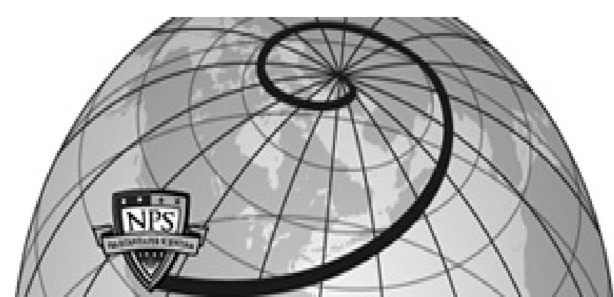


Calhoun: The NPS Institutional Archive

DSpace Repository

NPS Scholarship Theses

2022-12

INCREASING UTILITY AND FIDELITY OF A UXV NETWORKED CONTROL SYSTEM SIMULATION VIA PYTHON ADAPTATION

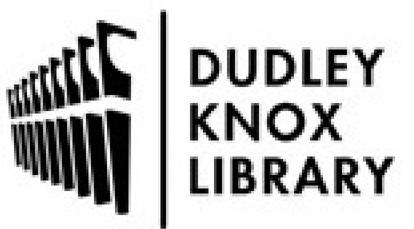
Faulk, Andrew J.; Muiruri, Antony K.

Monterey, CA; Naval Postgraduate School

https://hdl.handle.net/10945/71456

This publication is a work of the U.S. Government as defined in Title 17, United States Code, Section 101. Copyright protection is not available for this work in the United States.

Downloaded from NPS Archive: Calhoun



Calhoun is the Naval Postgraduate School's public access digital repository for research materials and institutional publications created by the NPS community. Calhoun is named for Professor of Mathematics Guy K. Calhoun, NPS's first appointed -- and published -- scholarly author.

Dudley Knox Library / Naval Postgraduate School
411 Dyer Road / 1 University Circle
Monterey, California USA 93943

http://www.nps.edu/library

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

INCREASING UTILITY AND FIDELITY OF A UXV NETWORKED CONTROL SYSTEM SIMULATION VIA PYTHON ADAPTATION

by

Andrew J. Faulk and Antony K. Muiruri

December 2022

Thesis Advisor: Geoffrey G. Xie
Second Reader: Duane T. Davis

Approved for public release. Distribution is unlimited.

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington, DC, 20503.

1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** December 2022	**3. REPORT TYPE AND DATES COVERED** Master's thesis

4. TITLE AND SUBTITLE INCREASING UTILITY AND FIDELITY OF A UXV NETWORKED CONTROL SYSTEM SIMULATION VIA PYTHON ADAPTATION	**5. FUNDING NUMBERS**
6. AUTHOR(S) Andrew J. Faulk and Antony K. Muiruri	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A	**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release. Distribution is unlimited.	**12b. DISTRIBUTION CODE** A

13. ABSTRACT (maximum 200 words)

While the use of unmanned vehicles (UxVs) within the Department of Defense (DOD) is prevalent, the ability of the DOD to operate these vehicles as a cooperative networked control system (NCS) is not fully developed. A MATLAB simulation modeling a UxV NCS that jointly optimizes sensing and data communications utilities currently exists. However, this implementation is of low fidelity and is not readily extensible. This thesis advances the NCS model by converting the MATLAB simulation into Python with an object-oriented design. We further extend the Python NCS simulation into a truly distributed system, where each node executes in an independent Docker container and communicates with other nodes via reliable message communications. Our implementation results demonstrated that the object-oriented Python simulation incurs a performance penalty with respect to execution times but provides for greater extensibility without changes to existing functionality. Furthermore, Docker containers offer a lightweight solution for modeling a more realistic version of UxV simulations, thus increasing the NCS simulation's fidelity.

14. SUBJECT TERMS networked control system, NCS, docker containers, object-oriented programming, network fidelity, extensibility, unmanned vehicles, UxVs	**15. NUMBER OF PAGES** 101
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

NSN 7540-01-280-5500 | Standard Form 298 (Rev. 2-89) Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release. Distribution is unlimited.

INCREASING UTILITY AND FIDELITY OF A UXV NETWORKED CONTROL SYSTEM SIMULATION VIA PYTHON ADAPTATION

Andrew J. Faulk
Lieutenant Commander, United States Navy
BS, United States Naval Academy, 2012

Antony K. Muiruri
Lieutenant, United States Navy
BS, Indiana University, Fort Wayne, 2007

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 2022

Approved by: Geoffrey G. Xie
Advisor

Duane T. Davis
Second Reader

Gurminder Singh
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

While the use of unmanned vehicles (UxVs) within the Department of Defense (DOD) is prevalent, the ability of the DOD to operate these vehicles as a cooperative networked control system (NCS) is not fully developed. A MATLAB simulation modeling a UxV NCS that jointly optimizes sensing and data communications utilities currently exists. However, this implementation is of low fidelity and is not readily extensible. This thesis advances the NCS model by converting the MATLAB simulation into Python with an object-oriented design. We further extend the Python NCS simulation into a truly distributed system, where each node executes in an independent Docker container and communicates with other nodes via reliable message communications. Our implementation results demonstrated that the object-oriented Python simulation incurs a performance penalty with respect to execution times but provides for greater extensibility without changes to existing functionality. Furthermore, Docker containers offer a lightweight solution for modeling a more realistic version of UxV simulations, thus increasing the NCS simulation's fidelity.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

1 Introduction **1**

1.1 Motivation 2

1.2 Problem Statement. 2

1.3 Research Questions 3

1.4 Overview 3

2 Background **5**

2.1 MATLAB Programming Language 5

2.2 The Python Programming Language. 10

2.3 Docker 16

2.4 Networked Control System (NCS) 18

2.5 Related Work 19

2.6 Chapter Summary 19

3 Methodology **21**

3.1 Automated MATLAB-to-Python Code Conversion 21

3.2 Function Mapping 21

3.3 Conceptualizing Python Classes 23

3.4 Chapter Summary 24

4 Implementation and Evaluation **27**

4.1 Python Design Implementation 27

4.2 Evaluation of Design Implementation 37

4.3 Chapter Summary 43

5 Docker Container Implementation and Evaluation **45**

5.1 Methodology 45

5.2 Design Implementation 48

5.3 Evaluation of Design Implementations 54
5.4 Chapter Summary 55

6 Conclusion **57**
6.1 Summary and Conclusions 57
6.2 Lessons Learned 58
6.3 Future Work 58

Appendix A SMOP Code **61**
A.1 MATLAB Test Code 61
A.2 SMOP Functionality Testing 62

Appendix B M2HTML Code **65**
B.1 M2HTML Demonstration 65

Appendix C Python Code **69**
C.1 Source Code Repository 69
C.2 Sample `Employees` Class Definition 69
C.3 Detailed Iterative Process Flow Diagram 71

Appendix D Cython Code **73**
D.1 Cython Conversion Test. 73
D.2 Conversion Results 75

List of References **77**

Initial Distribution List **81**

List of Figures

Figure 2.1	MATLAB-rendered cosine wave	10
Figure 2.2	Sample class hierarchy of an `Employee` class	15
Figure 2.3	Docker architecture	16
Figure 2.4	Depiction of applications running on Docker and VMs	17
Figure 3.1	Sample *GraphViz* dependency graph depiction	23
Figure 3.2	NCS function dependency graph	23
Figure 3.3	Proposed NCS Python class hierarchy	24
Figure 4.1	Iterative process flow chart	28
Figure 4.2	Finalized NCS Python class diagram	36
Figure 4.3	Initial NCS node placement Python-MATLAB plot comparison	38
Figure 4.4	Final NCS node placement Python-MATLAB plot comparison	39
Figure 4.5	Network utility plot comparison	40
Figure 5.1	Reliable node communications via MQTT protocol	47
Figure 5.2	Final output of Docker-compatible DS simulation	54
Figure C.1	Granular iterative process flow chart	71

THIS PAGE INTENTIONALLY LEFT BLANK

List of Tables

Table 4.1 Manufacturer specifications of benchmarking computer 41

Table 4.2 NCS simulation execution times 42

Table 4.3 Performance comparison between MATLAB, Python, and Cython 42

Table 5.1 Performance evaluation of the distributed NCS simulation 55

Table D.1 Python test code execution times 76

Table D.2 Python test code performance comparison 76

THIS PAGE INTENTIONALLY LEFT BLANK

List of Source Code

2.1 Demonstration of a Python class structure 12

4.1 MATLAB adjacency matrix calculation function 29

4.2 Python adjacency matrix calculation function 29

4.3 Python `calc_adj_mtx` function test script 30

4.4 Inheritance ("is a") relationship example written in Python 33

4.5 Composition ("has a") relationship example written in Python 34

5.1 Dockerfile source code for `USV2` image 53

A.1 MATLAB test code for automatic conversion 61

A.2 SMOP conversion of MATLAB code to Python 62

B.1 MATLAB test code to demonstrate M2HTML functionality 65

B.2 M2HTML output of function dependencies 66

C.1 *Employees* class definition 69

D.1 Python test script 73

D.2 Cythonized Python test script 73

D.3 Python script to compile target files 74

D.4 Python script to evaluate performance 75

THIS PAGE INTENTIONALLY LEFT BLANK

List of Acronyms and Abbreviations

API application programming interface

CPU central processing unit

DDG U.S. Navy destroyer

DOD Department of Defense

DS distributed submodularity

M2HTML MATLAB to Hypertext Markup Language

MANET mobile ad hoc network

MATLAB Matrix Laboratory

MQTT Message Queuing Telemetry Transport

MTX multi-thread exercise

NCS networked control system

NPS Naval Postgraduate School

NSW Naval Special Warfare

OOP object-oriented programming

OS operating system

SCI San Clemente Island

SMOP Small MATLAB and Octave to Python compiler

UAV unmanned aerial vehicle

UxV unmanned vehicle

VM virtual machine

THIS PAGE INTENTIONALLY LEFT BLANK

Acknowledgments

From Andrew: This work would not have been possible without the support of friends and family. To my comrade in arms, Antony. I seriously lucked out having you as a thesis partner. Your even temperament, humor, sharp intellect, and willingness to forgo all sleep for 3 a.m. coding sessions made this a genuinely gratifying experience. To my son, Jack, you were such a welcome distraction from my work and an endless source of joy and inspiration. Lastly, to my wife, Kendal. Thank you for your countless sacrifices, which enabled me to complete this work. Whether bringing meals to my office, holding down the home fort, or just being a listening ear, you were exactly what I needed. I will forever be indebted to you: thank you.

From Antony: I would like to express my gratitude to the supportive community of family and friends for their contributions in whatever special way. To my mom, dad, and the rest of the family, thank you for your support in taking care of my little princesses, Valentina and Juliah. To my friend Susan and many others, I cannot thank you enough for the free therapy sessions. Finally, I could not have asked for a better thesis partner. Your organization, coordination, and determination made the daunting thesis process an enjoyable experience. Thank you, Andrew.

From both of us: First and foremost, we would like to thank our thesis advisor, Dr. Geoffrey Xie, whose patience and enthusiasm afforded us the latitude to fully explore this thesis. His expert knowledge, guidance, and advice were invaluable in keeping us on the right path when the task seemed unachievable and the drain of thesis writing began to take its toll.

We also owe a debt of gratitude to our second reader, Dr. Duane Davis, for his constructive feedback and guidance. He was instrumental in spotlighting those issues we had overlooked and in helping us express our thoughts in a clear and concise manner.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1: Introduction

Technology is advancing at a rapid rate. Computations once conducted on highly specialized supercomputers requiring substantial infrastructure, energy, and resources can now be completed trivially by mobile devices. For example, in 1975, the Cray-1 supercomputer “had a raw computing power of 80 million floating-point operations per second” [1]. In contrast, the iPhone 13 Pro, released in 2021, has a five-core graphics processor capable of performing 15.8 trillion operations per second [2]. From a networking standpoint, this advancement in computing power has allowed for the implementation of robust mobile ad hoc networks (MANETs), or networks that do not rely on pre-existing infrastructure to operate [3].

One way this is manifested is in drone technology. Drones are becoming increasingly more capable, accessible, and affordable in both the private sector and military applications. Governments, militaries, and malicious actors who can leverage this technology effectively may gain a significant advantage in warfighting. Advances in MANETs have also shifted the dynamics towards employing systems of systems–individual units that can act in unison towards a single objective. Such systems are particularly advantageous to militaries, which often have multiple assets working together to complete specific goals.

One such system is a networked control system (NCS). An NCS is “a single system composed of multiple discrete entities with control loops that share feedback and control signals over a shared communications network” [4]. In NCSs, “both control signals and feedback signals can be exchanged among system components (e.g., sensors, controllers, actuators, and so on)” [5], enabling them to act on a shared set of information and control parameters. A key element of NCSs is that they manage coordination between many nodes. Recent developments in this field of study have inspired new applications for the Department of Defense (DOD) to gain and maintain a tactical advantage on the battlefield. To achieve this reality, the DOD must first develop, test, field, and deploy this technology faster than the adversary.

1.1 Motivation

This thesis advances the work of both ENS Noah Wachlin's *Robust Time-Varying Formation Control with Adaptive Submodularity* [6], and LT Brian Lowry's *Distributed Submodular Optimization for a UxV Networked Control System* [4] theses. In their work, Lowry and Wachlin utilized submodularity as a mathematical framework to optimize the positioning of NCS nodes to increase the joint utility of their sensing coverage and communications robustness. Their simulation was derived from a multi-thread exercise (MTX) scenario conducted on San Clemente Island (SCI), California, in November 2017. In this scenario, the nodes under the control of the NCS consisted of three aerial and two surface unmanned vehicles (UxVs) operating in support of a Naval Special Warfare (NSW) unit conducting a mission against a target located on SCI. The NSW unit traversed a road network from one end of the island to the other to close on the objective, while the UxVs positioned themselves to maintain optimal sensor coverage of the road network, the NSW team, and the target. Concurrently, the NCS optimized communications connectivity between the UxVs, the NSW unit, and a U.S. Navy destroyer (DDG) stationed off the coast. Lowry and Wachlin utilized the Matrix Laboratory (MATLAB) programming language to simulate and test their NCS submodularity framework. For the remainder of this thesis, the term NCS will refer to the MATLAB simulation created by Lowry and Wachlin.

While MATLAB can be a valuable tool for modeling and simulation in the field of robotics, employing the NCS code in real-world systems creates significant technology transfer gaps. We believe that we can effectively convert the NCS simulation from MATLAB code to Python to fill some of the gaps.

1.2 Problem Statement

The NCS simulation is very complex. A product of two theses and additional work by the Naval Postgraduate School (NPS) unmanned systems lab, it comprises 20 modules (i.e., .m files), 15 functions, and 1,178 lines of code. While this system is a significant achievement in and of itself, its functionality is not easy to extend and deploy to hardware because it is procedure-based rather than object-oriented. It is, therefore, advantageous to convert this MATLAB-based simulation into a code format more suitable for follow-on development, implementation, testing, and use. Our thesis aims to demonstrate that an

object-oriented programming (OOP)-based NCS implementation can achieve results similar to the MATLAB-based implementation in [4] and [6] while providing additional functionality. Specifically, it will more readily accommodate additional usability and extensibility requirements.

Lastly, the current MATLAB simulation lacks fidelity. Each node is simulated as a function call within the program, which is not how the system would truly operate in the real world. We postulate that we can develop lightweight Docker containers to house the code to simulate multiple independent nodes. This will allow future implementations to run the code on the UxVs and other platforms it was intended for.

1.3 Research Questions

Our thesis and associated simulation aim to answer the following basic research questions:

1. Can we demonstrate that the NCS simulation can be implemented in an OOP system that can be more readily deployed in real-world vehicles?
2. Is there a generalizable methodology for converting MATLAB modules into an OOP implementation?
3. How can an OOP implementation boost extensibility?
4. Are containers a good technology for modeling a more realistic version of the simulation?

1.4 Overview

The remainder of this thesis is organized as follows. Chapter 2 gives an introduction to MATLAB, discusses OOP as it applies to Python, introduces the concept of Docker containers, reviews the NCS in greater detail, and explores works related to our thesis. In Chapter 3, we discuss our methodology for converting MATLAB to Python. In Chapter 4, we present a detailed description of our software development process from design to implementation. We analyze the code conversion and its performance in various tests and present the results. In Chapter 5, we outline our methodology for re-designing our code to allow for a distributed simulation via Docker containers and furnish the results. Finally, in Chapter 6, we present our conclusions, discuss lessons learned, and propose future work.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 2: Background

This chapter explores the foundational tools, concepts, and bodies of work that are the building blocks for our thesis. Section 2.1 provides an overview of the MATLAB programming language, including its origins, basic syntax, examples of how it differs from Python, and a demonstration of its plotting capabilities. Section 2.2 provides an overview of the Python programming language, including discussions on OOP and class objects. Section 2.3 discusses Docker, introduces virtual machines (VMs), and explains our rationale for choosing Docker over VMs for our implementation. Lastly, we briefly review the high-level NCS node placement optimization in Section 2.4, and explore works related to our thesis in Section 2.5.

2.1 MATLAB Programming Language

MATLAB is a computing system for performing the calculations involved in scientific and engineering problems. MATLAB stands for Matrix Laboratory because the system was specifically designed to perform matrix computations with relative ease [7]. It was first made available for commercial use in 1984 and has since undergone substantial changes, and enhancements [8]. MATLAB is useful for modeling and simulation and includes "a large number of toolboxes available for license, as well as a number of community-provided toolboxes to solve common problems" [1]. However, MATLAB has some significant drawbacks, especially when compared to Python. Because MATLAB is a multi-paradigm programming language designed primarily for numeric computation, it is not as readily extensible as Python, which uses OOP. MATLAB is also not as user-friendly and requires a steeper learning curve. It has a smaller user base and ecosystem than Python and provides less functionality as a result. Lastly, MATLAB is proprietary software, whereas Python is open source (i.e., free). When direct comparison is warranted, this thesis displays MATLAB code against a light blue background and Python code against a light tan background. Index numbers are displayed on the left-hand side of code blocks as required to aid in referencing specific lines or to enhance readability.

