Capstone Project Phase B

SATELLITES, DISTANCE APPROXIMATION, FEASIBILITY TEST, COLLISION DETECTION, SATELLITE HARDWARE, ALGORITHMS IMPLEMENTATION

24-2-D-29

Supervisor: Mr. Ilya Zeldner
Shir Cohen 316216340
Cohensh96@gmail.com
Yotam Aharon 208608448
Yotamah14@gmail.com
Project Repository

BRAUDE
College of Engineering, karmiel

Table of Contents

Abstract

Our project is the continuation of project titled "Feasibility Analysis and Performance Testing of Collision Detection Algorithms for Satellites" **[7]**, which focused on developing an algorithm for calculating the minimal distance between two objects in space. In our project, we transition from the theoretical phase to the practical phase, optimizing and implementing SSA algorithms on actual hardware on satellite **On-Board Computer (OBC)**, following feasibility test results. The project has two goals, the first goal is to implement, deploy and execute the **Space Situational Awareness (SSA)** Algorithms on the selected OBC, the second goal is to conduct an in-depth performance analysis to assess the feasibility of executing these algorithms efficiently in real time while operating within the strict power, memory, and processing constraints of the selected satellite OBC. The project is based on Dr. Elad Dannenberg's research papers that introduced the algorithms **[5][6].**

Keywords

Satellite, Space debris, Minimal distance, Approximations algorithms, Feasibility test, Collision detection, Algorithm testing, Distance approximation, Orbiting objects, Satellite On-Board Computer (OBC), Raspberry pi 5 on-board computer, Satellite hardware, Algorithms implementation.

1. Introduction

One of the things that concerns satellite operators during a mission, is the risk of colliding into other objects. There is a significant amount of space debris orbiting Earth, including decommissioned satellites or parts of them, rockets, and other human-made objects, and there are also natural celestial bodies in space we should be aware of like asteroids. In order to avoid these threats, we start by keeping track of them, then we identify possible collisions and recalculate our path. Doing so is done by calculating the future orbit of 2 object and finding the point in time where the distance between them is the smallest, this time is called **Time of Closest Approach (TCA)** and the TCA and the respective distance is the values we are looking for.

With the increasing number of objects in Earth orbit, around 27,000 **[15]** and the shift to cluster of smaller satellites instead of a single big one **[5]** the cost of calculating the orbit of objects and finding the TCA for our satellite is only growing. To solve this problem a few cheaper algorithms were developed. The algorithms are supposed to be computationally cheap and fast enough to run on the satellite's own OBC.

At this stage of our project, we are actively conducting tests and implementing the algorithms on the selected satellite board. We have conducted extensive research to identify an optimal satellite board for evaluating these algorithms, even though the initial selection criteria did not specifically address space-related conditions. Our investigation has focused on potential board dimensions, particularly utilizing the **ISIS OBC [12]** satellite board data, selected by Dr. Elad Danenberg, with whom we are collaborating closely. The primary objective of this phase is to implement, deploy and execute these algorithms on the selected Raspberry Pi 5 OBC this step involves adapting the algorithms to the board's operating system, optimizing code for its CPU architecture, and establishing reliable data flows between the on-board software and ground-control test station. Dr. Elad Danenberg, the developer of the **Conjunction Assessment Through Chebyshev Polynomials (CATCH)** algorithm **[5]** and the **SBO-ANCAS** algorithm **[6]**, is relying on our expertise to ensure that these systems can autonomously calculate potential collisions, necessitating the development of rapid TCA algorithms. The second objective is to conduct an in-depth performance analysis to assess the feasibility of executing these algorithms efficiently in real time while operating within the strict power, memory, and processing constraints of the selected satellite OBC. To achieve these objectives, we are employing advanced approximation techniques, including the **CATCH [5]**, **SBO-ANCAS [6]**, and **Alfano/Negron Close Approach Software (ANCAS)** algorithms **[1]**. These algorithms undergo rigorous testing to verify their suitability for deployment in autonomous satellite operations and real-world implementation on the selected satellite board for performance and feasibility assessments.

2. General Description

2.1. Background

2.1.1. The Algorithms

In this project, we utilize three algorithms that were previously implemented and tested by the prior team **[7]**. Each of these algorithms can be used to calculate the TCA. The following sections provide a detailed description of the methodologies and principles underlying these algorithms. By relying on these established algorithms, we ensure continuity in the computational approach and base on the preceding research efforts of the prior team. The following is a description of the algorithms.

2.1.1.1. ANCAS

The first algorithm, ANCAS **[1]** uses cubic polynomial as an approximation of a function over an interval. Given n points in time and the respective location and velocity vectors for 2 objects, we can find the TCA by:

Algorithm 1: ANCAS on n points, (the original algorithms description can be found at **[1]**)

Input: $p[n], t[n]$
Output: $TCA + r_{TCA}$
$r_{TCA} = inf$
$t_{TCA} = inf$
for each set of 4 points **do:**
  Map the time points $t[1-4]$ to $\tau_0 =\leq \tau_1, \tau_2 \leq \tau_3 = 1$ on the interval $[0,1]$
  Calculate $\dot{f}(t), f_x, f_y, f_z$ using **[[1], Eq.2/5]** with the points $p[1-4]$
  Fit cubic polynomial to $\dot{f}(t)$ according to **[[1], Eq.1f-1j]** over $[0,1]$
  Find the cubic polynomial real roots in the interval $[0,1]$
  Fit cubic polynomials for f_x, f_y, f_z in the interval $[0,1]$
  for each root t^* **do:**
    calculate the distance $r(t^*)$ using **[[1], Eq.6]**
    if $r(t^*) < r_{TCA}$**:**
      $r_{TCA} = r(t^*)$
      $t_{TCA} = t^*$
    end
  end
end

The cubic polynomial coefficients calculations described in article **[1]**.
Finding the roots of a cubic polynomials can be done by solving the 3rd degree equation and we will find between 1 to 3 real solutions. There is a problem with the algorithm, the point in time must be relatively close because the algorithm can only find up to 3 extrema points, so working on a large time interval means we can miss possible points and even miss the actual point of the TCA. Because the root finding can be done fast using the 3rd degree equation the algorithm run relatively fast but the result can be inaccurate.

Variables and notation in ANCAS Algorithm:

To enhance the understanding of the ANCAS algorithm, here is a breakdown of the role of each variable, including the input, output, and other diffrenent notations used in the algorithm:

Input Variables:

- **p[n]:** This represents the position and velocity data of two orbiting objects at n distinct time points. Each element in p[n] is a vector that contains the 3D position and velocity information for the two objects involved in the calculation. This serves as the primary input data for the algorithm.

- **t[n]:** A corresponding time array that holds the timestamps associated with each data point in p[n]. This array represents the specific moments in time at which the positions and velocities of the two objects are known.These time points serve as the domain over which the polynomials are fitted. The algorithm works by fitting cubic polynomials to p[n] over the interval defined by t[n].

Output Variables:

- **t_TCA:** The time of closest approach (TCA), which is the time when the minimum distance between the two objects occurs. This is the primary result of the algorithm.
- **r_TCA:** The minimal distance between the two objects at the time of closest approach, calculated as the Euclidean distance between their positions at time t_TCA.

Other notations:

- The function $\dot{f}(t)$ represents the rate of change of the distance between the two objects over time.
- The components f_x, f_y, f_z represent the relative position between the objects in the 3D coordinate system, calculated using the data points.

2.1.1.2. SBO-ANCAS

The second algorithm, **SBO-ANCAS[6]** is based on the **ANCAS[1]** algorithm, still using cubic polynomial as an approximation of a function over an interval. But uses additional points to get better results. Given an initial set of n points in time, the respective location and velocity vectors for 2 objects, tolerance in time and tolerance in distance we can find the TCA by:

Algorithm 2: SBO-ANCAS on n points, (the original algorithms description can be found at **[1]**)

Input: $p[n], t[n], TOL_d, TOL_t$
Output: $t_{TCA} + r_{TCA}$
$r_{TCA} = inf$
$t_{TCA} = inf$
for each set of 4 points **do:**
  $\mathrm{t_{new}} = \mathrm{t_1}, \mathrm{t_2}, \mathrm{t_3}, \mathrm{t_4}$
  Do
    Map the time points $t[1-4]$ to $\tau_0 =\leq \tau_1, \tau_2 \leq \tau_3 = 1$ on the interval $[0,1]$
    Calculate $\dot{f}(t), f_x, f_y, f_z$ using **[[1], Eq.2/5]** with the points $p[1-4]$
    Fit cubic polynomial to $\dot{f}(t)$ according to **[[1], Eq.1f-1j]** over $[0,1]$
    Find the cubic polynomial real roots in the interval $[0,1]$
    Fit cubic polynomials for f_x, f_y, f_z in the interval $[0,1]$
    for each root $t*$ **do:**
      calculate the distance r at $t*$ using **[[1], Eq.6]**
      if $r < r_{min}$ **:**
        $r_{min} = r$
        $t_{min} = t*$
      end
    end
    Sample r_d and $\dot{r}_d$ at t_{min} using a propagator
    $\mathrm{t_{new}} = \mathrm{t_{new}} \setminus \left(t_j \mid \max\left(\left|t_j - t_{min}\right|\right) \cup t_{min}\right.$

  While $| r_{min} - \| r_d(t_{min}) \| | > TOL_d$ OR $\max\left(\left|t_j - t_{min}\right|\right) > TOL_t$
  $\mathrm{r_{tca}} = \| \mathrm{r_d}(t_{min}) \|$
  $\mathrm{t_{tca}} = t_{min}$
end

The cubic polynomial coefficients calculations described in article **[1].**
Finding the roots of a cubic polynomials can be done by solving the 3rd degree equation and we will find between 1 to 3 real solutions. In each iteration after finding the minimum point $\mathrm{t_{min}}$ we use the propagator to sample the location and velocity vectors at $\mathrm{t_{min}}$, we use the new and more accurate values and create a polynomial to find a more accurate minimum distance and so on until we reach the desired tolerance. This algorithm can give the best results, we can get the same results as checking every point with a small time-steps if we use small enough tolerance but with high cost in run time. **SBO-ANCAS [6]** have an additional loop in each iteration and sampling points with the propagator is an expensive operation.

Variables and notation in SBO-ANCAS Algorithm:
The SBO-ANCAS algorithm is an enhancement of ANCAS, incorporating iterative refinement and additional sampling to improve accuracy at the cost of longer runtime.To enhance the understanding of the SBO-ANCAS algorithm, here is a breakdown of the role of each variable, including the input, output, and other diffrenent notations used in the algorithm:

Input Variables:

- **p[n]:** The position and velocity vectors of two orbiting objects at n time points. This input provides the data needed to calculate the time and distance of closest approach between the objects.
- **t[n]:** The corresponding array of time values for each data point in p[n]. Each element represents the timestamp for the position and velocity of the objects.
- **Tolerance in time (TOL_t):** A small threshold that defines how close two time points must be for the algorithm to stop refining them. This determines the precision of the closest approach time.
- **Tolerance in distance (TOL_d):** A small threshold that specifies the acceptable error in the calculated minimal distance. It ensures the algorithm stops refining once the distance error is within this tolerance.

Output Variables:

- $\mathbf{t_{tca}}$: The time of closest approach (TCA) when the two objects are at their minimum distance.
- $\mathbf{r_{tca}}$: The smallest distance between the two objects during their encounter, calculated at $\mathbf{t_{tca}}$.

Other notations:

- $\mathbf{t_{new}}$: This variable represents the newly sampled time points introduced during each iteration of the **SBO-ANCAS [6]** algorithm.
  In each iteration of the algorithm, once the cubic polynomial fitting and root finding are complete, t_{new} is sampled using the propagator based on the previous approximate solution, and the algorithm recalculates the positions and distances at these new points. This iterative refinement continues until the tolerance in time and distance is satisfied.
- $\boldsymbol{r_d}$: represents the relative distance between the two orbiting objects at a given time. It is the Euclidean distance between their position vectors in three-dimensional space. The goal of the algorithm is to find the TCA, which is the time when $\boldsymbol{r_d}$ is minimized. During each iteration of the algorithm, $\boldsymbol{r_d}$ is calculated for various candidate times (including the initial and newly sampled time points) to determine when the two objects are at their closest.
- $\dot{\boldsymbol{r}}_{\boldsymbol{d}}$: represents the rate of change of the relative distance between the two objects with respect to time. In other words, it is the time derivative of the relative distance function $r_d(t)$.
- The algorithm uses $\dot{\boldsymbol{r}}_{\boldsymbol{d}}$ to help find candidate points for the closest approach. The critical points, where the distance between the objects could be minimal, occur when $\dot{\boldsymbol{r}}_{\boldsymbol{d}}(\boldsymbol{t}) = \mathbf{0}$. The algorithm fits a cubic

polynomial to approximate $\dot{\boldsymbol{r}}_{\boldsymbol{d}}$, and then finds its roots (where the derivative is zero) to identify potential times for the TCA.

2.1.1.3. CATCH

The third algorithm, **CATCH** [5], uses **Chebyshev Proxy Polynomial (CPP)** to approximate the functions. The CPP can give more accurate result, depending on the degree of polynomial we want to use. We can choose high enough degree to get the size of error we want. The algorithm work on time interval from 0 to $t_{\max}$, each iteration searches the minimal distance in an interval with size Γ. The degree of the CPP is part of the algorithm input and appear as N.

Algorithm 3: CATCH the original algorithms description can be found at **[[5],algorithm 2]**

Input: $p_1[], p_2[], t_{\max}, \Gamma, N$
Output: $TCA + r_{TCA}$
$r_{TCA} = \inf$
$t_{TCA} = \inf$
$a = 0$
$b = \Gamma$
While $b < t_{\max}$ **do**:
  Fit CPP $p_{\dot{f}}$ with order N to $\dot{f}(t)$ according to **[[5], Eq.15]** over the interval $[a,b]$
  Fit CPP with order N to f_x, f_y, f_z over the interval $[a,b]$
  Find the roots of $p_{\dot{f}}$
  for each root $t*$ **do:**
    calculate the distance r at $t*$ using **[[5], Eq.6]**
    if $r < r_{TCA}$**:**
      $r_{TCA} = r$
      $t_{TCA} = t*$
    end
  end
  $a = a + b$
  $b = b + \Gamma$
end

The algorithm needs N+1 points in time in each Gamma interval in order to create CPP of order N. After calculating the CPP coefficients we can use them to create a special NxN matrix called the companion Matrix **[[5],Eq.18]** and the eigen values of this matrix are the polynomial roots. Using the roots, we found and creating CPP for $p(t)_{f_x}, p(t)_{f_y}, p(t)_{f_z}$ we can calculate the minimal distance in each interval and eventually the TCA and respective distance in $[0, t_{\max}]$. The problem with CATCH is the cost of finding the roots, which is the cost of finding eigen values for an NxN matrix, to deal with it, Dr. Elad describe in his article **[[5],part 4]** that we can get sufficient results for both runtime and error size, using degree of 16 for the polynomial. Using a constant degree give us deterministic run times and the size of the error is small enough for the required result.

Variables and notation in CATCH Algorithm:
The CATCH algorithm uses Chebyshev Proxy Polynomials (CPP) for more precise approximations of the distance between orbiting objects. It is generally more computationally expensive than **SBO-ANCAS [6]**, but it can deliver higher accuracy.
Here is a breakdown of the role of each variable, including the input, output, and other different notations used in the algorithm:
Input Variables:

- $\boldsymbol{p_1}[]$**:** This array contains the **position vectors** of the first object (e.g., satellite) at multiple time points. Each element in $\boldsymbol{p_1}[]$ represents the 3D position of the object at a specific time, giving its location in space.
- $\boldsymbol{p_2}[]$**:** This array contains the **position vectors** of the second object (e.g., another satellite or debris) at multiple time points. Like $\boldsymbol{p_1}[]$, each element represents the 3D position of this second object at specific times.
- $\boldsymbol{t_{max}}$**: Maximum time interval** for which the algorithm will search for the closest approach between the two objects. It defines the range of time over which the CATCH algorithm looks for the Time of Closest Approach (TCA). It bounds the period in which the algorithm calculates the minimum distance. The algorithm works over this time interval to fit the Chebyshev polynomials and find roots.
- **Γ:** Initial estimate or threshold for the minimal distance between the two objects. It can also represent the minimal acceptable distance for conjunctions in certain scenarios. The algorithm refines its search for the TCA by continuously updating the estimated distance between the two objects.
- **Degree of the Chebyshev polynomial (N)**: A parameter that controls the degree of the polynomial approximation used in the algorithm. Higher degrees provide more accurate results but at the cost of increased computational complexity.

Output Variables:

- $\mathbf{t_{tca}}$: The TCA when the two objects are at their minimum distance.
- $\mathbf{r_{tca}}$: The smallest distance between the two objects during their encounter, calculated at $\mathbf{t_{tca}}$.

Other notations:

- $\boldsymbol{f_x}, \boldsymbol{f_y}, \boldsymbol{f_z}$: The relative position components in the x, y, and z directions, used to calculate the relative distance between the two objects.
- $\dot{\boldsymbol{f}}(\boldsymbol{t})$:The derivative of the relative distance function, used to identify critical points (minima or maxima) where the distance between the objects might be minimized (i.e., the TCA).
- $\boldsymbol{p_{\dot{f}}}$: The Chebyshev Proxy Polynomial coefficients, which approximate the position or distance functions over time.
- $\boldsymbol{a}, \boldsymbol{b}$: The time boundaries defining the interval over which the CATCH algorithm operates, typically from the start time $a = 0$ to the maximum time $b = t_{max}$.

2.1.2. The Propagator

For most of the existing satellite, and for each future autonomous satellite, calculating its future orbit is already an integrated part in the satellite task, the satellite uses this

calculation to do minor changes to its orbit to fit the desired orbit. This future orbit calculation done on the satellite system in a software called Propagator. Using the current location and velocity of an object in orbit (for example our satellite) and a point in time (for example ten minutes from now) the Propagator can calculate the object location and velocity in the given point in time. The propagator uses a forces model to find the future orbit, this force model is different between different propagators and can affect the result's accuracy and the calculation time. In this project we use a propagator called **Standard General Perturbations Satellite Orbit Model 4(SGP4) [24]**, SGP4 is old and famous propagator used for research that known for its fast run time and there are many available implementations we can use **[16][30]**. SGP4 get the object data using format called **Two Lines Elements (TLE)** which consist of two lines of data, including the object location, velocity, the corresponding time, the average number of revolutions per day (Mean Motion) and more.
To our propagator has two tasks, the first is creating the data for each of the algorithms runs. Given a set of TLE from the user we can create a set of point in time for two satellites and run the algorithms with it. The second task is using a propagator as part of the **SBO-ANCAS [6]** algorithm. SBO-ANCAS needs to sample new points as part of the algorithms so a propagator is needed.

2.1.3. Comparative Analysis of Satellite Boards for Algorithm Testing

2.1.3.1. Background and Research Approach

As part of the continuation of the project "Feasibility Analysis and Performance Testing of Collision Detection Algorithms for Satellites", an in-depth study we conducted on the satellite **ISIS OBC [12]** provided by Dr. Elad Dannenberg. This research involved gathering data through consultations with multiple companies, including SatSearch **[22]** , GOMSpace **[9]**, and ISILAUNCH **[13]**, and reviewing various datasheets **[17]** and other technical resources. The goal was to acquire a comprehensive understanding of the ISIS OBC's specifications, such as processor speed, RAM capacity, non-volatile storage, code storage, power consumption, and built-in Real-Time Clock (RTC). The goal was to find a cost-effective alternative that could support algorithm testing in a terrestrial environment without requiring space-grade durability.

2.1.3.2. Satellite ISIS OBC

To understand the environment in which our project will be executed, it was essential to study the ISIS OBC satellite system. The ISIS OBC is a specialized satellite board developed by Innovative **Solutions In Space B.V. (ISIS) [12]**, a company that provides various systems for small satellites, such as CubeSats.
Key Features of the ISIS OBC:

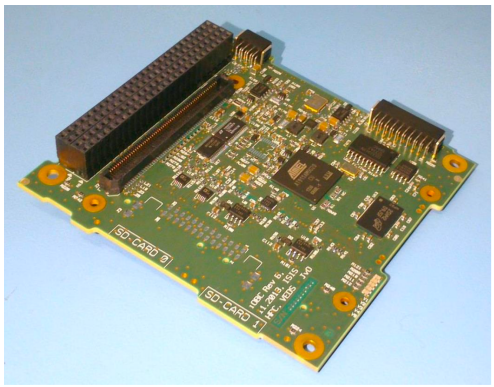
1. **Processor**: The ISIS OBC is equipped with a 400 MHz ARM9 (AT91SAM9G20) processor. This processor is widely used in embedded systems due to its balance of power efficiency and performance, making it ideal for satellite operations where energy conservation is critical.
2. **Memory**: It comes with 32MB of SDRAM, which provides sufficient space for running multiple satellite applications simultaneously, including real-time data processing, communication, and algorithm execution.
3. **Non-Volatile Storage**: The OBC is equipped with two 2GB SD cards configured with a FAT32 file system. This allows the board to store important mission data, including telemetry, payload data, and algorithm results, even in

the event of power loss. The redundancy in storage provides a safeguard against data corruption or failure in one of the storage systems.

4. **Real-Time Clock (RTC)**: A dual-redundant RTC ensures accurate and reliable timekeeping, which is critical for coordinating satellite operations and tasks such as collision detection. This feature ensures that the OBC can handle time-sensitive processes, especially those reliant on precise timing, such as navigation and communication.
5. **Power Consumption**: Operating at 380mW, the ISIS OBC is highly energy-efficient, which is essential for long-term satellite missions where power is often limited. This low power consumption also ensures minimal heat generation, preventing the system from overheating during extended operations.
6. **Operating System**: The OBC runs on FreeRTOS, a real-time operating system known for its efficient task scheduling and low resource usage. FreeRTOS is popular in space applications because it provides the necessary tools for real-time applications, such as collision detection algorithms, while maintaining a low memory and processing overhead.
7. **Modularity and Expandability**: The ISIS OBC is designed to be highly modular, meaning it can be customized to integrate with different payloads and subsystems required for specific satellite missions. This modularity makes it versatile for a wide range of CubeSat missions, from research and educational missions to commercial space operations.

The ISIS OBC's combination of processing power, memory, real-time capabilities, and power efficiency makes it an ideal fit for our system. It ensures that the collision detection algorithms will run efficiently and reliably under the constraints typically encountered in satellite environments. The OBC’s processing capacity, memory, and real-time accuracy align with the project’s requirements for handling complex algorithms, ensuring that the test results will be both relevant and reliable for the final satellite application.

Image 1: ISIS OBC



The ISIS OBC serves as the reference system for our project because of its robust design and suitability for small satellite missions. Its balance of processing power, energy efficiency, and real-time capability make it an ideal candidate for testing collision detection algorithms in a constrained environment. By closely studying this system, we aim to find alternative OBCs that align with its specifications for cost-effective algorithm testing in terrestrial environments.

Understanding the ISIS OBC's architecture helps us ensure that any algorithm optimizations and tests we perform will be relevant when transferred to the actual satellite system. This background is crucial for selecting appropriate alternative boards that can simulate the ISIS OBC's performance in a laboratory setting.

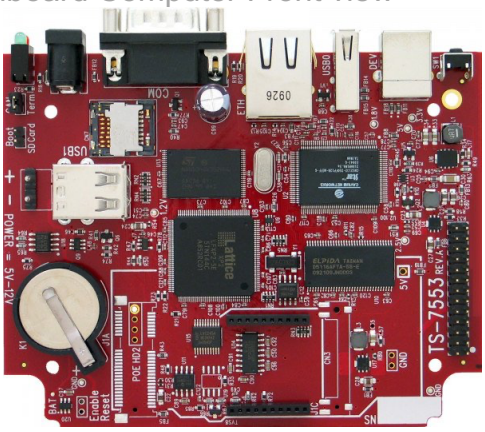
2.1.3.2.1. The Selected OBC in phase A of the project

In phase A of the project, after a thorough evaluation, the **TS-7553 [29]** has been identified as the closest match to the **ISIS OBC [12]** based on key specifications, including processor architecture, memory, and storage. Although it presents some drawbacks, such as higher power consumption and the absence of storage redundancy, its architectural similarity to the ISIS OBC makes it an optimal candidate for our algorithm testing.

TS-7553 [17][29] specifications:

- **Processor:** 250MHz ARM922T, slower than the ISIS OBC but with 64MB DDR2 RAM, double that of the ISIS OBC.
- **Power:** 3.2W, significantly higher than the ISIS OBC.
- **Cost:** $180 USD.
- **Modifications:** Requires additional MicroSD for storage ($5-$10) and FRAM ($10-$20). FreeRTOS needs to be ported from Linux (~$15-$30). It has a single real-time clock (RTC).

Image 2: TS-7553 Onboard Computer Front view



2.1.3.2.2. Transition from the TS-7553 OBC to the Raspberry Pi 5 for Algorithms Testing

As part of the continued development and optimization of our project, we re-evaluated our hardware platform to ensure that it provided the most suitable environment for implementing and testing algorithms. In Part A, we identified the **TS-7553 OBC [17][29]** as the closest match to the **ISIS OBC [12]**, with a 250MHz ARM9 processor, 64MB DDR2 RAM, and similar architecture, making it an ideal candidate for testing our collision detection algorithms in a constrained environment. This initial choice was based on the board's architectural similarities to the ISIS OBC **[12]** provided by Dr. Elad Dannenberg, ensuring that the performance insights we gathered would be transferable to a satellite grade system.

However, as we progressed through our implementation phase, a shortage of TS-7553 OBC **[17][29]** inventory forced us to move to another available option. After a detailed assessment of our requirements, including greater flexibility, higher computational efficiency, and improved software support while adhering to the real-time operational constraints of satellite systems, we moved to the **Raspberry Pi 5 [19]** as our primary hardware platform.

The **Raspberry Pi 5 [19]** features a 2.4GHz quad-core ARM Cortex-A76 processor, significantly enhancing computational efficiency while remaining within reasonable power consumption limits. Additionally, with up to 16GB of LPDDR4X RAM, high-speed USB 3.0, Gigabit Ethernet, and a real-time clock (RTC), it provides a versatile environment for testing our algorithms under conditions similar to those of satellite on-board computers. These specifications enable efficient execution of CATCH, SBO-ANCAS, and ANCAS algorithms, supporting high throughput numerical computations essential for determining Time of Closest Approach (TCA).

The board's Linux-based environment allows for flexible development, debugging, and optimization, ensuring that our algorithms can be adapted for deployment on satellite grade systems. Furthermore, the availability of PCIe 2.0, dual 4K HDMI outputs, and expanded I/O capabilities ensures a streamlined integration with our test station, facilitating efficient performance analysis and result validation. The only drawback to choosing the Raspberry Pi 5 is that it does not include an FPGA, which may limit the ability to perform certain hardware accelerated functions inherent to some satellite applications. However, this limitation can be mitigated by leveraging high performance software optimizations and the option to integrate external FPGA modules if needed, thereby reducing the overall impact on our testing and development processes.

By leveraging the **Raspberry Pi 5 [19]** , we ensure that our feasibility testing remains aligned with the objectives of our project while taking advantage of a modern, cost-effective, and high-performance computing platform for algorithm evaluation. This transition allows us to conduct a more comprehensive performance assessment while maintaining adaptability for future enhancements in satellite OBC.

Image 3: Raspberry Pi 5 [19] Front View

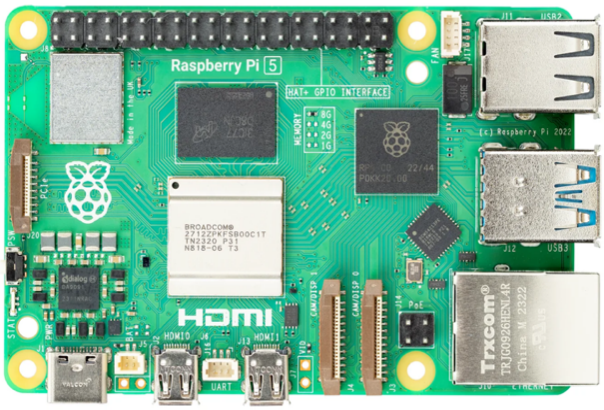
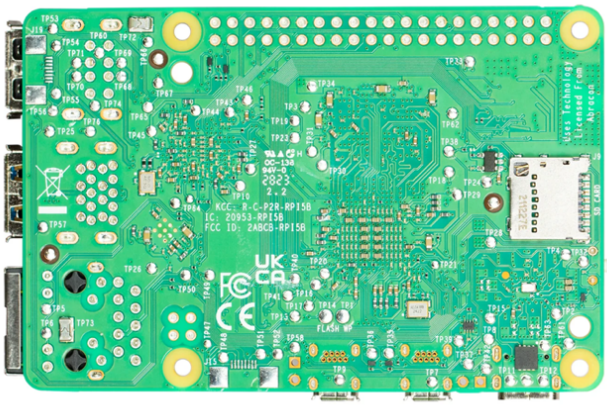


Image 4: Raspberry Pi 5 [19] Back View



2.1.3.2.3. Comparison Between Raspberry Pi 5 and ISIS OBC

The following table presents a structured comparison between the **Raspberry Pi 5 [19]** and the **ISIS OBC [12]**, based on the key criteria provided in the dataset:

Table 1: Comparative Analysis Table of the ISIS OBC **[12]** and the Raspberry Pi 5 **[19]**.

