

دانشگاه صنعتی امیرکبیر (پلی تکنیک تهران)

تمرين سري هفتم

آشنایی با روش های تخمین عمق، بازسازی سه بعدی و سنسور لایدار

مهندی فیروزبخت

۱۴۰۱۳۱۰۲۷

استاد درس : دکتر رضا صفابخش

۱۴۰۱
دانشگاه امیرکبیر
گروه هوش مصنوعی

سوال الف

برای سوال الف ، ابتدا لازم است تا داده ها را بارگذاری نماییم. برای اینکار ابتدا داده های اصلی را درون گوگل درایو آپلود میکنیم. سپس این داده ها را درون گوگل کولب بارگذاری میکنیم. برای اینکار از کد زیر استفاده میکنیم :

```
# Load the Drive helper and mount
from google.colab import drive
drive.mount('/content/drive')
import os

os.chdir("/content/drive/MyDrive/Colab Notebooks/CV_HW07"+
          "/kitti_sample/2011_09_29/2011_09_29_drive_0026_sync/velodyne_points/data")
!ls

0000000000.bin 0000000001.bin 0000000002.bin 0000000003.bin
```

قابل مشاهده است که به ۴ فایل bin دسترسی پیدا کرده ایم. سپس با استفاده از توابع open3D فایل ها را میتوانیم استفاده کنیم. برای اینکار ابتدا کتابخانه open3d را بر روی کولب نصب میکنیم. همچنین کتابخانه های مورد نیاز را بارگذاری میکنیم :

```
!pip install open3d

!python -c "import open3d as o3d; print(o3d.__version__)"

0.16.0

import struct
import open3d as o3d
import numpy as np
```

در مرحله بعدی تابع ای با نام convert_kitti_bin_to_pcd مینویسم که این تابع با دسترسی به فایل مورد نظر آن را خوانده و به فرمت قابل استفاده از آن در کتابخانه open3D در خواهد آورد. کد آن به صورت زیر خواهد بود:

```
def convert_kitti_bin_to_pcd(binFilePath):
    size_float = 4
    list_pcd = []
    with open(binFilePath, "rb") as f:
        byte = f.read(size_float * 4)
        while byte:
            x, y, z, intensity = struct.unpack("ffff", byte)
            list_pcd.append([x, y, z])
            byte = f.read(size_float * 4)
    np_pcd = np.asarray(list_pcd)
    pcd = o3d.geometry.PointCloud()
    pcd.points = o3d.utility.Vector3dVector(np_pcd)
    return pcd
```

سؤال الف

در این تابع ابتدا فایل مورد نظر را میخواند. سپس با استفاده از تابع `unpack` نقاط را استخراج میکند. سپس با استفاده از توابع `numpy.Vector3dVector` و `geometry.PointCloud` از حالت `utility` به فرمت `PointCloud` تبدیل میشود. نتیجه برای هر ۴ داده به صورت زیر است :

<pre>pcd1 = convert_kitti_bin_to_pcd("0000000001.bin") # show print(pcd1) print(np.asarray(pcd1.points)) PointCloud with 118987 points. [[74.66999817 12.2329998 2.76799989] [74.76300049 12.48999977 2.77200007] [74.39600372 12.66899967 2.76099992] ... [3.69000006 -1.38900006 -1.722] [3.70900011 -1.38300002 -1.73000002] [3.70799994 -1.37600005 -1.72800004]]</pre>	<pre>pcd0 = convert_kitti_bin_to_pcd("0000000000.bin") # show print(pcd0) print(np.asarray(pcd0.points)) PointCloud with 119339 points. [[75.37000275 12.34799957 2.79200006] [75.46499634 12.60700035 2.796] [75.38200378 12.71500015 2.79399991] ... [16.78800011 -2.15799999 -7.77600002] [16.78899956 -2.10400009 -7.77400017] [16.8029995 -2.079 -7.77899981]]</pre>
<pre>pcd3 = convert_kitti_bin_to_pcd("0000000003.bin") # show print(pcd3) print(np.asarray(pcd3.points)) PointCloud with 118056 points. [[73.46199799 12.39000034 2.72799993] [73.53299713 12.64000034 2.73200011] [73.19799805 12.81900024 2.72199988] ... [14.46100044 -3. -6.77099991] [14.45800018 -2.9519999 -6.76499987] [14.69299984 -2.9519999 -6.87200022]]</pre>	<pre>pcd2 = convert_kitti_bin_to_pcd("0000000002.bin") # show print(pcd2) print(np.asarray(pcd2.points)) PointCloud with 118491 points. [[74.03500366 12.24800014 2.74699998] [74.05000305 12.48999977 2.74799991] [73.91600037 12.58699989 2.74499989] ... [15.06299973 -3.0769999 -7.05299997] [15.04100037 -3.023 -7.03800011] [15.24800014 -3.03999996 -7.13399982]]</pre>

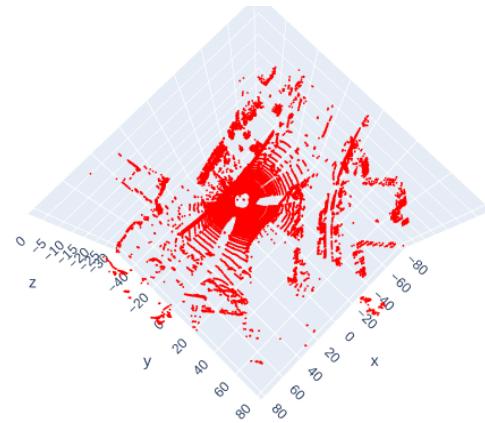
در این برنامه طبق صحبت های بیان شده در پیوست فایل ، مقادیر X نشان دهنده فاصله از سنسور و مقادیر Z مقادیر عمودی و مقادیر Y مقادیر افقی هستند. برای تولید تصاویر در زوایای مختلف از تابع زیر استفاده میکنیم. تصاویر با استفاده از این تابع قابل چرخاندن است پس در زوایای از بالا ، ۴۵ درجه و به حالتی که مقدار X ثابت باشد گرفته میشود :

```
o3d.visualization.draw_plotly([pcd0])
```

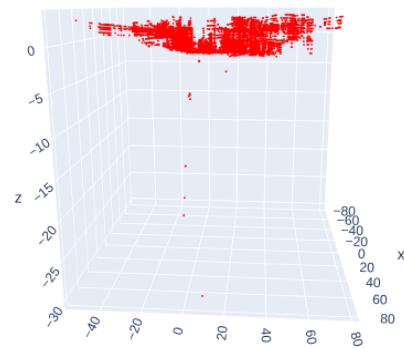
سؤال الف

تصویر اول :

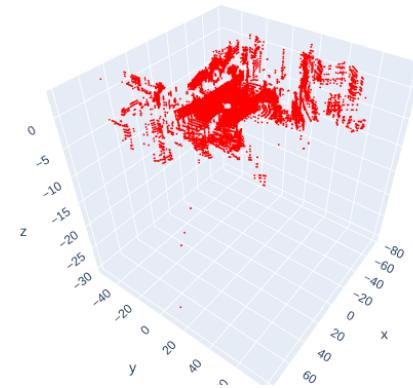
تصویر از بالا



تصویر با X ثابت



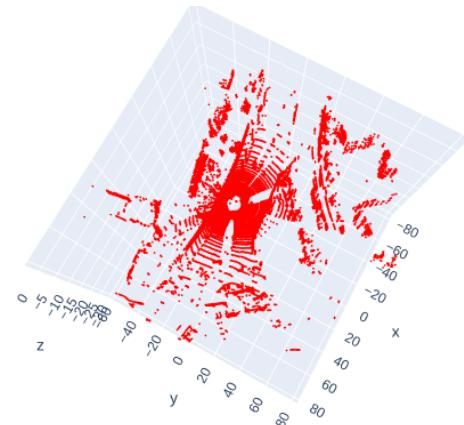
تصویر ۴۵ درجه



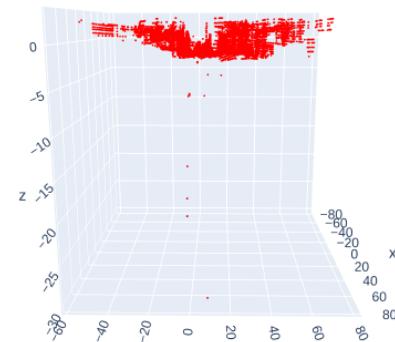
سؤال الف

تصویر دوم :