2.1.1 Basic Syntax

MATLAB is similar to the C programming language in that it uses specific characters (such as curly braces) to encode discrete flow control structures [1]. However, unlike in C, a semicolon is not strictly required as a termination character at the end of each line of code. When the optional semicolon is used as a termination character, the output of that line is *suppressed* on the display. For example:

```
y = 2 + 3;
```

does not display any output but stores the result in the variable `y`. However, if the semicolon is omitted, as in the statement

```
y = 2 + 3
```

the following output will result:

```
>> y = 2 + 3

y =

     5
```

Omitting the semicolon is useful for displaying the results of command execution to the user and for troubleshooting. Otherwise, programs normally include the semicolon to refrain from displaying extraneous information.

Variables

MATLAB is a weakly typed language, and variable types do not need to be declared before a variable can be used [1]. For example, in the code block

```
x = 4
three = 3
pi = 3.14
my_string = 'Hello World!'
```

the variables `x` and `three` are not explicitly declared as integers before assigning them integral values. Similarly, the variables `pi` and `my_string` did not have to be declared as float (i.e., floating point decimal value) and string types, respectively, before their assignments. This is one instance where MATLAB and Python behave in the same manner. In Python, it is also not necessary to declare a variable's type upon its creation (although

Python is considered a strongly typed language because of how variables are maintained after creation). After declaring a variable, it can be recalled to perform calculations. For example, based on the previous code snippets, the assignment statement

```
y = x * three
```

will have an output of

```
>> y = x * three

y =

    12
```

Additionally, both MATLAB and Python are dynamically typed and allow the reassignment of variables with different data types than previously assigned. For instance, given the previous variable assignments, the MATLAB variable `pi` can be dynamically reassigned to a string type in this manner:

```
pi = "sweet potato"
```

2.1.2 Comparison of MATLAB and Python Syntax

This section focuses on a few of the syntactic differences between MATLAB and Python. This list is by no means exhaustive and is primarily intended to give the reader a sense of the disparities between the two languages.

Semicolons

As discussed in Section 2.1.1, using a semicolon at the end of a line of code in MATLAB suppresses the display of the output from that line. In Python, the use of a semicolon has no effect whatsoever. For example, entry of the following lines that are slightly modified from [9]

```
1 >>> x = 30
2 >>> y = 40
3 >>> z = x + y
4 >>> z;
```

will result in the following output:

```
70
```

In the above code, we assigned the variable x the integer value 30, y the integer value 40, and z the sum of x and y. The value of the variable z was output to the terminal despite the inclusion of the semicolon in line 4 because the variable name is the only statement on the command line. The main point to notice from this example is that Python will output the result of the statement entered into the command line irrespective of the inclusion of a semicolon whenever the statement evaluates to anything other than None (i.e., Python's notion of a "null" or non-value). When executing a program or function, no output is sent to the terminal unless a statement specifically calls for it (e.g., a print statement).

Commenting

In MATLAB, single-line comments begin with a % character, while in Python they begin with a #. For this paper, all comments, whether written in Python or MATLAB, are highlighted in green to improve the readability of the code. By convention, inline documentation is written differently in MATLAB than in Python. In MATLAB, functions are documented with multiple single-line comments at the beginning of the function [9]:

```
function [product] = multiply(n,m)
% MULTIPLY  Multiplies two numbers
%   PRODUCT = MULTIPLY(N,M) multiplies N and M together
```

The Python convention, on the other hand, uses multi-line comments, referred to as documentation strings or *docstrings*, to document functions and classes [9]. For example, a Python version of the above function would use docstrings as follows:

```
def multiply(n, m):
    """Multiplies two numbers together.

    Example
    -------
    >>> product = multiply(30, 40)
    >>> product
    1200

    """
```

Leading Whitespace

In MATLAB, code blocks such as "if statements, for and while loops, and function definitions [terminate] with the end keyword" [9]. The following code snippet provides a

simple example.

```
var = input("Enter an integer: ");

if var == 20;
fprintf("the input value is 20")
else
fprintf("the input value is not 20")
end
```

The example code creates the variable, `var`, to store a user-input integer and then checks to see whether its value is equal to 20. If it is, line 4 displays the phrase "`the input value is 20`;" otherwise the `else` block displays "`the input value is not 20`." It is generally considered a best practice to indent the code within a block to make it more readable, but it is not syntactically necessary in MATLAB [9]. The following block of code is semantically identical to the previous block, but the indentation makes it much easier to visually distinguish the body of the `if` block from that of the `else` block.

```
var = input("Enter an integer: ");

if var == 20;
    fprintf("the input value is 20")
else
    fprintf("the input value is not 20")
end
```

In Python, indentation is used to indicate blocks of code, and statements at the beginning of a block typically end with a colon. The following code illustrates how the previous example might be coded in Python:

```
var = int(input("Enter an integer: "))

if var == 20:
    print("the input value is 20")
else:
    print("the input value is not 20")
```

All lines of code within a block must use the same indentation.

2.1.3 Data Visualization in MATLAB

MATLAB dedicates significant functionality to data visualization and plotting. The following code modified from [10] demonstrates a simple example.

```
x = [0:0.1:20];
y = cos(x);
plot(x,y)
```

This code produces the simple cosine wave graph shown in Fig. 2.1. Standard Python does not provide similar functionality, although a number of popular open-source libraries are available if this functionality is required.

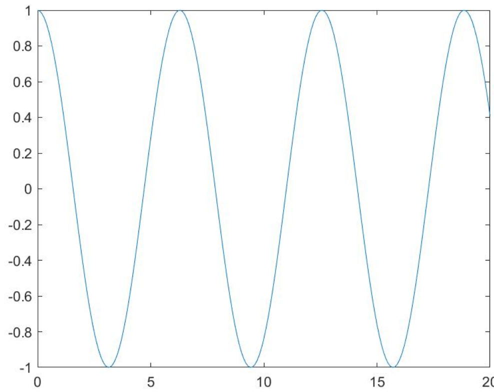


Figure 2.1. MATLAB-rendered cosine wave. Adapted from: [10].

2.2 The Python Programming Language

Python is a high-level, general-purpose OOP language with a user-friendly syntax. It is open-source and provides extensive libraries that support a wide range of applications with many pre-defined functions. Python has gained a broad user community in academia and industry. It is used in many areas, such as automation, software and web development, Artificial Intelligence, Machine Learning, system and network administration, research, and game development. We already covered some of the syntactic differences between Python and MATLAB in the previous section. In this section, we focus primarily on the major concepts of Python that pertain to our thesis. We begin by discussing the origins of OOP and provide a definition. We then introduce the concept of *classes* as a basic building block of OOP and conclude with an in-depth exploration of the four fundamental concepts of OOP.

2.2.1 Object-Oriented Programming (OOP)

The origins of OOP can be traced back to the efforts of two Norwegian computer scientists, Ole-Johan Dahl and Kristen Nygaard, in designing the programming language *Simula* that modeled real-world objects and their processes [11]. OOP is a programming paradigm that focuses on the functionality and interactions of objects. In everyday language, we describe an object as something that has a physical representation–something that can be seen and touched. In computer programming, "an object is a bundle of related state (variables) and behavior (methods)" [12]. As such, Gönther Blaschek notes that “objects are more than just data [in that they] can also perform actions” [13]. Blaschek captures this concept well with the equation

$$object = state + behavior \tag{2.1}$$

where *state* represents the data values in the object, and *behavior* encompasses the actions that the object can perform. Data is encoded as *attributes* that describe a specific object, and functionality is defined by a set of *methods* or functions that the object can perform. By grouping related attributes and methods together, OOP enables programmers to create complex programs that are implemented with relatively simple structures. These programs can effectively be summarized as a collection of largely autonomous agents (objects), each tasked with carrying out a specific job but interacting with each other as needed [14]. Glady Booch formally defines OOP as “a method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships” [15].

This definition introduces some fundamental concepts associated with OOP–that objects are the basic building blocks of a program, that the objects represent a class, and that there exists some type of inheritance relationship among these classes. Now that we have defined what objects are, we will next explore the fundamental structure for representing them in Python.

2.2.2 Classes

Classes are the foundational programming concept of OOP. We can think of a class as a group of objects that have similar properties and behaviors. In the real world, a class of automobiles, for example, can include sedans, buses, and trucks. A particular Ford Focus would be an instance of that automobile class. By the same reasoning that one would not include a bicycle in a class of automobiles, objects that do not have both a shared structure and behavior do not belong in the same class; they are unrelated by definition.

In OOP, a class is the programmatic representation of a type of object and provides a template for creating new instances of that object [16]. Source Code 2.1, adapted from [17], defines an `Employee` class to demonstrate this concept. In this example, the properties and behaviors of the class would be its attributes and methods. The values assigned to the `id`, `name`, and `job_title` attributes of each instance of the class uniquely define each object. Collectively, the attribute values of a specific instance specify the *state* of that `Employee` object.

```
class Employee:     # new Class definition
    # Initialize the attributes
    def __init__(self, id, name, pay, job_title=None):
        self.id = id
        self.name = name
        self.pay = pay
        if job_title != None:
            self.job_title = job_title

    # set the attributes (setter methods)
    def set_title(self, job_title):
        self.job_title = job_title

    # return the attributes (getter methods)
    def get_name(self):
        return self.name

    def calc_weekly_pay(self, hrs):
        ot = (hrs - 40)*1.5*self.pay
        return 40*self.pay + ot
```

Source Code 2.1: Sample Python class structure. See Appendix C.2 for a complete implementation. Adapted from: [17].

Classes simplify the creation of objects by facilitating reusability of code. The `Employee` class is a template from which uniform instances of individual `Employee` objects can

be created. Blaschek summarized a class as an abstract concept that aids programmers in identifying objects with the same characteristics and distinguishing objects with different structures and behaviors [13].

2.2.3 Fundamental Concepts of OOP

Four fundamental concepts capture the essence of OOP: *encapsulation*, *abstraction*, *inheritance*, and *polymorphism* [18]. In this section, we provide an overview of each.

Encapsulation

Encapsulation is a mechanism that prevents users from directly accessing variables that hold data within a class. It forms a sort of "protective barrier" around the objects of a class while providing a safe means of gaining access to those objects through *methods*. These methods provide external entities and users of the class with a "public interface" that they must use to interact with objects of the class. Users cannot see or manipulate an object's attributes or utilize methods that have not been provided as part of the public interface.

The class structure hides some information from external entities by only allowing controlled access to the data through its methods. In the definition of the `Employee` class in Source Code 2.1, for example, it might be prudent to restrict external entities from accessing the `id` and `name` attributes directly. The code accomplishes this by providing two methods as part of the public interface: `get_name()` and `set_title`. The `get_name()` method is a "getter" function that provides a read-only means of retrieving the employee's name. The `set_title` "setter" method provides a means of updating the title of the employee and could be written in a way that enforced specific naming requirements (this functionality is not incorporated into the example). No method is provided in the interface to change an employee's `name` attribute or access an employee's `id` attribute. Another useful aspect of encapsulation is that it allows the implementation to be changed without affecting the interface (i.e., objects of the class will retain the same functionality). For instance, if there was a future requirement to break up the `name` attribute into first and last names, we could do so without requiring changes beyond the class definition.

Blaschek stresses that this form of information hiding is vital to OOP for two reasons. First, some programs may have intricate data structures whose components should not be freely

accessed by clients. Second, if the underlying implementation of the class changes, the external interface would not need to be updated [13]. Without encapsulation, clients that relied on the class would all need to be updated to conform to any class implementation changes.

Abstraction

Abstraction is a form of implementation hiding, where the details of the inner workings of a system are hidden from its users. Once again, consider the `Employee` class example from above. Assume that the system user is a payroll administrator who needs to calculate an employee's pay for the week. To do this, they call the `calc_weekly_pay()` function and pass it the number of hours the employee worked for the week. The payroll administrator does not know whether the returned value resulted from a computation or a lookup of the employee's pay attribute; they only see that it accomplished their objective. By allowing us to "focus on what the [system] does, rather than on how it does it [18]," abstraction hides irrelevant details and simplifies how users interface with a system.

Inheritance

Inheritance allows a class to hierarchically derive its methods and properties from another class. It implies an "is a" relationship between two classes in which a *child* class or *subclass* is a more specific type of a more general *parent* class or *superclass*. Inheritance makes it easier to define new classes from existing ones by allowing child classes to inherit characteristics and functionality of the parent class while also adding their own. Inheritance is implicitly transitive, meaning that subclasses can inherit from superclasses multiple levels deep [14]. Figure 2.2 below illustrates the inheritance principle by defining new classes that inherit from the `Employee` class of the previous examples.

In Figure 2.2, the `SalaryEmployee` and `HourlyEmployee` subclasses are derived from the `Employee` superclass, and so they inherit all the attributes and methods of the `Employee` class. The `CommissionEmployee` class is a subclass of the `SalaryEmployee` and therefore inherits the attributes and methods of the `Employee` superclass in addition to those of the `SalaryEmployee` class. The inheritance property enables code reusability, consistency of interfaces, and allows polymorphism.

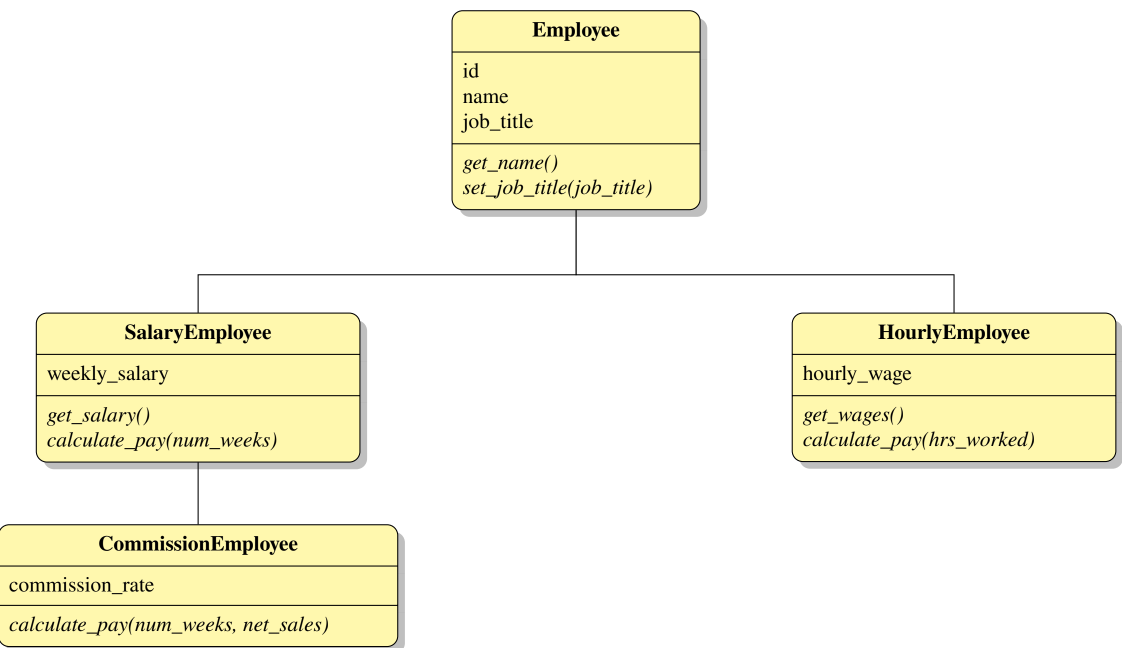


Figure 2.2. Sample class hierarchy of an `Employee` class showing the class attributes, class methods, and the inheritance relationships with child subclasses. See Appendix C.2 for a complete implementation. Adapted from: [17].

Polymorphism

In the context of OOP, polymorphism allows a subclass to redefine structures and methods inherited from its superclasses. That is, it can *override* any method or property inherited from a superclass by redefining it in its own class specification. In Figure 2.2, both the `SalaryEmployee` and `CommissionEmployee` subclasses have a `calculate_pay()` method. For objects of the `CommissionEmployee` class, the `calculate_pay()` method of the subclass overrides the inherited `SalaryEmployee` superclass implementation. Polymorphism implies that the correct implementation of a method is called for the correct class (i.e., the implementation most directly associated with the calling object will be invoked). In our example, this means that the `calculate_pay()` method of the `CommissionEmployee` class is invoked for objects of that class and the `SalaryEmployee` implementation is called for objects of the parent class. Polymorphism associated with higher-level components is automatically inherited by child classes lower in the inheritance tree.

2.3 Docker

Docker is an open-source platform for developing, distributing, and running applications through software called containers. A container can be defined as "a standard unit of software that packages up code and all its dependencies, so the application runs quickly and reliably from one computing environment to another" [19]. Instructions for building a container exist within a layered file system called a Docker image [20]. The basic components of a Docker image include a lightweight version of an operating system (OS), a runtime environment, application files, third-party libraries, and environmental variables [21]. Docker images are created by execution of a Dockerfile, which is a "text document that contains all the commands a user [executes] on the command line to assemble an image" [22].

The Docker architecture is depicted in Figure 2.3. In this depiction, a Docker client issues commands to the Docker daemon, which manages both the images and containers. The Docker registry is a service that stores and distributes Docker images which can be made either publicly or privately accessible. The Docker client, daemon, and various other Docker services can all be run by utilizing the Docker Desktop application, which is available for MacOS, Windows, or Linux environments [23]. Before the advent of Docker containers, VMs were the primary technology to offer similar functionality. Figure 2.4 provides a visual comparison of applications running on Docker containers and VMs.

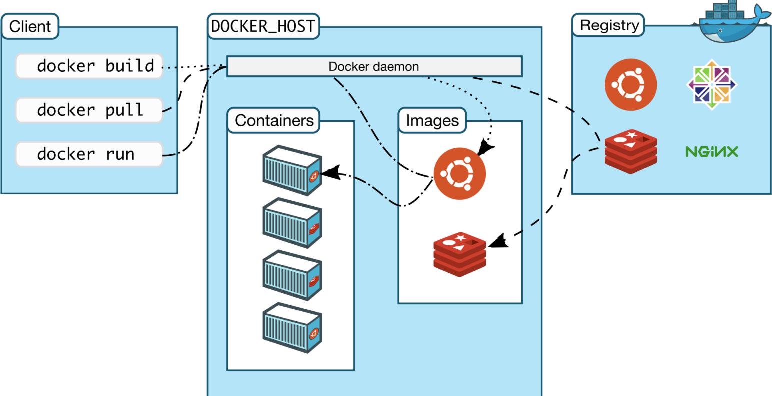


Figure 2.3. Docker architecture. Source: [23].

