Chapter 8

Software Developement

After the requirements of the test setup are met and kinematics are calculated, the prerequisites for programming are given. This includes hardware requirements like the electrical interface, necessary software and driver installations and the calculated kinematics and algorithms for multiple non unique object tracking.

8.1 Program Overview

The software, written in Python, can be divided into the main parts

- image acquisition,
- image processing,
- calculations,
- controlling,
- presentation.

The aim of this software is to hold tracked clearly distinguishable moving objects on an exact position of the image and control a two-axis mount through calculated reverse kinematics. The difficulty of this project is the identification of multiple non unique objects in a high speed video streams. The faster the shutter speed in correlation to the object movement, the smaller the deviation of the object between video frame n and frame n+1 is. Once again, the advantage of a fast software and shutter speed is shown. To track the non unique feature objects, their centre points are calculated through computer vision algorithms and saved into a temporary object database. After the next frame n+1 is read, the centre points of frame n and n+1 are compared with a Gaussian distribution to determine the most suitable positions of the objects in frame n+1. The result is saved in a persistent object database with a consistent numbering of every object. The database also includes the object number, the actual coordinates and an online indicator for each object. If an object leaves the field of view, the online indicator changes to 0 (= offline) and the coordinates will keep the last coordinates of the seen object. New objects, travelling into the image, won't be saved into the object database to keep the focus on the existing and desired objects.

The image acquisition part reads in the actual video data in a parallel program through a separate thread. In this project a special high sensitive astronomy camera is used to minimise the shutter speed and optimise the correlation between its speed and the tracked object movement. The data is read with a SDK (software development kit) written in the programming language C. To use it in Python, a wrapper for the SDK is required. With this library, all essential parameters such as shutter speed respectively the exposure time, image gain and white balance of the camera can be controlled live in the program. The image or video processing part is doing all the required calculation depending on the given pixel values. It starts with a dynamic threshold as the first step in segmentation, to reduce the data information for further processing.

In the calculation part the relative camera and kinematics behaviour is measured through a calibration method, which gives a relative coordinate system. This new rotated and shifted coordinate system is the base for all deviation measurements in the tracking process. The calibration uses a initial rotation of the kinematic axes to save the combined behaviour of camera orientation and kinematics to recalculate a reverse kinematics in order to control the mount.

To control the mount, the calculated relative coordinate system is set to the desired initial object position, which consists of a shift in x,y and a rotation about a calculated angle. The x,y values of the coordinate shift are the x,y coordinates of the desired object read out of the object database at the time of initialisation. If a deviation occurs in the transformed coordinate system, the values of the object are simultaneously the deviation values for controlling the closed loop system.

The presentation of the video stream, input buttons, camera parameters, coordinate information and camera controls are shown in an entire window, generated in PyGame.

8.2 Overview Software Functions

Table 8.1 gives an overview of essential functions of this program, which will be explained in more detail in this chapter. These functions are used in the first tests of the program, without a PyGame-GUI and also later, without any change.

Table 8.1: Function Overview

Function	**Explanation**
Initialize()	*Initialise GPIOs, load camera driver*
getCoordinates()	*calculate actual coordinates*
getInitialStarTable()	*read initial star database*
StarMatching(StarTable)	*apply the magic matching*
getsSlope()	*get slope of relative coordinate system*
RACalibration(StarTable, TrackedStar)	*right ascension calibration*
DECCalibration(StarTable, TrackedStar)	*declination calibration*
PrintLog()	*show calibration data points*
CoordinatesTransformation (StarTable,TrackedStar)	*transform coordinate system*
TrackingMarkers(StarTable, TrackedStar)	*highlight tracked object*
Tracking(StarTableTrans)	*track object*

8.3 Applied Computer Vision

8.3.1 Read Video Stream Introduction

In this thesis, two relevant types of cameras are used. The first test starts with the Raspberry-Pi camera, a out of the box solution for an easy beginning. The limits of this camera where quickly seen under low light conditions and the ability of adjusting parameters live due processing. For this reason, the implementation of an astronomy camera is necessary. This camera has a more light sensitive image sensor and live adjustable camera parameters via a SDK. Thus it is possible to detect darker objects and reduce the exposure time, which is an improvement factor in accurate tracking.

