深蓝学院
shenlanxueyuan.com

第六次作业讲解

主讲人 王一鸣

1. LK 光流　光流文献综述

(1) 按此文的分类,光流法可分为哪几类?

答: a. 根据是否估计incremental warp并且与当前估计组合分为additive和compositional;

b. 根据是否在模板图像计算梯度分为forwards和inverse;

c. 根据梯度下降中所用优化方法分为:

- Gauss-Newton (GN)
- Newton (N)
- Steepest-Descent (SD)
- Gauss-Newton Diagonal (Diag-GN)
- Newton Diagonal (Diag-N)
- Levenberg-Marquardt (LM)

In this paper, Part 1 in a series of papers, we begin in Section 2 by reviewing the Lucas-Kanade algorithm. We proceed in Section 3 to analyze the quantity that is approximated by the various image alignment algorithms and the warp update rule that is used. We categorize algorithms as either *additive* or *compositional*, and as either *forwards* or *inverse*. We prove the first order equivalence of the various alternatives, derive the efficiency of the resulting algorithms, describe the set of warps that each alternative can be

图1

1. LK 光流 光流文献综述

(2) 在 compositional 中,为什么有时候需要做原始图像的 wrap?该 wrap 有何物理意义?
答: warp即考虑图像块在不同相机中发生了仿射变换。带 warp 之后对旋转更加鲁棒.

(3)forward 和 inverse 有何差别?
答:forward 和 inverse 主要在计算梯度时有所不同。

the forwards compositional algorithm in Fig. 3 therefore need only be performed once as a pre-computation, rather than once per iteration. The only differences between the forwards and inverse compositional algorithms (see Figs. 3 and 4) are: (1) the error image is computed after switching the roles of I and T, (2) Steps 3, 5, and 6 use the gradient of T rather than the gradient of I and can be pre-computed, (3) Eq. (35) is used to compute $\Delta \mathbf{p}$ rather than Eq. (10), and finally (4) the incremental warp is inverted before it is composed with the current estimate in Step 9.

Note that inversion of the Hessian could be moved from Step 8 to Step 6. Moreover, the inverse Hessian can be pre-multiplied by the steepest-descent images. We have not described these minor improvements in Fig. 4 so as to present the algorithms in a unified way.

图2

1. LK 光流 FAGN光流的实现

(1) 从最小二乘角度来看,每个像素的误差怎么定义?

答: $error = I_1(x_i, y_i) - I_2(x_i + \Delta x_i, y_i + \Delta y_i)$ 等式1

(2) 误差相对于自变量的导数如何定义?

答: 导数即为该点当前估计位置的目标图像梯度

$$\frac{\partial e}{\partial p} = \nabla I \frac{\partial W}{\partial p} = [\frac{\partial I}{\partial x}, \frac{\partial I}{\partial y}]$$ 等式2

2.2. *Derivation of the Lucas-Kanade Algorithm*

The Lucas-Kanade algorithm (which is a Gauss-Newton gradient descent non-linear optimization algorithm) is then derived as follows. The non-linear expression in Eq. (4) is linearized by performing a first order Taylor expansion on $I(\mathbf{W}(\mathbf{x};\mathbf{p}+\Delta\mathbf{p}))$ to give:

$$\sum_{\mathbf{x}} \left[I(\mathbf{W}(\mathbf{x};\mathbf{p})) + \nabla I \frac{\partial \mathbf{W}}{\partial \mathbf{p}} \Delta\mathbf{p} - T(\mathbf{x}) \right]^2. \quad (6)$$

