SyntheticTrajectoryGeneration Documentation

Robin Jacobs

May 2024

1 Overview

This document contains an extended documentation for the `SyntheticTrajectoryGeneration` project [1].

1.1 Setup

The modeled scene consists of a user-defined number of cameras placed around a working volume. This allows to model the setup developed by Jonas Hein, where up to 5 Azure Kinects are present[2]. Additionally a optical tracker is present in the scene. In Jonas Hein's setup this corresponds to a FusionTrack500 system from Atracsys. This system uses a stereo camera setup to detect *markers* in space, which are composed of a number of infrared reflective spheres also known as *fiducials*. The detection are then used to estimate the pose of the markers. In our case, one of these markers is mounted on the checkerboard used for camera calibration. The checkerboard follows a user-defined trajectory, both in terms of position and orientation inside the working volume. In practice the working volume is limited by the constraints imposed by the sensors, e.g., for FusionTrack500 it limited by a view frustum of maximum depth of approximately 2.7m, 1.8m width and 1.3m height. A schematic view of the setup in the real world and in this simulation can be found in Figure **??** and 1.

1.2 Description

This Python package provides functionalities for generating synthetic checkerboard trajectories. It enables the user to:

- Generate synthetic trajectories of checkerboards.
- Project the generated checkerboards points onto the coordinate systems of different cameras and tracker frames.
- Save the generated trajectories for using it in e.g., for multi camera calibration.
- Control the sampling frequencies for both cameras and trackers.
- Introduce artificial recording time offsets to simulate discrepancies between camera and tracker data.

[1] The repository of this project can be found at `https://github.com/BAL-ROCS-BUT-COOL/SyntheticTrajectoryGeneration`

[2] see `https://arxiv.org/html/2305.03535v2`

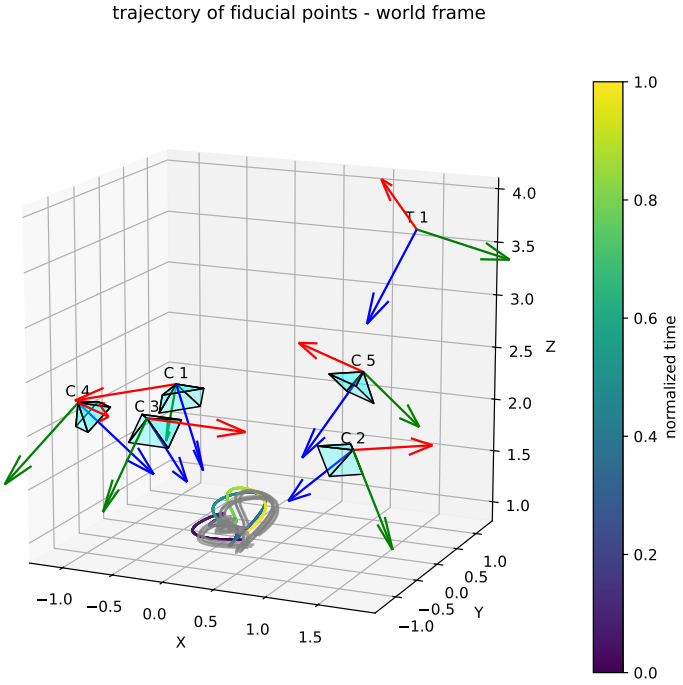


Figure 1: Setup overview. 5 Camera frames and frustums and 1 Tracker frame together with the checkerboard origin trajectory is shown.