8.3.2 Read Video Stream: Pi Camera

After the Pi-Camera is connected to the Raspberry Pi, it can be accessed via the command line prompt and also in Python. To read images [22] in a openCV understandable format, the existing camera driver must be overwritten by the command in the command prompt:

```
sudo modprobe bcm2835-v412
```

after are loaded drivers loaded and the camera can be initialized

```
cap = cv2.VideoCapture(0)
```

where 0 is the hardware device number of the camera. After the successful initialisation, a video stream can be started through

```
ret ,frame = cap.read()
```

The required image data is saved to the variable *frame* and is ready for further image processing. The number of frames per second in this case is depending on the cycle time and the shutter speed of the PiCamera.

8.3.3 Read Video Stream: ZWO Asi 120MM-S

In a similar way, but with many more requirements (chapter 3), the ZWO Asi camera can now be used in Python through:

```
camera = asi.Camera(0)
camera.start_video_capture()
frame = camera.capture_video_frame()
camera.stop_video_capture()
```

8.3.4 Converting the Video Stream

Again, the image data is saved into the variable *frame* in the same way and can be further processed as in the Pi Camera. The only difference is the data format of the image. The Pi Camera gathers an RGB image, which needs a three-dimensional data array while the ZWO Asi captures a grey level image, which is saved in a 2-dimensional array. At a certain point of the program, the same data format must be respected to fulfil the requirements of the applied algorithms. For this reason, the three-dimensional array must be converted to a two-dimensional grey level image.

```
gray = cv2.cvtColor(frame ,cv2.COLOR_BGR2GRAY)
```

After that conversion, a threshold algorithm is applied to filter the image for bright elements (segmentation), which stand out from the dark background. A simple dynamic threshold value *Threshold*, which is 80 percent of the maximum pixel value of the image is used.

```
Threshold = frame.max()*0.8
ret , thresh = cv2.threshold(gray ,Threshold ,255 ,0)
```

8.3.5 Feature Detection

After 'thresholding' the image, bright objects are represented by bright pixels (value=255), while the background is black-coloured (value=0), which are good conditions to apply an edge or contour detection algorithm. The existing algorithm *findContours* provided by the openCV library calculates all boundary pixels of the labelled objects and saves it into an array:

```
image, cnts, hierarchy =
cv2.findContours(thresh,cv2.RETR_TREE,cv2.CHAIN_APPROX_SIMPLE)
```

For further processing, the given data by the contours algorithm is used to determine the moments of each recognised object:

```
for c in cnts:
 M = cv2.moments(c)
  if(M['m10']!=0):
   cX = int(M['m10']/M['m00'])
   cY = int(M['m01']/M['m00'])
    TempCoordinates = np.append(TempCoordinates,np.array([[cX,cY]])
    ,axis=0)
```

The result is an array of found moments of the visible objects in the image:

$$\mathsf{C} = \begin{pmatrix} x_1 & y_1 \\ x_2 & y_2 \\ \vdots & \vdots \\ x_i & y_i \end{pmatrix} \tag{8.1}$$

8.3.6 Unique Labelling of Moving Objects

In case an object disappears or a new one is added and processed till the coordinate calculation, the position of the moments and their labels are changed in the moment matrix, which causes identification problems of single objects. This circumstance makes an supplementary calculation necessary to make every object clearly identifiable. For this reason the moments of frame n are saved temporary to a variable and compared to the next frame $n+1$ through a matching algorithm. This algorithm calculates all possible distances between the recognised objects, under consideration, that the dimension of each moment matrix matches the other. In case, the dimension doesn't match, additional calculations must be done. The aim of the matching algorithm is to make every object from a certain time, which is given by an user input, clearly identifiable and provide a persistent object database with a unique sorting. That means, that new objects won't be visible in the database, while disappeared objected are marked as offline (Table 8.2).