In this expression, $\nabla I = (\frac{\partial I}{\partial x}, \frac{\partial I}{\partial y})$ is the *gradient* of image I evaluated at $\mathbf{W}(\mathbf{x};\mathbf{p})$; i.e. ∇I is computed in the coordinate frame of I and then warped back onto the coordinate frame of T using the current estimate of the warp $\mathbf{W}(\mathbf{x};\mathbf{p})$. The term $\frac{\partial \mathbf{W}}{\partial \mathbf{p}}$ is the *Jacobian* of the warp. If $\mathbf{W}(\mathbf{x};\mathbf{p}) = (W_x(\mathbf{x};\mathbf{p}), W_y(\mathbf{x};\mathbf{p}))^{\mathrm{T}}$ then:

图3

1. LK 光流 FAGN光流的实现

代码实现:

The Lucas-Kanade Algorithm

Iterate:

(1) Warp I with $\mathbf{W}(\mathbf{x};\mathbf{p})$ to compute $I(\mathbf{W}(\mathbf{x};\mathbf{p}))$

(2) Compute the error image $T(\mathbf{x}) - I(\mathbf{W}(\mathbf{x};\mathbf{p}))$

(3) Warp the gradient ∇I with $\mathbf{W}(\mathbf{x};\mathbf{p})$

(4) Evaluate the Jacobian $\frac{\partial \mathbf{W}}{\partial \mathbf{p}}$ at $(\mathbf{x};\mathbf{p})$

(5) Compute the steepest descent images $\nabla I \frac{\partial \mathbf{W}}{\partial \mathbf{p}}$

(6) Compute the Hessian matrix using Equation (11)

(7) Compute $\sum_{\mathbf{x}} [\nabla I \frac{\partial \mathbf{W}}{\partial \mathbf{p}}]^{\mathrm{T}} [T(\mathbf{x}) - I(\mathbf{W}(\mathbf{x};\mathbf{p}))]$

(8) Compute $\Delta \mathbf{p}$ using Equation (10)

(9) Update the parameters $\mathbf{p} \leftarrow \mathbf{p} + \Delta \mathbf{p}$

until $\|\Delta \mathbf{p}\| \leq \epsilon$

图4

1. LK 光流 FAGN光流的实现

代码实现:

```
// compute cost and jacobian
for (int x = -half_patch_size; x < half_patch_size; x++)
    for (int y = -half_patch_size; y < half_patch_size; y++) {

        double error = GetPixelValue(img1, kp.pt.x + x, kp.pt.y + y) -
                       GetPixelValue(img2, kp.pt.x + x + dx, kp.pt.y + y + dy);
        Eigen::Vector2d J;
        if (inverse == false) {
            J = -1.0 * Eigen::Vector2d(
                    0.5 * (GetPixelValue(img2, kp.pt.x + dx + x + 1, kp.pt.y + dy + y) -
                           GetPixelValue(img2, kp.pt.x + dx + x - 1, kp.pt.y + dy + y)),
                    0.5 * (GetPixelValue(img2, kp.pt.x + dx + x, kp.pt.y + dy + y + 1) -
                           GetPixelValue(img2, kp.pt.x + dx + x, kp.pt.y + dy + y - 1))
            );
        } else {
            // NOTE this J does not change when dx, dy is updated, so we can store it and only compute error
            J = -1.0 * Eigen::Vector2d(
                    0.5 * (GetPixelValue(img1, kp.pt.x + x + 1, kp.pt.y + y) -
                           GetPixelValue(img1, kp.pt.x + x - 1, kp.pt.y + y)),
                    0.5 * (GetPixelValue(img1, kp.pt.x + x, kp.pt.y + y + 1) -
                           GetPixelValue(img1, kp.pt.x + x, kp.pt.y + y - 1))
            );
        }

        H += J * J.transpose();
        b += -error * J;
        cost += error * error;
    }

Eigen::Vector2d update = H.ldlt().solve(b);
```

$$J=\frac{\partial e}{\partial p}=[\frac{\partial e}{\partial dx},\frac{\partial e}{\partial dy}]^T$$

$$=-[\frac{\partial I(x+dx,y+dy)}{\partial dx},\frac{\partial I(x+dx,y+dy)}{\partial dy}]^T$$

$$=-\nabla I\frac{\partial W}{\partial p}=-[\frac{\partial I}{\partial x},\frac{\partial I}{\partial y}]\begin{bmatrix}1&0\\0&1\end{bmatrix}=-[\frac{\partial I}{\partial x},\frac{\partial I}{\partial y}]$$

