

Implementing the Backseat Driver Architecture

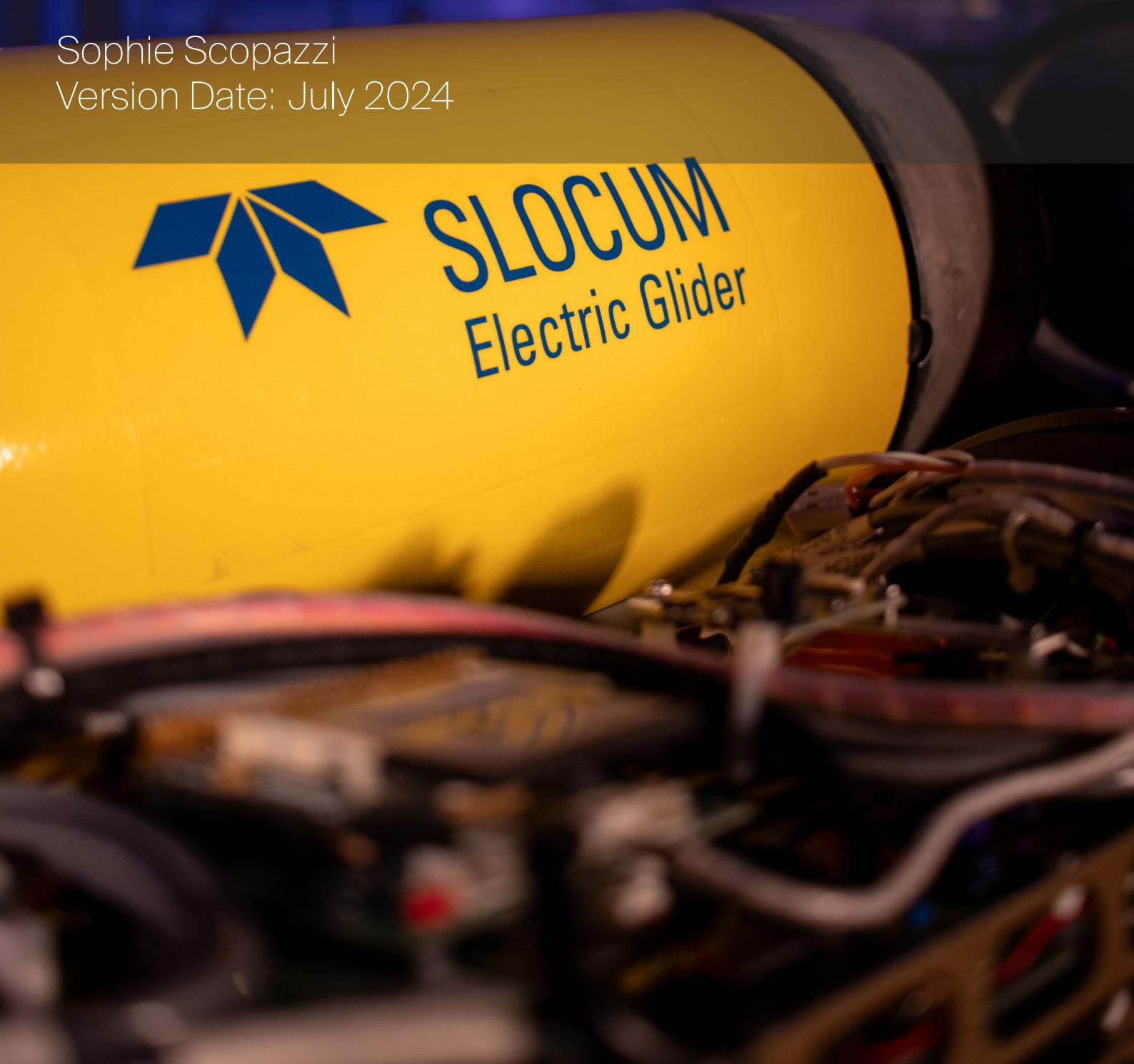
on a G3s Slocum Glider:
A Manual for Operators

Sophie Scopazzi

Version Date: July 2024



SLOCUM
Electric Glider



©2024 Sophie Scopazzi

Contents

Preface	vi
0.1 Intended Audience	vi
0.2 How to Use This Manual	vi
0.3 Piloting a Glider Autonomously: Proceed with Caution	vii
1 What is Backseat Driver...	1
1.1 What is an External Controller?	2
1.2 How the External Controller and Science Computer Communicate	2
2 External Controller Implementation	3
2.1 Raspberry Pi Specifications	3
2.2 Setup the Raspberry Pi	3
2.3 SSH access	3
2.3.1 Connecting to the Raspberry Pi via WinSCP	4
2.4 Automating the Script Using Crontab	4
2.4.1 Making the .py Launcher	4
2.4.2 Setting up the Crontab and Automating the .py Script	4
2.4.3 Viewing the output of the Raspberry Pi .py in the Cronlog	5
2.5 Troubleshooting Python Scripts on the Raspberry Pi	5
2.6 Connecting the External Controller to the Glider	5
2.6.1 Making the Connector	6
2.7 Which Files Go Where: Glider Pathways	9
2.7.1 extctl.ini	10
2.7.2 proglets.dat	10
2.7.3 sample90.ma	11
3 Running a Glider Mission in a Simulated Current Field	13
3.1 Using Slocum Fleet Mission Control (SFMC)	13
3.1.1 Sending Files to the Simulator	14
3.1.2 Getting Files off the Simulator	14
3.1.3 Logs, Scripts, Viewing Input Commands	14
3.2 Files Used in Test	15
3.2.1 bs_cft.mi	15
3.2.2 proglets.dat	15
3.2.3 extctl.ini	15
3.2.4 tbdlist.dat	16
3.2.5 simul.sim	16
3.2.6 sbdlist.dat	16
3.2.7 sample90.ma	16
3.2.8 yo99.ma	17
3.3 Local Mission Coordinates: Setting Location and Direction of Fast Current Zone	18
3.3.1 loadsim_cft.mi	18
3.4 Three ways to make a goto file	19
3.4.1 Manually	19
3.4.2 kml2goto.py	19
3.4.3 opencpn2goto.py	19
3.4.4 goto_l99.ma	19
3.5 RasPi script: depth_print.py	20
3.6 Starting the Simulated Mission	20
3.7 Plotting results using Python (current_field_test_simulation.ipynb/.py)	21

3.7.1	Getting the .sbd & .tbd files from the simulator	21
3.7.2	cac Files	21
3.7.3	Plots	21
4	Simulating a Large-Scale Ocean Current using an SFMC Script	22
4.1	Overview of SFMC Scripts	22
4.1.1	Breakdown of How a Script Works: sendWaterSpeedDir.xml	23
4.1.2	Changing the Surface Dialog: Adding Sensors to matchExpression With	23
4.2	Glider Simulator Limitations	24
4.3	Plotting Depth-Averaged Currents Using GGS: Yucatan Channel Example	24
4.4	Creating the SFMC Script to Control Simulated Variables (sendWaterSpeedDir.xml)	25
4.4.1	SFMC Script Part One: Building the simulated ocean current field	27
4.4.2	SFMC Script Part Two: Defining .xml script states	28
4.4.3	SFMC Script Part Three: Building the .xml script	29
4.4.4	Alternate Current Field	29
5	Verify .xml Script Behavior	30
5.1	Files Used	30
5.1.1	goto_l98/97.ma	30
5.2	Uploading and Selecting the SFMC .xml Script	31
5.2.1	Upload .xml script to SFMC	31
5.2.2	Select .xml script (sendWaterSpeedDir.xml)	31
6	Simple Backseat Driver Behavior Application: Commanding c_heading	34
6.1	Flying 90° From Depth-Averaged Currents	34
6.1.1	extctl.ini	34
6.1.2	set_he95.ma	35
6.1.3	right.mi	35
6.2	How the Raspberry Pi reads the Glider's Message	35
6.2.1	Simple Application: Raspberry Pi Script	36
6.3	90deg_bsd Test Results	37
7	Advanced Backseat Driver Behavior Application: Commanding c_heading, more sensors, make a plot	39
7.0.1	extctl.ini	39
7.0.2	sample90.ma	40
7.1	Advanced Application: Raspberry Pi Script	40
7.1.1	extctl Python class	40
7.1.2	while loop - parse strings from glider	41
7.1.3	while loop - continue when all values are present	42
7.1.4	crontab	43
7.2	Advanced Backseat Driver Output Plot	43
7.3	External Controller Python Script Run-Time Considerations	44
7.4	More Glider Simulator Limitations	45
8	Sending Files To/From External Controller	46
8.1	Baud Rate and Chunk Size	46
8.1.1	extctl Python class: sending/receiving files	47
8.1.2	extctl Python class: sending/receiving files	48
8.1.3	Python implementation for sending/receiving files	49
9	Science Sensor Integration, Testing, and Data File Management	50
9.1	Example A: Using an Actual CTD41CP Sensor	50
9.1.1	extctl.ini	50
9.1.2	proglets.dat	51
9.1.3	tbdlist.dat	51
9.1.4	sample01.ma	51
9.1.5	science.mi	51
9.2	Example B: Simulating a CTD41CP Sensor	52

9.2.1	Why Simulate a Sensor?	52
9.2.2	simCTD_synthetic.py	52
9.2.3	simCTD_file.py:	53
9.2.4	Making the Connector to Simulate a Science Sensor	53
9.2.5	Example B: simCTD_synthetic.py	54
9.2.6	Example B: simCTD_file.py	55
9.3	Example C: Retrieving and Processing the Most Recent .tbd File	56
9.3.1	Processing .tbd files onboard the EC	56
10	Alternative External Controllers	57
11	Conclusion	58
A	Most Optimal Heading Explanation	59
A.1	Deflection Distance	59
A.2	Calculating the Most Optimal Heading	62
A.2.1	Python function: def what_is_best_heading	62

Preface

The Backseat Driver (BSD) architecture on a G3S Slocum glider is unknown and complicated, even to those familiar with glider operations and piloting. The unfamiliar capabilities and implementation leads to uncertainty, and ultimately a lack of end-user use of the Slocum BSD architecture. For this reason, this manual was created.

This manual is only a starting point for implementation. It has been created for my Master's thesis *Sensor and Behavior Frameworks for Implementing Backseat Driver on a Slocum Glider* because there was no documentation of this type before.

All the files used throughout this document are available on my Github:
https://github.com/sscopazzi/Slocum_Backseat_Driver_Manual

0.1. Intended Audience

To the best of my ability, I've tried to approach the writing of this manual assuming no prior knowledge; however, prior knowledge of Slocum glider operation and piloting, along with experience with Python functional programming would be extremely helpful. If that sounds scary, please don't let that dissuade you from working through these topics. They can be difficult to work through, especially at first, but it'll be okay!

0.2. How to Use This Manual

This manual has been written so one can work sequentially through this document and the concepts within as they progressively get more advanced. If you feel you already know something, please feel free to use the Contents page to skip to the area you want to know more about!

- **Chapters 1-2:** Background about what the backseat driver architecture is, what an external controller is, how it is used and the type used in this manual, then how to implement the external controller onboard a glider or a glider simulator.
- **Chapters 3-6:** Learning glider piloting and simulation, then performing a simple backseat driver mission.
- **Chapters 7-9:** A more advanced backseat driver application, then reading and manipulating science data onboard the external controller. Sending a file processed on the external controller to the glider, for ability to retrieve via SFMC mid-mission.

I highly recommend reading through Teledyne Marine's G3S Backseat Driver Software Guide prior to using the examples in this manual. It will provide more of the software theory behind the examples then I will explain in this manual. At the time of writing I:

- Referenced version M315400-NFC, Rev A of the Slocum Backseat Driver Software Guide
- This manual uses Slocum glider software version 11.00 SW311356-NFC-RA

0.3. Piloting a Glider Autonomously: Proceed with Caution

In standard operation, a Slocum glider does not do anything not commanded of it by human pilots. Through using the BSD architecture, human pilots become one step removed. The Slocum BSD architecture, made by Teledyne Marine, allows us greater control over the glider system through allowance of onboard decision making capability. Because of this, it is important to note that in some manner we are removing ourselves from the in-situ piloting equation. Unbeknownst to us at the surface the glider may "decide" to do things while underwater we would never want it to do. This gets complicated by the sheer number of possible ways to use the Slocum BSD architecture. Thus, there are some tenants to keep in mind:

1. **Simulations Simulations Simulations:** Before putting a glider in the ocean controlled with the BSD architecture, simulate in the lab. This alone can get highly specific to your use case, based upon the question you are trying to answer. This manual details how to simulate science sensors, and more to meet this need.
2. **Log Everything Possible:** While testing your BSD architecture application, log everything, and not only about the glider. Ensure everything you could possibly want is saved in the glider files. How is the external controller performing? Log everything possible onboard. Simulations are real-time, meaning simulating a multi-day mission will take multiple days. Conduct short missions first, check your logs are in order, then continue. And then, if possible, ensure there is some sort of logging during a real mission. If anything goes awry, this will save you.

List of Figures

1.1	Backseat Driver Conceptual Layout (basic): Red box contains standard commercial-off-the-shelf glider configuration, with the addition of the external controller in green	1
2.1	Science Computer and External Controller Connector: Image of Parts During Simulation	6
2.2	Science Computer and Laptop Connector: simCTD RS-232 cable. Blue wire nuts for increased modularity	6
2.3	Science Computer and External Controller Connector: Individual parts labeled with words	7
2.4	Science Computer and External Controller Connector: Wire Diagram	8
2.5	Internal Glider File Pathways: Which Files Go Where	12
3.1	SFMC: Top right folders - to/from-glider, file drop zones, logs	14
3.2	SFMC: Selecting scripts and viewing their current state	14
3.3	SFMC: Viewing recently submitted commands	14
3.4	Waypoint Locations	19
3.5	Current Field Test: Cac files as seen from the glider terminal	21
3.6	Current Field Test: Output plots, viewing by LMC coordinates	21
4.1	SFMC Script Example: initialState, matchExpression, toState, state, action	22
4.2	SFMC Script: help command on glider	23
4.3	Yucatan Depth-Averaged Currents: An early output from the Glider Guidance System	25
4.4	SFMC Script: Blue dashed lines delineate the boundaries between changing currents, with the arrow in each vertical slice of longitude indicating the magnitude and direction of currents within.	26
4.5	SFMC Script Alternate Current Field: Blue dashed lines delineate the boundaries between changing currents, with the arrow in each vertical slice of longitude indicating the magnitude and direction of currents within.	29
5.1	SFMC Script: Where to upload scripts in SFMC	31
5.2	SFMC Script: Where to select scripts in SFMC glider terminal	31
5.3	SFMC Script: Location of waypoints used for both missions.	32
5.4	SFMC Script: c_heading comparison between both missions	32
5.5	SFMC Script: Blue dashed vertical lines are longitude current boundaries. Arrows are current direction and magnitude within each boundary. See figure 5.2 for waypoint locations.	33
6.1	Simple Backseat Driver: Cronlog from Simple Backseat Driver Application: The fields in variable cur_line on each glider cycle. Note intermittent "5:", processing, and sending of new heading from RasPi to glider.	37
6.2	Simple Backseat Driver: c_heading 90° from m_water_vel_dir during simple backseat driver application	37
6.3	Simple Backseat Driver Location Comparison: Green line is location of spectre ran with the BSD control architecture (rest of map the same as Figure 5.4). Note green's arrival to eastern boundary faster than red's with its southeast waypoint, but with greater deflection distance northward.	38
7.1	Advanced Backseat Driver: External controller output plot	43
7.2	Advanced Backseat Driver: Script Run Times	44
9.1	Example B: simCTD_synthetic.py's simulated CTD data	52
9.2	Example B: simCTD_synthetic.py's simulated CTD data as seen from the external controller	53
A.1	Idealized for illustration purpose only. Red is heading of 090°T and green is heading of 120°T	60

A.2 Two cases: the glider's velocity is constant while the river's velocity changes	60
A.3 All heading angles $000^{\circ}\text{T} - 180^{\circ}\text{T}$	61
A.4 Heading angles $100^{\circ}\text{T} - 180^{\circ}\text{T}$ (same plot as Figure A.3 above, but more zoomed in)	61
A.5 Gray arrow indicates current direction. Green arrow indicates optimal heading with green dashed line as the resultant course over ground (COG) and 3hr location. Red dashed line indicates the COG for the heading 090°T to the right of the current and 3hr location. Light blue circle is resultant location for all headings $000^{\circ}\text{T} - 360^{\circ}\text{T}$. Dark blue section headings of $045^{\circ}\text{T} - 135^{\circ}\text{T}$. For all cases of current direction $335^{\circ}\text{T} - 090^{\circ}\text{T}$, a heading of 135°T results in less deflection northward (while still retaining eastward velocity) than headings 90° to the right of the current.	63

List of Files

2.1	Backseat Driver Implementation: launcher.sh	4
2.2	Backseat Driver Implementation: Setup the the Crontab	5
2.3	Current Field Test: extctl.ini - science computer's external processor settings file	10
2.4	File Explanation: proglets.dat	10
2.5	File Explanation: sample90.ma	11
3.1	Current Field Test: bs_cft.mi - mission file	15
3.2	Current Field Test: proglets.dat - science computer's sensor file	15
3.3	Current Field Test: extctl.ini - used to control m_depth	15
3.4	Current Field Test: tbdlst.dat - science computer's data file	16
3.5	Current Field Test: simul.sim - must be on glider simulator for simulating	16
3.6	Current Field Test: sbdlist.dat - flight computer's data file	16
3.7	Current Field Test: sample90.ma - flight computer's sensor specific file	16
3.8	Current Field Test: yo99.ma - flight computer's control of dive and climbs (yo9)	17
3.9	Current Field Test: loadsim_cft.mi - setting simulated variables	18
3.10	Current Field Test: goto_l99.ma - commands the glider to go to a waypoint	19
3.11	Current Field Test: Raspberry Pi Script to change d_target_depth one time	20
4.1	SFMC Script: Building the simulated ocean current field	27
4.2	SFMC Script: Defining .xml script states	28
4.3	SFMC Script: Building the .xml script	29
5.1	Verify .xml Script Behavior: goto_l98/97.ma - waypoints	30
6.1	Simple Backseat Driver: extctl.ini - used to control c_heading	34
6.2	Simple Backseat Driver: set_he95.ma file for set_heading behavior	35
6.3	Simple Backseat Driver: right.mi - mission to fly 90° from ocean current	35
6.4	Simple Backseat Driver: sendHeadDir.py - Python script ran on the Raspberry Pi BSD	36
7.1	Advanced Backseat Driver: extctl.ini - control c_heading, send more sensors (lat/lon, etc)	39
7.2	Advanced Backseat Driver: sample90.ma - power the RasPi on only on surface and diving	40
7.3	Advanced Backseat Driver: sendHeadDir_WAIT.py - extctl Python class	40
7.4	Advanced Backseat Driver: sendHeadDir_WAIT.py - parse strings from glider	41
7.5	Advanced Backseat Driver: sendHeadDir_WAIT.py - continue when all values are present	42
7.6	Advanced Backseat Driver: Setup the Crontab	43
8.1	Advanced Backseat Driver: sendHeadDir_WAIT.py - continue when all values are present	47
8.2	Advanced Backseat Driver: sendHeadDir_WAIT.py - continue when all values are present	48
8.3	Advanced Backseat Driver: sendHeadDir_WAIT.py - continue when all values are present	49
9.1	Example A: extctl.ini - send CTD data	50
9.2	Example A: proglets.dat - include CTD41CP proget	51
9.3	Example A: extctl.ini - send CTD data	51
9.4	Example A: tbdlst.dat - include CTD science data	51
9.5	Example A: science.mi - calls sample01.ma as well	51
9.6	Example B: simCTD_synthetic.py	54
9.7	Example B: simCTD_file.py	55
9.8	Example C: Processing .tbd files onboard the EC	56
A.1	what_is_best_heading - Python function used to calculate heading	62

1

What is Backseat Driver... ...and how it relates to G3S Slocum Gliders

In **analogous terms**, a Backseat Driver (BSD) is like when you and your partner drive to visit a friend in a new-to-you town. It's your car so you drive with your partner in the passenger seat, and the local friend in the back seat. The local friend is scooted up and leaning forward through the front seats pointing across your face telling you to turn left so you don't miss the exit. Your friend, in the backseat, is seeing what's outside the car and making decisions that you, in the driver's seat, blindly follow. Maybe they even disagree with what the maps on your phone is saying because they know something better based on prior knowledge or what they see outside.

In this analogy, the driver is the flight computer, the passenger is the science computer, and the local friend is the BSD. The flight computer and science computer are already part of the glider system (you and your partner's car), hence the term "external controller" for the external controller (friend visiting your car). Conceptually, the external controller is given the name "external" because it resides outside the glider's normal system (Figure 1.1, red box to right). It's important to note that it is not outside or external to the glider housing, just the glider's standard control loop.

An important nomenclature note that can be confusing while researching this topic is that the phrase external controller and BSD can often be equated to mean the same thing, so long as we are not talking about backseat driver as a concept. For clarity, in this manual I will:

- Use external controller when the computer could be conceptually generic
- Use RP when talking about specific use-case examples in this manual, using the Raspberry Pi
- Use BSD to refer to the backseat driver concept

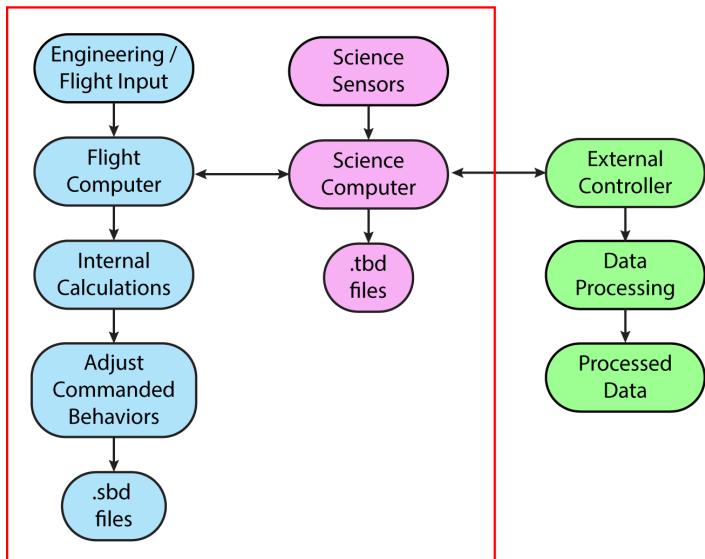


Figure 1.1: Backseat Driver Conceptual Layout (basic): Red box contains standard commercial-off-the-shelf glider configuration, with the addition of the external controller in green

1.1. What is an External Controller?

The external controller could technically be almost any kind of computer capable of receiving and sending serial data, but it also must exist within the size and energy consumption constraints that come from being housed internal to the Slocum glider during long-duration missions; i.e., it essentially must be a small form factor and energy efficient single-board computer. This has led to adoption of external controllers such as a BeagleBone Black or a Raspberry Pi 4, with other exciting newer possibilities like the NVIDIA Jetson Nano. However implemented, an external controller onboard the Slocum glider allows for greater decision making, control fidelity, and onboard data processing than possible without.

At a practical level The external controller runs a python script, so think of it more as user directed autonomous control. It does think for itself, but within constraints placed upon it by us, the creators of the Python script. Thus, what the external controller does is:

1. Receive glider values and/or a sensor's data file specified by the user
2. Processes and/or manipulates the data
3. Sends commands to the glider to alter its behavior and/or process a sensor's datafile into a smaller file and transfer it back to the science computer for sending ashore (process an onboard AD2CP file into a smaller file)

1.2. How the External Controller and Science Computer Communicate

The external controller and science computer communicate by configuring the external controller as a sensor, which is connected to a science computer's open port. The physical connection is made using a custom cable detailed in a later chapter. The external controller and science computer use the standard RS-232 communication protocol, allowing for two way communication. Character combinations specify the type of message and delimiters break up the values therein. There are two typical ways the Slocum backseat driver architecture is organized:

Table 1.1: Two Typical Approaches to Using the EC

Always Powered Example: Adaptive Control	Intermittent Power Example: File Processing
<ol style="list-style-type: none"> 1. The external controller subscribes to sensor values that are already onboard the glider. The values can be from either the flight computer (heading, etc.) or science computer (temperature, salinity, or other variable from science sensors) 2. Every glider cycle, the science computer sends each updated subscribed sensor value to the external controller. 3. The EC's Python script takes the incoming string and breaks it apart using delimiters built into the message (first (*), then (,), then (:)) to get the value from the science computer into a usable format on the EC 4. Using the same Python script, the external controller manipulates or otherwise uses the incoming values to make decisions not possible onboard the flight computer or science computer in real-time 5. The external controller then updates flight sensor values (b_args) on the flight computer via communicating back to the science computer. 	<ol style="list-style-type: none"> 1. The science computer powers the external controller on as defined in the typical glider sample behavior (15 choices). Once powered on, the Python script runs on external controller startup, requesting the most recent file from the glider. This can be any file on the science computer, but usually from a sensor whose file is too large to send ashore without onboard processing, like an AD2CP 2. The file is transferred to the EC, then processed, usually into a smaller file size. 3. The external controller sends the smaller file back to the science computer for sending ashore via Iridium

All the different message formats are described in the following tables. Some are "background" messages not directly controlled by the user, while others must be specified in the python script ran onboard the external controller.

