SLAM Trick 1: Undistort Image And Point

陈烁龙

2022 年 4 月 28 日

目录

插图

表格

摘要

在使用相机的时候，为了获取更好的摄影效果，我们一般会在相机上安装透镜组。透镜的使用使得相机成像更加鲁棒，但是也带来了畸变问题。所以当我们拿到相机的成像照片时，需要通过去畸变技术，才能得到原始没有畸变的相片。

关键词：畸变，去畸变，高斯-牛顿法

1 畸变模型

对于镜头带来的畸变，我们可以将其分为两种：径向畸变和切向畸变。

1.1 径向畸变

在针孔相机模型中，我们认为光线是直线传播进入相机到达像平面的。但是在现实中，由于透镜的作用，光线不再是一条直线，而是一条在透镜处发生转向的曲线。而且在像平面上，离像主点越远的像素点，受该种效应的影响越明显。该种畸变效应可用如下的数学公式描述：

$$\begin{cases} x_{dist} = x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \\ y_{dist} = y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \\ r = \sqrt{x^2 + y^2} \end{cases} \tag{1}$$

其中 $X(x, y)$ 表示的是一个未经过径向畸变的归一化像素点坐标，$X_{dist}(x_{dist}, y_{dist})$ 表示的是该点经过径向畸变的归一化像素点坐标。可以从公式 1 清晰的看到，当归一化像素点坐标离中心点越远，由于半径 r 越大，因而造成了 X_{dist} 相比 X 差别更大，也就是我们所说的畸变比较大。

另外我们注意到，当对于中心点 X_c 而言，由于其 $X_{cx} = 0$，$X_{cy} = 0$，故 $r_c = 0$，所以 $X_{dist,c} = X_c$。即：径向畸变对中心点不造成畸变效应[1]。

[1]注意：我们所谈论的畸变模型都是在归一化像素坐标平面上的，并不是像素坐标平面。

1.2 切向畸变

对于切向畸变，我们直接给出畸变模型数学公式：

$$\begin{cases} x_{dist} = x + 2p_1 xy + p_2(r^2 + 2x^2) \\ y_{dist} = y + p_1(r^2 + 2y^2) + 2p_2 xy \\ r = \sqrt{x^2 + y^2} \end{cases} \tag{2}$$

同样地，$X(x, y)$ 表示的是一个未经过径向畸变的归一化像素点坐标，$X_{dist}(x_{dist}, y_{dist})$ 表示的是该归一化像素点经过切向畸变的坐标。

但是，和径向畸变不一样的一点是，即使是中心点，其 $r_c = 0$，但是仍然会遭受切向畸变的影响。

1.3 综合畸变

基于上文的介绍，我们综合径向畸变和切向畸变的数学模型，给出最终的综合畸变数学模型：

$$\begin{cases} x_{dist} = x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \\ \qquad\quad + 2p_1 xy + p_2(r^2 + 2x^2) \\ y_{dist} = y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \\ \qquad\quad + p_1(r^2 + 2y^2) + 2p_2 xy \\ r = \sqrt{x^2 + y^2} \end{cases} \tag{3}$$

该综合畸变数学模综合考虑了径向畸变和切向畸变。

2 问题的引出

由上面所给出的径向畸变和切向畸变的模型公式可知，当我们知道了一个未发生畸变的像素点，为求得其畸变后的像素位置，我们可以先基于相机的内参，将其转换到归一化像素坐标平面上。而后基于已有的畸变参数，对其增添畸变效应，得到在归一化像素坐标平面上的畸变后的

点，而后再基于相机的内参得到畸变后的像素坐标。

但是，这个过程是畸变发生的过程。对于我们而言，我们一开始拿到的是一张已经畸变后的像片，我们的目的是要通过去畸变处理，得到无畸变的像片。这个过程和刚刚所描述的过程刚好相反。

目前，对于去畸变，我们有两种措施[2]：

- 整个图像去畸变

  对于该种需求，我们其实很好解决。首先我们准备一张用来存储去畸变后像素的图像，然后对于该图像上的每一个像素 p_i，我们通过畸变模型计算得到其畸变后应该在畸变图像上的位置。这个位置对应了我们一开始拿到的那张已畸变的原始图像上的位置。