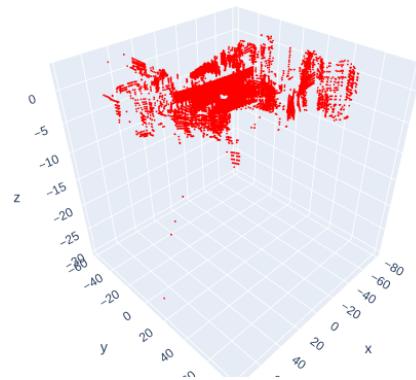
تصویر از بالا



تصویر با X ثابت



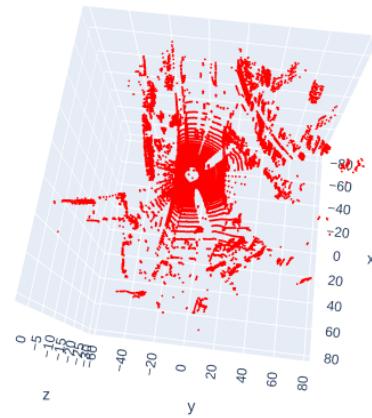
تصویر ۴۵ درجه



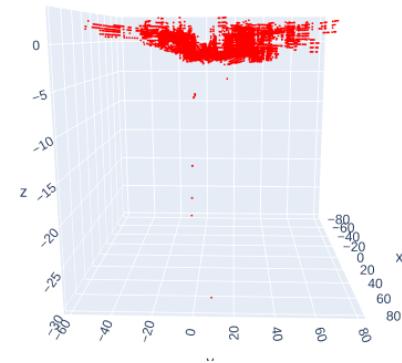
سؤال الف

تصویر سوم :

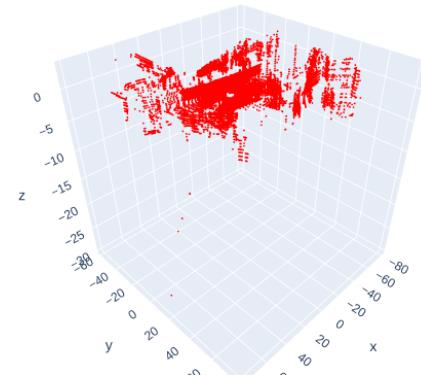
تصویر از بالا



تصویر با X ثابت



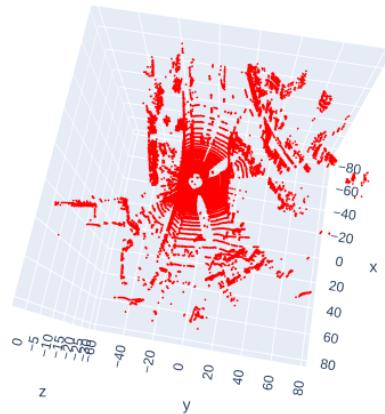
تصویر ۴۵ درجه



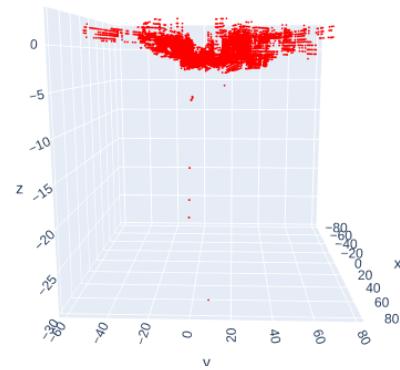
سؤال الف

تصویر چهارم :

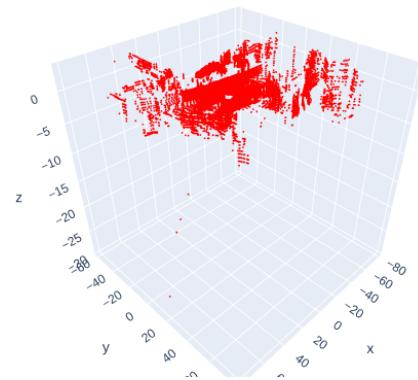
تصویر از بالا



تصویر با X ثابت



تصویر ۴۵ درجه



سوال الف

مزایا و معایب لایدار :

مزایا لایدار :

۱. داده ها به سرعت جمع آوری می شوند و دقت بالایی حفظ می شود. از آنجایی که لایدار یک فناوری سنجش هوایبرد است، جمعآوری دادهها فرآیند بسیار سریعتری است و بسیار دقیق است. این به دلیل مزیت موقعیتی است.

۲. داده های سطحی تراکم نمونه بالایی دارند. در مقایسه با سایر روش های مورد استفاده در جمعآوری دادهها، لایدار در مورد تراکم نمونه بالاتر دست بالا را دارد. این می تواند برای بهبود نتایج برنامه های خاص کار کند.

۳. توانایی جمع آوری داده های ارتفاعی حتی در جنگل های انبوه را دارد. فناوری LIDAR قابلیت نفوذ بالایی دارد. بنابراین به راحتی می تواند مناطق جنگلی متراکم را نقشه برداری کند و داده های ارتفاعی مورد نیاز را جمع آوری کند.

۴. هم روز و هم شب کار می کند. فناوری LIDAR از یک سنسور روشنایی فعال تشکیل شده است. به همین دلیل، تغییر نور روز و شب نمی تواند بر آن تأثیر بگذارد. این باعث کارآمدی آن می شود.

۵. هیچ گونه اعوجاج هندسی وجود ندارد. برخلاف سایر روش های جمعآوری دادهها، حسگرهای LIDAR در معرض هیچ یک از اعوجاج های هندسی نیستند.

۶. به حداقل نظارت انسانی نیاز دارد. تکنولوژی LIDAR بر خلاف سایر اشکال مانند نقشه برداری وابستگی کمی به انسان دارد. تعداد زیادی از فرآیندها در حال حاضر خودکار هستند و از این رو نیاز به وابستگی انسان را از بین می برند. این باعث صرفه جویی در زمان می شود.

۷. تحت تأثیر شرایط آب و هوایی شدید قرار نگرفته است. شرایط آب و هوایی شدید در عملکرد فناوری LIDAR اختلالی ایجاد نمی کند. هنوز هم می توان داده ها را زیر نور سوزان خورشید جمع آوری کرد و برای تجزیه و تحلیل فرستاد.

معایب لایدار :

۱. گران است. زمانی که نیاز به جمع آوری داده در مناطق وسیعی از زمین دارید، LIDAR مورد نیاز است. اگر این فقط یک پروژه کوچک است، پس من نمی دانم که چرا باید برای خرید فناوری LIDAR، بانک را شکست دهید. از سوی دیگر، ممکن است ارزش آن را داشته باشد که از هزاران دلار استفاده کنید، زمانی که می دانید این چیزی است که کار را به خوبی انجام می دهد.

۲. در باران های شدید و ابرهای کم آویزان بی تاثیر است. پالس های LIDAR ممکن است در شرایط آب و هوایی مانند باران های شدید یا ابرهای کم کارآمد نباشند. این به این دلیل است که انکساری وجود دارد که بر کل فرآیند تأثیر می گذارد.

سوال الف

۳. تحت تاثیر زوایای زیاد خورشید و بازتاب. این فناوری در مناطقی با زاویه و انعکاس خورشید زیاد کار نمی کند. به خاطر داشته باشید، پالس های لیزر به بازتاب ها متکی هستند.
۴. عدم دقت در عمق آب و امواج متلاطم. هنگامی که روی سطوح آبی که یکنواخت نیستند استفاده می شود، داده های جمع آوری شده معمولاً قابل اعتماد نیستند. عمق آب ممکن است بر انعکاس پالس ها تأثیر منفی بگذارد.
۵. داده خیلی زیاد فناوری LIDAR تمایل دارد مجموعه داده های زیادی را جمع آوری کند که نیاز به تجزیه و تحلیل و تفسیر رقابتی دارند. بنابراین، ممکن است زمان زیادی طول بکشد تا همه داده ها تجزیه و تحلیل شوند.
۶. پرتوهای قدرتمند لیزر می توانند به طور بالقوه بر چشم انسان تأثیر بگذارند. مواردی وجود دارد که پرتوهای لیزر فناوری LIDAR بسیار قوی هستند. این ممکن است چشم انسان را به شدت تحت تاثیر قرار دهد.
۷. به تحلیلگران ماهر داده نیاز دارد. به طور معمول، LIDAR مجموعه داده های عظیم و پیچیده ای را جمع آوری می کند. به همین دلیل، تکنیک های ماهری در تجزیه و تحلیل داده ها مورد نیاز است و این ممکن است هزینه ها را حتی گران تر کند.

مزایا و معایب ابر نقاط :

مزایا ابر نقاط :

۱. داده ها به سرعت جمع آوری می شوند و دقت بالایی حفظ می شود.
۲. داده های سطحی تراکم نمونه بالایی دارند.
۳. دقت کلی ابر نقطه نزدیک به ۱۰۰٪ است. این آستانه بسیار بالاتر از روش های سنتی تر است. هنگامی که ترجمه و در دنیای واقعی اعمال می شود، دقت بهتر به معنای کنترل بهتر هزینه است. اگر مسائل به طور دقیق و کارآمد شناسایی شوند، امکان رسیدگی به آنها را به همان شیوه فراهم می کند.
۴. در مورد جزئیات ها نیز بسیار عالی تمام جزئیات را در نظر گرفته به خوبی آن ها را نمایش میدهند.
۵. استفاده از ابری نقطه ای سه بعدی، تصمیمات حیاتی را بهتر تحت تاثیر میگذارد و روند برنامه ریزی را آگاه می کند. با توانایی شناسایی مشکلاتی که نیاز به رسیدگی اولیه دارند، می توانند به جلوگیری از افزایش هزینه های پیش بینی نشده کمک کنند.

