

Autonomous Waypoint Navigation Software for Raspberry Pi 5

Shawn Kim

Introduction

This software program enables autonomous waypoint navigation for a mobile platform using a Raspberry Pi 5.

The system uses GPS data to guide the platform from one location to another along a predefined series of static waypoints.



Requirements Coverage



Requirements Coverage: Requirements Matrix

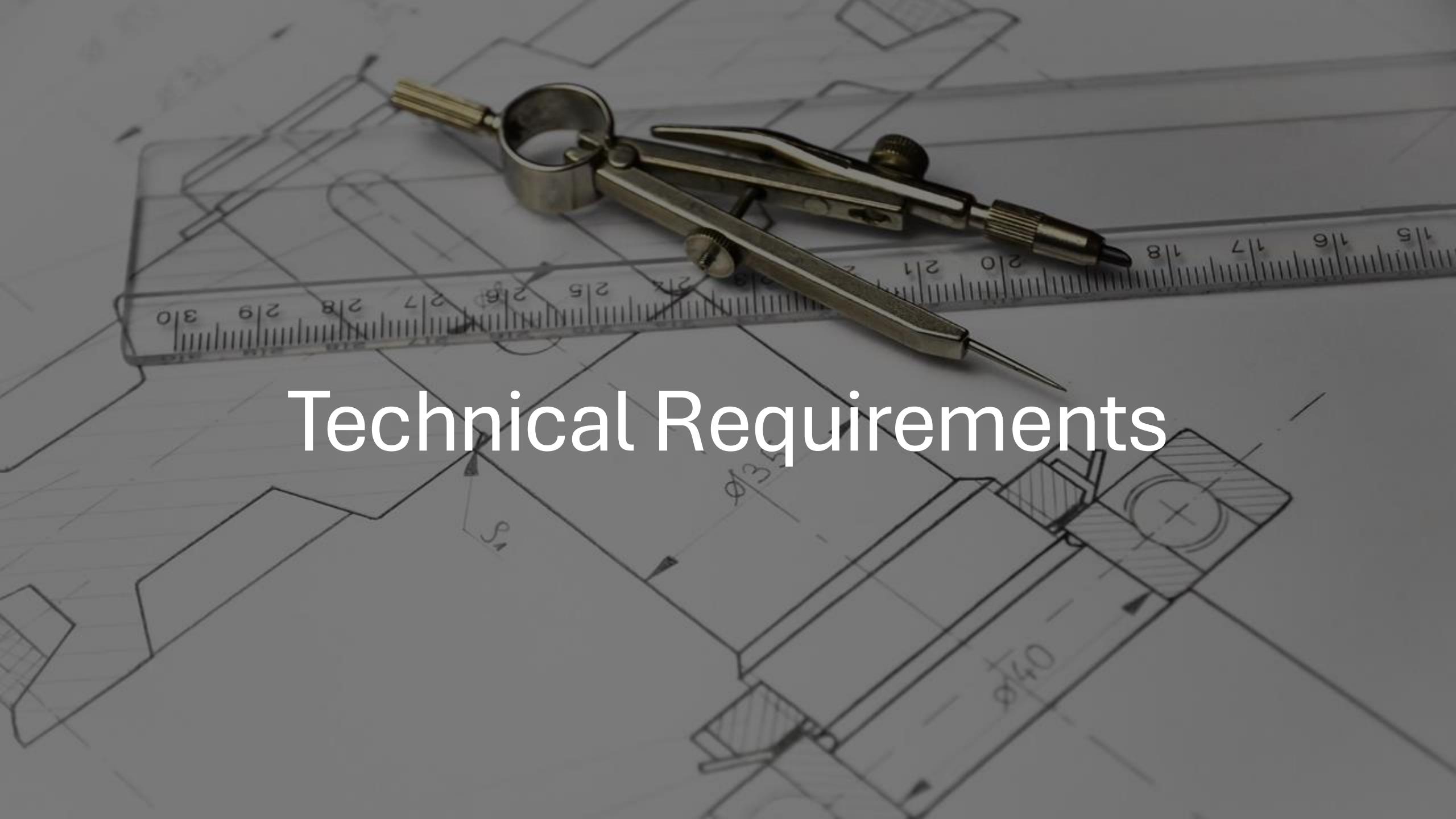
| Requirement | Status |
|--|-------------|
| Interface with GPS module via USB or GPIO serial | Implemented |
| Calculate real-time position and heading from GPS input | Implemented |
| Develop waypoint-following algorithm | Implemented |
| Implement proximity detection for waypoint arrival | Implemented |
| Output navigation heading and velocity commands | Implemented |
| Configurable waypoints via JSON or CSV file | Implemented |
| Documentation on waypoint file interpretation | Implemented |
| Fault detection for GPS signal loss | Implemented |
| Log navigation decisions with timestamps | Implemented |
| Use Raspberry Pi 5 with 64-bit Debian-based OS | Implemented |
| Programming language Python 3 or C++ | Implemented |
| Use pyserial, pynmea2, or gpsd for GPS communication | Implemented |
| Navigation math with numpy | Implemented |
| Optional visualization with matplotlib or folium | Deferred |
| Provide file I/O for waypoints and logs | Implemented |
| Optional CLI or GUI for system control | Deferred |
| Complete software package tested on Raspberry Pi 5 | Implemented |
| Provide biweekly status reports | Implemented |
| Documentation: code comments, README, setup and troubleshooting instructions | Implemented |
| Test Report for navigation between at least 3 waypoints | Implemented |
| Presentation of completed software | Implemented |

Requirements
Coverage:
References

Statement of Work for
Taewoo_051425.docx

Autonomous
Waypoint Navigation
System Test
Report.docx

Technical Requirements



Technical Requirements: Tool Stack

Platform: Raspberry Pi 5 OS Lite

Programming Languages: C/C++23, Shell, Python

GPS Driver: gpsd, gpsd-clients

Build Tools: git, cmake, clang/gcc, sanitizer

C++ Libraries: libgps, concorde, nlohmann_json

Python Libraries: matplotlib

Technical Requirements: Tool Stack (cont'd)

Raspberry Pi OS Lite (64-bit Debian Bookworm): Operating system platform for software deployment.

C++: Core programming language for system logic and navigation computations.

Python: Used data visualization of waypoint visitation order

gpsd/gpsd-clients: Daemon to interface and read GPS data from the dongle.

Git: Used to pull repo on Raspberry Pi

Cmake: Used to configure the build and compilation of the C++ project.

Clang/GCC & Address Sanitizer: Used to compile the project and address sanitizers were used as a tool for memory leak detection and ensuring software stability.

Libgps: C++ library to interface with gpsd-managed socket through API.

Concorde TSP Solver: Solver for Traveling Salesman Problem used for waypoint planning.

Nlohmann_json: C++ JSON library for navigation output.

Matplotlib: Python library to plot planned navigation route.

Technical Requirements: Interfaces

- Navigation software includes CLI to specify required files/directories:
 - ‘.csv’ file/directory to load in user-defined static waypoints
 - ‘.tsp’ directory to communicate with Concorde TSP Solver
 - ‘.sol’ directory to write out Concorde visiting order solution
 - ‘.log’ directory to log navigation output
 - ‘.png’ directory to plot tour of waypoints
- Details of these file interfaces are documented in the Git repository.

Test Automation

Test Automation: Summary

- **Automated Test Coverage:**
 - Over **50% of all required tests** have been automated.
 - Automated tests covers **navigation algorithm correctness**.
 - Tested in the “run” phase of the CI/CD pipeline after the system has been built and installed in Raspberry Pi OS-like environment (Debian Bookworm).
 - Success criteria is the successful completion of the “run” phase.
 - The other tests that were performed manually were navigation logging that demonstrated **GPS input handling, proximity detection, and output reliability**.
- **Execution Results:**
 - Automated tests include waypoint CSV scenarios ranging from **5 to 50 waypoints**.
 - Test results clearly documented in “Autonomous Waypoint Navigation System Test Report.docx”, demonstrating successful and consistent verification.
- **Automation Tools Used:**
 - GitLab CI/CD (.gitlab-ci.yml)

Static Analysis

Static Analysis: Summary

- **Static Analysis Tools:**
 - Utilized compiler-integrated tools (**Clang/GCC Address Sanitizer**) for static and dynamic analysis.
- **Summary of Findings:**
 - **No high or medium-severity issues** identified.
 - All identified minor issues were resolved during iterative development and testing cycles.
- **Risk Assessment:**
 - Final software is assessed as **low-risk**, demonstrating reliability, safety, and stability.
 - Memory leaks, undefined behaviors, and potential crashes thoroughly investigated and mitigated.
- **Evidence of Assurance:**
 - GitLab CI/CD process builds and runs program with a run-time address sanitizer that does not trigger during execution and after completion of program.

