Visual Odomentry 代码阅读报告

唐誉铭

2019 年 1 月 29 日

1　论文概要

1.1　论文要解决的问题

论文中提出了一种基于双镜摄像头拍摄的视频进行三维重建的方法。3D 感知是计算机视觉和机器人技术的核心课题之一。在实践中，相机的分辨率严重受限，并且对生成结果的准确性有着较高的要求，

2　代码分析

2.1　整体流程分析

2.1.1　demo 程序分析

```
VisualOdometryStereo::parameters param;

// calibration parameters for sequence 2010_03_09_drive_0019
param.calib.f  = 645.24; // focal length in pixels
param.calib.cu = 635.96; // principal point (u-coordinate) in
  ↪ pixels
param.calib.cv = 194.13; // principal point (v-coordinate) in
  ↪ pixels
param.base     = 0.5707; // baseline in meters
//新建一个参数，并且设置参数
// init visual odometry
```

```
VisualOdometryStereo viso(param);
//通过参数初始化一个VisualOdometryStereo实例
```

新建参数并且初始化一个实例

```
int32_t dims[] = {width,height,width};
if (viso.process(left_img_data,right_img_data,dims)) {

// on success, update current pose
pose = pose * Matrix::inv(viso.getMotion());

// output some statistics
double num_matches = viso.getNumberOfMatches();
double num_inliers = viso.getNumberOfInliers();
cout << ", Matches: " << num_matches;
cout << ", Inliers: " << 100.0*num_inliers/num_matches << "
↪  %" << ", Current pose: " << endl;
cout << pose << endl << endl;

} else {
cout << " ... failed!" << endl;
}
```

分别从从左右镜头读入一帧, 把图片中的信息存入数组, 调用 VisualOdemetryStereo 的 process 方法进行处理, 处理完成后输出匹配成功的点, 内围点的占比和当前摄像机的姿态

2.1.2 主要类分析

VisualOdomentry 简要分析

```
class VisualOdometry {
public:
    //相机校准参数
    struct calibration {
        ...
```

```
    };

    // bucketing 参数
    struct bucketing {
      ...
    };

    // 参数
    struct parameters {
    };
protected:
    Matrix                          Tr_delta;   // transformation (previous -> current frame)
    bool                            Tr_valid;   // 是否存在预测的下一帧
    Matcher                        *matcher;    // feature matcher
    std::vector<int32_t>            inliers;    // 内围点
    double                         *J;          // 雅克比行列式
    double                         *p_observe;  // observed 2d points
    double                         *p_predict;  // predicted 2d points
    std::vector<Matcher::p_match>   p_matched;  // feature point matches

private:

    parameters                     param;      // 参数
}
```

VisualOdometryStereo 简要分析

```
class VisualOdometryStereo {
    % TODO
}
```

2.1.3 主要过程分析

```
bool VisualOdometryStereo::process (uint8_t *I1,uint8_t *I2,int32_t* dims,bool replace) {

    // push back images
    matcher->pushBack(I1,I2,dims,replace);

    // bootstrap motion estimate if invalid
    // 如果是处理第一张图片就先建立动作预测
    if (!Tr_valid) {
        matcher->matchFeatures(2);
        matcher->bucketFeatures(param.bucket.max_features,param.bucket.bucket_width,param.bucket.bucket_height);
        p_matched = matcher->getMatches();
        updateMotion();
    }

    // match features and update motion
    if (Tr_valid) matcher->matchFeatures(2,&Tr_delta);
    else          matcher->matchFeatures(2);
    matcher->bucketFeatures(param.bucket.max_features,param.bucket.bucket_width,param.bucket.bucket_height);
    p_matched = matcher->getMatches();
    return updateMotion();
}
```

