Technische Universiteit
Eindhoven
University of Technology

Department of Mathematics And Computer Science
Discrete Algebra And Geometry
Postbus 513, 5600 MB Eindhoven
The Netherlands
www.tue.nl

Author
Ignas Šablinskas [redacted]

Date
January 5, 2025

2MBD20 CBL Assignment: Recognising Shapes

Group 23

Ignas Šablinskas [redacted]

Where innovation starts

Technische Universiteit
Eindhoven
University of Technology

Table of contents

Title
2MBD20 CBL Assignment: Recognising Shapes

Technische Universiteit
Eindhoven
University of Technology

Table of contents

Title
2MBD20 CBL Assignment: Recognising Shapes

1 Introduction

Media is universal. In all corners of contemporary society, we find the increasing use of pictures and videos. Industry, assisted technology and mechanics all employ its various implementations. With technology at the palm of our hand comes the need to improve its strength, efficiency, and accuracy. In the context of mathematics, this can involve fabricating a structured framework and analysis. So, the question might be: If we have a picture, what makes it good?

Image recognition is an involved process that necessitates identifying and recognising key pieces of information in visual media [Mathworks]. Sophisticated techniques often involve fields such as machine learning, but a vital component is the domain of computational geometry. Accordingly, we elect to study a subset of its key questions [Com].

In this report, we set out to create algorithms with the goal of shape recognition as a step towards advanced image recognition. We begin by defining a scenario of geometric objects that we seek to study through the context of modern algebra, computational geometry, numerical mathematics, complexity theory and probability. The diversity in applying different fields of math allows us to consider issues from receiving imperfect images with ineffective lighting and different angles to characterising the noise and error within our data.

We start with a consistent setting of a subset of $\mathbb{R}^2$, that is, Euclidean space. Note that on account of this choice, we are enabled to define the usual real inner product that describes a notion of distance or length and angle. Furthermore, as a metric, we establish an analytical basis that can be used for theoretical deduction and computation.

In each geometric object under consideration, we describe our model assumptions, approach and relevant algorithmic implementation, producing appropriate critique whenever possible. Once we complete an initial survey, we spend a measure of our report testing the limitations of our model in an attempt to either extend it or learn from alternative perspectives.

Concretely, we start with an examination of right-angled triangles, building toward regular polygons and finally circles. Ultimately, we will end with a conclusion and a reflection of our experience.

2 Recognizing A Right-Angled Triangle

2.1 Our Problem At Hand

We begin with an appropriate description of our data. Consider a set T with cardinality 3, of the form:

$$T = \{(x_1, y_1), (x_2, y_2), (x_3, y_3)\} \subseteq \mathbb{R}^2$$

Our setting of Euclidean space allows us to leverage its natural real inner product and produce a foundation built on Pythagoras'theorem. Observe that the exactness of our points is not guaranteed. Furthermore, whilst coordinates provide a mode of input, we can refashion the information they provide by considering corresponding side lengths obtained from a given collection of points in space.

Accordingly, we define $\bar{a} := \|(x_1, y_1) - (x_2, y_2)\|, \bar{b} := \|(x_1, y_1) - (x_3, y_3)\|$, and $\bar{c} := \|(x_2, y_2) - (x_3, y_3)\|$.

Next, we define a polynomial inspired by Pythagoras' theorem of the form

$$f(x, y, z) = x^2 + y^2 - z^2,$$

where our input of side lengths enables us to study an expression of our collection of data alongside a function whose zero set corresponds to the vertices of a right-angled triangle.

Predictably, it is likely that our data is aberrated by error. Thus, in contrast to exact arithmetic, we seek to define a background of study that aims to examine solutions with regards to predetermined numerical tolerance.

2.2 Our proposed model

In this mathematical model, we will focus on developing an algorithm and corresponding procedures that utilise side lengths rather than coordinates. We are presented with perturbed data

on side lengths. As a result, we start by defining errors corresponding to each side length:

$$\varepsilon_a = \bar{a} - a$$
$$\varepsilon_b = \bar{b} - b$$
$$\varepsilon_c = \bar{c} - c,$$

where a, b and c represent our true side lengths.

To continue describing our approach it is vital for us to take stock of the assumptions we employ. We will make appropriate references and develop them throughout this chapter. Consequently, in this model we make the following assumptions:

1. Input set T has cardinality 3 and at most 4.
2. Our side errors are sufficiently noisy to apply our mathematical analysis.
3. Our side errors have a small probability of lying outside our predetermined error bound.
4. Sides lengths are defined by the euclidean inner product.
5. Our side errors $\varepsilon_a, \varepsilon_b$ and ε_c are bounded by a predetermined theoretical bound ε where $\varepsilon > 0$.

Next, we set out to find a function that bounds our polynomial f with aberrated input in terms of our assumed bound ε. We will make use of this function to conduct an investigation to determine the presence of right-angled triangles. Subsequently, we aim to perform empirical experiments using implementation and further study of the limitations of our method.

2.3 Our Initial Solution

As described, we construct our bound function denoted by g such that $|f(\bar{a}, \bar{b}, \bar{c})| < g(\bar{a}, \bar{b}, \bar{c}, \varepsilon)$. It suitably follows that

$$\begin{aligned}
|f(\bar{a}, \bar{b}, \bar{c})| &= |\bar{a}^2 + \bar{b}^2 - \bar{c}^2| \\
&= |(a + \varepsilon_a)^2 + (b + \varepsilon_b)^2 - (c + \varepsilon_c)^2| \\
&= |(a^2 + 2a\varepsilon_a + \varepsilon_a^2) + (b^2 + 2b\varepsilon_b + \varepsilon_b^2) - (c^2 + 2c\varepsilon_c + \varepsilon_c^2)| \\
&= |(a^2 + b^2 - c^2) + 2(a\varepsilon_a + b\varepsilon_b - c\varepsilon_c) + (\varepsilon_a^2 + \varepsilon_b^2 - \varepsilon_c^2)| \\
&\leq 2|a\varepsilon_a + b\varepsilon_b - c\varepsilon_c| + |\varepsilon_a^2 + \varepsilon_b^2 - \varepsilon_c^2| \\
&\leq 2(|a\varepsilon_a| + |b\varepsilon_b| + |c\varepsilon_c|) + |\varepsilon_a^2 + \varepsilon_b^2 - \varepsilon_c^2| \\
&\leq 2\varepsilon(a + b + c) + 2\varepsilon^2 \\
&\leq 2\varepsilon|\bar{a} + \bar{b} + \bar{c}| + 8\varepsilon^2 \\
&=: g(\bar{a}, \bar{b}, \bar{c}, \varepsilon).
\end{aligned}$$

Consequently, we will abide by the following protocol to test for a right-angled triangle:

1. Fetch the input values $\bar{a}$, $\bar{b}$, $\bar{c}$, and ε.
2. Compute the values $f' = |f(\bar{a}, \bar{b}, \bar{c})|$ and $g' = g(\bar{a}, \bar{b}, \bar{c}, \varepsilon)$.
3. Check if $f' < g'$. If true, conclude that the triangle could have originated from a right triangle within the error ε. If false, repeat step 2 for $|f(\bar{a}, \bar{c}, \bar{b})|$ and $|f(\bar{c}, \bar{b}, \bar{a})|$. Due to symmetry, the other 3 comparisons namely, $\left|f(\bar{c}, \bar{a}, \bar{b})\right|$, $\left|f(\bar{b}, \bar{c}, \bar{a})\right|$, and $\left|f(\bar{b}, \bar{a}, \bar{c})\right|$ are irrelevant.

We will now inquire into empirical results based on data.

2.4 Implementation And Further Study

A program implementation on our provided test data produces the following output for the given test cases:

1. Sides: [3, 4, 5], ε: 0.01, Satisfies Right Triangle: True
2. Sides: [5, 4, 3], ε: 0.01, Satisfies Right Triangle: True
3. Sides: [3.01, 4.01, 4.99], ε: 0.01, Satisfies Right Triangle: True
4. Sides: [3.01, 4.99, 4.01], ε: 0.01, Satisfies Right Triangle: True
5. Sides: [3.01, 4.01, 4.99], ε: 0.002, Satisfies Right Triangle: False
6. Sides: [3.01, 5.01, 4.01], ε: 0.002, Satisfies Right Triangle: True
7. Sides: [3.1, 4.105, 4.901], ε: 0.1, Satisfies Right Triangle: True

The program does not output any false negatives due to the upper bound found via the aforementioned inequalities. However, we do observe that case 7 results in a false positive. The reasoning behind this relies on the impossibility to construct a right-angled triangle with sides $\bar{a} = 3.01$, $\bar{b} = 4.105$, and $\bar{c} = 4.901$ alongside the error bound $\varepsilon = 0.1$. Here it is evident that $a^2 + b^2 < c^2$, yet since our bound is not optimally minimised we find that it is a possibility. Finally, for case 5, we find that it is impossible to get such an error; a true answer due to similar principles holding as is for case 7.

2.5 Correlation Between The Errors

In our model, the errors are generally assumed to be independent. However, if the errors result from inaccuracies in the initial positioning of the three points, they might exhibit a correlation. To address this scenario, we can analyze the bounds and behaviors of these errors under specific assumptions. We will consider two cases, namely, when all of the errors are nonnegative and when all of the errors are nonpositive. We proceed as follows:

Assume that $\epsilon_a = a - \bar{a}, \epsilon_b = b - \bar{b}, \epsilon_c = c - \bar{c} \geq 0$, then

$$\begin{aligned}
|f(\bar{a},\bar{b},\bar{c})| &= |\bar{a}^2 + \bar{b}^2 - \bar{c}^2| \\
&= |(a+\varepsilon_a)^2 + (b+\varepsilon_b)^2 - (c+\varepsilon_c)^2| \\
&= |(a^2 + 2a\varepsilon_a + \varepsilon_a^2) + (b^2 + 2b\varepsilon_b + \varepsilon_b^2) - (c^2 + 2c\varepsilon_c + \varepsilon_c^2)| \\
&= |(a^2 + b^2 - c^2) + 2(a\varepsilon_a + b\varepsilon_b - c\varepsilon_c) + (\varepsilon_a^2 + \varepsilon_b^2 - \varepsilon_c^2)| \\
&\leq 2|a\varepsilon_a + b\varepsilon_b - c\varepsilon_c| + |\varepsilon_a^2 + \varepsilon_b^2 - \varepsilon_c^2| \\
&\leq 2|(a+b)\varepsilon - c\varepsilon_c| + 2\varepsilon^2 \\
&\overset{\text{Nonnegative errors } \epsilon_a, \epsilon_b, \epsilon_c \text{ and a + b > c}}{\leq} 2\varepsilon|a+b| + 2\varepsilon^2 \\
&= 2\epsilon|\bar{a} - \epsilon_a + \bar{b} - \epsilon_a| + 2\epsilon^2 \\
&\leq 2\epsilon|\bar{a} + \bar{b}| + 2\epsilon^2 \\
&=: g(\bar{a},\bar{b},\bar{c},\varepsilon).
\end{aligned}$$

Now, assume that $\epsilon_a = a - \bar{a}, \epsilon_b = b - \bar{b}, \epsilon_c = c - \bar{c} \leq 0$, we will show that it is similar to the one we just derived:

$$\begin{aligned}
|f(\bar{a},\bar{b},\bar{c})| &= \dots \\
&\leq 2|a\varepsilon_a + b\varepsilon_b - c\varepsilon_c| + |\varepsilon_a^2 + \varepsilon_b^2 - \varepsilon_c^2| \\
&\leq 2|-(a(-\varepsilon_a) + b(-\varepsilon_b) - c(-\varepsilon_c))| + 2\epsilon^2 \\
&\leq 2|a(-\varepsilon_a) + b(-\varepsilon_b) - c(-\varepsilon_c)| + 2\epsilon^2 \\
&\overset{\text{Nonnegative errors } -\epsilon_a, -\epsilon_b, -\epsilon_c \text{ and a + b > c}}{\leq} 2\varepsilon|a+b| + 2\varepsilon^2 \\
&= 2\epsilon|\bar{a} - \epsilon_a + \bar{b} - \epsilon_a| + 2\epsilon^2 \\
&\leq 2\epsilon(|\bar{a} + \bar{b}| + 2\epsilon) + 2\epsilon^2 \\
&\leq 2\epsilon|\bar{a} + \bar{b}| + 6\epsilon^2 \\
&=: g(\bar{a},\bar{b},\bar{c},\varepsilon).
\end{aligned}$$

Hence the upper bounds for $f(\bar{a},\bar{b},\bar{c})$ if all of the errors are nonnegative or nonpositive are $2\epsilon|\bar{a}+\bar{b}|+2\epsilon^2$ and $2\epsilon|\bar{a}+\bar{b}|+6\epsilon^2$ respectively. There are more ways to investigate the correlation between the errors, such as taking all the errors to be equal or only some of them nonpositive or nonnegative, but we will leave it at that and consider other interesting ways to look at the task at hand.

2.6 Input Containing The Vertices

We will now consider a scenario where instead of the side lengths, 3 points are given to as $(x_1, y_1), (x_2, y_2), (x_3, y_3)$. By leveraging mathematical bounds, we establish the constraints on the distances between them and derive new bounds for evaluating a right-angled triangle under error assumptions. Let $\bar{p}_1, \bar{p}_2, \bar{p}_3$ denote the approximate positions of the points (given to us) corresponding to the exact points p_1, p_2, p_3. Suppose the error for each of the points is at most ϵ, meaning:

$$dist(\bar{p}_1, p_1) < \epsilon, \quad dist(\bar{p}_2, p_2) < \epsilon, \quad dist(\bar{p}_3, p_3) < \epsilon.$$

From analysis, for $i, j \in \{1, 2, 3\}$ and $i \neq j$, we have:

$$dist(\bar{p}_i, \bar{p}_j) \leq dist(\bar{p}_i, p_i) + dist(p_i, p_j) + dist(p_j, \bar{p}_j).$$

Since $dist(\bar{p}_i, p_i) < \epsilon$ and $dist(\bar{p}_j, p_j) < \epsilon$, it follows that:

$$dist(\bar{p}_i, \bar{p}_j) < 2\epsilon + dist(p_i, p_j).$$

Similarly, we can derive:

$$dist(p_i, p_j) - 2\epsilon < dist(\bar{p}_i, \bar{p}_j),$$

where $dist(p_i, p_j)$ is the true length of one of the sides.