CI/CD Pipeline

CI/CD Pipeline: .gitlab-ci.yml

```
1 # Raspberry Pi 5 OS Lite
1 image: debian:bookworm
2 # Define pipeline stages
3 stages:
4   - build
5   - run
6 # Global variables for CMake build configuration
7 variables:
8   CMAKE_BUILD_TYPE: Release
9 # Common setup tasks performed before all stages
10 before_script:
11   - apt-get update && apt-get install -y build-essential cmake libgps-dev git
12 # Build Stage: Compiles and prepares the executable
13 build:
14   stage: build
15   script:
16     # Execute installation script (clone/build/install Concorde libraries and
17     # executable, install Python libraries and script, build/install Navigation
18     # System executable)
19     - ./scripts/install.sh
20   artifacts:
21     # Store compiled binary as an artifact for later stages
22     paths:
23       - app/build/awns-rpi5
24     expire_in: 1 week
25 # Run Stage: Executes tests using the built executable
26 run:
27   stage: run
28   dependencies:
29     - build
30   script:
31     # Ensure the executable has the necessary permissions
32     - chmod +x app/build/awns-rpi5
33     # Provide directory paths required by the executable via standard input to
34     # run waypoint route solver for CSV test suite
35     - |
36       printf "tests/csv\ntests/tsp\ntests/sol\ntests/graph\n" \
37       | ./app/build/awns-rpi5 solve
```

Software Bill of Materials

Software Bill of Materials



Project Name: Autonomous Waypoint Navigation Software



Platform: Raspberry Pi 5 (Debian-based OS)



Author of SBOM: Shawn Kim (Software Developer)



SBOM Assembly Date: June 25, 2025

SOBM: Project-developed Software

| Component | Version | Supplier | Libraries/Dependencies | Dependency Relationship | License | Notes |
|----------------------------|---------|----------------|-----------------------------------|-------------------------|-----------------------|------------------------|
| Autonomous Navigator (C++) | v1.0.0 | Team-developed | STL, filesystem, chrono, iostream | Core Application | MIT (project default) | Primary software logic |

SOBM: Open-source and Third-party Components

| Component | Version | Supplier/Source | Libraries Used | Dependency | License | Known Vulnerabilities | Notes |
|---------------------|----------|-------------------------------------|---|-----------------------|---------------------------------------|-----------------------|---|
| Concorde TSP Solver | 03.12.19 | Concorde TSP | Standalone Binary (no direct API/library calls) | Waypoint Optimization | Academic Free License | None Reported | Binary integration, no direct linking |
| GPSD | 3.25 | GPSD Project | libgps-dev | GPS Input | BSD License | None Reported | Used for GPS integration |
| nlohmann/json | 3.11.2 | JSON for Modern C++ | Header-only | JSON Config Parsing | MIT License | None Reported | Configuration parsing and output formatting |

SOBM: Development and Test Environment Tools

| Tool | Version | Supplier | Dependency Relationship | License | Known Vulnerabilities | Notes |
|--------------------------|--------------------|---------------------------------------|--------------------------|---------------------------------|-----------------------|--|
| Debian (Bookworm) | 12 | Debian | Base OS | GPLv2 | None Reported | Development and CI/CD pipeline container image |
| Clang/GCC Compiler Suite | Clang ≥16, GCC ≥13 | LLVM Project, GNU Compiler Collection | Compilation and Analysis | GPLv3 (GCC), Apache 2.0 (Clang) | None Reported | Address sanitizer used for static/dynamic analysis |

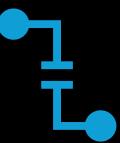
SOBM: Additional Information



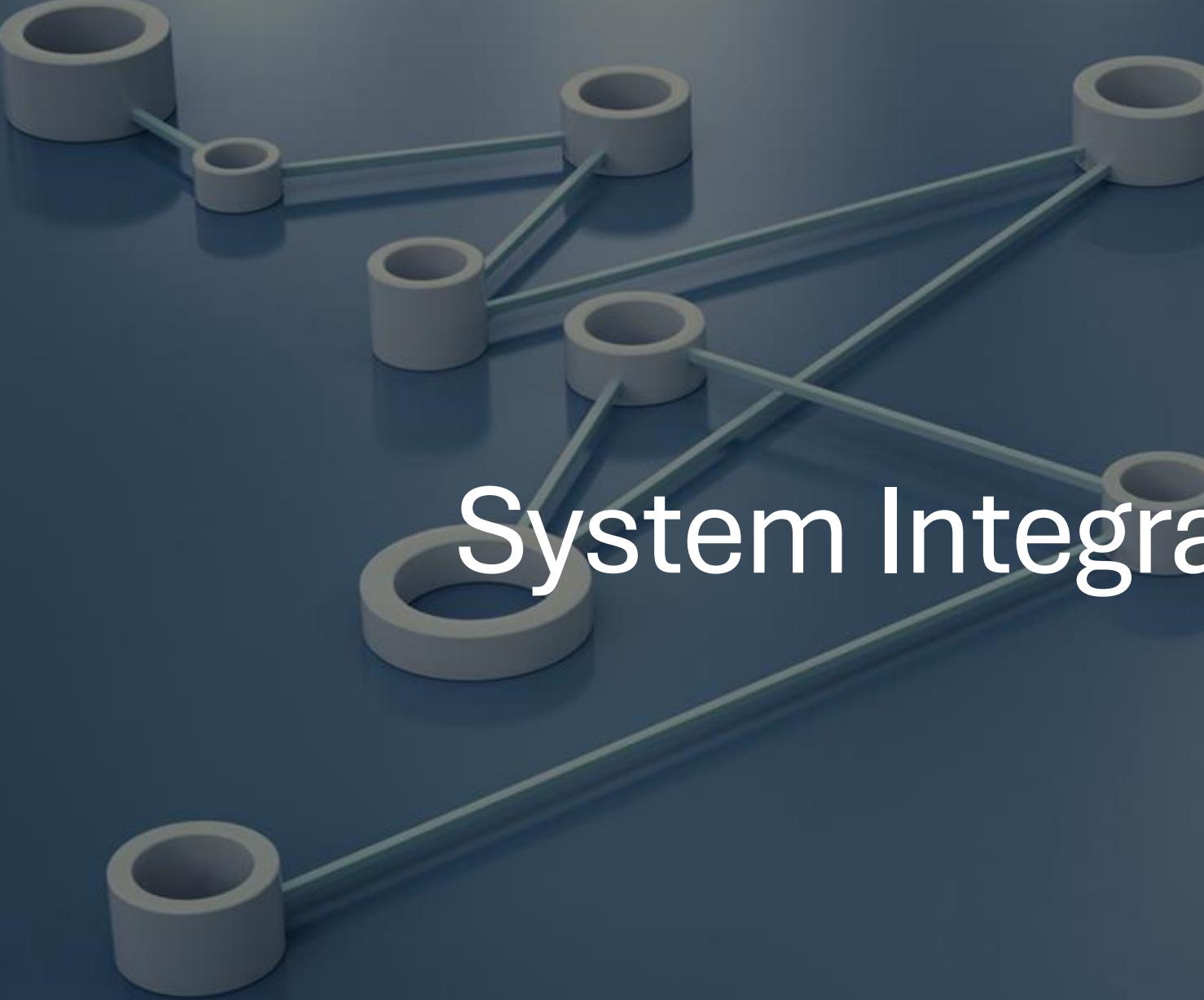
All source code components and build scripts are hosted in the project repository:
[github.com/kimsh02/awns-rpi5.](https://github.com/kimsh02/awns-rpi5)



Dependency relationships and licenses were verified as of SBOM assembly date.



No known vulnerabilities were reported in the components used at the time of assembly.



System Integration

System Integration: Hardware Modules

- Raspberry Pi 5 (16GB)
- Raspberry Pi 5 charger (CanaKit 45W USB-C Power Supply with PD for Raspberry Pi 5 (27W @ 5A))
- GPS Dongle (VK-162 G-Mouse USB GPS Dongle Navigation Module External GPS Antenna Remote Mount USB GPS Receiver for Raspberry Pi Support Google Earth Window Linux Geekstory)
- A flash drive (8GB+)

A photograph of a pushpin map on a desk. The map features a grid of colored pushpins in various colors like blue, yellow, green, and red, representing geographical features or routes. The background is slightly blurred.

Navigation Algorithm

Navigation Algorithm: Traveling Salesman Problem (TSP)



The Traveling Salesman Problem (TSP) is a classic optimization problem where a salesman must find the shortest route to visit a number of cities exactly once and return to the starting point.



This is the problem that the navigation algorithm is trying to solve.

Navigation Algorithm: Concorde TSP Solver

- The Concorde TSP solver is a software package designed to solve the symmetric traveling salesman problem (TSP) and related network optimization problems.
- It is known for its ability to find provably optimal solutions for large TSP instances using a branch-and-cut algorithm.
- For this reason, Concorde is the static waypoint planner chosen for the navigation system.

Navigation Algorithm: Concorde TSP Solver (cont'd)

- The authors didn't intend for anyone to use it as a C/C++ API in other projects; it was written as a research-grade standalone solver, not a reusable library.
- Academic software like Concorde entails
 - Solving a big math problem (TSP) and winning all the benchmarks
 - But not exporting most functions and little to no API documentation
- Acceptable usage is to build the full binary and call it as it does in the navigation system.
- Details are documented in the Git repository.

Navigation Algorithm: Proximity Detection



The GPS module is accurate up to around ~3-5 meters in an urban environment.



Idea is to set a proximity radius threshold for determining arrival at each waypoint.



Do this by comparing the current position of the navigation system to the position of the current waypoint that it is visiting.



Conceptually, we are computing the Euclidean distance between two points.

