Generic RIS Controller Detailed Development Document

### **Contents**

1. Introduction........................................................................................................... 4

2. Architecture............................................................................................................ 5

3. Work Flow.............................................................................................................. 7

3.1Initialization................................................................................................ 7

3.2 Operation................................................................................................... 7

4. Key Functionalities................................................................................................ 9

4.1 CE.py......................................................................................................... 9

4.1.1 Functional Details........................................................................ 9

4.1.2 API and Interfaces...................................................................... 11

4.2 LUT.py..................................................................................................... 12

4.2.1 Functional Details....................................................................... 12

4.2.2 API and Interfaces...................................................................... 13

4.3 GRC.py.................................................................................................... 14

4.3.1 Functional Details....................................................................... 15

4.3.2 API and Interfaces...................................................................... 16

4.4 RSC.py..................................................................................................... 17

4.4.1 Functional Details....................................................................... 17

4.4.2 API and Interfaces...................................................................... 19

5. Deployment and Execution ................................................................................. 21

### **List of Figures**

1. Testbed Architecture .............................................................................................. 6

2. Message flow in Lookup Table Calibration........................................................... 7

3. Generic RIS Control Operation Message Flow .................................................... 8

### **List of Tables**

1.API\_CE ................................................................................................................ 11

2. API\_LUT............................................................................................................ 13

3. API\_GRC ............................................................................................................ 16

4. API\_RSC ............................................................................................................. 19

### **Abbreviations**

1. RIS- Reconfigurable Intelligent Surface (RIS)

2. CE- Channel Estimator

3. GRC- Generic RIS Controller

4. RSC – RIS Specific Controller

5. LUT - Lookup table

6. API – Application programming Interface

### **1. Introduction**

This document outlines a comprehensive testbed for Reconfigurable Intelligent Surface (RIS) control. The testbed facilitates the following objectives:

1. RIS Fabricators to perform integration testing of their hardware with existing technologies and control algorithms without the need to develop the communication and control infrastructure,
2. Communications researchers, especially those working on Channel Estimators, can try out their RIS-aware channel estimation algorithms, without worrying about the control aspects and the RIS hardware interfacing,
3. Researchers working on autonomous/assisted RIS control using TSDSI interfaces to validate their algorithms without needing to develop infrastructure for communication and RIS.
4. The findings of the various users to be shared with TSDSI to enable enhancements in the standards.

Scope**:**  
This document describes the implementation details of the Generic RIS Controller Application, that will consist of four major components: Channel Estimator , Lookup table, Generic RIS Controller, and RIS specific controller.

This document will also guide external developers in integrating their RIS specific algorithms using the Generic RIS Controller. This document includes refined schematics and detailed descriptions of system components and user integration for the RIS test bed which follows TSDSI architecture.

### **2. Testbed Architecture**

### The testbed integrates hardware and software components to iteratively find the best beamforming angle and apply corresponding configurations. The core components communicate through REST APIs for modular functionality. The components of the testbed and their roles are:

**1.** Channel Estimator (**CE.py**) : Provides the current channel estimate via and API. The default example implementation measures the signal magnitude from a receiver and provides it via an API.

Features**:**

* + - Computes signal magnitude from complex data from receiver, in real time.
    - Exposes the magnitude via a Flask API.

**2.** Generic RIS Controller (**GRC.py**): Determines the optimal control command parameters (TSDSI commands) using proprietary algorithms, while interacting with the Channel Estimator module via the provided API. Retreives and exposes data via API.

Features**:**

* + - Retrieves valid angles from the RIS, through capability query.
    - Sends each angle to the RIS specific controller module, for RIS configuration.
    - Measures the magnitude using CE.py.
    - Finds and sends the best angle for RIS configuration via RIS specific controller module, using API.

### 3. **RIS Specific Controller (**RSC.py**)**: Acts as an end point for consuming the TSDSI commands for configuration/capability read/write commands and issues appropriate RIS hardware-specific electrical signals corresponding to the configuration write commands received from Generic RIS controller. Handles RIS configurations and communicates with the hardware using serial communication.