2

External Controller Implementation

Chapter Concepts:

1. Download the RasPi operating system and flash it to a microSD card.
2. Set up SSH access on the RasPi for remote access.
3. Remotely connect to the RasPi via SSH using WinSCP.
4. Automate the operation of our script using the RP's crontab.
5. Review handy troubleshooting commands that can be handy to have.
6. Make the physical connection between the RasPi and the glider's SC.

2.1. Raspberry Pi Specifications

In this manual's case, the external controller is a Raspberry Pi. A RasPi is a low-power single-board computer that's ubiquitous in classes that teach basic hands-on computer science.

2.2. Setup the Raspberry Pi

Before implementing the BSD capabilities, we will first download the operating system for the RP. The steps to do so are as follows:

1. Download Raspberry Pi Imager V 1.7.5 for your operating system.
 - (a) Insert the SD card into your laptop or card reader.
 - (b) Click "Choose OS" – choose the Raspberry Pi OS (64-bit) option. If it isn't listed on the main page, select Raspberry Pi OS (other).
 - (c) Click "Choose Storage" – select the appropriate storage for your project.
 - (d) Click "Write" – this will install the options you have selected above.
2. Insert SD card into slot in the Raspberry Pi and power on
3. Set up a username (rutgers) and password (rutgers)
4. Create a folder for your project in the '/home/rutgers/' directory. Something like a folder called `backseat_driver`. Make sure the folder has no spaces, hence the _.
5. In this folder, you will have your python scripts for your project and a shell script that executes those scripts, along with anything else needed for the mission by the RasPi (we will do this near the end).

2.3. SSH access

Access to the RasPi via SSH, secure shell, is required for ability to remotely send and receive files while the RasPi is inside a glider. SSH is a communication "protocol...for securely sending commands to a computer over an unsecured network." (*CloudFlare: What is SSH*). If your university has a VPN to connect to the universities network, the additional benefit of connecting the RasPi to an internet connected network (a universities, or departments) is the ability to remotely SSH in from home while running tests.

1. Enable SSH access by either, 1) From the RP's desktop GUI: Preferences -> Raspberry Pi Configuration -> Interfaces -> Enable SSH, or 2) From the RP's terminal: `sudo raspi-config -> interface options -> SSH`
2. Once SSH has been enabled, reboot the RasPi from the desktop GUI or from a terminal with the command: `sudo reboot`
3. Connect the RasPi to the same network as your computer (wireless network or Ethernet). Connecting to the same wifi network on the RasPi and your computer is probably the easiest option.
4. Get the RP's ip address it was assigned on the network in the RasPi terminal with command: `hostname -I`
5. Verify connectivity between your computer and the RasPi via SSH. From a command prompt or terminal window: `ssh rutgers@[RP's ip address]`. It probably will ask if you trust the source, so type `yes` then the password you setup earlier. When typing passwords into a terminal window it will not look like anything is happening (it doesn't show `*****` characters). This is normal behavior. Trust your keypresses then hit enter. If unsure halfway through, press delete a lot or hit enter so it asks you to retype.

```

1 hostname -I           # from the terminal shows RasPi ip address
2 sudo reboot            # from the terminal, reboots the RasPi
3 ssh rutgers@[RP's ip address]  # from windows/mac ssh into the RP

```

Now that the RasPi can be accessed via SSH, you can send and receive files over the shared network. This can be done via the command line from the Windows or Mac computer; however, it's far easier to use a program like WinSCP to send files.

2.3.1. Connecting to the Raspberry Pi via WinSCP

Section under construction.

(WinSCP Website). A Mac program one can use is Cyberduck.

2.4. Automating the Script Using Crontab

To automate the script on the RP, we require a two files:

1. **launcher.sh**: A launcher file that runs the python script (.py).
2. **python_script.py**: The script we will run is written in python and saved as a .py file. The program the RasPi runs makes decisions, runs calculations, etc. This script will be in our /backseat_driver folder on the RP, and is provided later in this manual for each chapter.

2.4.1. Making the .py Launcher

The launcher is required to avoid errors and to ensure the log outputs properly. This is the entirety of the launcher.sh file in the project's home directory `home/rutgers/backseat_driver`.

```

1 # launcher.sh file on the RP
2 cd /                         # make sure we are in the home directory
3 cd /home/rutgers/backseat_driver  # move to directory where the .py script is
4 sudo python3 [script name].py    # run the script
5 cd /                         # go back to home directory

```

File 2.1: Backseat Driver Implementation: launcher.sh

2.4.2. Setting up the Crontab and Automating the .py Script

1. Send command `sudo crontab -e` to open the crontab on the RP. This opens a file on the RP, *but only edit via this command*
2. Enter the lines below to run the script in the x time period you want.

```

1 # to open crontab thru the RasPi terminal (only open via terminal, not the file on the RP)
2 sudo crontab -e
3
4 # at end of file input these commands with the path to launcher
5 * * * * * sleep 5: sh /home/rutgers/backseat_driver/launcher.sh >> /home/rutgers/logs/cronlog
6 * * * * * sleep 10: sh /home/rutgers/backseat_driver/launcher.sh >> /home/rutgers/logs/cronlog
7 * * * * * sleep 15: sh /home/rutgers/backseat_driver/launcher.sh >> /home/rutgers/logs/cronlog
8 * * * * * sleep 20: sh /home/rutgers/backseat_driver/launcher.sh >>
9           ... etc ...
10 * * * * * sleep 55: sh /home/rutgers/backseat_driver/launcher.sh >> /home/rutgers/logs/cronlog
11 * * * * * sleep 60: sh /home/rutgers/backseat_driver/launcher.sh >> /home/rutgers/logs/cronlog
12           2>&1
13 # to run once every minute
14 * * * * * sh /home/rutgers/backseat_driver/launcher.sh >> /home/rutgers/logs/cronlog 2>&1
15
16 # to display updates written to cronlog while crontab is running
17 # tail -f cronlog

```

File 2.2: Backseat Driver Implementation: Setup the the Crontab

The crontab is the RP's way of running a script on reboot or a schedule. The shortest cronjob time period is every minute, but we can get around this limitation by using the sleep command, and running a cronjob every x seconds. The end result is a script that runs every x seconds. The end of each line above (2-11) is how the output of the script is saved in the cronlog folder (important for troubleshooting and viewing script running).

2.4.3. Viewing the output of the Raspberry Pi .py in the Cronlog

Now that we specified a location to log the output of the script, we can view the log as the RasPi runs the script. To display a static version of the entire cronlog first navigate to the cronlog file location described above using `cd [directory name]`. Once there, use `nano cronlog` or `vim cronlog`; however, to view the end of the file as new running scripts update the file, use `tail -f cronlog`. This prints the results of the script as it's run, as it's showing the updates to the tail (the end) of the cronlog. To get out of the `tail -f` command, use `control-c`.

2.5. Troubleshooting Python Scripts on the Raspberry Pi

```

1 # list all running python scripts
2 ps -aef | grep python
3
4 # if need to kill ALL scripts
5 sudo killall python3
6
7 # if need to kill specific scripts
8 sudo killall -e python3 sendHeadDir.py
9
10 To run on boot use this command in the crontab (for reference)
11 @reboot python /home/rutgers/bsd/script.py

```

If the RaspPi script runs fast you will probably not see it using the `grep` command, but if the .py is run in a program like Thonny it *should* show up as Thonny 'susends' instead of fully ending the script when ran. This also depends on the script, as an 'if' statement will end, but a 'while' statement could hang and wait for values (we'll get into this later).

2.6. Connecting the External Controller to the Glider

The external controller needs to communicate bidirectionally with the glider to read variables and/or files from the glider and to provide resultant control and/or files back to the glider. This necessitates a physical connection from the RasPi to the glider's science computer. This is done by using a custom cable and utilizing a UART communication method.

2.6.1. Making the Connector

The photo below (Figure 2.1) shows the parts as ran for simulations: top left is the RasPi; top right is the RS-232 and DC power adapter; bottom center are the shoebox simulator's science port. The parts used:

1. **10-pin Male Molex:** Words
2. **Weird small connector thingies and their tool:** Words
3. **DC Adapter:** Words
4. **RS-232 Adapter:** Words
5. **placeholder because I forgot:** Words



Figure 2.1: Science Computer and External Controller Connector: Image of Parts During Simulation

Figure 2.3 (next page) shows each component part for the external controller with corresponding label. Figure 2.4 (next next page) is the wiring diagram for making the external controller's connector.

Additionally, if you will be simulating a science sensor I'd recommend making the connector for that now (do it all at once). The photo below (Figure 2.2) shows the simCTD USB to RS232 to Molex connector.

How to Wiring Resources:

1. SparkFun resource for *how to read a wiring schematic*
2. KiCad Schematic Editor help documentation

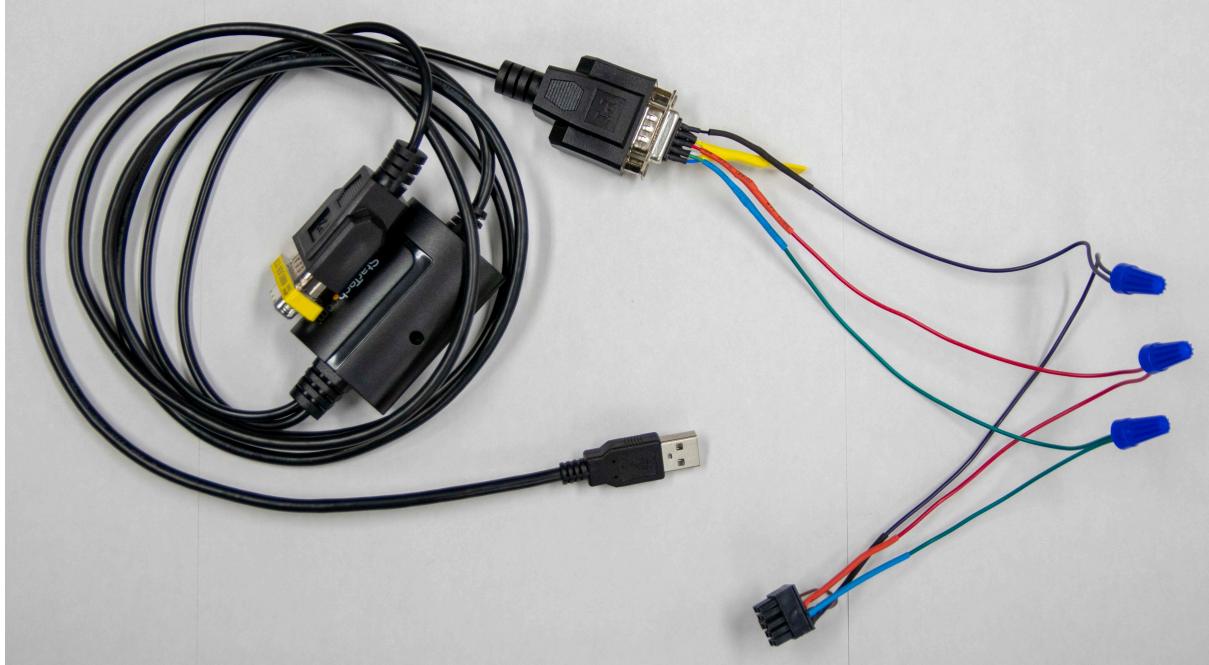


Figure 2.2: Science Computer and Laptop Connector: simCTD RS-232 cable. Blue wire nuts for increased modularity

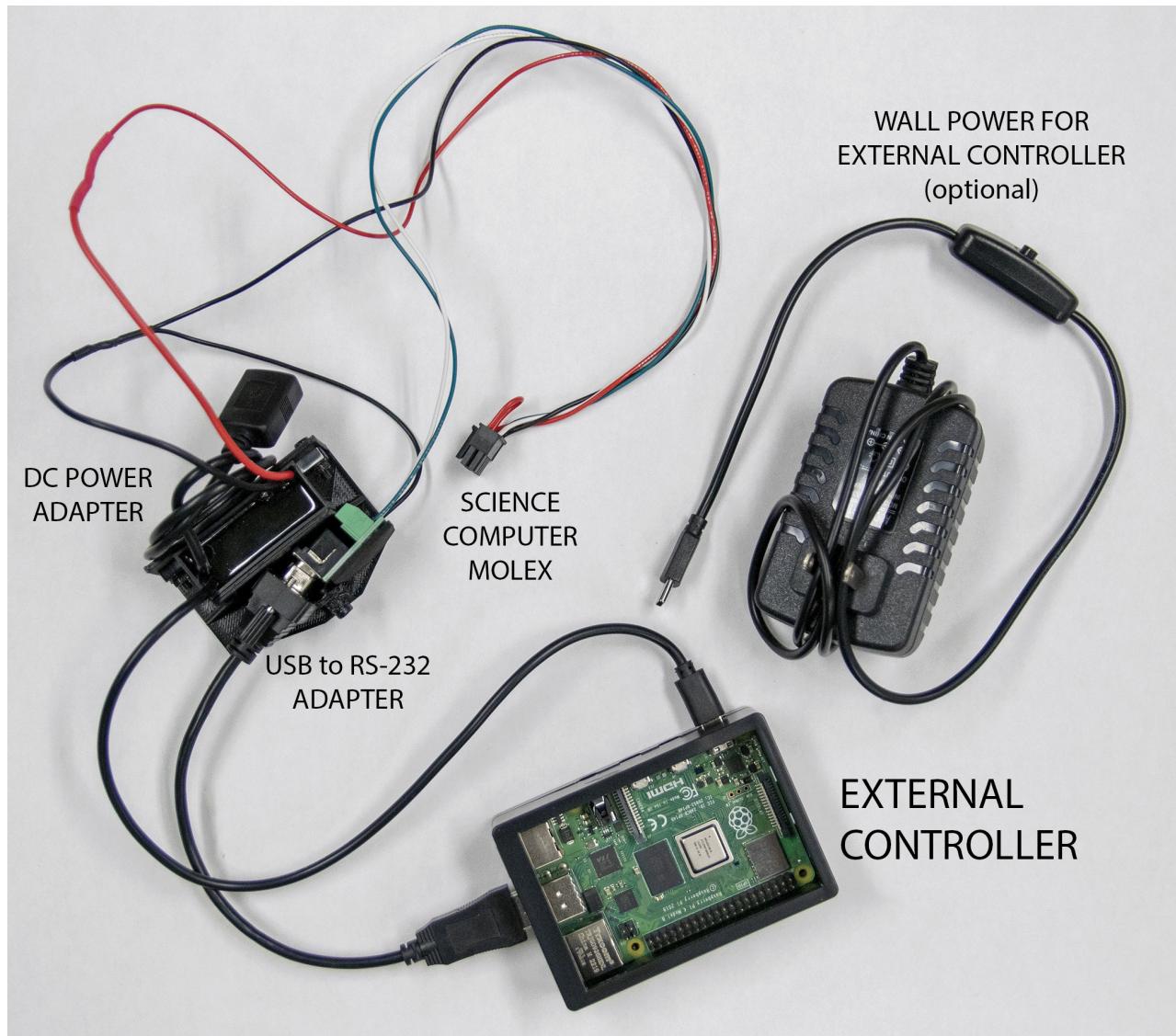


Figure 2.3: Science Computer and External Controller Connector: Individual parts labeled with words

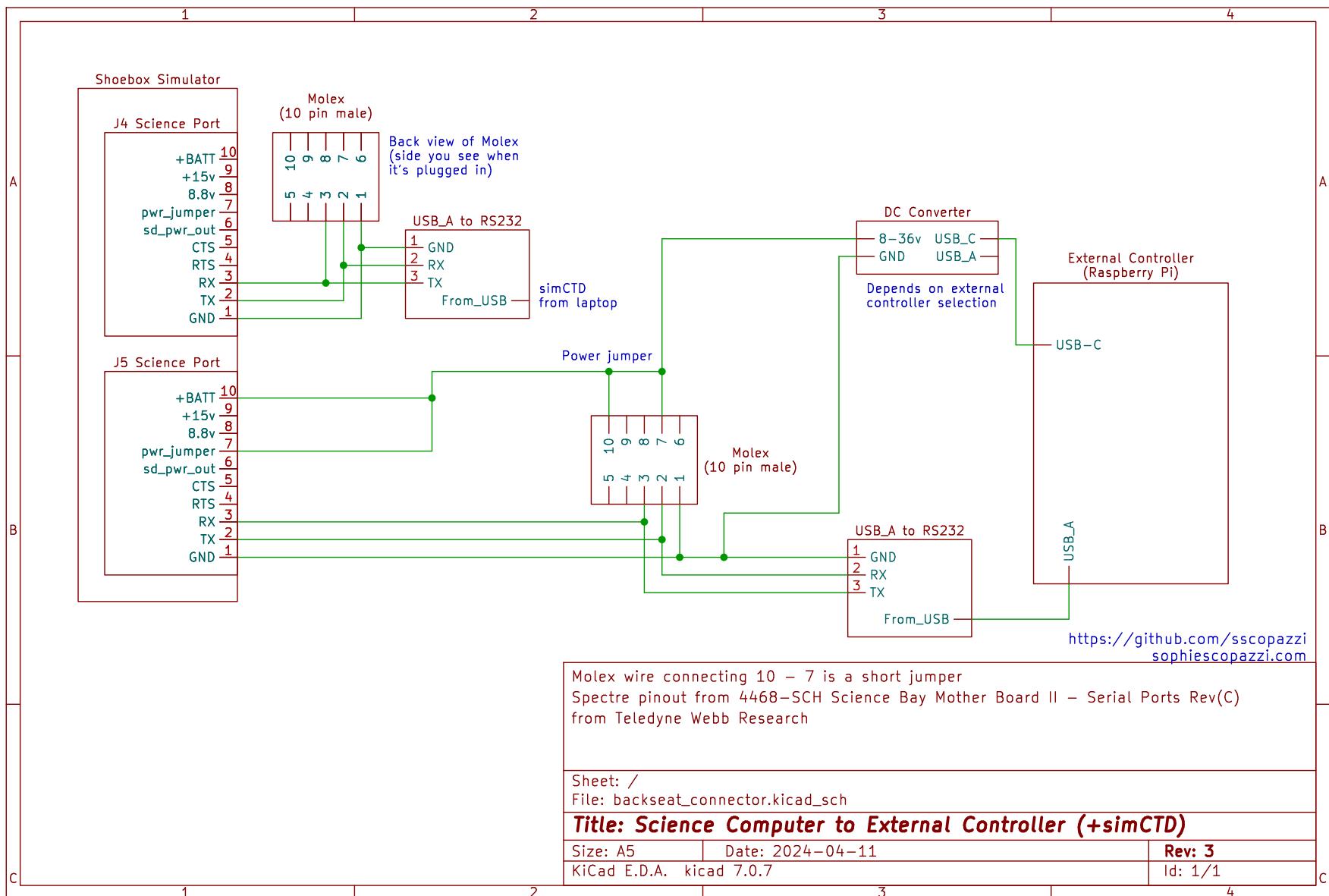


Figure 2.4: Science Computer and External Controller Connector: Wire Diagram

2.7. Which Files Go Where: Glider Pathways

The list of files we will reference or use for each chapter is large, and where they go on the glider is vitally important. In most cases, the file extension explains where the file goes; .mi for the /missions/ folder and .ma for the /mafiles/ on the FC. Where it can get confusing is with .dat, .ini, etc. When dropping files into the "To Glider" folder via SFMC and sending them to the glider using dockzr *.mi *.ma etc, the glider puts the files in the correct folder. Highlighted files are required for using the BSDA.

Table 2.1: Reference Documents

Filename	What it's for
masterdata.txt G3S Backseat Driver Software Guide	Defines all values used by the glider. These are the values used unless altered in future files (by autoexec.mi, for example) I will attempt to keep this manual up to date, but the version used for this work is P/N M315400-NFC-NFC, Rev. A. Software Version 11.0. Please contact Teledyne Marine for the most up-to-date copy.

Table 2.2: Flight Computer Files

Filename	What it does	Where it goes
autoexec.mi	Glider specific file containing everything that makes the glider that specific glider (calibrations for onboard equipment, etc.)	/config/
sbdlist.dat	Specify which FC values to save and how to save them (decimation)	/config/
simul.sim	Tells the glider that we are simulating, and how (electronics only, etc.)	/config/
bs_cft.mi	The mission file that sets b_args (values) and calls the .ma files	/missions/
loadsim.mi	Ran before starting the mission, this initializes the simulated variables	/missions/
sample90.ma	Controls the sample behavior of the extctl proglert (the external controller)	/mafiles/
yo99.ma	Controls the yoing of the glider - dives and climbs	/mafiles/
surfac10.ma	Controls surface timeouts and other parameters	/mafiles/
surfac40.ma	Each number relates to a different type of surfacing reason	/mafiles/
surfac42.ma	For example, this is mission end	/mafiles/

Table 2.3: Science Computer Files

Filename	What it does	Where it goes
extctl.ini	Controls and specifies which science computer values are sent to the BSD	/config/
proglets.dat	words	/config/
tbdlist.dat	Specify which science values to save in the .tbd files and how often to save them (decimation). See comments in file for more detail	/config/

Table 2.4: Raspberry Pi Files

Filename	What it does	Where it goes
current_field_test.py launcher.sh crontab	the BSD script ran on the RasPi Specified in the crontab, this runs the .py Runs the .py on a schedule	/backseat_driver /backseat_driver Accessed with command sudo crontab -e

2.7.1. extctl.ini

```

1 # EXTCTL.INI
2 # Maximum number of variables one can be subscribed to is 64 by default
3
4 # MISSION PARAMS SECTION - u_mission_param[a-1]
5 mp
6                               # INDEX NUMBER FOR RasPiPYTHON
7 u_mission_param_a          # 0
8 # c_heading in 7099.ma
9
10 # OUTPUT SENSORS SECTION
11 # there needs to be one sci_generic defined here per mission param used
12 os
13 sci_generic_a nodim      # 1
14
15 is # INPUT SENSORS FROM GLIDER SECTION
16
17 # FLIGHT COMPUTER - example variables
18 m_water_vel_dir rad     # 2
19
20 # SCIENCE COMPUTER - example variables
21 sci_water_cond S/M       # 3
22 sci_water_temp degC     # 4
23 sci_water_pressure bar   # 5
24
25 baud        # Baudrate of UART
26 9600        # default value
27
28 packet_size # specify packet_size of files transferred
29 600         # default value

```

File 2.3: Current Field Test: extctl.ini - science computer's external processor settings file

Both the baud and packet_size can be higher for faster file transfers, but a baud over 15200 is not recommended for stability. The max packet size is 730. If baud and packet_size arguments are left out of the extctl.ini, the science computer will use the default values. I leave them in all files used for clarity throughout this manual.

IMPORTANT: During each glider cycle, the message send to the external controller contains only the **updated** values specified in the extctl.ini, not **every** value asked for on **every** cycle. To restate, **if the sensor value has not been updated on that specific cycle of the glider, the value will not be sent during that cycle's message.** Teledyne Marine knows this is a desirable toggle-able setting and it may be included in a future BSD release.

