# 计算机视觉实践课程报告

# 几何视觉编程实践


班　　级：智创 2301

学 生 姓 名：汪天浩

学　　号：20231302005

大连理工大学

Dalian University of Technology

# 1 任务与方法介绍

main_estimateP.py: 相机投影矩阵估计

功能

main_estimateP.py 脚本主要关注从一组已知的二维图像点及其对应的三维世界点来估计**相机投影矩阵 (P)**。随后，它会将这个估计出的投影矩阵分解为相机内参 (K) 和外参 (R, t)，并与真实值进行比较。

方法和原理

1. **载入输入数据：**
    - 脚本首先载入预定义的数据：
        - **二维点 (points2D)**：代表在多张图像中检测到的特征点的二维坐标。
        - **三维点 (points3D)**：代表这些特征点在世界坐标系中的三维坐标。
        - **对应关系 (corr)**：将二维点与其对应的三维点关联起来。
        - **真实相机矩阵 (P)**：用于比较的预定义相机投影矩阵。
2. **投影与可视化：**
    - 脚本首先使用已知的 P[0] 将 points3D 投影到第一张图像上。这有助于直观地比较投影后的三维点与图像中实际的二维点。
    - matplotlib.pyplot 用于显示图像并绘制这些点，从而帮助可视化初始投影的准确性。
3. **用于 P 估计的数据准备：**
    - 脚本根据 corr 数据仅选择**可见的二维和三维点**，确保只使用可靠的对应关系进行估计。

- 二维点和三维点都被转换为**齐次坐标**，方法是在其向量末尾添加一个 '1'。这是投影几何中的标准做法，可以简化矩阵运算。

4. **相机投影矩阵 (P) 估计：**
   - 脚本的核心是 Pest = camera.Camera(sfm.compute_P(x, X)) 这行代码。sfm.compute_P 函数很可能实现了根据一组二维-三维对应关系来估计相机投影矩阵 P 的方法。
   - sfm.compute_P 所依据的底层原理通常是**直接线性变换 (DLT) 算法**。DLT 通过求解一个线性方程组来找到 P 的元素。每对二维-三维对应关系提供两个线性方程。拥有至少 6 对对应点（或者 11 个方程，因为 P 在缩放后有 11 个独立参数），DLT 就可以估计 P。如果可用点更多，则采用**最小二乘解**来找到最佳拟合。
5. **P 的分解：**
   - 一旦 Pest 被估计出来，它就会被分解为相机内参矩阵 K、旋转矩阵 R 和平移向量 t，通过 Kest, Rest, Test = Pest.factor() 实现。
   - 这种分解通常使用 **RQ 分解**（或 QR 分解，取决于约定）等技术，将内参（定义相机内部属性，如焦距和主点）与外参（定义相机在世界中的位置和方向）分离。Test 向量被重塑为 3x1 列向量，以确保正确的矩阵乘法。
6. **验证和比较：**
   - 估计出的 Pest 会被打印出来并与真实值 P[0] 进行比较。为了进行有意义的比较，两者都通过其最后一个元素 P[2,3] 进行归一化，因为投影矩阵通常在尺度因子下定义。
   - 最后，使用估计出的 Pest 将三维点重新投影到图像平面上 (xest)，并将这些投影点与原始二维点一起绘制。这种视觉比较有助于评估估计相机参数的准确性。

## main_3drecon.py: 两视图三维重建

### 功能

main_3drecon.py 脚本执行一个完整的**两视图三维重建**。它接收两张场景的灰度图像，识别它们之间的对应特征点，估计本征矩阵，恢复相机姿态，最后对三维点进行三角测量，以重建场景的稀疏三维点云。它还将重建的三维点投影回图像平面进行视觉验证。

### 方法和原理

1. **相机内参：** 脚本以一个预定义的**相机内参矩阵 K** 开始。该矩阵对于归一化图像坐标以及在像素坐标和归一化图像坐标之间进行转换至关重要。
2. **特征检测与匹配 (SIFT + BFMatcher)：**
    - **SIFT (尺度不变特征变换)：** cv2.SIFT_create() 用于初始化 SIFT 检测器。SIFT 是一种强大的算法，用于检测和描述图像中对尺度、旋转和光照变化具有不变性的局部特征。
    - **detectAndCompute：** 此方法用于查找关键点（图像中的独特特征点）并计算两个输入图像 (im1, im2) 的描述符（唯一表示每个关键点周围区域的向量）。
    - **BFMatcher (暴力匹配器)：** cv2.BFMatcher() 用于查找两个图像描述符集之间的对应关系。它将第一个集合中的每个描述符与第二个集合中的每个描述符进行比较，并找到最接近的匹配项。
    - **knnMatch 和 Lowe's 比率测试：** bf.knnMatch(descriptors1, descriptors2, k=2) 为每个描述符找到两个最近邻。然后，这用于基于 **Lowe's 比率测试** (m.distance < 0.6 * n.distance) 过滤匹配。此测试通过确保最佳匹配明显优于次佳匹配来帮助剔除模糊的匹配，从而产生更可靠的对应关系。
3. **本征矩阵估计 (F_from_ransac)：**

