Smart Physics with Video Tracking*

Muhammad Umar Hassan, Bilal Aftab Usman and Muhammad Sabieh Anwar
Center for Experimental Physics Education, Syed Babar Ali School of Science and Engineering, LUMS

V. 2017-1; February 23, 2017

The Smart Physics lab has been designed to introduce the students to new methods of data collection using commonly available consumer devices. This primer is meant as a general introduction to the experiments in the Smart Physics Lab in which video tracking is used to study several principles of physics. The aim is to explain the general methodology of conducting experiments with video tracking and extracting data for numerical analysis. For specific experiments, one may refer to the individual experiment manuals that are uploaded on our website.

1 Video Motion Tracking

Video analysis allows us to track the position and orientation of an object of interest with respect to time. We can process the data acquired to learn and investigate the underlying physics. The aim of this primer is to highlight different facets of the procedure developed in Physlab for this purpose. Examples of phenomena that can be studied using video analysis include the oscillatory motion of a pendulum, projectiles, collisions, rolling and sliding motion etc. Refer to our website for several useful ideas.

1.1 Overview of Video Processing

The usual practice is to record a high frames per second (fps) video of a moving object which has been marked with one or multiple track-points. This video is processed in Matlab using our video tracking library, PhysTrack. Using the tools included in this package the user can trim the ends of video, crop the frames and convert it to binary. The user then interactively selects or lets the software detect objects in the first frame for automated tracking. These objects or collections thereof are then automatically tracked through the rest of the video. A reference coordinate system is defined to interconvert trajectories between image and real spaces and a calibration constant is input. The positional data is then used to analyze different aspects of the underlying motion. All of these steps are achieved through an interactive graphical user interface (GUI).

Note that the codes we provide require installation of MATLAB R2010a or later along with `computer vision` and `image processing` toolboxes. The latest source code of PhysTrack can be downloaded from the GitHub repository[1] and a detailed description of important functions and GUI tools is available at the online wiki[2].

Following are, in general, the steps involved in a typical video tracking procedure.

1. **Data Extraction**

   Once the video is captured and transferred to the computer, PhysTrack is used to track the required object and extract the positional data.

   In order to investigate a diverse range of physical phenomenon we have made different variants of the code. The relevant programs are invoked depending upon the degrees of freedom of the moving object, the number and complication of objects and the positions of track-points. For example, the following scripts are readily available. Their usage is self explanatory.

   - `analyzeRotationOnAFixedPivot`
   - `analyzeSlidingFriction`
   - `analyzeRotationalFriction`
   - `analyzeProjectileMotion`
   - `analyze2DCollisions`
   - `analyze1DSHM`

   Once, any one of these scripts is invoked, a wizard starts which guides through the process. The main theme of this process is described below.

   - The user first selects a video file from the computer for processing.
   - Using a GUI tool, the user, marks the starting and ending frames of the video. Trimming is important because when capturing at high speed, delays of only a couple of seconds may result in hundreds of useless video frames.
   - The wizard then lets the user define a region of interest for cropping the video. It is not necessary to define a crop region but in many cases, it improves computer performance by saving processing time on unnecessary pixels.
   - For some experiments, we may also need to convert the video to binary for which the function presents a GUI tool. This tool is used to determine a threshold level on the grayscale video image which is used to convert the pixels to black or white, exposing the required objects against a contrasting background.
   - The user graphically specifies objects on the first frame of the video which need to be tracked in the consequent frames. The wizard guides about how the user needs to select these objects.
   - These objects are tracked automatically by one of the object trackers included in PhysTrack and trajectories are computed.

PhysTrack includes two kinds of trackers for automated object tracking. The first tracker is based on Kanade-Lucas-Tomasi's (KLT) algorithm [3]. KLT tries to find out the eigen features in a frame using a technique which may find multiple potential matches for a single feature. It, then, identifies the best match out of all the potential matches (while minimizing false positives). The second tracker is based on object detection using background difference method [4] and is called Binary Object Tracker (BOT). It detects all connected regions in a binary image and determines the centroids. It then matches these positions with the reference objects provided to identify there new location in the processed frame. The position of reference objects is updated and the procedure is repeated for the reset of the frames. Both of the trackers extract the trajectories of objects in form of `xdata` and `ydata`. Generically, this data is arranged in structs instead of arrays.

Once the desired points have been tracked, the remaining part of the script generates time stamps from the video frame rate information.

For the expert user, here is an example of code which is used by the "analyze motion" scripts to initialize the video and select the objects to be tracked.

```
% create a video reader2 object first.
vro = PhysTrack.VideoReader2;
% let the user select some objects of interest on the first frame.
obs = PhysTrack.GetObjects(vro);
```

Here is a snippet of the code for tracking the objects automatically through the rest of the video and extracting the positional data.

```
% Track the objects though out the video file. In case the tracking procedure
% failed midway, the new cropping information will be received in the new vro.
[traj, vro] = PhysTrack.KLT(vro, obs); % for RGB video
% [traj, vro] = PhysTrack.BOT(vro, obs); % for Binary video
% define a coordinate system and ppm
```

Note that the video tracking process tracks the objects in forward direction only and if the tracking fails midway, the video is trimmed automatically to adjust to the new length.