2.7.2. proglets.dat

```

1 proglet = extctl           # define name of proflet, corresponds with sample90.ma on the FC
2     uart      = j5          # science port it's plugged in (could also be j4)
3     bit       = 0           # power bit control, ability to turn the RasPion/off
4     start_snsr = c_extctl_on(sec)

```

File 2.4: File Explanation: proglets.dat

2.7.3. sample90.ma

```
1 behavior_name=sample
2
3 <start:b_arg>
4     b_arg: sensor_type(enum)           90 # c_extctl_on
5     b_arg: sample_time_after_state_change(s)    0 # start sampling right away
6     # Sampling Arguments
7         b_arg: intersample_time(sec)        0
8         b_arg: state_to_sample(enum)       15
9         b_arg: nth_yo_to_sample(nodim)     1
10 <end:b_arg>
```

File 2.5: File Explanation: sample90.ma

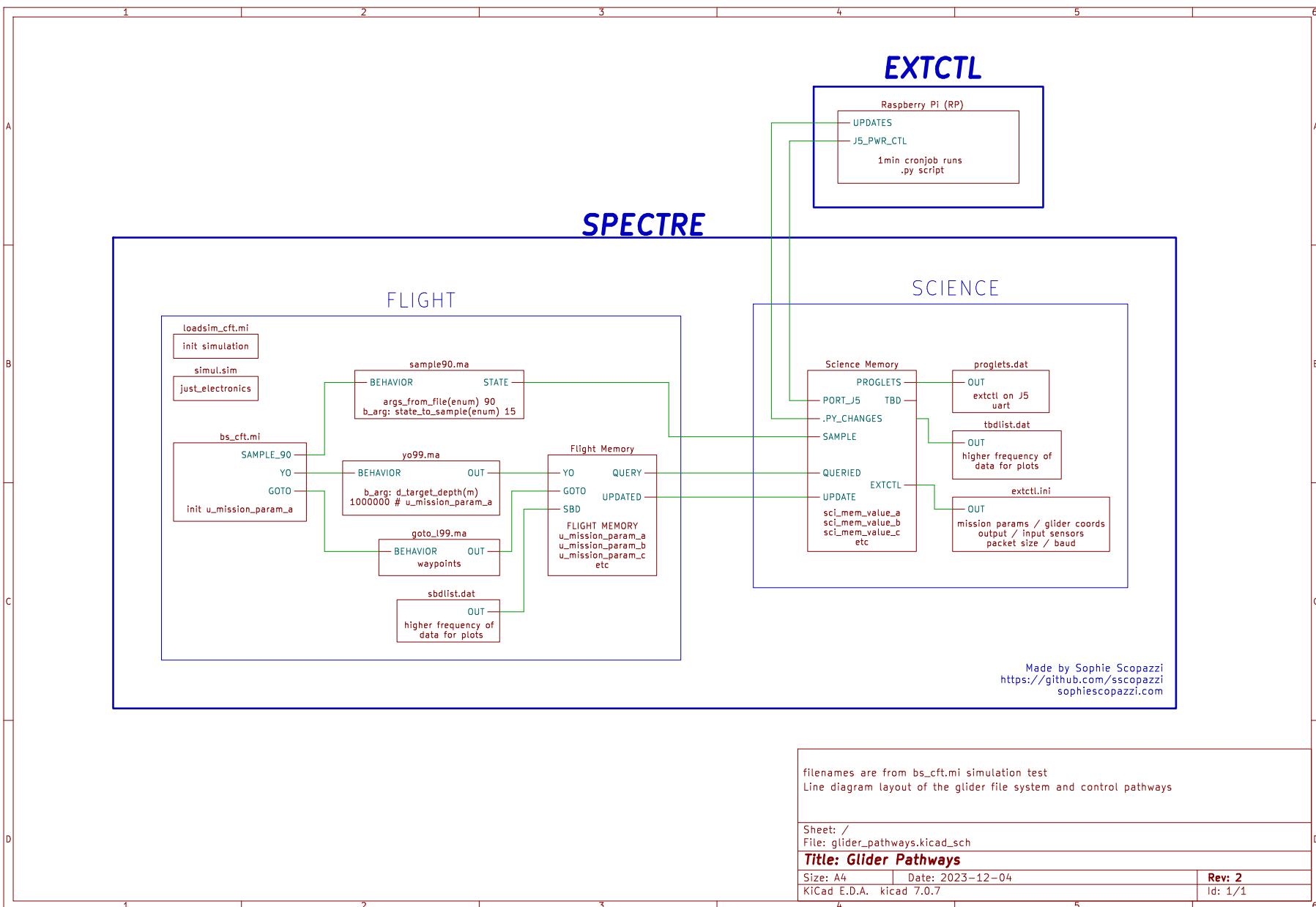


Figure 2.5: Internal Glider File Pathways: Which Files Go Where

3

Running a Glider Mission in a Simulated Current Field

Chapter Concepts:

1. Review every file required for piloting a glider in the simulator
2. Simulate a basic ocean current using the glider simulator's built-in fast speed zone
3. Control m_depth using the external controller

Now that we have a fully configured and ready external controller, our RasPi, and have created our wired connection to the glider science computer, we can get into using the glider simulator. This chapter familiarizes us with how to pilot a glider and all the files required and what they contain. We will also implement an extremely basic type of BSD control where we change the commanded depth of the glider after it dives. This chapter's information and files are the building blocks for all future chapters and glider missions.

3.1. Using Slocum Fleet Mission Control (SFMC)

SFMC is the web interface used to control gliders. Through SFMC we can send commands, send and receive files from the glider, and select SFMC .xml scripts (which we go over in future chapters) that control the glider automatically without any user input. There are many other features of SFMC, but we will only go over what is necessary for this manual here, starting with the typical glider terminal commands needed for this work.

Table 3.1: SFMC Terminal Commands

Command	What it does
{enter key}	hows ready for commands
dir	list files in the current directory
cd [directory name]	Changes terminal to that directory
type [filename]	print filename to glider terminal
dockzr [file]	Send files from SFMC to the glider
zs [file]	Send files from the glider to SFMC
*.[file extension]	The * acts as a wildcard, representing every file with that extension
sample*.ma	The same idea as above, but represents every file starting with sample
ver	Displays glider name and software version Ex: Vehicle Name: spectre, Glider: Version 11.00 SW311356-NFC-RA (0, 1)
consci	Transfer from the flight computer to the science computer
quit	Return from the science computer to the flight computer
control-C	Quits the current mission
control-R	Resumes the mission (glider dives)
exit reset	Restart the flight computer (will boot back up)
exit	Turns off glider (does not boot back up)
help	Shows a list of commands (far more than shown here)

3.1.1. Sending Files to the Simulator

On the top right is a blue box called "to-glider". Files in this folder can be sent to the glider. Below the blue box is the "To Glider File Drop Zone", the place to drop files that will be sent to the glider. However, unlike a standard glider which utilizes the "to-science" folder mid-mission, we will only be using the "to-flight" folder and that drop-zone.

An example of steps to send two files, `bs_cft.mi` and `sample90.ma`, to the glider is below:

1. Create (or download from Github) both files. To follow the Rutgers Glider Team's convention, make a duplicate of both files and add a `.cur` file extension. This is done to have an easily accessible record of what's actually on the glider, because once the `.ma` file has been sent it disappears from the folder. Thus, drop four files: `bs_cft.mi`, `bs_cft.mi.cur`, `sample90.ma`, `sample90.ma.cur`.
2. In the home directory of the flight computer when not in a mission the command to send files is `dockzr *.ma *.mi`. This will send all files ending with both extensions to the glider. One file can be sent by specifying the file (`dockzr sample90.ma`). When in a mission, all commands are began with an exclamation mark, so the command would be `!dockzr *.ma *.mi`.

3.1.2. Getting Files off the Simulator

To send files off the glider, you must be in the directory the file is within, then:

1. `zs [filename]`. This will send the specified file. Remember the use of wildcards (*), as they allow to batch send all `.sbd` files in one command, instead of one at a time.
2. Once sent, in the topmost right click the "from-glider" folder and download the files sent.

3.1.3. Logs, Scripts, Viewing Input Commands

Logs are on the lower right of the blue buttons. They are text files of the glider terminal's text and extremely useful for going back and checking what happened if you can't scroll up and find it. Even days later they can be downloaded and searched through using control-F: very handy!

Scripts are selected below the terminals enter-command box. The name of the selected script is on the left while the script state is on the right. Script states are gone over more in Chapter 4 and 5.



Figure 3.2: SFMC: Selecting scripts and viewing their current state

Below the script states is where to view recently submitted commands. If scripts are running uncheck "Show human user commands only".

Recent Commands Submitted				
<input type="checkbox"/> Show human user commands only				
Resend	Command		Submission Time	Submitter
	ver		2024-04-12 16:03:59	sscopazzi
	help ver		2024-04-12 16:03:47	sscopazzi
	help		2024-04-12 15:54:31	sscopazzi

Figure 3.3: SFMC: Viewing recently submitted commands

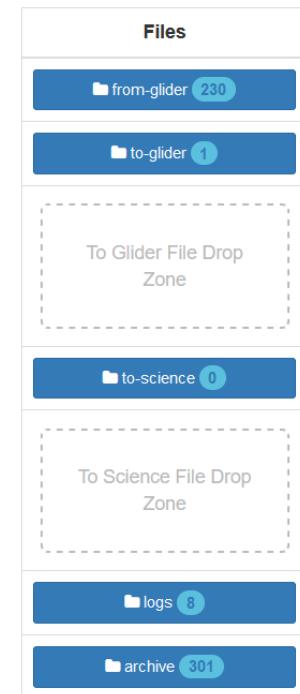


Figure 3.1: SFMC: Top right folders - to/from-glider, file drop zones, logs

3.2. Files Used in Test

All files for simulations in this manual are on GitHub. As the basic principle for the files are similar in all missions, I'll only explain the pertinent changed details in future chapters.

3.2.1. bs_cft.mi

```

1 ##### Onboard Processing Specific Parameters #####
2 sensor: u_allowable_cycle_overrun(msec) 3000 # how large x_cycle_overrun_in_ms can be
3 sensor: u_mission_param_a(nodim)          7      # z, initialize
4 sensor: u_reqd_depth_at_surface(m)         3      # in, depths less than this considered "at surface"
5 #
6 # - yo_199.ma      #
7 # - goto_199.ma    #
8 # - loadsim_cft.mi #
9 # GOTO_LIST : Waypoint list
10 behavior: goto_list
11   b_arg: args_from_file(enum)      99  # mafiles/goto_199.ma
12   b_arg: start_when(enum)        0    # 0-immediately
13 # YO : Flight params
14 behavior: yo
15   b_arg: args_from_file(enum)      99  # mafiles/yo99.ma
16   b_arg: start_when(enum)        2    # 2-depth idle
17   b_arg: end_action(enum)        2    # 2 resume
18 # SAMPLE : EXTCTL
19 behavior: sample
20   b_arg: args_from_file(enum)     90  # 90 v11 (89 v10.8)

```

File 3.1: Current Field Test: bs_cft.mi - mission file

For each behavior the args_from_file number calls the relevant file; i.e., 99 in the yo behavior calls yo99.ma in c:/mafiles on the flight computer. For this test I differentiated the goto and yo files from the norm of ending in 10 by ending them in 99.

3.2.2. proglets.dat

```

1 proglet = extctl
2     uart      = j5           # science port it's plugged in
3     bit       = 0           # power, battery, ability to turn the RasPi on/off
4     start_snsr = c_extctl_on(sec)

```

File 3.2: Current Field Test: proglets.dat - science computer's sensor file

Ensure the RasPi is plugged into the specified J port on the science computer (it must be J4 or J5).

3.2.3. extctl.ini

```

1                                     # INDEX NUMBER FOR RASPI PYTHON
2 u_mission_param_a                  # 0
3
4 os # OUTPUT SENSORS SECTION
5 sci_generic_a nodim               # 1
6
7 is # INPUT SENSORS SECTION
8 # Glider coords
9 m_present_time timestamp          # 2
10
11 # Specify the baudrate of the uart, don't technically have to specify but to not forget
12 baud
13 9600 # default value
14
15 packet_size # don't technically have to specify but to not forget
16 600 # default value

```

File 3.3: Current Field Test: extctl.ini - used to control m_depth

An extremely basic implementation of the external controller. It allows the control of one mission parameter, the commanded depth (`c_target_depth` in `yo99.ma`).

3.2.4. `tbdlist.dat`

```
1 # Basic Data
2 sci_m_present_time
```

File 3.4: Current Field Test: `tbdlist.dat` - science computer's data file

As there are no scientific sensors, the only line called in this file is `sci_m_present_time`. There are no numbers after it because this runs every glider cycle.

3.2.5. `simul.sim`

```
1 just_electronics
```

File 3.5: Current Field Test: `simul.sim` - must be on glider simulator for simulating

This file must be in the `c:/config` directory of the flight computer and needs only have the above line. This tells the simulator to purely use electronics; i.e., it's not an actual glider with a pump, etc.

3.2.6. `sbdlist.dat`

```
1 # INTERVAL           Seconds since last stored value, 0 is store every value
2 # STATE              State of glider (dive, hover, climb), 15 is always store
3 # HALFYOS            # of dive/climbs in this segment to record for, -1 store for all
4 # YO_DUTY_CYCLE      Store data every n'th yo in this segment, -1 is every dive/climb
5
6 #
7 # -----
8 # SENSOR NAME        INTERVAL   STATE     HALFYOS   YO_DUTY_CYCLE
9 # (Defaults)          0          15        -1         -1
10# -----
10m_tot_num_inflections 30         15        -1
11m_lat                  30         15        -1
12m_lon                  30         15        -1
13m_x_lmc                30         15        -1
14m_y_lmc                30         15        -1
15c_heading               30         15        -1
16m_heading               30         15        -1
17m_depth                30         15        -1
```

File 3.6: Current Field Test: `sbdlist.dat` - flight computer's data file

To have enough data to plot and thus see what transpired during the simulation over smaller timescales, for example if only running four yos to a depth of 40m for 30 min, we need data at a higher resolution. There are a lot of other values in the `sbdlist.dat`, but the above are the only ones changed. To cause less headaches later when processing and plotting the `.sbd` files put the interval and state to the same values.

3.2.7. `sample90.ma`

```
1 b_arg: sensor_type(enum)      90  # c_extctl_on
2 b_arg: state_to_sample(enum)  15    # all states
```

File 3.7: Current Field Test: `sample90.ma` - flight computer's sensor specific file

The big list of numbers in the file is only for reference, the number at the top (`sensor_type(enum)`) is what to change. This is where you change where it samples; i.e., when to turn the RasPi on and off. For this test it has been set to always be on so it doesn't turn off in the middle of the simulation.

3.2.8. yo99.ma

```
1 b_arg: d_target_depth(m)      1000000 # u_mission_param_a  
2 b_arg: c_target_depth(m)      5      ### visual flag ###
```

File 3.8: Current Field Test: yo99.ma - flight computer's control of dive and climbs (yos)

Nothing else in this file was changed from default values.

3.3. Local Mission Coordinates: Setting Location and Direction of Fast Current Zone

The fast zone is described by `s_water_speed_fast_zone_x_min/max`. The values input to these lines are Local Mission Coordinates (LMCs). They are set to zero upon mission start (`run bs_cft.mi`) and are x and y values with units of meters. Positive Y in the direction of magnetic north at mission start. Positive X 90 degrees to the right from positive Y. Having the LMCs 1:1 with latitude and longitude makes simulating an ocean environment and plotting easier (this is why `s_mag_var` is set to 0).

The glider always uses `xs_water_speed` as the simulated ocean current velocity. Thus, in the simulation file there need not be a `xs_water_speed` set to anything, only the `s_water_speed` because that's the base water speed in the sim. When `s_water_speed_fast` is enabled (not set to -1) it overwrites the `s_water_speed` based upon the `s_water_speed_fast_zone_x_min/max`.

3.3.1. loadsim_cft.mi

```

1 sensor: u_max_time_in_gliderdos(sec) -1.0 # disable run "sequence" after this much time in
     gliderdos without receiving a keystroke
2
3 # initial direction before it is changed using SFMC script
4 sensor: s_water_direction(rad) 0.00          # due north (direction current is going TO)
5
6 # to simulate a fast current between LMC coordinates
7 # fast zone start < measured x lmc coordinate < fast zone end
8 sensor: s_water_speed_fast(m/s) 1.5           # in, set to -1 to disable
9 sensor: s_water_speed_fast_zone_x_min(m) 200    # in
10 sensor: s_water_speed_fast_zone_x_max(m) 1500   # in
11
12 # current has a simulated location of higher speed between LMC coordinates. It looks like this:
13 #           |           |           |
14 #   slow   |           fast        |           slow
15 #           |           |           |
16 #   s_water_speed_fast_zone_x_min           |
17 #                           |           |
                           s_water_speed_fast_zone_x_max

```

File 3.9: Current Field Test: loadsim_cft.mi - setting simulated variables

This file is used to initialize the starting simulated glider and ocean environment by calling `loadmission loadsim_cft.mi` before running the mission. This file:

1. Sets the wind forcing to zero (do not want wind effecting the glider while on the surface)
2. The magnetic variation to zero (want LMC to equal latitude and longitude for easier plotting)
3. The ocean to be infinitely deep (1337m) with no bottom wavelength (don't want the glider to touch the bottom at varying depths; need consistency)
4. Initial glider location (this file sets it to east of the Yucatan Peninsula)
5. Sets the current field location using LMC (sets the *initial* current direction (in radians) to due north)

3.4. Three ways to make a goto file

Gliders use coordinates in longitude and latitude order ordered by [degrees][minutes].[decimal minutes]; i.e., -8440.0 2100.0. There are a few ways one can make a goto file:

3.4.1. Manually

Using Google Maps online, Google Earth Pro, OpenCPN, or other method to find GPS coordinates, manually create the goto file. If there are not many coordinates (1-4 or so) this is usually the easiest method. Because we have one waypoint, it's easy to make these goto files by hand.

3.4.2. kml2goto.py

Using Google Earth Pro, plan a route and export as a kml file, then use the .py. There are instructions in the .py file. Easier when there are many (5+) waypoints.

3.4.3. opencpn2goto.py

Using OpenCPN, plan a route, copy from the route planner into Excel, export as a .csv, then use the .py. There are instructions in the .py file. Easier when there are many (5+) waypoints.

3.4.4. goto_l99.ma

```

1 bbehavior_name=goto_list
2 # --- goto_l99.ma
3 <start:b_arg>
4   b_arg: num_legs_to_run(nodim)    -1      # loop through waypoints
5   b_arg: start_when(enum)          0       # BAW_IMMEDIATELY
6   b_arg: list_stop_when(enum)      7       # BAW_WHEN_WPT_DIST
7     b_arg: initial_wpt(enum)      0       # 0 to n-1, -1 first after last, -2 closest
8     b_arg: initial_wpt(enum)      0       # first (only) wpt in list
9     b_arg: num_waypoints(nodim)    1       # number of waypoints in list
10 <end:b_arg>
11
12 <start:waypoints>
13 #      LON           LAT           name
14 #    -8630.0        2100.0        | location of simulation start
15 #    -2522.0        2100.0        | far far away (westernmost point of Africa)
16
17 <end:waypoints>
18 # head due east of Yucatan

```

File 3.10: Current Field Test: goto_l99.ma - commands the glider to go to a waypoint

WPT Due East (right) shows the location of the waypoint.

The practice of putting a waypoint far away while crossing a location of strong currents is a standard practice for glider pilots. Thus, the waypoint is placed far away from the glider because we want the heading to remain as close as possible to 090 degrees while the glider is deflected northward when traversing the current field.

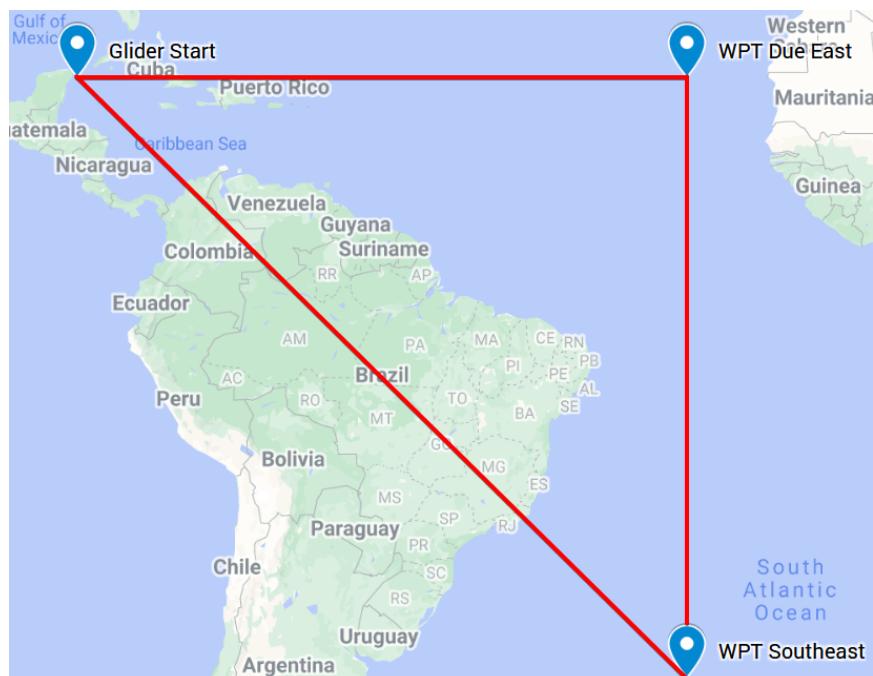


Figure 3.4: Waypoint Locations

3.5. RasPi script: depth_print.py

```

1 import serial
2
3 uart_baud = 9600           # Desired baudrate for uart comms
4 send_chunk_size = 600       # amount of bytes per chunk when writing file to progl
5 port_address = '/dev/ttyUSB0' # Define comms port address
6
7 class extctl(object):
8     def __init__(self, tty):
9         self.port = serial.Serial(tty, baudrate=uart_baud, xonxoff=False, rtscts=False,
10             dsrdr=False, stopbits=serial.STOPBITS_ONE, parity=serial.PARITY_NONE,
11             bytesize=serial.EIGHTBITS, timeout=None)
12
13     def send(self, s):
14         csum = 0
15         # bs = s.encode('ascii')
16         bs = s
17         for c in bs:
18             csum ^= c
19         nmea = b'$' + bs + (b'*%02X\r\n' % csum)
20         print(nmea)
21         self.port.write(nmea)
22
23     def write_SW(self, index, value):
24         self.send(b'SW,%d:%f' % (index, value))
25
26     def write_TXT(self, value):
27         self.send(b'TXT,%f' % (value))
28
29 ######
30
31 serial.Serial(port_address, baudrate=uart_baud).close()
32 x = extctl(port_address)
33 port = serial.Serial(port_address, baudrate=uart_baud)
34
35 x.write_SW(0, 42)    # update mission_param_a one time
36 x.write_TXT(42)      # print value to glider terminal
37
38 print('d_target_depth updated, message sent')

```

File 3.11: Current Field Test: Raspberry Pi Script to change d_target_depth one time

3.6. Starting the Simulated Mission

Once the RasPi is hooked up and all the files are on the simulator we can begin the simulation! To do so, in the main directory of the flight computer type:

1. loadmission loadsim_cft.mi: This will initialize the simulated ocean variables
2. report ++ m_depth: Displays m_depth, if this goes over 7m u_mission_param_a was updated
3. run bs_cft.mi: Run the mission
4. Once the glider starts diving, run the Python script to update the depth and print the value to the glider terminal

To run the script on the RasPi:

1. Open the RasPi's terminal and navigate to the folder containing the .py script using cd [folder] (following the manual's example, in the backseat_driver folder)
2. Once the glider has started diving, sudo python3 depth_print.py which will run the script. You should see the script's final print message in the terminal.

Once the script has been run, inspect the glider terminal for the message 42, and watch to see if m_depth goes > 7. If it does, control-C to cancel the mission. Restart the mission as before with the simulator and the RasPi, but don't use the report command. Once you update u_mission_param_a as before, let the simulation run for approximately 2.5 hours.

3.7. Plotting results using Python (current_field_test_simulation.ipynb/.py)

Before plotting we need to get the data files off the glider. This can be done a few ways (taking the simulator apart to get the CF card), but I recommend the following.

3.7.1. Getting the .sbd & .tbd files from the simulator

1. From the glider terminal change to the c:/logs folder using the command cd logs to get to the c:/logs directory.
2. dir to display all files in the directory.
3. The files at the bottom will have the most recent date. Inspect the ending number (for example, 2870012.sbd)
4. zs 28700*.sbd uses a wildcard to send all the files from the mission.

3.7.2. cac Files

cac files are located in c:/state/cache on both the flight and science computer (shown right). In basic terms, these files hold the keys for processing the .sbd and .tbd files (the cac files to process the .tbd files are in the same folder on the science computer).

3.7.3. Plots

Left: The blue line is the path of the glider through the current field boundaries, denoted by the vertical red dashed lines.

Right: The yo profile of the glider (blue line), with the max depth denoted by a red dashed line.

