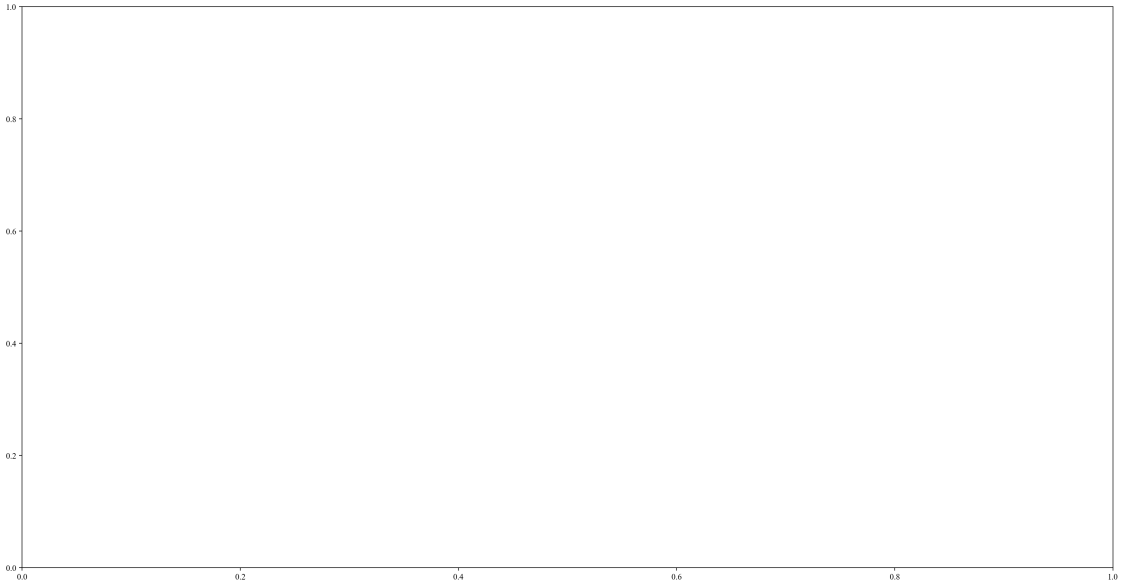


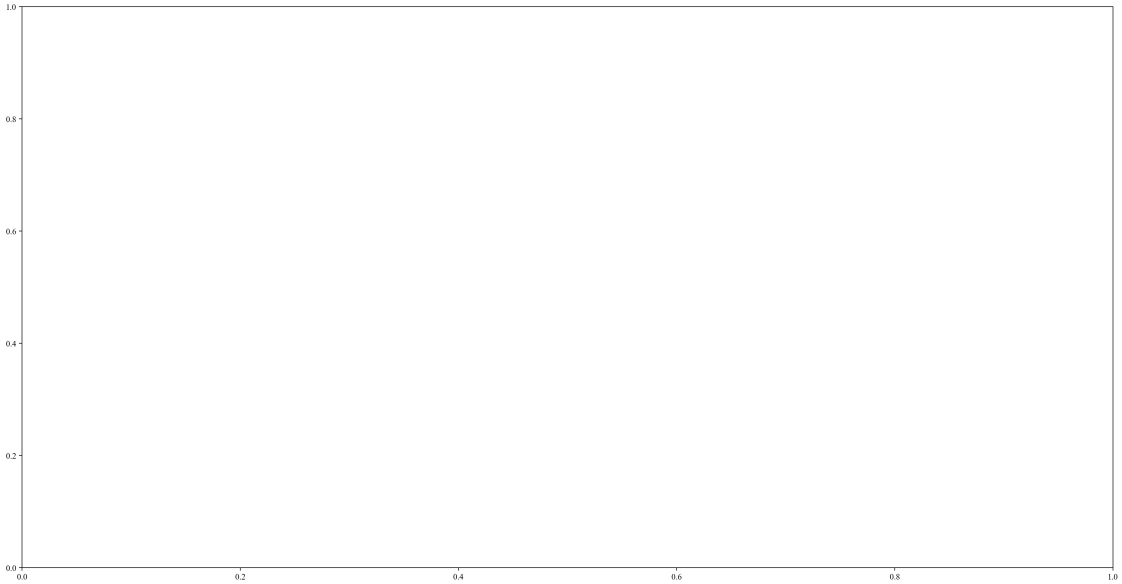


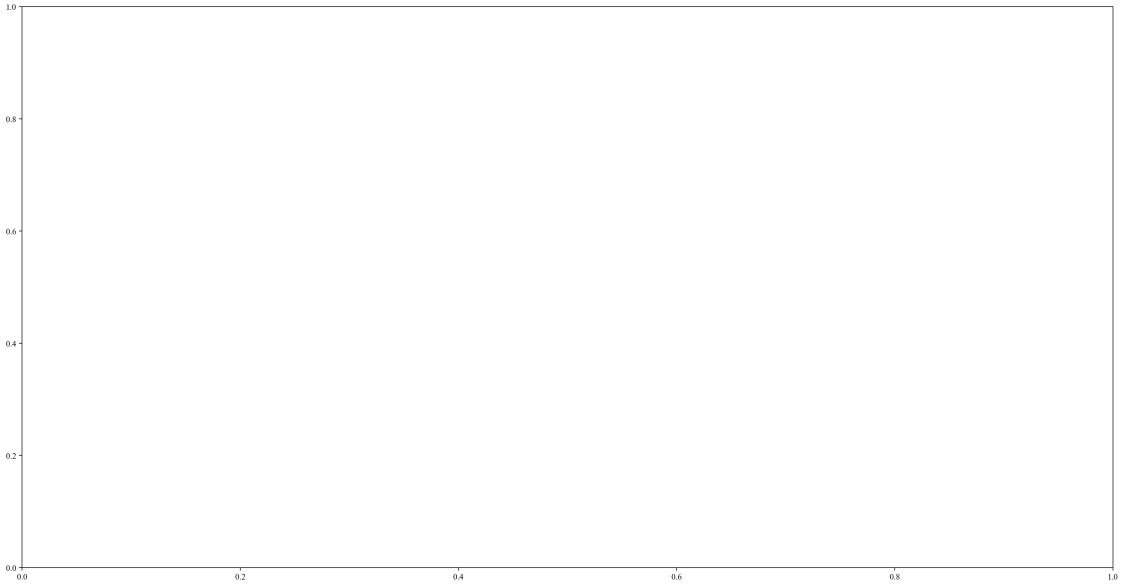

1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0

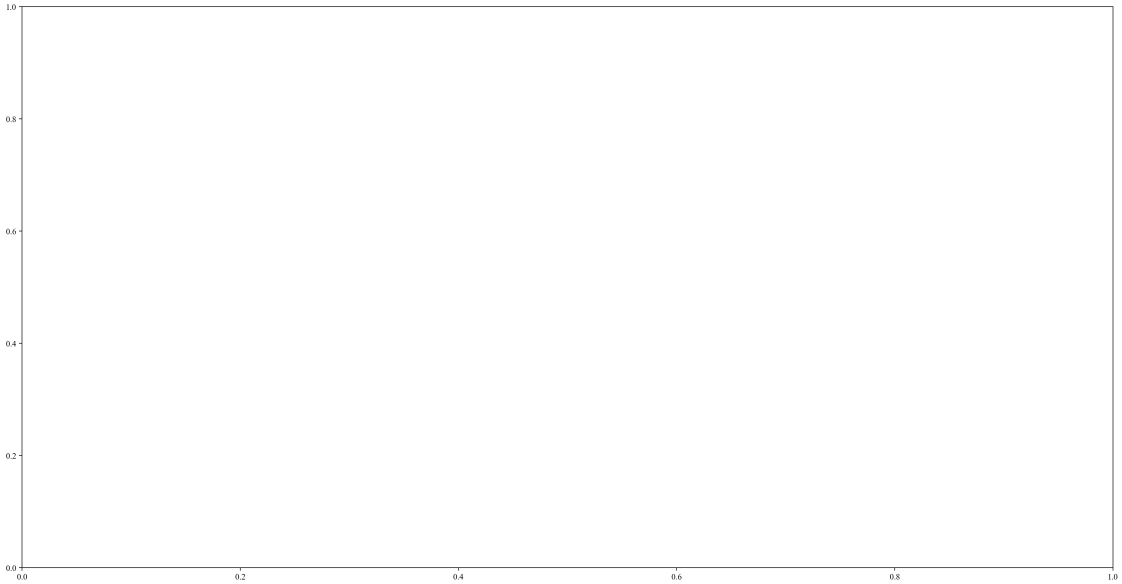

1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0

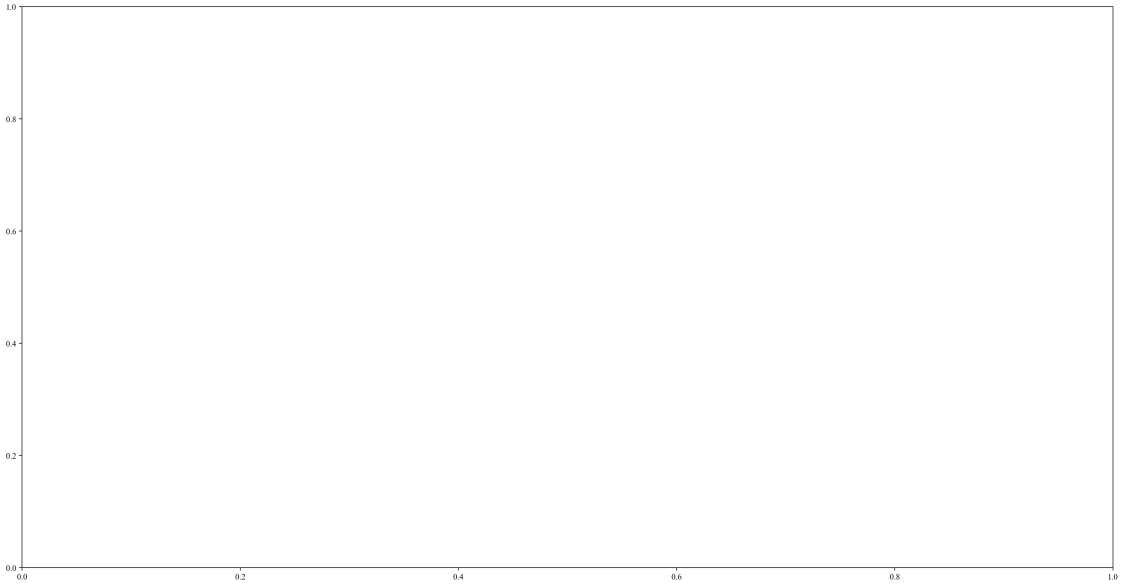

1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0

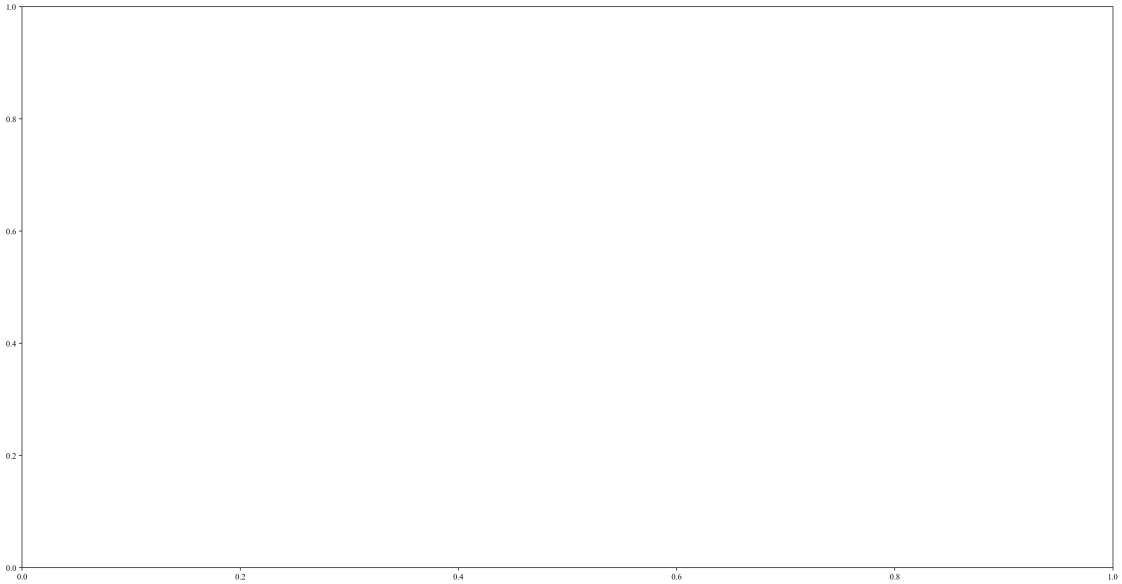

1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0

