bundle\_adjustment实例及pytorch3d代码

[1. Install and Import Modules 2](#_Toc178514503)

[2. Set up Cameras and load ground truth positions 2](#_Toc178514504)

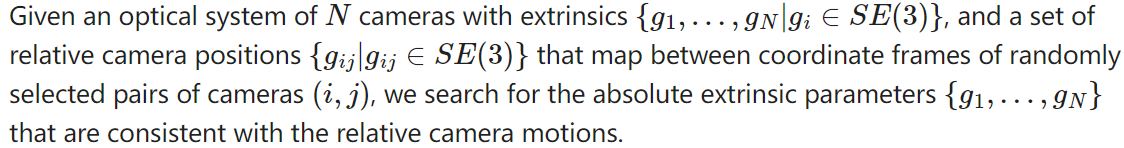
[3. Define optimization functions 3](#_Toc178514505)

[4. Optimization 4](#_Toc178514506)

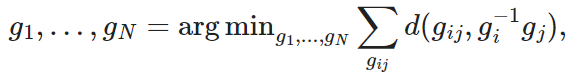
[5. Conclusion 5](#_Toc178514507)

[6. camera\_visualization 5](#_Toc178514508)

This tutorial showcases the cameras, transforms and so3 API. The problem we deal with is defined as follows:

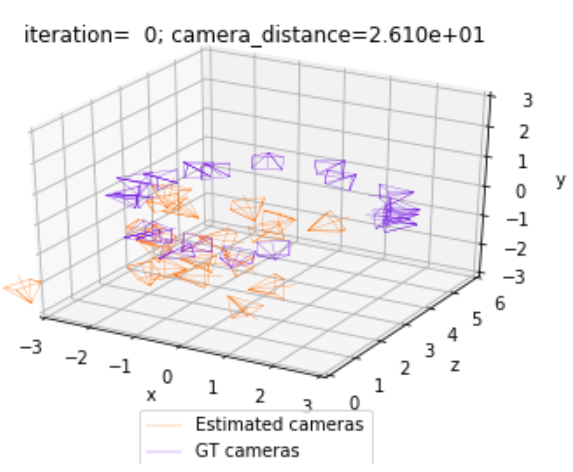


More formally:

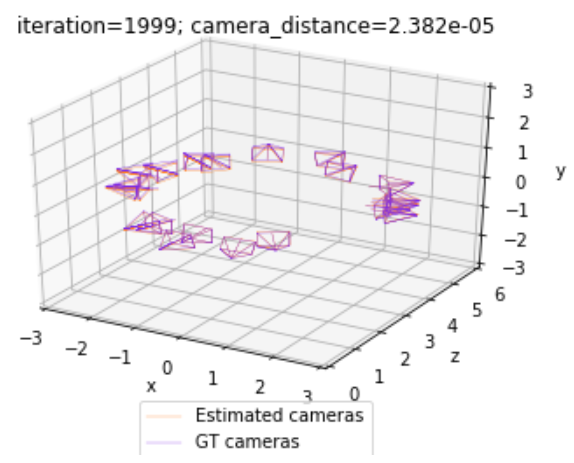


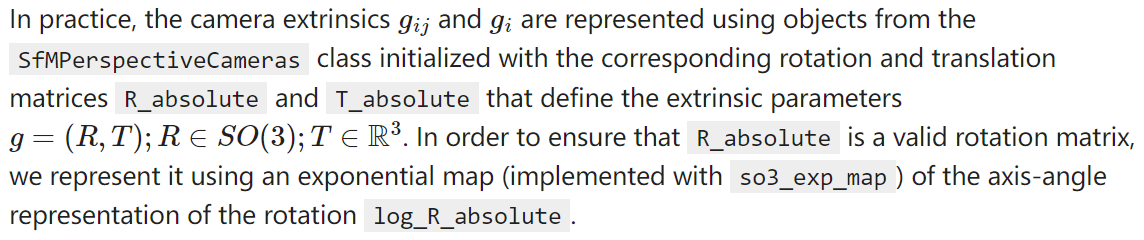


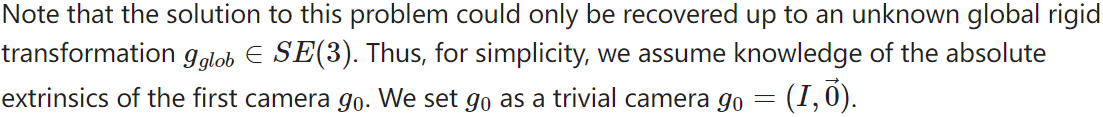
Visually, the problem can be described as follows. The picture below depicts the situation at the beginning of our optimization. The ground truth cameras are plotted in purple while the randomly initialized estimated cameras are plotted in orange:



Our optimization seeks to align the estimated (orange) cameras with the ground truth (purple) cameras, by minimizing the discrepancies between pairs of relative cameras. Thus, the solution to the problem should look as follows:







# Install and Import Modules

Ensure torch and torchvision are installed. If pytorch3d is not installed, install it using the following cell:

# imports

import torch

from pytorch3d.transforms.so3 import (

so3\_exp\_map,

so3\_relative\_angle,

)

from pytorch3d.renderer.cameras import (

SfMPerspectiveCameras,

)

# add path for demo utils

import sys

import os

from camera\_visualization import plot\_camera\_scene

sys.path.append(os.path.abspath(''))

device = torch.device("cpu") # device = torch.device("cuda:0")

# If using \*\*Google Colab\*\*, fetch the utils file for plotting the camera scene, and the ground truth camera positions:

# OR if running \*\*locally\*\* uncomment and run the following cell:

# from utils import plot\_camera\_scene

# Set up Cameras and load ground truth positions

# load the SE3 graph of relative/absolute camera positions

camera\_graph\_file = './data/camera\_graph.pth'

camera\_graph\_file = './camera\_graph.pth'

(R\_absolute\_gt, T\_absolute\_gt), (R\_relative, T\_relative), relative\_edges = torch.load(camera\_graph\_file)

# create the relative cameras

cameras\_relative = SfMPerspectiveCameras(

R = R\_relative.to(device),

T = T\_relative.to(device),

device = device,

)

# create the absolute ground truth cameras

cameras\_absolute\_gt = SfMPerspectiveCameras(

R = R\_absolute\_gt.to(device),

T = T\_absolute\_gt.to(device),

device = device,

)

# the number of absolute camera positions

N = R\_absolute\_gt.shape[0]

# Define optimization functions

# ### Relative cameras and camera distance

# We now define two functions crucial for the optimization.#

# \*\*`calc\_camera\_distance`\*\* compares a pair of cameras. This function is important as it defines the loss that we are minimizing. The method utilizes the `so3\_relative\_angle` function from the SO3 API.

# \*\*`get\_relative\_camera`\*\* computes the parameters of a relative camera that maps between a pair of absolute cameras. Here we utilize the `compose` and `inverse` class methods from the PyTorch3D Transforms API.

def calc\_camera\_distance(cam\_1, cam\_2):

"""

Calculates the divergence of a batch of pairs of cameras cam\_1, cam\_2.

The distance is composed of the cosine of the relative angle between the rotation components of the camera extrinsics and the l2 distance between the translation vectors.

"""

# rotation distance

R\_distance = (1.-so3\_relative\_angle(cam\_1.R, cam\_2.R, cos\_angle=True)).mean()

# translation distance

T\_distance = ((cam\_1.T - cam\_2.T)\*\*2).sum(1).mean()

# the final distance is the sum

return R\_distance + T\_distance

def get\_relative\_camera(cams, edges):

"""

For each pair of indices (i,j) in "edges" generate a camera that maps from the coordinates of the camera cams[i] to the coordinates of the camera cams[j]

"""

# first generate the world-to-view Transform3d objects of each

# camera pair (i, j) according to the edges argument

trans\_i, trans\_j = [

SfMPerspectiveCameras(

R = cams.R[edges[:, i]],

T = cams.T[edges[:, i]],

device = device,

).get\_world\_to\_view\_transform()

for i in (0, 1)

]

# compose the relative transformation as g\_i^{-1} g\_j

trans\_rel = trans\_i.inverse().compose(trans\_j)

# generate a camera from the relative transform

matrix\_rel = trans\_rel.get\_matrix()

cams\_relative = SfMPerspectiveCameras(

R = matrix\_rel[:, :3, :3],

T = matrix\_rel[:, 3, :3],

device = device,

)

return cams\_relative

# Optimization

# Finally, we start the optimization of the absolute cameras.

# We use SGD with momentum and optimize over `log\_R\_absolute` and `T\_absolute`.

# As mentioned earlier, `log\_R\_absolute` is the axis angle representation of the rotation part of our absolute cameras. We can obtain the 3x3 rotation matrix `R\_absolute` that corresponds to `log\_R\_absolute` with:

# `R\_absolute = so3\_exp\_map(log\_R\_absolute)`

# initialize the absolute log-rotations/translations with random entries

log\_R\_absolute\_init = torch.randn(N, 3, dtype=torch.float32, device=device)

T\_absolute\_init = torch.randn(N, 3, dtype=torch.float32, device=device)

# furthermore, we know that the first camera is a trivial one

# (see the description above)

log\_R\_absolute\_init[0, :] = 0.