Python code to create these plots, along with some others not known, is on [Sophie's GitHub](#).

```
Directory of c:/state/cache/
```

```
./
../
06a36d4e.cac 116841 2024-01-22 21:46:10
aad86c9b.cac 116841 2024-01-22 21:46:10
b3d61675.cac 116841 2024-01-22 21:46:12
b9bd3e49.cac 118098 2024-01-24 19:41:10
65acaf6c.cac 118098 2024-01-24 19:41:12
9b83f4ec.cac 118098 2024-01-24 19:41:12
bb978bca.cac 116841 2024-01-28 23:49:42
5a250d8f.cac 118098 2024-01-29 00:02:38
12ed5f92.cac 118098 2024-02-21 00:00:00
11 files
```

Figure 3.5: Current Field Test: Cac files as seen from the glider terminal

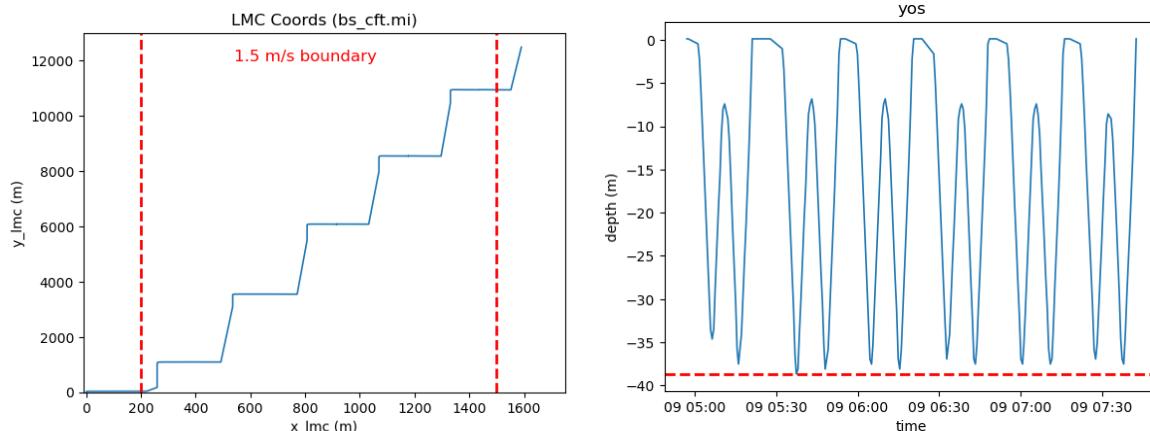


Figure 3.6: Current Field Test: Output plots, viewing by LMC coordinates

4

Simulating a Large-Scale Ocean Current using an SFMC Script

Chapter Concepts:

1. An explanation of SFMC scripts and their typical use-cases.
2. Changing the sensors in the surface dialog.
3. Using the Glider Guidance System to find depth-averaged currents, then turning that information into an SFMC script using Python to build the .xml file.

4.1. Overview of SFMC Scripts

SFMC scripts allow for automated control of gliders while on the surface by looking for exact phrases from the gliders surface output and sending a varying list of commands based on user configuration. This is typically done to automate the retrieval of .sbd and .tbd files and to send .ma files then re-read them before resuming the mission. In our case, however, we will be using this ability to send commands to the surfaced glider mid-mission. There are five types of commands we utilize in our .xml.

```
-<gliderScript>
  <!-- The start state of the script. -->
  -<initialState name="checkNoComms">
    -<transitions>
      -<!--
        Check what glider surfaced for. If no_comms, send "ctrl-R"
      -->
      <transition matchExpression="Because:no comms for a while" toState="sendWaterSpeedDir"></transition>
      <!-- Happens during testing so put here-->
      <transition matchExpression="Because:nothing commanded" toState="sendWaterSpeedDir"></transition>
    -</transitions>
  -</initialState>
-<!--
  From checkNoComms. This sends updated current direction before sendResume. This state "simulates" a current field
  by matching the Longitude from surface dialogs, then !put s_water_direction and s_water_speed to what it should be for
  the location of the glider before diving. It only reads the longitude part of the surface dialog, from following part
  in quotes: sensor:m_Lon(Lon)='-8630'

  -->
  -<state name="sendWaterSpeedDir">
    -<transitions>
      -<transition matchExpression="=-8630" toState="sendResume">
        <action type="glider" command="!put s_water_direction 0.7853981633974483"></action>
        <action type="glider" command="!put s_water_speed 0.6"></action>
      -</transition>
```

Figure 4.1: SFMC Script Example: initialState, matchExpression, toState, state, action

4.1.1. Breakdown of How a Script Works: sendWaterSpeedDir.xml

Using Figure 4.1 (above) as the example:

1. **initialState name ="checkNoComms"**: When selected in SFMC the script starts in this state, then waits for a glider surfacing to be detected. When a glider is detected it matches words in the surface dialog, then;
2. **matchExpression="Because: no comms for a while"**: From the initialState we match an expression (words and/or numbers from the surface dialog), then;
3. **toState="sendWaterSpeedDir"**: Send the script to another state named "sendWaterSpeedDir".
4. **state name="sendWaterSpeedDir"**: Once called, this state containing commands to send to the glider by first matching an expression (in this case -8630, longitude without a decimal), then;
5. **command="!put s_water_direction 0.78"**: The command send to the glider. For example, the command "Ctrl-R" would send the resume mission command.

In this way, any value of longitude can have any value for s_water_speed and s_water_direction within the sendWaterSpeedDir state.

4.1.2. Changing the Surface Dialog: Adding Sensors to matchExpression With

In these script examples, the script reads the phrase calculations COMPLETE, but if you desire the script to read a sensor name and/or value (like m_lon), it's possible to put a sensor in the surface dialog. View possible commands in the gliderDOS terminal by typing help. From there, we can find srf_display, which is how we control what sensors are displayed in the surface dialog. If we send help srf_display we get the following:

```
SRF_DISPLAY + <sensor_name ..> | - <sensor_name ..> | clearall | list ; config (or list) which
sensors to display in surface dialog
```

For example, srf_display + c_heading will show that sensor in the surface dialog. We can verify that it is in the surface dialog by sending srf_display list.

```
GliderDos I -3 >help

? ballast ballvalve boot
callback capture CAT CD
CHDIR CLRDEVERRS compass_cal consci
COPY core CP date
DEL DELLOG DEVICES? DF
digifin DIR drift_table echo
EXECUTE exit freport GET
glider_test HARDWARE? HEAP HELP
hs LAB_MODE LIST loadmission
logging LONGTERM LONGTERM_PUT LS
MBD MKDIR MV PRUNEDISK
PURGELOGS PUT RENAME REPORT
RM RMDIR run SBD
SEND sequence SETDEVLIMIT SETNUMWARN
SIMUL? SRF_DISPLAY strobe sync_time
SZR SZS talk tvalve
TYPE USE VER WHERE
whoami whoru WHY? wiggle
ZERO_OCEAN_PRESSUREZR ZS

    help list all commands in alphabetical order
    help -full list all commands with their usage and descriptions
    help -desc list all commands with their descriptions
    help -usage list all commands with their usage
    help <cmd> ... <cmd> prints the full help msgs for listed commands.
    | Note that -desc and -usage can also be used
    | command names in lower case are NOT executable in mission via !
```

Figure 4.2: SFMC Script: help command on glider

4.2. Glider Simulator Limitations

1. The simulation only runs in real-time, meaning there is no way to "speed" up the simulation to make it faster. This means if you're simulating a 200km ocean crossing it will take approximately five days. The simulator has been designed to operate real glider parts on a glider (was not originally made with the blue shoebox simulators in mind).
2. Ocean and atmosphere parameters are all static (with one exception). They are set once and stay that way all mission. Wind speed and direction, water depth, and current direction are all static. The one exception is water speed (as explained in Chapter 3, there is a built in method to make a static fast zone of current), but that solution does not allow for a variable speed and direction like an ocean current.

The glider simulators simulated fast current zone from Chapter 3 could be thought of as the spiritual predecessor to the following method of simulating an ocean current with a glider simulator. In order to: (1) achieve a (*mostly*) realistic current field similar to the Yucatan Current and (2) ensure the external controller's Python script correctly updated the heading based upon the measured current field we use an SFMC script to change the current direction based upon the longitude of the glider. This is achieved by the SFMC script as previously explained.

Besides the single-time initialization of current magnitude and direction, which we overcome here with the SFMC .xml script, the remaining limitation of the simulated ocean is the uniform current velocity with depth; however, for the purposes of the external controller's Python program that uses current vectors this is acceptable because near real-time depth-averaged current (DAC) data from ocean models is readily available. Thus, by using the DAC for 0-1000m we retain a close enough approximate for our purposes.

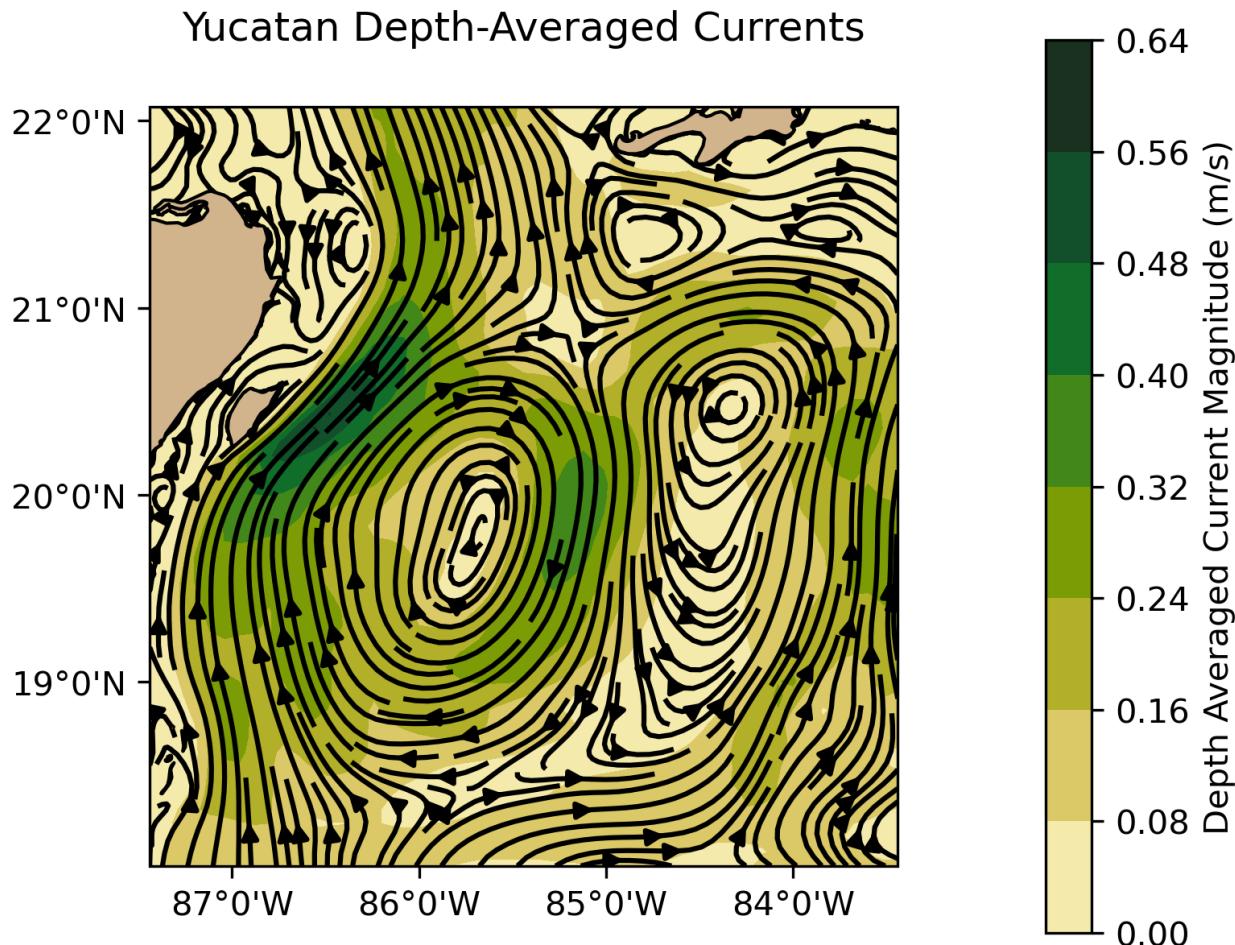
4.3. Plotting Depth-Averaged Currents Using GGS: Yucatan Channel Example

The Glider Guidance System (GGS) made by Sal Fricano has many uses. One of them is pulling data from the RTOFS (Global Real-Time Ocean Forecast System) ocean model, calculating the DAC, then creating a stream plot based on coordinate inputs. For information on how to use the GGS please see Sal Fricano's GitHub. Using (an early version of) GGS for depth-averaged current data for the Yucatan Current the following table was made by inspecting current speed and directions in a grid pattern:

Latitude	Direction °T / Magnitude m/s							
	000	340	350	355	350	000	000	000
21.50°	000	340	350	355	350	000	000	000
	0.0	0.1	0.2	0.3	0.1	0.1	0.1	0.1
21.25°	000	020	020	020	020	000	310	
	0.0	0.0	0.2	0.3	0.1	0.1	0.1	
21.00°	000	045	045	045	045	0	270	
	0.0	0.3	0.3	0.3	0.1	0.0	0.1	
20.75°	000	045	045	060	090	130	250	
	0.0	0.5	0.4	0.3	0.1	0.0	0.1	
20.50°	045	045	050	085	100	145	170	
	0.6	0.6	0.4	0.3	0.3	0.3	0.3	
Longitude	86.5°	86.25°	86.00°	85.75°	85.50°	85.25°	85.00°	

Table 4.1: Yucatan Depth-Averaged Currents: Current magnitude and direction in table form

Then, as the SFMC .xml script **only incorporates longitude**, I approximated a cross-section of longitude (a change in latitude wasn't required to verify proper behavior). Using this table as reference, the SFMC .xml script was made.



Generated by the Glider Guidance System (GGS) on 11-19-2023 at 16:21 UTC

Figure 4.3: Yucatan Depth-Averaged Currents: An early output from the Glider Guidance System

4.4. Creating the SFMC Script to Control Simulated Variables (sendWaterSpeedDir.xml)

As discussed, the glider sets the simulated ocean variables (`s_water_speed`, `s_water_direction`) once at mission start. To work around this we the SFMC script to read the glider's longitude from the surface dialog, then have the script change the ocean variables during the surfacing before commanding the glider to resume the mission.

The .xml script reads the expression "Calculations COMPLETE" instead of the typical Because: no comms for a while or Because: timeout expired because the simulator is always connected to SFMC. We also want to update and resume only after the glider has finished its water velocity calculations. The by-product of this method is a more accurate representation of time at the surface. If, for example, we used the timeout expired expression the glider would resume the mission and dive immediately upon surfacing, which would not: (1) give time to calculate the water velocity and not; (2) more accurately representing the sending of files when surfacing during a mission.

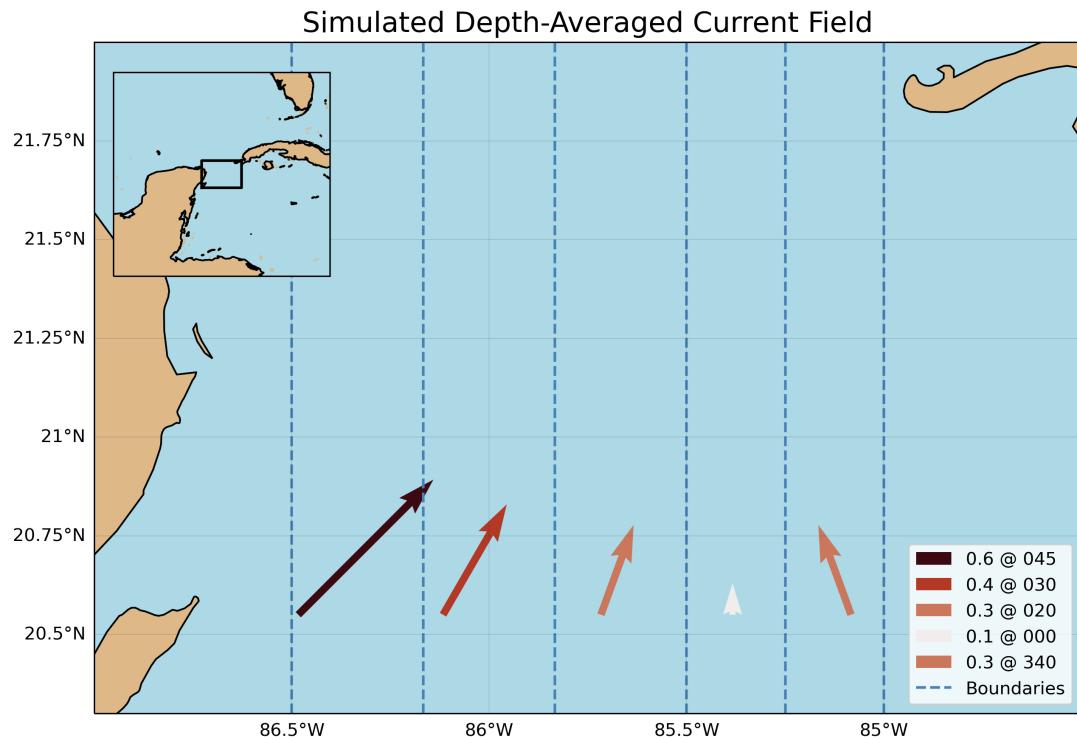


Figure 4.4: SFMC Script: Blue dashed lines delineate the boundaries between changing currents, with the arrow in each vertical slice of longitude indicating the magnitude and direction of currents within.

The current field only changes based upon the longitude. If one desired a more accurate current simulation in a glider simulator the code could be adapted to incorporate latitude as well. An attempt at this, that may help you get started, is located in the GitHub's files under this chapter called "attempt_at_incorporating_latitude.ipynb".

[this is empty space]

4.4.1. SFMC Script Part One: Building the simulated ocean current field

```

1 # IMPORTS
2
3 start_lon = -86.5
4 end_lon = -85
5 filename = 'znew6things_sendWaterSpeedDir.xml'
6 wheresave = '../f_SFMC_script/'
7 complete_filename = os.path.join(wheresave,filename)
8
9 # how wide to make the sections across the ocean area
10 # sections, 20, 20, 20, 15, 16
11 sec0 = 20
12 sec1 = 20
13 sec2 = 20
14 sec3 = 15
15 sec4 = 16
16
17 # location of sections
18 pltsec0 = 0
19 pltsec1 = sec0+sec1-1
20 pltsec2 = sec0+sec1+sec2-1
21 pltsec3 = sec0+sec1+sec2+sec3-1
22 pltsec4 = sec0+sec1+sec2+sec3+sec4-1
23
24 # for current direction, make length of each section
25 # if not all good lengths get lots of errors
26 west0 = [45]*sec0
27 west1 = [30]*sec1
28 west2 = [20]*sec2
29 west3 = [0]*sec3
30 west4 = [340]*sec4
31
32 # put all the things above into one
33 water_dir_deg = west0+west1+west2+west3+west4
34
35 # convert each degree into radians, needed for plots
36 rads = []
37 for i in water_dir_deg:
38     rads_loop = math.radians(i)
39     rads.append(rads_loop)
40
41 # for current speed, make length of each section
42 speed0 = [0.6]*sec0
43 speed1 = [0.4]*sec1
44 speed2 = [0.3]*sec2
45 speed3 = [0.1]*sec3
46 speed4 = [0.3]*sec4
47
48 # put all above into one
49 water_speed = speed0 + speed1 + speed2 + speed3 + speed4
50
51 # convert the start and end lon to proper Python mappable
52 lon_range_str,lon_map = decimal_degrees_to_degrees_minutes(start_lon,end_lon)
53 loopnum = len(lon_range_str) # for generating the .xml script, how many values there are (91 in
      this case)
54 lat_map = [20.55]*len(lon_map) # for plotting, but must be the same length as lon_map
55
56 # if these things aren't equal it's bad
57 print(f'Length of things should be equal: {len(lon_map)} = {len(lon_range_str)} = {len(rads)} = {len(water_speed)} = {len(lat_map)}')

```

File 4.1: SFMC Script: Building the simulated ocean current field

This part programmatically builds each section or vertical slice of longitude as seen in Figure 4.4 to have an associated magnitude and direction. We will use this same code chunk later to build the SFMC .xml script (that controls the simulator mid-mission) and to plot the glider locations to understand what happened during the simulation.

4.4.2. SFMC Script Part Two: Defining .xml script states

```

1 header = (
2     '<gliderScript>\n\n'
3 )
4
5 checkNoComms = (
6     '    <!-- The start state of the script. -->\n'
7     '    <initialState name="waterCalcComplete">\n'
8     '        <transitions>\n'
9     '#         <!-- start_when(enum) 9 says this at the top of surface. Send to to
10    #             sendWaterSpeedDir-->\n'
11    #             <transition matchExpression="Because:timeout expired" toState="
12    #                 sendWaterSpeedDir">\n'
13    #             </transition>\n'
14    #             <!-- Check what glider surfaced for. If no_comms, send to to
15    #                 sendWaterSpeedDir"-->\n'
16    #             <transition matchExpression="Because:no comms for a while" toState="
17    #                 sendWaterSpeedDir">\n'
18    #             </transition>\n'
19    #             <!-- Happens during testing so put here. Send to to sendWaterSpeedDir -->\n'
20    #             <transition matchExpression="Because:nothing commanded" toState="
21    #                 sendWaterSpeedDir">\n'
22    #             </transition>\n'
23    '        <!-- Must wait for glider to calculate water velocity! Send to to
24    #                 sendWaterSpeedDir -->\n'
25    '        <transition matchExpression="COMPLETE" toState="sendWaterSpeedDir">\n'
26    '        </transition>\n'
27    '    </transitions>\n'
28    '    </initialState>\n\n')
29
30 sendWaterSpeedDirHeader = (
31     '<!-- From checkNoComms. This sends updated current direction before sendResume. This state
32     "simulates" a current field\n'
33     '    by matching the longitude from surface dialogs, then !put s_water_direction and
34     s_water_speed to what it should be for\n'
35     '    the location of the glider before diving. It only reads the longitude part of the
36     surface dialog, from following part\n'
37     '    in quotes: sensor:m_lon(lon)=-8630\n'
38     '-->\n'
39     '    <state name="sendWaterSpeedDir">\n'
40     '        <transitions>\n'
41
42 footer = (
43     '<!-- SEND RESUME/DIVE ===== -->\n'
44     '    <state name="sendResume">\n'
45     '        <transitions>\n'
46     '            <transition matchExpression="xxx command verify fail xxx" toState="sendResume
47     ">\n'
48     '            </transition>\n'
49     '            <!-- Ready to dive, but send report first! MUST SEND TWO CTR-R'S OR ELSE IT
50     DOESN'T WORK\n'
51     '            Script needs more time after the !put so sends two -->\n'
52     '            <transition matchExpression="Hit Control-R to RESUME" toState="
53     waterCalcComplete">\n'
54     '                <action type="glider" command="Ctrl-R">\n'
55     '                </action>\n'
56     '                <action type="glider" command="Ctrl-R">\n'
57     '                </action>\n'
58     '            </transition>\n'
59     '            <transition timeout="10" toState="waterCalcComplete">\n'
60     '            </transition>\n'
61     '        </transitions>\n'
62     '    </state>\n'
63     '\n'
64     '    <finalState name="final">\n'
65     '    </finalState>\n'
66     '</gliderScript>\n')

```

File 4.2: SFMC Script: Defining .xml script states

4.4.3. SFMC Script Part Three: Building the .xml script

```

1 file = open(complete_filename, 'w')
2 file.write(header)
3 file.write(checkNoComms)
4 file.write(sendWaterSpeedDirHeader)
5 for zz in range(0, loopnum):
6     # file.write(f'          <transition matchExpression="    sensor:m_lon(lon)={lon_range_str[zz]}>" toState="sendResume">\n')
7     file.write(f'          <transition matchExpression="={lon_range_str[zz]}>" toState="'
8     file.write('            sendResume">\n')
9     file.write('              <action type="glider" command="!put s_water_direction {rads[zz'
10    file.write('                ]}">\n')
11    file.write('              </action>\n')
12    file.write('              <action type="glider" command="!put s_water_speed {water_speed['
13    file.write('                zz]}">\n')
14    file.write('              </action>\n')
15    file.write('              <action type="glider" command="!put X_PRIOR_SEG_WATER_VX 0">\n')
16    file.write('              </action>\n')
17    file.write('              <action type="glider" command="!put X_PRIOR_SEG_WATER_VY 0">\n')
18    file.write('              </action>\n')
19    file.write('              <action type="glider" command="!put M_WATER_VY 0">\n')
20    file.write('              </action>\n')
21    file.write('            </transition>\n')
22 file.write('          </transitions>\n')
23 file.write('\n')
24 file.write(footer)

```

File 4.3: SFMC Script: Building the .xml script

Within the python notebook used to make and export the .xml script used, there are diagnostic plots to visualize the simulated ocean and location of gliders. This would help in creating a simulated current in a new geographic location.