Converting these distances to the side lengths $\bar{a}, \bar{b}, \bar{c}$, we find that $\epsilon_a = a - \bar{a}, \epsilon_b = b - \bar{b}, \epsilon_c = c - \bar{c}$ for $-2\epsilon < \epsilon_a, \epsilon_c\epsilon_c < 2\epsilon$ for some $\epsilon > 0$. Now, the upper bound for evaluating whether the sides form a right-angled triangle is very similar to the initial one except that instead of ϵ we have 2ϵ. With that we get the following:

$$\begin{aligned}
|f(\bar{a}, \bar{b}, \bar{c})| &= \dots \\
&\leq 2(|a\varepsilon_a| + |b\varepsilon_b| + |c\varepsilon_c|) + |\varepsilon_a^2 + \varepsilon_b^2 - \varepsilon_c^2| \\
&\leq 4\varepsilon(a + b + c) + 8\varepsilon^2 \\
&\leq 4\varepsilon|\bar{a} + \bar{b} + \bar{c}| + 32\varepsilon^2 \\
&=: g(\bar{a}, \bar{b}, \bar{c}, \varepsilon).
\end{aligned}$$

This analysis demonstrates that even under bounded positional errors, the derived bounds on side lengths and their subsequent use in evaluating the Pythagorean condition remain robust.

The new upper bounds derived provide a tighter constraint, ensuring accurate testing for right-angled triangles in the presence of measurement inaccuracies. This methodology can be extended to other geometric error models for further applications.

2.7 Reducing the number of equations

In scenarios where a given set of three side lengths is provided, there are $3! = 6$ possible configurations to test whether they satisfy the Pythagorean theorem. However, due to the inherent commutativity of addition, i.e.,

$$\bar{a}^2 + \bar{b}^2 - \bar{c}^2 = \bar{b}^2 + \bar{a}^2 - \bar{c}^2$$

we have that we only need to evaluate three unique equations, such as

$$f(\bar{a}, \bar{b}, \bar{c}), \quad f(\bar{a}, \bar{c}, \bar{b}), \quad f(\bar{c}, \bar{b}, \bar{a}).$$

This reduces the computation complexity twofold. To see why, lets remember exercise 5 part a) from Week 2 homework, which asked to prove that the map $h \mapsto (1\ 2)h$ from subsets of S_n containing all even and odd permutations respectively, is bijective. There we concluded that there is an equal number of odd and even permutations. This relates to what we are doing right now. We can imagine as if we are permuting the letters $\bar{a}, \bar{b}, \bar{c}$ (in other words, the permutations group acts on the set of letters) as function parameters in in function $f(\bar{a}, \bar{b}, \bar{c})$ and when permutation $(1\ 2)$ is applied, we change the first and second entries of our function to get $f(\bar{b}, \bar{a}, \bar{c})$ and note that this doesn't change the outcome due to commutativity. Since the sets of permutations that produce these equal outcomes together have size $|S_n|$ and there are only two of those (since other permutations acting on the set of letter actually do change the outcomes of f) we can conclude that we indeed can reduce the number of equations twofold without losing any information.

We now take a look at what happens when a set of four distances is provided (e.g., due to preprocessing errors). There it is necessary to determine whether three of these distances form a right-angled triangle. Initially, there are $4! = 24$ permutations to consider when selecting three distances. The reason being is as follows. We can define a function f' that takes 4 arguments, namely, $\overline{a}, \overline{b}, \overline{c}, \overline{d}$ where $\overline{d}$ is the fourth distance that we are provided with. This function is defined as follows:

$$f'(\overline{a}, \overline{b}, \overline{c}, \overline{d}) = f(\overline{a}, \overline{b}, \overline{c})$$

All it does is map the first three arguments to the function f. Now, we can look at the problem similarly as for the 3 side case, where we permute the arguments of f' and check whether the desired property of a right angle triangles holds for all combinations of the sides. Again, due to the commutativity of expressions and by the similar argumentation as before, we reduce the number of equations to check to $\frac{4!}{2} = \frac{24}{2} = 12$, which again reduces the computation complexity twofold. This overall approach leverages geometric symmetries and algebraic properties to streamline the verification process for right-angled triangles, reducing the computational complexity twofold.

2.8 A Probabilistic Approach

Now, deterministic bounds represent vital information in terms of uncertainty. The choice of a constant bound and its study both along and around it is an oft-utilised method in engineering and practical application. Having acknowledged this perspective, we found alternative approaches to potentially be noteworthy. Let us consider a probabilistic approach to error propagation. We seek to develop this facet in two manners: initially, we will use our conventional elementary assumptions to produce a probabilistic deduction on the distributions of all data points throughout this report. Subsequently, we will wield this result to derive a probability bound that our points lie within a predetermined error-bound interval. Since we can adjust our interval to arbitrary translations, they produce a uniform remark on our error bounds.

Proposition 1 (Probablistic Nature Of Image Data) *Our Image Data Is I.I.D. (Independent And Identically Distributed)*

Proof. We argue using our assumptions and the Representation Theorem (Frequentist Version) [O'Neill, 2009].

Note by assumption our data is sufficiently noisy, in the sense that all our data points and/or computed objects respect a certain degree of error. Suppose not, then our data can be considered insufficient for the purpose of analysis or biased in nature. Alternatively, the resulting problem can also be considered to be ill-conditioned.

Given any finite set of points that form a geometric object, we can permute their positions using arbitrary permutations ρ from the symmetric group S_n. These permutations result in our data

points and objects either lying within an error bound or otherwise. Since our points are real values or objects, we can model these binary options and error values as Bernoulli-distributed random variables, it follows that their push forward measures (cumulative distribution function) are then invariant under the permutation ρ.

By the definition of exchangeability in [O'Neill, 2009], it follows that our values are exchangeable.

Therefore, by the the Representation Theorem, it follows that our elements are I.I.D. and we conclude our argument.

Proposition 2 (High Probability Of Bounded Error) *Our side length errors lie in the translation-invariant interval* $(-\varepsilon, \varepsilon)$ *with relatively high probability.*

Proof. We argue by theory found in [Royden and Fitzpatrick, 1963] and [Billingsley, 1999]. Define a product measure space $(\Omega, \mathcal{F}, \mu) = ([0,1]_1 \times [0,1]_2 \times [0,1]_3, \mathcal{B}_{1|[0,1]} \otimes \mathcal{B}_{2|[0,1]} \otimes \mathcal{B}_{3|[0,1]}, \lambda_{1|[0,1]} \otimes \lambda_{2|[0,1]} \otimes \lambda_{3|[0,1]})$ that acts as a universal probability space, where $\mathcal{B}_{i|[0,1]}$ denotes the Borel σ-algebra of $\mathbb{R}$ restricted to the interval $[0,1]$ and $\lambda_{i|[0,1]}$ denotes the Lebesgue measure restricted to the same interval. Observe that our space is σ-finite and our product measure is unique by the Carathéodory Extension Theorem. Moreover, let $\pi_i : [0,1]_1 \times [0,1]_2 \times [0,1]_3 \mapsto [0,1]_i$ denote the canonical projection maps from the product sample space to the corresponding sample space. First, we aim to use the inverse sampling method (or Smirnov transform) [Tang and Ng, 2011] to construct independent random variables that describe a probabilistic perspective of our error bounds. Let F_{X_i} denote the generalised inverse of the i-th random variable and U_i denote the standard unit uniform random variable. Note that the generalised inverse is a function given by

$$F_{X_i}^{-1}(p) := \inf\{x \in \mathbb{R} : F_{X_i}(x) \geq p\} \quad \forall p \in [0,1].$$

We then define a family of Bernoulli random variables $\{X_i\}_{i \in I}$ scaled by an indicator function $\mathbb{1}_{\{\varepsilon_i \notin (-\varepsilon, \varepsilon)\}}$:

$$X_i := F_{X_i}^{-1} \circ U_i \circ \pi_i.$$

By virtue of this construction, each random variable has a Cumulative Distribution Function of

the form:

$$F_{X_i}(t) = \mathbb{P}(X_i \leq t) = \begin{cases} 1-p & t = 0, \\ p & t = 1, \end{cases}$$

where $p > 0$ is small and each $1 \leq i \leq 3$ corresponds to a side length. Observe that due to our construction, these random variables are well-defined and model the independent and identically distributed nature provided by the random sample assumption of our error bounds. Employing the independence of our random variables now allows us to examine the instance that our side length errors all fall within our predetermined error bound. Due to the factorization of our probability measures, it holds that:

$$\begin{aligned} \mathbb{P}(X_1 = 0, X_2 = 0, X_3 = 0) &= \Pi_{i=1}^{3} \mathbb{P}(X_i = 0) \\ &= (1-p)^3 \\ &= 1^3 - 3 \cdot 1^2 \cdot p - 3 \cdot 1 \cdot p^2 - p^3. \end{aligned}$$

Since p is strictly positive and sufficiently small by assumption, we can quotient the impact of higher order terms and find that

$$\mathbb{P}\{\varepsilon_a, \varepsilon_b, \varepsilon_c \in (-\varepsilon, \varepsilon)\} \leq 1 - 3p,$$

implying that our side length errors behave well with relatively high probability and the probability of the event that our errors lie outside our bound is minor, arbitrarily scaling with p

Therefore, we conclude having demonstrated our proposed argument.

2.9 Consequences And Choices

Despite not knowing side lengths we bound the errors. Think of it as pixels

Currently, we note the relative efficacy of our progress resulting in the ability to recognize a particular shape. However, the overall picture is yet to be seen. Triangles are an apt start to our proposed problem, but what if we can approximate shapes with increasing vertices? Let us now explore this notion.

3 Recognizing Regular Polygons

3.1 General overview

We are given a set $P = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\} \subseteq \mathbb{R}^2$ of size n. The goal is to check whether these coordinates could have come from a regular polygon, which are polygons characterized by equal side lengths and equal internal angles. There are two interpretations of such a problem. One could assume that the points given to us simply lie on the sides of the polygon or that the points are the vertices of the polygon. An example of that would be a hexagon with 6 points distributed on only two sides of it or a hexagon with 6 points being the vertices of it. We will consider both scenarios in depth and provide our analysis on each of them. Finally, note that each coordinate (x_i, y_i) is off by at most ϵ in Euclidean norm.

3.2 A Primary Interpretation

Consider the set P, our goal is to figure out whether it is possible to construct a regular polygon such that all of the points lie on the boundary of it. We will work with n-gons, where n is the number of points we are given, but as we will see, similar idea can be used with any polygon to fit the n points on it and they might even yield better results. This algorithm finds the best match for the circle by minimizing the residual, inscribes a regular n-gon, proceeds to classify the points based on their position and finds whether it is possible to leverage the errors to make the points lie on the polygon.

3.2.1 Algorithm Overview

We will firstly use the least-squares method to find the best match for the circle. This is a known algorithm called Kasa's Circle Fit [Kasa, 1976]. The goal is to minimize the residual (error) E. Let $C = (C_x, C_y)$ be the center point of some circle and R its radius. We will use p_x, p_y to denote the x, y coordinates of some arbitrary point in P. Consider the following definition of the error:

$$E = \sum_{i=1}^{n} ((p_{i,x} - C_x)^2 + ((p_{i,y} - C_y)^2) - R^2)^2$$

Moreover, we know that any equation of a circle can be written as:

$$p_x^2 + p_y^2 + Ap_x + Bp_y + D = 0, \tag{3.1}$$

where A, B, and D are unknown parameters to be determined. For each point $(p_{i,x}, p_{i,y})$ let

$$z_i = p_{i,x}^2 + p_{i,y}^2.$$

Substituting z_i into the circle equation gives:

$$Ap_{i,x} + Bp_{i,y} + D = -z_i.$$

This is a linear equation in terms of A, B, and D.

Using n data points $(p_{i,x}, p_{i,y})$, we can rewrite the equations in matrix form as:

$$\begin{bmatrix} p_{1,x} & p_{1,y} & 1 \\ p_{2,x} & p_{2,y} & 1 \\ \vdots & \vdots & \vdots \\ p_{n,x} & p_{n,y} & 1 \end{bmatrix} \begin{bmatrix} A \\ B \\ D \end{bmatrix} = - \begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_n \end{bmatrix},$$

which simplifies to:

$$Mt = -z,$$

where:

$$M = \begin{bmatrix} p_{1,x} & p_{1,y} & 1 \\ p_{2,x} & p_{2,y} & 1 \\ \vdots & \vdots & \vdots \\ p_{n,x} & p_{n,y} & 1 \end{bmatrix}, \quad t = \begin{bmatrix} A \\ B \\ D \end{bmatrix}, \quad z = \begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_n \end{bmatrix}.$$

This is where we get n polynomial equations mentioned in the Goals section of the CBL description. Since $Mt = -z$ is generally overdetermined (more equations than unknowns), we solve it using the least squares approach. The least squares solution minimizes the residual $\|Mt + z\|^2$. The solution is given by:

$$t = -(M^T M)^{-1} M^T z,$$

where M^T is the transpose of M and $(M^T M)^{-1}$ is the inverse of the 3×3 matrix $M^T M$.

Once A, B, and D are determined, the circle's center (C_x, C_y) and radius R can be computed as follows:

- The x-coordinate of the center is:
$$C_x = -\frac{A}{2}.$$
- The y-coordinate of the center is:
$$C_y = -\frac{B}{2}.$$
- The radius is:
$$R = \sqrt{C_x^2 + C_y^2 - D}.$$

Substituting $C_x = -\frac{A}{2}$ and $C_y = -\frac{B}{2}$ gives:

$$R = \sqrt{\left(-\frac{A}{2}\right)^2 + \left(-\frac{B}{2}\right)^2 - D}.$$

To better see why this is the case consider viewing the left hand side of the initial circle equation (3.1) as follows:

$$\left(p_{i,x} - \left(-\frac{A}{2}\right)^2\right)^2 + \left(p_{i,y} - \left(-\frac{B}{2}\right)^2\right)^2 + D - \left(-\frac{A}{2}\right)^2 - \left(-\frac{B}{2}\right)^2$$

Now, knowing that each term of $||Mt + z||^2$ represents just that, we immediately see that we actually minimized exactly the error function E and thus we get the C and R such that the residual is minimized as much as possible.

The time complexity of the algorithm is as follows:

- Constructing the matrix M and vector z requires $O(n)$ operations.
- Computing $M^T M$ (a 3×3 matrix) requires $O(n)$ operations.
- Solving the 3×3 linear system (matrix inversion and multiplication) requires $O(1)$ operations. Note that we don't need to compute the inverse of the $M^T M$ matrix, we can simply use LU decomposition to solve the linear system of equations.

We end up with the center of the circle $C = (C_x, C_y)$ and the radius of the circle R, which is everything we need for the next step.

Secondly, we will take any point on the circle and rotate it by the angle $\frac{2\pi}{n}$, which is the central

angle of a n-gon. The rotation can be done numerically by multiplying with a complex number that represents the rotation by some angle. Let $(x, y) \in \mathbb{R}$ be an arbitrarily chosen point on the circle and consider the complex number $x + yi$ associated with it. Now, the following computation to rotate this point by an angle of $\frac{2\pi}{n}$ can be done:

$$(x+iy)\cdot\left(cos\left(\frac{2\pi}{n}\right)+isin\left(\frac{2\pi}{n}\right)\right) = xcos\left(\frac{2\pi}{n}\right)-ysin\left(\frac{2\pi}{n}\right)+i\left(xsin\left(\frac{2\pi}{n}\right)+ycos\left(\frac{2\pi}{n}\right)\right),$$

which yields us a coordinate

$$\left(xcos\left(\frac{2\pi}{n}\right)-ysin\left(\frac{2\pi}{n}\right),\ xsin\left(\frac{2\pi}{n}\right)+ycos\left(\frac{2\pi}{n}\right)\right).$$

We repeat this procedure n times or until the rotated point becomes as close to the original point as machine precision allows, which indicates that we've inscribed a regular n-gon and we are done with this step. Note, that this procedure can be done in $O(n)$ time.