A basic definition of a virtual machine (VM) is "software that runs programs or applications without being tied to a physical machine" [24]. It is a virtual (i.e., software-defined) com-

puter that exists like any other program installed on a physical computer. The host machine provides the infrastructure, which includes all the resources (e.g., memory, central processing unit (CPU), storage) shared by multiple hosted VMs. The hypervisor, or virtual machine monitor, is "the code responsible for managing virtual machine guests on a physical host machine" [25]. It allows multiple VMs to be hosted on the same physical machine at the same time, with each VM having its own guest OS, applications, and dependencies. Each VM is completely isolated from the others and is also isolated from the physical host's OS. VMs can take up significant resources to host each guest OS. VM images can be copied, deleted, or migrated to other physical hosts, just like regular computer files.

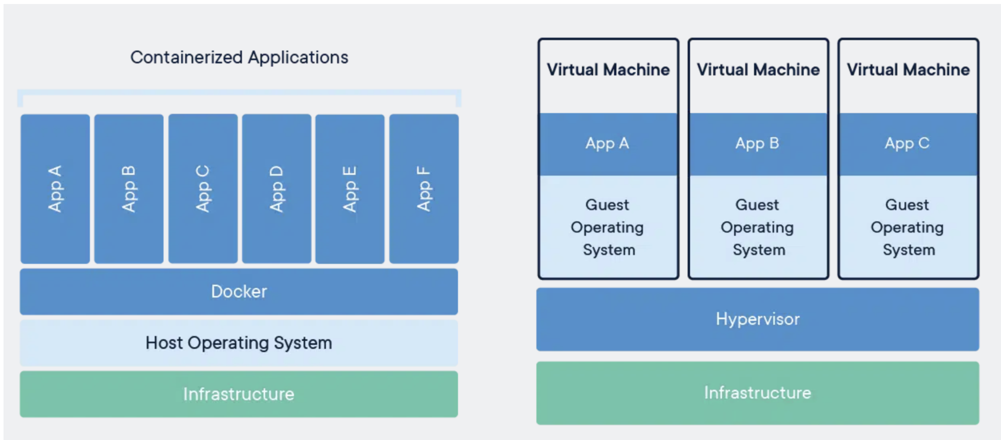


Figure 2.4. Depiction of applications running on Docker containers in comparison to VMs. Source: [19].

VMs differ from Docker containers in that they virtualize the hardware on the physical host, while Docker containers virtualize the OS in the application layer [19]. This allows containers to eliminate the additional overhead required to host a guest OS. When comparing Docker containers and VMs, Babu Kavitha and Perumal Varalakshni concluded that Docker containers are superior "in terms of CPU Performance, Memory throughput, Disk I/O, Load test, and operation speed measurement [benchmarks]" [26]. The findings that containers almost always equal or outperform VMs were mirrored in [27]. Allison Randal argues that containers provide more secure isolation than VMs because "containers take a modular approach to implementation that permits them to be more flexible over time and across different underlying software and hardware architectures" [25]. However, VMs retain an edge in some instances. For example, they can be useful when there is a need to maintain

support for legacy applications that can only run on obsolete OSs [24]. In contrast to the findings above, Manco, et al. [28] demonstrated that a VM instantiation of an application performed significantly faster than a Docker container instantiation of the same application.

Ultimately, while both architectures have their advantages and disadvantages, Docker containers are better suited for our work for two significant reasons. First, Docker containers are less resource intensive, which makes them easier to build, modify, and test on a single host. Second, their small size makes containers well-suited for deployment on memory-constrained platforms like UxVs.

2.4 Networked Control System (NCS)

As discussed in Chapter 1, this thesis builds on the works of Lowry [4] and Wachlin [6]. A key concept of their work was the idea of submodularity. Tyler Summers and Maryam Kamgarpour best describe submodularity as "a diminishing returns property, [whereby] adding an element to a smaller set gives a larger benefit than adding it to a larger set" [29]. This property was used to increase the NCS's total network utility. An in-depth analysis of how the submodular utility function was selected to evaluate the NCS is available in [6]. The resulting submodular total utility function, J, is represented by Equation 2.2

$$J(S) = \alpha_s f_s(S) + \alpha_r f_r(S) \tag{2.2}$$

In this function, the position set, S, is the optimal set of all possible node locations in a 100-meter by 100-meter grid, and f_s and f_r are the sensing and communications-robustness utility functions that are respectively assigned the weights α_s and α_r [4]. For the sensing function "in the MTX scenario, the road network, NSW team, and target objective all have high sensing values, so a node positioned closer to one of these would contribute to a higher sensing utility f_s than one positioned farther away" [4]. The robustness utility was calculated by measuring connectivity to ensure effective communications between nodes. This was done by measuring the effective graph resistance by increasing the edge weight as the distance between nodes increases so that the graph equates to increased resistance to communication (i.e., a less robust network) [4].

In their MATLAB implementation, Lowry and Wachlin compared two distinct methods for determining node placement: the centralized and distributed submodularity (DS) methods. The centralized method utilizes a single controller to calculate all of the nodes' optimal positions [4]. Node placement must be done in a specified sequence, and nodes cannot be repositioned after the initial placement. In contrast, the DS method uses an iterative approach to the node placement process that allows nodes to be placed in any order and repositioned multiple times until the best possible total network utility is achieved. The DS approach is more robust and is the method upon which most of this work focuses.

2.5 Related Work

In an ongoing research effort to convert MATLAB Bridge modules into a Python-integrated 3DSlicer, Sharon Peled and Andras Laso [30] began their implementation by mapping out the MATLAB Bridge module function dependencies. They concluded that MATLAB to Python converters would not work, and that conversion would have to be done manually. In [31], Sollfrank, et al. explored how "containerization supports platform independent development and deployment of secure and isolated applications," on a cyber-physical-system-based NCS. They determined that Docker containers are capable of supporting real-time NCS applications so long as the container is provided the highest real-time process priority. However, Sollfrank et al. noted a slight delay of approximately 43 μs in the round-trip-total time in the containerized version of a user datagram protocol client-server application [31]. Lastly, Gergely Imreh [32] proved that containerized Docker applications could successfully run on a unmanned aerial vehicle (UAV). During a keynote speech at Dockercon 2016, Imreh gave a live demonstration where he updated the drone's software while it was still in flight by creating a "user application [...] as a Docker container from the source pushed with git to the resin.io servers" [32]. The drone then synchronized with the servers and "pulled" an updated container once it was made available.

2.6 Chapter Summary

In this chapter, we introduced both the MATLAB and Python programming languages. We discussed basic syntax rules for MATLAB and provided examples whenever they differed from Python. For Python, we introduced the concept of OOP and `classes` and

discussed the four fundamental concepts of OOP: abstraction, encapsulation, inheritance, and polymorphism. We also introduced the concept of Docker containers, discussed VMs, and provided a justification for choosing Docker containers over VMs for our implementation. Lastly, we reviewed the NCS and discussed related works. Chapter 3 describes our methodology for converting the MATLAB code to Python.

CHAPTER 3: Methodology

This chapter discusses our methodology for converting MATLAB code to Python. We begin by evaluating the feasibility of automatic converters to automate the job for us in Section 3.1. Next, we use function mapping tools (discussed in Section 3.2) to better understand the functions in the MATLAB code and how they interact with each other. Lastly, we present our conceptualized design for the Python classes and propose a class hierarchy in Section 3.3.

3.1 Automated MATLAB-to-Python Code Conversion

Several tools exist to automatically parse and convert MATLAB code to Python. *Small MATLAB and Octave to Python compiler (SMOP)*, *matlab2python*, *Open-Source MATLAB to Python Compiler*, *Mat2py*, and *Libermate* are but a few. From this list, we selected SMOP for evaluation because, according to *Python Pool*, it is considered among the best available tools [33]. Although we ultimately could not get our test code to run with SMOP, our experimentation was partially fruitful. The test code and resulting conversion are provided in Appendix A. We concluded that SMOP and similar tools would more accurately be classified as code wrappers rather than code converters because they "wrap" the pre-existing code with an outer layer of code that a particular interpreter can understand. In essence, the translated code still reads like conventional MATLAB syntax but can be run in Python environments. Based on this observation, we determined that this tool was not suited for our thesis since our objective was the full implementation of the NCS in Python. The only option left was for manual code conversion.

3.2 Function Mapping

There are two types of MATLAB source code files (called m-files): scripts and functions. m-file *scripts* do not take inputs or return results–they simply operate on data in their *Workspace*. m-file *functions*, on the other hand, can accept inputs and return results. For this thesis, we use the term *function* interchangeably to refer to both m-file types. Our initial

approach to determining the dependencies between functions in the MATLAB code was a manual evaluation. We methodically examined the code to identify every function and then determined when and in what order they were called. Ultimately, this approach proved too cumbersome to be feasible. Our alternative approach was to identify a suitable automated tool for building function inter-dependencies.

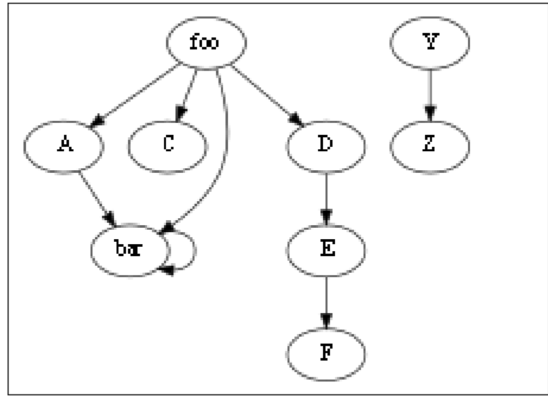
One tool that is widely used to generate documentation from source code for traditional OOP languages is Doxygen [34]. It is freely available and can generate user-friendly output in HTML format. A similar tool that automates the generation of HTML documentation for MATLAB files is MATLAB to Hypertext Markup Language (M2HTML) [35]. It evaluates a MATLAB file and generates a full list of functional dependencies. When used in conjunction with visualization software, this list can be displayed as a graphical rendering of the relationships between functions. M2HTML integrates well with *Graphviz*, an open-source graph visualization software for representing abstract structural information [36]. For our implementation, we settled on M2HTML. In demonstrating M2HTML's functionality, consider the following MATLAB list adapted from [37]. In this example, function dependencies are depicted as pairs ('X,' 'Y'), indicating that function X calls function Y:

```
calls = {'foo','A'; 'foo','C'; 'foo','D'; 'foo','bar'; 'D','E';
         'E','F';   'A','bar'; 'Y','Z';   'bar','bar'};
```

M2HTML utilizes the dot tool of *GraphViz* to generate a 2D rendering of the function dependencies. The output of this example is shown in Fig. 3.1, where the nodes represent the various functions and the directed edges represent the caller-callee relationships. The complete MATLAB m-file and the associated HTML document produced are provided in Appendix B.

```
digraph G {
   "foo" -> "A"
   "foo" -> "C"
   "foo" -> "D"
   "foo" -> "bar"
   "D" -> "E"
   "E" -> "F"
   "bar" -> "bar"
   "A" -> "bar"
   "Y" -> "Z"
}
```

(a) Dot file



(b) Graphic rendering

Figure 3.1. Sample *GraphViz* dependency graph depiction. *GraphViz* renders the dependencies defined in the Dot file as a directed graph in which nodes represent functions and vertices represent the caller-callee relationships. Adapted from: [37].

We used M2HTML and *GraphViz* to produce the dependency graph in Figure 3.2 from the MATLAB NCS m-files. We adopted this figure as a starting point for designing our Python class definitions.

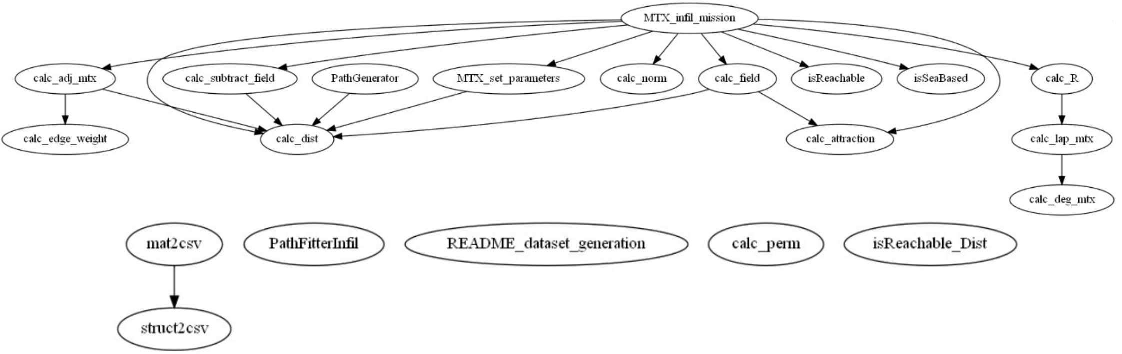


Figure 3.2. *GraphViz* rendering of the NCS functions and their dependencies.

3.3 Conceptualizing Python Classes

This section discusses the conceptualized Python classes and presents the initial design for our class hierarchy. Before commencing the code conversion process, we used a top-

down approach in order to gain a high-level understanding of the MATLAB program. From the dependency graph depicted in Figure 3.2, the logical starting point was the `MTX_infil_mission` file. We also wanted to understand what each function accomplished. We noted that the first function call was made to `MTX_set_parameters`. This function defines the general scenario parameters for the NCS simulation. Based on our review of all the functions in the dependency graph, we conceptualized our class objects and the class hierarchy design as depicted in Figure 3.3.

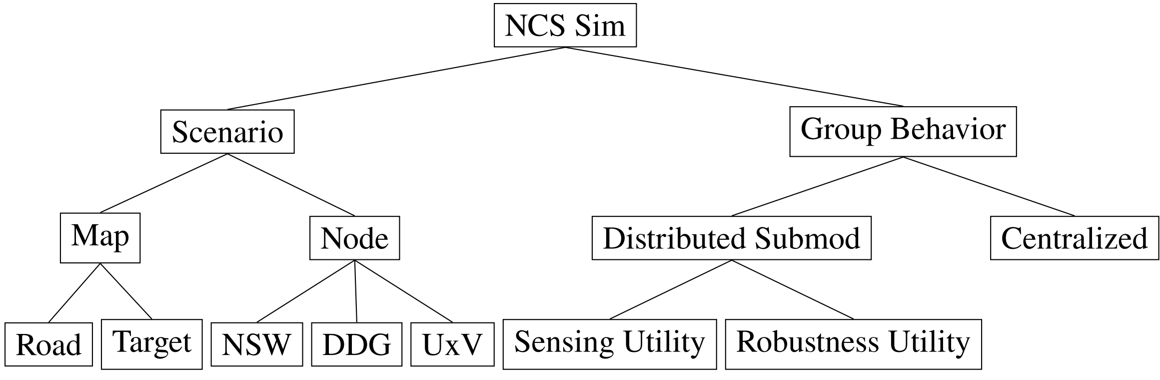


Figure 3.3. Proposed NCS Python class hierarchy.

In this hierarchy, `NCS Sim` is the class that runs the entire simulation. The `Scenario` class defines the physical parameters, including the location, players, and objective. The `Node` class contains the actors in this scenario. Its subclasses are the virtual leader, a `NSW` team class, a `DDG` class, and a `UxV` class representing different types of unmanned vehicles with distinct characteristics. The `Group Behavior` class represents functionality for manipulating collective node behaviors. Its `Distributed Submod` subclass represents node placement using the DS method, and its `Centralized` subclass represents node placement using the centralized method. The `Sensing Utility` class is used to maximize the coverage of the UxVs over high-value locations (i.e., the NSW team, the road network, and the target objective), and the `Robustness Utility` is used to maximize the communications connectivity between the nodes in the scenario.

3.4 Chapter Summary

This chapter outlined our methodology for converting the MATLAB code to Python. We explored the use of automatic converters, discussed function mapping tools, and demonstrated

the effectiveness of pairing M2HTML with *GraphViz* to generate function dependencies. Finally, we presented our conceptualized Python classes and proposed a class hierarchy for our Python NCS, the implementation of which we discuss in Chapter 4.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 4:
Implementation and Evaluation

This chapter covers the process we used to implement our code conversion. Section 4.1 describes the iterative software development process that we followed to refine and implement the initial design presented in Section 3.3. In Section 4.2, we validate the correctness of our final Python implementation by comparing its results with those of the original MATLAB implementation. We also present a performance evaluation based on execution times.

4.1 Python Design Implementation

We start this section with a step-by-step explanation of the iterative process we utilized to develop and debug our Python program. We then describe the different types of relationships that exist between elements in our implementation. Finally, we present the optimized class hierarchy and structure for the simulation.

4.1.1 The Multi-Step Iterative Process

We used an iterative approach to translating the MATLAB code to Python. Most importantly, translating the code in small blocks enhanced our ability to debug coding errors before the program expanded in length and complexity. Figure 4.1 provides a road map of our multi-step iterative process, and the following sub-sections elaborate on the methods employed at each stage of the process.

① **Pseudocode Review** Our first objective was to gain a high-level understanding of the code from the Lowry thesis [4, p. 25]. Reviewing this pseudocode gave us a scaled-out view of the original work's methodology and facilitated our understanding of the scope of our work. The pseudocode review included mapping each line of the pseudocode to its associated MATLAB implementation.