4.4.4. Alternate Current Field

To test in principle if another current field also works an alternate was created and tested. Further details on GitHub: "b_alternate_current_field_simpleGyre"

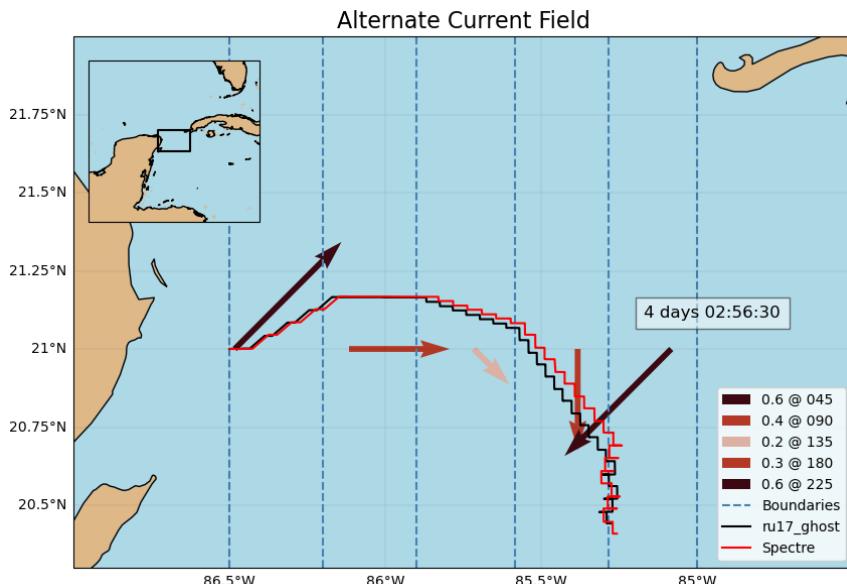


Figure 4.5: SFMC Script Alternate Current Field: Blue dashed lines delineate the boundaries between changing currents, with the arrow in each vertical slice of longitude indicating the magnitude and direction of currents within.

5

Verify .xml Script Behavior

Chapter Concepts:

In this mission we will use the goto_list behavior of the glider and:

1. Not use the external controller (RasPi)
2. Fly the simulation glider through our SFMC current-created field.

Before testing the external controller script, we must verify proper SFMC .xml script behavior. This also allows us to see what the glider will do when given waypoints instead of controlling the heading from the external controller.

5.1. Files Used

The rest of the glider files remain extremely similar to what was gone over in the current field test, so please see the GitHub for mission files.

5.1.1. goto_l98/97.ma

```
1 # goto_198.ma    # filename
2 <start:waypoints>
3 #      LON      LAT      name
4 # -8630.0    2100.0    | location of simulation start
5 -2522.0    2100.0    | far far away (westernmost point of Africa)
6
7 # goto_197.ma    # filename
8 <start:waypoints>
9 #      LON      LAT      name
10 # -8630.0   2100.0    | location of simulation start
11 -2522.0   -3612.0    | to give the initial heading hopefully 45deg more
```

File 5.1: Verify .xml Script Behavior: goto_l98/97.ma - waypoints

These waypoints were selected following the standard practice for glider piloting when crossing strong currents (Figure 5.2). By putting a waypoint far away in the desired direction of travel a nearly-consistent heading is maintained (Figure 5.3). Thus, I picked WPT Due East by continuing on the parallel of latitude, picking the westernmost point of Africa for longitude at 21N. For WPT Southeast I used navigational triangles and Google Maps to approximate a 45 degree angle and thus an initial heading of approximately 135T.

5.2. Uploading and Selecting the SFMC .xml Script

Now that we have our .xml script, we must put it in SFMC and activate it in the glider terminal.

5.2.1. Upload .xml script to SFMC

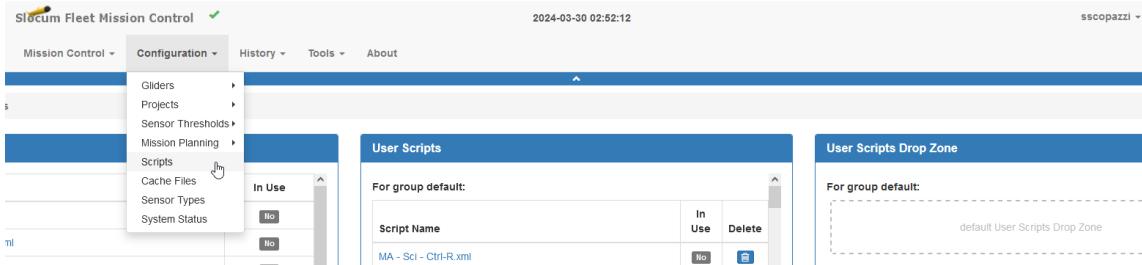


Figure 5.1: SFMC Script: Where to upload scripts in SFMC

Under "Configuration" select "Scripts". From there, drop the file in the "default User Scripts Drop Zone" on the far right. If you have many scripts (like we do at Rutgers) used for day-to-day piloting, I recommend using a prefix of "z_sim_" so the simulation scripts are only at the bottom and they can't be mistaken for anything else.

5.2.2. Select .xml script (sendWaterSpeedDir.xml)

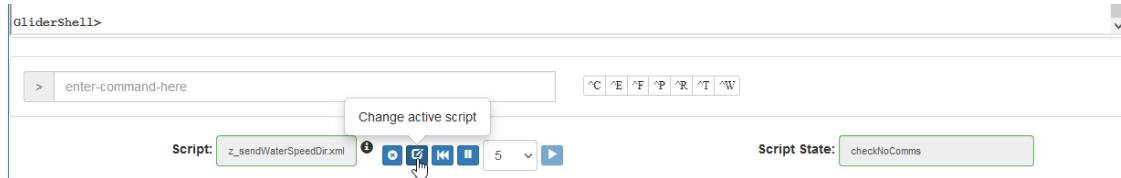


Figure 5.2: SFMC Script: Where to select scripts in SFMC glider terminal

In the glider's terminal, immediately below the enter command box, select "Change active script" and select the .xml script. If the script was made correctly, the name of the script will show up on the left side box, and the "Script State" will populate with checkNoComms. If after selecting the script nothing shows up in these boxes it means something is wrong with the formatting of the .xml file. There is a small bug in SFMC that causes an error message to not appear (through communication with Teledyne they now know this is a bug), but it's safe to assume if the box doesn't turn green with the name of the script, it has a formatting error.

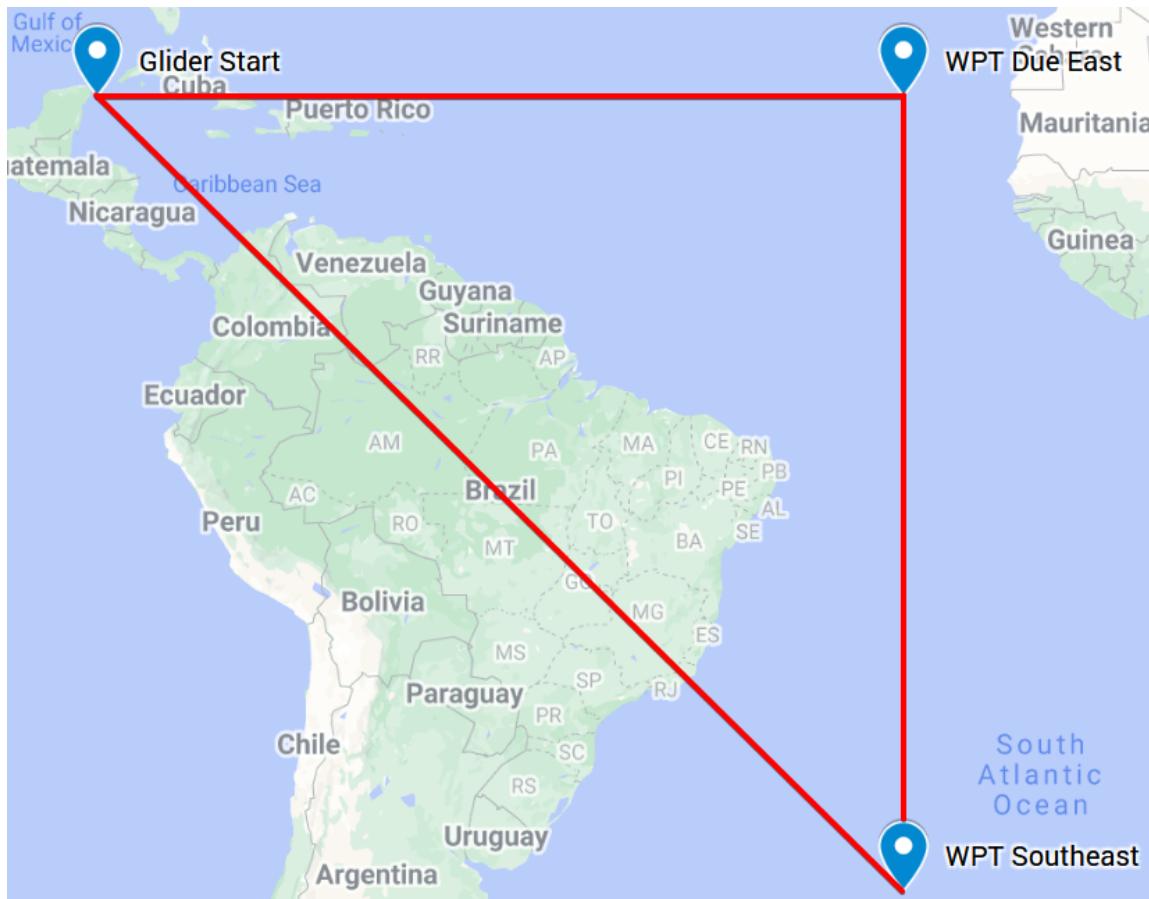


Figure 5.3: SFMC Script: Location of waypoints used for both missions.

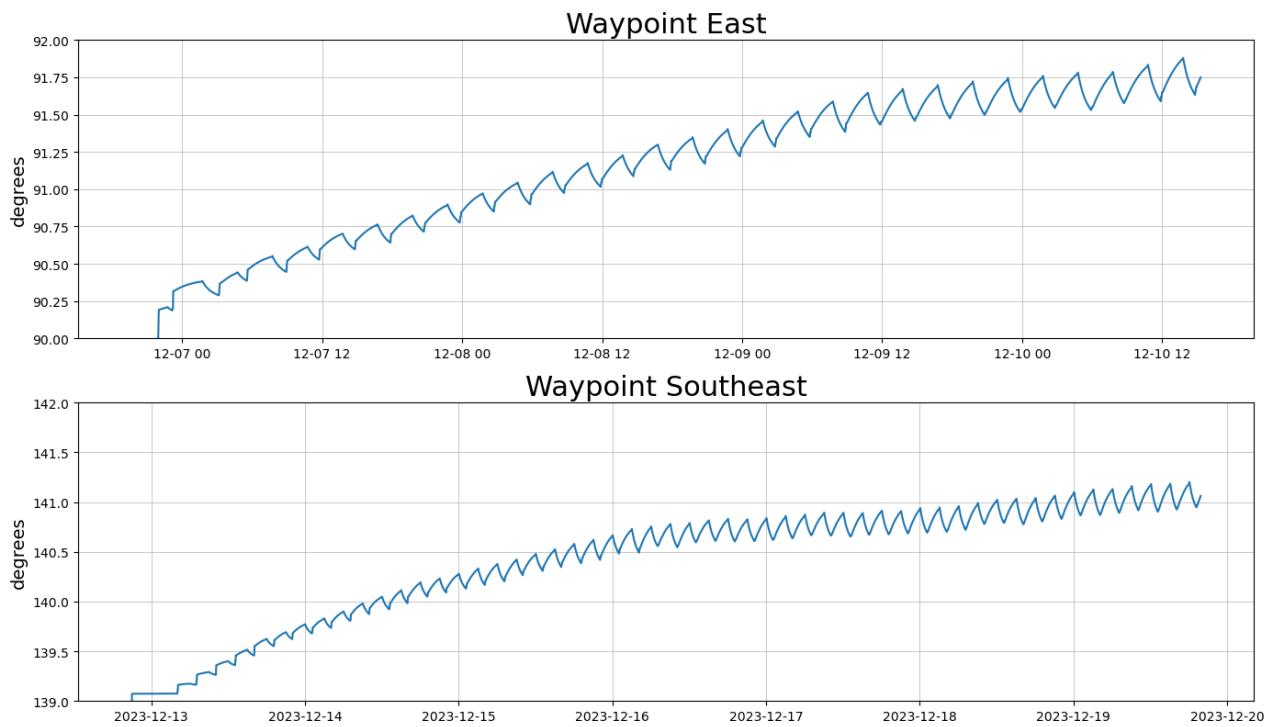


Figure 5.4: SFMC Script: `c_heading` comparison between both missions

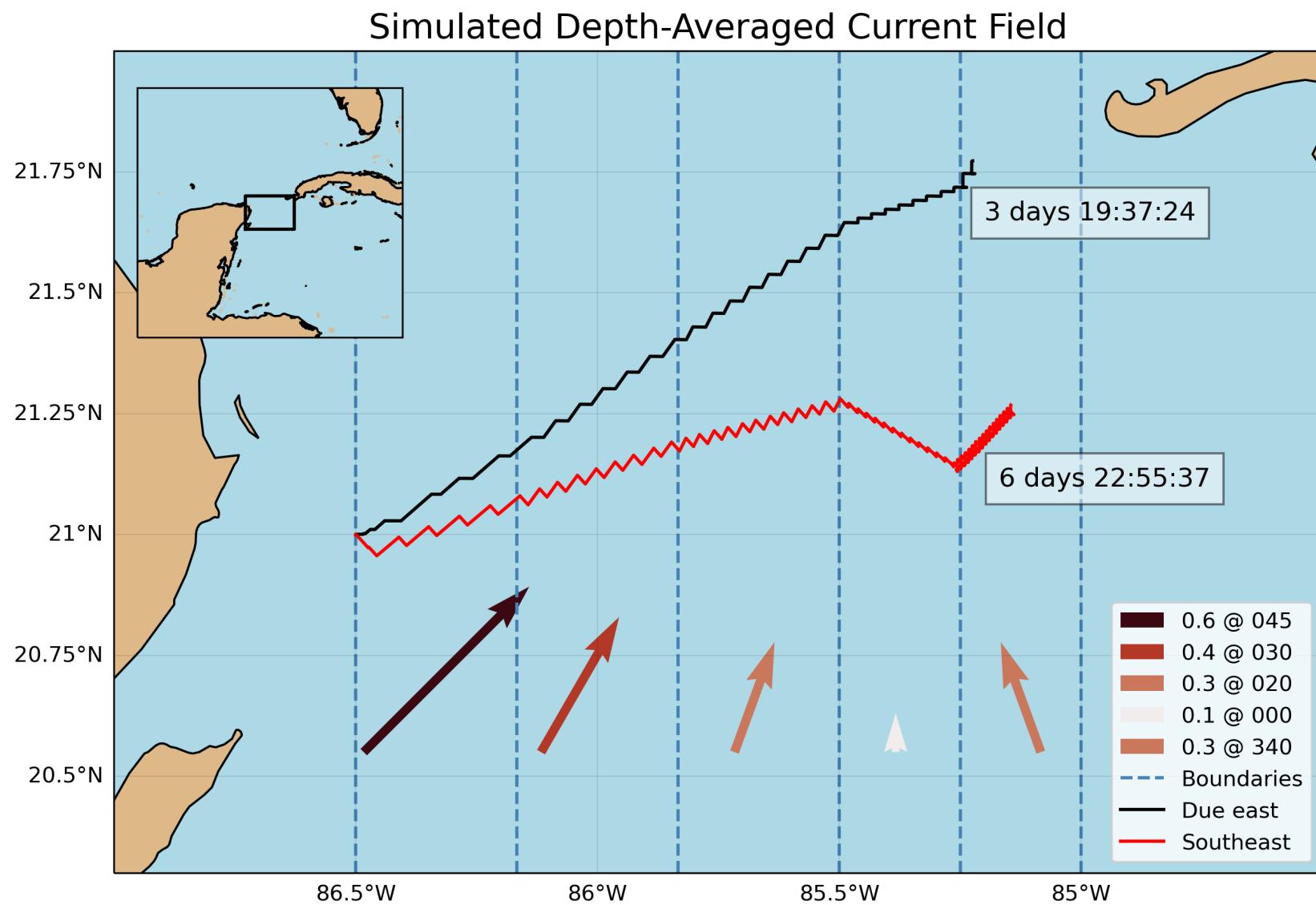


Figure 5.5: SFMC Script: Blue dashed vertical lines are longitude current boundaries. Arrows are current direction and magnitude within each boundary. See figure 5.2 for waypoint locations.

6

Simple Backseat Driver Behavior Application: Commanding c_heading

Chapter Concepts: In this mission we will use the set_heading behavior of the glider and:

1. Have the RasPi remain powered on for the duration of the mission.
2. Have the RasPi receive `m_water_vel_dir` from the glider.
3. Calculate the new heading 90° to the right of `m_water_vel_dir`.
4. Send the glider a new heading while also printing the new heading to the glider terminal for easy checking of proper behavior in the SFMC glider terminal (good for checking away from the lab, overnight).

6.1. Flying 90° From Depth-Averaged Currents

6.1.1. extctl.ini

Strictly speaking we do not need to define a `baud_rate` or `packet_size` as it defaults to 9600 and 600 respectively, but I left them in for visual clarity.

```
1 # c_heading in set_he95.ma, used with b_arg: heading_value(X) 1000000
2                                # INDEX NUMBER FOR RASPI PYTHON
3 u_mission_param_a          # 0
4
5 os # OUTPUT SENSORS SECTION
6 sci_generic_a nodim        # 1
7
8 is # INPUT SENSORS SECTION
9 # Glider coords
10 m_present_time timestamp   # 2
11 m_water_vel_dir rad       # 3
12
13 # Specify the baudrate of the uart, don't technically have to specify but to not forget
14 baud
15 9600 # default value
16
17 packet_size # don't technically have to specify but to not forget
18 600 # default value
```

File 6.1: Simple Backseat Driver: extctl.ini - used to control c_heading

6.1.2. set_he95.ma

The required .ma file for the set_heading behavior.

```

1 behavior_name=set_heading
2 <start:b_arg>
3     b_arg: use_heading(bool)           1
4     b_arg: heading_value(X)      1000000 # u_mission_param_a
5 <end:b_arg>
```

File 6.2: Simple Backseat Driver: set_he95.ma file for set_heading behavior

6.1.3. right.mi

Instead of goto behavior, we use the set_heading behavior so the RasPi can update the glider's heading.

```

1 sensor: u_allowable_cycle_overrun(msec) 3000      # how large x_cycle_overrun_in_ms can be
2 sensor: u_mission_param_a(nodim)        1.570796    # 90deg = radians. Must init something
3 sensor: u_extctl_debug(nodim)          1            # Received messages from RasPi displayed
4                                     only
5                                     # helps w/ debug and knowing if it's working
6
6 # HEADING: Command heading
7 behavior: set_heading
8     b_arg: args_from_file(enum) 95 # >= 0 enables reading from mafiles/set_he<N>.ma
9     b_arg: start_when(enum) 0      # Options: 0,1,2,6,7,8,15
10    b_arg: end_action(enum) 2     # 0 = quit, 2 = resume
```

File 6.3: Simple Backseat Driver: right.mi - mission to fly 90° from ocean current

6.2. How the Raspberry Pi reads the Glider's Message

The RasPi progressively splits the incoming cur_line from the glider using delimiters. Following each *.split command the output is shown.

```

1 cur_line = '$SW,0:0.12,1:508.0*3B$SW,0:0.24,1:520.0*3B'
2     # incoming line from the science computer to the RasPi
3 msg = cur_line.split('*')
4     # output: msg = ['$SW,0:0.12,1:508.0', '3B$SW,0:0.24,1:520.0', '3B']
5     # has been broken into two separate messages, or however many there are
6     # pulls the first, index 0 (I assume because this is the most recent?)
7
8 fields = msg[0].split(',')
9     # output: fields = ['$SW', '0:0.12', '1:508.0']
10    # further splits the message by commas
11
12 for subscription in fields:
13     # Loops through for each item in the fields variable
14     # subscription because this is what glider value's we asked for in
15     # the is section of the extctl.ini file
16     if '0:' in subscription:
17         # If a field starts with ':0' it does the following
18         subs = subscription.split(':')
19             # output: subs = ['0', '0.12']
20             # Splits the field's value that starts with '0:' into two parts
21
22         first_value = float(subs[1])
23             # put the first value into a variable, type float
24             # this is where I do the heading math, then write new heading to glider
25         print(f'First_Value:{first_value}')
26             # print value to RasPi terminal
27
28     if '1:' in subscription:
29         # does the same, but for '1:'
30         subs = subscription.split(':')
31         second_value = float(subs[1])
32         print(f'Second_Value:{second_value}')
```

If no value (in this example m_water_vel_dir) is found the script ends. To see how to wait until all values are received see the advanced version in Chapter 10.

6.2.1. Simple Application: Raspberry Pi Script

Class extctl is from Teledyne's manual, although here we define the baud, port_address and port at the top of the script initializing the values once.

```

1 import serial
2 import numpy as np
3
4 # Desired baudrate for uart comms
5 baud = 9600
6 # amount of bytes per chunk when writing file to proglert
7 send_chunk_size = 600
8
9 # Define comms port address
10 port_address = '/dev/ttyUSB0'
11 port = serial.Serial(port_address, baudrate=baud)
12
13 class extctl(object):
14     def __init__(self, tty):
15         self.port = serial.Serial(tty, baudrate=baud, xonxoff=False, rtscts=False,
16             dsrtr=False, stopbits=serial.STOPBITS_ONE, parity=serial.PARITY_NONE,
17             bytesize=serial.EIGHTBITS, timeout=None)
18
19     def send(self, s):
20         csum = 0
21         # bs = s.encode('ascii')
22         bs = s
23         for c in bs:
24             csum ^= c
25         nmea = b'$' + bs + (b'*%02X\r\n' % csum)
26         print(nmea)
27         self.port.write(nmea)
28
29     def write(self, index, value):
30         self.send(b'SW,%d:%f' % (index, value))
31
32     @staticmethod
33     def read(self):
34         #ser = serial.Serial(port_address, baudrate=uart_baud)
35         #line = ser.readline()
36         line = self.port.readline()
37         return line
38
39 x = extctl(port_address)
40
41 cur_line = str(port.readline())      # full line coming from glider
42 #print(f'cur_line: {cur_line}')      # view full line if viewed from RasPi
43 msg = cur_line.split('*')          # Split into separate fields
44 fields = msg[0].split(',')        # Split into separate fields
45 print(f'Fields: {fields}')
46
47 # the ifs MUST match the num order in extctl.ini
48 for subscription in fields:
49     if '3:' in subscription:           # index number of m_avg_speed
50         print('found 3:> processing!')
51         subs = subscription.split(':')    # from TWR
52         m_water_vel_dir = float(subs[1])
53
54         new_heading = m_water_vel_dir + (np.pi / 2)    # Adding 90 degrees still in radians
55         new_heading = np.round(new_heading,5)
56         print(f'From Glider ({m_water_vel_dir}): {m_water_vel_dir} From RasPi (perp_angle): {new_heading}') # see them for debug
57         x.write(0,new_heading)                      # change u_mission_param_a
58         glider_terminal_msg = new_heading
59         x.write_TXT(glider_terminal_msg)          # send the new_heading to the glider terminal to
59         # see when it is updated in SFMC
60         # this way you don't have to SSH into the RasPi and use tail -f on the cronlog

```

File 6.4: Simple Backseat Driver: sendHeadDir.py - Python script ran on the Raspberry Pi BSD

The rest of the files used in the mission remain unchanged from previous. All files are also on GitHub.

6.3. 90deg_bsd Test Results

For the first test with BSD controlling the heading we have the RasPi powered on the entire time. I would recommend this for a first test because it eliminates one place errors may occur and the ease of SSHing into the RasPi to see what fields are sent on each cycle for debugging purposes.

It is important to note that when the desired value is not found the RasPi prints the fields to end the script. If the script is not ended, because the CronJob runs the script every five seconds, the script "hangs" on the if statement. This leads to a "que" of scripts on the RasPi that "pile up", potentially leading to unintended consequences with RasPi behavior (memory usage crashes).

The following figure shows the `c_heading` for the backseat driver mission. Note the steps of `c_heading`, as it was updated every three hours upon new calculation of `m_water_vel_dir` as the heading driven glider, controlled by backseat driver, proceeded through the current field.

Not every message contains the desired value corresponding to `m_water_vel_dir` (Figure 6.1's mission it was position 5 and not 3 as shown previously).

