Visual Odometry

Lawrence Asamoah Adu-Gyamfi, Marc Fuster Rullan, Johanna Adele Järvsoo and Pol Rovira Escarrà

***Abstract*— In this work, we try to determine the position and orientation of a moving object from images of a camera attached to it.**

I. INTRODUCTION & OBJECTIVES

A. DESCRIPTION OF THE PROBLEM

Visual odometry is nothing more than the process of determining real parameters from the analysis of pictures. In order to achieve our goal of knowing the position and orientation of a moving object from images of a camera attached to it, we will distribute some balls (whose position is known) that should help us in determining our position and orientation.

But before trying to face the main objective of this work, we will start with the inverse situation, we will try to develop a code that allows us to find the distance between the center of the picture taken and the position of a ball knowing our position, the ball's position and also the direction of the photo. This will be called the direct problem.

Once we have done this, it will be time to develop the inverse problem, which is what we are asked to do. In this problem, we know the position of the balls and the distance found in the direct problem.

Firstly, we will try to solve both problems using trigonometrical relations between the known and unknown parameters. Moreover, since the inverse problem is more interesting and at the same time more difficult, we will try to make a different approach by trying to implement some machine learning functions. Then, we will compare and check both solutions and discuss the results.

B. APPLICATIONS

Classically, odometry has been the study of the calculation of the position and the distance travelled by a vehicle with wheels using the information of the wheel's rotation. However, with this new technological revolution that we are living, a new branch of odometry has appeared: visual odometry.

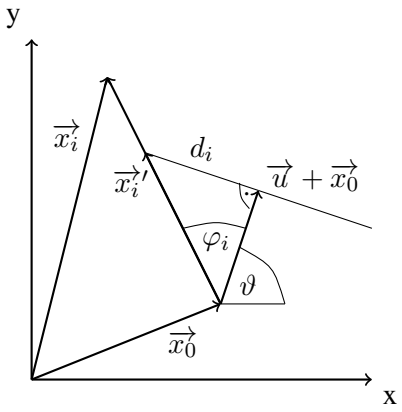
Visual odometry is widely known for its applications in robotics and computer vision. Basically, because the idea of knowing real parameters from images obtained with a camera seems to be made for space and ocean exploration, i.e. it seems to be fundamental for innovative and expensive researches. Visual odometry can also be applied in everyday things such as sports and non-invasive position detection.

It is obvious for non-invasive detection and more important that someone could think for sports. Nowadays we are living in a moment where the technology is more used in order to avoid human non-objective interpretations in sports that may affect the result erroneously. So, the application of visual odometry in football for detecting offside or in dance for having a better knowledge of the movements o the dancers will be fundamental if we want a future with more fair results.

II. PROBLEM IN 2 DIMENSIONS

In order to make a bit easier our problem we will consider a two-dimensional problem. Despite being a simplification of the general case that will be the three-dimensional problem, this makes a lot of sense and can be directly applied. We can imagine, for example, the case of a robot with a camera attached to it that can't go up or down, it's just attached in parallel to the ground, in this case we only need to discover the position of the robot ignoring its height.

So, for the two-dimensional problem we will have something like what is shown in the following picture, where $\overrightarrow{x_0}$ is our position, $\overrightarrow{x_i}$ is the position of one ball, $\overrightarrow{x_i'}$ is the apparent position of one ball in the photo, ϑ our orientation and φ_i is the angle between the direction of the photo $\overrightarrow{u}$ and position of the $i'th$



ball.

III. DIRECT PROBLEM

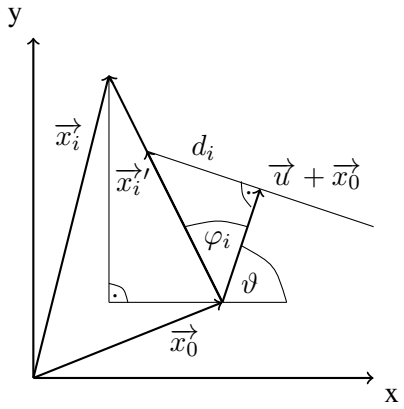
We know:

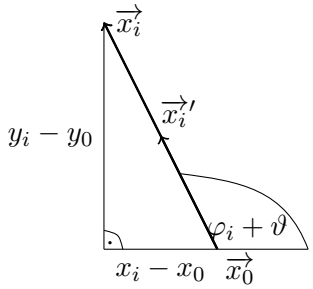
1) Our position $\overrightarrow{x_0} := (x_0, y_0)$
2) The position of balls $\overrightarrow{x_i} := (x_i, y_i)$
3) The direction of the photo $\overrightarrow{u} := (u, v)$

We want to find the distance in the photo between the center of the photo and the projection in the focal plane of the ball i, d_i.

Let f be the length of vector $\overrightarrow{u}$.

1) Computing $\varphi_i + \vartheta$

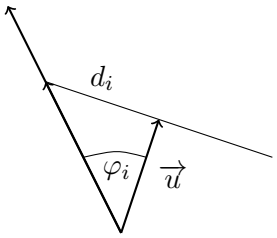




We have a triangle from which we get $\tan(\pi - (\varphi_i + \vartheta)) = \frac{|y_i - y_0|}{|x_i - x_0|}$. Now we can find $\varphi_i + \vartheta = -\arctan(\frac{|y_i - y_0|}{|x_i - x_0|})$.

2) Finding φ_i:
$\varphi_i = -\vartheta - \arctan(\frac{|y_i - y_0|}{|x_i - x_0|})$.

3) Finding d_i.



Finding the tangent of φ_i gives us $\tan(\varphi_i) = \frac{d_i}{|\overrightarrow{u}|} = \frac{d_i}{f}$. From here $d_i = \tan(\varphi_i) \cdot f$.
So we have found the distance of ball i from the center of the photo.

A. IMPLEMENTATION OF THE DIRECT PROBLEM

We have created a full implementation of the direct problem in the file "DirectProblem.py". This Python code has as inputs x_0, y_0 and ϑ (must be introduced in lines 29, 30 and 34 respectively) and outputs an image of what our view would be and an image of our position and where we are looking. In the figure 1 we can see the easiest example for $x_0 = 0$, $y_0 = 0$ and $\vartheta = 0$.

The code works perfectly for all possible configurations of x_0, y_0 and ϑ. In the figure 2 we can see en example of a more complex example for $x_0 = 0.6$, $y_0 = -0.4$ and $\vartheta = 125$.

