Motion Video Analysis of Dynamical Systems
Challenges and Solutions

Under the Supervision of Dr. Shibaji Banerjee

Debraj Dutta, Department of Physics, Semester 6, Roll number - 127

St. Xavier's College (Autonomous), Kolkata

May 10, 2021

Declaration

I affirm that I have identified all my sources and that no part of my dissertation paper uses unacknowledged materials.

Acknowledgment

This project would not have been possible without the efforts put in by Mousumi Mitra of my class. I would like to thank her for her active collaboration, hours of collective brainstorming. and her contribution in the parts involving differential equations modeling and digital image processing using openCV. I am grateful to all of those with whom I have had the pleasure to work during this project. Each of the members of my dissertation group has provided me extensive personal and professional guidance and taught me a great deal about both scientific research and life in general. I am especially indebted to Dr. Shibaji Banerjee, our mentor. He has shown me, by his example what a good scientist and person should be. Also, I would like to thank my friend, Rishita Sengupta who has been supportive of my ideas and ultimately fueled me into pursuing the project.

Nobody has been more important to me in the pursuit of this project than the members of my family. I would like to thank my parents, whose love and guidance are with me in whatever I pursue.Moreover, I am grateful to the Microsoft Teams platform provided by St. Xavier's College, that helped me communicate with my supervisor amidst the pandemic situation.

Abstract

The aim of the project is to develop a generalized sequence of steps that can extract the dynamics of a classical system. These steps are employed to obtain the position data from an experimental setup and then to fit the data to a set of modeled differential equations. A spherical pendulum and a magnetic pendulum is a non-linear system used for this purpose. The system is first modeled by a set of differential equations. The motion of the system is then captured using two cameras placed at different relative orientations with respect to the setup. The position data in subsequent video frames for the pair of un-synchronized cameras is reconstructed in 3D space using stereo geometry, triangulation and bundle adjustment and the experimentally captured data set is fit to the predicted set of differential equations using non-linear least square fit. The data-set and the values of the fit parameters are then used to verify the extent to which the set of differential equations can model the physical system.

Contents

1 Introduction

Dynamics is the theory of studying any ***time evolutionary process,*** and the set of *differential equations* governing the process constitutes a dynamical system. The *evolution rule* of a dynamical system is a function that describes how the system evolves once the current state is known. Most dynamical systems in nature are described by ***non-linear differential equations*** which are often unsolvable *analytically.* Hence it becomes difficult to experimentally verify the fact that physical systems in nature are indeed described by these differential equations. A non linear system like a spherical pendulum has its trajectory described in 3D space. The primary challenge is to somehow obtain the position data for the pendulum trajectory as a function of time and then use the data to verify the equations of motions and other parameters governing the system dynamics.
Motion capture can be used to ***digitally*** record the 3D trajectory of system. It is impossible to reconstruct a 3D scene from a single image without making prior assumptions about the scene structure. *Binocular spectroscopy* in a solution used by both biological and artificial systems to localize the position of a point in 3D via correspondences in two views. *Linear triangulation* used in ***stereo reconstruction*** is geometrically described as: the rays connecting each image location to its corresponding camera center intersect at the true 3D location of the point, as the two rays form a triangle with the baseline that connects the camera centers. However the triangulation constraint does not apply when the point moves between the image captures, since it is generally not possible to synchronize the independent camera shutters capturing the scene. The main purpose of this is project is to develop a generalized sequence of steps that captures the motion of a dynamical system using a pair of un-synchronized cameras and then use stereo geometry to simultaneously overcome the synchronization problem and reconstruct the motion of the system in 3D, to extract different motion parameters and verify the extent to which a set of modeled differential equations can describe a dynamical system. Both the ***spherical pendulum*** as well as the ***magnetic pendulum*** (two highly non-linear systems) are used for analyzing the validity of the process. A mathematical treatment is carried out for analyzing the various motion parameters of the spherical pendulum, and the experimentally obtained data is then used to fit the analytic results. The magnetic pendulum on the other hand cannot be handled analytically and a graphical and numerical method is adapted to study the system.

1.1 Challenges

Usually, achieving satisfactory results for 3D reconstruction requires sophisticated hardware. The focus has been shifted to build an elaborate sequence of steps to overcome the hardware restrictions and achieve satisfactory precision without any external software dependencies. The system setup has been built from scratch suiting the requirements for the process. Synchronizing cameras is a major technical challenge and requires extensive hardware support. This has been addressed using *stereo geometry* and *least squares minimization technique* purely as a mathematical problem. Linear triangulation suffers from noisy pixel data and *bundle adjustment* is used as a minimization process to refine the 3D reconstruction data. The magnetic pendulum tends to enter ***chaotic regime*** under certain conditions. As such, ordinary data fitting algorithms like *gradient descent* fail to fit the data to the differential equations. A more generalized approach is developed to fit the experimental data to the system trajectory using a damped non-linear least squares method.

1.2 Related work

1.2.1 Trajectory Reconstruction

Reconstructing a dynamic scene in 3D from a pair of image sequences is fundamentally an ill-posed problem. A large body of work has been done in this field to overcome this inherent ambiguity.
When point correspondences for the 2 camera views in a static scene is provided, the method proposed by *Longuet-Higgins* (1981)[1] estimates the relative camera poses and triangulates the point in 3D using *epipolar geometry.* The geometry involved in reconstructing 3D scenes has been systematically developed in subsequent research by *Hartley and Zisserman* (2004)[2]. While a static 3D point can be reconstructed

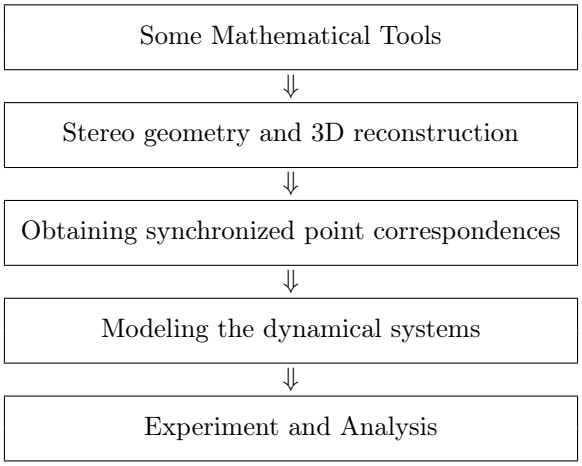
by linear triangulation, when a point moves between the frame captures, the linear triangulation method becomes inapplicable, since the line segments formed by the baseline and the rays from each camera center to the point no longer form a closed triangle.
Trajectory triangulation by *Avidan and Shashua* (2000)[3] demonstrated two cases where a moving point can be reconstructed: (1) if the point moves along a line (2) if the point moves along a conic section. Although this method provided a framework to reconstruct a 3D dynamic scene, the trajectory is restricted by certain geometrical constraints.

1.2.2 Parameter estimation for Differential Equations

Most ODE systems are not solvable analytically, so that conventional data-fitting methodology is not directly applicable. Exceptions are linear systems with constant coefficients, where the approach of the Laplace transform and transform functions plays a role, are solvable, and a statistical treatment of these is available in *Bates and Watts* (1988)[4] and *Seber and Wild* (1989)[5].

1.3 Sequence summary

A detailed description of the required mathematical and computational tools for the process have been provided in the subsequent sections [(2),(6)] in order of their usage, succeeding which the dynamical systems are being modeled to obtain their equations motion and other motion parameters (7). Finally an experiment is deviced to put everything together and test out the process. The python scripts required for different parts of the experiment are provided under the ***Appendix A***(A). The video and data files generated and used by the python scripts, are provided as links in ***Appendix B***(B). A summary of the order in which the paper flows is as shown below.



2 Mathematical Tools

2.1 Non-linear Least Squares Fitting

Non-linear least squares fitting[4, 5, 6] is a mathematical tool used to fit a set of m observations with a model that is non-linear in n number of unknown parameters. Let us consider that we have a set of data points with an experimentally measured set of $y\{y_i; i = 1, 2, 3, \ldots, m\}$ values for some corresponding $x\{x_i; i = 1, 2, 3, \ldots, m\}$ values.A model function $f(x; c_1, ..., c_n)$,of an independent variable x, of known analytical form depending on n parameters is to be fit to the data y_i. To fit the model function $f(x; \boldsymbol{c})$,where $\boldsymbol{c}$ is a vector of n parameters, the sum of squares of the errors (residuals) between the data

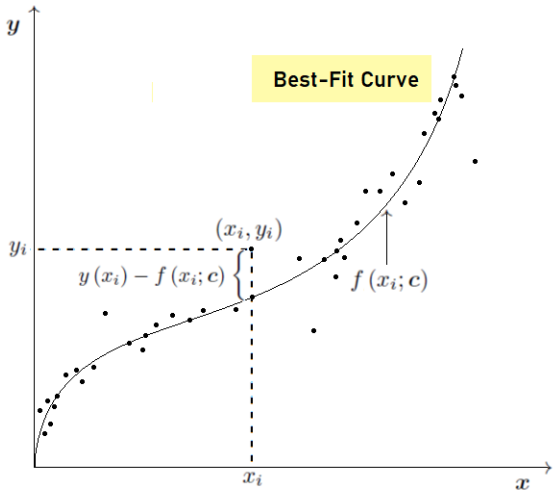


Figure 1: Obtaining residual for model function

$y_i$and the curve-fit function $f(x; \boldsymbol{c})$ is to be minimized.

$$\chi^2(\boldsymbol{c}) = \sum_{i=1}^{m} [y(x_i) - f(x_i; \boldsymbol{c})]^2 \tag{1}$$

This is called the ***chi-squared error criterion***. Expanding the right hand side of the expression explicitly, we can write,

$$\chi^2(\boldsymbol{c}) = \left(\frac{y(x_1) - f(x_1; \boldsymbol{c})}{\sigma_{y_1}}\right)^2 + \ldots + \left(\frac{y(x_m) - f(x_m; \boldsymbol{c})}{\sigma_{y_m}}\right)^2$$

Again, this can be written in the matrix form,

$$\chi^2(\boldsymbol{c}) = \begin{pmatrix} y(x_1) - f(x_1; \boldsymbol{c}) & \ldots & y(x_m) - f(x_m; \boldsymbol{c}) \end{pmatrix} \begin{pmatrix} y(x_1) - f(x_1; \boldsymbol{c}) \\ \vdots \\ \vdots \\ y(x_m) - f(x_m; \boldsymbol{c}) \end{pmatrix}$$

which can be written in a compressed format as,

$$\begin{aligned} \chi^2(\boldsymbol{c}) &= (\boldsymbol{y} - \boldsymbol{f}(\boldsymbol{c}))^T (\boldsymbol{y} - \boldsymbol{f}(\boldsymbol{c})) \\ &= \left(\boldsymbol{y}^T - \boldsymbol{f}^T(\boldsymbol{c})\right)(\boldsymbol{y} - \boldsymbol{f}(\boldsymbol{c})) \\ &= \boldsymbol{y}^T\boldsymbol{y} - \boldsymbol{f}^T(\boldsymbol{c})\,\boldsymbol{y} - \boldsymbol{y}^T\boldsymbol{f}(\boldsymbol{c}) + \boldsymbol{f}^T(\boldsymbol{c})\,\boldsymbol{f}(\boldsymbol{c}) \end{aligned}$$

using,

$$\boldsymbol{y} = \begin{pmatrix} y(x_1) \\ \vdots \\ \vdots \\ y(x_m) \end{pmatrix}_{(m\times 1)} \qquad \boldsymbol{f}(\boldsymbol{c}) = \begin{pmatrix} f(x_1; \boldsymbol{c}) \\ \vdots \\ \vdots \\ f(x_m; \boldsymbol{c}) \end{pmatrix}_{(m\times 1)}$$

Now as$\boldsymbol{y}^T_{(1\times m)}\boldsymbol{f}\left(\boldsymbol{c}\right)_{(m\times 1)}$is simply the dot product in vector notation. And, since dot product is commutative, we can have,

$$\begin{aligned}-\boldsymbol{f}^T\left(\boldsymbol{c}\right)\boldsymbol{y}-\boldsymbol{y}^T\boldsymbol{f}\left(\boldsymbol{c}\right)&=-\boldsymbol{y}^T\boldsymbol{f}\left(\boldsymbol{c}\right)-\boldsymbol{y}^T\boldsymbol{f}\left(\boldsymbol{c}\right)\\&=-2\boldsymbol{y}^T\boldsymbol{f}\left(\boldsymbol{c}\right)\end{aligned}$$

Putting this back into the equation for $\chi^2\left(\boldsymbol{c}\right)$,the final equation stands as,

$$\chi^2\left(\boldsymbol{c}\right)=\boldsymbol{y}^T\boldsymbol{y}-2\boldsymbol{y}^T\boldsymbol{f}\left(\boldsymbol{c}\right)+\boldsymbol{f}^T\left(\boldsymbol{c}\right)\boldsymbol{f}\left(\boldsymbol{c}\right)\tag{2}$$

If the function $f\left(x;\boldsymbol{c}\right)$ is nonlinear in the model parameters $\boldsymbol{c}$, then the minimization of $\chi^2$with respect to the parameters must be carried out iteratively. Each iteration is performed with the aim to find a perturbation $\boldsymbol{\Delta c}=\{\Delta c_i;i=1,2,3,\ldots,n\}$ to the parameters $\boldsymbol{c}$ that reduces χ^2.

2.1.1 The Gauss-Newton Method

[6]The Gauss-Newton method is a method for minimizing a sum-of-squares objective function. It consists of linearizing the model function using a Taylor series expansion around a set of initial parameter values called the preliminary estimates, whereby only the first order partial derivatives are considered. Expanding $f\left(x_i;\boldsymbol{c}+\boldsymbol{\Delta c}\right)$ in Taylor series,

$$f\left(x_i;\boldsymbol{c}+\boldsymbol{\Delta c}\right)\approx f\left(x_i;\boldsymbol{c}\right)+\left[\frac{\partial f\left(x_i;\boldsymbol{c}\right)}{\partial\boldsymbol{c}}\right]\boldsymbol{\Delta c}$$

Here, the term $\left[\frac{\partial f(x_i;\boldsymbol{c})}{\partial\boldsymbol{c}}\right]$ is the gradient of the fit function $f\left(x_i;\boldsymbol{c}\right)$ with respect to the parameters. This treatment can be extended to the matrix $\boldsymbol{f}\left(\boldsymbol{c}\right)$ as shown,

$$\boldsymbol{f}\left(\boldsymbol{c}+\boldsymbol{\Delta c}\right)\approx\boldsymbol{f}\left(\boldsymbol{c}\right)+\left[\frac{\partial\boldsymbol{f}}{\partial\boldsymbol{c}}\right]\boldsymbol{\Delta c}=\boldsymbol{f}\left(\boldsymbol{c}\right)+\boldsymbol{J}\boldsymbol{\Delta c}\tag{3}$$

Here,$\boldsymbol{J}=\left[\frac{\partial\boldsymbol{f}}{\partial\boldsymbol{c}}\right]$is the Jacobian matrix and is explicitly given by,

$$\boldsymbol{J}=\begin{pmatrix}\frac{\partial f(x_1;\boldsymbol{c})}{\partial c_1}&\cdots&\cdots&\frac{\partial f(x_1;\boldsymbol{c})}{\partial c_n}\\\vdots&\ddots&\ddots&\vdots\\\frac{\partial f(x_m;\boldsymbol{c})}{\partial c_1}&\cdots&\cdots&\frac{\partial f(x_m;\boldsymbol{c})}{\partial c_n}\end{pmatrix}_{(m\times n)}\tag{4}$$

Now, using the expression for χ^2(2), the expression for $\chi^2\left(\boldsymbol{c}+\boldsymbol{\Delta c}\right)$ is,

$$\chi^2\left(\boldsymbol{c}+\boldsymbol{\Delta c}\right)=\boldsymbol{y}^T\boldsymbol{y}-2\boldsymbol{y}^T\boldsymbol{f}\left(\boldsymbol{c}+\boldsymbol{\Delta c}\right)+\boldsymbol{f}^T\left(\boldsymbol{c}+\boldsymbol{\Delta c}\right)\boldsymbol{f}\left(\boldsymbol{c}+\boldsymbol{\Delta c}\right)$$

Substituting this approximation $\boldsymbol{f}\left(\boldsymbol{c}+\boldsymbol{\Delta c}\right)\approx\boldsymbol{f}\left(\boldsymbol{c}\right)+\boldsymbol{J}\boldsymbol{\Delta c}$ into the expression for $\chi^2\left(\boldsymbol{c}+\boldsymbol{\Delta c}\right)$,

$$\begin{aligned}\chi^2\left(\boldsymbol{c}+\boldsymbol{\Delta c}\right)&\approx\boldsymbol{y}^T\boldsymbol{y}-2\boldsymbol{y}^T\left(\boldsymbol{f}\left(\boldsymbol{c}\right)+\boldsymbol{J}\boldsymbol{\Delta c}\right)+\left(\boldsymbol{f}\left(\boldsymbol{c}\right)+\boldsymbol{J}\boldsymbol{\Delta c}\right)^T\left(\boldsymbol{f}\left(\boldsymbol{c}\right)+\boldsymbol{J}\boldsymbol{\Delta c}\right)\\&=\boldsymbol{y}^T\boldsymbol{y}-2\boldsymbol{y}^T\boldsymbol{f}\left(\boldsymbol{c}\right)-2\boldsymbol{y}^T\boldsymbol{J}\boldsymbol{\Delta c}+\boldsymbol{f}^T\left(\boldsymbol{c}\right)\boldsymbol{f}\left(\boldsymbol{c}\right)+\boldsymbol{\Delta c}^T\boldsymbol{J}^T\boldsymbol{f}\left(\boldsymbol{c}\right)\\&\quad+\boldsymbol{f}^T\left(\boldsymbol{c}\right)\boldsymbol{J}\boldsymbol{\Delta c}+\boldsymbol{\Delta c}^T\boldsymbol{J}^T\boldsymbol{J}\boldsymbol{\Delta c}\end{aligned}$$

Since, vector dot product is commutative $\boldsymbol{\Delta c}^T\boldsymbol{J}^T\boldsymbol{f}\left(\boldsymbol{c}\right)=\boldsymbol{f}^T\left(\boldsymbol{c}\right)\boldsymbol{J}\boldsymbol{\Delta c}$. therefore,

$$\chi^2\left(\boldsymbol{c}+\boldsymbol{\Delta c}\right)\approx\boldsymbol{y}^T\boldsymbol{y}+\boldsymbol{f}^T\left(\boldsymbol{c}\right)\boldsymbol{f}\left(\boldsymbol{c}\right)-2\boldsymbol{y}^T\boldsymbol{f}\left(\boldsymbol{c}\right)-2\left(\boldsymbol{y}-\boldsymbol{f}\left(\boldsymbol{c}\right)\right)^T\boldsymbol{J}\boldsymbol{\Delta c}+\boldsymbol{\Delta c}^T\boldsymbol{J}^T\boldsymbol{J}\boldsymbol{\Delta c}\tag{5}$$

The first-order Taylor approximation results in an approximation for χ^2that is quadratic in the perturbation $\boldsymbol{\Delta c}$.The parameter update $\boldsymbol{\Delta c}$that minimizes $\chi^2$is found from $\frac{\partial \chi^2}{\partial \boldsymbol{\Delta c}} = 0$.

$$\begin{aligned}
\frac{\partial \chi^2 (\boldsymbol{c} + \boldsymbol{\Delta c})}{\partial \boldsymbol{\Delta c}} &\approx -2 (\boldsymbol{y} - \boldsymbol{f} (\boldsymbol{c}))^T \boldsymbol{J} + \frac{\partial}{\partial \boldsymbol{\Delta c}} \left((\boldsymbol{\Delta c})^T \boldsymbol{J}^T \boldsymbol{J} \boldsymbol{\Delta c} \right) \\
&= -2 (\boldsymbol{y} - \boldsymbol{f} (\boldsymbol{c}))^T \boldsymbol{J} + \left(\frac{\partial}{\partial \boldsymbol{\Delta c}} \left((\boldsymbol{\Delta c})^T \boldsymbol{J}^T \boldsymbol{J} \right) \right) \boldsymbol{\Delta c} + (\boldsymbol{\Delta c})^T \frac{\partial}{\partial \boldsymbol{\Delta c}} \left(\boldsymbol{J}^T \boldsymbol{J} \boldsymbol{\Delta c} \right) \\
&= -2 (\boldsymbol{y} - \boldsymbol{f} (\boldsymbol{c}))^T \boldsymbol{J} + \boldsymbol{J}^T \boldsymbol{J} \boldsymbol{\Delta c} + (\boldsymbol{\Delta c})^T \boldsymbol{J}^T \boldsymbol{J} \quad \left[\because \frac{\partial}{\partial \boldsymbol{a}} \left(\left(\boldsymbol{a}^T \right)_{(1 \times n)} \boldsymbol{b}_{(1 \times n)} \right) = \boldsymbol{b}, \right] \\
\therefore \frac{\partial \chi^2 (\boldsymbol{c} + \boldsymbol{\Delta c})}{\partial \boldsymbol{\Delta c}} &\approx -2 (\boldsymbol{y} - \boldsymbol{f} (\boldsymbol{c}))^T \boldsymbol{J} + 2 \boldsymbol{J}^T \boldsymbol{J} \boldsymbol{\Delta c}
\end{aligned}$$

and the resulting normal equations for the Gauss-Newton update are,

$$\left[\boldsymbol{J}^{\boldsymbol{T}} \boldsymbol{J} \right] \boldsymbol{\Delta c}_{gn} = \boldsymbol{J}^T (\boldsymbol{y} - \boldsymbol{f} (\boldsymbol{c})) \tag{6}$$

2.1.2 The Levenberg-Marquardt Method

[7, 8]This method (also known as damped least squares method) can be considered as an interpolation between the Gauss-Newton and the gradient descent method. even if the initial guess is far from the solution corresponding to the minimum of the objective function, the iteration can can still converge towards the solution. By adding an extra term proportional to the identity matrix $\boldsymbol{I}$, the normal equation for Gauss-Newton method is modified as,

$$\left[\boldsymbol{J}^{\boldsymbol{T}} \boldsymbol{J} + \lambda \boldsymbol{I} \right] \boldsymbol{\Delta c}_{lm} = \boldsymbol{J}^T (\boldsymbol{y} - \boldsymbol{f} (\boldsymbol{c})) \tag{7}$$

Here, λ is the non-negative damping factor [6], which is to be adjusted at each iteration. Small values of the damping parameter λresult in a Gauss-Newton update and large values of λ result in a gradient descent update.The damping parameter λ is initialized to be large so that first updates are small steps in the steepest-descent direction. If any iteration happens to result in a worse approximation $\left(\chi^2 (\boldsymbol{c} + \boldsymbol{\Delta c}_{lm}) > \chi^2 (\boldsymbol{c}) \right)$, then $\lambda$is increased. Otherwise, as the solution improves, λ is decreased, the Levenberg-Marquardt method approaches the Gauss-Newton method, and the solution typically accelerates to the local minimum. In Marquardt's update relationship, the values of λ are normalized to the values of $\boldsymbol{J}^{\boldsymbol{T}} \boldsymbol{J}$.

$$\left[\boldsymbol{J}^{\boldsymbol{T}} \boldsymbol{J} + \lambda diag \left(\boldsymbol{J}^{\boldsymbol{T}} \boldsymbol{J} \right) \right] \boldsymbol{\Delta c}_{lm} = \boldsymbol{J}^T (\boldsymbol{y} - \boldsymbol{f} (\boldsymbol{c})) \tag{8}$$

In case the functional form of $f (x_i; \boldsymbol{c})$ is not known, the residual $\boldsymbol{R} = \boldsymbol{y} - \boldsymbol{f} (\boldsymbol{c})$, is linearized, and the matrix equation is modified as,

$$\left[\boldsymbol{J}^{\boldsymbol{T}} \boldsymbol{J} + \lambda diag \left(\boldsymbol{J}^{\boldsymbol{T}} \boldsymbol{J} \right) \right] \boldsymbol{\Delta c}_{lm} = \boldsymbol{J}^T \boldsymbol{R} \tag{9}$$

where,

$$\boldsymbol{J} = \begin{pmatrix} \frac{\partial r_1 (\boldsymbol{c})}{\partial c_1} & \dots & \dots & \frac{\partial r_1 (\boldsymbol{c})}{\partial c_n} \\ \vdots & \ddots & \ddots & \vdots \\ \frac{\partial r_m (\boldsymbol{c})}{\partial c_1} & \dots & \dots & \frac{\partial r_m (\boldsymbol{c})}{\partial c_n} \end{pmatrix}_{(m \times n)}$$

2.1.3 Numerical Implementation

The fitting is carried out iteratively by evaluating the matrix equation with the updated parameter list $\boldsymbol{c}$. For every iteration the Jacobian $\boldsymbol{J}$ is approximated numerically using forward differences,

$$\boldsymbol{J}_{ij} = \frac{\partial f (x_i; \boldsymbol{c})}{\partial c_j} = \frac{f(x_i; \boldsymbol{c} + \delta \boldsymbol{c}_j) - f(x_i; \boldsymbol{c})}{\| \delta \boldsymbol{c}_j \|} \tag{10}$$

```
def Jacobian(f, x_vect, p_vect, args, delta_p):
        n = len(p_vect)
        J = []

        #computing rows for the Jacobian
        for p_index in range(n):
                #computing the parameter vector for partial derivatives
                p_vect_dashed = p_vect.copy()
                p_vect_dashed[p_index] = p_vect_dashed[p_index] + delta_p
                #computing functional values for partial derivative
                f_p_vect_dashed = f(x_vect, p_vect_dashed, *args)
                f_p_vect = f(x_vect, p_vect, *args)
                #computing the partial derivatives for the Jacobian
                f_deriv_p = (f_p_vect_dashed - f_p_vect) / delta_p
                J.append(f_deriv_p)

        return np.matrix(J).T
```

In iteration i, the step $\boldsymbol{\Delta c}_{lm}$ is evaluated by comparing $\chi^2(\boldsymbol{c})$ to $\chi^2(\boldsymbol{c}+\boldsymbol{\Delta c}_{lm})$. The step is accepted if the metric ρ_i is greater than a user specified threshold, $\epsilon_4 > 0$[9]. This metric is a measure of the improvement in χ^2 as compared to the improvement of an LM update assuming the approximation $\boldsymbol{f}(\boldsymbol{c}+\boldsymbol{\Delta c}) \approx \boldsymbol{f}(\boldsymbol{c}) + \left[\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{c}}\right]\boldsymbol{\Delta c}$ is exact. The metric is evaluated by the expression,

$$\rho_i(\boldsymbol{\Delta c}_{lm}) = \frac{\chi^2(\boldsymbol{c}) - \chi^2(\boldsymbol{c}+\boldsymbol{\Delta c}_{lm})}{\boldsymbol{\Delta c}_{lm}^T\left(\lambda_i \text{diag}(\boldsymbol{J}^T\boldsymbol{J})\boldsymbol{\Delta c}_{lm} + \boldsymbol{J}^T(\boldsymbol{y}-\boldsymbol{f}(\boldsymbol{c}))\right)} \tag{11}$$

If in the i^{th} iteration, $\rho_i(\boldsymbol{\Delta c}_{lm}) > \epsilon_4$ then $\boldsymbol{c}+\boldsymbol{\Delta c}_{lm}$ is sufficiently better than $\boldsymbol{c}$, then $\boldsymbol{c}$ is replaced by $\boldsymbol{c}+\boldsymbol{\Delta c}_{lm}$ and then λ is reduced by a factor. Otherwise λ is increased by a factor, and the algorithm proceeds to the next iteration. The values of λ and parameters $\boldsymbol{c}$ are initialized and updated using the following relations:

- $\lambda_{initial} = \lambda_0$ is specified by the user, and is generally chosen to be large, so that first updates are small steps in the steepest-decsent direction. $\lambda_0$chosen as 1.0 seemed to show good convergence properties.
- if $\rho_i(\boldsymbol{\Delta c}_{lm}) > \epsilon_4$: $\boldsymbol{c} \leftarrow \boldsymbol{c}+\boldsymbol{\Delta c}_{lm}; \lambda_{i+1} = max\left[\lambda_i/L_\downarrow, 10^{-7}\right]$; otherwise : $\lambda_{i+1} = min\left[\lambda_i/L_\uparrow, 10^7\right]$, with $L_\uparrow \approx 11$ and $L_\downarrow \approx 9$, exhibiting good convergence properties[9].

```
#computing the metric measuring the goodness of the fit
rho = (chi_squared_p - chi_squared_ph)/((h.T).dot(lambda_0*A_diag.dot(h)+B))

#checking the threshold for the rho
if rho > eps[3]:
        #updating the fit parameters
        p0+=h_flat
        #down-scaling the damping factor lambda
        lambda_0=max((lambda_0/L_scale_down), 1.0e-7)

else:
        #up-scaling the damping factor lambda
        lambda_0=min((lambda_0*L_scale_up), 1.0e7)
```

The iterative process is halted once the convergence criteria for the fit is met. Convergence is achieved and the iteration terminates when one of the following three criteria[6] is satisfied,

- Convergence in the gradient : $\max \left| \boldsymbol{J}^T \left(\boldsymbol{y} - \boldsymbol{f} \left(\boldsymbol{c} \right) \right) \right| < \epsilon_1$
- Convergence in the parameters : $\max |h_i / p_i| < \epsilon_2$
- Convergence in χ^2 : $\chi^2_v = \frac{\chi^2}{(m-n-1)} < \epsilon_3$

```
#expression for the convergence criterion
exp1 = np.abs(B).max()
exp2 = np.abs(h/p0).max()
exp3 = chi_squared/(m-n-1)

#checking the convergence in gradient
if  exp1 < eps1:
        return True

#checking the convergence in parametets
if  exp2 < eps2:
        return True

#checking the convergence in the chi-squared
if  exp3 < eps3:
        return True
return False
```

[*usage in sections* (4.5), (8.4.7), (8.4.8)]

2.2 Singular Value Decomposition

Singular value decomposition[10] is a method of decomposing a matrix into three other matrices, in the following manner:

$$\boldsymbol{A} = \boldsymbol{U}\boldsymbol{S}\boldsymbol{V}^T \tag{12}$$

where,

- $\boldsymbol{A}$ is an $m \times n$ matrix
- $\boldsymbol{U}$ is an $m \times n$ orthogonal matrix
- $\boldsymbol{S}$ is an $n \times n$ diagonal matrix
- $\boldsymbol{V}$ is an $n \times n$ orthogonal matrix

In index notation, the decomposition is explicitly given by,

$$a_{ij} = \sum_{k=1}^{n} u_{ik} s_k v_{jk}$$

The variables $\{s_k\}$ are called the singular values of the diagonal matrix S and are normally arranged from largest to smallest, i.e.

$$s_{i+1} \leq s_i$$

The columns of $\boldsymbol{U}$ are called *left singular vectors,* while those of V are called r*ight singular vectors.* The matrices $\boldsymbol{U}$ and $\boldsymbol{V}$ being orthogonal, we can write:

$$\boldsymbol{U}^T\boldsymbol{U} = \boldsymbol{V}\boldsymbol{V}^T = \boldsymbol{I}$$

where $\boldsymbol{I}$ is the identity matrix. Also since $\boldsymbol{U}$ is not a square matrix, we cannot have $\boldsymbol{U}\boldsymbol{U}^T = \boldsymbol{I}$. Thus $\boldsymbol{U}$ is orthogonal only in one direction. Using this orthogonality properties, we can rearrange equation (12) into the following pair of eigenvalue operations,

$$\boldsymbol{A}\boldsymbol{A}^T\boldsymbol{U} = \boldsymbol{U}\boldsymbol{S}^2 \tag{13}$$

$$\boldsymbol{A}\boldsymbol{A}^T\boldsymbol{V} = \boldsymbol{V}\boldsymbol{S}^2 \tag{14}$$

Since $\boldsymbol{A}^T\boldsymbol{A}$ is the same size or smaller that $\boldsymbol{A}\boldsymbol{A}^T$, $\boldsymbol{V}$ and $\boldsymbol{S}^2$ is computed by solving the eigenvalue equation (14), and then $\boldsymbol{U}$ is found out by rearranging the equation (12),

$$\boldsymbol{U} = \boldsymbol{A}\boldsymbol{V}\boldsymbol{S}^{-1}$$

[*usage in sections* (4.3), (6.3)]

Besides these, following are the list of the common mathematical tools used in the following discussion:

- **Eigenvalue Decomposition**[11] [*usage in sections* (4.3), (4.4), (6.3)]
- **Vector Algebra and Vector Calculus**[12] [*usage in section* (7)]
- **Linear Algebra**[11] [*usage in sections* (2.1), (3), (4), (6)]
- **Numerical Integration**[13] [*usage in sections* (32)]
- **Numerical Solution to Ordinary Differential Equations[14]** [*usage in sections* (7.1.4), (7.2.3), (8.4.8)]

3 Camera Modeling

3.1 Pinhole Camera Model

[15]Let us consider a construction with a small aperture between the 3D object and photographic plane functioning as a pinhole camera as shown in the figure. Each point of the 3D object corresponds to a point on the image plane because the aperture allows only one of the multiple rays emitted from each point. Thus, we are able to establish a **one-to-one mapping** between the points on the 3D object and the film.
Let $\overrightarrow{P} = \begin{bmatrix} x & y & z \end{bmatrix}^T$ be any point on the 3D object that is visible to the pinhole camera. Now, let $\overrightarrow{P}' = \begin{bmatrix} x' & y' \end{bmatrix}^T$be the point mapped onto the image plane corresponding to $\overrightarrow{P}$. Every point including the camera center O is mapped onto the image plane. Let O be mapped to C'. The coordinate system centered at the O is called the ***camera reference system*** or ***camera coordinate system***. OC′ defines the ***optic axis*** of this camera system. The main aim is to derive a relationship between 3D point $\overrightarrow{P}$ and image point $P'$to understand how this transformation from 3D to 2D coordinates takes place. When a lens is used, O corresponds to the center of the lens. The distance between the point of focus and O is known as the ***focal length*** f**.** The point where the z axis meets the 3D object is named C.

$$\triangle B'C'O \sim \triangle BCO \;\; and \;\; \triangle A'C'O \sim \triangle ACO$$

Using law of similar triangles,

$$\frac{x'}{f} = \frac{x}{z} \quad \frac{y'}{f} = \frac{y}{z} \tag{15}$$

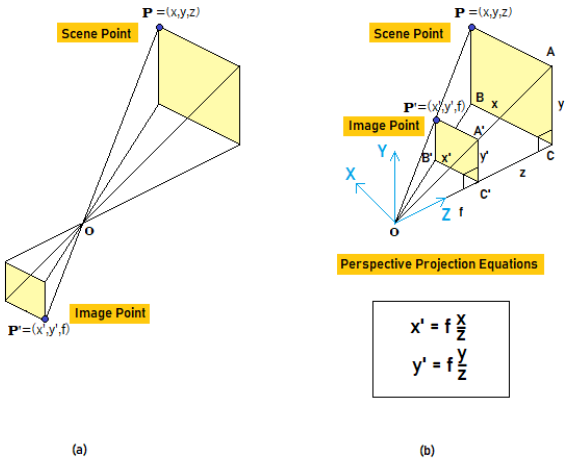


Figure 2: Pinhole camera projection (a) without inversion (b) with inversion

therefore, the point $\overrightarrow{P}'$ can be written as,

$$\overrightarrow{P}' = \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} f\frac{x}{z} \\ f\frac{y}{z} \end{bmatrix} \tag{16}$$

The mapping of a point $\overrightarrow{P}$ in 3D space into a 2D point $\overrightarrow{P}'$in the image plane $\left(\mathbb{R}^3 \rightarrow \mathbb{R}^2\right)$ is referred to as a ***projective transformation***[16]. This projection of 3D points into the image plane does not directly correspond to what we see in actual digital images for several reasons. Firstly, points in the digital images are, in general, in a different reference system than those in the image plane. Secondly, digital images are divided into discrete pixels, whereas points in the image plane are continuous. Finally, the physical sensors can introduce non-linearity such as distortion to the mapping. To account for these differences, we will introduce a number of additional transformations that allow us to map any point from the 3D world to pixel coordinates.