Thirdly, we will prove a protocol for checking whether the points can come from a polygon, as required by the Goals section of the CBL description. For each of the points $p_i = (x_i, y_i), i \in \{1, 2, \ldots, n\}$ that were given to us, we check which one of the following holds:

1. Point is outside of the circle. For this we check if $dist(p_i, C) > R$. If so, then it is outside of the circle and inside otherwise.

2. Point is inside of the polygon. For this, we have to take more complicated measures. We take a reference point C (center of the polygon). We use binary search (O(log n)) on the vertices V_i of the polygon (in their natural order) and in each iteration check whether the point lies to the left or to the right of the vector $(V_{i,x} - C_x, V_{i,y} - C_y)$. The orientation can be checked by using the 2D cross product (we will not go into the details here, the point is that the computation is possible and $O(1)$). At the end of the binary search we end up knowing a sector where the point lies in. What's left is to take the common side of the sector and our polygon and check whether the points lies on the left or to the right of it (taking the vector in the direction of the natural order of the vertices of the polygon). Depending on the order of the vertices of the polygon we conclude if the point is in the polygon. If so, we are done.

3. Point is in between the circle and the polygon. This is the last option that is automatically correct if the other two are not.

Note: We need to split into these 3 cases, because otherwise we are blind of where the points p_i are and simply using disjunction between the inequality bounds (see in the later part) does

not work. For example, if a point is very close to the center, knowing that ϵ is larger than the gap between the polygon and circle does not guarantee the possibility of the point lying on the boundary of a polygon, but if aforementioned point is in the gap between the circle and the polygon, it does guarantee that.

After that, we check if the point p_i can be placed on the boundary of a polygon. For that we need to know the largest distance from the boundary of a polygon to the circle's circumference, call it Δ. It is computed as follows: $\Delta = R(1-cos(\frac{\pi}{n}))$. We again distinguish this into 3 cases:

1. Point is outside of the circle. We check if $\epsilon \geq dist(p_i, C) - R + \Delta$, as $dist(p_i, C) - R + \Delta$ is the largest distance from the point outside the circle to the boundary of the polygon. If so, we conclude that the point can lie on the boundary of a polygon and continue with the next point.

2. Point is inside of the polygon. We check if $\epsilon \geq R - dist(p_i, C)$, as $R - dist(p_i, C))$ this is the largest distance from the point inside of the polygon to the boundary of the polygon. -,-

3. Point is in between the circle and the polygon. We check if $\epsilon \geq \Delta$, as Δ is the largest distance from the point in between circle's circumference and the polygon. -,-

If all the results are return positive, we conclude that the points can lie on the polygon.

3.2.2 Conclusive Remarks

This method differs from all the other ones in this report, because it allows for false negative to occur, i.e., we could say that it is impossible that these points have originated from a regular polygon, but that might just be the wrong choice of the polygon (since we started with an arbitrary point on the circle). We could have also chosen a polygon not only with a different orientation, but with the different number of sides. However, if we say that it could have originated from a polygon, it is certainly true, since we constructed a polygon and showed that if we leverage the errors, all points could end up on the boundary of it.

3.3 A Secondary Interpretation

3.3.1 Intial Description

This interpretation focuses on verifying if a given set of points $P = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$ can form a regular polygon using the points as vertices. Here we will use the characterization of regular polygons having equal side lengths and equal internal angles. Firstly, we will show an easy method of checking whether the given points can form a regular polygon by considering all the possible ways to connect the vertices to form a polygon and checking whether it is equilateral and equiangular. The second algorithm described here is a little bit more involved and uses the Convex Hull Algorithm to find the convex hull followed by the incorporation of the interior points to the convex hull to form a n-gon (not necessarily regular). Furthermore, it compares the side lengths and interior angles of the polygon to figure out whether there could be a way of constructing a regular polygon out of them, in a similar way as the easier version of this algorithm. The former method (with some assumptions) runs in $O(n^2)$, whereas the latter method (with some assumptions) runs in $O(n \log n)$ time and provides a sub-optimal way of verifying if points can form a regular polygon.

3.3.2 Method 1

The following offers a straightforward yet exhaustive approach to verifying the regularity of polygons. This method involves evaluating all possible configurations of the given points to determine whether they can form a regular polygon. By checking the side lengths and angles for equilateral and equiangular properties, this method assesses the possibility of regularity of polygon under consideration.

3.3.3 Polynomial Equations For Regular Polygons

Given points P we consider the side lengths and interior angles of the polygon. The side length equations are the distances between any two points:

$$s_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}, \quad \text{for } i, j \in \{1, \ldots, n\}, i \neq j. \tag{3.2}$$

Now, assuming we have found the sides of the polygon, we can order the vertices and take triplet $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ as consecutive vertices of a polygon. Here we only need

to check for angles in the following way:

$$\begin{aligned}\cos(\theta_i) &= \frac{v_{i-1} \cdot v_{i+1}}{\|v_{i-1}\|\|v_{i+1}\|}, \quad \text{for } i \in \{2, \ldots, n-1\},\\ \cos(\theta_1) &= \frac{v_n \cdot v_2}{\|v_n\|\|v_2\|},\\ \cos(\theta_n) &= \frac{v_{n-1} \cdot v_1}{\|v_{n-1}\|\|v_1\|}.\end{aligned} \tag{3.3}$$

As n increases, the number of side equations grows quadratically and the number of angle equations grows linearly. However, if we do not make the assumption that sides of the polygon were already found, the number of equations would grow cubicly, since there are $\binom{n}{3} = \frac{n(n-1)(n-2)}{6}$ total angles to check.

3.3.4 Error Model

The error model assumes perturbed coordinates $(\bar{x}_i, \bar{y}_i)$ such that:

$$(\bar{x}_i, \bar{y}_i) = (x_i, y_i) + \epsilon_i, \quad \|\epsilon_i\| < \epsilon. \tag{3.4}$$

For side lengths, the error propagation is bounded by:

$$|\Delta s| = |s - \bar{s}| \leq 2\epsilon. \tag{3.5}$$

Where $s, \bar{s}$ are any true and perturbed side lengths respectively. We now will prove that the inequality is correct. Let $(\bar{x}_i, \bar{y}_i), (\bar{x}_{i_1}, \bar{y}_{i+1}), (x_i, y_i), (x_{i_1}, y_{i+1})$ be two adjacent points given to us with their true counterparts in some polygon. Call the points $\bar{p}_i, \bar{p}_{i+1}, p_i, p_{i+1}$ Consider the following:

$$\begin{aligned}|\Delta s| = |s - \bar{s}| = |\bar{s} - s| &= |dist(\bar{p}_i, \bar{p}_{i+1}) - dist(p_i, p_{i+1})\\ &= |dist(p_i + \epsilon_i, p_{i+1} + \epsilon_{i+1}) - dist(p_i, p_{i+1})|\\ &\leq |dist(p_i, p_{i+1}) + dist(p_{i+1}, p_{i+1} + \epsilon_{i+1}) + dist(p_i, p_i + \epsilon_i) - dist(p_i, p_{i+1})|\\ &= |dist(p_{i+1}, p_{i+1} + \epsilon_{i+1}) + dist(p_i, p_i + \epsilon_i)|\\ &\leq |dist(p_{i+1}, p_{i+1} + \epsilon_{i+1})| + |dist(p_i, p_i + \epsilon_i)|\\ &= \|\epsilon_i\| + \|\epsilon_{i+1}\|\\ &\leq \epsilon + \epsilon\\ &= 2\epsilon\end{aligned}$$

We conclude that the bound is correct. It gets a little bit more complicated for angles. Consider the following set up:

We take 3 consecutive points $(\bar{x}_{i-1}, \bar{y}_{i-1}), (\bar{x}_i, \bar{y}_i), (\bar{x}_{i+1}, \bar{y}_{i+1})$ for $i \in \{2, \dots, n-1\}$ (and corresponding points for $i = 1, n$). They are the measured values and hence, have a deviation from the true error described as follows:

$$\begin{aligned}(\bar{x}_{i-1}, \bar{y}_{i-1}) &= (x_{i-1}, y_{i-1}) + \epsilon_{i-1},\\ (\bar{x}_i, \bar{y}_i) &= (x_i, y_i) + \epsilon_i,\\ (\bar{x}_{i+1}, \bar{y}_{i+1}) &= (x_{i+1}, y_{i+1}) + \epsilon_{i+1},\end{aligned}$$

Define the vectors:

$$\begin{aligned}v'_{i-1} &= (\bar{x}_{i-1} - \bar{x}_i, \bar{y}_{i-1} - \bar{y}_i),\\ v'_{i+1} &= (\bar{x}_{i+1} - \bar{x}_i, \bar{y}_{i+1} - \bar{y}_i),\\ v_{i-1} &= (x_{i-1} - x_i, y_{i-1} - y_i),\\ v_{i+1} &= (x_{i+1} - x_i, y_{i+1} - y_i).\end{aligned}$$

Let θ be the true angle calculated by $\theta = \frac{\pi(n-2)}{n}$ and θ' be the angle between the measured edges (at point $(\bar{x}_i, \bar{y}_i)$).

We claim that the perturbation bound is given by:

$$|\Delta\theta| = |\theta - \theta'| \leq 2 \arcsin\left(\frac{\epsilon}{\|{v'}_{i-1}\|}\right) + 2 \arcsin\left(\frac{\epsilon}{\|{v'}_{i+1}\|}\right). \tag{3.6}$$

To argue the bound we have to refer to the image below. Note, that colored sides are the potential sides of our actual polygon, but not necessarily the ones that give the largest or the smallest angle (even though we tried to make it that way), this figure is merely there to provide some visuals next to the discussion. We firstly look at the first term in the bound. We now only consider the perturbation of the point $(\bar{x}_{i-1}, \bar{y}_{i-1})$ and keep $(\bar{x}_i, \bar{y}_i)$ fixed. Imagine drawing a tangent from $(\bar{x}_i, \bar{y}_i)$ to the circumference of the error ball of point $(\bar{x}_{i-1}, \bar{y}_{i-1})$. I claim that this gives the largest error perturbation out of all similar configurations. The angle perturbation is then $\arcsin\left(\frac{\epsilon}{\|{v'}_{i-1}\|}\right)$. Now, we have to take into account the movement of the point $(\bar{x}_i, \bar{y}_i)$ and for that my claim is that it is equivalent to moving a point $(\bar{x}_{i-1}, \bar{y}_{i-1})$ in the opposite direction introducing another angle shift of $\arcsin\left(\frac{\epsilon}{\|{v'}_{i-1}\|}\right)$ (this actually skips one small step of the

angles being the same after the shift), which gives the factor of 2. We can do the same thing for the second point, namely, $(\bar{x}_{i+1}, \bar{y}_{i+1})$, which yields us the second term of the inequality. We admit the hand wavy explanation and it should be taken with a grain of salt. However, that is precisely the reason why we decided to find a bound on the angle rigorously.

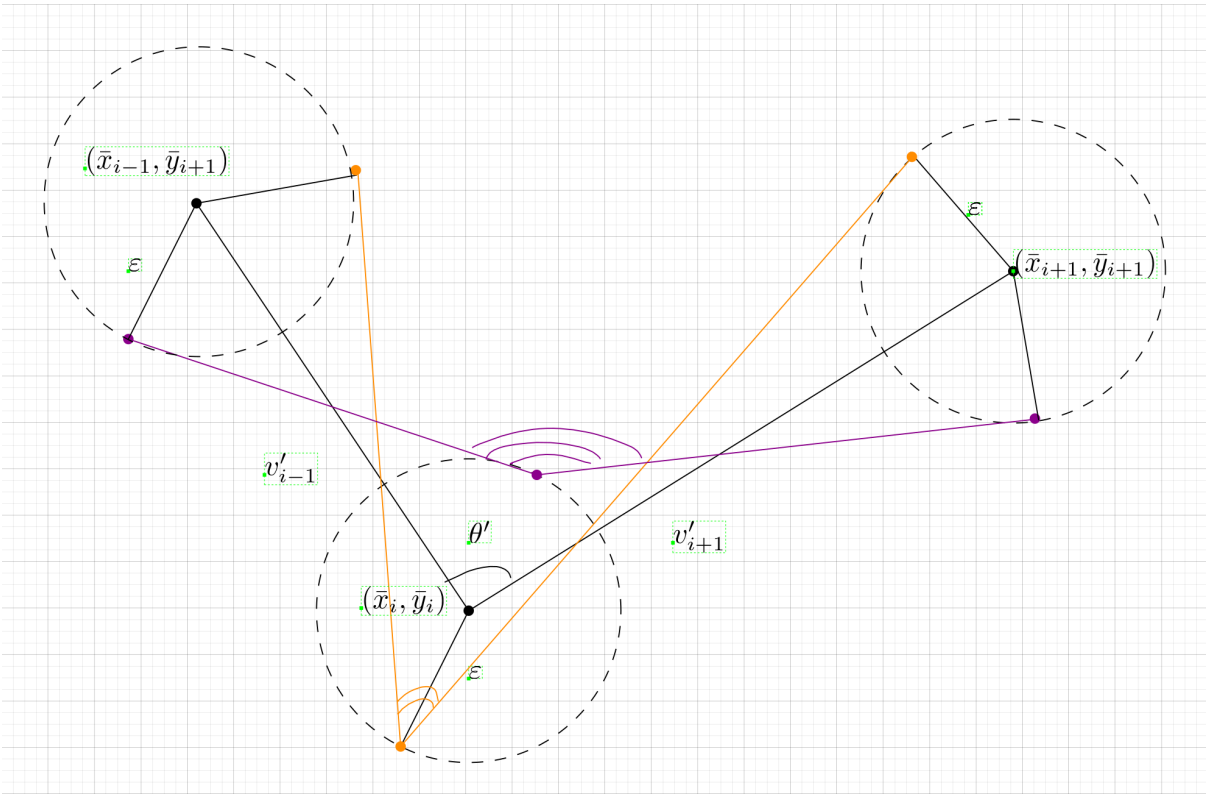

Figure 3.1: Angle perturbation

For convenience, we use $v_1 = v_{i-1}$, $v_2 = v_{i+1}$, $v'_1 = v'_{i-1}$, $v'_2 = v'_{i+1}$, $c_1 = \cos(\theta)$, $c_2 = \cos(\theta')$. We get the following bound rigorously:

$$|\cos(\theta) - \cos(\theta')| \leq \frac{4\epsilon}{\|v'_1\|} + \frac{4\epsilon}{\|v'_2\|} + \frac{24\epsilon^2}{\|v'_1\|\|v'_2\|}. \tag{3.7}$$