The system selected for this problem is based on 8 color balls equally distributed in a circle of radius 1. The colors go from 0 to 360 as: red, blue, green, cyan, magenta, yellow, black and pink. These colors have an associated RGB color which can be found in the following snippet 1:

Listing 1. Distribution of balls

```
x=1 # radius of the circle

red    =    [x*np.cos(0*np.pi/2),
    x*np.sin(0*np.pi/2) , 0,'r'] # r
blue   =    [x*np.cos(0.5*np.pi/2) ,
    x*np.sin(0.5*np.pi/2) ,0,'b'] # b
green  =    [x*np.cos(1*np.pi/2),
    x*np.sin(1*np.pi/2)   ,0,'g'] # g
cyan   =    [x*np.cos(1.5*np.pi/2)   ,
    x*np.sin(1.5*np.pi/2) ,0,'c'] # c
magenta =     [x*np.cos(2*np.pi/2)   ,
    x*np.sin(2*np.pi/2)   ,0,'m'] # m
yellow  = [x*np.cos(2.5*np.pi/2)   ,
    x*np.sin(2.5*np.pi/2)   ,0,'y'] # y
```

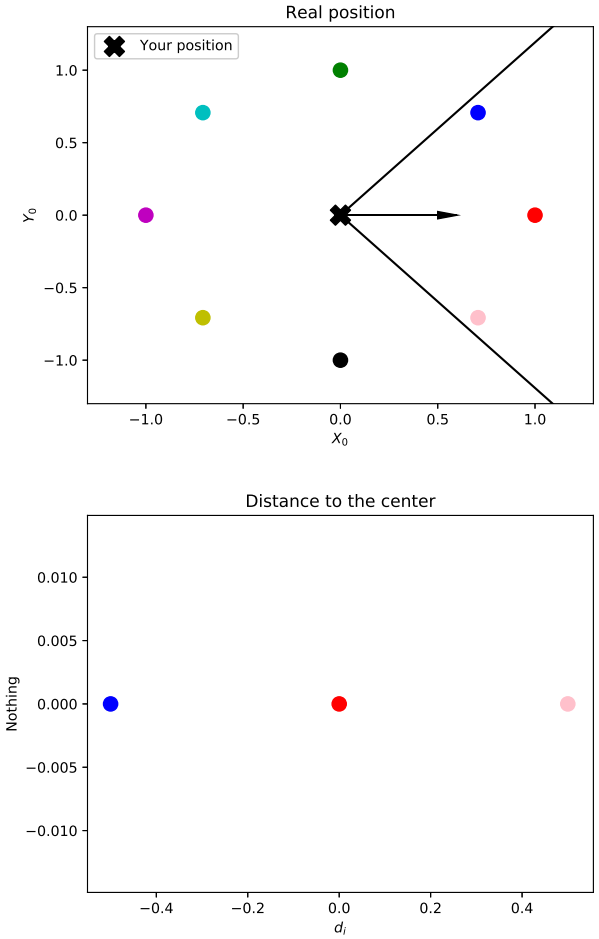


Fig. 1. Direct problem code for $x_0 = 0$, $y_0 = 0$ and $\vartheta = 0$.

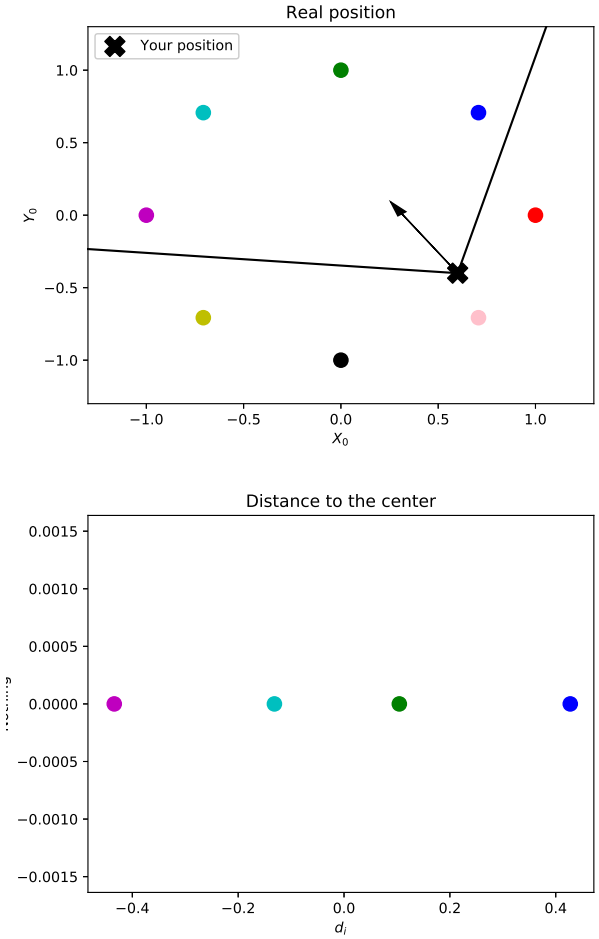


Fig. 2. Direct problem code for $x_0 = 0.6$, $y_0 = -0.4$ and $\vartheta = 125$.

```
black = [x*np.cos(3*np.pi/2) ,
    x*np.sin(3*np.pi/2) ,0,'k'] # k
pink = [x*np.cos(3.5*np.pi/2) ,
    x*np.sin(3.5*np.pi/2) ,0,'pink'] #w
colors=[red,blue,green,cyan,
    magenta,yellow,black,pink]


red_c=[254,0,0]
blue_c=[0,0,254]
green_c=[0,128,1]
cyan_c=[0,192,191]
magenta_c=[191,0,191]
yellow_c=[191,191,0]
black_c=[0,0,0]
pink_c=[255,192,203]
```

The RGB values will be needed later so as to read the images and determine which ball is which. Moreover, the code has plenty of visualizations, functions. However, the core of the code is the implementation of the computation of the distances that are contained in the following Python function 2:

Listing 2. Distance to Center of Image

```
Field_of_view=114
Field_of_view=Field_of_view*np.pi/180
Half_Field_of_view=Field_of_view/2

def distance_to_center_point_i(x0,y0,xi,yi,f,
theta, Half_Field_of_view):

  # angle where the point is phi=np.arctan2(yi-y0,xi-x0)

  angle_max_left= theta - Half_Field_of_view
  angle_max_right= theta + Half_Field_of_view
  #print(angle_max_left,phi,angle_max_right)

  if (angle_max_left < phi < angle_max_right)
  or (angle_max_left < phi-2*np.pi < angle_max_right)
  or (angle_max_left < phi+2*np.pi < angle_max_right) :
     #compute vector to where we point
     distance = -f* np.tan(phi-theta)
  else:
     distance=np.NaN

  return distance
```