2. **Preparing the Positional Data**

Phystrack originally returns data in the image's native reference coordinate system in pixel units. Clearly, we need to convert this data to a real world coordinate system. For this purpose, the script automatically presents a coordinate system tool which lets the user draw a coordinate system on the screen and define a distance calibration constant. The trajectories are then transformed to the new coordinate system and scaled accordingly.

Here is a snippet of the code which defines a reference coordinate system and transforms the trajectories.

```
% define a coordinate system and distance calibration constant. The user will
% define the parameters on the screen.
[rwRCS, ppm] = PhysTrack.DrawCoordinateSystem(vro);
```

```
% transform the trajectories.
traj = PhysTrack.TransformCart2Cart(trajs, rwRCS);
% finally, convert the units from pixels to meters. Since the trajectories
% are in the form of structs, we cannot directly operate them with ppm. we
% use the  PhysTrack helper function for this purpose
traj = PhysTrack.StructOp(traj, ppm, '/');
```

- Once the trajectories of all the objects are acquired, the user identifies a reference coordinate system for coordinates transformation using a GUI. The tool also allows to manually mark two points inside the video, with known real distance between them. This is used to define a calibration constant in units of pixels per meter, called `ppm`, which is used to inter-convert distances between the video and the real world.

Here is a code snippet which is used to create time stamps for the converted positional data.

```
% Create time stamps for the data. Use the updated vro for this purpose.
t = PhysTrack.GenerateTimeStamps(vro);
```

3. **Analysis**

   The third step involves post processing. A wealth of information can be extracted from the trajectory of track-points obtained in previous steps. For example, if a body oscillating with a hanging spring was captured and processed using the `analyze1DSHM` script, one will obtain the x and y coordinates of the object's position recorded at successive instances. The `ydata` of this trajectory, say, will represent the vertical motion undergone by the oscillating object. A plot of these coordinates against time can be made, expecting to observe a sinusoid representing simple harmonic motion. The first numerical derivative of the `ydata` taken on time stamps will yield the object's oscillatory speed.

   Depending upon the type of motion selected in step 1, the script will automatically perform some preliminary analysis which will be displayed as figures. The exact analysis will, however, depend on the experiment and we leave this step largely to the experimenter. The key is that `xdata`, `ydata` and their time stamps can generate any information of interest. Other useful tools for analysis and data processing are discussed in Section 2 of this document.

   The following code snippet exemplifies the usage of the data structs produced by PhysTrack.

```
% Lets say we get the trajectory as 'traj' and time as 't'.
% The following line will plot the xdata of first tracked point against time.
plot(t, traj.tp1.x);
% The following line will plot the ydata of the second point against xdata.
plot(traj.tp2.x, traj.tp2.y);
```

1.2 Some Helpful Considerations about Video Recording

In this section, we list some useful tips for recording video. These considerations will help in video tracking and analysis.

(a) It is important to mark a point on the object which is big enough to fill some tens of pixels in the recorded video. Although this video will be rescaled automatically to higher resolution during the processing, but still, bigger the marker, lesser will be chances that one loses the object during the course of the video.

(b) If a plain object is recorded for tracking, the algorithm still works but there are bright chances that the trackpoints will be lost as the video proceeds or will give wrong location information. Hence, special care has to be taken while using plain objects which cannot be otherwise physically marked.

(c) Experiments where the force of gravity is important, it is recommended to use a spirit-level which is also visible in the video. This will help defining a reference coordinate system.

(d) It is a good practice to place the camera about 5–6 feet away from the plane of motion. This reduces perspective distortion.

(e) An important decision to make in high-speed video motion analysis is to consider which is more important: high frame rate, high shutter speed, high resolution, shallow depth of field or better sensitivity for light (ISO number). For example, higher the frame rate, more frames will be captured during the same amount of time which will give more sample readings for position. Higher the resolution, lesser will be the error in measuring the instantaneous position of an object. Higher the shutter speed, lower will be the ghosting caused due to movement. This will also improve the process of object tracking and reduce the error in positional measurement. Smaller the aperture size (denoted by f/n in camera) larger will be the depth of field which will improve image sharpness. Higher the ISO, more sensitive will be the camera to light enabling it to see vividly in dark environments or when optical zoom is used. Most of the digital cameras can increase the ISO number automatically in darker environment but this is always on the cost of more image noise. For best video recording, all of these factors have to be optimized.

   For example, we use a `Canon PowerShot SX280 HS` in our lab. This camera lets us choose only two: fps and resolution. These factors are also interconnected. Higher the fps, poorer is the resolution. All the other factors are either fixed in the hardware or are configured automatically. Increasing the ambient light automatically reduces image noise and increases the shutter speeds in all of the selectable fps modes. High level of ambient light and highest fps setting is recommended for capturing very fast objects e.g. a tossed coin. However, for capturing very small objects, high resolution is preferred over high fps.