So the matrix of moments of objects of frame n with its dimensions [i x 2] is given by

Table 8.2: Object database

StarTable			
Object Number	***x-Coordinates***	***y-Coordinates***	***online indicator***
1	x_1	y_1	0 or 1
2	x_2	y_2	0or 1
$\vdots$	$\vdots$	$\vdots$	$\vdots$
n	x_n	y_n	0 or 1

$$\mathsf{C}_n = \begin{pmatrix} x_1 & y_1 \\ x_2 & y_2 \\ \vdots & \vdots \\ x_i & y_i \end{pmatrix}. \tag{8.2}$$

And the matrix of moments of objects of frame $n+1$ with its dimensions [j x 2] by

$$\mathsf{C}_{n+1} = \begin{pmatrix} \hat{x}_1 & \hat{y}_1 \\ \hat{x}_2 & \hat{y}_2 \\ \vdots & \vdots \\ \hat{x}_j & \hat{y}_j \end{pmatrix}. \tag{8.3}$$

Regarding these two matrices of moments, where i and j are the number of objects, the distances between all objects of frame n and $n+1$ are calculated with equation 8.4 and summarised in a distance matrix, given by equation 8.5.

$$r_{ij} = \sqrt{(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2} \tag{8.4}$$

$$\mathsf{R} = \begin{pmatrix} \sqrt{(x_1-\hat{x}_1)^2+(y_1-\hat{y}_1)^2} & \sqrt{(x_1-\hat{x}_2)^2+(y_1-\hat{y}_2)^2} & \dots & \sqrt{(x_1-\hat{x}_i)^2+(y_1-\hat{y}_i)^2} \\ \sqrt{(x_2-\hat{x}_1)^2+(y_2-\hat{y}_1)^2} & \dots & \dots & \sqrt{(x_2-\hat{x}_i)^2+(y_2-\hat{y}_i)^2} \\ \vdots & \ddots & \vdots & \vdots \\ \sqrt{(x_i-\hat{x}_1)^2+(y_i-\hat{y}_1)^2} & \dots & \dots & \sqrt{(x_i-\hat{x}_j)^2+(y_i-\hat{y}_j)^2} \end{pmatrix} \tag{8.5}$$

Underlying a Gaussian distribution (equation 8.6), the single values are weighted by their probability of best suitable distance between frame n and $n+1$. The range of this probability starts at 0 - which is a very unlikely match, till 1, which gives the best match for the new position of an object of frame n in the frame $n+1$.