As we can see, this function computes the distance to the center only if the ball is inside the field of view. We

have used the human field of view which is 114 degrees [1]. We need a structure in order to save to which ball the distance is computed. Therefore, we will save it as an array $\{d_1, d_2, d_3, d_4, d_5, d_6, d_7, d_8\}$ where if the ball i is not in the field of view, $d_i = NaN$. Remember that the balls are sorted in the following structure {red,blue,green,cyan,magenta,yellow,black,pink}. In our case, for the examples 1 and 2, the distances are the following vectors respectively:

{-0.0, -0.5, nan, nan, nan, nan, nan, 0.5}
{nan, 0.4274, 0.1045, -0.1315, -0.4341, nan, nan, nan}

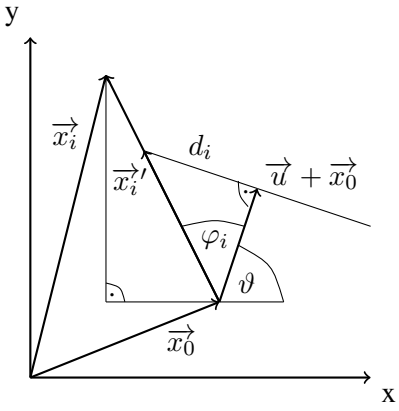
IV. INVERSE PROBLEM

We know:

1) The position of balls $\overrightarrow{x_i}$
2) The distance d_i
3) The length of vector $\overrightarrow{u}$, f

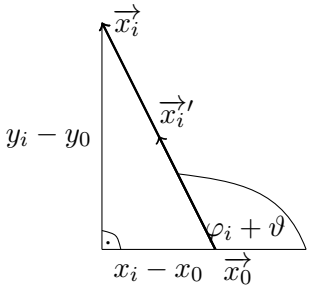
We want to find our position $\overrightarrow{x_0}$ and the angle ϑ.

1) Finding φ_i.



$\tan(\varphi_i) = \frac{d_i}{f}$ and so $\varphi_i = \arctan(\frac{d_i}{f})$.

2) Finding a relation between $\overrightarrow{x_0}$ and ϑ.



We have a triangle from which we get $\tan(\pi - (\varphi_i + \vartheta)) = \frac{y_i - y_0}{x_i - x_0}$. o for each point on the photo we get the equation $-\tan(\varphi_i + \vartheta) = \frac{y_i - y_0}{x}_i - x_0$

We have three unknowns x_0, y_0 and ϑ. So we will need three equations to solve the system i.e. we need at least three balls on the photo. This system of equations is not very easy to solve.

A. IMAGE EXTRACTION

In this section, we will use two different approaches to solve the inverse problem, one solving numerically the system of equations and the other with Machine Learning. The input of both methods are images with balls such as the figure 3:

Fig. 3. Example of real input of the Inverse Problem.

The first thing to do is to read the image pixel by pixel and detect all points that have the RGB colors detailed in the listing 1 within a certain range of 10. Once we know all the positions that contain each color, we fit a circle so as to determine the center of the ball. The method used to fit a circle is Least-Square method and the function was obtained in [2]. After applying this function, we obtain the center of each circle knowing which color it is.

Now we need to transform from pixels to distances. In our case, we have will always work with images of (640, 480) pixels[1]. Since we know the center of the image is in the pixel 320, we make a transformation to go from pixels to distances[2]. At this point, we have extracted all the information from the image and we have the real input structure:

[1]It is a completely arbitrary decision. We chose this one because is how Python save as default the images.

[2]In our system, the pixel to distance transformation is multiplying by 0.00176

$\{d_1, d_2, d_3, d_4, d_5, d_6, d_7, d_8\}$ where if the ball i is not in the field of view, $d_i = NaN$.

B. SYSTEM OF EQUATIONS APPROACH: GEOMETRY

As it has been introduced, in order to solve the inverse problem it is required to solve the following system of equations:

$$\tan(\varphi_1 + \vartheta) + \frac{y_1 - y_0}{x_1 - x_0} = 0, \quad (1)$$

$$\tan(\varphi_2 + \vartheta) + \frac{y_2 - y_0}{x_2 - x_0} = 0, \quad (2)$$

$$\tan(\varphi_3 + \vartheta) + \frac{y_3 - y_0}{x_3 - x_0} = 0, \quad (3)$$

where the unknowns are x_0, y_0 and ϑ and the rest of the variables are known. This system is non-linear and contains trigonometric functions, so it was decided not trying to solve it analytically. Therefore, the first numerical method that it was used was Newton Method [3], which is a method for finding successively better approximations to the roots (or zeroes) of a real-valued function using the iterative equation:

$$x_{n+1} = x_n - J_F(x_n)^{-1} F(x_n) \quad (4)$$

where $x = \{x_0, y_0, \vartheta\}$ is the vector of solutions and $J_F(x)$ is the Jacobian of the system of equations evaluated with the solutions x. The main problem with this method, and of all other methods tried, is that they require an initial seed from where to start the iterations. If the system of equations was smooth, the election of the initial seed lacks importance. However, for this particular system, the initial seed is crucial because the system has plenty of local minima as we can see in the figures 4:

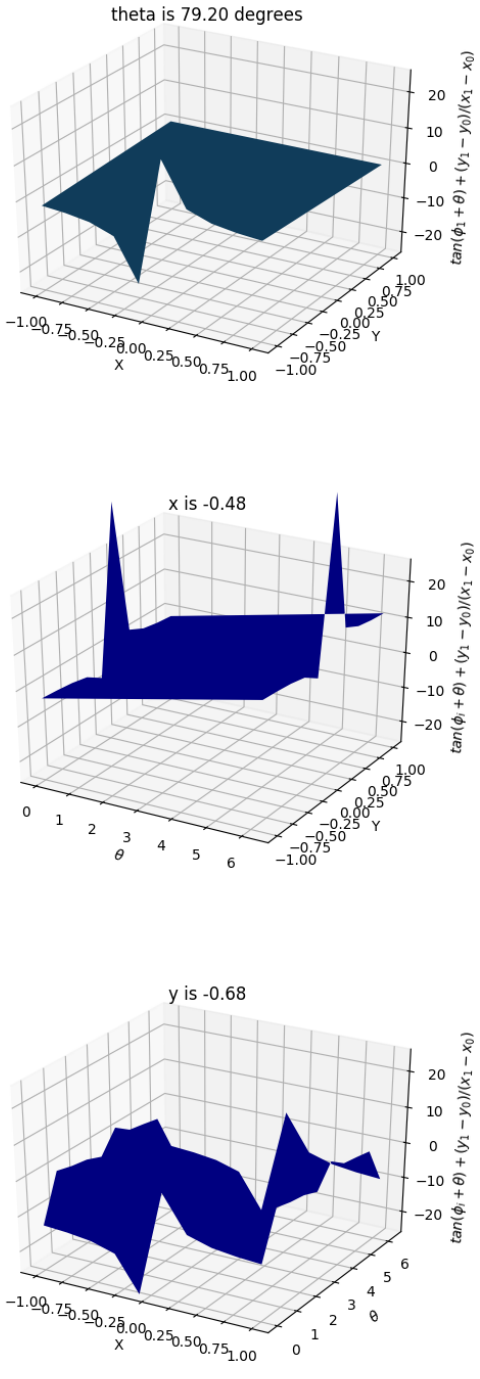


Fig. 4. Example of representation of equation 1 of the system of equations. There are 3 representations, each one is plotting the surface of two of the variables and giving a value for the third one. In this case, the distances that were given as variables were $d_1 = 0.3756$, $d_2 = -0.0272$, $d_3 = -0.4129$

To represent an equation which depends on 3 variables, we require 3 figures, each one is plotting the surface of two of the variables and giving a value for the third one. The numerical values used to represent these figures are not important. **It is difficult to choose the seed as it needs to be suitably close to the root. Because our functions have very steep places it is complicated as even for two values very close to each other the function values may be very far from each other.**

In this project, two different Newton method's have been implemented. The first one is an implementation from scratch which can be found in the code 3. The second implementation is based on using the optimized method of Scipy library, fsolve [4].

Listing 3. Newton Method

```
def iter_newton_short(X, alpha, imax=1e6, tol=1e-3):

    for i in range(int(imax)):
      # calculate jacobian J = df(X)/dY(X)
      J=jacobian(X,x1,y1,phi1,x2,y2,phi2,x3,y3,phi3)
      # calculate function Y = f(X)
      Y=function(X,x1,y1,phi1,x2,y2,phi2,x3,y3,phi3)
      # invert J
      Jinv=np.linalg.inv(J)
      dX=alpha*Jinv.dot(Y)
      #print(dX)
      # step X by dX
      X -= dX
      print(X)
      if np.linalg.norm(dX)<tol:
        #break if converged
        print('converged.')
        break
    return X
```

In order to analyze the convergence of both methods, we have used a known example. We know that the solution of the system of equations is $x_{solution} = \{0.2545, 0.3592, 2.5735\}$ and we use different seeds around the actual solution. Unfortunately, we observe that if the seed is moved slightly (in average 0.001 for x_0 and y_0, and 0.0005 for θ) both methods do not converge to the actual solution $x_{solution}$. In fact, the optimized method fsolve converges to other local minima, and the method is done by scratch sometimes diverges and others converge to other local minima[3].

In fact, this is a very bad result. Let's try to estimate the probability to choose a seed within the convergence ratio. We can choose the right x_0 if we choose within 0.001 of the real solution where the all possible values are from -1 to 1. So we have a probability of $5 \cdot 10^{-4}$ to choose the right x_0. For y_0 is exactly the same. For θ we have possible values from 0 to 2π and we need a precision of 0.0005, so we have more or less a probability of finding the correct θ of 10^{-4}. If we only needed to choose correctly one of the three variables it would not be a problem because we can compute 10000 Newton's method relatively fast. However, the three variables must be chosen within the convergence radius at the same time, so the probability is the multiplication of the three probabilities. Therefore, we have a probability of $2.5 \cdot 10^{-11}$ to choose the correct seed. In fact, this is an extremely low value. We would need, on average, $4 \cdot 10^{10}$ attempts to choose randomly a seed within the convergence radius. Therefore, we need a more clever way to choose the seed.

[3]We have tried this method for different stepsizes (alpha)

We tried using the Continuation Method [5] to get a good seed. This method is based on replacing your system of equations $f = 0$ for $\lambda f + (1-\lambda)g = 0$, where g is a system of equations which we know the solution, and λ is a parameter that we will be ranging from 0 to 1. This method is based on giving as an initial seed the known solution of $g = 0$ and solve the sum system $\lambda f + (1-\lambda)g = 0$ for $\lambda = 0$ using Newton Method. The solution of Newton's method for $\lambda = 0$ is used as a seed for the sum system for $\lambda = 0.01$. Iteratively, until reaching $\lambda = 1$ that the solution we obtain is the solution of $\lambda = 0$.

In this project, we used as a known system the identity $g = \mathbb{1} = 0$. Which corresponds to the set of equations $x_0 - 1 = 0$, $y_0 - 1 = 0$ and $\theta - 1 = 0$. However, all approaches failed to get a correct solution. The continuation method converged to different local minima depending on the number of steps of λ.

C. SYSTEM OF EQUATIONS APPROACH: CAMERA CALIBRATION

A second approach to solve it analytically that has been tried to use is based on the Camera Calibration [6]. The problem of the equations that we had is that they are highly non-linear:

$$\tan(\varphi_i + \vartheta) + \frac{y_i - y_0}{x_i - x_0} = 0$$

Both the fraction on the right and specially the tangent because $\varphi_i = arctan\,(d_i/f)$. Therefore, we have a term which is:

$$\tan(\arctan\,(d_i/f) + \vartheta)$$

which is highly non-linear. Therefore, we need easier equations to be able to solve them easier. **The Camera Calibration approach aims to get a different system of equations to be solved easier.** This approach is based on the following schema:
Since our problem is in 2D, we will set the $Y = 0$. We also need to adapt their notation to ours. The change of notation is the following: $X \equiv x_i$, $Z \equiv y_i, x \equiv d_i$, $p_x \equiv x_0$ and $p_z \equiv y_0$. The main idea behind this approach is to represent the two transformations done (translation and rotation) in a matricial form. Let's briefly detail this method:

The second picture of the model shows that the projection of the ball is inside the line that joins the camera and the ball. In fact, it is the intersection of