等式3

1. LK 光流 推广至金字塔

(1)所谓 coarse-to-fine 是指怎样的过程?
答: coarse-to-fine 是说从最粗糙的顶层金字塔开始向下迭代,不断细化估计的过程。

(2)光流法中的金字塔用途和特征点法中的金字塔有何差别?
答: 特征点法的金字塔主要用于不同层级之间的匹配,以使得匹配对缩放不敏感。光流中金字塔主要用于 coarse-to-fine 的估计。

1. LK 光流 推广至金字塔

代码实现：

```
    // create pyramids
    vector<Mat> pyr1, pyr2; // image pyramids
    for (int i = 0; i < pyramids; i++) {
        if (i == 0) {
            pyr1.push_back(img1);
            pyr2.push_back(img2);
        } else {
            Mat img1_pyr, img2_pyr;
            cv::resize(pyr1[i - 1], img1_pyr,
                       cv::Size(pyr1[i - 1].cols * pyramid_scale, pyr1[i - 1].rows * pyramid_scale));
            cv::resize(pyr2[i - 1], img2_pyr,
                       cv::Size(pyr2[i - 1].cols * pyramid_scale, pyr2[i - 1].rows * pyramid_scale));
            pyr1.push_back(img1_pyr);
            pyr2.push_back(img2_pyr);
        }
    }
```

1. LK 光流 推广至金字塔

代码实现：

```
    // LK tracking in pyramids
    vector<KeyPoint> kp1_pyr, kp2_pyr;
    for (auto &kp:kp1) {
        auto kp_top = kp;
        kp_top.pt *= scales[pyramids - 1];
        kp1_pyr.push_back(kp_top);
    }

    for (int level = pyramids - 1; level >= 0; level--) {
        // from coarse to fine
        success.clear();
        OpticalFlowSingleLevel(pyr1[level], pyr2[level], kp1_pyr, kp2_pyr, success, inverse);

        if (level > 0) {
            // update kp1_pyr and kp2_pyr to next level
            for (auto &kp: kp1_pyr)
                kp.pt /= pyramid_scale;
            for (auto &kp: kp2_pyr)
                kp.pt /= pyramid_scale;
        }
    }

    // set to kp2
    for (auto &kp: kp2_pyr)
        kp2.push_back(kp);
```

1. LK 光流　讨论

(1)我们优化两个图像块的灰度之差真的合理吗?哪些时候不够合理?你有解决办法吗?
答:灰度不变假设不满足时即不合理。可以考虑去掉均值(Zero-mean),加入曝光时间等做法.

(2)图像块大小是否有明显差异?取 16x16 和 8x8 的图像块会让结果发生变化吗?
答:差异不大,用大图像块时比较耗时,太小时结果不稳定。

(3)金字塔层数对结果有怎样的影响?缩放倍率呢?
答:差别均不大,2 倍缩放倍率比较容易计算。层数太少时结果不稳定。

2. 直接法 单层直接法

深蓝学院 shenlanxueyuan.com

(1) 该问题中的误差项是什么?

答: $e_i(\xi) = I_1(p_1, i) - I_2(p_2, i)$ 等式4

(2) 误差相对于自变量的雅可比维度是多少?如何求解?