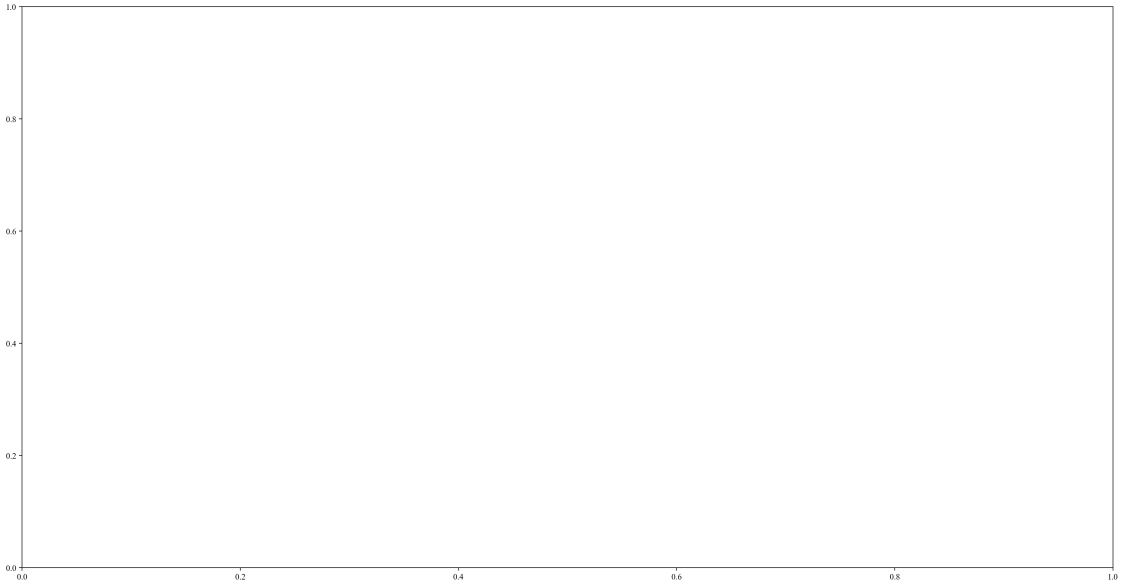

1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0

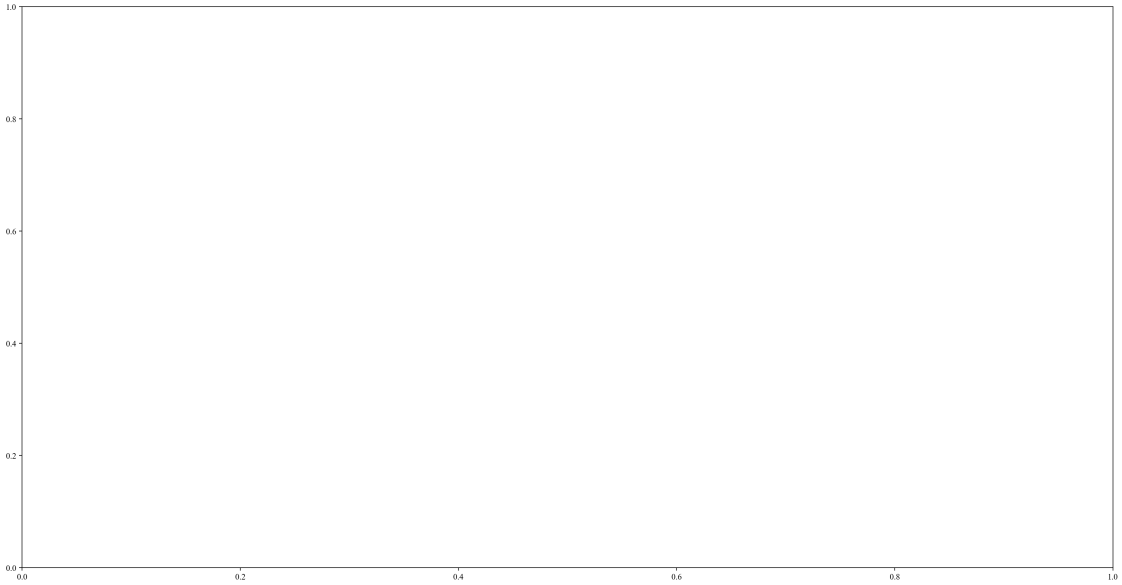

1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0


1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0


1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0


1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0


1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0



[164]:
```
# This is going to be the function to automate the powerpoint... (nothing here
↪really yet)
instance.generate_results_slides()
```

```
Generating results slides…
Done.
```

[165]:
```
# TEMPORARY HERE TO AVOID RE-RUNNING EVERYTHING ABOVE
# (Just to help me as I keep writing this tutorial)
import numpy as np
import pandas as pd
import os

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

# Get main afccp folder path
index = dir_path.find('afccp')
dir_path = dir_path[:index + 6]

# Update working directory
os.chdir(dir_path)
print('updated working directory:', dir_path)

# Import module and data
from afccp.core.main import CadetCareerProblem
instance = CadetCareerProblem("Random_1")
```

```
instance.set_value_parameters()

# Shorthand
p, vp = instance.parameters, instance.value_parameters
```

```
initial working directory: /Users/griffenlaird/Desktop/Coding Projects/afccp/
updated working directory: /Users/griffenlaird/Desktop/Coding Projects/afccp/
Importing 'Random_1' instance…
Instance 'Random_1' initialized.
```