Figure 6.2 illustrates how the RasPi properly updated the heading of the glider mid-mission every three hours as desired. However, the heading has a lot more steps than five (one for each current boundary it passed through). This is due to how the glider simulator itself calculates `m_water_vel_dir`.

```
Fields: ["b'$SD", '4:0.231009111066112']
Fields: ["b'$SD", '2:1703229641.789', '3:1.65108147238664']
Fields: ["b'$SD", '5:0']
found 5: -> processing!
From Glider (m_water_vel_dir): 0.0 | From RasPi (perp_angle): 1.5708
b'$SW,0:1.570800*37\r\n'
Fields: ["b'$SD", '4:0.231192270006687']
Fields: ["b'$SD", '4:0.23123812680972']
Fields: ["b'$SD", '2:1703229661.811', '3:1.59872159482681']
Fields: ["b'$SD", '4:0.231097803694824']
Fields: ["b'$SD", '4:0.231143754964168']
Fields: ["b'$SD", '5:0']
found 5: -> processing!
From Glider (m_water_vel_dir): 0.0 | From RasPi (perp_angle): 1.5708
b'$SW,0:1.570800*37\r\n'
Fields: ["b'$SD", '2:1703229681.831', '3:1.54461638801498']
Fields: ["b'$SD", '4:0.231326730991636']
```

Figure 6.1: Simple Backseat Driver: Cronlog from Simple Backseat Driver Application: The fields in variable `cur_line` on each glider cycle. Note intermittent "5:", processing, and sending of new heading from RasPi to glider.

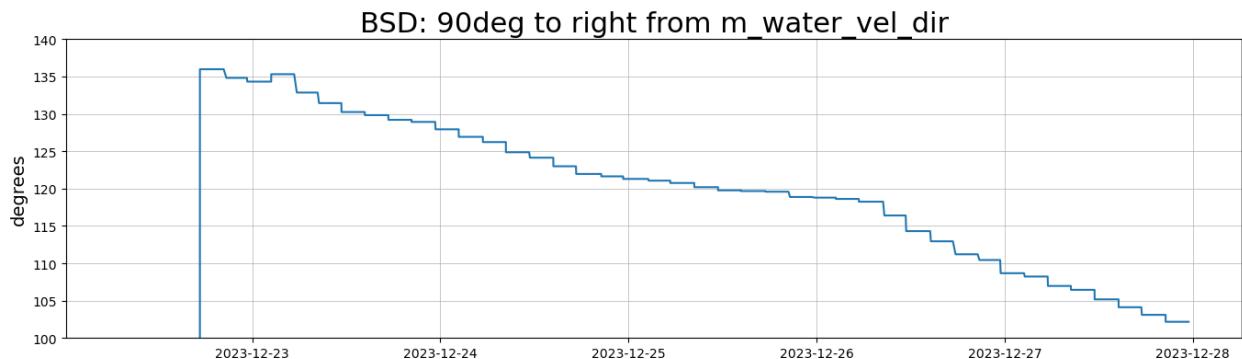


Figure 6.2: Simple Backseat Driver: `c_heading` 90° from `m_water_vel_dir` during simple backseat driver application

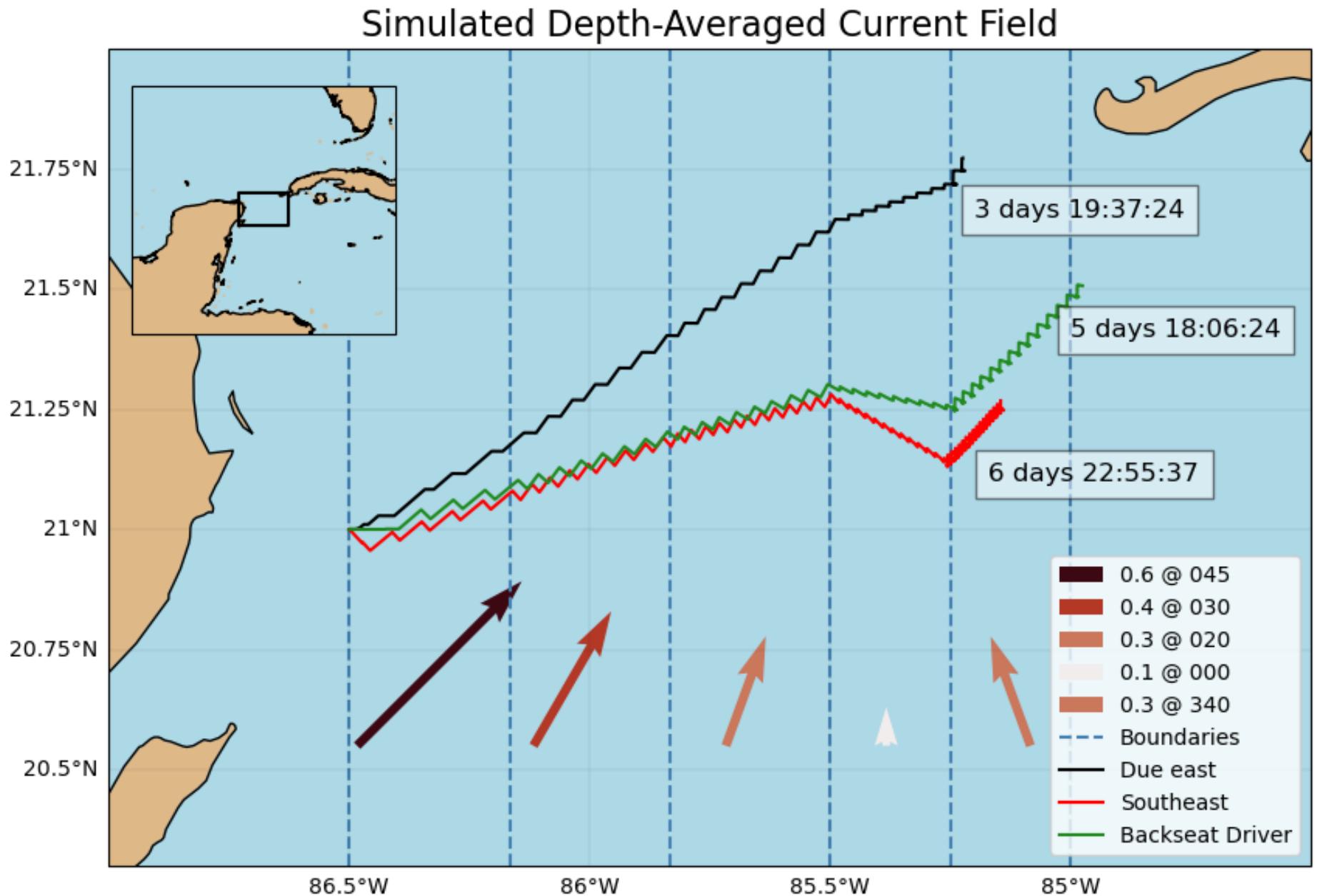


Figure 6.3: Simple Backseat Driver Location Comparison: Green line is location of spectre ran with the BSD control architecture (rest of map the same as Figure 5.4). Note green's arrival to eastern boundary faster than red's with its southeast waypoint, but with greater deflection distance northward.

7

Advanced Backseat Driver Behavior Application: Commanding c_heading, more sensors, make a plot

Chapter Concepts: We will ingest more sensors from the glider and do more calculations than the simple case.

1. The RasPi will be powered on for surfacing and diving only
2. The RasPi script contains a while loop to wait for multiple sensor values before continuing (in contrast to running the script despite the presence of desires values).
3. The RasPi upon receiving all sensor values required for the calculation, will calculate the most optimal heading for the prevailing current conditions, glider speed, and heading.
4. The RasPi will generate a script run-time file, used for debugging and troubleshooting before deployment.

For this application "most optimal" means the heading that results with the least northward deflection distance compared to flying a heading 90° from the current. For a more in-depth explanation, see [Appendix A](#).

7.0.1. extctl.ini

```
1 mp # MISSION PARAMS SECTION - u_mission_param[a-1]
2 # c_heading in set-he94.ma, used with b_arg: heading_value(X) 1000000
3 # INDEX NUMBER FOR RASPI PYTHON
4 u_mission_param_a # c_heading
5
6 os # OUTPUT SENSORS SECTION
7 sci_generic_a nodim # 1
8
9 is # INPUT SENSORS SECTION
10 # Glider coords
11 #flight
12 m_present_time timestamp # 2
13 m_heading rad # 3
14 m_avg_speed m/s # 4
15 m_water_vel_dir rad # 5
16 m_water_vel_mag m/s # 6
17 m_lat lat # 7
18 m_lon lon # 8
19
20 # Specify the baudrate of the uart, don't technically have to specify but to not forget
21 baud
22 9600 # default value
23
24 packet_size # don't technically have to specify but to not forget
25 600 # default value
```

File 7.1: Advanced Backseat Driver: extctl.ini - control c_heading, send more sensors (lat/lon, etc)

7.0.2. sample90.ma

```

1 behavior_name=sample
2 # SAMPLE90.MA (EXTCTL)
3
4 <start:b_arg>
5   b_arg: sensor_type(enum)          90  # c_extctl_on
6   b_arg: sample_time_after_state_change(s) 0  # start sampling right away
7
8   # Sampling Arguments
9     b_arg: intersample_time(sec)    0
10    b_arg: state_to_sample(enum)   9  # diving | on_surface
11    b_arg: nth_yo_to_sample(nodim) 1  # +n: first and nth: 1,3,5
12 <end:b_arg>

```

File 7.2: Advanced Backseat Driver: sample90.ma - power the RasPi on only on surface and diving

The rest of the glider's files remain the same as in the previous chapter's basic case.

7.1. Advanced Application: Raspberry Pi Script

This script is long, so these are the most important parts to explain. This is best looked at as an example for you to expand upon based upon your specific use-case requirements.

7.1.1. extctl Python class

```

1 # SKIPPING IMPORTS, SEE GITHUB
2 timer_start = datetime.datetime.now()  # TIMER START
3 values_present = 3      # PROGRESS MESSAGES SENT VIA GLIDER TERMINAL
4 all_done = 42
5
6 class extctl(object):
7     def __init__(self, tty):
8         self.port = serial.Serial(tty, baudrate=baud, xonxoff=False, rtscts=False,
9             dsrdr=False, stopbits=serial.STOPBITS_ONE, parity=serial.PARITY_NONE,
10            bytesize=serial.EIGHTBITS, timeout=None)
11
12     def send(self, s):
13         csum = 0
14         # bs = s.encode('ascii')
15         bs = s
16         for c in bs:
17             csum ^= c
18         nmea = b'$' + bs + (b'*%02X\r\n' % csum)
19         print(nmea)
20         self.port.write(nmea)
21
22     def write_SW(self, index, value):
23         self.send(b'SW,%d:%f' % (index, value))
24
25     def write_TXT(self,value):
26         self.send(b'TXT,%f' % (value))
27
28     def mode(self, mask, value):
29         self.send(b'MD,%d,%d' % (mask, value))
30
31     @staticmethod
32     def read(self):
33         #ser = serial.Serial(port_address, baudrate=uart_baud)
34         #line = ser.readline()
35         line = self.port.readline()
36         return line

```

File 7.3: Advanced Backseat Driver: sendHeadDir_WAIT.py - extctl Python class

7.1.2. while loop - parse strings from glider

```

1 m_present_time_norm = None # 2
2 m_heading_rad = None # 3
3 m_avg_speed = None # 4
4 m_water_vel_dir_rad = None # 5
5 m_water_vel_mag = None # 6
6 m_dr_time = None # 7
7 m_lat = None # 8
8 m_lon = None # 9
9
10 port = serial.Serial(port_address, baudrate=baud)
11 x = extctl(port_address)
12
13 while done_with_for_loop == 0:
14     print('start\u00d7while')
15     cur_line = str(port.readline())      # full line coming from glider
16     print(f'cur_line:{cur_line}')       # view full line if viewed from RasPi
17     msg = cur_line.split('*')          # Split into separate fields
18     fields = msg[0].split(',')        # Split into separate fields
19     # print(f'{datetime.datetime.now(): Fields: {fields}}')    # see fields not cur_line
20
21     # number must match index number in extctl.ini
22     for subscription in fields:
23         if '2:' in subscription:           # index number of m_present_time_epoch
24             subs = subscription.split(':')  # from TWR
25             m_present_time_epoch = float(subs[1])
26             m_present_time_norm = datetime.datetime.fromtimestamp(m_present_time_epoch) # convert to human timestamp
27             print(f'found\u00d72\u00d7{m_present_time_norm}')
28
29         if '3:' in subscription:           # index number of m_heading
30             subs = subscription.split(':')  # from TWR
31             m_heading_rad = float(subs[1])
32             print(f'found\u00d73\u00d7{m_heading_rad}')
33             m_heading_deg = math.degrees(m_heading_rad)
34
35         if '4:' in subscription:           # index number of m_water_vel_mag
36             subs = subscription.split(':')  # from TWR
37             m_avg_speed = float(subs[1])
38             print(f'found\u00d74\u00d7{m_avg_speed}')
39
40         if '5:' in subscription:           # index number of m_water_vel_dir_mag
41             subs = subscription.split(':')  # from TWR
42             m_water_vel_dir_rad = float(subs[1]) #
43             print(f'found\u00d75\u00d7{m_water_vel_dir_rad}')
44             m_water_vel_dir_deg = math.degrees(m_water_vel_dir_rad)
45
46         if '6:' in subscription:           # index number of m_water_vel_mag
47             subs = subscription.split(':')  # from TWR
48             m_water_vel_mag = float(subs[1]) #
49             print(f'found\u00d76\u00d7{m_water_vel_mag}')
50
51         if '7:' in subscription:           # index number of m_water_vel_mag
52             subs = subscription.split(':')  # from TWR
53             m_dr_time = float(subs[1])
54             print(f'found\u00d77\u00d7{m_dr_time}')
55
56         if '8:' in subscription:           # index number of m_water_vel_mag
57             subs = subscription.split(':')  # from TWR
58             m_lat = float(subs[1])
59             print(f'found\u00d78\u00d7{m_lat}')
60
61         if '9:' in subscription:           # index number of m_water_vel_mag
62             subs = subscription.split(':')  # from TWR
63             m_lon = float(subs[1])
64             print(f'found\u00d79\u00d7{m_lon}')

```

File 7.4: Advanced Backseat Driver: sendHeadDir_WAIT.py - parse strings from glider

7.1.3. while loop - continue when all values are present

```

1   if (m_present_time_norm           != None) and \
2       (m_heading_rad              != None) and \
3       (m_avg_speed                != None) and \
4       (m_water_vel_dir_rad        != None) and \
5       (m_water_vel_mag            != None) and \
6       (m_dr_time                  != None) and \
7       (m_lat                      != None) and \
8       (m_lon                      != None):
9       #print(m_present_time_norm,m_heading_rad,m_avg_speed,m_water_vel_dir_rad,
10          m_water_mag,m_dr_time,m_lat,m_lon)
11
12   print(f'{datetime.datetime.now().strftime("%Y-%m-%d_%H:%M:%S")}_values_present,lets go
13      !')
14   x.write_TXT(values_present) # first progress message
15
16   if 270 <= m_water_vel_dir_deg < 360 or 0 <= m_water_vel_dir_deg < 45:
17       # use optimal heading
18       new_heading_send = np.round(heading_with_min_deflection_rad,3)           # round to
19       three digits ##### because we don't need all
20       x.write_SW(0,new_heading_send)                                              # change
21       heading on glider
22       x.write_TXT(new_heading_send)                                              # print
23       message in glider terminal
24       print(f'{current_time}: new_heading={new_heading_send}') # print new heading with
25       timestamp in RasPi terminal
26
27   elif 45 <= m_water_vel_dir_deg < 90:
28       # use 90 degrees from current (relative to current)
29       new_heading_send = np.round(math.radians(m_water_vel_dir_deg + 90),3)
30       x.write_SW(0,new_heading_send)                                              # change
31       heading on glider
32       x.write_TXT(new_heading_send)                                              # print
33       message in glider terminal
34       print(f'{current_time}: new_heading={new_heading_send}') # print new heading with
35       timestamp in RasPi terminal
36
37   elif 90 <= m_water_vel_dir_deg < 270:
38       # use 90 degrees (due east)
39       new_heading_send = math.radians(90)
40       x.write_SW(0,new_heading_send)                                              # change
41       heading on glider
42       x.write_TXT(new_heading_send)                                              # print
43       message in glider terminal
44       print(f'{current_time}: new_heading={new_heading_send}') # print new heading with
45       timestamp in RasPi terminal
46
46   else:
47       x.write_TXT(er_m_water_vel_dir)
48       pass
49
50   done_with_for_loop = 1 # true, finish while loop and done with script
51
52   # MORE CODE USING THE VALUES IF DESIRED
53
54   # after the while loop
55   x.write_TXT(all_done)
56
57   # TIMER END AND CALCULATE SCRIPT RUN TIME
58   timer_end = datetime.datetime.now()
59   script_time = timer_end - timer_start
60   elapsed_seconds = str(round(script_time.total_seconds(),2))
61
62   # PUT SCRIPT RUN TIME IN A FILE
63   with open('script_times.txt','a') as file:
64       file.write(f'{elapsed_seconds}\n')
65
66   print(f'script is done! ({elapsed_seconds}s)')

```

File 7.5: Advanced Backseat Driver: sendHeadDir_WAIT.py - continue when all values are present

7.1.4. crontab

```

1 # to open crontab thru the RasPi's terminal (only open via terminal, not the file on the RasPi)
2 sudo crontab -e
3
4 # to run once every minute, as the script takes between 18-48 seconds
5 * * * * * sh /home/rutgers/backseat_driver/launcher.sh >> /home/rutgers/logs/cronlog 2>&1
6
7 # to display updates written to cronlog while crontab is running
8 # tail -f cronlog

```

File 7.6: Advanced Backseat Driver: Setup the Crontab

As we'll see later, this script takes 45 seconds max to run; thus, the crontab runs every minute. Manually run your script a few times on the RasPi (`sudo python3 [script_name.py]`) before using the crontab so you can see how long it takes.

7.2. Advanced Backseat Driver Output Plot

The Python file for this outputs two files: 1) a plot (Figure 7.1) and 2) a `script_times.txt` that contains the run-time of the script. In a real deployment this type of plot would not be generated onboard the glider using the external controller - it would be generated shore-side. This was done onboard the external controller to 1) show the flexibility and computing improvement of adding an external controller to the glider system, and 2) allow for ease of troubleshooting mid-mission.

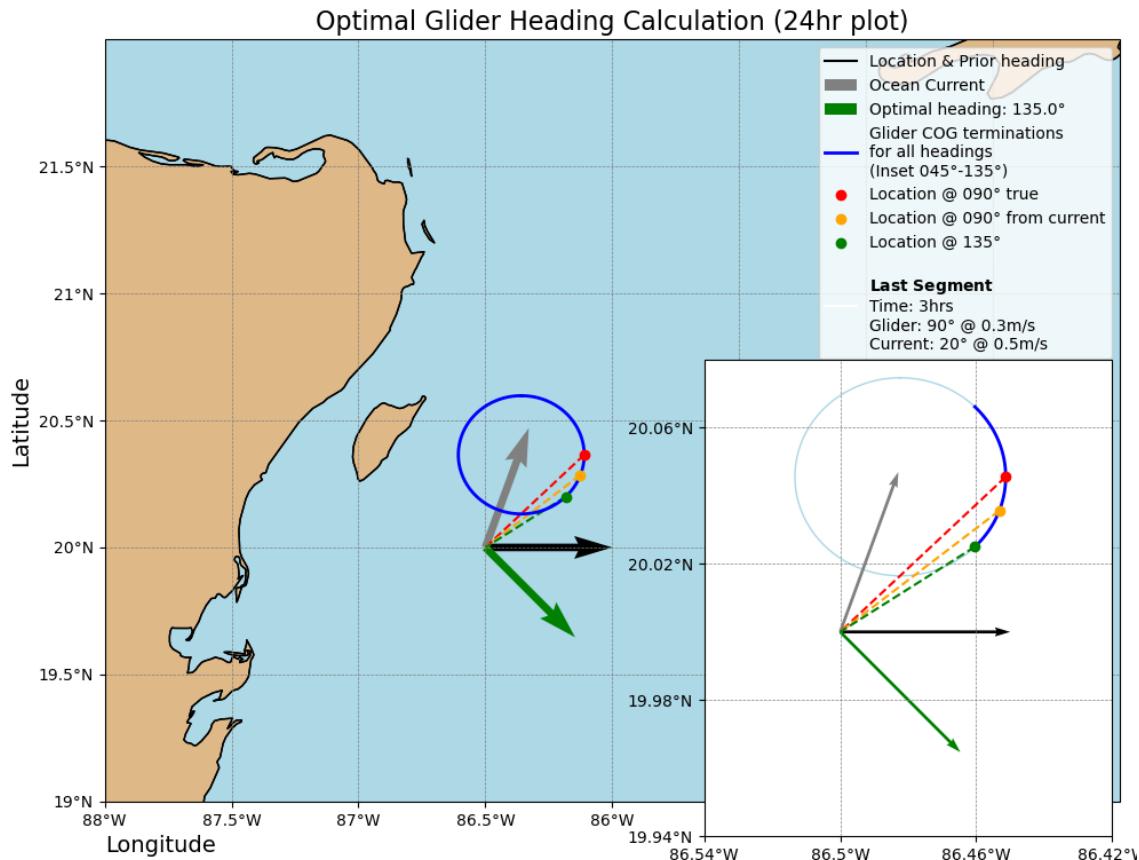


Figure 7.1: Advanced Backseat Driver: External controller output plot

If you are planning on making a diagnostic plot from values sent ashore from the glider, especially if using the external controller to read a sensor's output (like an AD2CP) and then generate some intermediary file for mid-mission data, I recommend doing it all onboard the external controller while simulating. You can immediately

know if any part breaks (external controller reading the sensor data or your diagnostic plot's code) without having to wait for the mission to end, going through SFMC, or troubleshooting during a real deployment in the ocean.

7.3. External Controller Python Script Run-Time Considerations

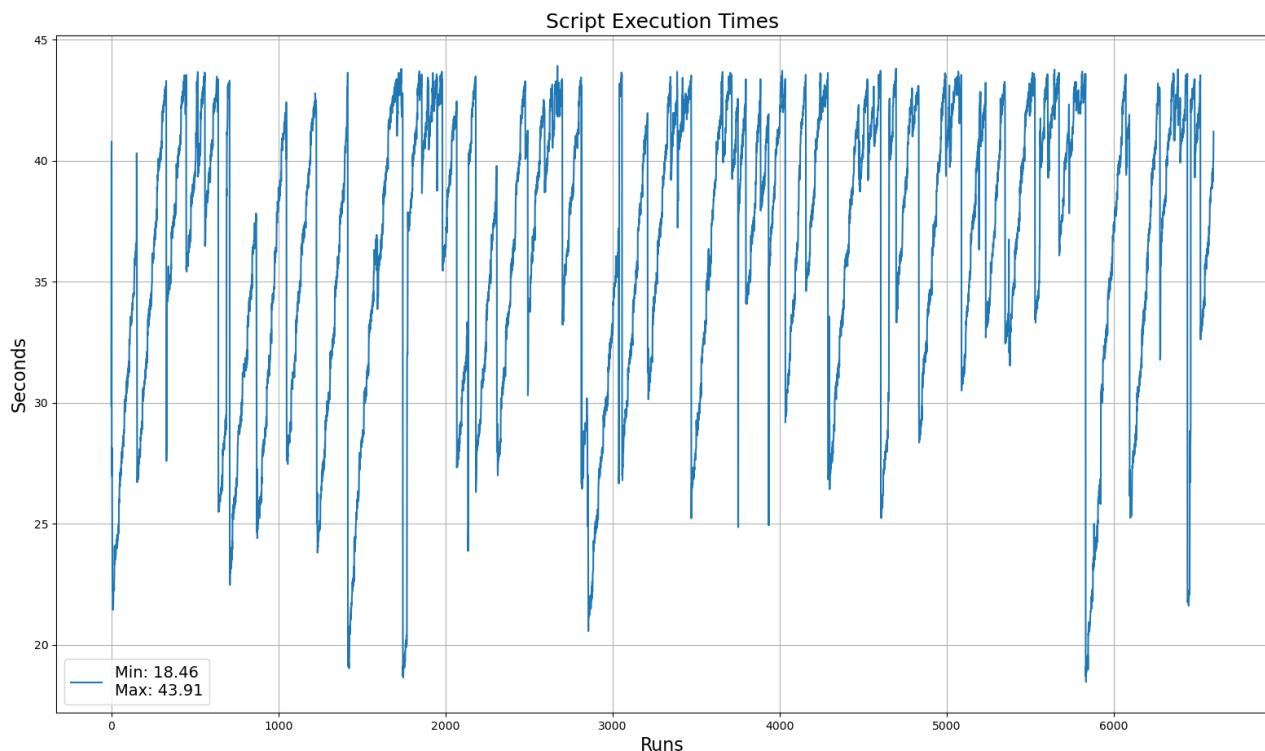


Figure 7.2: Advanced Backseat Driver: Script Run Times

The external controller script has a highly variable run-time, ranging from 18.5 seconds to 44 seconds. As the Python script itself can run in under 18 seconds, the majority of this time is waiting for all sensor values to be sent before continuing.