معایب ابر نقاط :

۱. داده های عظیم

- داده ها یک فایل عظیم با مجموعه های نقطه ای است که حاوی اطلاعاتی درباره اشیاء در فضای سه بعدی است. بنابراین، ذخیره این داده ها و انتقال آن، چه به پایگاه داده یا برای پردازش، مشکل ساز است.

سوال الف

۲. پردازش پذیری

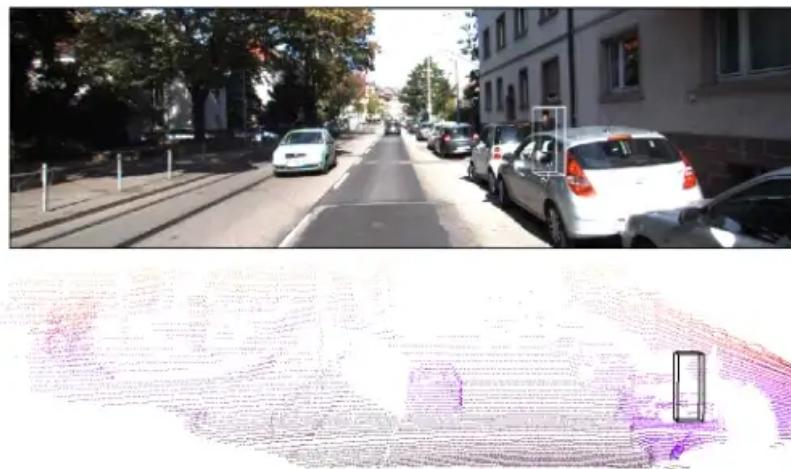
پردازش داده های انبوه نیز سخت است - برای تجزیه و تحلیل یا پردازش فایل یا گروهی از فایل ها به قدرت محاسباتی بسیار زیادی نیاز است. هنگامی که با فضای ذخیره سازی بزرگ مورد نیاز برای کار با این نوع داده ترکیب می شود، فرآیند پرهزینه می شود. و افزایش هزینه ها سودآور استفاده از داده های ابری نقطه ای را کاهش می دهد.

۳. پردازش داده های چندوجهی

تأثیر مستقیم مشکلات ذکر شده در بالا. ترکیب چندین نوع داده به خودی خود یک چالش است. وقتی صحبت از کار با انواع بسیار متفاوت داده می شود، چالش حتی بزرگتر می شود.

سوال ب

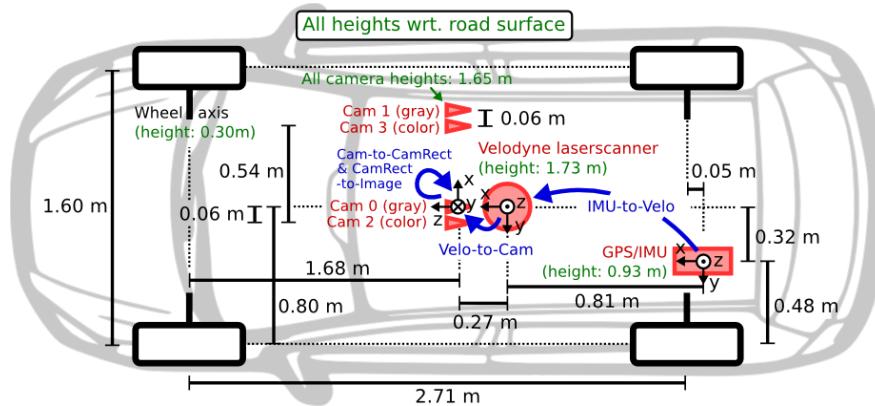
وقتی نوبت به تصاویر متراکم تر و غنی تر می‌رسد، دوربین از لایدار بهتر عمل می‌کند. با توجه به شکل زیر، با نگاه کردن به ابر نقطه پراکنده، تشخیص درست جعبه سیاه به عنوان عابر پیاده نسبتاً دشوار است. با این حال، با توجه به تصویر RGB، حتی در صورتی که پشت فرد قابل رویت باشد، به راحتی می‌توانیم متوجه شویم که جسم شبیه یک عابر پیاده است. علاوه بر این، سایر ویژگی‌های بصری مفیدی که می‌توان استخراج کرد شامل چراغ راهنمایی و علائم جاده‌ای است که لایدار با آن‌ها مشکل دارد.



در مقابل، لایدار در استخراج اطلاعات مسافت برتری دارد. اندازه گیری فاصله با استفاده از دوربین در نمای پرسپکتیو استاندارد بسیار دشوار است.

با ادغام اطلاعات از هر دو سنسور، ایده این است که می‌توانیم از مزایای هر دو سنسور استفاده کنیم و بر محدودیت‌های فردی غلبه کنیم. وجود سنسورهای متعدد روی برد نیز امکان افزونگی را فراهم می‌کند، که یک عنصر حیاتی در رانندگی خودکار ایمن در صورت خرابی سنسور است.

در ابتدا لازم است اطلاعات مربوط به قرارگیری و نوع سنسور‌ها داشته باشیم. با توجه به پیوست قرار گرفته در صورت سوال، شکل کلی سنسور‌ها به صورت زیر است :



سوال ب

در این تصویر دوربین ۰ و ۱ دوربین تصویر سطح خاکستری هستند. دوربین ۰ به عنوان دوربین مرجع در نظر گرفته میشود.

دوربین های ۲ و ۳ دوربین های رنگی هستند.

در وسط نیز لایدار قرار داده شده است.

سیستم مختصات: وسیله نقلیه دارای سیستم مختصات سمت چپ است.

فرآیند استفاده از ابر نقاط و تصاویر از چندین مرحله تشکیل شده است که در زیر هر مرحله شرح داده میشود. اما ابتدا لازم است توضیحاتی در مورد استفاده از فایل های متفاوت داشته باشیم. در این فرآیند از چندین فایل استفاده میشود. ابتدا فایلی شامل نقاط لایدار است که در سوال الف از آنها استفاده شده است. فایل دیگر فایلهای تصاویر برای دوربین های متفاوت است. فریم های متفاوتی از تصویر هم به صورت اطلاعات ابر نقاط و هم تصویر آن درون فایل ها قرار دارد. فایل دیگر شامل اطلاعات مربوطه برای نگاشت لایدار به روی تصویر مرجع است که به نام فایل calib_velo_to_cam است. فایل دیگری شامل اطلاعات داخلی و خارجی هر دوربین و متغیرات لازم برای نگاشت دوربین ها به یکدیگر قرار دارد.

حال باید از این فایل ها استفاده کرده تا بتوانیم نگاشتی از اطلاعات لایدار (ابر نقاط) به روی تصویر دوربین شماره ۲ ایجاد کنیم.

برای این کار ابتدا لازم است نگاشت این نقاط را به روی تصویر دوربین شماره ۲ ایجاد کنیم. برای اینکار باید مختصات نقاط را به روی تصویر به دست بیاوریم.

در این حالت لازم است ابتدا با استفاده از ماتریس تبدیل نقاط ، ابر نقاط را به تصویر مرجع که تصویر دوربین ۰ است تبدیل شود. برای اینکار لازم است ماتریس مورد نظر از فایل calib_velo_to_cam استخراج شود. با توجه به توضیحات ارائه شده در جلسه توجیهی ، برای حالت هوموژنس ، میتوانیم با ضرب یک ماتریس هم چرخش و هم انتقال را اعمال کرد که به همین علت از این ماتریس استفاده میشود.

در مرحله بعدی لازم است تا تصویر به دست آمده را به صورت اصلاح شده آن محاسبه کرد. استفاده از تصویر اصلاح شده باعث میشود تا از تراز مسطح بین دوربین ها استفاده شود و دقت کار را بهبود ببخشد. برای این کار لازم است از calib_cam_to_cam.txt استفاده کرده و از R_rect_xx استفاده کرده و با ضرب ماتریس مربوطه در نقطه به دست آمده نقطه جدید در تصویر اصلاح شده به دست خواهد آمد.

در مرحله آخر نیاز است تا با استفاده از P_rect_xx مختصات نقطه به دست آمده در تصویر اصلاح شده را به تصویر اصلی مورد نظر انتقال دهیم. برای این کار نیاز است ماتریس به دست آمده از این بخش را در مقدار قبلی ضرب کنیم تا در نتیجه مختصات در ماتریسنهایی به دست آید. در این مرحله لازم است اگر مقدار Z غیر ۱ بود ، مقدار آن را بر تمام مقادیر تقسیم کنیم تا مختصات صحیح به دست آید.