3.2 The Camera Matrix Model and Homogeneous Coordinates

[15]The image plane coordinates have their center at the image center, while typically digital image coordinates have their origin at the upper-left corner of the image. Thus, 2D points in the image plane and 2D points in the image are offset by a translation vector $\begin{bmatrix} c_x & c_y \end{bmatrix}^T$. Thus, $\overrightarrow{P}'$becomes,

$$\overrightarrow{P}' = \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} f\frac{x}{z} + c_x \\ f\frac{y}{z} + c_y \end{bmatrix} \tag{17}$$

Points in digital images are expressed in pixels, while points in image plane are represented in physical measurements, therefore a scale factor has to be multiplied. Therefore,

$$\overrightarrow{p} = \begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} fk\frac{x}{z} + kc_x \\ fl\frac{y}{z} + lc_y \end{bmatrix} = \begin{bmatrix} \alpha\frac{x}{z} + C_x \\ \beta\frac{y}{z} + C_y \end{bmatrix} \tag{18}$$

Here k and l are parameters with units $\frac{pixels}{cm}$,which are different in general because the aspect ratio of a pixel is not guaranteed to be one.As it is evident the projection $\overrightarrow{P} \rightarrow \overrightarrow{P}'$is non-linear (owing to the division by one of the input parameters z),so to represent this projection as a matrix-vector product ***homogeneous coordinates*** are used. In ***homogeneous coordinate system***, the image coordinates is transformed to,

$$\overrightarrow{U} = (u, v) \rightarrow (u, v, 1) \tag{19}$$

Similarly, any point $\overrightarrow{P} = (x, y, z)$ becomes $(x, y, z, 1)$. To convert a Euclidean vector $(v_1, ..., v_n)$ to homogeneous coordinates, a 1 is appended in a new dimension to get $(v_1, ..., v_n, 1)$. The equality between any vector and its homogeneous coordinates can be established only if the final coordinate is equal to 1.

$$Homogeneous\ coordinates\ (v_1, ..., v_n, w) \rightarrow Euclidean\ coordinates\ \left(\frac{v_1}{w}, ..., \frac{v_n}{w}\right)$$

Therefore the relationship between any point in 3D space and its image coordinates (18) in homogeneous coordinates can be written as:

$$\overrightarrow{U}' = \begin{bmatrix} u' \\ v' \\ z \end{bmatrix} = \begin{bmatrix} \alpha x + C_x z \\ \beta y + C_y z \\ z \end{bmatrix} = \begin{bmatrix} \alpha & 0 & C_x & 0 \\ 0 & \beta & C_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} \alpha & 0 & C_x & 0 \\ 0 & \beta & C_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} P = \boldsymbol{M}P \tag{20}$$

where,

$$u = \frac{u'}{z} \quad v = \frac{v'}{z} \tag{21}$$

This is the matrix vector relationship, and can be written as,

$$\overrightarrow{U}' = MP = \begin{bmatrix} \alpha & 0 & C_x \\ 0 & \beta & C_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} I & 0 \end{bmatrix} \overrightarrow{P} = \boldsymbol{K} \begin{bmatrix} \boldsymbol{I} & 0 \end{bmatrix} \overrightarrow{P} \tag{22}$$

$\boldsymbol{K}$ is the ***camera matrix*** and is written as,

$$\boldsymbol{K} = \begin{bmatrix} \alpha & 0 & C_x \\ 0 & \beta & C_y \\ 0 & 0 & 1 \end{bmatrix} \tag{23}$$

3.3 Extrinsic Parameters

[15]Usually, the information about the 3D world is available in a different coordinate system, so an additional transformation is required to relate points from the world reference system to the camera reference system.This transformation can be carried out by performing a rotation operation followed by a translation. This is captured by a rotation matrix $\boldsymbol{R}$ and translation vector $\overrightarrow{T}$. Thus, any point in the world reference system $\overrightarrow{W}$ can be written as:

$$\overrightarrow{P} = \begin{bmatrix} \boldsymbol{R} & \boldsymbol{T} \\ 0 & 1 \end{bmatrix} \overrightarrow{W} \tag{24}$$

Substituting this in $\overrightarrow{U}' = \boldsymbol{K} \begin{bmatrix} \boldsymbol{I} & 0 \end{bmatrix} \overrightarrow{P}$ (22), we get,

$$\overrightarrow{U}' = \boldsymbol{K} \begin{bmatrix} \boldsymbol{R} & \overrightarrow{T} \end{bmatrix} \overrightarrow{W} = \boldsymbol{M}\overrightarrow{W} \tag{25}$$

$\boldsymbol{R}$ and $\overrightarrow{T}$ are know as the **extrinsic parameters** as they are dependent on the external world reference system and not on the camera. This is the mapping from a 3D point in an arbitrary world reference system to the image plane. The full projection matrix $\boldsymbol{M}$ consists of the **intrinsic** and **extrinsic** parameters. $\boldsymbol{K}$consists of all the intrinsic parameters that change with the type of camera used.The extrinsic parameters include the rotation and translation, which do not depend on the camera's build. It can be observed that the 3×4 projection matrix $\boldsymbol{M}$ has 11 degrees of freedom: 5 from the intrinsic camera matrix, 3 from extrinsic rotation, and 3 from extrinsic translation.

World Coordinates		*Camera Coordinates*		*Film Coordinates*		*Pixel Coordinates*
X		x		x'		u
Y	$\rightarrow$	y	$\rightarrow$	y'	$\rightarrow$	v
Z		z				

3.4 Numerical Implementation

A script is used to implement the mathematical model of a camera. A pair of cameras C_{m1} and C_{m2} are used to project 3D points onto the image planes of the cameras[16]. The camera matrices are considered identical and are given by:

Camera	$\boldsymbol{C_{m1}}$	$\boldsymbol{C_{m2}}$
Camera Matrices	$\begin{pmatrix} 300 & 0 & 150 \\ 0 & 300 & 150 \\ 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 300 & 0 & 150 \\ 0 & 300 & 150 \\ 0 & 0 & 0 \end{pmatrix}$

The snippet defining the camera parameters and the projection matrices is as follows:

```
#setting the camera intrinsic matrix for camera 1
M_int_c1=[[300,0,150, 0],
                  [0,300,150, 0],
                  [0,0,1,0]]

#setting the camera intrinsic matrix for camera 2
M_int_c2=[[300,0,150, 0],
                  [0,300,150, 0],
                  [0,0,1,0]]

M_int_c1=np.matrix(M_int_c1)
M_int_c2=np.matrix(M_int_c2)
```

Consider a set of 3D world points on the surface of a cube. The homogeneous $(X, Y, Z, 1)$ coordinates for the points are listed in the table below.

X	0	0	0	0	0	0	0	0	0	1	2	1	2	1	2
Y	2	1	0	2	1	0	2	1	0	0	0	0	0	0	0
Z	0	0	0	-1	-1	-1	-2	-2	-2	0	0	-1	-1	-2	-2
H	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Let the 3D coordinates be defined under a matrix named P_M. The camera projection matrix for C_{m1} is obtained by defining the rigid translation and rotation of the camera with respect to the world frame in which the 3D points are defined. Let the rigid rotations about the individual axes be given by.

$$\theta_x = 120^{\circ}$$
$$\theta_y = 0^{\circ}$$
$$\theta_z = 60^{\circ}$$

The corresponding rotation matrices for an arbitrary angles θ_x, θ_y and θ_z are given by,

$$\boldsymbol{R}_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta_x & -\sin\theta_x \\ 0 & \sin\theta_x & \cos\theta_x \end{bmatrix} \quad \boldsymbol{R}_y = \begin{bmatrix} \cos\theta_x & 0 & -\sin\theta_x \\ 0 & 1 & 0 \\ \sin\theta_x & 0 & \cos\theta_x \end{bmatrix} \quad \boldsymbol{R}_z = \begin{bmatrix} \cos\theta_x & -\sin\theta_x & 0 \\ \sin\theta_x & \cos\theta_x & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

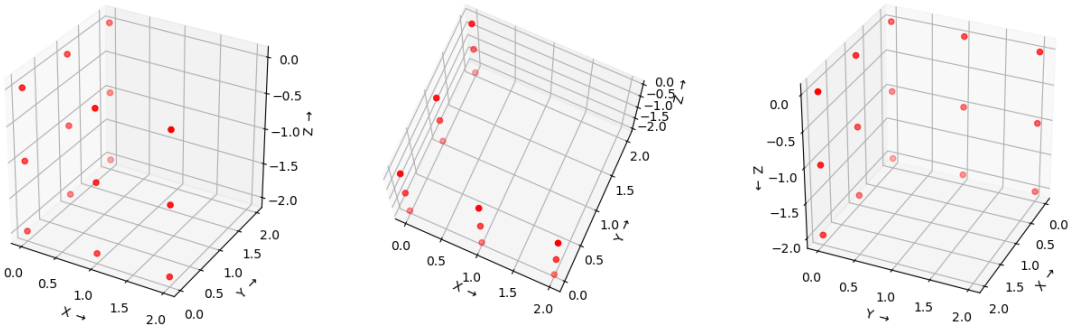


Figure 3: 3D points on the surface of cube

The combined rotation and the translation for C_{m1} can be obtained as,

```
#calculating the effective rotation matrix
R_m_c1=R_x*R_y*R_z

#defining the translation vector for  model wrt to the camera 1
T_m_c1=np.matrix([[0],[0],[5]])
```

The final camera projection matrix for C_{m1} is then computed by combining the extrinsic camera matrix with the intrinsic camera matrix.

```
#setting the camera extrinsic matrix for camera 1
M_ext_c1=np.vstack((np.hstack((R_m_c1,T_m_c1)),np.r_[0,0,0,1]))

#setting the effective transformation matrix from the
#world coordinates to the image coordinates for camera 1
M_c1=M_int_c1*M_ext_c1
```

Similarly the camera projection matrix for C_{m2} is obtained by calculating the homography between the two camera scenes. This is done by defining the relative translation and rotation of C_{m2} with respect to that of C_{m1} and then using the computed homography to obtain the camera extrinsic matrix. Finally the camera projection matrix for C_{m2} is obtained.

```
#using the homography to find the extrinsic matrix for camera 2
M_c2_c1=np.vstack((np.hstack((R_c2_c1,T_c2_c1)),np.r_[0,0,0,1]))
M_c1_c2=np.linalg.inv(M_c2_c1)
M_ext_c2=M_c1_c2*M_ext_c1

#setting the effective transformation matrix from the world coordinates
#to the image coordinates for camera 2
M_c2=M_int_c2*M_ext_c2
```

The image points are then obtained by projecting the world points onto the image planes using the camera projection matrices.

```
#obtaing the homogeneous image coordinates for camera 1
#by applying the transformation matrix for camera 1
p1=M_c1*P_M

```

```
#obtaing the homogeneous image coordinates for camera 2
#by applying the transformation matrix for camera 2
p2=M_c2*P_M
```

The computed values of the pixels coordinates are in homogeneous form. The euclidean values of the image coordinates are then obtained by dividing the first two array coordinate elements with the last element. The images seen by C_{m1} and C_{m2} can be visualized as shown.

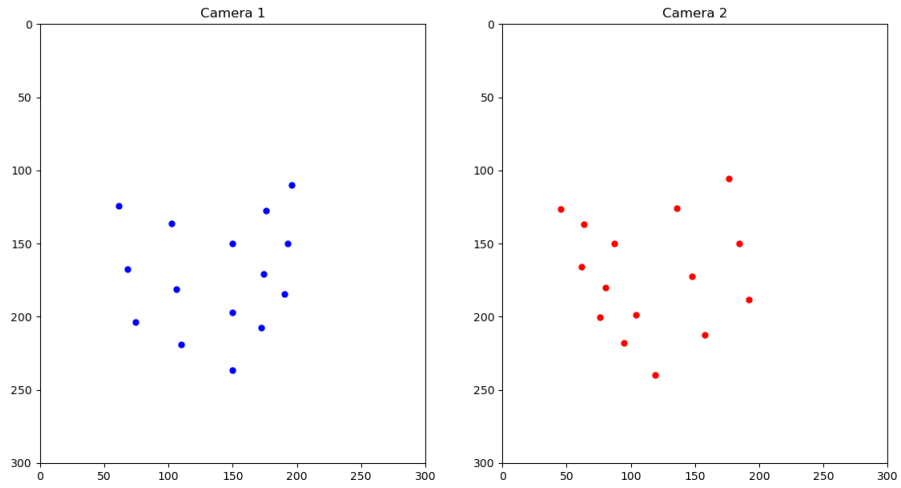


Figure 4: Model camera views

4 Fundamental Matrix and 3D reconstruction

4.1 Epipolar Geometry

[15]In general, it is not possible to extract complete information about 3D structures from just a single camera. This is due to the intrinsic ambiguity of the mapping from 3D world to the camera plane; some information is simply lost. The *epipolar geometry* of a pair of cameras expresses the fundamental relationship between any two corresponding points in the two image planes, and leads to a key constraint between the coordinates of these points that underlies visual reconstruction. Structure from motion employs epipolar geometry to estimate the relative orientation between camera poses an hence reconstruct the 3D information of the scene under study. Given an arbitrary scene, we can determine the 3D position of points in the scene as well as the ***relative camera positions*** in terms of rotation and translation. However the positions will vary by an ***unknown scale factor*** in terms of the point positions and the camera translations, which cannot be determined without a separate ***calibration procedure***. First a generalized treatment is done to obtain the ***fundamental matrix*** encoding the information about the camera ***relative orientation***. Next we can use the fundamental matrix and the corresponding images to obtain a 3D reconstruction of the scene.
A standard epipolar geometry setup consists of two cameras (C_{m1} and C_{m2}) placed at a fixed relative orientation, observing the same 3D world point $\overrightarrow{P}$. Let the projections of this point on the two camera planes be $\overrightarrow{p}_1$ and $\overrightarrow{p}_2$ respectively and O_1 and O_2be the ***camera centers***. The line joining the two camera centers is the ***baseline*** and the plane defined by the two camera centers and the ***world point*** $\overrightarrow{P}$ is called the ***epipolar plane***. The points of intersection of the baseline with the image planes are the ***epipoles*** e_1 and e_2. The lines formed by the intersection of the image and the epipolar planes are the ***epipolar lines*** l_1 and l_2. Note that the epipolar lines intersect the baseline at the epipoles.
[15]Let $\boldsymbol{M_1}$ and $\boldsymbol{M_2}$ be the ***camera projection matrices*** for the two cameras that maps the 3D points

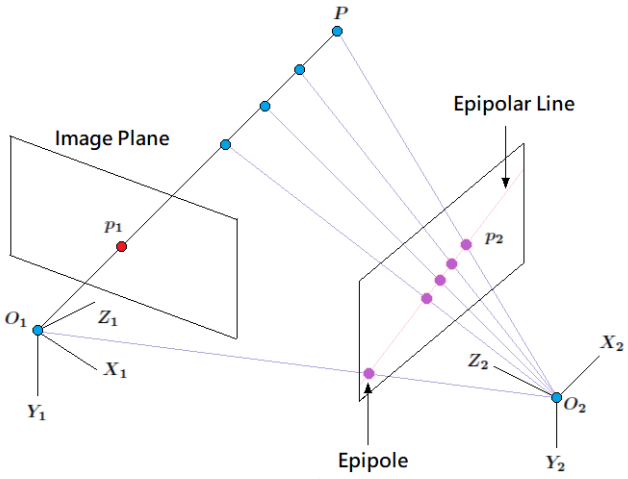


Figure 5: Epipolar geometry setup

from the world frame to their corresponding image points on the 2D camera planes. The position of camera C_{m1} is chosen such that its camera center coincides with the origin of the world coordinate system, with the second camera C_{m2} offset by some rotation $\boldsymbol{R}$ and translation $\overrightarrow{T}$. Thus the camera projection matrices (25) can be written as,

$$\boldsymbol{M_1} = \boldsymbol{K_1}\begin{bmatrix}\boldsymbol{I} & \overrightarrow{O}\end{bmatrix} \tag{26}$$

$$\boldsymbol{M_2} = \boldsymbol{K_2}\begin{bmatrix}\boldsymbol{R} & \overrightarrow{T}\end{bmatrix} \tag{27}$$

4.2 Essential Matrix

For simplicity let us consider the case of ***canonical cameras***, for which $\boldsymbol{K_1} = \boldsymbol{K_2} = \boldsymbol{I}$. Thus the camera matrices are now given by (28)(29),

$$\boldsymbol{M_1} = \begin{bmatrix}\boldsymbol{I} & \overrightarrow{O}\end{bmatrix} \tag{28}$$

$$\boldsymbol{M_2} = \begin{bmatrix}\boldsymbol{R} & \overrightarrow{T}\end{bmatrix} \tag{29}$$

Using canonical forms for the cameras, the image coordinates $\overrightarrow{p}_1$ and $\overrightarrow{p}_2$, are now transformed into their respective canonical forms $\overrightarrow{p}_{c1}$ and $\overrightarrow{p}_{c2}$ obtained by the operation of the modified camera projection matrices $\boldsymbol{M_1}$ and $\boldsymbol{M_2}$ respectively on the 3D point $\overrightarrow{P}$. Since an image point is measured in the camera's own reference frame, the rigid transformation(24) between the two ***camera coordinate frames*** is given by $\overrightarrow{p}_{c2} = \boldsymbol{R}\left(\overrightarrow{p}_{c1} - \overrightarrow{T}\right)$. Solving for $\overrightarrow{p}_{c1}$, the reverse transformation is then given by,

$$\begin{aligned}\overrightarrow{p}_{c1} &= \boldsymbol{R}^T\overrightarrow{p}_{c2} + \overrightarrow{T} \\ &= \boldsymbol{R}^T\left(\overrightarrow{p}_{c2} + \boldsymbol{R}\overrightarrow{T}\right)\end{aligned} \tag{30}$$

Thus if the rotation and the translation for the reverse transformation are given by $\boldsymbol{R}'$ and $\overrightarrow{T}'$, then

$$\boldsymbol{R}' = \boldsymbol{R}^T \tag{31}$$

$$\overrightarrow{T}' = -\boldsymbol{R}\overrightarrow{T} \tag{32}$$

Thus when expressed in the reference frame of C_{m1}, the directions of the projection rays from 3D point $\overrightarrow{P}$ through the corresponding image points with coordinates $\overrightarrow{p}_{c1}$ and $\overrightarrow{p}_{c2}$ are along the vectors $\overrightarrow{p}_{c1}$ and $\boldsymbol{R}^T\overrightarrow{p}_{c2}$. Also the baseline in this reference frame is along the ***translation vector*** $\overrightarrow{T}$. Epipolar geometry dictates that the vectors $\overrightarrow{p}_{c1}$, $\boldsymbol{R}^T\overrightarrow{p}_{c2}$ and $\overrightarrow{T}$ are confined to the ***epipolar plane***.[17] ***Co-planarity of the vectors*** requires the ***vector triple product*** to be zero.

$$\begin{aligned}\left(\boldsymbol{R}^T\overrightarrow{p}_{c2}\right)\cdot\left(\overrightarrow{T}\times\overrightarrow{p}_{c1}\right)&=0\\ \left(\boldsymbol{R}^T\overrightarrow{p}_{c2}\right)^T\left(\overrightarrow{T}\times\overrightarrow{p}_{c1}\right)&=0\\ \left(\overrightarrow{p}_{c2}\right)^T\boldsymbol{R}\left(\overrightarrow{T}\times\overrightarrow{p}_{c1}\right)&=0\end{aligned}\tag{33}$$

From linear algebra, we can introduce a different and compact expression for the ***cross product***: we can represent the cross product between any two vectors $\overrightarrow{a}$ and $\overrightarrow{b}$ as a matrix vector multiplication:

$$\overrightarrow{a}\times\overrightarrow{b}=\begin{bmatrix}0&-a_z&a_y\\a_z&0&-a_x\\-a_y&a_x&0\end{bmatrix}\begin{bmatrix}b_x\\b_y\\b_z\end{bmatrix}=[\overrightarrow{a}]_\times\overrightarrow{b}\tag{34}$$

Using this notation we can express $\overrightarrow{T}\times\overrightarrow{p}_{c1}$ as $\left[\overrightarrow{T}\right]_\times\overrightarrow{p}_{c1}$ and thus **equation number** [16]can be rewritten as,

$$\begin{aligned}\left(\overrightarrow{p}_{c2}\right)^T\left(\boldsymbol{R}\left[\overrightarrow{T}\right]_\times\right)\overrightarrow{p}_{c1}&=0\\ \left(\overrightarrow{p}_{c2}\right)^T\boldsymbol{E}\overrightarrow{p}_{c1}&=0\end{aligned}\tag{35}$$

where $\boldsymbol{E}=\boldsymbol{R}\left[\overrightarrow{T}\right]_\times$ is called the *essential matrix* and the **equation number** is called the *epipolar constraint*. The **equation number** expresses the ***co-planarity between any two points*** on the same epipolar plane for two fixed cameras. The essential matrix is a 3×3 singular matrix with ***5 degrees of freedom*** that relates the image of a point in one camera to its image in the other camera, given a translation and rotation. It can be shown that the essential matrix has ***rank 2*** for any non-zero $\overrightarrow{T}$. Note that the matrix $\left[\overrightarrow{T}\right]_\times$ has rank 2 due to the fact that,$\left[\overrightarrow{T}_\times\right]\overrightarrow{T}=\overrightarrow{T}\times\overrightarrow{T}=0$ so the null space of $\left[\overrightarrow{T}\right]_\times$ is the line through the origin along $\overrightarrow{T}$. Since $\boldsymbol{R}$ is full rank, thus $\boldsymbol{E}=\boldsymbol{R}\left[\overrightarrow{T}\right]_\times$ must have rank 2.

4.3 Fundamental Matrix

[15]Now we can extend this treatment for non-canonical cameras, by retaining the camera intrinsic matrix in the camera projection matrix(25):

$$\boldsymbol{M_1}=\boldsymbol{K_1}\begin{bmatrix}\boldsymbol{I}&\overrightarrow{O}\end{bmatrix}\tag{36}$$

$$\boldsymbol{M_2}=\boldsymbol{K_2}\begin{bmatrix}\boldsymbol{R}&\overrightarrow{T}\end{bmatrix}\tag{37}$$

Thus if $\overrightarrow{p}_{c1}$ and $\overrightarrow{p}_{c2}$ are the ***normalized/canonical coordinates*** for canonical cameras C_{m1} and C_{m2} respectively then, we can write:

$$\overrightarrow{p}_{c1}=\boldsymbol{K_1^{-1}}\left(\boldsymbol{K_1}\begin{bmatrix}\boldsymbol{I}&\overrightarrow{O}\end{bmatrix}\overrightarrow{P}\right)=K_1^{-1}\overrightarrow{p}_1\tag{38}$$

$$\overrightarrow{p}_{c2}=\boldsymbol{K_2^{-1}}\left(\boldsymbol{K_2}\begin{bmatrix}\boldsymbol{R}&\overrightarrow{T}\end{bmatrix}\overrightarrow{P}\right)=K_2^{-1}\overrightarrow{p}_2\tag{39}$$

where $\overrightarrow{p}_1$ and $\overrightarrow{p}_2$ are the ***unnormalized coordinates*** for the cameras C_{m1} and C_{m2} respectively. Substituting the values of the normalized coordinates into the equation for epipolar constraint (35), gives:

$$\begin{aligned}\left(\boldsymbol{K}_2^{-1}\overrightarrow{p}_2\right)^T\left(\boldsymbol{R}\left[\overrightarrow{T}\right]_\times\right)\boldsymbol{K}_1^{-1}\overrightarrow{p}_1 &= 0\\ \overrightarrow{p}_2^T\left[\left(\boldsymbol{K}_2^{-1}\right)^T\boldsymbol{R}\left[\overrightarrow{T}\right]_\times\boldsymbol{K}_1^{-1}\right]\overrightarrow{p}_1 &= 0\\ \overrightarrow{p}_2^T\boldsymbol{F}\overrightarrow{p}_1 &= 0\end{aligned} \tag{40}$$

The matrix $\boldsymbol{F} = \left(\boldsymbol{K}_2^{-1}\right)^T\boldsymbol{R}\left[\overrightarrow{T}\right]_\times\boldsymbol{K}_1^{-1} = \left(\boldsymbol{K}_2^{-1}\right)^T\boldsymbol{E}\boldsymbol{K}_1^{-1}$ is the ***fundamental matrix***, which encodes the information about the camera intrinsic matrices $\boldsymbol{K_1}$, $\boldsymbol{K_2}$ and the realtive translation $\overrightarrow{T}$ and rotation $\boldsymbol{M}$ between the ***camera poses***. The fundamental matrix is also a 3×3 singular matrix with ***7 degrees of freedom*** and has ***rank 2***. [17]Knowing the fundamental matrix we obtain an easy constraint of the corresponding point in the other image as given by (40)**.** Now it is possible to estimate the fundamental matrix given two images of the same scene and without knowing the intrinsic and extrinsic camera parameters. The *Eight-Point algorithm* is one such method of obtaining the fundamental matrix given a ***minimum of 8 point correspondences***. Each correspondences $\overrightarrow{p}_{1i} = \begin{bmatrix}u_{1i} & v_{1i} & 1\end{bmatrix}^T$ and $\overrightarrow{p}_{2i} = \begin{bmatrix}u_{2i} & v_{2i} & 1\end{bmatrix}^T$ gives us the epipolar constraint,

$$\overrightarrow{p}_{2i}^T F\overrightarrow{p}_{1i} = 0 \tag{41}$$

$$\begin{bmatrix}u_{2i} & v_{2i} & 1\end{bmatrix}\begin{bmatrix}f_{11} & f_{12} & f_{13}\\ f_{21} & f_{22} & f_{23}\\ f_{31} & f_{32} & f_{33}\end{bmatrix}\begin{bmatrix}u_{1i}\\ v_{1i}\\ 1\end{bmatrix} = 0$$

$$u_{1i}u_{2i}f_{11} + v_{1i}u_{2i}f_{12} + u_{2i}f_{13} + u_{1i}v_{2i}f_{21} + v_iv_{2i}f_{22} + v_{2i}f_{23} + u_{1i}f_{31} + v_{1i}f_{32} + f_{33} = 0 \tag{42}$$

The above equation can be written as the matrix product of a 1×9 matrix $\boldsymbol{A_i}$ and a 9×1 matrix $\overrightarrow{X}$ as shown,

$$\begin{bmatrix}u_{1i}u_{2i} & v_{1i}u_{2i} & u_{2i} & u_{1i}v_{2i} & v_{1i}v_{2i} & v_{2i} & u_{1i} & v_{1i} & 1\end{bmatrix}\begin{bmatrix}f_{11}\\ f_{12}\\ f_{13}\\ f_{21}\\ f_{22}\\ f_{23}\\ f_{31}\\ f_{32}\\ f_{33}\end{bmatrix} = 0$$

$$\boldsymbol{A_i}\overrightarrow{X} = 0 \tag{43}$$

Since this constraint is a scalar equation, it only constraints ***one degree of freedom***. Since, we can only know the fundamental matrix up to a scale we require a minimum of eight of these point correspondences to determine the fundamental matrix. Thus we can construct a matrix $\boldsymbol{A}$ with $\boldsymbol{A_i}$ as its rows for each of the point correspondences.

$$\boldsymbol{A}\overrightarrow{X} = 0 \tag{44}$$

$\boldsymbol{A}$ is an $N\times9$ matrix derived from $N\geq8$ point correspondences and $\overrightarrow{X}$ contains the coefficients of the fundamental matrix **defined up to a scale factor**[16]. Thus we can restrict the solution for $\overrightarrow{X}$ to have norm 1. We usually have more than 8 points but these are perturbed by noise so we look for a least square solution[17]:

$$\underset{\overrightarrow{X}}{minimize}\ \left\|\boldsymbol{A}\overrightarrow{X}\right\|^2 \tag{45}$$

As $\left\|\boldsymbol{A}\overrightarrow{X}\right\|^2 = \overrightarrow{X}^T\boldsymbol{A}^T\boldsymbol{A}\overrightarrow{X}$, this amounts to finding the ***eigenvector associated with the smallest eigenvalue*** of the 9×9 symmetric, positive semi-definite normal matrix $\boldsymbol{A}^T\boldsymbol{A}$. *Eigenvalue decomposition* will give us an estimate of the fundamental matrix $\hat{\boldsymbol{F}}$ which may have full rank. But we know that the true fundamental matrix has rank 2: as such we can impose this constraint to find the best possible value of $\hat{\boldsymbol{F}}$. This formulation does not enforce the rank constraint, so a second step is used to project the solution $\boldsymbol{F}$ onto the rank 2 subspace. This can be done by taking the *Singular Value Decomposition* of $\hat{\boldsymbol{F}}$and setting the smallest singular value to zero[2]. Let the SVD decomposition be described by:

$$\boldsymbol{F} = \boldsymbol{U}\sum\boldsymbol{V}^T \tag{46}$$

where $\sum$ is diagonal, and $\boldsymbol{U}$ and $\boldsymbol{V}$ are orthogonal. Setting the smallest diagonal element of $\sum$ to 0 and reconstituting gives the desired result. Now, for practical consideration this method can often get ***quite unstable***[2]. For example a typical image coordinate in a 512×512 image might be ~ 200. So, some of the entries in a typical row of $\boldsymbol{A}$ are $u_m u_n \sim (200)^2$, others are $u_m \sim 200$ and the last entry is 1, so there is a variation in size of $\sim (200)^2$ among the entries of A, and hence of $(200)^4 \sim 2 \times (10)^9$ among the entries of $\boldsymbol{A}^T\boldsymbol{A}$, making it numerically ill-conditioned. This problem is tackled by ***normalizing the pixel points*** before constructing $\boldsymbol{A}$. $\boldsymbol{A}$ is pre-conditioned by applying both a translation and a scaling. Firstly the origin of the new coordinate system is made to ***coincide with the centroid*** of the image points. And second, the ***mean square distance*** of the transformed image points from the origin should be 2 pixel. This is done by ***pre-multiplying the image points*** by transformation matrices $\boldsymbol{T}_1$ and $\boldsymbol{T}_2$ that translate by the centroid and scale by the scale factor,

$$\left(\frac{2N}{\sum_{i=1}^{N}\|u_i - u_{centroid}\|}\right)^{\frac{1}{2}} \tag{47}$$

for respective images. The normalize coordinates are then obtained by,

$$\overrightarrow{q}_1 = \boldsymbol{T}\overrightarrow{p}_1 \quad \overrightarrow{q}_2 = \boldsymbol{T}\overrightarrow{p}_2 \tag{48}$$

This provides a well balanced matrix $\boldsymbol{A}$ and much more stable and accurate results for $\boldsymbol{F}$. However the calculated fundamental matrix is for the normalized coordinates and must be de-normalized before it can be used for the next step[17].

$$\boldsymbol{F} = \boldsymbol{T}_2^T\boldsymbol{F}_q\boldsymbol{T}_1 \tag{49}$$

4.4 3D reconstruction using Linear Triangulation

[16]Using the fundamental matrix and the camera intrinsic matrices the location of a 3D point is now determined using ***linear triangulation*** given the projection of the points into 2 images.The relative orientation of the cameras 1 and 2 is obtained from the fundamental matrix. This is done by first calculating the essential matrix from the fundamental matrix and the intrinsic camera matrices obtained by camera calibration. Thus the essential matrix is given by(41),

$$\boldsymbol{E} = \boldsymbol{K}_2^T\boldsymbol{F}\boldsymbol{K}_1 \tag{50}$$

With the essential matrix $\boldsymbol{E} = \boldsymbol{R}\left[\overrightarrow{T}\right]_\times$ contains the information for the relative rotation and translation between the cameras. We can extract the rotation and translation by taking the *SVD* of $\boldsymbol{E}$:$\boldsymbol{E} = \boldsymbol{U}\sum\boldsymbol{V}^T$and form the following combinations[2, 16]:

- $\overrightarrow{T}$ is either $\boldsymbol{U}_3$ or $-\boldsymbol{U}_3$, where $\boldsymbol{U}_3$ is the last column of $\boldsymbol{U}$
- $\boldsymbol{R}$ is either $\boldsymbol{U}\boldsymbol{W}\boldsymbol{V}^T$ or $\boldsymbol{U}\boldsymbol{W}^T\boldsymbol{V}^T$, where $\boldsymbol{W} = \begin{bmatrix}0 & -1 & 0\\ 1 & 0 & 0\\ 0 & 0 & 1\end{bmatrix}$

Thus there are 4 possible solutions of $\boldsymbol{R}\left[\overrightarrow{T}\right]_\times$ obtained by using the different combinations of $\boldsymbol{R}$ and $\overrightarrow{T}$. But 3 of them are ***nonsensical***, meaning that they represent situations where the scene is behind one or more of the cameras. Only one of the four solutions correspond to the case where the ***scene points are in front of other cameras***. To find the correct one, we need to reconstruct a ***test scene point*** and see if it is in front of both the cameras. Since the results are only up to a scale factor, the translation $\overrightarrow{T}$ will have an arbitrary magnitude, i.e. we can only know the direction of $\overrightarrow{T}$, although the rotation $\boldsymbol{R}$ is correct. Thus the 3D points are also scaled by the same amount the translation $\overrightarrow{T}$ is scaled.
Given a hypothesized pose between the cameras, we want to reconstruct the 3D position of a point from its 2 image projections. The projection of world coordinate $\overrightarrow{P}$, onto the camera C_{m1} and camera C_{m2} are[16]:

$$\lambda\overrightarrow{p}_1 = \boldsymbol{M}_1\overrightarrow{P} \quad \lambda\overrightarrow{p}_2 = \boldsymbol{M}_2\overrightarrow{P} \tag{51}$$

where λ is the unknown scaling factor. Considering the coordinate of C_{m1} to be the world coordinate, we can write $\boldsymbol{M_1} = \boldsymbol{K_1}\begin{bmatrix}\boldsymbol{I} & \overrightarrow{O}\end{bmatrix}$ and $\boldsymbol{M_2} = \boldsymbol{K_2}\begin{bmatrix}\boldsymbol{R} & \overrightarrow{T}\end{bmatrix}$. Evidently, $\overrightarrow{p}_1$ and $\boldsymbol{M}_1\overrightarrow{P}$ must be parallel, so their c***ross product must be 0***. Similarly the cross product of $\overrightarrow{p}_1$ and $\boldsymbol{M}_1\overrightarrow{P}$ must be 0, and $\overrightarrow{P}$ should satisfy both,

$$\overrightarrow{p}_1 \times \boldsymbol{M}_1\overrightarrow{P} = 0 \tag{52}$$

$$\overrightarrow{p}_2 \times \boldsymbol{M}_2\overrightarrow{P} = 0 \tag{53}$$

We can explicitly use the equalities generated by the cross product to form the ***6 constraints***:

$$\begin{aligned}
u_1\left(\boldsymbol{M}_{13}\overrightarrow{P}\right) - \left(\boldsymbol{M}_{11}\overrightarrow{P}\right) &= 0\\
v_1\left(\boldsymbol{M}_{13}\overrightarrow{P}\right) - \left(\boldsymbol{M}_{12}\overrightarrow{P}\right) &= 0\\
u_1\left(\boldsymbol{M}_{12}\overrightarrow{P}\right) - v_1\left(\boldsymbol{M}_{13}\overrightarrow{P}\right) &= 0
\end{aligned}$$

$$\begin{aligned}
u_2\left(\boldsymbol{M}_{23}\overrightarrow{P}\right) - \left(\boldsymbol{M}_{21}\overrightarrow{P}\right) &= 0\\
v_2\left(\boldsymbol{M}_{23}\overrightarrow{P}\right) - \left(\boldsymbol{M}_{22}\overrightarrow{P}\right) &= 0\\
u_2\left(\boldsymbol{M}_{22}\overrightarrow{P}\right) - v_2\left(\boldsymbol{M}_{23}\overrightarrow{P}\right) &= 0
\end{aligned}$$

where $\boldsymbol{M}_{1i}$ is the i-th row of the matrix $\boldsymbol{M}_1$ and $\boldsymbol{M}_{2j}$ is the j-th row of the matrix $\boldsymbol{M}_2$. Using the constraints from both images, a linear equation is constructed which is of the form:

$$\boldsymbol{A}\overrightarrow{P} = \overrightarrow{O} \tag{54}$$

$$where \quad \boldsymbol{A} = \begin{bmatrix} u_1\boldsymbol{M}_{13} - \boldsymbol{M}_{11} \\ v_1\boldsymbol{M}_{13} - \boldsymbol{M}_{12} \\ u_2\boldsymbol{M}_{23} - \boldsymbol{M}_{21} \\ v_2\boldsymbol{M}_{23} - \boldsymbol{M}_{22} \end{bmatrix}$$

This equation can again be solved by ***eigenvalue decomposition as a least square problem*** to obtain the eigenvector corresponding to the ***smallest eigenvalue***. This eigenvector will give the required solution for the triangulated 3D point $\overrightarrow{P}$.

4.5 Bundle Adjustment

[15]Now, for practical purposes $\overrightarrow{p}_1$and $\overrightarrow{p}_2$ are noisy and the camera calibration parameters are not precise, as such the $\overrightarrow{P}$ obtained by linear triangulation is quite unreliable. The error in the computed $\overrightarrow{P}$

can be reduced by framing a minimization problem. We seek to find a $\hat{P}$ in 3D that best approximates $\overrightarrow{P}$ by fining the best least-squares estimate of the *re-projection error* of $\hat{P}$ in both images.The re-projection error for a 3D point in a image is the distance between the projection of that point in the image and the corresponding observed point in the image plane. Thus the minimization problem can be framed as,

$$\underset{\hat{P},\boldsymbol{R},\overrightarrow{T}}{minimize} \quad \left\|\boldsymbol{M}_1\hat{P} - \overrightarrow{p}_1\right\|^2 + \left\|\boldsymbol{M}_2\hat{P} - \overrightarrow{p}_2\right\|^2 \tag{55}$$

This is a non-linear least square problem because the projection into the image plane often involves a division by the homogeneous coordinate. A standard non-linear least square method is used to solve for the minimization problem, that finally gives the best values for $\hat{P}$, $\boldsymbol{R}$ and $\overrightarrow{T}$.