   Using optical zoom reduces the amount of light which falls on the image sensor, increasing image noise. To use optical zoom, one must sufficiently increase the ambient light flux.

(f) Recording video in fluorescent tube lights or bulbs can sometimes confuse those cameras which have automatic iso compensation and automatic shutter speed selection. Canon PowerShot SX280 HS possesses this feature which may result in video flickering which drastically reduces the efficiency of point-tracking algorithm. Therefore LED lights fed on stable DC or ambient-sunlight can also be utilized to overcome this issue without having to change the fps.

2 Analyzing Data

2.1 Least-Squares Curve Fitting

Once we have acquired the data, we need to be able to describe the physics behind it. To test the fidelity of data with respect to the physical principles, collected either from video analysis or from a smartphone's sensor, we fit our acquired data to a suitable physical model. For the this purpose, we have prepared a generic function `PhysTrack.lsqCFit` based on MATLAB's built-in function `cftool`. Here is an example of how this tool is used.

```
function [fitResult, GoF] = PhysTrack.lsqCFit(xData, yData, dependent, model,
independent, optionalStartPoints)
% Returns a fit result and goodness of fit parameter.
%
% This function can be used to fit points with xData and yData on the model
% represented by symbol, "model".
%
% It uses cftool to compute a Fit object which is a struct containing the unknown
% variables from model and other fit  parameters.
%
% User can also specify some start points for fitting purpose.
%
% optionalStartPoints must be a linear array with length equal to the number of
% unknowns in the model.
```

For example, we have the time stamps in the array `tData` to serve as `xdata` and height measurements in the array `hData` to serve as `ydata`, and we want to fit this data to the equation.

$$h = h_0 + v_i t + \frac{1}{2} g t^2. \tag{1}$$

The dependent variable is `h` and the independent variable is `t`. The following code can be used to fit the data to selected model.

```
hFit = PhysTrack.lsqCFit(tData, hData, 'h', 'h0 + vi * t + 1/2*9.81*t^2', 't');
% If the fit process doesn't converge it is suggested that some start points
% are also given. For example, for the above example, there are two unknowns
% and thus two start points have to be given.
```

```
hFit = PhysTrack.lsqCFit(tData, hData, 'h', 'h0 + vi * t + 1/2*9.81*t^2', 't', ...
                [0.5, -1.8]);
```

The variable `hFit` is a `cfit` type variable. It not only contains the fitted values of unknown variables `h0` and `vi` but also the model function. We can use this variable as a function to find out the value of dependant variable from the fit result.

```
fittedH = hFit(t);
```

2.2 Numerical Differentiation

New physical quantities emerge from the calculation of derivatives (e.g. velocities from displacement data). Numerical differentiation can directly be performed on the acquired data. For this purpose, the generic function `deriv` has been developed which uses the four-point numerical differentiation method for accurate derivatives. The following code exemplifies usage of this function.

```
[xd, yd] = PhysTrack.deriv(xdata, ydata, order);
```

Here `xdata` and `ydata` are input datasets, `order` is the order of differentiation (1 for the first derivative or 2 for the second derivative). The variable `yd` contains the derivative of `ydata` with respect to `xdata`. Since differentiation changes the size of the data steps, dropping points here and there, one also needs to define new points along the independent axis (in this case, along x). The variable `xd` captures the essence of this interleaving process. See Figure 1 describing the outcome of the `deriv` function.

From Figure 1 it is also clear that the length of resulting dataset will be reduced due to the limitation of four point derivative algorithm. The first derivative formula used in this function is

$$\left(\frac{dy}{dx}\right)_i = \frac{2y_{i+3} - 9y_{i+2} + 18y_{i+1} - 11y_i}{6h}. \tag{2}$$

whereas the four-point second derivative formula used in this function is

$$\left(\frac{d^2y}{dx^2}\right)_i = \frac{-y_{i+3} + 4y_{i+2} - 5y_{i+1} + 2y_i}{h^2}. \tag{3}$$

where `h` is the common difference along the x dimension. Our differentiation code implicitly assumes that the original data is sampled at equal intervals.

2.3 Data Interpolation

Data Interpolation is sometimes also a necessary post-processing task. For example, numerical differentiation can be performed on the acquired data to obtain new physical quantities. This