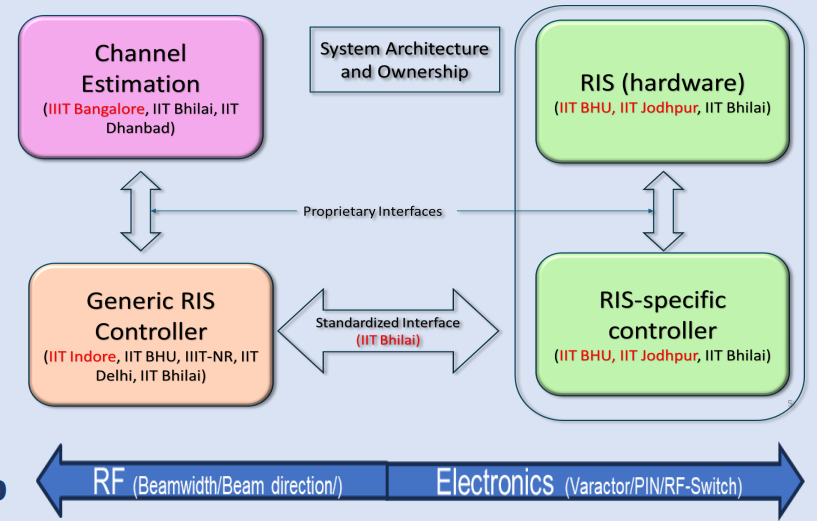
Features**:**

* + - Queries valid beamforming angles from a lookup table (LUT).
    - Configures RIS based on the received beamforming angle, as per the corresponding configuration in the lookup table.
    - Sends configuration patterns to RIS via serial communication.

**4.** Lookup Table Generator (**LUT.py**)**:** RIS hardware provider needs to use this to populate the translation table required to map the TSDSI commands parameters (received from the GRC) to the electrical signals to be sent to the RIS hardware. Populates a LUT with beamforming angles and corresponding patterns to maximize signal magnitude.

Features**:**

* + - Iterates through possible beamforming angles.
    - For each angle, generates random patterns and evaluates their performance.
    - Stores the optimal angle-pattern pair in the LUT.



**Figure 1: Testbed Architecture**

**3. Workflow**

A typical example interaction among the various entities described above is provided in this section.

3.1 Initialization**:**

* + - Run CE.py to start the magnitude measurement service.
    - Use LUT.py to populate the LUT by determining optimal beamforming patterns for different angles.