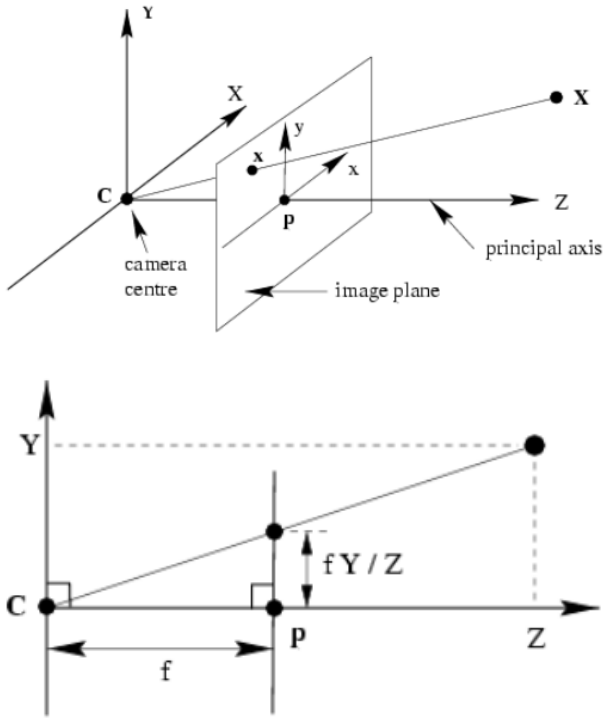


Fig. 5. Schema of Camera Calibration approach.

this line with the focal plane. Therefore, we know that the distance d_i is $d_i = f \cdot x_i/y_i$. The transformation that we are aiming to obtain so as to get the projection is the following:

$$(x_i, y_i) \rightarrow (f \cdot x_i/y_i)$$

So the matrix form that we are looking for is the following:

$$\begin{bmatrix} f \cdot x_i \\ y_i \end{bmatrix} = \begin{bmatrix} f & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix}$$

The first transformation that we apply is the translation, we translate the system of reference (x_0, y_0), therefore, we modify our transformation to:

$$\begin{bmatrix} f \cdot x_i + x_0 \\ y_i + y_0 \end{bmatrix} = \begin{bmatrix} f & 0 & x_0 \\ 0 & 1 & y_0 \end{bmatrix} \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}$$

Now we should apply the Rotation of angle θ which corresponds to the matrix:

$$\begin{bmatrix} cos(\vartheta) & -sin(\vartheta) \\ sin(\vartheta) & cos(\vartheta) \end{bmatrix}$$

So if before we replace the left part of the matrix by this rotation matrix obtaining:

$$\begin{bmatrix} cos(\vartheta) & -sin(\vartheta) & x_0 \\ sin(\vartheta) & cos(\vartheta) & y_0 \end{bmatrix} \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}$$

$$= \begin{bmatrix} x_i cos(\vartheta) - y_i sin(\vartheta) + x_0 \\ x_i sin(\vartheta) + y_i cos(\vartheta) + y_0 \end{bmatrix}$$

Now we calibrate the camera by introducing the parameter λ setting this transformation equal to:

$$\begin{bmatrix} \lambda \cdot d_i \\ \lambda \end{bmatrix} = \begin{bmatrix} x_i cos(\vartheta) - y_i sin(\vartheta) + x_0 \\ x_i sin(\vartheta) + y_i cos(\vartheta) + y_0 \end{bmatrix}$$

Setting $\lambda = \lambda$, we obtain that the final equation to be solved is the following one:

$$x_i cos(\vartheta) - y_i sin(\vartheta) + x_0 - d_i x_i sin(\vartheta) - d_i y_i cos(\vartheta) - d_i y_0 = 0$$

D. Solving the Equations

So from the previous section we got the system of equations

$$\begin{aligned}
(x_1 - d_1 y_1)\cos(\vartheta) - (y_1 + d_1 x_1)\sin(\vartheta) + x_0 - d_1 y_0 = 0 \\
(x_2 - d_2 y_2)\cos(\vartheta) - (y_2 + d_2 x_2)\sin(\vartheta) + x_0 - d_2 y_0 = 0 \\
(x_3 - d_3 y_3)\cos(\vartheta) - (y_3 + d_3 x_3)\sin(\vartheta) + x_0 - d_3 y_0 = 0
\end{aligned}$$

From here we can separate the variable x_0 in the third equation

$$x_0 = -(x_3 - d_3 y_3)\cos(\vartheta) + (y_3 + d_3 x_3)\sin(\vartheta) + d_3 y_0$$

and substituting it into the other two equations we get

$$\begin{aligned}
&(x_1 - d_1 y_1 - x_3 + d_3 y_3)\cos(\vartheta) - \\
&(y_1 + d_1 x_1 - y_3 - d_3 x_3)\sin(\vartheta) - (d_1 - d_3) y_0 = 0 \\
&(x_2 - d_2 y_2 - x_3 + d_3 y_3)\cos(\vartheta) - \\
&(y_2 + d_2 x_2 - y_3 - d_3 x_3)\sin(\vartheta) - (d_2 - d_3) y_0 = 0
\end{aligned}$$

from which by separating y_0 from the second equation, getting

$$y_0 = \Big(\big(x_2 - d_2 y_2 - (x_3 - d_3 y_3)\big)\cos(\vartheta) - \big(y_2 + d_2 x_2 - (y_3 + d_3 x_3)\big)\sin(\vartheta) \Big) \frac{1}{(d_2 - d_3)}$$

and substituting it into the first one, we get

$$\Big(x_1 - d_1 y_1 - (x_3 - d_3 y_3) + \big(x_2 - d_2 y_2 - (x_3 - d_3 y_3)\big)\frac{d_3 - d_1}{d_2 - d_3}\Big)\cos(\vartheta) - \Big(y_1 + d_1 x_1 - (y_3 + d_3 x_3) + \big(y_2 + d_2 x_2 - (y_3 + d_3 x_3)\big)\frac{d_3 - d_1}{d_2 - d_3}\Big)\sin(\vartheta) = 0$$

Now denoting

$$a := \Big(x_1 - d_1 y_1 - \big(x_3 - d_3 y_3\big) + (x_2 - d_2 y_2 - (x_3 - d_3 y_3))\frac{d_3 - d_1}{d_2 - d_3}\Big)$$

$$b := \Big(y_1 + d_1 x_1 - \big(y_3 + d_3 x_3\big) + (y_2 + d_2 x_2 - (y_3 + d_3 x_3))\frac{d_3 - d_1}{d_2 - d_3}\Big)$$

the equation becomes

$$a\cos(\vartheta) - b\sin(\vartheta) = 0,$$

where a and b are known constants.