در این مرحله با استفاده از مقدار فاصله به دست آمده از لایدار ، رنگ نقاط را معین کنیم.

سوال ب

برای پیاده سازی این عملیات از تابع lidar2image استفاده میکنیم. ورودی این تابع فریم تصویر مورد نظر ، نام دوربین مورد نظر برای تبدیل و نقاط لایدار فریم مورد نظر هستند.
در ابتدای این تابع مشخصات مورد نظر در مورد دوربین ها و لایدار را استخراج میکنیم :

```
def lidar2image(img , P_rect_xx , frame):
    os.chdir("/content/drive/MyDrive/Colab Notebooks/CV_HW07"+
    "/kitti_sample/2011_09_29/")
    calib_cam_to_cam = read_calib_file("calib_cam_to_cam.txt")
    calib_velo_to_cam = read_calib_file("calib_velo_to_cam.txt")

    R_array = np.asarray(calib_velo_to_cam['R']).reshape(3,3)
    T_array = np.asarray(calib_velo_to_cam['T']).reshape(3,1)
```

برای اینکار تابع read_calib_file کمک میگیریم :

```
def read_calib_file(filepath):
    data = {}
    with open(filepath, 'r') as f:
        for line in f.readlines():
            line = line.rstrip()
            if len(line) == 0: continue
            key, value = line.split(':', 1)
            # The only non-float values in these files are dates, which
            # we don't care about anyway
            try:
                data[key] = np.array([float(x) for x in value.split()])
            except ValueError:
                pass

    return data
```

این تابع خط به خط فایل را خوانده و اطلاعات را درون یک دیکشنری ذخیره میکند که در این صورت به سادگی میتوانیم به اطلاعات مورد نظر دسترسی پیدا کنیم.

در مرحله بعد ماتریس P_velo2cam_ref را به کمک ۲ ارایه R & T میسازیم. سپس به هر نقطه لایدار یک خانه ۱ اضافه میکنیم تا با توضیحات سوال همخوانی داشته باشد. سپس ماتریس نقاط را ترانهاده میکنیم و درون ماتریس P_velo2cam_ref ضرب میکنیم در نهایت نتیجه را ترانهاده میکنیم . خروجی این مرحله تبدیل نقاط لایدار به نقاط درون دوربین مرجع خواهد بود (points_on_ref) :

سوال ب

```
# Adding column to numpy array
#P_velo2cam_ref Matrix
P_velo2cam_ref = np.r_[np.hstack((R_array, T_array)), [[0, 0, 0, 1]]]

#Calculate for input frame
points = np.asarray(frame.points)
new_points = np.hstack((points, np.ones(points.shape[0]).reshape(points.shape[0], 1)))

#Convert lidar to ref
points_on_ref = np.matmul(P_velo2cam_ref, new_points.T).T
```

سپس با اطلاعات دریافت شده از درون فایل ماتریس تبدیل مرجع به حالت اصلاح شده را بازیابی کرده و ماتریس جدید را درون نقاط درون دوربین مرجع همانند مرحله قبلی میکنیم. که در نهایت نقاط points_on_rect بدست خواهد آمد. در این مرحله لازم است تا با کمک از ماتریس تبدیل هر دوربین، نقاط را به آن دوربین ببریم که نیاز داریم تا ماتریس را از دیکشنری بازیابی کرده که برای این کار از ورودیتابع که نام دوربین را دریافت میکند استفاده میکنیم. سپس ماتریس بازیابی شده را درون ماتریس نقاط درون دوربین اصلاح شده ضرب میکنیم. نتیجه در نهایت به صورت points_on_cam2 خواهد بود :

```
#Convert ref to rect
#R_rect_00 Matrix
R_rect_00 = calib_cam_to_cam["R_rect_00"].reshape(3,3)
R_rect_00 = np.r_[np.hstack((R_rect_00, [[0], [0], [0]])), [[0, 0, 0, 0]]]

points_on_rect = np.matmul(R_rect_00, points_on_ref.T).T

#Convert rect to new Cam
P_rect_02 = calib_cam_to_cam[P_rect_xx].reshape(3,4)
points_on_cam2 = np.matmul(P_rect_02, points_on_rect.T).T
```

در مرحله بعد به علت اینکه عنصر درون مقدار Z هر خانه ماتریس غیر ۱ است لازم است تا داده هر نقطه را نرمال کنیم و بر Z هر نقطه تقسیم میکنیم. در مرحله بعد نیاز است تا نقاطی که بیرون از تصویر هستند را از تصویر حذف کنیم. پس از طول و عرض تصویر استفاده کرده و همچنین نقاطی که پشت به دوربین قرار میگیرند را حذف میکنیم.

نتیجه نهایی به صورت lidarOnImage قرار دارد :

```
#Remove points out of the image frame
mask = (main_points_cam2[:,0] >= 0) & (main_points_cam2[:,0] <= img.shape[1]) &
       (main_points_cam2[:,1] >= 0) & (main_points_cam2[:,1] <= img.shape[0])
mask = mask & (points_on_rect[:,2] > 2)
points_2d = main_points_cam2[mask,0:2]

lidarOnImage = np.concatenate((points_2d, points_on_rect[mask,2].reshape(-1,1)), 1)
```

سوال ب

در مرحله بعد نياز داريم تا نقشه عمق را با استفاده از تصوير و لايدار به دست آوريم. در اين حالت از تابع کمک ميگيريم. ورودي تابع نقاط به دست آمد و طول و عرض تصوير است. تابع ذكر شده به صورت زير است :

```
def dense_map(Pts, n, m, grid):
    ng = 2 * grid + 1

    mX = np.zeros((m,n)) + np.float16("inf")
    mY = np.zeros((m,n)) + np.float16("inf")
    mD = np.zeros((m,n))
    mX[np.int32(Pts[1]),np.int32(Pts[0])] = Pts[0] - np.round(Pts[0])
    mY[np.int32(Pts[1]),np.int32(Pts[0])] = Pts[1] - np.round(Pts[1])
    mD[np.int32(Pts[1]),np.int32(Pts[0])] = Pts[2]

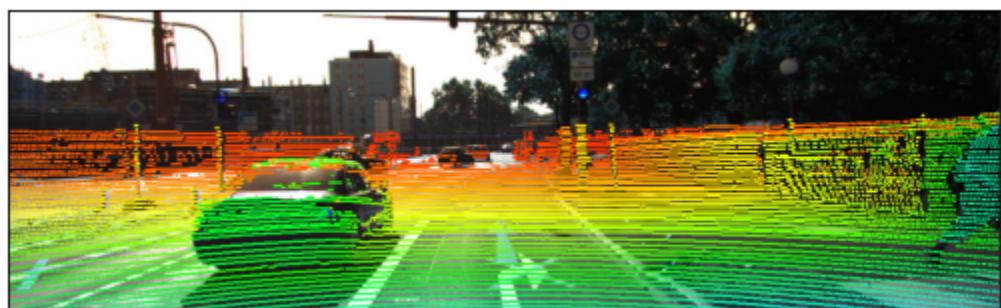
    KmX = np.zeros((ng, ng, m - ng, n - ng))
    KmY = np.zeros((ng, ng, m - ng, n - ng))
    KmD = np.zeros((ng, ng, m - ng, n - ng))

    for i in range(ng):
        for j in range(ng):
            KmX[i,j] = mX[i : (m - ng + i), j : (n - ng + j)] - grid - 1 + i
            KmY[i,j] = mY[i : (m - ng + i), j : (n - ng + j)] - grid - 1 + i
            KmD[i,j] = mD[i : (m - ng + i), j : (n - ng + j)]
    S = np.zeros_like(KmD[0,0])
    Y = np.zeros_like(KmD[0,0])

    for i in range(ng):
        for j in range(ng):
            s = 1/np.sqrt(KmX[i,j] * KmX[i,j] + KmY[i,j] * KmY[i,j])
            Y = Y + s * KmD[i,j]
            S = S + s

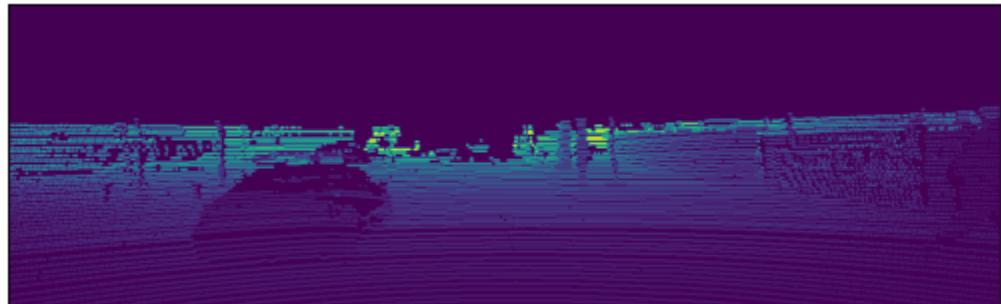
    S[S == 0] = 1
    out = np.zeros((m,n))
    out[grid + 1 : -grid, grid + 1 : -grid] = Y/S
    return out
```