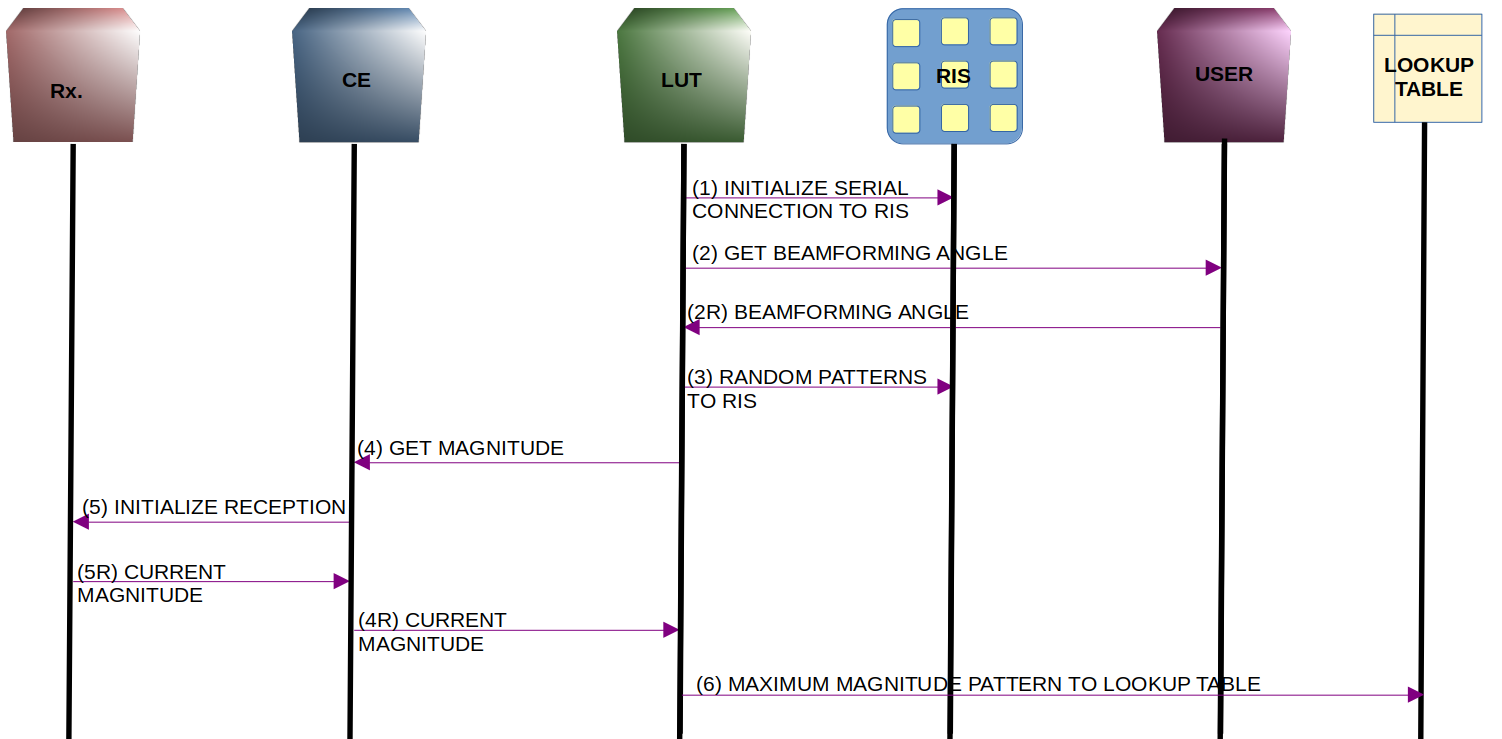
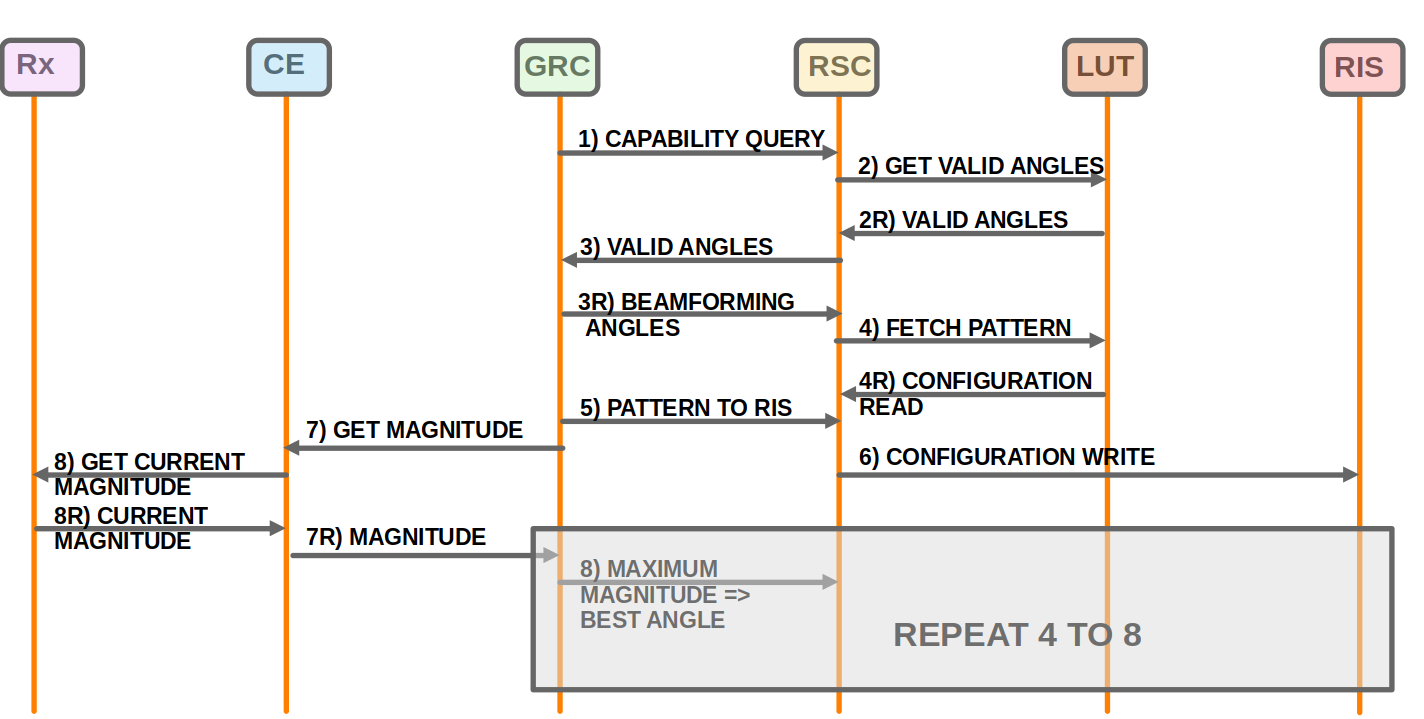