4.6 Numerical implementation

[16, 2]Consider a 3D (numerical implementation under **section number**) scene captured by two cameras C_{m1} and C_{m2}. The cameras are calibrated, and the point correspondences for the two scenes are known. The camera scenes are shown in the figure below: The camera matrices for C_{m1} and C_{m2} as given in

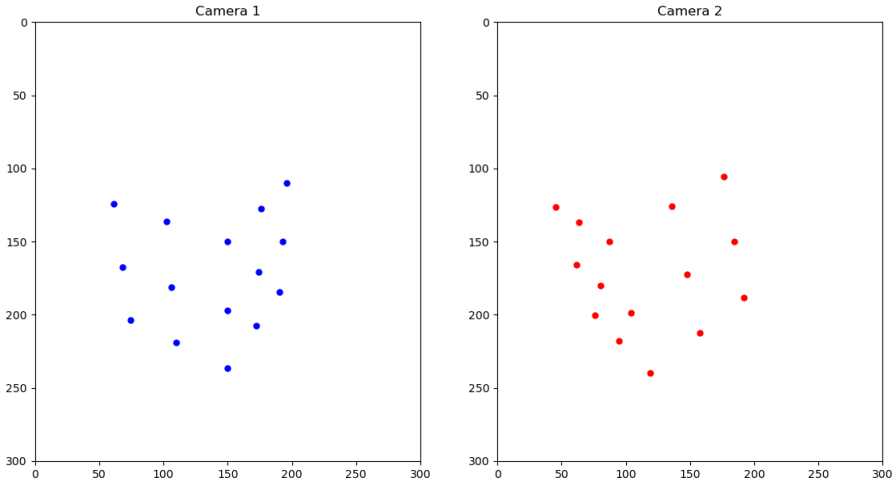


Figure 6: Model camera views

section number are as follows:

Camera	$\boldsymbol{C_{m1}}$	$\boldsymbol{C_{m2}}$
Camera Matrices	$\begin{pmatrix} 300 & 0 & 150 \\ 0 & 300 & 150 \\ 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 300 & 0 & 150 \\ 0 & 300 & 150 \\ 0 & 0 & 0 \end{pmatrix}$

A sequence of steps are employed to compute the fundamental matrix using eight point algorithm, use the fundamental matrix to obtain the essential matrix and then reconstruct the scene in 3D using linear triangulation. Finally the results of linear triangulation are refined using bundle adjustment. Initially a pre-scaling is required to precondition the point correspondences that that translate by the centroid and scale by the scale factor,

$$\left(\frac{2N}{\sum_{i=1}^{N}\|u_i - u_{centroid}\|}\right)^{\frac{1}{2}}$$

The translation by the centroid for the data points are calculated using the snippet,

```
#computing the cetroid for the data points
centroid_x=(1.0/n)*np.sum(x_vect)
centroid_y=(1.0/n)*np.sum(y_vect)

#translating the points by the centroid
x_vect_centroid=x_vect-centroid_x
y_vect_centroid=y_vect-centroid_y
```

The scale factor is computed according to the relation (47) and the points are preconditioned by applying the required transformation to the image points,

```
#computing the average distance of the data points from the data centroid
centroid_distance=np.sqrt(x_vect_centroid**2+y_vect_centroid**2)
average_distance=(1.0/n)*np.sum(centroid_distance)

#calculating the required scale factor for the transformation
scale_factor=np.sqrt(2.0)/average_distance

#computing the preconditioning matrix for the transformation
T_scale=np.matrix([
[scale_factor, 0, -scale_factor*centroid_x],
[0, scale_factor, -scale_factor*centroid_y],
[0,0,1]
])

#computing the prescaled coordinates
pts_scaled=T_scale.dot(pts.T)
```

The list of scaled points for the data is as shown below:

$\boldsymbol{camera\ C_{m1}}$		$\boldsymbol{camera\ C_{m2}}$	
$unscaled\_x$	$unscaled\_y$	$unscaled\_x$	$unscaled\_y$
-1.962	-1.192	-1.734	-1.117
-0.921	-0.891	-1.286	-0.865
0.301	-0.539	-0.700	-0.533
-1.784	-0.098	-1.331	-0.134
-0.818	0.257	-0.867	0.226
0.301	0.668	-0.270	0.680
-1.632	0.836	-0.978	0.728
-0.731	1.227	-0.505	1.160
0.301	1.673	0.092	1.704
0.967	-1.116	0.513	-1.138
1.480	-1.560	1.532	-1.646
0.914	-0.008	0.814	0.024
1.395	-0.539	1.736	-0.533
0.868	0.936	1.071	1.019
1.322	0.346	1.913	0.430

The preconditioned data points are then used to compute the fundamental matrix using the eight point algorithm. This is done by first computing the matrix $\boldsymbol{A}$ (44) for which the homogeneous equation $\boldsymbol{A}\overrightarrow{X} = 0$ is to be solved, where $\overrightarrow{X}$ contains the coefficients of the fundamental matrix.

```
A_f = np.zeros((pts1_scaled.shape[0], 9))

#constructing the matrix for eigen value decomposition
for i in range(pts1_scaled.shape[0]):
    A_f[i, :] = [ pts2_scaled[i,0]*pts1_scaled[i,0],
                  pts2_scaled[i,0]*pts1_scaled[i,1],
                  pts2_scaled[i,0],
                  pts2_scaled[i,1]*pts1_scaled[i,0],
                  pts2_scaled[i,1]*pts1_scaled[i,1],
                  pts2_scaled[i,1],
                  pts1_scaled[i,0],
                  pts1_scaled[i,1], 1  ]
```

The homogeneous equation is solved as a eigenvalue problem in the least squares sense. Eigenvalue decomposition of the matrix is $\boldsymbol{A}^T\boldsymbol{A}$ is carried out and the eigenvector corresponding to the smallest eigenvalue is computed. This eigenvector is then reshaped into the required form of the 3×3 fundamental matrix.

```
#eigenvalue decomposition to obtain the least square solution
e_vals, e_vecs = np.linalg.eig(np.dot(A_f.T, A_f))

#obtaining the eigenvector corresponding to the smallest eigen value
F_stacked=e_vecs[:, np.argmin(e_vals)]

#reshaping the eigenvector in the form of fundamental matrix
F_scaled = F_stacked.reshape(3,3)
```

A transformation is carried out on the fundamental matrix to revert the effect of preconditioning.

```
#reverting the effect of preconditioning on the fundamental matrix
unscaled_F = (T_pts1_scaled.T).dot(F_scaled).dot(T_pts2_scaled)
```

Finally the rank condition is enforced on the fundamental matrix to obtain the value of the fundamental matrix.

```
#enforcing the rank condition on the fundamental matrix
U,D,V_T=np.linalg.svd(unscaled_F)
unscaled_F=U*np.diag([D[0],D[1],0])*(V_T)
```

The essential matrix is then obtained from the fundamental matrix using the relation, $E = \boldsymbol{K}_2^T\boldsymbol{F}\boldsymbol{K}_1$. The computed fundamental and essential matrices for the system are:

fundamental matrix	***essential matrix***
$\begin{pmatrix} -1.72e-19 & +7.01e-06 & -1.05e-03 \\ +1.94e-05 & -1.30e-18 & -2.03e-02 \\ -2.91e-03 & +1.66e-02 & +5.58e-01 \end{pmatrix}$	$\begin{pmatrix} -1.55e-14 & +6.31e-01 & +4.67e-04 \\ +1.74e+00 & -1.17e-13 & -5.22e-00 \\ -1.26e-03 & +5.31e+00 & +7.71e-03 \end{pmatrix}$

The essential matrix contains 4 combinations of rotation and translation for the relative camera pose. We can extract the rotation and translation by taking the SVD of $\boldsymbol{E}$:$\boldsymbol{E} = \boldsymbol{U}\sum\boldsymbol{V}^T$and form the following combinations:

- $\overrightarrow{T}$ is either $\boldsymbol{U}_3$ or $-\boldsymbol{U}_3$, where $\boldsymbol{U}_3$ is the last column of $\boldsymbol{U}$
- $\boldsymbol{R}$is either $\boldsymbol{U}\boldsymbol{W}\boldsymbol{V}^T$ or $\boldsymbol{U}\boldsymbol{W}^T\boldsymbol{V}^T$, where $\boldsymbol{W} = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$

```
#enforcing the fact that the rotation matrices preserve orientation
if np.linalg.det(U.dot(W).dot(V))<0:
    W = -W

#obtaining the list of all possible rotations and translations
M2s = np.zeros([3,4,4])
M2s[:,:,0] = np.concatenate([U.dot(W).dot(V),\
                        U[:,2].reshape([-1, 1])/abs(U[:,2]).max()], axis=1)
M2s[:,:,1] = np.concatenate([U.dot(W).dot(V),\
                        -U[:,2].reshape([-1, 1])/abs(U[:,2]).max()], axis=1)
M2s[:,:,2] = np.concatenate([U.dot(W.T).dot(V),\
                        U[:,2].reshape([-1, 1])/abs(U[:,2]).max()], axis=1)
M2s[:,:,3] = np.concatenate([U.dot(W.T).dot(V),\
                        -U[:,2].reshape([-1, 1])/abs(U[:,2]).max()], axis=1)
```

One of the four combinations is the true value of the relative pose between the cameras C_{m1} and C_{m2}. The triangulation is carried out for all the values of the point correspondences using the four combinations of the relative pose. Linear triangulation is equivalent to solving the homogeneous matrix equation:

$$\boldsymbol{C}\overrightarrow{P} = \overrightarrow{O} \; where,$$

$$\boldsymbol{C} = \begin{bmatrix} u_1\boldsymbol{M}_{13} - \boldsymbol{M}_{11} \\ v_1\boldsymbol{M}_{13} - \boldsymbol{M}_{12} \\ u_2\boldsymbol{M}_{23} - \boldsymbol{M}_{21} \\ v_2\boldsymbol{M}_{23} - \boldsymbol{M}_{22} \end{bmatrix}$$

This is done for all the point correspondences. The matrix $\boldsymbol{C}$ for each of the point correspondences is constructed as shown:

```
#obtaining the matrix A for the homogeneus equation AX=0
A = np.array([pts1[i,0]*C1[2,:] - C1[0,:] ,
              pts1[i,1]*C1[2,:] - C1[1,:] ,
              pts2[i,0]*C2[2,:] - C2[0,:] ,
              pts2[i,1]*C2[2,:] - C2[1,:]])
```

The homogeneous equation is solved as an eigenvalue problem in the least squares sense. This is done by using EVD for the matrix $\boldsymbol{C}$ and then using the eigenvector corresponding to the smallest eigenvalue as the solution to the system of equations.

```
#solving the homogeneous equation using EVD in least squares sense
e_vals, e_vecs = np.linalg.eig(np.dot(A.T, A))
X = e_vecs[:, np.argmin(e_vals)]

#obtaining the correct value of the euclidean coordinates
X = X/X[-1]
```

The resulting array of 3D points obtained by linear triangulation must have all the z values greater than zero, since the reconstructed scene must lie in front of the camera C_{m1}. The true value of the camera relative pose and the reconstructed 3D points is obtained by applying this condition. The other three combinations of the camera pose are simply meaningless.

```
P, err = triangulate(C1, pts1, C2, pts2, K1, K2, M1, M2)
#obtaining the z values of the triangulated points
z_vect = P[:, 2]
#testing if all the triangulated 3D points are in front of the cameras
if all(z>0 for z in z_vect):
    #storing the valid 3D points, relative orientation
        #and the second camera projection matrix
    err_true = err
    P_true = P.copy()
    M2_true = M2.copy()
    C2_true = C2.copy()
    break
```

The 3D points obtained by linear triangulation is then refined using bundle adjustment to reduce the re-projection error. This is framed as a minimization problem, that minimizes the re-projection error as a function of the triangulated 3D points and the relative camera pose matrix.

$$\underset{\hat{P}, \boldsymbol{R}, \overrightarrow{T}}{minimize} \quad \left\| \boldsymbol{M}_1 \hat{P} - \overrightarrow{p}_1 \right\|^2 + \left\| \boldsymbol{M}_2 \hat{P} - \overrightarrow{p}_2 \right\|^2$$

The residual is calculated by re-projecting the 3D points onto the camera planes using the camera projection matrices and then subtracting from the image coordinates of the point correspondences as captured by C_{m1} and C_{m2} to obtain the euclidean distance between the $\left(\boldsymbol{M}\hat{P} - \overrightarrow{p}\right)$ as the re-projection error. The script given below re-projects the 3D points onto the camera planes in homogeneous coordinates.

```
#projecting 3D points onto the image planes using the projection matrices
p1_hat_homo = np.matmul( C1, P_homo )
p2_hat_homo = np.matmul( C2, P_homo )
```

The residual is then obtained by computing the euclidean distance between the captured image points and the re-projected points from the camera projection matrices.

```
#computing the array of residuals for the reprojected points
residuals = np.concatenate([(p1-p1_hat).reshape([-1]),\
                        (p2-p2_hat).reshape([-1]), np.zeros(5)])
```

The re-projection error is minimized using the function *nlsq_ residual.levenberg()* in the non-linear least square sense. The function returns the best 3D points and the best relative pose matrix obtained by the minimization procedure.

```
#projecting 3D points onto the image planes using the projection matrices
#arranging the parameters into a 1D array
x_init  = np.hstack( [ P_init[:,0].reshape([-1]), P_init[:,1].reshape([-1]),\
                  P_init[:,2].reshape([-1]), r_init, t_init ]  )

#minimizing the reprojection error using non linear least squares
x_new = levenberg(residual, x_init, args=(K1, M1, p1, K2, p2))
```

The chi-squared value before the fit is obtained to be $3.79 \times 10^{-1}\text{pixel}^2$ and comes out to be $1.89 \times 10^{-6}\text{pixel}^2$ after the fit. The refined values of the 3D point obtained after bundle adjustment is visualized in a 3D plot as show below.

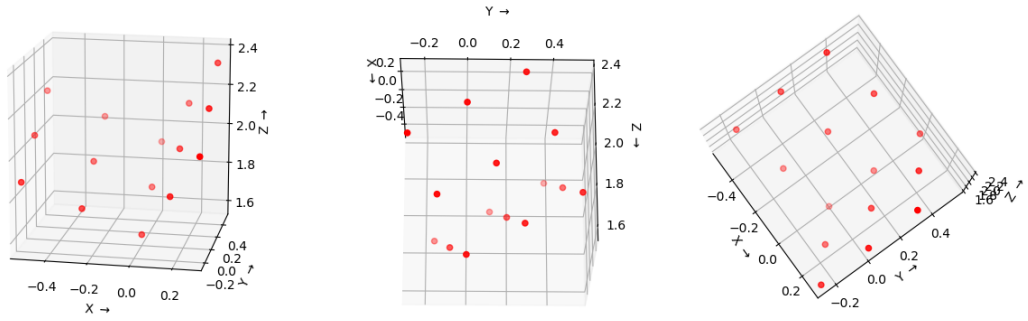


Figure 7: 3D reconstruction

5 Object Tracking

[18] *Object tracking* refers to the sequence of steps employed to locate a moving object over time, constrained within the image frame of a camera. 3D reconstruction using stereo geometry requires us to have a knowledge of ***synchronized point correspondences*** for the two camera views. This is achieved using a object tracking algorithm that uses a ***color mask technique*** to distinguish the object of interest from the surroundings. The following subsection describes the process in detail.

5.1 Algorithm

The general sequence of steps employed to track the object of interest[18] is given below:

- The object to be tracked is given a unique ***HSV*** signature. The lower and upper bounds of the object in the HSV color space is defined.

```
colLower=(95,50,50)
colUpper=(130,255,255)
```

- The video frames are grabbed iteratively in ***BGR*** color space along with their time stamps.

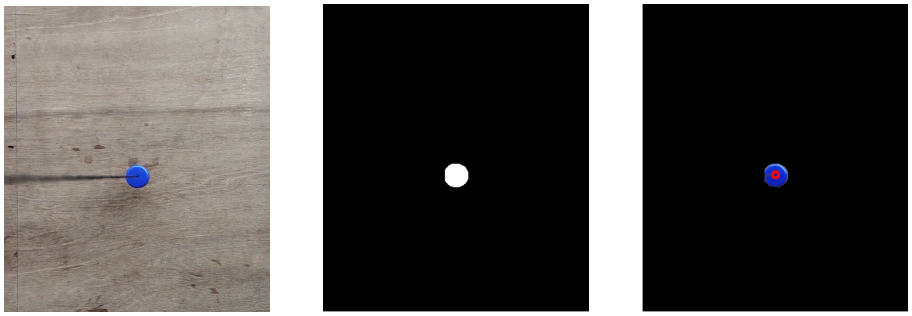


Figure 8: Object tracking using mask creation

- A pre-processing schedule is carried out to remove noise and other image distortions.

```
#Frame preprocessing to remove noise
blurred=cv.GaussianBlur(frame,(7,7),0)
```

- The image is then transformed from the BGR spce to the ***HSV*** space.

```
#Conversion to HSV color space
hsv=cv.cvtColor(blurred,cv.COLOR_BGR2HSV)
```

- A mask is constructed for the specific HSV range as required for the object to be tracked.

```
#mask of the specific color is formed
mask=cv.inRange(hsv,colLower,colUpper)
```

- ***Erosion*** and ***dilutions*** are carried out to remove noisy mask blobs.
- A ***contour*** is created for the largest tracked white blob in the mask.
- The object is located in the image frame by finding out the ***centroid*** of this contour.

```
#the largest contour  in the mask is found out
contours=max(contours,key=cv.contourArea)

M=cv.moments(contours)
#computing the centroid of the contours
centroid=(int(M["m10"]/M["m00"]),int(M["m01"]/M["m00"]))
```

- The positon and time data so obtained are stored in a 2D matrix for subsequent computations.

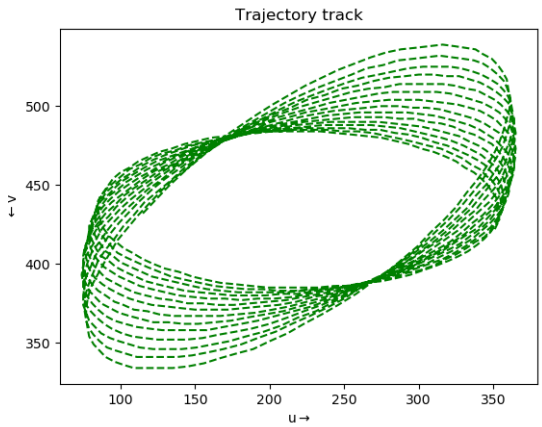


Figure 9: Track trajectory

6 Video Synchronization

6.1 Posing the problem

A pair of un-synchronized cameras capturing a dynamic scene cannot produce synchronized point correspondences for 3D reconstruction. This is because the correspondence is both a function of time as well as the space variables. An analytic treatment is developed to synchronize the camera pair and obtain the point correspondences.
Consider two static un-synchronized cameras C_{m1} and C_{m2} capturing a dynamic scene at a fixed frame rate placed at a fixed relative pose. The position of a 3D point moving in a trajectory in space can be described by:

$$\overrightarrow{Q}(t) = \begin{bmatrix} Q_1(t) \\ Q_2(t) \\ Q_3(t) \\ 1 \end{bmatrix} \tag{56}$$

where t denotes time. Projecting $\overrightarrow{Q}(t)$ onto the camera image planes to produce two 2D trajectories $\overrightarrow{q}_1(t)$ and $\overrightarrow{q}_2(t)$. Let the first camera take frames at an interval of f_{c1} seconds and the second camera take frames at intervals of f_{c2} seconds. This leads to a sequence of samples

$$\overrightarrow{p}_1(m) = \begin{bmatrix} u_{1m} \\ v_{1m} \\ 1 \end{bmatrix} \quad \overrightarrow{p}_2(n) = \begin{bmatrix} u_{2n} \\ v_{2n} \\ 1 \end{bmatrix} \tag{57}$$

- of $\overrightarrow{q}_1(t_{1m})$ at times $t_{1m}(m) = t_{10} + mf_{c1}$ from camera 1, where $m = \{0, 1, 2, 3, ..., M\}$ and t_{10} is the time at which the 0^{th} frame is captured
- of $\overrightarrow{q}_2(t_{2n})$ at times $t_{2n}(n) = t_{20} + nf_{c2}$ from camera 2, where $n = \{0, 1, 2, 3, ..., N\}$ and $t_{20}$is the time at which the 0^{th} frame is captured

Rearranging $t_{2n} = t_{20} + nf_{c2}$, we have,

$$n = \frac{t_{2n} - t_{20}}{f_{c2}} \tag{58}$$

Since for synchronization the times at which the m_{th} and the n_{th} frames are captured must be the same, we can have $t_{1m} = t_{2n}$[19]. Substituting this into (58) we obtain,

$$\begin{aligned} n(m) &= \frac{t_{1m} - t_{20}}{f_{c2}} \\ n(m) &= \frac{t_{10} + mf_c - t_{20}}{f_{c2}} \\ n(m) &= \left(\frac{t_{10} - t_{20}}{f_{c2}}\right) + m\frac{f_{c1}}{f_{c2}} \end{aligned} \tag{59}$$

Writing $\alpha = \frac{t_{10} - t_{20}}{f_{c2}}$ and $\beta = \frac{f_{c1}}{f_{c2}}$, we can rewrite the equation as : $n(m) = \alpha + m\beta$. This equation is linear in m. Note that n in general is not integer for integral values of m. The constant α represents the ***systematic shift*** in time between the camera frames whereas the constant β represents a ***time scaling between the camera frame*** sequences[20, 21]. In order to obtain $\overrightarrow{p}_2(\alpha + m\beta)$ an ***cubic interpolation function*** is obtained for the samples $\overrightarrow{p}_2(n)$. Thus the point correspondences are now given by,

$$\overrightarrow{p}_1(m) \longleftrightarrow \overrightarrow{p}_2(\alpha + m\beta)$$

6.2 Linearizing $\overrightarrow{p}$ around optimal α

[21]In order to obtain the point correspondences, we need to estimate the value of the constants α and β. Now the frame rates of the cameras are in general provided by the manufacturers, and hence $\beta = \frac{f_{c1}}{f_{c2}}$

is known. α can be estimated by linearizing $\overrightarrow{p}_2(\alpha + m\beta)$ towards the true value of α. Expanding $\overrightarrow{p}_2(\alpha + m\beta)$ using Taylor Series approximation, around $n = \alpha_0 + m\beta$,

$$\overrightarrow{p}_2(\alpha + m\beta) = \overrightarrow{p}_2(\alpha + m\beta)|_{\alpha=\alpha_0} + \frac{d}{dn}[\overrightarrow{p}_2(\alpha + m\beta)]\Big|_{\alpha=\alpha_0} [\overrightarrow{p}_2(\alpha + m\beta) - \overrightarrow{p}_2(\alpha_0 + m\beta)] + \dots$$

$$\overrightarrow{p}_2(\alpha + m\beta) \approx \overrightarrow{p}_2(\alpha_0 + m\beta) + \frac{d}{dn}[\overrightarrow{p}_2(\alpha_0 + m\beta)](\alpha - \alpha_0) \tag{60}$$

The derivative $\frac{d}{dn}[\overrightarrow{p}_2(\alpha_0 + m\beta)]$ can be expressed as the finite difference since n must vary by integral values only. Thus we can write,

$$\frac{d\overrightarrow{p}_2}{dn}\Big|_{\alpha=\alpha_0} \equiv \frac{\triangle\overrightarrow{p}_2}{\triangle n}\Big|_{\alpha=\alpha_0} = \overrightarrow{p}_2(\alpha_0 + m\beta + 1) - \overrightarrow{p}_2(\alpha_0 + m\beta) \tag{61}$$

Writing $\triangle\overrightarrow{p}_2 = \overrightarrow{w}(m) \equiv \overrightarrow{w}(\alpha_0 + m\beta)$ and substituting $\frac{d\overrightarrow{p}_2}{dn}\Big|_{\alpha=\alpha_0}$ this into (60) for $\overrightarrow{p}_2(\alpha + m\beta)$ we obtain,

$$\begin{aligned}\overrightarrow{p}_2(\alpha + m\beta) &= \overrightarrow{p}_2(\alpha_0 + m\beta) + (\alpha - \alpha_0)\overrightarrow{w}(m) \\ &= \overrightarrow{g}(m) + \alpha\overrightarrow{w}(m)\end{aligned} \tag{62}$$

where, $\overrightarrow{p}_2(\alpha_0 + m\beta) - \alpha_0\overrightarrow{w}(m)$ is rewritten as $\overrightarrow{g}(m) \equiv \overrightarrow{g}(\alpha_0 + m\beta)$.

6.3 Solving for α

Now to estimate the value of α, we make use of the fact that $\overrightarrow{p}_1(m)$ and $\overrightarrow{p}_2(\alpha + m\beta)$ are constrained by the ***epipolar constraint*** (40) i.e. any point image point $\overrightarrow{p}_2(\alpha + m\beta)$ in C_{m2} image is constrained to lie on the epipolar line corresponding to image point $\overrightarrow{p}_1(m)$ in C_1 image. Thus finding α can be viewed as a ***minimization problem***: estimating α for which the error due to the image points not lying on the ***corresponding epipolar lines is minimized***. Now, substituting the values of $\overrightarrow{p}_1(m)$ and $\overrightarrow{p}_2(\alpha + m\beta)$ into the epipolar constraint, we obtain

$$\begin{aligned}\overrightarrow{p}_2(\alpha + m\beta)^T \boldsymbol{F}\overrightarrow{p}_1(m) &= 0 \\ [\overrightarrow{g}(m) + \alpha\overrightarrow{w}(m)]^T \boldsymbol{F}\overrightarrow{p}_1(m) &= 0 \\ \overrightarrow{g}(m)^T \boldsymbol{F}\overrightarrow{p}_1(m) + \alpha\overrightarrow{w}(m)^T \boldsymbol{F}\overrightarrow{p}_1(m) &= 0\end{aligned} \tag{63}$$

- The first term $\overrightarrow{g}(m)^T \boldsymbol{F}\overrightarrow{p}_1(m)$, can be written as the product of a 1×9 matrix and a 9×1 matrix : $\boldsymbol{A}_{1m}\overrightarrow{X}$
- Similarly the second term $\alpha\overrightarrow{w}(m)^T \boldsymbol{F}\overrightarrow{p}_1(m)$ can be written as : $-\gamma\boldsymbol{A}_{2m}\overrightarrow{X}$ where $\gamma = -\alpha$

Thus we can construct matrices $\boldsymbol{A}_1$and $\boldsymbol{A}_2$ with each of their rows for each of the point correspondences. Thus the matrix equation can be written in the form:

$$\begin{aligned}\boldsymbol{A}_1\overrightarrow{X} &= \gamma\boldsymbol{A}_2\overrightarrow{X} \\ \boldsymbol{M}\overrightarrow{X} &= \gamma\overrightarrow{X}\end{aligned} \tag{64}$$

where $\boldsymbol{M} = \left(\boldsymbol{A}_2^T\boldsymbol{A}_2\right)^{-1}\left(\boldsymbol{A}_2^T\boldsymbol{A}_1\right)$. Then (64) is reduced to an eigenvalue problem, with the eigenvalues $-\alpha$ and the corresponding eigenvectors containing the coefficients for the fundamental matrix. Only the real eigenvalues for the problem correspond to a valid solution for α. Knowing α, the point correspondences can be obtained as:

$$\overrightarrow{p}_1(m) \longleftrightarrow \overrightarrow{p}_2(\alpha + m\beta)$$

In case there are more than one real values of α, the one giving minimum re-projection error as a result of linear triangulation is the best possible solution for α[19]. The process is carried out iteratively updating the value of α at each iteration till the convergence criteria is satisfied.

6.4 Numerical Implementation

Consider a synthetic data set of video frames captured by cameras C_{m1} and C_{m2}. The camera frame rates for C_{m1} and C_{m2} are both ≈ 16.67s. The true difference between the start of the video sequences of C_{m1} and C_{m2} is $(t_{20} - t_{10})$=0.555s, giving $\alpha_{true} = -9.25$. Then,

- m^{th} frame of camera C_{m1} is captured at time $t_{1m}\,(m) = t_{10} + 0.06m$
- n^{th} frame of camera C_{m2} is captured at time $t_{2n}\,(n) = t_{20} + 0.06n$

The trajectory captured by the cameras is that of a simulated spherical pendulum. The cameras are placed static with respect to the system and hence with respect to each other. The points are preconditioned prior to constructing the matrices $\boldsymbol{A}_1$ and $\boldsymbol{A}_2$ by applying a translation an scaling. This is done by pre-multiplying the image points by transformation matrices $\boldsymbol{T}_1$ and $\boldsymbol{T}_2$ that translate by the centroid and scale by the scale factor,

$$\left(\frac{2N}{\sum_{i=1}^{N}\|u_i - u_{centroid}\|}\right)^{\frac{1}{2}}$$

This provides a well balanced matrices $\boldsymbol{A}_1$ and $\boldsymbol{A}_2$ and much more stable and accurate results for α. However the calculated eigenvector is used as the fundamental matrix then it must be de-normalized before it can be used for the next step.

$$\boldsymbol{F} = \boldsymbol{T}_2^T \boldsymbol{F}_q \boldsymbol{T}_1$$

Vectors are defined for the frame indices m and n and a cubic interpolation function is created for the trajectory $\overrightarrow{q}_2\,(t)$ using the scipy method *interp1d()* so that the mapping from $m \to n$ can be an integer to real number mapping. The vector $\overrightarrow{w} = \overrightarrow{p}_2\,(\alpha_0 + m\beta + 1) - \overrightarrow{p}_2\,(\alpha_0 + m\beta)$ is calculated as the finite forward difference, since n can only vary by an integer,

```
#computing the cubic interpolation function for the trajectory q_2(t)
f_u_pts2=interp1d(n,pts2_scaled[0])
f_v_pts2=interp1d(n,pts2_scaled[1])

#computing the vector w = finite difference delta(pts2)
u_w=f_u_pts2(alpha_0+m*beta+1)-f_u_pts2(alpha_0+m*beta)
v_w=f_v_pts2(alpha_0+m*beta+1)-f_v_pts2(alpha_0+m*beta)
w=np.vstack((u_w,v_w))
```

The vector $\overrightarrow{g} = \overrightarrow{p}_2\,(\alpha_0 + m\beta) - \alpha_0\overrightarrow{w}\,(m)$ is again calculated by the same treatment. The matrices $\boldsymbol{A}_1$ and $\boldsymbol{A}_2$ for matrix equation is stacked using the vector $\overrightarrow{p}_1\,(m)$ computed vectors $\overrightarrow{w}\,(m)$ and $\overrightarrow{g}\,(m)$.

$$\boldsymbol{A}_{1i} = \begin{bmatrix} u_{p_1 i}u_{gi} & v_{p_1 i}u_{gi} & u_{gi} & u_{p_1 i}v_{gi} & v_{p_1 i}v_{gi} & v_{gi} & u_{p_1 i} & v_{p_1 i} & 1 \end{bmatrix}$$

$$\boldsymbol{A}_{2i} = \begin{bmatrix} u_{p_1 i}u_{wi} & v_{p_1 i}u_{wi} & u_{wi} & u_{p_1 i}v_{wi} & v_{p_1 i}v_{wi} & v_{wi} & u_{p_1 i} & v_{p_1 i} & 1 \end{bmatrix}$$

where the index i represents the row number for the matrices $\boldsymbol{A}_1$ and $\boldsymbol{A}_2$.The matrix $\boldsymbol{M}$ is finally decomposed to obtain its eigenvalues and the corresponding eigen vectors. The eigenvalue with the least re-projection error is used as the best value for alpha.

```
#Computing the matrix whose eigenvalues are to be determined
B=(A_2.T).dot(A_2)
C=(A_2.T).dot(A_1)
M=(np.linalg.inv(B)).dot(C)
#EVD for the matrix M
alpha_list, X=np.linalg.eig(M)
```

The convergence criteria is obtained by dividing the update in alpha value by the last alpha value. The iteration breaks whenever this ratio becomes smaller than a user defined epsilon.

```
#convergence criteria for obtaining alpha
def convergence(delta_alpha, alpha_0, eps):
        if abs(delta_alpha/alpha_0)<eps:
                return True
        return False
```

With the guess value for $(t_{20} - t_{10}) = 0.65$, the computed value of α converged after 5 iterations and the best value comes out to be -9.268.

7 Modeling the Dynamical Systems

The spherical pendulum and the magnetic pendulum are the dynamical systems, that are used for the purpose of the experiment. The following subsections go into detailed analysis for deriving the equations of motion for the systems and the expressions for the other motion parameters.