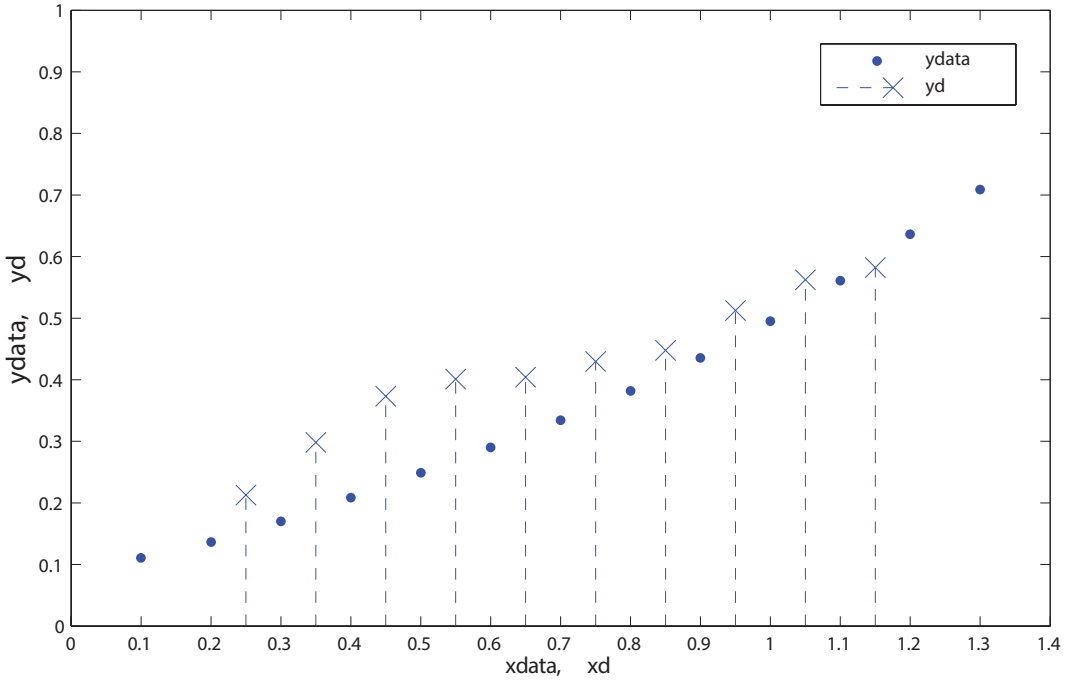


Figure 1: Example of input provided to the `PhysTrack.deriv` function and its output `yd` and `xd`. Note that the output `yd` is sampled at newly generated points `xd`.

reduces the number of datapoints. It may then be possible that the sampling rate may not be high enough for post-processing. This is a typical scenario where data interpolation comes into play.

Interpolation means creating new datapoints based on the available data. The most common method is through curve fitting. The acquired data is fitted to a suitable physical model and a time array is generated as per the requirement, e.g., step size can be 10× smaller for 10× interpolation. The data points are then calculated by finding the value of the fitted function at each timestamp thus generated.

The following MATLAB code demonstrates how the data acquired in a simple vertically oscillating object experiment is fit to the damped simple harmonic oscillator model and then interpolated.

```
% traj; % this variable has already been acquired using the KLT algorithm
% since we tracked only one point, it is convenient to extract the point from
% the trajectory.
traj = traj.tp1;
model = 'c1 * cos(c2 * t + c3) * exp(c4 * t) + c5'; % Damped SHM oscillator model
hFit = PhysTrack.lsqCFit(t, traj.y, 'h', model, 't'); % least-squares curve fitting
% interpolation on time array.
tInt = t(1):(t(2)-t(1))/factor:t(end);
% interpolation on H data.
fittedHData = hFit(tInt);
```

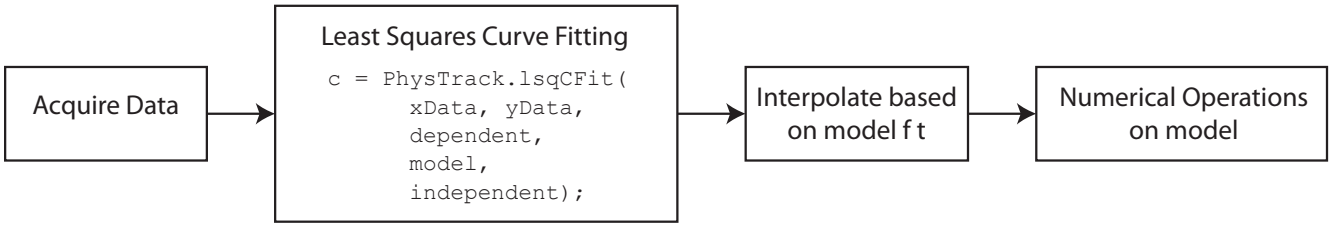


Figure 2: An example of sequence for post-processing the acquired data.

2.4 Some Other Useful Scripts

In this section, we summarize operations of some useful complementary scripts that are included in the package you've downloaded.

1. `TransformCart2Cart`, `InverseTransformCart2Cart` In computer graphics, points are conventionally defined in the coordinate system of the image. This coordinate system is Cartesian and centered on the upper left corner of the image, the `+x-axis` is along the right side and `+y-axis` is downwards. See Figure 3 for a depiction.

   Since the video frames used in video motion tracking represent real world scenes, the reference coordinate system of an image becomes inappropriate for defining points in the real world. For example, a certain point situated at the extreme upper left corner of the image will have coordinates $(0, 0)$ according to the reference coordinate system of the image, however the real world point situated at the same location may have coordinates $(3, -2.1)$, both in meters, when referred to some other real world coordinate system.