Figure 1: Lookup Table Calibration Message Flow

Figure 2: Message flow in Lookup Table Calibration

3.2 Operation**:**

* + - Run CE.py to start the magnitude measurement service.
    - Start RSC.py to handle RIS hardware configuration.
    - Run GRC.py to optimize and apply beamforming angles dynamically during operation.

**Figure 3: Generic RIS Control Operation Message Flow**

### 4. **Key Functionalities**

4.1 CE.py

CE.py is designed to handle channel estimation (CE) in environments utilizing Reconfigurable Intelligent Surfaces (RIS). It serves two key purposes:

Lookup Table Calibration: Integrates with LUT.py to pre-calculate channel responses and store them for later use in RIS optimization.

Runtime Usage in GRC: Exposes magnitude of signal, for real-time updates and adjustments in RIS configurations during operations.

Channel Estimation Algorithm:

Calculates the channel response between transmitter, RIS, and receiver.

Uses inputs like signal strength, beamforming angles, and path loss models.

Inputs and Outputs:

Inputs:

Transmitter/receiver coordinates.

Signal properties (e.g., frequency, power).

Outputs:

Estimated channel response.

##### **4.1.1 Function Details:**

##### 1. \_\_init\_\_

Class constructor : This method initializes a class, which represents a signal receiver flowgraph. The flowgraph includes configuration for the source device and GNU Radio blocks to process received signals.

#### Key Features:

* Configures the sample rate.
* Sets up a source (uhd\_usrp\_source\_0) to receive signals:
* Configures the center frequency and gain.
* Creates and connects GNU Radio processing blocks:
* Converts received complex signals to their magnitude (complex\_to\_mag).
* Probes the magnitude using a signal probe (probe\_signal\_f).
* Lays the foundation for real-time signal analysis.

##### Parameters:

* None (the method automatically initializes class attributes).

### 2. get\_current\_magnitude

Fetches the current magnitude of the signal being received: This method retrieves the current magnitude of the signal from the blocks\_probe\_signal\_f probe block. It is primarily used to monitor and log real-time signal strength.

