General constant convention:

- n: *dimensionality of full problem space*
- m_i: *dimensionality of problem vectors*
- r_i: *number of vectors in problem space*

Problem definition: searching for intersections/nearest points of approaches of vectors in n dimensional space.

- Further generalisation to increase scope – "vectors" (at least in my mind) tends to suggest 2D lines: i.e, $p + \mu a$ – can be expanded to m dimensional vectors up to n dimensions $p + \mu_1 a + \mu_2 b + \mu_3 c + \cdots \mu_i x$
    - Where r is some initial starting point, a, b, c ... are directions, μ_i are scalars.
- Further generalisation to include more than 2 vectors
    - Points where all vectors intersect? Points where more than/less than x vectors intersect? Points where there exists an intersection? Etc.

Results in all methods having three primary performance metrics to be compared: How well it scales with:

a. Dimensionality of problem space n
b. Dimensionality of problem vectors m
c. Number of vectors r

In addition, further properties of each method may include:

a. Does it identify the existence/lack of intersections?
b. Will it find the closest point of approach should an intersection not exist?
c. Do all vectors need to have the same dimensionality?
d. Will it identify all possible intersections?

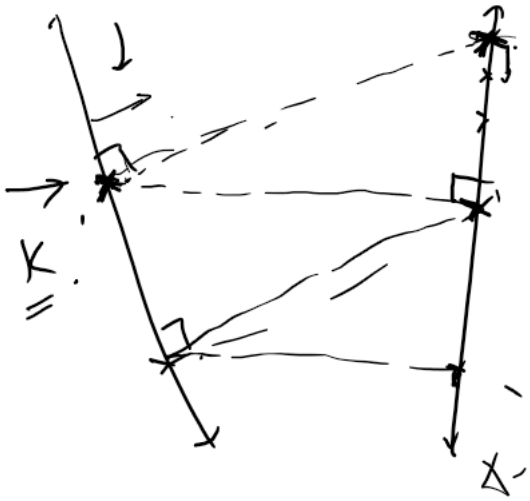
Example method: rudimentary GCSE method for finding intersections of 2 2D vectors in 3D space.

- Identifies the existence/lack of intersections.
- Will not find the closest point of approach.

1. Formulate the two vector equations as a set of simultaneous equations, with 3 equations (one per dimension n) and 2 unknowns (one per dimension m)
2. Arbitrarily choosing 2 of the 3 equations, formulate as a 2x2 matrix and solve the inversion, resulting in some values for μ_1, μ_2.
3. Confirm values are consistent with the last equation.

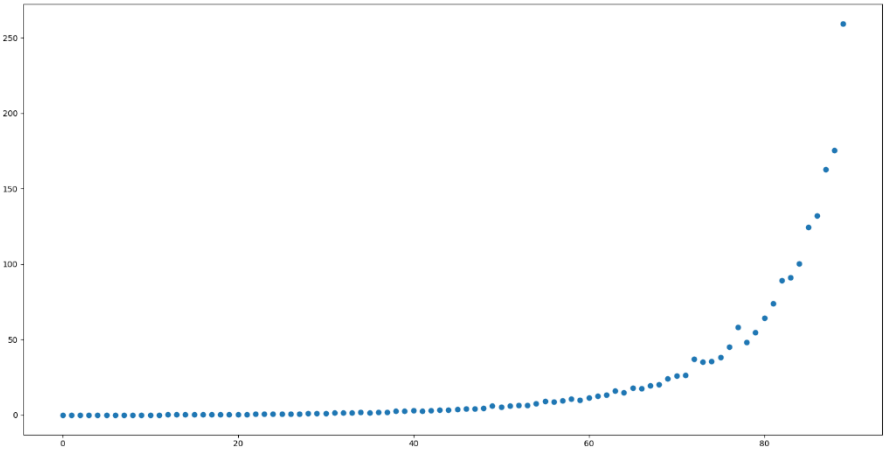
Currently benchmarked methods:

1. “Gradient descent-esque” greedy optimisation algorithm – Take two arbitrary starting points on each vector, and probe in all dimensions, comparing various distances between all probed points. Then, iterate greedily and take new points as the starting points for the next iteration.
2. Orthogonality-informed descent – taking the orthogonally closest point on the line, and repeating. (As vaguely shown in the following image)



Current performance comparisons:

Time (seconds) against vector dimensionality in 1000D space – axis and captions to be labelled.



Iterative and numerical comparison between greedy and orthogonality methods:

```
No size array provided, setting to default.
(array([   1. , -166.3,  312.6,  143.1,  -15.5, -163.4,  -17.9, -237.1,
         12.3,   95.8]), array([  17. , -279.1, -274.4, -166.3,   23.8, -224.1,  109.9,  -29.4,
        -84.6,  -21.8]), 12583)
Made it here!
(array([   1.0155751 , -166.29187912,  312.64925603,  143.12072228,
        -15.54382458, -163.378709  ,  -17.90770774, -237.06126441,
         12.26956715,   95.77559908]), array([  16.98606249, -279.07444188, -274.35807456, -166.27144871,
         23.75668287, -224.06748873,  109.93861973,  -29.41294294,
        -84.61245244,  -21.78368758]), 7)
PS C:\Users\oweng\Desktop\University Work\Y4\Project\testbench>
```