	ISIS OBC	**Raspberry Pi 5**
Processor	400MHz, 32-bit ARM9 (AT91SAM9G20)	Quad-core ARM Cortex-A76 @ 2.4 GHz
RAM	32MB SDRAM	8GB LPDDR4X
Non-volatile data storage	2x2GB SD-Cards with FAT32 file system 256kB FRAM (high endurance and fast read/writes)	microSD card slot, NVMe (via PCIe 2.0)
Code storage	1MB NOR-Flash	microSD / external SSD storage
Timing	2 redundant real-time Clocks	Real-Time Clock (RTC)
Average power consumption	380mW, typical usage @ 3.3V supply	~5W (higher than ISIS OBC)
Operating System	FreeRTOS	Linux-based OS (Raspberry Pi OS, Ubuntu, etc.)
Price ranges	6000 EUR	**$100 USD**

The transition from the TS-7553 to the Raspberry Pi 5 was initially driven by the unavailability of the TS-7553 board in stock. However, beyond necessity, the Raspberry Pi 5 offers several key advantages that make it a strong candidate for testing and developing our collision detection algorithms, its quad-core ARM Cortex-A76 processor (2.4GHz) far surpasses the 400MHz ARM9 in the ISIS OBC, enabling faster execution of complex algorithms such as CATCH, SBO-ANCAS, and ANCAS. The large RAM capacity (8GB LPDDR4X) allows for efficient handling of large datasets compared to the limited 32MB SDRAM in the ISIS OBC, it runs a Linux-based environment, which provides a more flexible development ecosystem than the ISIS OBC's FreeRTOS, allowing for seamless debugging, profiling, and performance

optimization. Furthermore, the Raspberry Pi's widespread availability and strong community support mean that additional development requirements can be met with open-source tools and libraries, making it easier to implement, test, and refine our algorithms before transitioning them to a satellite-grade OBC. The main limitations are its higher power consumption (~5W) compared to the ISIS OBC (380mW), meaning it is less energy-efficient for space applications; the absence of an FPGA, which prevents the direct implementation of certain real time hardware acceleration features, although this can be mitigated through software optimizations or by integrating an external FPGA module if necessary; and the lack of dual-redundant RTCs, which are crucial for ensuring fault tolerance in timing for space applications. While the Raspberry Pi 5 is not a direct replacement for the ISIS OBC in an actual space environment, it provides an affordable, high-performance, and flexible platform for terrestrial testing of our algorithms, and its superior processing power, RAM, storage capabilities, and development environment make it an excellent choice for feasibility testing, ensuring that our algorithms can be optimized before being transferred to a more constrained space-grade OBC.

2.2. Goals

In this project, we aim to create a comprehensive and adaptable system for testing collision detection algorithms under conditions approximating those of an operational satellite on-board computer (OBC). This process includes implementing, developing and executing the algorithms on the selected satellite OBC, evaluating their feasibility, and conducting in depth data analyses. Our goal is to generate a comprehensive set of experimental results, derived from testing a variety of input types, that will either confirm or refute the feasibility of executing TCA calculations on the satellite's OBC. Moreover, this project aims to provide Dr. Elad Dannenberg with a generic and sufficiently flexible testing system that can support any future update that might be needed, for example adding algorithms or algorithms variants, adding communication types and so on. we want our application to be available for any follow up team trying to test similar things or different algorithms in a similar environment, to be used as a base or reference for future uses.

Accordingly, we have defined three main goals for this project :

2.2.1. Implementing and Deploying the Algorithms

Our first goal is to implement and deploy collision detection algorithms on a satellite's OBC. Until now, the **SBO-ANCAS [6]** and **CATCH [5]** algorithms were primarily implemented and tested in MATLAB as part of the initial research studies **[5] [6]** To ensure their practical usability, we need to adapt these algorithms to run efficiently on a satellite OBC while considering the hardware constraints, such as processing power, memory, and energy consumption. Our work involves converting the algorithms into optimized C++ implementations, adapting them to the operating system of the selected satellite board, and ensuring their compatibility with real-time execution.

2.2.2. Creating a Testing System

To test the feasibility of running the algorithms on satellites OBC we needed a system fitting for running test and collecting data and results. We needed to run the algorithm on a satellite OBC with a given set of data and parameters as input, to get the output and run time and to save the results and test related parameters in our data set in order to collect enough data on the algorithms expected run time and accuracy in different scenarios. The Testing System needed to be flexible enough to run the algorithms on different machine and environments and manage and collect the data

well. We needed the system both for running the feasibility analysis ourself and for leaving it for Dr. Elad to use for his research.

2.2.3. Feasibility Testing and Analysis

The last part of our project is to conduct a feasibility analysis of running the TCA algorithms on the selected satellite OBC using our system. We needed to run our system with different types of inputs, different types of algorithms and different parameters for each algorithm. Through this analysis, we aim to determine whether these algorithms can operate autonomously on a satellite, ensuring timely and accurate collision detection while meeting power and hardware limitations and what the optimal parameters values are that yield a balance between good accuracy and moderate runtime for the algorithms.

In this project we expect to finish with concrete and comprehensive proof of feasibility or infeasibility of the task of running the TCA calculations on the satellite OBC.

2.3. Users

Our system has two types of users. The first is Dr. Elad and his associates, who aim to test the algorithms on a real satellite's OBC. The second group includes our team, conducting feasibility tests on a real satellite's OBC as part of our project, as well as the previous team that worked on the project and any future teams that may contribute to the ongoing effort of developing and validating autonomous satellite capabilities. (Several other teams are already working on projects related to this goal).

3. The solution

3.1. Algorithms Analysis

3.1.1. Algorithms Runtime Complexity

3.1.1.1. ANCAS Time Complexity

In ANCAS **[1],** for each set of 4 data points we need to create 4 cubic polynomials, one for the relative distance derivative and 3 for the relative distance X, Y, Z. each cubic polynomials required coefficients calculation which consist of 4 equations **[[1],Eq 1f-1j]**, meaning the complexity of finding the polynomial coefficients is constant. To map the time points to the interval [0,1] we use a simple calculation for each point 4 times, one for each point.

Finding the solution for a 3rd degree equations is quite simple, using a given formula with a constant run time we get between 1 to 3 real result.

For each of the roots we found, we calculate the distance using **[[1], Eq.6]** and check if we found a smaller distance. In the worst case we check 3 times.

Meaning for each set of 4 data points the complexity is (where k is a constant number):

$$O(4*k_{coefficients}+k_{roots}+3*k_{dist})=O(1)$$

Calculating the complexity for finding the TCA over **n** data points means we check the first 4 points and for each iteration after that we use the last point from the previous iteration as the first points meaning we need 3 new points, so we need to do $\left\lceil\frac{n-4}{3}\right\rceil+1=\left\lceil\frac{n-1}{3}\right\rceil\leq\frac{n}{3}$ iterations.

The complexity of running ANCAS on **n** data points is:

$$O\left(\frac{n}{3}*k_{ancasIteration}\right)=O(n)$$

3.1.1.2. SBO-ANCAS Time Complexity

In **SBO-ANCAS [6]**, we are going over a set of n initial points, the outer loop check the first 4 points and for each iteration after that we use the last point from the previous iteration as the first points meaning we need 3 new points, so we need to do $\left\lceil \frac{n-4}{3} \right\rceil + 1 = \left\lceil \frac{n-1}{3} \right\rceil \leq \frac{n}{3}$ outer iterations.

In the inner loop we run until we reach the desire tolerance in distance and time, thus the number of inner iterations depends on the size of tolerance in distance, the size of tolerance in time, the error of the polynomial approximation, the change in relative distance in time between the 2 objects and the distance between the initial time points. For each inner iteration we use the propagator to sample a single point in time.

Let's start by looking at the tolerance in time condition for the inner loop, say we have 4 time points, $t_1 - t_4$, with the initial distance between 2 time points of $t_{distance}$, and tolerance in time TOL_t. To get to the desired tolerance we need the distance between t_m to the other points to be smaller than TOL_t. which means that at the last iteration we get:

$interval\ size_i = TOL_t \leq t_{4i} - t_{1i} \leq 2TOL_t$.

To find the worst-case scenario for the number of iterations we need to consider the smallest possible decrement in the total time interval per iteration. In the following example we can see that theoretically there is no limit to how many iterations we get.

We start with a set of 4 points, $t_1 - t_4$:

$t_{m1} = t_1 + \varepsilon \rightarrow t_{new} = \{t_1, t_{m1}, t_2, t_3\}$

$t_{m2} = t_3 - \varepsilon \rightarrow t_{new} = \{t_{m1}, t_2, t_{m2}, t_3\}$

$t_{m3} = t_{m1} + \varepsilon \rightarrow t_{new} = \{t_{m1}, t_{m3}, t_2, t_{m2}\}$

$t_{m4} = t_{m2} - \varepsilon \rightarrow t_{new} = \{t_{m3}, t_2, t_{m4}, t_{m2}\}$

And so on.

And if we look at the interval size in each step:

$interval\ size_0 = t_3 - t_1 = 2 * t_{distance}$

$interval\ size_1 = t_3 - t_1 - \varepsilon = 2 * t_{distance} - \varepsilon$

$interval\ size_2 = t_3 - t_1 - \varepsilon - \varepsilon = 2 * t_{distance} - 2 * \varepsilon$

...

$interval\ size_i = 2 * t_{distance} - i * \varepsilon$

And we continue until:

$interval\ size_k \leq TOL_t$

$2 * t_{distance} - k * \varepsilon \leq TOL_t$

$2 * t_{distance} - TOL_t \leq k * \varepsilon$

$$\frac{2 * t_{distance} - TOL_t}{\varepsilon} \leq k \rightarrow O\left(\frac{t_{distance} - TOL_t}{\varepsilon} * P\right)$$

where P = complexity of getting a single point from the propogator

We expect the distance between 2 points, $t_{distance}$, to be bigger than the tolerance and with a small enough $\varepsilon \rightarrow 0$ we get:

$$\lim_{\varepsilon \to \infty} \left(\frac{t_{distance} - TOL_t}{\varepsilon} * P\right) = \infty$$

Practically that not the case because there is a limit on how many small numbers we can fit between any set of 2 initial values, depending on the value of $t_{distance}$, the specific implementation and the precision of the variables. For example, if we use an

IEEE 754 double-precision variable, the smallest possible value is about 5×10^{-324} so we will get a large but final number of iterations.

Let's look at the tolerance in distance, we compare two values of the distance in the same point in time. The first is the value we got from the polynomial approximation and the second is the value we got from the propagator. The different is because the error of the polynomial approximation. The closer the points in time will be, the smaller the change in distance will be and we can expect smaller errors. So, with a small enough time step we will reach the desired tolerance.

3.1.1.3. CATCH Time Complexity

In **CATCH** [5] algorithm we iterate through the number of time points in our external loop, $n = \frac{t_{max}}{\Gamma} \cdot N_{deg} = number\ of\ \ points$

Where t_{max} is the is end boundry in the time range where we're looking for mininal disdance, and Γ equal to half of the smaller revolution time of the object **[**[5]**,part 4]**, The value of N_{deg} is the order of the polynomial, while we can change the chosen value of N, it was determined that $N=16$ give sufficient results.
Inside the loop we're doing the following steps:

1. Fit the CPP of order N_{deg} to $\dot{f}(t)$, p_x, p_y, p_z over each interval of points:
   Assuming the arithmetic operations we use are basic operation done in time complexity of $O(1)$, we calculate the Chebyshev polynomials **[5]**. We'll iterate through $N_{deg}+1$ points, which is a constant in our case, meaning that the time complexity will also be constant. Each iteration requires us to sample a new time point which will be our input parameter x, calculating the interpolation matrix with size of $(N_{deg}+1)(N_{deg}+1)$ which is also constant.
   The complexity of this step is: $O(N_{deg}) \cdot \left(O(N_{deg}) \cdot \left(O(N_{deg}) \cdot O(N_{deg})\right) + O(1)\right) = O(1)$

2. Finding the roots for P_f will consist of calculating the companion matrix with a size of ${N_{deg}}^2$ and finding the eigen values, using the complexity of matrix multiplication for this step, the complexity will be $O\left({N_{deg}}^3\right) = O(1)$, rescaling each eigen value to the actual coefficient value also takes constant time.

3. For each time point we'll calculate in our interval we'll check if we found a new minimal distance, if we did, we'll update the minimum distance and the time of occurrence. This step also has a constant time complexity.

It means that the only inputs that determines our time complexity are the values of how long each interval time, and how long in the future we want to look it,
meaning the complexity equals the number of different time-points we measure, which is: $O(n)$

3.1.2. Space complexity

The space complexity of the algorithms is the same. SBO-ANCAS, ANCAS and CATCH uses a constant number of internal variables to help with the calculations. Because our task is finding a minimum, we only need one variable to store the current minimum without any dependency for the input size. We also use some internal

variables representing the polynomial and other related logics. The only memory that is related to the size of the input is the input itself. The input consists of 2 location vectors, 2 velocity vectors and the time point value for each time point in our data set, so we can see that the size of memory the input uses is linear to the number of points we need to test. We get constant space complexity for the algorithms themselves and linear to the number of points for the input: $O(n+1) = O(n)$

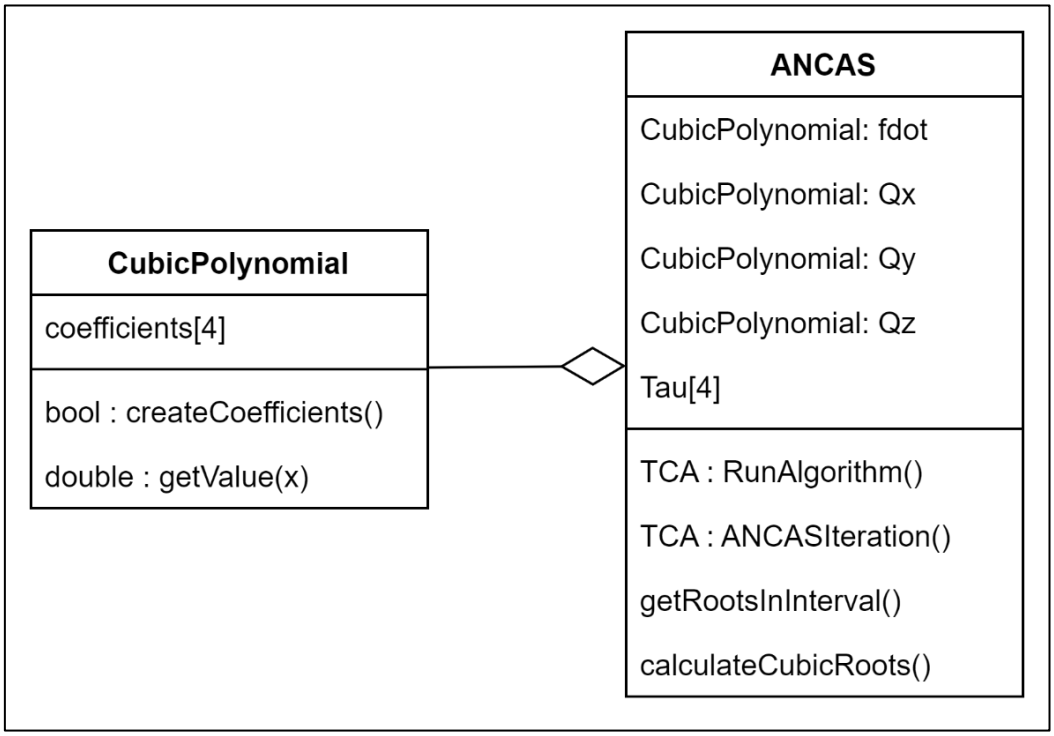
*For SBO-ANCAS, we assume that the propogator uses a constant amount of memory, but this may not always be the case.

3.2. Algorithms implementation

3.2.1. ANCAS Implementation

ANCAS **[1]** implementation is pretty straight forward, there are no inner loop or complicated algorithms in use here, we only need to find the roots of a cubic polynomial, and it can be done using a formula. We kept the implementation as simple and straight forward as possible, only taking out the code for each iteration logic, including fitting the polynomial and finding the roots, into a different function so we can reuse the code for SBO-ANCAS **[6]**. We created a class representing a cubic polynomial with functions for creating the coefficients and for getting a value at a point x. we created a function for finding the roots using the cubic polynomial formula and created unit tests that check the roots finding for a polynomial with 0 to 3 real roots in range.

Diagram 1: Class diagram for ANCAS. Including a function for calculating the roots of a cubic polynomial used by a function for getting the roots in the interval 0,1. The function for ANCAS Iteration return the found minimum and time, used for a single ANCAS iteration.

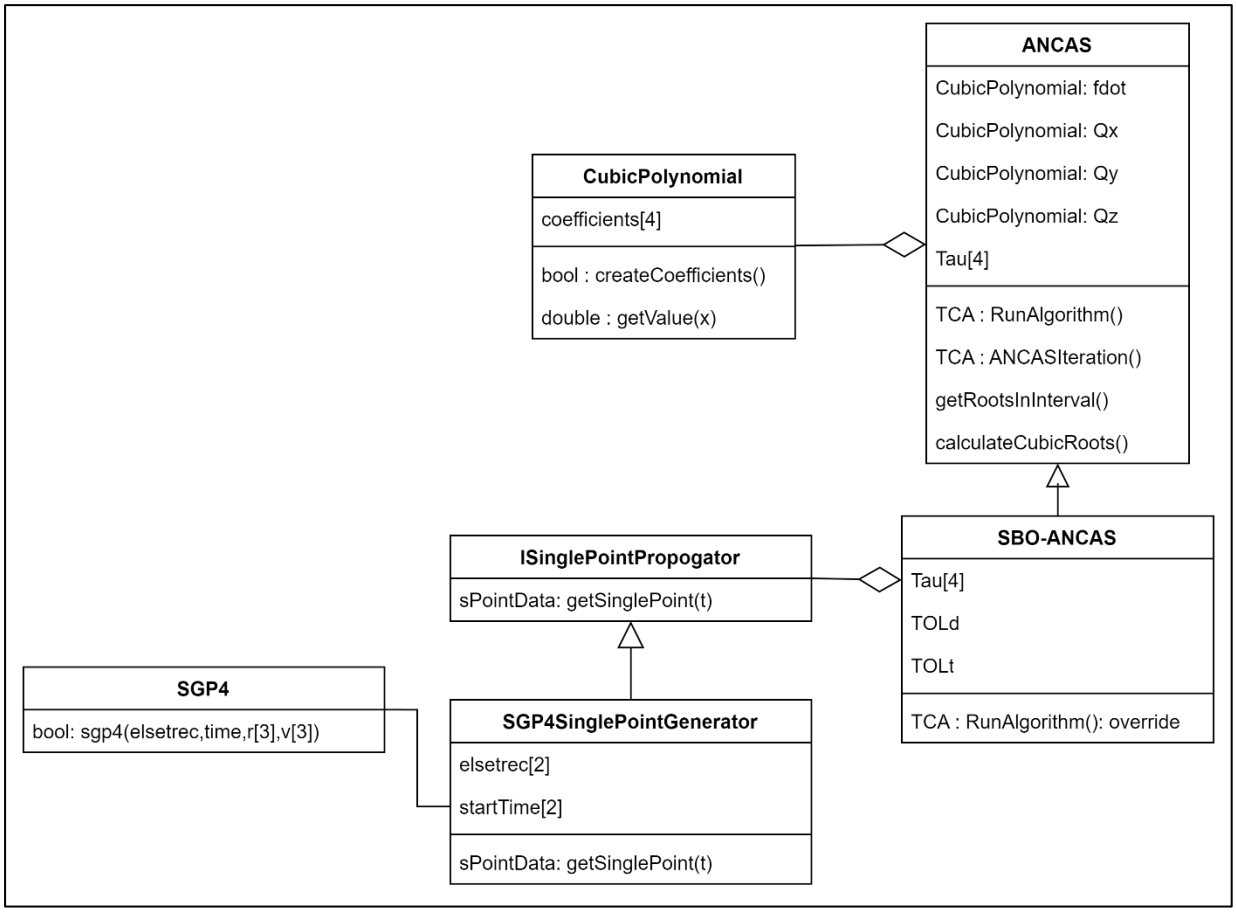


3.2.2. SBO-ANCAS Implementation

SBO-ANCAS [6] acts similar to ANCAS **[1]** in every iteration, initialize the polynomials, finding the roots and so on. To avoid rewriting the same code we inherited ANCAS and only needed to override the RunAlgorithm function. We added an interface for the propagator SBO-ANCAS uses, because we only need a single point in time every time,

we called it SinglePointPropogator. We implemented the interface using **SGP4 [25]** and used it for our testing. Additionally, SBO-ANCAS needed the tolerances in both time and distance as input.

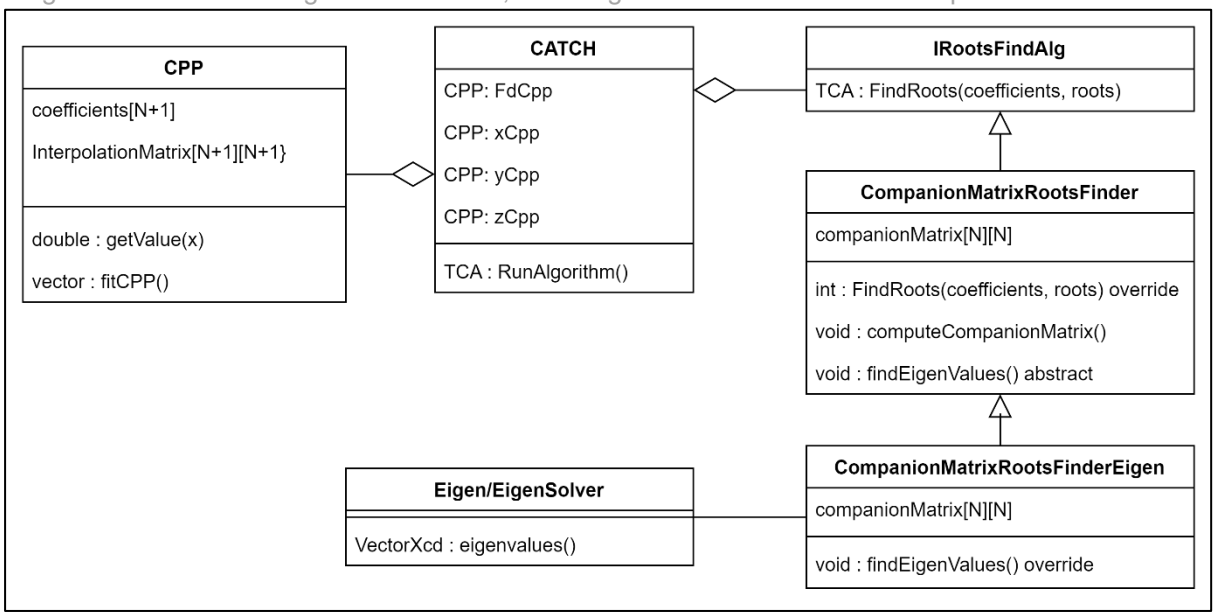
Diagram 2: Class diagram for SBO-ANCAS. Including the Propagator interface and implementation and the tolerances.



3.2.3. CATCH Implementation

CATCH implementation required implementing the Chebyshev Proxy Polynomials (CPP) class, with function for calculating the polynomial coefficients and to get the value at a point x. we needed the freedom to use different variations of the roots finding to check different libraries so we separated the root finding problem into a different interface. The CATCH class uses 4 CPP, for Fd,x,y,z, additionally it uses the Rootfinder interface to get the polynomial roots in each step. We implemented the CompanionMatrixRootFinder based on the algorithm described in the CATCH's article [5], and we tried two libraries for finding the Eigenvalues of the Companion Matrix. We implemented using **Eigen [28]**.

Diagram 3: Class diagram for CATCH, including Rootfinder interface and implementation.



3.3. Raspberry Pi 5 for Algorithm Testing

Following an initial assessment of the **TS-7553 OBC [17][29]**, which was selected for its resemblance to the **ISIS OBC [12]** and preliminary suitability for real-time applications, we found that it would remain out of stock for an extended period. Furthermore, new project requirements emerged, calling for enhanced software flexibility, a wider range of input/output interfaces, and more robust computational power to accommodate increasingly complex testing environments. Therefore, after evaluating multiple alternatives, we elected to transition to the **Raspberry Pi 5 [19]**. The **Raspberry Pi 5 [19]** features a quad-core ARM Cortex-A76 processor operating at 2.4 GHz, up to 16 GB of LPDDR4X memory, and an integrated real-time clock (RTC). These capabilities enable efficient execution of the **CATCH [5]**, **SBO-ANCAS [6]**, and **ANCAS[1]** algorithms, while maintaining power levels appropriate for satellite boards. In parallel, the Linux-based operating system simplifies development and debugging, and enhances our ability to optimize and analyze performance. In addition, the broad connectivity options, such as PCIe 2.0, USB 3.0, and Fast Ethernet facilitate more seamless integration with our test station and support a range of load and data management scenarios. Beyond resolving supply limitations, the transition to the **Raspberry Pi 5 [19]** also meets the core demands of satellite OBC, addressing both performance requirements and support for real-time operations and future hardware expansions. As such, it proves an ideal fit for the project's needs and enables more in depth and varied algorithm testing, all while preserving scalability and operational flexibility moving forward.

Image 5: Raspberry Pi 5 [19] Front View

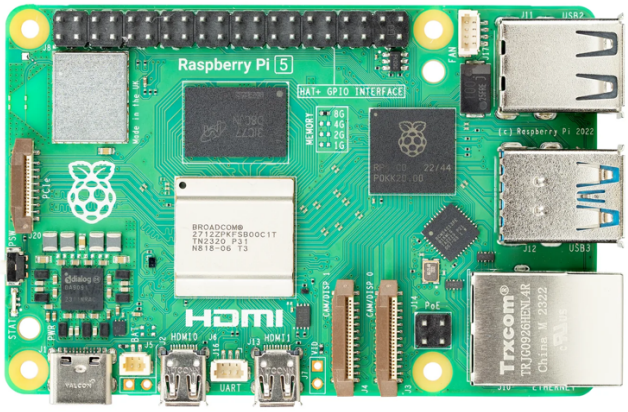
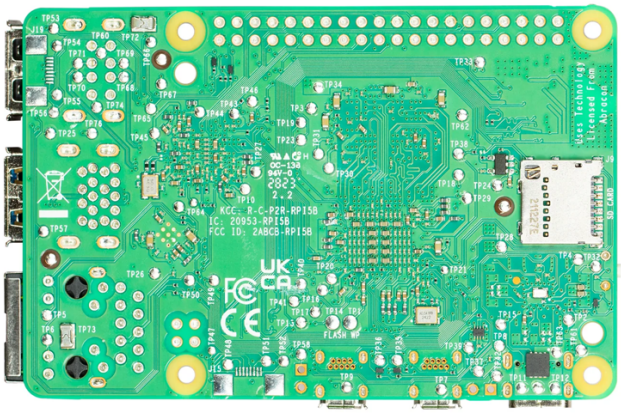


Image 6: Raspberry Pi 5 [19] Back View



3.3.1. Selection of the Optimal Communication Protocol for Our System

As we transitioned from the **TS-7553 OBC[17][29]** to the **Raspberry Pi 5 [19]** , we reassessed our communication protocol to ensure it best supports real-time data exchange between the selected OBC **Raspberry Pi 5 [19]** and the Test Station System. Given our project's focus on simulating real-world satellite operations as closely as possible, we selected Cross-Platform TCP over Wi-Fi as our primary communication method.