در اين تابع از فاصله و همسایگی نقاط استفاده کرده و نقشه عمق را ايجاد ميکنیم. سپس از اين نقشه استفاده ميکنیم. ابتدا يك كپي از تصوير اصلی ميگيريم. سپس يك ارایه از رنگ ها ايجاد ميکنیم. سپس با استفاده از نقشه عمق و تصوير اصلی ، اگر در نقطه اي عمق غير باشد رنگ را از ارایه با يك فرمول خاص به دست مى آوریم و اين رنگ را درون تصوير اصلی درون آن پیکسل قرار ميدهیم. نتیجه نهايی به صورت زير است :



سوال ج

ابتدا برای شکل نقشه عمق ، از تابع بیان شده در قسمت قبل (dense_map) استفاده کرده و تصویر نقشه عمق را ایجاد میکنیم. برای فریم اول به صورت زیر میباشد :



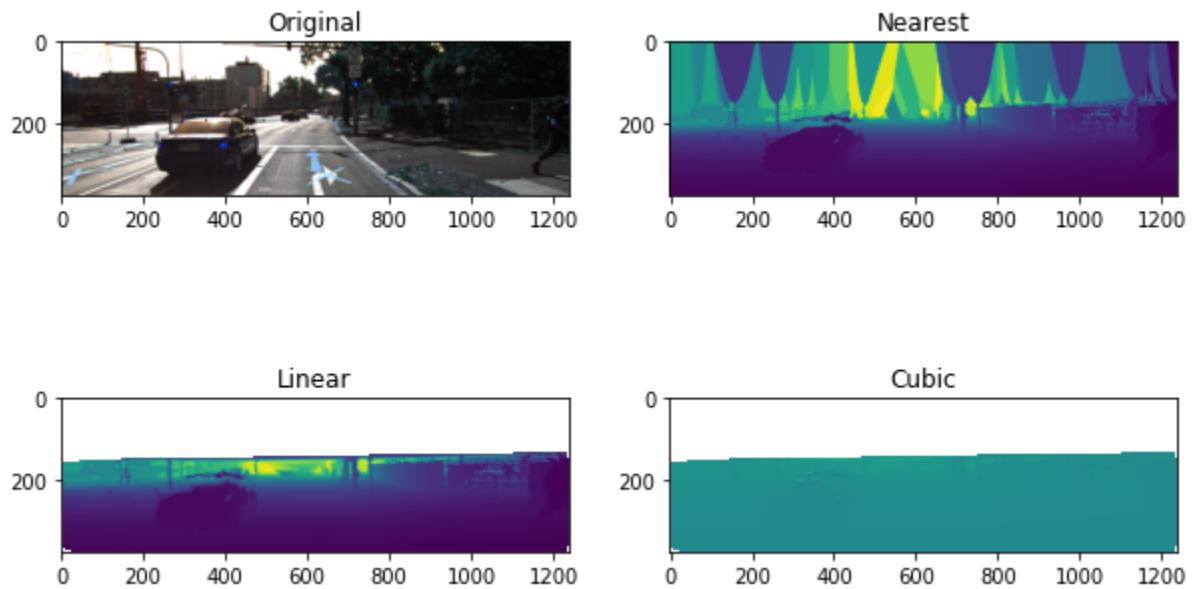
در ادامه طبق سوال نیاز است تا از روش های درون یابی استفاده کنیم تا به بهترین نتیجه برسیم. ابتدا یک gridded از طول و عرض تصویر ایجاد میکنیم. سپس با استفاده از griddata و استفاده از نقاط کلی و نقاطی که مقدار آنها را داریم مدل های مختلف را ایجاد میکنیم :

```
from scipy.interpolate import griddata
grid_z0 = griddata(points_lidar_image[:, 0:2], points_lidar_image[:, 2], (grid_x, grid_y), method='nearest')
grid_z1 = griddata(points_lidar_image[:, 0:2], points_lidar_image[:, 2], (grid_x, grid_y), method='linear')
grid_z2 = griddata(points_lidar_image[:, 0:2], points_lidar_image[:, 2], (grid_x, grid_y), method='cubic')
```

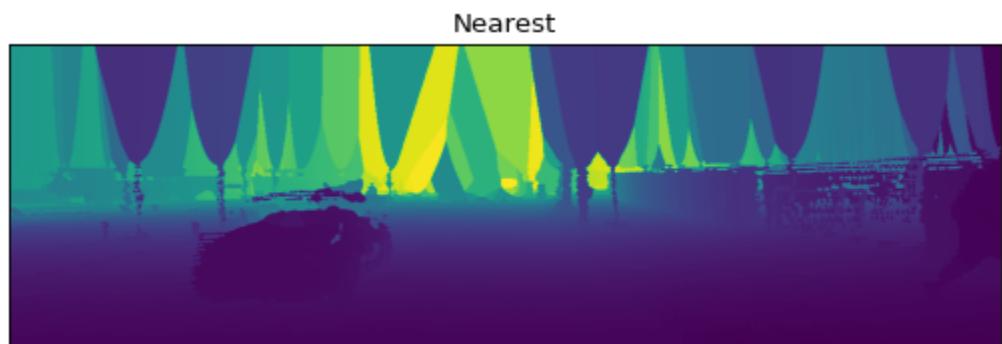
از ۳ حالت نزدیک ترین همسایه ، درون یابی خطی و درون یابی مکعبی استفاده کرده ایم. نتیجه به دست آمده به صورت زیر است .

```
import matplotlib.pyplot as plt
plt.subplot(221)
plt.imshow(img)
# plt.plot(points[:,0], points[:,1], 'k.', ms=1)
plt.title('Original')
plt.subplot(222)
plt.imshow(grid_z0.T)
plt.title('Nearest')
plt.subplot(223)
plt.imshow(grid_z1.T)
plt.title('Linear')
plt.subplot(224)
plt.imshow(grid_z2.T)
plt.title('Cubic')
plt.gcf().set_size_inches(6, 6)
plt.show()
```

سوال ج



با توجه به نتایج به دست آمده مدل نزدیکترین همسایه بهترین نتیجه را ایجاد میکند.



سوال د

در این سوال ابتدا به فولدر مربوط به تصاویر خاکستری رفته و تصاویر آن را بازگیری میکنیم.