Taking $\sin(\vartheta)$ to the other side, and dividing bu $\cos(\vartheta)$ we can express ϑ by

$$\vartheta = \arctan(\frac{a}{b}).$$

Values for x_0 and y_0 can be found from previous expressions using the known value for ϑ. Now we have solved these equations. However, we have had no time to implement the solutions to the complete algorithm that reads the images and extract their information. This implementation was done in the machine learning approach as we will see in the following section.

E. MACHINE LEARNING APPROACH

This section describes the implementation of machine learning to solve the inverse problem i.e. estimate the position of the camera with some information about the image produced by the camera.

In reference to the parameter position, we predict[4] the x,y coordinates of the possible position of the camera as well as the posture of the camera (angle of the line of the vision).

For the purpose of this implementation, we have restricted ourselves to regression algorithms[5] of supervised machine learning (based on previous data), as the parameters we intend to predict are all numerical values. The outline of the procedure followed for implementing the machine learning model for predictions is shown in figure 6.

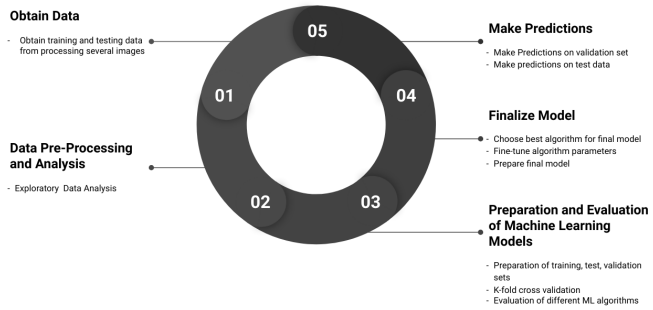


Fig. 6. Outline of procedure for machine learning model

Due to the specificity of the problem, we have trialed several machine learning methods in order to identify which will attain the best prediction accuracy. The final prediction model is then prepared based on the results of the tested algorithms.

The models are trained with data generated from the processing of the images randomly generated with the camera in different positions. Separate models are then prepared to predict each of the three (3) different components of the position parameter.

The accuracy of the predictions of the models are then evaluated.

1) Dataset Description: The data used for the training of the models are obtained from processing the random images generated by alternating the position and view angle of the camera through several different random coordinates.

[4]For the purpose of this document, estimate and predict will be used interchangeably to mean the same thing

[5]Algorithms for predicting continuous variables

The image pre-processing step produces two datasets which are used as the input and labels dataset for training the models. These are described in detail below.

i. Input Dataset
The input dataset is a table with each record representing the results of processing a single image. The variable is all the possible colors of balls that can be present in an image as in 7.

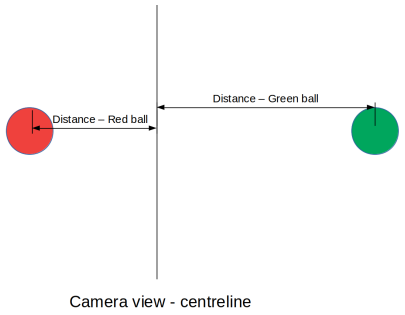


Fig. 7. Image showing parameters captured for input dataset

At the end of the image processing, a list of all balls identified in the image together with their calculated distances to the camera are represented in each record. Balls which do not appear in the image are represented as Nans in the record.

In figure 8 is a typical output of the input dataset.

	red	blue	green	cyan	magenta	yellow	black	pink
0	NaN	NaN	0.374014	-0.416549	NaN	NaN	NaN	NaN
1	NaN	NaN	0.374014	-0.416549	NaN	NaN	NaN	NaN
2	0.363488	-0.022626	-0.412821	NaN	NaN	NaN	NaN	NaN
3	-0.054330	-0.416866	NaN	NaN	NaN	NaN	NaN	0.371825
4	-0.400580	NaN	NaN	NaN	NaN	NaN	NaN	0.375536

Fig. 8. Typical view of the input dataset

ii. Output Dataset (Labels)
The x, y and theta values for each record in the input dataset are recorded and saved as another output of the image processing stage.

A typical view of this table is shown in figure 9.

2) Evaluating Machine Learning Algorithms: As we are not aware in advance which machine learning

	x0	y0	Theta
0	-0.398371	-0.054269	1.542349
1	0.482887	-0.019398	2.232313
2	-0.474068	-0.292890	0.714516
3	-0.100153	-0.416075	0.301375
4	0.281090	0.434339	5.892689

Fig. 9. Typical view of table of x, y, ϑ values

algorithm would work well on our dataset and our problem, we have tested a series of different algorithms in order to make the final choice based on the resulting accuracies of predictions.

In addition, this has been done as an academic exercise to evaluate the strength of different algorithms on our problem. The algorithms chosen are a mixture of linear and non-linear machine learning algorithms.

The steps followed for the preparation of the models and the subsequent evaluation are described in detail in the following sections.

i. Preparation of Training and Test Sets
The datasets for the training of the models and for the final testing are generated separately from the image processing step. This is to avoid the potential effects of over-fitting on our model.

ii. Separate out a validation dataset
In order to limit the effect of over-fitting and possible data leakage during training of the different models, we set aside a validation dataset that is used to evaluate the performance of the models on unseen data. In order to ensure we train each model on the same set of a data, we have specified a random set during the splitting of the data in order to be able to repeat the experiment for each model using more or less the same circumstance.

iii. Setup a test harness to use 10-fold cross -validation
We have taken into account the potential impact of the difference in our datasets on our estimation of the performance of the models. As a result, we

have further pre-processed the datasets to limit the possibility of high variance in our final trained models.

We make use of a cross-validation method known as K-Fold cross-validation where we split our dataset into k parts or folds. We hold back one of the folds and proceed to train our models on k-1 folds. This is repeated k times so that each fold can be used as a test set at some point of the training. The final result is a mean of the performance of the different k iterations.

We have used a value of k=10 as the number of folds for training and cross-validation of our models.

iv. Build 6 different models to predict positions and angle

As part of the spot-check to identify which algorithm best fits the solution of our problem, we have trialed six(6) machine learning algorithms for regression. A combination of linear and non-linear methods have used as part of the list of algorithms. We have utilized already-built classes or packages from sci-kit learn to implement these algorithms in the setup of our models. Below is a list of all the learning algorithms that have trialed in order to pick the best and use for the final model:

Linear Machine Learning Algorithms

- Linear Regression
- Least Absolute Shrinkage and Selection Operator (LASSO) Linear Regression
- Elastic Net Regression

Non-Linear Machine Learning Algorithms

- K-Nearest Neighbours (KNN)
- Classification and Regression Trees
- Support Vector Machines

v. Evaluate Performance of Algorithms and Choose Best

As described earlier, the test harness designed for evaluating the different algorithms makes use of 10-fold cross-validation to estimate the performance of each method. The dataset is split into 10 partitions; training of the model is performed on 9 folds while the remaining is kept for validating the performance of the trained model. The performance of the models are then evaluated based on the resulting mean squared error from the different combination of tests.