答:雅可比为 1 × 6, 用链式法则求解

$$\boldsymbol{J} = -\frac{\partial \boldsymbol{I}_2}{\partial \boldsymbol{u}} \frac{\partial \boldsymbol{u}}{\partial \delta \boldsymbol{\xi}}. \quad \frac{\partial \boldsymbol{u}}{\partial \delta \boldsymbol{\xi}} = \begin{bmatrix} \frac{f_x}{Z} & 0 & -\frac{f_x X}{Z^2} & -\frac{f_x XY}{Z^2} & f_x + \frac{f_x X^2}{Z^2} & -\frac{f_x Y}{Z} \\ 0 & \frac{f_y}{Z} & -\frac{f_y Y}{Z^2} & -f_y - \frac{f_y Y^2}{Z^2} & \frac{f_y XY}{Z^2} & \frac{f_y X}{Z} \end{bmatrix}$$

等式5

(3) 窗口可以取多大?是否可以取单个点?

答:窗又可以取 4x4, 8x8 等, 单点亦可, 但要求点数要够多

2. 直接法 单层直接法

代码实现：

```
    for (size_t i = 0; i < px_ref.size(); i++) {

        // compute the projection in the second image
        Eigen::Vector3d point_ref =
            depth_ref[i] * Eigen::Vector3d((px_ref[i][0] - cx) / fx, (px_ref[i][1] - cy) / fy, 1);
        Eigen::Vector3d point_cur = T21 * point_ref;
        if (point_cur[2] < 0)
            continue;

        float u = fx * point_cur[0] / point_cur[2] + cx, v = fy * point_cur[1] / point_cur[2] + cy;
        if (u < half_patch_size || u > img2.cols - half_patch_size || v < half_patch_size ||
            v > img2.rows - half_patch_size)
            continue;

        double X = point_cur[0], Y = point_cur[1], Z = point_cur[2], Z2 = Z * Z;
        nGood++;
        goodProjection.push_back(Eigen::Vector2d(u, v));
```

计算当前像素点在输入图像上的投影

标记投影在内部的点

2. 直接法 单层直接法

计算光度误差

```
// compute cost and jacobian
for (int x = -half_patch_size; x < half_patch_size; x++)
    for (int y = -half_patch_size; y < half_patch_size; y++) {

        double error = GetPixelValue(img1, px_ref[i][0] + x, px_ref[i][1] + y) -
                       GetPixelValue(img2, u + x, v + y);
        Matrix26d J_pixel_xi;
        Eigen::Vector2d J_img_pixel;

        J_pixel_xi(0, 0) = fx / Z;
        J_pixel_xi(0, 1) = 0;
        J_pixel_xi(0, 2) = -fx * X / Z2;
        J_pixel_xi(0, 3) = -fx * X * Y / Z2;
        J_pixel_xi(0, 4) = fx + fx * X * X / Z2;
        J_pixel_xi(0, 5) = -fx * Y / Z;

        J_pixel_xi(1, 0) = 0;
        J_pixel_xi(1, 1) = fy / Z;
        J_pixel_xi(1, 2) = -fy * Y / Z2;
        J_pixel_xi(1, 3) = -fy - fy * Y * Y / Z2;
        J_pixel_xi(1, 4) = fy * X * Y / Z2;
        J_pixel_xi(1, 5) = fy * X / Z;
```

$$\frac{\partial u}{\partial \delta \xi}$$

2. 直接法 单层直接法

```
            J_img_pixel = Eigen::Vector2d(
                    0.5 * (GetPixelValue(img2, u + 1, v) - GetPixelValue(img2, u - 1, v)),
                    0.5 * (GetPixelValue(img2, u, v + 1) - GetPixelValue(img2, u, v - 1))
            );

            Vector6d J = -1.0 * (J_img_pixel.transpose() * J_pixel_xi).transpose();    // should be 1x6

            H += J * J.transpose();
            b += -error * J;
            cost += error * error;
        }
    }

    Vector6d update = H.ldlt().solve(b);
    T21 = Sophus::SE3::exp(update) * T21;
    cost /= nGood;
```

