Capstone Project Phase B

FEASIBILITY ANALYSIS AND PERFORMANCE TESTING OF COLLISION DETECTION ALGORITHMS FOR SATELLITES

23-2-D-17

Supervisor: Mr. Ilya Zeldner

Hallel Weil 316484153 Hallel.Weil@e.braude.ac.il

Omer Shamir 314763970 Omer.Shamir@e.braude.ac.il

Abstract

The project we present is based on algorithms for finding the minimal distance between two objects in space, and the time of occurrence, which will have to work on satellite on-board computer (OBC). The project has two goals, the first goal is to implement the algorithms and deliver them ready to integrate and use for future testing and projects, and the second goal is to create a system for testing the algorithms performance on a satellite OBC or an emulated environment and conduct a feasibility study.
The project is based on Dr. Elad Denenberg's research papers that introduced the algorithms **[1][3]**.

Keywords

Satellite, Space debris, Minimal distance, Approximations algorithms, Feasibility test, Collision detection, Algorithm testing, Distance approximation, Orbiting objects, Satellite on-board computer.

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**[4]** and the shift to cluster of smaller satellites instead of a single big one **[1]** 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 ***on-board computer (OBC)***. These algorithms have not been tested and we need to prove the task feasibility, to implement the algorithms and to show the calculation time and memory requirements in an environment with limited calculation power and memory simulating an actual satellite on-board computer. We are working with Dr. Elad Denenberg, who created the **Conjunction Assessment Through Chebyshev Polynomials (CATCH)** algorithm **[1]** and the **SBO-ANCAS** algorithm **[3]**. Dr. Elad is working on creating an autonomous satellite and as part of his work he need our help. An autonomous satellite needs to calculate the possible collisions by itself and for doing that a fast algorithm for finding the TCA is needed, faster calculations are possible using approximations like the algorithms CATCH**[1]**, SBO-ANCAS**[3]** and **Alfano\Negron Close Approach Software (ANCAS) [2]**. These algorithms were never tested on an actual satellite on-board computer, and proved fitting to run on an autonomous satellite and this is we come in.

2. General Description

2.1. Background

2.1.1. The Algorithms

In this project we implemented and tested three algorithms, all three can be used to calculate the TCA. The following is a description of the algorithms.

2.1.1.1. ANCAS

The first algorithm, ANCAS **[2]** 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 [2])

Input: $p[n], t[n]$
Output: $t_{TCA} + r_{TCA}$
$r_{TCA} = inf$
$t_{TCA} = inf$
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 **[[2],Eq.2/5]** with the points $p[1-4]$
  Fit cubic polynomial to $\dot{f}(t)$ according to **[[2],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 **[[2],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 **[2]**.
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.

2.1.1.2. SBO-ANCAS

The second algorithm, SBO-ANCAS **[3]** is based on the ANCAS 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 **[3]**)

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 **[[2],Eq.2/5]** with the points $p[1-4]$
Fit cubic polynomial to $\dot{f}(t)$ according to **[[2],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 **[[2],Eq.6]**
if $r < r_{min}$ **:**
$r_{min} = r$
$t_m = t*$
end
end
Sample r_d and $\dot{r}_d$ at t_{min} using a propagator
$\mathrm{t_{new}} = \mathrm{t_{new}} \setminus \left(t_j \,|\, \max\left(|t_j - t_m|\right)\right) \cup t_m$

While $| r_{min} - \| r_d(t_m) \| | > TOL_d$ OR $\max\left(|t_j - t_m|\right) > TOL_t$

$\mathrm{r_{tca}} = \| \mathrm{r_d(t_m)} \|$

$\mathrm{t_{tca}} = \mathrm{t_m}$

end

The cubic polynomial coefficients calculations described in article **[2].**
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 $\mathrm{t_m}$ we use the propagator to sample the location and velocity vectors at $\mathrm{t_m}$, 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 have an additional loop in each iteration and sampling points with the propagator is an expensive operation.

2.1.1.3. CATCH

The third algorithm, CATCH **[1]**, uses **Chebyshev Proxy Polynomial (CPP)[1]** 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 **[[1], 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 **[[1], 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 **[[2],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 **[[1],Eq.18]** and the eigen values of this matrix are the polynomial roots. Using the roots, we found and creating CPP for $p(t)_{f_x}, p(t)_{fy}, p(t)_{fz}$ 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**[[1],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.

2.1.2. The 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) [10]**, 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 **[5][6]**. 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.
We used the propagator for 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 algorithm. SBO-ANCAS needs to sample new points as part of the algorithms so a propagator is needed.

2.2. Goals

In this project we have three main goals.

2.2.1. Implementing the Algorithms

Our first goal is to implement the algorithms themselves and doing so while considering the environment the algorithm will have to work on. Until now SBO-ANCAS and CATCH were only implemented in MATLAB as part of the initial article and testing [1][3]. To work on a satellite OBC well the algorithms need to run efficiently on various systems and computer boards. We started implementing the algorithms in the first part of the project as a feasibility proof for our project and completed the implementation in this part.

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 dedicated system or emulator 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 using our system. We needed to run our system with different types of inputs, different types of algorithms and different parameters for each algorithm.

2.3. Users

For our system we have two types of users, the first is DR. Elad and his associates wanting to test the algorithms against an emulator for now and a real satellite's OBC in the future. The second type of user is us, wanting to run a feasibility test as part of our project, and any possible follow up team that will take part in the ongoing effort of creating and proving the feasibility of the autonomous satellites. (there are already other teams working in their projects toward this goal, that might need to use our system or expend it).

3. The solution

3.1. Algorithms Analysis

3.1.1. Algorithms Runtime Complexity

3.1.1.1. ANCAS Time Complexity

In ANCAS [2], 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 **[[2],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 [2] 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 **[[2]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 [3], 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[1] 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 **[[1],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[1]. 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 $\left(N_{deg}+1\right)\left(N_{deg}+1\right)$ 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: $O(n+1) = O(n)$

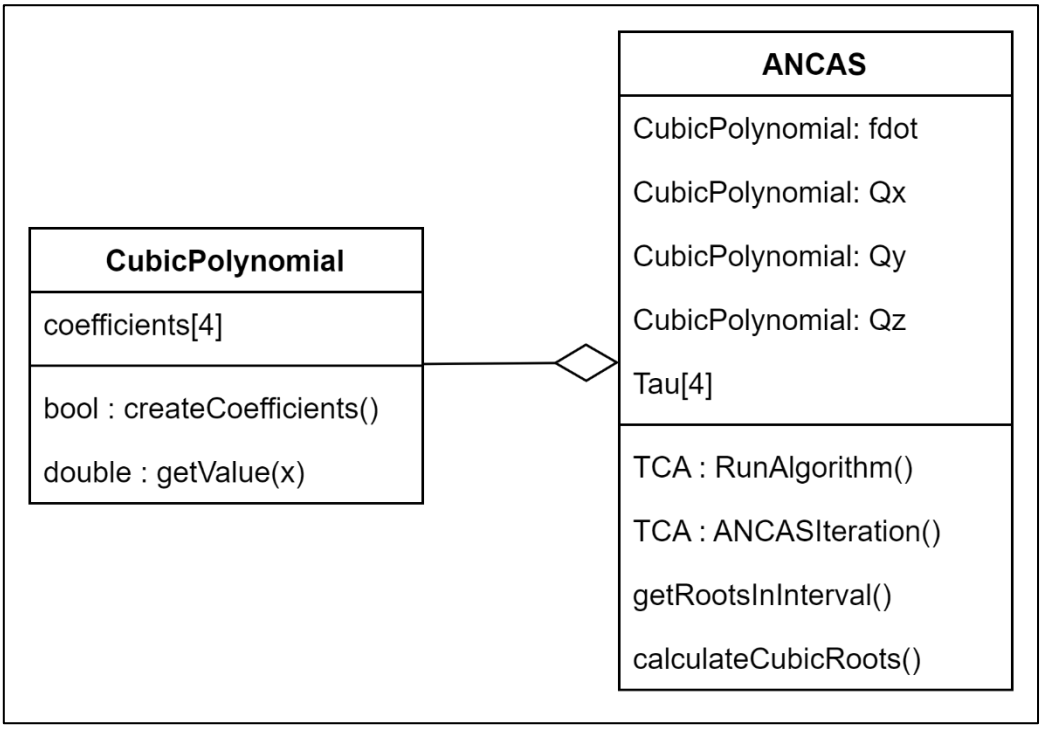
*For SBO-ANCAS, we assume that the propogator uses a constant amount of memory, but this may not always be the case.

3.2. Algorithms implementation

3.2.1. ANCAS Implementation

ANCAS 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. 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.

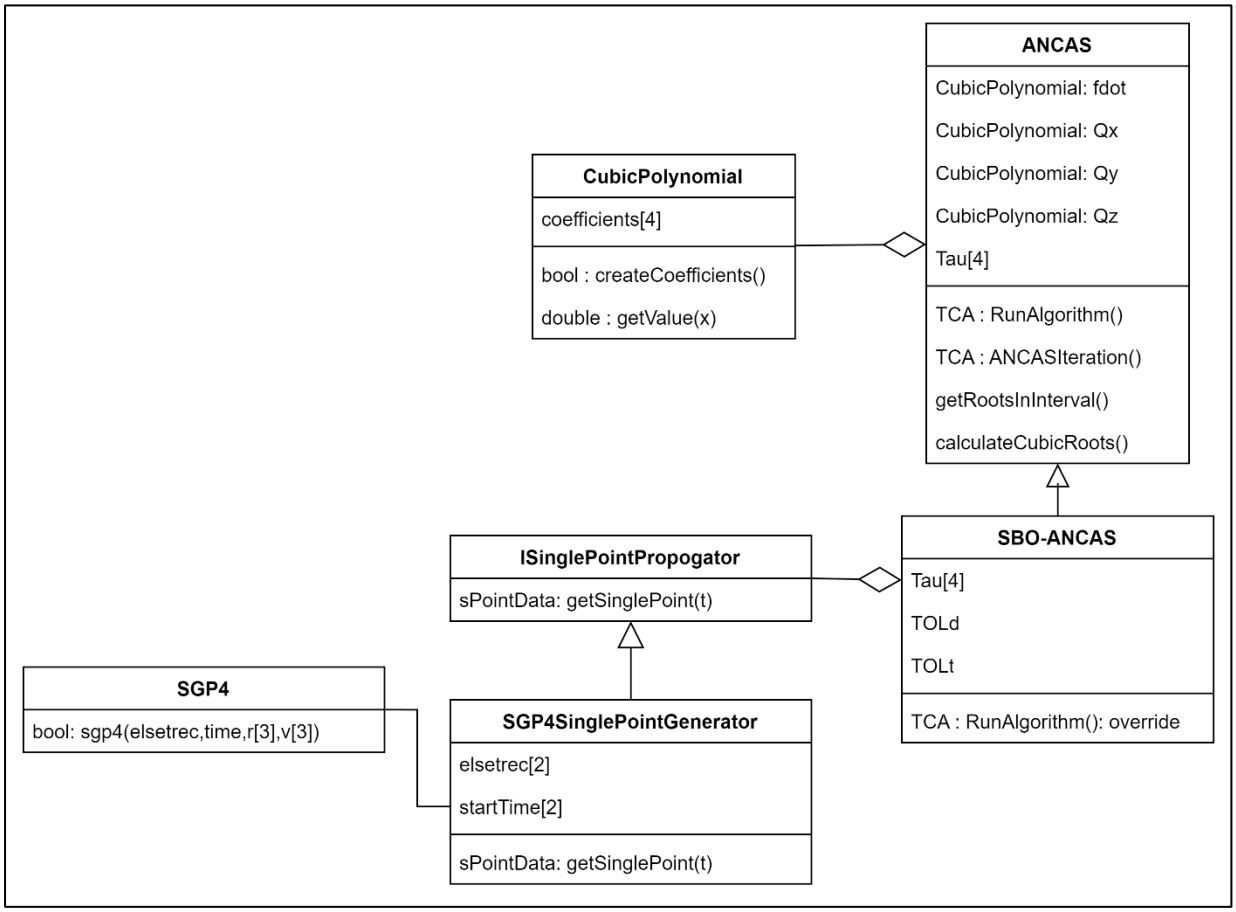
Diagram 1: Class diagram for ANCAS. Including a function for calculating the roots of a cubic polynomial used by a function for getting the roots in the interval 0,1. The function for ANCAS Iteration return the found minimum and time, used for a single ANCAS iteration.



3.2.2. SBO-ANCAS Implementation

SBO-ANCAS acts similar to ANCAS 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 and used it for our testing. Additionally, SBO-ANCAS needed the tolerances in both time and distance as input.