#### Key Features:

* Logs the current signal magnitude.
* Returns the real-time signal magnitude as a floating-point value.

#### Returns:

* float: The magnitude of the received signal.

#### Usage:

This function is typically called during the execution of the flowgraph to fetch the instantaneous magnitude of the signal.

### 3. get\_magnitude

Exposes signal magnitude via a Flask API endpoint: This Flask route handles HTTP GET requests to /api/v1/magnitude and returns the real-time magnitude of the received signal in JSON format.

#### Key Features:

* Calls get\_current\_magnitude() from the constructed class to fetch the signal magnitude.
* Returns the magnitude as part of a JSON response.

#### Returns:

* JSON: A JSON object containing the magnitude:

{ "magnitude": <current\_signal\_magnitude> }

**Example Request:** curl http://<server\_ip>:5000/api/v1/magnitude

**Example Response:** { "magnitude": 0.345 }

### 4. run\_flask

Starts the Flask server for the API: This method launches a Flask web server that listens for incoming API requests. It runs in a separate thread to allow simultaneous operation of the GNU Radio flowgraph.

#### Key Features:

* Configures the Flask app to run on host='0.0.0.0' and port=5000.
* Enables multithreaded request handling with threaded=True.
* Provides API access while keeping the main application running.

#### Parameters:

* None (the function runs independently).

#### Exceptions:

* Logs errors if the server encounters issues during startup.

4.1.2. APIs and interfacing in **CE.py**:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **API** | **URL** | **Method** | **Description** | **Response** | **Response Codes** |
| Get Magnitude | /api/v1/magnitude | GET | Fetches the current magnitude of the received signal from the cos\_rx flowgraph.   1. Retrieves live signal magnitude from flowgraph. 2. Provides real-time signal strength measurement. | Success: Returns a JSON object containing the magnitude value.  { "magnitude": <value> }  Error: Returns a server error response (500) if magnitude can’t be fetched due to flowgraph issues. | 200 (success), 500 (server error) |

**Table1:API\_CE**

### Interfaces

### 1. F**lowgraph**

### This GNU Radio flowgraph handles signal reception and processing.

### **Components**:

* Source: Configured to receive signals at particular frequency,gain and a sample rate..
* blocks.complex\_to\_mag: Converts the complex signal to its magnitude.
* blocks.probe\_signal\_f: Probes the signal for its magnitude, which is accessible via the get\_current\_magnitude method.

Functionality:

* Receives and processes signals to compute the magnitude in real-time.
* Provides this magnitude as the output for the API.

2. Flask App

* Serves as the interface for the flowgraph, allowing external systems to query the signal magnitude via HTTP requests.
  + Threading: Runs in a separate thread to ensure non-blocking operation of the flowgraph.

#### Usage:

This function is invoked in a separate thread to ensure that the Flask server does not block the main application.

**Customization and Extension for Advanced Applications**

Developers interested in working on channel estimation for RIS environments can achieve the required functionality by modifying the \_\_init\_\_ function.

E.g.- To adapt CE.py for cellular communication in RIS environments, developers only need to incorporate path loss models (e.g., 3GPP), support MIMO, and account for Doppler shifts. For or WiFi communication, integrate IEEE 802.11 standards and optimize for high-frequency multipath effects (e.g., 5 GHz, 6 GHz). These additions enable the flowgraph to handle multi-user scenarios, long delay spreads, and short-range high-frequency communications, ensuring seamless channel estimation.

4.2 LUT.py

The LUT.py script uses the magnitude to determine optimal beamforming patterns and populates a Lookup Table (LUT). Its Role in the Testbed is

* Signal Processing: Extracts specific parameters (like magnitude) from raw input signals.
  + Data Communication: Interfaces with the testbed by sending data to the LUT for configuration or decision-making.
  + System Integration: Acts as a bridge between raw signal processing and dynamic system adjustments (e.g., RIS beamforming).
* **Reliable support for testbed** applications like RIS configuration.