Rationale for Selecting Cross-Platform TCP Over Wi-Fi:

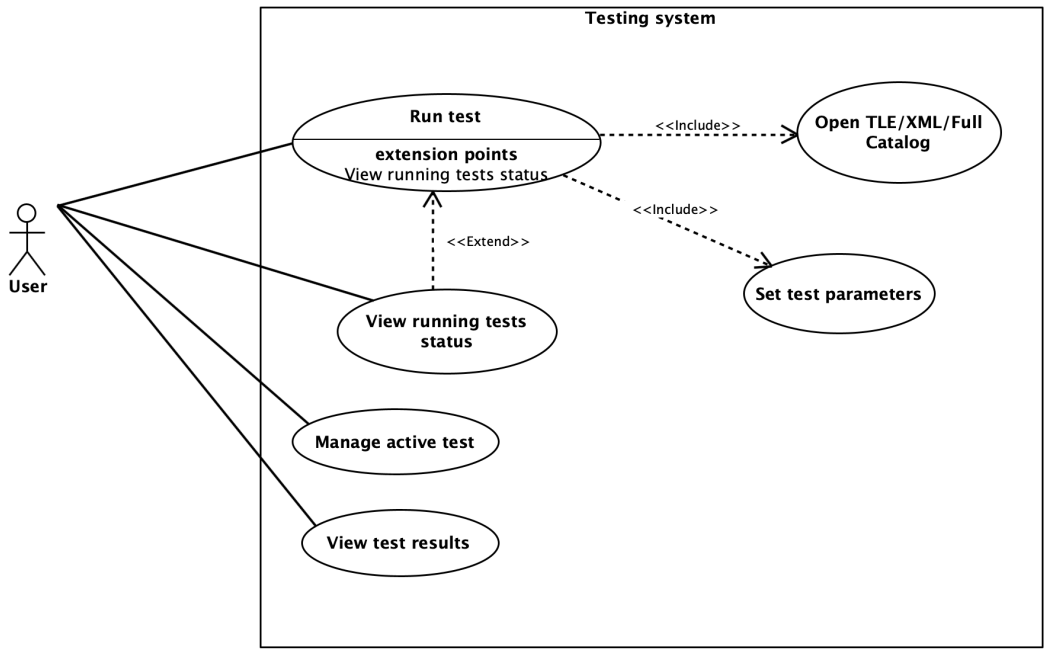
- Simulation of Real Satellite Communication:
  Unlike direct-wired connections, wireless communication more accurately represents the data transmission challenges of real satellite operations.
  Satellites typically rely on wireless protocols, such as S-band, X-band, or Ka-band for transmitting telemetry and scientific data to ground stations, and while TCP over Wi-Fi does not replicate space communication exactly, it provides a closer approximation than a traditional wired Ethernet connection.
- Reliability and Data Integrity:
  TCP ensures error detection, retransmission, and ordered data delivery, making it highly reliable for transmitting critical data such as collision detection results.
  Unlike UDP, which allows packet loss, TCP guarantees that all data packets are received correctly, ensuring accurate algorithm performance analysis.
- Flexibility Across Platforms and Remote Testing Capability:
  Using TCP over Wi-Fi eliminates the constraints of physical cabling, allowing for more flexible test setups.
  It enables remote communication between the **Raspberry Pi 5 [19]** and the Test Station, simulating how ground control would receive data from an actual satellite.
- High-Speed and Low-Latency Performance:
  The **Raspberry Pi 5 [19]** supports Gigabit Wi-Fi (802.11ac), allowing for high-speed and low-latency communication, crucial for real-time monitoring and execution of collision detection algorithms.
  This ensures that data transmission is efficient and responsive, supporting continuous algorithm performance evaluation.
- Ease of Implementation and Expandability:
  TCP is platform-independent, meaning it can run seamlessly across different operating systems, allowing for smooth communication between the Linux-based **Raspberry Pi 5 [19]** and the Windows-based Test Station.
  This wireless approach also provides an easy path for future expansions, such as integrating additional OBCs or testing in different environments without physical connection constraints.

By implementing Cross-Platform TCP over Wi-Fi, we ensure that our testing environment remains as close as possible to real-world satellite communication constraints, while maintaining high reliability, scalability, and efficient data exchange between the **Raspberry Pi 5 [19]** and the Test Station System. This decision allows us to effectively test and refine our algorithms in a way that aligns with the challenges of autonomous satellite operations.

3.4. The Testing System

In order to test the algorithms on the **Raspberry Pi 5 [19]** we needed to create a system fitting to run such tests, for each test we need to start by creating the data set for the test, the data set is composed of 3 parts, we need data in the points of time for ANCAS, CATCH and SBO-ANCAS. Lastly, we want to find out what the actual TCA is, to compare the algorithms results to. In order to find out the actual TCA we need to run the propagator with small time steps using the **SGP4 [24]** propagator. Creating the data will be done on both our own personal computer and the **Raspberry Pi 5 [19]**, and the testing equipment will be composed of 2 parts, the Test Station, our own computer, creating the data for each test and handling the test result. The second part is the Tested OBC, connected via **WIFI using Cross-Platform TCP Communication Channel** receiving the data from the Test station, generate data points using SGP4 **[24]**, running the algorithms, checking the run time and memory used, and sending the result back to the test station.

Diagram 4: System Use case diagram from the User point of view



Looking at the full application, the process of creating and running a test begins when the user navigates to the Test Creation page. At this point, the user may choose to create a single test or select a full catalog of satellites. In single-test mode, the user simply provides the necessary details for one test.

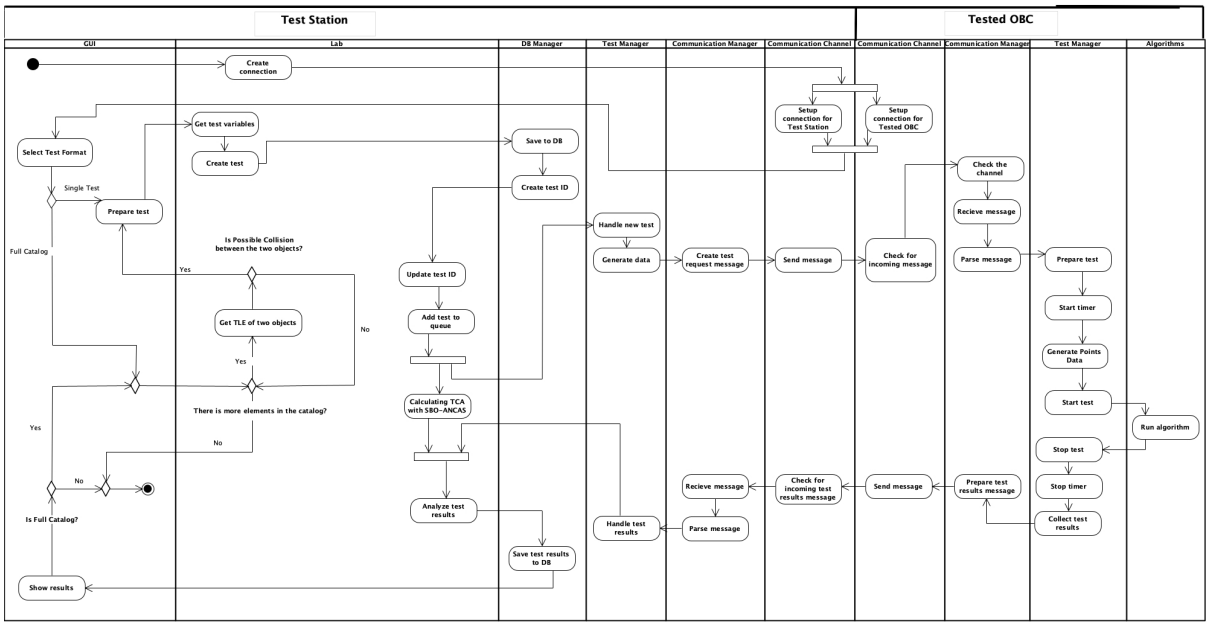
If the user opts for the catalog-based mode, the Lab component automatically iterates over each entry in a user-provided text file. To ensure only relevant satellites remain in the test queue, the Lab applies a 10 km collision threshold by computing the orbital parameters “a” (semi-major axis) and “e” (eccentricity) for the first satellite in the file and for each candidate satellite, in order to derive their respective perigees and apogees. It then defines:

$$q = \max(perigeeS, perigeeD)\ and\ Q = \min(apogeeS, apogeeD)$$

Any Satellite for which $(q - Q) > D$ $(Where\ D = 10Km)$ is automatically excluded from further testing.

Once the user's selection (single test or filtered catalog) is finalized, the GUI managers collect the input and call the Lab to create the test. The Lab then generates the data, saves the test, and passes it to the Test Manager, which forwards it to the Tested OBC App via the communication channels. Finally, the Tested OBC App uses the incoming data, generates data points, executes the algorithm, and returns the results all the way back to the Test Results page for display to the user.

Diagram 5: Top view activity diagram for running a test, with the full 2 Apps system



3.4.1. Test station system

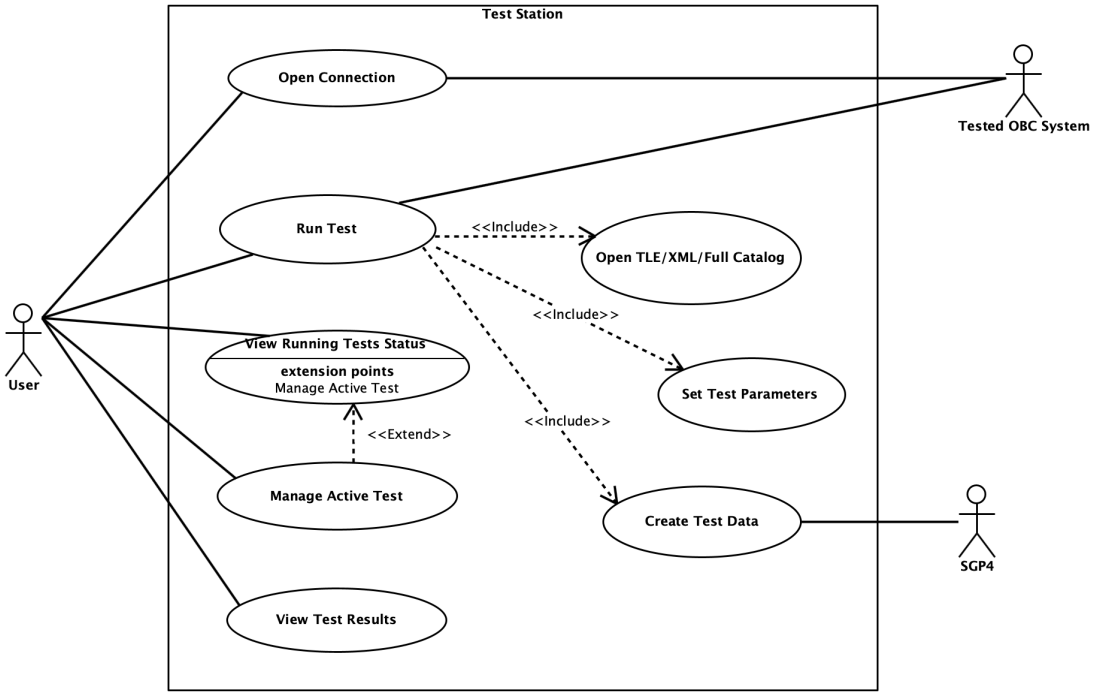
In order to accommodate a wide variety of tests with distinct inputs and expected outcomes, we developed a special message structure (described in detail in the following sections). This approach supports both single test runs and catalog-based runs, eliminating the need to modify code for every new test scenario. When running a catalog-based test, the Test Station parses a user-provided text file, configured in "one with all" or "all with all" mode, to automatically generate and queue multiple tests. As part of this process, it applies a 10 km collision threshold to ensure only relevant satellites remain in the test queue. Specifically, the system calculates the semi-major axis (a) and eccentricity (e) for the first satellite in the file and each candidate satellite, deriving their perigees and apogees. It then defines:

$$q = \max(perigeeS, perigeeD)\ and\ Q = \min(apogeeS, apogeeD)$$

Any Satellite for which $(q - Q) > D$ $(Where\ D = 10Km)$ is automatically excluded from further consideration.

Using these specialized messages, the testing station begins by creating fitting data for the tested OBC, sending the data to the tested OBC via our TCP communication channel. While awaiting the OBC's response, the Test station calculates the actual TCA using a high-precision **SGP4 [24]** propagator. After finishing the calculation and getting the test result from the tested on-board computer, the Test station analyzes the result and saves it into our result data set.

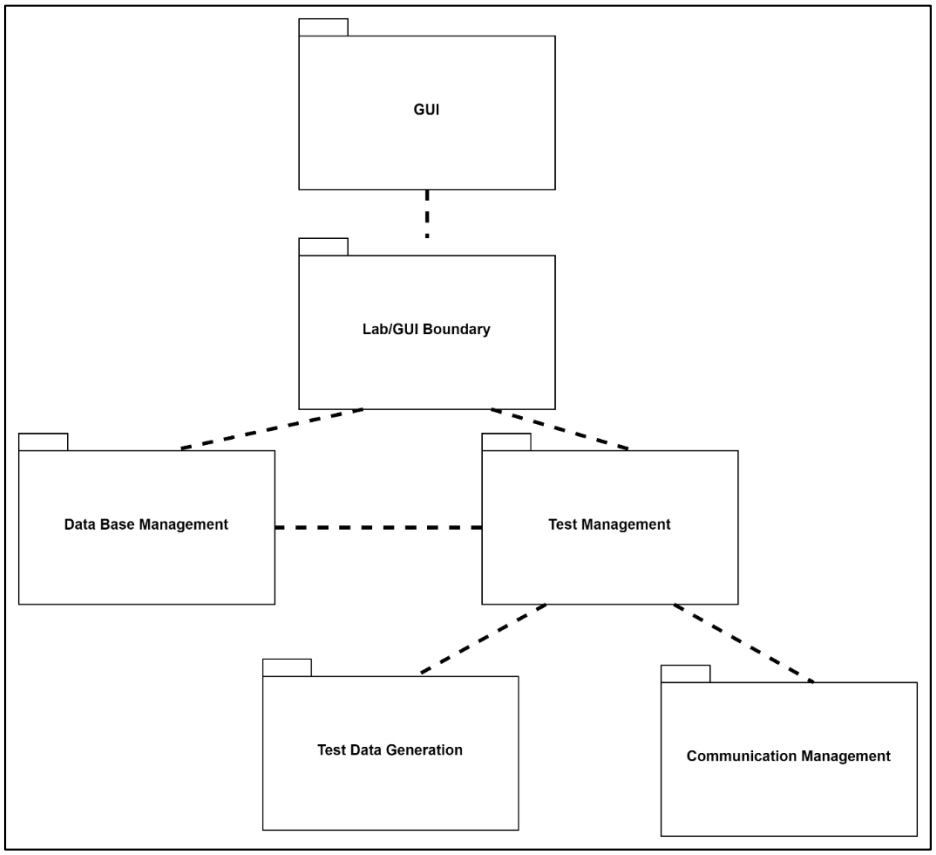
Diagram 6: System Use case diagram for the Test Station system



3.4.1.1. App Structure and Architecture

Package diagram and top-down explanation of the system, class diagram for each package with explanations. Activity for running a test.

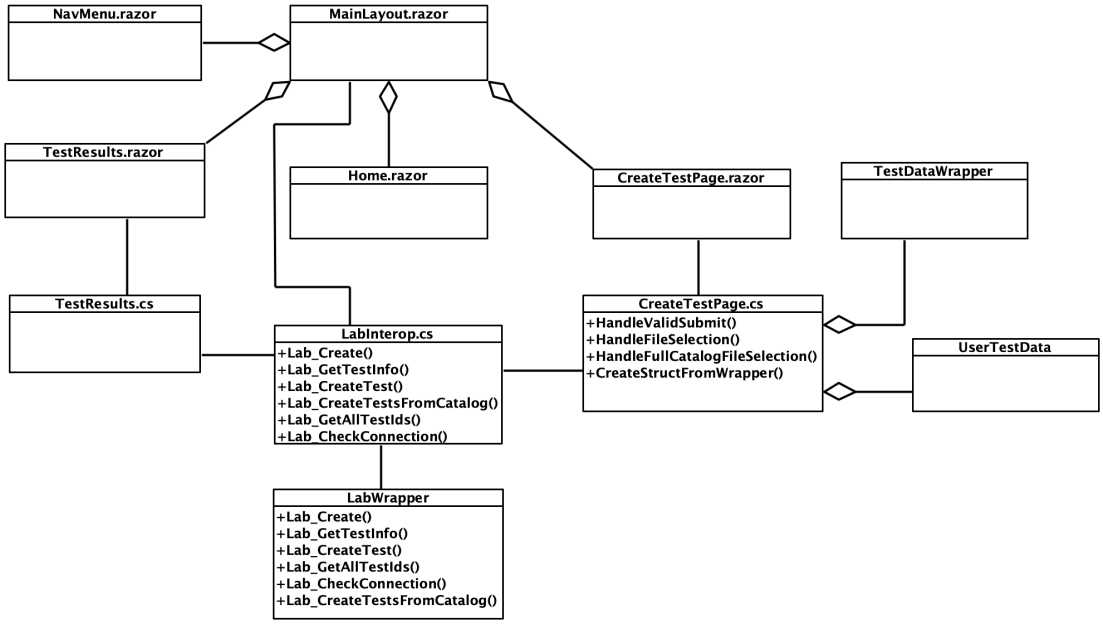
Diagram 7: The Testing Station App Package Diagram



GUI Package

The GUI package include all the application windows (or pages) and their managers. This is where every test starts, the user can go to the test creation page and create a single test or tests list from catalog file. After that he can go to the test results page and watch the test results. We have the main layout, holding all the application pages and the top bar (with connection status + links) and the navigation bar. We have 3 pages, the home page with no manager and the test creation and test results pages, each with its own manager. The test creation manager uses 2 data structs, the TestDataWrapper arrive from the GUI and the manager handle the input and place it in the internal application test data structure, the UserTestData struct. Additionally, the manager has a function for handling files as input. The test results manager displays the full tests list on start and each test information on select. All the interface with the rest of the app is done using the LabInterop static cast, setting the API against the LabWrapper (part of the Lab Package).

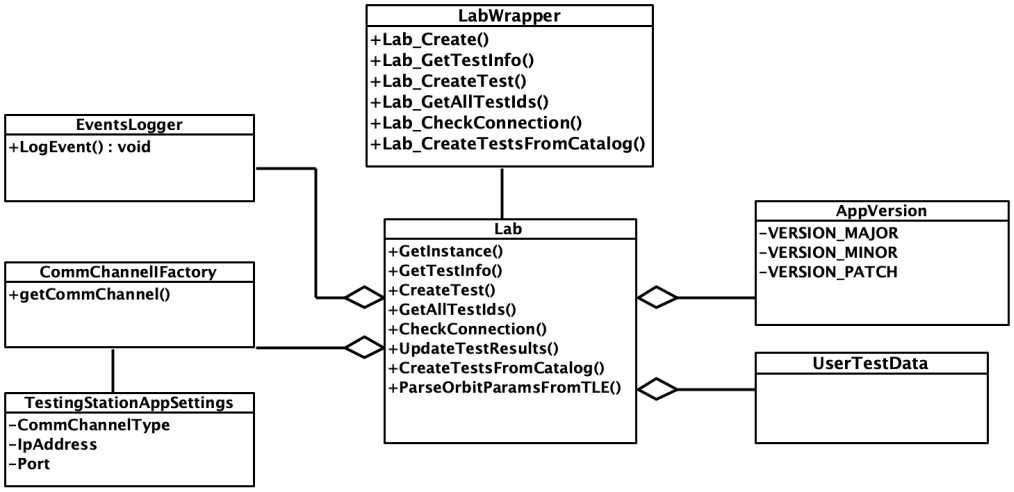
Diagram 8: GUI Package class diagram.


NavMenu.razor
MainLayout.razor
TestResults.razor
Home.razor
CreateTestPage.razor
TestDataWrapper
TestResults.cs
LabInterop.cs
+Lab_Create()
+Lab_GetTestInfo()
+Lab_CreateTest()
+Lab_CreateTestsFromCatalog()
+Lab_GetAllTestIds()
+Lab_CheckConnection()
CreateTestPage.cs
+HandleValidSubmit()
+HandleFileSelection()
+HandleFullCatalogFileSelection()
+CreateStructFromWrapper()
UserTestData
LabWrapper
+Lab_Create()
+Lab_GetTestInfo()
+Lab_CreateTest()
+Lab_GetAllTestIds()
+Lab_CheckConnection()
+Lab_CreateTestsFromCatalog()

Lab/GUI Boundary Package

The Lab Package is the main package of our application, the Lab Wrapper is the boundary between the GUI and the rest of the application, simply making the Lab class functions available withing the GUI. The Lab class is the main class of the application, managing the rest of the application initialization and use. We use the Comm Channel Factory and the Settings file to initialize the communication, initializing the database and the Logger and so on. Additionally, the Lab holds and manage the application capabilities and processes.

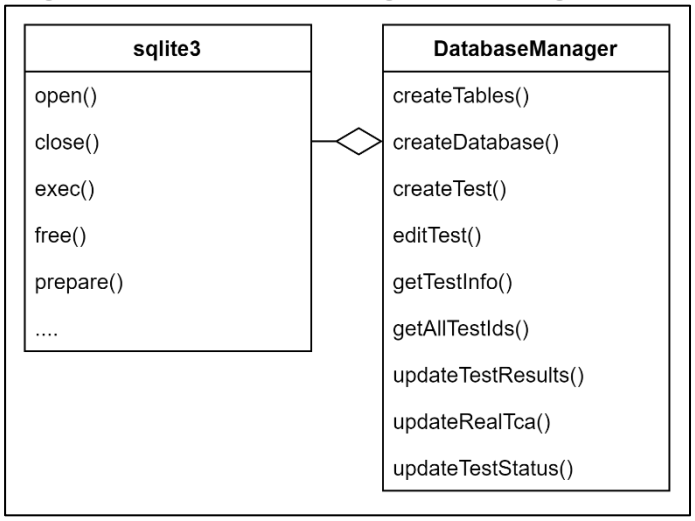
Diagram 9: Lab Package class diagram.



Database Management Package

In this Package we have two classes, the **SQLite3 [26]** implementation we used and the manager we created, wrapping the SQLite and implementing the creation and management of our tables and data.

Diagram 10: Database Management Package class diagram

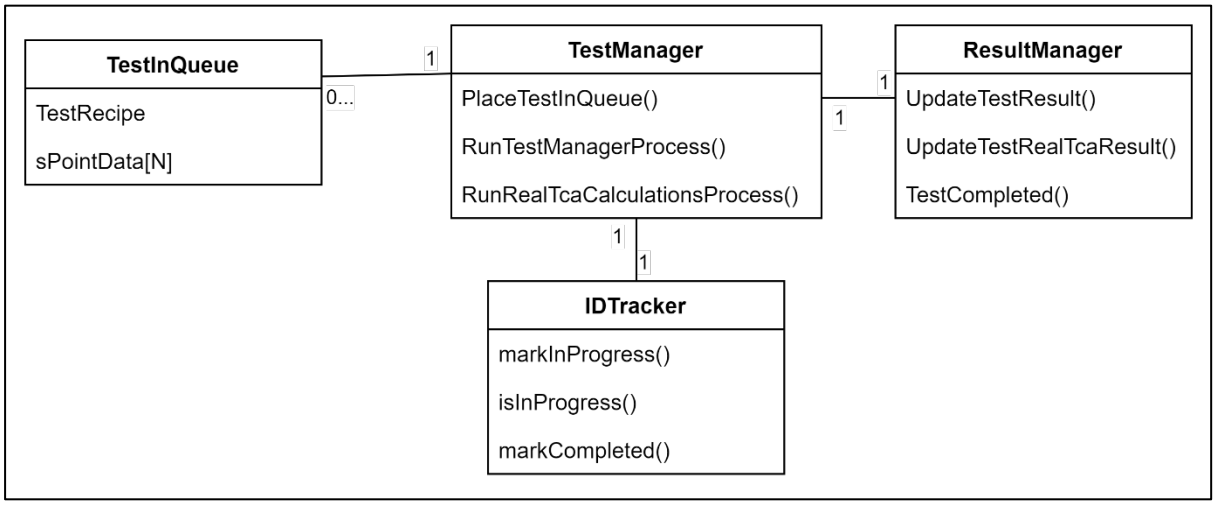


Test Management Package

The package manages the tests and results, there are 2 parts for each test, testing the algorithm and finding the real TCA. The test manager has to asynchronized threads each handling one of the tasks, the Lab call place test in queue when creating

a new test and the new test is place inside 2 queues, one for each thread. Each of the threads do its task and update the test manager with the results. Using the shared ID Tracker the thread that completed its task last update the test manager that the test was completed and free any used memory. We use mutex and a “Safe Queue” (synchronized queue) to manage access to any of our shared objects (queues, tracker and shared memory).

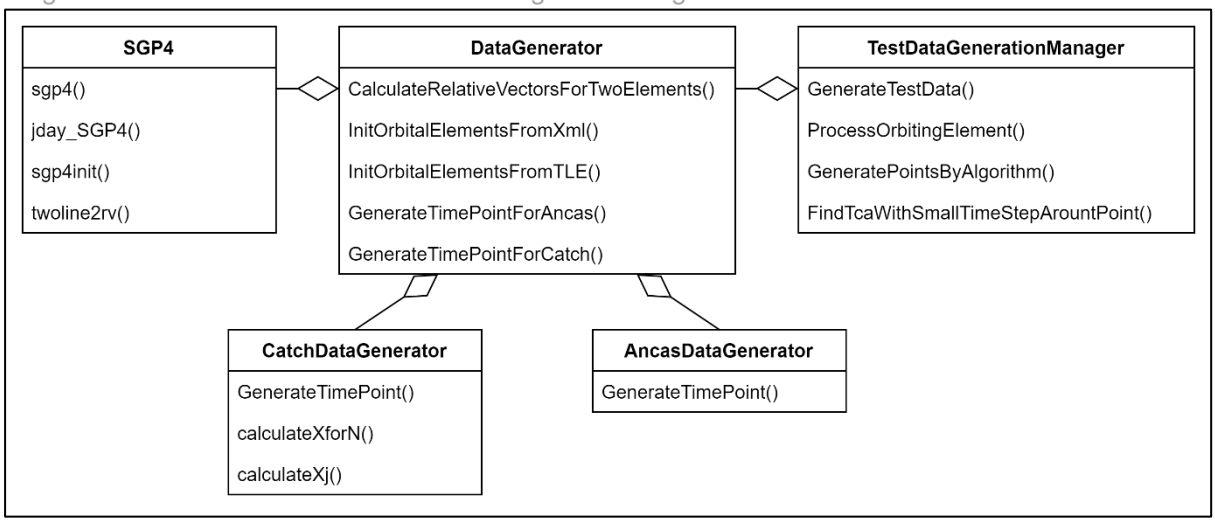
Diagram 11: Test Management Package class diagram.



Test Data Generation Package

The Test Generation package has a central class call the Test Data Generation Manager, using the Data Generator it wraps complicated or sets of action into a single function we use in the rest of our app. The Data Generator do the actual work of initializing the objects, calculating the time points and using SGP4 to generate the data.

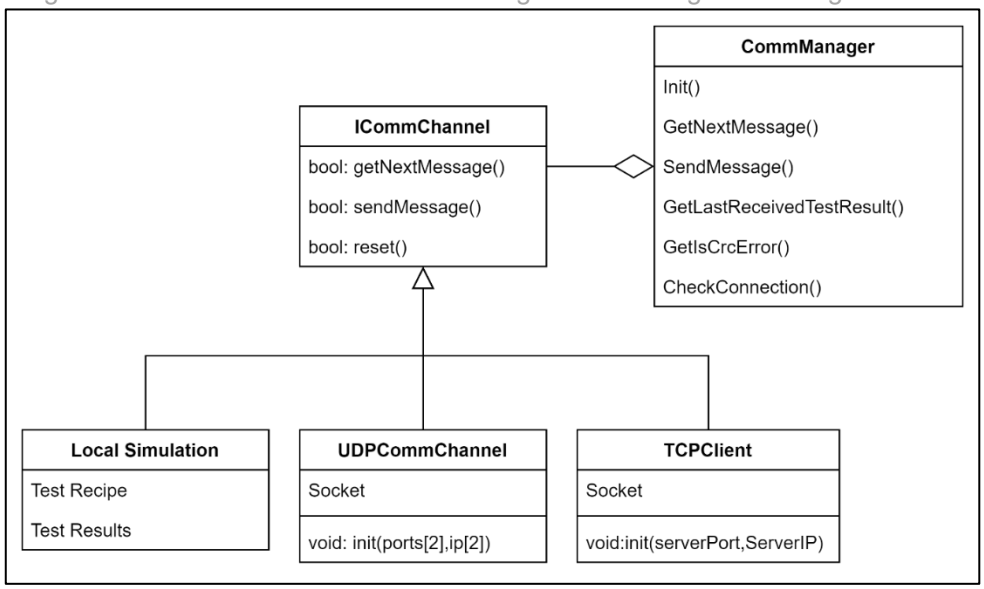
Diagram 12: Test Data Generation Package class diagram.