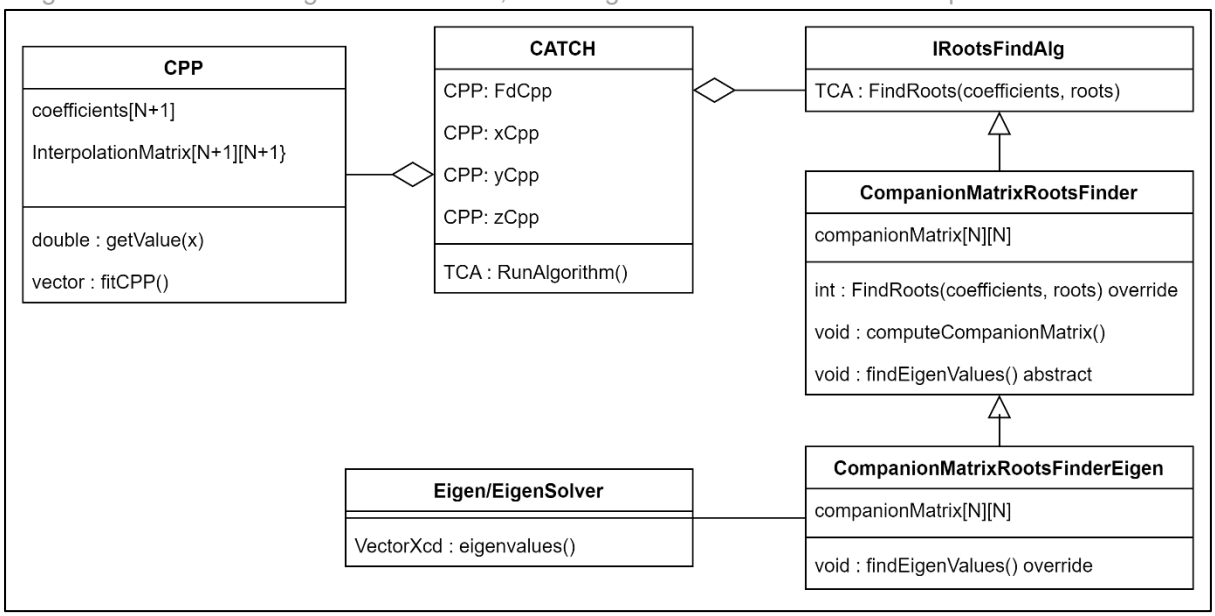
Diagram 2: Class diagram for SBO-ANCAS. Including the Propagator interface and implementation and the tolerances.



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 [1], and we tried two libraries for finding the Eigenvalues of the Companion Matrix. We implemented using Eigen [12] and Armadillo [13]. Unfortunately, the Armadillo library is quite heavy (while using the library the code uses around 400MB) and its too much for the satellite's OBC so we removed the Armadillo implementation.

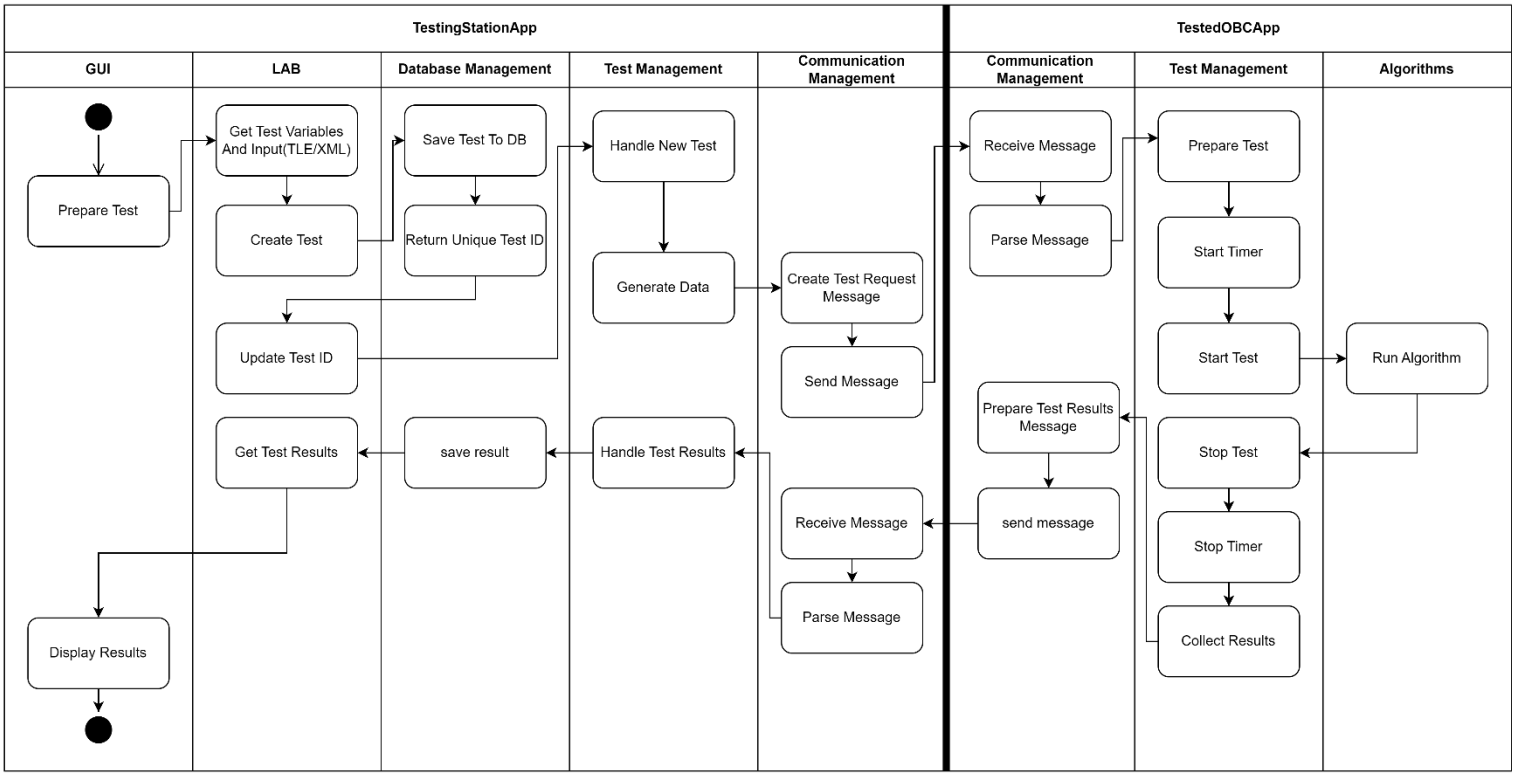
Diagram 3: Class diagram for CATCH, including Rootfinder interface and implementation.



3.3. The Testing system

Looking at the full application, the process of creating and running a test starts from the user, 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 App via the communication channels. The Tested OBC App take the input, 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 4: Top view activity diagram for running a test, with the full 2 Apps system

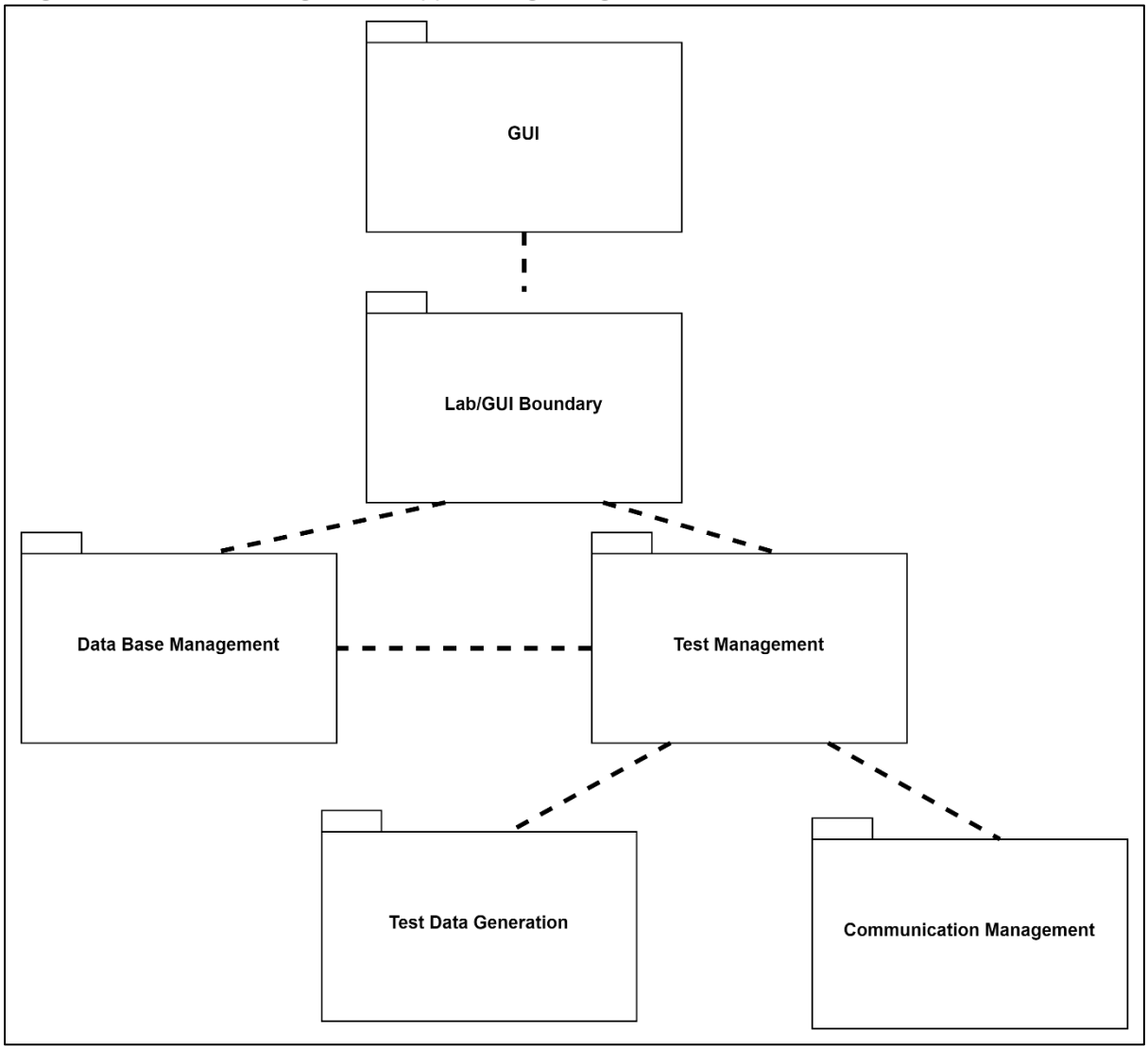


3.3.1. Testing Station App

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

3.3.1.1. App Structure and Architecture

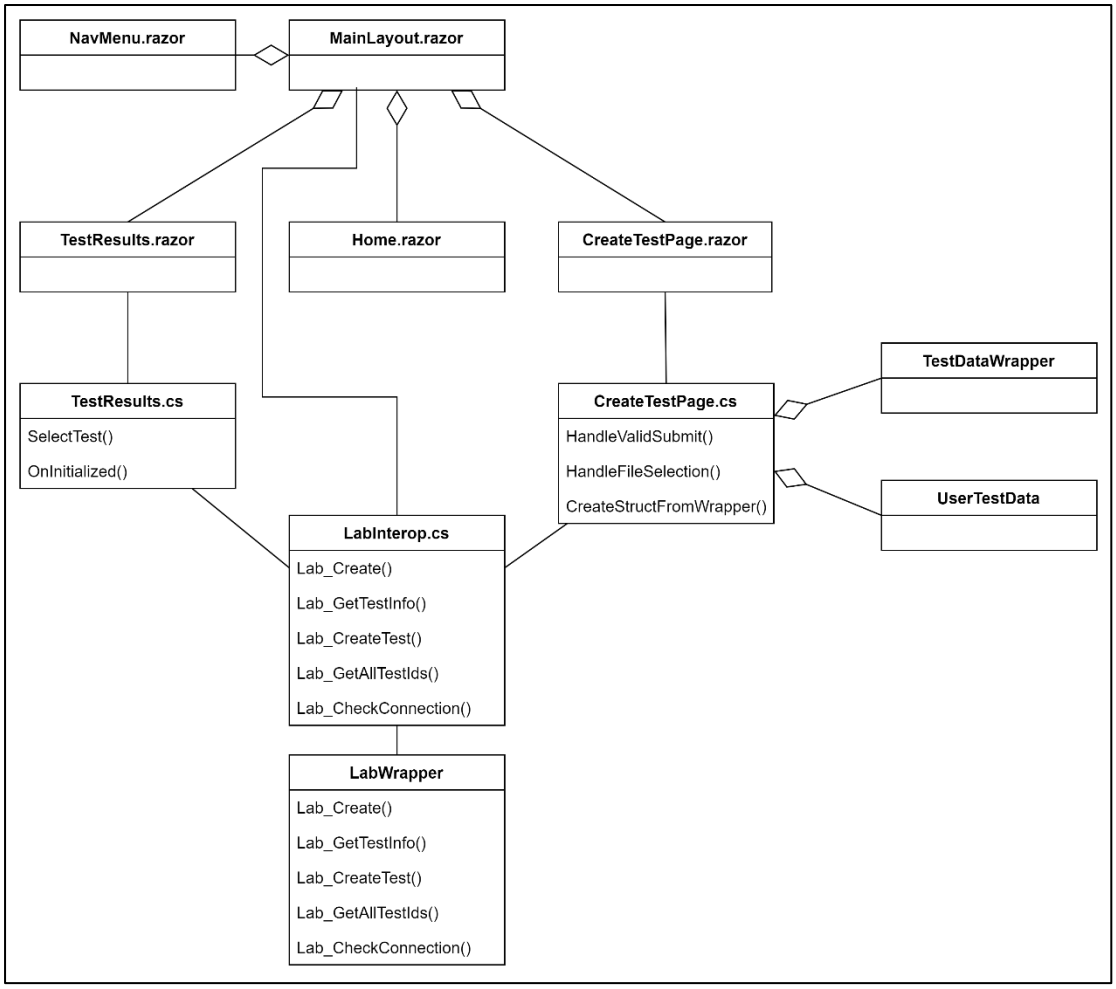
Diagram 5: The Testing Station App Package Diagram



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 test. 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 in 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).