② **Scenario Initialization Review**

After gaining a high-level perspective of the code, we next sought to understand the program flow starting from initialization. The *GraphViz* dependency graph of Figure 3.2

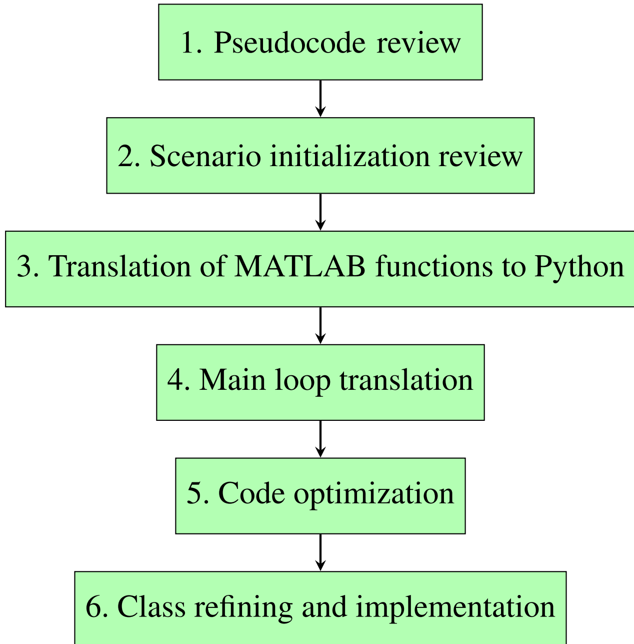


Figure 4.1. Flow chart of the iterative process used for converting MATLAB code to Python. See Appendix C.1 for a more granular flow chart.

indicates that `MTX_infil_mission.m` is the entry point for the program and initializes and runs the scenario. However, while this rendering accurately depicts the function dependencies, it does not indicate the order in which the functions are called. For example, `MTX_infil_mission.m` is depicted with ten dependencies. Only by examining the actual code does it become clear that `MTX_set_parameters.m` is the first function called. This function defines the parameters that characterize the SCI Scenario. These parameters include variables derived from the geographical topography and user-defined variables. For example, `MTX_set_parameters.m` reads a .jpg image of a map of SCI and stores it in an "image" variable that is then scaled according to a user-defined variable. The `MTX_infil_mission.m` file saves all the parameters to a `.mat` file. This file is imported into our Python program to initialize the SCI scenario.

③ Translation of MATLAB Functions to Python

Armed with an understanding of the overall structure of the code and how it is initialized, we commenced our code translation. Because some functions depended on others, we started by first converting the leaf nodes on the dependency graph in Figure 3.2. An unintended

benefit of this approach was that these functions were generally only a few lines of code and, thus, the easiest to convert. After converting each MATLAB function to an equivalent Python function, we tested for correctness. For each MATLAB function, we noted both the input arguments at the start of the function execution and the output values just before the function returned. We then used this data to verify the functionality of our Python functions.

We demonstrate a successful conversion of the `calc_adj_mtx` function, which calculates a communications adjacency matrix from MATLAB to Python in Source Codes 4.1 and 4.2. Source Code 4.3 shows the test script we utilized to check for correctness in the function.

```
function [A] = calc_adj_mtx(pos,rmax)
% The Adjacency matrix describes the interaction strength among the agents in the group
% and is given by a_ij(q)=\rho_h(||q_j-q_i||_\sigma / d_x) \in [0,1]
% where d_x is a constant representing the maximum communication distance

[n,~]=size(pos);
A = zeros(n);

for i=1:n
    for j=1:n
        if (i~=j)
            r = calc_dist(pos(i,:), pos(j,:));
            if r < rmax
                A(i,j) = calc_edge_weight(r,rmax);  %% rmax = cmax = 500
            end
        end
    end
end
end
```

Source Code 4.1: MATLAB `calc_adj_mtx` function for calculating the communication system adjacency matrix.

```
# ---------- calc_adj_mtx -----------------------------------------------------
def calc_adj_mtx(pos, rmax):
    """The Adjacency matrix describes the interaction strength among the
    agents in the group and is given by:

    a_ij(q)=\rho_h(||q_j-q_i||_\sigma / d_x) \in [0,1]

    where d_x is a constant representing the maximum communication distance.
    """

    n = np.shape(pos)[0]
    A = np.zeros([n, n])

```

```
14     for i in range(n):
15         for j in range(n):
16             if i != j:
17                 r = calc_dist(pos[i, :], pos[j, :])
18                 if r < rmax:
19                     A[i, j] = calc_edge_weight(r, rmax)
20     return A
```

Source Code 4.2: Python version of the MATLAB `calc_adj_mtx` function.

The `calc_adj_mtx` function in Source Code 4.2 serves as an example of the importance of understanding the function dependencies before code conversion. The function `calc_dist` is called in line 17 and `calc_edge_weight` in line 19. Therefore, the Python version of `calc_adj_mtx` would not work unless these two functions were translated to Python first.

```
1 # ---------- TESTING: calc_adj_matrix -----------------------------------------
2 # test parameters set
3 cmax = 500
4 pos = np.array([[444.7388, 452.4329],
5                 [853.9999, 220.0796],
6                 [1.0, 1.0]])
7
8 # calcualte adjacency matrix and round values to 4 decimals
9 A = calc_adj_mtx(pos, cmax)
10 A = np.round(A, decimals=4)
11
12 # A_test is our expected output to test against
13 A_test = np.array([[0.,    0.9144, 0.],
14                    [0.9144,    0., 0.],
15                    [0.,        0., 0.]])
16
17 # compare our output with expected output to test for correctness
18 if np.array_equal(A, A_test):
19     print("+ calc_adj_matrix test SAT")
20 else:
21     print("- calc_adj_matrix test FAILED")
```

Source Code 4.3: Python test script for the `calc_adj_mtx` function translation.

In the Source Code 4.3 test script, the test input and anticipated output are encoded in lines 3-6 and lines 13-15, respectively. Both were captured by setting breakpoints in Source Code 4.2 at lines 6 and 19, respectively. Some test failures necessitated testing with additional input-output combinations due to the requirement for some functions to accept multiple data types as inputs. For example, a function might take a float type input on one

iteration and an array type on the next. Line 10 rounds the output to four decimal places to match MATLAB's output; otherwise, the test would fail due to float data type precision rounding. Line 18 performs the comparison that determines whether or not the test was a success.

④ Main Loop Translation

Once all the functions were individually validated, we began translating the program's main loop. The methodology was similar to the one described in the previous section for converting and testing individual functions. We initially intended to execute the Python program in one main loop as in the MATLAB version. We refer to this initial approach as the *streamlined* approach. It allowed us to understand the program flow better and made resolving conversion errors easier.

We began by setting up the SCI scenario, which involved loading the initialization parameters from the imported `.mat` file. We then iterated through the MATLAB main loop line by line and translated the code into its Python equivalent. This step was not trivial as translations between MATLAB and Python are not always direct. For example, consider the MATLAB command

```
snapshots = [100 500:300:length(NSW_Path)]
```

which creates an array variable with values starting at 100, followed by 500, and increments by 300 thereafter. There is no equivalent Python command, so multiple steps are required to achieve the intended result:

```
# create array of starting value
head = np.array([100])

# create array of tail end
tail = np.arange(500, len_NSW_Path, 300)
tail = np.array([tail])

# concatenate the them into one array
snapshots = np.concatenate((head, tail), axis =None)
```

⑤ Code Optimization

In an effort to improve execution times, we considered several performance-enhancing solutions and ultimately adopted Cython for its utility, relative ease of use, and compatibility. Cython is a superset of the Python language that acts as a bridge between Python and

C or C++ [38]. Cython improves the performance of Python code without requiring any prior understanding of C++ by translating it into optimized C/C++ code that can be compiled [38]. A setup file is used to specify all the target files to be compiled. Simply compiling a Python file can bring notable performance improvements. For additional performance improvements, Python code can be "Cythonized" by using Cython-specific syntax to declare variable types. This approach allows a programmer to run the code either as a Python or as a Cython script and allows for easy performance comparison. Additional analysis of Cython and a "Cythonized" code sample is provided in Appendix D.

When we Cythonized our code at this stage, the largest file was 652 lines long and took approximately 1 hour and 37 minutes to compile. Once compiled, the results were promising–execution time was improved by over two orders of magnitude. As much as this was an improvement, however, the compilation time was intolerably long. We assessed that decomposing the main loop into smaller files as described in Section (6) would be necessary to improve the compilation time.

(6) Class Refining and Implementation

The product of the first phase of our conversion process–the streamlined Python version–turned out to be too complex, as evidenced by the unacceptably long Cython compilation time. The most extensive file, `NCS_sim_f.py`, had 598 lines of code and contained multiple nested loops that were costly in execution time. In our final step, we broke this large file into smaller modular files. We more rigorously defined how we wanted to represent our data and, more importantly, the class structures in which the data would be stored. The following sections detail the resolution of the relationships between classes, which resulted in the comprehensive Python class hierarchy portrayed in Figure 4.2.

4.1.2 Relationship Types

We considered three central relationship types when implementing the class hierarchy of Figure 3.3: inheritance ("is a"), composition ("has a"), and semantic relationships.

Inheritance ("is a") Relationships

Inheritance is characterized by an "is a" relationship where a child class is a subtype of its parent class. In this relationship, the parent class is referred to as the superclass, and the

child class is referred to as the subclass. The subclass is said to "derive from" the superclass. For example, in the inheritance relationship of Source Code 4.4, the child `UAV` subclass derives from the parent `Node` superclass. This indicates that a `UAV` is a specialized kind of `Node`, and similarly, a `Node` is a generalization of a `UAV`. The `UAV` subclass inherits the structure and functions of the `Node` superclass. Our finalized NCS class hierarchical representation in Figure 4.2 depicts inheritance relationships as solid lines.

```
class Node():       # Node class definition
    num_nodes = 0
    def __init__(self, id=00, name="defaultName", vel=00):
        self.id = id
        self.name = name
        self.vel = vel
        Node.num_nodes += 1

    def identify_node(self):
        print("Node id: %d of %d, name: %s, vel: %d" %(self.id, Node.num_nodes, self.name,
     self.vel))

    def set_num_nodes(self, num):
        Node.num_nodes = num

class UAV(Node):
    # constructor for initialization
    def __init__(self, id, name, vel, path):
        self.UAV_path = path
        super().__init__(id, name, vel)

# ======== Demonstrating the Node-UAV relationship ========
# instantiating a Node object
>>>nd = Node()
>>>nd.identify_node()
[Out] Node id: 0 of 1, name: defaultName, vel: 0

# instantiating a UAV object
>>>uav1 = UAV(1, "UAV1", 5, "uav_path")
>>>uav1.identify_node()
[Out] Node id: 1 of 2, name: UAV1, vel: 5

# demonstrating a UAV is-a Node
>>>uav1.set_num_nodes(9)
>>>uav1.identify_node()
[Out] Node id: 1 of 9, name: UAV1, vel: 5
>>>nd.identify_node()
[Out] Node id: 0 of 9, name: defaultName, vel: 0
```

Source Code 4.4: Inheritance ("is a") relationship example written in Python with typical runtime results.

Composition ("has a") Relationships

A composition relationship is characterized by an instance of one class being contained as a data element of another class. In the Source Code 4.5 example, the `Map` class contains a `Target` class object as one of its member variables. The definition of this composition relationship enables us to reference methods of the `Target` class instance using a `Map` class object as demonstrated in the `display_map()` class method. The NCS class hierarchical representation of Figure 4.2 depicts composition relationships as dotted lines.

```
class Target():     # Target class definition
    def __init__(self, x, y):
        self.x_cord = x
        self.y_cord = y

    def display_target(self):
        print("(%d, %d)" %(self.x_cord, self.y_cord))

class Map():        # Map class definition
    # constructor for initialization
    def __init__(self, height, width):
        self.height = height
        self.width = width
        self.target_obj = Target(350, 120)

    def display_map(self):
        # some plot map code here
        self.target_obj.display_target()

# ======== Demonstrating the Map-Target relationship ========
# instantiating a Map object
>>>mp = Map(750, 750)

# accessing the Target object via Map instance
>>>mp.display_map()
[Out] (350, 120)

# another direct method of access
>>>mp.target_obj.display_target()
[Out] (350, 120)
```

Source Code 4.5: Composition ("has a") relationship example written in Python with typical runtime results.

Semantic Relationships

The third type of relationship is one in which two classes have no inherent connection outside a specific context. A pumpkin, flowers, cornucopia, vase, and candles are seemingly

unrelated when viewed as individual objects but are related in the context of items used for decorating a Thanksgiving dinner table. Similarly, a `Target` object and a `Node` object are not strongly coupled since a `Target` can exist independent of a `Node`, and vice-versa. However, viewed in the context of the NCS simulation, the two are related in that the `Target` is the objective towards which the `Node(s)` navigate.

Although the proposed design of Figure 3.3 was correct, we incorrectly assumed that every node in the hierarchy represented an inheritance relationship. This created significant hardships during the implementation phase of our conversion process. Once we gained clarity on the different types of relationships, we adopted the following guidelines to assist in identifying them accurately [39]:

1. To determine *inheritance* relationships, ask if a child class is a type of its parent class. If so, reverse the question and ask if the parent is a generalized type of its child. There should be a clear "is a" child and "is a" parent relationship, respectively.
2. For components that can be reused by more than one class, use a *composition* relationship.
3. For classes representing behavior groups that are interchangeable with other classes, use a *composition* relationship.
4. When there seems to be no clear relationship, default to a *semantic* relationship.

4.1.3 Refined NCS Python Class Definitions

Figure 4.2 depicts a comprehensive hierarchical view of the final NCS class definitions and their relationships. Each class is represented as a rectangular box with the class name displayed across the top. The class variables are listed below the class name and the class methods below the class variables. Due to space constraints, we listed only the critical components for some of the classes. "Has a" composition-type relationships are represented as dotted lines, and "is a" inheritance-type relationships are represented as solid lines. Semantic relationships are not depicted.

To address the performance concerns identified earlier in the iterative design process, our last step was to Cythonize the Python code to improve performance. The Cythonized version of the Python NCS consisted of 13 files and over 1,250 combined lines of code, with each file averaging around 100 lines of code. Having smaller files drastically reduced the compilation

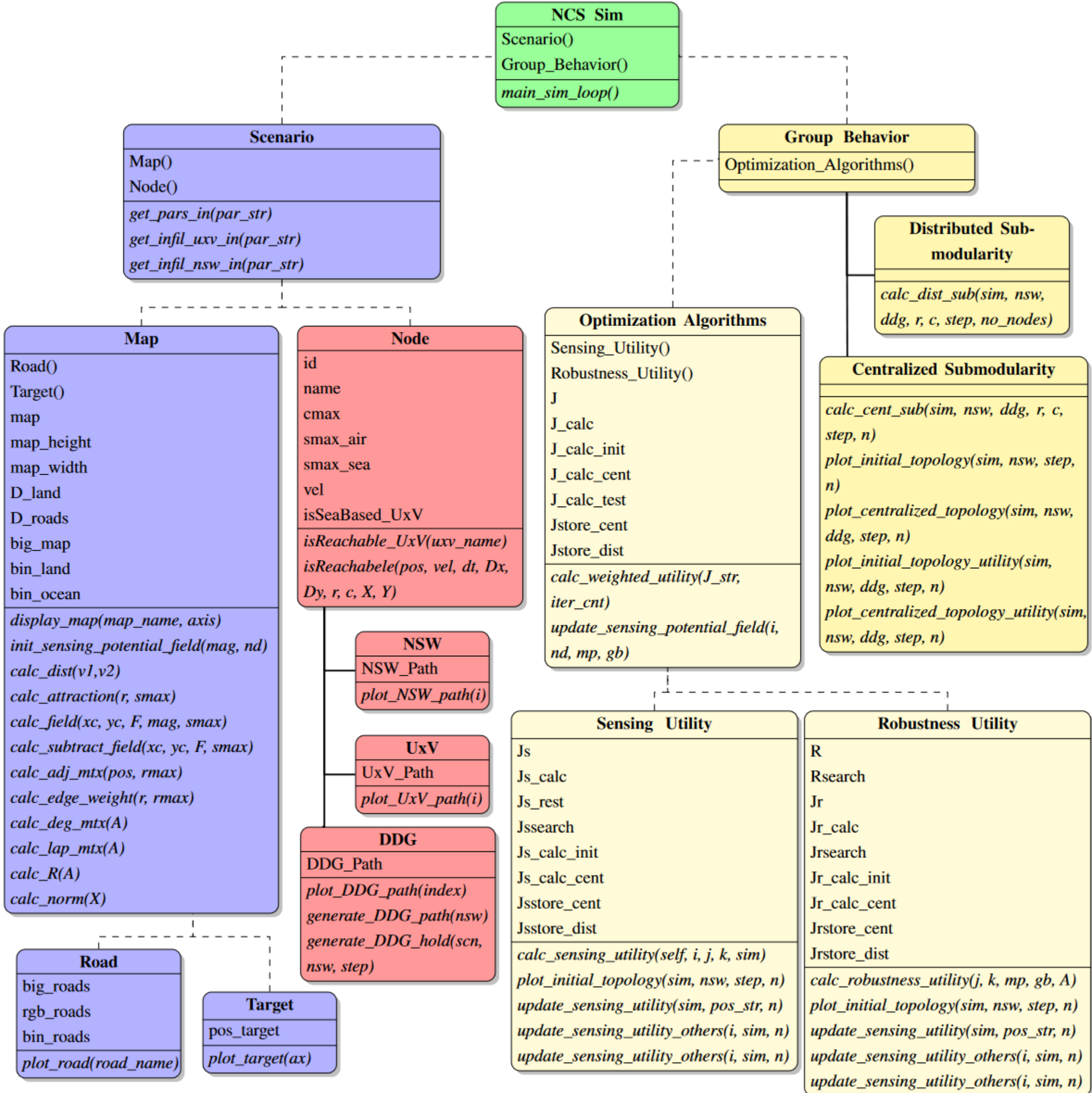


Figure 4.2. Finalized Python NCS diagram showing class structures and relationships (solid lines indicate inheritance and dashed lines indicate composition).

time to a more tolerable 5 minutes. A detailed description of our performance evaluation is discussed in Section 4.2.2.