Navigation Algorithm: Proximity Detection (cont'd)

- Because the Earth is (roughly) a sphere, Euclidean (flat) distance formulas are inaccurate over large distances. The Haversine formula accounts for Earth's curvature.
- The **Haversine formula** is a mathematical method used to calculate the **great-circle distance** (shortest distance over the Earth's surface) between two points specified by their **latitude and longitude**.

```
/* Helper method to check whether destination has been reached */
/// Returns true if 'current' is within proximityRadius_ of 'destination'.
/// current    {latitude, longitude} in degrees
/// destination {latitude, longitude} in degrees
/// return true if distance ≤ proximityRadius_
bool Navigator::waypointReached(
    const std::pair<double, double> &current,
    const std::pair<double, double> &destination) const noexcept
{
    // 1) convert to radians
    constexpr double degToRad = std::numbers::pi / 180.0;
    double φ1      = current.first * degToRad;
    double λ1      = current.second * degToRad;
    double φ2      = destination.first * degToRad;
    double λ2      = destination.second * degToRad;
    // 2) haversine "a"
    double dφ     = φ2 - φ1;
    double dλ     = λ2 - λ1;
    double sinDφ2 = std::sin(dφ / 2);
    double sinDλ2 = std::sin(dλ / 2);
    double a =
        sinDφ2 * sinDφ2 + std::cos(φ1) * std::cos(φ2) * sinDλ2 * sinDλ2;
    a = std::clamp(v: a, lo: 0.0, hi: 1.0);
    // 3) central angle c
    double c = 2 * std::atan2(std::sqrt(a), std::sqrt(1 - a));
    // 4) distance and compare
    double distance = earthRadius_ * c;
    return distance ≤ proximityRadius_;
}
```

Autonomous Control Input



Autonomous Control Input: Navigation Output

Logged to stdout and
optionally to a log file if the
user specifies a log directory
in the CLI



Generated in JSON string
format

- JSON is a popular and universal
format choice for data exchange
- Suitable for interpretation of
directions by a downstream motion
controller or vehicle controller.

Autonomous Control Input: Navigation Output for Simulated Velocity

- Bearing: reported as degrees from true North
- Destination: reported in latitude, longitude, and waypoint which is the order number in which the coordinate was read from the .csv file
- GPS Position: reported in latitude, longitude as it was polled from the GPS dongle
- Simulated Position: reported in latitude, longitude as it was computed from the user-set simulated velocity (for debugging)
- Timestamp: reported in YYYY-MM-DD Hour-Minute-Second format
- Velocity: user-set simulated velocity reported in meters per second
- ***Used for debugging and/or when there is no downstream motor controller.***

```
{  
    "bearing": 25,  
    "destination": {  
        "latitude": 32.410998,  
        "longitude": -110.990288,  
        "waypoint": 2  
    },  
    "gps_position": {  
        "latitude": 32.398866923,  
        "longitude": -110.997004779  
    },  
    "sim_position": {  
        "latitude": 32.399682088352556,  
        "longitude": -110.99655458005157  
    },  
    "timestamp": "2025-06-16 19:54:51",  
    "velocity": 20.0  
}
```

Autonomous Control Input: Navigation Output for GPS

- Bearing: reported as degrees from true North
- Destination: reported in latitude, longitude, and waypoint number as it was read from the .csv file
- GPS Position: reported in latitude, longitude as it was polled from the GPS dongle
- Timestamp: reported in YYYY-MM-DD Hour-Minute-Second format
- ***Used for passing directions to a downstream motor controller***
- ***Realistically, only the `bearing` value is of interest to the controller***

```
{  
    "bearing": 24,  
    "destination": {  
        "latitude": 32.410998,  
        "longitude": -110.990288,  
        "waypoint": 2  
    },  
    "gps_position": {  
        "latitude": 32.398829749,  
        "longitude": -110.996981211  
    },  
    "timestamp": "2025-06-19 18:40:25"  
}
```

Configurability

Configurability: Waypoints

- The program will read in a series of static waypoints via a .csv file
 - Through a user-specified file/directory input in the CLI.
- Must have two columns with headers specifying latitude and longitude and in that order.
- A tests/csv directory is included in the Git repository providing test CSV files.
- Documentation of this is also specified in the Git repository.

Configurability: Waypoints (cont'd)

```
1 latitude,longitude
1 32.398800,-110.997000
2 32.310543,-110.923034
3 32.416638,-111.073390
4 32.384877,-110.982565
5 32.426101,-111.040624
6 32.326244,-111.066260
7 32.480459,-111.088236
8 32.484397,-110.967560
9 32.401417,-111.047077
10 32.398097,-110.915550
11 32.322838,-110.897429
12 32.354010,-111.026832
13 32.468800,-110.999670
14 32.412453,-111.054416
15 32.415779,-110.907931
16 32.383876,-110.946210
17 32.498529,-110.995976
18 32.471005,-111.087000
19 32.406681,-111.052260
20 32.336800,-111.037324
21 32.323546,-111.085184
22 32.371574,-110.932296
23 32.397830,-111.055515
```

10_allaround

| latitude | longitude |
|------------------|-------------|
| 32.398800 | -110.997000 |
| 32.491510 | -111.082205 |
| 32.414900 | -110.931113 |
| 32.448865 | -111.065148 |
| 32.299868 | -111.061833 |
| 32.337376 | -111.058102 |
| 32.330259 | -111.088128 |
| 32.474241 | -110.924415 |
| 32.442561 | -110.908860 |
| 32.490015 | -110.940281 |

A photograph of a large pile of cut logs stacked in a pyramidal shape. The logs are light-colored with dark, textured bark. In the background, a dense forest of evergreen trees is visible against a sky transitioning from blue to orange and yellow, suggesting either sunrise or sunset.

Fail-Safes and Logging

Fail-Safes and Logging: Fault Detection

- Gracefully logs errors and exits upon failure to connect to the GPS stream or upon detection of GPS signal loss during execution.
- Handles other signal jitters including
 - GPS stream read errors
 - Insufficient GPS data (it at least needs to report 2D coordinates)
 - Stale GPS data in socket
 - GPS read timeouts
- Lenient GPS polling logic is policy in case of failed attempts due to the above

```
/* Fix-reading with appropriate error handling */
std::optional<GPSFix> GPSClient::readFix(void)
{
    /* Poll GPS daemon's socket for data */
    if (gps_waiting(&data_, timeout_us_)) {
        /* Read GPS data into data_ struct */
        if (gps_read(&data_, message: nullptr, message_len: 0) < 0) {
            /* If gps_read() returns less than 0, report error and
             * return nullopt */
            std::cerr << "gps_read error: " << gps_errstr(errno)
                  << "\n";
            return std::nullopt;
        }
        /* If GPS reports at least latitude and longitude (and maybe not
         * altitude), return GPSFix struct */
        if (data_.fix.mode >= MODE_2D) {
            /* Check for fresh fix */
            timespec t      = data_.fix.time;
            double   fix_ts = static_cast<double>(t.tv_sec) +
                             static_cast<double>(t.tv_nsec) * 1.0e-9;
            /* If stale fix, return nullopt */
            if (fix_ts <= last_ts_) {
                return std::nullopt;
            }
            last_ts_ = fix_ts;
            return GPSFix{ .latitude=data_.fix.latitude,
                          .longitude=data_.fix.longitude,
                          .heading data_.fix.track };
        }
    }
    /* Either timeout has expired and no data/not enough data has arrived
     * (for whatever reason), or another error has occurred such as the
     * socket closing */
    return std::nullopt;
}
```

Fail-Safes and Logging: Fault Detection (cont'd)

```
/* Wrapper for readFix(), attempts to get 2D fix reading */
std::optional<GPSFix> GPSClient::waitReadFix(void)
{
    /* If GPSClient is not connected, return nullopt */
    if (!connected_)
        return std::nullopt;
    int tries = max_tries_;
    /* Try to get GPS fix */
    while (tries) {
        auto optFix: std::optional<GPSFix>{ readFix() };
        if (optFix) {
            /* If we get 2D fix, then return fix */
            return optFix;
        }
        tries--;
        /* Rate limit tries */
        std::this_thread::sleep_for(d: std::chrono::milliseconds(r: 1000));
    }
    /* If we don't get any 2D fix max_tries_, return nullopt */
    return std::nullopt;
}
```

```
/* Helper method for GPS output */
std::optional<json> Navigator::gpsOutput(void)
{
    /* Get GPS reading */
    auto optFix: std::optional<GPSFix>{ gps_.waitReadFix() };
    /* If can't get GPS reading, return null */
    if (!optFix) {
        logPrint(message: "(System Message) GPS signal lost. Ending output.",
                 timeStamp: true);
        return std::nullopt;
    }
    /* Get current position */
    GPSFix fix{ *optFix };
    /* Update 'currPos_' */
    currPos_.first = fix.latitude;
    currPos_.second = fix.longitude;
    /* ... */
}
```

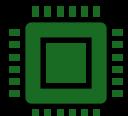
```
/* GPS poll, if exit is true then exit program */
void Navigator::gspoll(bool exit)
{
    /* Test GPS connection */
    while (true) {
        /* If GPS connection test was successful, proceed */
        if (testGPSConnection()) {
            break;
        }
        /* Else, ask user whether to retry connection */
        retryPrompt(message: "GPS connection failed.");
    }
    if (exit) {
        std::exit(0);
    }
}
```

Documentation

Documentation: API



The developer can modify the source code as they see fit.



As-is, the navigation system provides a simple API that can be integrated with the development of a downstream motor controller.



The idea is to work with a 'Navigator' object in the code

Has a total of 5 methods that you can call

Documentation: API (cont'd)

Headers

- #include "navigator.hpp"

Methods

- Navigator(int argc, const char **argv) noexcept
- void start(void)
- void setProximityRadius(double r) noexcept
- void setSimulationVelocity(double v) noexcept
- std::optional<json> getOutput(void)

Git repository documents specifics of the API

- 'main.cpp' contains an example usage of initializing and running the navigator.