Communication Management Package

The package handles anything related to the communication with the Tested OBC App. Using a Comm Channel we received when initialized by the Lab, the manager sends outgoing massages and collect and parse incoming messages. We have a few implementations of a Comm Channel including the Local Simulation one.

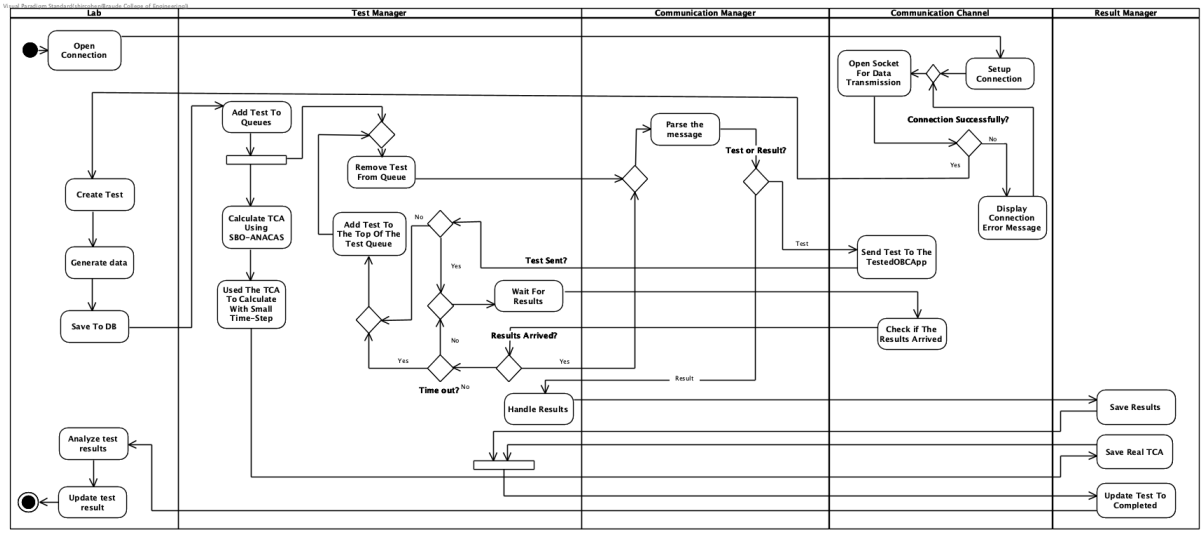
Diagram 13: The Communication Management Package class diagram.



3.4.1.1.1. Running A Test - Activity Diagram

Initially, the test station system create TCP communication over WIFI to establish how tests will be transmitted and results received. For TCP communication, the system sets up a WIFI connection, checks for successful connection, and configures sockets for data transmission. After setting up the communication,the Lab creates the test and places it into 2 queues in the Test Manager. Inside the Test manager we have 2 independent thread each with his own task, and each with his own incoming tests queue. The first get the test from the queue and send it to the Tested OBC App, waits for the response and update the Results Manager with the test results. The second get the test, calculate the real TCA and update it using the Results Manager. Only after both of them finished with the test we can update its state to Completed and display the results.

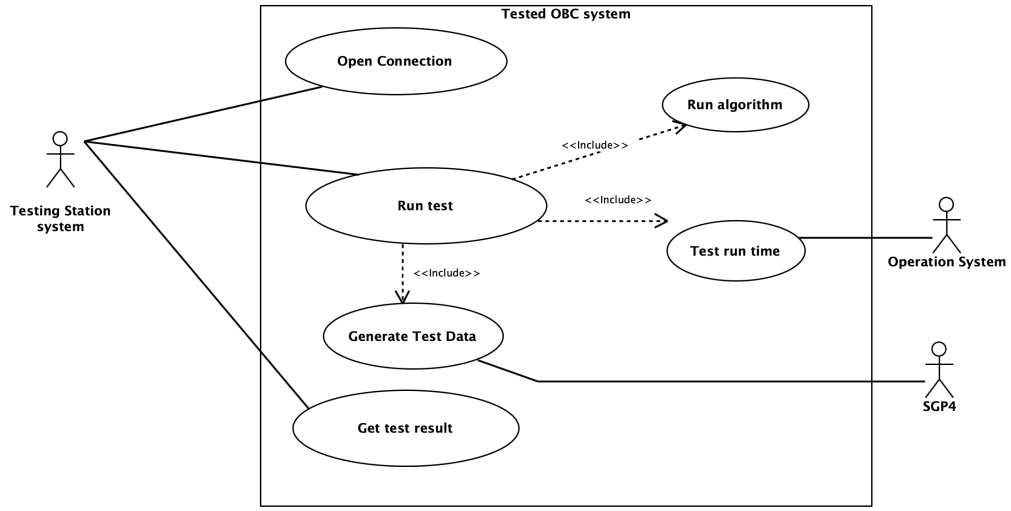
Diagram 14: Test Manager Process activity diagram.



3.4.2. Tested OBC system

On the other side, the software running on the **Raspberry Pi 5 [19]** continuously listens to the communication channel for incoming messages. Once it receives a command from the Test station following by a test data, the system starts a timer, generate data points using the specified algorithm, executes the test with the selected algorithm and the computed data set, stops the timer, and calculates the run time. Upon completion, the results, including the runtime, are sent back to the Testing Station for analysis.

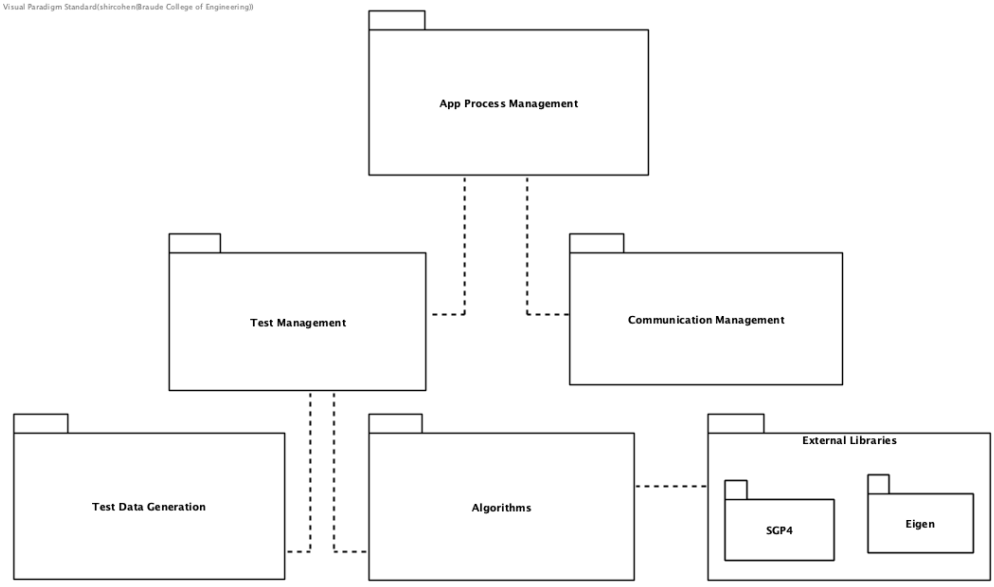
Diagram 15: System Use case diagram for the Tested OBC system



3.4.2.1. App Structure and Architecture

The Tested OBC App works around messages from the Testing Station App, we wait for an incoming message, get the Test Recipe and Test Data from the message, run the algorithm and return the results and run time. When not running a test the Tested OBC App waits for the next message.

Diagram 16: The Tested OBC App Package Diagram

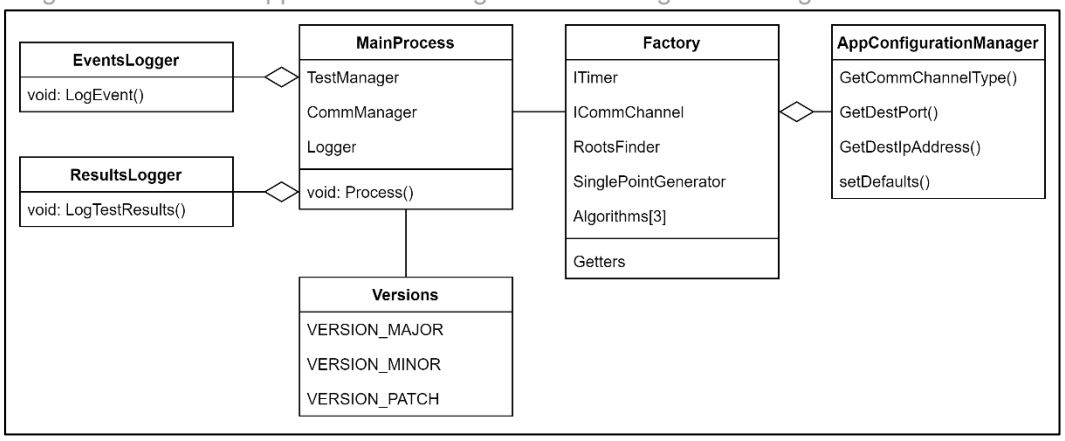


App Process Managements Package

The App process management package handles the main app process, calling the Communication Management and checking for incoming messages, starting tests

using the Test Managements package and handling anything else related to the App creation and process.

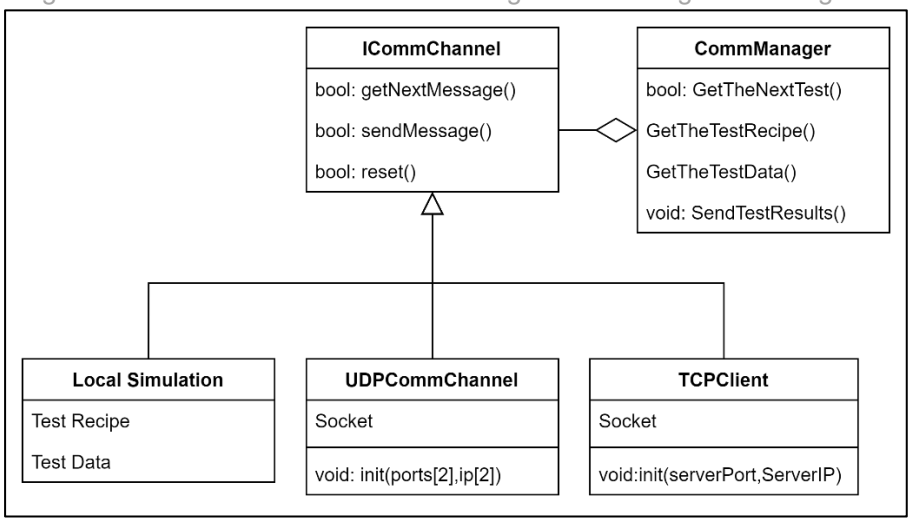
Diagram 17: The App Process Managements Package class diagram.



Communication Management Package

The Communication Management package handles anything related to the communication with the Testing Station. The Factory create the Comm Channel based on the App Configuration and the Comm Manager handle incoming messages, parsing the messages and checking for errors. Additionally, the Comm Manager send the outgoing Results Message based on results set it receive.

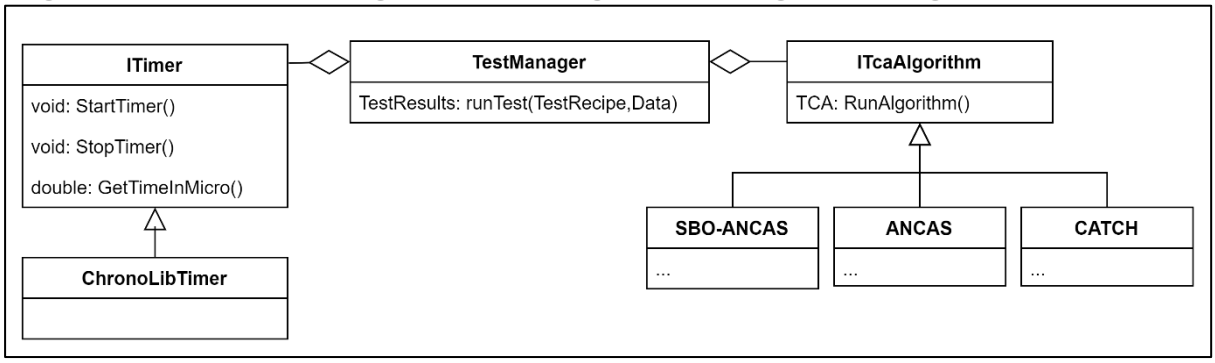
Diagram 18: The Communication Management Package class diagram.



Test Management and the Algorithms Packages

The Test Management package handles the test itself, after receiving a Test Recipe, it uses the **SGP4 [24]** propagator to generate a set of points, which are then stored alongside the Test Recipe in the Test Data, the Test Manager uses the Factory to get the required algorithms object, initialize to the correct degree or with the correct roots finding algorithm based on the Recipe. After receiving the Algorithm, the Test Manager start the timer and run the algorithm. After the algorithm completed the call the Test Manager stop the timer, collect the run time, algorithm output and the test data into the Test Results Set. Additionally, if the test should run over a few additional iterations the Test Manager run the algorithm again and return the Test Results at the end. The Algorithms package contains the Algorithms implementations and variations.

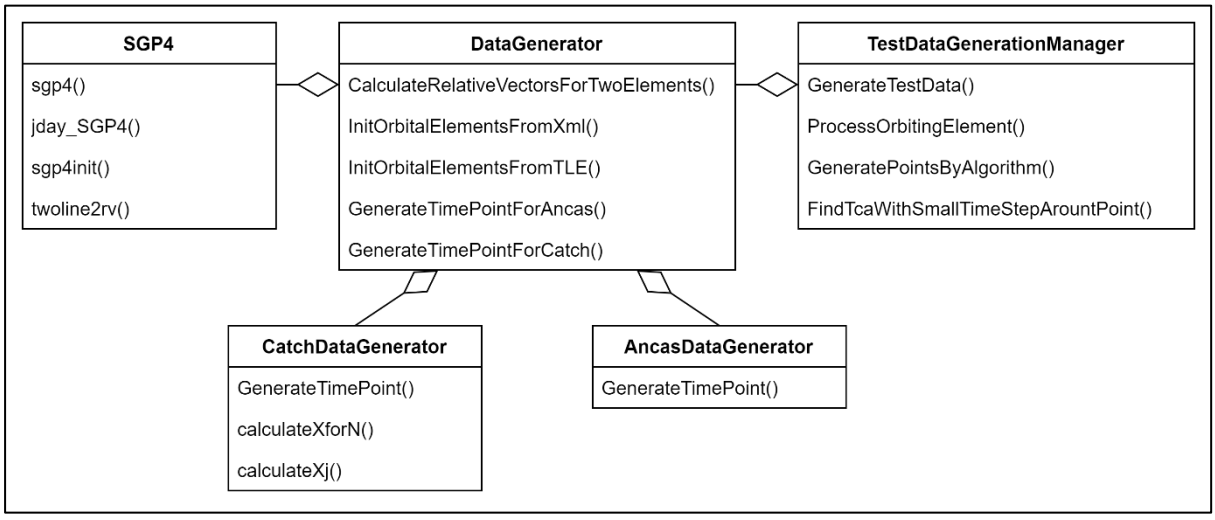
Diagram 19: The Test Management and the Algorithms Packages class diagram.



Test Data Generation Package

The Test Generation package has a central class call the Test Data Generation Manager, using the Data Generator it wraps complicated or sets of action into a single function we use in the rest of our app. The Data Generator do the actual work of initializing the objects, calculating the time points and using **SGP4[24]** to generate the data.

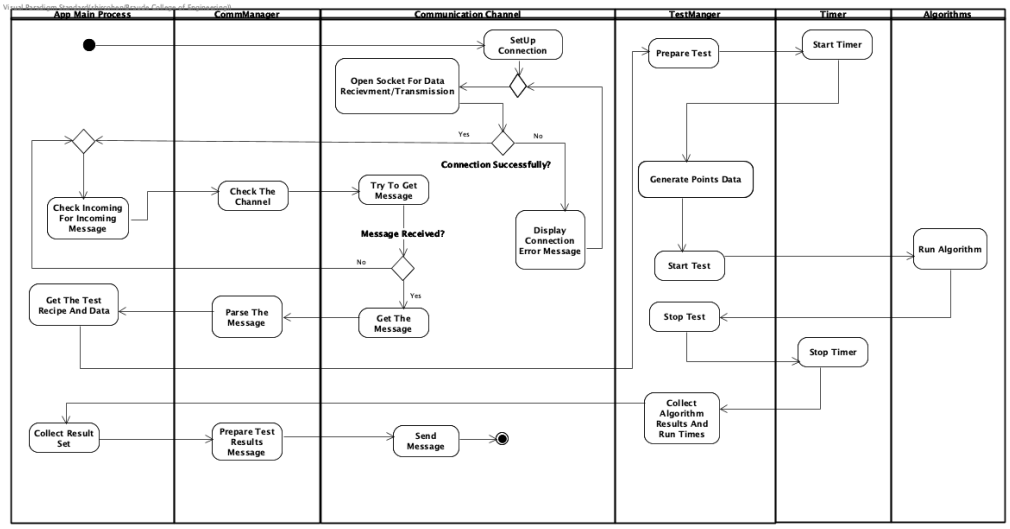
Diagram 20: Test Data Generation Package class diagram.



3.4.2.2. Running A Test - Activity Diagram

The Tested OBC App continuously checks for incoming messages. Once a message arrives, it parses the message and retrieves the Test Recipe. It then starts a timer, generates a set of points using the SGP4 **[24]** propagator, prepares and executes the test, stops the timer, and creates a results set that includes the run time. Finally, it sends the results back to the Testing Station App.

Diagram 21: Activity diagram of running a test on the Tested OBC System via Ethernet Communication Method and Protocol.



3.4.3. Full system

Looking at the full system, the process of creating and running a test starts from the user after openning communication. Then the user goes to the Test Creation page and create a test, filling all the necessary fields, after that the GUI managers collect the input and call the Lab to create a test, the Lab generate the data, save the test and give it to the test manager who forward it to the Tested OBC via the communication channels. The Tested OBC take the input,generate the data, run the algorithm and send back the results who go all the way back to the tests results page and displayed to the user.

Diagram 22: Top view Use Case diagram for running a test, with the full 2 systems

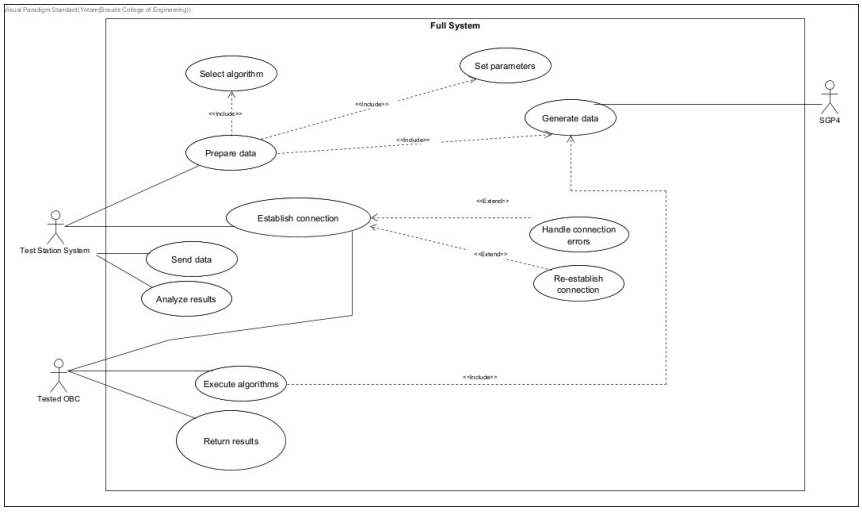
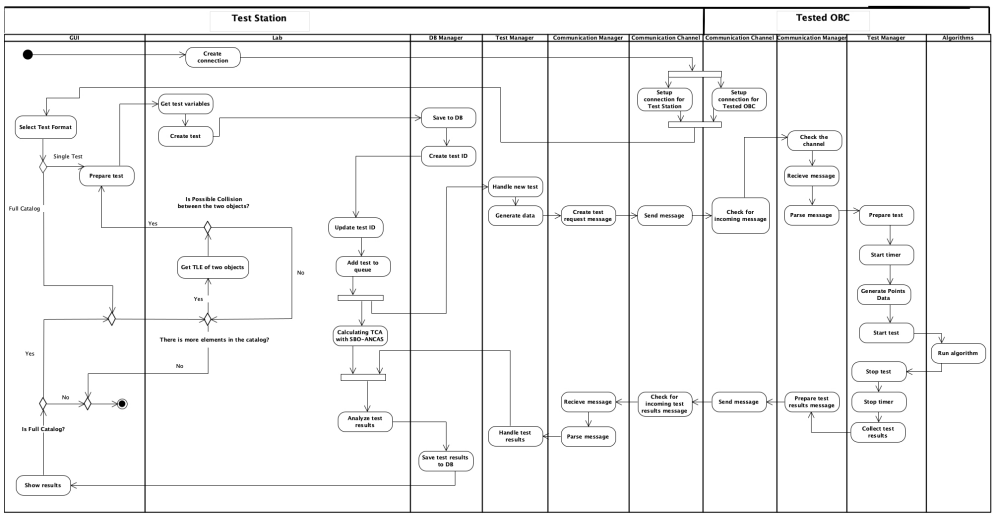


Diagram 23: Top view activity diagram for running a test, with the full 2 Apps system



3.4.4. Communication Protocol and Channels

3.4.4.1. The Communication Protocol

We created a simple protocol for the communication between the Testing Station App and the Tested OBC App. The protocol contains 2 messages, one from the Testing Station App to the Tested OBC App and the other one back. The application work in a Master-Slave like manner, the Tested OBC App never start the communication, only waits for a test request message and answering with the test results message after completing the test. The Testing Station App send one test at a time and wait for the reply up to a given Timeout.

Testing Station to Tested OBC – Test Request Message

Message from the Test Station to the Tested OBC:

This message contains the test data necessary for the algorithm to run on the Tested OBC. It includes:

- The **test recipe**: specifying which algorithm to run (CATCH **[5]** , ANCAS **[1]**, or SBO-ANCAS **[6]**), and any additional algorithm-specific parameters (e.g., the polynomial degree, tolerance levels, etc.).
- The **test data**: consisting of an array of time points, position vectors, and velocity vectors for two objects in orbit.

Once the Tested OBC receives this message, it proceeds to execute the appropriate algorithm based on the provided data.

Table 2: Test Request Message Description.

Bytes	Type	Field Name	Field Description	Expected Values
0-1	Unsigned Short	Opcode	Unique identifier for the message, used for sync and opcode, identifying the message start and type.	0x1234
2-5	Unsigned Int	Data Length	The size of the test data, can be different in every message.	$[0, 2^{32} - 1]$

6-9	Unsigned Int	CRC	4 Bytes CRC, to identify errors in the message without relying on the communication type.	Calculated on the message
10-209	Struct	Test Recipe	Struct containing all the required options for running the test, including the polynomial degree, number of points, tested algorithm and more.	can vary
210-N	Array	Points Data	The input for SGP4, array of time points, where for each time point, the propagator generates velocity and position vectors that will serve as input for the algorithms.	can vary

Tested OBC to Testing Station – Test Results Message

Message from the Tested OBC to the Test Station:

After executing the algorithm, the Tested OBC sends a message back to the Test Station. This message contains:

- The **calculated results**: including the TCA and the minimum distance between the two objects.
- **Performance metrics**: such as the total runtime for the algorithm, memory usage, and detailed breakdowns of the time taken for individual operations (e.g., polynomial fitting, root-finding, etc.).

Table 3: Test Results Message Description.

Bytes	Type	Field Name	Field Description	Expected Values
0-1	Unsigned Short	Opcode	Unique identifier for the message, used for sync and opcode, identifying the message start and type.	0x4321
2-5	Unsigned Int	Data Length	The size of the test data, a constant size.	188
6-9	Unsigned Int	CRC	4 Bytes CRC, to identify errors in the message without relying on the communication type.	Calculated on the message
10-197	Struct	Test Results	Struct containing all the test results data, the found TCA and distance, the run time, average and minimal and more.	188

3.4.4.2. UDP Communication Channel

The first implementation we did was UDP, the easiest to implement and use. But unfortunately, UDP have few major flaws. The first problem is a limit on the message size, in the IP layer we have a total length field in an unsigned short variable, limiting the total size of each IP packet to around 65,500 bytes of data (after subtracting the headers size) so we need to send our message in blocks ourself, and here we get to the second problem, reliability. The protocol doesn't assure as we get the blocks in the order we sent them or that we will get them at all, meaning that we will have to track the blocks arrival order ourself, send ACK of some kind and resend it if necessary. In each message we can have a lot of data, for example for a test of time period of a week we can easily get 6000 points, each point contains the time value and 4 vectors,

location and velocity of two objects, meaning we have 13 double precision variables, each of them is 8 Bytes, the data array will be $6000 * 13 * 8 > 600KB$ making the risk of losing parts of the message much higher.

3.4.4.3. TCP Communication Channel

Unlike the UDP protocol, TCP is a much better option for our needs. The protocol handles the full message, sending it fragmented if necessary, collecting and making sure we can the full message in the correct order. The cost is in run time but we only care about the run time when the algorithm is running, in other times it doesn't really matter. We used the protocol as a client and server duo, the Testing Station being the server, running on a PC and with resources to spare. Additionally, we only need to know the IP address and port of the Testing Station. The Tested OBC connect to the station as a client, and waiting for incoming test request + answering with the results.

3.4.5. Feasibility Testing Environments

We designed the Tested OBC App to be adaptable across multiple computing environments, ensuring flexibility in testing and deployment. While our primary goal is to run the application on the selected OBC (**Raspberry Pi 5 [19]**), we also tested its functionality on three different systems to verify its compatibility and performance across various platforms. This approach allows us to evaluate the software under different operating conditions, ensuring robustness and scalability while maintaining consistency in algorithm execution and data handling.

3.4.5.1. Linux on Raspberry Pi 5 (The Selected OBC)

Using the **Raspberry Pi 5 [19]**, we conducted tests in a resource-constrained environment to evaluate the feasibility of running our collision detection algorithms under realistic conditions. Our focus was on validating cross-platform TCP communication, analyzing algorithm performance with limited CPU and memory, and ensuring real-time processing capabilities on an ARM-based system. Additionally, we tested the **SGP4 [24]** propagator on the **Raspberry Pi 5 [19]** to verify its ability to compute position and velocity vectors accurately. Running tests on this platform ensures compatibility with future satellite-grade OBCs, providing a robust evaluation framework for algorithm optimization and deployment.

Image 7: Our Raspberry Pi 5 [19] system specs (Using lscpu command)

```
Architecture:            aarch64
  CPU op-mode(s):        32-bit, 64-bit
  Byte Order:            Little Endian
CPU(s):                  4
  On-line CPU(s) list:   0-3
Vendor ID:               ARM
  Model name:            Cortex-A76
    Model:               1
    Thread(s) per core:  1
    Core(s) per cluster: 4
    Socket(s):           -
    Cluster(s):          1
    Stepping:            r4p1
    CPU(s) scaling MHz:  62%
    CPU max MHz:         2400.0000
    CPU min MHz:         1500.0000
    BogoMIPS:            108.00
    Flags:               fp asimd evtstrm aes pmull sha1 sha2 crc32 atomics fphp asimdhp cpuid asimdrdm lrcpc dcpop asi
                         mddp
```

3.4.5.2. Windows/MacOS

The easiest option is to run our application on the same computer as the Testing Station App, either using TCP over localhost (when both systems operate on the same machine) or using two separate computers (one running macOS and the other Windows) connected via Wi-Fi. This setup is particularly convenient for real-time testing, debugging, or for scenarios that do not require the actual embedded

environment (for example, exploring the relationship between different root-finding algorithms and their error magnitudes).

We also implemented a Local Simulation mode. In this mode, only the Tested OBC App is executed, while a local simulator generates a Test Request Message and simulates receiving it through the communication channel. This approach is especially useful for preparing and evaluating test cases without relying on physical communication or synchronization with another application. It also enables running tests asynchronously in a self-contained environment.