$$G_{ij} = e^{-\frac{r_{ij}^2}{2\sigma^2}} \tag{8.6}$$

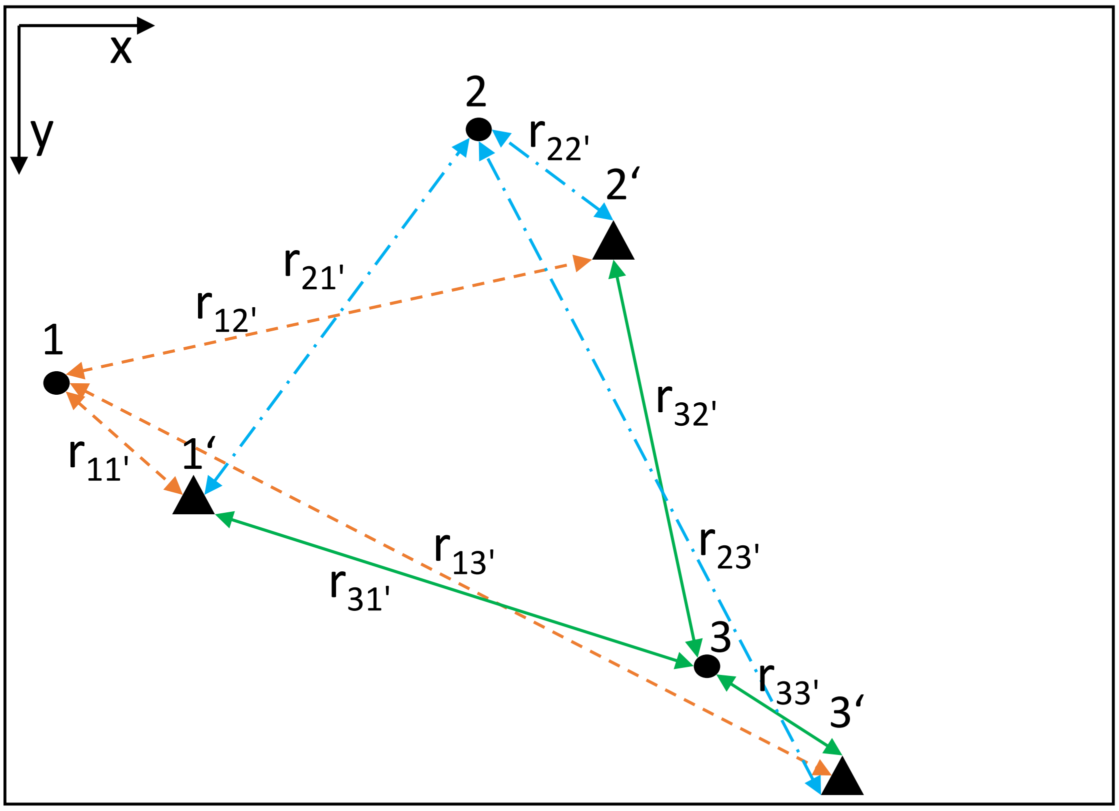

Figure 8.1: Distance determination

$$\mathsf{G} = \begin{pmatrix} e^{-\frac{r_{11}^2}{2\sigma^2}} & \dots & e^{-\frac{r_{1j}^2}{2\sigma^2}} \\ \vdots & \ddots & \vdots \\ e^{-\frac{r_{i1}^2}{2\sigma^2}} & \dots & e^{-\frac{r_{ij}^2}{2\sigma^2}} \end{pmatrix} \tag{8.7}$$

In the case, that

$$dim(\mathsf{C_n}) = dim(\mathsf{C_{n+1}}), \tag{8.8}$$

which means, that the number of objects of frame n and $n+1$ did not change, a diagonal matrix will be the result

$$\hat{\mathsf{G}} = round(\mathsf{G}) = \begin{pmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \ddots & \vdots & \\ 0 & 0 & \dots & 1 \end{pmatrix}. \tag{8.9}$$

As already mentioned, it is possible, that the number of objects decreases or increases in the following image frame, which leads to different dimensions of the compared matrices:

Case a: Number of objects decreasing:

$$dim(\mathsf{C_n}) > dim(\mathsf{C_{n+1}}) \tag{8.10}$$

Case b: Number of objects increasing:

$$dim(\mathsf{C_n}) < dim(\mathsf{C_{n+1}}) \tag{8.11}$$

Considering both cases and the requirement to calculate the distance matrix or proximity matrix, the dimensions of the two input matrices have to be adjusted by the following steps:
Extract the x any y coordinates from $\mathsf{C_n}$ and $\mathsf{C_{n+1}}$ through

$$\begin{pmatrix} \boldsymbol{x} \\ \boldsymbol{y} \end{pmatrix} = \mathsf{C_n}^{\mathrm{T}} \tag{8.12}$$

and

$$\begin{pmatrix} \hat{\boldsymbol{x}} \\ \hat{\boldsymbol{y}} \end{pmatrix} = \mathsf{C_{n+1}}^{\mathrm{T}}, \tag{8.13}$$

further

$$\boldsymbol{x} = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}, \boldsymbol{y} = \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix}, \boldsymbol{I} = \begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix}, \tag{8.14}$$

where the length of the unit vector $\boldsymbol{I}$ corresponds to the length of C_{n+1}. Similar the matrix $\hat{\boldsymbol{x}}$, $\hat{\boldsymbol{y}}$ and $\hat{\boldsymbol{I}}$ are formed, where the length of this unit vector $\hat{\boldsymbol{I}}$ corresponds to $\mathsf{C_n}$.