The derivation of the bound is on the page after the next one and is quite extensive. We suggest glossing over it (unless you have time), as it is merely there to suggest the direction we were taking and to show that we can a bound derived rigorously. It somewhat the resembled the bound made by observation as we have an error bound ϵ in the nominator and length of the vectors in the denominator. Besides that, we have a second order term with a factor of 24, but if we take some time to analyze it we can see that it does not make the bound tighter by a lot. Side lengths being 10 units long and error bound being 0.5 contributes 0.06 units which is almost negligible when considering that we are bounding the different in cosines (turns out to

be around $5°$). However, the first two terms show a different story. The reason why this whole bound is not very tight is precisely because we have the factor of 4 in the first two terms. Taking the lengths to be 10 units longs and error bound 0.5 as before, we get that the first two terms contribute 0.4 units to the cosine difference, which when added to the previous term results in 0.46 units for the difference of cosines (turns out to be around $50°$. For such a small error, this is a big perturbation of the angle. For the same example, the $arcsin$ bound yields $11°$, which is a fivefold decrease, which makes it a clear priority (if correct).

$$
\begin{aligned}
|c_1 - c_2| &= \left| \frac{v_1 \cdot v_2}{||v_1|| ||v_2||} - \frac{v_1' \cdot v_2'}{||v_1'|| ||v_2'||} \right| \\
&\leq \frac{|||v_1'|| ||v_2'|| v_1 \cdot v_2 - ||v_1|| ||v_2|| v_1' \cdot v_2'|}{||v_1'|| ||v_2'|| ||v_1|| ||v_2||} \\
&\leq \frac{|(||v_1|| + ||\delta v_1||)(||v_2|| + ||\delta v_2||) v_1 \cdot v_2 - ||v_1|| ||v_2|| v_1' \cdot v_2'|}{||v_1'|| ||v_2'|| ||v_1|| ||v_2||} \\
&\leq \frac{|(||v_1|| + ||\delta v_1||)(||v_2|| + ||\delta v_2||) v_1 \cdot v_2 - ||v_1|| ||v_2|| (v_1 \cdot v_2 + \delta v_1 \cdot v_2 + \delta v_2 \cdot v_1 + \delta v_1 \cdot \delta v_2)|}{||v_1'|| ||v_2'|| ||v_1|| ||v_2||} \\
&\leq \frac{|(||\delta v_1|| ||v_2|| + ||v_1|| ||\delta v_2|| + ||\delta v_1|| ||\delta v_2||) v_1 \cdot v_2 - ||v_1|| ||v_2|| (\delta v_1 \cdot v_2 + \delta v_2 \cdot v_1 + \delta v_1 \cdot \delta v_2)|}{||v_1'|| ||v_2'|| ||v_1|| ||v_2||} \\
&\leq \frac{|(||\delta v_1|| ||v_2|| + ||v_1|| ||\delta v_2|| + ||\delta v_1|| ||\delta v_2||) ||v_1|| ||v_2|| - ||v_1|| ||v_2|| (\delta v_1 \cdot v_2 + \delta v_2 \cdot v_1 + \delta v_1 \cdot \delta v_2)|}{||v_1'|| ||v_2'|| ||v_1|| ||v_2||} \\
&\leq \frac{|||v_1|| ||v_2|| ((||\delta v_1|| ||v_2|| + ||v_1|| ||\delta v_2|| + ||\delta v_1|| ||\delta v_2||) - (\delta v_1 \cdot v_2 + \delta v_2 \cdot v_1 + \delta v_1 \cdot \delta v_2))|}{||v_1'|| ||v_2'|| ||v_1|| ||v_2||} \\
&\leq \frac{|(||\delta v_1|| ||v_2|| + ||v_1|| ||\delta v_2|| + ||\delta v_1|| ||\delta v_2||) - (\delta v_1 \cdot v_2 + \delta v_2 \cdot v_1 + \delta v_1 \cdot \delta v_2)|}{||v_1'|| ||v_2'||} \\
&\leq \frac{|(||\delta v_1|| ||v_2|| + ||v_1|| ||\delta v_2|| + ||\delta v_1|| ||\delta v_2||)| + |(\delta v_1 \cdot v_2 + \delta v_2 \cdot v_1 + \delta v_1 \cdot \delta v_2)|}{||v_1'|| ||v_2'||} \\
&\leq \frac{|(||\delta v_1|| ||v_2|| + ||v_1|| ||\delta v_2|| + ||\delta v_1|| ||\delta v_2||)| + |(||\delta v_1|| ||v_2|| + ||v_1|| ||\delta v_2|| + ||\delta v_1|| ||\delta v_2||)|}{||v_1'|| ||v_2'||} \\
&\leq \frac{2(|(||\delta v_1|| ||v_2|| + ||v_1|| ||\delta v_2|| + ||\delta v_1|| ||\delta v_2||)|)}{||v_1'|| ||v_2'||} \\
&\leq \frac{2(||\delta v_1|| ||v_2|| + ||v_1|| ||\delta v_2|| + ||\delta v_1|| ||\delta v_2||)}{||v_1'|| ||v_2'||} \\
&\leq \frac{2(||\delta v_1|| (||v_2'|| + ||\delta v_2||) + (||v_1'|| + ||\delta v_1||) ||\delta v_2|| + ||\delta v_1|| ||\delta v_2||)}{||v_1'|| ||v_2'||} \\
&\leq \frac{2(||v_2'|| ||\delta v_1|| + ||\delta v_1|| ||\delta v_2|| + ||v_1'|| ||\delta v_2|| + ||\delta v_1|| ||\delta v_2|| + ||\delta v_1|| ||\delta v_2||)}{||v_1'|| ||v_2'||} \\
&\leq \frac{2(||v_2'|| ||\delta v_1|| + ||v_1'|| ||\delta v_2|| + 3||\delta v_1|| ||\delta v_2||)|}{||v_1'|| ||v_2'||} \\
&\leq \frac{2(||v_2'|| \cdot 2\epsilon + ||v_1'|| \cdot 2\epsilon + 3 \cdot 2\epsilon \cdot 2\epsilon)}{||v_1'|| ||v_2'||} \\
&\leq \frac{2(||v_2'|| \cdot 2\epsilon + ||v_1'|| \cdot 2\epsilon + 12\epsilon^2)}{||v_1'|| ||v_2'||} \\
&\leq \frac{||v_2'|| \cdot 4\epsilon + ||v_1'|| \cdot 4\epsilon + 24\epsilon^2}{||v_1'|| ||v_2'||} \\
&\leq \frac{4\epsilon}{||v_1'||} + \frac{4\epsilon}{||v_2'||} + \frac{24\epsilon^2}{||v_1'|| ||v_2'||}
\end{aligned}
$$

3.3.5 Algorithm For Regular Polygon Verification

We now move to the algorithm for finding whether the given perturbed points could have originated from vertices of a polygon. The algorithm consists of the following steps:

1. Compute pairwise side lengths s_{ij} for all $i \neq j$. This gives us $\binom{n}{2}$ side lengths to work with.
2. Sort the side lengths by size in increasing order.
3. Use sliding window approach with size n starting with the smallest n side lengths to check whether a polygon can be constructed with them. If a working configuration is found, move to the next step.
4. Compute angles at each vertex using dot products of adjacent edge vectors.
5. Check that all angles are within the angle perturbation bound difference (using the error model) from the true angle (that we know).

The assumption to be made is that the points given to us are relatively spaced out and have small enough errors. Giving the bound for the errors to satisfy this vague assumption turns out to be complicated, so we will skip to the meat of the algorithm instead. The first 2 steps are trivial, although keep in mind that at the end of them we do not get n^2 side lengths, but $\binom{n}{2}$ as we don't take the pairs to be reflexive. The third steps takes a window of size n, which initially contains all the smallest n vertices. It might look like that it is the only thing we need to check, because for a regular polygon the smallest n pairwise distances are indeed the side lengths, but it might not be the case here, when we have an error around each point to consider. This means that there might be a number of small distances coming from one point that is close to a few other points and distort the sequence of side lengths. In each iterations of the sliding window we check whether or not the largest side length and smallest side length differ by at most 4ϵ, because this indicates that the polygon could contain equal side lengths, since in the worst case the largest side could have originated from a side of length 2ϵ smaller and the shortest side could have originated from a side 2ϵ longer. We also check whether or not each point contributed to exactly 2 side lengths. For that, next to each side length we would have to store the points corresponding to it, which would make the initial sorting of the points a little bit more complicated, since we have to introduce a custom data structure, however, it is still feasible and could be implemented without any hurdles if needed. One thing we glossed over was the fact

that we take n consecutive points. As you might think that there could be a configuration such that we take $k > 0$ consecutive points of some subarray and then $n - k$ consecutive points of another subarray to get a valid configuration. This we assume never happens. One could somehow argue that this assumption then also invalidates the need to consider all n consecutive vertices instead of the first n and while this might be the case, we simply leave it at that. If we had considered also taking non-consecutive vertices it seems to us that this would be similar to finding a Hamiltonian Cycle in a complete graph K_n of n nodes, them being the points (with a constraint that you can't have edges whose lengths, or weights so to say, differ by at least 4ϵ), which is NP-complete and thus makes it unreasonable to use here. Moreover, note that we terminate after finding a valid configuration, so we do not consider the case where there might be more configurations due to the looseness of the errors). Last thing to mention on step 3 would be that this step does not change the time complexity of the whole algorithm, since sliding window has the same time complexity as the array it is used on, in this case it would run in $O(n^2)$ time. Finally, moving on to steps $4, 5$ which are straightforward. For the valid configuration that we've found (and we assume it is only one) we compute the angles for each adjacent sides and use an angle perturbation bound found before to check whether there is a possibility of making the true angles by leveraging the errors that we have. If there there is such a possibility we conclude that a regular polygon could have been made.

3.3.6 Algorithm Complexity:

The algorithm involves $O(n^2)$ side length comparisons and $O(n)$ angle calculations, resulting in an overall complexity of $O(n^2)$.

3.3.7 Closing Observations

This approach provides a systematic way to verify the regularity of polygons under coordinate perturbations and the assumption of non-degenerate points. By combining polynomial equations with an error model, we ensure robustness while maintaining a manageable computational complexity. For larger n, the quadratic growth in side length checks becomes the limiting factor, hence, we have to switch to a quicker method, but at the same time a more complicated method.

3.3.8 Method 2

The following method introduces a more geometric approach, incorporating the Convex Hull Algorithm to address scenarios where some points may lie inside the polygon. This method builds on the concepts introduced in Method 1 but adopts a more systematic strategy to verify polygon regularity.

3.3.9 Algorithm For Regular Polygon Verification

We will use the same error model as in **Method 1** and the polynomial equations that we will have are very similar to **Method 1**, but instead of checking side lengths s_{ij} for $i, j \in \{1, \ldots, n\}$, $i \neq j$ we will only need to check n such side lengths as the algorithm will return all vertices in the natural order of the convex polygon. Thus, we can instantly skip to the algorithm itself that we also implemented in C++ (provided in the abstract). The algorithm outlined in the implementation `Polygons.cpp` does the following:

1. Uses Graham's scan to compute the convex hull of the input points.
2. If all of the points are in the convex hull it immediately skips to the next step. However, if there are some interior points, it finds the shortest distance to some side of the polygon and inserts that point in between the vertices that formed that side if the distance to that side was smaller than 2ϵ. If the smallest distance to the boundary of the polygon is larger than 2ϵ the algorithm terminates. If not, it continues to do the same for all of the interior points.
3. It then proceeds to use angle and side length regularity conditions to verify whether or not it is possible to make a regular polygon based on them.

There is an important assumption to be made for this algorithm to work that we will cover later, but for now, lets talk about this algorithm in more general sense. Computing the convex hull helps us to find the convex set and the set of interior points. The convex set is really useful, especially if it contains all of the given points, because then we have a blueprint for our regular polygon as every regular polygon is convex. If the convex hull does not contain all of the points, it means that you can't form a polygon will all interior angles being smaller than $180°$ and thus more work is needed. to see why this is the case, we proceed to give a short outline of the Convex Hull Algorithm (which is implemented in the function `convex_hull_computation`

of our program):

1. Identify the point p_0 with the lowest y-coordinate. In case of a tie, the leftmost point is chosen.
2. Sort points in counter-clockwise order based on their polar angle relative to p_0. This is done by defining a new sorting function.
3. Construct the convex hull by iterating through sorted points and applying orientation tests to eliminate clockwise turns.

We can see that eliminating clockwise turns thus ensures all interior angles are less than $180°$. What is left is to consider the situation where the number of interior angles is not zero. In that case we must check whether the smallest distance to any of the sides of the polygon is smaller than 2ϵ. If it is not, then there is no possibility of making the set of points convex by leveraging the errors and thus, making the set of points the vertices of a polygon. If it is the case, then we can still make a convex polygon out of that (and perhaps a regular one as well) by leveraging the errors. We omit the description of the calculation of the distances in this report as they are part of the `Polygons.cpp`, specifically, in function `check_convex_correctness`. To see why we check for 2ϵ instead of simply ϵ we have to look at the image below, lets focus on the interior point with its error radius of length ϵ and vertices to which it is closest to. We can see that the distance between the closest side length is larger than epsilon, however if we move the top left vertex and interior point to specific positions (the points colored magenta) within the error bounds we can obtain a convex polygon. Hence, we not only need to account for the interior point translations, but also the translations of the vertices to which the point is closest as well. This is precisely the reason for checking with 2ϵ. Now, whenever we checked this condition for the interior point, we still need to know where to insert it. This turns out to be a problem of its own that we don't have a satisfactory answer to. We claim that for sufficiently small epsilon, it is always optimal to insert the vertex to the side to which it is closest to. We do not prove this here, but assume that such an epsilon exist and that the data provided to us satisfies such a condition. We repeat the insertion procedure for all interior points that could be part of the convex polygon and if there exist an interior point that does not satisfy the condition, the algorithm terminates. After all, we get n ordered vertices and we proceed to check the regularity conditions for side lengths and angles as we did in **Method 1**, but with $O(n)$ time complexity as we only need to find the shortest and largest side lengths, alongside

checking the conditions for angles. This is done in the function `check_valid_polygon` in our program.

