### ANCAS

The first algorithm, ANCAS **[**[**2**](#Reference2)**]** 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**](#Reference2)**]**)

**Input**:

**Output**:

**for** each set of 4 points **do:**

Map the time points to on the interval

Calculate using **Eq.([5](#eq5)/**[**2**](#eq2)**)** with the points

Fit cubic polynomial to according to **[**[**2**](#Reference2) **,Eq.1f-1j]** over

Find the cubic polynomial real roots in the interval

Fit cubic polynomials for in the interval

**for** each root **do:**

calculate the distance using **Eq.([6](#eq6))**

**if** **:**

**end**

**end**

**end**

The cubic polynomial coefficients calculations described in article **[**[**2**](#Reference2)**]**.

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.

### SBO-ANCAS

The second algorithm, SBO-ANCAS **[**[**2**](#Reference2)**]** 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 **[**  **]**)

**Input**: ,,

**Output**:

**for** each set of 4 points **do:**

**Do**

Map the time points to on the interval

Calculate using **Eq.([5](#eq5)/**[**2**](#eq2)**)** with the points

Fit cubic polynomial to according to **[**[**2**](#Reference2) **,Eq.1f-1j]** over

Find the cubic polynomial real roots in the interval

Fit cubic polynomials for in the interval

**for** each root  **do:**

calculate the distance at  using **Eq.([6](#eq6))**

**if** **:**

**end**

**end**

Sample and at using a propagator

**While**  OR

**end**

The cubic polynomial coefficients calculations described in article **[**[**2**](#Reference2)**]**.

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 we use the propagator to sample the location and velocity vectors at , 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.

### CATCH

The third algorithm, CATCH **[**[**1**](#Reference1)**]**, uses **Chebyshev Proxy Polynomial (CPP)[[1](#Reference1)]** 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 , each iteration searches the minimal distance in an interval with size . 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**](#Reference1)**, algorithm 2]**

**Input**: 

**Output**: 









**While** **do**:

Fit CPP  with order N to  according to **[**[**1**](#Reference1) **, Eq.15]** over the interval 

Fit CPP with order N to  over the interval 

Find the roots of 

**for** each root  **do:**

calculate the distance  at  using Eq.([6](#eq6))

**if** **:**



**end**

**end**

****

**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**](#Reference1)**,Eq.18]** and the eigen values of this matrix are the polynomial roots. Using the roots, we found and creating CPP for  we can calculate the minimal distance in each interval and eventually the TCA and respective distance in . 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](#Reference1),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.

## Algorithms Complexity

### ANCAS Time Complexity

In ANCAS **[**[**2**](#Reference2)**]**, 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**](#Reference2)**,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**](#Reference2)**]** 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 **Eq.([6](#eq6)),** 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):



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  iterations.

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



### SBO-ANCAS Time Complexity

In SBO-ANCAS **[** **]**, we are going over a set of 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  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, , with the initial distance between 2 time points of , and tolerance in time . To get to the desired tolerance we need the distance between to the other points to be smaller than . which means that at the last iteration we get:

.

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, :

And so on.

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

And we continue while:

We expect the distance between 2 points, , to be bigger than the tolerance and with a small enough we get:

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 , 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 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.

### CATCH Time Complexity

In CATCH**[**[**1**](#Reference1)**]** algorithm we iterate through the number of time points in our external loop, 

Where {"mathml":"<math style=\"font-family:stix;font-size:16px;\" xmlns=\"http://www.w3.org/1998/Math/MathML\"><mstyle mathsize=\"16px\"><msub><mi>t</mi><mrow><mi>m</mi><mi>a</mi><mi>x</mi></mrow></msub></mstyle></math>"} is the is end boundry in the time range where we're looking for mininal disdance, and equal to half of the smaller revolution time of the object **[**[**1**](#Reference1)**,part 4]**, The value of {"mathml":"<math style=\"font-family:stix;font-size:16px;\" xmlns=\"http://www.w3.org/1998/Math/MathML\"><mstyle mathsize=\"16px\"><msub><mi>N</mi><mrow><mi>d</mi><mi>e</mi><mi>g</mi></mrow></msub></mstyle></math>"} is the order of the polynomial, while we can change the chosen value of N, it was determined that {"mathml":"<math style=\"font-family:stix;font-size:16px;\" xmlns=\"http://www.w3.org/1998/Math/MathML\"><mstyle mathsize=\"16px\"><mi>N</mi><mo>=</mo><mn>16</mn></mstyle></math>"} give sufficient results.