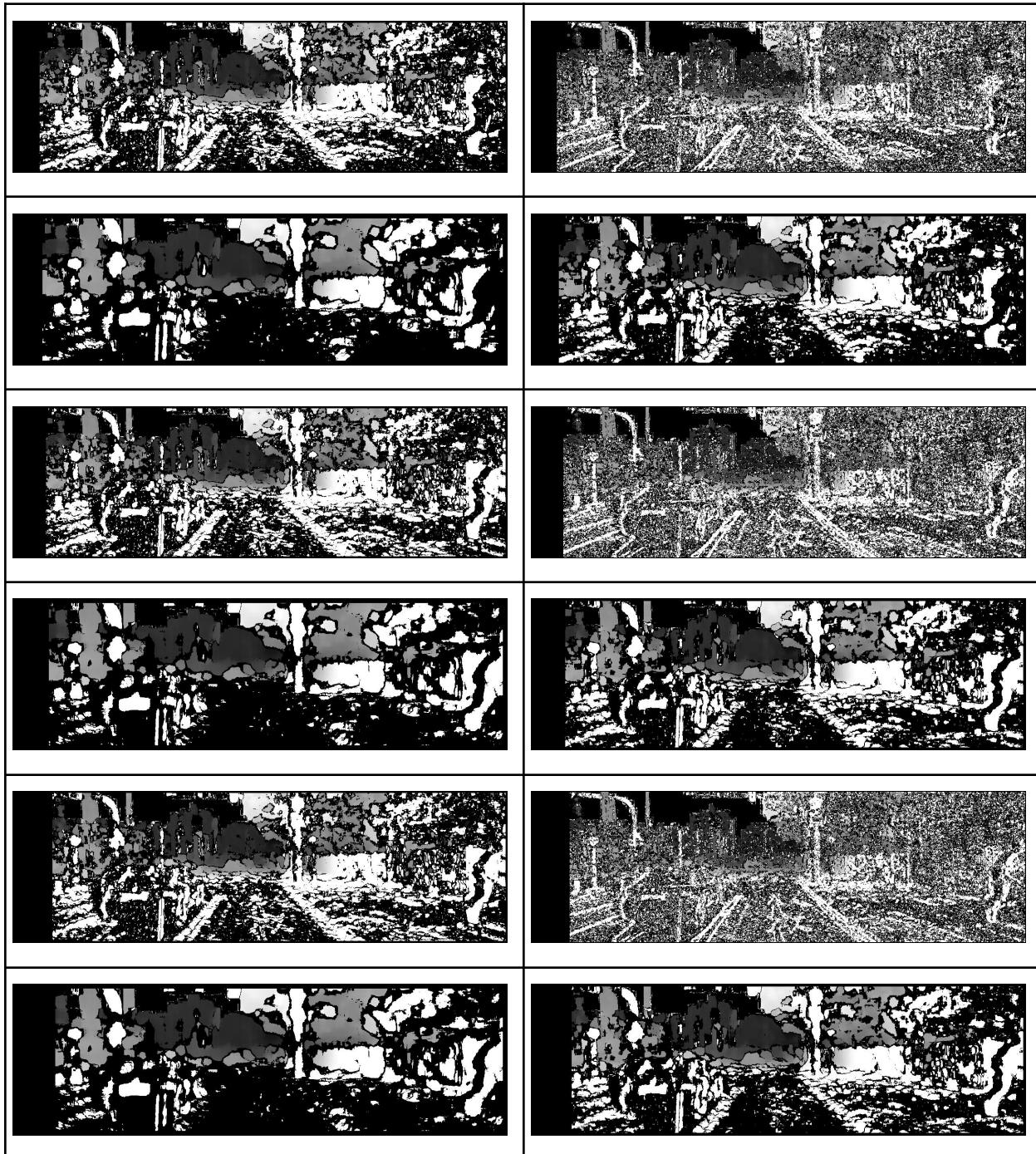
```
os.chdir("/content/drive/MyDrive/Colab Notebooks/CV_HW07"+  
        "/kitti_sample/2011_09_29/2011_09_29_drive_0026_sync/image_00/data")  
!ls  
  
0000000000.jpg 0000000001.jpg 0000000002.jpg 0000000003.jpg  
  
images0 = []  
for item in np.sort(os.listdir()):  
    images0.append(cv2.cvtColor(cv2.imread(item), cv2.COLOR_BGR2GRAY))  
  
os.chdir("/content/drive/MyDrive/Colab Notebooks/CV_HW07"+  
        "/kitti_sample/2011_09_29/2011_09_29_drive_0026_sync/image_01/data")  
!ls  
  
0000000000.jpg 0000000001.jpg 0000000002.jpg 0000000003.jpg  
  
images1 = []  
for item in np.sort(os.listdir()):  
    images1.append(cv2.cvtColor(cv2.imread(item), cv2.COLOR_BGR2GRAY))
```

در مرحله بعد برای تصویر فریم اول از ۲ دوربین خاکستری و استفاده میکنیم . برای این کد نیاز است تا مقدار های متفاوتی از BlockSize و num_disparities را بررسی کنیم . برای همین از کد زیر استفاده میکنیم .

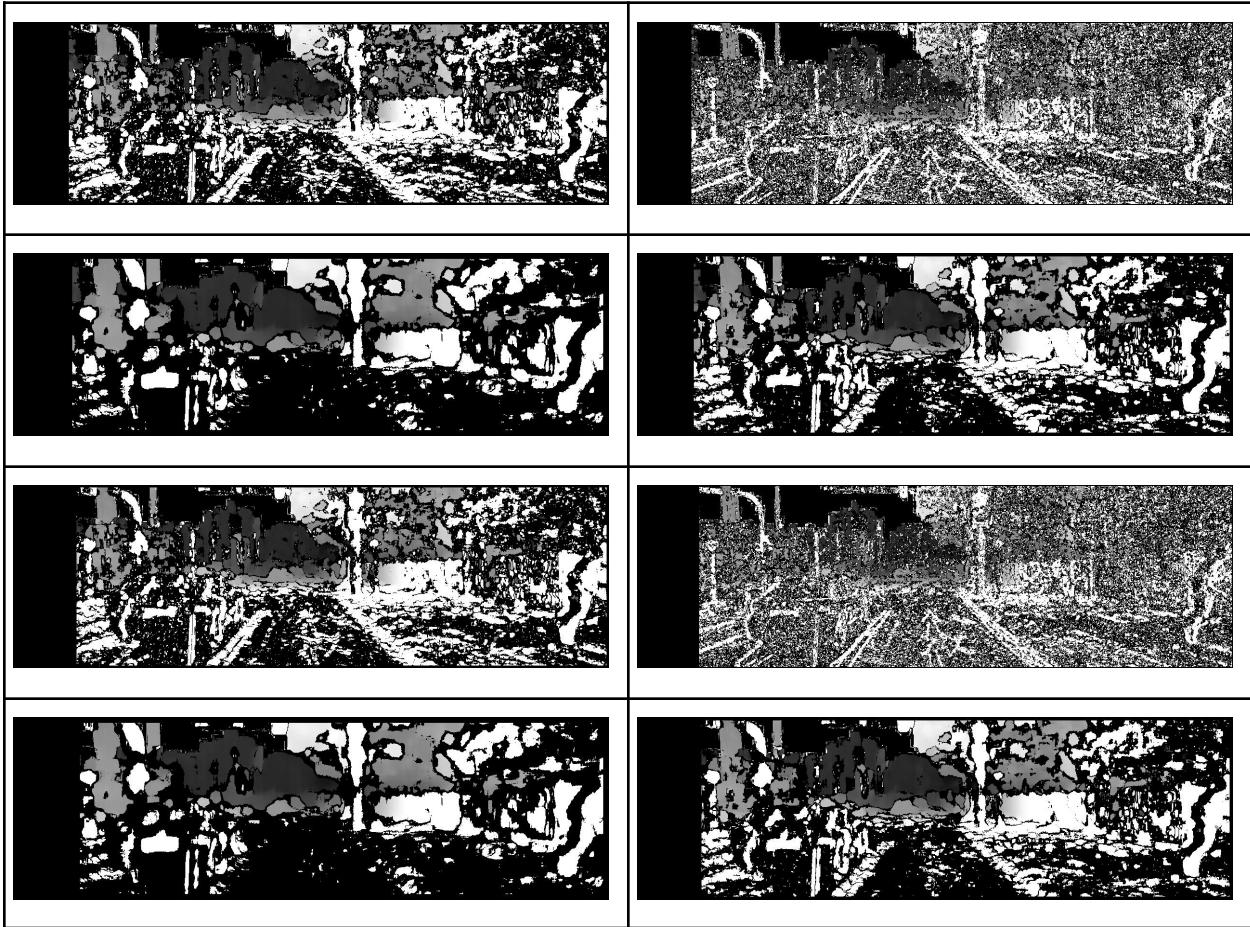
```
left_RGB = images0[0] # left RGB image  
right_RGB = images1[0] # right RGB image  
# compute depth map from stereo  
depth_maps = []  
blocks = [5,11 , 15,25]  
for item in range(3,8):  
    for i in blocks:  
        stereo = cv2.StereoBM_create()  
        num_disparities = 16*(item+1)  
        stereo.setNumDisparities(num_disparities)  
        stereo.setBlockSize(1*(i))  
        stereo_depth_map = stereo.compute(left_RGB,right_RGB)  
        title = "depth map : disparity =" + str(16*(item+1)) +  
               " Block size =" + str(1*(i))+".jpg"  
        cv2.imwrite(title , stereo_depth_map)  
        depth_maps.append(stereo_depth_map)
```

سوال د

نتیجه به دست آمده به صورت زیر خواهد بود. در این کد از ۲۰ حالت استفاده شده است.



سوال د



در این حالت برای سایز بلاک ها از سایز ۵ ، ۱۱ ، ۱۵ و ۲۵ استفاده شده است. برای مقدار num_disparities نیز از ضریب های ۳ ، ۴ ، ۵ ، ۶ ، ۷ برابر ۱۶ استفاده شده است.

این نقشه عمق ساخته شده در مقابل نقشه عمق ساخته شده توسط ابر نقاط بدون در نظر گرفتن درون یابی ، کامل قابل مشاهده است که این نقشه ها بسیار متراکم تر و پر ویژگی تر از نقشه ابر نقاط هستند. فقط در این حالت در نقاطی قابل مشاهده است که مقداری در نظر گرفته نشده است که با استفاده از درونیابی یا استفاده از یک هموارسازی یا کرنلی برای رفع نویز می تواند نتیجه بسیار بهتری را حتی بهتر از ابر نقاط با درونیابی نتیجه ایجاد کند. در مجموع قابل مشاهده است که نقشه ایجاد شده بسیار بهتر از نقشه ایجاد شده توسط ابر نقاط خواهد بود.