   One solution to this problem is to define a real world coordinate system referred to the image's coordinate system. See Figure 4 for example. The new reference coordinate system is defined by three points which are situated inside the image. Among these three points is the new origin $O'(280, 140)$, a point on the `+x-axis`, $A'(80, 190)$ and a point along the `+y-axis`, $B'(250, 60)$. The coordinates of these points are in pixels and are stated with respect to the image system. Clearly, to relate a certain distance between two points in the image to the real world distance, a unit conversion constant, pixels per meter (`ppm`) is also required.

   Hence, in the file initialization step, a real world 2D reference coordinate system is set up. The same system is used to define the object trajectories and distances in the motion tracking scripts. These trajectories are originally calculated from the video frames but are represented in real world coordinates instead. Since the videos are shot with a fixed camera, the same coordinate system can be used with all of the frames in the video. The file initialization process leaves two constants among all the others in the base workspace: `ppm` (pixels per meter) and `rwRCS` (real world Reference Coordinate System).

   In video motion analysis, one often needs to know the real and graphic coordinates of a point. For example, after performing some calculations on the object trajectories which were calculated using the motion tracking scripts, one may want to plot some results on the original video frames. Similarly, one may look at a certain point in the video frame and want

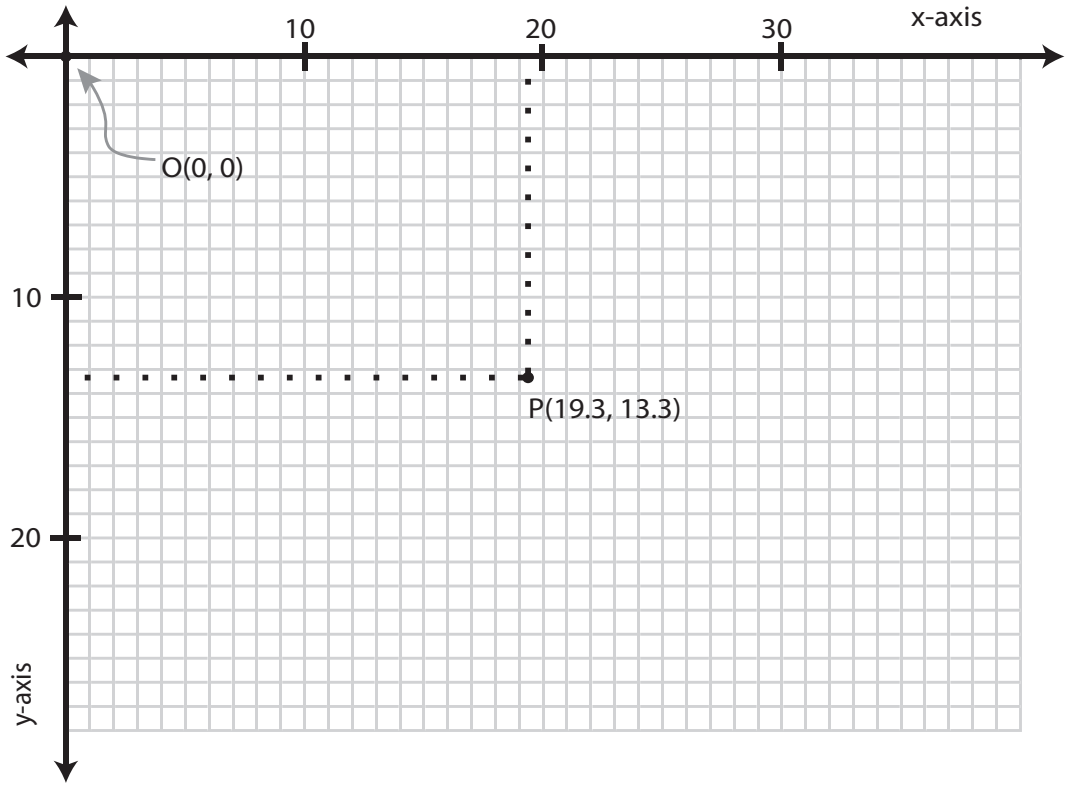


Figure 3: An illustration showing the coordinate system used to represent images. O is the origin and P is a general point with coordinates $(19.3, 13.3)$ inside the coordinate system.

to know its real coordinates. For this purpose, a transformation function has been defined that can be used to inter-convert points between the two coordinate frames.

To help with this transformation, two functions have been developed: `transformCart2Cart` and `inverseTransformCart2Cart`. `transformCart2Cart` can be used to calculate the real world Cartesian coordinates of a set of points using the image coordinates by providing the already calculated constants `rwRCS` and `ppm`. The syntax for using the function is quite simple. Also refer to Fig. 4 for definitions of `pImage` and `pReal` which are merely coordinates of the point in the two coordinate systems.

```
pReal = PhysTrack.TransformCart2Cart(pImage, rwRCS) / ppm;
% pImage must be an n x 2 dimensional array.PhysTrack.TransformCart2Cart can also
% work with structs extracted from video trackers.
% rwRCS is a 3x2 array of three points defining the real world coordinate system.
```

Similarly, knowing the real world Cartesian coordinates of a certain point, the following code can be used to determine the respective image coordinates.

```
pImage = PhysTrack.InverseTransformCart2Cart(pReal, rwRCS) * ppm;
% pReal must be an n x 2 dimensional array.
```