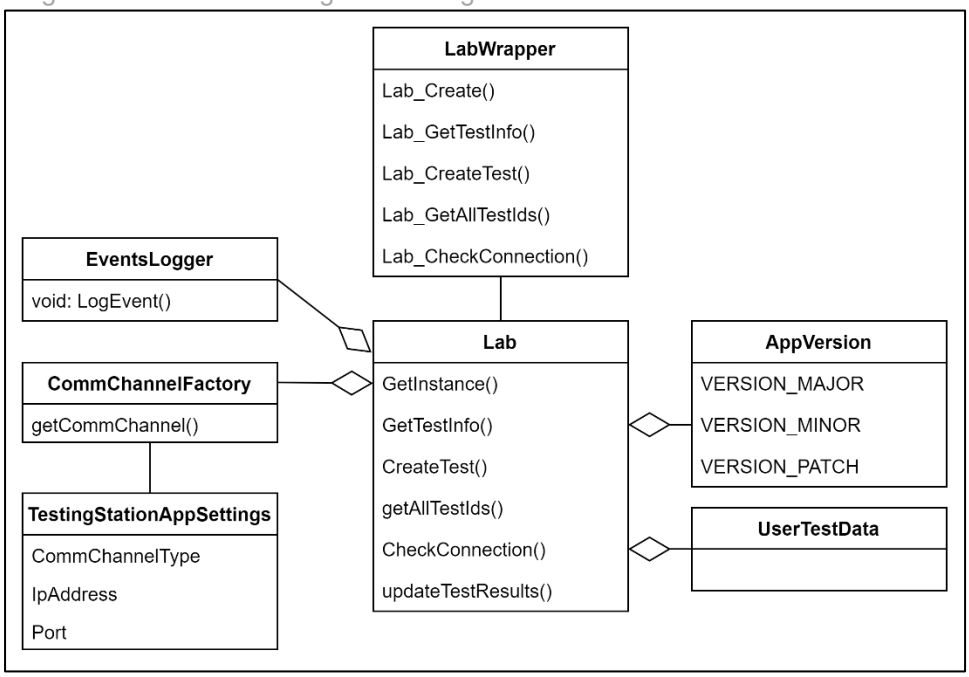
Diagram 6: GUI Package class diagram.



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.

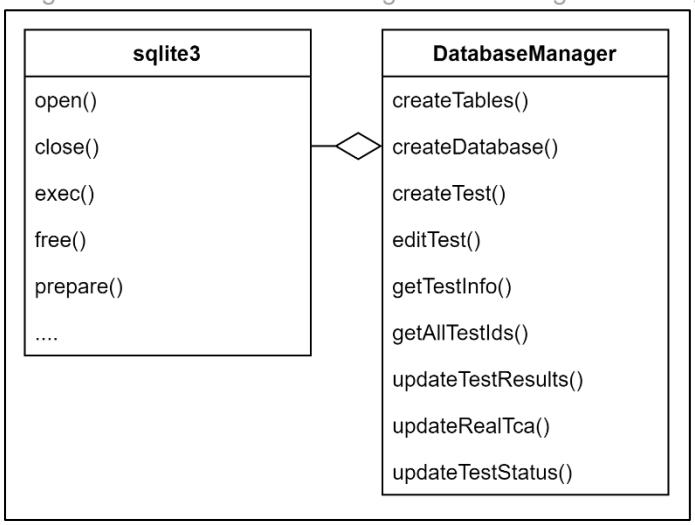
Diagram 7: Lab Package class diagram.



Database Management Package

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

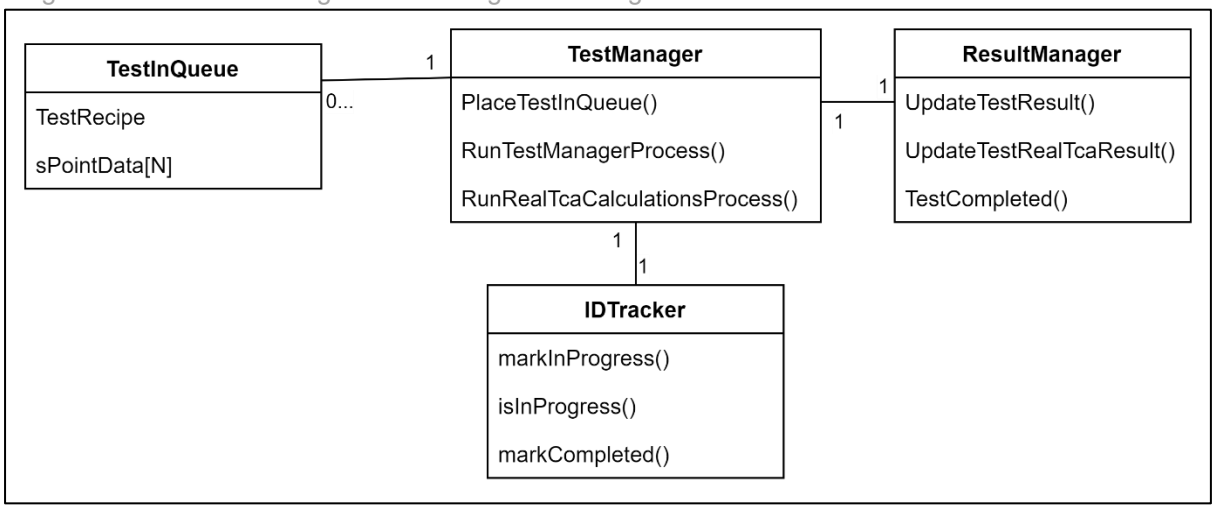
Diagram 8: Database Management Package class diagram



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).

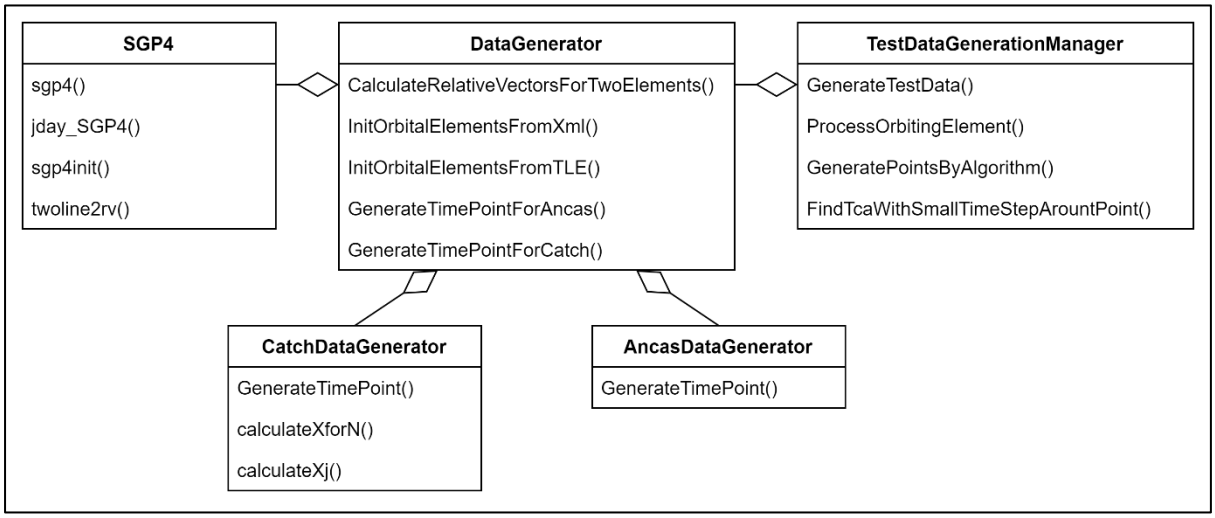
Diagram 9: Test Management Package class diagram.



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.

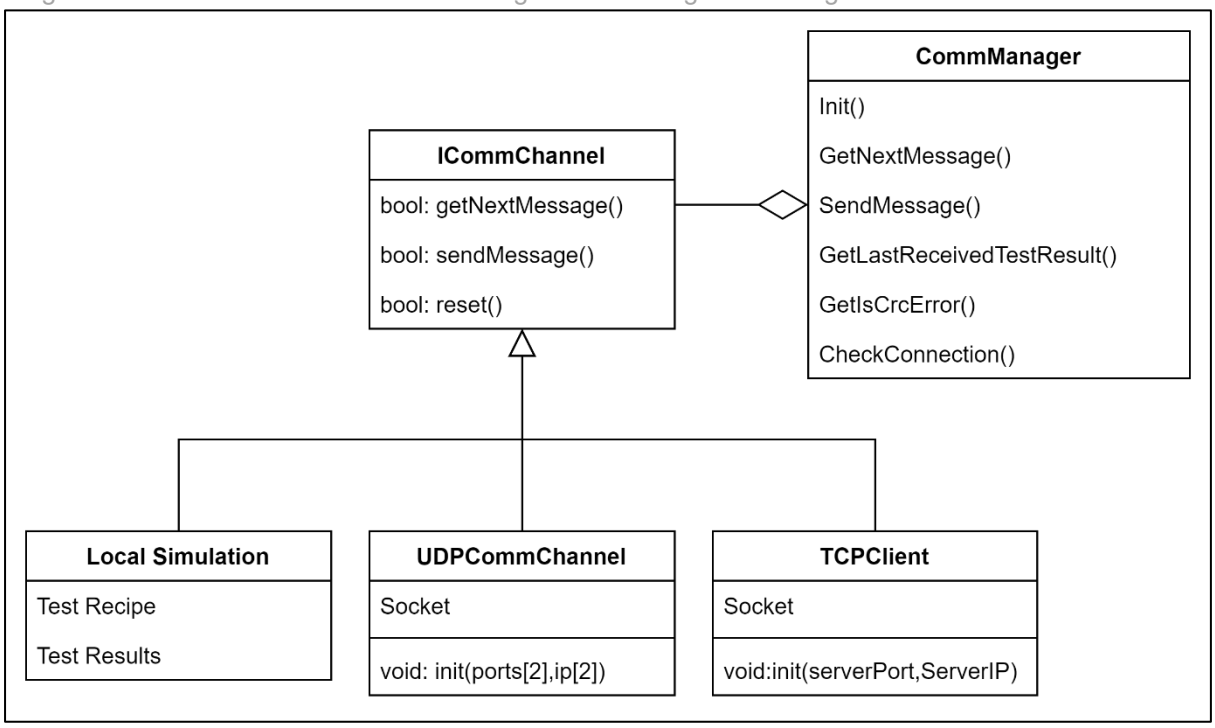
Diagram 10: Test Data Generation Package class diagram.



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.

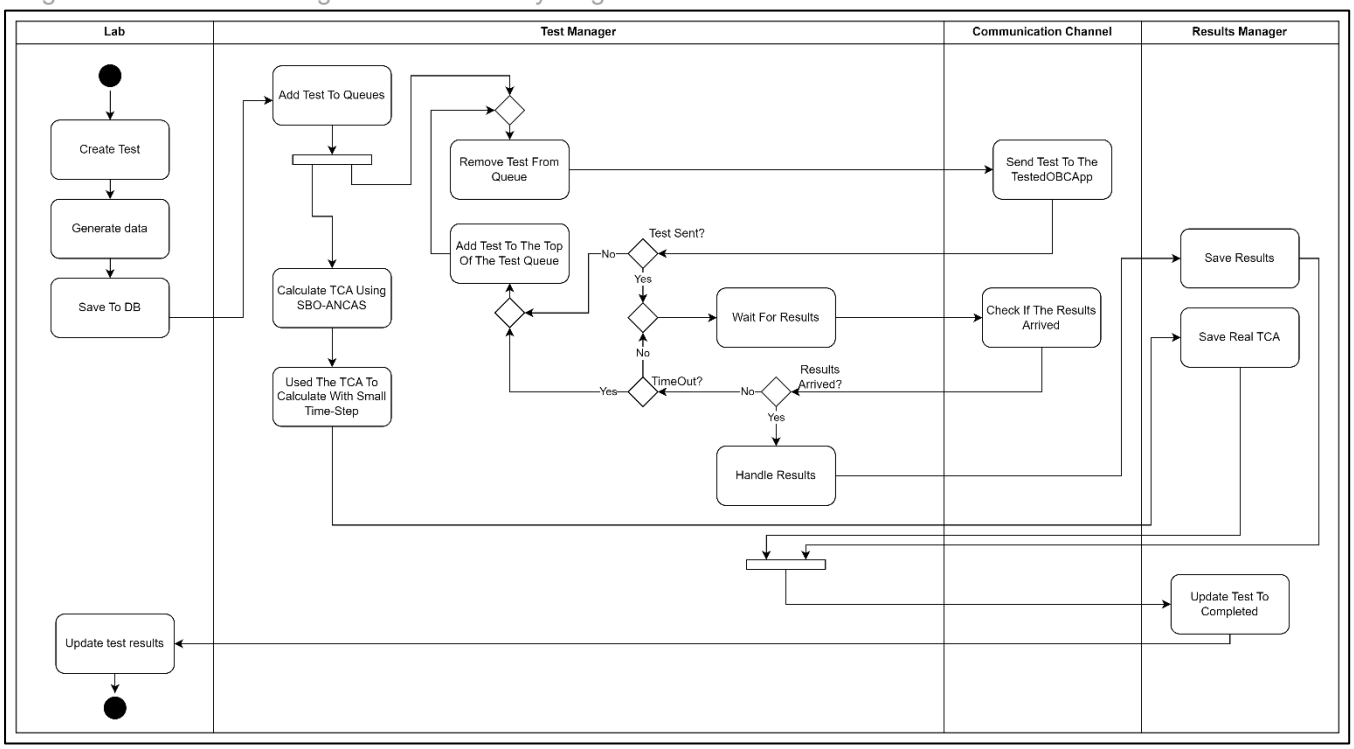
Diagram 11: The Communication Management Package class diagram.