4. Research and Development process

4.1. Algorithms Analysis and Implementation

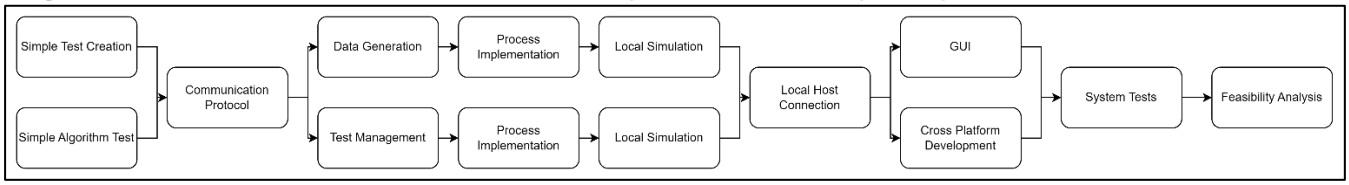
We conducted a comprehensive analysis and implementation of the ANCAS, CATCH, and SBO-ANCAS algorithms, evaluating their computational complexity and runtime performance. Each algorithm was implemented and tested rigorously to ensure accuracy, efficiency, and adaptability in a constrained computing environment. The final validation was performed on the selected **Raspberry Pi 5 [19]**, where we analyzed execution time, memory consumption, and overall feasibility, ensuring the algorithms' suitability for real-world deployment.

4.2. The Development Process

For our development process we worked in an Agile like process, we started developing our system with a simple API and classes, adding more features each iteration. We met every week for a short sprint where we divided the tasks and discussed the project. The number of tasks we decided on each week varied greatly depending on the availability of each of us.

We started with developing the app separately, each app with a local simulation, a simple implementation of a fake communication channel that created the needed messages we needed, used to debug and test ourself in each step.

Diagram 24: Flowchart of milestones and notable points in the development process.



4.3. Unit Testing and Debugging

We created and used the unit tests for testing different logic processes in our code, debugging a specific case if needed and making sure we didn’t make any new bugs after every change. Additionally, we used the local simulation and loggers we created to test more complex, full system cases and scenarios, like running a full test,running full catalog or faulty inputs.

Image 8: Example of some of the unit tests, testing the Tested OBC App Comm Manager parser with different types of possible incoming messages (fragmented, big message in parts).

```
[ RUN      ] TestCommManagerParser.TEST_test_SimpleFullMessageInBuffer
[       OK ] TestCommManagerParser.TEST_test_SimpleFullMessageInBuffer (0 ms)
[ RUN      ] TestCommManagerParser.TEST_test_FragmentedMessage
[       OK ] TestCommManagerParser.TEST_test_FragmentedMessage (0 ms)
[ RUN      ] TestCommManagerParser.TEST_test_BigFragmentedMessage
[       OK ] TestCommManagerParser.TEST_test_BigFragmentedMessage (10 ms)
[ RUN      ] TestCommManagerParser.TEST_test_SimpleFullMessageInMiddleOfBuffer
[       OK ] TestCommManagerParser.TEST_test_SimpleFullMessageInMiddleOfBuffer (0 ms)
[ RUN      ] TestCommManagerParser.TEST_test_FragmentedMessageOnHeader
[       OK ] TestCommManagerParser.TEST_test_FragmentedMessageOnHeader (3 ms)
```

5. Development tools

5.1. Development environment

For software development, we used Visual Studio 2022, which supported both C++ development for our core system and C# development for the Blazor framework used in the Testing Station App. In addition, we utilized CLion as an integrated development environment (IDE) for C++ development, particularly for working with the **Raspberry Pi 5 [19]**.

One of the key advantages of using CLion was its Remote Development capabilities. By leveraging SSH connections, we were able to develop, compile, and debug the code on our local machines while executing it directly on the **Raspberry Pi 5 [19]**. This setup provided a seamless development workflow, allowing us to efficiently test and refine our implementation in the target embedded environment, while maintaining the convenience of working from our development machines.

5.2. Languages

Earlier on we decided to develop our system with C++. The core of our system, the main purpose is the algorithm and we needed to implement them efficiently and have a well-structured with OOP implementation. We also needed to create an app that can work well on a limited system, the Tested OBC App that needed to work well on a relatively weak computer so the decision to create the Tested OBC App with C++ was quite easy. We chose to also develop the Testing Station App with C++, because it's easier to manage communication and common structures and it give as faster calculation when needed (creating the input data, calculating the real TCA with a small time-step). To create the GUI for our project we used Blazor because this framework is modern and easy to work with. Also, we already had some experience with this framework, and integrating Blazor with the rest of our C++ code was relatively simple.

5.3. External libraries

5.3.1. Eigen

We used the **Eigen library [28]** for finding the eigenvalues of CATCH's companion matrix, part of the root finding algorithm **CATCH [5]** uses for finding the root of a high degree polynomial. The library provides easy to use and relatively fast solutions for many algebraic problems. Additionally, Eigen is a header only library meaning easy to use on different platform and compiled with our project and any optimization we included. The library doesn't require a lot of memory like other options we checked so it a good option for the low memory system we need to use. The library can be found here **[28]**.

5.3.2. INIH – INI Files Reader Library

We used this library for reading INI files easily, used for a setting or configuration files in both the Testing Station App and the Tested OBC App. Can be found here **[11]**.

5.3.3. SQLITE3

We used SQLITE3 implementation for our local database, can be found here **[26].** because it's quite old implementation it required some specific configuration to work properly. The problem was that it couldn't be compiled while supporting CLR, so we disabled it for the relevant files only.

5.3.4. SGP4

For our propagator we used SGP4 C implementation, can be found here **[24]**. We use it to generate test data and SBO-ANCAS's additional points. The propagator also used to initialize the objects necessary from the TLE input.

5.4. Additional Tools

5.4.1. Git

We used Git for source control, creating a repository on GitHub [18] and using additional applications with easier user interface like Sourcetree and TortoiseGit.

5.4.2. GTest and GMock

We used GTest and GMock for our unit tests, GTest and GMock **[10]**. are frameworks for unit tests in C++, providing the tools required for creating unit tests, tests cases and mocks easily. It's easy to use and debug and there was a lot of information and tips for it online.

5.4.3. CMake

Because we needed to have easy option for cross – platform compiling, and to leave the option to compile easily for whatever OBC needed, we used **CMake [20]** for the Tested OBC App. We created a CMakeLists file that we can use to compile the application with whatever compiler or configurations we might need. We used it to compile our application for our **Raspberry Pi 5 [19]**.

5.4.4. Semantic Versioning

We used the guidelines found in here **[23]**. for versioning our applications.

5.4.5. Coding Conventions

We tried to based our coding on the convention found here **[2]**, mostly for variable and functions naming.

6. Problems and solutions

6.1. Cross Platform Communication

One of the problems we expected we might come across, and unfortunately, we did is communicating with common objects over different platforms. Each compiler can use different sized for the basic types, for example we found out that the "long" type was 8 bytes signed integer in one system and 4 bytes in another. This can obviously be a problem when trying to parse incoming data from a different system. So, in order to solve the problem, we used the types from the cstdint library, giving us constant types sizes over any platform for all the integer types. The floating-point types, float and double already use the IEEE 754 standard that defined their sizes and behaver consistently over different platform (not true for the "long double" type which we opted not to use).

Another problem we encounter is padding. Compilers might add padding to any data structure in order to align it in memory or as a part of some optimization for memory access. For example, rounding a structure size to the nearest multiple of 8 bytes.

When trying to parse a structure arriving from a different platform, we must have consistence structures and structure sizes. To solve the inconsistent padding, we used the “pragma pack” directive to specify a consistent padding for different possible compilers.
The last problem we had is implementing a common code, classes or functions, for different platforms. Some libraries or functions can vary between different platforms and to ensure our code worked universally, we sometimes created platform-specific implementation and used the “#ifdef” directive with preprocessor defines to compile the required variation each time.

6.2. Communication Error Detection

Because our code is not limited to one type of communication, and not all communication protocol created equally, some might have size limitation forcing us to send messages in blocks, other might have limited error detection or won’t make sure packages arrive to their destination (like UDP). After facing some communication problems, duo to different types sizes and padding, and having difficulty identifying these issues because we received the messages exactly as sent, but parsed them incorrectly, we decided to add our own error detection to messages. A simple checksum could be problematic because if messages in blocks arrived out of order, the checksum won’t find the error. Therefore, we decided to go for a more reliable solution by using a 4-Byte CRC.

6.3. Debugging Different Asynchronized Apps

Another problem we faced while developing is debugging two separate applications, working independently and sometimes relying on the other side to continue in some logic process. We decided from the start to implement a Local Simulation, simulating for each application the other one, but still part of the application and can easily be used locally without and dependencies on the other application. The Local Simulations proved valuable when trying to find problems or check complicated scenarios. But even the Local Simulations had their limits. When trying to find problems in the actual communication and interaction between the applications. Using the traditional debugging tools like breakpoints could be problematic because some problems occur due to timing or synchronization problems. To help as find problems and bugs we implemented Event Logger for both of our applications. The logger saves different events with time stamps and can be used to compare events, detect unwonted event, errors and more.

6.4. Cross Language Development

Our GUI framework choice was Blazor which uses C#, but our system was built in C++ language only, to create a bridge between those two languages, we had to use P/Invoke. Using cross platforms added challenges to the way we configured the project's solution, when debugging because when we debugged the C# project the debugger didn't go into the C++ code. We had to be careful how we build the common structures in C# and C++ and how we moved certain types of data without getting memory violations.

6.5. SGP4 in C++ Had Only Partial Implementation

When we initially started working with **SGP4 [24]** propagator from python just for data gathering, we had the option to get the satellites data from OMM (Orbiting Mean-Elements Message) XML files. Later on we needed to integrate SGP4 as part of our system, but parsing the XML files into usable data was missing, and we needed to create our own implementation to parse those files. After implementing, we were

concerned about the results, because we got different property's values from XML files and TLEs which supposed to give the same data. After investigation we discovered that this is not our code's fault, but in TLEs due to the format nature, the values are rounded, therefore we can expect a small difference in the values.

6.6. Optimizing Resource Allocation for Accurate Algorithm Execution

One of our key objectives was to ensure that the execution of collision detection algorithms on the **Raspberry Pi 5 [19]** was performed under optimal conditions, allowing us to obtain the most precise and consistent results across multiple test runs. Since the **Raspberry Pi 5 [19]** is a multi-core system, background processes and system tasks could interfere with the algorithm execution by consuming CPU resources, leading to variations in execution time and performance measurements.

To address this, we allocated a dedicated CPU core exclusively for the Tested OBC system. By configuring the system to pin the Tested OBC process to a specific core, we ensured that all other user processes and background tasks were restricted from running on that core. This setup allowed us to isolate the execution environment, ensuring that the full processing power of the designated core was available for running the algorithms. As a result, we achieved consistent performance measurements, minimizing external system influences and improving the reliability and accuracy of our test results.

7. Results and conclusion

7.1. Feasibility Analysis and Test Results

We used the same data set for all test cases while varying the numbers of points per segment and the number of segments per minimal revolution.

The Data Set:

We used a catalog of 11,000 objects and obtained the TLE data for all active satellites from Celestrak [4] as of 27/02/25 14:45:33 UTC. The algorithms were tested on pairs, where the first satellite in the list was tested against every other satellite. After filtering the catalog by removing satellites that had no chance of colliding with the first satellite (with collision threshold of 10 KM), we obtained 2,487 different test cases. We tested with different numbers of points per segment (8 or 16 points) and different number of segments per minimal revolution (8 or 16 segments), with **CATCH [5]** degree set to 15 or 7 according to the selected number of points and time interval set to a week (2419200 Seconds), with the following SBO-ANCAS tolerances:
TOLd = 0.00000001, TOLt = 0.0001.
The values of the numbers of points per segment (8 or 16 points) and different number of segments per minimal revolution control how many data points the algorithms must process.

7.1.1. Run Time Comparison

To compare the runtimes, we ran four variations over the data set and checked the results.

A Test Case 1:

Here we have an example of a single run of the 3 algorithms over the same set of data:

- numbers of points per segment (Number of Input Points): 8
- number of segments per minimal revolution (Tmin Factor): 8
- CATCH degree:7

- SBO-ANCAS tolerances: TOLd = 0.00000001, TOLt = 0.0001

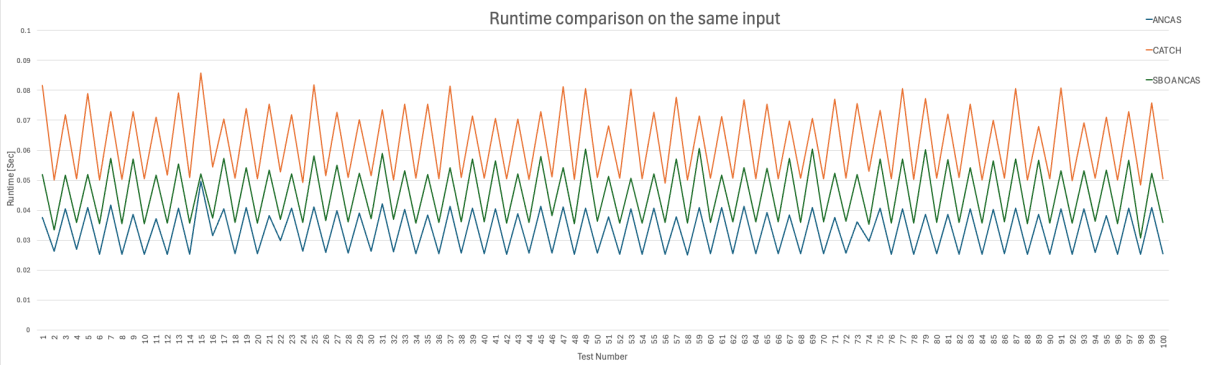
Table 4: Test data for the STARLINK-30375 satellite and COSMO-SKYMED 3 satellite in the catalog.

Test ID	Algorithm Name	Run Time[Sec]	TCA Distance[Km]	TCA[Sec]	Real TCA Distance[Km]	Real TCA[Sec]	Error Time[Sec]	Error Distance[Km]
14	ANCAS	0.025244	9.31388256	922993.6313	9.313926564	922993.633	0.001648891	4.40041E-05
14	SBO ANCAS	0.035631	9.313926563	922993.633	9.313926564	922993.633	8.05419E-06	8.27539E-10
14	CATCH	0.050847	9.313926218	922993.633	9.313926564	922993.633	5.78351E-06	3.46059E-07

Analyzing The Data:

From the runtime measurements in Test Case 1, where each algorithm received 8 points per segment, 8 segments per minimal revolution (Tmin Factor = 8), and CATCH used degree = 7, we observe that CATCH consistently takes the longest time, while SBO-ANCAS runs faster than CATCH but is still slower than ANCAS. Interestingly, SBO-ANCAS does not exhibit the highest runtime despite its additional iterative steps for sampling new points, indicating that the overhead from repeatedly refining time points is still less expensive than the companion matrix eigenvalue calculations required by CATCH. As expected, ANCAS demonstrates the lowest runtime, owing to its straightforward cubic polynomial approach with minimal overhead.

Graph 1: Algorithms Runtime [Sec] per test, only on the first 100 tests for Tmin=8 and 8 points per segment.



These results confirm that CATCH's higher complexity is largely due to its root finding approach, which requires eigenvalue computations for a 7th degree polynomial. In contrast, SBO-ANCAS uses a loop based refinement process that, although more computationally expensive than ANCAS, remains comparatively efficient. ANCAS, with its simple cubic polynomial fitting method, consistently achieves very short run times, although this simplicity may come at the expense of accuracy in cases where the relative distance between objects changes rapidly. In Test Case 1, ANCAS proves to be the fastest algorithm, completing calculations quickly due to its streamlined approach, however, it may miss critical roots or yield less precise results when the data changes abruptly. SBO-ANCAS strikes a balanced middle ground between speed and accuracy by using iterative refinement to improve precision, resulting in a lower runtime than CATCH. CATCH consistently shows the highest computation time, mainly because its companion matrix method for polynomial root finding requires

intensive eigenvalue calculations, which significantly increase the runtime even though they can provide robust accuracy.

A Test Case 2:

Here we have an example of a single run of the 3 algorithms over the same set of data:

- numbers of points per segment (Number of Input Points): 8
- number of segments per minimal revolution (Tmin Factor): 16
- CATCH degree:7
- SBO-ANCAS tolerances: TOLd = 0.00000001, TOLt = 0.0001

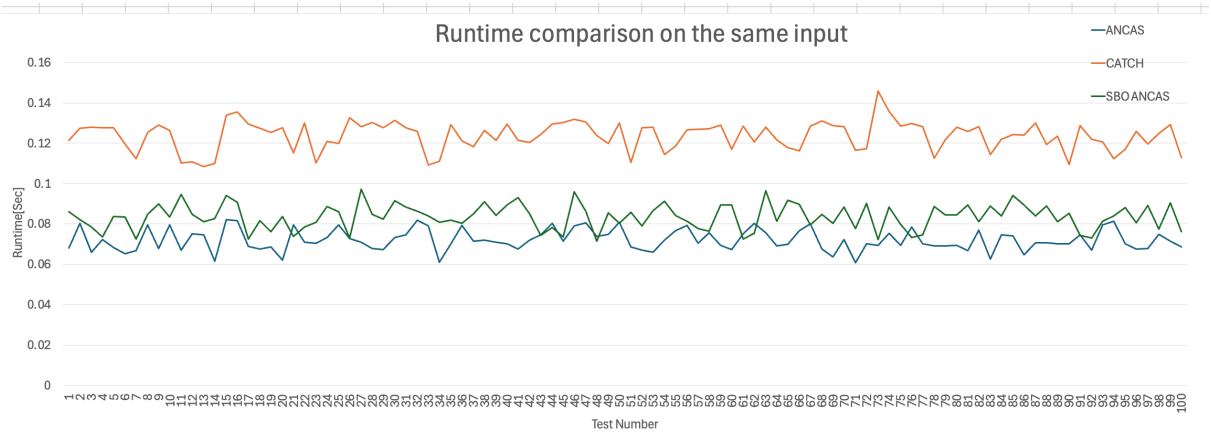
Table 5: Test data for the STARLINK-30375 satellite and COSMO-SKYMED 3 satellite in the catalog.

Test ID	Algorithm Name	Run Time[Sec]	TCA Distance[Km]	TCA[Sec]	Real TCA Distance[Km]	Real TCA[Sec]	Error Time [Sec]	Error Distance [Km]
14	ANCAS	0.061559999 9999999	9.31393067 045039	922993.632 954832	9.31392656 375749	922993.632 981267	0.00002643547 48651385	4.106692895 07836E-06
14	SBO ANCAS	0.082564999 9999999	9.31392656 28991	922993.632 979321	9.31392656 375749	922993.632 981267	1.94599851965 904E-06	4.106692895 07836E-06
14	CATCH	0.110103	9.31392656 496836	922993.632 979541	9.31392656 373572	922993.632 971266	8.27549956738 949E-06	1.232633550 78507E-09

Analyzing The Data:

The runtime measurements for Test Case 2 clearly reveal clear performance differences between the algorithms. CATCH again exhibits the highest runtime, depending on the eigenvalue calculations of its associated matrix. In contrast, SBO-ANCAS, although it requires additional inner loops for iterative refinement, still runs faster than CATCH. This suggests that solving a cubic polynomial multiple times is less expensive than computing eigenvalues for a higher degree polynomial. As expected, ANCAS remains the fastest approach due to its minimal computational overhead, although its simplicity may limit accuracy in rapidly changing scenarios. With an increased number of segments per round (16 instead of 8), SBO-ANCAS experiences a more pronounced effect, as it runs multiple refinement steps over more intervals but still maintains a lower overall runtime than CATCH. Additionally, increasing the number of points per segment increases the data size for CATCH and increases the cost of root finding, which demonstrates how sensitive its running time is to the choice of Tmin value. In contrast, ANCAS is almost unaffected by these parameter changes, which makes its computational load simple and robust.

Graph 2: Algorithms Runtime [Sec] per test, only on the first 100 tests for Tmin=8 and 16 points per segment.



A Test Case 3:

Here we have an example of a single run of the 3 algorithms over the same set of data:

- numbers of points per segment (Number of Input Points): 16
- number of segments per minimal revolution (Tmin Factor): 8
- CATCH degree:15
- SBO-ANCAS tolerances: TOLd = 0.00000001, TOLt = 0.0001

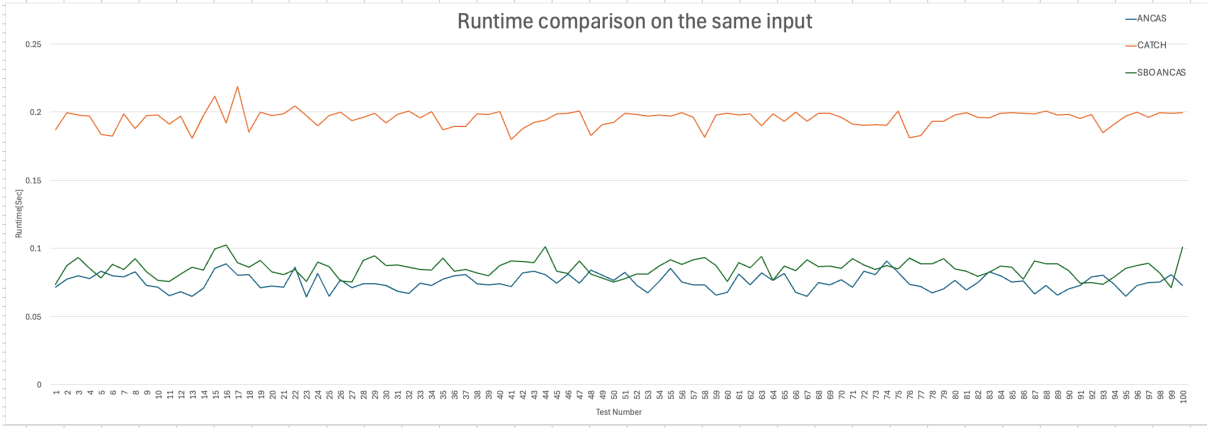
Table 6: Test data for the STARLINK-30375 satellite and COSMO-SKYMED 3 satellite in the catalog.

Test ID	Algorithm Name	Run Time[Sec]	TCA Distance[Km]	TCA[Sec]	Real TCA Distance[Km]	Real TCA[Sec]	Error Time[Sec]	Error Distance[Km]
14	ANCAS	0.070674	9.31392852502409	922993.632961531	9.31392656415478	922993.632971268	9.736977517604 83E-06	1.960869310480 01E-06
14	SBO ANCAS	0.083794	9.31392656332724	922993.632979322	9.31392656415478	922993.632971268	8.054194040596 49E-06	8.275389262735 19E-10
14	CATCH	0.197636	9.37806147 52586	914370.913044776	9.31392656375571	922993.632981267	8622.719936491 11	0.064134911502 8849

Analyzing The Data:

As illustrated by the runtime measurements for Test Case 3, CATCH emerges as the slowest among the three. Its companion matrix eigenvalue calculations for a 15th degree polynomial impose a heavy computational burden, surpassing the cost of SBO-ANCAS's iterative loops. Although SBO-ANCAS samples additional points until tolerances are satisfied, it's still outperforms CATCH in runtime, suggesting that refining cubic polynomials multiple times can be more efficient than high degree polynomial root finding. Meanwhile, ANCAS remains the fastest, reaffirming that its direct cubic approach yields minimal overhead, albeit with potential accuracy limitations if the input data does not capture sudden distance changes. This test scenario amplifies the performance gap for CATCH, raising the polynomial degree to 15 greatly increases eigenvalue computation time. Conversely, SBO-ANCAS scales more gracefully with additional segments or points, as long as the iterative refinement does not escalate into too many loops. ANCAS, by design, has a near constant runtime cost per data segment, making it consistently the fastest option at the possible expense of missing certain rapid changes in distance.

Graph 3: Algorithms Runtime [Sec] per test, only on the first 100 tests for Tmin=8 and 16 points per segment.



A Test Case 4:

Here we have an example of a single run of the 3 algorithms over the same set of data:

- numbers of points per segment (Number of Input Points): 16
- number of segments per minimal revolution (Tmin Factor): 16
- CATCH degree:15
- SBO-ANCAS tolerances: TOLd = 0.00000001, TOLt = 0.0001

Table 7: Test data for the STARLINK-30375 satellite and COSMO-SKYMED 3 satellite in the catalog.

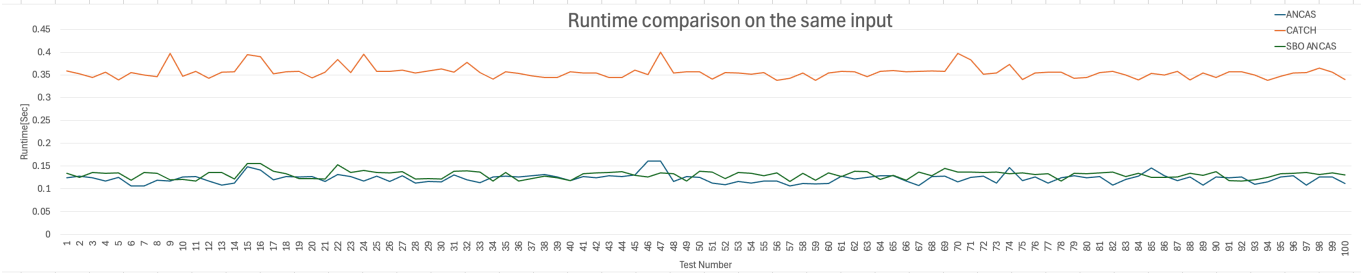
Test ID	Algorithm Name	Run Time[Sec]	TCA Distance[Km]	TCA[Sec]	Real TCA Distance[Km]	Real TCA[Sec]	Error Time[Sec]	Error Distance[Km]
14	SBO ANCAS	0.083794	9.313926563	922993.633	9.313926564	922993.633	8.05419E-06	8.27539E-10
14	CATCH	0.197636	9.378061475	914370.913	9.313926564	922993.633	8622.719936	0.064134912
14	ANCAS	0.070674	9.313928525	922993.633	9.313926564	922993.633	9.73698E-06	1.96087E-06

Analyzing The Data:

In this test case, each algorithm processes 16 points per segment, 16 segments per minimal revolution (Tmin Factor = 16). At a polynomial degree of 15, CATCH demands the most time, primarily because each interval requires eigenvalue computations for a relatively large companion matrix. While it can offer strong accuracy, the time cost escalates significantly with higher polynomial degrees. SBO-ANCAS, while performing iterative sampling until it meets the specified tolerances, still manages to maintain lower runtimes than CATCH, indicating that repeatedly solving cubic equations remains more efficient than high degree eigenvalue computations. We can see that the SBO-ANCAS delivers a good balance between accuracy and speed. Meanwhile, ANCAS continues to be the fastest, relying on a single cubic polynomial per interval with minimal overhead. Compared to scenarios with fewer points per segment or a lower Tmin factor, increasing both parameters impacts SBO-ANCAS because it must iterate over a larger number of intervals. However, this increase does not cause its runtime to exceed that of CATCH.

ANCAS, with its simple approach, has a nearly constant runtime cost per data segment, making it consistently the fastest option at the expense of missing some rapid changes in distance.

Graph 4: Algorithms Runtime [Sec] per test, only on the first 100 tests for Tmin=16 and 16 points per segment.



7.1.1.1. Overall Analyzing and Conclusion of the 4 Test Cases

Analyzing The Data:

Table 5 summarizes the average algorithm runtime across four different configurations of Number of Points per Segment and Tmin Factor (8,8), (8,16), (16,8) and (16,16). Each cell represents the sum of run times in seconds for multiple tests within that configuration, then averaged per algorithm. By examining these values, we can see how each algorithm's runtime scales as the input size and complexity increase.

Table 8: Data of Average Algorithm Runtime in seconds vs. Number of Points per Segment and Tmin Factor.

Sum of Run Time[Sec]	Column Labels			
Row Labels	ANCAS	CATCH	SBO ANCAS	Grand Total
8,8	0.03288992	0.06266644	0.04534436	0.14090072
8,16	0.07203348	0.12364917	0.0834865	0.27916915
16,8	0.07522706	0.1953885	0.08541386	0.35602942
16,16	0.12247283	0.35556683	0.13068292	0.60872258
Grand Total	**0.30262329**	**0.73727094**	**0.34492764**	**1.38482187**

Analyzing of the 4 Test Cases:

ANCAS

ANCAS is consistently the fastest across all four scenarios. Its runtime increases from ~0.033 seconds at (8,8) to ~0.122 seconds at (16,16), reflecting a roughly fourfold rise. This relatively modest growth suggests that ANCAS scales linearly or near linearly with the number of input points.