7.1 Spherical Pendulum

7.1.1 Equations of motion from Lagrangian

Figure (10) shows a spherical pendulum setup. A bob of mass m_{bob} is suspended using a rigid rod of mass m_{rod} and length L, from a rigid support considered to be the system origin. The suspension rod makes an angle θ with the positive x-axis and angle ϕ with the positive z-axis. The bob is restricted to move along the surface of sphere of radius $r = L$, that is center on the suspension point. Thus the motion of the pendulum can be completely described by the polar angle θ and the azimuthal angle ϕ and their time derivatives. We will be using the *Euler Lagrange*[23] equations to find the equations of motion for the system. The ***total kinetic energy*** and the ***total potential energy*** of the system is given by,

$$T_{total} = T_{bob} + T_{rod} \tag{65}$$

$$U_{total} = U_{bob} + U_{rod} \tag{66}$$

where T_{bob} and T_{rod} are the kinetic energies of the bob and the rod respectively. U_{bob} is the ***gravitational*** potential energy of the bob, U_{rod} the gravitational potential energy of the rod. Using the transformation equations between the Cartesian and the spherical polar coordinates

$$\begin{aligned} x &= L\sin\phi\cos\theta \\ y &= L\sin\phi\sin\theta \\ z &= L\cos\phi \\ L &= \sqrt{x^2 + y^2 + z^2} \end{aligned}$$

the time derivatives $\dot{x}$, $\dot{y}$ and $\dot{z}$ are then given by,

$$\begin{aligned} \dot{x} &= L\cos\phi\cos\theta\dot{\phi} - L\sin\phi\sin\theta\dot{\theta} \\ \dot{y} &= L\cos\phi\sin\theta\dot{\phi} + L\sin\phi\cos\theta\dot{\theta} \\ \dot{z} &= -L\sin\phi\dot{\phi} \end{aligned}$$

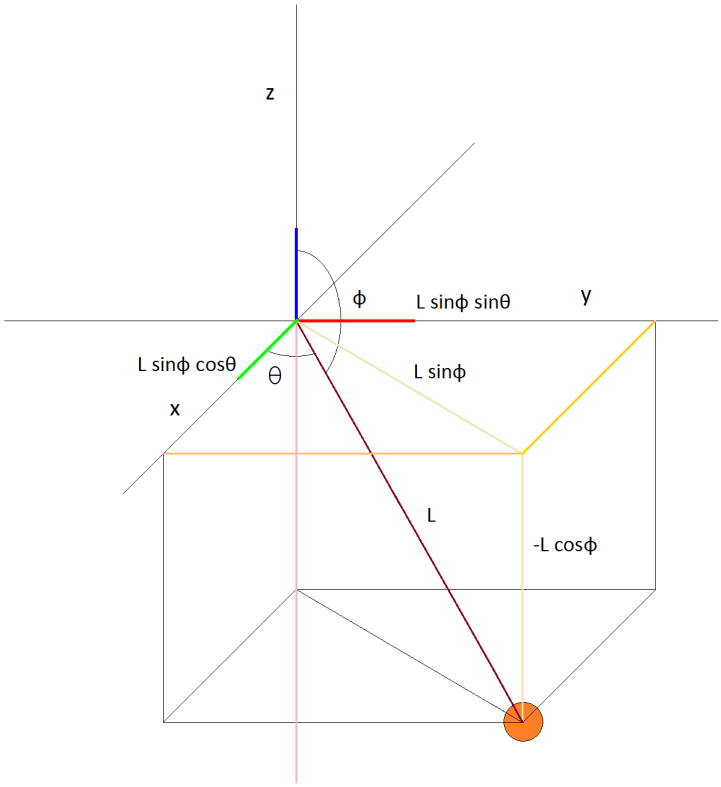


Figure 10: The magnetic pendulum system

Thus the expressions for $T_{bob}$and T_{rod} are derived as follows,

$$\begin{aligned} T_{bob} &= \frac{1}{2} m_{bob}(\dot{x}^2 + \dot{y}^2 + \dot{z}^2) \\ &= \frac{1}{2} m_{bob} L^2 \left[\left(\cos\phi\cos\theta\dot{\phi} - \sin\phi\sin\theta\dot{\theta} \right)^2 + \left(\cos\phi\sin\theta\dot{\phi} + \sin\phi\cos\theta\dot{\theta} \right)^2 + \left(-\sin\phi\dot{\phi} \right)^2 \right] \\ &= \frac{1}{2} m_{bob} L^2 \left(\dot{\phi}^2 + \sin^2\phi\dot{\theta}^2 \right) \end{aligned} \tag{67}$$

$$\begin{aligned} T_{rod} &= \frac{1}{2} \int_0^L l^2 \left(\dot{\phi}^2 + \sin^2\phi\dot{\theta}^2 \right) \frac{m_{rod}}{L} dl \\ &= \frac{1}{2} \left(\dot{\phi}^2 + \sin^2\phi\dot{\theta}^2 \right) m_{rod} \frac{L^2}{3} \\ &= \frac{m_{rod}}{6} L^2 \left(\dot{\phi}^2 + \sin^2\phi\dot{\theta}^2 \right) \end{aligned} \tag{68}$$

T_{rod} is calculated by integrating the kinetic energies for ***infinitesimal mass*** components $dm = \frac{m_{rod}}{L} dl$, over the entire length of the rod from 0 to L. Note that the kinetic energy of the rod can also be obtained by using the expression $T = \frac{1}{2} I_{rod} \omega^2$, where $I_{rod} = \frac{1}{3} m_{rod} L^2$ is the moment of inertia of the rod about an end point, and $\omega = \dot{\phi}^2 + \sin^2\phi\dot{\theta}^2$ is the angular velocity of the rod. Similarly we have the expressions

for U_{bob} and U_{rod} given by,

$$U_{bob} = m_{bob} g L \cos\phi \tag{69}$$

$$\begin{aligned} U_{rod} &= \int_0^L \frac{m_{rod}}{L} g l \cos\phi dl \\ &= m_{rod} g \frac{L}{2} \end{aligned} \tag{70}$$

The potential energies are calculated with respect to the system origin at the suspension point. U_{rod} is obtained by integrating the potential energies of the infinitesimal mass elements dm over the entire length of the rod. This is equivalent to calculating the potential energy of the rod using its center of mass, i.e. $U_{rod} = m_{rod} g \times (position\, of\, COM)$. Now we put everything together to obtain T_{total} and U_{total} and the *Lagrangian* $\mathscr{L}$ for the system can be given by,

$$\begin{aligned} \mathscr{L} &= T_{total} - U_{total} \\ &= \frac{L^2}{2}\left(m_{bob} + \frac{m_{rod}}{3}\right)\left(\dot{\phi}^2 + \sin^2\phi\dot{\theta}^2\right) - \left(m_{bob} + \frac{m_{rod}}{2}\right) g L \cos\phi \\ &= M_1 \frac{L^2}{2}\left(\dot{\phi}^2 + \sin^2\phi\dot{\theta}^2\right) - M_2 g L \cos\phi \end{aligned} \tag{71}$$

There are two variables for the system involved, ϕ and θ. So we will have one differential equation by using ϕ as the variable for the Euler Lagrange equations and the other differential equation by using θ as the variable. For the variables ϕ and θ we have the differential equation for the system given by,

$$\frac{\partial \mathscr{L}}{\partial \phi} = \frac{d}{dt}\left(\frac{\partial \mathscr{L}}{\partial \dot{\phi}}\right) \tag{72}$$

$$\frac{\partial \mathscr{L}}{\partial \theta} = \frac{d}{dt}\left(\frac{\partial \mathscr{L}}{\partial \dot{\theta}}\right) \tag{73}$$

Expanding the derivatives we obtain the ***equations of motion*** for the system, one for $\ddot{\theta}$ and the other for $\ddot{\phi}$,

$$\ddot{\phi} = \sin\phi\cos\phi\dot{\theta}^2 + \frac{g}{L}\left(\frac{M_2}{M_1}\right)\sin\phi \tag{74}$$

$$\ddot{\theta} = -2\cot\phi\dot{\theta}\dot{\phi} \tag{75}$$

7.1.2 Effective potential and other motion parameters

The equation (74), $\frac{\partial \mathscr{L}}{\partial \theta} = \frac{d}{dt}\left(\frac{\partial \mathscr{L}}{\partial \dot{\theta}}\right)$ suggests that $\frac{d}{dt}\left(M_1 L^2 \sin^2\phi\dot{\theta}\right) = 0$. In other words, the quantity $M_1 L^2 \sin^2\phi\dot{\theta}$ is constant of time. This quantity is the z-component l_z of the ***angular momentum*** vector for the pendulum and is conserved for the system[22, 24]. This is conserved because neither gravity nor reaction from the suspension rod exert any torque on the pendulum about the z axis. Moreover, the Lagrangian (71) $\mathscr{L}$ ***does not have any explicit dependence on time***, and the potential energy(66) ***does not depend explicitly on the time derivatives*** of the coordinates, the total energy $E = T_{total} + U_{total}$ remains conserved for the system. Since l_z is a constant of the motion we can replace $\dot{\theta}$ by the expression $\frac{l_z}{M_1 L^2 \sin^2\phi}$ in the expression for total energy.

$$E = \frac{1}{2} M_1 L^2 \dot{\phi}^2 + \frac{l_z^2}{2 M_1 L^2 \sin^2\phi} + M_2 g L \cos\phi \tag{76}$$

This makes E a function of only one coordinate $\dot{\phi}$, reducing the system to a one body problem, such that $E = T_{\text{eff}} + U_{\text{eff}}$ with,

$$T_{\text{eff}} = \frac{1}{2} M_1 L^2 \dot{\phi}^2 \; and \; U_{\text{eff}} = \frac{l_z^2}{2 M_1 L^2 \sin^2\phi} + M_2 g L \cos\phi \tag{77}$$

The first term of the effective potential energy is the ***centrifugal*** term that tends to ***throw the particle outwards*** due to the pendulum rotating about the z axis, whereas the second term tends to ***pull the pendulum inwards***. Plotting for the effective potential energy U_{eff} as a function of the azimuthal coordinate ϕ,

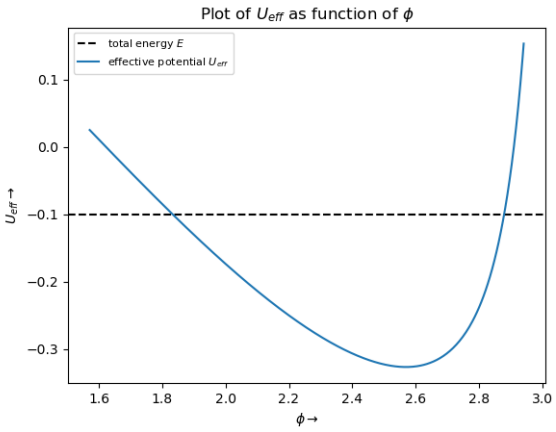


Figure 11: Plot for U_{eff} as a function of ϕ in the range $(\pi/2, \pi - 0.2)$

The plot (11) shows that U_{eff} shows a ***potential well*** for the system which is the result of the ***superposition*** of the gravitational and the centrifugal potential terms. The total energy of the system can have any value ***greater than the minimum*** value of the effective potential, since the effective kinetic energy for the system must be positive. Thus depending on the total energy of the system, there can be two cases:

- when the total energy of the system is greater than the minimum effective potential[22], i.e. $E > U_m$, the system oscillates between two values of $\phi(\phi_{min}, \phi_{max})$ (12). At any point, the effective kinetic energy of the system is given by $T_{\text{eff}} = E - U_{\text{eff}}$. This implies that ϕ becomes ϕ_{min} and ϕ_{max} when the effective kinetic energy of the system becomes 0. Thus the total energy of the system must be equal to the effective potential energy at these points.

$$E = \frac{l_z^2}{2M_1L^2\sin^2\phi} + M_2gL\cos\phi$$

$$2M_2M_1gL^3\left(\cos\phi - \cos^3\phi\right) - 2M_1EL^2\left(1 - \cos^2\phi\right) + l_z^2 = 0 \tag{78}$$

  Solving the quadratic equation for $\cos\phi$ we obtain the values for ϕ_{min} and ϕ_{max}, between which the system oscillates. It is to be noted that no matter how high the total energy of the system is, the system is always bounded between some ϕ_{min} and some ϕ_{max}. The plot below shows ϕ oscillating between ϕ_{min} and ϕ_{max} with time. It is seen that around the maxima, the derivative is a lot aggressive as compare to around the minima. This is due to the potential well having a steep rise towards ϕ_{max} and a gradual rise towards the ϕ_{min}. Also, let T_ϕ be the ***time period of oscillation in*** ϕ (84), an expression for which has been derived later. Moreover the motion of the system is always bounded between $\phi = 0$ and $\phi = \pi$. This is due to the effective potential blowing up at $\phi = 0, \pi$.

- when the total energy of the system is ***equal to the minimum effective potential***, i.e. $E = U_m$, then $\phi_{max} = \phi_{min}$ and the system remains fixed at this value of ϕ. This is the case of a conical pendulum, when the bob rotates around the z-axis with a uniform speed in a ***horizontal plane***.

Now, substituting for $\dot{\theta}$, the equations of motion (74)(75) for the system can now be given by[24],

$$\ddot{\phi} = \frac{l_z^2\cos\phi}{M_1^2L^4\sin^3\phi} + \frac{g}{L}\left(\frac{M_2}{M_1}\right)\sin\phi \tag{79}$$

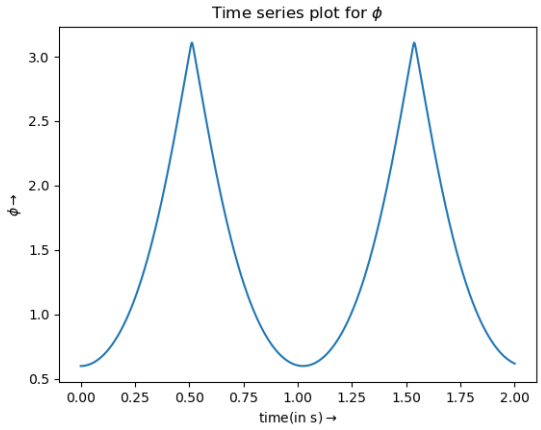


Figure 12: ϕ oscillating with time

$$\dot{\theta} = \frac{l_z}{M_1 L^2 \sin^2 \phi} \tag{80}$$

Again there can be two special cases for this equation at this point.

- ***The polar coordinate does not change with time***, i.e. $\theta = \theta_0 = (\text{constant}) \Rightarrow \dot{\theta} = 0$. Thus the z-component of the angular momentum l_z must be zero, and the equation of motion is reduced to,

  $$\ddot{\phi} = \frac{g}{L}\left(\frac{M_2}{M_1}\right)\sin\phi \tag{81}$$

  This is the case of a ***simple pendulum*** with its motion constrained to a vertical plane.

- ***The azimuthal coordinate ϕ does not change with time***, i.e. $\phi = \phi_0 = (\text{constant}) \Rightarrow \dot{\theta} = \frac{l_z}{M_1 L^2 \sin^2 \phi} = \text{constant}(80)$. This is the case of the ***conical pendulum***, with the pendulum rotating around the z-axis with ***uniform*** angular velocity $\dot{\theta} = \sqrt{\frac{g}{L\cos\phi_0}\left(\frac{M_2}{M_1}\right)}$, in a horizontal plane.

Now, using the one body energy expression (76) for the system, we can have, the time derivative of ϕ given as,

$$\dot{\phi} = \sqrt{\frac{2\left[E - U_{\text{eff}}\left(\phi\right)\right]}{M_1 L^2}} \tag{82}$$

Rearranging an integrating with respect to t and ϕ we obtain,

$$t = t_{init} + \sqrt{M_1} L \int_{\phi_{init}}^{\phi} \frac{d\phi}{\sqrt{2\left[E - U_{\text{eff}}\left(\phi\right)\right]}} \tag{83}$$

Thus we can obtain the value of $T_\phi$by writing,

$$T_\phi = 2\sqrt{M_1} L \int_{\phi_{min}}^{\phi_{max}} \frac{d\phi}{\sqrt{2\left[E - U_{\text{eff}}\left(\phi\right)\right]}} \tag{84}$$

7.1.3 System trajectory

Consider the expression for $\dot{\theta}(80)$and $\dot{\phi}(82)$. Using chain rule we can write,

$$\frac{d\theta}{dt} = \frac{d\theta}{d\phi}\frac{d\phi}{dt}$$
$$\Rightarrow \frac{d\theta}{d\phi} = \frac{\dot{\theta}}{\dot{\phi}} = \frac{l_z}{L\sin^2\phi}\sqrt{\frac{1}{2M_1\left[E - U_{\text{eff}}\left(\phi\right)\right]}}$$
$$\Rightarrow d\theta = \frac{l_z}{L\sin^2\phi\sqrt{2M_1}}\sqrt{\frac{1}{\left[E - U_{\text{eff}}\left(\phi\right)\right]}}d\phi$$

Integrating for θ and ϕ we can obtain θ as a function of the azimuthal coordinate ϕ. Thus we can have,

$$\theta = \theta_{\text{init}} + \frac{l_z}{L\sqrt{2M_1}}\int_{\phi_{\text{init}}}^{\phi_{\text{final}}} \frac{1}{\sin^2\phi\sqrt{\left[E - U_{\text{eff}}\left(\phi\right)\right]}}d\phi \tag{85}$$

The ***effect of damping*** can be incorporated into the system with the assumption that the damping force is due bob moving through a viscous medium and is ***proportional to the tangential velocity*** of the system. Now there are two such velocities associated with the motion,

- one is due to $\dot{\phi}$ and the corresponding damping is $\alpha L\dot{\phi}$
- the other is due to $\dot{\theta}$and the corresponding damping is $\alpha L\sin\phi\dot{\theta}$

Here α is the ***damping constant***. Note that the tangential velocity due to $\dot{\theta}$ is obtained by multiplying it with the distance of the bob from the z axis and not the suspension point of the pendulum $(L\sin\phi)$, as shown in the figure (13). Thus the modified equations of motion can be represented by the equations

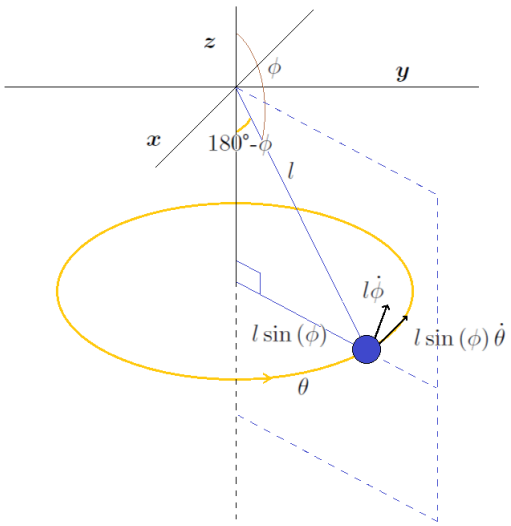


Figure 13: Tangential velocities for the pendulum bob

of motion,

$$\ddot{\phi} = \sin\phi\cos\phi\dot{\theta}^2 + \frac{3g}{2L}\left(\frac{2m_{bob} + m_{rod}}{3m_{bob} + m_{rod}}\right)\sin\phi - \alpha L\dot{\phi} \tag{86}$$
$$\ddot{\theta} = -2\cot\phi\dot{\theta}\dot{\phi} - \alpha L\sin\left(\phi\right)\dot{\theta} \tag{87}$$

7.1.4 Obtaining System trajectory using numerical methods

The equations of motion derived previously can be integrated numerically using ***rk-methods*** or ***adaptive rk-methods***[14] to obtain the θ and ϕ as a function of t. This gives us an approximate information about the pendulum's position as a function of time. The differential governing the system dynamics are given as,

$$\ddot{\phi} = \sin\phi\cos\phi\dot{\theta}^2 + \frac{3g}{2L}\left(\frac{2m_{bob}+m_{rod}}{3m_{bob}+m_{rod}}\right)\sin\phi - \alpha L\dot{\phi}$$
$$\ddot{\theta} = -2\cot\phi\dot{\theta}\dot{\phi} - \alpha L\sin(\phi)\,\dot{\theta}$$

Writing out $\dot{\theta} = \omega_\theta$ and $\dot{\phi} = \omega_\phi$, we reduce the system to a set of four first order differential equations as shown,

$$\dot{\theta} = \omega_\theta \quad \dot{\phi} = \omega_\phi$$

$$\dot{\omega}_\theta = -2\cot\phi\dot{\theta}\dot{\phi} - \alpha L\sin(\phi)\,\dot{\theta}$$
$$\dot{\omega}_\phi = \sin\phi\cos\phi\dot{\theta}^2 + \frac{3g}{2L}\left(\frac{2m_{bob}+m_{rod}}{3m_{bob}+m_{rod}}\right)\sin\phi - \alpha L\dot{\phi}$$

Depending upon the various values of initial conditions and input parameters, namely, $\theta, \dot{\theta}, \phi, \dot{\phi}, L, g$, m_{rod}, m_{bob}, α, we obtain various plots for the space variables. Some of them are shown in figures(15)(16):

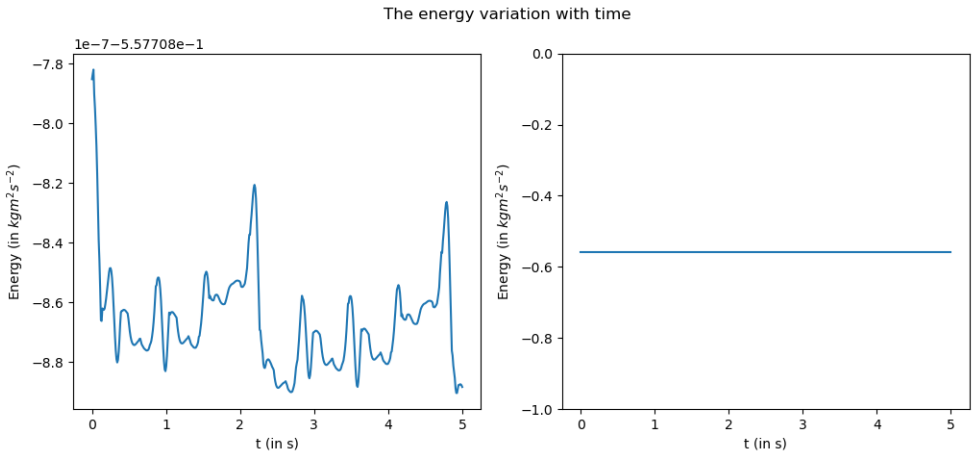


Figure 14: The plot on the left shows the variation of total energy ($T_{total} + U_{total}$) with time for the case with no damping ($\alpha = 0$). The change is of the order of 10^{-7} kgm^2s^{-2}. The plot on the right shows the same with a zoomed out view showing that the energy is almost constant.

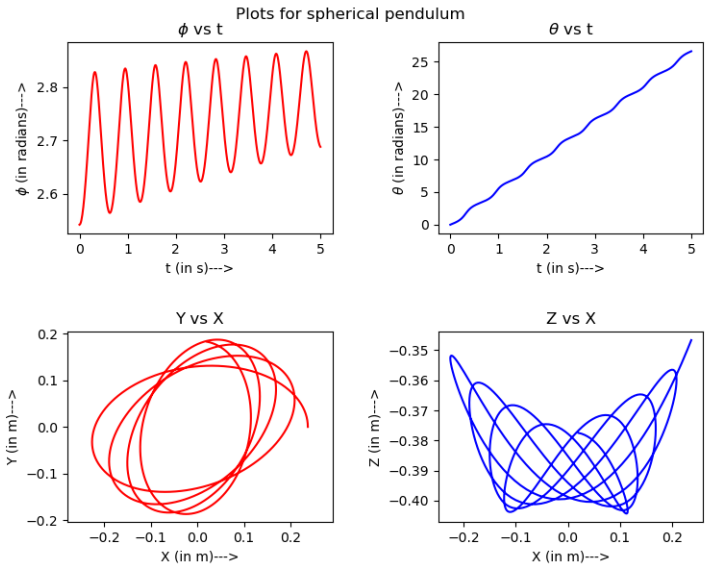


Figure 15: Plots for the spherical pendulum with $\theta = 0.0$, $\dot{\theta} = (\pi - 0.6)\ s^{-1}$, $\phi = 3.0$, $\dot{\phi} = 0.0\ s^{-1}$ (the angles are in radians) and, $L = 0.42\ m$, $g = 9.8\ ms^{-2}$, $m_{rod} = 0.2\ kg$, $m_{bob} = 0.2\ kg$, $\alpha = 0.5\ m^{-1}s^{-1}$

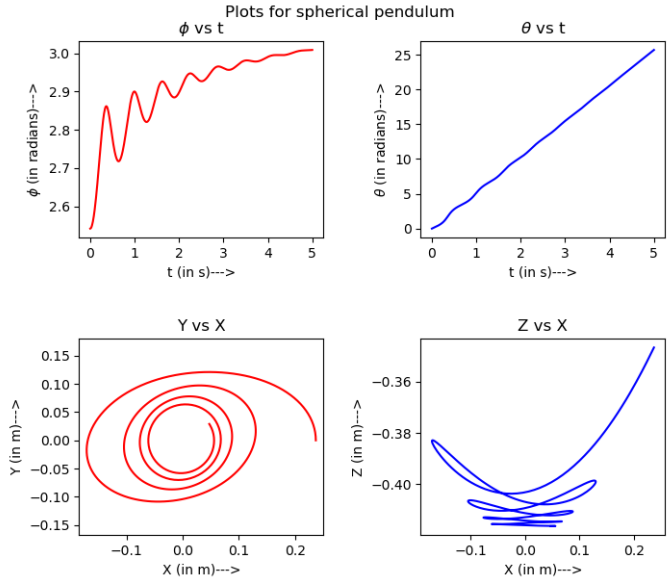


Figure 16: Plots for the spherical pendulum with $\theta = 0.0$, $\dot{\theta} = (\pi - 0.6)\ s^{-1}$, $\phi = 3.0$, $\dot{\phi} = 0.0\ s^{-1}$ (the angles are in radians) and, $L = 0.42\ m$, $g = 9.8\ ms^{-2}$, $m_{rod} = 0.2\ kg$, $m_{bob} = 0.2\ kg$, $\alpha = 5.0\ m^{-1}s^{-1}$

7.2 Magnetic Pendulum

Before getting into the dynamics of the magnetic pendulum, it is necessary to model the magnets to obtain an expression for their interaction potential energy needed for constructing the system Lagrangian.

7.2.1 Modeling cylindrical magnets

Consider a cylindrical magnet with radius r_{magnet}, length l_{magnet} and frozen ***uniform magnetization*** along the axial direction $\overrightarrow{M} = M\hat{z}$[12]. For a permanent magnet with ***no external field*** applied, we have the free current density $\overrightarrow{J_f} = 0$ everywhere. This makes $\nabla \times \overrightarrow{H} = 0$ everywhere and the magnetic field, $\overrightarrow{H}$ can be described as the gradient of a scalar potential ϕ_s,

$$\overrightarrow{H} = -\nabla\phi_s \tag{88}$$

Now, using the relations $\overrightarrow{H} = \frac{\overrightarrow{B}}{\mu_0} - \overrightarrow{M}$ and $\nabla \cdot \overrightarrow{B} = 0$, where $\overrightarrow{B}$ is the magnetic induction and μ_0 is the magnetic permeability of free space, and taking divergence on both sides we arrive at,

$$\begin{aligned} \nabla \cdot \overrightarrow{H} &= -\nabla \cdot \overrightarrow{M} \\ \nabla^2\phi_s &= \nabla \cdot \overrightarrow{M} \end{aligned} \tag{89}$$

This is mathematically identical to the *Poisson's equation* in electrostatics,

$$\nabla^2 V = -\frac{\rho}{\epsilon_0} \tag{90}$$

With $-\nabla \cdot \overrightarrow{M}$ in place of $\frac{\rho}{\epsilon_0}$ as the "*source*". And for $-\nabla \cdot \overrightarrow{M}$ ***going to zero at infinity***, the solution can be given by the same prescription as in case of the *Poisson's equation*,

$$\phi_s = -\frac{1}{4\pi}\int_{\tau'} \frac{\nabla \cdot \overrightarrow{M}\,(\overrightarrow{r}')\,d\tau'}{\xi} = \frac{1}{4\pi}\int_{\tau'} \frac{\nabla' \cdot \overrightarrow{M}\,(\overrightarrow{r}')\,d\tau'}{\xi} \tag{91}$$

where $\xi = |\overrightarrow{r} - \overrightarrow{r}'|$, $\overrightarrow{r}$ the position vector of the point P at which the potential is being calculated, $\overrightarrow{r}'$ the position vector of the source, and the integration is over the volume $\tau'$with $d\tau'$ being the elemental volume (17). Using the *Gauss's Divergence Theorem* the above equation can be expressed as a closed surface integral over the surface S' enclosing volume τ'.

$$\phi_s = \oint_{S'} \frac{\left(\overrightarrow{M} \cdot \hat{n}\right) dS'}{\xi}$$

The idea of ***multipole expansion*** can be employed to expand ϕ_s in terms of the power series in $\frac{1}{\xi}$, provided that the points of interest P is outside the minimum bounding sphere that encompasses the magnet. If ξ is sufficiently large, the series will be dominated by the lowest non-vanishing contribution, and the higher terms can be ignored.

$$\frac{1}{\xi} = \frac{1}{\sqrt{r^2 + (r')^2 - 2rr'\cos\alpha}} = \frac{1}{r}\sum_{n=0}^{\infty}(\frac{r'}{r})^n P_n\left(\cos\alpha\right) \tag{92}$$

where α is the angle between $\overrightarrow{r}$ and $\overrightarrow{r}'$and $P_n\left(\cos\alpha\right)$ is the n^{th} order Legendre Polynomial. Accordingly the scalar potential ϕ_s (91) can be written as,

$$\begin{aligned} \phi_s &= \frac{1}{4\pi}\sum_{n=0}^{\infty}\frac{1}{r^{n+1}}\oint (r')^n \overrightarrow{M} \cdot \hat{n} P_n\left(\cos\alpha\right) dS' \\ &= \frac{1}{4\pi}\left[\frac{1}{r}\oint_{S'} \overrightarrow{M} \cdot \hat{n} dS' + \frac{1}{r^2}\oint_{S'} r'\overrightarrow{M} \cdot \hat{n}\left(\cos\alpha\right) dS' + \frac{1}{r^3}\oint_{S'} (r')^2 \overrightarrow{M} \cdot \hat{n}\left(\frac{3}{2}\cos^2\alpha - \frac{1}{2}\right) dS' + \ldots\right] \end{aligned}$$

The ***monopole term vanishes*** because $\overrightarrow{M} \cdot \hat{n}$ is zero for the ***curved surface of a cylinder*** and the contribution of the integrals from the upper and lower flat surfaces of the cylinder ***cancel each other out***. In absence of any monopole contribution, the dominant term is the dipole,

$$\phi_{dipole} = \frac{1}{4\pi r^2}\oint_{S'} \left(r'\cos\alpha\right)\overrightarrow{M} \cdot \hat{n} dS' = \frac{1}{4\pi}\frac{\hat{r}}{r^2} \cdot \oint_{S'} r'\overrightarrow{M} \cdot \hat{n} dS' \tag{93}$$

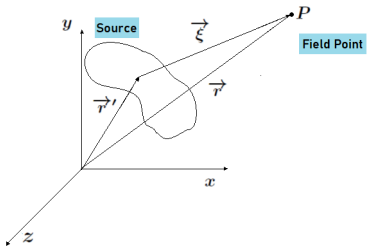


Figure 17: Positions for Source and Point P

Thus the magnetic induction $\overrightarrow{B}$ due to the magnet at point P can be given by[25],

$$\begin{aligned}\overrightarrow{B}_{dipole} &= \mu_0 \overrightarrow{H}_{dipole} \\ &= -\mu_0 \nabla \phi_{dipole} \\ &= -\frac{\mu_0}{4\pi} \nabla \left[\left(\frac{\hat{r}}{r^2} \right) \cdot \left(\oint_{S'} r' \overrightarrow{M} \cdot \hat{n} dS' \right) \right] \\ &= -\frac{\mu_0}{4\pi} \nabla \left[\left(\frac{\hat{r}}{r^2} \right) \cdot \overrightarrow{c} \left(\overrightarrow{r}' \right) \right] \end{aligned} \tag{94}$$

here the the integral $\oint_{S'} r' \overrightarrow{M} \cdot \hat{n} dS'$ is replaced by $\overrightarrow{c} \left(\overrightarrow{r}' \right)$. Using the vector result,

$$\nabla \left(\overrightarrow{a} \cdot \overrightarrow{b} \right) = \left(\overrightarrow{a} \cdot \nabla \right) \overrightarrow{b} + \left(\overrightarrow{b} \cdot \nabla \right) \overrightarrow{a} + \overrightarrow{a} \times \left(\nabla \times \overrightarrow{b} \right) + \overrightarrow{b} \times \left(\nabla \times \overrightarrow{a} \right)$$

$\overrightarrow{B}_{dipole}$can be expanded in the form,

$$\overrightarrow{B}_{dipole} = -\frac{\mu_0}{4\pi} \left[\left(\frac{\hat{r}}{r^2} \cdot \nabla \right) \overrightarrow{c} \left(\overrightarrow{r}' \right) + \left(\overrightarrow{c} \left(\overrightarrow{r}' \right) \cdot \nabla \right) \frac{\hat{r}}{r^2} + \frac{\hat{r}}{r^2} \times \left(\nabla \times \overrightarrow{c} \left(\overrightarrow{r}' \right) \right) + \overrightarrow{c} \left(\overrightarrow{r}' \right) \times \left(\nabla \times \frac{\hat{r}}{r^2} \right) \right] \tag{95}$$

The divergence and the curl operators are defined with respect to $\overrightarrow{r}$. Thus the terms $\left(\frac{\hat{r}}{r^2} \cdot \nabla \right) \overrightarrow{c} \left(\overrightarrow{r}' \right)$ and $\frac{\hat{r}}{r^2} \times \left(\nabla \times \overrightarrow{c} \left(\overrightarrow{r}' \right) \right)$ must be zero[12]. Also $\nabla \times \frac{\hat{r}}{r^2}$ is zero, making the term $\overrightarrow{c} \left(\overrightarrow{r}' \right) \times \left(\nabla \times \frac{\hat{r}}{r^2} \right)$ zero as well. Thus we are left with,

$$\begin{aligned}\overrightarrow{B}_{dipole} &= -\frac{\mu_0}{4\pi} \left(\overrightarrow{c} \left(\overrightarrow{r}' \right) \cdot \nabla \right) \frac{\hat{r}}{r^2} \\ &= \frac{\mu_0}{4\pi r^3} \left[3 \left(\overrightarrow{c} \cdot \hat{r} \right) \hat{r} - \overrightarrow{c} \right] \end{aligned} \tag{96}$$

This is the exact expression for the magnetic field at point P due to a dipole moment $\overrightarrow{c}$ ***placed at the origin***. It can be shown that $\overrightarrow{c}$ is indeed the dipole of the cylindrical magnet with magnetization $\overrightarrow{M}$ by evaluating the integral $\overrightarrow{c} = \oint_{S'} r' \overrightarrow{M} \cdot \hat{n} dS'$ (94), with the origin considered at the center of the magnet. $\overrightarrow{M} \cdot \hat{n}$ is zero for the curved surface of the cylinder. Thus the integral is evaluated for the flat surfaces of the magnet.

$$\begin{aligned}\overrightarrow{c} &= \int_{S_1} r' \overrightarrow{M} \cdot \hat{n} dS' + \int_{S_2} r' \overrightarrow{M} \cdot \hat{n} dS' \\ &= \int_{S_1} r' M \left(\hat{z} \cdot \hat{z} \right) dS' + \int_{S_2} r' M \left(\hat{z} \cdot -\hat{z} \right) dS' \\ &= 2\pi M \left[\int_0^{r_{maget}} \left(\frac{l_{magnet}}{2} \hat{z} + \rho \hat{\rho} \right) \rho d\rho - \int_0^{r_{magnet}} \left(-\frac{l_{magnet}}{2} \hat{z} + \rho \hat{\rho} \right) \rho d\rho \right] \\ &= \pi M l_{magnet} r_{magnet}^2 \end{aligned} \tag{97}$$

But $\pi l_{magnet} r^2_{magnet}$ is the ***volume of the magnet***[25], making $\overrightarrow{c}$ the magnetic dipole moment $\left(= \overrightarrow{M} \times total\ volume\right)$ of the magnet. This allows us to define the ***magnetic interaction energy*** between two dipoles,

$$\begin{aligned} U_m &= -\overrightarrow{c}' \cdot \overrightarrow{B}_{dipole} \\ &= -\frac{\mu_0}{4\pi r^3} \left[3\left(\overrightarrow{c} \cdot \hat{r}\right)\left(\overrightarrow{c}' \cdot \hat{r}\right) - \overrightarrow{c} \cdot \overrightarrow{c}'\right] \end{aligned} \tag{98}$$

7.2.2 Equations of motion from the Lagrangian

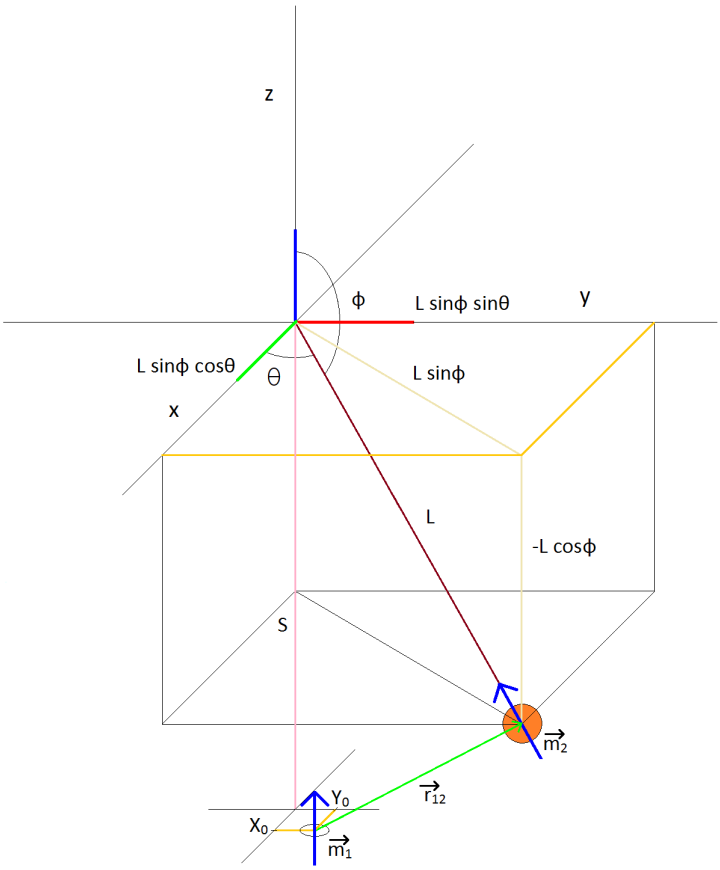


Figure 18: Magnetic pendulum setup

A magnetic pendulum setup is obtained by replacing the non-magnetic bob by a magnetic bob. Another magnet is fixed at position $(X_0, Y_0, -S)$ with $S > L$. The magnets are modeled as solid cylinders with uniform magnetization along their axial directions as shown in the section (7.2.1). Let the magnets have magnetic dipole moments $\overrightarrow{m}_1$and $\overrightarrow{m}_2$ with equal magnitudes as shown in figure (18). The magnetic dipole interaction energy for the system as obtained in the section (7.2.1) is given by(98),

$$U_m = -\frac{\mu_0}{4\pi r^3_{12}} \left[3\left(\overrightarrow{m}_1 \cdot \hat{r}_{12}\right)\left(\overrightarrow{m}_2 \cdot \hat{r}_{12}\right) - \overrightarrow{m}_1 \cdot \overrightarrow{m}_2\right]$$

where $\overrightarrow{r}_{12}$ is the position vector of $\overrightarrow{m}_2$ with respect to $\overrightarrow{m}_1$and μ_0 is the magnetic permeability of free space. Now, we can expand U_m by writing out the expressions for $\overrightarrow{m}_1 \cdot \hat{r}_{12}$, $\overrightarrow{m}_2 \cdot \hat{r}_{12}$ and $\overrightarrow{m}_1 \cdot \overrightarrow{m}_2$. In

general we have,

$$\begin{aligned}\overrightarrow{r}_{12} &= \overrightarrow{r}_2 - \overrightarrow{r}_1 \\ &= position\ of\ m_2\ (x,y,z) - position\ of\ m_1\ (X_0, Y_0, -S) \\ \overrightarrow{r}_{12} &= (L\sin\phi\cos\theta - X_0)\,\hat{i} + (L\sin\phi\sin\theta - Y_0)\,\hat{j} + (L\cos\phi + S)\,\hat{k}\end{aligned} \tag{99}$$

Thus the magnitude r_{12} and the unit vector $\hat{r}_{12}$ for $\overrightarrow{r}_{12}$are given by,

$$r_{12} = \sqrt{L^2 + S^2 + X_0^2 + Y_0^2 + 2LS\cos\phi - 2L\left(X_0\cos\theta + Y_0\sin\theta\right)\sin\phi} \tag{100}$$

$$\hat{r}_{12} = \frac{1}{r_{12}}\left[(L\sin\phi\cos\theta - X_0)\,\hat{i} + (L\sin\phi\sin\theta - Y_0)\,\hat{j} + (L\cos\phi + S)\,\hat{k}\right] \tag{101}$$

Again$\overrightarrow{m}_1 = m_1\hat{k}$ since dipole moment $\overrightarrow{m}_1$ is fixed in place with its magnetization pointing towards positive. Note that the magnitude of the magnetic dipole moment as calculated in the section (7.2.1) is $m_1 = M\pi r_{magnet}^2 l_{magnet}$(97). Similarly $\overrightarrow{m}_2 = -\frac{m_2}{L}\left(\sin\phi\cos\theta\hat{i} + \sin\phi\sin\theta\hat{j} + \cos\phi\hat{k}\right)$ where $m_2 = M\pi r_{magnet}^2 l_{magnet}$ and $\overrightarrow{m}_2$is pointing in the direction from the pendulum bob to the suspension point. Thus,

$$\overrightarrow{m}_1 \cdot \overrightarrow{m}_2 = -m_1 m_2\cos\phi \tag{102}$$

$$\overrightarrow{m}_1 \cdot \hat{r}_{12} = \frac{m_1}{r_{12}}\left(L\cos\phi + S\right) \tag{103}$$

$$\overrightarrow{m}_2 \cdot \hat{r}_{12} = -\frac{m_2}{r_{12}}\left[\sin\phi\cos\theta\left(L\sin\phi\cos\theta - X_0\right) + \sin\phi\sin\theta\left(L\sin\phi\sin\theta - Y_0\right) + \cos\phi\left(L\cos\phi + S\right)\right] \tag{104}$$

Substituting the values of these expressions into U_m (98)we obtain,

$$U_m = \frac{\mu_0 m_1 m_2}{4\pi r_{12}^3}\left[\frac{3}{r_{12}^2}\left(L\cos\phi + S\right)\left\{L - \sin\phi\left(X_0\cos\theta + Y_0\sin\theta\right) + S\cos\phi\right\} - \cos\phi\right] \tag{105}$$

Thus the total potential energy of the system is given by,

$$U_{total} = U_{rod} + U_{bob} + U_m \tag{106}$$

Thus the equations of motion for the system are now modified as(86)(87),

$$\ddot{\phi} = \sin\phi\cos\phi\dot{\theta}^2 + \frac{3g}{2L}\left(\frac{2m_{bob} + m_{rod}}{3m_{bob} + m_{rod}}\right)\sin\phi - \frac{1}{\left(m_{bob} + \frac{m_{rod}}{3}\right)}\frac{\partial U_m}{\partial\phi} - \alpha L\dot{\phi} \tag{107}$$

$$\ddot{\theta} = -2\cot\phi\dot{\theta}\phi - \frac{\dot{1}}{\sin^2\phi L^2\left(m_{bob} + \frac{m_{rod}}{3}\right)}\frac{\partial U_m}{\partial\theta} - \alpha L\sin\left(\phi\right)\dot{\theta} \tag{108}$$

7.2.3 Studying system trajectory under different conditions

The equations of motions can be integrated numerically [14] to obtain the θ and ϕ as a function of t. This gives us an approximate information about the pendulum's position as a function of time. Under mild magnetic effects the system, shows periodic and predictable behaviors. However with the increase in magnetic influence trajectory that the system follows becomes greatly dependent on the initial conditions. The plot in figure (19) shows an example of the sensitive dependence of the motion of the system on the initial conditions[26].