3.3.1.2. Running A Test - Activity Diagram

After the Lab create the test, it 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 sate to Completed and display the results.

Diagram 12: Test Manager Process activity diagram

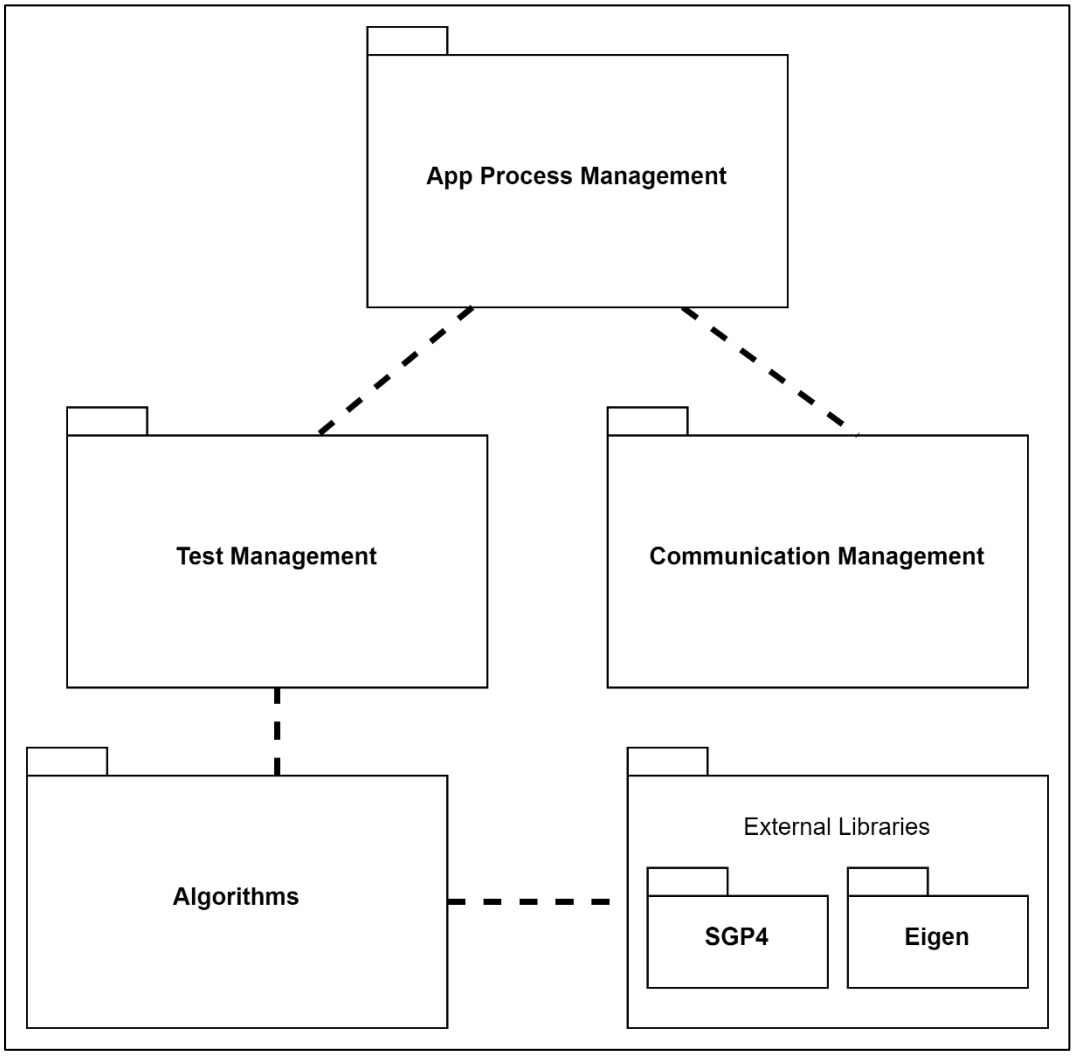


3.3.2. Tested OBC App

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.

3.3.2.1. App Structure and Architecture

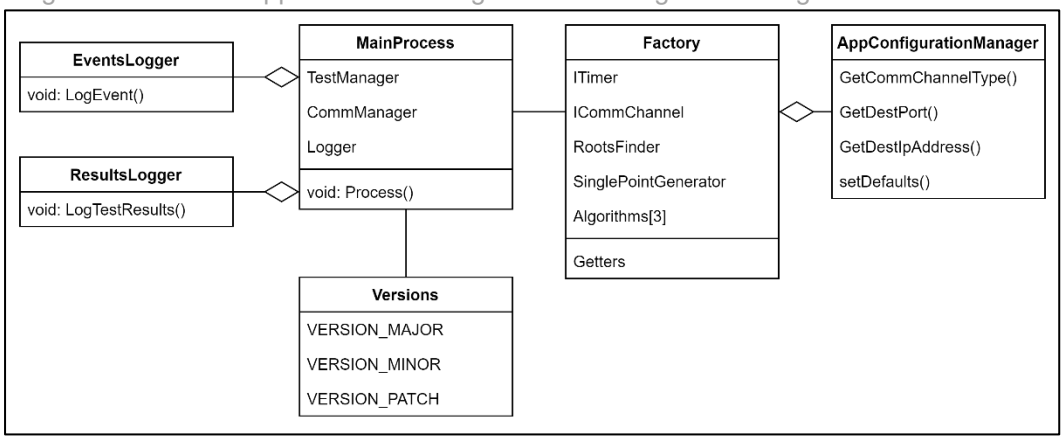
Diagram 13: The Tested OBC App Package Diagram



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.

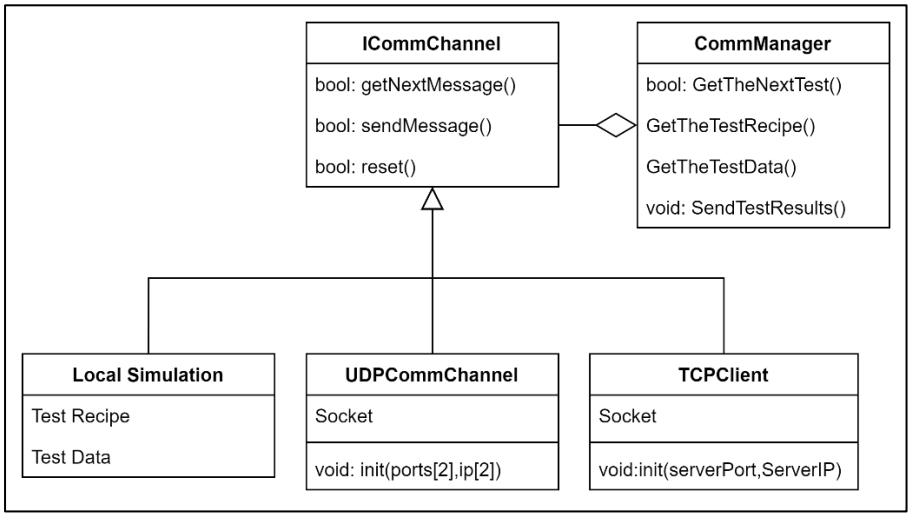
Diagram 14: The App Process Managements Package class diagram.



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.

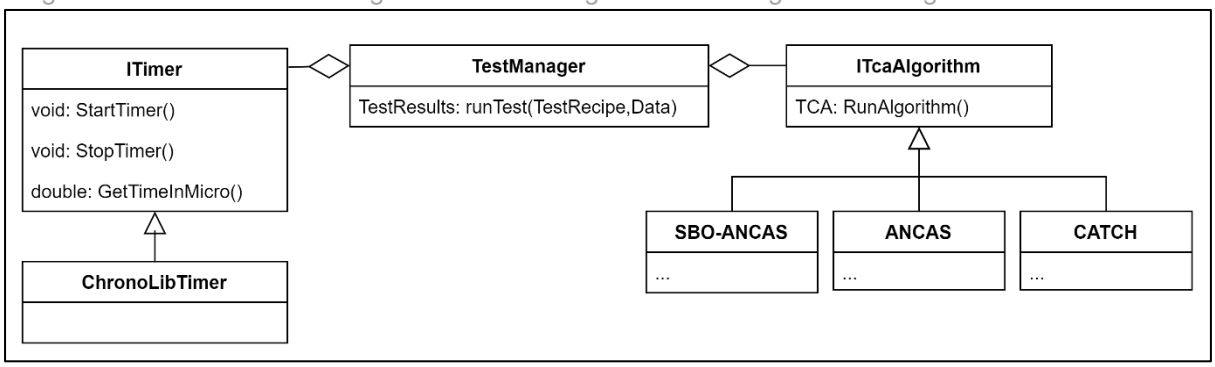
Diagram 15: The Communication Management Package class diagram.



Test Management and the Algorithms Packages

The Test Management package handles the test itself, after receiving a Test Recipe and a set of points as 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.

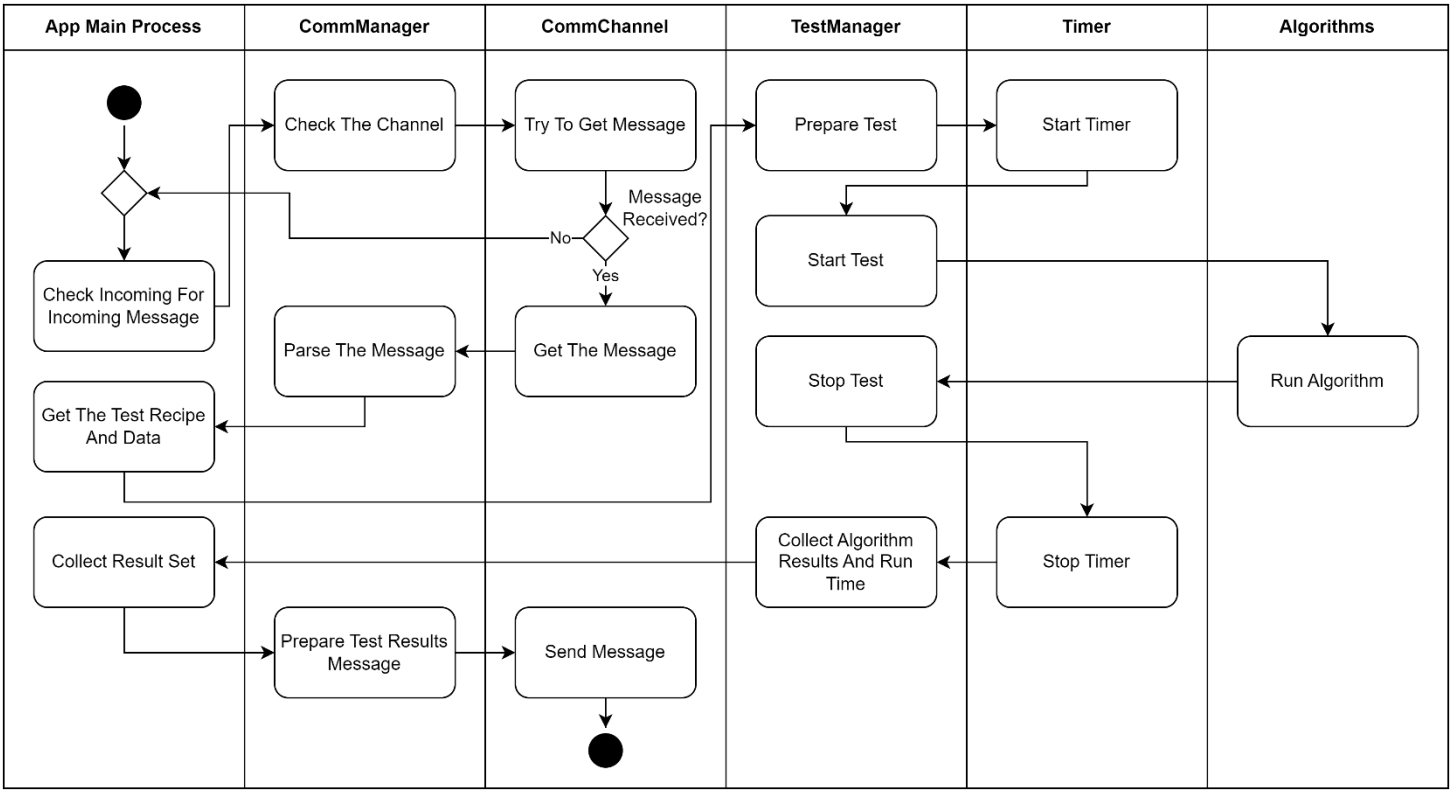
Diagram 16: The Test Management and the Algorithms Packages class diagram.