SBO-ANCAS

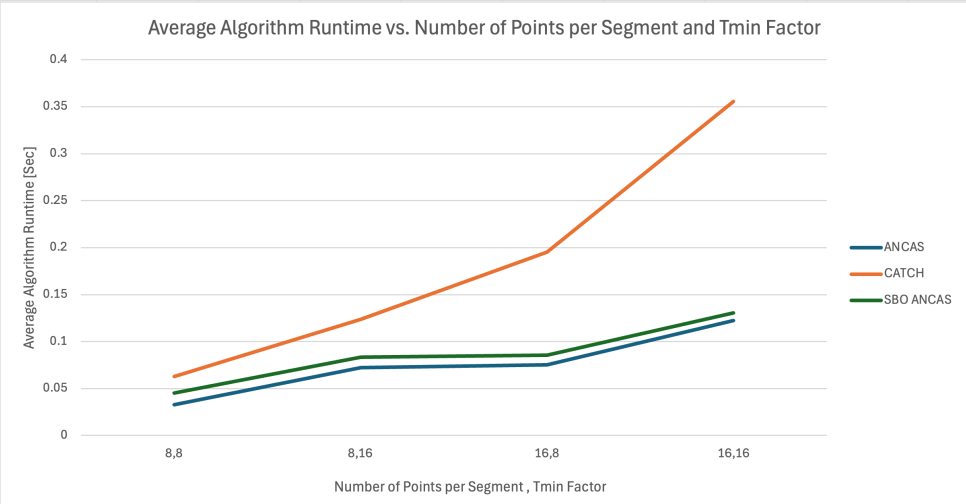
SBO-ANCAS maintains a middle ground in terms of speed. Even though it involves iterative refinement to meet distance and time tolerances, it remains faster than CATCH. As the input size grows from (8,8) to (16,16), SBO-ANCAS also shows a clear but moderate increase in runtime, from ~0.045 seconds to ~0.131 seconds.

CATCH

CATCH consistently demonstrates the highest runtime, showing a substantial jump when both the number of points per segment and the Tmin Factor increase. At (8,8),

it sits around ~0.063 seconds, but by (16,16) it escalates to ~0.356 seconds. This indicates that constructing the companion matrix and computing eigenvalues for each segment becomes increasingly expensive as the dataset expands.

Graph 5: Average Algorithm Runtime in seconds vs. Number of Points per Segment and Tmin Factor.



The experimental results across the four cases clearly show that increasing the number of points per segment and the Tmin Factor leads to higher runtimes for all algorithms. Moving from 8 to 16 points per segment roughly doubles the total number of data points, and as a result, all algorithms take longer to run. However, ANCAS shows only a small relative increase in runtime, which confirms that its simple cubic polynomial approach remains efficient even as the input size grows. SBO-ANCAS also scales moderately well; its iterative refinement steps add some overhead but do not become excessively costly, keeping its runtime lower than that of CATCH. In contrast, CATCH experiences a pronounced increase in runtime when the Tmin Factor is raised from 8 to 16, indicating that each additional interval significantly adds to the computational load due to the construction and diagonalization of a companion matrix for a high-degree polynomial. Overall, these findings demonstrate a consistent pattern: ANCAS is the fastest, SBO-ANCAS occupies a middle ground by combining speed and accuracy, and CATCH is the slowest yet potentially more precise. This suggests that ANCAS is best suited for applications requiring rapid execution, even though it may sacrifice some accuracy. In contrast, SBO-ANCAS strikes a balance by combining speed with improved accuracy, while CATCH, although slower, delivers higher precision. Therefore, when computation speed is critical, ANCAS and SBO-ANCAS are preferable, whereas CATCH might be chosen when maximum accuracy is required and longer computation times are acceptable.

7.1.2. Change in the number of points

In order to find some correlation between the input and the algorithms runtime we did four test variations, one with only a change to the number of points and one with only a change to the time interval.

A Test Case 1:

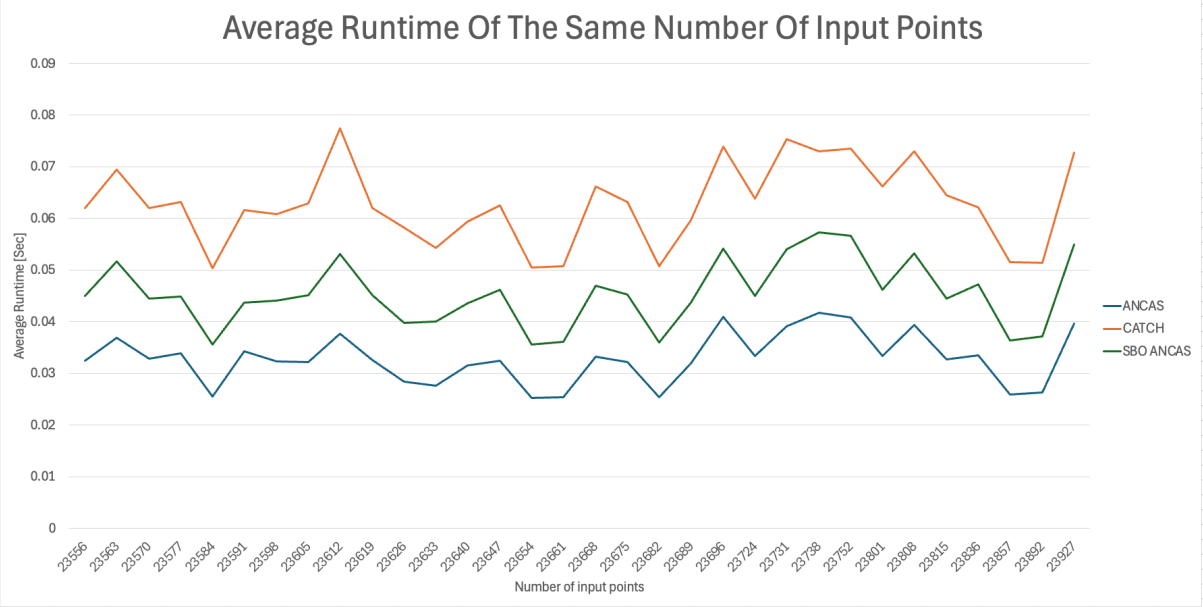
Here we have an example of a single run of the 3 algorithms over the same set of data:

- numbers of points per segment (Number of Input Points): 8
- number of segments per minimal revolution (Tmin Factor): 8
- CATCH degree:7
- SBO-ANCAS tolerances: TOLd = 0.00000001, TOLt = 0.0001

Table 9: Test data for the STARLINK-30375 satellite and COSMO-SKYMED 3 satellite in the catalog.

Test ID	Algorithm Name	Run Time[Sec]	TCA Distance[Km]	TCA[Sec]	Real TCA Distance[Km]	Real TCA[Sec]	Error Time[Sec]	Error Distance[Km]	Number of Input Points
14	ANCAS	0.025244	9.31388256	922993.6313	9.313926564	922993.633	0.001648891	4.40041E-05	23556
14	SBO ANCAS	0.035631	9.313926563	922993.633	9.313926564	922993.633	8.05419E-06	8.27539E-10	23556
14	CATCH	0.050847	9.313926218	922993.633	9.313926564	922993.633	5.78351E-06	3.46059E-07	23556

Graph 6: Algorithms Runtime [Sec] per Number of Input Points, where Tmin=8 and 8 points per segment.



A Test Case 2:

Here we have an example of a single run of the 3 algorithms over the same set of data:

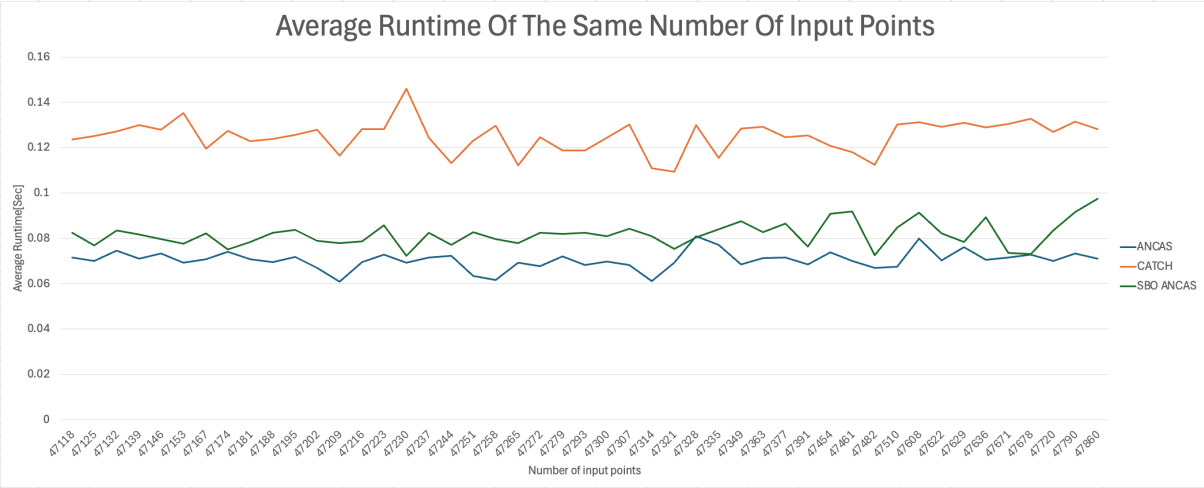
- numbers of points per segment (Number of Input Points): 8
- number of segments per minimal revolution (Tmin Factor): 16
- CATCH degree:7
- SBO-ANCAS tolerances: TOLd = 0.00000001, TOLt = 0.0001

Table 10: Test data for the STARLINK-30375 satellite and COSMO-SKYMED 3 satellite in the catalog.

Test ID	Algorithm Name	Run Time[Sec]	TCA Distance[Km]	TCA[Sec]	Real TCA Distance[Km]	Real TCA[Sec]	Error Time [Sec]	Error Distance [Km]	Number Of Input Points

14	ANCAS	0.0615599999999999	9.31393067045039	922993.632954832	9.31392656375749	922993.632981267	0.0000264354748651385	4.10669289507836E-06	47118
14	SBO ANCAS	0.0825649999999999	9.3139265628991	922993.632979321	9.31392656375749	922993.632981267	1.94599851965904E-06	4.10669289507836E-06	47118
14	CATCH	0.110103	9.31392656496836	922993.632979541	9.31392656373572	922993.632971266	8.27549956738949E-06	1.23263355078507E-09	47118

Graph 7: Algorithms Runtime [Sec] per Number of Input Points, where Tmin=16 and 8 points per segment.



A Test Case 3:

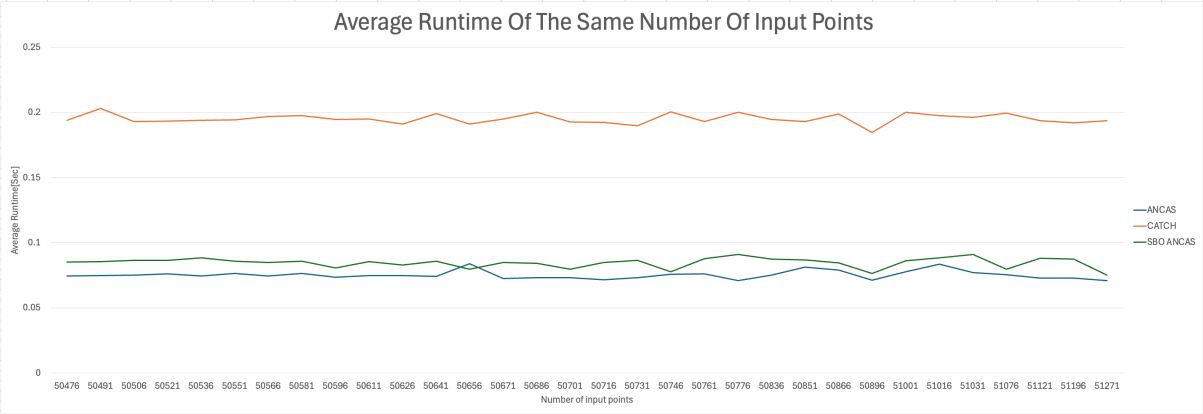
Here we have an example of a single run of the 3 algorithms over the same set of data:

- numbers of points per segment (Number of Input Points): 16
- number of segments per minimal revolution (Tmin Factor): 8
- CATCH degree:15
- SBO-ANCAS tolerances: TOLd = 0.00000001, TOLt = 0.0001

Table 11: Test data for the STARLINK-30375 satellite and COSMO-SKYMED 3 satellite in the catalog.

Test ID	Algorithm Name	Run Time[Sec]	TCA Distance[Km]	TCA[Sec]	Real TCA Distance[Km]	Real TCA[Sec]	Error Time[Sec]	Error Distance [Km]	Number Of Input Points
14	ANCAS	0.070674	9.31392852502409	922993.632961531	9.31392656415478	922993.632971268	9.73697751760483E-06	1.96086931048001E-06	50476
14	SBO ANCAS	0.083794	9.31392656332724	922993.632979322	9.31392656415478	922993.632971268	8.05419404059649E-06	8.27538926273519E-10	50476
14	CATCH	0.197636	9.3780614752586	914370.913044776	9.31392656375571	922993.632981267	8622.71993649111	0.0641349115028849	50476

Graph 8: Algorithms Runtime [Sec] per Number of Input Points, where Tmin=8 and 16 points per segment.



A Test Case 4:

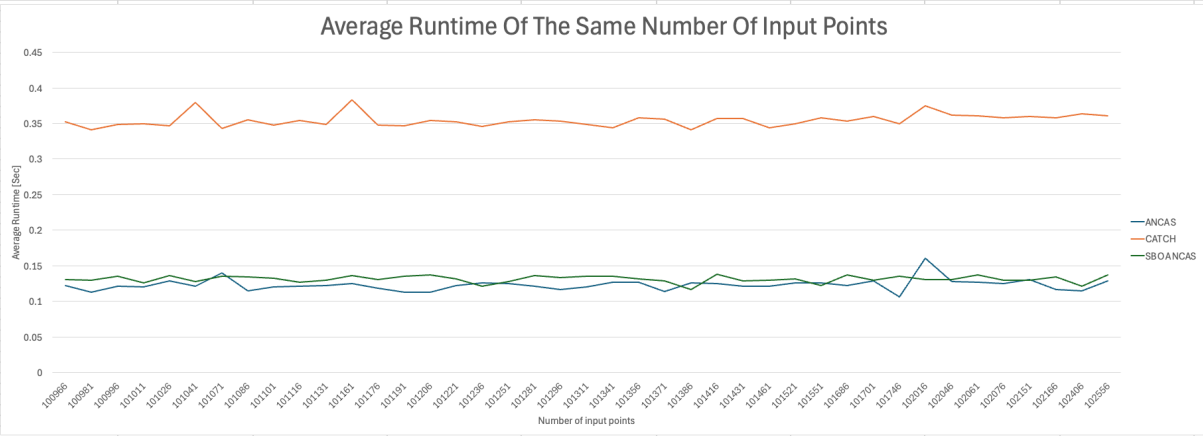
Here we have an example of a single run of the 3 algorithms over the same set of data:

- numbers of points per segment (Number of Input Points): 16
- number of segments per minimal revolution (Tmin Factor): 16
- CATCH degree:15
- SBO-ANCAS tolerances: TOLd = 0.00000001, TOLt = 0.0001

Table 12: Test data for the STARLINK-30375 satellite and COSMO-SKYMED 3 satellite in the catalog.

Test ID	Algorithm Name	Run Time[Sec]	TCA Distance[Km]	TCA[Sec]	Real TCA Distance[Km]	Real TCA[Sec]	Error Time[Sec]	Error Distance[Km]	Number Of Input Points
14	SBO ANCAS	0.083794	9.313926563	922993.633	9.313926564	922993.633	8.05419E-06	8.27539E-10	100966
14	CATCH	0.197636	9.378061475	914370.913	9.313926564	922993.633	8622.719936	0.064134912	100966
14	ANCAS	0.070674	9.313928525	922993.633	9.313926564	922993.633	9.73698E-06	1.96087E-06	100966

Graph 9: Algorithms Runtime [Sec] per Number of Input Points, where Tmin=16 and 16 points per segment.



Analyzing The Data:

Like you can see at the graphs [Graph 6,Graph 7,Graph 8, Graph 9], across the four test cases (8–8, 8–16, 16–8, and 16–16), we see that ANCAS consistently scales in a

near-linear with the number of input points, confirming our expectation that its straightforward cubic polynomial approach imposes minimal overhead even as inputs grow. SBO-ANCAS, while generally taking longer than ANCAS due to iterative refinement, also increases predictably with additional points but maintains a more moderate rise in runtime than CATCH. CATCH, by contrast, shows the steepest increase in runtime as input size and polynomial degree grow, underscoring how constructing a companion matrix and computing eigenvalues becomes more expensive under larger or more complex datasets. Overall, these observations highlight that ANCAS remains the fastest option, SBO-ANCAS provides a balanced middle ground, and CATCH exhibits the most significant jump in runtime when the number of input points and Tmin Factor increase.

7.1.2.1. Correlations and Conclusions

After testing the changing the number of points we arrived to the following observations:

ANCAS

As anticipated, ANCAS is linear to the number of input data, with constant time interval size (week), this is similar to our algorithm analysis, where the number of iterations is dependent on the number of points and in each iteration we have a constant number of operations.

SBO ANCAS

After testing the effect of changing the number of points, we observed that SBO-ANCAS's runtime is influenced by both the total number of input points and the data characteristics. Increasing the number of points per segment leads to a higher number of outer iterations, which in turn increases the overall runtime. Additionally, when more input points are provided, the algorithm may require more inner iterations, especially if the time interval is larger or if the satellite data shows gradual changes that necessitate finer refinement to meet the desired tolerances. In other words, SBO-ANCAS's performance is highly dependent on the size of the dataset and the specific dynamics of the input data, for instance, slower satellite motion tends to require additional inner iterations to reach the required distance tolerance.

CATCH

In CATCH, we expected to see linear correlation between the number of input point and the algorithm runtime, and it seems like that’s the case. Additionally CATCH seems to have steeper increase rate, resulting from the high cost of calculating the roots in each iteration, giving higher increase in time the more iterations we have.

7.1.3. Errors Analysis

To test the errors, we ran four variations over a data set and checked the results.

A Test Case 1:

Here we have an example of a single run of the 3 algorithms over the same set of data:

- numbers of points per segment (Number of Input Points): 8
- number of segments per minimal revolution (Tmin Factor): 8
- CATCH degree:7

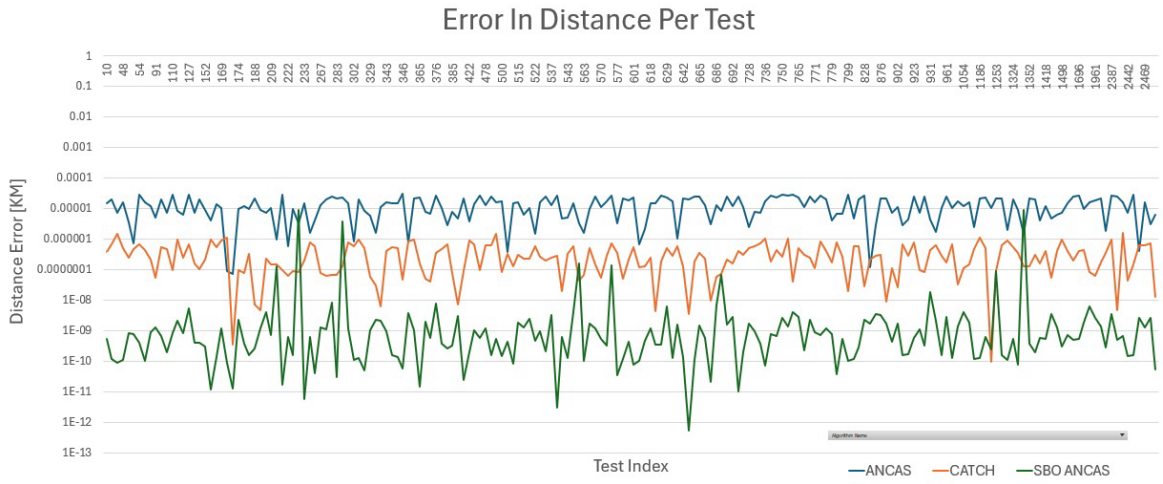
- SBO-ANCAS tolerances: TOLd = 0.00000001, TOLt = 0.0001

Table 13: Test data for the STARLINK-30375 satellite and STARLINK-1362 satellite in the catalog.

Test ID	Algorithm Name	Run Time[Sec]	TCA Distance[Km]	TCA[Sec]	Real TCA Distance[Km]	Real TCA[Sec]	Error Time[Sec]	Error Distance[Km]	Passed Tests Error $Error\ Distance[Km] \leq 0.00001$
226	ANCAS	0.025416	15.73760844	1606879.915	15.73761768	1606879.915	0.0003189	9.2419E-06	79
226	SBO ANCAS	0.035748	15.73761768	1606879.915	15.73761768	1606879.915	4.52292E-05	1.49925E-10	2219
226	CATCH	0.050354	15.73761759	1606879.915	15.73761768	1606879.915	4.93787E-05	8.91634E-08	2388

Analyzing The Data:

Graph 10: Algorithms distance error [KM] per test in logarithmic scale, only for the tests that passed the defined tolerance 0.00001 for Tmin=8 and 8 points per segment.



This chart shows the test cases that successfully pass the predefined tolerance threshold. The results illustrate significant differences in error distance performance between the three algorithms.

SBO-ANCAS consistently achieves the lowest error distances, mainly in the range of 10^{-10} to 10^{-7} km, highlighting its exceptional accuracy under the specified test conditions. CATCH exhibits moderate accuracy, with error distances typically spanning 10^{-7} to 10^{-6} km, suggesting a balance between computational efficiency and accuracy. In contrast, ANCAS exhibits the highest error distances, often in the range of 10^{-6} to 10^{-5} km, indicating reduced accuracy under these experimental conditions. These results place SBO-ANCAS as the most reliable algorithm when requiring high accuracy, while CATCH exhibits a balance between accuracy and computational efficiency, making it a possible alternative in scenarios where computational resources are a constraint. ANCAS, despite its functionality, exhibits the most pronounced error margins, making it less suitable where stringent error minimization is necessary.

A Test Case 2:

Here we have an example of a single run of the 3 algorithms over the same set of data:

- numbers of points per segment (Number of Input Points): 8

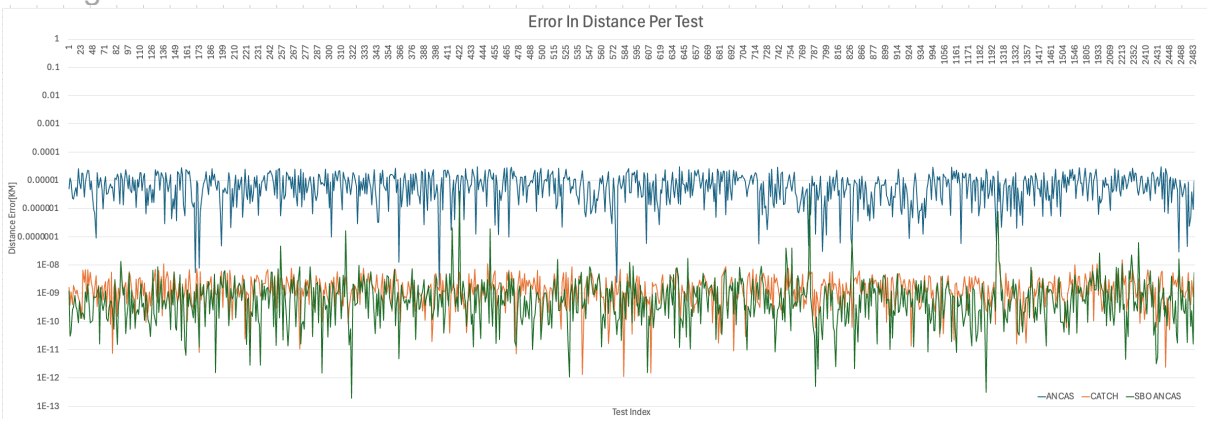
- number of segments per minimal revolution (Tmin Factor): 16
- CATCH degree:7
- SBO-ANCAS tolerances: TOLd = 0.00000001, TOLt = 0.0001

Table 14: Test data for the STARLINK-30375 satellite and STARLINK-1362 satellite in the catalog.

Test ID	Algorithm Name	Run Time[Sec]	TCA Distance[Km]	TCA[Sec]	Real TCA Distance[Km]	Real TCA[Sec]	Error Time[Sec]	Error Distance[Km]	Passed Tests $Error\ Distance[Km] \leq 0.00001$
226	ANCAS	0.069776	15.7376194	1606879.915	15.73761768	1606879.915	3.06275E-05	1.71636E-06	613
226	SBO ANCAS	0.082644	15.73761768	1606879.915	15.73761768	1606879.915	4.52292E-05	1.49379E-10	2314
226	CATCH	0.124484	15.73761769	1606879.915	15.73761768	1606879.915	4.50362E-05	1.52018E-09	2354

Analyzing The Data:

Graph 11: Algorithms distance error [KM] per test in logarithmic scale,only for the tests that passed the defined tolerance 0.00001 for Tmin=16 and 8 points per segment.



First, it can be observed that for this data, a much larger number of tests have passed the desired tolerance. The analysis of the average error distance across different methods ANCAS, CATCH, and SBO ANCAS demonstrates significant variations in accuracy under the given test conditions. SBO-ANCAS consistently produces the lowest error values. This outcome is attributable to its iterative refinement process, which repeatedly samples point around the detected minimum to better approximate the true distance function. CATCH also yields lower errors than ANCAS by using a Chebyshev Proxy Polynomial to capture more complex variations in the distance function, although its computational burden remains high due to the eigenvalue calculations involved. In contrast, ANCAS, which relies on a simple cubic polynomial approximation, shows the highest errors because it may miss critical extrema when the function behavior is more complex. The overall trend highlights the efficiency of SBO ANCAS in minimizing errors, making it a favorable approach when required high precision.

A Test Case 3:

Here we have an example of a single run of the 3 algorithms over the same set of data:

- numbers of points per segment (Number of Input Points): 16

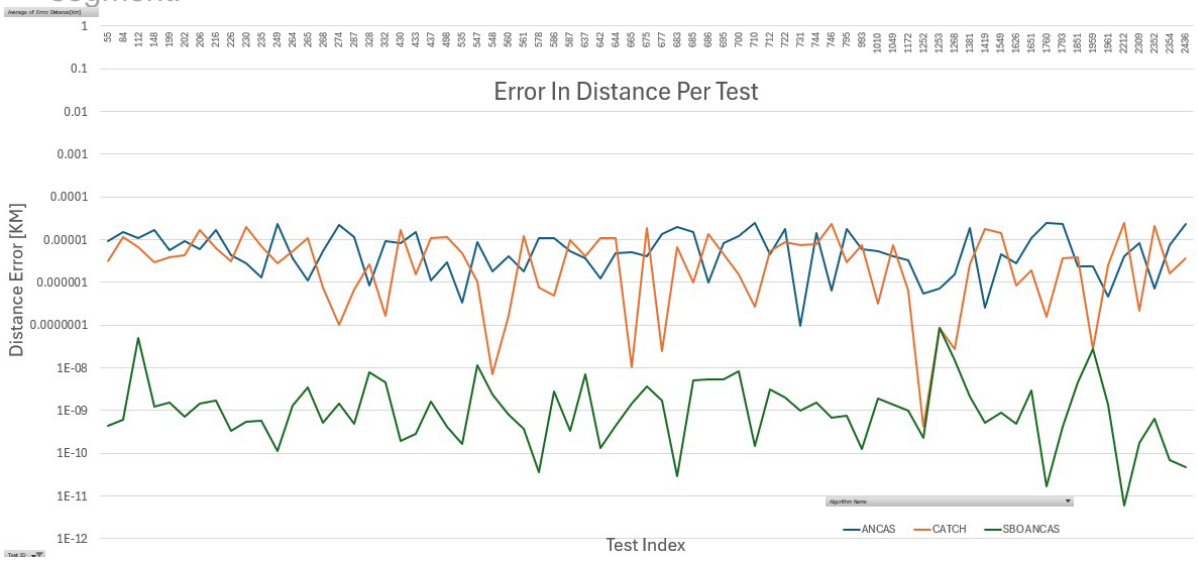
- number of segments per minimal revolution (Tmin Factor): 8
- CATCH degree:15
- SBO-ANCAS tolerances: TOLd = 0.00000001, TOLt = 0.0001

Table 15: Test data for the STARLINK-30375 satellite and STARLINK-1362 satellite in the catalog.