Inside the loop we're doing the following steps:

1. Fit the CPP of order {"mathml":"<math style=\"font-family:stix;font-size:16px;\" xmlns=\"http://www.w3.org/1998/Math/MathML\"><mstyle mathsize=\"16px\"><msub><mi>N</mi><mrow><mi>d</mi><mi>e</mi><mi>g</mi></mrow></msub></mstyle></math>"} to {"mathml":"<math style=\"font-family:stix;font-size:16px;\" xmlns=\"http://www.w3.org/1998/Math/MathML\"><mstyle mathsize=\"16px\"><mover><mi>f</mi><mo>&#x2D9;</mo></mover><mfenced><mi>t</mi></mfenced><mo>,</mo><mo>&#xA0;</mo><msub><mi>p</mi><mi>x</mi></msub><mo>,</mo><mo>&#xA0;</mo><msub><mi>p</mi><mi>y</mi></msub><mo>,</mo><mo>&#xA0;</mo><msub><mi>p</mi><mi>z</mi></msub></mstyle></math>"} over each interval of points:

Assuming the arithmetic operations we use are basic operation done in time complexity of {"mathml":"<math style=\"font-family:stix;font-size:16px;\" xmlns=\"http://www.w3.org/1998/Math/MathML\"><mstyle mathsize=\"16px\"><mi>O</mi><mfenced><mn>1</mn></mfenced></mstyle></math>"}, we calculate the Chebyshev polynomials**[**[**1**](#Reference1)**]**. We'll iterate through {"mathml":"<math style=\"font-family:stix;font-size:16px;\" xmlns=\"http://www.w3.org/1998/Math/MathML\"><mstyle mathsize=\"16px\"><msub><mi>N</mi><mrow><mi>d</mi><mi>e</mi><mi>g</mi></mrow></msub><mo>+</mo><mn>1</mn></mstyle></math>"} 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 {"mathml":"<math style=\"font-family:stix;font-size:16px;\" xmlns=\"http://www.w3.org/1998/Math/MathML\"><mstyle mathsize=\"16px\"><mfenced><mrow><msub><mi>N</mi><mrow><mi>d</mi><mi>e</mi><mi>g</mi></mrow></msub><mo>+</mo><mn>1</mn></mrow></mfenced><mfenced><mrow><msub><mi>N</mi><mrow><mi>d</mi><mi>e</mi><mi>g</mi></mrow></msub><mo>+</mo><mn>1</mn></mrow></mfenced></mstyle></math>"}which is also constant.

The complexity of this step is: {"mathml":"<math style=\"font-family:stix;font-size:16px;\" xmlns=\"http://www.w3.org/1998/Math/MathML\"><mstyle mathsize=\"16px\"><mi>O</mi><mfenced><msub><mi>N</mi><mrow><mi>d</mi><mi>e</mi><mi>g</mi></mrow></msub></mfenced><mo>&#xB7;</mo><mfenced><mrow><mi>O</mi><mfenced><msub><mi>N</mi><mrow><mi>d</mi><mi>e</mi><mi>g</mi></mrow></msub></mfenced><mo>&#xB7;</mo><mfenced><mrow><mi>O</mi><mfenced><msub><mi>N</mi><mrow><mi>d</mi><mi>e</mi><mi>g</mi></mrow></msub></mfenced><mo>&#xB7;</mo><mi>O</mi><mfenced><msub><mi>N</mi><mrow><mi>d</mi><mi>e</mi><mi>g</mi></mrow></msub></mfenced></mrow></mfenced><mo>+</mo><mi>O</mi><mfenced><mn>1</mn></mfenced></mrow></mfenced><mo>&#xA0;</mo><mo>=</mo><mo>&#xA0;</mo><mi>O</mi><mfenced><mn>1</mn></mfenced></mstyle></math>"}

1. Finding the roots for {"mathml":"<math style=\"font-family:stix;font-size:16px;\" xmlns=\"http://www.w3.org/1998/Math/MathML\"><mstyle mathsize=\"16px\"><msub><mi>P</mi><mi>f</mi></msub></mstyle></math>"} will consist of calculating the companion matrix with a size of {"mathml":"<math style=\"font-family:stix;font-size:16px;\" xmlns=\"http://www.w3.org/1998/Math/MathML\"><mstyle mathsize=\"16px\"><msup><msub><mi>N</mi><mrow><mi>d</mi><mi>e</mi><mi>g</mi></mrow></msub><mn>2</mn></msup></mstyle></math>"} and finding the eigen values, using the complexity of matrix multiplication for this step, the complexity will be {"mathml":"<math style=\"font-family:stix;font-size:16px;\" xmlns=\"http://www.w3.org/1998/Math/MathML\"><mstyle mathsize=\"16px\"><mi>O</mi><mfenced><msup><msub><mi>N</mi><mrow><mi>d</mi><mi>e</mi><mi>g</mi></mrow></msub><mn>3</mn></msup></mfenced><mo>&#xA0;</mo><mo>=</mo><mo>&#xA0;</mo><mi>O</mi><mfenced><mn>1</mn></mfenced></mstyle></math>"}, rescaling each eigen value to the actual coefficient value also takes constant time.
2. 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: {"mathml":"<math style=\"font-family:stix;font-size:16px;\" xmlns=\"http://www.w3.org/1998/Math/MathML\"><mstyle mathsize=\"16px\"><mi>O</mi><mfenced><mi>n</mi></mfenced></mstyle></math>"}

### 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 time point for the input: .

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