像素梯度

计算雅克比矩阵

2. 直接法 多层直接法

```
    // create pyramids
    vector<cv::Mat> pyr1, pyr2; // image pyramids
    for (int i = 0; i < pyramids; i++) {
        if (i == 0) {
            pyr1.push_back(img1);
            pyr2.push_back(img2);
        } else {
            cv::Mat img1_pyr, img2_pyr;
            cv::resize(pyr1[i - 1], img1_pyr,
                       cv::Size(pyr1[i - 1].cols * pyramid_scale, pyr1[i - 1].rows * pyramid_scale));
            cv::resize(pyr2[i - 1], img2_pyr,
                       cv::Size(pyr2[i - 1].cols * pyramid_scale, pyr2[i - 1].rows * pyramid_scale));
            pyr1.push_back(img1_pyr);
            pyr2.push_back(img2_pyr);
        }
    }

    // scale fx, fy, cx, cy in different pyramid levels
    double fxG = fx, fyG = fy, cxG = cx, cyG = cy;
    for (int level = pyramids - 1; level >= 0; level--) {
        VecVector2d px_ref_pyr;
        for (auto &px: px_ref) {
            px_ref_pyr.push_back(scales[level] * px);
        }
        fx = fxG * scales[level];
        fy = fyG * scales[level];
        cx = cxG * scales[level];
        cy = cyG * scales[level];

        DirectPoseEstimationSingleLayer(pyr1[level], pyr2[level], px_ref_pyr, depth_ref, T21);
    }
```

2. 直接法 延伸讨论

(1)直接法是否可以类似光流,提出 inverse, compositional 的概念?它们有意义吗?
答:可以。inverse 即取原始图像中梯度,compositional 即同时估计仿射变换参数。

(2)请思考上面算法哪些地方可以缓存或加速?
答:图像梯度可以事先算好,直接使用。用 inverse 方法的话,只需要计算残差,H 不变。

(3) 在上述过程中,我们实际假设了哪两个 patch 不变?
答:关键点灰度值不变,和深度信息不变

(4) 为何可以随机取点?而不用取角点或线上的点?那些不是角点的地方,投影算对了吗?
答:因为没有匹配过程,算法不依赖角点。

2. 直接法 延伸讨论

(5)请总结直接法相对于特征点法的异同与优缺点。

答:

方法	优点	缺点
直接法	(1)省略特征提取的时间 (2)只需要有像素梯度而不必是角点 (3)可稠密或半稠密	(1)灰度不变难以满足，易受曝光和模糊影响 (2)单像素区分性差 (3)相机发生大尺度移动或旋转时，无法很好的追踪，非凸优化，容易局部极值
特征点法	运动过大时，只要匹配点还在像素内，则不会引起误匹配，相对于直接法有更好的鲁棒性	(1)特征过多过少都无法正常工作 (2)只能用来构建稀疏地图 (3)环境特征少，或者无法提取角点，例如渐变色等状况，都无法正常工作 (4)计算量大

3. 使用光流计算视差

本题包含两步:

1. 使用LK光流法追踪得到对应点对;
2. 计算视差,即左右图的横坐标之差.

disparity=left_keypoint.x-right.keypoint.x

```
    // track these keypoints and estimate the disparity
    // note that you need to reject some wrong tracking results
    vector<KeyPoint> kp2;
    vector<bool> success;
    OpticalFlowMultiLevel(left_img, right_img, kp1, kp2, success, true);

    float error = 0;
    int n_error = 0, bad = 0;
    for (size_t i = 0; i < kp1.size(); i++) {
        if (success[i]) {
            float dx = kp1[i].pt.x - kp2[i].pt.x, dy = kp1[i].pt.y - kp2[i].pt.y;
            int disparity_gt = disparity_img.at<uchar>(kp1[i].pt.y, kp1[i].pt.x);
            error += (disparity_gt - dx) * (disparity_gt - dx);
            cout << "true: " << disparity_gt << ", estimated: " << dx << endl;
            if (fabs(disparity_gt - dx) > 2)
                bad++;
            n_error++;
        }
    }
```

感谢各位聆听!

Thanks for Listening