3.3.10 Algorithm Complexity:

The time complexity of this algorithm is $O(n\ log\ n)$. The reason is that Convex Hull Algorithm itself has time complexity, because it sorts the point by the inverse slope and the lower bound of any comparison sorting function exactly that. Moreover, if there are a lot of interior points, then checking the shortest distance to the boundary of the polygon for all of the points might take $O(n^2)$ as you have to iterate through all of the sides of the polygon for each of the interior points. To optimize this function and reduce the time complexity from $O(n^2)$ to $O(log\ n)$ we can use ternary search to find the side where the minimum distance between the interior point and the polygon side occurs. Ternary search is applicable here because the distance function between a fixed point and a line segment (in the context of convex polygons) is unimodal, meaning it decreases to a minimum and then increases. Ternary search, just like binary search, runs in $O(log\ n)$ time and thus this portion of the algorithm results in a worst case $O(n\ log\ n)$ time complexity. All other parts of the program do not heavily contribute to the time complexity and thus we can conclude that we can find a

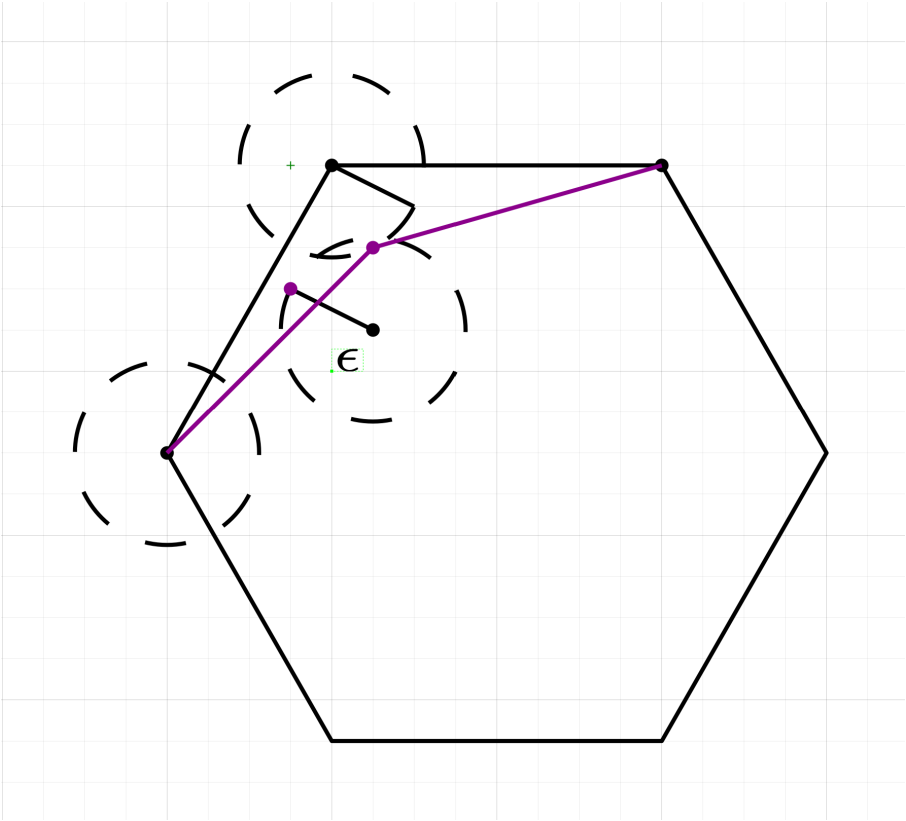

Figure 3.2: Example of an interior point

3.3.11 Further Commentary

The algorithm provides a method to verify the regularity of polygons under measurement errors with sub-optimal time complexity of $O(n \log n)$. By combining convex hull analysis with perturbation bounds and assumptions on the errors and nature of the given points we deduce whether it is possible to construct a regular polygon or not without introducing false negatives (false positive are obviously possible). Further refinements could incorporate tighter bounds and proofs of the claims that were used, but not proven.

3.4 A Second Probabilistic Approach

Similar to our procedure for the case of right-angled triangles, we may attempt a probabilistic output that can complement our discussion on deterministic numerical bounds. We seek to develop this facet through construction via Measure-theoretic Probability. Initially, we will use conventional elementary assumptions to produce a probabilistic deduction and we will use coordinate error translation invariance as previously to follow suit and create a commentary on

our probability results.

Proposition 3 (High Probability Of Bounded Error - Polygon Edition) *Our points in Euclidean Space have errors lie in the translation-invariant interval* $(-\varepsilon, \varepsilon)$ *with relatively high probability.*

Proof. Once more, we argue by theory found in [Royden and Fitzpatrick, 1963] and [Billingsley, 1999]. This time, we define a product measure space such that $(\Omega, \mathcal{F}, \mu) = ([0,1]_1 \times [0,1]_2 \cdots \times [0,1]_{2n}, \mathcal{B}_{1|[0,1]} \otimes \mathcal{B}_{2|[0,1]} \cdots \otimes \mathcal{B}_{2n|[0,1]}, \lambda_{1|[0,1]} \otimes \lambda_{2|[0,1]} \cdots \otimes \lambda_{2n|[0,1]})$ that acts as a universal probability space, where $\mathcal{B}_{i|[0,1]}$ denotes the Borel σ-algebra of $\mathbb{R}$ restricted to the interval $[0,1]$ and $\lambda_{i|[0,1]}$ denotes the Lebesgue measure restricted to the same interval. Observe that our space is σ-finite and our product measure is unique by the Carathéodory Extension Theorem. Moreover, let $\pi_i : [0,1]_1 \times [0,1]_2 \times \ldots [0,1]_{2n} \mapsto [0,1]_i$ denote the canonical projection maps from the product sample space to the corresponding sample space. Subsequently,

we aim to use the Inverse Sampling Method to construct independent random variables that describe a probabilistic perspective of our error bounds. Let F_{X_i} denote the generalised inverse of the i-th random variable and U_i denote the standard unit uniform random variable. Note that the generalised inverse is a function given by

$$F_{X_i}^{-1}(p) := \inf\{x \in \mathbb{R} : F_{X_i}(x) \geq p\} \quad \forall p \in [0,1].$$

We then define a family of Bernoulli random variables $\{X_i\}_{i \in I}$ scaled by an indicator function $\mathbb{1}_{\{\varepsilon_i \notin (-\varepsilon, \varepsilon)\}}$ by

$$X_i := F_{X_i}^{-1} \circ U_i \circ \pi_i.$$

By virtue of this construction, each random variable has a Cumulative Distribution Function of the form:

$$F_{X_i}(t) = \mathbb{P}(X_i \leq t) = \begin{cases} 1-p & t = 0, \\ p & t = 1, \end{cases}$$

where $p > 0$ is small and each $1 \leq i \leq 2n$ corresponds to an element in Euclidean Space. Observe that due to our construction, these random variables are well-defined and model the

independent and identically distributed nature provided by the random sample assumption of our error bounds. Leveraging the independence of our random variable now allows us to examine the instance that our errors all fall within our predetermined error bound. Due to the factorization of our probability measures, it holds that:

$$\begin{aligned}\mathbb{P}(X_1 = 0, X_2 = 0, \dots X_n = 0) &= \Pi_{i=1}^{n}\mathbb{P}(X_i = 0) \\ &= (1-p)^n \\ &\text{(By Bernoulli's Inequality)} \\ &\leq 1 - np.\end{aligned}$$

Alternative to Bernoulli's Inequality [Portegies, 2023] we may also make use of the Binomial Theorem and quotient higher order terms as is within the proof of **??**. Observe that since p is strictly positive and sufficiently small by assumption, we find that

$$\mathbb{P}\{\varepsilon_i \in (-\varepsilon, \varepsilon)\} \leq 1 - np,$$

implying that our coordinate errors behave well with relatively high probability and that the probability of the event that our errors lie outside our bound is insignificant.

Accordingly, we conclude having demonstrated our proposed argument.

4 Recognizing circles

4.1 General overview

We are given a set $C := (x_1, y_1), (x_2, y_2), ..., (x_n, y_n) \subseteq \mathbb{R}^2$ of size n. The goal is to check whether these coordinates could have been obtained from points lying on a circle. In the case $n \leq 3$ there is always a circle passing through any three points, so we assume $n \geq 4$.

4.2 Exact Arithmetic Algorithms

To begin with, we aim to find a collection of polynomial equations that can be used to test whether the points in C lie on a circle, assuming the coordinates are exact.

Our strategy will be to use the 3-point form of a circle equation. Let $(x_1, y_1), (x_2, y_2)$ and (x_3, y_3) be three points that are not on a line. Then the equation of the circle that passes through these points is given by

$$\frac{((x - x_1)(x - x_2) + (y - y_1)(y - y_2)}{(y - y_1)(x - x_2) - (y - y_2)(x - x_1)} = \frac{(x_3 - x_1)(x_3 - x_2) + (y_3 - y_1)(y_3 - y_2)}{(y_3 - y_1)(x_3 - x_2) - (y_3 - y_2)(x_3 - x_1)}.$$

Now we want to rearrange the terms in a more convenient way. To do so, we first cross-multiply and then subtract from the LHS the RHS of the equation. As a result we obtain the polynomial

$$\begin{aligned} f(x, y) = &[(x - x_1)(x - x_2) + (y - y_1)(y - y_2)][(y_3 - y_1)(x_3 - x_2) - (y_3 - y_2)(x_3 - x_1)] \\ &- [(x_3 - x_1)(x_3 - x_2) + (y_3 - y_1)(y_3 - y_2)][(y - y_1)(x - x_2) - (y - y_2)(x - x_1)]. \end{aligned}$$

Remark: $f(x, y) = 0$ if and only if the point (x, y) lies on the circle that passes through $(x_1, y_1), (x_2, y_2), (x_3, y_3)$.

Using this polynomial, we can develop an algorithm for checking whether the points in C lie on a circle. Assuming that the coordinates are exact, it suffices to choose any three points in C that are not on a straight line and check if all other points satisfy the 3-point equation of the circle that passes through the chosen points. I.e. we can check if $f(x_i, y_i) = 0$ for all $i \in \{1, ..., n\}$

where f is the corresponding polynomial defined as described above. In case that all points in C are on a line, we can conclude that there does not exist a circle that passes through all of the points.

As a result, our collection of polynomials consists of all polynomials of the form

$$\begin{aligned} f_{i,j,k}(x,y) = &[(x-x_i)(x-x_j)+(y-y_i)(y-y_j)][(y_k-y_i)(x_k-x_j)-(y_k-y_j)(x_k-x_i)] \\ &-[(x_k-x_i)(x_k-x_j)+(y_k-y_i)(y_k-y_j)][(y-y_i)(x-x_j)-(y-y_j)(x-x_i)] \end{aligned}$$

where $(x_i, y_i), (x_j, y_j), (x_k, y_k) \in C$ and $i \neq j \neq k$.

4.3 Error Analysis

In this part of the report we want to investigate what happens if the data contains some error and how the error behaves for the polynomial defined in the previous section. The setting is as follows:

- $(x_1, y_1), (x_2, y_2)$ and (x_3, y_3) are three points that are not on a line.
- T is the circle that passes through these points.

Assume that we are given the coordinates $\bar{x}, \bar{y}$ that are perturbed from the actual coordinates x, y by some error, i.e. the point $(\bar{x}, \bar{y})$ is an approximation of the point (x, y). We let $\epsilon_x := \bar{x} - x$ and $\epsilon_y := \bar{y} - y$ where $-\epsilon < \epsilon_x, \epsilon_y < \epsilon$. Now, we want to find a function g in terms of $\bar{x}, \bar{y}$ and ϵ such that if the point $(x, y) \in T$, then $|f(\bar{x}, \bar{y})| < g(\bar{x}, \bar{y}, \epsilon)$.

Remark: $(x, y) \in T \implies f(x, y) = 0$.

Assume that $(x, y) \in T$. We consider the polynomial

$$\begin{aligned} f(x,y) = &[(x-x_1)(x-x_2)+(y-y_1)(y-y_2)]\overbrace{[(y_3-y_1)(x_3-x_2)-(y_3-y_2)(x_3-x_1)]}^{\text{const. } A} \\ &-\underbrace{[(x_3-x_1)(x_3-x_2)+(y_3-y_1)(y_3-y_2)]}_{\text{const. } B}[(y-y_1)(x-x_2)-(y-y_2)(x-x_1)]. \end{aligned}$$

For simplicity we define the constants $A := (y_3 - y_1)(x_3 - x_2) - (y_3 - y_2)(x_3 - x_1)$, $B := (x_3 - x_1)(x_3 - x_2) + (y_3 - y_1)(y_3 - y_2)$ and $C := |A| + |B|$. For the derivation of the function g that satisfies $|f(x, y)| < g(\bar{x}, \bar{y}, \epsilon)$ for $(x, y) \in T$, we performed the following computations

$$
\begin{aligned}
|f(\bar{x},\bar{y})| &= |A[(\bar{x}-x_1)(\bar{x}-x_2)+(\bar{y}-y_1)(\bar{y}-y_2)] - B[(\bar{y}-y_1)(\bar{x}-x_2)-(\bar{y}-y_2)(\bar{x}-x_1)]| \\
&= |A[(x+\epsilon_x-x_1)(x+\epsilon_x-x_2)+(y+\epsilon_y-y_1)(y+\epsilon_y-y_2)] - B[(y+\epsilon_y-y_1)(x+\epsilon_x-x_2)-(y+\epsilon_y-y_2)(x+\epsilon_x-x_1)]| \\
&= |A[(x-x_1)(x-x_2)+\epsilon_x(x-x_1)+\epsilon_x(x-x_2)+\epsilon_x^2+(y-y_1)(y-y_2)+\epsilon_y(y-y_1)+\epsilon_y(y-y_2)+\epsilon_y^2] \\
&\quad - B[(y-y_1)(x-x_2)+\epsilon_y(x-x_2)+\epsilon_x(y-y_1)+\epsilon_x\epsilon_y-(y-y_2)(x-x_1)-\epsilon_y(x-x_1)-\epsilon_x(y-y_2)-\epsilon_x\epsilon_y]| \\
&= |A[\epsilon_x(x-x_1)+\epsilon_x(x-x_2)+\epsilon_x^2+\epsilon_y(y-y_1)+\epsilon_y(y-y_2)+\epsilon_y^2] - B[\epsilon_y(x-x_2)+\epsilon_x(y-y_1)+\epsilon_x\epsilon_y-\epsilon_y(x-x_1)-\epsilon_x(y-y_2)-\epsilon_x\epsilon_y] \\
&\quad + \underbrace{A[(x-x_1)(x-x_2)+(y-y_1)(y-y_2)] - B[(y-y_1)(x-x_2)-(y-y_2)(x-x_1)]}_{f(x,y)}| \\
&= |A[\epsilon_x(x-x_1)+\epsilon_x(x-x_2)+\epsilon_x^2+\epsilon_y(y-y_1)+\epsilon_y(y-y_2)+\epsilon_y^2] - B[\epsilon_y(x-x_2)+\epsilon_x(y-y_1)+\epsilon_x\epsilon_y-\epsilon_y(x-x_1)-\epsilon_x(y-y_2)-\epsilon_x\epsilon_y]| \\
&\leq |A||\epsilon_x(x-x_1)+\epsilon_x(x-x_2)+\epsilon_x^2+\epsilon_y(y-y_1)+\epsilon_y(y-y_2)+\epsilon_y^2| + |B||\epsilon_y(x-x_2)+\epsilon_x(y-y_1)+\cancel{\epsilon_x\epsilon_y}-\epsilon_y(x-x_1)-\epsilon_x(y-y_2)-\cancel{\epsilon_x\epsilon_y}| \\
&\leq |A|\epsilon|x-x_1| + |A|\epsilon|x-x_2| + |A|\epsilon^2 + |A|\epsilon|y-y_1| + |A|\epsilon|y-y_2| + |A|\epsilon^2 + |B|\epsilon|x-x_2| + |B|\epsilon|y-y_1| + |B|\epsilon^2 + |B|\epsilon|x-x_1| + |B|\epsilon|y-y_2| \\
&\leq C\epsilon|x-x_1| + C\epsilon|x-x_2| + C\epsilon|y-y_1| + C\epsilon|y-y_2| + C\epsilon^2 \\
&\leq C\epsilon|\bar{x}-\epsilon_x-x_1| + C\epsilon|\bar{x}-\epsilon_x-x_2| + C\epsilon|\bar{y}-\epsilon_y-y_1| + C\epsilon|\bar{y}-\epsilon_y-y_2| + C\epsilon^2 \\
&\leq C\epsilon(|\bar{x}-x_1| + |\epsilon_x|) + C\epsilon(|\bar{x}-x_2| + |\epsilon_x|) + C\epsilon(|\bar{y}-y_1| + |\epsilon_y|) + C\epsilon(|\bar{y}-y_2| + |\epsilon_y|) + C\epsilon^2 \\
&< C\epsilon(|\bar{x}-x_1| + |\bar{x}-x_2| + |\bar{y}-y_1| + |\bar{y}-y_2|) + 4C\epsilon^2 + C\epsilon^2 \\
&= C\epsilon(|\bar{x}-x_1| + |\bar{x}-x_2| + |\bar{y}-y_1| + |\bar{y}-y_2|) + 5C\epsilon^2 =: g(\bar{x},\bar{y},\epsilon)
\end{aligned}
$$

As a result we obtain a function g with the desired property.