Test Report



Test Report: Brief

Navigation system can plan visiting order of 50+ waypoints in less than 1 second.

Plots a visualization of the visiting order via an installed Python script.

Full testing suite and results are uploaded in the ‘tests/csv’ and ‘tests/graph’ directories in the Git repository.

- Sample navigation logs are uploaded in the ‘tests/log’ directory.

Test Report: Navigation Planner Correctness

118.920

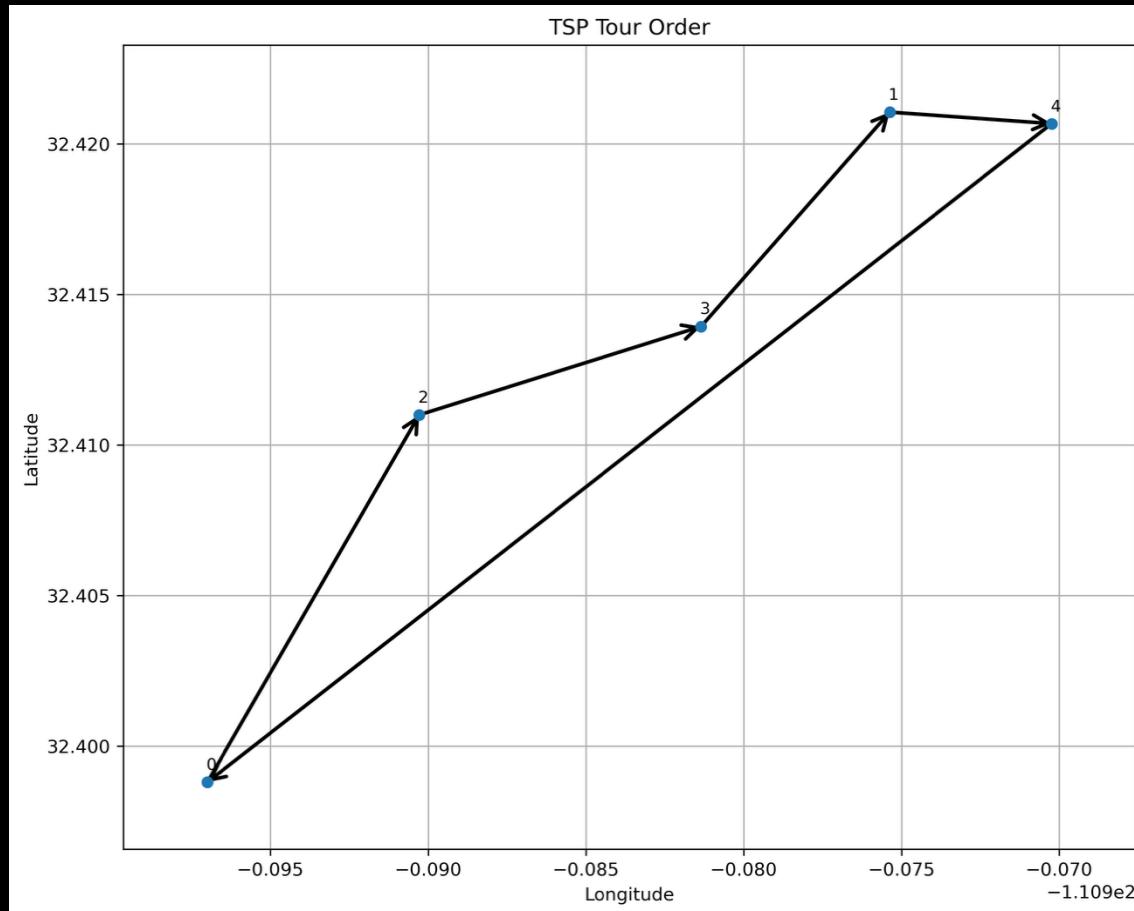
380.330

187.170

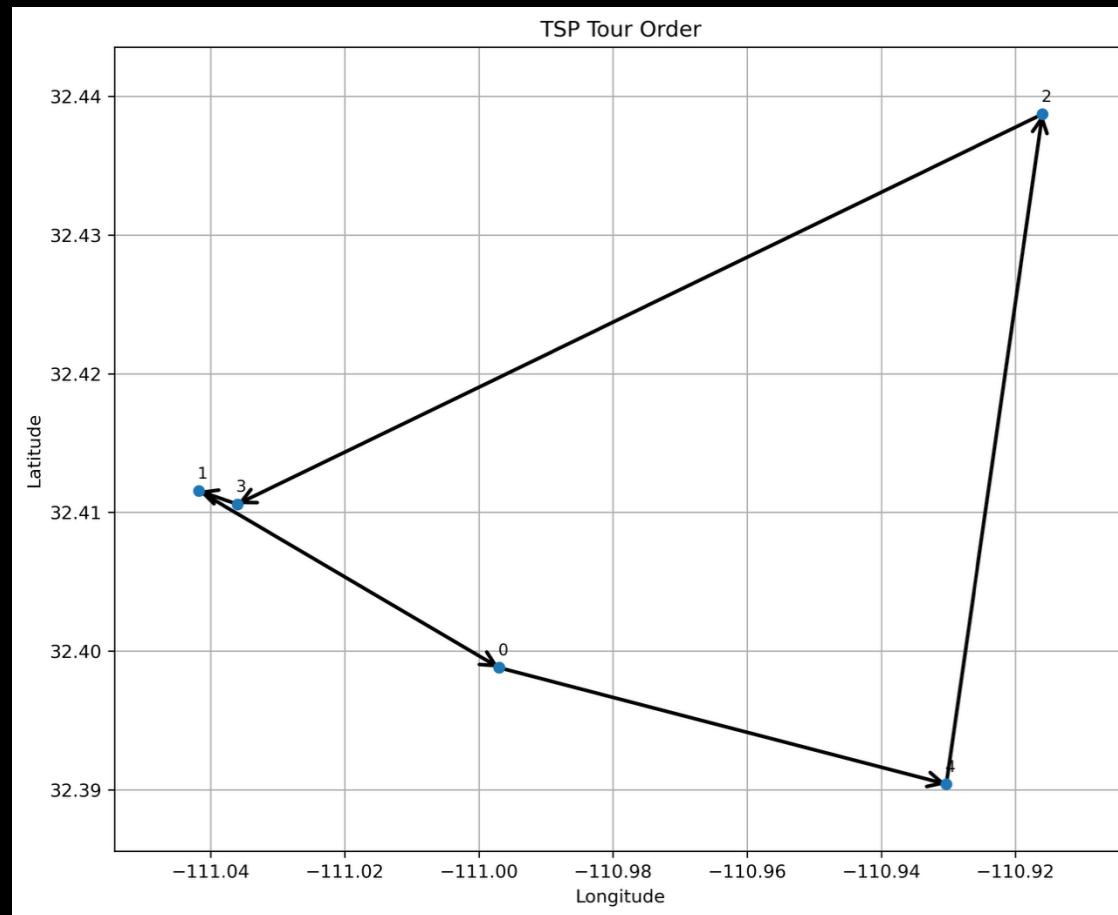
154.360



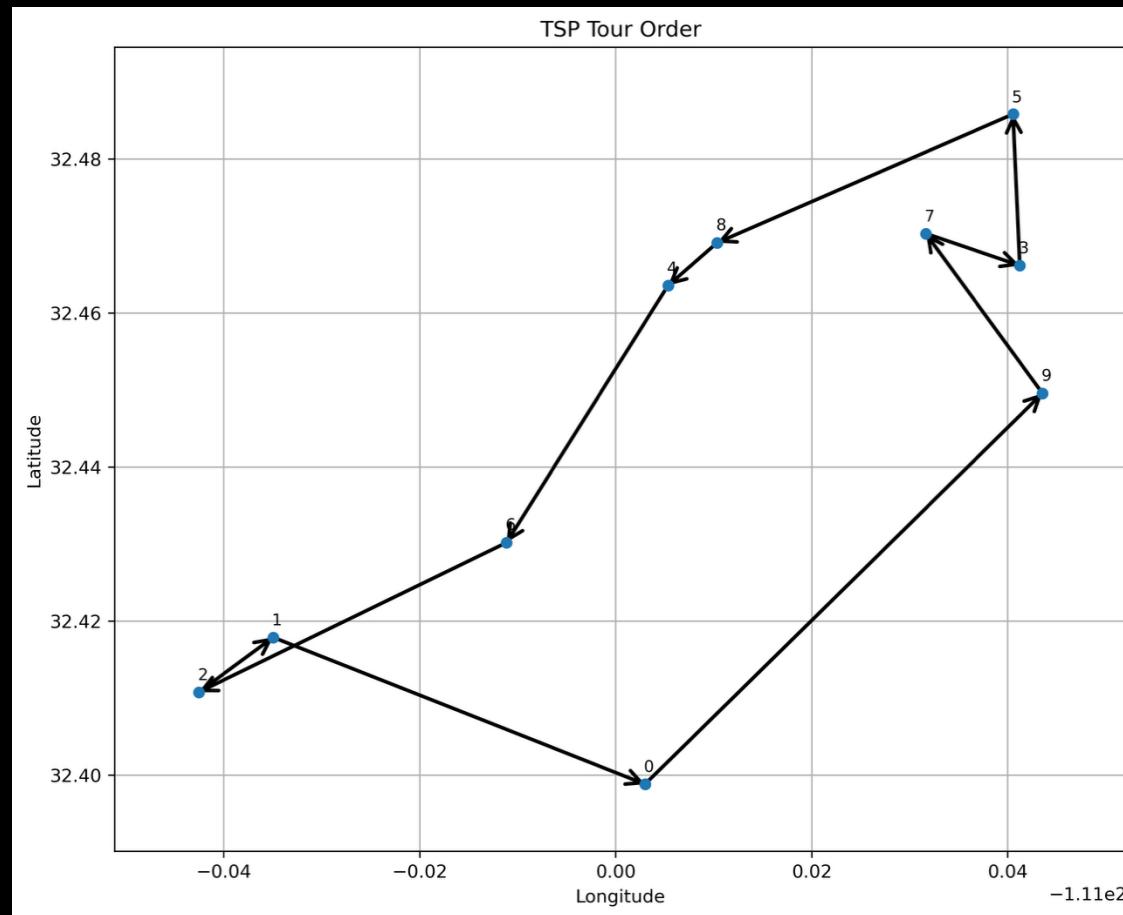
Test Report: 5_line.csv (0.004235 s)