  当然，这个求得的位置一般不再是整数，而是浮点数，所以不会和某个畸变像素 $p_{dist,j}$ 对应上。这时我们可以通过插值操作，获得该浮点位置的像素值，然后直接赋值给去畸变后的像素。一般的插值方法有：最邻近法、双线性插值法[3]。这种方法在数字图象处理中被成为间接法。

  当然，如果通过畸变模型计算得到的畸变像素超过了图像的边界，我们可以简单的将其像素值设置为 0 或 255 或某一值的常数即可。

- 部分点去畸变

  我们的需求是给定一个已畸变的像素点 $p_{dist,i}$ 坐标，通过已有的相机模型和畸变参数，计算其对应的未畸变的像素点 p_i 坐标。一般我们是通过迭代的方式计算的。在本文中，我们通过高斯-牛顿法进行迭代求解。

[2]或者说需求更加合适。

[3]最邻近法速度最快，但是效果不佳。相比而言，双线性插值法速度虽然较其慢，但是能够得到比较鲁棒的插值结果。

3 求解

高斯-牛顿法首先通过计算误差函数对待求解参数的雅可比矩阵，以此构造一个线性方程。进而求解这个线性方程得到的每一次迭代更新的变化量。所以，求解雅可比矩阵就变得比较重要。

3.1 畸变过程描述

对于我们这个待求解的问题，我们要求解的参数是去畸变后的像素点坐标 $p(u,v)$。对于相机针孔模型，我们知道：

$$\begin{cases} u = f_x x + c_x \\ v = f_y y + c_y \end{cases} \tag{4}$$

其中 $X(x,y)$ 仍然表示的是归一化像素坐标平面上的点，f_x、f_y、c_x、c_y 表示的是相机的内参。

所以从未畸变像素点 $p(u,v)$ 到已畸变像素点 $p_{dist}(u_{dist}, v_{dist})$，我们有如下的转换公式：

$$\begin{cases} x = (u - c_x)/f_x \\ y = (v - c_y)/f_y \end{cases}$$

$$\rightarrow \begin{cases} x_{dist} = x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \\ \qquad\qquad + 2p_1 xy + p_2(r^2 + 2x^2) \\ y_{dist} = y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \\ \qquad\qquad + p_1(r^2 + 2y^2) + 2p_2 xy \\ r = \sqrt{x^2 + y^2} \end{cases}$$

$$\rightarrow \begin{cases} u_{dist} = f_x x_{dist} + c_x \\ v_{dist} = f_y y_{dist} + c_y \end{cases}$$

3.2 误差函数

基于上文的分析，我们很容易写出误差函数，即：

$$\begin{cases} e_u = \tilde{u}_{dist} - u_{dist} \\ e_v = \tilde{v}_{dist} - v_{dist} \end{cases} \tag{5}$$

其中 $p_{dist}(u_{dist}, v_{dist})$ 是我们通过模型计算得到的畸变像素坐标，$\tilde{p}_{dist}(\tilde{u}_{dist}, \tilde{v}_{dist})$ 是真实的畸变像素坐标。

3.3 雅可比矩阵求解

首先明确我们的误差项为 e_u、e_v，我们的待求解参数为 u、v。接下来我们基于每一个误差项，分别对每一个带求解参数进行求到，得到最终的雅可比矩阵。

1. e_u 对 u 的求导 $\partial e_u / \partial u$

$$\begin{aligned} \frac{\partial e_u}{\partial u} =& \frac{\partial e_u}{\partial u_{dist}} \times \frac{\partial u_{dist}}{\partial x_{dist}} \times \frac{\partial x_{dist}}{\partial x} \times \frac{\partial x}{\partial u} \\ =& \frac{\partial e_u}{\partial u_{dist}} \times \frac{\partial x_{dist}}{\partial x} \end{aligned}$$

$$\rightarrow \frac{\partial e_u}{\partial u_{dist}} = -1$$

$$\begin{aligned} \rightarrow \frac{\partial x_{dist}}{\partial x} =& (1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \\ & + x(2k_1 x + 4k_2 r^2 x + 6k_3 r^4 x) \\ & + 2p_1 y + 6p_2 x \end{aligned}$$

$$\begin{aligned} \rightarrow J_{uu} = \frac{\partial e_u}{\partial u} =& -(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \\ & - x(2k_1 x + 4k_2 r^2 x + 6k_3 r^4 x) \\ & - 2p_1 y - 6p_2 x \end{aligned}$$