$$\hat{\boldsymbol{x}} = \begin{pmatrix} \hat{x}_1 \\ \vdots \\ \hat{x}_n \end{pmatrix}, \hat{\boldsymbol{y}} = \begin{pmatrix} \hat{y}_1 \\ \vdots \\ \hat{y}_n \end{pmatrix}, \hat{\boldsymbol{I}} = \begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix} \tag{8.15}$$

with the outer product of

$$\hat{\boldsymbol{I}}\boldsymbol{x}^{\mathrm{T}} = \begin{pmatrix} x_1 & x_1 & \dots & x_1 \\ x_2 & x_2 & \dots & x_2 \\ \vdots & \ddots & \vdots & \vdots \\ x_n & \dots & \dots & x_n \end{pmatrix} = \mathsf{x_n} \tag{8.16}$$

$$\hat{\boldsymbol{I}}\boldsymbol{y}^{\mathrm{T}} = \begin{pmatrix} y_1 & y_1 & \cdots & y_1 \\ y_2 & y_2 & \cdots & y_2 \\ \vdots & \ddots & \vdots & \vdots \\ y_n & \cdots & \cdots & y_n \end{pmatrix} = \mathsf{y_n} \tag{8.17}$$

and

$$\boldsymbol{I}\hat{\boldsymbol{x}}^{\mathrm{T}} = \mathsf{x_{n+1}} \tag{8.18}$$

$$\boldsymbol{I}\hat{\boldsymbol{y}}^{\mathrm{T}} = \mathsf{y_{n+1}} \tag{8.19}$$

the matrices $\mathsf{x_n}$, $\mathsf{y_n}$, $\mathsf{x_{n+1}}$ and $\mathsf{x_{y+1}}$ are calculated. The result is a uniform length (Equation 8.20) of all necessary matrices for further calculations like the proximity matrix.

$$dim(\mathsf{x_n}) = dim(\mathsf{y_n}) = dim(\mathsf{x_{n+1}}) = dim(\mathsf{y_{n+1}}) \tag{8.20}$$

To this, the calculation is done in Python by:

```python

# Extract x,y values of frame n and frame n+1
#xO,yO ... x-old, y-old
#xN, yN ... x-new, ynew
xO,yO = KoordinatenFrameN0.T
xN, yN = KoordiantenFrameN1.T

#Create ones-matrices of the coordinates
#of frame n and n+1
onesArrA = np.ones((KoordinatenFrameN0.shape[0],1))
onesArrB = np.ones((KoordiantenFrameN1.T.shape[0],1))

#calculate the outer products of ones and x0,xN
xO = np.outer(xO,onesArrB)
xN = np.outer(onesArrA,xN)

#calculate the outer products of ones and y0,yN
yO = np.outer(yO,onesArrB)
yN = np.outer(onesArrA,yN)

#calculate the deviations of all
#possible coordinates of frame n
#and n+1
dx = np.power(xO-xN,2)
dy = np.power(yO-yN,2)
R = np.sqrt(dx+dy)
```

```

#Probibility function declaration
sigma = 5
G = np.exp(-(R*R)/(2*sigma*sigma))
```

Correlation Matrix

The probability matrix G shows with its rows and columns the correlation between objects of frame n and frame $n+1$. An example is shown in the following:

$$\hat{\mathsf{G}} = round(G) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \tag{8.21}$$

G shows, that every object of frame n and frame $n+1$ is in a single row. The row indicates the position of the corresponding object n and $n+1$ in the following frame $n+1$. For example, the object 1 of frame n corresponds with object 1 of frame $n+1$ and object 3 of frame n corresponds to object 4 of frame $n+1$. So the aim of the probability matrix evaluation is the classification of objects resulting into a dynamic and easy to use correlation matrix, which assigns ongoing objects into the persistent object database. The evaluated probability matrix result for the given example leads to the correlation matrix

$$\mathsf{CorrMatrix} = \begin{pmatrix} 1 & 1 \\ 2 & 2 \\ 3 & 4 \\ 4 & 3 \end{pmatrix}. \tag{8.22}$$