Matcher 类分析

```

```

2.2 局部模块分析

2.2.1 功能类分析

2.2.2 功能函数分析

Matcher::pushBack 把左右各一帧图像放入 ringbuffer，并且计算两张图像的各个特征

```cpp
void Matcher::pushBack (uint8_t *I1,uint8_t* I2,int32_t*
↪ dims,const bool replace) {

    // 定义图片大小
    int32_t width  = dims[0];
    int32_t height = dims[1];
    int32_t bpl    = dims[2];

    // sanity check
    if (width<=0 || height<=0 || bpl<width || I1==0) {
        cerr << "ERROR: Image dimension mismatch!" << endl;
        return;
    }

    if (replace) {
        ... //如果规定了replace，释放上上张图片的各项参数
    } else {
        ... //释放上上张图片的各项参数
        ... //将“当前”图片的各项信息设为“上张”图片的各项信息
    }

    // 对齐内存
    dims_c[0] = width;
    dims_c[1] = height;
    dims_c[2] = width + 15-(width-1)%16;

```

```
    I1c = (uint8_t*)_mm_malloc(dims_c[2]*dims_c[1]*sizeof(uint8_t),16);
    I2c = (uint8_t*)_mm_malloc(dims_c[2]*dims_c[1]*sizeof(uint8_t),16);
    ... // 对齐放置图片信息

    // 计算图片各项特征
    computeFeatures(I1c,dims_c,m1c1,n1c1,m1c2,n1c2,I1c_du,I1c_dv,I1c_du_full,I1c_dv_full);
    if (I2!=0)
        computeFeatures(I2c,dims_c,m2c1,n2c1,m2c2,n2c2,I2c_du,I2c_dv,I2c_du_full,I2c_dv_full);
    }
```

Matcher::computeFeatures

```
void Matcher::computeFeatures (...) {

    ...

    if (!param.half_resolution) {
        ... // demo不涉及这个部分
    } else {
        uint8_t* I_matching = createHalfResolutionImage(I,dims);
        getHalfResolutionDimensions(dims,dims_matching);
        // 将图像缩小一半
        ... // 为各个滤波结果分配空间
        filter::sobel5x5(I_matching,I_du,I_dv,dims_matching[2],dims_matching[1]);
        // 对缩略图进行sobel滤波求梯度
        filter::sobel5x5(I,I_du_full,I_dv_full,dims[2],dims[1]);
        // 对原图图进行sobel滤波求梯度
```

```
        filter::blob5x5(I_matching,I_f1,dims_matching[2],dims_matching[1]);
        // 对缩略图使用blob滤波
        filter::checkerboard5x5(I_matching,I_f2,dims_matching[2],dims_matching[1]);
        // 对缩略图求边界
        _mm_free(I_matching);
    }

    // 使用非极大抑制提取稀疏极大值
    vector<Matcher::maximum> maxima1;
    if (param.multi_stage) {
        int32_t nms_n_sparse = param.nms_n*3;
        if (nms_n_sparse>10)
            nms_n_sparse = max(param.nms_n,10);
        nonMaximumSuppression(I_f1,I_f2,dims_matching,maxima1,nms_n_sparse);
        computeDescriptors(I_du,I_dv,dims_matching[2],maxima1);
    }

    // 使用非极大抑制提取稠密极大值
    vector<Matcher::maximum> maxima2;
    nonMaximumSuppression(I_f1,I_f2,dims_matching,maxima2,param.nms_n);
    computeDescriptors(I_du,I_dv,dims_matching[2],maxima2);

    ...

    if (num1!=0) {
        ... // 将稀疏最大值对齐内存存入返回变量
    }

```

```
    if (num2!=0) {
        ... // 将稠密最大值对齐内存存入返回变量
    }
}
```

Matcher::matchFeatures 匹配特征值

```
void Matcher::matchFeatures(int32_t method, Matrix *Tr_delta) {

    ... // 检查完整性

    // double pass matching
    if (param.multi_stage) {

        // 1st pass (sparse matches)
        matching(m1p1,m2p1,m1c1,m2c1,n1p1,n2p1,n1c1,n2c1,p_matched_1,method,false,Tr_delta);
        // 使用2d Delaunay三角剖分消除异常值
        removeOutliers(p_matched_1,method);

        // compute search range prior statistics (used for speeding up 2nd pass)
        // 为加速第二次处理计算搜索范围优先级数据
        computePriorStatistics(p_matched_1,method);

        // 2nd pass (dense matches)
        matching(m1p2,m2p2,m1c2,m2c2,n1p2,n2p2,n1c2,n2c2,p_matched_2,method,true,Tr_delta);
        if (param.refinement>0)
            // 通过抛物线拟合的子像素细化来进一步改进特征定位
            refinement(p_matched_2,method);
        removeOutliers(p_matched_2,method);