### **4.2.1. Functional Details:**

#### **1.** get\_magnitude()

#### Processes raw signal data to compute its magnitude.

* Handles input data in complex or raw form.
* Applies mathematical operations (e.g., FFT or absolute value computations).
* Ensures compatibility with subsequent processing steps.

Returns:

* A numeric value representing the computed magnitude of the input signal.

Example Request: magnitude = get\_magnitude(signal\_data)

Example Response: 25.6 # Magnitude of the signal

Usage:  
Call this function to compute magnitude before sending data to the LUT.

2. **send\_to\_lut()**

Sends computed magnitude or related parameters to the LUT server via a REST API.

* Formats data into the required JSON structure.
* Sends an HTTP POST request to the LUT server.
* Handles responses, ensuring the server has processed the data successfully.

Returns:

* A status code indicating success (e.g., 200) or failure (e.g., 400, 500).

Example Request: status = send\_to\_lut({"magnitude": magnitude, "angle": beamforming\_angle})

Example Response: json { "status": "success",

"message": "Data received and processed."}

Usage:  
This function populates LUT with processed parameters and supports calibration of RIS with optimal beamforming patterns for improved signal reception.

### **4.2.2 APIs and Interfaces Used in** LUT.py

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **API** | **URL** | **Method** | **Description** | **Response Codes** |
| **Magnitude Endpoint** | /get\_magnitude | GET | Retrieves magnitude values from a signal data source. | 200 (success), 400 (invalid request) |
| **LUT API** | /send\_to\_lut | POST | Sends magnitude and/or beamforming angle to LUT server for configuration. | 200 (success), 500 (server error) |
| **Logging API** | /log\_status | POST | Logs the status of LUT operations to a centralized monitoring server. | 200 (success), 400 (bad request) |

**Table2: API\_LUT**

##### **Endpoints Consumed**:

* Magnitude API: GET {magnitude\_url}: Fetches signal magnitude data from CE.

Functionality:

1. Serial Communication:

* Port: /dev/ttyUSB0
* Baud Rate: 115200

Used to send random patterns to the RIS hardware for configuration.

1. Pattern Evaluation:
   * Generates random 16-character patterns.
   * Sends the pattern to the RIS.
   * Retrieves magnitude for each pattern using the Magnitude API.
   * Compares magnitudes to find the best pattern.

##### **LUT Population**:

* Stores beamforming angle and the best pattern in a text file.
* Example- beamforming\_angle, best\_pattern : 90, !0XFFFF0000FFFF0000

1. **Customization and Extension for Advanced Applications**
2. The resolution of beamforming angles and the resulting beamforming patterns can be configured based on the specifications of the RIS board. The communication interface with the RIS board is also board-specific; for example, it can utilize a serial port, as in the current setup, or alternative modes such as Bluetooth or other protocols, depending on the implementation requirements.

**4.3 GRC.py**

Manages beamforming optimization.

* Query Valid Beamforming Angles: Interacts with RSC.py to fetch valid beamforming angles from the lookup table.
* Select Optimal Angle: Iterates through valid angles, sending them to RSC.py to achieve maximum signal magnitude.
* Send Best Beamforming Angle to RSC: Posts the final optimal angle after multiple iterations for RIS configuration.
* Find Most Frequent Angle: Analyzes multiple best angles and determines the most effective one statistically.

### 4.3.1. **Function Details**:

#### 1. get\_magnitude ()

* Retrieves the current signal magnitude from CE.py.

Syntax:get\_magnitude() -> float or None

Parameters: None

Returns:

* float: Signal magnitude if successful.
* None: If an error occurs.

Throws: requests.exceptions.RequestException: For HTTP-related issues.

#### 2. send\_to\_rsc ()

* Sends a beamforming angle to the RSC for configuration.

Syntax: send\_to\_rsc(beamforming\_angle: int) -> dict or None

Parameters: beamforming\_angle (int): Angle to configure the RIS.