T\_absolute\_init[0, :] = 0.

# instantiate a copy of the initialization of log\_R / T

log\_R\_absolute = log\_R\_absolute\_init.clone().detach()

log\_R\_absolute.requires\_grad = True

T\_absolute = T\_absolute\_init.clone().detach()

T\_absolute.requires\_grad = True

# the mask the specifies which cameras are going to be optimized

# (since we know the first camera is already correct,

# we only optimize over the 2nd-to-last cameras)

camera\_mask = torch.ones(N, 1, dtype=torch.float32, device=device)

camera\_mask[0] = 0.

# init the optimizer

optimizer = torch.optim.SGD([log\_R\_absolute, T\_absolute], lr=.1, momentum=0.9)

# run the optimization

n\_iter = 8000 # fix the number of iterations

for it in range(n\_iter):

# re-init the optimizer gradients

optimizer.zero\_grad()

# compute the absolute camera rotations as

# an exponential map of the logarithms (=axis-angles)

# of the absolute rotations

R\_absolute = so3\_exp\_map(log\_R\_absolute \* camera\_mask)

# get the current absolute cameras

cameras\_absolute = SfMPerspectiveCameras(

R = R\_absolute,

T = T\_absolute \* camera\_mask,

device = device,

)

# compute the relative cameras as a composition of the absolute cameras

cameras\_relative\_composed = get\_relative\_camera(cameras\_absolute, relative\_edges)

# compare the composed cameras with the ground truth relative cameras

# camera\_distance corresponds to $d$ from the description

camera\_distance = calc\_camera\_distance(cameras\_relative\_composed, cameras\_relative)

# our loss function is the camera\_distance

camera\_distance.backward()

# apply the gradients

optimizer.step()

# plot and print status message

if it % 200==0 or it==n\_iter-1:

status = 'iteration=%3d; camera\_distance=%1.3e' % (it, camera\_distance)

plot\_camera\_scene(cameras\_absolute, cameras\_absolute\_gt, status)

print('Optimization finished.')

# Conclusion

# In this tutorial we learnt how to initialize a batch of SfM Cameras, set up loss functions for bundle adjustment, and run an optimization loop.

# camera\_visualization

import matplotlib.pyplot as plt

from mpl\_toolkits.mplot3d import Axes3D # noqa: F401 unused import

from pytorch3d.vis.plotly\_vis import get\_camera\_wireframe

def plot\_cameras(ax, cameras, color: str = "blue"):

"""

Plots a set of `cameras` objects into the maplotlib axis `ax` with color `color`.

"""

# cam\_wires\_canonical = get\_camera\_wireframe().cuda()[None]

cam\_wires\_canonical = get\_camera\_wireframe()[None]

cam\_trans = cameras.get\_world\_to\_view\_transform().inverse()

cam\_wires\_trans = cam\_trans.transform\_points(cam\_wires\_canonical)

plot\_handles = []

for wire in cam\_wires\_trans:

# the Z and Y axes are flipped intentionally here!

x\_, z\_, y\_ = wire.detach().cpu().numpy().T.astype(float)

(h,) = ax.plot(x\_, y\_, z\_, color=color, linewidth=0.3)

plot\_handles.append(h)

return plot\_handles

def plot\_camera\_scene(cameras, cameras\_gt, status: str):

"""

Plots a set of predicted cameras `cameras` and their corresponding ground truth locations `cameras\_gt`. The plot is named with a string passed inside the `status` argument.

"""

# print("----------------tpxxx-------------")

fig = plt.figure()

ax = fig.add\_subplot(projection="3d")

ax.clear()

ax.set\_title(status)

# print("----------------tpxxx7-------------")

handle\_cam = plot\_cameras(ax, cameras, color="#FF7D1E")

handle\_cam\_gt = plot\_cameras(ax, cameras\_gt, color="#812CE5")

plot\_radius = 3

# print("----------------tpxxx8-------------")

ax.set\_xlim3d([-plot\_radius, plot\_radius])

ax.set\_ylim3d([3 - plot\_radius, 3 + plot\_radius])

ax.set\_zlim3d([-plot\_radius, plot\_radius])

ax.set\_xlabel("x")

ax.set\_ylabel("z")

ax.set\_zlabel("y")

labels\_handles = {

"Estimated cameras": handle\_cam[0],

"GT cameras": handle\_cam\_gt[0],

}

ax.legend(

labels\_handles.values(),

labels\_handles.keys(),

loc="upper center",

bbox\_to\_anchor=(0.5, 0),

)

plt.show()

plt.close()

return fig