```

```
    // single pass matching
    } else {
        matching(m1p2,m2p2,m1c2,m2c2,n1p2,n2p2,n1c2,n2c2,p_matched_2,method,false,Tr_delta);
        if (param.refinement>0)
        refinement(p_matched_2,method);
        removeOutliers(p_matched_2,method);
    }
}
```

Matcher::matching

```
void Matcher::matching (int32_t *m1p,int32_t *m2p,int32_t *m1c,int32_t *m2c,
                        int32_t n1p,int32_t n2p,int32_t n1c,int32_t n2c,
                        vector<Matcher::p_match> &p_matched,int32_t method,bool use_prior,Matrix *Tr_delta) {



    // loop variables
    int32_t* M = (int32_t*)calloc(dims_c[0]*dims_c[1],sizeof(int32_t));
    int32_t i1p,i2p,i1c,i2c,i1c2,i1p2;
    int32_t u1p,v1p,u2p,v2p,u1c,v1c,u2c,v2c;

    double t00,t01,t02,t03,t10,t11,t12,t13,t20,t21,t22,t23;
    if (Tr_delta) {
        ... // 如果存在之前的姿态变化量，就初始化变量
    }

    ///////////////////////////////////////////////////////
```

```
    // method: flow
    if (method==0) {
        ... // demo不涉及这个部分
    }

    //////////////////////////////////////////////////////
    // method: stereo
    } else if (method==1) {
        ... // demo不涉及这个部分
    }

    //////////////////////////////////////////////////////
    // method: quad matching
    } else {

        // create position/class bin index vectors
        createIndexVector(m1p,n1p,k1p,u_bin_num,v_bin_num);
        createIndexVector(m2p,n2p,k2p,u_bin_num,v_bin_num);
        createIndexVector(m1c,n1c,k1c,u_bin_num,v_bin_num);
        createIndexVector(m2c,n2c,k2c,u_bin_num,v_bin_num);

        // for all points do
        for (i1p=0; i1p<n1p; i1p++) {
            // 对所有“前一帧”左边视图的所有特征点执行下面的操作

            // 读取每个特征点的坐标
            u1p = *(m1p+step_size*i1p+0);
            v1p = *(m1p+step_size*i1p+1);

            // compute row and column of statistics bin to
            ↪  which this observation belongs
            int32_t u_bin = min((int32_t)floor((float)u1p/(fl⌋
            ↪  oat)param.match_binsize),u_bin_num-1);
```

```
int32_t v_bin = min((int32_t)floor((float)v1p/(float)param.match_binsize),v_bin_num-1);
int32_t stat_bin = v_bin*u_bin_num+u_bin;

// match in circle
// 开始环形匹配
findMatch(m1p,i1p,m2p,step_size,k2p,u_bin_num,v_bin_num,stat_bin,i2p,0,false,use_prior);
// 匹配“前一帧”左视图和“前一帧”右视图的稀疏特征

u2p = *(m2p+step_size*i2p+0);
v2p = *(m2p+step_size*i2p+1);

if (Tr_delta) {

    double d = max((double)u1p-(double)u2p,1.0);
    double x1p = ((double)u1p-param.cu)*param.base/d;
    double y1p = ((double)v1p-param.cv)*param.base/d;
    double z1p = param.f*param.base/d;

    double x2c = t00*x1p + t01*y1p + t02*z1p + t03 - param.base;
    double y2c = t10*x1p + t11*y1p + t12*z1p + t13;
    double z2c = t20*x1p + t21*y1p + t22*z1p + t23;

    double u2c_ = param.f*x2c/z2c+param.cu;
    double v2c_ = param.f*y2c/z2c+param.cv;