Returns:

* dict: Response from the RSC API.
* None: If an error occurs.

Throws: requests.exceptions.RequestException: For HTTP-related issues.

#### 3. get\_valid\_angles ()

* Queries the RSC for valid beamforming angles.

Syntax: get\_valid\_angles() -> list[int] or None

Parameters: None

Returns:

* list[int]: List of valid angles.
* None: If an error occurs.

Throws: requests.exceptions.RequestException: For HTTP-related issues.

#### 4. find\_most\_frequent\_angle ()

* Determines the most frequent angle in a list.

Syntax: find\_most\_frequent\_angle(angles: list[int]) -> int or None

Parameters: angles (list[int]): List of beamforming angles.

Returns:

* int: The most frequent angle.
* None: If the list is empty.

### 4.3.2. **APIs and Interfaces**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **API** | **Method** | **Endpoint** | **Purpose** | **Response Codes** |
| **Magnitude API** | GET | /api/v1/magnitude | Retrieve signal magnitude: provided by CE.py | 200: Success 500: Internal Error |
| Capability Query | GET | /api/v1/configuration\_write/capability\_query | Query valid beamforming angles, provided by: RSC.py | 200: Success 500: No Angles Found |
| **RIS Configuration Write** | POST | /api/v1/configuration\_write | Send beamforming angle configuration, provided by: RSC.py | 200: Success |

Table 3: API\_GRC

### **Endpoints Consumed**

1. /api/v1/magnitude:

* Consumed by: GRC.py, LUT.py
* Functionality: Provides real-time signal magnitude from CE.py.

1. /api/v1/configuration\_write/capability\_query:

* Consumed by: GRC.py
* Functionality: Provides a list of valid beamforming angles.

1. /api/v1/configuration\_write:

* Consumed by: GRC.py

Functionality: Accepts the best beamforming angle for RIS configuration.

#### **Integration**:

* CE.py: Magnitude source for calibration and beamforming.
* LUT.py: Ensures RIS patterns and angles are optimized.
* GRC.py: Orchestrates beamforming, making decisions based on magnitude and RIS capabilities.
* RSC.py: Manages the physical RIS configuration.

**Customization and Extension for Advanced Applications:**

The GRC.py code is a generic and modular framework designed for flexibility across different RIS implementations. Developers can modify the channel estimator module to align with the communication system in use, such as cellular or WiFi. Similarly, the RSC.py module can be replaced with one specific to the RIS system being used. The serial communication mode implemented here is also adaptable and can be configured based on the communication method required for the specific RIS setup.

4.4 RSC.py

The RSC.py script facilitates communication between the RIS controller and the RIS board. Handles RIS configuration via serial communication based on optimal beamforming angles.

* Capability Query: Identifies valid beamforming angles from the LUT.
* RIS Configuration: Sends beamforming angles and associated patterns to the RIS.

### **4.4.1 Function Details:**

#### 1. **capability\_query( )**

#### Handles a GET API request to provide a list of all valid beamforming angles stored in the lookup table (LUT).

Process:

* Calls extract\_angles\_from\_lut() to get a list of unique angles.
* Returns the angles in a JSON response if successful.
* Sends an error response (500) if the LUT cannot be read or contains no valid angles.

Endpoint: /api/v1/configuration\_write/capability\_query

#### 2. **extract\_angles\_from\_lut( )**

#### Reads the LUT file and extracts all beamforming angles.

Process:

* Opens the file specified by lookup\_table\_file.
* Parses each line to separate the angle and pattern.
* Collects all angles in a set (to ensure uniqueness).
* Returns a list of unique angles or None if the file can't be read.

Parameters: None (uses the LUT file path defined globally).

Returns:

* list[int]: List of unique beamforming angles.
* None: If the LUT is missing or contains errors.