سوال ۵

برای این سوال از قطعه کد قرار گرفته در سوال استفاده میکنیم :

```
def img_to_pointcloud(img, depth, K , RT):
    rgb = o3d.geometry.Image(img)
    depth = o3d.geometry.Image(depth)
    rgbd = o3d.geometry.RGBDImage.create_from_color_and_depth(rgb, depth, convert_rgb_to_intensity=False)
    fx, fy, cx, cy = K[0, 0], K[1, 1], K[0, 2], K[1, 2]
    intrinsic = o3d.camera.PinholeCameraIntrinsic(int(cx*2), int(cy*2), fx, fy, cx, cy)
    pc = o3d.geometry.PointCloud.create_from_rgbd_image(rgbd, intrinsic , RT)
    # o3d.visualization.draw_geometries([pc])
    return pc
```

نکته ای که در این مورد وجود دارد این است که درون کولب نمیتوانیم عملیات draw_geometries را انجام دهیم برای همین ابتدا مقدار pcd را محاسبه میکنیم. سپس این مقدار را ذخیره کرده و درون برنامه دیگری درون سیستم اجرا میکنیم که در ادامه به بررسی آن میپردازیم.

در مرحله بعدی تصویر مورد نظر را بارگیری میکنیم. در این کد نیازی داریم تا ویژگی های خروجی را نیز بارگیری کنیم برای همین از دیکشنری استفاده کرده و ارایه T & R را بارگیری میکنیم. سپس با اتصال آنها به یکدیگر ماتریس Rt را ایجاد میکنیم که در تابع به آن نیاز داریم. همچنین تصویر اصلی و نقشه آن را نیز به عنوان ورودی به تابع میدهیم. در این جا نیاز داریم تا مقدار K را که ماتریس مربوط به این دوربین است ، بارگیری کنیم که دوباره برای آن از دیکشنری استفاده میکنیم.

```
os.chdir("/content/drive/MyDrive/Colab Notebooks/CV_HW07"+
"/kitti_sample/2011_09_29/")
calib_cam_to_cam = read_calib_file("calib_cam_to_cam.txt")
calib_velo_to_cam = read_calib_file("calib_velo_to_cam.txt")
K = np.asarray(calib_cam_to_cam['K_02']).reshape(3,3)
R = np.asarray(calib_cam_to_cam['R_02']).reshape(3,3)
T = np.asarray(calib_cam_to_cam['T_02']).reshape(3,1)

#RT Matrix
Rt = np.r_[np.hstack((R, T)), [[0 , 0 , 0 , 1]]]

!ls
0000000000.jpg  0000000001.jpg  0000000002.jpg  0000000003.jpg  point_clo

img = cv2.imread("0000000000.jpg")
```

سوال ۵

در مرحله بعد برای تمام ۲۰ نقشه ایجاد شده مقدار pcd را ایجاد میکنیم. همچنین برای بهبود عملکرد از یک تابع گاسی استفاده میکنیم تا نقشه را اندکی بهتر نمایش دهد. سپس تابع را فراخوانی کرده و نتیجه را ذخیره میکنیم:

```
os.chdir("/content/drive/MyDrive/Colab Notebooks/CV_HW07/Files")
for i , depth in enumerate(depth_maps):
    # blur
    depth = cv2.GaussianBlur(depth, (0,0), sigmaX=1, sigmaY=1)

    pc = img_to_pointcloud(img, depth, K , Rt)
    title = str(i)+".pcd"
    o3d.io.write_point_cloud(title, pc)
    print("Number " , i , ": Finished")
```

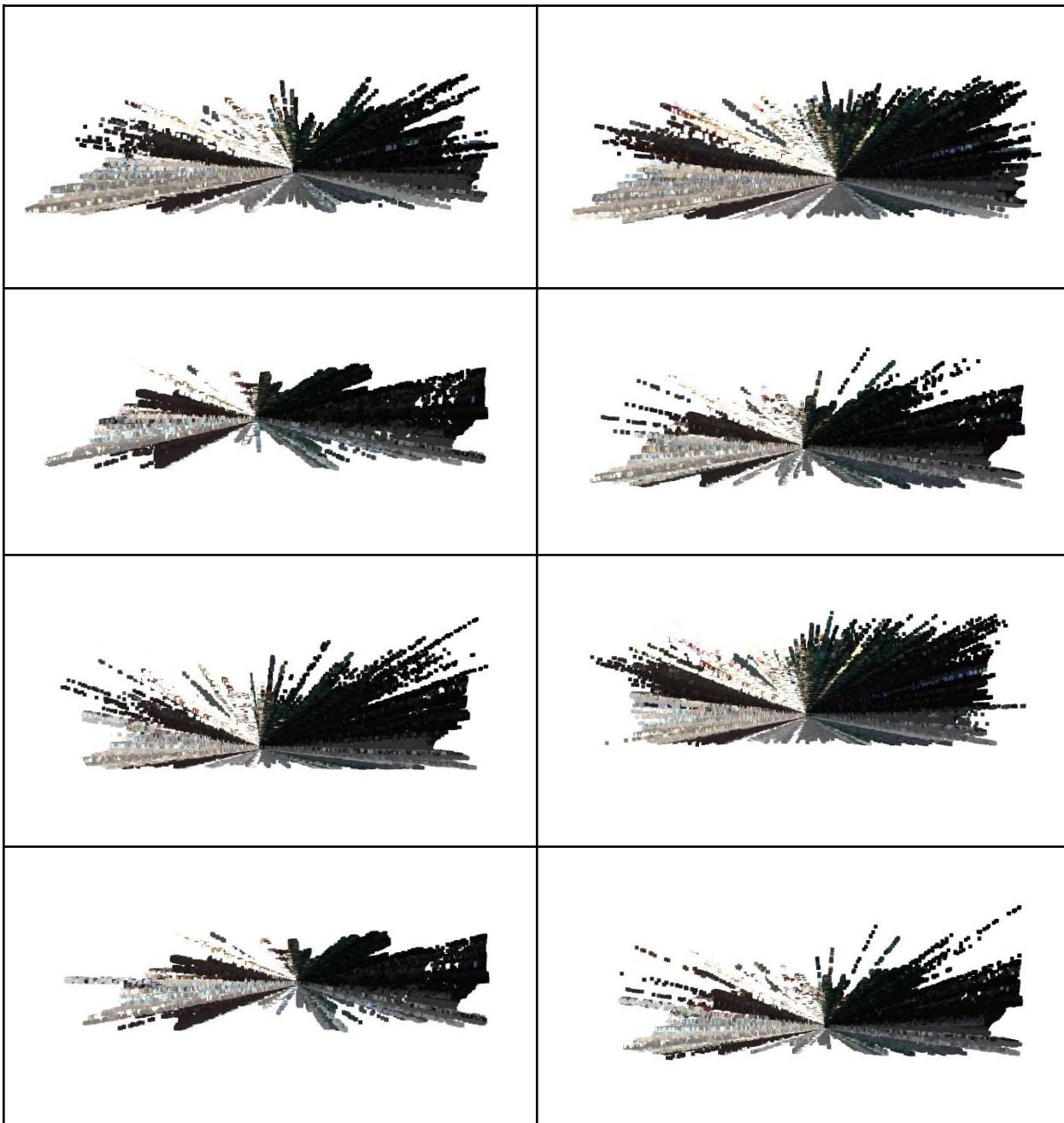
در ادامه این فایل ها را دانلود کرده و از کد زیر برای نمایش تصاویر استفاده میکنیم :

```
def custom_draw_geometry(pcd, name):
    vis = o3d.visualization.Visualizer()
    vis.create_window(visible=False)
    vis.add_geometry(pcd)
    vis.update_geometry(pcd)
    vis.poll_events()
    vis.update_renderer()
    title = str(name) + ".jpg"
    vis.capture_screen_image(title)
    vis.destroy_window()

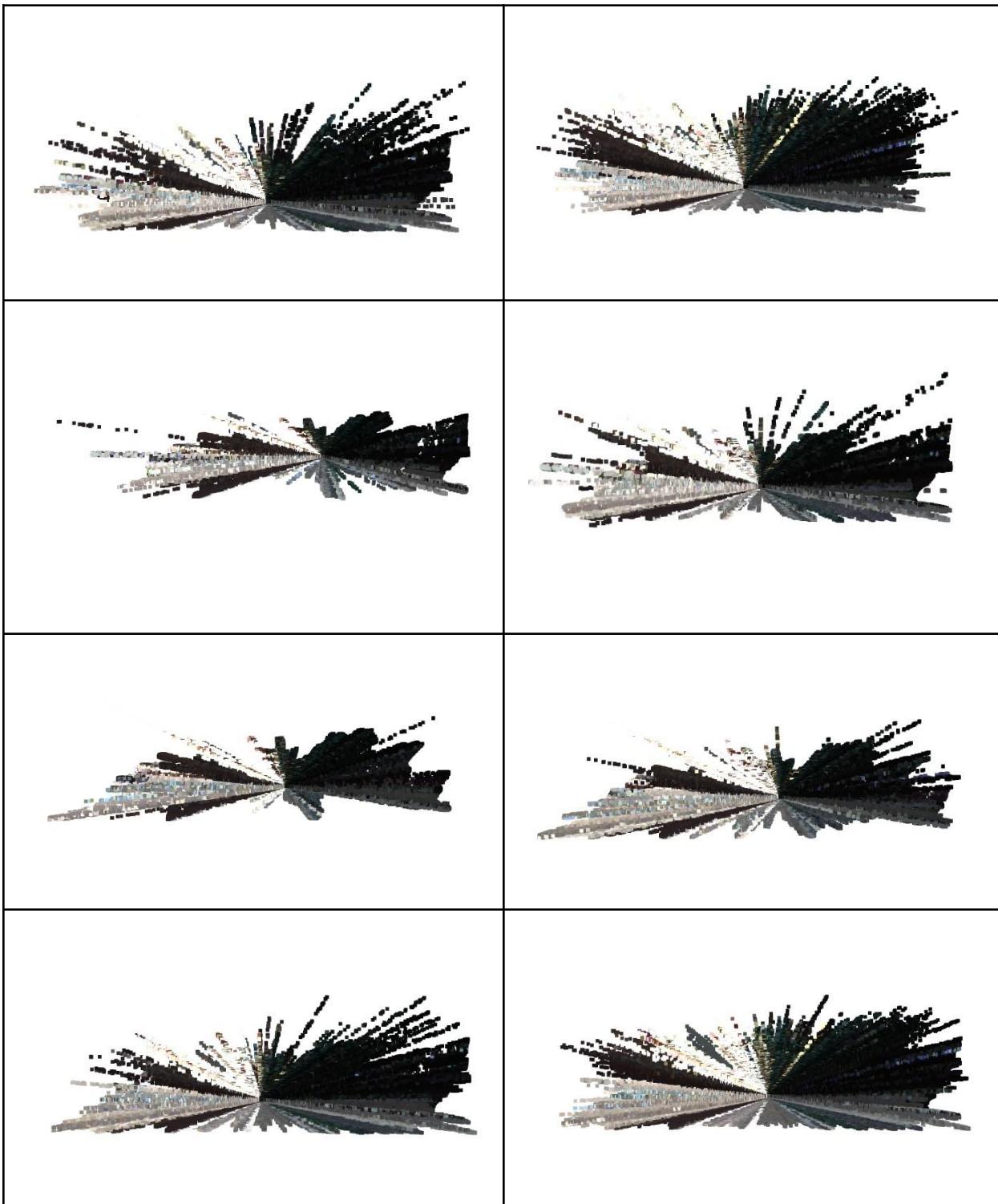
for i, item in enumerate(np.sort(os.listdir())):
    if item.endswith("pcd"):
        pcd = o3d.io.read_point_cloud(item)
        # Flip it, otherwise the pointcloud will be upside down
        pcd.transform([[1, 0, 0, 0], [0, -1, 0, 0], [0, 0, -1, 0], [0, 0, 0, 1]])
        custom_draw_geometry(pcd, i)
```