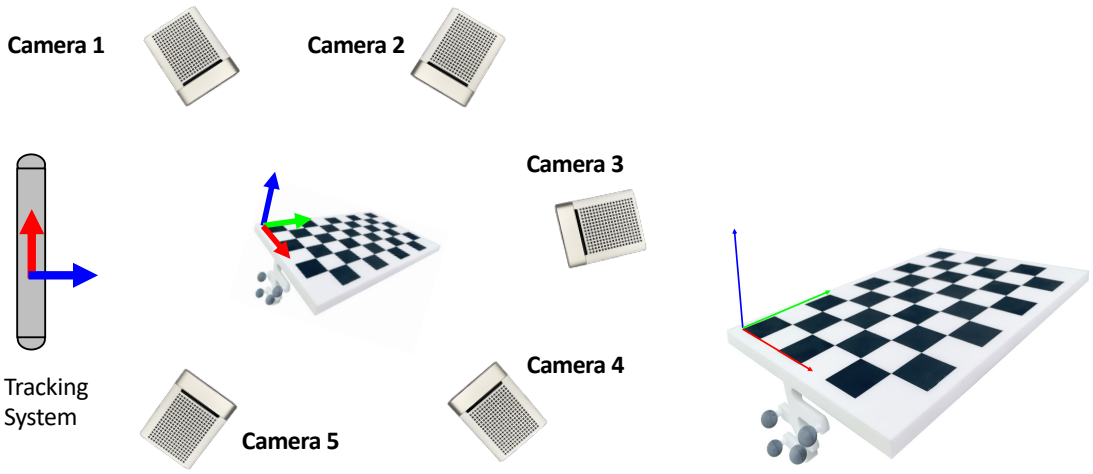
2 Checkerboard

We support checkerboards of variable sizes. Initialization is achieved using the `Checkerboard` class, an example of a checkerboard showing the numbering of keypoints on the board can be seen in Figure 3a. In the following we define the checkerboard origin to be M and the frame $\mathcal{M}$. Additionally, our checkerboard description also includes the geometry of fiducial markers mounted on the plane used by the tracker.

3 Trajectory Generation

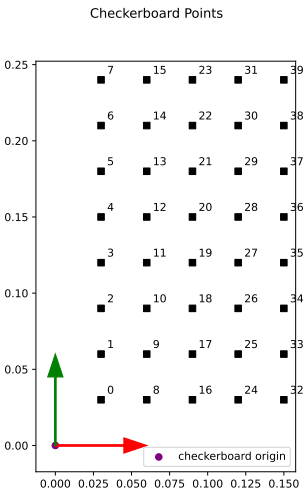
The core functionality for trajectory generation resides within the `generation.py` script. Trajectory definitions are specified using the Python-based configuration class `GeneratorConfig`. This class encapsulates the following sub-configurations:

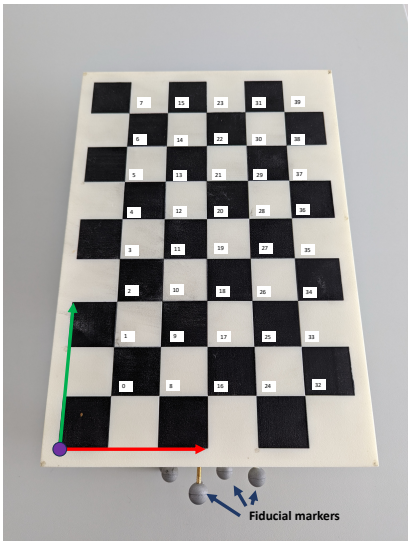
- **Position Trajectory**: Defines the path the checkerboard origin will follow in 3D space.
- **Orientation Trajectory** Specifies the checkerboard's frame rotational changes throughout the motion.
- **Camera and Optical Tracker** configures parameters related to camera and tracker data generation.



(a) Setup (b) Checkerboard model with fiducials

Figure 2: Schematic of tracker, cameras and checkerboard





(a) Checkerboard points and origin representation in code (b) Real checkerboard

- **Time**: Controls the timing and duration of the trajectory, as well as temporal sampling.

Note that in the current implementation the generation of position and orientation trajectories are treated to be independent from each other. In the following, the individual modules are explained.

3.1 Time

The user can define a starting time for the trajectory t_{start}, and an end time t_{end}. Additionally the sampling frequencies of the cameras f_{C} and tracker f_{T} need to be set.

Trajectories are internally calculated using a simulation sampling frequency which is calculated to be $f_{\text{sim}} = \text{lcm}(f_{\text{C}}, f_{\text{T}})$, where lcm denotes the least common multiple of both arguments. This ensures that all sensors can be simulated using the correct sampling frequency. Note this mandates that both f_{C} and f_{T} are set to be integers.