2. e_u 对 v 的求导 $\partial e_u / \partial v$

$$\frac{\partial e_u}{\partial v} = \frac{\partial e_u}{\partial v_{dist}} \times \frac{\partial v_{dist}}{\partial y_{dist}} \times \frac{\partial y_{dist}}{\partial y} \times \frac{\partial y}{\partial v}$$

由于：

$$\frac{\partial e_u}{\partial v_{dist}} = 0$$

所以：

$$\rightarrow J_{uv} = \frac{\partial e_u}{\partial v} = 0$$

3. e_v 对 u 的求导 $\partial e_v / \partial u$

$$\frac{\partial e_v}{\partial u} = \frac{\partial e_u}{\partial u_{dist}} \times \frac{\partial u_{dist}}{\partial x_{dist}} \times \frac{\partial x_{dist}}{\partial x} \times \frac{\partial x}{\partial u}$$

由于：

$$\frac{\partial e_v}{\partial u_{dist}} = 0$$

所以：

$$\rightarrow J_{vu} = \frac{\partial e_v}{\partial u} = 0$$

4. e_v 对 v 的求导 $\partial e_v / \partial v$

$$\frac{\partial e_v}{\partial v} = \frac{\partial e_v}{\partial v_{dist}} \times \frac{\partial v_{dist}}{\partial y_{dist}} \times \frac{\partial y_{dist}}{\partial y} \times \frac{\partial y}{\partial v}$$

可以得到：

$$\begin{aligned} \rightarrow J_{vv} = \frac{\partial e_v}{\partial v} =& -(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \\ & - y(2k_1 y + 4k_2 r^2 y + 6k_3 r^4 y) \\ & - 2p_2 x - 6p_1 y \end{aligned}$$

综上可知：

$$J = \begin{pmatrix} J_{uu} & J_{vu} \\ J_{uv} & J_{vv} \end{pmatrix} = \begin{pmatrix} \frac{\partial e_u}{\partial u} & 0 \\ 0 & \frac{\partial e_v}{\partial v} \end{pmatrix} \tag{6}$$

令

$$\delta X = \begin{pmatrix} \delta u \\ \delta v \end{pmatrix}$$

则对于高斯牛顿法，有：

$$H \delta X = g \tag{7}$$

其中：

$$\begin{cases} H = JJ^T = \begin{pmatrix} J_{uu}^2 & 0 \\ 0 & J_{vv}^2 \end{pmatrix} \\ g = -Je(u,v) = \begin{pmatrix} -J_{uu}e_u \\ -J_{vv}e_v \end{pmatrix} \end{cases}$$

进而对公式 7 求解，得到：

$$\begin{cases} \delta u = -e_u / J_{uu} \\ \delta v = -e_v / J_{vv} \end{cases} \tag{8}$$

所以每一次迭代时，进行的增量为：

$$\begin{cases} u_{i+1} = u_i + \delta u \\ v_{i+1} = v_i + \delta v \end{cases} \tag{9}$$