4.4 Error Protocol

Now, our main goal is to develop a protocol for testing whether the points in C lie on a circle. However, the tricky part is that these points may contain some error. Fortunately, the result from section 4.3 will come in handy. To tackle this problem we will use our knowledge about the behavior of the error. As a result, we propose the following algorithm:

For every three points $(x_i, y_i), (x_j, y_j), (x_k, y_k) \in C$ such that $i \neq j \neq k$ we define the polynomials

$$f_{i,j,k}(x,y) = [(x-x_i)(x-x_j)+(y-y_i)(y-y_j)][(y_k-y_i)(x_k-x_j)-(y_k-y_j)(x_k-x_i)] \\ -[(x_k-x_i)(x_k-x_j)+(y_k-y_i)(y_k-y_j)][(y-y_i)(x-x_j)-(y-y_j)(x-x_i)]$$

$$g_{i,j,k}(x,y,\epsilon) = C\epsilon(|x-x_1|+|x-x_2|+|y-y_1|+|y-y_2|)+5C\epsilon^2$$

where C is a constant given by the following expression

$$C := |(y_3-y_1)(x_3-x_2)-(y_3-y_2)(x_3-x_1)|+|(x_3-x_1)(x_3-x_2)+(y_3-y_1)(y_3-y_2)|.$$

Then, we iterate through the points and check if $|f_{i,j,k}(x,y)| < g_{i,j,k}(x,y,\epsilon)$ for all $(x,y) \in C$. In case all points in C satisfy this condition for the polynomials $f_{i,j,k}$, $g_{i,j,k}$ the algorithm terminates and outputs "The points are obtained from a circle". Otherwise, the algorithm proceeds to the next 3-tuple of points.

If the algorithm did not terminate while checking the criterion for all 3-tuples, we know that the points in C are not obtained from points lying on a circle. In this case, the algorithm outputs "The points are not obtained from a circle".

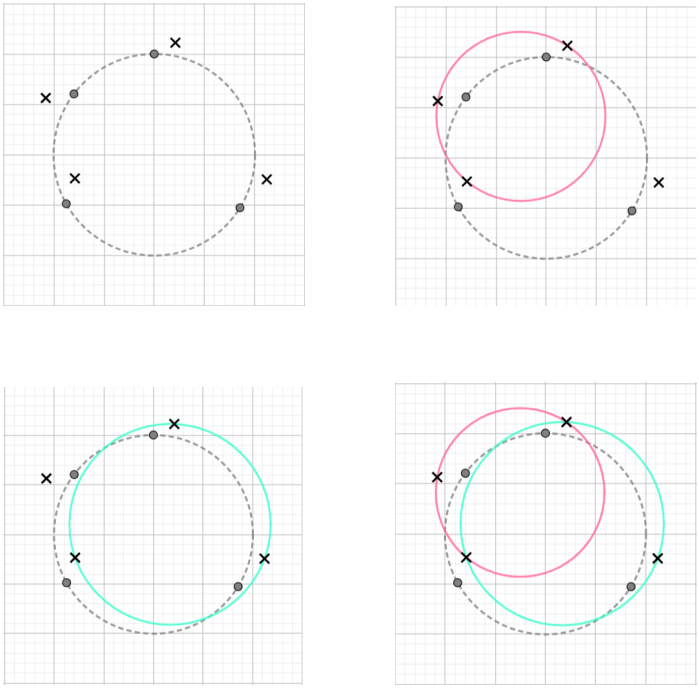

Figure 4.1: Two circles computed by the algorithm vs the circle from which the points originate

Since we want to estimate how efficient our algorithm is we will analyse its time complexity. We observe that if we consider every (ordered) 3-tuple of points $(x_i, y_i), (x_j, y_j), (x_k, y_k) \in C$ such that $i \neq j \neq k$ the algorithm has a running time of $O(n^4)$. However, we can improve that sixfold by exploiting the symmetry of the points. The main idea is that the 3-tuples $((x_i, y_i), (x_j, y_j), (x_k, y_k))$ and $((x_j, y_j), (x_i, y_i), (x_k, y_k))$ are inscribed within the same circle. As a result, it suffices to check $\{(x_i, y_i), (x_j, x_j), (x_k, y_k)\}$.

An interesting question is what is the largest number of points that the algorithm can handle. Assuming that the allowed run-time for the program is one second and given the fact that a modern computer can process around 5 billion instructions per second [BBC, 2023] [Padmanabhan, 2011], the algorithm should be able to work for up to 150 points. However, with the optimization due to the symmetry, this number could be pushed to 250. Furthermore, if we are able to run the program for 10 minutes, rather than a single second, we could process more than a thousand points.

4.5 Limitations Of The Algorithm

It is also important to investigate under what assumptions the algorithm proposed in section 4.4 behaves nicely. Since we have based our testing procedure on the 3-point form of a circle equation, it is reasonable to take a closer look at that expression once again.

$$\frac{((x - x_1)(x - x_2) + (y - y_1)(y - y_2)}{(y - y_1)(x - x_2) - (y - y_2)(x - x_1)} = \frac{(x_3 - x_1)(x_3 - x_2) + (y_3 - y_1)(y_3 - y_2)}{(y_3 - y_1)(x_3 - x_2) - (y_3 - y_2)(x_3 - x_1)}.$$

We observe that if the points $(x_1, y_1), (x_2, y_2)$ and (x_3, y_3) are 'close' to each other, then the term $(y_3 - y_1)(x_3 - x_2) - (y_3 - y_2)(x_3 - x_1)$ becomes 'very small'. i.e. the RHS of the equation tends to infinity. Therefore, our algorithm would be unpredictable in such situations.

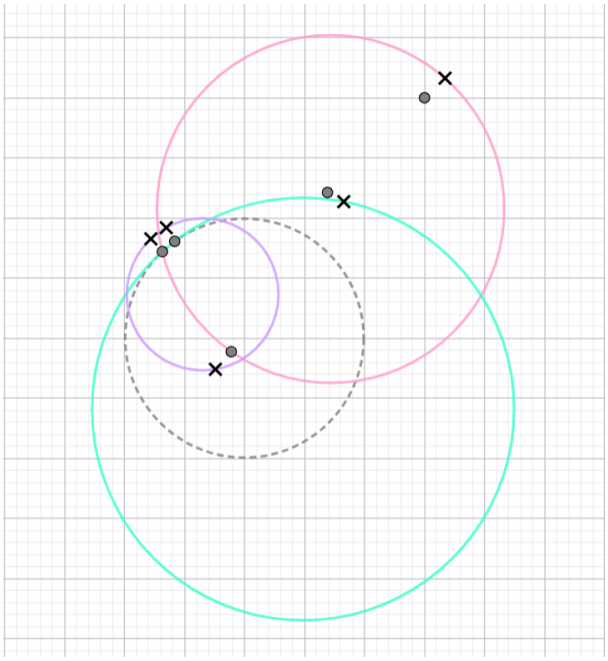

Figure 4.2: An example of a data that contains dense points

The above image represents a case in which the data is clearly not obtained from a circle and it also contains two points that are 'close' to each other. We ran our program with the exact coordinates of the given points (the ones represented by a cross) and $\epsilon = 0.3$ (since the distance between one of the original points and its approximation is almost equal to 0.3). Unfortunately, the algorithm concluded that the points are obtained from a circle. This confirms our statement that the dense points make the algorithm behave unpredictably.

In conclusion, the proposed algorithm is accurate only under the assumption that there are no dense points in the given data.

4.6 Optimization Of The Algorithm

In this section we propose one possible optimization of the algorithm. The idea is to sort the unordered 3-tuples by a certain criteria, instead of taking them at random. More precisely, we sort in non-increasing order the unordered 3-tuples by the sine of the largest angle in the triangle formed by these points plus $\frac{\pi}{6}$ radians. Assuming that the data is obtained from a circle, the intuition behind our idea is that a 3-tuple consisting of points that are 'equally' distributed on the circle, will produce a circle that is 'closer' to the one on which the original points lie on. We used the sine of the angle plus $\frac{\pi}{6}$ radians, since its maximum value is achieved when the

largest angle is $\frac{\pi}{3}$ radians, which implies that the points are 'equally' distributed on the circle.

We define an array Arr that consist of all unordered 3-tuples. This array is of size $m = \binom{n}{3}$. Then for every $l \in \{1, ..., m\}$ we have that Arr(l) is of the form $\{(x_i, y_i), (x_j, x_j), (x_k, y_k)\}$ where $i \neq j \neq k$. For each l we let α to be the largest angle in the triangle defined by the points $(x_i, y_i), (x_j, x_j), (x_k, y_k)$. Then we sort the array in non-increasing order by the value $\sin(\alpha - \frac{\pi}{6})$.

As a result of the optimization, we greatly increase the probability of finding a circle which fits the points during our first initial guesses. Therefore, the algorithm would terminate far more quickly in the case that the data is obtained from a circle.

As a final remark, we investigate the time complexity of the optimization. The sorting part of the algorithm would need $O(n^3 log_2(n))$ computational time and now the overall complexity becomes $O(n^4 + n^3 log_2(n))$. We see that the term added by the optimization is negligible compared to the rest. This means that it would not make the algorithm worse in the case that the data does not come from a circle. Otherwise, it becomes the dominant term since we expect the algorithm to terminate after a small amount of checked 3-tuples.

4.7 Order Of The Data Points

Another important aspect of the algorithm analysis is whether the order of the given points influences the method. In general, the order of the points does not have a significant impact on the algorithm, since in the worst case scenario we still need to check all 3-tuples. However, if we implement the algorithm without the optimization, in some specific situation the ordering may reduce the running-time. This occurs when we only need to make a few checks.

On the other hand, that is not the case when we implement the algorithm with the optimization, since we are checking the 3-tuples in a very specific order.

4.8 Ideal Fit Of A Circle

Curve fitting is a well-known problem. Here we discuss the sub-problem of finding a circle that best matches the given data. Given that this a popular topic, there are a lot of resources

proposing different methods. In this section we present a few of them. One of these methods is a natural extension of our algorithm described in section 4.4.

Let $T : (x-a)^2+(x-b)^2 = r^2$ denotes the best fitting circle. Now we are interested in finding the values of a, b and r. To determine these, we are going to consider two methods different in their nature.

4.8.1 Method 1

Our first approach will be to choose a function

$$R(a,b,r) := \sum_{i=1}^{n}(||(x_i,y_i)-(a,b)||_2 - r)^2 = \sum_{i=1}^{n}\left(\sqrt{(x_i-a)^2+(y_i-b)^2}-r\right)^2.$$

This function provides a reasonable measure of how well the circle with center (a, b) and radius r fits the given data. Now to obtain the best possible values of a, b and r we are going to minimize the function R. This turns out to be complicated problem when using exact arithmetics. Thus we may consider finding the minimiser numerically. The problem is known as Total Least Squares and can be solved using Gauss-Newton minimisation algorithm. However, this approach is shown to be inefficient and extremely sensitive to the presence of outliers [Coope].

Before jumping into the second method for finding the best fitting circle, we compute the partial derivatives of the function R:

$$\frac{\partial R}{\partial a} = 2r\sum_{i=1}^{n}\frac{x_i-a}{\sqrt{(x_i-a)^2+(y_i-b)^2}} - 2n\bar{x} + 2na$$

$$\frac{\partial R}{\partial b} = 2r\sum_{i=1}^{n}\frac{y_i-b}{\sqrt{(x_i-a)^2+(y_i-b)^2}} - 2n\bar{y} + 2nb$$

$$\frac{\partial R}{\partial r} = -2\sum_{i=1}^{n}\sqrt{(x_i-a)^2+(y_i-b)^2} + 2nr.$$

Equating these partial derivatives to zero does not produce a closed form solution for a, b and r. That is the reason why we consider a numerical approximation of the minimiser of R. However, one useful takeout is that if we have the values of a and b, we can express r as

$$r = \frac{1}{n}\sum_{i=1}^{n}\sqrt{(x_i - a)^2 + (y_i - b)^2}.$$

4.8.2 Method 2

Our second strategy will be to use the same function R as defined for Method 1. More specifically we want to be able to express the radius r in terms of a, b.

The idea is to compute the equations of all circles that passe through three distinct points that are not on a line. Then we will obtain $\binom{n}{3}$ equations. Now for the radius of T we will average the radii of all these circles. We will also choose the center (a, b) of the best fitting circle T to be the center of mass of the points $(a_1, b_1), ..., (a_m, b_m)$, where $m = \binom{n}{3}$ and $(a_1, b_1), ..., (a_m, b_m)$ are the centers of the circles that passe through three distinct points. Then we obtain a circle T, which is a reasonable choice if we want a circle that fits the given data.

One important remark is that this approach has the same limitations as the algorithm we described in section 4.4, i.e. we need the assumption that there are no dense points in the data.

4.8.3 Linear Least squares

There is another method proposed by Coope. The main idea of this approach is to linearize the Total Least Squares problem. However, we are not going to dive into details for this method, but it is worth mentioning it since it is preferable over the Total Least Squares.