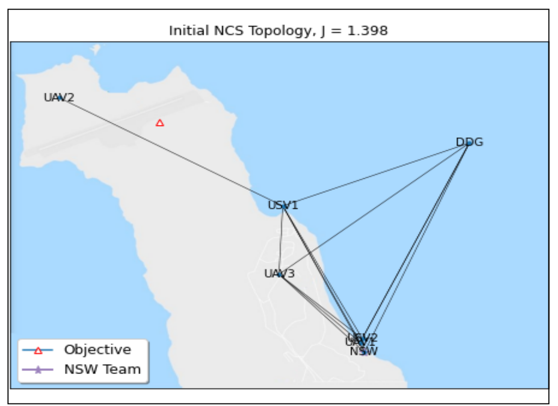
4.2 Evaluation of Design Implementation

This section provides an evaluation of our Python implementation of the NCS simulation, including an assessment of the correctness of the system (i.e., does it get the desired results) and the system's performance (i.e., how quickly it obtains a final solution). In Section 4.2.1, our results are compared against the outputs of the MATLAB implementation to validate the correctness of our code conversion. In Section 4.2.2, we compare the performance of our code to that of MATLAB.

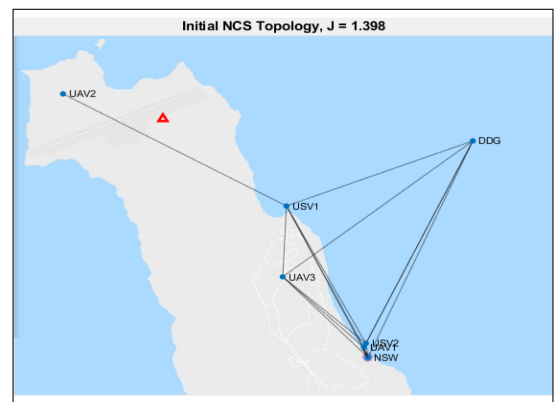
4.2.1 Correctness Validation

Figures 4.3 through 4.5 present a side-by-side comparison of the results from the Python and MATLAB implementations. In Figure 4.3, plots (a) and (b) show the initial node placements for the NCS simulation, (c) and (d) the node placements after the first iteration of the centralized method, and (e) and (f) the node placements after the first iteration of the DS method. Figure 4.4 shows the final node topologies for the NCS simulation, centralized, and DS methods, respectively.

The "J" values at the top of each plot represent the total network utility for the NCS topology at the completion of an iteration. This utility value indicates that the topology has achieved the best placement of nodes for that iteration–rearranging the nodes in any manner cannot achieve a better network utility. We compared the J values of the Python figures on the left to the MATLAB figures on the right and performed a visual inspection of the node topologies to confirm that our results were accurate. By demonstrating that our total utility values are equal to those of MATLAB, we showed that our Python implementation achieved node placements equal to the MATLAB implementation. Figure 4.5 displays the utility values for the sensing and communications robustness and the combined total utilities for each iteration. Plots (a) and (b) show the network utilities after the first iteration of the simulation, plots (c) and (d) show the final utilities after the last iteration of the simulation, and plots (e) and (f) summarizes the total utilities over all the iterations of the simulation. Our results demonstrate that the Python conversion and implementation were successful–the outputs were precisely equivalent to those of MATLAB.

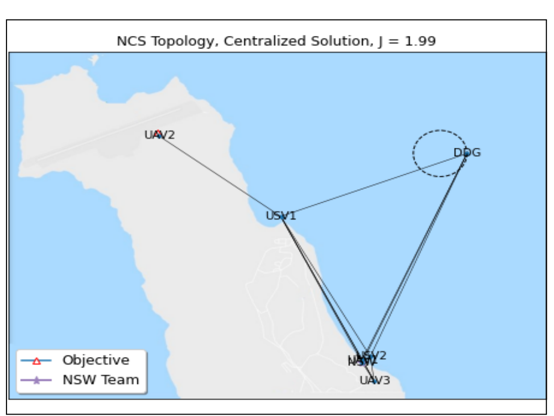


(a) Python

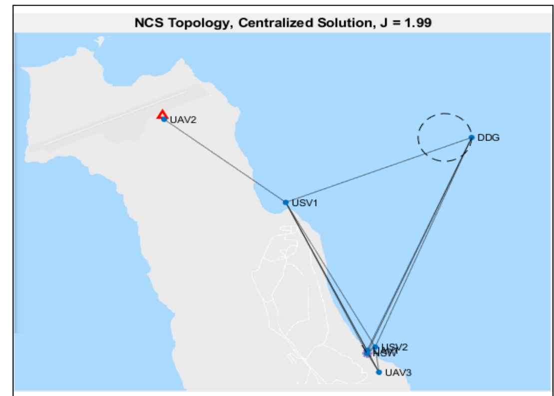


(b) MATLAB

Initial NCS node placement at beginning of infiltration phase.

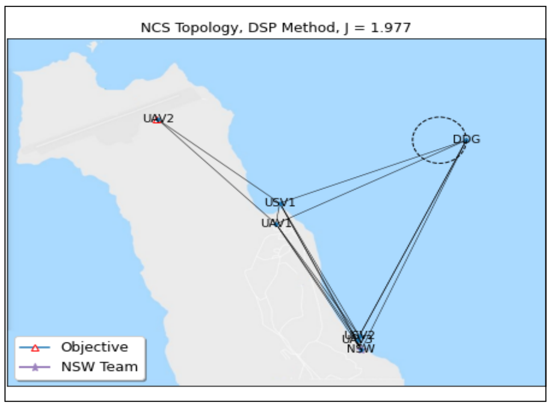


(c) Python

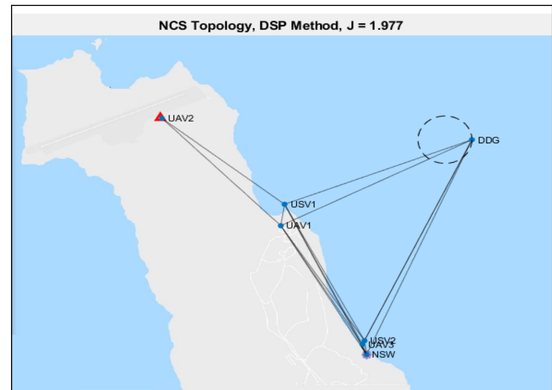


(d) MATLAB

NCS topology for the centralized method after first simulation iteration.



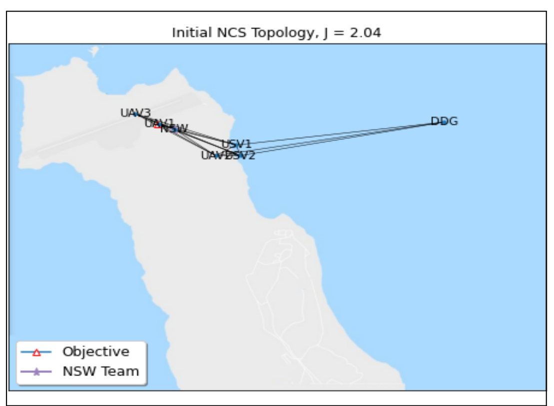
(e) Python

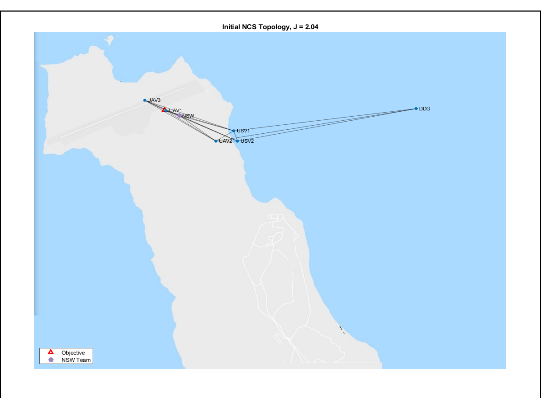


(f) MATLAB

NCS topology for the DS method after first simulation iteration.

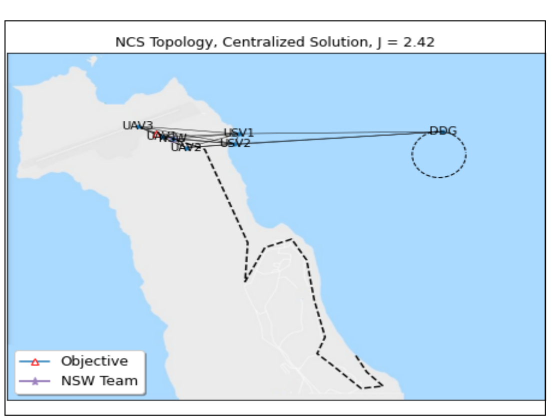
Figure 4.3. Comparison of the Python and MATLAB topology plots after the first iteration of the simulation.

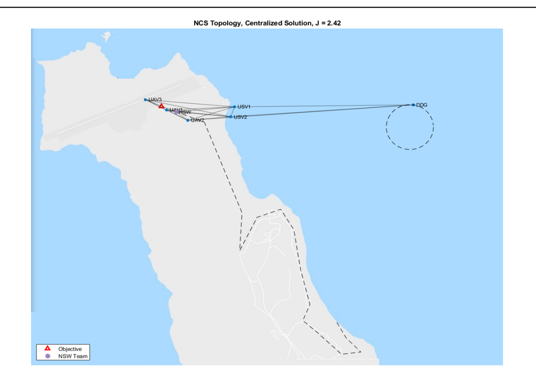




(a) Python (b) MATLAB

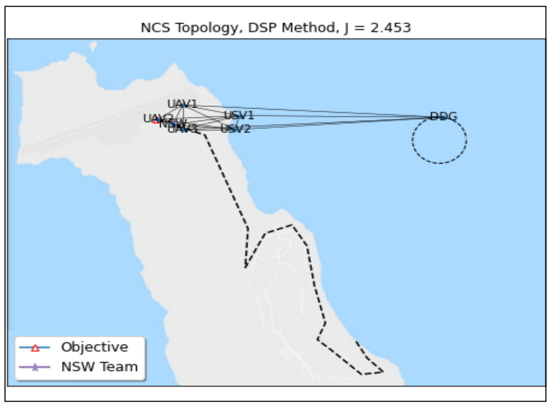
NCS topology node placement at beginning of the final simulation iteration.

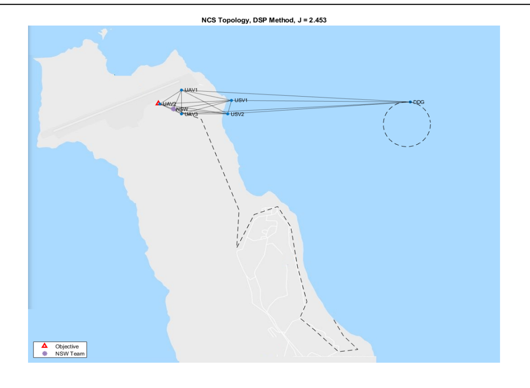




(c) Python (d) MATLAB

NCS topology for the centralized method after the final simulation iteration.

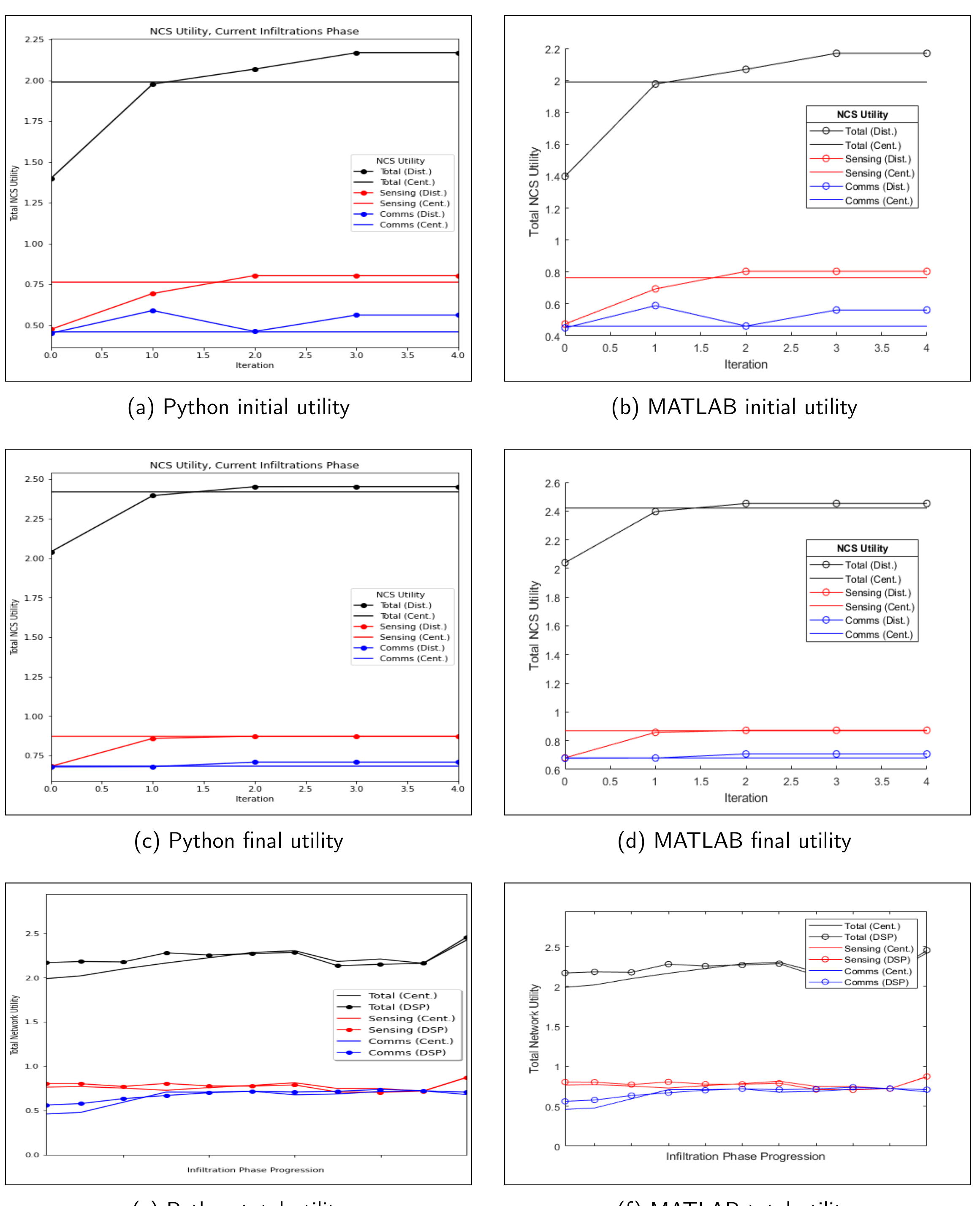




(e) Python (f) MATLAB

NCS topology for the DS method after the final simulation iteration.

Figure 4.4. Comparison of the final Python and MATLAB plots after the final iteration of the simulation.



(a) Python initial utility

(b) MATLAB initial utility

(c) Python final utility

(d) MATLAB final utility

(e) Python total utility

(f) MATLAB total utility

Figure 4.5. Comparison of the network utilities after the first simulation iteration (a)-(b) and final simulation iteration (c)-(d). Plots (e)-(f) show the overall network utility for the simulation.

4.2.2 Performance Evaluation Results

Table 4.1 lists the manufacturer's specifications for the host used to benchmark our performance evaluations.

General	
Model	MacBook Pro (14-inch, 2021)
Processor	Apple M1 Pro chip
Operating System	macOS Monterey Version 12.6
Storage	512GB SSD
Memory	16GB unified memory
Processor Details	
CPU	8-core CPU with 6 performance cores and 2 efficiency cores
Speed	8x 3.22 GHz P-cores and 2x 2.06 GHz E-cores
Graphics Card	14-core GPU
WiFi	802.11ax Wi-Fi 6 wireless networking IEE 802.11a/b/g/n/ac/ax compatible
I/O Ports	
Display port	HDMI
USB ports	3x Thunderbolt 4 USB-C ports
Card Slot	SDXC card slot
Aux Port	3.5 mm headphone jack with advanced support for high-impedance headphones

Table 4.1. Specifications of the Apple MacBook Pro (2021) laptop used for benchmarking.

The results of our execution time assessments are presented in Tables 4.2 and 4.3. Table 4.2 compares the individual run times for five iterations of the MATLAB, Python, and Cython versions of the simulation and the overall average time. The MATLAB column contains the run times for the original MATLAB code. The Python column contains the run times for our finalized NCS simulation, which incorporates the refined class definitions discussed in Section 4.1.3. The Cython column includes the run times for our Cythonized version of the code. The MATLAB code executed the fastest, averaging 2 minutes and 10 seconds. As expected, our Python code had relatively slower execution times, averaging 33 minutes and 44 seconds. However, our Cythonized code showed significant improvement over the Python variant, averaging 14 minutes and 58 seconds. The Cythonized version incorporates

special variable declarations and is compiled into a C/C++ file that significantly improves performance (see Appendix D for a detailed discussion of Cython conversion).

Execution Times (min:sec.ms)			
	MATLAB	Python	Cython
Run 1	02:11.21	33:27.53	14:56.84
Run 2	02:11.68	34:11.46	15:01.59
Run 3	02:11.94	33:44.27	15:02.40
Run 4	02:11.31	33:43.66	14:54.13
Run 5	02:07.51	33:33.10	14:56.59
Average	02:10.73	33:44.00	14:58.31

Table 4.2. Observed NCS simulation execution times for MATLAB, Python, and Cython.

Table 4.3 provides a further breakdown of the performance comparison. Column two compares the relative execution times between Python and MATLAB, and column three between Cython and MATLAB. On average, Python executed 15.48x slower than MATLAB. However, the Cython version performed only 6.87x slower than MATLAB–thus showing that Cython improved the average performance time of our Python script by a notable 225 percent. We postulate that the MATLAB version ran significantly faster than the Python and Cython versions because its optimized matrix processing allowed for much more runtime parallelization than the nested loops of our conversions. However, additional analysis beyond the scope of this work would be required to verify this hypothesis.