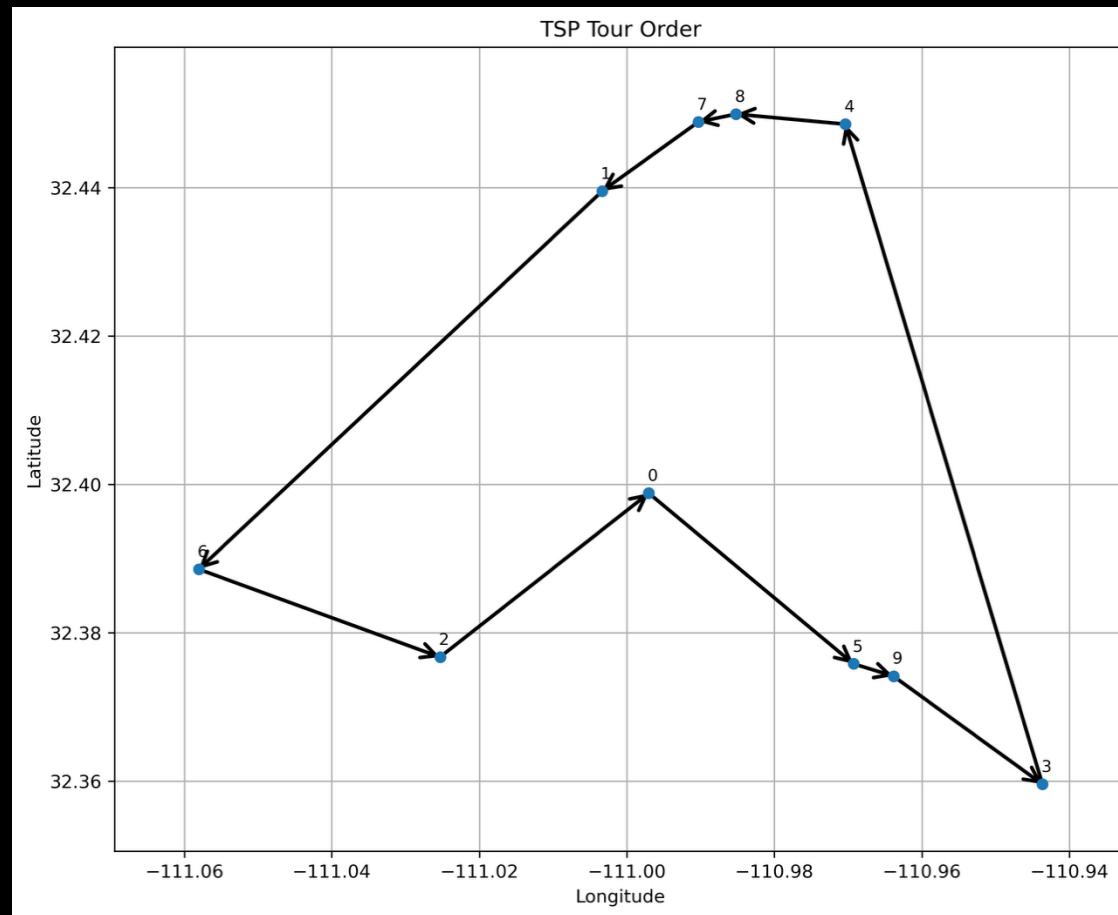
Test Report: 5_allaround.csv (0.003660 s)



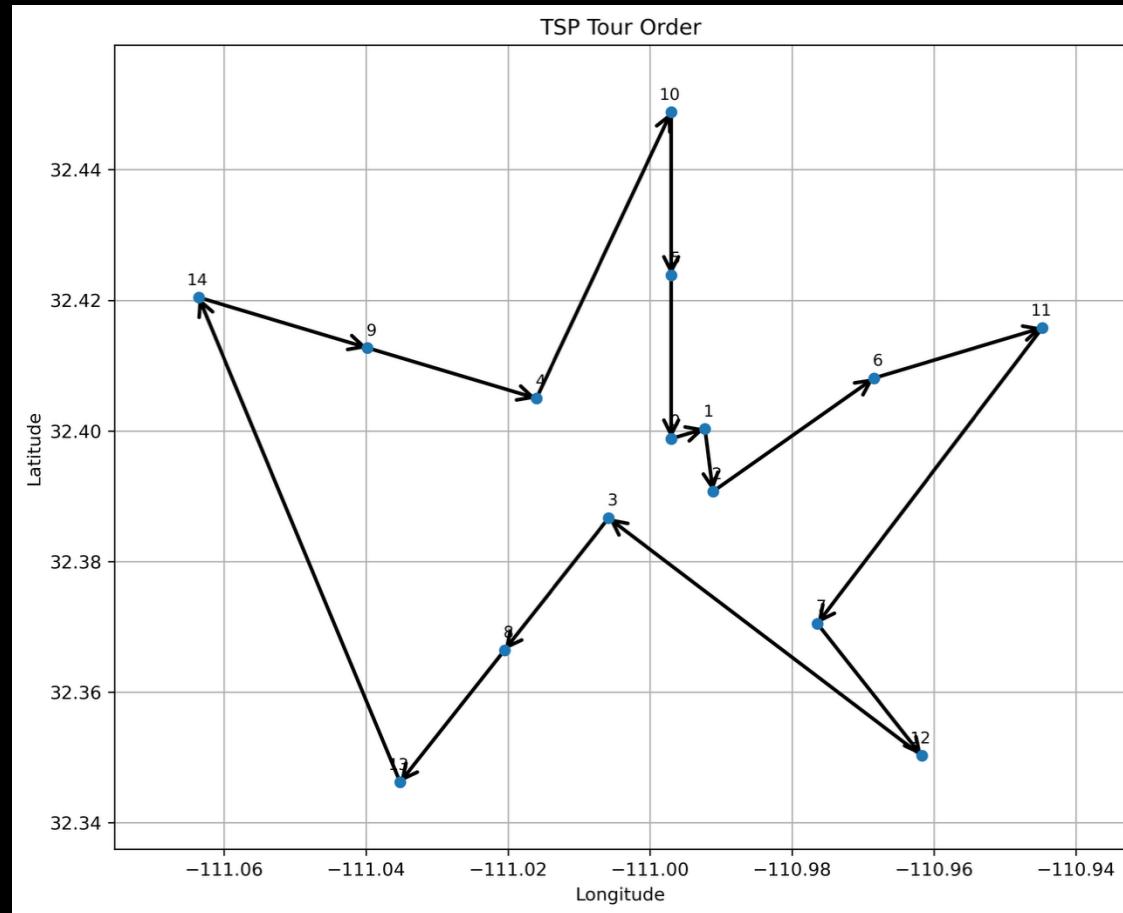
Test Report: 10_oneside.csv (0.002295 s)