- **提取匹配点：** 提取 good_matches 的像素坐标并转换为**齐次坐标**。
- **归一化：** 使用内参矩阵 K 的逆矩阵对二维图像点 (x1, x2) 进行**归一化** (x1n = np.dot(np.linalg.inv(K),x1))。这将像素坐标转换为归一化图像坐标，这对于基础矩阵和本征矩阵的计算至关重要，因为它们消除了相机内参的影响。
- **本征矩阵 (E)：** 极线几何估计的核心是 E,inliers = sfm.F_from_ransac(x1n,x2n,model)。此函数旨在估计**本征矩阵 (E)**。
  - **极线几何：** 本征矩阵 (E) 描述了从不同视点拍摄的两幅图像之间的极线约束，假设相机已标定（即，内参已知）。它关联了两幅图像中的对应点。
  - **RANSAC (随机采样一致性)：** RANSAC 是一种稳健的估计技术，此处用于估计 E。它迭代地：
    1. 随机选择最小数量的对应关系（对于 E 的 8 点算法需要 8 个点）。
    2. 从该子集中估计一个本征矩阵。
    3. 计算“内点”（在一定容差范围内与估计的 E 一致的点）的数量。
    4. 重复这些步骤多次，并选择内点数量最多的 E。这使得估计对异常值（不正确的匹配）具有鲁棒性。

4. **相机姿态恢复 (compute_P_from_essential)：**
   - 第一个相机的投影矩阵 P1 被设置为单位矩阵 np.array([[1,0,0,0],[0,1,0,0],[0,0,1,0]])，代表世界坐标系的原点。
   - 本征矩阵 E 可以分解为第二个相机的四对可能的旋转 (R) 和平移 (t) 矩阵。P2 = sfm.compute_P_from_essential(E) 生成这四个可能的投影矩阵。
5. **歧义消除 (手性检查)：**

- **三角测量：** 对于四种可能的 P2 解中的每一种，脚本都会执行**三角测量**(sfm.triangulate)，以重建与内点二维匹配相对应的三维点。三角测量是给定一点在两个或多个图像中的投影以及已知的相机姿态，找到该点三维位置的过程。
- **手性检查：** 选择正确 P2 的关键是确保重建的三维点位于**两个相机前方**（即，它们在两个相机坐标系中的 Z 坐标都为正）。脚本遍历四个 P2 解，投影三维点，并计算有多少点在两个相机前方具有正深度 (d1 > 0) & (d2 > 0))。产生最多点位于两个相机前方的 P2 解被选为 best_ind。

6. **最终三维重建与可视化：**
   - 使用 best_ind 相机姿态重新对三维点进行三角测量，并过滤掉只包含那些位于两个相机前方的点。
   - 然后使用 matplotlib.pyplot 将重建的三维点绘制在三维散点图中，提供稀疏三维场景的视觉表示。
7. **投影与验证：**
   - 重建的三维点 (X) 使用估计的相机投影矩阵 (cam1.project(X) 和 cam2.project(X)) 投影回两个原始图像平面。
   - 然后，这些投影的二维点通过 K 重新缩放，以将其转换回像素坐标。
   - 最后，将投影点绘制在原始图像上方，并与原始匹配的二维点一起绘制。这种视觉比较是关键的验证步骤，它显示了重建的三维点与原始图像特征的对齐程度，从而指示了整个重建流程的准确性。

main_estimateP.py 展示了如何估计和分解相机投影矩阵，而 main_3drecon.py 则展示了从两幅图像重建稀疏三维场景的完整流程，包括特征匹配、极线几何、相机姿态估计和三维三角测量。