Performance Comparison		
	Python v. MATLAB	Cython v. MATLAB (Cython Speedup)
Run 1	15.30x	6.84x (224%)
Run 2	15.58x	6.85x (228%)
Run 3	15.34x	6.84x (224%)
Run 4	15.41x	6.81x (226%)
Run 5	15.79x	7.03x (225%)
Average	15.48x	6.87x (225%)

Table 4.3. Performance comparison between MATLAB, Python, and Cython.

4.3 Chapter Summary

This chapter covered the multi-step iterative process of converting the MATLAB code to Python. We discussed the three types of relationships used in implementing our class hierarchy. We evaluated the system for correctness and analyzed its performance based on execution times. We also discuss Cython's improvement to the overall performance of the system. In Chapter 5, we discuss system modifications to enable the implementation of UxV nodes on Docker containers.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 5:
Docker Container Implementation and Evaluation

Chapters 3 and 4 focused on the general process for converting the MATLAB simulation to Python. This chapter follows by discussing the specific development of our Python NCS simulation code for implementation in Docker containers in order to increase fidelity. Section 5.1 discusses the methodology used to plan and structure our code redesign. We next explore the implementation of our re-design in Section 5.2. Finally, in Section 5.3, we present the performance evaluation of our implementation.

5.1 Methodology

The first step in developing our redesign was to tabulate the functionality of the NCS implemented in Chapter 4 to determine which things we could prune and which were essential for our implementation.

The NCS simulation is executed in eleven steps. These steps correlate with equally spaced points along the NSW team's predetermined path to a target location. This implementation is summarized in Algorithm 1. For each step, s_c, the program determines the optimum placement of each node to maximize the system's total utility, J. While the implementation described in Chapter 4 utilized both the centralized and DS methods, this chapter focuses solely on the DS method because the centralized method's reliance on a single controller for the placement of nodes is not representative of a distributed simulation that we desired to model.

For each step in the NSW team's path, the determination of the node placement is iteratively updated to maximize total network utility. The total network utility, J^k, following iteration k, is calculated as the sum of the individual utilization contributions:

$$J^k = \sum_{i=1}^{n} j_i^k \tag{5.1}$$

where j_i^k is the individual contribution of node i to the total utility. For each update from k to $k+1$, a single node, v^{k+1}, is repositioned based on the following equations:

$$\Delta j_{max}^{k+1} = \max_i(\Delta j_i^{k+1}) \tag{5.2}$$

$$v^{k+1} = \underset{i}{argmax}(\Delta j_i^{k+1}) \tag{5.3}$$

where Δj_{max}^{k+1} is the maximum increase possible by a movement of any single node and Δj_i^{k+1} is the maximum that each node, i, can increase its contribution by moving unilaterally. A more detailed description of the DS method used in this simulation can be found in [4].

Algorithm 1: Centralized DS Simulation

```
1  n: number of UxV nodes under control
2  hasMoves: whether new move exists to increase overall utility J
3  s: total number of NSW path snapshots
4  s_c: step count to cover the entire NSW path
5  for s_c = 1 to S do
6      hasMoves = true
7      while hasMoves do
8          for i = 1 to n do
9              compute (Δj_i)
10         determine max(Δj_i) and v
11         if max(Δj_i) <= 0 then
12             hasMoves = false
13         else
14             reposition node v to effect max(Δj_i)
```

As specified in Algorithm 1, the DS method is not truly "distributed" in its execution. A more accurate characterization would be the "centralized simulation" of the DS method. This is because the simulation for the DS method runs as a single process where each node "knows" about all the others. Our goal was to create a more realistic simulation that distributed the computational load among all the nodes by running each node in an isolated Docker container. For nodes to operate independently, we needed to develop an algorithm where the nodes communicated via messages. Thus, for this implementation to

work seamlessly, we needed a reliable message exchange system. The Message Queuing Telemetry Transport (MQTT) protocol was a good fit.

MQTT is a "client server publish/subscribe messaging transport protocol [that] is lightweight," [40] relatively easy to implement and supports reliable message broadcast. We used the Eclipse Mosquitto MQTT v5/v3.1.1 broker and the associated application programming interface (API) for this thesis. Figure 5.1 depicts our design for achieving reliable communications between nodes. A node communicates (i.e., *publishes*) its message to a specific *topic* managed by the MQTT broker, and the broker forwards the message to all nodes that have *subscribed* to that topic. This approach allows any connected node to communicate reliably with all other nodes.

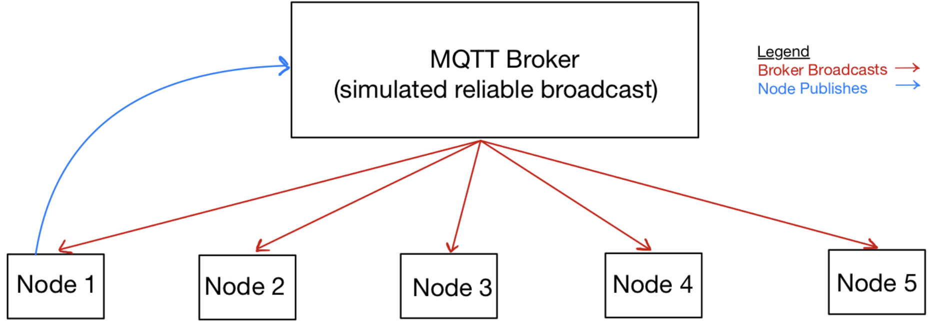


Figure 5.1. Reliable communications via MQTT broadcast.

To incorporate MQTT communications into our simulation, we utilized the following high-level conceptualization for an individual node:

1. calculate Δj_i
2. publish Δj_i to MQTT broker
3. **while** number of messages received is < 5
   (a) receive Δj_i broadcast message
   (b) store $max(\Delta j_i)$
4. **if** node with $max(\Delta j_i)$ **then** move node
   else update location of node with $max(\Delta j_i)$
5. repeat steps 1-4 until simulation is complete

A detailed pseudocode specification of this concept as implemented is provided in Algorithm 2.

5.2 Design Implementation

This section details the two-phased process we used to achieve our desired end state of a truly distributed simulation of the DS method. We first present the process of redesigning the Python code to be more modular, utilizing MQTT for broadcast communications. We then detail how we furthered our design to house the execution of each UxV node in an individual Docker container.

5.2.1 Phase I: Modular DS Simulation with MQTT

In this step, we developed a message-driven approach. Since the program flow execution depends on the receipt of all messages (i.e., all nodes have published their Δj_i values), it seemed reasonable to implement the program flow control logic at the point in the code where each node waited to receive the messages from the other nodes. That is, we needed to get the preexisting functionality into the `on_connect()` and `on_message()` functions. To do this, we needed to reformulate the loops in Algorithm 1 into modular, reusable functions. Our analysis of each loop necessitated functions that initialized loop parameters before executing the body of the loops. Algorithm 2 provides an outline of this message-driven approach.

We used the `on_connect()` and `on_message()` functions from the MQTT API to implement our flow control mechanism. `on_connect()` is the callback function that executes when a client requests a connection to the MQTT broker and receives a positive acknowledgment, while `on_message()` is the callback function that executes when a client receives a message from the broker [41]. We used a looping mechanism within the `on_message()` function to pause the program execution until all messages were received.

Upon establishing a connection to the broker via the `on_connect()` function, a new node completes function initialization and then calculates and publishes its initial Δj_i value. The `on_message()` function then takes over and manages flow control. Whenever a new message is received, the function extracts the node id, utility value, and node coordinates

Algorithm 2: Event-Based DS Simulation

```
1  n: number of UxV nodes under control
2  m: message count, initialized to 0
3  s: total number of NSW path snapshots
4  s_c: step counter to cover the entire NSW path, initialized to 0
5  if on_connect() then
6      initialize global parameters
7      initialize step parameters
8      initialize calculate_utility_J() function parameters
9      calculate and publish initial utility Δj_i
10 if on_message() then
11     extract node ID i and utility Δj_i from message
12     increment m by 1
13     if m >= n then
14         reset m to 0
15         if max(Δj_i) <= 0 then
16             if s_c ≥ s then
17                 return (Simulation Complete)
18             else
19                 increment s_c by 1
20         reposition node i with max(Δj_i)
21         reinitialize step parameters
22         reinitialize calculate_utility_J() function parameters
23         calculate and publish Δj_i
```

and increments a message counter. It compares the received utility value to the current known maximum. For simplicity, the function only keeps track of these three values from the message in order to calculate the largest Δj_i value received during each *cycle*. A cycle is defined by the receipt of n messages, one from each node. Upon receiving the nth message, the node with the maximum utility repositions to the proposed coordinates. The rest of the nodes update their reference to the node that repositioned with its new coordinates. All nodes then calculate and publish their updated utility values.

Multiple cycles are completed for each step until all nodes have achieved the placement

that maximizes the total utility (i.e., until no individual movement is possible without decreasing the total utility). This is identified by the $max(\Delta j_i)$ value being less than or equal to zero, indicating that the total utility has reached a local maximum. This is also the trigger condition to advance to the next step. The simulation is complete once there are no more steps to take (i.e., the NSW team has reached the objective).

An unintended consequence of the message-driven approach was that reinitializing the step parameters after the conclusion of each step also reset the node coordinates to their locations at the start of the simulation. We needed to store the position of the nodes at the end of each step so that they maintained their locations from the end of the last step into the next one.

Reliable Broadcast Configuration

When a new node connects to the MQTT broker, it provides its ID, subscribes to a topic (or topics), and sets a message retention flag. The broker identifies the node (i.e., client) by a `client_id` field. The message topic determines what messages the broker will forward to the client. The retain message flag instructs the broker to store the last message received. Initially, we erroneously assigned all our nodes the same ID. This caused the connected node to be disconnected from the broker with each subsequent node's connection since only one node can be connected with any unique `client_id`. MQTT automatically attempts to reestablish dropped connections, resulting in an infinite cycle of nodes kicking each other off the broker. Renaming the nodes with unique `client_ids` was a simple resolution.

Another configuration modification dealt with how the broker handled published messages. By design, when the message retention flag is set, the broker retains only the last message published to a topic [42]. By structuring our nodes to all publish to the same topic, the nodes were overwriting each other's messages. This would not be a problem if all of the nodes were connected to the broker simultaneously and only one node transmitted at a time (as depicted in Figure 5.1). It does become a problem if all the nodes connect to the broker sequentially (as was the case in our implementation). For example, consider the following sequence of events.

1. Node 1 connects to the broker, subscribes to the `NCS/Node` topic, and publishes its message to that topic
2. The broker broadcasts node 1's message, which is received by node 1

3. Node 2 connects to the broker, subscribes to the `NCS/Node` topic, and publishes its message to that same topic, overwriting the message published by node 1
4. The broker broadcasts node 2's message, which is received by nodes 1 and 2

In this example, node 2 does not receive the message that node 1 published in step one because it overwrote that message upon publishing its own message in step three. The remaining nodes experience issues similar to node 2 upon connecting and publishing to the broker.

To overcome this issue, we needed to retain the last message published by each node so that any new node that established a connection to the broker would immediately receive the retained messages. We accomplished this by having each node subscribe to a topic heading but publish to its own subtopic. To illustrate, assume nodes 1 and 2 published to the `NCS/Node 1` and `NCS/Node 2` topics, respectively, and that the retain flags are set. If node 3 later connected to the broker and subscribed to `NCS/#`, it would receive both messages from the `NCS/Node 1` topic and also from `NCS/Node 2` topic. MQTT treats "`#`" as a multi-level wildcard character that, in this case, allows nodes to receive messages from all subtopics of the top-level `NCS` topic.

Phase I Final Product

Phase I efforts resulted in a message-based, modular version of the Python DS implementation. This version consisted of 17 `.py` files, three `.mat` files, and 1,520 lines of code. To run the simulation, we configured the code to accept a node name as an input argument string. This way, the same file could be utilized multiple times–once for each node. For example, to run the program for `USV1`, the user would enter the following command in a terminal or shell.

```
python3 d_node_main.py USV1
```

The next and last step was to implement the redesigned code in Docker containers to create a fully distributed NCS simulation.

5.2.2 Phase II: Docker Implementation of the DS Simulation

Proof of Concept

Before implementing our redesigned code in Docker containers, we first developed simple Python test scripts that ran in separate containers and communicated via MQTT protocol: `ping.py` and `pong.py`. The first script performed a simple calculation and then "pinged" the other by publishing its result. The second script extracted the result from the received message, performed another simple calculation, and returned its result to the sender. Each script repeated this cycle three times and then ended the program.

While these scripts communicated perfectly in a conventional environment, additional configuration changes were required for them to run correctly in containers. Initially, we designed the scripts to connect to the broker when `on_connect()` was called with the command

```
client.connect("localhost",1883)
```

which accounted for the MQTT broker's default security settings that only allow connections from the "localhost" address (i.e., 127.0.0.1) on port 1883. We modified the Mosquitto software configuration to permit connections from anonymous hosts. Once the security settings were updated, we were able to modify the scripts to utilize the internet protocol address of the machine hosting the Mosquitto broker rather than connecting to the loopback address:

```
ipaddr = "172.20.145.225"     # where ipaddr is the IP address of the broker
client.connect(ipaddr,1883)
```

After making these changes, the scripts had no issues communicating with one another from separate Docker containers. Once we confirmed the functionality of this pattern, our last step was to implement the DS functionality of our nodes in the Docker containers.

Docker Implementation

Our first step in implementing our code in Docker containers was to make five copies of the `node_files` folder from Phase I–one for each node in our simulation. Second, we implemented the programming pattern from our `ping.py` and `pong.py` examples to enable the nodes' containers to communicate via MQTT. Lastly, we renamed each folder

to node_files_docker1 through node_files_docker5 and added a Dockerfile to each. This step ensured that we generated a unique docker image for each node. Source Code 5.1 provides an example of the Dockerfile used to create the USV2 node image.

```
FROM python:3

# copy all files from the current directory into the app directory of the container
COPY . /app

# set working directory. all following instructions now assume we are in
# this directory.
WORKDIR /app

RUN pip install --upgrade pip
RUN pip3 install matplotlib
RUN pip3 install numpy
RUN pip3 install datetime
RUN pip3 install networkx
RUN pip3 install scipy
RUN pip3 install python-math
RUN pip3 install paho-mqtt
RUN pip3 install cython

# execute the d_node_main.py file
CMD [ "python3", "./d_node_main.py" ]
```

Source Code 5.1: Dockerfile source code for USV2 image.

To build each Docker image, we executed the following command from the respective directories (the example is for USV2):

```
docker build -t node_files_docker1 .
```

This command produced a 1.25 gigabyte (GB) image containing everything USV2 needed to run in a container. We repeated this process for the remaining four nodes in four separate terminals. We executed the script for the USV2 node by entering the command

```
docker run -t -i node_files_docker1
```

in the same terminal (use of "docker1" for USV2 intentionally duplicates the numbering of the original source [4]). We executed the same command for the remaining four nodes (i.e., node_files_docker2 through node_files_docker5) in their respective terminals. Each node then ran its simulation within its containerized environment, communicating with the other nodes through the MQTT broker until the simulation terminated. The final

implementation of this system consisted of five 1.25 GB Docker images, each containing a folder with the 17 `.py` files, the three `.mat` files, and one Dockerfile.

5.3 Evaluation of Design Implementations

This section presents a validation and performance evaluation of our initial modular DS implementation and final Docker-compatible version.

Correctness Validation

We used utility values (i.e., J-values) and node coordinates to validate the correctness of our implementations (as shown in Figure 5.2). The final total network utility obtained by both versions of our implementation was 2.453, which matched the results presented in Figure 4.4 plots (e) and (f). This value could only be obtained if all of the previous steps in the simulation were executed correctly, thus showing that our code performed precisely as the original MATLAB version.

```
 < rcvd msg: id_str=USV2, nid=2, iter_cnt=4, step=10, J_val=2.452948013667349, coords=[424.0, 197.0]
 < rcvd msg: id_str=UAV1, nid=4, iter_cnt=4, step=10, J_val=2.452948013667349, coords=[326.0, 144.0]
 < rcvd msg: id_str=UAV3, nid=6, iter_cnt=4, step=10, J_val=2.452948013667349, coords=[326.0, 197.0]
 < rcvd msg: id_str=UAV2, nid=5, iter_cnt=4, step=10, J_val=2.452948013667349, coords=[280.0, 175.0]
 < rcvd msg: id_str=USV1, nid=3, iter_cnt=4, step=10, J_val=2.452948013667349, coords=[432.0, 167.0]
<max node> nid= 2 j= 2.452948013667349 loc= [424.0, 197.0]
```

Figure 5.2. Final output of Docker-compatible DS simulation used for correctness validation.

Performance Evaluation Results

In this section, we compare execution times of the Modular DS simulation discussed in Section 5.2.1 to that of the Docker-compatible DS simulation in 5.2.2. Table 5.1 shows the results of execution times and relative performance for five runs.

Execution Times (min:sec.ms)			Comparison
	Modular DS	Docker DS	Docker Speedup
Run 1	11:18.27	05:18.59	213%
Run 2	11:15.69	05:12.95	216%
Run 3	11:19.66	05:15.14	216%
Run 4	11:25.06	05:12.11	219%
Run 5	11:17.98	05:12.05	217%
Average	11:19.33	05:14.17	216%

Table 5.1. Performance comparison of the modular and Docker container simulation of the DS methods.

Surprisingly, the Docker implementation ran, on average, 2.16 times faster than the Python implementation. We hypothesize that the host used for benchmarking may support multi-threading differently for Docker images than for the modular implementation. A full investigation of this phenomenon is left for future work.

5.4 Chapter Summary

This chapter described the methodology for revising our Python NCS for deployment on Docker containers. We outlined the steps taken to convert the implementation from a centralized version of the DS method to a distributed one using MQTT as a reliable broadcast medium between Docker containers. Lastly, we detailed the process of creating Docker images from Python scripts to implement our UxV nodes into Docker containers. In Chapter 6, we summarize our work and provide recommendations for future works.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 6:
Conclusion

In this chapter, we first summarize the major findings of our work and present our conclusions and lessons learned. We conclude by suggesting areas for further exploration as an extension of this thesis.