When evaluating the trajectory functions, a normalized time is used $t_{\text{norm}} = \frac{t_{\text{end}} - t_{\text{start}}}{t_{\text{start}}}$.

3.2 Position Trajectory

To describe the 3D checkerboard origin position trajectory, a curve function $\mathbf{p}_c(t_{\text{norm}})$ must be provided mapping normalized time to 3D positions. Currently, 2 different curve types are implemented: B-spline-based and a simple Helix curve function. Other curves can be easily implemented by following the same function-based formulation.

3.2.1 Helix

$$\mathbf{p}_c^{\text{helix}}(t) = c \cdot \begin{pmatrix} a\cos(wt) \\ a\sin(wt) \\ bt \end{pmatrix} \tag{1}$$

where w, a, b, c controls the frequency, radius, z height increase rate and scale respectively. Figure 4 shows an example.

3.2.2 B-Spline

To achieve more complex trajectories that closely resemble real-life paths while maintaining continuity and smoothness in three-dimensional space, we have implemented B-spline-based curve generation. By approximating a curve through a number of 3D sample vectors in space. The implementation is based on scipy's `splprep` function[3] An example trajectory is plotted in Figure 5.

Randomization To introduce randomness into the curve, we have employed two randomization methods for randomizing the sample vectors: random walk and uniform sampling within a designated working volume. Specifically:

- The **random walk** samples N points sequentially from a initial point $\mathbf{x}_0$. Subsequent points are generated using the recursive rule $\mathbf{x}_i = \mathbf{x}_{i-1} + \gamma$ where γ is uniformly drawn from 6 axis direction $(\pm\mathbf{e}_x, \pm\mathbf{e}_y, \pm\mathbf{e}_z)$.
- The **uniform sampling** samples N points inside a box-shaped working volume.

[3] see `https://docs.scipy.org/doc/scipy/reference/generated/scipy.interpolate.splprep.html`

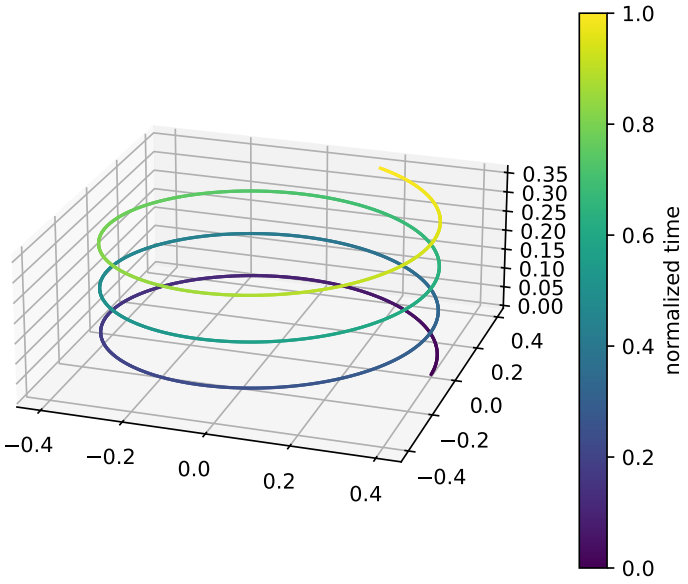


Figure 4: Example checkerboard origin trajectory using the helix function

3.3 Orientation Trajectory

As with the position trajectory, the orientation trajectory is defined using a curve function. Since orientation lives in SO(3) interpolation between different control points is more involved. Due to this currently we are only supporting simple interpolation schemes relying on defining a start and end orientation and interpolating in between a define start orientation R_0 and a goal orientation R_1. As with position trajectories, we want to have a function mapping normalized time to orientation of the checkerboard, i.e., $\mathbf{q}(t_{\text{norm}}) \in \text{SO}(3)$

3.3.1 Slerp

Spherical linear interpolation (slerp) is a method for allowing for smooth interpolation betwent two rotations. We use scipy's Slerp implementation[4] for interpolating between start and end orientation.