## 2. 代码分析

`compute_P` 函数：计算相机投影矩阵

```
def compute_P(x: np.ndarray, X: np.ndarray) -> np.ndarray:
    """
    通过 2D-3D 点对应关系计算相机投影矩阵
    该实现基于显式地引入尺度因子 lambda_i 的 DLT 方法。
    参数:
        x: 2D 点的齐次坐标 (形状为 3xN 的 numpy array，其中 N 是点的数量)
            x[0, i] 是 u_i, x[1, i] 是 v_i, x[2, i] 是 w_i
        X: 3D 点的齐次坐标 (形状为 4xN 的 numpy array，其中 N 是点的数量)
            X[0, i] 是 X_i, X[1, i] 是 Y_i, X[2, i] 是 Z_i, X[3, i] 是 W_i
    返回:
        3x4 的相机投影矩阵 P (numpy array)
    """
    n = x.shape[1]
    if X.shape[1] != n:
        raise ValueError("Number of points don't match.")

    M = np.zeros((3 * n, 12 + n))

    for i in range(n):
        u_i, v_i, w_i = x[:, i]
        X_i_vec = X[:, i]

        M[3 * i, 0:4] = X_i_vec
        M[3 * i, 12 + i] = -u_i
```

```
        M[3 * i + 1, 4:8] = X_i_vec
        M[3 * i + 1, 12 + i] = -v_i

        M[3 * i + 2, 8:12] = X_i_vec
        M[3 * i + 2, 12 + i] = -w_i

    _, _, Vt = np.linalg.svd(M)

    return Vt[-1, :12].reshape((3, 4))
```

功能描述：

这个函数使用直接线性变换（DLT）算法来估计一个 3x4 的相机投影矩阵 P。它需要输入一组 2D 图像点（齐次坐标 x）和它们对应的 3D 世界点（齐次坐标 X）。投影矩阵 P 能够将 3D 世界点投影到 2D 图像平面上。

原理和分析：

DLT 算法基于以下投影方程：

sx=PX

其中 s 是一个非零的尺度因子，x 是 2D 图像点（齐次坐标），X 是 3D 世界点（齐次坐标），P 是 3x4 的投影矩阵。

将方程展开，对于每个对应点 $(u_i, v_i, w_i)$ 和 $(X_i, Y_i, Z_i, W_i)$：

$$s_i u_i = p_{11} X_i + p_{12} Y_i + p_{13} Z_i + p_{14} W_i$$

$$s_i v_i = p_{21} X_i + p_{22} Y_i + p_{23} Z_i + p_{24} W_i$$

$$s_i w_i = p_{31} X_i + p_{32} Y_i + p_{33} Z_i + p_{34} W_i$$

通过引入尺度因子 $s_i$（在代码中是 `lambda_i`），我们可以将这些方程重写为齐次线性方程组的形式：

$(p_{11}X_i+\cdots+p_{14}W_i)-s_iu_i=0$

$(p_{21}X_i+\cdots+p_{24}W_i)-s_iv_i=0$

$(p_{31}X_i+\cdots+p_{34}W_i)-s_iw_i=0$

这些方程可以堆叠成一个大型矩阵 `M` 乘以一个包含 `P` 的 12 个未知元素和 `n` 个尺度因子 $s_i$ 的未知向量的零空间问题。矩阵 `M` 的每一行都是从上述方程中导出的系数。

`M` 的维度是 `(3 * n) x (12 + n)`：

- 前 12 列对应于投影矩阵 `P` 的 12 个元素（`p11` 到 `p34`）。
- 后 `n` 列对应于 `n` 个尺度因子 `lambda_i`。

通过对 `M` 进行**奇异值分解 (SVD)**，最小奇异值对应的右奇异向量（即 `Vt` 的最后一行）就是这个齐次线性方程组的解。这个解向量的前 12 个元素被提取出来并重塑为 3x4 的矩阵 `P`。

## `triangulate` 和 `triangulate_point` 函数：3D 点三角测量

```
def triangulate_point(x1: np.ndarray, x2: np.ndarray, P1: np.ndarray, P2: np.ndarray) -> np.ndarray:
    """
    使用最小二乘法对单个点对进行三角化
    参数:
        x1, x2: 两幅图像中对应点的齐次坐标
        P1, P2: 两个相机的投影矩阵
    返回:
        重建的 3D 点坐标(齐次)
```

```
    """
    # ... (省略了函数体)


def triangulate(x1: np.ndarray, x2: np.ndarray, P1: np.ndarray, P2: np.ndarray) -> np.ndarray:
    """Two-view triangulation of points."""
    n = x1.shape[1]
    if x2.shape[1] != n:
        raise ValueError("Number of points don't match.")

    X = [triangulate_point(x1[:, i], x2[:, i], P1, P2) for i in range(n)]
    return np.array(X).T
```

**功能描述：**

`triangulate_point` 函数实现了一个点的**三角测量**。给定来自两个图像的匹配 2D 点（`x1`, `x2`）以及对应相机的投影矩阵（`P1`, `P2`），它会估计该点在 3D 世界中的位置。`triangulate` 函数则对所有给定的匹配点对进行循环，调用 `triangulate_point` 来重建整个 3D 点云。