6.1 Summary and Conclusions

At the onset of this thesis, we sought to answer four research questions. Through our research and simulation experimentation, we reached several conclusions that addressed them.

The NCS simulation can be implemented in an OOP language (in this case, Python) that can be more readily deployed in real-world vehicles. Although there was a considerable performance decline in the Python implementation relative to MATLAB, the benefits potentially outweigh the cost in performance. Furthermore, this work demonstrated that performance improvement tools like Cython can be utilized to significantly speed up Python code execution.

Currently, no single generalizable methodology exists for converting MATLAB modules into an OOP implementation. Tools available in the industry can aid in this effort, but they are only partially effective and better suited for small code snippets than for large projects. Open-source Python converters are not suitable for programs of considerable complexity. Unfortunately, the most reliable way to achieve a viable conversion is to do it manually.

The structures for implementing OOP provide the flexibility to incorporate additional functionality. For example, adding new node placement optimization algorithms can be achieved by designing and implementing new subclasses of the "Group Behavior" class. In addition, the internal implementations of objects can be modified to meet new specifications without affecting the objects' external interfaces. This provides the ability to update and extend functionality without requiring a complete overhaul of existing program structures.

Docker containers offer a lightweight solution for modeling a more realistic version of UxV

simulations. As independent discrete virtual machines, they introduce features into the simulation that better represent actual interactions between real-life vehicles. We consider the Docker DS implementation a step forward in moving the NCS software from a lab environment to actual physical devices.

6.2 Lessons Learned

Challenges encountered at the infancy stages of this project and the lessons learned from them propagated throughout the rest of the thesis and can be summarized in two categories:

Software engineering: Developing a plan of attack and breaking down complex problems into smaller incremental tasks was instrumental in managing code conversion and allowed us to build a few lines of code at a time and test each incremental step.

Technical lessons: Simplicity in implementation is preferable. The MATLAB implementation of the NCS simulation had multiple nested loops. In our first iteration of the conversion, we maintained the iterative structures in Python. This implementation worked seamlessly in a centralized control simulation but did not translate well to a truly decentralized control model. In particular, this implementation contained race conditions that led to buggy and unreliable code. Our final, better implementation relied on modular functions in a distributed simulation environment and proved much more reliable as a result.

6.3 Future Work

We have demonstrated the concept of Docker containers simulating the NCS nodes. A natural progression of this work would be to explore an implementation of the images on actual physical models. In our simulation, the nodes were pre-programmed with scenario-specific information ahead of time. The next iteration of this thesis would have the nodes ingest incoming data communication and other sensory data as inputs from their natural environment and perform the necessary calculations in real-time to achieve optimal positioning.

Our implementation utilized MQTT to simulate reliable communication between nodes. This potentially introduces a single point of failure since the broker acts as the central link in the communication infrastructure. Alternative communication models are a topic worthy of further consideration. A mesh architecture, for example, involves each node

communicating directly with every other node, but it might be operationally infeasible to establish or maintain. A relay system where each node communicates with its immediate neighbors while forwarding messages on behalf of other nodes may be a more realistic model.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A: SMOP Code

A.1 MATLAB Test Code

Source Code A.1 is a MATLAB script adapted from [43] that establishes a client-server session between a single host and a server.

```
%Find Host Name and Address
[~,hostname] = system('hostname');
hostname = string(strtrim(hostname));
address = resolvehost(hostname,"address");

%%%%%% Create Server %%%%%%
server = tcpserver(address,4999,"ConnectionChangedFcn",@connectionFcn);

%Read Binary Data Using Byte Callback Mode
configureCallback(server,"byte",7688,@readDataFcn);

%%%%%% Create Client Session %%%%%%%
client = tcpclient(server.ServerAddress,server.ServerPort,"Timeout",5);
pause(1);

% Read Data and Display
rawData = read(client,961,"double");
reshapedData = reshape(rawData,31,31);
surf(reshapedData);

%Write Data to the Server
write(client,rawData,"double");

%Clear the Client
clear client

%Callback Functions

%This function calls write data to the connected TCP/IP client.
function connectionFcn(src, ~)
if src.Connected
    disp("Client connection accepted by server.")
    data = membrane(1);
    write(src,data(:),"double");
end
end
```

```
%This function calls read to read BytesAvailableFcnCount number of bytes of data.
function readDataFcn(src, ~)
disp("Data was received from the client.")
src.UserData = read(src,src.BytesAvailableFcnCount/8,"double");
reshapedServerData = reshape(src.UserData,31,31);
surf(reshapedServerData);
end
```

Source Code A.1: MATLAB test code for automatic conversion. Adapted from: [43].

A.2 SMOP Functionality Testing

Source Code A.2 illustrates the functionality of SMOP in converting the MATLAB code in Source Code A.1 into Python.

```
# Generated with SMOP  0.41
from libsmop import *
# ClientServerSession.m

    #Find Host Name and Address
    __,hostname=system('hostname',nargout=2)
# ClientServerSession.m:2
    hostname=string(strtrim(hostname))
# ClientServerSession.m:3
    address=resolvehost(hostname,'address')
# ClientServerSession.m:4
    ###### Create Server ######
    server=tcpserver(address,4999,'ConnectionChangedFcn',connectionFcn)
# ClientServerSession.m:8
    #Read Binary Data Using Byte Callback Mode
    configureCallback(server,'byte',7688,readDataFcn)
    ###### Create Client Session #######
    client=tcpclient(server.ServerAddress,server.ServerPort,'Timeout',5)
# ClientServerSession.m:14
    pause(1)
    # Read Data and Display
    rawData=read(client,961,'double')
# ClientServerSession.m:18
    reshapedData=reshape(rawData,31,31)
# ClientServerSession.m:19
    surf(reshapedData)
    #Write Data to the Server
    write_(client,rawData,'double')
    #Clear the Client
    clear('client')
    #Callback Functions
```

```
    #This function calls write to write data to the connected TCP/IP client.

@function
def connectionFcn(src=None,__=None,*args,**kwargs):
    varargin = connectionFcn.varargin
    nargin = connectionFcn.nargin

    if src.Connected:
        disp('Client connection accepted by server.')
        data=membrane(1)
# ClientServerSession.m:34
        write_(src,ravel(data),'double')

    return

if __name__ == '__main__':
    pass

    #This function calls read to read BytesAvailableFcnCount number of bytes of data.

@function
def readDataFcn(src=None,__=None,*args,**kwargs):
    varargin = readDataFcn.varargin
    nargin = readDataFcn.nargin

    disp('Data was received from the client.')
    src.UserData = copy(read(src,src.BytesAvailableFcnCount / 8,'double'))
# ClientServerSession.m:42
    reshapedServerData=reshape(src.UserData,31,31)
# ClientServerSession.m:43
    surf(reshapedServerData)
    return

if __name__ == '__main__':
    pass
```

Source Code A.2: SMOP conversion of MATLAB code to Python. Adapted from: [43].

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B:
M2HTML Code

B.1 M2HTML Demonstration

The `functionDependanciesDemo.m` file (adapted from [37]) in Source Code B.1 below creates a list of function dependency pairs ('X', 'Y') where function X calls function Y. The produced HTML document is provided in Source Code B.2.

```
1  function functionDependenciesDemo(calls)
2  % This MATLAB code creates function dependancies dot file and renders
3  % a 2D graphical representation of these functions interdependencies.
4  % The nodes represent the functions and the directed vertices
5  % represent the function calls.
6
7  % Calls (cellstr) is an n-by-2 cell array in format {caller,callee;...}.
8  calls = { 'foo','A'; 'foo','C'; 'foo','D'; 'foo','bar'; 'D', 'E';
9            'E', 'F'; 'bar','bar'; 'A', 'bar'; 'Y', 'Z'};
10
11 % Create a GraphViz DOT diagram functions dependencies for "calls"
12 fileName = 'dependenciesDemo';
13 dotFile = [fileName '.dot'];
14 fid = fopen(dotFile, 'w');
15 fprintf(fid, 'digraph G {\n');
16 for i = 1:size(calls,1)
17     [parent,child] = calls{i,:};
18     fprintf(fid, '    "%s" -> "%s"\n', parent, child);
19 end
20 fprintf(fid, '}\n');
21 fclose(fid);
22
23 % Render image to create a 2D graphical representation
24 imageFile = [fileName '.png'];
25 % Assumes the GraphViz bin dir is on the path; if not, use full path to
26 % dot.exe
27 cmd = sprintf('dot -Tpng -Gsize="2,2" "%s" -o"%s"', dotFile, imageFile);
28 system(cmd);
29 fprintf('Image file created: %s\n', imageFile);
30
31 end
```

Source Code B.1: MATLAB test code to demonstrate M2HTML functionality. Adapted from: [37].

Below is the HTML code for the dependencies of the functions defined in Source Code B.1 above.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
                "http://www.w3.org/TR/REC-html40/loose.dtd">
<html>
<head>
  <title>Description of functionDependenciesDemo</title>
  <meta name="keywords" content="functionDependenciesDemo">
  <meta name="description" content="This MATLAB code creates function dependancies dot
    file and renders">
  <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
  <meta name="generator" content="m2html v1.5 &copy; 2003-2005 Guillaume Flandin">
  <meta name="robots" content="index, follow">
  <link type="text/css" rel="stylesheet" href="../m2html.css">
</head>
<body>
<a name="_top"></a>
<div><a href="../index.html">Home</a> &gt;  <a href="index.html">CallsDemo</a> &gt;
    functionDependenciesDemo.m</div>

<!--<table width="100%"><tr><td align="left"><a href="../index.html"><img alt="<" border="
    0" src="../left.png"> Master index</a></td>
<td align="right"><a href="index.html">Index for CallsDemo <img alt=">" border="0"
    src="../right.png"></a></td></tr></table>-->

<h1>functionDependenciesDemo
</h1>

<h2><a name="_name"></a>PURPOSE <a href="#_top"><img alt="^" border="0" src="../up.png"></
    a></h2>
<div class="box"><strong>This MATLAB code creates function dependancies dot file and
    renders</strong></div>

<h2><a name="_synopsis"></a>SYNOPSIS <a href="#_top"><img alt="^" border="0" src="../up.
    png"></a></h2>
<div class="box"><strong>function functionDependenciesDemo(calls) </strong></div>

<h2><a name="_description"></a>DESCRIPTION <a href="#_top"><img alt="^" border="0" src="
    ../up.png"></a></h2>
<div class="fragment"><pre class="comment"> This MATLAB code creates function dependancies
     dot file and renders
 a 2D graphical representation of these functions interdependencies.
 The nodes represent the functions and the directed vertices
 represent the function calls.</pre></div>

<!-- crossreference -->
<h2><a name="_cross"></a>CROSS-REFERENCE INFORMATION <a href="#_top"><img alt="^" border="
    0" src="../up.png"></a></h2>
```

```
37 This function calls:
38 <ul style="list-style-image:url(../matlabicon.gif)">
39 </ul>
40 This function is called by:
41 <ul style="list-style-image:url(../matlabicon.gif)">
42 </ul>
43 <!-- crossreference -->
44
45
46
47 <h2><a name="_source"></a>SOURCE CODE <a href="#_top"><img alt="^" border="0" src="../up.png"></a></h2>
48 <div class="fragment"><pre>0001 <a name="_sub0" href="#_subfunctions" class="code">function functionDependenciesDemo(calls)</a>
49 0002 <span class="comment">% This MATLAB code creates function dependancies dot file and renders</span>
50 0003 <span class="comment">% a 2D graphical representation of these functions interdependencies.</span>
51 0004 <span class="comment">% The nodes represent the functions and the directed vertices</span>
52 0005 <span class="comment">% represent the function calls.</span>
53 0006
54 0007 <span class="comment">% Calls (cellstr) is an n-by-2 cell array in format {caller, callee;...}.</span>
55 0008 calls = { <span class="string">'foo'</span>,<span class="string">'A'</span>; <span class="string">'foo'</span>,<span class="string">'C'</span>; <span class="string">'foo'</span>,<span class="string">'D'</span>; <span class="string">'foo'</span>,<span class="string">'bar'</span>; <span class="string">'D'</span>, <span class="string">'E'</span>;
56 0009           <span class="string">'E'</span>, <span class="string">'F'</span>; <span class="string">'bar'</span>,<span class="string">'bar'</span>; <span class="string">'A'</span>, <span class="string">'bar'</span>; <span class="string">'Y'</span>, <span class="string">'Z'</span>};
57 0010
58 0011 <span class="comment">% Create a GraphViz DOT diagram functions dependencies for &quot;calls&quot;</span>
59 0012 fileName = <span class="string">'dependenciesDemo'</span>;
60 0013 dotFile = [fileName <span class="string">'.dot'</span>];
61 0014 fid = fopen(dotFile, <span class="string">'w'</span>);
62 0015 fprintf(fid, <span class="string">'digraph G {\n'</span>);
63 0016 <span class="keyword">for</span> i = 1:size(calls,1)
64 0017     [parent,child] = calls{i,:};
65 0018     fprintf(fid, <span class="string">'   &quot;%s&quot; -&gt; &quot;%s&quot;\n'</span>, parent, child);
66 0019 <span class="keyword">end</span>
67 0020 fprintf(fid, <span class="string">'}\n'</span>);
68 0021 fclose(fid);
69 0022
70 0023 <span class="comment">% Render image to create a 2D graphical representation</span>
71 0024 imageFile = [fileName <span class="string">'.png'</span>];
```

```
72 0025 <span class="comment">% Assumes the GraphViz bin dir is on the path; if not, use full
     path to</span>
73 0026 <span class="comment">% dot.exe</span>
74 0027 cmd = sprintf(<span class="string">'dot -Tpng -Gsize=&quot;2,2&quot; &quot;%s&quot; -
    o&quot;%s&quot;'</span>, dotFile, imageFile);
75 0028 system(cmd);
76 0029 fprintf(<span class="string">'Image file created: %s\n'</span>, imageFile);
77 0030
78 0031 <span class="keyword">end</span></pre></div>
79 <hr><address>Generated on Thu 12-May-2022 01:31:38 by <strong><a href="http://www.artefact
    .tk/software/matlab/m2html/" title="Matlab Documentation in HTML">m2html</a></strong>
    &copy; 2005</address>
80 </body>
81 </html>
```

Source Code B.2: M2HTML output of function dependencies. Adapted from: [37].

APPENDIX C:
Python Code

C.1 Source Code Repository

A link to the GitLab repository, which includes all of the source code for our Python NCS simulation, can be found at:

https://gitlab.nps.edu/andrew.faulk/thesis-matlab-to-python#

C.2 Sample `Employees` Class Definition

Usually, the first argument to any method of a class is the `self` argument, which refers to the object itself and the particular instance of the object being operated on. To use the `Employee` class, you can instantiate the class like this:

```
emp = Employee()
```

This will instantiate an object instance of the class. Class methods can now be called in this format:

```
emp.Classmethod()
```

Note that there is no need to supply the `self` parameter when calling class methods–Python runtime automatically takes care of that.

```
class Employees:   # PARENT class
    def __init__(self, id, name, job_title):
        self.id = id
        self.name = name
        self.job_title = job_title

    def get_name(self):
        return self.name
    def get_title(self):
        return self.job_title

    def set_title(self, job_title):
```

```
        self.job_title = job_title

class SalaryEmployees(Employees):   # child to Employees class
    def __init__(self, id, name, job_title, weekly_salary):
        super().__init__(id, name, job_title)
        self.weekly_salary = weekly_salary

    def get_salary(self):
        return self.weekly_salary

    def calc_pay(self, num_weeks):
        return (self.weekly_salary * num_weeks)

class HourlyEmployees(Employees):   # child to Employees class
    def __init__(self, id, name, job_title, hourly_wage):
        super().__init__(id, name, job_title)
        self.hourly_wage = hourly_wage

    def get_wages(self):
        return self.hourly_wage

    def calc_pay(self, hrs_worked):
        return (self.hourly_wage * hrs_worked)

class CommissionedEmployees(SalaryEmployees):
    def __init__(self, id, name, job_title, weekly_salary,
                 sales_commission):
        super().__init__(id, name, job_title, weekly_salary)
        self.commission_rate = sales_commission/100

    def calc_pay(self, num_weeks, total_sales):
        fixed = super().calc_pay(num_weeks)
        return (fixed + total_sales * self.commission_rate)

# ==================== Testing Code ==================================
n_empl = Employees(1, "Emperor", "worker")    # new base employee
print('Name:', n_empl.get_name(), '\tTitle:\t', n_empl.get_title())

s_empl = SalaryEmployees(2, "Sally", "manager", 1200)
print('Name:', s_empl.get_name(), '\tSalary:\t$', s_empl.calc_pay(2))

w_empl = HourlyEmployees(3, "Walle", "secretary", 17.50)
print('Name:', w_empl.get_name(),'\tWages:\t$', w_empl.calc_pay(40))

c_empl = CommissionedEmployees(4, "Carla", "hr", 900, 10)
print('Name:', c_empl.get_name(), '\tComissioned Salary: $',
      c_empl.calc_pay(2, 8000))

# ==================== Output ========================================
'''
```

```
Name: Emily   Title:   worker
Name: Sally   Salary: $ 2400
Name: Walle   Wages:  $ 700.0
Name: Carla   Comissioned Salary: $ 2600.0
'''
```

Source Code C.1: *Employees* class definition. Adapted from: [17].

C.3 Detailed Iterative Process Flow Diagram

The flow chart below is a more detailed version of the flow diagram in Section 4.1.

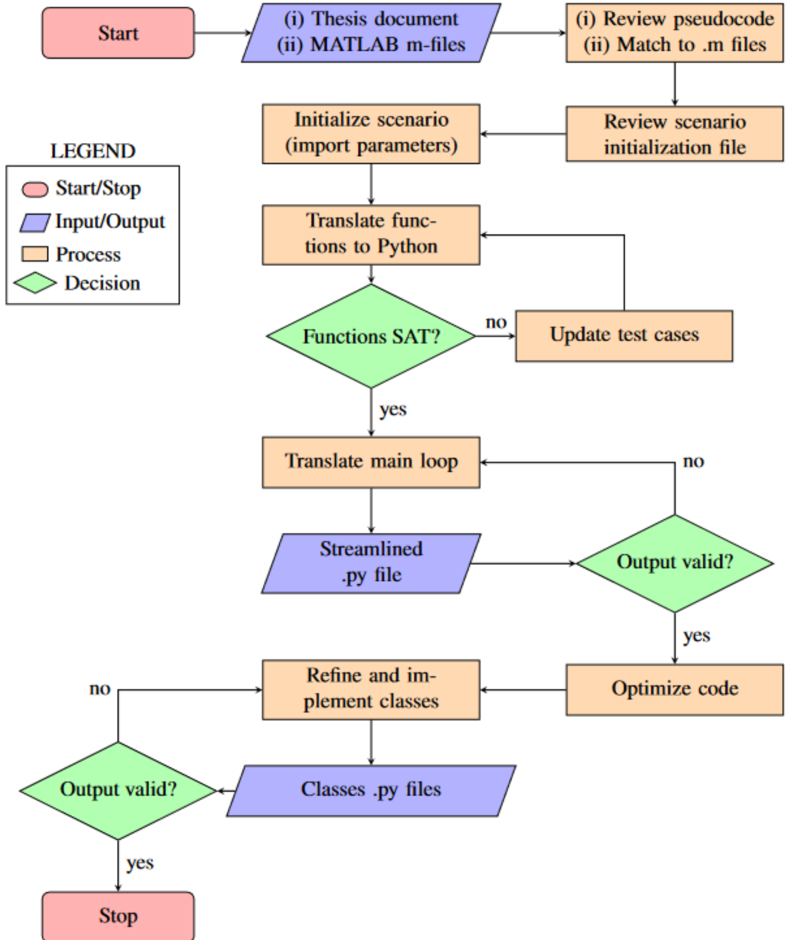


Figure C.1. Granular flow chart detailing the iterative process used for converting MATLAB code to Python.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX D:
Cython Code

D.1 Cython Conversion Test

To illustrate Cython functionality, we use Source Code D.1 through D.3 below, slightly adapted from [38]. Source Code D.1 is a simple Python program file we named `primes.py` that takes a number and determines if it is prime.