If one starts requesting for many sensor values, this time will only increase and should be tested before any deployment. Additionally, through pure speculation from looking at the output from the glider for many hours, certain sensors are updated more frequently while others come through often in some states and not others.

For example, `m_avg_speed` shows up basically every cycle, while values like `m_water_vel_dir` show up more often while surfacing and the first five or so minutes of diving behavior before it seems to taper off. Thus, if you are interested in one specific sensor value (like I was with `m_water_vel_dir`) test things out and get a "feel" for how the sensor value will appear. Basically, sit and watch for it to see how it comes in, or quantify

It bears repeating that "there is not currently a way to have the BSD send every sensor requested each cycle. From Teledyne, "this was done to reduce the number of calls over the clothesline. When the SC attempts to send a \$SD message the proget will check which sensors were updated and only request those that have been updated. Only sensors that were updated will be sent in the \$SD. A way of keeping everything in sync on the external controller side of the system would be to have a accessible struct that is continuously updated based on inputs over the UART. This will help keep a external controller side memory of what all the values are."

7.4. More Glider Simulator Limitations

During testing for this chapter I ran into an issue with how the glider simulator itself calculates water velocities. After communicating with Teledyne (much of the following text in this section comes from communication with Teledyne, see email in GitHub files) and troubleshooting myself. An overview for how the glider calculates water velocities is as follows:

Non-simulation glider (real glider and in the ocean): Water velocities get recalculated whenever `m_gps_x_lmc` is updated. In a normal mission this will occur when the glider surfaces and gets a GPS fix.

Simulation glider (shoebox, in a lab): The `s_water_speed` and `s_water_direction` are used in the dead reckoning while in simulation mode and act as the true water velocity for how the simulated glider positions will change while underway. `m_water_vel_dir` and `m_water_vel_mag` however are used for heading correction and every time a surfacing occurs will be calculated, see the method below. Glider tries goes underway running calculations with `m_water_vel_dir` and `m_water_vel_mag`. When it surfaces it compares dead reckoned positions to actual position to calculate water velocities.

- `m_dr_x_ini_err = m_gps_fix_x_lmc - m_dr_surf_x_lmc`
- `m_dr_y_ini_err = m_gps_fix_y_lmc - m_dr_surf_y_lmc`
- `delta_vx = m_dr_x_ini_err / m_dr_time`
- `delta_vy = m_dr_y_ini_err / m_dr_time`
- `m_water_vx = delta_vx + x_prior_seg_water_vx`
- `m_water_vy = delta_vx + x_prior_seg_water_vy`

I've bolded the end of the last two lines where `m_water_vel_dir` and `m_water_vel_mag` have memory of the prior segments. Because of this it was recommended to reset the following values (instead of only `m_water_vel_dir` and `m_water_vel_mag`):

- `x_prior_seg_water_vx`
- `x_prior_seg_water_vy`
- `m_water_vx`
- `m_water_vy`
- `m_water_vel_mag`
- `m_water_vel_dir`

I ran a few missions attempting to use an SFMC .xml script to !put these sensor values to zero before resuming the mission (immediately before diving), to no success. Putting them to zero seemed to have no effect on the calculations onboard the glider. As this was not "mission critical" for my thesis work to continue, no further development into solving this issue was pursued. A potential way forward, that also avoids at surface calculations overwriting any of the puts would be for the external controller to update these sensor values. But, as that would basically defeat the purpose of the external controller calculating from these values, it has little application in this specific example, with potential uses in other external controller applications.

Furthermore, `m_dr_time` does not seem to work in the simulator. Again, I accepted this and moved on to further work.

8

Sending Files To/From External Controller

Chapter Concepts:

1. The extctl Python class used
2. How to use the extctl functions
3. The Python code required

The required functions for this operation are much more involved. As this is a Python heavy chapter, please be reminded that all files used throughout this manual are on GitHub. Click [here](#) to open.

For this chapter, the RasPi will be powered on the entire time. We can use the external controller to send and/or receive a file anytime in the mission, so we begin the mission, let the glider dive, then work through examples in this chapter

8.1. Baud Rate and Chunk Size

The `extctl.ini` file specifies the baud rate and chunk size the science computer and external controller use to communicate. Changing these will change how long it takes to transfer files back and forth.

Baud Rate: The default value is 9600, and the max recommended in Teledyne's Backseat Driver Software Guide is 115200 for the most stable communications. This can be pushed, see following table.

Chunk size: This sets the max size of data sent back and forth (in bytes per packet). The default value is 500, and the max value in Teledyne's Backseat Driver Software Guide is 730. However, if sending a comparatively big file it may be desired to push this some. Do so requires a faster baud rate, see following table.

Table 8.1: Recommended Baud Rates and Chunk Sizes

Baud Rate	Chunk Size	Details
9600	500	Default communications that will be slower than other settings
115200	730	Fastest Teledyne recommended settings
230400	1200	Upward capability possible from the system from our testing

While it's certainly possible to use values beyond Teledyne's recommended fastest communication settings, extensive lab-based testing should be conducted to ensure stability of the system. The manufacturer recommends the settings for a reason - they don't want a glider to have an issue in the ocean - so properly test all systems in the lab before a deployment. This should be done anyways, but especially when attempting to push settings towards there limits.

8.1.1. extctl Python class: sending/receiving files

```

1 # define locations and generics
2 uart_baud = 9600
3 send_chunk_size = 600
4 port_address = '/dev/ttyUSB0' # CHANGE x TO PORT USED
5
6 # Set path to save processed data files to give back to glider
7 to_dir='/home/rutgers/backseat_driver/to_glider/'
8
9 # Set path for .tbd files from glider
10 from_dir = '/home/rutgers/backseat_driver/from_glider/'
11
12 # Set path for completed files
13 pros_dir = '/home/rutgers/backseat_driver/files_processed/'
14
15 class extctl(object):
16     def __init__(self, tty):
17         self.port = serial.Serial(tty, baudrate=uart_baud, stopbits=serial.STOPBITS_ONE, parity=serial.PARITY_NONE, bytesize=serial.EIGHTBITS)
18
19     def send(self, s):
20         csum = 0
21         # bs = s.encode('ascii')
22         bs = s
23         for c in bs:
24             csum ^= c
25         nmea = b'$' + bs + (b'*%02X\r\n' % csum)
26         print("Sending:\u21d2", nmea)
27         self.port.write(nmea)
28
29     def write(self, index, value):
30         self.send(b'SW,%d:%f' % (index, value))
31
32     def mode(self, mask, value):
33         self.send(b'MD,%d,%d' % (mask, value))
34
35     def read(self):
36         return self.port.readline()
37
38     @staticmethod
39     def send_file_data(data):
40         # Encode the data in b64
41         output_string = b'FO,' + base64.b64encode(data)
42         x.send(output_string)
43
44     def file_read(self, filename, sci_dir, data_type):
45         # Send open read message to glider and open destination file
46         command = b'FR,' + bytes(filename, 'utf-8') + b',' + bytes(sci_dir, 'utf-8')
47         self.send(command)
48         print(f'opening\u21d2file\u21d2{filename}')
49         if data_type:
50             file = open(f'/home/rutgers/backseat_driver/from_glider/{filename}', 'wb')
51         else:
52             file = open(f'/home/rutgers/backseat_driver/from_glider/{filename}', 'w')
53         self.recieve_file_data(file, data_type)
54
55     def GO(self):
56         self.send(b'GO')
57
58     # This function sends the command for the extctl proglot to list files in 'directory'
59     # matching 'wild_card'
60     def list_directory(self, directory, wild_card):
61         self.send(b'LS,' + bytes(directory, 'utf-8') + b',' + bytes(wild_card, 'utf-8'))
62
63     # File write but with a specifier for
64     def file_write_new(self, source_file, directory):
65         # Send command to open a file in directory
66         self.send(b'FW,' + bytes(source_file, 'utf-8') + b',' + bytes(directory, 'utf-8'))

```

File 8.1: Advanced Backseat Driver: sendHeadDir_WAIT.py - continue when all values are present

8.1.2. extct Python class: sending/receiving files

```

1  # Runs full loop to send a file
2  @staticmethod
3  def send_file(uart_port, file_path, dest_name, dest_dir, byte_bool):
4      # Bool for if need confirmation message
5      x.file_write_new(dest_name, dest_dir)
6      # open the specified file.
7      with open(file_path, 'rb') as file_object:
8          while True:
9              # This for binary based files
10             if byte_bool:
11                 data = file_object.read(send_chunk_size)
12             # This for text data
13             else:
14                 data = bytes(file_object.read(send_chunk_size), 'utf-8')
15             # End of file, break loop
16             if not data:
17                 print("file\u2022send\u2022break")
18                 break
19             # Send data and set waiting bool
20             x.send_file_data(data)
21             waiting = 1
22             # Read the uart line until you read get the confirmation message from the
23             # proglet
24             while waiting:
25                 check_line = uart_port.readline()
26                 check_line = check_line.decode('utf-8')
27                 check_line = check_line.split('*')[0]
28                 # Recieved ok to send next chunk, otherwise keep checking for it
29                 if check_line == '$NEXT':
30                     print('Got\u2022the\u2022next')
31                     waiting = 0
32
33     def receive_file_data(self, file, data_type):
34         line = self.read()
35         print(line)
36         # print('chunk sent')
37         if line.startswith(b'$ER'):
38             return
39         if line.startswith(b'$FI'):
40             if line.startswith(b'$FI,'):
41                 line = line.split(b',')
42                 line = line[1]
43                 # print(f'line: {line}')
44                 content = line.split(b'*')[0]
45                 # print(f'content: {content}')
46                 if data_type == 'bin':
47                     content = base64.decodebytes(content) # binary data
48                 else:
49                     # original
50                     # content = base64.b64decode(content).decode('utf-8') # other data
51
52                     # newtry
53                     content = base64.decodebytes(content)
54                     print(f'content=\u2022{content}')
55                     # content = base64.b64decode(content) # other data
56
57                     file.write(content)
58                     self.GO()
59                     self.receive_file_data(file,data_type)
60             else:
61                 print('Reached\u2022EOF,\u2022closing\u2022file\u2022and\u2022exiting')
62                 file.close()
63                 return
64             self.receive_file_data(file,data_type)

```

File 8.2: Advanced Backseat Driver: sendHeadDir_WAIT.py - continue when all values are present

8.1.3. Python implementation for sending/receiving files

```

1 serial.Serial(port_address, baudrate=uart_baud).close()
2 x = extctl(port_address)
3 port = serial.Serial(port_address, baudrate=uart_baud)
4
5 test=0
6 count=0
7
8 # ask SC to list files in a directory, sends names to EC)
9 # init file_list, place to put the file names we get back from the list_directory command
10 file_list=[]
11 x.list_directory("logs",".tbd")
12
13 print('before while')
14 while test == 0:
15     cur_line = str(port.readline())
16     # print(cur_line) results in
17     # ["b'$DIR",'c:/logs/0010000.tbd', 'c:/logs/00110000.tbd', 'c:/logs/00120000.tbd', '']
18     # Split into separate fields
19     msg = cur_line.split('*')
20     fields = msg[0].split(',')
21     # print(fields) results in
22     # ["b'$DIR", 'c:/logs/0010000.tbd', 'c:/logs/00110000.tbd', 'c:/logs/00120000.tbd',
23     #   '']
24
25     for subscription in fields:
26         if 'c:' in subscription: # get files from the list_dir function
27             subs=subscription.split(':')
28             pfile=subs[1]
29             # print(subs)
30             file_list.append(subscription)
31             print(f'file_list:{file_list}')
32
33     if (file_list != None):
34         # print(f'file_list -1: {file_list[-1]}' )           # for viewing
35         # tbd_file_name = os.path.basename(file_list[-1])    # use -1 to get the most recent
36         # file
37         tbd_file_name = 'ru34-2024-089-0-22.tbd' # or ^, but easy to specify for testing
38
39         # Send file from SC to EC
40         x.file_read(tbd_file_name,"logs",1)
41         # add path to most recent filename
42         print('file transferred, trying to process now... ')
43         file = '/home/rutgers/backseat_driver/from_glider/'+tbd_file_name
44
45         # the .tbd file is now on the glider and can be used
46
47         # send file from EC to SC
48         # to get a tbd (or other file) on the glider for testing mid-mission, use these lines
49         # saves having to end the mission, dockzr etc
50         # x.send_file(port, "/home/rutgers/backseat_driver/"+str(name), str(name), "logs", 1)
51         # send_file(uart_port, file_path, dest_name, dest_dir, byte_bool):
52
53     test = 1

```

File 8.3: Advanced Backseat Driver: sendHeadDir_WAIT.py - continue when all values are present

9

Science Sensor Integration, Testing, and Data File Management

Chapter Concepts: In this chapter we will have the RasPi read values from a CTD41CP sensor. There are three ways we will accomplish this, each with their own example:

Example A: Plugging a real CTD41CP sensor into the science port's of the simulator. Because this is rather simple to execute, we also append these values to a file on the RasPi so it can access and/or make decisions using the prior x amount of information.

Example B: Simulating a CTD41CP sensor using a .py script. Ran on a laptop that sends serial data through the same type of cable as the real sensor, we decide what ocean the glider is "seeing" mid-mission.

Example C: Sending the latest .tbd file from the science computer and processing the values therein, ready to make a decision using them.

9.1. Example A: Using an Actual CTD41CP Sensor

This is predicated upon having an actual sensor and it's connecting wire in-hand. If you don't, use all the glider files for Example A and go to Example B, simulating the data.

9.1.1. extctl.ini

```
1 mp # MISSION PARAMS SECTION - u_mission_param[a-1]
2 # no mission params
3
4 os # OUTPUT SENSORS SECTION
5 sci_generic_a nodim          # 0
6
7 is # INPUT SENSORS SECTION
8 # Glider coords
9 #flight
10 m_present_time timestamp    # 1
11
12 #science values
13 sci_water_cond S/m          # 2
14 sci_water_temp degC          # 3
15 sci_water_pressure bar      # 4
16
17 baud
18 9600 # default value
19 packet_size
20 600 # default value
```

File 9.1: Example A: extctl.ini - send CTD data

9.1.2. proglets.dat

```

1 proglet = extctl
2     uart      = j5          # place it's plugged in
3     bit       = 0          # power, battery
4     start_snsr = c_extctl_on(sec)
5
6 proglet = ctd41cp
7     uart      = j4          # connector
8     bit       = 0          # power, battery
9     start_snsr = c_profile_on(sec)
10    simulator = ctd41cp_sim

```

File 9.2: Example A: proglets.dat - include CTD41CP proglet

The only addition from our usual is the CTD41CP proglet found in `motherproglets.dat`

9.1.3. tbdlist.dat

```

1 # Basic Data
2 sci_m_present_time
3
4 # CTD41CP Sea-bird CTD(SBE-41)
5 # proglet = ctd41cp SIMULATING THE DATA THIS SENSOR SENDS
6 # -----
7 sci_water_cond      30   15   -1   3
8 sci_water_pressure  30   15   -1   3
9 sci_water_temp      30   15   -1   3

```

File 9.3: Example A: extctl.ini - send CTD data

Now that there is science data, we can include it in the `tbdlist.dat`. In Example C we will send the most recent `.tbb` (or other) file from the science computer to the external controller.

9.1.4. sample01.ma

```

1 behavior_name=sample
2
3 # sample01.ma (CTD)
4
5 <start:b_arg>
6     b_arg: sensor_type(enum)           1    # c_profile_on, is CTD
7     b_arg: sample_time_after_state_change(s) 0    # start sampling right away
8
9     # Sampling Arguments
10    b_arg: intersample_time(sec)      1    # if = 0 then fast as possible, > 0 secs
11    b_arg: state_to_sample(enum)     15   # diving|hovering|climbing, all
12    b_arg: nth_yo_to_sample(nodim)   1    # after the first yo, sample only every nth to
13 <end:b_arg>

```

File 9.4: Example A: tbdlist.dat - include CTD science data

The generic sample file for the CTD41CP that has it always on (during all behaviors). It's important to note that if ran in air there will be no salinity value, which, while it looks weird, is fine.

9.1.5. science.mi

```

1 # FULL FILE ON GITHUB
2 # SAMPLE : CTD41CP, THE ADDED SENSOR
3 behavior: sample
4     b_arg: args_from_file(enum)      01 # 01 CTD

```

File 9.5: Example A: science.mi - calls sample01.ma as well

The mission file for this example is almost the same for Example B and Example C, as it utilizes both the flight computer's and science computer's CTD41CP proglet and `sample01.ma` files.

9.2. Example B: Simulating a CTD41CP Sensor

In this example, we fully simulate the sensor and send data to the science computers port via an RS232 USB adapter and a Python .py on a laptop. To test this, and verify proper operation, we use all the same files as Example A, plus the additional Python .py script to simulate ocean variables. We don't run the Python .py script that simulates the ocean on the external controller so we can observe and verify proper BSDA behavior when reading specific science sensor values. We will simulate our ocean two ways, by sending data using:

simCTD_synthetic.py: Creating fake science data that roughly approximates an ocean, stored in a Pandas DataFrame, then sending values therefrom.

simCTD_file.py: Reading .ebd files containing real ocean variable data from an oceangoing glider mission.

9.2.1. Why Simulate a Sensor?

Depending on the type of BSDA application it may not be possible to properly test the application without reading a specific type of ocean variables and/or ocean conditions from the glider itself. In a lab setting this can often be impossible using an actual sensor because the glider is not actually in the ocean. Thus, to allow certain types of external controller application tests we must simulate the incoming science sensor data.

9.2.2. simCTD_synthetic.py

Figure 9.1 is just one example of a type of data possible can send. Using simCTD_synthetic.py, you could send any values desired to the science computer and thus external processor.

1. First we define the ctd41cp Python class with the required commands.
2. Then we approximate changing ocean conditions following a typical depth profile. Figure 9.1 (left) shows the three values that are sent to the glider. The rate of change of temperature and salinity is approximated with cosine functions, while pressure assumes a moment of initialization at the surface before a linear downward trajectory. In this, we assume the change is over ten minutes, resulting in 600 data values, as seen on the left.
3. Lastly, we open the computer's COM port and send the simulated data to the glider's science computer at 1hz through the USB to RS232 adapter (using

Note that all variables must be the same length as one another, and the length changes the overall time the script will send the variables to the science computer. Furthermore, while the script sends data at 1hz (simulating the CTD41CP sensor) the external controller won't read the data that fast. Using a crontab running Example A's script every 5 seconds the external controller only read in 105 values. The values were correct, just in less total number (figure 9.2, next page)

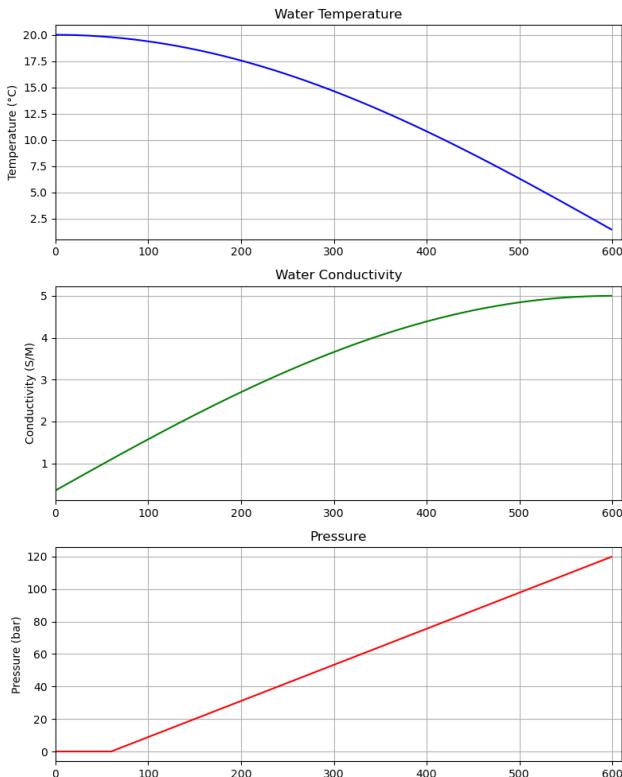


Figure 9.1: Example B: simCTD_synthetic.py's simulated CTD data

Once simCTD_synthetic.py is ready to run, you have a connector made to connect your laptop to the science computer (next section), and you have the external controller ready from Example A:

1. Start the glider's science.mi mission. Approximately five or so seconds later, approximately once you see the mission stack;
2. Start sending the simulated science data with the Python script, then;
3. Use the crontab, or run one iteration of the file, to run Example A (or another program) on the external controller.

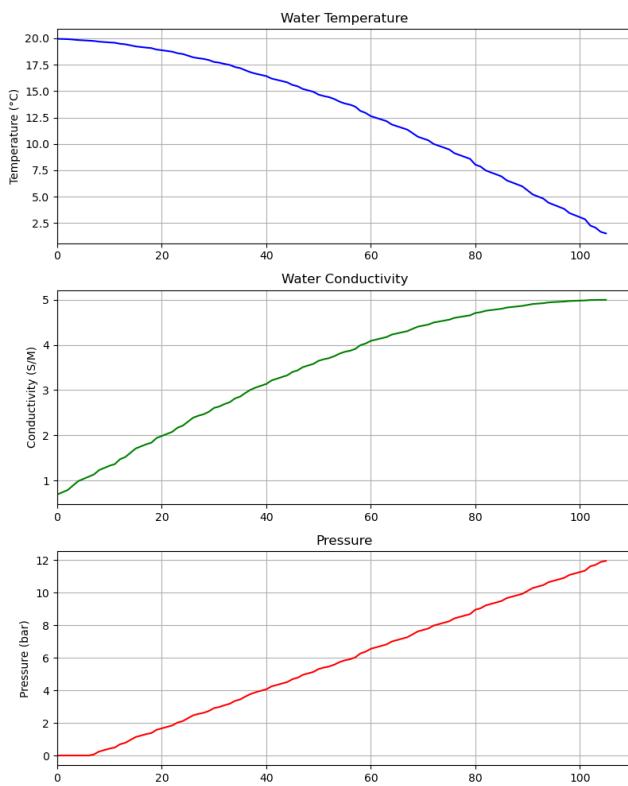


Figure 9.2: Example B: simCTD_synthetic.py's simulated CTD data as seen from the external controller

9.2.3. simCTD_file.py:

This example is from Ella Wawrynek who first used this method using .ebd files. This Python program:

1. Defines the ctd41cp Python class
2. Decodes .ebd files, from a real glider mission
3. Sends the data as before

See file on GitHub or on later page of this manual.

9.2.4. Making the Connector to Simulate a Science Sensor

For this connector we need a USB to RS232 adapter and a 10pin male Molex, like before. We will be connecting the two, allowing the connection of a laptop's USB to the science computer.