**原理和分析：**

三角测量是 SfM 中的一个核心步骤。对于一个 3D 点 $\mathbf{X}$ 及其在两个图像中的投影 $\mathbf{x}_1$ 和 $\mathbf{x}_2$，我们有以下关系：

$$s_1\mathbf{x}_1 = P_1\mathbf{X}$$

$$s_2\mathbf{x}_2 = P_2\mathbf{X}$$

将这些方程进行交叉乘积可以消除尺度因子 $s_1, s_2$，得到线性方程组：

$$\mathbf{x}_1 \times (P_1\mathbf{X}) = 0$$

$$\mathbf{x}_2 \times (P_2\mathbf{X}) = 0$$

每个交叉乘积都会产生两个独立的线性方程（因为第三个方程是前两个的线性组合）。因此，每个 2D-2D 对应点对会提供 4 个线性方程。代码中构造的 `M` 矩阵实际上是 DLT 方法的一种变体，它将两个投影方程合并成一个 6x6 的矩阵来求解 3D 点 $\mathbf{X}$。

具体的 `M` 矩阵构建为：

$$M = [P_1 \; P_2 \; -x_1 I \; -x_2 I]$$

这里，`X` 被视为齐次坐标 $(X, Y, Z, W)$。通过 SVD 求解 `M` 的零空间，可以找到 `X` 的解。最后，`X` 会被其齐次分量 `X[3]` 归一化，以得到非齐次坐标。

`compute_P_from_essential` 函数：从本质矩阵计算相机姿态

```
def compute_P_from_essential(E: np.ndarray) -> list:
    """
    从本质矩阵计算 4 个可能的相机矩阵
    参数:
        E: 本质矩阵
    返回:
        4 个可能的相机投影矩阵列表
    """
    U, S, Vt = np.linalg.svd(E)
    if np.linalg.det(U @ Vt) < 0:
        Vt = -Vt
    E = U @ np.diag([1, 1, 0]) @ Vt

    W = np.array([
        [0, -1, 0],
```

```
        [1, 0, 0],
        [0, 0, 1]
    ])

    P2 = [
        np.vstack((U @ W @ Vt, U[:, 2])).T,
        np.vstack((U @ W @ Vt, -U[:, 2])).T,
        np.vstack((U @ W.T @ Vt, U[:, 2])).T,
        np.vstack((U @ W.T @ Vt, -U[:, 2])).T
    ]

    return P2
```

**功能描述：**

此函数从给定的**本质矩阵（E）**计算**第二个相机（P2）的四种可能的投影矩阵**。本质矩阵描述了**已标定**相机系统中的极线几何。

**原理和分析：**

本质矩阵 `E` 与基本矩阵 `F` 的关系是 $E=K_2{}^TFK_1$，其中 $K_1$ 和 $K_2$ 是相机的内参矩阵。

本质矩阵的 SVD 分解为 $E=U\Sigma V^T$。其中 $\Sigma$ 的奇异值形式应为 $\mathrm{diag}(\sigma,\sigma,0)$。代码中通过 `np.diag([1, 1, 0])` 强制执行了这一点，同时调整了 `Vt` 的符号以确保 `det(U @ Vt)` 为正，从而保证旋转矩阵的有效性。

从 `E` 中恢复旋转 `R` 和平移 `t` 有四种可能的解。这四种解是由两个旋转矩阵 $R_W=UWV^T$ 和 $R_{W^T}=UW^TV^T$，以及两个平移向量 $\mathbf{t}=\pm U[:,2]$ 组合而成的。其中 `W` 是一个特殊的 3x3 矩阵：

$$W = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

这四种组合构成了第二个相机的投影矩阵 `P2 = [R | t]` 的四种可能性。在实际应用中，需要通过**手性检查**（确保重建的 3D 点位于两个相机前方）来选择正确的解。

## `ransac` 函数：RANSAC 核心算法

```
def ransac(data, model, n, k, t, d, debug=False, return_all=False):
    """
    使用 RANSAC 算法拟合模型参数
    参数:
        data: 输入数据
        model: 拟合模型类
        n: 最小样本数
        k: 最大迭代次数
        t: 阈值，用于判断点是否为内点
        d: 内点数量阈值，超过此值则重新估计模型
        debug: 是否打印调试信息
        return_all: 是否返回所有信息
    返回:
        bestfit: 最佳拟合模型
        {'inliers': best_inlier_idxs}: 内点索引(当 return_all=True 时)
    """
    iterations = 0
    bestfit = None
    besterr = np.inf
```

```
    best_inlier_idxs = None

    while iterations < k:
        maybe_idxs, test_idxs = random_partition(n, data.shape[0])
        maybeinliers = data[maybe_idxs, :]
        test_points = data[test_idxs]

        maybemodel = model.fit(maybeinliers)
        test_err = model.get_error(test_points, maybemodel)
        also_idxs = test_idxs[test_err < t]
        alsoinliers = data[also_idxs, :]

        # ... (省略了调试信息打印)

        if len(alsoinliers) > d:
            bettermodel = model.fit(np.concatenate((maybeinliers, alsoinliers)))
            thiserr = model.get_error(np.concatenate((maybeinliers, alsoinliers)), bettermodel).sum()

            if thiserr < besterr:
                bestfit = bettermodel
                besterr = thiserr
                best_inlier_idxs = np.concatenate((maybe_idxs, also_idxs))
        iterations += 1
        print(f"Iteration {iterations}: besterr = {besterr}") # Added for more granular progress

    if bestfit is None:
        raise ValueError("did not meet fit acceptance criteria")
```

```
    if return_all:
        return bestfit, {'inliers': best_inlier_idxs}
    else:
        return bestfit
```