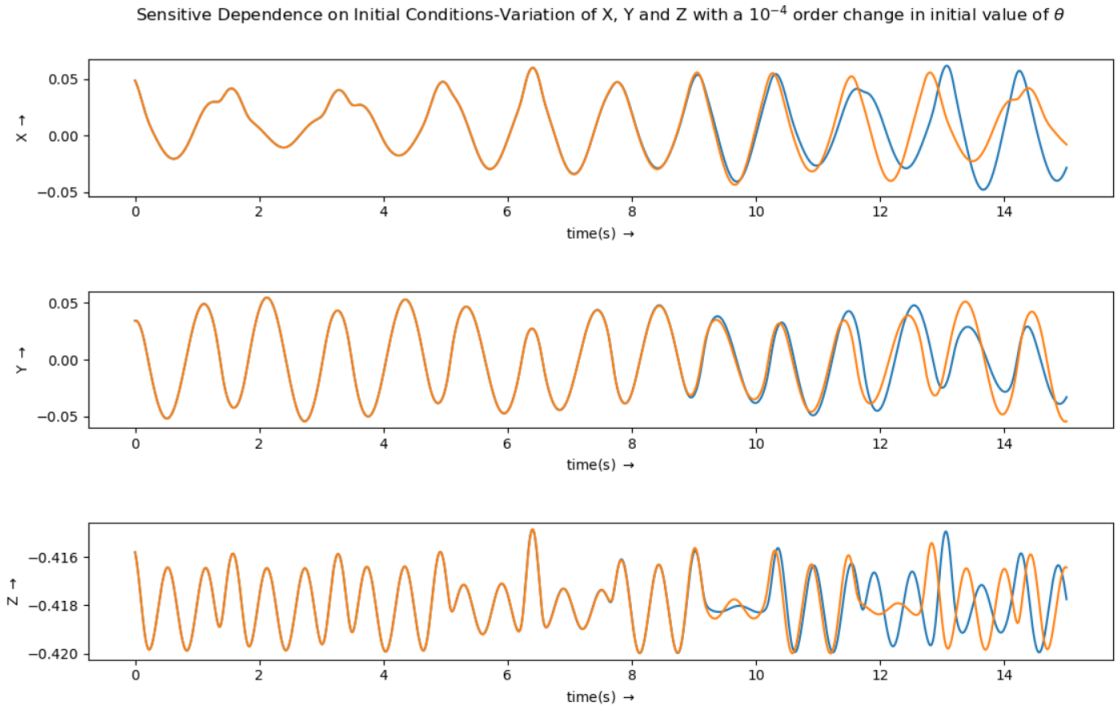


Figure 19: The two curves in each of the plots differ only in the initial θ value. Blue curve corresponds to $\theta = 0.6136$ $(radians)$ while the orange curve corresponds to $\theta = 0.6136 + 0.0001$ $(radians)$. $\dot{\theta} = 1.2657\ s^{-1}$, $\phi = 3.0$, $\dot{\phi} = 0.1883\ s^{-1}$ (the angles are in radians) and, $S = 0.46\ m$, $X_0 = 0.05\ m$, $Y_0 = 0.00\ m$, $m_1 = 2.5\ Am^2$, $m_2 = 2.5\ Am^2$, $L = 0.42\ m$, $g = 9.8\ ms^{-2}$, $m_{rod} = 0.2\ kg$, $m_{bob} = 0.2\ kg$, $\alpha = 0.0\ m^{-1}s^{-1}$.

As is evident from the plots (19), the two curves with a slightly different value of initial θ superimpose on each other up to a certain time and then show completely different behaviors. This deviation in the behavior between the two curves increases with the increase in value of the dipole moments of the two magnets. The deviation increases with the increase in difference between the initial conditions.

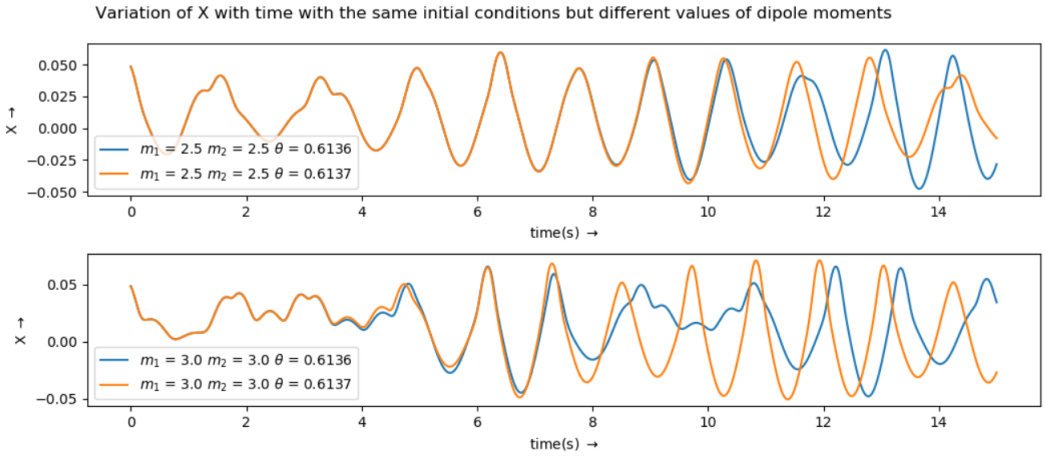


Figure 20: Similar curves as the previous figure with (i) $m_1 = 2.5\ Am^2$, $m_2 = 2.5\ Am^2$and (ii) $m_1 = 3.0\ Am^2$, $m_2 = 3.0\ Am^2$.

In figure (20), the deviation in the curves for the first subplot with lower values of dipole moments for the two magnets starts at around 9.074 s, while that for the second subplot with higher values of dipole moment starts at around 3.483 s. So we can conclude that higher the magnetic effect in the spherical pendulum system greater is the sensitive dependence on initial conditions. The case of the non-magnetic spherical pendulum, showed no such behavior.

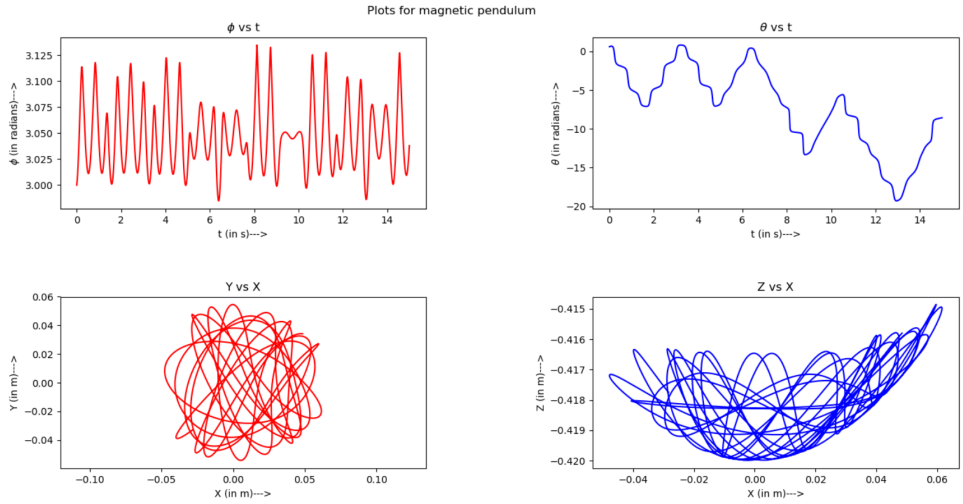


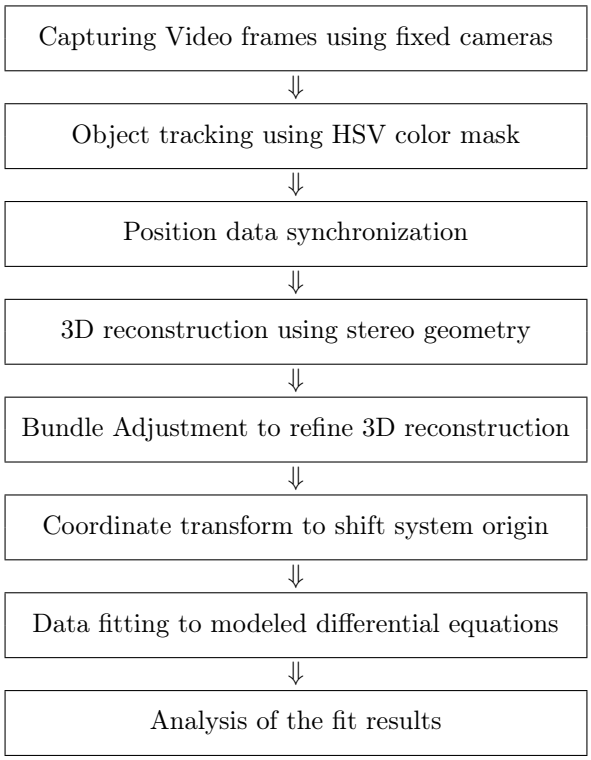
Figure 21: Plots for magnetic pendulum when the magnetic force between bob and the external magnet is attractive.

Figure (21) shows that in the first two plots, no periodicity of the motion is observed in the specific time interval for which the simulation is carried out. The third and the fourth plots basically shows the trajectory of the motion of the bob from two different views. This motion is completely different from that obtained in the case without any magnetic influence.

8 Experiment

8.1 Experiment Summary

The goal of this experiment is to develop a ***generalized sequence of steps*** that captures the motion of a pendulum system using a ***pair of cameras*** and then use stereo geometry to reconstruct the the motion of the system in 3D, to verify the extent to which a set of modeled differential equations can describe the physical system. An experimental setup is constructed for the spherical pendulum which constitutes the system to be studied. A pair of ***un-synchronized*** cameras are positioned at a ***fixed relative orientation*** with respect to the experimental setup. The video capture ***starting time is noted*** for each of the cameras and the video frames are captured sequentially to track the pendulum bob using an ***HSV*** color mask (5). The tracked data for the pendulum bob is then synchronized and the motion is reconstructed using ***stereo geometry*** and ***linear triangulation*** (4.3)(4.4). A non-linear least square algorithm is then used to refine the pendulum trajectory in 3D (4.5). Finally the data set for the pendulum trajectory is fit to a set of modeled differential equations using non-linear least square fitting(2.1). The flowchart below summarizes *the* ***sequence of steps required to experimentally analyze the motion of a dynamical system using 3D reconstruction***:



8.2 Hardware Setup

The setup consists of a ***cylindrical brass bob*** suspended from a rigid support using a rigid rod. A solid flat board is chosen to serve as the base for the pendulum stand. The stand is made in a way that the pendulum can be lifted or lowered without changing the length of the suspending rod. For this, holes are made at different heights in the two opposite pillars and provision is made for clamping the suspension bar to the pair of holes at a particular height. The rod is suspended from the suspension bar using a thin cotton thread to minimize friction and torsion at the suspension point.
For the magnetic pendulum setup, a thin cylindrical magnet is fixed to the base of the brass bob with its geometric axis aligned with the geometric axis of the brass bob. Another identical magnet is fixed to the base with its axis perpendicular to the surface of the base. The magnets are placed in a way that when the bob is suspended vertically similar poles of the magnets are pointing up. A detailed description for the setup build is given below:

8.2.1 Pendulum bob

A cylindrical bob with a brass base and a blue plastic covering is used as the pendulum bob. A small hole is bored into the cylindrical bob through one of the flat faces, along its axis for inserting the rigid suspension rod. A thin cylindrical neodymium magnet is fixed to the other flat surface of the bob using a heat resistant adhesive. The magnet is fixed in a way that its geometrical axis is approximately (*mm precision*) aligned to the geometrical axis of the bob. The magnet is modeled (7.2.1) as a solid cylinder with uniform magnetization along the magnetic axis. The magnetic induction due to the modeled magnetic cylinder is given by the expression(96):

$$\overrightarrow{B}_{magnet} = \frac{\mu_0}{4\pi r^3}\left[3\left(\overrightarrow{m}\cdot\hat{r}\right)\hat{r} - \overrightarrow{m}\right]$$

Figure 22: Picture of the pendulum bob

where $\overrightarrow{m} = \overrightarrow{M}\pi r_{magnet}^2 l_{magnet}$ is the magnetic dipole moment of the ***neodymium magnet*** approximated to be a magnetic dipole, $\overrightarrow{r}$ is the vector pointing from the center of the magnet to the point where the magnetic induction is to be calculated and μ_0 is the magnetic permeability of free space. The bob and the magnet specifications are provided in the table below -

measurement quantity	***value***	***l.c.***
radius of the cylindrical bob (r_{bob})	1.3 cm	0.1 cm
height of the cylindrical bob (l_{bob})	3.1 cm	0.1 cm
mass of the cylindrical bob (m_{bob})	33.17 g	0.01 g
RGB signature for the bob	#1636CC	-
radius of the magnetic cylinder (r_{magnet})	0.5 cm	0.1 cm
height of the magnetic cylinder (l_{magnet})	0.1 cm	0.1 cm

Table 1: Bob and magnet specifications

8.2.2 Suspension Rod

The bob is suspended using a stainless steel rod. A part of this rod is inserted into the bob and fixed in place using adhesive. A small hook is attached to the other end of the rod. A cotton thread is passed through this hook to suspend it from the rigid support. A steel rod is used since it has moderately high bending stiffness. The rod specifications are listed in the table below -

measurement quantity	***value***	***l.c.***
mass of the rod (m_{rod})	6.7 g	0.01 g
diameter of the rod (d_{rod})	0.16 cm	-
length of the rod from the suspension point to bob center (l_{rod})	42.3 cm	0.1 cm

Table 2: Suspension rod specifications

8.2.3 Wooden platform

A flat plywood of rectangular shape is used as the base for the setup. The pillars for the support is fixed to the base using nuts and bolt. Markings are made to mark the center of the plywood and field of camera views. An identical neodymium magnet as the one attached to the base of the pendulum bob, is

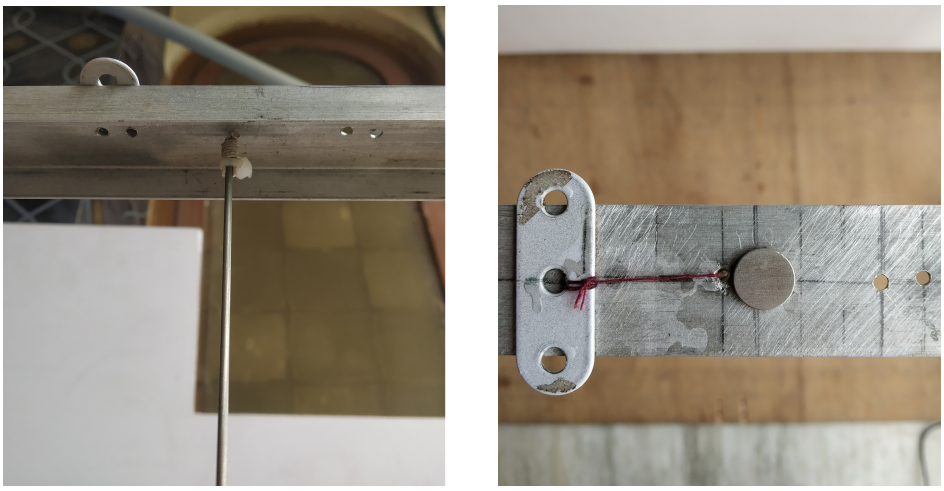


Figure 23: Picture of suspension rod and suspension point

fixed on the wooden base within a radius of 2 cm to 4 cm centering the equilibrium projection point of the pendulum bob on the wooden base. The magnet should be placed in a way that similar poles of the pendulum magnet and this magnet should be pointing upwards. The dimensions of the plywood board are given as follows -

measurement quantity	***value***	***l.c.***
length of the base (l_{base})	61.0 cm	0.1 cm
breadth of the base (b_{base})	40.0 cm	0.1 cm
thickness of the base (t_{base})	0.5 cm	0.1 cm

Table 3: Wooden base dimensions

8.2.4 Suspension frame

A suspension frame is made from aluminium channels to suspend the pendulum. It consists of two aluminium pillars attached upright to the breadth-wise center of the wooden board at the lengthwise extremes. The pillars are bridged at the top by another aluminium channel screwed at right angles to the pillars. Now the vertical pillars have holes drilled at gaps of 1 cm along their length. This facilitates lowering or raising the horizontal bridging frame by clamping it to the holes in the vertical pillars, to adjust for various suitable heights. A small hole is drilled at the approximate center of the horizontal bar to suspend the pendulum rod using a cotton thread. The dimensions of the various frame attachments are as follows -

measurement quantity	***value***	***l.c.***
length of the vertical pillars	47.2 cm	0.1 cm
distance between the pillars	59.0 cm	0.1 cm

Table 4: Suspension frame dimensions

8.2.5 Cameras and their positions

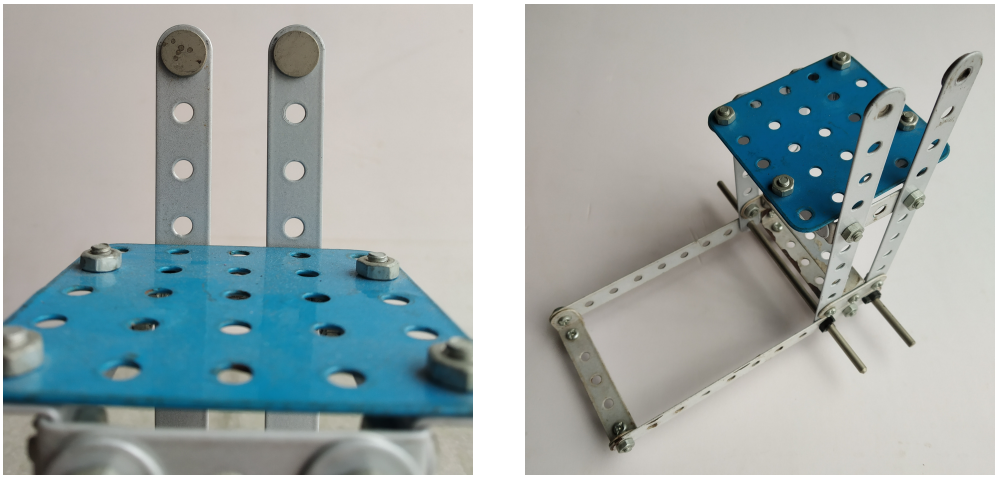


Figure 24: Picture of camera stand

A pair of ***un-synchronized phone cameras*** are use to capture the pendulum trajectory from a fixed relative orientation. The ***relative orientation needs to stay fixed*** while capturing the motion of the system. One of the cameras is ***attached to the horizontal bar*** of the suspension frame capturing the scene from above. The camera fits into a ***magnetic clamp*** fixed to the platform. The other camera is similarly attached to a separate stand and captures the system from the side. This orientation of the cameras is found to produce the best results for 3D reconstruction. This seems to be quite intuitive as cameras can capture almost all the orthogonal degrees of freedom of the the system in this orientation. There is no ***hardware or software synchronizations*** between the cameras. The video capturing process can be formally described as follows -

- Let the position of the tracked object moving in a ***smooth trajectory in 3D space*** be described by:

$$\overrightarrow{Q}(t) = \begin{bmatrix} Q_1(t) \\ Q_2(t) \\ Q_3(t) \\ 1 \end{bmatrix}$$

  where t denotes time.

- Un-synchronized cameras C_{m1} and C_{m2} are projecting $\overrightarrow{Q}(t)$ onto their respective camera planes, proucing two 2D trajectories, $\overrightarrow{q}_1(t)$ and $\overrightarrow{q}_2(t)$. Let C_{m1} capture frames at an interval of f_{c1} seconds and the second camera take frames at intervals of f_{c2} seconds. This leads to a sequence of samples

$$\overrightarrow{p}_1(m) = \begin{bmatrix} u_{1m} \\ v_{1m} \\ 1 \end{bmatrix} \quad \overrightarrow{p}_2(n) = \begin{bmatrix} u_{2n} \\ v_{2n} \\ 1 \end{bmatrix}$$

  - of $\overrightarrow{q}_1(t_{1m})$ at times $t_{1m}(m) = t_{10} + mf_{c1}$ from camera C_{m1}, where $m = \{0, 1, 2, 3, ..., M\}$ and t_{10} is the time at which the 0^{th} frame is captured
  - of $\overrightarrow{q}_2(t_{2n})$ at times $t_{2n}(n) = t_{20} + nf_{c2}$ from camera C_{m2}, where $n = \{0, 1, 2, 3, ..., N\}$ and $t_{20}$is the time at which the 0^{th} frame is captured

The variations in the interval at which the the cameras capture successive frames is of the ***order of microseconds*** as compared to the frame capture interval which is of the order of 10 milliseconds and

is neglected. ***The camera calibrations are carried out prior to the experiment*** to obtain the camera intrinsic parameters and the distortion coefficients. The specifications for camera 1 and camera 2 are summarized below.

camera model	Xiaomi Redmi Note 5 pro
video resolution	1920 × 1080 pixels
fps	60
aperture	f/2.2
pixel color	8 bit

camera model	Xiaomi Redmi Note 5 pro
video resolution	1920 × 1080 pixels
fps	60
aperture	f/2.2
pixel color	8 bit

Table 5: Camera specifications

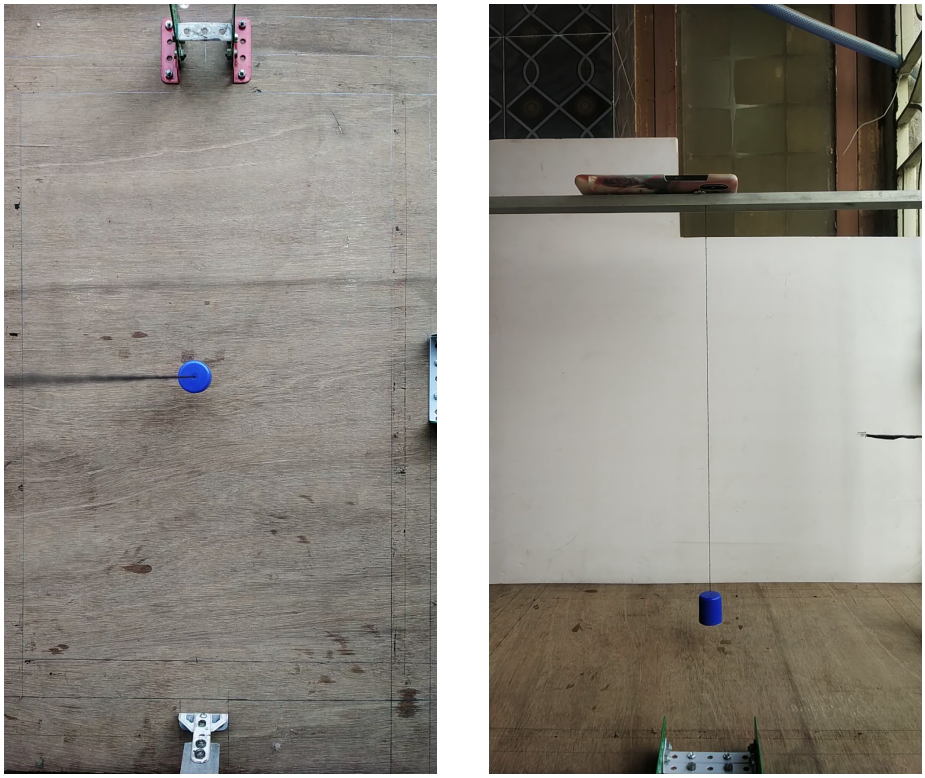


Figure 25: Camera views for camera 1 and camera 2

8.2.6 Hardware Approximations

Following are the list of hardware approximations considered while performing the experiment -

- The camera frame rate for cameras C_{m1} and C_{m2} remains constant. The variations in the time interval between the frame captures is of the order of few microseconds and hence is neglected.
- The neodymium magnet attached to the base of the pendulum bob is considered to be exactly aligned with the axis of the bob cylinder.
- The suspension rod and the suspension frame are ***completely rigid***.
- The bob is considered to be a ***point mass***.

- ***Torsional resistance*** and ***friction*** at the point of suspension of the pendulum is neglected.
- The damping in the motion is due to the bob ***moving through a viscous medium***.
- The magnets are being modeled as magnetic dipoles with dipole moment $\overrightarrow{m} = \overrightarrow{M}\pi r^2_{magnet} l_{magnet}$, where $\overrightarrow{M}$ is the magnetization of the magnetic cylinder, ***uniform throughout its volume*** and pointing along the cylinder axis.

8.3 Analysis Software

The numerical analysis for the experiment is carried out using **Python** (**v 2.7.18**). The library dependencies used are listed below:

- Numerical Python Library - **Numpy** (v 1.18.1)
- Open Source Computer Vision Library - **OpenCV** (v 4.2.0)
- Scientific Python Library - **Scipy** (v 1.6.2)
- Python Mathematical Plotting Library - **Matplotlib** (v 3.4.1)

Once the videos for the system trajectory is captured using the cameras, the data is to be analyzed for reconstruction and curve fitting. This is performed by a list of python scripts each of them performing a specific task.

8.3.1 camera_calibrate.py

(A)A video of a checkerboard pattern captured from different angles is used to obtain the camera intrinsic parameters. The script grabs a sequence of 40 frames captured at regular intervals from the video file. It then uses ***openCV's camera calibration module*** to compute the camera's orthogonal focal lengths and camera centers, and finally outputs the camera intrinsic matrix(B).

calibration video file → **computes camera center and orthogonal focal lengths** → camera intrinsic matrix

8.3.2 object_track.py

AThis python script performs the tracking of the object of interest (5). It inputs the video files, and grabs the frames one by one using the openCV library. A ***general image pre-processing*** is carried out using *Gaussian Blurring* and a color mask is created by subtracting everything in the scene except for the pendulum bob with a specific color mask. Noisy blobs of unwanted masks are removed using erosion and dilution and then a contour is created for the largest available mask. The ***center of the contour*** is computed in pixels and stored as the image position data along with the time stamps for the video frames into a *.txt* data file(B). The working of the script is summarized below.

system video file → **tracks bob position in image plane with time** → tracked data file

8.3.3 frame_capture.py

(A)frame_capture.py is used to capture a single video frame. It simply inputs a video file and a frame number $m = 1, 2, 3, ..., M$ and outputs the frame to a matplotlib window. It also enables us to find the HSV color or the coordinate of a particular pixel by clicking on it in the real time window.

system video file, frame number m → **real time image processing** → image color data, coordinate data

8.3.4 data_synchronize.py

(A)The purpose of this script is to synchronize the position data from the two cameras over time using epipolar constraint (6). It takes in the image position data and the time data from the data files for the two cameras and a guess value for the time shift between the videos as the input. The script then uses epipolar geometry to improve the time shift iteratively using a minimization technique(6.4). The synchronized point correspondences and time data for the two views are then saved to .npz files(B).

un-synchronized data, guess value for time shift → **synchronizes image position data using epipolar geometry** → synchronized data, synchronized time data

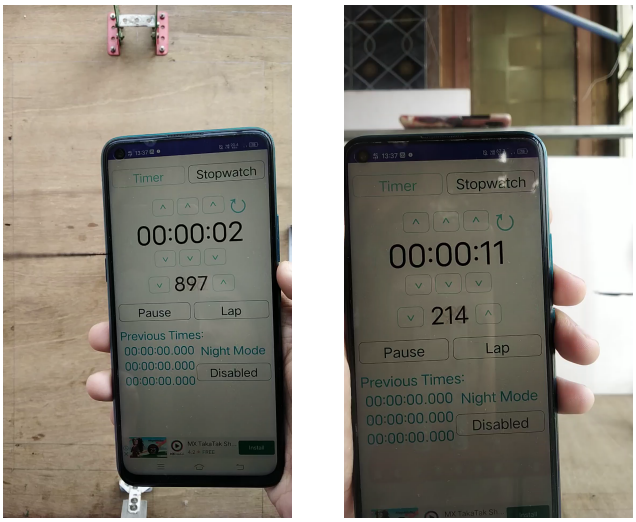


Figure 26: First frame capture time for two views

8.3.5 find_fundamental_matrix_eightpoint.py

(A)This script computes the *fundamental matrix* using *eight point algorithm* (4.6). It takes in the synchronized point correspondences as input and then uses eight point algorithm to compute the fundamental matrix.

point correspondences → **computes fundamental matrix using eight point algorithm** → fundamental matrix

8.3.6 reconstruction.py

ALinear triangulation (4.4) is used to reconstruct the 3D trajectory from the point correspondences and the fundamental matrix. It takes in the point correspondences, the fundamental matrix and the camera intrinsic matrices as inputs. The essential matrix is computed using the fundamental matrix and the intrinsic matrices. It then computes the four possible combination of the camera relative rotations and the translations and figures out the correct relative orientation using linear triangulation on a test data. This correct pose matrix is use to figure out the camera matrices for the two cameras and triangulate the trajectory in 3D space(4.6).

point correspondences, camera intrinsic parameters, fundamental matrix → **computes the 3D trajectory using linear triangulation** → 3D data points, the projection matrices, reprojection error

8.3.7 bundle_adjust.py

(A)This script uses non-linear least squares to minimize the re-projection error (4.5) for the 3D reconstruction. It takes in the 3D data points generated by linear triangulation, the image points and the camera projection matrices as the inputs, calculates the re-projection error for the cameras and then uses non-linear least squares method (2.1.3)to reduce this error by adjusting the 3D data points and projection matrices. The script generates the ***refined 3D points*** and the ***refined camera projection matrices*** as the output.

point correspondences, unrefined projection matrices, unrefined 3D data points → **refines the 3D trajectory using non-linear triangulation** → refined 3D data points, refined projection matrices,

8.3.8 generate_3D_reconstruction.py

(A)This script executes the sequence of processes required for 3D reconstruction and then finally renders the data points into a 3D plot. It imports the data for the point correspondences and the camera intrinsic parameters to execute the scripts *find_fundamental_matrix_eightpoint.py, reconstruction.py* and *bundle_adjust.py* sequentially and obtain the reconstructed trajectory. It then plots the 3D data into a 3D plot and outputs the 3D data to a .npz file(B).

point correspondences, camera intrinsic parameters, → **performs and renders 3D reconstruction** → refined 3D data points, refined projection matrices, 3D plot

8.3.9 find_suspension_point.py

(A)Fits the trajectory to a sphere surface to find out its center as the suspension point of the pendulum(8.4.6). It uses the 3D trajectory data as the input and outputs the suspension point of the pendulum to the .npz file(B) containing 3D data.

3D trajectory data → **computes the pendulum suspension point** → pendulum suspension point,

8.3.10 coordinate_transform.py

(A)Performs a rigid transformation from the camera frame to the pendulum suspension point. Using the suspension point as the system origin and the vector pointing from the suspension point to the bob in equilibrium, it computes the required ***rotation and the translation matrices*** required for the coordinate transformation (8.4.7). Finally it outputs 3D data points in the transformed coordinate to a .npz file(B).

3D data in camera coordinates → **performs rigid coordinate transform** → 3D data in changed coordinates

8.3.11 nlsq_differential.py

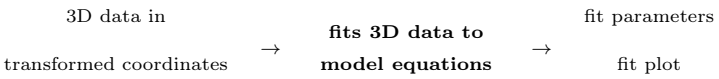
(A)Library to implement the *Newton-Gauss* (2.1.1) and the *Levenberg-Marquardt* (2.1.2) *algorithm* to perform least square fitting for differential equations. It inputs a model differential, an array of experimental data points and ***guess values*** for the fit parameters. The chosen algorithm is applied to obtain the ***best fit parameter values***. It finally outputs the best fit parameters and the ***chi-squared value*** for the fit.

8.3.12 nlsq_residual.py

(A)Library to implement the *Newton-Gauss*(2.1.1) and *Levenberg-Marquardt algorithm*(2.1.1) to perform least square fitting on a residual function. It also provides a function to solve a matrix equation using least squares. The fit functions input a model residual function and the guess value for the parameters and then the chosen algorithm is used to compute the best fit parameters iteratively. The ***matrix solver minimizes the residual*** $\boldsymbol{A}\overrightarrow{x} - \overrightarrow{b}$ as a function of the vector $\overrightarrow{x}$. It takes in the matrix $\boldsymbol{A}$ and the vector $\overrightarrow{b}$ as the inputs, computes the vector $\overrightarrow{x}$ that minimizes the residual.

8.3.13 fit_data.py

(A)(A)This script inputs the refined and transformed 3D data and implements the script nlsq_differential.py to fit the data to a modeled set of differential equations, transforms the data from the ***Cartesian coordinate system to spherical polar coordinate*** system and finally outputs the best fit parameters for the fit. It also plots the best fit curve over the experimental data points.



8.4 Procedure and Experimental Data

The experiment is performed in two different ways, once by having the system as a simple spherical pendulum, and then by inserting the magnets for the magnetic pendulum. The setup is placed in a ***wind free region*** with ***good lighting conditions***. The cameras C_{m1} and C_{m2} are clamped at their respective positions. The clamps are adjusted to keep the bob image near the center of the image plane for both C_{m1} and C_{m2}. The bob is manually brought to equilibrium position and made as still as possible. A millisecond stopwatch C_s is started and keeping its display in the frame of C_{m1}carefully so that it does not touch the pendulum suspension rod, video capture for C_{m1} is started. Now, C_s is removed and and without stopping it, is brought into the frame of C_{m2} before C_{m2} starts recording. After around a minute or so the recordings are stopped, by first stopping C_{m2} and then C_{m1}. This will give a lengthier data set for C_{m1} as compared to C_{m2}. This is necessary as will be explained later. This process is carried out for the simple pendulum case as well as the magnetic pendulum case. Now that the physical data taking process is over, the video files are named as follows -

	camera	***video file name***
spherical pendulum	C_{m1}	spherical_cam1.mp4
	C_{m2}	spherical_cam2.mp4
magnetic pendulum	C_{m1}	magnetic_cam1.mp4
	C_{m2}	magnetic_cam2.mp4

Table 6: Video files for tracking

8.4.1 Tracking pendulum bob

The HSV ranges for the color mask are set in the python script *object_ track.py*(8.3.2)(A) as given below.

- for view C_{m1}: the range is set from $(95, 50, 50)$ to $(130, 255, 255)$
- for view C_{m2}: the range is set from $(105, 17, 0)$ to $(125, 255, 200)$

	camera	***track-data file name***
spherical pendulum	C_{m1}	spherical_track_1.txt
	C_{m2}	spherical_track_2.txt
magnetic pendulum	C_{m1}	magnetic_track_1.txt
	C_{m2}	magnetic_track_2.txt

Table 7: Un-synchronized track data and time data

The video files are fed one by one first for view C_{m1} and then for view C_{m2}. A set of six data files are generated giving the position and time data for each set corresponding to C_{m1} and C_{m2}. The data files are named as follows.