3.3.2.2. Running A Test - Activity Diagram

The Tested OBC App is repeatedly checking for incoming messages until a message arrived, then parsing the message and using the Test Recipe preparing and running the test, creating the result set and sending it back to the Testing Station App.

Diagram 17: Activity diagram of running a test on the Tested OBC App.



3.3.3. Communication Protocol and Channels

3.3.3.1. 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

Table 1: 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 algorithms input, array with the data in each point in time, each entry containing the time at the point and 4 3d vectors, the location and velocity of 2 objects in this point in time.	can vary

Tested OBC to Testing Station – Test Results Message

Table 2: 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.	can vary

3.3.3.2. 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.

3.3.3.3. 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.

3.3.4. Feasibility Testing Environments

We created the Tested OBC App with the capabilities to work on different systems, with the intention of eventually running the application on an actual satellite's OBC, unfortunately we still didn't manage to get access to one but we wanted to leave the application with the ability to do so. In the meantime, we tested our app on 3 different systems.

3.3.4.1. Windows

The easiest option is to run our app on the same computer as the Testing Station App, working with TCP over the Local Host. This option is more fitting for testing and debugging the application in real time or even running tests that doesn't require the actual system (for example exploring the relation between the root finding algorithm we use and the size of the error). Another option we implemented is running the application in a Local Simulation mode. In this option we only use the Tested OBC App and the Local Simulation create the Test Request Message and simulate receiving the message via the communication channel. This option is really convenient for test a prepare test case and testing the system logic without relying on an actual communication and synchronization with the second app. Additionally it can be used to run tests on a system in an asynchronized manner.

3.3.4.2. Linux on Raspberry Pi 4

Using a Raspberry Pi 4 we had to run tests on a more limited system, with actual communication and a different operating system. We used it to test the cross-platform communication between the Testing Station App and the Tested OBC App.

Image 1: Our Raspberry Pi 4 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-A72
    Model:               3
    Thread(s) per core:  1
    Core(s) per cluster: 4
    Socket(s):           -
    Cluster(s):          1
    Stepping:            r0p3
    CPU(s) scaling MHz:  33%
    CPU max MHz:         1800.0000
    CPU min MHz:         600.0000
    BogoMIPS:            108.00
    Flags:               fp asimd evtstrm crc32 cpuid
```

3.3.4.3. Emulator using gem5

The last option we tested is running the Tested OBC App on and emulator. Finding an emulator that can emulate the CPU frequencies we needed reliably wasn't simple until we come across gem5 [14]. gem5 is a community led project, providing a modular platform for creating and researching computer systems. We can use gem5 to run our application with a simulated architecture, CPU type Memory type and size and more.

Image 2: gem5 configuration, we used the TimingSimpleCPU to get accurate timing, with CPU clock frequency of 400MHz, instruction and data caches of 32KB each and a SimpleMemory as the memory type. We used the X86 se script provided with gem5.

```
CPU: X86/TimingSimpleCPU
cpu-clock=400MHz
caches:
l1d_size=32kB
l1i_size=32kB
mem-type=SimpleMemory
```

We created a script for running our app with similar properties to the OBC [15] we wanted to test. The only problems with gem5 are that its design to run on Linux and that it can be slow.

Image 3: Example of gem5 stats, simulating 0.02 seconds took 131.5 real seconds.

```
---------- Begin Simulation Statistics ----------
simSeconds                                   0.020757                       # Number of seconds simulated (Second)
hostSeconds                                    131.50                       # Real time elapsed on the host (Second)
```

Image 4: Another example of gem5 stats, simulating 372 seconds took almost 12 real hours.