Test ID	Algorithm Name	Run Time[Sec]	TCA Distance[Km]	TCA[Sec]	Real TCA Distance[Km]	Real TCA[Sec]	Error Time[Sec]	Error Distance[Km]	Passed Tests $Error\ Distance[Km] \leq 0.00001$
226	ANCAS	0.066284	15.73762201	1606879.915	15.73761768	1606879.915	6.86524E-06	4.32641E-06	692
226	SBO ANCAS	0.077593	15.73761768	1606879.915	15.73761768	1606879.915	1.52194E-05	3.33596E-10	2298
226	CATCH	0.195762	15.73762087	1606879.912	15.73761768	1606879.915	0.003148539	3.19001E-06	591

Analyzing The Data:

Graph 12: Algorithms distance error [KM] per test in logarithmic scale, only for the tests that passed the defined tolerance 0.00001 for Tmin=8 and 16 points per segment.



The test results reveal that the average error distance is significantly influenced by the initial input parameters. Unlike previous cases, where ANCAS was consistently the method with the highest error values, the current results indicate that its error rates are not always distinctly worse than CATCH. However, it remains evident that ANCAS generally exhibits higher error distances compared to SBO-ANCAS and CATCH. SBO-ANCAS once again demonstrates the lowest error values, reinforcing its superior precision across various test cases. This result confirms that SBO-ANCAS consistently outperforms the other methods in minimizing error distances, making it the most effective choice among the tested approaches. Additionally, in this test case, we observe unexpectedly high error values for CATCH, an outcome that runs contrary to prior expectations, since CATCH typically achieves higher accuracy than ANCAS because it uses higher degree Chebyshev polynomials to approximate the relative distance function, whereas ANCAS relies on a single cubic polynomial. By employing more polynomial terms, CATCH can capture more subtle variations in the satellite trajectories, thus producing a closer fit to the true distance function. In addition, the

companion matrix approach used in CATCH systematically identifies all relevant roots of the high-degree polynomial, further enhancing its accuracy in locating the minimum distance point (Time of Closest Approach). Conversely, the cubic polynomial in ANCAS can only approximate a narrower range of possible function shapes, making it inherently less precise under most conditions.
A likely explanation for the anomalous CATCH results in this scenario is that, with a polynomial degree of 16, the dataset grows significantly, increasing the complexity of the companion matrix eigenvalue calculation. This can lead to numerical instabilities or other computational issues that inflate the distance error. Additionally, the satellite's trajectory may include challenging segments or singular points that exacerbate these numerical effects. Because of these factors, the results in this test case are not sufficient to draw a definitive conclusion regarding CATCH's overall performance. Further testing, especially under different input configurations and degrees, will be necessary to isolate and understand the conditions under which CATCH might produce anomalous errors. Dr. Elad and his future research teams will conduct a separate, in-depth study to investigate the factors affecting the accuracy of CATCH in greater detail, especially in cases where the degree of the polynomial is 16.

A Test Case 4:

Here we have an example of a single run of the 3 algorithms over the same set of data:

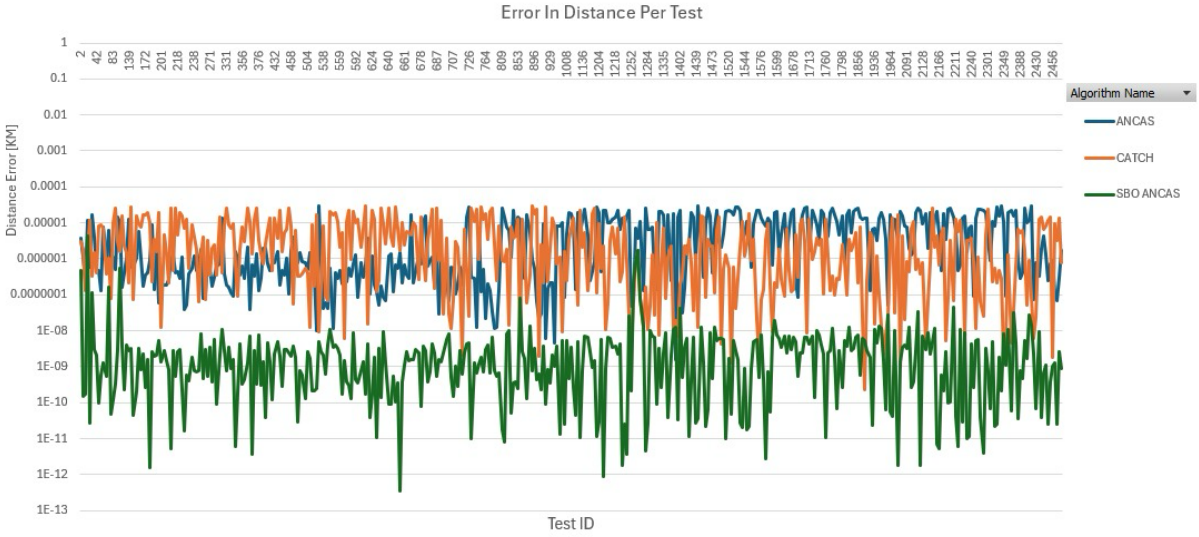
- numbers of points per segment (Number of Input Points): 16
- number of segments per minimal revolution (Tmin Factor): 16
- CATCH degree:15
- SBO-ANCAS tolerances: TOLd = 0.00000001, TOLt = 0.0001

Table 16: Test data for the STARLINK-30375 satellite and STARLINK-1362 satellite in the catalog.

Test ID	Algorithm Name	Run Time[Sec]	TCA Distance[Km]	TCA[Sec]	Real TCA Distance[Km]	Real TCA[Sec]	Error Time[Sec]	Error Distance[Km]	Passed Tests $Error\ Distance[Km] \leq 0.00001$
226	ANCAS	0.11078	15.73761729	1606879.915	15.73761768	1606879.915	7.19919E-05	3.9088E-07	1202
226	SBO ANCAS	0.126707	15.73761768	1606879.915	15.73761768	1606879.915	6.481E-05	4.00791E-10	2296
226	CATCH	0.356273	15.73761821	1606879.914	15.73761768	1606879.915	0.001311346	5.27668E-07	931

Analyzing The Data:

Graph 13: Algorithms distance error [KM] per test in logarithmic scale, only for the tests that passed the defined tolerance 0.00001 for Tmin=16 and 16 points per segment.



In this test case, as seen in previous test cases, SBO-ANCAS consistently demonstrates the lowest error values, confirming its superior precision and making it the most effective approach among the three algorithms, while ANCAS continues to produce relatively higher error values, which aligns with earlier findings that it is generally less reliable and tends to produce larger deviations from the true distance. In this test case, we observe unexpectedly high error values for CATCH, similar to those seen in Test Case 3 (with 16 points per segment and a Tmin Factor of 8), an outcome that contradicts our prior expectations, this is because CATCH typically achieves greater accuracy than ANCAS by employing higher degree Chebyshev polynomials to capture the finer nuances of satellite trajectories. Notably, when the polynomial degree is set to 16, the dataset increases substantially, which in turn heightens the complexity of calculating the eigenvalues of the associated companion matrix. This increased complexity can introduce numerical instabilities or other computational challenges that result in inflated distance errors, and the satellite's trajectory in this scenario may be particularly complex, featuring several singular points or rapid orbital changes that further amplify these numerical sensitivities.
Because of these factors, the results in this test case are not sufficient to draw a definitive conclusion regarding CATCH's overall performance. Further testing, especially under different input configurations and degrees, will be necessary to isolate and understand the conditions under which CATCH might produce anomalous errors. Dr. Elad and his future research teams will conduct a separate, in-depth study to investigate the factors affecting the accuracy of CATCH in greater detail, especially in cases where the degree of the polynomial is 16.

7.1.3.1. Overall Analyzing and Conclusion of the 4 Test Cases

Analyzing The Data:

Table 6 summarizes the average error distance in kilometers across four different configurations of Number of Points per Segment and Tmin Factor: (8,8), (8,16), (16,8), and (16,16). Each cell represents the average error distance for multiple tests within that configuration, calculated per algorithm. By analyzing these values, we can observe how the accuracy of each algorithm varies as the input parameters change.

Table 17: Data of Average Algorithm Distance Error in Km vs. Number of Points per Segment and Tmin Factor.

Average of Avg Error Distance [Km]	Column Labels			
Row Labels	ANCAS	CATCH	SBO ANCAS	Grand Total
8,8	1.33326E-05	3.47184E-07	1.145E-07	4.5981E-06
8,16	8.78415E-06	2.95664E-09	1.04548E-08	2.93252E-06
16,8	8.04129E-06	5.92527E-06	4.17849E-09	4.65691E-06
16,16	6.7953E-06	5.32866E-06	2.2904E-08	4.04895E-06
Grand Total	**9.23834E-06**	**2.90102E-06**	**3.80092E-08**	**4.05912E-06**

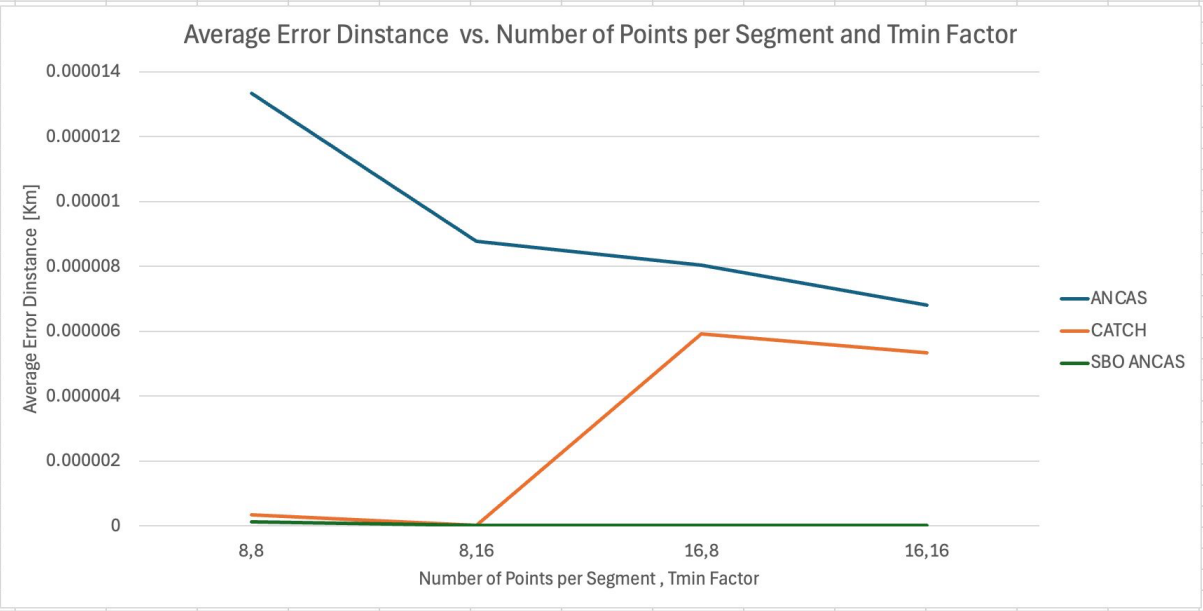
Analyzing of the 4 Test Cases:

ANCAS
consistently exhibits the highest average error across all four test scenarios. At (8,8), its error is approximately 1.33E-05 Km, and it decreases slightly to 6.79E-06 Km at (16,16). This reduction suggests that ANCAS benefits from increasing the number of points per segment and Tmin Factor, likely due to better trajectory fitting. However, even with this improvement, ANCAS still maintains the highest error compared to other algorithms, indicating that while it scales well, its inherent accuracy is lower.

SBO-ANCAS
SBO-ANCAS consistently achieves the lowest error across all configurations. At (8,8), it starts with 1.145E-07 Km, which further decreases to 2.29E-08 Km at (16,16). This suggests that SBO-ANCAS is highly effective in refining trajectory accuracy, even as input size increases. The minimal increase in error at larger input sizes highlights its robustness and ability to maintain precision under varying conditions.

CATCH
CATCH exhibits varying accuracy depending on input parameters. It achieves low error at (8,8) 3.47E-07 Km and even lower at (8,16) 2.96E-09 Km, indicating strong performance in these cases. However, at (16,8), the error increases sharply to 5.93E-06 Km, and at (16,16), it remains high at 5.33E-06 Km. This suggests that CATCH performs well when the number of points per segment is lower but struggles as it increases. This behavior is likely due to the companion matrix computation, where certain input conditions introduce numerical instability. While CATCH can achieve high accuracy in some cases, its performance is inconsistent across all configurations.

Graph 14: Graph showing the average error values for each algorithm over the different data configurations we presented above.



The experimental results show that the average error distance varies across algorithms and test configurations, influenced by their underlying computational methods. ANCAS consistently has the highest error, though it decreases as the number of points per segment and Tmin Factor increase. This suggests that while its simple polynomial approach benefits from more data points for better trajectory fitting, it lacks advanced refinement steps to further reduce errors. The gradual decline in error indicates that its accuracy improves with higher input density but remains the least precise among the tested algorithms. SBO-ANCAS maintains the lowest error in all cases, highlighting the effectiveness of its iterative refinement process. This suggests that SBO-ANCAS can consistently correct deviations, regardless of the number of input points or Tmin Factor. Its minimal fluctuation in error indicates that it is robust to changes in input complexity, making it the most reliable choice for accuracy-driven applications. CATCH demonstrates mixed performance, achieving low error at (8,8) and (8,16) but experiencing a sharp increase at (16,8) and maintaining relatively high error at (16,16). This variability is likely due to the numerical properties of its companion matrix approach, which may introduce instability as the number of points increases without a corresponding Tmin Factor adjustment. This suggests that while CATCH can achieve high accuracy under certain conditions, its performance is highly sensitive to the input configuration.

Overall, SBO-ANCAS is the most reliable algorithm in terms of accuracy, making it the preferred choice when minimizing error is the priority. The (16,16) configuration is the most efficient for SBO-ANCAS, as it provides the lowest recorded error while leveraging increased data points for refinement.
Surprisingly, the results for CATCH did not align with the findings of Dr. Elad's research, where he determined that a polynomial degree of 16 is the most suitable for this algorithm. Our results, however, show that when the polynomial degree is changed to 16, CATCH begins to significantly increase its error. This discrepancy suggests that additional factors, such as numerical stability or dataset characteristics, may influence CATCH's performance under different conditions.

To address this inconsistency, Dr. Elad will continue investigating this case in future research with his upcoming teams. Further analysis will help clarify the factors affecting CATCH's accuracy and determine whether modifications to its polynomial computation can improve its stability.

7.2. Conclusion

The overall analysis of the experimental results reveals clear trends regarding the trade-offs between runtime, accuracy, and input size.

From a runtime perspective, increasing the number of points per segment and the Tmin Factor leads to longer computation times for all algorithms. ANCAS demonstrates the fastest execution, with only a modest increase in runtime as the input size grows, thanks to its simple cubic polynomial approach. SBO-ANCAS shows moderate scaling behavior, where its iterative refinement introduces some overhead but does not lead to excessive computation times. CATCH, however, exhibits the most significant runtime increase, particularly when the Tmin Factor rises from 8 to 16, as constructing and diagonalizing its companion matrix for a high-degree polynomial becomes computationally expensive. This makes CATCH the slowest of the three algorithms.

Regarding accuracy, SBO-ANCAS consistently achieves the lowest average error across all configurations, indicating its robustness and ability to refine results effectively. ANCAS, in contrast, has the highest error in all cases, though it benefits from increasing the number of points per segment. CATCH delivers low error in some cases (8,8 and 8,16) but experiences a significant rise in error at (16,8) and (16,16). Surprisingly, this contradicts previous findings, which suggested that CATCH would perform optimally with a polynomial degree of 16. The fact that CATCH's error increased significantly in these cases suggests that numerical instabilities or other computational challenges that result in inflated distance errors, and the satellite's trajectory in this scenario may be particularly complex, featuring several singular points or rapid orbital changes that further amplify these numerical sensitivities. Because of these factors, the results in this test case are not sufficient to draw a definitive conclusion regarding CATCH's overall performance. Further testing, especially under different input configurations and degrees, will be necessary to isolate and understand the conditions under which CATCH might produce anomalous errors. Dr. Elad and his future research teams will conduct a separate, in-depth study to investigate the factors affecting the accuracy of CATCH in greater detail, especially in cases where the degree of the polynomial is 16.

When analyzing the effect of total data points, increasing the number of points per segment from 8 to 16 approximately doubles the total number of data points, leading to increased runtime across all algorithms. However, the results suggest that increasing both the number of points and the Tmin Factor together (e.g., moving from 8,8 to 16,16) does not always yield proportional improvements in accuracy. SBO-ANCAS benefits most from larger datasets, as it maintains high accuracy with only a moderate runtime increase. CATCH, however, struggles with larger inputs, showing a drastic error increase, particularly at (16,8) and (16,16), which raises concerns about its numerical stability when handling larger data matrices.

Based on both runtime and accuracy data, we can see that SBO-ANCAS is the most efficient algorithm overall. It provides the best balance, maintaining low error and moderate execution times, making it the most suitable for applications requiring both speed and precision.

The optimal input configuration for SBO-ANCAS is (16,16), as it offers the best accuracy for SBO-ANCAS while keeping its runtime manageable. This configuration ensures that SBO-ANCAS has sufficient data points for refinement, while still preventing excessive computational overhead. Regarding CATCH, our current data indicate that an (8,16) configuration provides the best compromise between accuracy and runtime, given the unexpected error spikes we observed at (16,8) and (16,16). Although previous research suggested that degree 16 would be optimal, our tests revealed anomalies that require more thorough investigation, especially to understand the numerical instabilities affecting CATCH under certain conditions. As for ANCAS, higher point counts clearly improve accuracy with a near-linear increase in runtime, making 16-16 the optimal input configuration for maximizing accuracy without making the runtime too long. In future studies, Dr. Elad Dannenberg hopes to clarify the factors influencing CATCH's performance in order to better leverage its potential.

In conclusion, in our project the two main principles we tried to implement in each decision is reusability and scalability, we tried to make our implementation generic enough to support any future update that might be needed, for example adding algorithms or algorithms variants, adding communication types and so on. Additionally, we want our application to be available for any follow up team trying to test similar things or different algorithms in a similar environment, to be used as a base or reference for future uses. We tried to make our code as readable as possible and took the time to document and create diagrams for every part we could. We know that there is already future related project on the way.
We completed the tasks we placed in the first part of the project, the algorithms analysis and implementation, the testing system and the feasibility testing and analysis.

We completed the tasks we placed in the first part of the project, analyzing and implementing the algorithms on the selected OBC, developing the testing system, and conducting feasibility testing and analysis. While ANCAS achieves the fastest runtime but offers lower accuracy, and CATCH can deliver higher accuracy at a considerable computational cost, our findings indicate that SBO-ANCAS offers the best balance between accuracy and runtime

In addition, we found that, when running CATCH on the board with a polynomial degree of 16, its accuracy and runtime conflicted with the findings previously reported by Dr. Elad Danenberg. Consequently, further testing, with varied input data and across different satellites, is warranted to identify the factors influencing CATCH's runtime and accuracy. Although we did not arrive at a definitive conclusion regarding CATCH, we are glad that subsequent teams working with Dr. Elad Dannenberg can utilize, and if necessary, extend, our testing system to collect substantial data for deeper analysis and future research.

8. User guide

8.1. Testing Station App

Configuring the System

The application might need specific configuration in order to run properly, because the IP address we used when communicating with the Tested OBC App can be different between system and operation modes. All the application configuration can be done using TestingStationAppSettings.INI file, found where the application is located. In the configuration file you can chose the communication types, UDP (Not recommended), TCP(Recommended) and Local Simulation. The local simulation is an asynchronized mode running without the Tested OBC App while simulating the full application capabilities and can be used to test the environment. For the UDP you need to place both the Testing Station PC IP in the local Ip address field and the Tested OBC PC IP in the destination IP address filed. For TCP you only need to update the local IP and port.

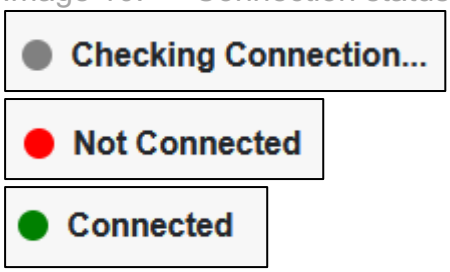
Image 9: Testing Station App configuration. The Source and Destination ports should be the opposite of the Tested OBC App ports.

```
[General]
#The type of the Communication channel. options: LocalSimulation , Tcp , Udp
CommChannelType = Tcp
#IP for any IP related comm channel
LocalIpAddress = 127.0.0.1
DestIpAddress = 127.0.0.1
#Local port
SourcePort = 8889
#destination port
DestinationPort = 8888
```

Top Menu

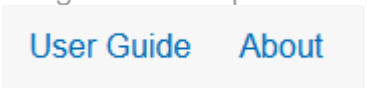
In the top of our application we have a menu, containing 3 things:
Tested OBC App Connection indicator:

Image 10: Connection status indicators.



About link, leading to the project book, and User Guide, Leading to a PDF with this guide.

Image 11: Top Menu options.



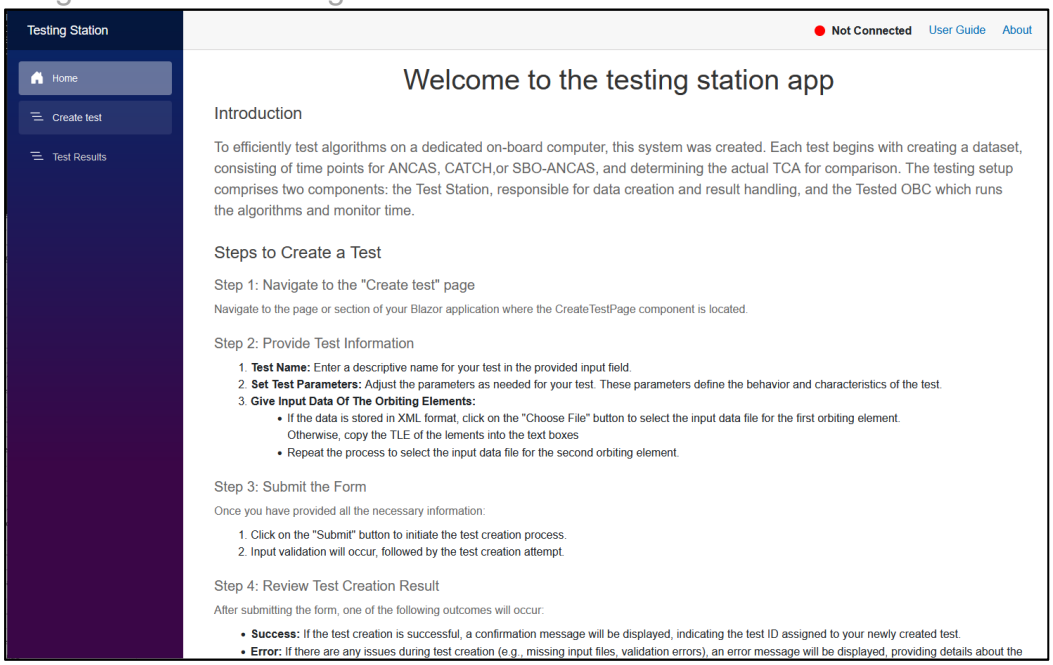
Menu

We have 3 options in the menu, leading to the different pages.

Home Page

The Home Page contain instructions on how to create a new test and view the results.

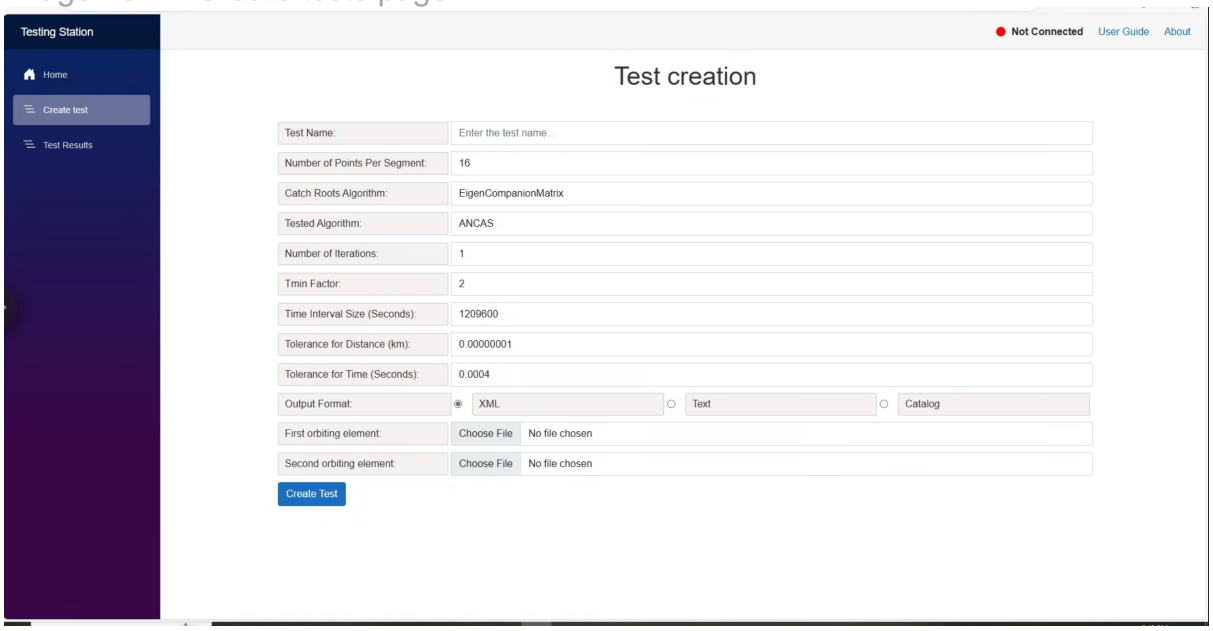
Image 12: Home Page and the Menu.



Creating a new test

To create a new test, start by navigating to the Create test page:

Image 13: Create tests page.

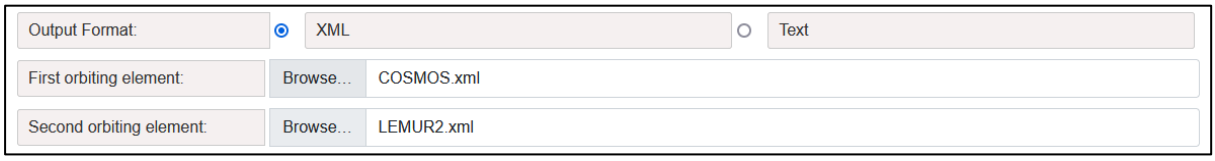


The create test form come with some default values, with number of points per segment set to 16, number of iterations to 1, TminFactor to 2, the time interval to 1 week and the tolerances to their default values.

You start by entering the test name, doesn't need to be a unique name. after that choose the desired algorithm and enter the input satellites data.

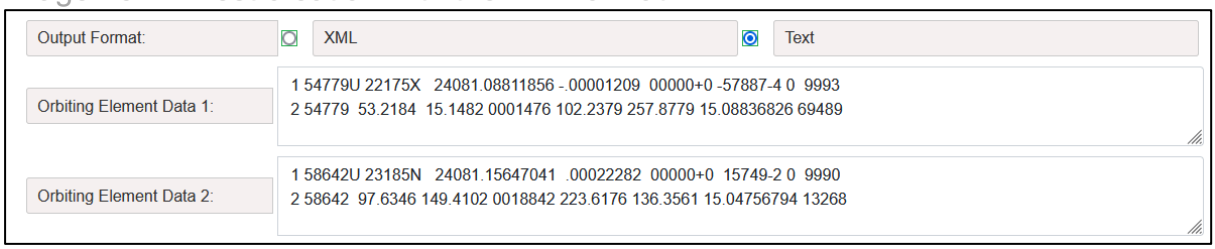
The first input option is XML files, simply choose 2 files from your file system.

Image 14: Test creation with XML format.



The second input option is TLE, placing the 2 lines of input for each satellite inside the text box as 2 lines.

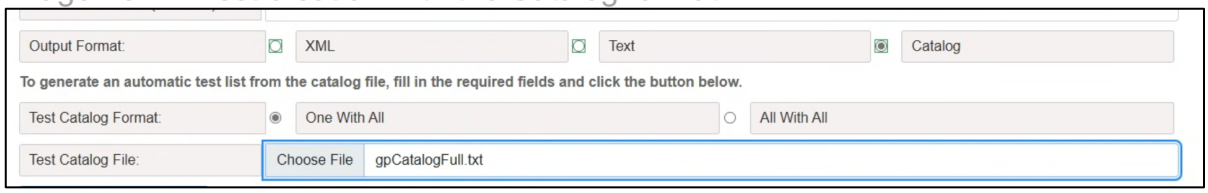
Image 15: Test creation with the TLE format.