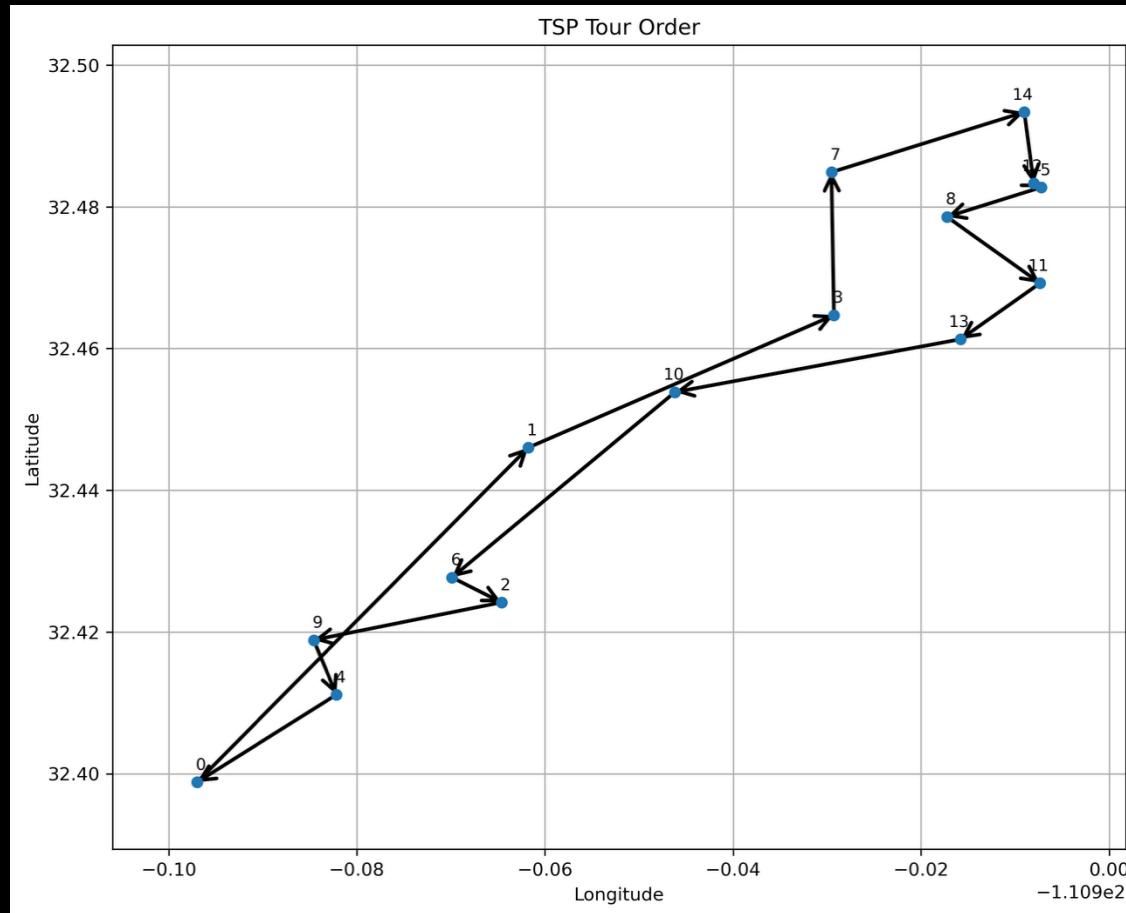
Test Report: 10_clusters.csv (0.002183 s)



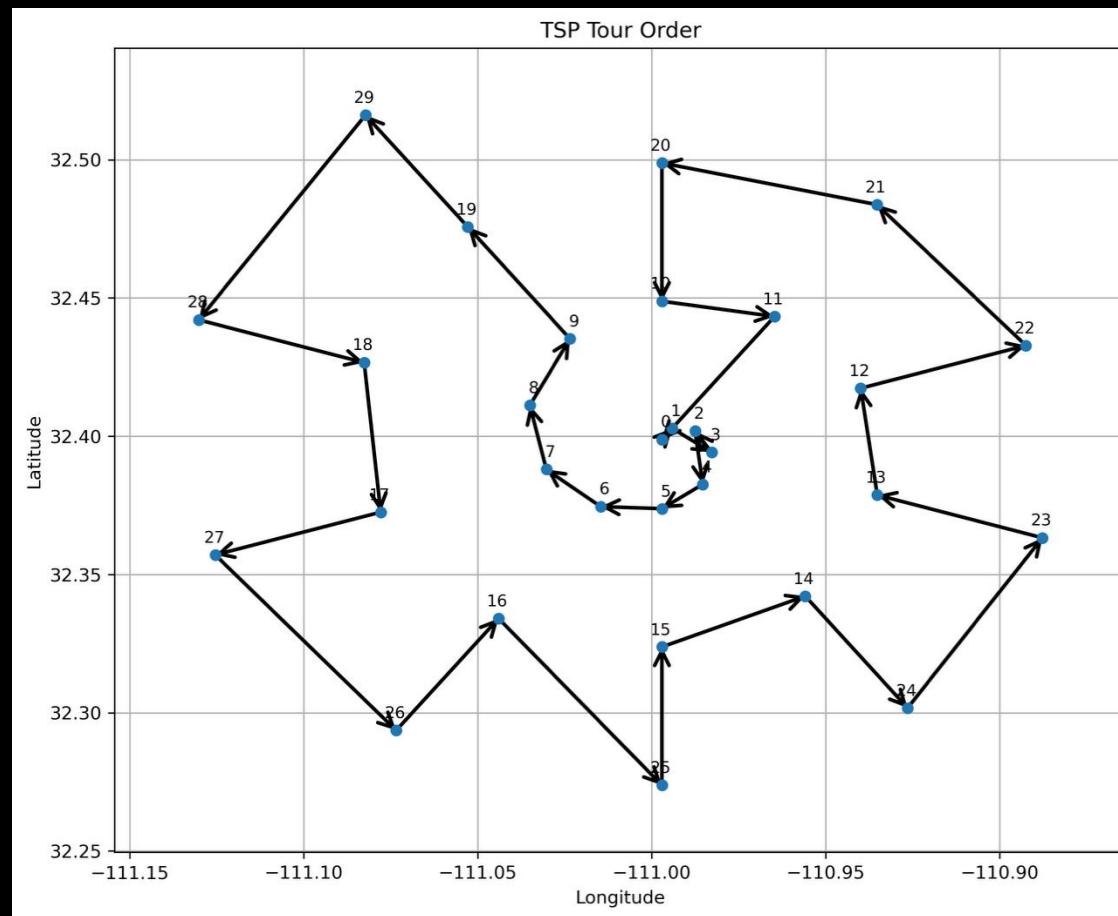
Test Report: 15_spiral.csv (0.008457 s)



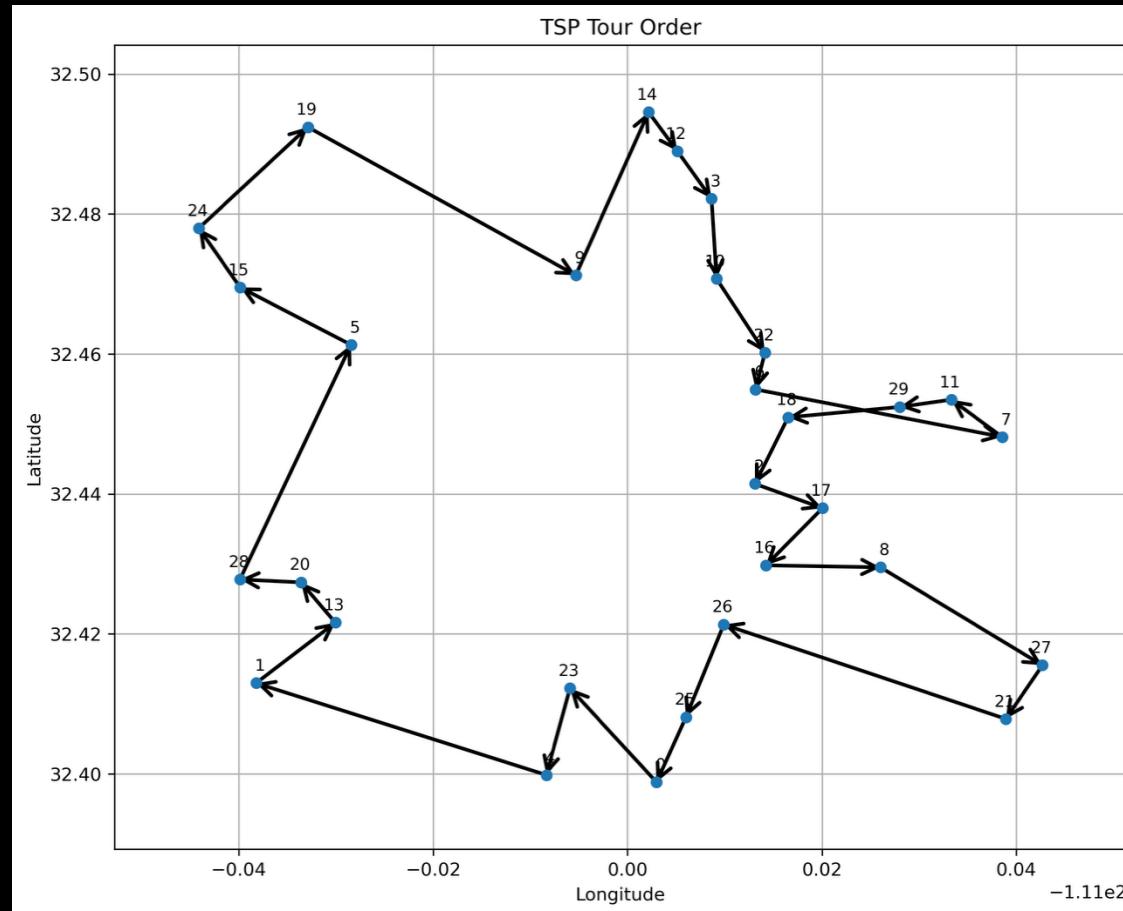
Test Report: 15_line.csv (0.007012 s)