```
---------- Begin Simulation Statistics ----------
simSeconds                                 372.163036                       # Number of seconds simulated (Second)
hostSeconds                                  41985.61                       # Real time elapsed on the host (Second)
```

4. Research and Development process

4.1. Algorithms Analysis and Implementation

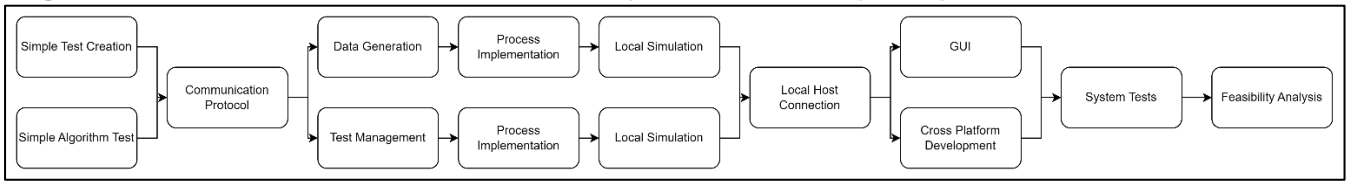
We started our process with the algorithms analysis and implementation, we started with ANCAS and CATCH in the previous part of our project. Using our implementation of the algorithms as a feasibility proof for our project. We continue with implementing SBO-ANCAS and analyzing the complexity and run time of the algorithms.

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 18: 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 or faulty inputs.

Image 5: 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

We used Visual Studio 22 for our development, supporting both the C++ development and C# development for the Blazor framework.

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 [12] for finding the eigenvalues of CATCH's companion matrix, part of the root finding algorithm CATCH 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 [12].

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 [16].

5.3.3. SQLITE3

We used SQLITE3 implementation for our local database, can be found here [17]. 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 [6]. 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 [11] 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 [] 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, the gem5 emulator and WSL virtual machine.

5.4.4. Semantic Versioning

We used the guidelines found in here [20] for versioning our applications.

5.4.5. Coding Conventions

We tried to based our coding on the convention found here [19], 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 mysterious “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 (I’m looking at you 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 and 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 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.

7. Results and conclusion

7.1. Feasibility Analysis and Test Results

7.1.1. SBO-ANCAS Errors Analysis

We notice that sometimes the error we get from SBO-ANCAS TCA and distance is bigger than the tolerances we defined. To test the errors, we ran two SBO-ANCAS variations and ANCAS over a data set and checked the results.

The Data Set:

We used a catalog of 9727 objects, we got the TLE of all the active satellites from Celestrak [8] as of 29/04/24 12:05:33UTC. The algorithms were tested on pairs made from the first satellite in the list with every other satellite. Giving us 9726 different tests. We used 32 point per minimal revolution and tested over a week with the following tolerances:

$TOLd = 0.00000001\ TOLt = 0.0001$

The Algorithms:

We used 2 variations of SBO-ANCAS, the first is the regular algorithm and in the second after finding the t_{new} set in each iteration we save the first and last points and created 2 additional points to get a set of evenly spaced point in the time interval.

A Test Case:

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

Table 3: Test data for the satellites 1 and 5 in the catalog - CALSPHERE 1 and CALSPHERE 4A

Test ID	Algorithm	Distance[KM]	TCA[Sec]	Real Distance[KM]	Real TCA[Sec]	TCA Error [Sec]	Distance Error [KM]	No Roots	Tolerance Reached	Number of outer iterations
10	ANCAS	127.186963	551415.4845	127.1656473	551415.4912	0.006735	0.021316	581	-	959
11	SBO ANCAS	127.1656612	551415.4845	127.1656473	551415.4912	0.006728	1.38E-05	887	72	959
12	SBO ANCAS ES	127.1656473	551415.4912	127.1656473	551415.4912	1.37E-05	1.10E-10	581	378	959

The No Roots column is the number of iterations ended when no roots were found in range for a cubic polynomial, the Tolerance Reached column is the number of iterations ended because we reached both the tolerance in distance and in time. Like you can see in the example, we have an initial number of iteration where no roots can be found at the first inner iteration – 581, the additional $887 - 581 = 301$ iteration SBO-ANCAS failed to find root before reaching the tolerance, meaning we only reached the tolerance in 72 iterations. Unlike SBO-ANCAS in the variation we got to the desired tolerance every iteration we could, except the ones when no roots were found from the start. It seems like the cubic polynomial approximation perform better when the points are evenly spaces, and by running the algorithm we can get extremely uneven distribution of points from time to time and it can lead to slightly worse performance.

Analyzing The Data:

We checked how many tests didn't reach the desired tolerances and got the following data:

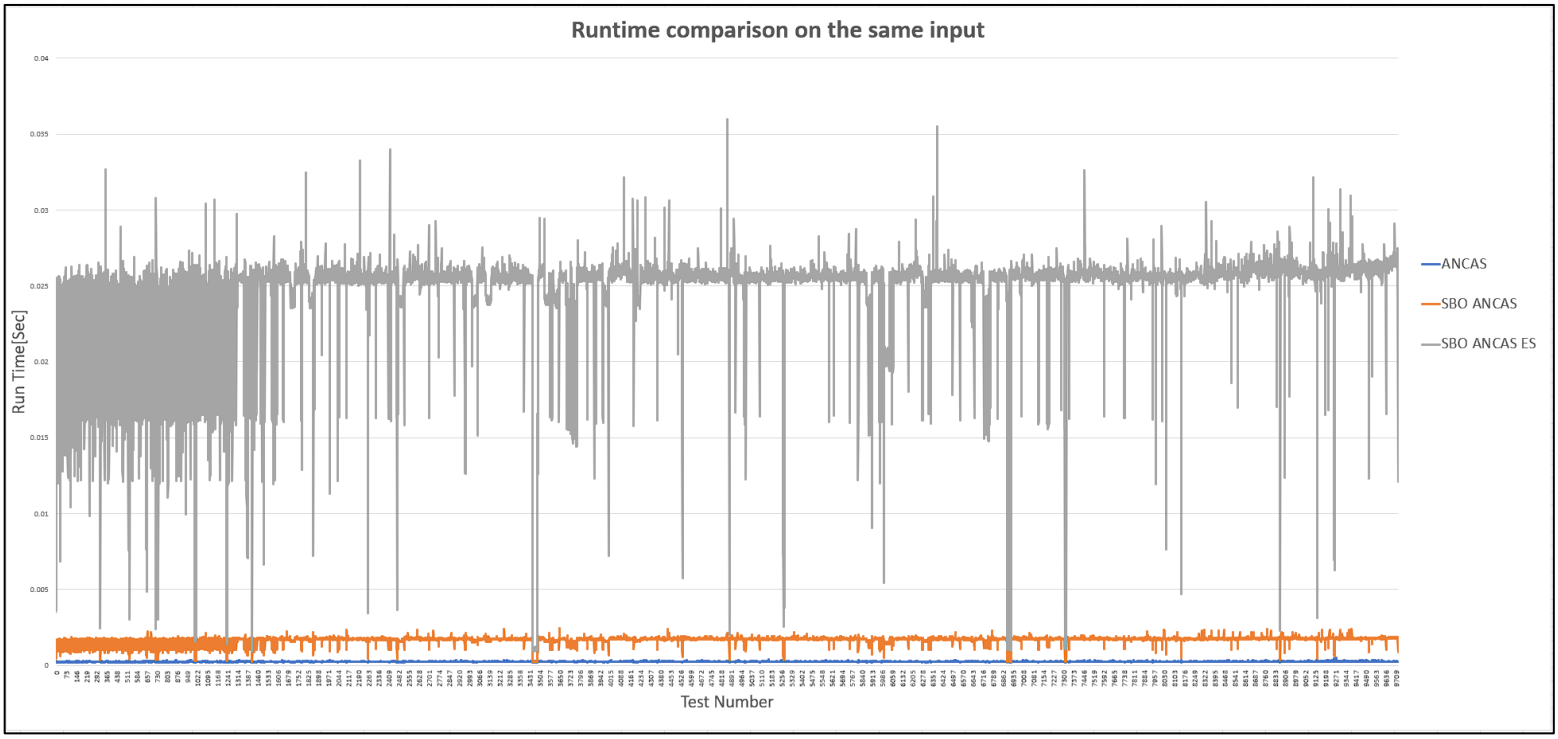
Table 4: Total fails (not within the one of the tolerances) and average values.

	Fails	Total Tests	Percentage	Avg Time Error on Fail	Avg Distance Error on Fail
SBO ANCAS ES	2067	9636	0.214508	0.001970586	3.6774E-06
SBO ANCAS	2692	9712	0.277183	0.023549618	0.008712228

*Note: because we calculated the real TCA and real distance by running SBO ANCAS and then running with a smalltime-step (smaller than the tolerance) 1 second around the TCA we ignored data with time error bigger than 1 second.

Even with the SBO ANCAS variation there are still high percentage of tests ended with error bigger than the tolerance, we think it's a problem with the cubic polynomial, unable to find roots in range when working with small or unevenly spaced values.

Graph 1: Run time comparison, in each test (on the X axis) the algorithms use the exact same input. SBO ANCAS ES (grey) is the SBO ANCAS variation that uses equally spaced points.



Conclusion:

Even though we got better results with the SBO ANCAS variation, the number of points the algorithm use can be up to 10 times the number of points SBO ANCAS used and the run- time can be even worse. So practically SBO ANCAS is still a better option, running significantly faster and getting accurate results most of the time, and still getting only a small error when not.

7.1.2. Run Time Comparison

The Data Set:

We used a catalog of 9727 objects, we got the TLE of all the active satellites from Celestrak [8] as of 29/04/24 12:05:33UTC. The algorithms were tested on pairs made from the first satellite in the list with every other satellite. Due to the simulation limitations, working relatively slow (the hosting timing, the simulated run time is not that slow) we were unable to process the full catalog and completed tests with only around 19.6% of the catalog, resulting in 5709 different tests. We used 32 points per minimal revolution, with CATCH degree set to 15 and time interval set to a week, with the following SBO-ANCAS tolerances:

$TOLd = 0.00000001\ TOLt = 0.0001$

A Test Case:

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

Table 5: Test data for the satellites 1 and 5 in the catalog - CALSPHERE 1 and CALSPHERE 4A

Test ID	Algorithm	Runtiem[Sec]	Distance[KM]	TCA[Sec]	Real Distance[KM]	Real TCA[Sec]	TCA Error [Sec]	Distance Error [KM]
10	ANCAS	0.02526	127.186962953645	551415.484504414	127.165647312815	551415.491239301	0.006735	0.021316
11	SBO ANCAS	0.175659	127.165661156987	551415.484511341	127.165647312815	551415.491239301	0.006728	1.38E-05
12	CATCH	0.623122	127.165647281093	551415.491225964	127.165647312815	551415.491239301	7.66E-06	3.16E-08

Analyzing The Data:

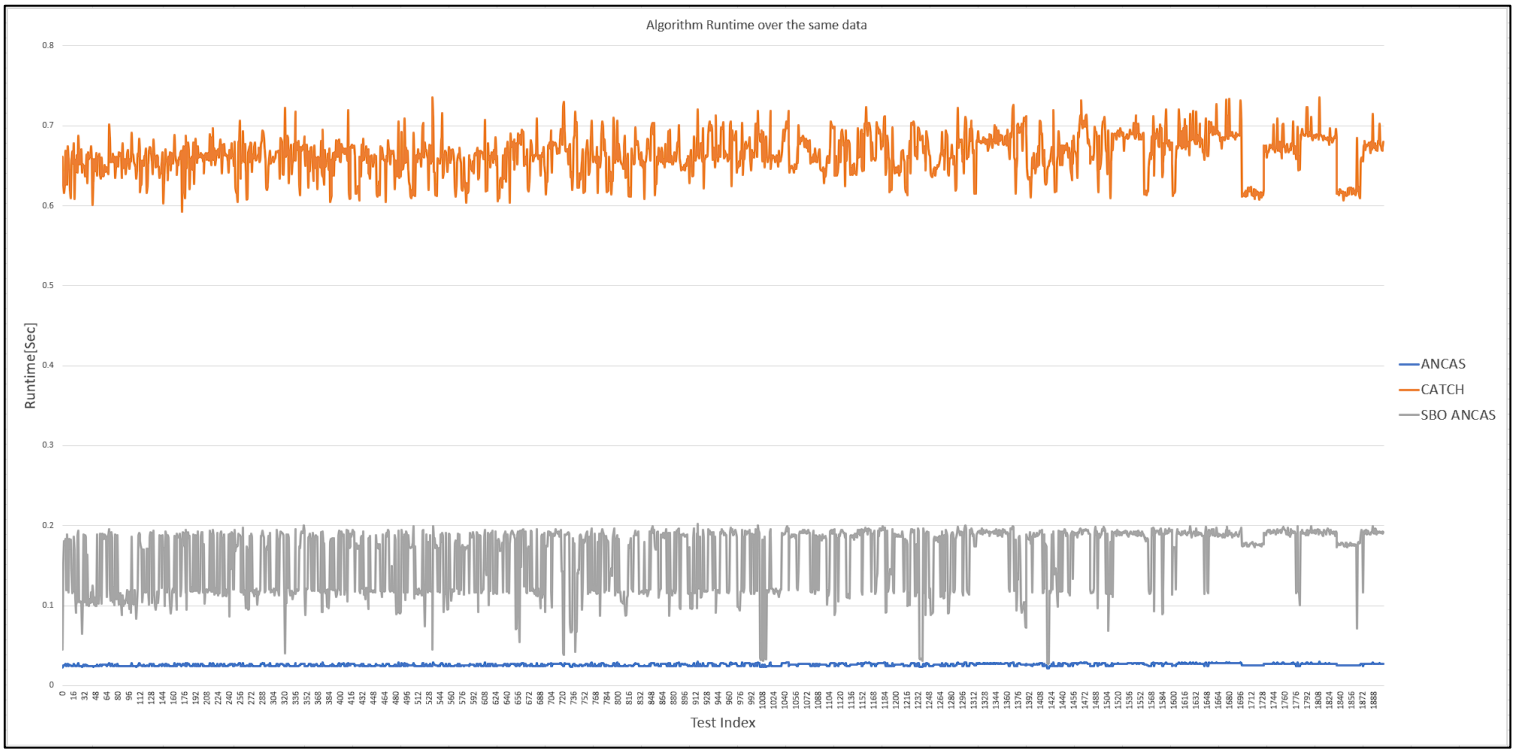
Like you can see in the provided graphs, the CATCH demonstrated the slowest performance among the tested algorithms, with average runtime of 0.664 seconds per test, which is much higher compare to SBO ANCAS and ANCAS, challenging our initial expectation. Interestedly, SBO ANCAS wasn't the slowest despite its algorithmic complexity, with the additional inner loops and sampling of new points. It performed better then CATCH suggesting that CATCH higher runtime is due to the relatively high cost of finding the roots of a high degree polynomial, comparing to the almost neglectable computation cost of finding a cubic polynomial roots. And as expected, ANCAS was the fastest algorithm, however its speed comes at a compromise in terms accuracy.