8.4.2 Getting video start time estimates

The script *frame_ capture.py*(8.3.3)(A) is executed to capture the first frame of a particular video file. The stopwatch time as captured from the first frame in seconds is noted as follows:

	camera	***video first frame time***
spherical pendulum	C_{m1}	4.874
	C_{m2}	13.318
magnetic pendulum	C_{m1}	4.125
	C_{m2}	11.254

Table 8: Starting time for video sequences

8.4.3 Synchronizing the point correspondences over time

The captured track data C_{m1} and C_{m2} are not synchronized in time. The starting time for the videos give a ***rough estimate*** of the time shift between the camera video sequences. This estimate is fed into the python script *data_ synchronize.py* (6.4)(A) along with the track data for the two camera views to obtain the ***synchronized point correspondences*** and the relevant time data. Instead of taking the entire data set for analysis a set of 200 data points are used. Moreover 1500 of the initial data points are removed to cut off the portions where the bob is set to motion manually. Since the camera frame rates (60 fps) are assumed to be approximately same then from (58), $\beta = \frac{f_{c1}}{f_{c2}} = 1$. The calculated values of α for the data sets are given below:

type of system	$\boldsymbol{\alpha}$
spherical pendulum	506.646
magnetic pendulum	427.597

Table 9: Computed values of α

The synchronized point correspondences for the data sets and the time data are saved to the files named as *point_ correspondences.npz* and *time_ data.npz respectively.*

8.4.4 Camera Calibration

The cameras C_{m1} and C_{m2} are calibrated for the intrinsic parameters using the calibration video files(C). A video of an 8×6 checkerboard pattern from different angles is captured for both the cameras. The video files are then fed as inputs into the python script *camera_ calibrate.py*(8.3.1)(A) to obtain the camera intrinsic matrices. The obtained values of camera calibration matrices are as follows:

camera	$\boldsymbol{C_{m1}}$	$\boldsymbol{C_{m2}}$
intrinsic matrix ($\boldsymbol{K}$)	$\begin{pmatrix} 1484.0 & 0.0 & 534.4 \\ 0.0 & 1485.7 & 957.1 \\ 0.0 & 0.0 & 1.0 \end{pmatrix}$	$\begin{pmatrix} 1493.0 & 0.0 & 537.4 \\ 0.0 & 1496.7 & 959.9 \\ 0.0 & 0.0 & 1.0 \end{pmatrix}$

Table 10: Camera intrinsic matrices

The intrinsic matrices are saved to data files named as *camera1_ intrinsic_ matrix1920.npz*(B) and *camera2_ intrinsic_ matrix1920.npz*(B).

8.4.5 Estimating the Fundamental matrix, 3D reconstruction and Bundle Adjustment

The *generate_ 3D_ reconstruction.py* (8.3.8)(A) script is executed to generate the 3D reconstruction from the synchronized point correspondences and the camera intrinsic matrices. The fundamental matrices for the data sets are given below:

	fundamental matrix
spherical pendulum	$\begin{pmatrix} \text{-1.41e-6} & \text{+2.79e-7} & \text{+2.52e-3} \\ \text{+4.15e-7} & \text{+4.10e-6} & \text{-6.31e-3} \\ \text{+8.99e-4} & \text{+6.24e-3} & \text{-1.02e+1} \end{pmatrix}$
magnetic pendulum	$\begin{pmatrix} \text{+1.15e-3} & \text{-1.99e-4} & \text{-2.52e+0} \\ \text{-4.49e-4} & \text{-2.18e-3} & \text{+3.83e+0} \\ \text{-4.53e-1} & \text{-1.24e+0} & \text{+2.65e+3} \end{pmatrix}$

Table 11: Camera Fundamental matrices

The fundamental matrices are fed into *reconstruction.py* (8.3.6)(A) to obtain the 3D reconstructed points using *linear triangulation*. The 3D points are refined by bundle adjustment using the script *bundle_ adjust.py* (8.3.7)(A) and the optimized values of the 3D points are saved to a file named *3D_ data.npz*(B). The total re-projection error for the 3D reconstruction are listed in the table below.

	total re-projection error
spherical pendulum	15.808 pixel^2
magnetic pendulum	9.506 pixel^2

8.4.6 Estimating the suspension point

The pendulum trajectory obtained from 3D reconstruction is fit to the surface of a sphere to obtain the radius and the center of the sphere as the suspension point. The general equation of a sphere of radius r and center (x_0, y_0, z_0) in Cartesian coordinates is given by,

$$(x - x_0)^2 + (y - y_0)^2 + (z - z_0)^2 = r^2$$

$$\Rightarrow x^2 + y^2 + z^2 = 2xx_0 + 2yy_0 + 2zz_0 + r^2 - x_0^2 - y_0^2 - z_0^2 \tag{109}$$

This equation can be written in the form of a matrix equation $\boldsymbol{A}\overrightarrow{x} = \overrightarrow{b}$, where the matrices and the vectors are given by:

$$\boldsymbol{A} = \begin{bmatrix} 2x_1 & 2y_1 & 2z_1 & 1 \\ 2x_2 & 2y_2 & 2z_2 & 1 \\ \vdots & \vdots & \vdots & \vdots \\ 2x_n & 2y_n & 2z_n & 1 \end{bmatrix} \quad \overrightarrow{x} = \begin{bmatrix} x_0 \\ y_0 \\ z_0 \\ r^2 - x_0^2 - y_0^2 - z_0^2 \end{bmatrix} \quad \overrightarrow{b} = \begin{bmatrix} x_1^2 + y_1^2 + z_1^2 \\ x_2^2 + y_2^2 + z_2^2 \\ \vdots \\ x_n^2 + y_n^2 + z_n^2 \end{bmatrix} \tag{110}$$

The above equation can be solved to obtain the least squares solution for $\overrightarrow{x}$. The python function *solve_mat_nlsq()* under the script *nlsq_residual.py* (8.3.12)(A) precisely does this to a matrix equation. The obtained value of $\overrightarrow{x}$ then gives the sphere center and the sphere radius. The python script *find_suspension_point.py* (8.3.9)(A) takes in the trajectory data as input and append to it the coordinate for the suspension point. The computed values of the pendulum suspension points are listed below:

system	***suspension point***
spherical pendulum	[-0.0421 -0.1937 -0.0033]^T
magnetic pendulum	[0.1975 -0.0060 -0.1021]^T

Table 12: Computed suspension point

8.4.7 Coordinate transformation

The resulting 3D data is computed in the frame of the camera. A coordinate transformation is required which makes the new coordinate system,

- with its origin at the suspension point of the pendulum
- and its z-axis pointing in the direction of the vector ***pointing from pendulum bob under gravitational equilibrium*** (only) to the pendulum suspension point

This is equivalent to a translation $\overrightarrow{T}_{coord}$ and a rotation $\boldsymbol{R}_{coord}$ of all the points. The translation is obtained by simply subtracting $\overrightarrow{T}_{coord}$ from all the points. The rotation $\overrightarrow{\boldsymbol{R}}_{coord}$ requires a more formal treatment.

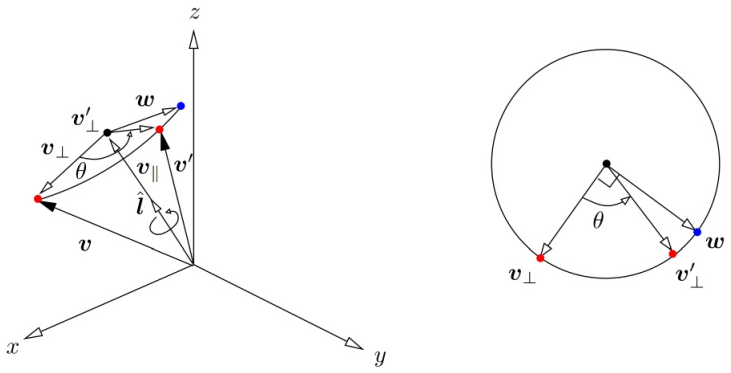


Figure 27: Vector v is rotated to v' about the axis $\hat{l}$

Consider a vector undergoing a rotation by an arbitrary angle θ about some arbitrarily oriented axis passing through the origin. Let $\hat{l} = (l_x, l_y, l_z)^T$ be the ***unit vector along the rotation axis***, $\overrightarrow{v}$

represent any arbitrary vector in the coordinate system and $\overrightarrow{v}'$ represent the corresponding rotated vector after the required rotation. Decomposing $\hat{v}$ into two components, one ***parallel*** to $\hat{l}$ and the other ***perpendicular*** to $\hat{l}$, we have:

$$\overrightarrow{v}_{\parallel} = \left(\overrightarrow{v} \cdot \hat{l}\right) \hat{l} \tag{111}$$

$$\overrightarrow{v}_{\perp} = \overrightarrow{v} - \left(\overrightarrow{v} \cdot \hat{l}\right) \hat{l} \tag{112}$$

Figure (27) shows that the component $\overrightarrow{v}_{\parallel}$ ***is not affected by rotation***. Thus the only required component is $\overrightarrow{v}'_{\perp}$ that results from applying some rotation to $\overrightarrow{v}_{\perp}$. Consider the plane Π as shown in (27). The plane is spanned by the orthogonal vectors: $\overrightarrow{v}_{\perp}$ and $\overrightarrow{w}$, where $\overrightarrow{w}$ is given by:

$$\begin{aligned}
\overrightarrow{w} &= \hat{l} \times \overrightarrow{v}_{\perp} \\
&= \hat{l} \times \left(\overrightarrow{v} - \left(\overrightarrow{v} \cdot \hat{l}\right) \hat{l}\right) \\
&= \hat{l} \times \overrightarrow{v}
\end{aligned}$$

The vector $\overrightarrow{v}'_{\perp}$ lying in the plane Π can be represented in terms of the orthogonal vectors $\overrightarrow{v}_{\perp}$ and $\overrightarrow{w}$.

$$\overrightarrow{v}'_{\perp} = \overrightarrow{v}_{\perp} \cos\theta + \overrightarrow{w} \sin\theta \tag{113}$$

Thus the rotated vector $\overrightarrow{v}'$obtained by rotating $\overrightarrow{v}$ about $\hat{l}$ by an arbitrary angle θ is given by:

$$\begin{aligned}
\overrightarrow{v}' &= \overrightarrow{v}_{\perp} + \overrightarrow{v}_{\parallel} \\
&= \left(\overrightarrow{v} \cdot \hat{l}\right) \hat{l} + \overrightarrow{v}_{\perp} \cos\theta + \overrightarrow{w} \sin\theta \\
&= \left(\overrightarrow{v} \cdot \hat{l}\right) \hat{l} + \left(\overrightarrow{v} - \left(\overrightarrow{v} \cdot \hat{l}\right) \hat{l}\right) \cos\theta + \left(\hat{l} \times \overrightarrow{v}\right) \sin\theta \\
&= \overrightarrow{v} \cos\theta + \left(\hat{l} \times \overrightarrow{v}\right) \sin\theta + \hat{l} \left(\hat{l} \cdot \overrightarrow{v}\right) (1 - \cos\theta)
\end{aligned} \tag{114}$$

In matrix notation the rotation can be expressed in the form,

$$\overrightarrow{v}' = \left(\boldsymbol{R}_a + (\sin\theta)\, \boldsymbol{R}_b + (1 - \cos\theta)\, \boldsymbol{R}_c\right) \overrightarrow{v} = \boldsymbol{R}_{coord}\, \overrightarrow{v} \tag{115}$$

where the matrices $\boldsymbol{R}_a$, $\boldsymbol{R}_b$ and $\boldsymbol{R}_c$ are given by,

$$\boldsymbol{R}_a = \begin{bmatrix} \cos\theta & 0 & 0 \\ 0 & \cos\theta & 0 \\ 0 & 0 & \cos\theta \end{bmatrix} \quad \boldsymbol{R}_b = \begin{bmatrix} 0 & -l_z & l_y \\ l_z & 0 & -l_x \\ -l_y & l_x & 0 \end{bmatrix} \quad \boldsymbol{R}_c = \begin{bmatrix} l_x^2 & l_x l_y & l_x l_z \\ l_y l_x & l_y^2 & l_y l_z \\ l_z l_x & l_z l_y & l_z^2 \end{bmatrix}$$

If $\overrightarrow{v}$ represents an arbitrary 3D point in the camera frame, $(x_{sus}, y_{sus}, z_{sus})$ represents the pendulum suspension point and $(x_{equib}, y_{equib}, z_{equib})$ represents the ***gravitational equilibrium*** position of the pendulum bob, then we have, the unit vector along the axis of rotation given by,

$$\hat{l} = \frac{\overrightarrow{l}}{\left|\overrightarrow{l}\right|} \quad with \quad \overrightarrow{l} = \overrightarrow{z}_{pend} \times \overrightarrow{z}_{cam} \tag{116}$$

where $\overrightarrow{z}_{pend} = \begin{bmatrix} x_{sus} - x_{equib} \\ y_{sus} - y_{equib} \\ z_{sus} - z_{equib} \end{bmatrix}$ and $\overrightarrow{z}_{cam} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$ and the angle of rotation θ about the axis of rotation is,

$$\theta = \cos^{-1}\left(\frac{\overrightarrow{z}_{pend} \cdot \overrightarrow{z}_{cam}}{|\overrightarrow{z}_{pend}|\, |\overrightarrow{z}_{cam}|}\right) \tag{117}$$

Also the translation $\overrightarrow{T}_{coord}$ is given by the vector,

$$\overrightarrow{T}_{coord} = \begin{bmatrix} x_{sus} \\ y_{sus} \\ z_{sus} \end{bmatrix}$$

Thus the net transformation can be represented by the matrix equation,

$$\overrightarrow{v}' = \boldsymbol{R}_{coord} \left(\overrightarrow{v} - \overrightarrow{T}_{coord} \right) \tag{118}$$

The python script *coordinate_ transform.py*(8.3.10)(A) does this transformation and outputs the transformed 3D coordinates to a file named *XYZ_ data.npz*. The rotation $\boldsymbol{R}_{coord}$ and the translation $\overrightarrow{T}_{coord}$ for the data sets are given in the table below:

system	***rotation and translation***
spherical pendulum	$\begin{pmatrix} +0.8827 & +0.4697 & -0.0146 \\ +0.4697 & -0.8809 & 0.0583 \\ +0.0145 & -0.0583 & -0.9982 \end{pmatrix}$ $\begin{bmatrix} +0.0421 & 0.1937 & 0.0033 \end{bmatrix}^T$
magnetic pendulum	$\begin{pmatrix} -0.9654 & -0.2128 & -0.1507 \\ -0.2128 & +0.9769 & -0.0163 \\ +0.1507 & +0.0163 & -0.9884 \end{pmatrix}$ $\begin{bmatrix} -0.1974 & 0.0060 & 0.1021 \end{bmatrix}^T$

Table 13: Computed rotations $\boldsymbol{R}_{coord}$ and translations $\overrightarrow{T}$

The final 3D trajectory for the spherical pendulum and magnetic pendulum obtained after linear triangulation, bundle adjustment and coordinate transform is shown in the figures(28)(29).

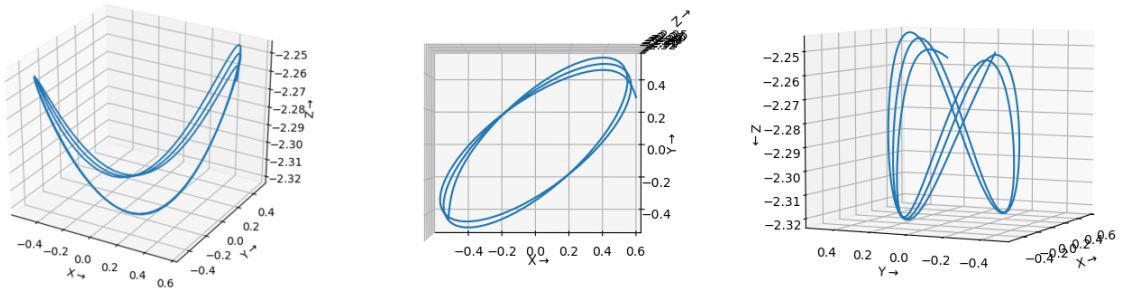


Figure 28: Refined and coordinate transformed 3D data for spherical pendulum

8.4.8 Fitting trajectory data to the modeled equations of motion

The trajectory data from the the from file *XYZ_ data.npz*(B) and the time data from the file *time_ data.npz*(B) is fit to the modeled set of differential equations using non linear least squares. But the modeled equations are in spherical coordinates. So a transformation is required to ***obtain the data in spherical coordinates***. The transformation is given by the equations -

$$\theta = \arctan_2 (Y, X) \tag{119}$$

$$\phi = \arccos (Z/L) \tag{120}$$

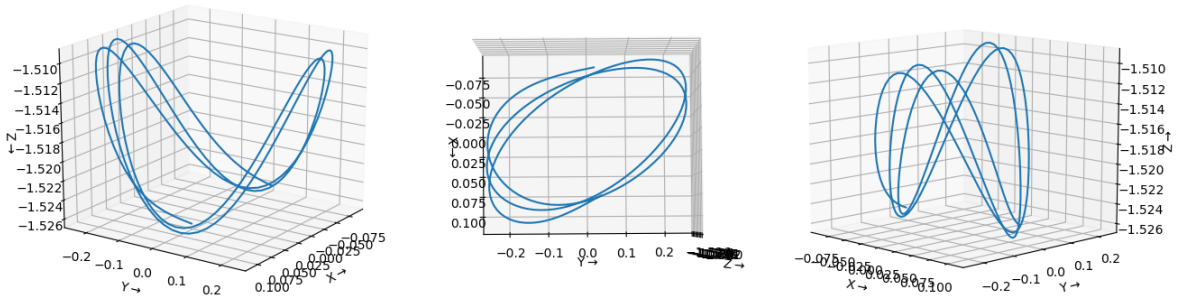


Figure 29: Refined and coordinate transformed 3D data for magnetic pendulum

where (X, Y, Z) are the 3D reconstructed points and L is the estimated pendulum length from fitting trajectory to sphere surface. The script *fit_ data.py* (8.3.13) is executed to carry out the transformation and perform the fit . The modeled differential equations of motion for the spherical pendulum are(7.1.1):

$$\ddot{\phi} = \sin\phi\cos\phi\dot{\theta}^2 + \frac{3g}{2L}\left(\frac{2m_{bob}+m_{rod}}{3m_{bob}+m_{rod}}\right)\sin\phi - \alpha L\dot{\phi}$$
$$\ddot{\theta} = -2\cot\phi\dot{\theta}\dot{\phi} - \alpha L\sin(\phi)\,\dot{\theta}$$

Any combination of the motion parameters in the differential equation in the form of ***products or fractions cannot be determined uniquely*** by the fit. As such the fit parameters for the system are redefined as follows:

$$F_{p1} = \frac{3g}{2L}\left(\frac{2m_{bob}+m_{rod}}{3m_{bob}+m_{rod}}\right) \tag{121}$$
$$F_{p2} = \alpha L \tag{122}$$

Thus the ***equations of motion*** to which the data is fit can be rewritten in the form,

$$\ddot{\phi} = \sin\phi\cos\phi\dot{\theta}^2 + F_{p1}\sin\phi - F_{p2}\dot{\phi} \tag{123}$$
$$\ddot{\theta} = -2\cot\phi\dot{\theta}\dot{\phi} - F_{p2}\sin(\phi)\,\dot{\theta} \tag{124}$$

Similarly the equations of motion for the ***magnetic pendulum*** are given by,

$$\ddot{\phi} = \sin\phi\cos\phi\dot{\theta}^2 + F_{p1}\sin\phi - F_{p2}\dot{\phi} - \frac{F_{p3}}{\left(m_{bob}+\frac{m_{rod}}{3}\right)}V_1(\theta,\phi) \tag{125}$$
$$\ddot{\theta} = -2\cot\phi\dot{\theta}\dot{\phi} - F_{p2}\sin\phi\dot{\theta} - \frac{F_{p3}}{\sin^2\phi L^2\left(m_{bob}+\frac{m_{rod}}{3}\right)}V_2(\theta,\phi) \tag{126}$$

where, $F_{p3}V_1(\theta,\phi) = \frac{\partial U_m}{\partial\phi}$, $F_{p3}V_2(\theta,\phi) = \frac{\partial U_m}{\partial\theta}$ with $F_{p3} = m_2 m_2$ and F_{p1}, F_{p2} are same as in the case of spherical pendulum. The other parameters of the equations of motion cannot be estimated uniquely. These parameters are either in the form of products or fractions with each other and would require multi-parametric fitting techniques, which is beyond the scope of this project. The values of the constants provided for the fit are given in the table below:

constant	*value*
S	47 cm
L	43.2 cm
X_0	0.01 cm
Y_0	0.02 cm
m_{bob}	33.17 g
m_{rod}	6.70 g

8.5 Data Analysis

8.5.1 Spherical Pendulum

The best fit trajectory of the pendulum along with the data points are shown in the figure (30).

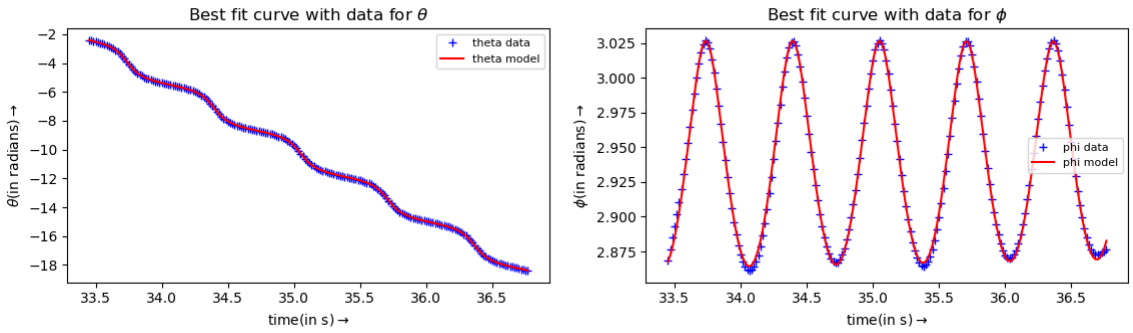


Figure 30: Spherical Pendulum Data fit to model equations using Levenberg-Marquardt method$\left(\chi^2 = 8.65 \times 10^{-4}\text{rad}^2\right)$

The table below lists the estimated value of the fit parameters obtained from the least square fit:

fit parameter	*guess value*	*fitted value*
F_{p1}	20.0	23.1132
F_{p2}	0.05	0.0273
θ_0	-2.0	-2.4168
ϕ_0	2.0	2.8677
$\dot{\theta}_0$	-2.0	-2.0887
$\dot{\phi}_0$	0.5	0.2204

Table 14: Fit Parameters

- ***Determination of g and α from the fit parameters*** : Using the measured values of m_{bob}, m_{rod} and L, the values of g and α can be computed from the fit parameters F_{p1} and F_{p2} (123)(124) respectively.

$$g = \frac{2F_{p1}L}{3}\left(\frac{3m_{bob} + m_{rod}}{2m_{bob} + m_{rod}}\right) = \mathbf{9.68}\text{m s}^{\mathbf{-2}}$$

And the ***maximum relative error*** for the measurement is obtained to be,

$$\left.\frac{\delta g}{g}\right|_{max} = \left|\frac{\delta L}{L}\right| + \left|\frac{3\delta m_{bob} + \delta m_{rod}}{3m_{bob} + m_{rod}}\right| + \left|\frac{2\delta m_{bob} + \delta m_{rod}}{2m_{bob} + m_{rod}}\right| = 0.315\%$$

Similarly, the damping factor α is calculated as,

$$\alpha = \frac{F_{p2}}{L} = \mathbf{0.063}\mathrm{m}^{\mathbf{-1}}\mathrm{s}^{\mathbf{-1}}$$

with the ***maximum relative error*** given by,

$$\left.\frac{\delta\alpha}{\alpha}\right|_{max} = \left|\frac{\delta L}{L}\right| = 0.236\%$$

It is to be noted that the error in the estimation of the fit parameters have not been taken into account here. A better and a more detailed analysis of the measurement error can be done by computing the error in parameter estimation using the reconstruction error.

- ***Variations in the angular momentum*** l_z ***and total energy*** E ***of the system*** : The z-component of the angular momentum of the pendulum l_z is given by the expression (80),

$$l_z = M_1 L^2 \sin^2\phi\dot{\theta}$$

$\dot{\theta}$ for the i^{th} data is obtained numerically by taking finite central difference,

$$\dot{\theta}_i = \frac{\theta_{i+1} - \theta_{i-1}}{2\delta t} \quad \text{where, } \delta t = f_c = (1/60)\,\mathrm{s}$$

The plot for the angular momentum of the system is as shown in figure (31). Similarly, the total energy of the system, is given by the expression (76),

$$E = \frac{1}{2}M_1 L^2\dot{\phi}^2 + \frac{l_z^2}{2M_1 L^2\sin^2\phi} + M_2 gL\cos\phi$$

Here again $\dot{\phi}$ for the i^{th} data is obtained numerically by taking finite central difference,

$$\dot{\phi}_i = \frac{\phi_{i+1} - \phi_{i-1}}{2\delta t} \quad \text{where, } \delta t = f_c = (1/60)\,\mathrm{s}$$

The plot for the total energy of the system is as shown in figure(31)
The gradual decrease in the energy and magnitude of angular momentum of the system shows the

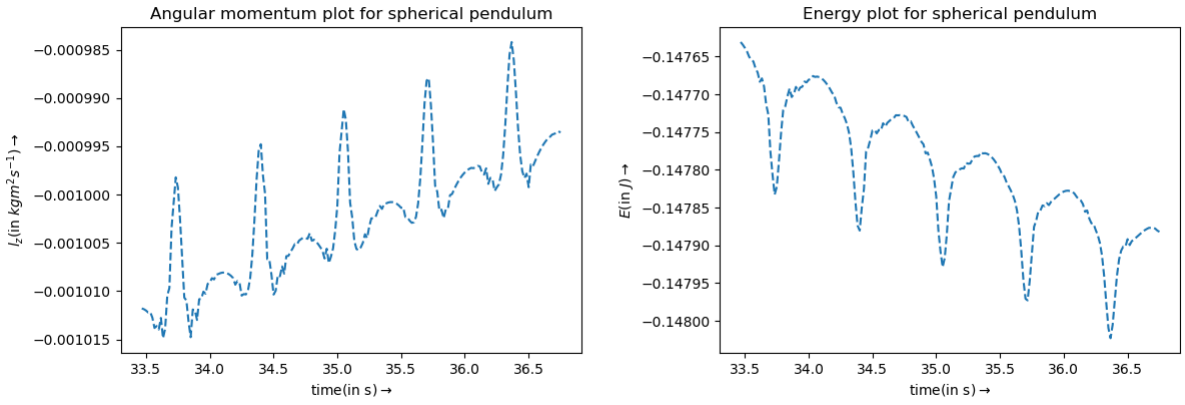


Figure 31: Variation of l_z and E with time for the experimental data

presence of dissipative forces. The sudden humps along the energy and the momentum plots are the result of error in the numerically calculated values of $\dot{\theta}$ and $\dot{\phi}$.

- ***Trajectory of the spherical pendulum neglecting damping effects*** : The expression for the polar coordinate θ as a function of the azimuthal coordinate $\phi(85)$ is given by,

$$\theta = \theta_{\text{init}} + \frac{l_z}{L\sqrt{2M_1}} \int_{\phi_{\text{init}}}^{\phi_{\text{final}}} \frac{1}{\sin^2\phi\sqrt{[E - U_{\text{eff}}(\phi)]}} d\phi$$

Since we are neglecting the effects of damping in this case, the total energy and angular momentum of the system is assumed to be constant and is computed by taking their mean values over time, i.e.,

$$E \equiv \bar{E} = \frac{1}{N}\sum_{i=1}^{N} E_i \quad l_z \equiv \bar{l}_z = \frac{1}{N}\sum_{i=1}^{N} l_{zi}$$

where N is the total number of data points. The value of theta is then computer for each θ_i with θ_{init}being θ_0. The expression for $U_{\text{eff}}(\phi)$ is given by,

$$U_{\text{eff}} = \frac{l_z^2}{2M_1L^2\sin^2\phi} + M_2gL\cos\phi$$

The plot in figure show the θ values obtained experimentally along with the values obtained using the expression (85).

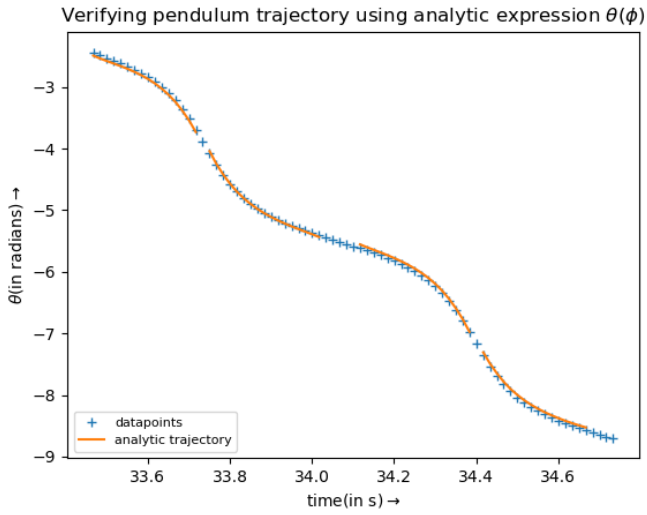


Figure 32: Plot showing the experimentally obtained data points for θ and the analytically obatined θ

- ϕ_{min} ***and*** ϕ_{max} ***for oscillating*** ϕ : Neglecting the effects of damping, the roots of the expression (78),

$$y(\cos\phi) = 2M_2M_1gL^3\left(\cos\phi - \cos^3\phi\right) - 2M_1EL^2\left(1 - \cos^2\phi\right) + l_z^2$$

give the ***maximum and minimum*** values of ϕ between which it oscillates. Assuming $E \equiv \bar{E}$ and $l_z \equiv \bar{l}_z$, the roots for the expression can be obtained graphically by finding out the values of $\cos\phi$ where y touches the x-axis. The plot for y as a function of cosϕ in the range of ϕ=($^5\pi/_6, \pi$) is shown in figure (33),

The plot shows, the curve intersecting the x-axis at two points. The corresponding values of ϕ_{min} and ϕ_{max} are computed as follows:

$$\phi_{max} = \cos^{-1}(-0.9934) = \mathbf{3.0266rad}$$
$$\phi_{min} = \cos^{-1}(-0.9627) = \mathbf{2.8676}\text{ rad}$$

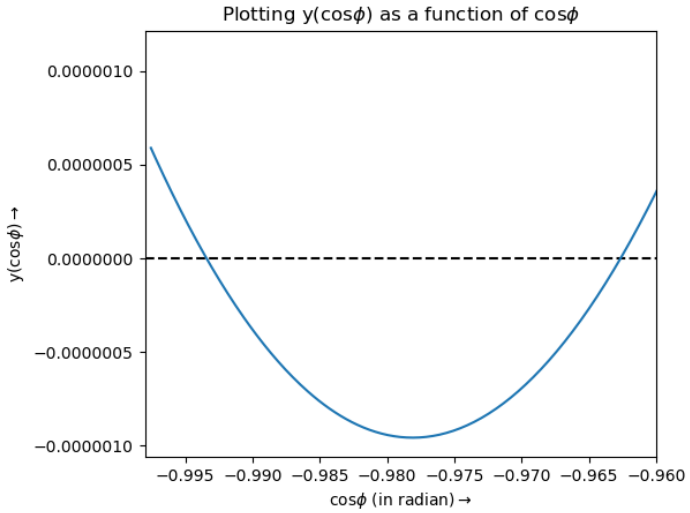


Figure 33: The roots for $y\left(\cos\phi\right)$ give the values for ϕ_{min} and ϕ_{max}

The time series plot (34) for ϕ data shows that the minimum and maximum values of oscillating ϕ (as shown in the table below) are sufficiently close $\left(\pm 10^{-2}\text{ rad}\right)$ to analytic results .

obs. no.	***maxima (rad)***	***minima (rad)***
1	3.0268	2.8613
2	3.0264	2.8668
3	3.0269	2.8645
4	3.0265	2.8698
5	3.0272	2.8727

Table 15: Table for experimental ϕ_{min} and ϕ_{max}

- ***Time period*** T_ϕ ***for the spherical pendulum*** : The time period T_ϕ can be estimated using the expression,

$$T_\phi = 2\sqrt{M_1}L\int_{\phi_{min}}^{\phi_{max}} \frac{d\phi}{\sqrt{2\left[E - U_{\text{eff}}\left(\phi\right)\right]}}$$

where ϕ_{min} and ϕ_{max} are the analytically obtained results. Like the preceding cases, the total energy and angular momentum are given as $E \equiv \bar{E}$ and $l_z \equiv \bar{l}_z$. Then the value of T_ϕ is computed to be,

$$T_\phi = \mathbf{0.658}\text{s}$$

The time interval between two subsequent maxima of ϕ in its time series plot (34) matches well with the computed result as event from the table below.

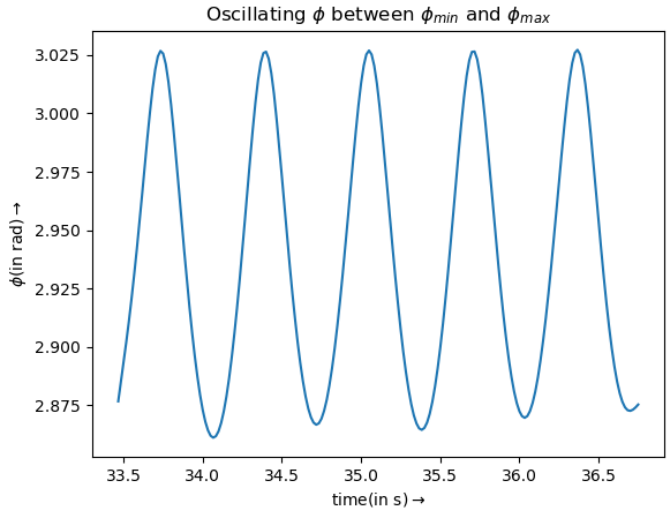


Figure 34: Time series plot for experimental ϕ

obs. no.	T_ϕ ***(s)***
1	0.6608
2	0.6544
3	0.6479
4	0.6608

Table 16: Table for experimental T_ϕ

- ***Discussion :*** The following conclusions can be drawn from the above obtained results:
  - The re-projection error for each 3D point comes out to be 0.079pixel^2, which is sufficiently good for the purpose.
  - The value of g obtained from the value of fit parameters is a ***direct measure of the accuracy*** of the reconstruction process. The true value of g at the experimental location is obtained using an online application, and is 9.78 m s^{-2} $\left(\text{latitude} = 22.6196^\circ,\ \text{longitude} = 88.3920^\circ\right)$. Considering the measurement and reconstruction error, this is a good value for the measured g.
  - The ***gradual decrease in total energy*** and in the ***magnitude of the angular momentum*** of the system shows the effect of the damping effect on the motion of pendulum.
  - The analytically obtained trajectory using the initial conditions and the constants of the system, shows good correspondence with the experimentally obtained data (32). The ***discontinuities in the analytic data*** is due to the integral(85) blowing up under certain conditions.
  - Moreover, the values of ϕ_{min} and ϕ_{max} obtained graphically, and the computed T_ϕ matches well with the data especially considering the fact that the ***damping effects have been neglected***. In case, the damping effects are taken into account, we obtain multiple solutions for ϕ_{max} and ϕ_{min} and hence for T_ϕ.

Thus it can be concluded that the motion of the spherical pendulum can be well established by the modeled set of differential equations.

8.5.2 Magnetic Pendulum

The best fit trajectory for the magnetic pendulum along with the data points are as shown in the figures.