In conclusion, there are various methods for fitting a circle and each of them has its own strengths and limitations.

4.9 A Closing Probabilistic Outlook

In our preceding scenarios, we studied geometric objects with a finite set of vertices. A circle can be viewed as an analogue that describes a polygon with an infinite set of vertices. It is apparent that within the context of our problem, we only highlight the study of finite sets of data brought due to the constraint of computing power and numerical simulation. However, an *a priori* result can potentially provide a path for investigation without this restriction while simultaneously providing outcomes that are significant to our analysis.

Reslying on our previous assumptions, we will advance this argument in stages. First, we rely on primary intuition to obtain a rudimentary probabilistic intuition of our setting and how this may be ineffectual. We follow up using tools from Measure-Theoretic Probability and close with an animalistic theorem that ensures an effective bound for our case.

Proposition 4 (High Probability Of Bounded Error - Circle Edition) *Our points in Euclidean Space have errors lie in the translation-invariant interval* $(-\varepsilon, \varepsilon)$ *with relatively high probability.*

Proof. We argue by tools of Measure-Theoretic Probability and assume Zermelo-Fraenkel-Choice axioms or any set of axioms consistent with it.

We start by defining a product measure space such that $(\Omega, \mathcal{F}, \mu) = ([0,1]_1 \times [0,1]_2 \ldots, \mathcal{B}_{1|[0,1]} \otimes \mathcal{B}_{2|[0,1]} \ldots, \lambda_{1|[0,1]} \otimes \lambda_{2|[0,1]} \ldots)$ that acts as a universal probability space, where $\mathcal{B}_{i|[0,1]}$ denotes the Borel σ-algebra of $\mathbb{R}$ restricted to the interval $[0,1]$ and $\lambda_{i|[0,1]}$ denotes the Lebesgue measure restricted to the same interval. The proof on its existence can be found in [Sae]. Observe that our space is σ-finite and our product measure is unique by Kolmogrov's Extension Theorem ([Cohn], [Bauer, 1996],[Halmos, 1974]). Moreover, let $\pi_i : [0,1]_1 \times [0,1]_2 \times \cdots \mapsto [0,1]_i$ denote the canonical projection maps from the product sample space to the corresponding sample space. Subsequently, we aim to use the Inverse Sampling Method to construct independent random variables that describe a probabilistic perspective of our error bounds. Let F_{X_i} denote the generalised inverse of the i-th random variable and U_i denote the standard unit uniform random variable. Note that the generalised inverse is a function given by

$$F_{X_i}^{-1}(p) := \inf\{x \in \mathbb{R} : F_{X_i}(x) \geq p\} \quad \forall p \in [0,1].$$

We then define a family of Bernoulli random variables $\{X_i\}_{i \in I}$ scaled by an indicator function $\mathbb{1}_{\{\varepsilon_i \notin (-\varepsilon, \varepsilon)\}}$ by

$$X_i := F_{X_i}^{-1} \circ U_i \circ \pi_i.$$

We observe that our construction is supported by the Kolmogrov Extension Theorem as our Borel σ-algebras contain the Euclidean topology on $\mathbb{R}$ which is Hausdorff and our measures are inner-regular as $\mathbb{R}$ is a Polish Space and thus Radon [Rad, 2020]. By virtue of this construction,

each random variable has a Cumulative Distribution Function of the form:

$$F_{X_i}(t) = \mathbb{P}(X_i \leq t) = \begin{cases} 1-p & t = 0, \\ p & t = 1, \end{cases}$$

where $p > 0$ is small and each $i \geq 1$ corresponds to an element in Euclidean Space. Observe that due to our construction, these random variables are well-defined and model the independent and identically distributed nature provided by the random sample assumption of our error bounds. Leveraging the independence of our random variable now allows us to examine the instance that our errors all fall within our predetermined error bound.

Due to the factorization of our probability measures, it holds that:

$$\begin{aligned} \mathbb{P}(X_1 = 0, X_2 = 0, \dots) &= \Pi_{i=1}^{\infty} \mathbb{P}(X_i = 0) \\ &= \Pi_{i=1}^{\infty}(1-p). \end{aligned}$$

We note that we can view this infinite product as a series through a logarithm transform [Sci, 2012]. However, due to the non-zero value of p, our infinite products are reduced to geometric sequences that cannot provide us any useful information in the setting of our defined measure space.

To further investigate possible values we define $\{E_n\}_{n=1}^{\infty}$ to be the sequence of events that our errors lie within our error bound and their tail field [Sheffield] by

$$\tau = \bigcap_{n=1}^{\infty} \sigma.$$

By Kolmogrov's Zero-One Law [Sheffield] it follows that the probability of our event E_n taking place lies within the set $\{0,1\}$.

Therefore, we turn to the aid of the second Borel-Cantelli Lemma ([Cohn], [Bauer, 1996],[Halmos, 1974]). It holds that,

$$\sum_{n=0}^{\infty} \mathbb{P}(E_n) = \sum_{n=0}^{\infty} (1-p)$$

which diverges as it is the infinite sum of a non-zero number independent of the summation index.

Thus, by the lemma we have that

$$\mathbb{P}(\limsup_{n\to\infty} E_n) = 1.$$

By definition of the limit superior of a sequence of sets, we find that

$$\begin{aligned}\limsup_{j\to\infty} E_n &:= \bigcup_{j\geq 1}\bigcap_{n\geq j} E_n \\ &= E_n \text{i.o.},\end{aligned}$$

where i.o. abbreviated the term 'infinitely often'.

Therefore, we find that the probability of any of our elements being within their error bound will almost surely converge to 1.

We now correspond the finite events to our set of coordinates. We have that the second Borel-Cantelli lemma implies the Infinite Monkey Theorem.

We define a subsequence of our sequence of events $\{E_{n_k}\}_{k\geq 1}$ that correspond to the events that all our points meet our theoretical error bound. Then by the Infinite Monkey Theorem ([Isaac, 1995], [Gut, 2005]), the probability that infinitely many of our events E_{n_k} occur is 1.

Collectively, our argument implies that our coordinate errors behave well with relatively high probability and that the probability of the event that our errors lie outside our bound is negligible.

Accordingly, we conclude having demonstrated our proposed argument.

5 Conclusion

In this report, we explored the task of recognizing shapes, specifically, right-angled triangles, regular polygons, and circles. All was done within the framework of computational geometry, numerical analysis, and probability theory. Our aim was to develop algorithms that could reliably identify whether the given data points could have originated from the respective shapes in the presence of data perturbations. For that we established multiple errors models based on the assumptions we made on the data and used them to provide valuable results.

For the right-angled triangle detection we used Pythagoras' theorem and bounded errors in side lengths. This approach proved effective in identifying right-angled triangles with high accuracy under predefined error tolerances.

Regarding the regular polygon verification, three distinct methods were proposed: one using least-squares method and inscribing a polygon, another one employing exhaustive geometric checks and the last one using a convex hull computation. The first one took a different approach to solve the problem and resulted an algorithm giving no false positives, whereas the later one took a standard, but optimized approach making it suitable for large datasets.

As far as the circle recognition is concerned, we proposed one method for testing whether the perturbed points are obtained from some circle. Then we analyzed the time complexity. Moreover, we suggested an optimization that improves the running-time significantly in the case that the original points do lie on a circle. Unfortunately, we also discovered that the algorithm is unstable when the data contains dense points. In addition, we investigated some methods for finding a best fitting circle, which is a well-known problem in image recognition. The study showcased how different approaches to the three main tasks at hand can yield reliable geometric recognition algorithms. These methods are foundational for more advanced image recognition tasks and can have real world applications from computer vision to object detection.

In summary, our work within this course has supplied us with a chance to grow in the skill of applying various forms of mathematics. Despite the pitfalls of our proposed protocols, we believe that our model provides an effective start to the mathematical modelling cycle and are confident in our ability to further question, research and collaborate to improve on what we have developed. All things considered, mathematical models are iterative by design; it is natural for them to evolve.

6 Appendices

Triangle Code

```
"""
Function to test if the given inputs form a right-angled triangle within the bounds.
Args:
    inputs (list): List of tuples, where each tuple contains the side lengths and error value.
                   Example: [[a, b, c], epsilon].
"""
def check_right_triangle(inputs, scope):
    results = []

    for entry in inputs:
        # Get the sides array and epsilon
        sides, epsilon = entry
        # Get the corresponding sides from sides array
        a, b, c = sides

        # Define the function f and the upper bound g
        def f(x, y, z):
            return abs(x**2 + y**2 - z**2)

        def g(a, b, c, epsilon):
            return 2 * epsilon * (a + b + c) +  8 * epsilon**2
        # def g_s3(a, b, c, epsilon):
        #     return 2 * epsilon * (a + b) + 2 * epsilon**2
        # Compute the values of f for all permutations and g
        f_values = [f(a, b, c), f(a, c, b), f(c, b, a)]
        if (scope == 0):
            g_value = g(a, b, c, epsilon)
        # elif (scope == 3):
        #     g_value = g_s3(a, b, c, epsilon)
        else:
            print("NO SUCH SCOPE FOUND");


        # Check if any of the f values satisfy the inequality
        satisfies = any(f_value < g_value for f_value in f_values)

        # Append the result for this entry
        results.append((sides, epsilon, satisfies))

    return results

# Input list
inputs = [
    [[3, 4, 5], 0.01],
    [[5, 4, 3], 0.01],
    [[3.01, 4.01, 4.99], 0.01],
    [[3.01, 4.99, 4.01], 0.01],
    [[3.01, 4.01, 4.99], 0.002],
```

```
    [[3.01, 5.01, 4.01], 0.002],
    [[3.1, 4.105, 4.901], 0.1]
]
scope = 0

# Run the function and print the results
results = check_right_triangle(inputs, scope)

for n, (sides, epsilon, satisfies) in enumerate(results):
    if scope == 0:
        print(f"Sides: {sides}, Epsilon: {epsilon}, Satisfies Right Triangle: {satisfies}")
    # if scope == 3 and n == 5:
    #     print(f"Sides: {sides}, Epsilon: {epsilon}, Satisfies Right Triangle: {satisfies}")