Note. Scipy's slerp interpolation supports interpolation between more than two rotations at the same time which we currently do not use.

3.4 Camera and Tracker

The current implementation supports up to N cameras and one optical tracker sensor.

[4]https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.transform.Slerp.html

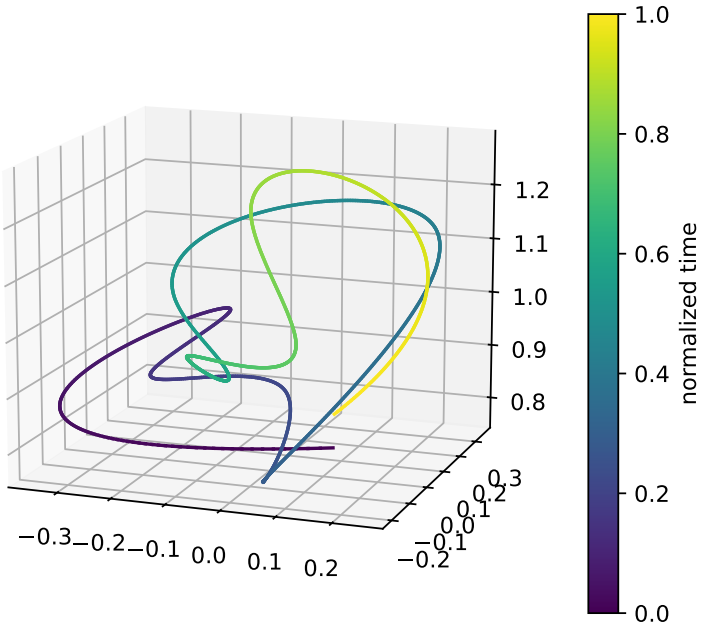


Figure 5: Example checkerboard origin trajectory using the b-spline function

3.4.1 Cameras

Cameras are defined using an instance of `CamerasConfig`. For each camera, extrinsic and intrinsic parameters need to be provided. The camera extrinsics consist of a rotation vector $\mathbf{v}_{\mathcal{CW}} \in \mathrm{SO}(3)$ transforming world frame coordinates to camera frame, and a vector pointing from camera center to world origin and which expressed in the camera frame ${}_{\mathcal{C}}\mathbf{r}_{CW} \in \mathbb{R}^3$. The intrinsics values $[f_x, f_y, c_x, c_y]$ are the focal point and the principal point in x, y image coordinates, such that the calibration matrix is written:

$$\mathsf{P} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} [\mathsf{R}_{\mathcal{CW}} | {}_{\mathcal{C}}\mathbf{r}_{CW}] \tag{2}$$

Note. Nonlinear effects in the camera projection such as distortions are currently not supported.

Pose Randomization Camera pose's can be randomly sampled using `sample_camera_extrinsics`, which uniformly samples camera position inside a cylindrical surface above the working volume. The camera's orientation is set so that the z-axis of the camera frame will point towards the working volume's center and the x-axis is parallel to the xy plane.

Projections Points projections are calculated following standard pinhole-based camera projection model, i.e.,

$$\mathbf{p} = \mathsf{P}\mathbf{X} \tag{3}$$

where $\mathbf{X}$ are the 3D Points to be projected. We save 2D projections in either pixel coordinates and/or ideal coordinates.

Note. We do not discretize pixel coordinates of the points.

Occlusion To achieve a degree of physical realism within the simulation, we support some basic physical occlusion detection for the cameras. This means, when the checkerboard faces a specific camera with its back, 2D points are not rendered. This is achieved by testing whether or not the line of sight vector between camera center and checkerboard origin $\mathbf{r}_{CM}$ opposes the normal vector of the checkerboard plane $\mathbf{e}_z^{\mathcal{M}}$, i.e., we test