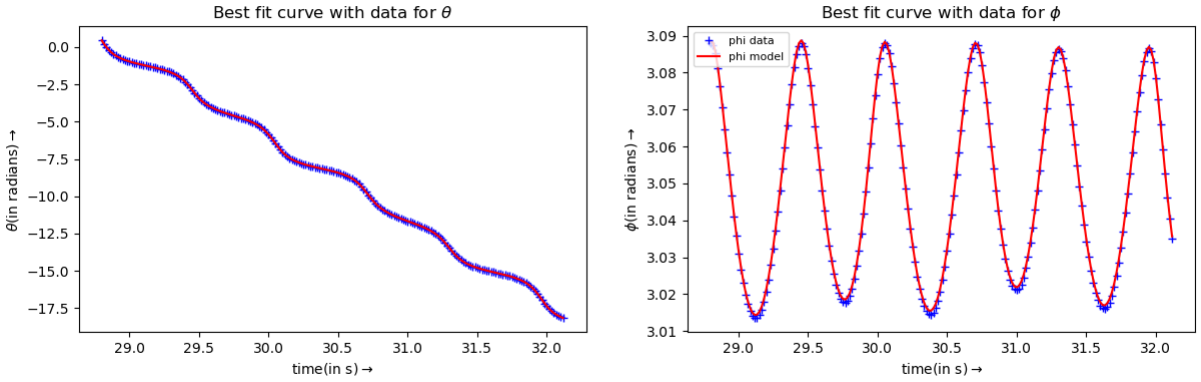


Figure 35: Magnetic Pendulum Data fit to model equations using Levenberg-Marquardt method $\left(\chi^2 = 1.23 \times 10^{-4}\text{rad}^2\right)$

The table below lists the estimated value of the fit parameters obtained from the least square fit.

fit parameter	***guess value***	***fitted value***
F_{p1}	20.0	23.789
F_{p2}	0.04	0.0302
F_{p3}	0.30	0.3086
θ_0	1.0	0.4281
ϕ_0	1.0	3.0862
$\dot{\theta}_0$	-8.0	-13.9310
$\dot{\phi}_0$	1.0	0.0310

Table 17: Fit Parameters

- ***Determination of g,α dipole moment m from the fit parameters*** : Using the measured values of m_{bob}, m_{rod} and L, the values of g and α can be computed from the fit parameters F_{p1} and F_{p2} (123)(124) respectively.

$$g = \frac{2F_{p1}L}{3}\left(\frac{3m_{bob} + m_{rod}}{2m_{bob} + m_{rod}}\right) = \mathbf{9.75}\text{m/s}^{\mathbf{2}}$$

with maximum relative error of 0.315%. Similarly, we have the damping factor α calculated as,

$$\alpha = \frac{F_{p2}}{L} = \mathbf{0.071}\text{m}^{\mathbf{-1}}\text{s}^{\mathbf{-1}}$$

with max relative error of 0.236%. Assuming the dipole moment for the two magnets to be ***identical***, the value of the dipole moment $\boldsymbol{m}$ for each magnet can be computed as,

$$m = \sqrt{F_{p3}} = \mathbf{0.56}\text{A m}^{\mathbf{-2}}$$

- ***Variation of the total energy E of the system*** : The total energy of the magnetic pendulum is given by the expression,

$$E = \frac{L^2}{2}\left(m_{bob} + \frac{m_{rod}}{3}\right)\left(\dot{\phi}^2 + \sin^2\phi\dot{\theta}^2\right) + \left(m_{bob} + \frac{m_{rod}}{2}\right)gL\cos\phi + U_m$$

where $\dot{\theta}$ and $\dot{\phi}$ for the i^{th} data is obtained numerically by taking finite central difference,

$$\dot{\theta}_i = \frac{\theta_{i+1} - \theta_{i-1}}{2\delta t} \quad \dot{\phi}_i = \frac{\theta_{i+1} - \theta_{i-1}}{2\delta t}$$

where, $\delta t = f_c = (1/60)\,\text{s}$. The plot for the total energy of the system is plotted in the figure(36), The total energy of the system decreases gradually as in the case of spherical pendulum due to the

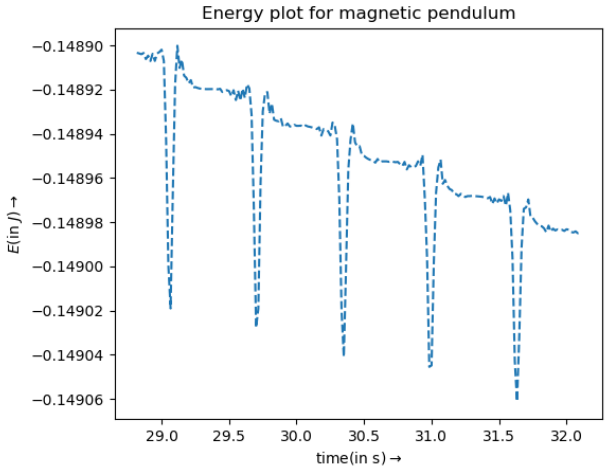


Figure 36: Variation of the total energy E of the system with time

effect of the dissipative forces. The sudden spikes in the energy plot (36) is due to the error in the numerically calculated values of $\dot{\theta}$and $\dot{\phi}$.

- ***Discussion*** : The experimental data leads to the following conclusions:
  - A ***low value of the re-projection error*** per 3D data point $\left(= 0.0475\text{pixel}^2\right)$ suggests a good 3D reconstruction for the system trajectory.
  - The obtained value of g is consistent with the value obtained using the spherical pendulum case and matches well with its true value at the experimental location.
  - As in the case of a spherical pendulum, the gradual decrease in the total energy E, ***shows the effect of damping*** on the pendulum motion.
  - The time series plot for θ and ϕ shows the ***perturbation in the trajectory*** of a spherical pendulum under magnetic influence.
  - Under the applied magnetic influence, the motion appears more or less ***periodic*** in nature. A greater magnetic influence could not be applied by increasing the number of neodymium magnets due to the fact that the stainless steel suspension rod ***considered to be stiff starts vibrating*** under the aggressive potential dips of the magnet. As such the approximation that the rod is completely stiff ***fails*** and the equations no longer apply to the system. However with a suitable setup for the pendulum, the chaotic nature of the system can be studied using the same procedure.

The experimental results show that the data matches well with the modeled set of equations obtained for the magnetic pendulum. The parameters in the equation which are in the form of ***products or fractions*** with each other, could not be uniquely determined, using ordinary fitting and requires ***multi-parameter fitting***. As such good estimate values for the constants are supplied for fitting the data to the equations.

8.6 Error analysis

The stereo vision based 3D reconstruction of a dynamic scene is sensitive to several types of errors. Since the uncertainty is a critical issue when using the trajectory data for the system analysis, the estimation and the assessment of errors in the final reconstruction needs to be carried out carefully. In general the errors in obtaining the desired 3D information is estimated using expressions determined in terms of the system parameters and the error sources likely to be observed. The primary sources of error are related to camera calibration and camera modeling. The 3D reconstruction results are also majorly affected by correspondence errors occurring during the tracking and the synchronization process. The errors are magnified during the standard triangulation process, resulting in reconstruction discrepancies. Bundle adjustment reduces these errors to a great extent using minimization techniques. The persisting error can still lead to inconsistent results for the experiments and must be accounted for. Similarly the uncertainties due to the least count of the measuring instruments are to be taken into account while calculating the various motion parameters. Taking into account all the sources of error will help gain a better understanding of the deviation of results from the expected values.

9 Conclusion

The experimental results show that the data obtained from the reconstruction are in good agreement with the analytic results. The values of g obtained for both the cases support the fact that the data generated by the proposed sequence of steps can satisfactorily describe the dynamics of the system. It has been shown that precise trajectory data can be obtained using a primitive setup and minimal hardware, without any external dependencies. However, certain improvisations can be made to improve the process. Increasing the number of views for the scene by using more number of cameras can lead to a much better 3D reconstruction, though the mathematics gets more complicated in that case. Cameras with a higher frame rate would yield a denser data-set and can detect abrupt changes in the motion. The method suggested in the project for synchronizing the position data over time assumes that the time interval between the successive frames captured by the cameras remains constant. However, this is not the case in general on account of hardware noise. A more general treatment can be done to synchronize the data frame by frame over time using stereo geometry. Another approach to go about the synchronization problem is to use external hardware for camera shutter synchronization. Multi-parameter estimation for the case of magnetic pendulum, is a major challenge and would require a much more involved treatment using machine learning.
Any physical system in nature can be modeled using a set of differential equations. Whether be linear or non-linear, the method can be generalized to study these systems with the mentioned improvements.

References

[1] Longuet-Higgins, H. C. (1981), *A computer algorithm for reconstructing a scene from two projections*

[2] Richard Hartley (2004), *Multiple View Geometry in Computer Vision*

[3] Avidan, S., and Shashua, A. (2000). *Trajectory triangulation: 3D reconstruction of moving points from a monocular image sequence. IEEE Transactions on Pattern Analysis and Machine Intelligence*

[4] Douglas G. Bates and Donald G. Watts (1988), *Nonlinear Regression Analysis and Its Applications*

[5] George A. F. Seber, C.J. Wild (1989), *Nonlinear Regression*

[6] Henri P. Gavin (2020), *The Levenberg-Marquardt algorithm for nonlinear least squares curve-fitting problems*

[7] K. Levenberg (1944), *A Method for the Solution of Certain Non-Linear Problems in Least Squares*

[8] D.W. Marquardt (1963), *An algorithm for least-squares estimation of nonlinear parameters*

[9] A. Bjorck (1966), *Numerical methods for least squares problems*

[10] Peter Mills (2017), *Singular Value Decomposition (SVD) Tutorial: Applications, Examples, Exercises,* *https://blog.statsbot.co/singular-value-decomposition-tutorial-52c695315254*

[11] K. F. Riley, M.P. Hobson and S.J. Bence (2006), *Mathematical Method for Physics and Engineering*

[12] David J. Griffiths (2017), *Introduction to Electrodynamics*

[13] Suparna Roychowdhury (2020), *Integration and Differentiation*

[14] William H. Press (2002), *Numerical Recipes in C: The Art of Scientific Computing*

[15] Kenji Hata and Silvio Savarese, *Course Notes: Camera Models, Epipolar Geometry, Stereo Systems* *https://web.stanford.edu/class/cs231a/course_notes.html*

[16] William Hoff (2012), *Lecture Series on Epipolar geometry,* *https://www.youtube.com/watch?v=skaQfPQFSyY&list=PL4B3F8D4A5CAD8DA3*

[17] Bill Triggs, *Estimating the Fundamental Matrix,* https://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/MOHR_TRIGGS/node50.html

[18] Adrian Rosebrock (2015), Ball Tracking using OpenCV

[19] Cheng Lei and Yee-Hing Yang (2006), *Trifocal tenson based multiple video synchronization with subframe optimization*

[20] A. Elhayek, C. Stoll , N. Hasler, K. I. Kim, H. P. Seidel and C. Theobalt (2012), *Spatio Temporal motion tracking with unsynchronized cameras.*

[21] Cenek Albl, *On the Two-View Geometry of Unsynchronized Cameras*

[22] R. G. Takwale and P. S. Puranik (1979), *Introduction to Classical Mechanics*

[23] John R. Taylor (2005), *Classical Mechanics*

[24] Richard Fitzpatrick (2011), Spherical Pendulum, http://farside.ph.utexas.edu/teaching/336k/Newtonhtml/node82.html

[25] John D. Jackson (1962), Classical Electrodynamics

[26] James Christian and Holly Middleton-Spencer, *Chaos in the Magnetic Pendulum*, https://ima.org.uk/13908/chaos-in-the-magnetic-pendulum/

A Python Scripts

This section includes the main python scripts which were used for the software analysis part of the process. The rest of the scripts are given as links at the end of the section.

camera_calibrate.py

```
import cv2 as cv
import numpy as np
import argparse
import imutils
import os

#checkerboard matrix for calibration
CHECKERBOARD=(8,6)

#termination criteria
criteria=(cv.TERM_CRITERIA_EPS+cv.TERM_CRITERIA_MAX_ITER,30,0.001)
#cv.TERM_CRITERIA_EPS - stop the algorithm iteration if
#specified accuracy, epsilon, is reached.
#cv.TERM_CRITERIA_MAX_ITER - stop the algorithm after
#the specified number of iterations, max_iter.
#cv.TERM_CRITERIA_EPS + cv.TERM_CRITERIA_MAX_ITER -
#stop the iteration when any of the above condition is met.

points3D=[]#object points in real world space
points2D=[]#image points in image plane.

objectp3D=np.zeros((1,CHECKERBOARD[0]*CHECKERBOARD[1],3)\
,np.float32)#shape:(1L, 48L, 3L)
#It is a matrix with 1 row and 48 columns with each
#elements as array([0., 0., 0.], dtype=float32)
#making a row array to store the 3d data points

objectp3D[0, :, :2] =2.3*10e-2*np.mgrid[0:CHECKERBOARD[0],\
0:CHECKERBOARD[1]].T.reshape(-1, 2)
#forming a grid and reshaping it to mantain the shape of objectp3D
#0:8->each row is [0,1,...7]*2.3*10e-2
#0:6->each column is [0,1....5]*2.3*10e-2
#objectp3D[0, :, :2]->leaving out the last entry for each element of size (1,3)

prev_img_shape=None

ap=argparse.ArgumentParser()
ap.add_argument("--video")
args=vars(ap.parse_args())

total_frames=int(vs.get(cv.CAP_PROP_FRAME_COUNT))
frame_interval=total_frames/40

gray=None
frame=None
frame_count=0
while True:
        #grabbing frames one by one for faster scan
        _=vs.grab()
        if frame_count%frame_interval==0:
                #reading frame at specific intervals
```

```
                frame=vs.read()

                if args.get("video",False):
                        frame=frame[1]

                if frame is None:
                        break

                frame = cv.rotate(frame, cv.ROTATE_90_CLOCKWISE)
                #converting to GRAYSCLAE space for corner detection
                gray= cv.cvtColor(frame, cv.COLOR_BGR2GRAY)
                ret, corners = cv.findChessboardCorners(gray, CHECKERBOARD,\
                flags=cv.CALIB_CB_ADAPTIVE_THRESH + cv.CALIB_CB_FAST_CHECK \
                + cv.CALIB_CB_NORMALIZE_IMAGE)

                #storing corner locations to a matrix both in image
                #coordinates and in world coordinates
                if ret == True:
                        points3D.append(objectp3D)

                        corners2D=cv.cornerSubPix(gray,corners,(11,11),(-1,-1),criteria)

                        points2D.append(corners2D)
                        frame = cv.drawChessboardCorners(frame,CHECKERBOARD,corners2D, ret)

                cv.imshow("window",frame)

        else:
                pass

        #displaying %completion
        _=os.system('cls')
        print round(((float)(frame_count)/(total_frames)*100.0),2),'% completed'
        frame_count+=1


        if cv.waitKey(1)==ord('q'):
                break

cv.destroyAllWindows()
_=os.system('cls')

#using openCV's calibrate camera module for obtaining camera
#intrinsic matrix and the distortion coefficients
ret, matrix, distortion, r_vecs, t_vecs =
#cv.calibrateCamera(points3D, points2D, gray.shape[::-1], None, None)

#storing the camera intrinsic parameter into a file
np.savez('camera_intrinsic_parameters.npz',
#camera_intrinsic_matrix=matrix,distortion_coefficients=distortion)
```

object_track.py

```
import numpy as np
import argparse
import cv2 as cv
import imutils
import os

#taking user input at command line as video file input
ap=argparse.ArgumentParser()
ap.add_argument("-v","--video")
args=vars(ap.parse_args())

#setting HSV bounds
colLower=(95,50,50)
colUpper=(130,255,255)

#minimum observable area for a valid track contour
min_observable_contour_area=80
vs=cv.VideoCapture(args["video"])
total_frames=int(vs.get(cv.CAP_PROP_FRAME_COUNT))

#file for position-time data
data_file=open("track_data.txt","w")
print "Writing data file..."

frame_count=1

while True:
        #reading video frames
        frame=vs.read()


        if args.get("video",False):
                frame=frame[1]

        #obtaining valid video frames
        if frame is None:
                break

        frame = cv.rotate(frame, cv.ROTATE_90_CLOCKWISE)

        #Gaussian blurring for noise filtering
        blurred=cv.GaussianBlur(frame,(7,7),0)

        #converting to HSV color space
        hsv=cv.cvtColor(blurred,cv.COLOR_BGR2HSV)

        #creating colour mask for tracking
        mask=cv.inRange(hsv,colLower,colUpper)

        #mask postprocessing
        mask=cv.erode(mask,None,iterations=5)
```

```
52          mask=cv.dilate(mask,None,iterations=5)
53
54          #mask for object tracking
55          contours, _=cv.findContours(mask,cv.RETR_EXTERNAL,cv.CHAIN_APPROX_SIMPLE)
56
57          if len(contours)>0:
58                  #selecting mask with largest available area
59                  contours=max(contours,key=cv.contourArea)
60
61                  M=cv.moments(contours)
62                  #computing contour center
63                  centroid=(int(M["m10"]/M["m00"]),int(M["m01"]/M["m00"]))
64
65                  if cv.contourArea(contours)>min_observable_contour_area:
66                          #saving centroid and time data to data file
67                          data_file.write(str(vs.get(cv.CAP_PROP_POS_MSEC)/1000.0)\
68                          +"\t"+str(centroid[0])+"\t"+str(centroid[1])+"\n")
69                          cv.circle(frame,centroid,int(5),(0,0,255),3)
70
71                  else:
72                          #saving  (0,0) as centroid data incase no valid contour is obtained
73                          data_file.write(str(vs.get(cv.CAP_PROP_POS_MSEC)\
74                          /1000.0)+"\t"+str(0)+"\t"+str(0)+"\n")
75
76          else:
77                  data_file.write(str(vs.get(cv.CAP_PROP_POS_MSEC)\
78                  /1000.0)+"\t"+str(0)+"\t"+str(0)+"\n")
79
80          #displaying % completion
81          _=os.system('cls')
82          print round(((float)(frame_count)/(total_frames)*100.0),2),'% completed'
83
84          frame_count+=1
85
86          if cv.waitKey(1)==ord('q'):
87                  cv.imwrite('snapshot.png',frame)
88                  break
89
90  if not args.get("video",False):
91          vs.stop()
92
93  else:
94          vs.release()
95
96  cv.destroyAllWindows()
97  data_file.close()
98  print "Process ended"
```

frame_capture.py

```
1  import numpy as np
2  import argparse
```

```
import cv2 as cv
import imutils

#accepting video file as command line argument
ap=argparse.ArgumentParser()
ap.add_argument("-v","--video")
args=vars(ap.parse_args())

vs=cv.VideoCapture(args["video"])

#frame number of the frame to be capture
frame_number=1

frame_count=0
#iterating to the required frame
while frame_count<frame_number:
        frame=vs.read()
        frame_count+=1

if args.get("video",False):
                frame=frame[1]

if frame is None:
        pass

frame=imutils.resize(frame,width=900)
frame = cv.rotate(frame, cv.ROTATE_90_CLOCKWISE)
#saving frame as an image file
cv.imwrite("track_window.png",frame)
cv.destroyAllWindows()
```

data_synchronize.py

```
import numpy as np
from scipy.interpolate import interp1d
import matplotlib.pyplot as plt
from reconstruction import *
import cv2 as cv

#function for converting cartesian to homogeneous coordinates
def cartesian2homogeneous(arr):
        if arr.ndim == 1:
                return np.hstack([arr, 1])
        else:
                return np.hstack((arr, np.ones((arr.shape[0],1))))

#function for converting homogeneous to cartesian coordinates
def homogeneous2cartesian(arr):
    if arr.ndim == 1:
        return np.asarray([arr[:-1]])
    else:
        return (arr.T[:-1]).T
```

```
20
21  def prescaling(pts):
22      #obtaining the x an y data from the pixel coordinates
23      x_vect=np.asarray(pts[:,0])
24      y_vect=np.asarray(pts[:,1])
25
26      #obtaining the number of data points
27      n=len(x_vect)
28
29      #computing the cetroid for the data points
30      centroid_x=(1.0/n)*np.sum(x_vect)
31      centroid_y=(1.0/n)*np.sum(y_vect)
32
33      #translating the points by the centroid
34      x_vect_centroid=x_vect-centroid_x
35      y_vect_centroid=y_vect-centroid_y
36
37      #computing the average distance of the data points from the data centroid
38      centroid_distance=np.sqrt(x_vect_centroid**2+y_vect_centroid**2)
39      average_distance=(1.0/n)*np.sum(centroid_distance)
40
41      #calculating the required scale factor for the transformation
42      scale_factor=np.sqrt(2.0)/average_distance
43
44      #computing the preconditioning matrix for the transformation
45      T_scale=np.matrix([
46      [scale_factor, 0, -scale_factor*centroid_x],
47      [0, scale_factor, -scale_factor*centroid_y],
48      [0,0,1]
49      ])
50
51      #computing the prescaled coordinates
52      pts_scaled=T_scale.dot(pts.T)
53
54      return T_scale, pts_scaled.T
55
56  #convergence criteria for obtaining alpha
57  def convergence(delta_alpha, alpha_0, eps):
58          if abs(delta_alpha/alpha_0)<eps:
59                  return True
60          return False
61
62
63
64  def estimate_alpha(PTS1_scaled, PTS2_scaled, f_c_pts1, alpha_0, beta, eps, max_iter=20):
65          #iterating the process for maximum number of iteration
66          for iteration_count in range(max_iter):
67                  #copying the scaled points into a temporary variable for the current iteration
68                  pts1_scaled=PTS1_scaled.copy()
69                  pts2_scaled=PTS2_scaled.copy()
70
71                  #computing the shape of data points for frame indices
72                  n1=pts1_scaled.shape[-1]
```

```
73                  n2=pts2_scaled.shape[-1]
74
75                  #array for the frame indices
76                  m=np.arange(0,n1,1)
77                  n=np.arange(0,n2,1)
78
79                  #computing the cubic interpolation function for the trajectory q_2(t)
80                  f_u_pts2=interp1d(n,pts2_scaled[0])
81                  f_v_pts2=interp1d(n,pts2_scaled[1])
82
83                  #setting the starting point for the first camera frame indices
84                  l,=(int(np.ceil(-alpha_0*(1.0/beta))),)
85
86                  #slicing the first camera data points to keep within
87                  #the interpolation range of the second camera capture data
88                  pts1_scaled=pts1_scaled[:,l:]
89                  m=m[l:]
90
91                  #computing the vector w = finite difference delta(pts2)
92                  u_w=f_u_pts2(alpha_0+m*beta+1)-f_u_pts2(alpha_0+m*beta)
93                  v_w=f_v_pts2(alpha_0+m*beta+1)-f_v_pts2(alpha_0+m*beta)
94                  w=np.vstack((u_w,v_w))
95
96                  #computing vector g = pts2(n(m))-alpha*delta(pts1)
97                  u_g=f_u_pts2(alpha_0+m*beta)-alpha_0*u_w
98                  v_g=f_v_pts2(alpha_0+m*beta)-alpha_0*v_w
99                  g=np.vstack((u_g,v_g))
100
101                 #declaring the matrix A1
102                 A_1 = np.zeros((g.shape[1], 9))
103
104                 #defining the matrix A1
105                 for i in range(g.shape[1]):
106                         A_1[i, :] = [ g[0,i]*pts1_scaled[0,i],
107                                       g[0,i]*pts1_scaled[1,i],
108                                       g[0,i],
109                                       g[1,i]*pts1_scaled[0,i],
110                                       g[1,i]*pts1_scaled[1,i],
111                                       g[1,i],
112                                       pts1_scaled[0,i],
113                                       pts1_scaled[1,i], 1  ]
114
115                 #defining the matrix A2
116                 A_2 = np.zeros((w.shape[1], 9))
117
118                 #defining the matrix A2
119                 for i in range(w.shape[1]):
120                         A_2[i, :] = [ w[0,i]*pts1_scaled[0,i],
121                                       w[0,i]*pts1_scaled[1,i],
122                                       w[0,i],
123                                       w[1,i]*pts1_scaled[0,i],
124                                       w[1,i]*pts1_scaled[1,i],
125                                       w[1,i],
```

```
                                        pts1_scaled[0,i],
                                        pts1_scaled[1,i], 1  ]

                A_1=np.matrix(A_1)
                A_2=np.matrix(A_2)

                #Computing the matrix whose eigenvalues are to be determined
                B=(A_2.T).dot(A_2)
                C=(A_2.T).dot(A_1)
                M=(np.linalg.inv(B)).dot(C)

                #EVD for the matrix M
                alpha_list, X=np.linalg.eig(M)

                #obtaing the smallest real eigenvalue as the value for -alpha
                for index in range(len(alpha_list)-1,0,-1):
                        if abs(1.0-alpha_list[index].real)>1e-5 and\
                        abs(alpha_list[index].imag)<1e-1:
                                alpha=-alpha_list[index].real
                                break

                #change in the computed alpha for the convergence condition
                delta_alpha=abs(alpha-alpha_0)

                #checking for the alpha convergence
                if convergence(delta_alpha,alpha_0,eps):
                        break

                #updating the computed alpha
                alpha_0=alpha

        return alpha

residual

if __name__=='__main__':
        #importing the data points for the captured data set
        pts=np.load('point_correspondences.npz')
        pts1=pts['pts1']
        pts2=pts['pts2']

        #converting to homogeneous coordinates for prescaling
        hpts1=cartesian2homogeneous(pts1.T)
        hpts2=cartesian2homogeneous(pts2.T)

        #carrying out preconditioning on the data points
        T_pts1_scaled, pts1_scaled = prescaling(hpts1)
        T_pts2_scaled, pts2_scaled = prescaling(hpts2)

        #reconverting scaled image points to cartesian coordinates
        PTS1_scaled=homogeneous2cartesian(pts1_scaled).T
        PTS2_scaled=homogeneous2cartesian(pts2_scaled).T

```

```
        #defining frame intervals for the cameras
        f_c_pts1=0.06
        f_c_pts2=0.06

        #initial guess time for the first frame for the cameras
        t_0_pts1=0.0
        t_0_pts2=0.75

        #calculating guess value for alpha
        alpha_0=(t_0_pts1-t_0_pts2)/f_c_pts2
        #evaluating beta
        beta=f_c_pts1/f_c_pts2

        #epsilon for the convergence of alpha value
        eps=1.0e-6

        #calling function to estimate alpha using epipolar constraint
        alpha=estimate_alpha(PTS1_scaled, PTS2_scaled, f_c_pts1, alpha_0, beta, eps)

        print alpha
        pts1=pts['pts1']
        pts2=pts['pts2']

        n1=pts1.shape[-1]
        n2=pts2.shape[-1]

        m=np.arange(0,n1,1)
        n=np.arange(0,n2,1)

        f_u_pts2=interp1d(n,pts2[0])
        f_v_pts2=interp1d(n,pts2[1])

        l,=(int(np.ceil(-alpha*(1.0/beta))),)
        pts1=pts1[:,l:]
        m=m[l:]
        t=np.load('time_data.npz')
        t1=t['T1'][l:]

        u_pts2=f_u_pts2(alpha+m*beta)
        v_pts2=f_v_pts2(alpha+m*beta)

        pts1=pts1.T
        pts2=np.vstack((u_pts2,v_pts2)).T

        np.savez('synced_point_correspondences.npz',pts1=pts1,pts2=pts2)
        np.savez('synced_time_data.npz',T=t1)
```

find_fundamental_matrix_eightpoint.py

```
from __future__ import division
import numpy as np
import scipy.optimize
```

```
4
5   #function for converting cartesian to homogeneous coordinates
6   def cartesian2homogeneous(arr):
7           if arr.ndim == 1:
8                   return np.hstack([arr, 1])
9           else:
10                  return np.hstack((arr, np.ones((arr.shape[0],1))))
11
12  #function for converting homogeneous to cartesian coordinates
13  def homogeneous2cartesian(arr):
14      if arr.ndim == 1:
15          return np.asarray([arr[:-1]])
16      else:
17          return (arr.T[:-1]).T
18
19
20  #function for singularising the fundamental matrix using SVD
21  def singularize_F(F):
22          U, S, VT = np.linalg.svd(F)
23          S[-1]=0
24          F=np.dot(U,np.dot(np.diag(S),VT))
25
26          return F
27
28  #residual function for optimising the value
29  #of the fundamental matrix (Hartley and Zisserman)
30  def objective_F(F_stacked, pts1, pts2):
31          F=singularize_F(F_stacked.reshape([3,3]))
32          num_points=pts1.shape[0]
33          #converting from cartesian to homogeneous
34          hpts1 = cartesian2homogeneous(pts1)
35          hpts2 = cartesian2homogeneous(pts2)
36
37          F_p1=np.dot(F, hpts1.T)
38          FT_p2=np.dot(F.T, hpts2.T)
39          #computing the residual for optimising F
40          sum_of_residual_squared=0
41          for fp1, fp2, hp2 in zip(np.asarray(F_p1.T),\
42          np.asarray(FT_p2.T), np.asarray(hpts2)):
43                  sum_of_residual_squared+=(np.dot(hp2,fp1))**2\
44                  * (1/(fp1[0]**2 + fp1[1]**2) + 1/(fp2[0]**2 + fp2[1]**2))
45          print sum_of_residual_squared
46          return sum_of_residual_squared
47
48  #function for refining the value of F
49  def refine_F(F, pts1, pts2):
50      F_stacked=F.reshape([-1])
51      #using scipy's minimize function for optimising F
52      result=scipy.optimize.minimize(objective_F, F_stacked, args=(pts1, pts2))
53      F_stacked=result.x
54      #reshaping F into a 3x3 matrix
55      F_refined=F_stacked.reshape([3,3])
56      F_refined_singularized=singularize_F(F_refined)
```

```
57
58      return F_refined_singularized
59
60  def prescaling(pts):
61      #obtaining the x an y data from the pixel coordinates
62      x_vect=np.asarray(pts[:,0])
63      y_vect=np.asarray(pts[:,1])
64
65      #obtaining the number of data points
66      n=len(x_vect)
67
68      #computing the cetroid for the data points
69      centroid_x=(1.0/n)*np.sum(x_vect)
70      centroid_y=(1.0/n)*np.sum(y_vect)
71
72      #translating the points by the centroid
73      x_vect_centroid=x_vect-centroid_x
74      y_vect_centroid=y_vect-centroid_y
75
76      #computing the average distance of the data points from the data centroid
77      centroid_distance=np.sqrt(x_vect_centroid**2+y_vect_centroid**2)
78      average_distance=(1.0/n)*np.sum(centroid_distance)
79
80      #calculating the required scale factor for the transformation
81      scale_factor=np.sqrt(2.0)/average_distance
82
83      #computing the preconditioning matrix for the transformation
84      T_scale=np.matrix([
85      [scale_factor, 0, -scale_factor*centroid_x],
86      [0, scale_factor, -scale_factor*centroid_y],
87      [0,0,1]
88      ])
89
90      #computing the prescaled coordinates
91      pts_scaled=T_scale.dot(pts.T)
92
93      return T_scale, pts_scaled.T
94
95  def eight_point(pts1, pts2):
96      #converting to homogeneous coordinates for preconditioning the data points
97      hpts1=cartesian2homogeneous(pts1)
98      hpts2=cartesian2homogeneous(pts2)
99
100     #preconditioning the data points by translating
101     #by the centroid and scaling by the required factor
102     T_pts1_scaled, pts1_scaled = prescaling(hpts1)
103     T_pts2_scaled, pts2_scaled = prescaling(hpts2)
104
105     #recoverting from cartesian to homogeneous coordinates
106     pts1_scaled=homogeneous2cartesian(pts1_scaled)
107     pts2_scaled=homogeneous2cartesian(pts2_scaled)
108
109     A_f = np.zeros((pts1_scaled.shape[0], 9))
```

```
110
111     #constructing the matrix for eigen value decomposition
112     for i in range(pts1_scaled.shape[0]):
113         A_f[i, :] = [ pts2_scaled[i,0]*pts1_scaled[i,0],
114                       pts2_scaled[i,0]*pts1_scaled[i,1],
115                       pts2_scaled[i,0],
116                       pts2_scaled[i,1]*pts1_scaled[i,0],
117                       pts2_scaled[i,1]*pts1_scaled[i,1],
118                       pts2_scaled[i,1],
119                       pts1_scaled[i,0],
120                       pts1_scaled[i,1], 1  ]
121
122     #eigenvalue decomposition to obtain the least square solution
123     e_vals, e_vecs = np.linalg.eig(np.dot(A_f.T, A_f))
124
125     #obtaining the eigenvector corresponding to the smallest eigen value
126     F_stacked=e_vecs[:, np.argmin(e_vals)]
127
128     #reshaping the eigenvector in the form of fundamental matrix
129     F_scaled = F_stacked.reshape(3,3)
130
131     #carrying out non-linear least squares to refine the fundamental matrix
132     F_scaled = refine_F(F_scaled, pts1_scaled, pts2_scaled)
133
134     #reverting the effect of preconditioning on the fundamental matrix
135     unscaled_F = (T_pts1_scaled.T).dot(F_scaled).dot(T_pts2_scaled)
136
137     #enforcing the rank condition on the fundamental matrix
138     U,D,V_T=np.linalg.svd(unscaled_F)
139     unscaled_F=U*np.diag([D[0],D[1],0])*(V_T)
140
141     return unscaled_F
```

reconstruction.py

```
1  from find_fundamental_matrix_eightpoint import *
2  from bundle_adjust import *
3
4  #function for computing the essential matrix
5  def compute_essential(F, K1, K2):
6          E=np.dot(np.dot(K2.T, F), K1)
7          return E
8
9  def triangulate(C1, pts1, C2, pts2, K1, K2, M1, M2):
10     #declaring empty list for the 3D triangulated points
11     P_i = []
12
13     #iteration for each pair of image point correpondences
14     for i in range(pts1.shape[0]):
15         #obtaining the matrix A for the homogeneus equation AX=0
16         A = np.array([pts1[i,0]*C1[2,:] - C1[0,:] ,
17                       pts1[i,1]*C1[2,:] - C1[1,:] ,
```

```
                                pts2[i,0]*C2[2,:] - C2[0,:] ,
                                pts2[i,1]*C2[2,:] - C2[1,:]])

            #solving the homogeneous equation using EVD in least squares sense
            e_vals, e_vecs = np.linalg.eig(np.dot(A.T, A))
            X = e_vecs[:, np.argmin(e_vals)]

            #obtaining the correct value of the euclidean coordinates
            X = X/X[-1]
            #appending triangulated point to the list of 3D points
            P_i.append(X)

        P_i = np.asarray(P_i)

        #reprojecting the 3D points onto the camera image planes
        reprojected_pts1 = C1.dot(P_i.T)
        reprojected_pts2 = C2.dot(P_i.T)

        reprojected_pts1 = reprojected_pts1.T
        reprojected_pts2 = reprojected_pts2.T

        #obtaining the true euclidean coordinates for the image points
        for i in range(reprojected_pts1.shape[0]):
            reprojected_pts1[i,:] = reprojected_pts1[i,:] / reprojected_pts1[i, -1]
            reprojected_pts2[i,:] = reprojected_pts2[i,:] / reprojected_pts2[i, -1]

        reprojected_pts1 = reprojected_pts1[:, :-1]
        reprojected_pts2 = reprojected_pts2[:, :-1]

        reprojection_err = 0
        #calculating the total reprojection error
        for i in range(reprojected_pts1.shape[0]):
            reprojection_err = reprojection_err + np.linalg.norm( pts1[i,:]\
            - reprojected_pts1[i,:] )**2 + np.linalg.norm( pts2[i,:] - reprojected_pts2[i,:] )**2

        P_i = P_i[:, :-1]

        return P_i, reprojection_err


def true_M2(pts1, pts2, F, K1, K2):

        #using the fundamental matrix and the camera intrinsic s to obtain the essential matrix
        E = compute_essential(F,K1,K2)

        #considering the first camera frame to be the world frame
        M1 = np.array([[1,0,0,0],
                                [0,1,0,0],
                                [0,0,1,0]])

        #obtaining the list of possile rigid transformations for the relative camera pose
        M2_list = find_M2s(E)