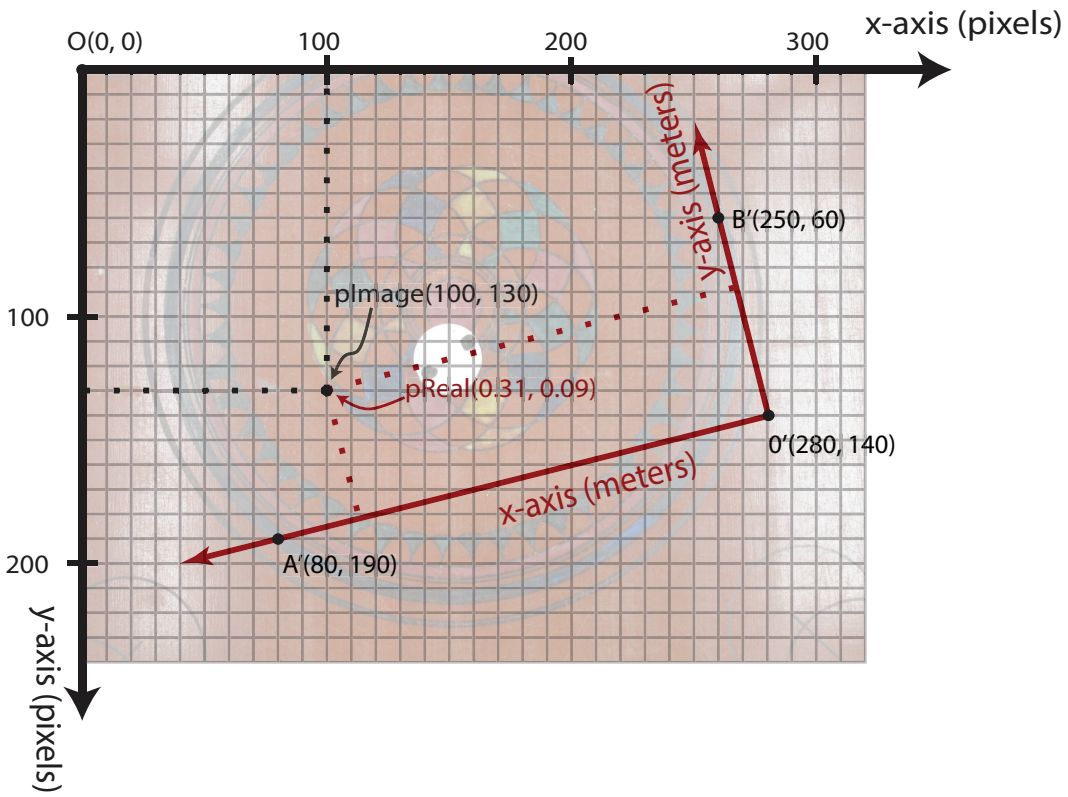


Figure 4: An illustration of reference coordinate system transformation.

2. `GetAngDispFrom2DtrackPoints(trackPoints1, trackPoints2)`

   In experiments where rotational motion is recorded and analyzed, the angular displacement is a vital parameter. It can be calculated by tracing the trajectories of two points on a rigid body rotating about an axis of rotation. Once these trajectories have been calculated using the video tracking algorithm, `getAngDispFrom2DtrackPoints` can be used to calculate the angular displacement of the rigid body.

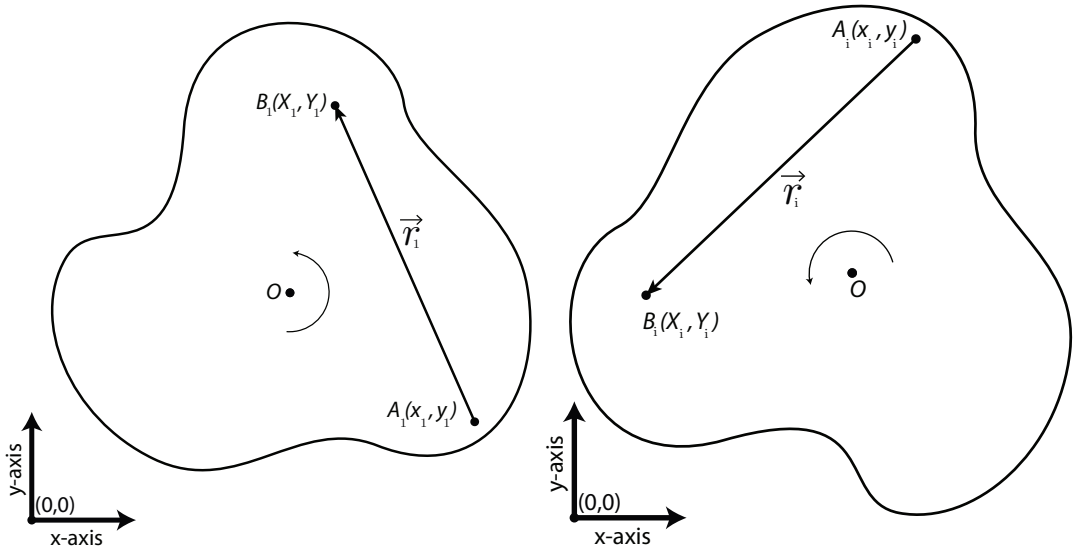
   Let us say that we have calculated the trajectories of a pair of points A and B which are situated on a rigid body rotating about a center of rotation, O. These trajectories will be in the form of arrays of order $n \times 2$ containing `x` and `y` coordinates of the positions of the points at n instances. We want to calculate the angular displacement of the rigid body about the center of rotation O at some instance t_i. See Figure 5 which shows the orientation of the body in the first frame and in some later frame (index i).