9.2.5. Example B: simCTD_synthetic.py

```

1 class ctd41cp(object):
2     def __init__(self, port):
3         self.port = serial.Serial(port, baudrate=baud, stopbits=serial.STOPBITS_ONE, parity=
4             serial.PARITY_NONE,
5             timeout=timeout, bytesize=serial.EIGHTBITS)
6
7     def write(self, s,count):
8         print('Sending: ', s,'| count is',count)
9         self.port.write(s)
10
11    def read(self):
12        return self.port.readline()
13
14 min_to_run = 10
15 time_to_run = np.arange(0,min_to_run*60,1)
16 length = len(time_to_run)
17
18 # warmer surface, decrease with depth (time)
19 surface_temp = [20]
20 iterate_temp = np.cos(np.arange(0,1.5,(1.5/length)))
21 water_temp = surface_temp * iterate_temp
22
23 # fresher surface, increase with depth
24 depth_salt = [5]
25 iterate_salt = np.cos(np.arange(-1.5,0,(1.5/length)))
26 water_cond = depth_salt * iterate_salt
27
28 pressure_pause = [0]*60 # surface time before diving
29 pressure_data = (np.arange(0,120,(120/(length-60))))
30 pressure = pressure_pause.__iadd__(pressure_data)
31
32 sci_data = pd.DataFrame(index=time_to_run)
33 sci_data['cond'] = water_cond
34 sci_data['temp'] = water_temp
35 sci_data['pressure'] = pressure
36 indx = len(sci_data)-1
37
38 port = 'COM5' # USB port
39 baud = 9600
40 timeout = 0 # Non-blocking read
41
42 print('Opening port...')
43 u = ctd41cp(port)
44
45 print('Sending CTD data to simulator...')
46 index_len = len(sci_data)-1 #-1 cuz zero starting list
47 count = 0
48 while indx > 0:
49     round = 4
50     cond = np.round(sci_data.cond[count],round)
51     temp = np.round(sci_data.temp[count],round)
52     pres = np.round(sci_data.pressure[count],round)
53     write = (f'{pres},{temp},{cond}\r\n')
54     encoded_bytes = write.encode('utf-8')
55     u.write(encoded_bytes,count)
56
57     time.sleep(1) # approx 1 Hz data rate
58     indx = index_len - 1
59     count = count + 1
60
61 u.port.close() # have to close or can't reopen and run again
62 print('all data sent')

```

File 9.6: Example B: simCTD_synthetic.py

Used for creating the simulated CTD values sent to the glider from a laptop. These values are then sent to the external controller by the glider.

9.2.6. Example B: simCTD_file.py

```

1 port = 'COM5' # USB port
2 baud = 9600
3 timeout = 0 # Non-blocking read
4
5 class ctd41cp(object):
6     def __init__(self, port):
7         self.port = serial.Serial(port, baudrate=baud, stopbits=serial.STOPBITS_ONE, parity=
8             serial.PARITY_NONE,
9             timeout=timeout, bytesize=serial.EIGHTBITS)
10
11     def write(self, s):
12         print("Sending: ", s)
13         self.port.write(s)
14
15     def read(self):
16         return self.port.readline()
17
18 def get_tCTDxy_data_array(files, comp=False):
19     xbd=dbdreader.MultiDBD(files, complemented_files_only=comp, complement_files=comp)
20     t_ctd,C,T,P=xbd.get_CTD_sync()
21     tCTDxy_data = np.array([t_ctd,C,T,P])
22     return tCTDxy_data
23
24 def get_data_array(files, comp=False):
25     xbd=dbdreader.MultiDBD(files, complemented_files_only=comp, complement_files=comp)
26     raw_array = xbd.get("sci_water_cond", "sci_water_temp", "sci_water_pressure", return_nans=
27         True)
28     return raw_array
29
30 # %% Get CTD data from previous ebd files
31
32 datapath = r''
33 datafiles = [datapath+"00810000.ebd", datapath+"00820000.ebd", datapath+"00910002.ebd"]
34
35 print('Decoding CTD data...')
36 ctd_data = get_tCTDxy_data_array(datafiles)
37 #ctd_data_raw = get_data_array(datafiles)
38
39 print('Opening port...')
40 u = ctd41cp(port)
41 # u.write(b'!--start logging--\r\n') # This line appears to be unnecessary
42
43 print('Sending CTD data to simulator...')
44 indx = len(ctd_data[0])-1
45 while indx > 0:
46     u.write(b'%f, %f, %f\r\n' %(ctd_data[1][0], ctd_data[2][0], ctd_data[3][0]))
47     time.sleep(1) # approx 1 Hz data rate
48     indx = indx - 1

```

File 9.7: Example B: simCTD_file.py

Used for pulling CTD values from a file.

9.3. Example C: Retrieving and Processing the Most Recent .tbd File

This process can be used with any file on the science computer, but here we will use a .tbd file as we generated one in Example A. If you have a specific sensor file in mind, use it instead.

Once the file has been sent to the external controller from the science computer there are a few ways to process the .tbd file. This example shows the same method to send the file as in Chapter 8, with the addition of:

1. Processing the .tbd file into a Pandas DataFrame using numerous functions
2. Saving the Pandas DataFrame containing all the file data as a netCDF.nc
3. Sending the processed .nc file from the external controller to the science computer

9.3.1. Processing .tbd files onboard the EC

```

1 # we save the file as in Chapter 8 (all files for this example are on GitHub)
2 # this is still within the while loop
3
4 file = '/home/rutgers/backseat_driver/from_glider/' + tbd_file_name
5
6 if (os.path.isfile(file) > 0):
7     # must have the cac files for the glider on the RP
8     tbd, tbd_meta = dbd_to_dict(file, 'rutgers-cac/')
9
10    # imports the data into Pandas DataFrame using proper time for type of file
11    tbd_data = pd.DataFrame.from_dict(tbd, orient='columns', dtype=None, columns=None).
12        set_index('sci_m_present_time')
13
14    # cleans up the DataFrame time indices.
15    tbd_data.index = pd.to_datetime(tbd_data.index, unit='s').round(freq='S')
16
17    # make xarray dataset
18    tbd_xr = xr.Dataset.from_dataframe(tbd_data)
19    xr_file_name = f'{tbd_file_name}.nc'
20
21    # Save the Dataset to a NetCDF file
22    tbd_xr.to_netcdf(to_dir+xr_file_name)
23
24    ncname = os.path.basename(to_dir+xr_file_name)
25
26    #send nc from pi to glider
27    x.send_file(port, "/home/rutgers/backseat_driver/to_glider/" + str(ncname), str(ncname), "
28        logs", 1)
29    #once sent to glider, move processed nc to files_processed folder
30    # os.remove(file)
31
32    elif (os.stat(tbd_file_name).st_size == 0):
33        print(tbd_file_name, 'created, but data did NOT write to file.')
34        os.remove(tbd_file_name)
35    elif not os.path.isfile(tbd_file_name):
36        print('Failed to create', tbd_file_name)
37        os.remove(tbd_file_name)
38        print('Finished')
39    else:
40        print('File does not exist.')

```

File 9.8: Example C: Processing .tbd files onboard the EC

10

Alternative External Controllers

Practically every computer that supports serial (UART) communication could be used as an external controller; thus, there are many possible choices, all with trade-offs from one another. Because I have thus far been unable to get hands-on with the following, this list is rough at best. Please use it only as a starting point.

1. **Nvidia Jetson Nano:** <https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-nano/product-development/>
2. **BeagleBone Black:** <https://www.beagleboard.org/boards/beaglebone-black>
3. **Odroid:** <https://www.hardkernel.com/shop/odroid-m1-with-4gbyte-ram/>
4. **Teensy 4.1:** <https://www.pjrc.com/store/teensy41.html>

A good resource to read more about UART communication is <https://docs.arduino.cc/learn/communication/uart/>. While that contains Arduino-specific information, the generic UART info is displayed in a user-friendly manner.

11

Conclusion

In this manual we've gone over many aspects of using the backseat driver software onboard a G3S Slocum glider, or it's simulation counterpart, the shoebox simulator. There are undoubtedly many many more ways to use the backseat driver architecture (part of the reason I find this work so interesting) that is limited only by our imagination and ingenuity.

That said however, it can be complicated to work through how exactly to best utilize this control scheme. If you are lacking someone to talk with about this, I am more than happy to lend an ear. Please email or reach out in any of the forms listed on my website or in the main GitHub readme.

sophiescopazzi.com

Lastly, if you've made it this far, I hope your work is going well! If you've found this manual helpful, but there is anything you wish was included I'd love to hear about it.

A

Most Optimal Heading Explanation

As discussed in [Chapter 7](#), the standard practice for glider piloting when crossing strong currents is to put a waypoint far away in the desired direction of travel; i.e., orthogonal to the current direction (the East waypoint in [Figure 3.4](#)). This leads to a mostly consistent commanded heading as in [Figure 5.4](#).

The thought process behind this methodology is to spend as little time as possible in the fast moving current. Piloting in this manner can lead to at least two issues: What to do when 1) the mission's objective is to sample the fast moving current and 2) a shorter deflection distance compared to 90° from the current is desired.

There are additional questions arising from these two issues: what is the optimal heading for crossing ocean currents? We used 90° to the right of the DAC for our BSD program, but is that the "most optimal" heading to use? What can "most optimal" mean anyway?

A.1. Deflection Distance

This discussion is predicated upon the understanding that the glider is slower than the ocean current, because if the glider is faster than the current it can fly anywhere commanded. To better understand this works definition of deflection distance, we begin with an idealized situation of a boat crossing a river. The current is flowing due north and the boat is traveling west to east. Observing the boat from above, top down with north upwards and east to the right, like with a nautical chart, we can watch how its path would bend northward as it set off on its journey.

When attempting to traverse the ocean there is an idealized location the glider would arrive at if there was no current - what I've labeled "ideal end" in figure 7.1. However, with an ocean current the glider, when commanded with a heading of 090°T deflects from the ideal path ([Figure A.1](#), idealized paths shown in red and green).

Despite having the shortest possible time ([Figure A.3](#)) in crossing the current 90 degrees will also have a greater deflection distance than turning partway into the current. Thus, the question becomes how far to turn into the current before we get diminishing returns. This will depend on glider speed and heading, and ocean current and magnitude.

Because the glider is turned more into the current the deflection distance at a heading of 120°T is less than with a heading of 090°T; however, the former takes more time while the latter is the shortest time to cross the current ([Figure A.2](#)). Keep in mind that a heading perpendicular to the current will always be the shortest time.

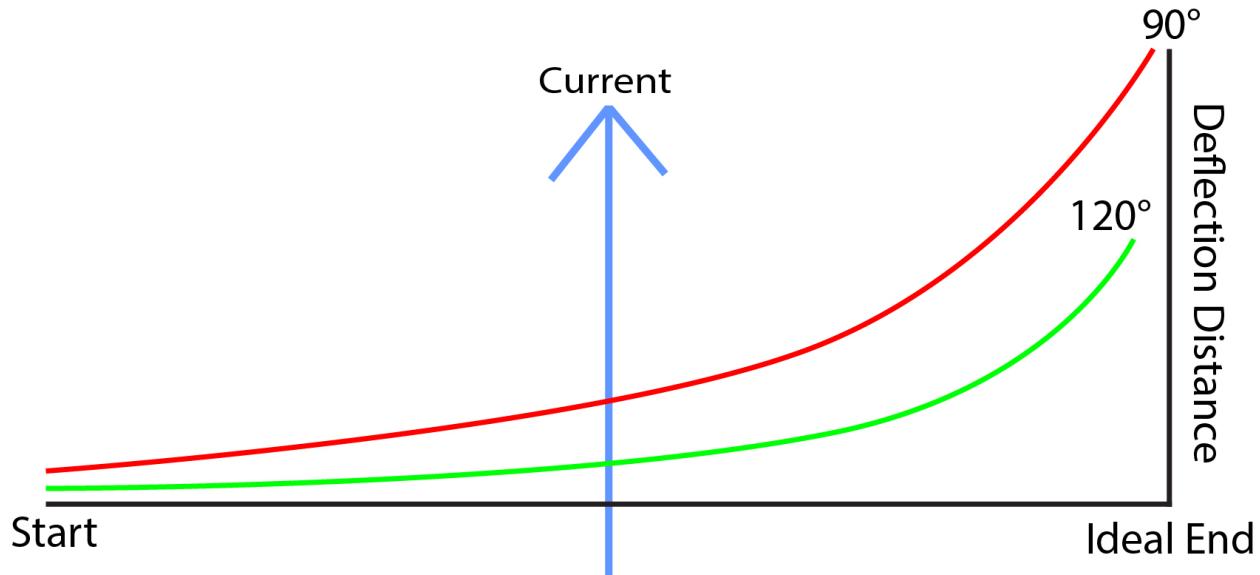
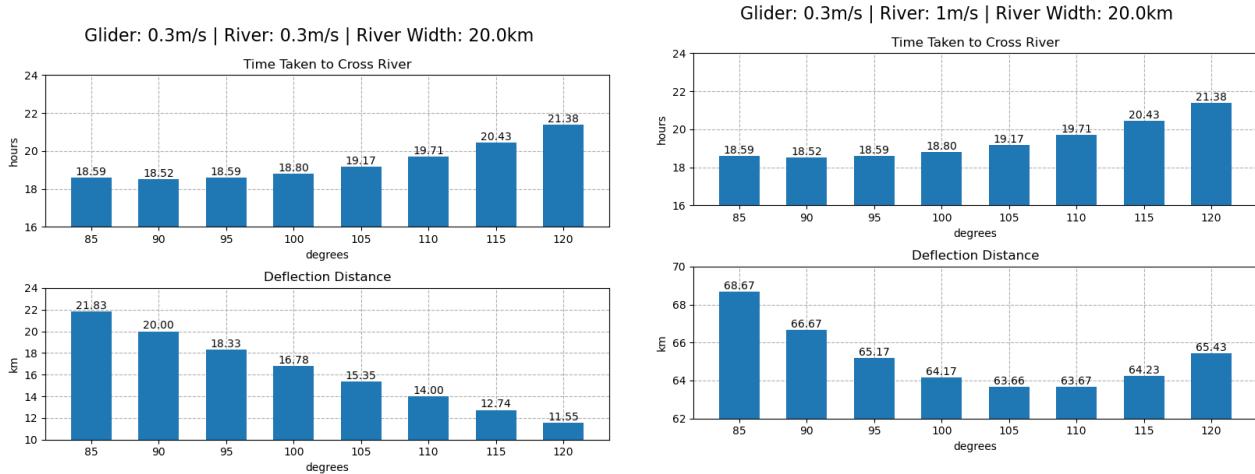


Figure A.1: Idealized for illustration purpose only. Red is heading of 090°T and green is heading of 120°T



(a) When the glider speed is the same as the current it is possible to pick any desired deflection distance, with the trade-off of far more time. The deflection distance heads from infinity towards zero, while the time taken exponentially increases

(b) When the glider speed is a fraction (in this case 33%) of the ocean current the time taken to cross remains the same with each heading as the fraction of east/west velocity remains the same. What changes, however, is the deflection distance northward as the north/south velocity of the glider fights against, or runs with, the river's current

Figure A.2: Two cases: the glider's velocity is constant while the river's velocity changes



Figure A.3: All heading angles 000°T - 180°T

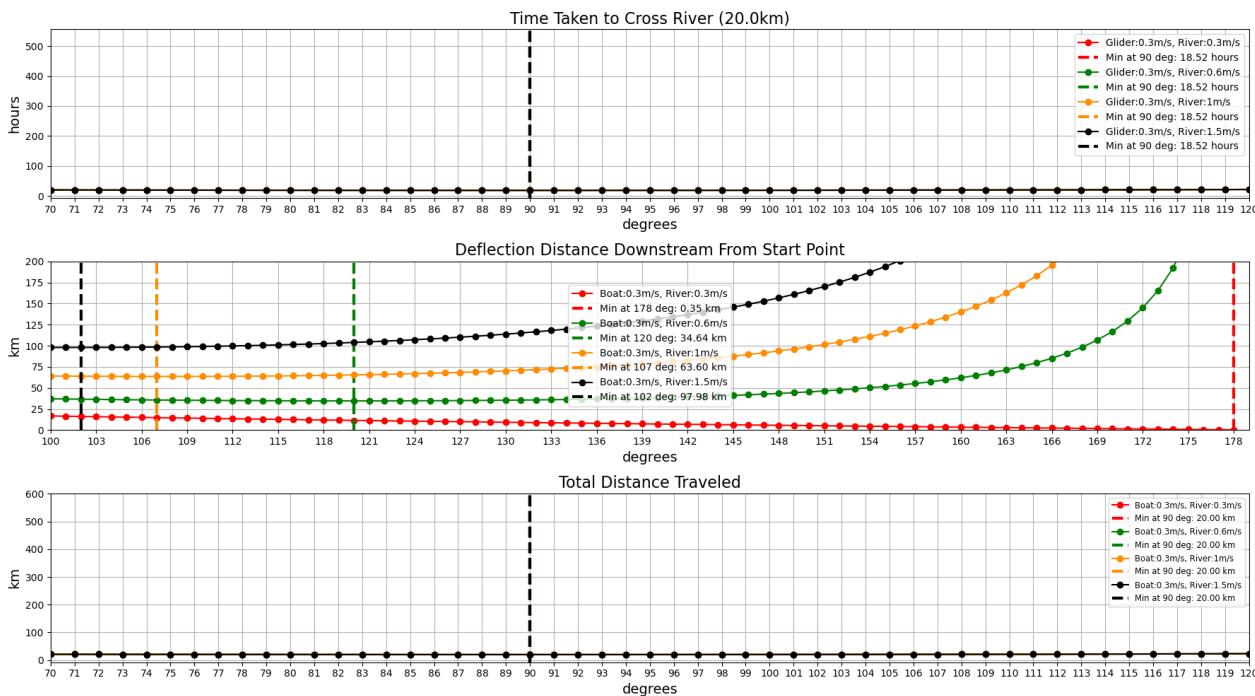


Figure A.4: Heading angles 100°T - 180°T (same plot as Figure A.3 above, but more zoomed in)

Top: The shortest time, regardless of glider or ocean velocity, is always a heading of directly across the river (90° from the current direction) due to the entirety of the boat's velocity residing in cross-current direction.

Middle: The boat velocity remains the same while the river's velocity changes. As the river velocity continues to increase the heading for minimum deflection distance downstream approaches 90°. The orange and black case are most representative of a real-life scenario, showing an optimal heading for least deflection is approximately a heading of 100°-110°.

Bottom: The shortest distance traveled over the ground (COG) is always shortest at a heading of 90° from the current direction.

A.2. Calculating the Most Optimal Heading

As established, if the mission is to get across the current with the shortest time possible, perpendicular to the current will always be the fastest. In the case of the shortest time, the deflection distance doesn't matter. However, if the mission is to have the shortest deflection distance possible (time is less a factor) the most optimal heading now depends on the glider's vector and current's vector. Using both vectors we can calculate the heading that results in the smallest deflection distance.

A.2.1. Python function: def what_is_best_heading

```

1 def what_is_best_heading(underwater_time, m_avg_speed_mps, m_heading_rad, m_water_vel_dir_rad,
2                         m_water_vel_mag_mps):
3     """
4     Calculate the best heading based on input parameters.
5
6     Parameters:
7     - underwater_time (float): Time spent underwater in hours (prior leg time, if using in a
      glider)
8     - m_avg_speed_mps (float): Average speed of the glider in meters per second
9     - m_heading_rad (float): Glider's heading in radians
10    - m_water_vel_dir_rad (float): Water velocity direction in radians
11    - m_water_vel_mag_mps (float): Water velocity magnitude in meters per second
12
13    Returns:
14    - final_vel_mag_mps (float): Magnitude of the final velocity in meters per second
15    - final_vel_dir_rad (float): Direction of the final velocity in radians
16    - deflection_distance_km (float): Distance glider would travel at m_heading_rad in km
17    """
18    # glider vel components (horiz and vert)
19    glider_vx_mps = m_avg_speed_mps * math.sin(m_heading_rad) # m_heading is radians
20    glider_vy_mps = m_avg_speed_mps * math.cos(m_heading_rad) # take average of last x number
      somehow???
21
22    # water vel components (horiz and vert)
23    water_vx_mps = m_water_vel_mag_mps * math.sin(m_water_vel_dir_rad) # m_water_vel_dir is
      radians
24    water_vy_mps = m_water_vel_mag_mps * math.cos(m_water_vel_dir_rad)
25
26    # combine glider and water velocities
27    final_vx_mps = glider_vx_mps + water_vx_mps
28    final_vy_mps = glider_vy_mps + water_vy_mps
29
30    # calculate the magnitude of the resultant velocity
31    final_vel_mag_mps = math.sqrt(final_vx_mps ** 2 + final_vy_mps ** 2)
32
33    # calculate the direction of the resultant velocity
34    final_vel_dir_rad = math.atan2(final_vy_mps, final_vx_mps)
35
36    # what is new estimated distance?
37    new_dist_m = final_vel_mag_mps * 3600 * underwater_time # underwater time to seconds
38
39    # d=s*t --> t=d/s, used for calculating distance value
40    deflect_time_sec = new_dist_m / final_vel_mag_mps
41
42    # Calculate the deflection distance
43    deflection_distance_km = (final_vel_mag_mps * deflect_time_sec)/1000
44
45    return final_vel_mag_mps, final_vel_dir_rad, deflection_distance_km

```

File A.1: what_is_best_heading - Python function used to calculate heading

This Python code function is used in [Chapter 7](#) onboard the external controller as an example at using the BSD capability of the glider to make real-time onboard decisions about heading based upon the observed ocean current vector.

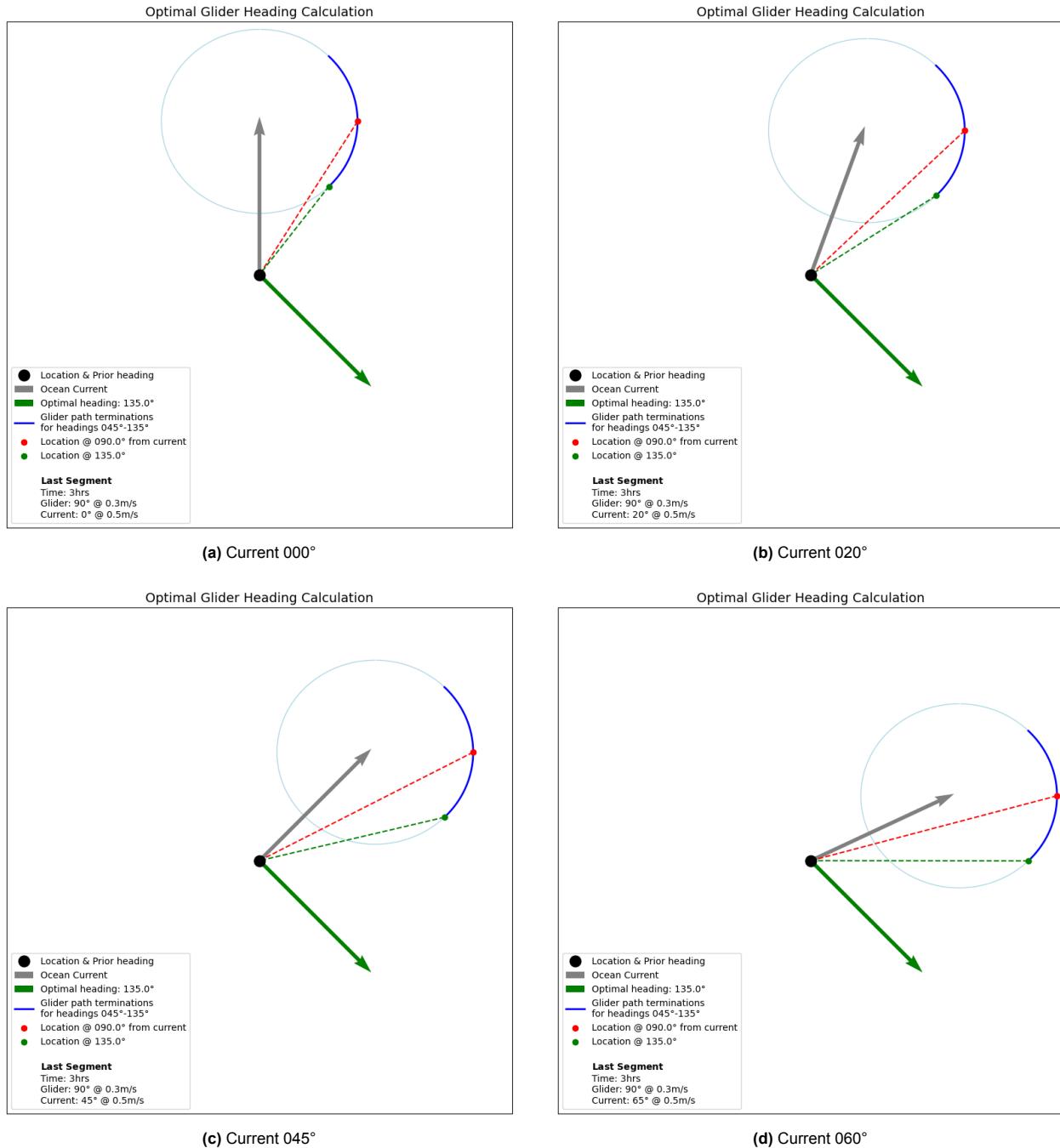


Figure A.5: Gray arrow indicates current direction. Green arrow indicates optimal heading with green dashed line as the resultant course over ground (COG) and 3hr location. Red dashed line indicates the COG for the heading 090°T to the right of the current and 3hr location. Light blue circle is resultant location for all headings 000°T - 360°T. Dark blue section headings of 045°T - 135°T. For all cases of current direction 335°T - 090°T, a heading of 135°T results in less deflection northward (while still retaining eastward velocity) than headings 90° to the right of the current.