The third input option is a full catalog text file. Simply select a text file from your file system and choose the format for creating tests from the catalog (The maximum size of the file should be less then 512000 bytes):

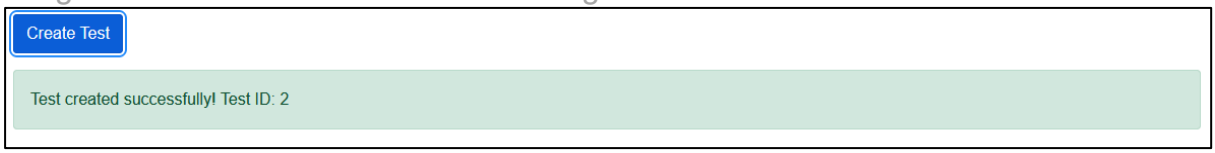
- One with all - The first object in the file is tested against all other objects in the file.
- All with all-All objects in the file are tested against each other. The default value is One with all.

Image 16: Test creation with the Catalog format.



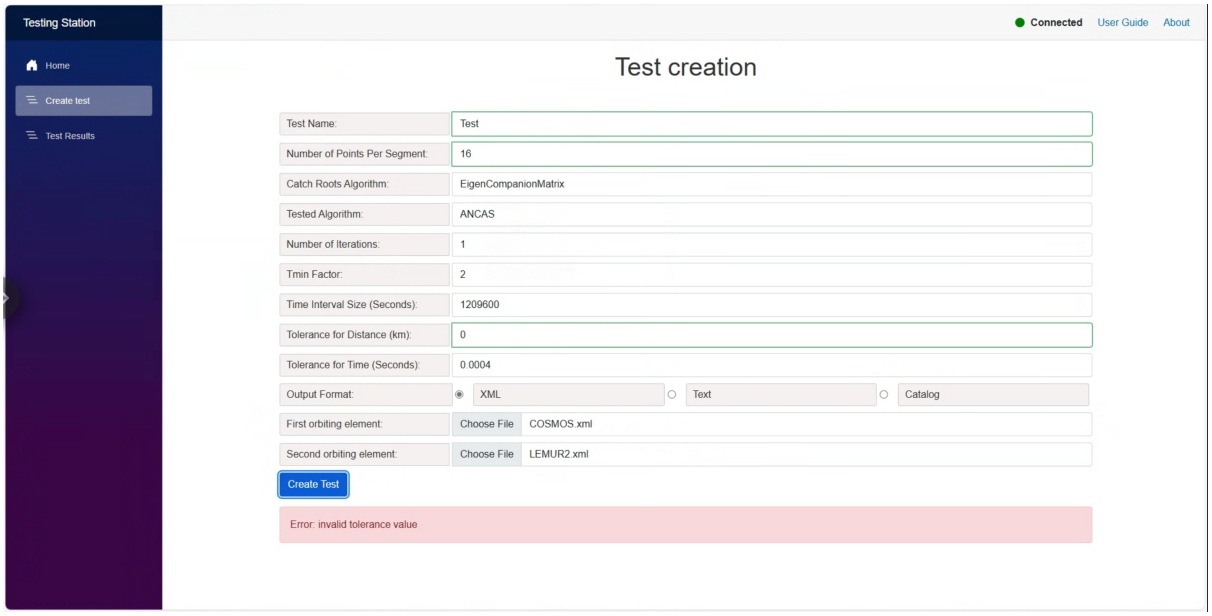
For XML or TLE format selection, after pressing the "Create Test" button, if your input was correct, the test will be created and you'll get the test ID with successful message.

Image 17: Test create success message.



For XML or TLE format selection, after pressing the "Create Test" button, if any of the inputs are incorrect, the test will not be created, and an error message will be displayed indicating the issue.

Image 18: Failed to create test message.

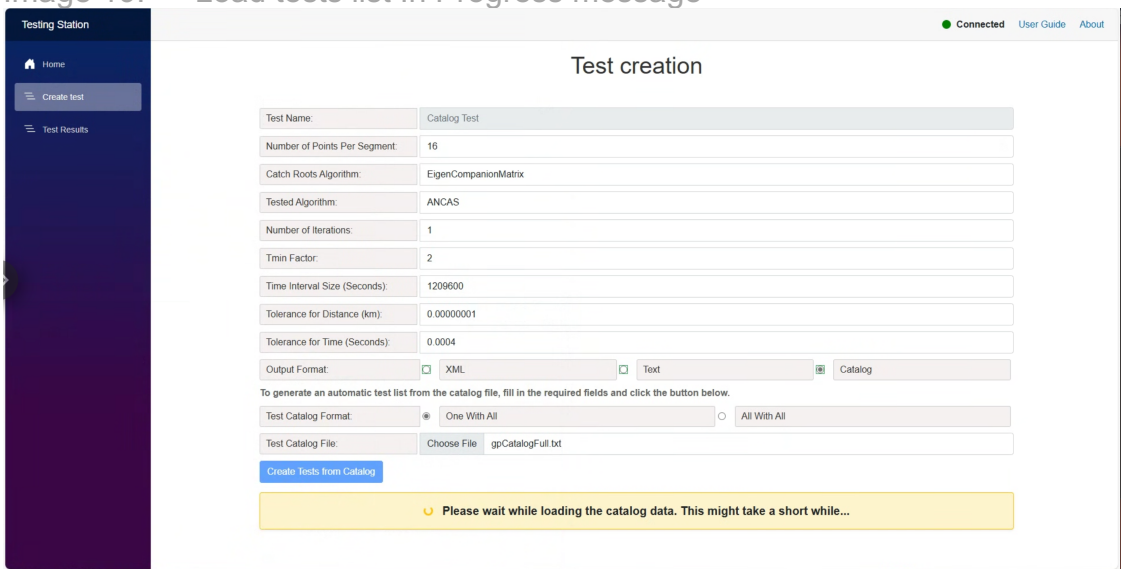


For catalog format selection, after pressing the "Create Test List" button, if your input was correct, the system will generate a list of tests based on the user's selection, One with all or All with all.

During this process, a message will be displayed informing the user that the operation is in progress and that they must wait.

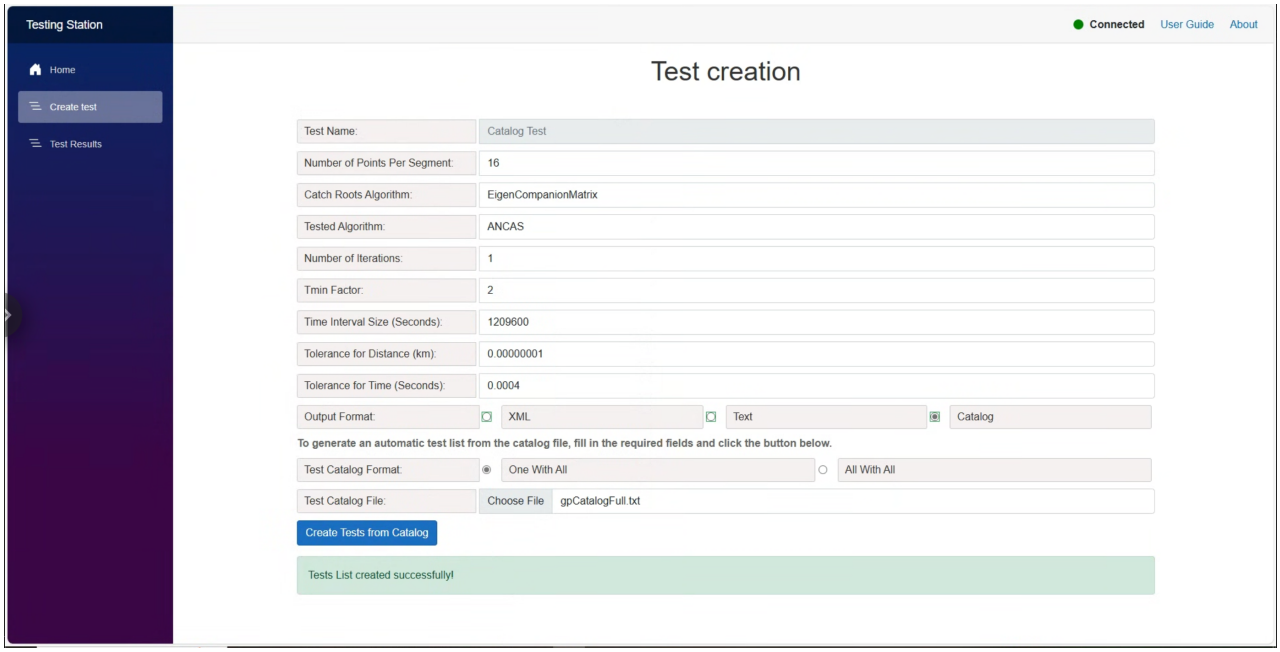
The user will not be able to create additional tests until the test list generation is successfully completed.

Image 19: Load tests list In Progress message



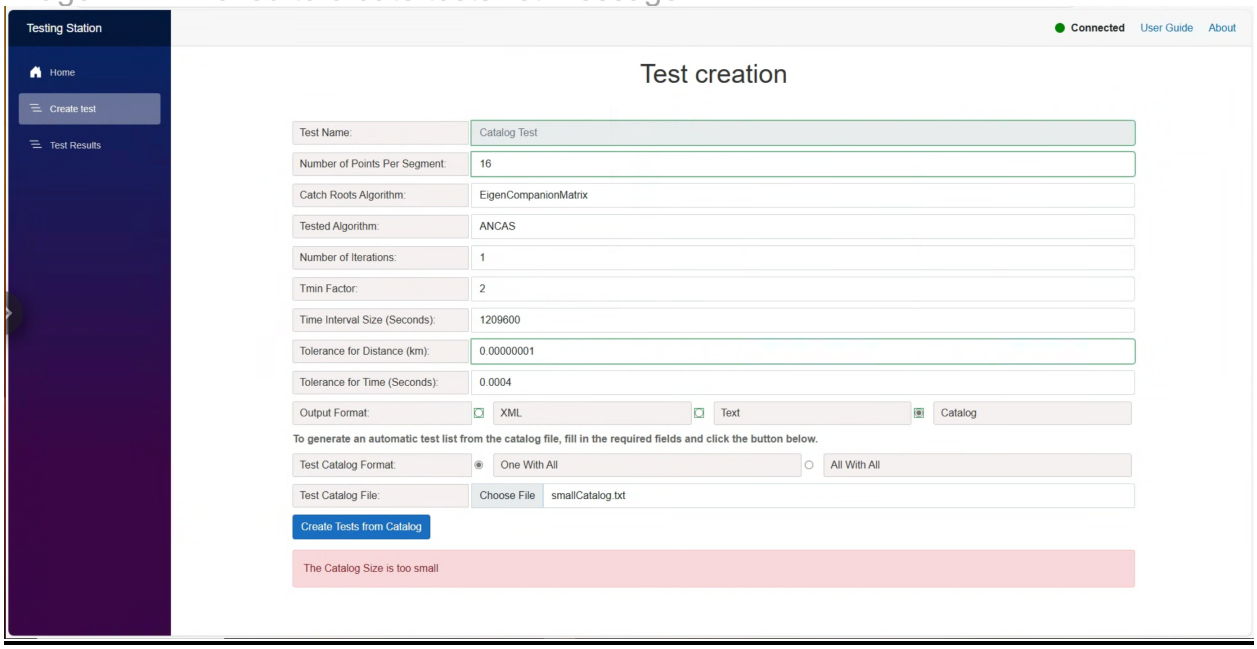
Once the entire catalog has been scanned and all tests have been successfully created and passed, the user will see a success message stating "Test List created successfully".

Image 20: Tests list create success message.



For Catalog format selection, after pressing the "Create Tests List" button, if any of the inputs are incorrect, the tests list will not be created, and an error message will be displayed indicating the issue.

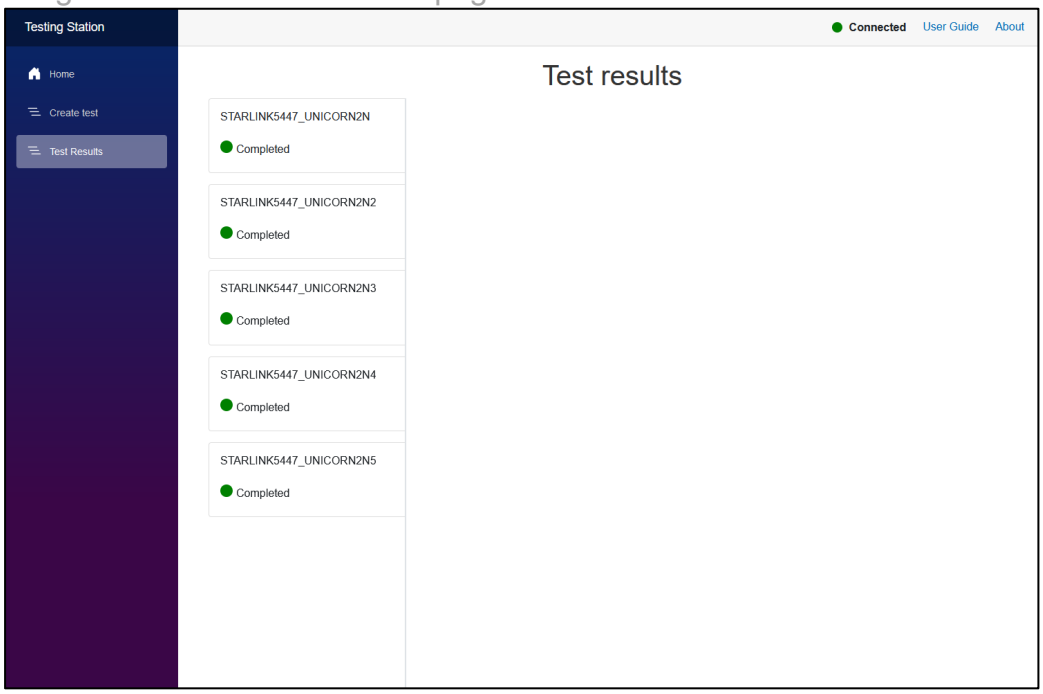
Image 21: Failed to create tests list message.



Watching the tests results

After creating a test go to the Test results page, when opening you can see a list of the existing tests each with the test name and test statues (Completed, In Progress, Failed).

Image 22: The test Results page.



When pressing on one of the tests the test results will be displayed. Almost every value includes tooltip for additional information (formulas, how it was calculated and so on).

Image 23: The test results view.

Test results

Test Name:	STARLINK5447_UNICORN2N5
Test ID:	5
Test Status:	Completed
Number Of Points Per Segment:	16
Tested Algorithm:	SBO_ANCAS
Number Of Iterations:	1
Tmin Factor:	2
Time Interval Size (Sec):	1209600
TOLdKM:	1E-08
TOLtSec:	0.0004
Initial Number Of Points:	6331
Segment Size (Sec):	2863.1326632267655
Format:	Text
TCA (Sec):	577578.3970222491
Distance Of TCA (KM):	0.13874966805570968
Number Of Points The Algorithm Used:	10534
Average Run Time (Micro):	13959
Minimum Run Time (Micro):	13959
Real TCA (Sec):	577578.4064600052
Real Distance Of TCA (KM):	0.011885297650049195
Distance Of TCA Error (KM):	0.12686437040566048
TCA Error (Sec):	0.009437756147235632

8.2. Tested OBC App

To run the Tested OBC App you only need to set the wanted configuration in the INI file and start the application, found in the released Dll versions folder, the application will continually try to connect to the Testing Station, and when connected will wait for a test request message.
The configuration file contains a few important options you will need to consider.

Image 24: Tested OBC App configuration. The Source and Destination ports should be the opposite of the Testing Station App ports.

```
#The type of the Communication channel. options: LocalSimulation , Tcp
CommChannelType = Tcp
#IP for any IP related comm channel
LocalIpAddress = 127.0.0.1
DestIpAddress = 127.0.0.1
#Local port
SourcePort = 8888
#destination port/server port
DestinationPort = 8889
```

We can decide between three operational modes, Tcp, Local Simulation and Udp (only on windows).

The Local Simulation doesn't require the Testing Station and run a simple tests case when activated, can be used for testing the target system.

The Tcp option used for communicating with the Testing Station. For TCP we only care about the destination Ip address and port, both should be updated in the file with the Testing System Ip and port. The Udp option is only supported on windows.

After configuring the INI file it should be place in the same folder as the Tested OBC.

9. Maintenance Guide

The Tested OBC application, the on-board-computer side and the Testing Station application were successfully built and tested on the following platforms:

- Windows 10/11 (x64)
- macOS (Intel-based, Apple Silicon builds are also possible with minor adjustments)
- Raspberry Pi 5 (running a standard 64-bit Linux distribution such as Raspberry Pi OS 64-bit)

The Tested OBC and Testing Station applications rely on:

- C++17 or newer
- CMake (3.15+) on Linux/macOS
- Visual Studio 2022 on Windows (or MSVC from the command line)
- Optionally: Command-line builds for Raspberry Pi 5
- Optionally: CLion for remote development on Raspberry Pi 5

Below are the detailed instructions for building and installing on each environment.

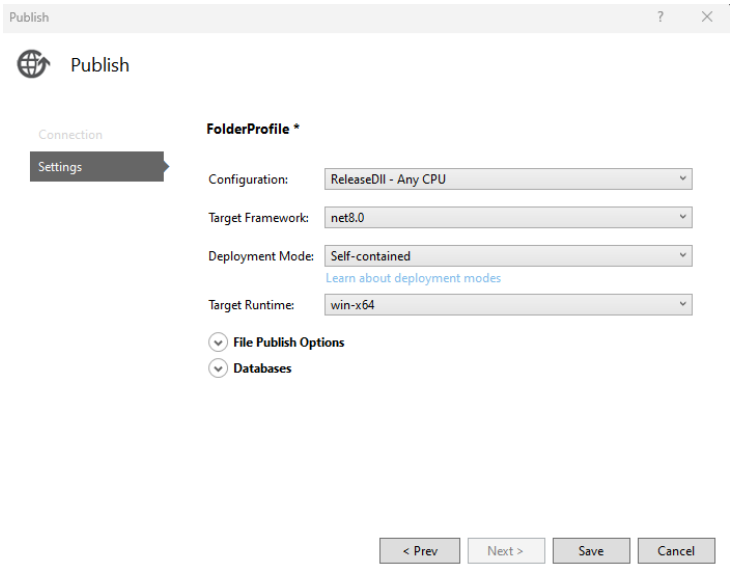
9.1. Testing Station App

9.1.1. Installing and Running the Application

The Testing Station on Windows can be run immediately using the precompiled Blazor executable found in the ReleasedVersions folder. Double-clicking the executable will launch the GUI in your default browser, displaying real-time status indicators for on-board computer connectivity. To rebuild the application, you can use Visual Studio and Publish the blazor project with the existing settings for ReleaseDll – ANY CPU, while choosing the deployment mode as "Self-contained" and Target Runtime as "win-x64". Once compiled, ensure that the TestingStationAppSettings.ini file is in the same folder as the executable so that TCP or UDP ports, IP addresses, and logging parameters are correctly configured.

In addition, if there is no Logger folder present, create a new folder named Logger in the same location where the INI file and the executable are extracted. All log files and results files will be saved in this Logger folder.

Image 25: Visual Studio for Publish the blazor project with the existing settings for ReleaseDll – ANY CPU



9.2. Tested OBC App

9.2.1. Installing and Running the Application

9.2.1.1. Running on Windows

On Windows, the Tested OBC application is distributed as a precompiled .exe located in the ReleasedVersions folder. Simply place it alongside the TestedObcAppSettings.ini configuration file, then double click the .exe or run it from the command line. If the configuration file specifies TCP, the OBC app will attempt to connect to your Testing Station's IP and port, for UDP, ensure ports are open as needed. To compile the application for windows you can either compile a released version via Visual Studio or use the CMakeLists file located at Code\TestedOBCApp\TestedOBCAppCMake` and build a new windows version. After compilation, copy the resulting .exe and .ini files together to run in a single folder. In addition, if there is no Logger folder present in that directory, create a new folder named Logger in the same location where the INI and EXE files are located. All log and result files will be stored in this Logger folder.

9.2.1.2. Running on MacOS

To run the Tested OBC on macOS, clone or copy the TestedOBCApp source folder onto your system. From a terminal, navigate to the root of the **CMake [20]** project and create a build directory (for example, mkdir macOSRelease && cd macOSRelease). Run cmake .. -DCMAKE_BUILD_TYPE=Release, then cmake --build . to compile the application. Alternatively, you can open the project in CLion, select a Release configuration, and let the IDE handle the CMake workflow automatically. When the build completes, place the generated binary (TestedOBCApp) in the same directory as the TestedObcAppSettings.ini file.If there is no Logger folder present in that directory, create a new folder named Logger in the same location where the INI file is located. All log and result files will be stored in this Logger folder. Finally, Launch the app by calling ./TestedOBCApp from a terminal, and it will attempt to connect to the configured Testing Station or perform a local simulation depending on the .ini settings.

9.2.1.3. Running on Linux (Raspberry Pi 5)

On Linux, you can build the Tested OBC application using the provided `build_Linux.sh` script or use the `CMakeLists` file to build the project from the command line. If using the script, mark it executable (chmod +x build_linux.sh) and run ./build_linux.sh to generate a Release build in a dedicated folder, such as LinuxRelease. Alternatively, you can use CLion Remote Development to configure a remote toolchain for the Raspberry Pi 5, build the project, and deploy it directly on the board. Place the TestedObcAppSettings.ini file in the same folder, adjusting IP addresses and ports to match your Testing Station or local simulation preference. If there is no Logger folder present in that directory, create a new folder named Logger in the same location where the INI file is located, all log files and result files will be stored in this Logger folder. Finally, start the application by executing ./TestedOBCApp from the terminal. The console output and logs will confirm successful initialization, communication attempts, or any errors encountered.

Image 26: The build Linux script, simply building with CMake in release.

```
#!/bin/bash
set -e

# Define build directory
BUILD_DIR_LINUX="LinuxRelease"

# Clean and build for Linux
if [ -d "$BUILD_DIR_LINUX" ]; then
    rm -rf "$BUILD_DIR_LINUX"
fi

mkdir "$BUILD_DIR_LINUX"
pushd "$BUILD_DIR_LINUX"

# Configure and build for Linux
cmake .. -DCMAKE_BUILD_TYPE=Release
cmake --build . --config Release

popd
```

9.3. Error Detection and Debugging

9.3.1. Log Files

Our application has a few logging mechanisms, all log outputs should be created in the Logger folder. The first one is an Event Logger, logging system event like starting the system, configuration loaded, receiving messages and starting tests and errors, both the Testing Station and Tested OBC application use this logger. Additionally, both

applications have a Result Logger, saving tests results, but with somewhat different format and implementation. All log files are created when the application starts and their names contain the creation time and date, creating a new file for every system run. The event logger can be used to identify problems with the system, like connection or communication errors.

9.3.2. Local Simulation

The local simulation is a great debugging option, with an asynchronized run and no dependencies and a single threaded application debugging is as easy as it gets. The only problem that can't be identified in the local simulation is a specific communication problem. You can run the local simulation mode in Visual Studio or Clion by making sure that the configuration INI file (found under the main filter in VS) is set to local simulation. For the Tested OBC App there are additional local simulation related parameters in there, like what algorithms to test and more.

9.4. Implementing Changes

We collected a few possible future needed updates and how to implement them easily.

9.4.1. Additional Communication Types

Although we think that the TCP communication should suffice, you might want to run the application on a very specific satellite's computer with limited access to communication and connection, and might need to use a different communication type (serial communication for example). We designed our application to allow adding such changes easily. All you need to do in order to add a new communication type is to implement the ICommChannel interface, and add the new channel to the applications and the creation to the Factories.

The comm managers on both applications doesn't assume any protocol or the correctness of the input and can still work and identify errors even with a less reliable communication.

9.4.2. Additional Algorithms Variations

Adding a new algorithm can be done by adding the algorithm to the algorithms Enum (for test request and so on) and implementing the creation of the object in the Tested OBC App Factory and Test Manager. Adding a new root finding algorithm for **CATCH [5]** is the same, updating the relevant Enum and factory, while implementing the root finder interface.

9.4.3. Additional Test Creation Options

Adding additional options for the user can be quite easy, for example setting a set of tests at once instead of manually, or running over a set of inputs and creating a test for each one. All we need to do is add an appropriate function to the Lab and Lab Wrapper classes and create the appropriate GUI.

9.4.4. Testing Different Algorithms Types

You can use our application as a base and add different tests, for example testing other required algorithms for satellites. To do so might be more complicated, we recommend adding a test type to the test request message header allowing you to parse the data required for your test by adding code instead of changing the full application.

9.4.5. Additional GUI Features

For creating additional features and make them available in the GUI level, for example adding an option for generating graph form tests results, you need to implement the feature in your code then add it to the Lab and Lab Wrapper to make it available in the

GUI environment, there you simply import it in the LabInterop class and its available to use withing the GUI managers.

10. REFERENCES:

[1] Alfano, S. (1994). Determining Satellite Close Approaches-Part II. ADVANCES IN THE ASTRONAUTICAL SCIENCES, 87, 233 -233.

[2] C++ Coding Conventions

[3] CelesTrak

[4] CelesTrak Current Data

[5] Denenberg, E. (2020). Satellite closest approach calculation through chebyshev proxy polynomials. Acta Astronautica, 170, 55-65.

[6] Denenberg, E., & Gurfil, P. (2016). Improvements to time of closest approach calculation. Journal of Guidance, Control, and Dynamics, 39(9), 1967-1979.

[7] Feasibility Analysis and Performance Testing of Collision Detection Algorithms for Satellites Project's GitHub repository

[8] Freethink article about space debris

[9] GOMSpace's website

[10] Google test framework

[11] INIH- INI File Reader Library.

[12] ISIS On Board Computer

[13] ISILAUNCH's website

[14] Kessler, D. J., Johnson, N. L., Liou, J. C., & Matney, M. (2010). The kessler syndrome: implications to future space operations. Advances in the Astronautical Sciences, 137(8), 2010.

[15] NASA on orbital debris

[16] NASA Procedural Requirements for Limiting Orbital Debris

[17] OBC's Datasheets, GitHub repository

[18] Our GitHub repository

[19] Raspberry Pi 5

[20] Repository for CMake in github

[21] Satellite Orbital Conjunction Reports Assessing Threatening Encounters in Space

[22] SatSearch's website

[23] Semantic Versioning

[24] SGP4 and implementation for C++

[25] SGP4 python library

[26] SQLITE 3

[27] The Armadillo Library

[28] The Eigen Library

[29] TS-7553 OBC's data

[30] UN Space Debris Mitigation Guidelines

[31] Vallado, D. A., Crawford, P., Hujsak, R., & Kelso, T. S. (2006, August). Revisiting spacetrack report# 3. AIAA-2006-6753. AIAA Astrodynamics Specialists Conference and Exhibit. Keystone, CO: American Institute of Aeronautics and Astronautics.

11. AI Prompts

During the research phase of our project, we used AI tools such as ChatGPT to validate our technology choices, improve grammar, and explore new feature ideas for our application. The following are examples of the prompts we used and how the AI's feedback influenced our research and development decisions.

[32] Write me short literature review on the Propagator SGP4. Remember to include references.
https://chatgpt.com/share/66e9f25b-5580-8006-873e-73a4bb0a32d4

[33] Create for me a two page literature review about [2] Denenberg, E. (2020). Satellite closest approach calculation through chebyshev proxy polynomials. Acta Astronautica, 170, 55-65. Remember to include references.
https://chatgpt.com/share/66e9f307-3c88-8006-8ab4-ed7fb3743687

[34] Create for me a two page literature review about Alfano, S. (1994). Determining Satellite Close Approaches-Part II. ADVANCES IN THE ASTRONAUTICAL SCIENCES, 87, 233-233. Remember to include references.
https://chatgpt.com/share/66e9f437-28d4-8006-985f-e0e22ec792b6

[35] Create for me a two page literature review about [3] Denenberg, E., & Gurfil, P. (2016). Improvements to time of closest approach calculation. Journal of Guidance, Control, and Dynamics, 39(9), 1967-1979. Remember to include references.
https://chatgpt.com/share/66e9f4ad-e3ec-8006-afd4-0fbe2e85b792

[36] Sort the reference articles in alphabetical order according to the first author of the article.
https://chatgpt.com/c/67d14a47-cbb0-8006-9847-23de161bdf27