   Say the coordinates of point A are $(x_1, y_1; x_2, y_2; ...; x_N, y_N)$ and point B are $(X_1, Y_1; X_2, Y_2; ...; X_N, Y_N)$. Now, $\overrightarrow{r}_1$ defines an initial vector formed by the first pair of points.

$$\overrightarrow{r}_1 = (X_1 - x_1)\hat{\mathbf{i}} + (Y_1 - y_1)\hat{\mathbf{j}} \tag{4}$$

   where $\hat{\mathbf{i}}$ and $\hat{\mathbf{j}}$ are unit vectors in some reference frame. Further, $\overrightarrow{r}_i$ and $\overrightarrow{r}_{i-1}$ define two vectors formed by the same points in frames i and $i-1$ respectively.

$$\overrightarrow{r}_i = (X_i - x_i)\hat{\mathbf{i}} + (Y_i - y_i)\hat{\mathbf{j}}. \tag{5}$$



(a) Initial orientation of the rigid body. Vector $\overrightarrow{r}_1$ is defined by A_1 and B_1.

(b) Orientation of the rigid body at frame number i. Vector $\overrightarrow{r}_i$ is defined by A_i and B_i.

Figure 5: Orientation of the rigid body at frame number 1 and i. The point O represents the axis of rotation.

Let $\Delta\overrightarrow{\theta}_i$ be the angular displacement between two consecutive frames, i and $i-1$. It is defined by the angle between the vectors $\overrightarrow{r}_i$ and $\overrightarrow{r}_{i-1}$. (where $i \in [2, N]$). See Figure 6b which shows the vectors $\overrightarrow{r}_i$ and $\overrightarrow{r}_{i-1}$ in the same figure. Precisely,

$$\Delta\theta_i = \left|\Delta\overrightarrow{\theta}_i\right| = \measuredangle(\overrightarrow{r}_i, \overrightarrow{r}_{i-1}) = \frac{\cos^{-1}(\overrightarrow{r}_i \cdot \overrightarrow{r}_{i-1})}{|\overrightarrow{r}_i|\,|\overrightarrow{r}_{i-1}|}. \tag{6}$$

The total angular displacement of i^{th} pair of points from the first pair of points can be represented by:

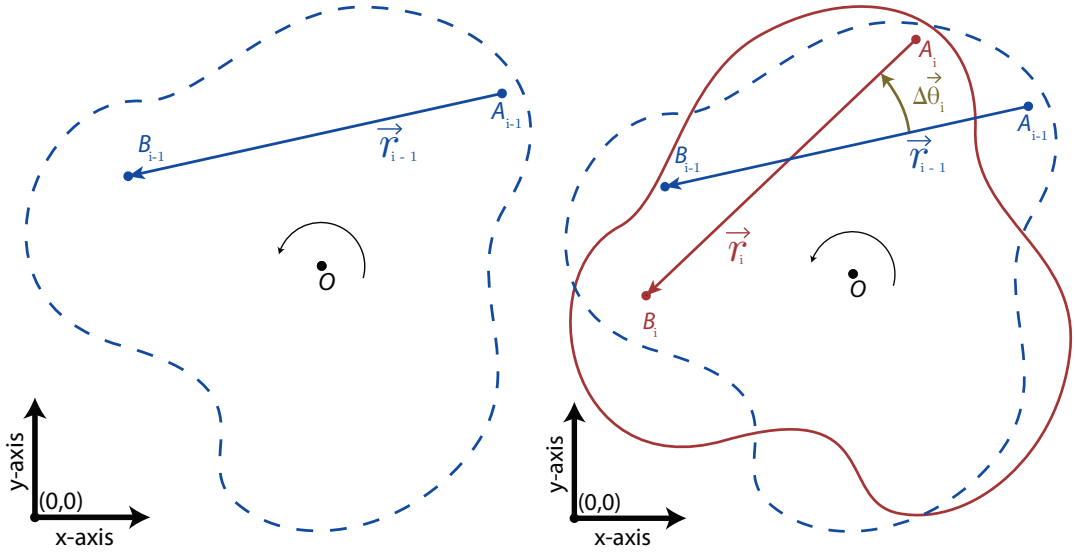
$$\Theta_i = \sum_{k=2}^{i} \Delta\theta_k \tag{7}$$

Clearly, the angular displacement for the first pair of points will be zero.