```

```
        // 如果有之前运动矩阵，就沿用这个运动矩阵
        // 这里假设运动是连续的，运动的变化较小
        findMatch(m2p,i2p,m2c,step_size,k2c,u_bin_num
        ↪ ,v_bin_num,stat_bin,i2c, 1,true
        ↪ ,use_prior,u2c_,v2c_);
    } else {
        findMatch(m2p,i2p,m2c,step_size,k2c,u_bin_num
        ↪ ,v_bin_num,stat_bin,i2c, 1,true
        ↪ ,use_prior);
    }
    // 匹配“前一帧”右侧图像和“当前帧”右侧图像
    findMatch(m2c,i2c,m1c,step_size,k1c,u_bin_num,v_b
    ↪ in_num,stat_bin,i1c,
    ↪ 2,false,use_prior);
    // 匹配”当前帧“左侧图像和”当前帧“右侧图像
    if (Tr_delta)
        findMatch(m1c,i1c,m1p,step_size,k1p,u_bin_num
        ↪ ,v_bin_num,stat_bin,i1p2,3,true
        ↪ ,use_prior,u1p,v1p);
    else
        findMatch(m1c,i1c,m1p,step_size,k1p,u_bin_num
        ↪ ,v_bin_num,stat_bin,i1p2,3,true
        ↪ ,use_prior);
    // 匹配”当前帧“左侧图像和”前一帧“右侧图像
    // circle closure success?
    if (i1p2==i1p) {

        // extract coordinates
        u2c = *(m2c+step_size*i2c+0); v2c =
        ↪ *(m2c+step_size*i2c+1);
        u1c = *(m1c+step_size*i1c+0); v1c =
        ↪ *(m1c+step_size*i1c+1);