This matrix is computed in Python via two loops, which reads the position of ones, evaluated for their positions and assigns the values to a new matrix CorrMatrix. With the new information of the correlation matrix, the updated object database can be calculated:

```
i=0
j=0

while(i<G.shape[0]):
 while(j<G.shape[1]):
  if(G[i,j]==1):
   CorrMatrix = np.append(CorrMatrix,np.array([[i,j]]),axis=0)
  j+=1
 j=0
 i+=1
```

```
for C in CorrMatrix:
 c1,c2 = C
 starnumber,x,y,online = StarTable[c1]
 xnew,ynew = CurrentCoordinates[c2]
 StarTable[c1]=starnumber,xnew,ynew,1
return StarTable
```

8.3.7 Camera Calibration

The position of the camera in the guidescope has no fixed orientation and can be rotated in the eyepiece of the scope, which makes a calibration of the position essential. The kinematics must also be considered, so its behaviour also influences the result of the camera calibration, a essential step of relative and further reverse kinematics calculations. To do the calibration process, the given kinematics is moved into both possible directions, while a single object is tracked. The information of the tracking process, more precisely the x and y data of the movement, is logged into two arrays. One array for each axis. These coordinates are saved in RAlog and DECLog. Depending on the focal length, the number of measurement points are set regarding to the distance of the movement of the objects while calibration. (RA = right ascension axis, DEC = declination axis)

$$\mathsf{RALog} = \begin{pmatrix} x_1 & y_1 \\ \vdots & \vdots \\ x_n & y_n \end{pmatrix}, \mathsf{DECLog} = \begin{pmatrix} x_1 & y_1 \\ \vdots & \vdots \\ x_n & y_n \end{pmatrix}. \tag{8.23}$$

The calibration of the right ascension axis is done by:

```
global RACalibIndicator
global RALog
if(len(RALog) <= (CalibrationSteps-1)):
 #move mount in RA
 GPIO.output(20,1)
 #save x and y values to array
 RALog = np.append(RALog,np.array ([[StarTable[TrackedStar,1],StarTable
     [TrackedStar,2]]])
 ,axis=0)
if(len(RALog) == CalibrationSteps):
 #Calibration done, set indicator to 1
 #and disable mount movement in RA
 RACalibIndicator = 1
 GPIO.output(20,0)
 cv2.putText(frame,'RA Calibration on: '+str(len(RALog))+' Steps'
     ,(10,20),1,1,(255,255,255),1,0)
```

And the calibration of the declination axis by:

```
global DECCalibIndicator
global DECLog
if(len(DECLog) <= (CalibrationSteps-1) and RACalibIndicator == 1):
 #move mount in DEC
 GPIO.output(21,1)
 #save x and y values to array
 DECLog = np.append(DECLog,np.array
 ([[StarTable[TrackedStar,1],StarTable[TrackedStar,2]]])
 ,axis=0)
if(len(DECLog) == CalibrationSteps):
 #Calibration done, set indicator to 1
 #and disable mount movement in DEC
 GPIO.output(21,0)
 DECCalibIndicator = 1
 cv2.putText(frame,'DEC Calibration on: '+str(len(DECLog))+' Steps'
     ,(10,32),1,1,(255,255,255),1,0)
```

8.3.8 Fitting

After successfully creating the log files of right ascension and declination axes, a linear function is fitted through the data points of each axis (RALog and DECLog). Used is a least square algorithm provided by the open source numpy library of Python [23]:

```
#calculate slope of fitted function through xRA, yRA,
#degree of function = 1
slopeRA, zRA = np.polyfit(xRA,yRA,1)
#calculate slope of fitted function through xDEC, yDEC,
#degree of function = 1
slopeDEC, zDEC = np.polyfit(xDEC,yDEC,1)
#convert slope from radians to degree
RAangle = np.degrees(np.arctan(slopeRA))
DECangle = np.degrees(np.arctan(slopeDEC))
```

The variables $slopeRA$ and $slopeDEC$ are the slopes of the two fitted linear functions. The argument of the fitting function requires the data points (x,y) and degree of the polynomial - in this case a first order degree. The slope, calculated as the quotient of y and x values, can converted to radians through the evaluation of its tangent value. So the result can be expressed in radians or converted to degrees with the function $np.degrees(radiansValue)$ for easy legibility of the user.