$$\mathbf{r}_{CM} \cdot \mathbf{e}_z^{\mathcal{M}} < 0 \tag{4}$$

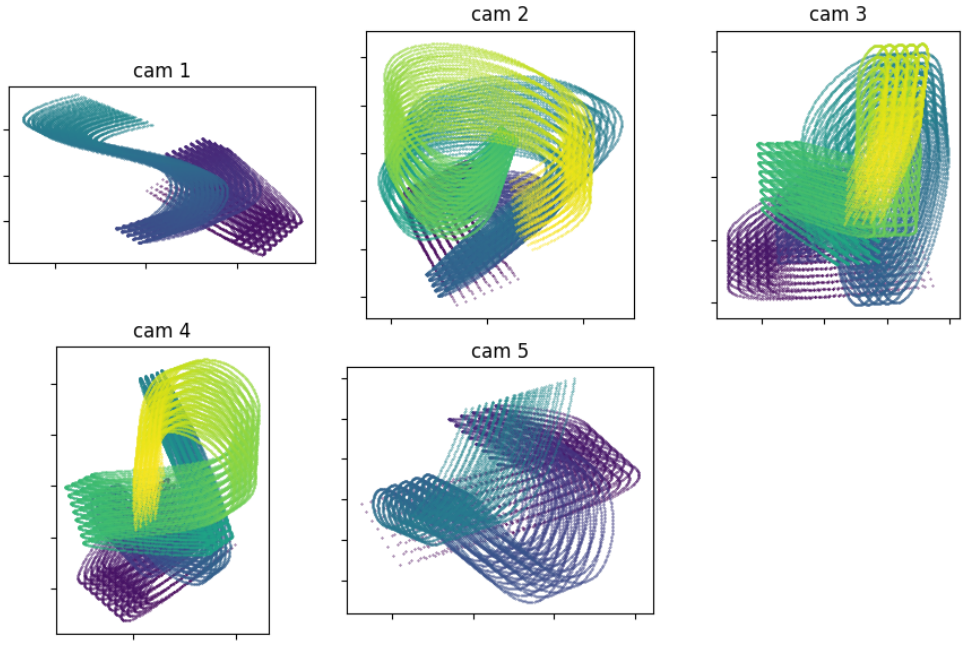


Figure 6: Example camera projection of checkerboard points trajectory using the b-spline function.

Clipping (Only relevant when use pixel coordinates). Cut off projected points lying outside the specified image boundaries.

3.4.2 Optical Tracker

A tracker is implemented as an additional frame denoted $\mathcal{T}$ to which all the checkerpoint points and fiducial points as well as cameras are calculated. It does not use any projections. The tracker frame pose follows the same convention as the camera extrinsics, i.e., $[\mathbf{v}_{\mathcal{TW},\mathcal{C}}\, \mathbf{r}_{TW}] \in \mathrm{SE}(3)$ and is set in `world_to_tracker_tf`.

4 Algorithm

A simplified overview on how the generation algorithm works can be found in Algorithm 1

Algorithm 1: Simplified `generate()`

```
for t = t_start, ... t_end do
    Calculate checkerboard pose in W, T frame using parametrized curve;
    Calculate checkerboard points in W, T frame using pose and checkerboard geometry;
    for camera in cameras do
        Project each checkerboard point in camera frame;
        Remove occluded points;
        Clip points;
    end
end
```

5 Time offset

Since in practice there is usually a temporal offset between device clocks, i.e., especially between the optical tracker and the cameras, one needs to account for this when calibrating individual devices, which is e.g., done in Jona's work. To simulate this temporal offset, the variable `cameras_t_offset_s` can be set in the configuration. This variable is a list, specifying for each camera its time offset with respect to the global simulation clock[5]. A non-zero offset Δt_{offset} as an entry in the list will result in the camera recordings (i.e., projections) starting at $t_{0,\text{cam}} + \Delta t_{\text{offset}}$ instead of $t_{0,\text{cam}}$ while the returned camera device time will not account for this offset and still start at $t_{0,\text{cam}}$ leading to a difference between devices clocks of the cameras and if used to the optical tracker, which we set to the global simulation clock.

Note. Whether or not the exact time offset can be achieved depends on whether Δt_{offset} can be attained exactly by the cameras sampling interval (related to f_C)

[5]This means if we want to model intra camera offset we need to formulate them with respect to a global reference time offset first