```

Polygon Code

```cpp
#include <bits/stdc++.h>
#define double long double
using namespace std;

struct Point
{
    double x, y;
};

// A global point needed for  sorting points with reference
Point p0;

// A utility function to return square of distance
// between p1 and p2
double distSq(Point p1, Point p2)
{
    return (p1.x - p2.x)*(p1.x - p2.x) +
          (p1.y - p2.y)*(p1.y - p2.y);
}

// To find orientation of ordered triplet (p, q, r).
// The function returns following values
// 0 --> p, q and r are collinear
// 1 --> Clockwise
// 2 --> Counterclockwise
int orientation(Point p, Point q, Point r)
{
    double val = (q.y - p.y) * (r.x - q.x)
              (q.x - p.x) * (r.y - q.y);
    if (val == 0) return 0;  // collinear
    return (val > 0)? 1: 2; // clock or counterclock wise
}

//  Check if two points are equal
int compare(Point& p1, Point& p2)
{

   // Find orientation
   int o = orientation(p0, p1, p2);
   if (o == 0)
     return (distSq(p0, p2) >= distSq(p0, p1))? false : true;

   return (o == 2)? false: true;
}


// Compute the convex set of points
vector<Point> convex_hull_computation(vector<Point> points, vector<Point>&
points_in_order) {
```

```
    int m = 0;
    for (int i = 1; i < points.size(); i++) {
        if (points[i].y < points[m].y) m = i;
    }
    swap(points[0], points[m]);
    p0 = points[0];
    sort(points.rbegin(), points.rend(), compare);
    points_in_order = points;
    vector<Point> convex_hull;
    convex_hull.push_back(points[0]);
    convex_hull.push_back(points[1]);
    int j = 1; // index of the last element in Convex Hull
    for (int i = 2; i < points.size(); i++) {

        while(orientation(convex_hull[j - 1], convex_hull[j], points[i]) == 1) { // If clockwise then remove
            convex_hull.pop_back(); j--;
        }
        convex_hull.push_back(points[i]); j++;
    }

    return convex_hull;

}

double calculate_angle(Point& left, Point& middle, Point& right) {
    // Vectors from points
    double v1x = - middle.x + left.x;
    double v1y = - middle.y + left.y;
    double v2x = right.x - middle.x;
    double v2y = right.y - middle.y;

    // Dot product
    double dot = v1x * v2x + v1y * v2y;

    // Magnitudes of the vectors
    double mag1 = sqrt(v1x * v1x + v1y * v1y);
    double mag2 = sqrt(v2x * v2x + v2y * v2y);

    // Calculate the angle
    if (mag1 == 0 || mag2 == 0) return 0.0; // Avoid division by zero
    double angle = acos(dot / (mag1 * mag2));

    // Convert to degrees if needed
    return angle * (180.0 / 3.1415926535);
}

// Helper function to calculate the magnitude of a vector
double magnitude(double x, double y) {
    return sqrt(x * x + y * y);
}
```

```
 99
100 pair<double, double> calculate_perturbation(double epsilon, double polygon_angle,
    Point& left, Point& middle, Point& right) {
101
102     // Vectors from points
103     double v1x = - middle.x + left.x;
104     double v1y = - middle.y + left.y;
105     double v2x = right.x - middle.x;
106     double v2y = right.y - middle.y;
107
108     // Dot product
109     double dot = v1x * v2x + v1y * v2y;
110
111     // Magnitudes of the vectors
112     double mag1 = sqrt(v1x * v1x + v1y * v1y);
113     double mag2 = sqrt(v2x * v2x + v2y * v2y);
114
115     // Calculate the angle
116     double angle = acos(dot / (mag1 * mag2));
117
118     // Convert to degrees if needed
119     angle =  angle * (180.0 / 3.1415926535);
120
121     // Compute the maximum angle perturbation using the formula
122     double delta_theta_max =
123         2 * asin(epsilon / mag1) +
124         2 * asin(epsilon / mag2);
125     // Calculate the bounds
126     double lower_bound = polygon_angle - delta_theta_max * (180.0 / 3.1415926535);
127     double upper_bound = polygon_angle + delta_theta_max * (180.0 / 3.1415926535);
128
129     return make_pair(lower_bound, upper_bound);
130
131 }
132
133 bool equal_points(Point& p1, Point& p2) {
134     return (p1.x == p2.x && p1.y == p2.y);
135 }
136
137 // Takes a vector of 3 values (you only need 1)
138 bool check_convex_correctness(vector<Point>& points, double epsilon, vector<Point>&
    points_sorted) {
139     double min_dist = numeric_limits<double>::max();
140     int min_index = -1;
141     Point middle = points[1];
142
143     // Function to calculate the distance from the interior point to the segment
    between `points_sorted[i]` and `points_sorted[i+1]`
144     auto distance_to_segment = [&](int i) {
145
146         Point left = points_sorted[i];
```

```
147         Point right = (i == points_sorted.size() - 1) ? points_sorted[0] :
    points_sorted[i + 1];
148
149         double slope_1 = (right.y - left.y) / (right.x - left.x);
150         double constant_1 = left.y - left.x * slope_1;
151
152         double slope_2 = -1 / slope_1;
153         double constant_2 = middle.y - middle.x * slope_2;
154
155         // Intersection point
156         double x = (constant_1 - constant_2) / (slope_2 - slope_1);
157         double y = slope_1 * x + constant_1;
158         Point intersection = {x, y};
159
160         return distSq(intersection, middle);
161     };
162
163     // Perform ternary search on indices
164     int left = 0;
165     int right = points_sorted.size() - 1;
166
167     while (right - left > 2) {
168         int mid1 = left + (right - left) / 3;
169         int mid2 = right - (right - left) / 3;
170
171         double dist1 = distance_to_segment(mid1);
172         double dist2 = distance_to_segment(mid2);
173
174         if (dist1 < dist2) {
175             right = mid2;
176         } else {
177             left = mid1;
178         }
179     }
180
181     // Check the remaining indices (usually 2-3 indices after ternary search)
182     for (int i = left; i <= right; i++) {
183         double dist = distance_to_segment(i);
184         if (dist < min_dist) {
185             min_dist = dist;
186             min_index = i;
187         }
188     }
189
190     // Check if the minimum distance is smaller than 4 * epsilon * epsilon
191     if (min_dist <= 4 * epsilon * epsilon) {
192         // Insert the middle point in between the vertices to which the distance was
    the smallest
193         points_sorted.insert(points_sorted.begin() + min_index + 1, middle);
194         return true;
195     }
```

```
196
197     return false;
198 }
199
200
201 bool check_valid_polygon(vector<double>& side_lengths, vector<pair<double, double>>&
    angle_upper_lower, double epsilon, double polygon_angle) {
202     // Check if the minimum and maximum side lengths are within bounds of 4*epsilon
203     double min_side = *min_element(side_lengths.begin(), side_lengths.end());
204     double max_side = *max_element(side_lengths.begin(), side_lengths.end());
205     if (max_side - min_side > 4 * epsilon) {
206         return false;
207     }
208
209     // Check if each angle is within the perturbation range
210     for (int i = 0; i < angle_upper_lower.size(); i++) {
211         double lower_bound = angle_upper_lower[i].first;
212         double upper_bound = angle_upper_lower[i].second;
213         if (polygon_angle < lower_bound || polygon_angle > upper_bound) {
214             return false;
215         }
216     }
217
218     return true;
219 }
220
221 int main() {
222
223     /*  Input order:
224     n (the number number of vertices (or sides of the polygon) in any order)
225     x_1 y_0 (first point)
226     x_2 y_2 (second point)
227     .
228     .
229     .
230     x_n y_n (nth points)
231     epsilon (the error upper bound)
232     */
233
234     int n; cin >> n;
235     vector<Point> points(n);
236     for (int i = 0; i < n; i++) {
237         cin >> points[i].x >> points[i].y;
238     }
239     double epsilon; cin >> epsilon;
240
241     // Calculating the actual polygon angle
242     double polygon_angle = 180.0 * (n - 2) / n;
243
244     // Calculates points in order
245     vector<vector<Point>> points_removed;
```

```
246     vector<Point> points_in_order;
247     vector<Point> points_sorted = convex_hull_computation(points, points_in_order);
248 
249     cout << "\n<———————————————————— BEGINNING OF THE PROGRAM ————————————————————
    ———> \n\n";
250     // Not proceed if not equal
251     int j = 3;
252     if (points_sorted.size() != points_in_order.size()) {
253         // We will now check if the non-convexity of a polygon could have been
    caused by the errors in the vectors
254         for (int i = 3; i < n; i++) {
255             if (equal_points(points_in_order[i], points_sorted[j])) {
256                 j++;
257                 continue;
258             } else {
259                 vector<Point> temp;
260                 if (i == n-1) {
261                     temp.push_back(points_in_order[i-1]);
262                     temp.push_back(points_in_order[i]);
263                     temp.push_back(points_in_order[0]);
264                 } else {
265                     temp.push_back(points_in_order[i-1]);
266                     temp.push_back(points_in_order[i]);
267                     temp.push_back(points_in_order[i+1]);
268                 }
269                 points_removed.push_back(temp);
270             }
271         }
272 
273 
274         // Iterate through the points removed and check if the distance between the
    side formed from
275         // Points next to it is greater then epsilon, then end the process,
    otherwise, continue.
276         cout << "Points removed due to non-convexity: \n";
277         for (vector<Point> vec : points_removed) {
278             cout << vec[1].x << " " << vec[1].y << '\n';
279             cout << '\n';
280             if (!check_convex_correctness(vec, epsilon, points_sorted)) {
281                 cout << "NOT CONVEX. UNEQUAL VERTICES OF CONVEX HULL" << '\n';
282                 return 0;
283             }
284         }
285         cout << "IMPORTANT: Points were added back as the error could have caused
    the non-convexity" << "\n\n";
286     }
287 
288 
289 
290     vector<double> angles_sorted;
291     vector<pair<double, double>> angle_upper_lower;
```

```
292
293     cout << "Points in order" << '\n';
294     for (int i = 0; i < points_sorted.size(); i++) {
295         cout << "Vertex " << i << ": " << points_sorted[i].x << " " <<
    points_sorted[i].y << '\n';
296     }
297
298     // Calculate angles
299     vector<Point> points_sorted_angle = vector<Point> (points_sorted.begin(),
    points_sorted.end());
300     points_sorted_angle.insert(points_sorted_angle.begin(),
    points_sorted[points_sorted.size() - 1]);
301     points_sorted_angle.push_back(points_sorted[0]);
302     vector<double> angles;
303     for (int i = 1; i <= points_sorted.size(); i++) {
304         Point middle = points_sorted_angle[i];
305         Point left = points_sorted_angle[i - 1];
306         Point right = points_sorted_angle[i + 1];
307         angles.push_back(calculate_angle(left, middle, right));
308         angle_upper_lower.push_back(calculate_perturbation(epsilon, polygon_angle,
    left, middle, right));
309     }
310
311     cout << "\nAngle between consecutive points" << '\n';
312     for (int i = 0; i < angles.size(); i++) {
313         cout << "Angle for vertex: " << i << " " << setprecision(5) << angles[i] <<
    " Perturbation Bounds: [" << angle_upper_lower[i].first << ", " <<
    angle_upper_lower[i].second << "] degrees\n";
314     }
315
316     // Calculate the side lengths
317     vector<double> side_lengths;
318     for (int i = 0; i < points_sorted.size(); i++) {
319         if (i == points_sorted.size() - 1) {
320             side_lengths.push_back(sqrt(distSq(points_sorted[i],
    points_sorted[0])));
321         } else {
322             side_lengths.push_back(sqrt(distSq(points_sorted[i],
    points_sorted[i+1])));
323         }
324     }
325
326     cout << "\nSide lengths of corresponding 2 consecutive vertices" << '\n';
327     for (int i = 0; i < side_lengths.size(); i++) {
328         if (i == side_lengths.size() - 1) {
329             cout << i << " - " << 0 << " side length: " << side_lengths[i] << '\n';
330         } else {
331             cout << i << " - " << i + 1 << " side length: " << side_lengths[i] <<
    '\n';
332         }
333     }
```

```
334
335     cout << '\n';
336
337     // Check if the polygon is valid
338     if (check_valid_polygon(side_lengths, angle_upper_lower, epsilon,
    polygon_angle)) {
339         cout << "It is possible to create a polygon." << endl;
340     } else {
341         cout << "It is not possible to create a polygon." << endl;
342     }
343
344     return 0;
345 }
346
347 /*
348 5
349 0 0
350 1 2
351 1 3
352 0 4
353 -1 2
354
355 9
356 0 0
357 4 1
358 6 4
359 4 10
360 0 10
361 -3 8
362 -5 5
363 -5 2
364 -2 1
365
366 6
367 10 0
368 5 9
369 -5 9
370 -10 0
371 -5 -9
372 5 -9
373
374
375 */
```

Circle Code

```cpp
#include <iostream>
#include <math.h>
struct Point
{
double x, y;
Point(double _x = 0, double _y = 0)
{
x = _x;
y = _y;
}
void input()
{
std::cin >> x >> y;
}
};
double get_first_constant(Point A, Point B, Point C)
{
double value = (C.y - A.y) * (C.x - B.x) - (C.y - B.y) * (C.x - A.x);
return value;
}
double get_second_constant(Point A, Point B, Point C)
{
double value = (C.x - A.x) * (C.x - B.x) + (C.y - A.y) * (C.y - B.y);
return value;
}
double get_third_constant(Point A, Point B, Point C)
{
//std::cout << "line: " << get_first_constant(A, B, C) << " " <<
get_second_constant(A, B, C) << std::endl;
return fabs(get_first_constant(A, B, C)) + fabs(get_second_constant(A, B, C));
}
double calculate_f(Point A, Point B, Point C, Point V)
{
double first_constant = get_first_constant(A, B, C);
double second_constant = get_second_constant(A, B, C);
//std::cout << "take f " << ((V.x - A.x) * (V.x - B.x) + (V.y - A.y) * (V.y - B.y))
*
first_constant << " " << second_constant * ((V.y - A.y) * (V.x - B.x) - (V.y - B.y)
*
(V.x - A.x)) << std::endl;
double value = ((V.x - A.x) * (V.x - B.x) + (V.y - A.y) * (V.y - B.y)) *
first_constant -
second_constant * ((V.y - A.y) * (V.x - B.x) - (V.y - B.y) * (V.x - A.x
));
return value;
}
double calculate_g(Point A, Point B, Point C, Point V, double eps)
{
double constant = get_third_constant(A, B, C);
```

```
48  //std::cout << "value of C: " << constant << std::endl;
49  //std::cout << "here " << constant * (fabs(V.x - A.x) + fabs(V.x - B.x) + fabs(V.y -
50  A.y) + fabs(V.y - B.y)) + 5 * constant * eps * eps << std::endl;
51  double value = constant * eps * (fabs(V.x - A.x) + fabs(V.x - B.x) + fabs(V.y - A.y)
52  + fabs(V.y - B.y)) +
53  5.0 * constant * eps * eps;
54  return value;
55  }
56  const int MAXN = 1e4 + 10;
57  int n;
58  double eps;
59  Point data[MAXN];
60  void input_data()
61  {
62  std::cin >> n >> eps;
63  for (int i = 1; i ⩽ n; i ++)
64  data[i].input();
65  }
66  bool check_tuple(Point A, Point B, Point C)
67  {
68  for (int i = 1; i ⩽ n; i ++)
69  {
70  //std::cout << "points: " << A.x << " " << A.y << " " << B.x << " " << B.y << " "
71  << C.x << " " << C.y << std::endl;
72  //std::cout << "cs " << get_second_constant(A, B, C) << std::endl;
73  double f = calculate_f(A, B, C, data[i]);
74  double g = calculate_g(A, B, C, data[i], eps);
75  //std::cout << "here " << i << " " << fabs(f) << " " << g << std::endl;
76  if (!(fabs(f) < g))
77  return false;
78  }
79  return true;
80  }
81  void choose_tuples()
82  {
83  //std::cout << n << std::endl;
84  //exit(0);
85  for (int i = 1; i ⩽ n; i ++)
86  for (int j = i + 1; j ⩽ n; j ++)
87  for (int k = j + 1; k ⩽ n; k ++)
88  {
89  //std:: cout << i << " : " << j << " : " << k << std::endl;
90  if (check_tuple(data[i], data[j], data[k]))
91  {
92  std::cout << "All points lie on a circle!" << std::endl;
93  return;
94  }
95  }
96  std::cout << "No circle was found" << std::endl;
97  }
98  void model()
```

```
{
input_data();
choose_tuples();
}
int main()
{
model();
return 0;
}

/*
4 0.0001
1 1
1 -1.00001
-1 1
-1 -1.00001
9 0.15
1.90117 3.38488
2.03587 3.58694
4.26316 1.64331
100 100
1500 -2500
16000 4000
16000 -8000
-10000 -8000
-10000 4000
*/
```

Bibliography

URL https://computational-geometry.org/.

2012. URL https://www.sciencedirect.com/book/9780123846549/mathematical-methods-for-physicists.

2020. URL https://encyclopediaofmath.org/wiki/Radon_measure.

Feb 2023. URL https://www.bbc.co.uk/bitesize/guides/zws8d2p/revision/2.

H. Bauer. Probability theory, 1996. URL https://books.google.com/books/about/Probability_Theory.html?id=w76IHsPHybcC.

P. Billingsley. *Convergence of probability measures*. Wiley Series in Probability and Statistics: Probability and Statistics. John Wiley & Sons Inc., New York, second edition, 1999. ISBN 0-471-19745-9. A Wiley-Interscience Publication.

D. L. Cohn. Measure theory. URL https://link.springer.com/book/10.1007/978-1-4614-6956-8.

I. D. Coope. Circle fitting by linear and nonlinear least squares - journal of optimization theory and applications. URL https://link.springer.com/article/10.1007/BF00939613.

A. Gut. Probability: A graduate course, 2005. URL https://link.springer.com/book/10.1007/978-1-4614-4708-5.

P. R. Halmos. Measure theory, 1974. URL https://link.springer.com/book/10.1007/978-1-4684-9440-2.

R. Isaac. The pleasures of probability, 1995. URL https://link.springer.com/book/10.1007/978-1-4612-0819-8.

I. Kasa. A circle fitting procedure and its error analysis. *IEEE Transactions on Instrumentation and Measurement*, IM–25(1):8–14, Mar 1976. doi: 10.1109/tim.1976.6312298.

Mathworks. What is image recognition? URL https://nl.mathworks.com/discovery/image-recognition-matlab.html#:~:text=Image%20recognition%20is%20the%20process,medical%20imaging%2C%20and%20security%20surveillance.

B. O'Neill. Exchangeability, correlation, and bayes' effect. *International Statistical Review*, 77(2):241–250, 2009. doi: https://doi.org/10.1111/j.1751-5823.2008.00059.x. URL https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1751-5823.2008.00059.x.

N. Padmanabhan. Informatics history, 2011. URL https://sde.uoc.ac.in/sites/default/files/sde_videos/InformaticsHistory247.pdf.

J. Portegies. *Lecture Notes Analysis (2MBA40) and (2MBA60)*. Technische Universiteit Eindhoven, 2023.

H. Royden and P. Fitzpatrick, 1963. URL https://www.scirp.org/reference/referencespapers?referenceid=1774969.

S. Sheffield. 18.175: Lecture 10 zero-one laws and maximal inequalities. URL https://math.mit.edu/~sheffield/175/Lecture10.pdf.

M. L. Tang and H. K. T. Ng. *Inverse Sampling*, pages 688–690. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. ISBN 978-3-642-04898-2. doi: 10.1007/978-3-642-04898-2_313. URL https://doi.org/10.1007/978-3-642-04898-2_313.