In figure 10 is the results of the cross-validation test performed with the chosen algorithms.

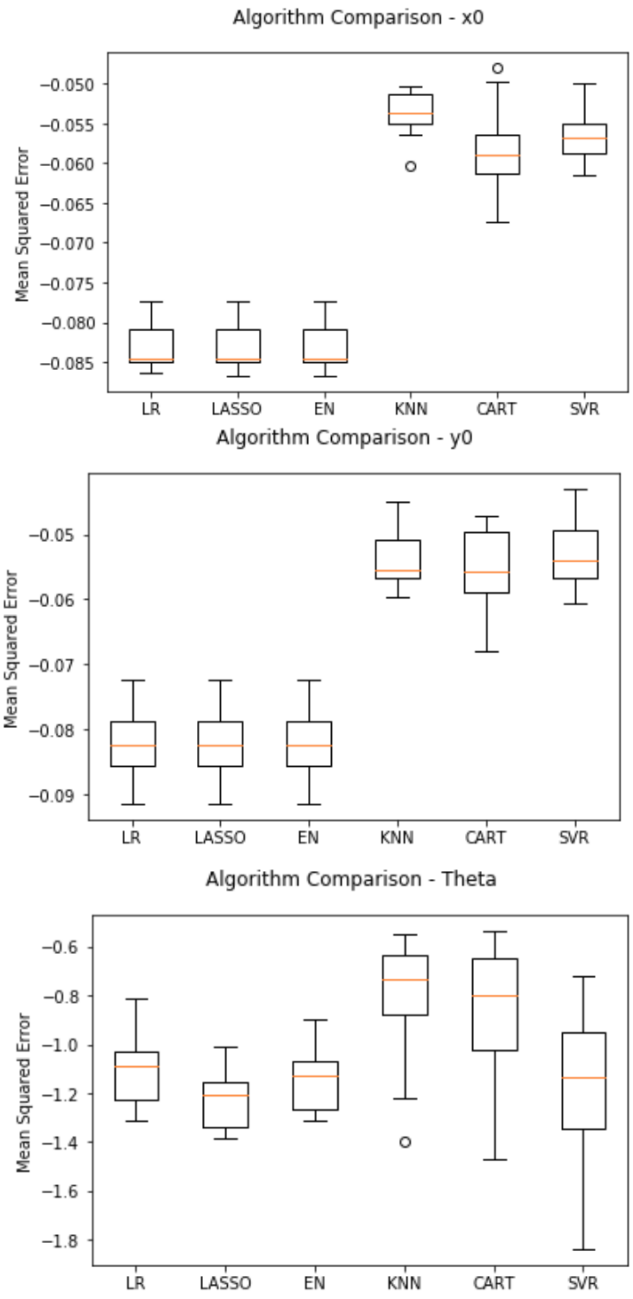


Fig. 10. Cross-validation results for chosen algorithms

After the evaluation of the different models, the K-Nearest Neighbor (KNN) algorithm was chosen to build the final model for the predictions.

We present here a brief overview of how the KNN algorithm works.

3) K-Nearest Neighbours (KNN) Algorithm Overview: The KNN [7] algorithm is a supervised machine learning algorithm employed typically in classification and regression problems. As it is utilized for a regression problem in our case, our description of the method will focus on this function of the algorithm.

The algorithm takes as input the closest training [8] examples in the variable (feature) space. For each test example, it outputs the property value (continuous) which is the average of the values of its 'k' nearest neighbors.

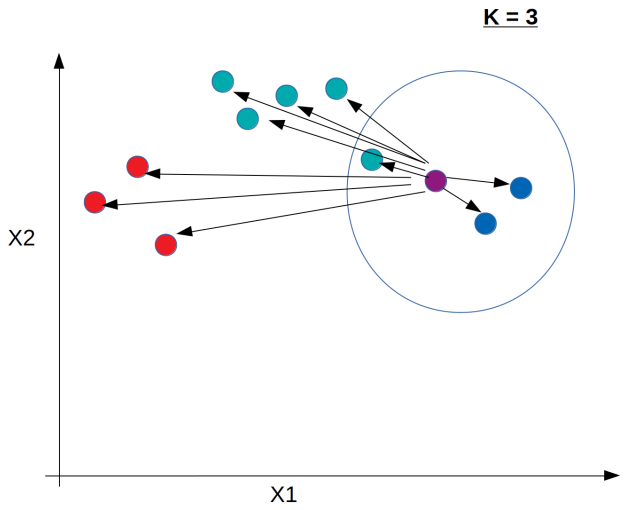


Fig. 11. Example of an implementation of KNN Algorithm for using the three (3) nearest neighbors.

4) Further Tuning of KNN Algorithm: The KNN algorithm is sensitive to the local structure of the data, and that the choice of K for the number of neighbors parameter has a significant impact on the results. As a result, we decided to make a test to identify the best value of K to use in setting up the final model.

We make use of grid search parameter tuning, which builds and evaluates the model for each combination of the parameter pre-defined in a grid. In our case, the parameter 'n_neighbors' is specified in the grid and the model is set-up and evaluated for different numbers of neighbors.

In the end, we get an output of the performance of the model for each of the values specified, from which we proceed to pick the best to use for the preparation of the final model.

Below are the results from the grid search parameter tuning stage. For every query or test example, the Euclidean or Manhattan distance from the labeled (training) examples is computed. Based on the increasing distance, the labeled examples are ordered and the mean value of distances of the K nearest points to the example is calculated.

In figure 12 are the results from testing for the best number of neighbors parameter to use in our model.

```
Best: -0.049667 using {'n_neighbors': 21}
-0.085487 (0.009550) with: {'n_neighbors': 1}
-0.058771 (0.004125) with: {'n_neighbors': 3}
-0.053817 (0.002901) with: {'n_neighbors': 5}
-0.053492 (0.003172) with: {'n_neighbors': 7}
-0.052379 (0.003313) with: {'n_neighbors': 9}
-0.051125 (0.003344) with: {'n_neighbors': 11}
-0.050615 (0.003194) with: {'n_neighbors': 13}
-0.050151 (0.003185) with: {'n_neighbors': 15}
-0.050154 (0.003642) with: {'n_neighbors': 17}
-0.049916 (0.004185) with: {'n_neighbors': 19}
-0.049667 (0.004129) with: {'n_neighbors': 21}
```

Fig. 12. Mean square errors for different number of neighbors highlighting the impact of the number of neighbors parameters on the performance of the model.