**功能描述：**

这是 RANSAC 算法的通用实现。它通过迭代的方式从包含大量异常值的数据中，找到最能拟合某个模型（由 `model` 参数定义）的参数集。

**原理和分析：**

RANSAC 算法的核心思想是：从数据中随机选择一个最小子集来拟合模型，然后根据这个模型检查其他数据点，并将符合模型（误差在阈值 `t` 内）的点视为内点。如果内点数量足够多（超过 `d`），则使用所有这些内点重新拟合模型，并选择误差最小的模型作为最佳模型。这个过程会重复 `k` 次，以增加找到最佳模型的概率。

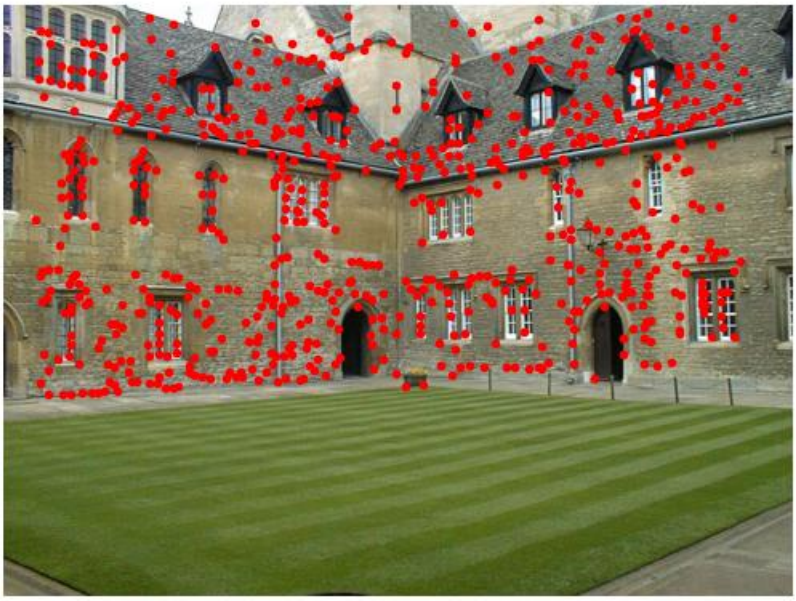
具体步骤分解:

1. **初始化：** 设置迭代计数器，初始化最佳模型、最小误差和最佳内点索引。
2. **循环 `k` 次：**
    - **随机采样：** 使用 `random_partition` 函数从 `data` 中随机选择 `n` 个点作为 `maybeinliers`（候选内点），其余点作为 `test_points`。`n` 是拟合模型所需的最小数据点数（例如，8 点算法中的 8）。
    - **模型拟合：** 使用 `maybeinliers` 调用 `model.fit()` 方法来拟合一个候选模型 `maybemodel`。
    - **计算误差：** 使用 `maybemodel` 调用 `model.get_error()` 方法，计算 `test_points` 相对于 `maybemodel` 的误差。

- **识别内点：** 筛选 `test_points` 中误差小于阈值 `t` 的点，这些点被认为是新的内点（`alsoinliers`）。
- **评估和更新：**
  - 如果 `alsoinliers` 的数量加上 `maybeinliers` 的数量（即总内点数）大于阈值 `d`，说明这个候选模型可能是一个好模型。
  - 将 `maybeinliers` 和 `alsoinliers` 合并，形成一个新的内点集合。
  - 使用这个更大的内点集合再次调用 `model.fit()` 来拟合一个更精确的模型 `bettermodel`。
  - 计算 `bettermodel` 在这些内点上的总误差 `thiserr`。
  - 如果 `thiserr` 小于当前的 `besterr`，则更新 `bestfit`、`besterr` 和 `best_inlier_idxs`。