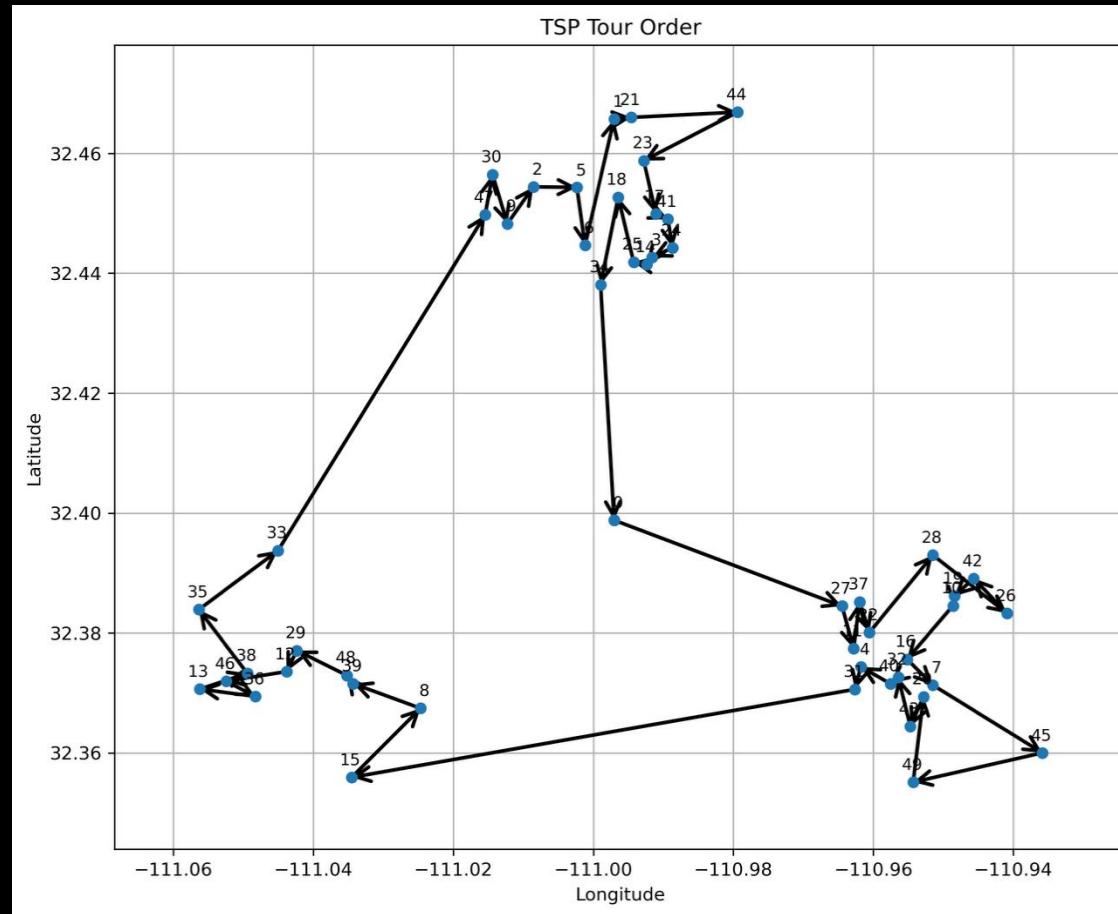
Test Report: 30_spiral.csv (0.072919 s)



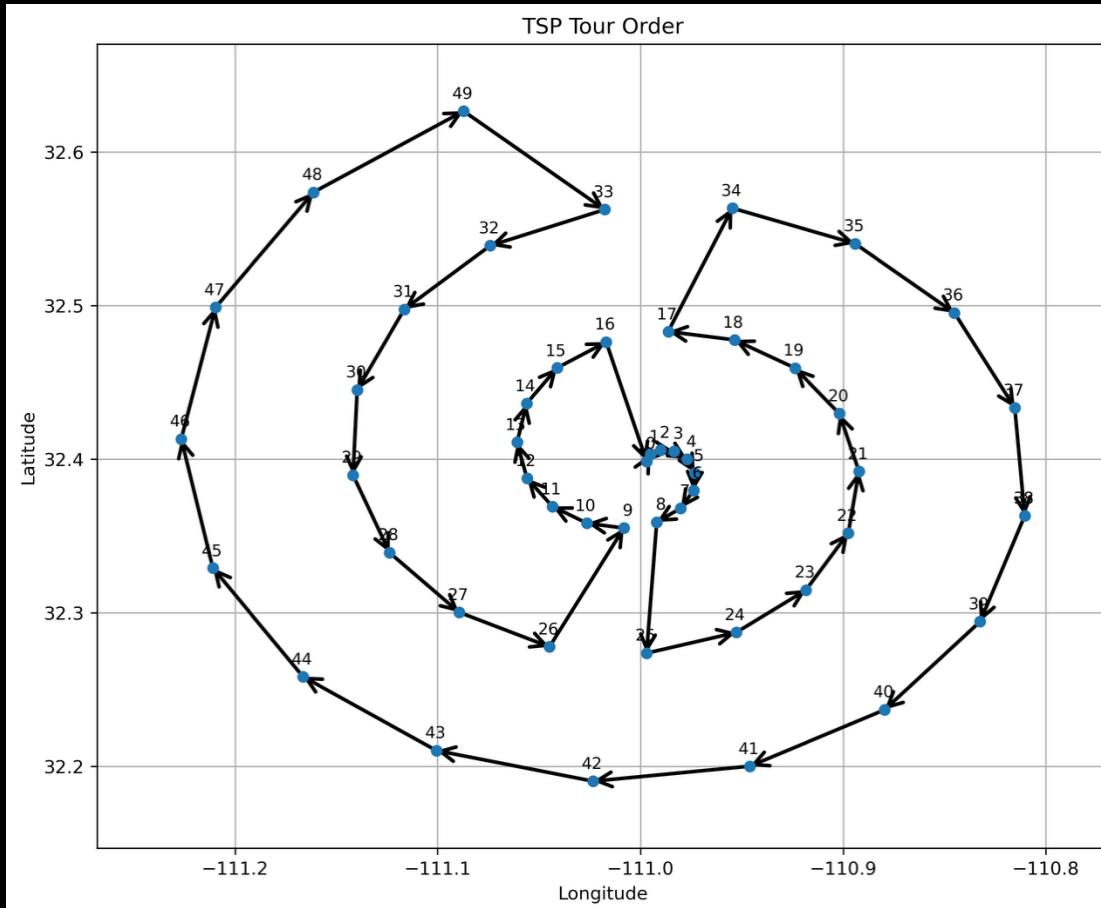
Test Report: 30_oneside.csv (0.042657 s)



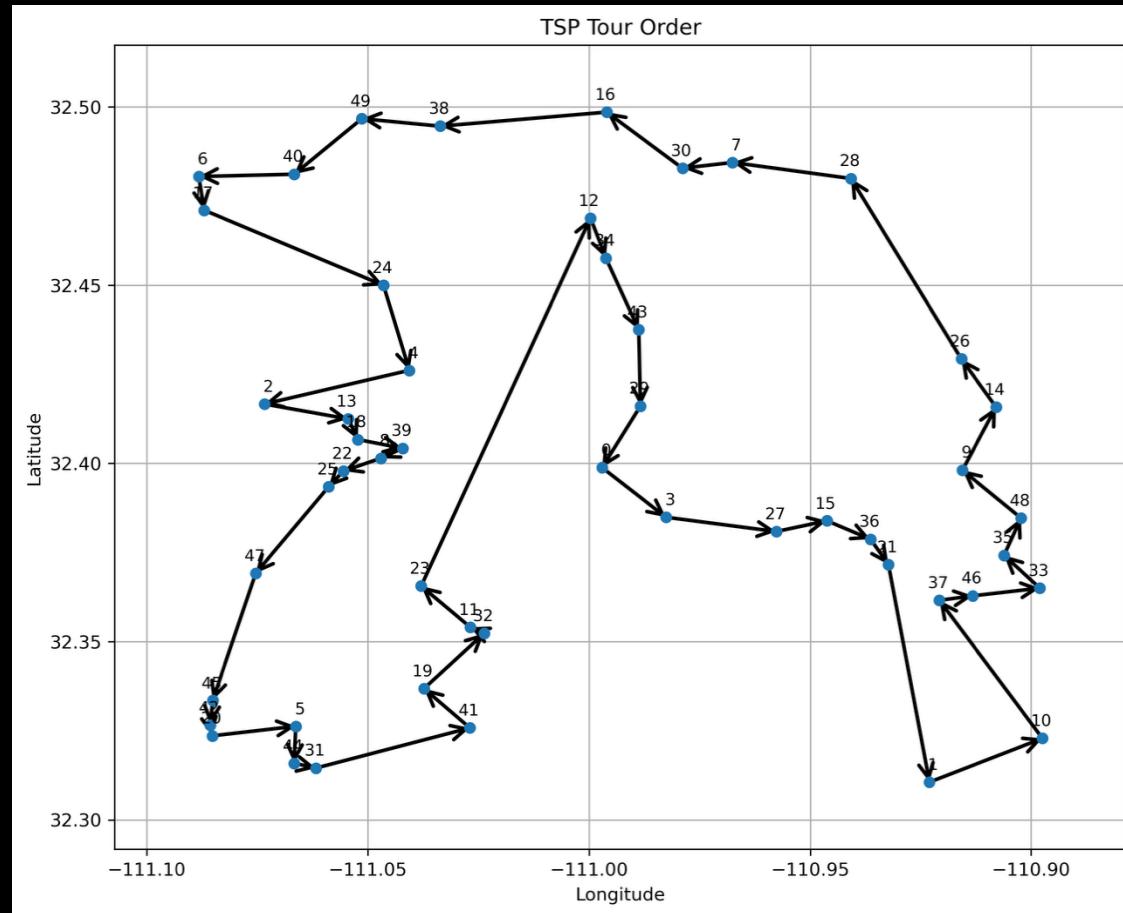
Test Report: 50_clusters.csv (0.123383 s)



Test Report: 50_spiral.csv (0.083751 s)



Test Report: 50_allaround.csv (0.141592 s)