8.3.9 Coordinate Transformation

The calculated slopes of the data points can now be used to determine a relative coordinate system, which includes the camera position in the guidescope and also the behaviour of the used kinematics. That circumstance makes the whole tracking algorithm independent of the kinematics, the observation site and the camera orientation, which guarantees a maximum of flexibility. The slope of the declination axis gives the rotation of the coordinate system, assuming orthogonality. Additionally a single object can be set as origin of the new coordinate system, so the relative coordinate system is rotated and shifted. The advantage of the additional shift is the easy way to evaluate deviations within a target-actual comparison. Every value unequal to zero gives the deviation of the object in pixels in the relative coordinate system.
The transformation of the object database into the relative coordinate system in Python is done with:

```python
StarTableTrans = np.empty((0,4),float)
for Star in StarTable:
 #extract object number, coordinates and online
 #of each object of the database
 number, x, y, online = Star
 #shift
 x = x-xreference
 y = y-yreference
 #rotation by RA-angle
 xtrans = x*np.cos(slopeRA)+y*np.sin(slopeRA)
 ytrans = y*np.cos(slopeRA)-x*np.sin(slopeRA)
 #write transformed database
 StarTableTrans = np.append(StarTableTrans,np.array
 ([[number,xtrans,ytrans,1]]),axis=0)
```

The transformation process includes the object database values and sets up a new transformed database, while maintaining the order of objects. A successful transformation can be recognised by the indication, that the value of the desired tracked object in the transformed object database at time of transformation is equal to zero or very close to zero.

8.4 Tracking

The tracking functions uses the calculated transformed object database, to track objects by minimisation of the deviation of the object to the origin of the relative coordinate system. The minimisation is done by a kinematics movement corresponding to the object deviation in a closed loop function [24]. The Raspberry Pi creates pulses, send them

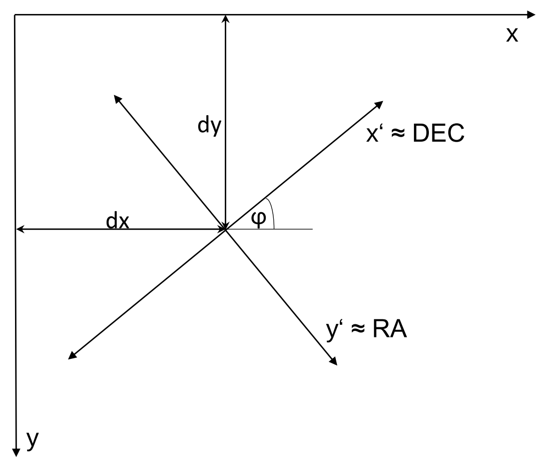

Figure 8.2: Transformation: Rotation by φ und shift by dx and dy

to the electrical interface, which translates them to the ST-4 standard input signal of the telescope mount. The behaviour of the switching time depends, as already mentioned, on the used type of interface, such as relays or transistors. Relays need a minimum on-time cause of their inertia. The speed of the resulting mount speed is set on the EQ-6 hand controller between 0.25 and 1x absolute sidereal speed. A certain inertia of the mount has to be considered, and also atmospheric induced distortions, which affect the apparent object position. The deviation is split into four cases: $dx > 0$, $dx < 0$, $dy > 0$,$dy < 0$. Programmed in Python, considering the minimum on-time of relays:

```
if(dx > 0 ):
GPIO.output(20,1) #RA+
GPIO.output(26,0) #RA-
time.sleep(sleeptime)
if(dx < 0 ):
GPIO.output(26,1) #RA-
GPIO.output(20,0) #RA+
time.sleep(sleeptime)
if(dy < 0):
GPIO.output(19,1) #DEC+
GPIO.output(21,0) #DEC-
time.sleep(sleeptime)
if(dy > 0 ):
GPIO.output(21,1) #DEC-
GPIO.output(19,0) #DEC+
time.sleep(sleeptime)
```

The correction movement takes place as long as there is a deviation, until $dx = 0$, $dy = 0$.