当 δX 降低到一定的程度时，认为迭代收敛，可跳出循环，得到最终的结果。

4 实践

针对本文，我们进行了两个实验：基于整个图像的去畸变 (直接法) 和基于单个像素点的去畸变 (间接法 + 高斯牛顿)。

4.1 图像去畸变

首先描述以下我们自定义的一些结构体：

Listing 1: 自定义结构体

```
namespace ns_st0 {
  /**
   * @brief a structure for camera' inner parameters'
       organization
   */
  struct CameraInnerParam {
    float fx, fy, cx, cy;
    CameraInnerParam(const float &fx, const float &fy, const
        float &cx, const float &cy)
       : fx(fx), fy(fy), cx(cx), cy(cy) {}
  };

  /**
   * @brief a structure for camera' distortion parameters'
       organization
   */
  struct CameraDistCoeff {
    float k1, k2, k3;
    float p1, p2;

    CameraDistCoeff(const float &k1, const float &k2, const
        float &k3,
                    const float &p1, const float &p2)
       : k1(k1), k2(k2), k3(k3), p1(p1), p2(p2) {}
  };

  /**
   * @brief the different interpolation choices
   */
  enum class Interpolation {
    NEAREST_NEIGHBOR,
    BILINEAR
  };
}
```

以下代码是对于整张图像去畸变的关键代码。在该代码函数中，我们首先使用 CV_Assert 宏对传入的图像数据进行判断。而后基于用户传入的插值算法，构建相应的插值函数 (传入浮点像素位置和图像头部指针，返回插值结果)。而后基于构建的去畸变图像对象，遍历其每一个像素，通过直接法和畸变模型，得到其在畸变图像上的浮点位置，并插值得到最终的结果。

Listing 2: 图像去畸变

```
namespace ns_st1 {
  /**
   * @brief undistort a gray image
   *
   * @param src the distorted gray image [CV_8UC1]
   * @param innerParam the camera's inner parameters
   * @param distCoff the camera's distortion parameters
   * @param methods the interpolation choice
   * @return cv::Mat the undistorted gray image [CV_8UC1]
   */
  static cv::Mat undistortImage(
      cv::Mat src,
      const CameraInnerParam &innerParam,
      const CameraDistCoeff &distCoff,
      Interpolation methods = Interpolation::NEAREST_NEIGHBOR)
          {
    // assert
    CV_Assert(src.type() == CV_8UC1);
    CV_Assert(!src.empty());

    // get parameters
```

```
    int rows = src.rows, cols = src.cols;
    float fx = innerParam.fx, fy = innerParam.fy, cx =
        innerParam.cx, cy = innerParam.cy;
    float k1 = distCoff.k1, k2 = distCoff.k2, k3 = distCoff.k3
        ;
    float p1 = distCoff.p1, p2 = distCoff.p2;

    // define the interpolation function
    std::function<char((float, float, uchar *))> interpolation
        ;

    switch (methods) {
    case Interpolation::NEAREST_NEIGHBOR: {
      interpolation = [cols, rows](float u, float v, uchar *
          headPtr) {
        return *(headPtr + int(v + 0.5) * cols + int(u + 0.5))
            ;
      };
    } break;
    case Interpolation::BILINEAR: {
      interpolation = [cols, rows](float u, float v, uchar *
          headPtr) {
        // four corners
        int u_lt = int(u), v_lt = int(v);
        int u_rt = u_lt + 1, v_rt = v_lt;
        int u_lb = u_lt, v_lb = v_lt + 1;
        int u_rb = u_lt + 1, v_rb = v_lt + 1;

        // range check for u
        (u_lt < 0) ? (u_lt = 0) : (0), (u_lt > cols - 1) ? (
            u_lt = cols - 1) : (0);
        (u_rt < 0) ? (u_rt = 0) : (0), (u_rt > cols - 1) ? (
            u_rt = cols - 1) : (0);
        (u_lb < 0) ? (u_lb = 0) : (0), (u_lb > cols - 1) ? (
            u_lb = cols - 1) : (0);
        (u_rb < 0) ? (u_rb = 0) : (0), (u_rb > cols - 1) ? (
            u_rb = cols - 1) : (0);

        // range check for v
        (v_lt < 0) ? (v_lt = 0) : (0), (v_lt > rows - 1) ? (
            v_lt = rows - 1) : (0);
        (v_rt < 0) ? (v_rt = 0) : (0), (v_rt > rows - 1) ? (
            v_rt = rows - 1) : (0);
        (v_lb < 0) ? (v_lb = 0) : (0), (v_lb > rows - 1) ? (
            v_lb = rows - 1) : (0);
        (v_rb < 0) ? (v_rb = 0) : (0), (v_rb > rows - 1) ? (
            v_rb = rows - 1) : (0);

        // the gray values for four corners
        uchar lt = *(headPtr + v_lt * cols + u_lt);
        uchar rt = *(headPtr + v_rt * cols + u_rt);
        uchar lb = *(headPtr + v_lb * cols + u_lb);
        uchar rb = *(headPtr + v_rb * cols + u_rb);

        // bilinear
        float v1 = (u - int(u)) * rt + (1 - u + int(u)) * lt;
        float v2 = (u - int(u)) * rb + (1 - u + int(u)) * lb;

        uchar v3 = (v - int(v)) * v2 + (1 - v + int(v)) * v1
            + 0.5;

        return v3;
      };
    } break;
    default: {
      interpolation = [cols, rows](float u, float v, uchar *
          headPtr) {
        if (u < 0 || u > cols - 1 || v < 0 || v > rows - 1) {
          return uchar(0);
        } else {
          return *(headPtr + int(v) * cols + int(u));
        }
      };
    } break;
    }

    // the undistorted image
    cv::Mat dst(rows, cols, CV_8UC1);
    uchar *srcHead = src.ptr<uchar>(0);

    for (int v = 0; v != rows; ++v) {
      auto dstPtr = dst.ptr<uchar>(v);

      for (int u = 0; u != cols; ++u) {

        // transfrom to the normalized pixel coordinate
        float x = (u - cx) / fx;
        float y = (v - cy) / fy;
        float r2 = x * x + y * y, r4 = r2 * r2, r6 = r4 * r2;

        // the distortion model
        float x_dist = x * (1 + k1 * r2 + k2 * r4 + k3 * r6) +
            2 * p1 * x * y + p2 * (r2 + 2 * x * x);
        float y_dist = y * (1 + k1 * r2 + k2 * r4 + k3 * r6) +
            2 * p2 * x * y + p1 * (r2 + 2 * y * y);

        // transfrom to pixel coordinate
        float u_dist = x_dist * fx + cx;
        float v_dist = y_dist * fy + cy;

        // range check
        if (u_dist < 0 || u_dist > cols - 1 || v_dist < 0 ||
            v_dist > rows - 1) {
          dstPtr[u] = 0;
        } else {
          // interpolation
          dstPtr[u] = interpolation(u_dist, v_dist, srcHead);
        }
      }
    }

    return dst;
  }
} // namespace ns_st1
```