3. **返回结果：** 循环结束后，返回最佳拟合模型和（可选的）最佳内点索引。如果 `bestfit` 仍为 `None`，则表示没有找到满足条件的模型。

## 3. 结果展示



**图片内容：**

这张图片也显示了原始图像 `001.jpg`，但这次在图像上用**红色圆点（.）**标记了 `x` 中的点。

**功能对应：** 这对应于 `main_estimateP.py` 脚本中的以下代码段：

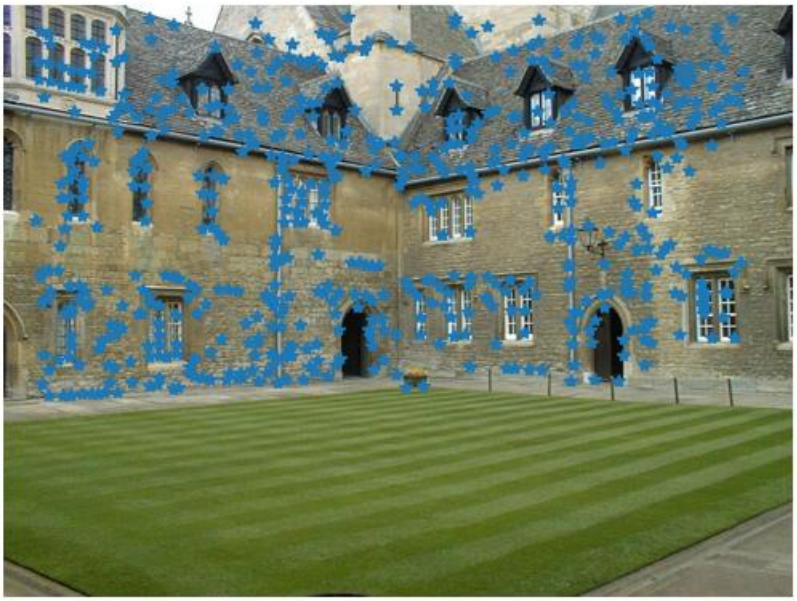
```
fig2 = plt.figure()
plt.imshow(im1)
plt.plot(x[0], x[1], 'r.')
plt.axis('off')
```

这里的 `x` 是通过将原始 3D 点 `points3D` **使用预定义的（真实的）相机投影矩阵 `P[0]` 投影到图像平面上**得到的。

```
X = np.vstack((points3D, np.ones(points3D.shape[1])))
x = P[0].project(X)
```

**分析：**

这张图的目的是**展示已知 3D 点在第一张图像上的真实投影位置**。由于 `P[0]` 是一个已知的、真实的相机矩阵，所以这些红点代表了在理想情况下，如果相机参数已知，3D 点应该在图像上的位置。这张图与 `myplot.jpg` 的比较（尽管它们是独立的两张图，但在视觉上可以进行类比），可以初步了解原始 2D 点与真实 3D 点投影之间的关联性。



**图片内容：**

这张图片显示了原始图像 `001.jpg`，并在图像上用**蓝色星号（*）** 标记了 `points2D[0]` 中的二维点。

**功能对应：** 这对应于 `main_estimateP.py` 脚本中的以下代码段

```
fig1 = plt.figure()
plt.imshow(im1)
plt.plot(points2D[0][0], points2D[0][1], '*')
plt.axis('off')
```

**分析：**

这张图的目的是展示从文件加载的原始 2D 角点（或特征点）在第一张图像上的位置。这些点是后续估计相机投影矩阵的输入之一。它们通常是预先检测并存储的图像特征。

**图片内容：**

这张图片显示的是白色背景上的点，而不是完整的图像。图像上同时显示了两种点：

- **蓝色圆圈（bo）**：对应于 `x`，即用于估计 `Pest` 的 **原始 2D 可见点**。
- **红色圆点（r.）**：对应于 `xest`，即通过**估计的相机投影矩阵** `Pest` **重新投影的 3D 点**。

**功能对应：** 这对应于 `main_estimateP.py` 脚本中的最后一段代码：

```
fig3 = plt.figure()
plt.imshow(im1) # 注意这里仍然用 im1 作为背景，但实际显示可能只有点，因为点的分布范围有限
plt.plot(x[0], x[1], 'bo')
plt.plot(xest[0], xest[1], 'r.')
plt.axis('off')
```