نتیجه ای تصاویر به صورت زیر خواهد بود :

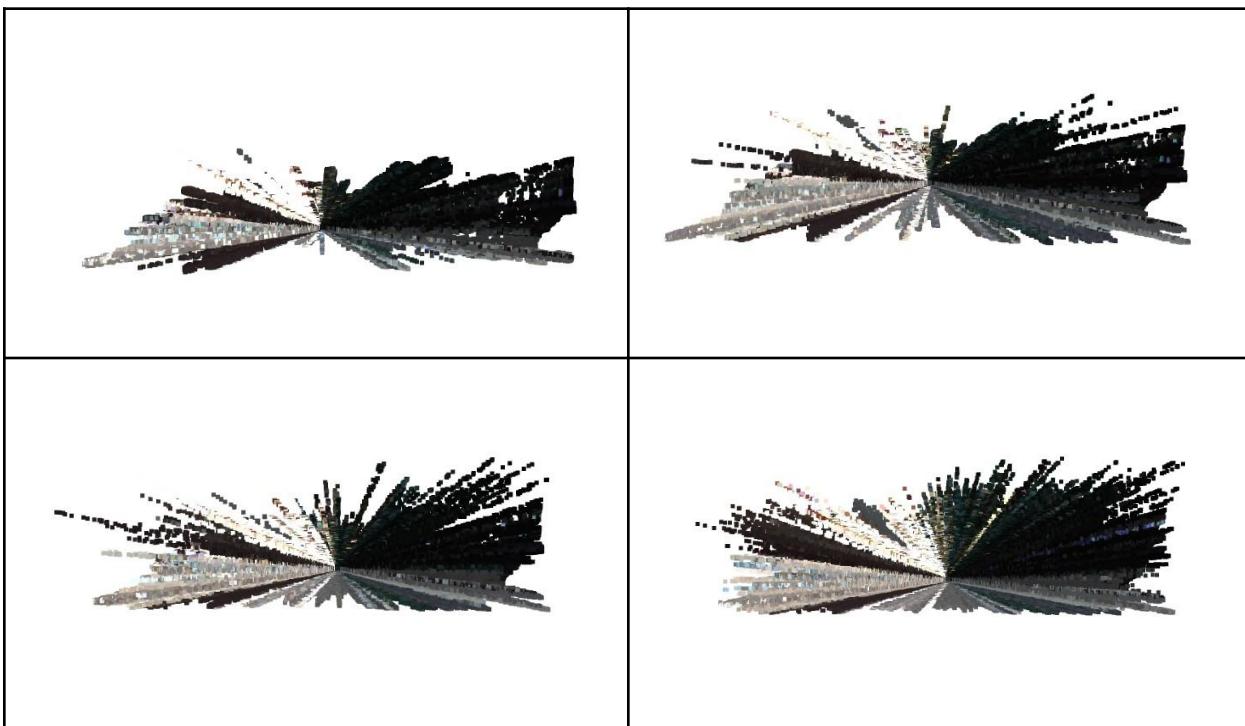
سوال ۵



سوال ۵



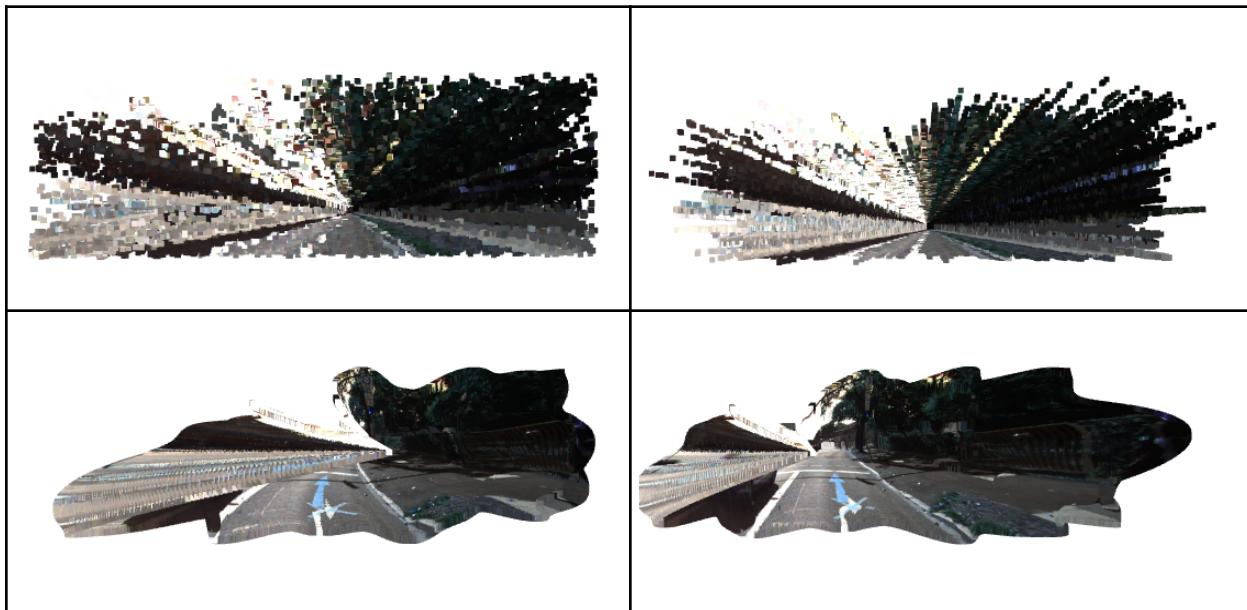
سوال ۵



تصاویر بالا تصاویری است که با استفاده از توابع ذکر شده ایجاد شده است.

سوال ۵

در ادامه برای بهبود تصویر چند مدل از هموارسازی های متفاوت را ایجاد کرده ایم که به صورت زیر است :



با عوض کردن پارامتر ها میتوانیم به تصاویر بسیار بهتری برسیم. به علت ضعیف بودن کیفیت تصاویر در فایل پیوست تصاویر قرار داده شده است.

با تشکر فراوان
مهندی فیروزبخت
۴۰۰۱۳۱۰۲۷