```
def primes(nb_primes):
    p = []
    n = 2
    while len(p) < nb_primes:
        # Is n prime?
        for i in p:
            if n % i == 0:
                break

        # If no break occurred in the loop
        else:
            p.append(n)
        n += 1
    return p
```

Source Code D.1: Python test script. Source: [38].

Next, we Cythonize this code to show the improvement in performance. This involves declaring variables using special Cython notation, such as `i: cython.int` (equivalent to `int i` in C++) seen on line 5 of Source Code D.2. We name this file `primes_cythonized.py`. The `import cython` statement on line 1 enables Cython functionality.

```
import cython

def primes(nb_primes: cython.int):
    i: cython.int
    p: cython.int[1000]     # An array of size 1000. any larger and it will break!

    if nb_primes > 1000:
        nb_primes = 1000

    if not cython.compiled: # Only if regular Python is running
```

```
11        p = [0] * 1000      # Make p work almost like a C array. Zeros out array
12
13    len_p: cython.int = 0   # The current number of elements in p.
14    n: cython.int = 2
15    while len_p < nb_primes:
16        # Is n prime?
17        for i in p[:len_p]:
18            if n % i == 0:
19                break
20
21        # If no break occured in the loop, we have a prime.
22        else:
23            p[len_p] = n
24            len_p += 1
25        n += 1
26
27    # Copy the result into a Pyhton list:
28    result_as_list = [prime for prime in p[:len_p]]
29    return result_as_list
```

Source Code D.2: Cythonized Python test script. Adapted from: [38].

Lastly, we create a setup file, `setup.py`, to compile the target files `primes_cythonized.py` and `primes_compiled.py`. The latter is a compiled version of the `primes.py` file in Source Code D.2. The file `primes_compiled.py` is used as a benchmark to compare execution times against `primes.py` and `primes_cythonized.py` in Section D.2.

```
1  from setuptools import setup
2  from Cython.Build import cythonize
3
4  setup(
5      ext_modules=cythonize(
6          ['primes_compiled.py',              # Python code that is compiled
7           'primes_cythonized.py'])           # Python code that is compiled and Cythonized
8  )
9
10 # to run, do:
11 #    python3 setup.py build_ext --inplace
```

Source Code D.3: Python script to compile target files. Adapted from: [38].

To run `setup.py`, the user must enter the following command in a terminal or shell.

```
python3 setup.py build_ext --inplace
```

The `cythonize` command on line 5 of Source Code D.3 "takes a `.py` or `.pyx` file and compiles it into a C/C++ file. It then compiles the C/C++ file into an extension module which

is directly importable from Python" [44]. This means that two files are generated from each target file. For example, `primes_compiled.py` generates `primes_compiled.c` and an associated binary file. The setup file must be re-run whenever a change is made to any of the target files.

D.2 Conversion Results

Source Code D.4 is a test script that was modified from [45] to evaluate the performance of the code in Section D.1 above. To run this script, execute the following command in a terminal or shell.

```
python3 time_test.py
```

```
import timeit

primes = timeit.timeit('primes.primes(2000)', setup='import primes', number = 3)
primes_compiled = timeit.timeit('primes_compiled.primes(2000)', setup='import
    primes_compiled', number = 3)
primes_cythonized = timeit.timeit('primes_cythonized.primes(2000)', setup='import
    primes_cythonized', number = 3)

print("primes=", primes)
print("primes_compiled=", primes_compiled)
print("primes_cythonized=", primes_cythonized)
print('primes_compiled  is {}x faster than primes'.format(primes/primes_compiled))
print('primes_cythonized is {}x faster than primes_compiled'.format(primes_compiled/
    primes_cythonized))
print('primes_cythonized is {}x faster than primes'.format(primes/primes_cythonized))

# to test, do:
#   python3 time_test.py
```

Source Code D.4: Python script to evaluate performance. Adapted from: [45].

Below is a snapshot of the output produced by the above Python script.

```
********************* BEGIN EXAMPLE OUTPUT ***************************

    andrewfaulk@Andrews-MBP Source Code % python3 time_test.py
    primes= 0.15482049999991432
    primes_compiled= 0.0749014589964645
    primes_cythonized= 0.0015983749908627942
    primes_compiled  is 2.066989109080267x faster than primes
    primes_cythonized is 46.8610053489595x faster than primes_compiled
    primes_cythonized is 96.86118769685145x faster than primes
```

```
******************** END EXAMPLE OUTPUT ******************************
```

Tables D.1 and D.2 show the results of five iterations of running the test script. On average, `primes_compiled.py` ran approximately 1.96x faster than `primes.py`. This is a significant improvement accomplished by simply compiling the file. However, the performance improvements are even more impressive when benchmarking against `primes_cythonized.py`. On average, `primes_cythonized.py` performed 41.06x faster than `primes_compiled.py` and an impressive 82.77x faster than `primes.py`. This simple illustration shows the powerful performance-enhancing potential of Cython.

Execution Times (in seconds)			
	primes.py	primes_compiled.py	primes_cythonized.py
Run 1	0.160	0.102	0.006
Run 2	0.155	0.075	0.002
Run 3	0.154	0.075	0.002
Run 4	0.153	0.075	0.002
Run 5	0.155	0.075	0.002
Average	0.155	0.080	0.002

Table D.1. Comparison of execution times.

Performance Comparison			
	primes_comp v. primes	primes_cyth v. primes_comp	primes_cyth v. primes
Run 1	1.57x	16.74x	26.28x
Run 2	2.06x	47.25x	97.36x
Run 3	2.05x	46.87x	96.31x
Run 4	2.04x	47.56x	97.06x
Run 5	2.07x	46.86x	96.86x
Average	1.96x	41.06x	82.77x

Table D.2. Python test code performance comparison.

List of References

[1] S. I. Gordon and B. Guilfoos, *Introduction to Modeling and Simulation with MATLAB and Python*. Boca Raton, FL, USA: Chapman and Hall/CRC, 2017 [Online]. Available: https://doi.org/10.1201/9781315151748

[2] Apple Inc., "iPhone 13 pro and 13 pro max - technical specifications," Oct. 13, 2022 [Online]. Available: https://www.apple.com/am/iphone-13-pro/specs/

[3] J. Ortiz and A. Cruz, *Ad Hoc Networks*. London, United Kingdom: IntechOpen, 2017 [Online]. Available: http://doi.org/10.5772/62746

[4] B. Lowry, "Distributed submodular optimization for a UxV networked control system," M.S. thesis, Dept. of Mech. and Aero. Eng., NPS, Monterey, CA, USA, 2020 [Online]. Available: https://calhoun.nps.edu/handle/10945/64926

[5] X.-M. Zhang, Q.-L. Han, and X. Yu, "Survey on recent advances in networked control systems," *IEEE transactions on industrial informatics*, vol. 12, no. 5, pp. 1740–1752, Oct. 2016 [Online]. Available: https://doi.org/10.1109/TII.2015.2506545

[6] N. Wachlin, "Robust time-varying formation control with adaptive submodularity," M.S. thesis, Dept. of Mech. and Aero. Eng., NPS, Monterey, CA, USA, 2018 [Online]. Available: http://hdl.handle.net/10945/59612

[7] D. T. Valentine, *Essential MATLAB for Engineers and Scientists*, 3rd ed. Amsterdam, The Netherlands: Butterworth Heinemann, 2007.

[8] C. Moler, "A brief history of MATLAB," 2018 [Online]. Available: https://www.mathworks.com/company/newsletters/articles/a-brief-history-of-matlab.html

[9] B. Weber, A. Santos, G. Hjelle, and J. Jablonski, "MATLAB vs Python: Why and how to make the switch – Real Python," Accessed May 13, 2022 [Online]. Available: https://realpython.com/matlab-vs-python/

[10] H. Huang, *50 Basic Examples for MATLAB*, ver. 2012.3, 2012 [Online]. Available: https://www.public.asu.edu/~hhuang38/hph_matlab_basic2013_1.pdf

[11] A. P. Black, "Object-oriented programming: Some history, and challenges for the next fifty years," *Information and Computation*, vol. 231, pp. 3–20, Oct. 2013 [Online]. Available: https://doi.org/10.1016/j.ic.2013.08.002

[12] S. Gowrishankar and A. Veena, *Introduction to Python Programming*, 1st ed. Boca Raton, FL: CRC Press, 2018. Available: https://doi.org/10.1201/9781351013239

[13] G. Blaschek, *Object-Oriented Programming with Prototypes*, 1st ed. Berlin, Germany: Springer-Verlag, 1994.

[14] T. Budd, *An Introduction to Object-Oriented Programming*. Reading, MA, USA: Addison-Wesley Pub. Co., 1991, pp. 15–85.

[15] G. Booch, *Object Oriented Design with Applications* (Benjamin/Cummings Series in Ada and Software Engineering). Redwood City, CA, USA: Benjamin/Cummings Pub. Co., 1991, pp. 35–99.

[16] P. Wegner, "Concepts and paradigms of object-oriented programming," *SIGPLAN OOPS Mess.*, vol. 1, no. 1, pp. 7–87, Aug. 1990 [Online]. Available: https://doi-org.libproxy.nps.edu/10.1145/382192.383004

[17] Stack Overflow, "Employee classes - Python," April 4, 2017 [Online]. Available: https://stackoverflow.com/q/43160641

[18] B. Cyganek, "Delving into Object-Oriented Programming," in *Introduction to Programming with C++ for Engineers*. Chichester, UK: John Wiley Sons, Inc, 2020 [Online], pp. 227–348. Available: https://doi.org/10.1002/9781119431152

[19] Docker, "What is a container?" Accessed Feb. 24, 2022 [Online]. Available: https://www.docker.com/resources/what-container

[20] C. Pahl, A. Brogi, J. Soldani, and P. Jamshidi, "Cloud container technologies: A state-of-the-art review," *IEEE Transactions on Cloud Computing*, vol. 7, no. 3, pp. 677–692, July 2019 [Online]. Available: https://doi.org/10.1109/TCC.2017.2702586

[21] M. Hamedani. "Docker Tutorial for Beginners," Mar. 30, 2021. [YouTube video]. Available: https://www.youtube.com/watch?v=pTFZFxd4hOI

[22] Docker Docs, "Dockerfile reference." Nov. 27, 2022 [Online]. Available: https://docs.docker.com/engine/reference/builder/

[23] Docker Docs, "Docker overview." Nov. 27, 2022 [Online]. Available: https://docs.docker.com/get-started/overview/

[24] K. Shaw, "What is a virtual machine, and why are they so useful?" *Network World*, Sep. 2020 [Online]. Available: https://libproxy.nps.edu/login?url=https://www.proquest.com/trade-journals/whatis-virtual-machine-why-are-they-so-useful/docview/2446042086/se-2?accountid=12702

[25] A. Randal, "The ideal versus the real: Revisiting the history of virtual machines and containers," *ACM computing surveys*, vol. 53, no. 1, pp. 1–31, Feb. 2020 [Online]. Available: https://doi.org/10.1145/3365199

[26] B. Kavitha and P. Varalakshmi, "Performance analysis of virtual machines and docker containers," in *Data Science Analytics and Applications* (Communications in Computer and Information Science). Singapore: Springer Singapore, 2018, pp. 99–113.

[27] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2015 [Online]. Available: https://doi.org/10.1109/ISPASS.2015.7095802

[28] F. Manco, C. Lupu, F. Schmidt, J. Mendes, S. Kuenzer, S. Sati, K. Yasukata, C. Raiciu, and F. Huici, "My VM is lighter (and safer) than your container," in *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017 [Online]. Available: https://doi.org/10.1145/3132747.3132763

[29] T. Summers and M. Kamgarpour, "Performance guarantees for greedy maximization of non-submodular controllability metrics," in *2019 18th European Control Conference (ECC)*, 2019 [Online]. Available: https://doi.org/10.23919/ECC.2019.8795800

[30] S. Peled and A. Laso, "Conversion of MATLAB Bridge modules to integrated 3DSlicer modules." Accessed Nov. 28, 2022 [Online]. Available: https://projectweek.na-mic.org/PW28_2018_GranCanaria/Projects/MatlabToPython/

[31] M. Sollfrank, F. Loch, and B. Vogel-Heuser, "Exploring docker containers for time-sensitive applications in networked control systems," in *2019 IEEE 17th International Conference on Industrial Informatics (INDIN)*, 2019 [Online]. Available: https://doi.org/10.1109/INDIN41052.2019.8972165

[32] G. Imreh, "How we updated a drone while flying - and how you can too!" balenaBlog, blog, July 14, 2016 [Online]. Available: https://www.balena.io/blog//how-we-updated-a-drone-while-flying-dockercon2016/

[33] Python Pool, "5 ways to convert MATLAB to Python." Accessed Nov. 25, 2022 [Online]. Available: https://www.pythonpool.com/convert-matlab-to-python/

[34] E. Schlogl, *Quantitative Finance : An Object-Oriented Approach in C++*, 1st ed. (Chapman and Hall/CRC Financial Mathematics Series). Boca Raton, FL, USA: Chapman and Hall/CRC, 2018.

[35] "M2HTML: Documentation system for MATLAB in HTML," Accessed May 12, 2022 [Online]. Available: https://www.gllmflndn.com/software/matlab/m2html/

[36] Graphviz, "What is Graphviz?" Aug. 10, 2021 [Online]. Available: https://graphviz.org/

[37] A. Janke, “Answer to "automatically generating a diagram of function calls in MATLAB",” Apr. 1, 2011 [Online]. Available: https://stackoverflow.com/a/5519365

[38] Cython, “Basic tutorial - Cython 3.0.0a11 documentation.” Accessed Oct. 12, 2022 [Online]. Available: https://cython.readthedocs.io/en/latest/src/tutorial/cython_tutorial.html

[39] I. Rodriguez, “Inheritance and composition: A Python OOP guide - Real Python,” Accessed Sep. 22, 2022 [Online]. Available: https://realpython.com/inheritance-composition-python/

[40] T. H. Team, “Introducing the MQTT protocol - MQTT essentials: Part 1,” Accessed Nov. 25, 2022 [Online]. Available: https://www.hivemq.com/blog/mqtt-essentials-part-1-introducing-mqtt/

[41] I. Craggs, “Eclipse paho - The Eclipse Foundation,” Accessed Nov. 25, 2022 [Online]. Available: https://www.eclipse.org/paho/index.php?page=clients/python/docs/index.php

[42] T. H. Team, “MQTT topics, wildcards, & best practices - MQTT essentials: Part 5,” Accessed Nov. 25, 2022 [Online]. Available: https://www.hivemq.com/blog/mqtt-essentials-part-5-mqtt-topics-best-practices/

[43] MathWorks, “Communicate between a TCP/IP client and server in MATLAB - MATLAB & Simulink.” Accessed May 5, 2022 [Online]. Available: https://www.mathworks.com/help/instrument/communicate-between-a-tcpip-client-and-server-in-matlab.html

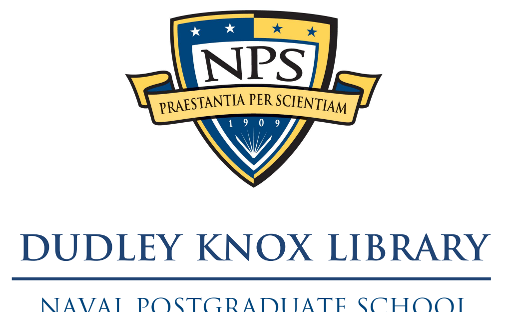
[44] Cython, “Source files and compilation - Cython 3.0.0a11 documentation.” Accessed Oct. 30, 2022 [Online]. Available: https://cython.readthedocs.io/en/latest/src/userguide/source_files_and_compilation.html

[45] Python Programming, “Optimizing with Cython introduction - Cython tutorial.” Accessed Oct. 30, 2022 [Online]. Available: https://pythonprogramming.net/introduction-and-basics-cython-tutorial/

Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California



WHERE SCIENCE MEETS THE ART OF WARFARE