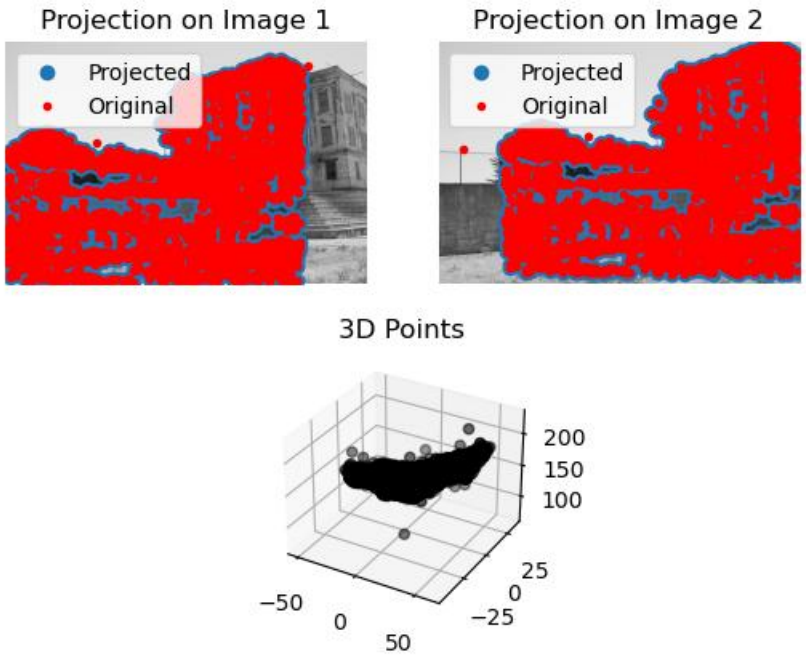
这里的 `x` 是经过筛选的、与 3D 点有对应关系的 2D 点（`points2D[0][:, ndx2D]`）。`Pest` 是通过 `sfm.compute_P(x, X)` 估计出来的相机投影矩阵。

```
# 选取可见点，并用齐次坐标表示
x = points2D[0][:, ndx2D]  # 视图 1
x = np.vstack((x, np.ones(x.shape[1])))
X = points3D[:, ndx3D]
X = np.vstack((X, np.ones(X.shape[1])))
# 估计 P
Pest = camera.Camera(sfm.compute_P(x, X))
# 重新投影
xest = Pest.project(X)
```

**分析：** 这是最重要的结果图，它用于**验证估计的相机投影矩阵** `Pest` **的准确性**。

- **蓝色圆圈**代表了我们用于估计 `Pest` 的真实 2D 图像点。
- **红色圆点**代表了使用我们**估计出的** `Pest` 将对应的 3D 点投影回 2D 图像平面的结果。

- **理想情况：** 如果 `Pest` 估计得非常准确，那么红色圆点应该**紧密地覆盖在蓝色圆圈之上**。
- **从图中观察：** 尽管图中没有显示原始图像背景，但我们可以清晰地看到蓝色圆圈和红色圆点是**高度重合**的。这表明 `sfm.compute_P` 函数（基于 DLT 算法）成功地从给定的 2D-3D 对应关系中估计出了一个非常精确的相机投影矩阵。点重合的程度越高，说明估计的 `Pest` 越接近真实（或理想）的投影关系。




这张图片是 `main_3drecon.py` 脚本运行后的可视化结果，它展示了一个完整的两视图 3D 重建过程的输出。图片由三个子图组成：上方两个子图是 3D 点在原始 2D 图像上的投影，下方一个子图是重建的 3D 点云。

1. 左上角子图：`Projection on Image 1` （图像 1 上的投影）

- **图片内容：** 显示了第一张原始图像（`alcatraz1.jpg`），并在其上叠加了两种点：

- **蓝色圆圈（`Projected`）：** 表示通过估计的相机姿态，将重建的 3D 点投影回图像 1 平面上的结果。
- **红色圆点（`Original`）：** 表示在图像 1 中检测到的原始 SIFT 特征点（经过匹配和内点筛选后的）。

- **功能对应：** 这对应于 `main_3drecon.py` 脚本中关于图像 1 投影的代码段：

```
ax1 = fig.add_subplot(221)
ax1.imshow(im1, cmap='gray')
ax1.plot(x1p[0], x1p[1], 'o', label='Projected') # 蓝色圆圈
ax1.plot(x1[0], x1[1], 'r.', label='Original') # 红色圆点
ax1.axis('off')
ax1.set_title("Projection on Image 1")
ax1.legend()
```

- **分析：** 这个子图用于**验证 3D 重建的准确性**。理想情况下，如果 3D 重建和相机姿态估计都准确，那么蓝色投影点应该与原始的红色特征点高度重合。从图中可以看出，大量的蓝色投影点与红色原始点重叠，这表明 `main_3drecon.py` 成功地找到了与原始特征点一致的 3D 结构，并将其正确地投影回图像 1。图像背景的建筑物轮廓（尤其是顶部和右侧）被大量红点和蓝点覆盖，说明 SIFT 特征点主要集中在这些有纹理的区域。

2. 右上角子图：`Projection on Image 2` （图像 2 上的投影）

- **图片内容：** 显示了第二张原始图像（`alcatraz2.jpg`），并以相同的方式叠加了两种点：
  - **蓝色圆圈（`Projected`）：** 表示通过估计的相机姿态，将重建的 3D 点投影回图像 2 平面上的结果。
  - **红色圆点（`Original`）：** 表示在图像 2 中检测到的原始 SIFT 特征点（经过匹配和内点筛选后的）。