Table 6: Average run time and error (Distance)

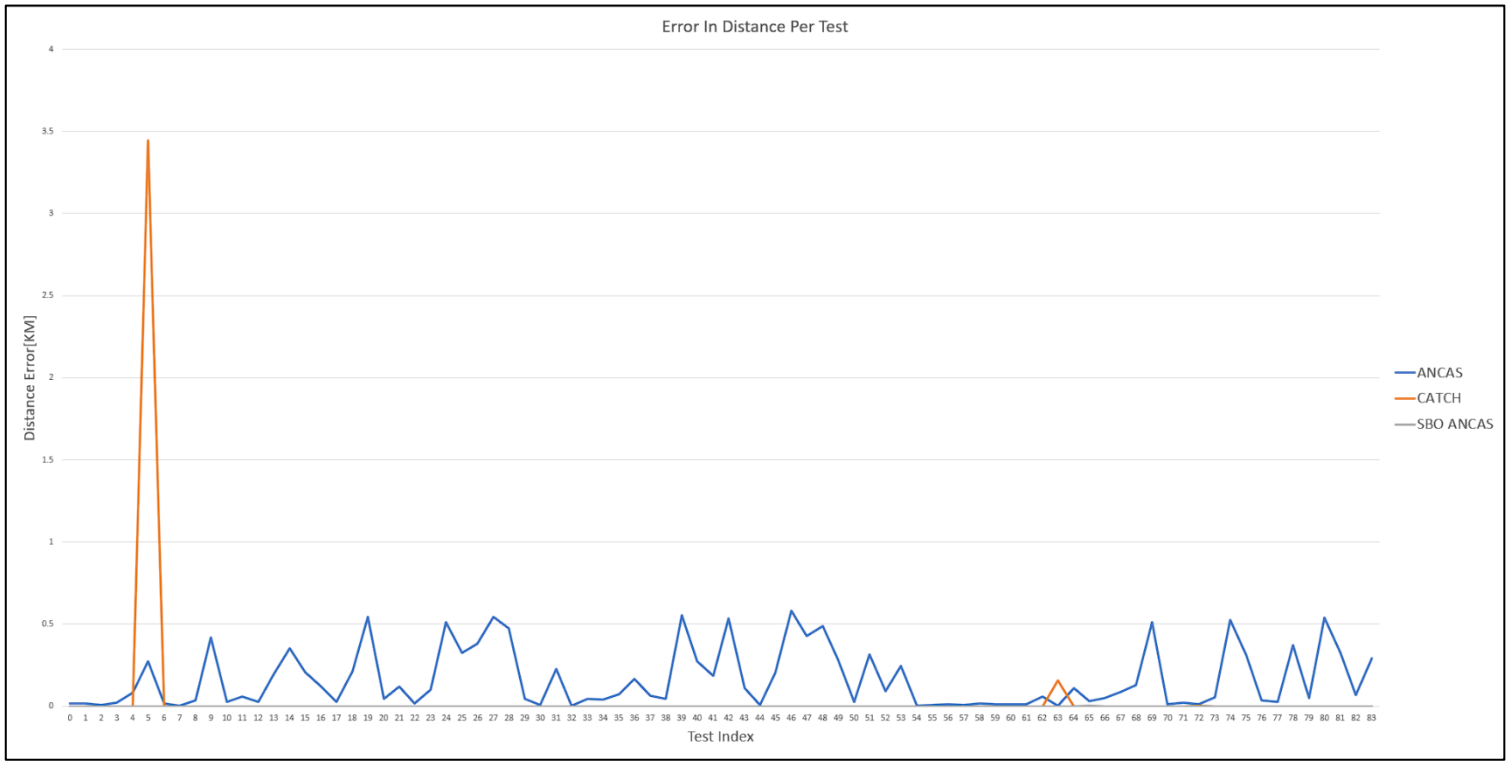
Algorithm	Average Runtime[Sec]	Average Distance Error[KM]	Average #Points	Test With Distance Error > 0.00001[KM]	Test With Distance Error > 0.100[KM]
ANCAS	0.026057568	0.222462893	3019	1902	783
SBO ANCAS	0.159591541	0.007744792	3019	91	8
CATCH	0.663982872	0.022729184	3019	41	12

Unfortunately analyzing the errors in time is not very practical because sometimes a small error in calculating the distance, the minimum, can cause a big error in time. For example, if there are 2 minimal points in the data even a small error in calculating the distance can make the algorithm choose the other point and we can get an error point in time. For this reason, we decided to mostly focus on the errors in distance.

Graph 2: Algorithms runtime, the X axis is the test index, in each test the algorithm working on the same input.



Graph 3: Algorithms distance error [KM] per test, only on the first 83 tests.



Conclusion:

In conclusion, after evaluating the three algorithms in a simulated environment, we arrived to the following observations:

ANCAS

As anticipated, ANCAS was extremely fast, completing the calculation rapidly. However, this speed comes at a cost of limited accuracy, meaning it can be problematic to use for actual collision detection by itself.

SBO ANCAS

SBO ANCAS performed well, delivering accurate results, though not always within the desire tolerance but still with the smallest errors between the tested algorithms. However, it's important to notice that SBO ANCAS can come with its own set of limitation, because of the algorithm nature, running up to a given tolerance, some set of input points may trigger extended or near-infinite loops, and possibly creating problems in the highly scheduled environment of satellite systems. Implementing and upper bound to the number of inner iterations could mitigate this risk, ensuring more predictable and reliable performance.

CATCH

with the current implementation for the roots-finding component, CATCH have the highest computation time among the tested algorithms. To enhance performance, we might need to consider other solutions for finding the roots. For example, a specific implementation for calculating eigenvalues of the companion matrix, tailored to the characteristics of the known matrix, could prove more efficient. Alternatively, utilizing the low expected number of roots and known interval bound, a different root finding algorithm might be used.

7.1.3. Inputs and Algorithms Runtime Correlation

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

7.1.3.1. Change in the number of points

The Data Set:

We used a catalog of 9727 objects, we got the TLE of all the active satellites from Celestrak [8] as of 29/04/24 12:05:33UTC. The algorithms were tested on pairs made from the first satellite in the list with every other satellite. We only tested with the first 30 satellites. We tested with CATCH degree set to 15 and time interval set to a week, with the following SBO-ANCAS tolerances:

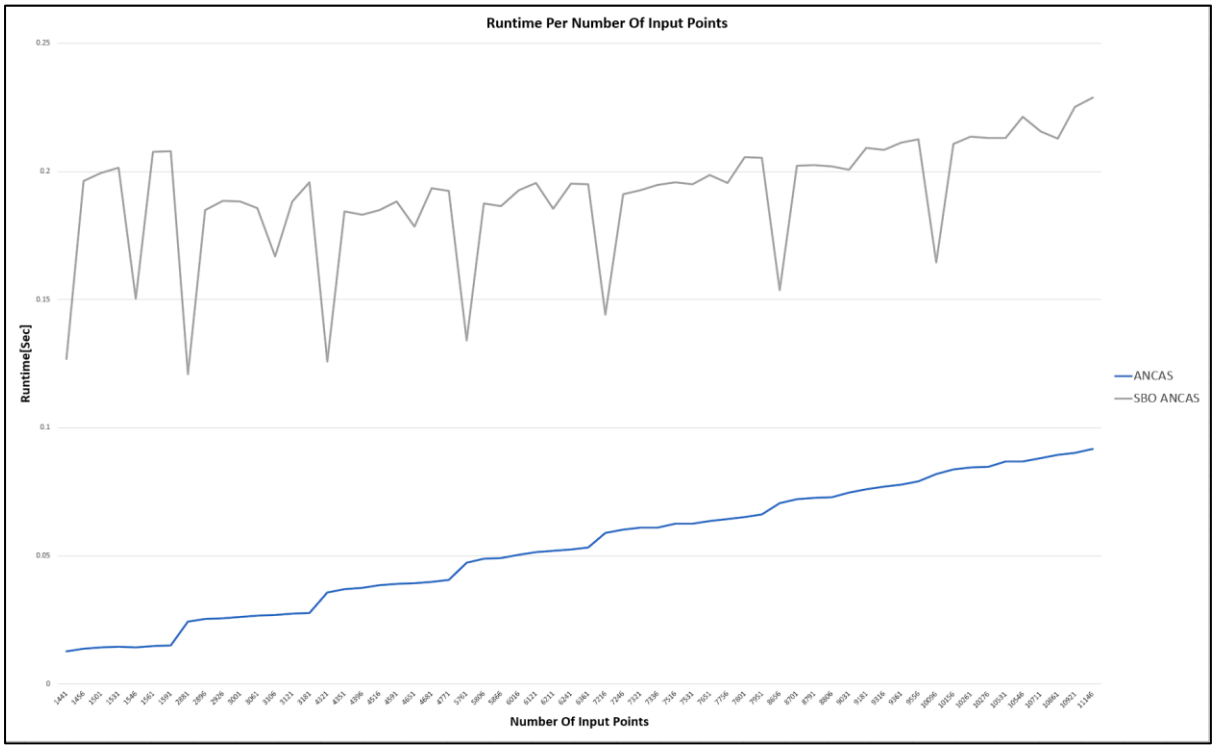
$TOLd = 0.00000001$ $TOLt = 0.0001$

To get different number of points over the same time interval we only changed TminFactor, resulting in more point per minimal revolution, and more points in total, going over 7 different values of TminFactor and getting a total of 609 tests.

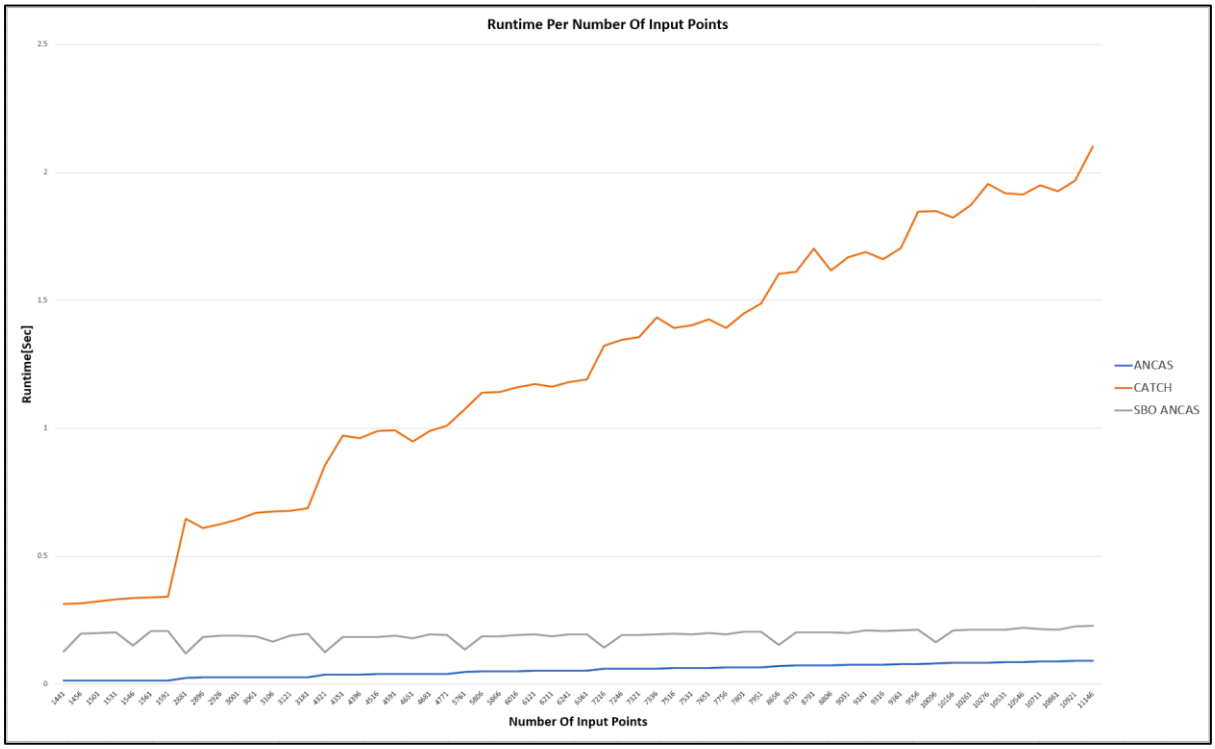
Analyzing The Data:

Although, due to time limitations (creating this data set alone took over 12 hours), we don't have a lot of data to compare, we can still identify some correlations between the size of the input, the number of points, to the run time. The easiest to identify is ANCAS, being linear to the number of points like we expected. SBO ANCAS seems to take more time for more points but the runtime is not that consistence so other factor might have a part in that. And lastly CATCH also seems to be linear to the number of input points but steeper, rising faster with the change in input size.

Graph 4: SBO ANCAS and ANCAS runtime over the change in the number of input points.



Graph 5: The algorithms runtime over the change in the number of input points.



7.1.3.2. Change in the time interval

The Data Set:

We used a catalog of 9727 objects, we got the TLE of all the active satellites from Celestrak [8] as of 29/04/24 12:05:33UTC. The algorithms were tested on pairs made from the first satellite in the list with every other satellite. We only tested with the first 30 pairs. We tested with CATCH degree set to 15 and with the following SBO-ANCAS tolerances:

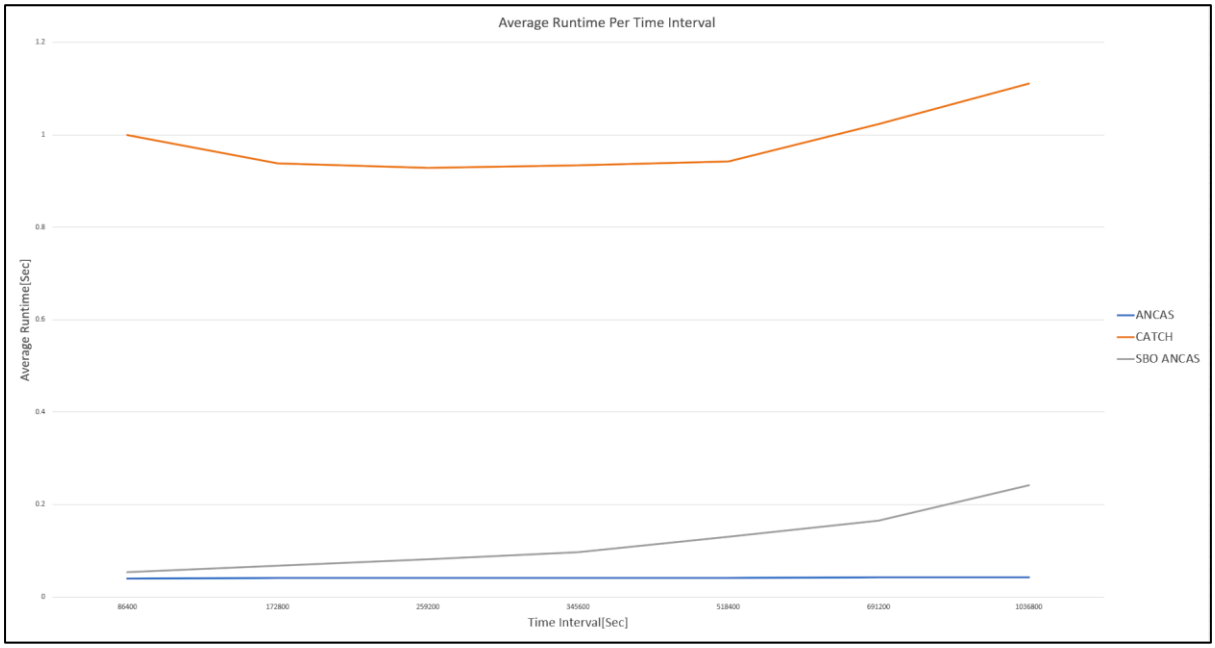
$TOLd = 0.00000001\ TOLt = 0.0001$

To get different a constant number of points over a changing time interval we use a specific combination of TminFactor and the time interval, for example, if we use a time interval of a week with a factor of 2 and a time interval of 2 weeks with a factor of 1, we should get a similar number of points.

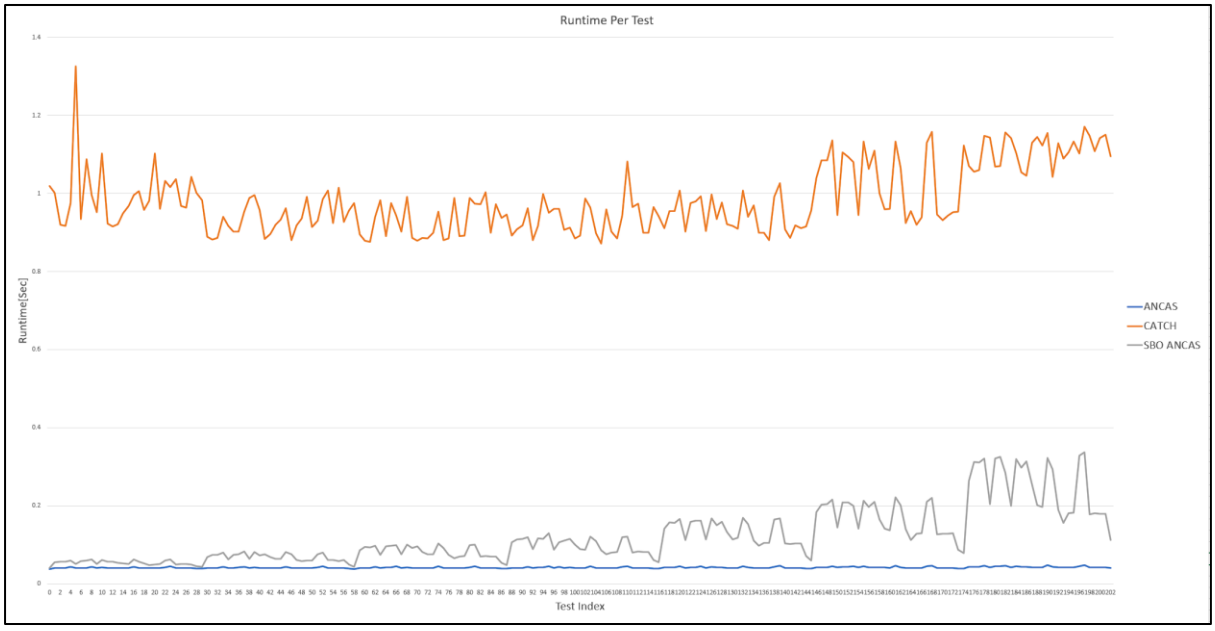
Analyzing The Data:

Like you can see at the graphs [Graph 6:,Graph 7:], ABO ANCAS have high correlation between the size of the time interval of the input and the run time, ANCAS has no change related to the time interval and CATCH is generally the same but have higher runtime over a small time interval and a large one.

Graph 6: Average runtime for each time interval, with a constant number of input point per input data (29 variation at each time interval).



Graph 7: Runtime per tests, the tests are over an increasing time interval, each set of 29 tests is over the same time interval.



7.1.3.3. Correlations and Conclusions

After testing the 2 variations, changing the time interval without changing the number of points and changing the number of point without changing the time interval we arrived to the following observations:

ANCAS

As anticipated, ANCAS is linear to the number of input data, with no change when changing the time interval size, 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

SBO ANCAS runtime is affected by the number of input points, changing the number of outer iterations, additionally it highly dependent on the actual data, bigger time interval meaning more inner iteration until reaching the desired time tolerance, and we can deduce that the actual satellites relative speed and locations will have similar effect, the slower they move the more inner iterations we need to reach the desired 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. We also notice how CATCH runtime change when changing the time interval, even if the number of points stays the same, we suspect the reason for that is the fact that the bigger the time interval the more roots we will have and with a constant number of points, the more roots we have per iteration, interestingly it means that our roots finding algorithms take more time when we have high number of roots and a very small number of roots for the polynomial.

7.2. 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 communications 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 are already future related project on the way, for example algorithms for the satellite to keep track on other objects by itself (collecting the data needed for finding the TCA) and it will require more testing in the same environment.

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 only regret not being able to run our test and system on the actual satellite OBC, but we intend to stay available for it and provide support when the OBC will be available.

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.

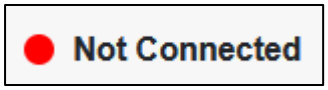
Image 6: Testing Station App configuration. The Source and Destination ports should be the opposite of the Tested OBC App ports.