#### 3. **send\_to\_ris ()**

#### Sends the beamforming pattern corresponding to a given angle to the RIS hardware using serial communication.

Process:

* Connects to the RIS device via a serial port (/dev/ttyUSB0 by default).
* Clears the serial buffers and sends the beamforming pattern.
* Waits for and logs a response from the RIS (or logs an error if no response is received).

Parameters:

* beamforming\_angle: The angle to configure (for logging purposes).
* best\_pattern: The beamforming pattern to send.

Returns: None (logs success or failure)

#### 4. **configure\_ris ( )**

#### Handles a POST API request to configure the RIS with a specific beamforming angle.

Process**:**

* Receives the requested angle from the API payload.
* Calls fetch\_best\_pattern\_from\_lut() to retrieve the corresponding beamforming pattern.
* Invokes send\_to\_ris() to apply the pattern to the RIS.
* Returns a success response (200) if the configuration is applied successfully.
* Returns an error response (404) if the angle is not found in the LUT.

Endpoint**:** /api/v1/configuration\_write.

5. fetch\_best\_pattern\_from\_lut ( )

#### **Finds the beamforming pattern corresponding to a specific angle in the LUT.**

Process:

* Opens the LUT file.
* Iterates through each line to match the given angle.
* Returns the associated beamforming pattern if found.
* Logs an error if the LUT is missing or unreadable.

Parameters: beamforming\_angle (int): The angle for which to retrieve the pattern.

Returns:

* str: The beamforming pattern for the angle.
* None: If the angle is not found or an error occurs.

#### 6. **find\_most\_frequent\_angle ( )**

#### Identifies the most frequently occurring angle in a list of beamforming angles.

#### **Syntax**: find\_most\_frequent\_angle(angles: list[int]) -> int or None

Parameters**:** angles (list[int]): A list of beamforming angles to analyze.

Returns:

int: The angle with the highest frequency.

None: If the list is empty.

### **4.4.2 APIs and Interfaces Table**

|  |  |  |  |
| --- | --- | --- | --- |
| **API Endpoint** | **Purpose** | **Input** | **Output** |
| /api/v1/configuration\_write/capability\_query | Returns valid beamforming angles from LUT | None | {"valid\_angles": [...] } |
| /api/v1/configuration\_write | Configures RIS with a given beamforming angle | { "beamforming\_angle": ... } | {"status": "Success" } |
| /api/v1/magnitude | Provides real-time signal magnitude | None | { "magnitude": ... } |

### Table 4 : API\_RSC

### **Consumed Endpoints**

1. /api/v1/configuration\_write/capability\_query: Consumed by GRC.py to retrieve valid angles for testing.
2. /api/v1/configuration\_write: Used by GRC.py to apply the best angle and pattern configuration to the RIS.
3. /api/v1/magnitude: Accessed by CE.py and LUT.py for real-time magnitude measurements during calibration and optimization.

### **RIS Board Communication**

The RIS board communicates through serial connections (/dev/ttyUSB0). Instructions include:

* Beamforming Pattern: Sent as a string to configure RIS elements.
* Response Handling: Receives acknowledgment or errors from the RIS.
  + 1. Customization and Extension for Advanced Applications
    2. The RSC.py module is designed to work seamlessly with the GRC.py framework, facilitating smooth interaction with any RIS board and its specific control systems. While the current implementation establishes serial communication with the RIS board via /dev/ttyUSB0, the module is adaptable to other hardware setups. Developers can modify the serial communication protocols and endpoints to align with the requirements of their specific RIS hardware. Similarly, the flexible design of GRC.py enables integration with any RIS-specific control program, making it straightforward to interface custom hardware with the existing generic controller architecture.

### **5. Deployment and Execution**

* Environment Setup**:**

Hardware: USRP, RIS, Serial Connection.

Software: Python, Flask, GNU Radio.

* Steps to Run**:**

1. Start CE.py to provide magnitude values.
2. Execute LUT.py to generate the LUT.
3. Run GRC.py to determine beamforming angles.
4. Use RSC.py for configuring RIS based on angles.