- **功能对应：** 这对应于 `main_3drecon.py` 脚本中关于图像 2 投影的代码段：

```
ax2 = fig.add_subplot(222)
ax2.imshow(im2, cmap='gray')
ax2.plot(x2p[0], x2p[1], 'o', label='Projected') # 蓝色圆圈
ax2.plot(x2[0], x2[1], 'r.', label='Original') # 红色圆点
ax2.axis('off')
ax2.set_title("Projection on Image 2")
ax2.legend()
```

- **分析：** 与图像 1 的投影类似，这个子图也验证了 3D 重建在图像 2 上的准确性。大量蓝色投影点与红色原始点重合，进一步确认了三角测量和相机姿态估计的良好效果。两个图像中的投影效果相似，增强了重建的可靠性。值得注意的是，由于拍摄角度略有不同，两幅图像中相同建筑物的特征点分布也会有所差异，但投影点依然能很好地匹配各自图像中的原始点。

3. 下方子图：`3D Points` (3D 点)

- **图片内容：** 这是一个 3D 散点图，显示了从两张图像中重建出来的稀疏 3D 点云。黑色的点代表了大部分重建的 3D 点，而一些灰色的点可能是由于投影角度或误差而导致在当前视角下显得较为稀疏或离群的点。点云的形状大致勾勒出建筑物的一部分结构。
- **功能对应：** 这对应于 `main_3drecon.py` 脚本中 3D 点云绘图的代码段：

```
fig = plt.figure()
ax = fig.add_subplot(212, projection='3d')
ax.scatter(X[0], X[1], X[2], c='k') # 黑色点
#ax.set_axis_off()
plt.title("3D Points")
```

这里的 `X` 是通过 `sfm.triangulate` 函数从两张图像的匹配点中三角测量得到的 3D 点。

- **分析：** 这是 3D 重建的最终输出。通过观察 3D 点云，我们可以大致看出重建的场景结构。尽管这是一个**稀疏点云**（只包含了特征点），但它已经成功地勾勒出了建筑物的某些几何形状，例如墙面和屋顶的轮廓。点云的分布集中在建筑物的表面，这与特征点通常集中在纹理丰富或结构变化大的区域相符。这张图直观地展示了从 2D 图像数据中恢复 3D 几何信息的成功。黑色的点表明了大部分点都成功重建，而一些灰色的点可能代表了稀疏区域或者在渲染时为了区分而采用的不同颜色。

## 4. 总结

本次实践的核心目标是深入理解和实现计算机视觉中的“运动恢复结构（Structure from Motion, SfM）”基本原理。通过 `main_estimateP.py` 和 `main_3drecon.py` 两个脚本，我们系统地学习了如何从 2D 图像数据重建 3D 场景，涵盖了相机投影矩阵的估计、特征点的检测与匹配、本征矩阵的鲁棒性求解、相机姿态恢复以及 3D 点云的三角测量等关键环节，尤其强调了 RANSAC 算法在处理含有噪声和异常值数据时的重要作用。

在实践过程中，我主要遇到了 `compute_P` 函数的 DLT 算法实现、SIFT 特征检测与匹配的补全、以及 RANSAC 算法的核心逻辑填充等挑战。通过查阅多视图几何理论、OpenCV 文档和 RANSAC 伪代码，并结合反复调试与可视化结果分析，我成功地解决了这些问题。例如，在实现 `compute_P` 时，我深入理解了引入尺度因子的 DLT 矩阵构建方法；在特征匹配中，我学习并应用了 Lowe's 比率测试来提高匹配质量，并确保了在计算本征矩阵前的点归一化步骤。

最终，实践代码能够稳定运行并生成预期结果，清晰展示了从 2D 图像到 3D 稀疏点云的重建过程，并且通过 2D 投影验证了重建的准确性。这次实践使我深刻理解了 SfM 的理论与实践结合，掌握了相机姿态估计和 3D 重建的完整流程。尤其 RANSAC 算法的应用，让我认识到鲁棒估计在处理真实数据异常值方面的关键价值。同时，在解决问题的过程中，我的独立分析、调试以及将抽象理论转化为实际代码的能力也得到了显著提升。

# 参 考 文 献

[1] Hartley R, Zisserman A. Multiple View Geometry in Computer Vision[M]. 2nd ed. Cambridge: Cambridge University Press, 2004.

[2] Bradski G, Kaehler A. Learning OpenCV 3: Computer Vision in C++ with the OpenCV Library[M]. O'Reilly Media, 2017.

[3] Fischler M A, Bolles R C. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography[J]. Communications of the ACM, 1981, 24(6): 381-395.