8.5 PyGame

The whole GUI is programmed as a combination of openCV and PyGame and the requirement to convert image and video data between the libraries considering their compatibility. To use the PyGame library, the initialisation has to be done by:

8.6 Initialise PyGame

```
import pygame
from pygame.locals import *
```

A new window with a resolution of 800 by 600 pixels is created by:

```
pygame.display.set_caption("Window Name")
screen = pygame.display.set_mode([800,600])
```

For additional program outputs via textual overlays in the PyGame window, a font and its size must be initialised by:

```
pygame.font.init()
#...SysFont('FontName',TextSize)
myfont2 = pygame.font.SysFont('Arial',15)
```

8.6.1 User Inputs: Buttons

An easy way to implement buttons in PyGame is to load pre created button images and place them to a specific area in the PyGame window.

```
#with path/filename.png
button_name = pygame.image.load("buttons/button_name.png")
#display button at x,y in the window
screen.blit(button_name,(651,5))
```

To connect a called function and the button, the position of the mouse cursor is checked when pressed and evaluated by its coordinates in the window as following:

```

#define function
def CheckMousePos():

#read mouse cursor position
xmouse,ymouse = pygame.mouse.get_pos()
```

```

#check an area, where x is betweet
#x1 and x2
#and y is between
#y1 and y2
if(ymouse > y1Value and ymouse < y2Value and xmouse > x1Value and
    xmouse < x2Value):
 #execute the desired button function
 ButtonFunction()
```

The called function is executed by a transition-function. For clarity in the program code these functions are outsourced and called by:

```
#define the button function

def ButtonFunction():
 doSomething()
 callOtherFunctions()

#Other used functions
#as example:

def Search():
 Search()
 global StarTable
 print("Select")
 StarTable = getInitialStarTable()
 print(StarTable)

#Do starmatching an track objects visually
#without mount movement
def TrackButton():
 print("Track")
 global StarTable
 StarTable = StarMatching(StarTable)
 print(StarTable)

#capture image and save it to a file
def CapImgButton():
 print('CapImg')
 im = Image.fromarray(tframe)
 im.save(strftime("%Y-%m-%d-%H:%M:%S",gmtime())+"img.jpeg")
 print("CapImg")
```

8.7 Threading

Due to image acquisition, depending on the exposure time of the camera, the program is blocked and allows no user inputs and no program outputs. This circumstance causes a difficult operability of the program, because the user inputs are only recognised, when the image acquisition is inactive. The same applies to the output such as image visualisation and calculations. For this reason a parallel operation of the image acquisition and the rest of the program is required. In Python, the available threading library provides that kind of desired parallelism.
In the first step, the library must be imported by:

```
import threading
```

A global variable, which stores the image data is initialised and preallocated:

```
global tframe
#x,y,z, wehre z is the image depth
#z = 3 = RGB
#z= 1 = grey level
tframe = np.zeros((480,640,3))
```

The actual function is packed into a threading class:

```
#tframe ... threaded frame

class getFrame(threading.Thread):
def __init__ (self):
 threading.Thread.__init__(self)
 self.stopThread = False

 def run(self):
  while True:
    global tframe
    global ExpTime
    #set camera controls
    camera.set_control_value(asi.ASI_EXPOSURE, ExpTime)
    camera.set_control_value(asi.ASI_GAIN, CamGain)
    #save image data to tframe
    tframe = camera.capture_video_frame()

    #stop the thread
    if self.stopThread == True:
    break
 def stopThread(self, stopThread):
  self.stopThread = stopThread
```

To start the function of the thread class, it is called by:

```
t1 = getFrame()
t1.start()
```

Now the camera starts a continuous video capture and updates the global variable *tframe* with actual data. This data is read parallel to this by the main program for calculation and presentation.