The script `getAngDispFrom2DtrackPoints` returns a column of angular displacements of the vectors defined by pairs of the two track points. Here is an example of using this function.

```
% trackPointA; trajectory of point A calculated using the video tracking
% algorithm. It may be an nx2 array or a struct.
% trackPointB; trajectory of point A calculated using the video tracking
% algorithm. It may be an nx2 array or a struct.
angDisp = PhysTrack.GetAngDispFrom2DtrackPoints(trackPointA, trackPointB);
```

It may be deduced from this explanation that if the trajectories of two fixed points on a rigid body are known, it is not necessary to define an axis of rotation. The script



(a) Orientation of the body at some time, t_{i-1}.

(b) The angle $\Delta\overrightarrow{\theta}$ representing the angular displacement between vectors r_i and r_{i-1}.

Figure 6: Angular displacement of the rigid body between two consecutive instances.

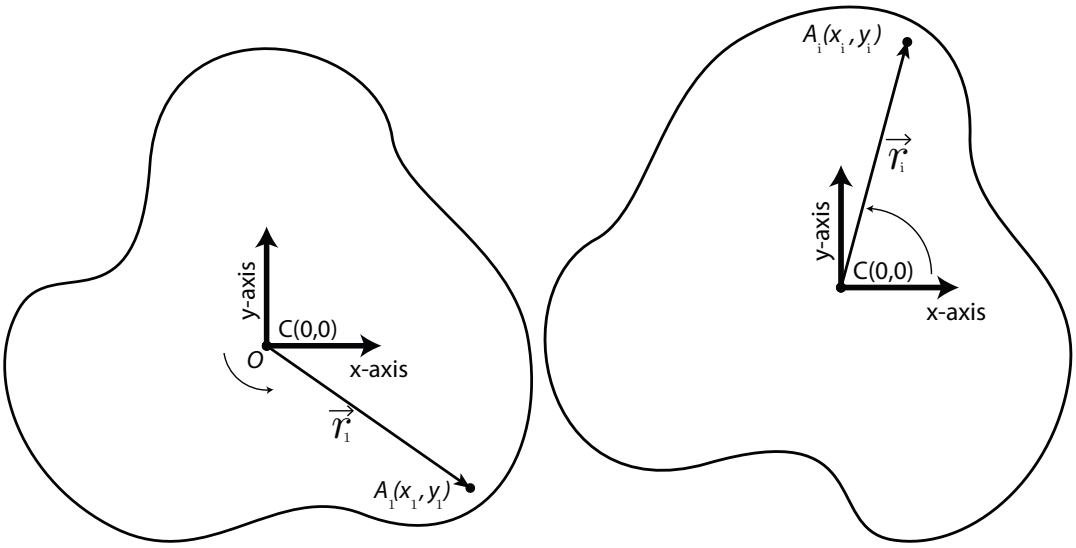
`getAngDispFrom2DtrackPoints` only requires the trajectories of any two fixed points on moving rigid body. However, if the trajectory of only one point is known, the algorithm would not be able to define the vector $\overrightarrow{r}$ using a single point and hence will not be able to calculate the angular displacement.

To solve this problem, if only one trajectory is given, the algorithm assumes that the object's axis of rotation passes through the origin of the real world reference coordinate system $C(0,0)$. So, the vector $\overrightarrow{r}$ is now defined by two points $C(0,0)$ and A. See Figure 7 which shows another rigid body with only one track point. Say the coordinates of point A are $(x_1, y_1; x_2, y_2; ...; x_N, y_N)$. Now, vector $\overrightarrow{r}_i$ can be defined in similar fashion.

$$\overrightarrow{r}_i = (x_i - 0)\hat{\mathbf{i}} + (y_i - 0)\hat{\mathbf{j}} = x_i\hat{\mathbf{i}} + y_i\hat{\mathbf{j}}. \tag{8}$$

Here is a script exemplifying the usage of `getAngDispFrom2DtrackPoints` using a single track point. You will use this format if the origin coincides with the axis of rotation.

```
% trackPointT; % trajectory of point A calculated using the video tracking
% algorithm.
angDisp = PhysTrack.GetAngDispFrom2DtrackPoints(trackPointT);
```



(a) Initial orientation of the rigid body. Vector $\overrightarrow{r}_1$ is defined by A_1 and fixed point $C(0,0)$.

(b) Orientation of the rigid body at frame number i. Vector $\overrightarrow{r}_i$ is defined by A_i and fixed point$C(0,0)$.

Figure 7: An illustration showing how a single point can be used to measure the angular displacement.

3 Further Reading and References

References

[1] PhysTrack source code on GitHub, alongwith sample codes and videos. `https://goo.gl/4qchpJ`

[2] PhysTrack wiki. `https://goo.gl/hVyWih`

[3] Lucas, Bruce D. and Takeo Kanade. "An Iterative Image Registration Technique with an Application to Stereo Vision", Proceedings of the 7th International Joint Conference on Artificial Intelligence, pp. 674–679 (1981).

[4] "Video motion analysis with automated tracking: An insight", European Journal of Physics, Eur. J. Phys. 36 (2015) 065049 (13pp)