Test Report: Software Assurance



Test Report: Software Assurance

- Navigation system does not have any memory leaks.
 - Clang/GCC (compiler) address sanitizer was used to profile the code at compile-time and run-time.

```
mirror_mod = modifier_obj
set mirror object to mirror
mirror_mod.mirror_object

operation == "MIRROR_X":
    mirror_mod.use_x = True
    mirror_mod.use_y = False
    mirror_mod.use_z = False

operation == "MIRROR_Y":
    mirror_mod.use_x = False
    mirror_mod.use_y = True
    mirror_mod.use_z = False

operation == "MIRROR_Z":
    mirror_mod.use_x = False
    mirror_mod.use_y = False
    mirror_mod.use_z = True

selection at the end -add
modifier.select= 1
modifier.select=1
bpy.context.scene.objects.active.name
("Selected" + str(modifier))
modifier.select = 0
bpy.context.selected_objects
data.objects[one.name].select

int("please select exactly one object")
- OPERATOR CLASSES -
types.Operator):
    X mirror to the selected object.mirror_mirror_x"
    or X"
context):
    ext.active_object is not None
```

Test Report: Software Assurance (cont'd)

- Navigation system logging was tested on a planned tour for `30_lines.csv` with proximity radius set to 20 meters and simulation velocity set to 20 meters per second.
- Simulation lasted for ~1 hour, and navigation output was generated to completion without crashing.

```
1 {
2   "bearing": 355,
3   "destination": {
4     "latitude": 32.419003,
5     "longitude": -110.998734,
6     "waypoint": 7
7   },
8   "gps_position": {
9     "latitude": 32.398799759,
10    "longitude": -110.996986175
11  },
12  "sim_position": {
13    "latitude": 32.398799759,
14    "longitude": -110.996986175
15  },
16  "timestamp": "2025-06-16 21:16:33",
17  "velocity": 20.0
18 }
```

```
19 {
20   "bearing": 245,
21   "destination": [
22     {
23       "latitude": 32.3988,
24       "longitude": -110.997,
25       "waypoint": 0
26     },
27     {
28       "gps_position": {
29         "latitude": 32.398799759,
30         "longitude": -110.996986175
31       },
32       "sim_position": {
33         "latitude": 32.399013241483665,
34         "longitude": -110.99645226812663
35       },
36       "timestamp": "2025-06-16 22:16:39",
37       "velocity": 20.0
38     }
39   ],
40   "t[2025-06-16 22:16:41] (System Message) Waypoint reached: [Latitude: 32.3988, Longitude: -110.9970]
41   [2025-06-16 22:16:41] (System Message) Navigation has completed.
42 }
```

Test Report: Software Assurance (cont'd)

- Officially, the navigation system can run up to 1 hour in providing output.
 - In theory, the navigation system runs indefinitely if the GPS stream provides data.
- The navigation system has not been observed to crash during the various stages of testing and is not designed to crash but rather catch and exit on all errors gracefully.
 - A ‘crash’ could be defined as an instance where the system gracefully logged and exited on a GPS read error during navigation output which has been observed and is expected and good behavior.

Known Anomalies

Known Anomalies: Risk Assessment

| ID | Anomaly Description | Observed Conditions | Frequency | Impact Assessment | Risk Level | Mitigation Strategy |
|----|--------------------------------------|---|-----------|---|------------|--|
| 1 | GPS Signal Jitters | Brief drops in GPS connection or intermittent data reception. | Rare | Minor disruptions in real-time navigation accuracy. Temporary reduced reliability of navigation instructions. | Low | Implemented lenient GPS polling logic; gracefully logs errors and resumes normal operations automatically. |
| 2 | Dependency on Concorde Binary | Lack of direct API integration; external Concorde solver binary dependency. | Always | Limits system flexibility; potential performance impacts due to external process invocation overhead. | Medium | Clearly documented usage; potential future refactoring to integrate a dedicated, native TSP solver. |
| 3 | Waypoint File Formatting | CSV input strictly requires latitude and longitude columns in exact order and format. | Always | Risk of user input error leading to incorrect waypoint parsing or runtime failures. | Medium | Comprehensive documentation provided; rigorous CSV format validation implemented. |
| 4 | No GUI Interface Implemented | CLI-based usage with no graphical user interface. | Always | Reduced ease-of-use, particularly for less technical users; may impact broader user adoption. | Low | Planned future GUI implementation; current CLI includes thorough documentation. |

Known Anomalies: Overall Risk Assessment



The software is stable and reliable, with no anomalies posing critical or high risk.



Identified anomalies have documented mitigations and clear guidance provided in the user documentation.



The product can be safely released with current anomalies, given the existing mitigation strategies.

Technical Debt

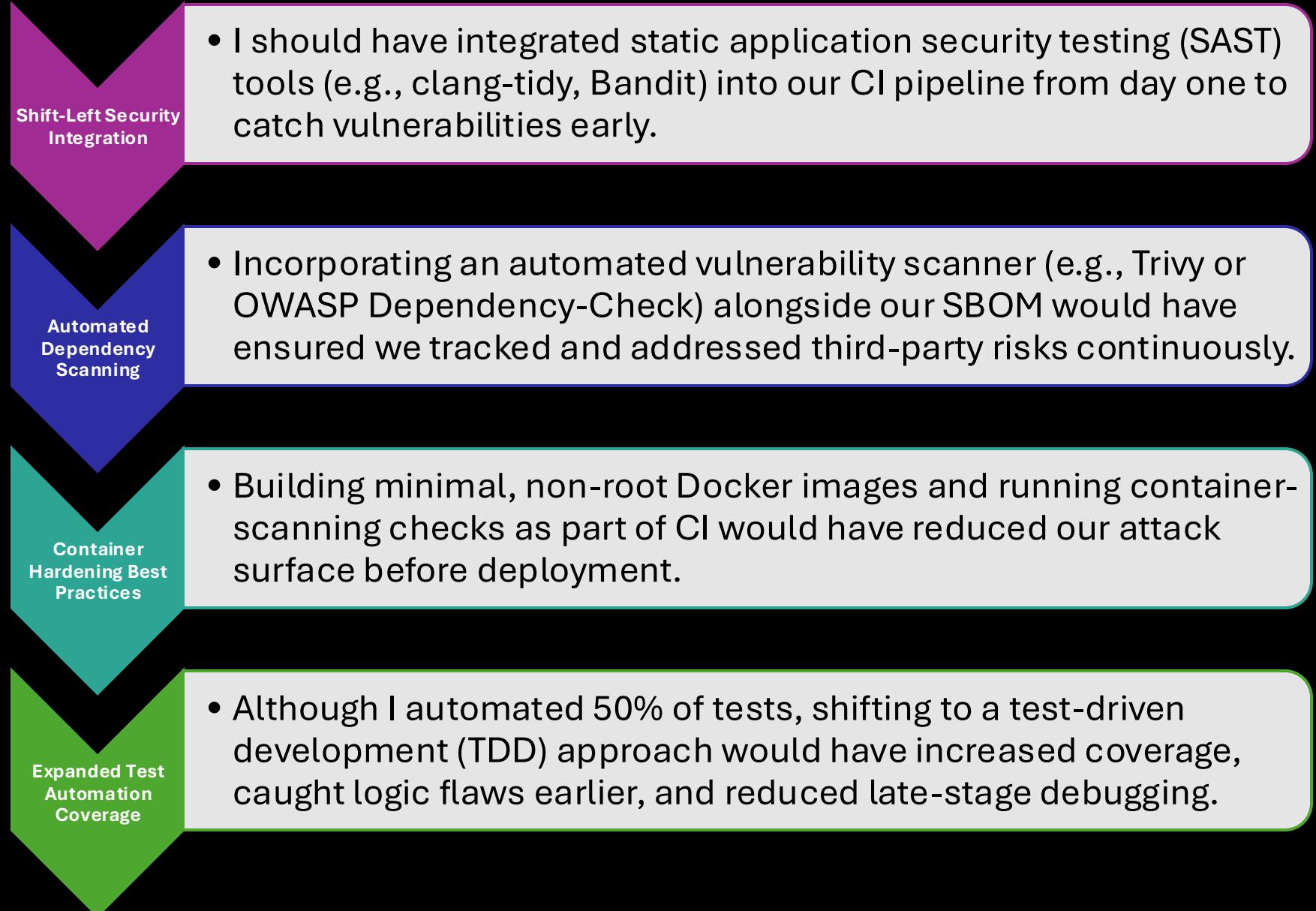
Technical Debt: Summary

| ID | Technical Debt Description | Reason for Deferral | Impact if Unaddressed | Recommended Future Action |
|----|---|--|--|--|
| 1 | No GUI Interface Implementation | Prioritization of core navigation logic and reliability. | Reduced user-friendliness; higher learning curve. | Plan incremental development of a lightweight GUI for future releases. |
| 2 | Dependency on External Concorde TSP Binary | Project timeline constraints; complexity of native solver integration. | Limits scalability, adds process overhead, reduces flexibility. | Evaluate and integrate a native or API-based TSP solver. |
| 3 | Strict CSV Waypoint Format Requirement | Initial requirement prioritization and time constraints. | User errors and potential runtime failures if CSV format deviates. | Add enhanced CSV input validation and error handling. |



Lessons Learned

Lessons Learned



Success Criteria



Success Criteria: Checklist

- System reliably reads GPS data on Raspberry Pi 5  (slide 22)
- Waypoint algorithm guides system from point to point  (slide 30)
- Navigation decisions are logged and reproducible  (slide 16)
- Software runs continuously without crashing for ≥ 30 minutes  (slide 43)
- Clear, well-documented code and setup instructions 
(<https://github.com/kimsh02/awns-rpi5>)

Thank You