```
[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

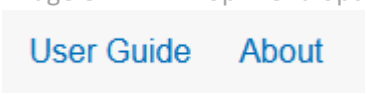
In the top of our application we have a menu, containing 3 things:
Tested OBC App Connection indicator:

Image 7: Connection status indicators.



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

Image 8: Top Menu options.



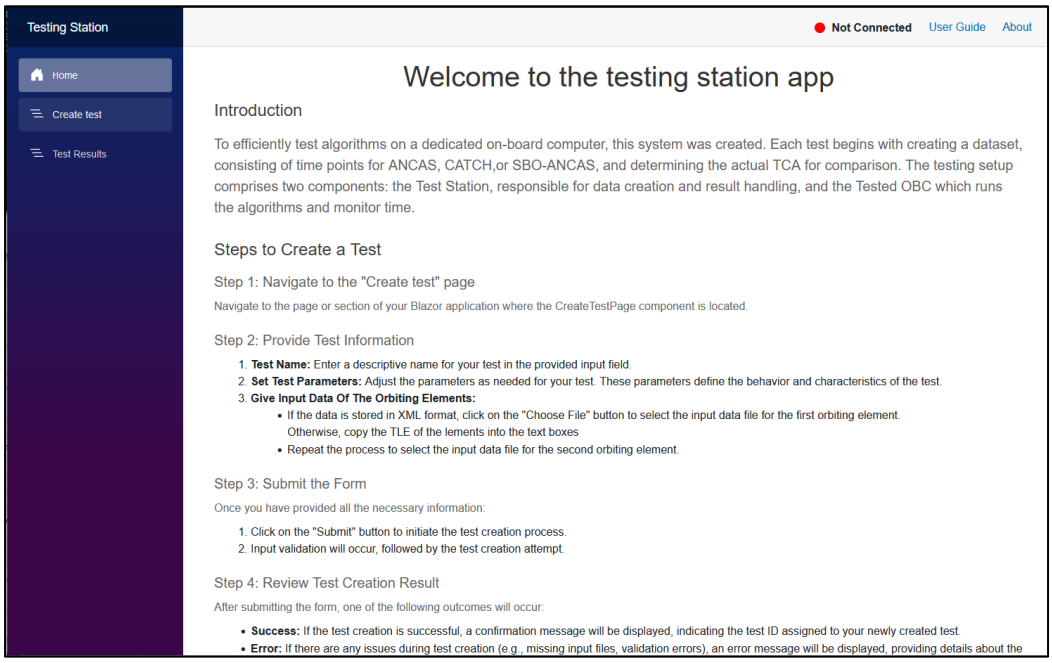
Menu

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.

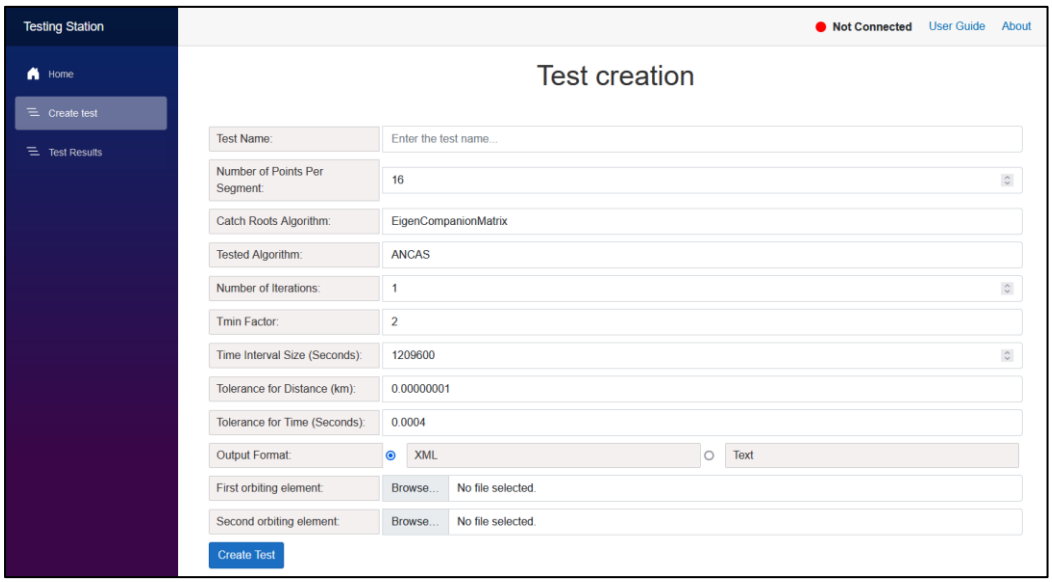
Image 9: Home Page and the Menu.



Creating a new test

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

Image 10: Create tests page.

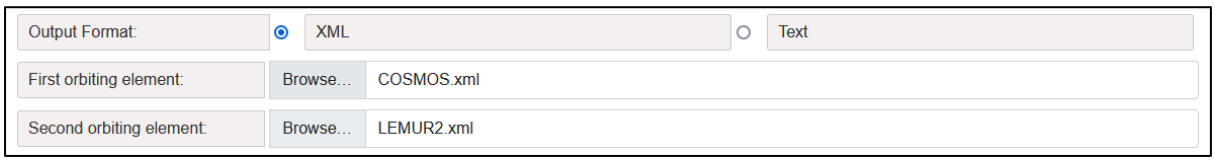


The create test form come with some default values, with number of points per segment set to 16, number of iterations to 1, TminFactor to 2, the time interval to 1 week and the tolerances to their default values.

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.

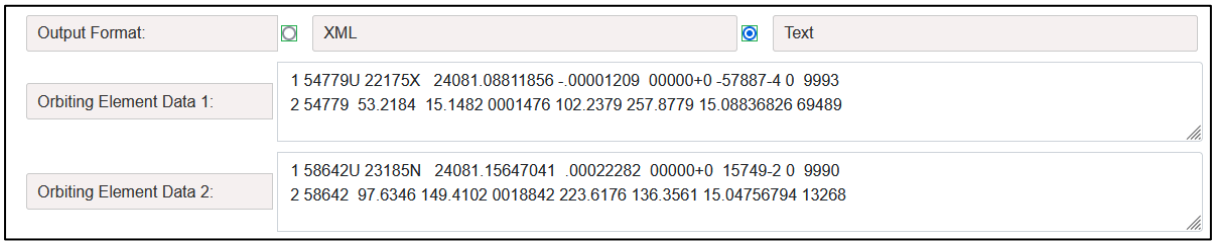
The first input option is XML files, simply choose 2 files from your file system.

Image 11: Test creation with XML format.



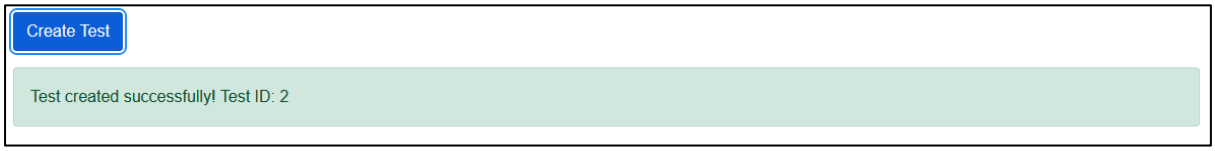
The second input option is TLE, placing the 2 lines of input for each satellite inside the text box as 2 lines.

Image 12: Test creation with the TLE format.



After pressing the Create Test button, if your input was correct, the test will be created and you'll get the test ID.

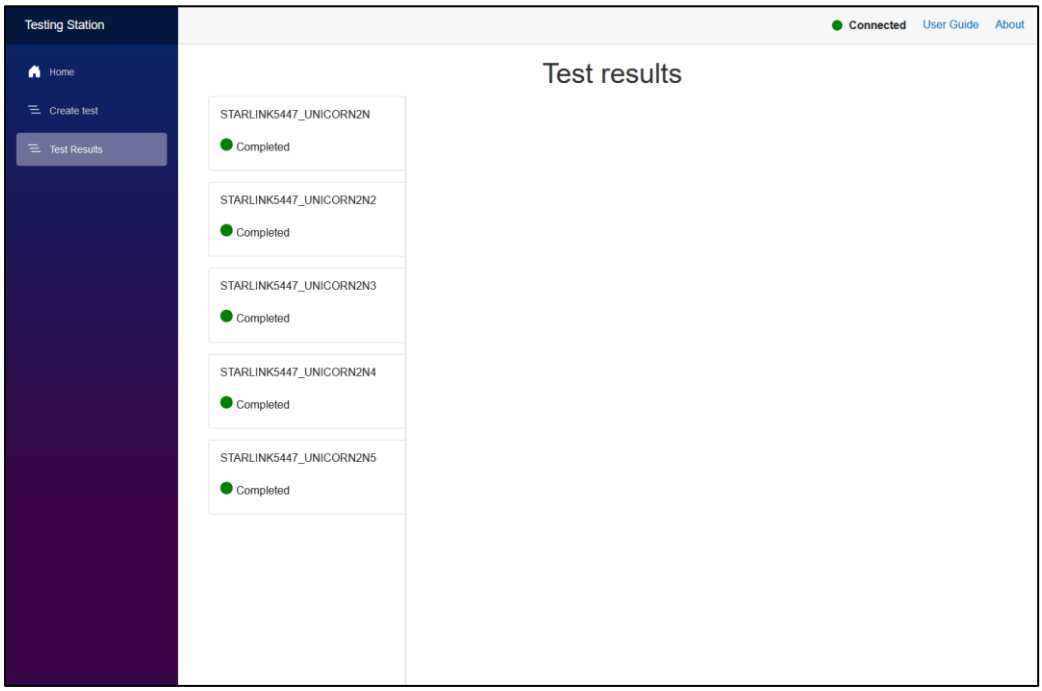
Image 13: Test create success message.



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 14: 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 15: 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):	0.1387496680557096
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):	0.1268643704056604
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 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 16: 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.

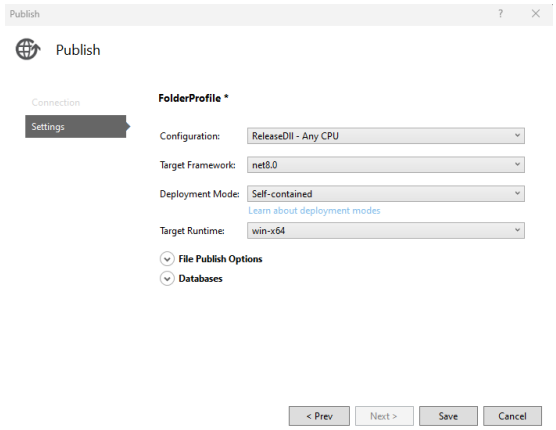
After configuring the INI file it should be place in the same folder as the Tested OBC.

9. Maintenance Guide

9.1. Testing Station App

9.1.1. Installing and Running the Application

The testing station app should be ready to use and can be found in the released versions folder, with a setting INI file near it. 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 "Seld-contained" and Target Runtime as "win-x64".



9.2. Tested OBC App

9.2.1. Installing and Running the Application

9.2.1.1. Running with gem5 Simulation

To run the gem5 simulation you need to clone, build and only then run the required script. The simulation has a few requirements and can only run on Linux. To save us the trouble we created scripts for every step. In the `ReleasedVersions\TestedObcApp` we have the following files:

1. **InstallGem5X86:** Shell script, doing anything required to run the gem5 emulator. Starting with installing requirements, cloning the gem5 repository and building the emulator. The building is quite a long process but only needed to be done once.
2. InstallGem5BuildOnlyX86: Shell script, if you already cloned and just want to build the emulator.
3. **TestedOBCAppX86:** The Tested OBC application binary, already compiled.
4. TestedObcAppSettings: The Tested OBC configuration file, make sure to update the file with your Testin Station IP address.
5. **BuildAppForX86:** Shell script for building the Tested OBC application binary.
6. **RunGem5X86:** Shell script, after installing gem5, having the Tested OBC application and setting ready you can use this script to run the application on the emulator.

9.2.1.2. Running on Windows

In the ReleasedVersions folder you can find executable of the latest Tested OBC App ready to use. 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.

9.2.1.3. Running on any other systems

For other system you can either use the existing ` build_Linux` script or use the `CMakeLists` file to build the project from the command line.

Image 17: 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 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 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 the m available in the GUI level, for example adding an option for creating a few tests at once, 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] Denenberg, Elad. "Satellite closest approach calculation through Chebyshev Proxy Polynomials." CATCH

[2] Alfano, S. "Determining Satellite Close Approaches-Part II." ANCAS

[3] Denenberg, Elad. Gurfil, Pini. “Improvements to Time of Closest Approach Calculation.” SBO ANCAS

[4] NASA on orbital debris: https://www.nasa.gov/mission_pages/station/news/orbital_debris.html

[5] SGP4 python library: https://pypi.org/project/sgp4/

[6] SGP4 and implementation for C++: https://celestrak.org/publications/AIAA/2006-6753/

[7] CelesTrak: https://celestrak.org/

[8] CelesTrak Current Data: https://celestrak.org/NORAD/elements/index.php?FORMAT=tle

[9] The Kessler Syndrome: Implications to Future Space operations: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=227655e022441d1379dfdc395173ed2e776d54ee

[10] SGP4: Revisiting Spacetrack Report: https://celestrak.org/publications/AIAA/2006-6753/AIAA-2006-6753.pdf

[11] Google test framework: https://github.com/google/googletest

[12] The Eigen Library: https://eigen.tuxfamily.org/index.php?title=Main_Page

[13] The Armadillo Library: https://arma.sourceforge.net/

[14] The gem5 simulator: https://www.gem5.org/

[15] On Board Computer: https://www.isispace.nl/product/on-board-computer/

[16] INIH- INI File Reader Library: https://github.com/benhoyt/inih

[17] SQLITE 3: https://www.sqlite.org/index.html

[18] Our GitHub repository: https://github.com/ommersh/Final-Project

[19] C++ Coding Conventions: https://github.com/cpp-best-practices/cppbestpractices/blob/master/03-Style.md

[20] Semantic Versioning: https://semver.org/spec/v2.0.0.html