5) Final Model For Prediction: In this section we prepare the final model, using the identified hyperparameters, to be trained and evaluated with the validation dataset. As mentioned earlier, we prepare and train different models for each of the three (3) variables to be predicted, (x_0, y_0, and ϑ). However, this is done using the same data to ensure consistency.

The model is evaluated again based on the results obtained from the predictions on the validation set. We make further adjustments to the model and its parameters if required.

If not we proceed to make predictions on the test dataset (new data not used as part of the training) and evaluate the results.

6) Make Predictions on Validation set: In figure 13 are results obtained after making predictions on the validation data set using the final trained model.

7) Make Predictions on Test Dataset: In figure 14 are results obtained after making predictions on the test

```
Mean Squared Error when predicting X: 0.05200796263021012
Mean Squared Error when predicting Y: 0.05432525991554999
Mean Squared Error when predicting Theta: 0.7526966119581668
```

Fig. 13. Performance of trained model on validation data set

dataset using the final trained model.

```
Mean Squared Error when predicting X: 0.05032149175667659
Mean Squared Error when predicting Y: 0.048246316662602345
Mean Squared Error when predicting Theta: 0.7170940115917734
```

Fig. 14. Results of performance of trained model on completely new data, generated as test dataset.

The table in figure 15 shows some of the estimated data (x, y and theta) compared with the actual values.

	x0_Actual	x0_Predicted	y0_Actual	y0_Predicted	theta_Actual	theta_Predicted	balls_visible
1	-0.035476	-0.084160	-0.178177	-0.196584	0.892034	0.884403	3
2	0.290907	0.333189	-0.312847	-0.310776	5.431745	5.368066	1
3	-0.089383	-0.103759	-0.096043	-0.075799	5.249794	5.180877	2
4	0.053209	0.048064	0.216779	0.236352	5.288363	5.263949	3
5	0.210150	0.232627	-0.048525	0.000213	3.137546	3.153378	3
6	-0.193757	-0.182538	0.177304	0.171203	5.633370	5.628566	3
7	0.055058	0.059747	0.218630	0.261517	4.164792	4.179675	3
8	-0.052423	-0.016917	-0.275745	-0.269401	2.036927	2.059280	3
9	0.088117	0.122815	-0.075971	-0.073069	0.532184	0.511330	2
10	-0.134376	-0.153877	0.244011	0.291746	1.932301	1.943045	2
11	-0.340791	-0.314553	-0.134899	-0.104606	0.563508	0.562567	3
12	-0.122621	-0.103759	-0.093747	-0.075799	5.149243	5.180877	2
13	0.042703	0.049451	-0.195996	-0.203793	1.628148	1.641602	3
14	-0.026916	0.012640	-0.367985	-0.361545	4.792945	4.797428	1
15	-0.072906	-0.059403	0.035526	-0.001677	2.797737	2.743913	2
16	-0.006558	-0.013031	-0.013921	-0.014907	3.376192	3.451033	2

Fig. 15. Predicted data compared with actual values

V. CONCLUSIONS

To sum up, first of all we have successfully developed the inverse situation of our initial problem. This problem that can look like something stupid will be very important for the inverse problem: first, because it allows us to check if the solutions of our direct problem are right, and second, because it allows us to generate the dataset required for training the computer in the machine learning section.

About the inverse problem we have to say that the first implementation, the geometric approach, fails easily due to the highly non-linearity of the equation obtained, for this reason the algorithms applied for solving it numerically had failed. On the other side, the camera calibration method looks better for two principal reasons: seems to be more lineal and the focal doesn't appear in the final expression. However, we have to say that we have discovered this method the last days and we only had time to understand it and develop it mathematically, unfortunately we haven't had the opportunity to implement this in the full code that should give us the solutions.

Our last approach for the inverse problem has been the implementation of machine learning algorithms. For doing this we have collected some data with an application based on the solution of the direct problem. As we didn't know which machine learning algorithm would be the best, we have tested a list of them, and we have concluded that K-Nearest Neighbor (KNN) was the best option for solving our problem. Applying KNN we had solutions that only differ in ranges of around 2.5% for the position coordinates (x_o and y_o) and 12% for the angle ϑ. These errors are very little compared to what we've had in the trigonometrical approach, for this reason we conclude that KNN algorithm works great and moreover, we mostly attribute these errors to the image processing application rather than the machine learning algorithm itself.

VI. FURTHER WORK

There are two major improvements that should be done in order to obtain better results. The first one is to solve a variant of our equations as it is explained in the Appendix. The first one is to modify the Machine Learning algorithms so as to predict the three quantities (x_0, y_0 and ϑ) one depending on the other. Now the three variables are predicted independently and it gets a good accuracy. However, the results would improve if the prediction was dependant because these three variables are correlated. The second improvement will be the implementation of penalty or barrier methods to do not allow solutions outside the circle of radius 1. Although the solutions outside the circle are correct, the aim of the problem is to predict our position inside the circle, not outside.

REFERENCES

[1] Brian J Howard, Ian P.; Rogers. Binocular vision and stereopsis. https://books.google.es/books?id=I8vqITdETe0Clpg=PA32ots=JhaI-4InNvpg=PA32redir_esc=yv=onepageqf=false.

[2] MeshLogic. Fitting a circle to cluster of 3d points. https://meshlogic.github.io/posts/jupyter/curve-fitting/fitting-a-circle-to-cluster-of-3d-points/.

[3] Wikipedia. Newton method for non-linear system of equations. https://en.wikipedia.org/wiki/Newton%27s_methodNonlinear_systems_of_equations.
[4] Scipy. *Fsolve, root-finding*.
[5] Wikipedia. Numerical continuation. https://en.wikipedia.org/wiki/Numerical_continuation.
[6] Zakaria Laskar Juho Kannala, Santiago Cortes Reina. Lecture 8: Computer vision. https://mycourses.aalto.fi/pluginfile.php/438972/mod_resource/content/1/Lecture08_JK.pdf.
[7] Wikipedia. k-nearest neighbors algorithm.
[8] Analyticsvidhya. k-nearest-neighbor-introduction-regression-python.