(a) 原始畸变图像

(b) 去畸变图像 (最近邻插值)

(c) 去畸变图像 (双线性插值)

图 1: 对整张图像去畸变

图 1 为对整张图像去畸变得到的结果，其中图 1(a) 为原始的已畸变图像，图 1(b) 为使用最邻近插值法获得的去畸变图像，而图 1(c) 为使用双线性插值法获得的去畸变图像。可以明显的看到，相较于最邻近插值，双线性插值处理德更加精细，但是其耗时也更多。在本人的电脑上，使用 *Release* 编译模式时，最邻近插值法耗时 5.18229(MS)，而双线性插值法耗时 8.25476(MS)，基本上相差了 1.5 倍。结果如下列表所示。

Listing 3: 不同插值方法对比

```
{'undistorted-nearest': 5.18229(MS)}
{'undistorted-bilinear': 8.25476(MS)}
```

4.2 像素点去畸变

以下代码列表时基于我们之前推导的高斯-牛顿法，求解一个已畸变点对应的未畸变像素坐标。首先我们使用已畸变像素点的坐标作为待求解去畸变像素点的初值。而后在每一次迭代计算中，我们首先将现有的去畸变点的坐标转换到归一化像素坐标平面上，而后计算相应的雅各比矩阵，畸变模型，再将其转到畸变后的像素坐标里，得到误差。最后根据雅各比矩阵和误差，求解更新量，并对带估计参数进行更新。当更新量小到一定程度或者达到迭代次数限制的时候，结束循环，返回结果。