```

```
71      #projection matrix for the first camera
72      C1 = np.dot(K1,M1)
73
74      #defining the maximum reprojection error in case a valid rotation translation is not found
75      err_true = np.inf
76
77      #iteration to determine the true rotation and translation
78      for i in range(M2_list.shape[2]):
79          #obtaining a particular relative orientation
80          M2 = M2_list[:, :, i]
81          #test camera matrix for the second camera
82          C2 = np.dot(K2,M2)
83          #triangulating the 3D points with the selected relative orientation
84          P, err = triangulate(C1, pts1, C2, pts2, K1, K2, M1, M2)
85          #obtaining the z values of the triangulated points
86          z_vect = P[:, 2]
87          #testing if all the triangulated 3D points are in front of the cameras
88          if all(z>0 for z in z_vect):
89              #storing the valid 3D points, relative orientation and
90                          #the second camera projection matrix
91              err_true = err
92              P_true = P.copy()
93              M2_true = M2.copy()
94              C2_true = C2.copy()
95              break
96
97      return P_true, C2_true, M2_true, err_true
98
99
100 def find_M2s(E):
101     #SVD decomposition of essential matrix to obtain the
102         #required combinations of rotation and translation
103     U,S,V = np.linalg.svd(E)
104
105     #enforcing the rank condition on the essential matrix
106     m = S[:2].mean()
107     E = U.dot(np.array([[m,0,0], [0,m,0], [0,0,0]])).dot(V)
108     U,S,V = np.linalg.svd(E)
109
110     #defining the W matrix for obtaining the required rotations
111     W = np.array([[0,-1,0], [1,0,0], [0,0,1]])
112
113     #enforcing the fact that the rotation matrices preserve orientation
114     if np.linalg.det(U.dot(W).dot(V))<0:
115         W = -W
116
117     #obtaining the list of all possible rotations and translations
118     M2s = np.zeros([3,4,4])
119     M2s[:,:,0] = np.concatenate([U.dot(W).dot(V),\
120         U[:,2].reshape([-1, 1])/abs(U[:,2]).max()], axis=1)
121     M2s[:,:,1] = np.concatenate([U.dot(W).dot(V),\
122         -U[:,2].reshape([-1, 1])/abs(U[:,2]).max()], axis=1)
123     M2s[:,:,2] = np.concatenate([U.dot(W.T).dot(V),\
```

```
124         U[:,2].reshape([-1, 1])/abs(U[:,2]).max()], axis=1)
125     M2s[:,:,3] = np.concatenate([U.dot(W.T).dot(V),\
126         -U[:,2].reshape([-1, 1])/abs(U[:,2]).max()], axis=1)
127 
128     return M2s
```

bundle_adjust.py

```
1  from __future__ import division
2  import numpy as np
3  #from scipy.optimize import minimize, least_squares
4  from nlsq_residual import levenberg
5  
6  #function for converting euler angles to rotation matrix
7  def eulerAnglesToRotationMatrix(r):
8      angle_x, angle_y, angle_z = r
9  
10     #rotation matrix for rotation about x axis
11     R_x = np.matrix([
12     [ 1, 0, 0],
13     [ 0, np.cos(angle_x),-np.sin(angle_x)],
14     [ 0, np.sin(angle_x), np.cos(angle_x)]
15     ])
16 
17     #rotation matrix for rotation about y axis
18     R_y = np.matrix([
19     [ np.cos(angle_y), 0, np.sin(angle_y)],
20     [ 0, 1, 0],
21     [-np.sin(angle_y), 0, np.cos(angle_y)]
22     ])
23 
24     #rotation matrix for rotation about z axis
25     R_z = np.matrix([
26     [np.cos(angle_z),-np.sin(angle_z), 0],
27     [np.sin(angle_z), np.cos(angle_z), 0],
28     [0, 0, 1]
29     ])
30 
31     #obtaining the net rotation matrix
32     R = np.dot(R_z, np.dot( R_y, R_x ))
33 
34     return R
35 
36 def rotationMatrixToEulerAngles(R) :
37 
38     sy = np.sqrt(R[0,0] * R[0,0] +  R[1,0] * R[1,0])
39     singular = sy < 1e-6
40 
41     #calculating rotation angles for non singular matrix
42     if  not singular :
43         x = np.arctan2(R[2,1] , R[2,2])
44         y = np.arctan2(-R[2,0], sy)
```

```
45          z = np.arctan2(R[1,0], R[0,0])
46
47      #calculating rotation angles for singular matrix
48      else :
49          x = np.arctan2(-R[1,2], R[1,1])
50          y = np.arctan2(-R[2,0], sy)
51          z = 0
52
53      return np.array([x, y, z])
54
55  def residual(x, K1, M1, p1, K2, p2):
56      #obtaining the 3D points, the rotation angles and the
57      #translation vector from the flattened parameter list
58      n = p1.shape[0]
59      P = np.hstack( ( x[0:n, None], x[n:2*n, None], x[2*n:3*n, None] ) )
60      r = x[3*n:3*n+3]
61      t = x[3*n+3:3*n+6, None]
62
63      #obtaining the rotation matrix from the euler angles
64      R=eulerAnglesToRotationMatrix(r)
65
66      #reconstructing the relative camera pose matrix
67      M2 =  np.hstack( [R, t] )
68
69      #computing the camera projection matrices
70      C1 = np.dot(K1,M1)
71      C2 = np.dot(K2,M2)
72
73      #converting from cartesian to homogeneous coordinate
74      P_homo = np.vstack( [P.T, np.ones(n) ] )
75
76      #projecting 3D points onto the image planes using the projection matrices
77      p1_hat_homo = np.matmul( C1, P_homo )
78      p2_hat_homo = np.matmul( C2, P_homo )
79
80      p1_hat = np.asarray(p1_hat_homo.T)
81      p2_hat = np.asarray(p2_hat_homo.T)
82
83      #obtaining the true euclidean coordinates by dividing by the last coordinate
84      for i in range(p1_hat.shape[0]):
85          p1_hat[i,:] = p1_hat[i,:] / p1_hat[i, -1]
86          p2_hat[i,:] = p2_hat[i,:] / p2_hat[i, -1]
87
88      #excluding the last coordinate from the homogeneous coordinate
89      p1_hat = p1_hat[:, :-1]
90      p2_hat = p2_hat[:, :-1]
91
92      #computing the array of residuals for the reprojected points
93      residuals = np.concatenate([(p1-p1_hat).reshape([-1]),\
94          (p2-p2_hat).reshape([-1]), np.zeros(5)])
95
96      return residuals
97
```

```
def bundleAdjustment(K1, M1, p1, K2, M2_init, p2, P_init):
    #obtaining the rotation and translation from the relative pose matrix
    R_init = M2_init[:, 0:3]
    t_init = M2_init[:, 3].reshape([-1])

    #obtaining the euler angles from the rotation matrix
    r_init=rotationMatrixToEulerAngles(R_init)

    #arranging the parameters into a 1D array
    x_init  = np.hstack( [ P_init[:,0].reshape([-1]),\
        P_init[:,1].reshape([-1]), P_init[:,2].reshape([-1]), r_init, t_init ]  )

    #minimizing the reprojection error using non linear least squares
    x_new = levenberg(residual, x_init, args=(K1, M1, p1, K2, p2))


    #extracting the best values of the 3D points, the rotation and the translation
    n = p1.shape[0]
    P_new = np.hstack( ( x_new[0:n, None], x_new[n:2*n, None], x_new[2*n:3*n, None] ) )
    r_new = x_new[3*n:3*n+3]
    t_new = x_new[3*n+3:3*n+6, None]
    R_new = eulerAnglesToRotationMatrix(r_new)

    #constructing relative pose matrix from the rotation and the translation matrix
    M2_new =  np.hstack( [R_new, t_new] )

    return M2_new, P_new
```

generate_3D_reconstruction.py

```
from bundle_adjust import *
from reconstruction import *
from find_fundamental_matrix_eightpoint import *
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from scipy import signal
import cv2 as cv

#visualizing the 3D plot
def points_3d_visualize(P_best):
    fig = plt.figure()
    ax = fig.gca(projection='3d')
    ax.set_aspect('equal')

    X = P_best[:,0]
    Y = P_best[:,1]
    Z = P_best[:,2]

    #plotting 3D data points onto a 3D plot
    ax.scatter(X, Y, Z)

    max_range = np.array([X.max()-X.min(), Y.max()-Y.min(), Z.max()-Z.min()]).max() / 2.0
```

```
23
24      mid_x = (X.max()+X.min()) * 0.5
25      mid_y = (Y.max()+Y.min()) * 0.5
26      mid_z = (Z.max()+Z.min()) * 0.5
27
28      #setting axes limits
29      ax.set_xlim(mid_x - max_range, mid_x + max_range)
30      ax.set_ylim(mid_y - max_range, mid_y + max_range)
31      ax.set_zlim(mid_z - max_range, mid_z + max_range)
32
33      plt.show()
34
35  if __name__ == '__main__':
36
37      #loading synchronised point correspondences
38      correspondence_data=np.load('synced_point_correspondences.npz')
39
40      pts1=correspondence_data['pts1']
41      pts2=correspondence_data['pts2']
42
43      pt1=pts1.T
44      pt2=pts2.T
45
46      #mild signal filtering for better reconstruction
47      b, a = signal.butter(3, 0.1)
48      pts1 = np.vstack((signal.filtfilt(b, a, pt1[0]),signal.filtfilt(b, a, pt1[1]))).T
49      pts2 = np.vstack((signal.filtfilt(b, a, pt2[0]),signal.filtfilt(b, a, pt2[1]))).T
50
51      #loading thecamera intrinsic matrices
52      intrinsics1=np.load('camera1_intrinsic_parameters1920.npz')
53      intrinsics2=np.load('camera2_intrinsic_parameters1920.npz')
54
55      K1=intrinsics1['camera_intrinsic_matrix']
56      K2=intrinsics2['camera_intrinsic_matrix']
57
58      #taking camera Cm1 as the world coordinate
59      M1 = np.array([[1,0,0,0],
60                             [0,1,0,0],
61                             [0,0,1,0]])
62      #computing projection matrix for camera Cm1
63      C1 = K1.dot(M1)
64
65      #computing the fundametal matrix using eight-point algorithm
66      F=eight_point(pts1,pts2)
67
68      #3D reconstruction using linear triangulation
69      P_optimized, C2_best, M2_best, err_best = true_M2(pts1, pts2, F, K1, K2)
70      #refining on the 3D reconstruction using bundle adjustment
71      M2_optimized, P_optimized=bundleAdjustment(K1, M1, pts1, K2, M2_best, pts2, P_optimized)
72
73      #computing optimised projection matrix
74      M2_optimized=np.asarray(M2_optimized)
75      C2_optimized=np.dot(K2,M2_optimized)
```

```
76
77      #pendulum bob position for gravitational equilibrium position
78      pts1_extra=np.array([[486.0,921.0]])
79      pts2_extra=np.array([[534.0,1432.0]])
80      #triangulating the equilibrium point
81      P_extra, _=triangulate(C1, pts1_extra, C2_optimized, pts2_extra, K1, K2, M1, M2_optimized)
82      P_optimized=np.vstack((P_optimized, P_extra))
83
84      #displaying 3D reconstrution
85      points_3d_visualize(P_optimized)
86
87      np.savez('3D_data_temp.npz',data=P_optimized)
```

find_suspension_point.py

```
1   import numpy as np
2   import matplotlib.pyplot as plt
3   from mpl_toolkits.mplot3d import Axes3D
4   from nlsq_residual import solve_mat_nlsq
5
6   #function for finding the center and radius of sphere by nlsq fitting
7   def Fit_Sphere(spX,spY,spZ):
8       #matrix A for the sphere equation Ax=b
9       A = np.zeros((len(spX),4))
10      A[:,0] = spX*2
11      A[:,1] = spY*2
12      A[:,2] = spZ*2
13      A[:,3] = 1
14      #vector b
15      f = np.zeros((len(spX),1))
16      f[:,0] = (spX*spX) + (spY*spY) + (spZ*spZ)
17      #solving for Ax=b using non-linear least square fitting
18      C=solve_mat_nlsq(A,f)
19      #computing the sphere radius
20      t = (C[0]*C[0])+(C[1]*C[1])+(C[2]*C[2])+C[3]
21      radius = np.sqrt(t)
22
23      return radius, C[0], C[1], C[2]
24
25  #loadind trajectory data
26  DATA=np.load('3D_data_temp.npz')['data'].T
27  x=DATA[0]
28  y=DATA[1]
29  z=DATA[2]
30
31  R,X,Y,Z=np.asarray(Fit_data(x,y,z)).flatten()
32  p_test=[X,Y,Z,R]
33
34  #appending suspension point coordinates to track data
35  x=np.append(x,X)
36  y=np.append(y,Y)
37  z=np.append(z,Z)
```

```
38
39 np.savez('3D_data_sp.npz',x=x,y=y,z=z,R=R)
40 plt.show()
```

coordinate_transform.py

```
1  import numpy as np
2
3  #function for applying the coorinate transformation
4  def transform(x,y,z,X0,Y0,Z0):
5
6          #vector defining camera z-axis
7          vect_cam_z=np.array([0,0,1])
8          #vector defining system z-axis
9          vect_pend_z=np.array([(X0[1]-X0[0]),(Y0[1]-Y0[0]),(Z0[1]-Z0[0])])
10
11         #computing unit vector for rotation axis
12         axis=np.matrix(np.cross(vect_pend_z,vect_cam_z)).T
13         axis=axis/np.linalg.norm(axis)
14
15         #computing rotation angle
16         theta=np.arccos(np.dot(vect_pend_z,vect_cam_z)\
17         /(np.linalg.norm(vect_pend_z)*np.linalg.norm(vect_cam_z)))
18
19         #Rotation matrix R1
20         A=np.matrix([[np.cos(theta),0,0],
21         [0,np.cos(theta),0],
22         [0,0,np.cos(theta)]])
23
24         #Rotation matrix R2
25         B=np.sin(theta)*np.matrix([[0,-axis[2,0],axis[1,0]],
26         [axis[2,0],0,-axis[0,0]],
27         [-axis[1,0],axis[0,0],0]])
28
29         #Rotation matrix R3
30         C=(1-np.cos(theta))*np.matrix([[axis[0,0]*axis[0,0],axis[0,0]\
31         *axis[1,0],axis[0,0]*axis[2,0]],
32         [axis[1,0]*axis[0,0],axis[1,0]*axis[1,0],axis[1,0]*axis[2,0]],
33         [axis[2,0]*axis[0,0],axis[2,0]*axis[1,0],axis[2,0]*axis[2,0]]])
34
35         #net rotation matrix
36         R=(A+B+C)
37         #net translation vector
38         T=np.matrix([[-X0[1]],[-Y0[1]],[-Z0[1]]])
39
40         H=np.matrix(np.vstack((x,y,z)))
41         #applying translation on the data points
42         H=H+T
43         #applying rotation on the data points
44         H=np.asarray(np.dot(R,H))
45
46         return H
```

```
47
48  #loading trajectory data
49  DATA_3D=np.load('3D_data.npz')
50
51  x=DATA_3D['x']
52  y=DATA_3D['y']
53  z=DATA_3D['z']
54  rad=DATA_3D['R']
55
56  X0=x[-2:]
57  Y0=y[-2:]
58  Z0=z[-2:]
59
60  #applying coordinate transform
61  D=transform(x,y,z,X0,Y0,Z0)
62
63  x=D[0][:-2]
64  y=D[1][:-2]
65  z=D[2][:-2]
66
67  X0=D[0][-2:]
68  Y0=D[1][-2:]
69  Z0=D[2][-2:]
70
71  np.savez('XYZ_data.npz',x=x,y=y,z=z,X=X0,Y=Y0,Z=Z0)
72  plt.show()
```

nlsq_differential.py

```
1   import numpy as np
2
3   def chi_squared(residual, f, x_data_vect, y_data_vect, p0):
4           R = residual(f, x_data_vect, y_data_vect, p0)
5           return np.sum(R**2)
6
7
8   def convergence(B, h, p0, chi_squared, m, n, eps):
9           eps1, eps2, eps3 = eps
10
11          #expressions for the convergence criterion
12          exp1 = np.abs(B).max()
13          exp2 = np.abs(h/p0).max()
14          exp3 = chi_squared/(m-n-1)
15
16          #checking the convergence in gradient
17          if  exp1 < eps1:
18                  return True
19          #checking the convergence in parameters
20          if  exp2 < eps2:
21                  return True
22          #checking the convergence in the chi-squared
23          if  exp3 < eps3:
```

```
                return True

        return False


def Jacobian(f, x_vect, p_vect, args, delta_p):
        n = len(p_vect)
        J = []

        #computing columns for the Jacobian
        for p_index in range(n):
                #computing the parameter vector for partial derivatives
                p_vect_dashed = p_vect.copy()
                p_vect_dashed[p_index] = p_vect_dashed[p_index] + delta_p
                #computing the partial derivatives for the Jacobian
                f_deriv_p = (f(x_vect, p_vect_dashed, *args) \
                - f(x_vect, p_vect, *args)) / delta_p
                J.append(f_deriv_p)

        return np.matrix(J).T


def newton_gauss(f, x_data_vect, y_data_vect, p0, args=(),\
delta_p=0.2, max_iter=1000, eps = (1.0e-8, 1.0e-8,1.0e-2)):

        p0=np.asarray(p0)
        residual = lambda f, x_data_vect, y_data_vect, p0, args :
               y_data_vect - f(x_data_vect, p0, *args)

        iter_count = 1
        while iter_count <= max_iter:
                J = Jacobian(f, x_data_vect, p0, args, delta_p)
                R = residual(f, x_data_vect, y_data_vect, p0, args)
                chi_squared_p = np.sum(R**2)

                R=np.matrix(R).T
                A = (J.T).dot(J)
                B = (J.T).dot(R)

                h = np.linalg.inv(A).dot(B)
                h = np.asarray(h).flatten()

                p0 += h
                iter_count += 1

                if convergence(B, h, p0, chi_squared_p,
        len(x_data_vect), len(p0), eps):
                        break

        return p0


def levenberg(f, x_data_vect, y_data_vect, p0, args=(), delta_p=1.0e-1, max_iter=500,\
```

```
77  eps = (1.0e-8, 1.0e-8,1.0e-2, 0.05), lambda_0=100, L_scale_up=11, L_scale_down=9):
78
79          p0=np.asarray(p0)
80          #creating function for computing the residual to be minimized
81          residual = lambda f, x_data_vect, y_data_vect, p0, args :\
82          y_data_vect - f(x_data_vect, p0, *args)
83
84          #counter for keeping track of number of iterations
85          iter_count = 1
86
87          while iter_count <= max_iter:
88                  #obtaining the Jacobian matrix
89                  J = Jacobian(f, x_data_vect, p0, args, delta_p)
90
91                  #obtaining the residual array
92                  R = residual(f, x_data_vect, y_data_vect, p0, args)
93                  R=np.matrix(R).T
94
95                  #solving the matrix equation to improve the fit parameters
96                  A = (J.T).dot(J)
97                  A_diag = np.diag(np.diag(A))
98                  B = (J.T).dot(R)
99                  C = A + lambda_0 * A_diag
100
101                 h = np.linalg.inv(C).dot(B)
102                 h_flat = np.asarray(h).flatten()
103
104                 #computing the chi squared value for the current and the improved parameters
105                 chi_squared_p=chi_squared(residual,
106                 f, x_data_vect, y_data_vect, p0, args)
107                 chi_squared_ph=chi_squared(residual,
108                 f, x_data_vect, y_data_vect, p0+h_flat, args)
109
110                 #computing the metric measuring the goodness of the fit
111                 rho = (chi_squared_p - chi_squared_ph)/((h.T).dot(lambda_0*A_diag.dot(h)+B))
112
113                 #checking the threshold for the rho
114                 if rho > eps[3]:
115                         #updating the fit parameters
116                         p0+=h_flat
117                         #down-scaling the damping factor lambda
118                         lambda_0=max((lambda_0/L_scale_down), 1.0e-7)
119
120                 else:
121                         #up-scaling the damping factor lambda
122                         lambda_0=min((lambda_0*L_scale_up), 1.0e7)
123
124                 #checking for fit convergence
125                 if convergence(B, h_flat, p0, chi_squared_ph,\
126                 len(x_data_vect), len(p0), eps[:3]):
127                         break
128
129                 iter_count += 1
```

```
130
131         return p0
```

nlsq_residual.py

```
1   import numpy as np
2
3   #function for calculating the chi-squared value
4   def compute_chi_squared(residual, p, args):
5           R = residual(p, *args)
6           return np.sum(R**2)
7
8   #function for determining the convergence of the fit
9   def convergence(B, h, p0, chi_squared, m, n, eps):
10          eps1, eps2, eps3 = eps
11
12          #expressions for the convergence criterion
13          exp1 = np.abs(B).max()
14          exp2 = np.abs(h/p0).max()
15          exp3 = chi_squared/(m-n-1)
16
17          print exp1,exp2,exp3
18          #checking the convergence in gradient
19          if  exp1 < eps1:
20                  return True
21          #checking the convergence in parameters
22          if  exp2 < eps2:
23                  return True
24          #checking the convergence in the chi-squared
25          if  exp3 < eps3:
26                  return True
27
28          return False
29
30
31  def Jacobian(residual, p, args=(), delta_p=1.0e-2):
32          J=[]
33          m=len(p)
34
35          #computing columns for the Jacobian
36          for index in range(m):
37                  #computing the parameter vector for partial derivatives
38                  p_change=p.copy()
39                  p_change[index]+=delta_p
40                  #computing residual for the partial derivatives
41                  f_p=residual(p,*args)
42                  f_p_change=residual(p_change,*args)
43                  #computing partial derivatives for the Jacobian
44                  f_deriv_p=(f_p_change-f_p)/delta_p
45                  J.append(f_deriv_p)
46
47          return np.matrix(J).T
```

```
48
49
50  def newton_gauss(residual, p, args=(),
51  eps=(1.0e-5, 1.0e-5, 1.0e-5), max_iter=50):
52
53          p=np.asarray(p)
54          #counter for keeping track of number of iterations
55          iter_count=1
56
57          while iter_count <= max_iter:
58                  #obtaining the Jacobian matrix
59                  J = Jacobian(residual, p, args)
60
61                  #obtaining the residual array
62                  R = np.matrix(residual(p,*args)).T
63
64                  #solving the matrix equation to improve the fit parameters
65                  A = (J.T).dot(J)
66                  B = (J.T).dot(R)
67
68                  h=-np.linalg.inv(A).dot(B)
69                  h=np.asarray(h).flatten()
70
71                  p+=h
72                  iter_count+=1
73
74                  #computing the chi squared value for the improved parameters
75                  chi_squared=compute_chi_squared(residual, p, args)
76
77                  #checking for fit convergence
78                  if convergence(B, h, p, chi_squared, len(R), len(p), eps):
79                          break
80
81          return p
82
83  def levenberg(residual, p, args=(), eps=(1.0e-8, 1.0e-8, 1.0e-5, 1.0e-2),
84  max_iter=50, lambda_0=5.0, L_scale_up=11, L_scale_down=9):
85
86          p=np.asarray(p)
87          #counter for keeping track of number of iterations
88          iter_count=1
89
90          while iter_count <= max_iter:
91                  #obtaining the Jacobian matrix
92                  J = Jacobian(residual, p, args)
93
94                  #obtaining the residual array
95                  R = np.matrix(residual(p,*args)).T
96
97                  #solving the matrix equation to improve the fit parameters
98                  A = (J.T).dot(J)
99                  A_diag = np.diag(np.diag(A))
100                 B = (J.T).dot(R)
```

```
                C = A + lambda_0 * A_diag

                h=-np.linalg.inv(C).dot(B)
                h_flat=np.asarray(h).flatten()

                #computing the chi squared value for the current and the improved parameters
                chi_squared_p=compute_chi_squared(residual, p, args)
                chi_squared_ph=compute_chi_squared(residual, p+h_flat, args)
                #computing the metric measuring the goodness of the fit
                rho = (chi_squared_ph - chi_squared_p)/((h.T).dot(lambda_0*A_diag.dot(h)+B))

                #checking the threshold for the rho
                if rho > eps[3]:
                        #updating the fit parameters
                        p+=h_flat
                        #down-scaling the damping factor lambda
                        lambda_0=max((lambda_0/L_scale_down), 1.0e-7)

                else:
                        #up-scaling the damping factor lambda
                        lambda_0=min((lambda_0*L_scale_up), 1.0e7)

                #checking for fit convergence
                if convergence(B, h, p, chi_squared_ph, len(R), len(p), eps[:3]):
                        break

                iter_count += 1

        return p

#function for solving matrix equation Ax=b in least square sense
def solve_mat_nlsq(A,b):
        n=A.shape[-1]
        x=tuple(np.ones(n))

        residual=lambda x, A, b: np.asarray(A.dot(np.matrix(x).T)-b).flatten()
        params=newton_gauss(residual, x, args=(A,b))

        return params
```

fit_data.py ***(for spherical pendulum)***

```
import numpy as np
from numpy import sin, cos, tan
from nlsq_diff import levenberg, newton_gauss
import matplotlib.pyplot as plt
from scipy.integrate import odeint

#differential model for data fitting
def differential_model(initial_conditions,t,Fp1,Fp2):
        #unpacking initial conditions
        theta,phi,theta_dot,phi_dot=initial_conditions
```

```
11
12          #damping for the theta and phi tangential components
13          theta_damping=Fp2*np.sin(phi)*theta_dot
14          phi_damping=Fp2*phi_dot
15
16          #computing theta and phi accelarations
17          theta_dot_dot=(-2.0/tan(phi))*phi_dot*theta_dot-theta_damping
18          phi_dot_dot=sin(phi)*cos(phi)*(theta_dot**2)+Fp1*sin(phi)-phi_damping
19
20          #returning the state variable derivatives
21          return (theta_dot,phi_dot,theta_dot_dot,phi_dot_dot)
22
23
24  #function for calculating the sum of residual squares
25  def fit_func(t, p0):
26
27          #tuple for the initial conditions of the differential equations
28          initial_conditions=p0[:4]
29
30          #params for differential equations
31          params=p0[4:]
32
33          #using odeint() to solve for the coupled differential equations
34          d=odeint(differential_model,initial_conditions,t,args=params).T
35
36          #returning the model data
37          return np.append(d[0],d[1])
38
39  #loading track data
40  DATA=np.load('XYZ_data.npz')
41  T=np.load('time_data.npz')['T']
42
43  X=DATA['x']
44  Y=DATA['y']
45  Z=DATA['z']
46
47  #transforming from cartesian to polar coordinates
48  THETA=np.arctan2(Y,X)
49  PHI=np.arccos(Z/R)
50
51  #since theta is not a bound variable, the constraint on theta
52  #being defined between 0 and 180 degrees is removed
53  for theta_index in range(len(THETA)-1):
54          if (THETA[theta_index]-THETA[theta_index+1])>np.pi:
55                  THETA[(theta_index+1):]=THETA[(theta_index+1):]+2*np.pi
56          elif (THETA[theta_index+1]-THETA[theta_index])>np.pi:
57                  THETA[(theta_index+1):]=THETA[(theta_index+1):]-2*np.pi
58          else:
59                  pass
60
61
62  no_dp=len(T)
63  T=T[:no_dp]
```

```
64  THETA=THETA[:no_dp]
65  PHI=PHI[:no_dp]
66
67
68  #setting guess values for the parameters
69  theta_0=THETA[0]
70  phi_0=PHI[0]
71  theta_dot_0=(THETA[1]-THETA[0])/(T[1]-T[0])
72  phi_dot_0=(PHI[1]-PHI[0])/(T[1]-T[0])
73  Fp1=23.0
74  Fp2=0.02
75
76
77  p0=(theta_0,phi_0,theta_dot_0,phi_dot_0,Fp1,Fp2)
78  Y_data=np.append(THETA,PHI)
79
80  #using minimize function to minmize the 'sum of the residual squares' using
81  #least square method and hence finding out the optimum values for the parameters
82  p0 = levenberg(fit_func, T, Y_data, p0,max_iter=50)
83
84  #printing fit parameters
85  print p0
86
87  initial_conditions=p0[:4]
88  params=p0[4:]
89
90  #simuating model with the fit parameters
91  d=odeint(differential_model,initial_conditions,T,args=params).T
92
93  #plotting fit results
94  plt.subplot(1,2,1)
95  plt.title(r'Best fit curve with data for $\theta$')
96  plt.xlabel(r'time(in s)$\rightarrow$')
97  plt.ylabel(r'$\theta$(in radians)$\rightarrow$')
98  plt.plot(T,THETA,'b+',label='theta data')
99  plt.plot(T,d[0],'r',label='theta model')
100
101 plt.subplot(1,2,2)
102 plt.title(r'Best fit curve with data for $\phi$')
103 plt.xlabel(r'time(in s)$\rightarrow$')
104 plt.ylabel(r'$\phi$(in radians)$\rightarrow$')
105 plt.plot(T,PHI,'b+',label='phi data')
106 plt.plot(T,d[1],'r',label='phi model')
107
108 plt.legend(fontsize=8)
109
110 np.savez('polar_data.npz',theta=d[0],theta1=THETA,phi=d[1],\
111 phi1=PHI,theta_dot=d[2],phi_dot=d[3],)
112 plt.show()
```

fit_data.py ***(for magnetic pendulum)***

```
import numpy as np
from numpy import sin, cos, tan, abs
from nlsq_residual import levenberg
import matplotlib.pyplot as plt
from scipy.integrate import odeint

#function for modelling the differential equation
def differential_model(initial_conditions, t,
Fp1, Fp2, Fp3, s, l, x_0, y_0, m_rod, m_bob):
        #specifying the initial conditions for the system
        theta,phi,theta_dot,phi_dot=initial_conditions
        #mu_0/(4*pi)
        k=1.0e-7

        #the damping terms affecting the motion of the pendulum
        phi_damping=Fp2*phi_dot
        theta_damping=Fp2*np.sin(phi)*theta_dot

        r_12_2=-2*l*(sin(theta)*y_0+cos(theta)*x_0)*
        sin(phi)+2*l*s*cos(phi)+y_0**2+x_0**2+s**2+l**2

        #derivative of magnetic potential energy wrt phi
        dUm_dphi=Fp3*k*((-(3*l*sin(phi)*(-(sin(theta)*y_0+cos(theta)*x_0)
        *sin(phi)+s*cos(phi)+l))/(r_12_2)+(3*(l*cos(phi)+s)*(-s*sin(phi)-
        (sin(theta)*y_0+cos(theta)*x_0)*cos(phi)))/(r_12_2)-(3*(l*cos(phi)+s)
        *(-2*l*s*sin(phi)-2*l*(sin(theta)*y_0+cos(theta)*x_0)*cos(phi))
        *(-(sin(theta)*y_0+cos(theta)*x_0)*sin(phi)+s*cos(phi)+l))/(r_12_2)**2
        +sin(phi))/(r_12_2*abs(r_12_2)**(1.0/2.0))-(3*(-2*l*s*sin(phi)-2*l*
        (sin(theta)*y_0+cos(theta)*x_0)*cos(phi))*((3*(l*cos(phi)+s)*
        (-(sin(theta)*y_0+cos(theta)*x_0)*sin(phi)+s*cos(phi)+l))/(r_12_2)
        -cos(phi)))/(2*(r_12_2**2*abs(r_12_2)**(1.0/2.0))))

        #derivative of magnetic potential energy wrt theta
        dUm_dtheta=-Fp3*k*((3*sin(phi)*(x_0*sin(theta)-y_0*cos(theta))
        *((3*l*sin(phi)*s+l**2*cos(phi)*sin(phi))*y_0*sin(theta)+(3*l*
        sin(phi)*s+l**2*cos(phi)*sin(phi))*x_0*cos(theta)+(s+2*l*
        cos(phi))*y_0**2+(s+2*l*cos(phi))*x_0**2+s**3-l*cos(phi)
        *s**2+(-l**2*cos(phi)**2-4*l**2)*s-3*l**3*cos(phi)))
        /(-(r_12_2)**3*(r_12_2)**(1.0/2.0)))

        #equations of motion for the system
        theta_dot_dot=(-2.0/tan(phi))*phi_dot*theta_dot-
        (1.0/(np.sin(phi)**2*((l**2)*(m_bob+(m_rod/3.0)))))*dUm_dtheta-theta_damping
        phi_dot_dot=sin(phi)*cos(phi)*(theta_dot**2)+Fp1
        *sin(phi)-(1.0/(l**2*(m_bob+(m_rod/3.0))))*dUm_dphi-phi_damping

        #returning the time  derivatives  of the initial conditions
        return (theta_dot,phi_dot,theta_dot_dot,phi_dot_dot)

def fit_func(p0, T, Y_data, s, l, x_0, y_0, m_rod, m_bob):
```

```
        #tuple for the initial conditions of the differential equations
        initial_conditions=p0[:4]

        #params for differential equations
        params=tuple(np.append(p0[4:],np.array([s,l,x_0,y_0,m_rod,m_bob])))

        #using odeint to solve for the coupled differential equations
        d=odeint(differential_model,initial_conditions,T,args=params).T

        #returning the model data
        return (Y_data-np.append(d[0],d[1]))

#loading position-time data for the pendulum
DATA=np.load('XYZ_data.npz')
T=np.load('time_data.npz')['T']

X=DATA['x']
Y=DATA['y']
Z=DATA['z']

R=np.sqrt(X**2+Y**2+Z**2)

#transforming from the cartesian to the polar coordinates
THETA=np.arctan2(Y,X)
PHI=np.arccos(Z/R)

#since theta is not a bound variable, the constraint on theta
#being defined between 0 and 180 degrees is removed
for theta_index in range(len(THETA)-1):
        if (THETA[theta_index]-THETA[theta_index+1])>np.pi:
                THETA[(theta_index+1):]=THETA[(theta_index+1):]+2*np.pi
        elif (THETA[theta_index+1]-THETA[theta_index])>np.pi:
                THETA[(theta_index+1):]=THETA[(theta_index+1):]-2*np.pi
        else:
                pass

no_dp=200
T=T[:no_dp]
THETA=THETA[:no_dp]
PHI=PHI[:no_dp]

#defining the constants of the motion
s=0.47
l=0.432
x_0=0.01
y_0=0.02
m_rod=0.0067
m_bob=0.03317

#guess values for the parametes
theta_0=THETA[0]
phi_0=PHI[0]
theta_dot_0=(THETA[1]-THETA[0])/(T[1]-T[0])
```

```
phi_dot_0=(PHI[1]-PHI[0])/(T[1]-T[0])
Fp1=9.5
Fp2=0.05
Fp3=0.26

p0=(theta_0,phi_0,theta_dot_0,phi_dot_0,Fp1,Fp2,Fp3)
Y_data=np.append(THETA,PHI)

#using minimize function to minmize the 'sum of the residual squares' using
#least square method and hence finding out the optimum values for the parameters
p0 = levenberg(fit_func, p0,args=(T, Y_data, s, l, x_0, y_0, m_rod, m_bob))
#printing fit parameters
print p0

initial_conditions=p0[:4]
params=tuple(np.append(p0[4:],np.array([s,l,x_0,y_0,m_rod,m_bob])))

#simuating model with the fit parameters
d=odeint(differential_model,initial_conditions,T,args=params).T

#plotting fit results
plt.subplot(1,2,1)
plt.title(r'Best fit curve with data for $\theta$')
plt.xlabel(r'time(in s)$\rightarrow$')
plt.ylabel(r'$\theta$(in radians)$\rightarrow$')
plt.plot(T,THETA,'b+',label='theta data')
plt.plot(T,d[0],'r',label='theta model')

plt.subplot(1,2,2)
plt.title(r'Best fit curve with data for $\phi$')
plt.xlabel(r'time(in s)$\rightarrow$')
plt.ylabel(r'$\phi$(in radians)$\rightarrow$')
plt.plot(T,PHI,'b+',label='phi data')
plt.plot(T,d[1],'r',label='phi model')

plt.legend(fontsize=8)
plt.show()
```

All the python scripts associated with the project are linked to the given URL : `https://drive.google.com/drive/folders/126q-B5NasxbafJAvmGK43AoFjAVO053U?usp=sharing`

B Links to Data and Video Files

This section contains the link to the video files and data files used and generated while performing the experiment.

File Type	***File Name***	***Link***
Video	spherical_cam_1.mp4	`https://cutt.ly/2bY0wIY`
	spherical_cam_2.mp4	`https://cutt.ly/CbY0yAn`
	magnetic_cam_1.mp4	`https://cutt.ly/bbOoLpB`
	magnetic_cam_2.mp4	`https://cutt.ly/DbOoCev`
Data	camera1_intrinsic_parameters1920.npz	`https://cutt.ly/0bY1oDD`
	camera2_intrinsic_parameters1920.npz	`https://cutt.ly/1bY1dKZ`
	spherical_track_1.txt	`https://cutt.ly/TbY1lNc`
	spherical_track_2.txt	`https://cutt.ly/ybY1cxl`
	magnetic_track_1.txt	`https://cutt.ly/FbI3clp`
	magnetic_track_2.txt	`https://cutt.ly/JbI3nzs`
	synced_point_correspondences_sp.npz	`https://cutt.ly/QbY1nNX`
	time_data_sp.npz	`https://cutt.ly/pbY1WlO`
	synced_point_correspondences_mp.npz	`https://cutt.ly/SbI38zv`
	time_data_mp.npz	`https://cutt.ly/IbI35CH`
	3D_data_sp.npz	`https://cutt.ly/7bYM8vU`
	XYZ_data_sp.npz	`https://cutt.ly/2bY1R0c`
	3D_data_mp.npz	`https://cutt.ly/sbI3Xb7`
	XYZ_data_mp.npz	`https://cutt.ly/DbI31bp`

Table 18: List of data and video files

C Camera Calibration

A 3D point $\boldsymbol{P}$ in world co-ordinates is projected onto a point in the image plane using intrinsic camera parameters. The process of estimating these parameters is know as ***camera calibration***. There are essentially, two types of parameters, namely,

- ***Internal parameters*** of the camera or lens system like focal length, optical center, and radial distortion coefficients of the camera lens.
- ***External parameters*** refers to the orientation of the camera with respect to some world coordinate system.

There are different kinds of camera calibration methods, the one that will be employed in the experiment is the ***checkerboard based method***. The calibration process is expressed via the following steps:

- ***Defining the real world co-ordinates with checkerboard pattern :*** In this process, the camera parameters are estimated by a known set of 3D world points (X, Y, Z) and their corresponding pixel co-ordinates (u, v) in the image. To obtain the 3D points a checkerboard pattern ***with known dimensions*** is photographed at many different orientations. The corner points of the checkerboard

are taken to be the world coordinates with the origin as one of the corners. Now, since all the corner points lie on a plane, Z co-ordinate of the world co-ordinates can be arbitrarily chosen to be ***zero***. Thus, the 3D points can easily be defined by taking one point as the reference $(0, 0)$and defining the remaining with respect to that reference point.

- ***Capture multiple images of the checkerboard from different viewpoints :*** Keeping the checkerboard pattern static, multiples images of the checkerboard are taken by moving the camera aroun.

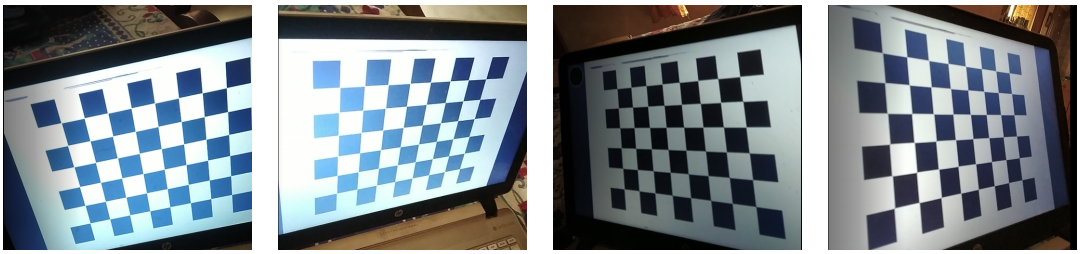


Figure 37: Images of the checkerboard from different view points.

- ***Finding 2D coordinates of checkerboard :*** The 3D location of points on the checkerboard in world coordinates are known to us. For calibration, the 2D pixel locations of these checkerboard corners in the images are to be obtained. OpenCV provides a builtin function called *findChessboardCorners* that looks for a checkerboard and returns the coordinates of the corners. OpenCV's function *cornerSubPix* takes in the original image, and the location of corners, and looks for the best corner location inside a small neighbourhood of the original location. The algorithm is iterative in nature and therefore a termination criteria is required to terminate the process.
- ***Calibrate Camera :*** The final step of calibration is to pass the 3D points in world coordinates and their corresponding 2D locations in all images to openCV's *calibrateCamera* method. The details of the calibration mathematics can be found in https://web.stanford.edu/class/cs231a/course_notes/01-camera-models.pdf.

Camera Calibration Flowchart

Defining real world coordinates of 3D points using checkerboard pattern of known size.

↓

Capturing the images of the checkerboard from different viewpoints.

↓

Finding the pixel coordinates (u, v) for each 3D point in different images.

↓

Finding the camera parameters using the 3D points and the pixel coordinates.