Listing 4: 像素点去畸变

```
namespace ns_st1 {
  /**
   * @brief undistort a pixel point
   *
   * @param srcPt the distorted point
   * @param innerParam the camera's inner parameters
   * @param distCoff the camera's distortion parameters
   * @param threshold the threshold to stop iterator
   * @param iterator the interator times
   * @return cv::Point2f the undistoerted pixel point
   */
  static cv::Point2f undistortPoint(
      cv::Point2f srcPt,
      const CameraInnerParam &innerParam,
      const CameraDistCoeff &distCoff,
      float threshold = 1E-5,
      int iterator = 5) {
    // check input parameters
    CV_Assert(threshold > 0.0f);
    CV_Assert(iterator > 0);

    // get parameters
    float fx = innerParam.fx, fy = innerParam.fy, cx =
        innerParam.cx, cy = innerParam.cy;
    float k1 = distCoff.k1, k2 = distCoff.k2, k3 = distCoff.k3
        ;
    float p1 = distCoff.p1, p2 = distCoff.p2;
    float x_est = srcPt.x, y_est = srcPt.y;
```

```
27
28      for (int i = 0; i != iterator; ++i) {
29        // transform to normalized pixel coordinate plane
30        float x = (x_est - cx) / fx;
31        float y = (y_est - cy) / fy;
32        float r2 = x * x + y * y, r4 = r2 * r2, r6 = r4 * r2;
33
34        // compute the jacobian matrix
35        float Juu = -(1.0f + k1 * r2 + k2 * r4 + k3 * r6) -
36                    x * (2.0f * k1 * x + 4.0f * k2 * r2 * x + 6.0
                            f * k3 * r4 * x) -
37                    2.0f * p1 * y - 6.0f * p2 * x;
38
39        float Jvv = -(1.0f + k1 * r2 + k2 * r4 + k3 * r6) -
40                    y * (2.0f * k1 * y + 4.0f * k2 * r2 * y + 6.0
                            f * k3 * r4 * y) -
41                    2.0f * p2 * x - 6.0f * p1 * y;
42
43        // the distortion model
44        float x_dist = x * (1 + k1 * r2 + k2 * r4 + k3 * r6) + 2
                * p1 * x * y + p2 * (r2 + 2 * x * x);
45        float y_dist = y * (1 + k1 * r2 + k2 * r4 + k3 * r6) + 2
                * p2 * x * y + p1 * (r2 + 2 * y * y);
46
47        // transfrom to pixel coordinate plane
48        float u_dist = x_dist * fx + cx;
49        float v_dist = y_dist * fy + cy;
50
51        // comput the error
52        float eu = srcPt.x - u_dist;
53        float ev = srcPt.y - v_dist;
54
55        // the variation
56        float delte_u = -eu / Juu;
57        float delte_v = -ev / Jvv;
58
59        // update
60        x_est += delte_u;
61        y_est += delte_v;
62
63        float variation = delte_u * delte_u + delte_v * delte_v;
64
65        if (variation < threshold) {
66          break;
67        }
68      }
69
70      return cv::Point2f(x_est, y_est);
71    }
72  } // namespace ns_st1
```

如下列表所示，我在同一台机器上统计了使用 *OpenCV* 内置的去畸变函数 *cv* :: *undistortPoints* 和我实现的去畸变算法。可以看到，对于同一个待去畸变的像素点，通过高斯-牛顿法解算的结果误差是 *OpenCV* 版本的 1/15 左右，且速度比其快 5 倍，效果非常鲁棒。

Listing 5: 不同像素点去畸变版本对比

```
1  ————————————
2  gauss—newton
3  ————————————
4  {'undistort point cost': 0.00903(MS)}
5  srcPt(distorted): [188.00000, 120.00000]
6  dstPt(undistorted): [174.34048, 110.19159]
7  error: 0.000043158372137
8  ———————
9  OpenCV
10 ———————
11 {'undistort point cost': 0.05889(MS)}
12 srcPt(distorted): [188.00000, 120.00000]
13 dstPt(undistorted): [174.34122, 110.19208]
14 error: 0.000723787932657
```

5 GitHub

以下链接为该项目在 *GitHub* 上的地址，点击他，克隆它，使用它:

https://github.com/Unsigned-Long/slam-tricks/tree/master/st1-undistort