```

```
                // if disparities are positive
                if (u1p>=u2p && u1c>=u2c) {

                    // 如果匹配成功，就把匹配的结果放入p_match中
                    p_matched.push_back(Matcher::p_match(u1p,v1p,i1p,u2p,v2p,i2p,u1c,v1c,i1c,u2c,v2c,i2c));
                }
            }
        }
    }

    // free memory
    ...
}
```

```
void Matcher::bucketFeatures(int32_t max_features,float bucket_width,float bucket_height) {

    // 找到”当前帧“左侧图像u-v坐标做大的特征点
    float u_max = 0;
    float v_max = 0;
    for (vector<p_match>::iterator it = p_matched_2.begin(); it!=p_matched_2.end(); it++) {
        if (it->u1c>u_max) u_max=it->u1c;
        if (it->v1c>v_max) v_max=it->v1c;
    }

    // 分配需要的buckets
    int32_t bucket_cols = (int32_t)floor(u_max/bucket_width)+1;
    int32_t bucket_rows = (int32_t)floor(v_max/bucket_height)+1;
```

```
    vector<p_match> *buckets = new vector<p_match>[bucket_cols*bucket_rows];

    // 把每个特征点放入其位置对应的bukect中
    for (vector<p_match>::iterator it=p_matched_2.begin(); it!=p_matched_2.end(); it++) {
        int32_t u = (int32_t)floor(it->u1c/bucket_width);
        int32_t v = (int32_t)floor(it->v1c/bucket_height);
        buckets[v*bucket_cols+u].push_back(*it);
    }

    p_matched_2.clear();
    // 清空原特征点
    for (int32_t i=0; i<bucket_cols*bucket_rows; i++) {
        // 对每个bucket

        std::random_shuffle(buckets[i].begin(),buckets[i].end());

        // 每个bucket随机填入max_features个特征点
        int32_t k=0;
        for (vector<p_match>::iterator it=buckets[i].begin(); it!=buckets[i].end(); it++) {
            p_matched_2.push_back(*it);
            k++;
            if (k>=max_features)
                break;
        }
    }

    // free buckets
    delete []buckets;
}
```

```cpp
void Matcher::computePriorStatistics(vector<Matcher::p_match> &p_matched,int32_t method) {

    // 计算区域 (bin) 数量
    int32_t u_bin_num = (int32_t)ceil((float)dims_c[0]/(float)param.match_binsize);
    int32_t v_bin_num = (int32_t)ceil((float)dims_c[1]/(float)param.match_binsize);
    int32_t bin_num   = v_bin_num*u_bin_num;

    ...

    for (vector<Matcher::p_match>::iterator it=p_matched.begin(); it!=p_matched.end(); it++) {
        // 对每一组配对好的点组
        // method flow: compute position delta
        if (method==0) {
            ...
        } else if (method==1) {
            ...
        } else {
            delta_curr.val[0] = it->u2p - it->u1p;
            delta_curr.val[1] = 0;
            delta_curr.val[2] = it->u2c - it->u2p;
            delta_curr.val[3] = it->v2c - it->v2p;
            delta_curr.val[4] = it->u1c - it->u2c;
            delta_curr.val[5] = 0;
            delta_curr.val[6] = it->u1p - it->u1c;
            delta_curr.val[7] = it->v1p - it->v1c;
            // 计算这个点组的各个偏移量
        }

        // 计算哪些区域 (bin) 包含了这个点组
```

```
        int32_t u_bin_min,u_bin_max,v_bin_min,v_bin_max;

        // flow + stereo: use current left image as reference
        if (method<2) {
            ...

        // quad matching: use current previous image as reference
        } else {
            u_bin_min = min(max((int32_t)floor(it->u1p/(float)param.match_binsize)-1,0),u_bin_num-1);
            u_bin_max = min(max((int32_t)floor(it->u1p/(float)param.match_binsize)+1,0),u_bin_num-1);
            v_bin_min = min(max((int32_t)floor(it->v1p/(float)param.match_binsize)-1,0),v_bin_num-1);
            v_bin_max = min(max((int32_t)floor(it->v1p/(float)param.match_binsize)+1,0),v_bin_num-1);
        }

        // 在相关区域对应的accumulator中加入计算出的偏移量
        for (int32_t v_bin=v_bin_min; v_bin<=v_bin_max; v_bin++)
            for (int32_t u_bin=u_bin_min; u_bin<=u_bin_max; u_bin++)
                delta_accu[v_bin*u_bin_num+u_bin].push_back(delta_curr);
    }

    ranges.clear();

    // 对每个区域 (bin) 计算最大偏移量
    for (int32_t v_bin=0; v_bin<v_bin_num; v_bin++) {
        for (int32_t u_bin=0; u_bin<u_bin_num; u_bin++) {
```

```
        // 如果此区域（bin）没有相关记录，就使用默认值
        delta delta_min(-param.match_radius);
        delta delta_max(+param.match_radius);

        // 通过每个区域（bin）的accumulator中的各个偏移值计算出最大偏移值
        if (delta_accu[v_bin*u_bin_num+u_bin].size()>0) {

            // init displacements 'delta' to 'infinite'
            delta_min = delta(+1000000);
            delta_max = delta(-1000000);

            // find minimum and maximum displacements
            for (vector<Matcher::delta>::iterator it=delta_accu[v_bin*u_bin_num+u_bin].begin();
                it!=delta_accu[v_bin*u_bin_num+u_bin].end(); it++) {
                for (int32_t i=0; i<num_stages*2; i++) {
                    if (it->val[i]<delta_min.val[i]) delta_min.val[i] = it->val[i];
                    if (it->val[i]>delta_max.val[i]) delta_max.val[i] = it->val[i];
                }
            }
        }

        // 将最大偏移值进一步处理为搜索范围
        range r;
        for (int32_t i=0; i<num_stages; i++) {

            // bound minimum search range to 20x20
```

```
                float delta_u =
                ↪ delta_max.val[i*2+0]-delta_min.val[i*2+0];
                if (delta_u<20) {
                    delta_min.val[i*2+0] -=
                    ↪ ceil((20-delta_u)/2);
                    delta_max.val[i*2+0] +=
                    ↪ ceil((20-delta_u)/2);
                }
                float delta_v =
                ↪ delta_max.val[i*2+1]-delta_min.val[i*2+1];
                if (delta_v<20) {
                    delta_min.val[i*2+1] -=
                    ↪ ceil((20-delta_v)/2);
                    delta_max.val[i*2+1] +=
                    ↪ ceil((20-delta_v)/2);
                }

                // set range for this bin
                r.u_min[i] = delta_min.val[i*2+0];
                r.u_max[i] = delta_max.val[i*2+0];
                r.v_min[i] = delta_min.val[i*2+1];
                r.v_max[i] = delta_max.val[i*2+1];
            }
            ranges.push_back(r);
        }
    }

    // free bin accumulator memory
    delete []delta_accu;
}
```

```
bool VisualOdometry::updateMotion () {

  // 调用estimateMotion预测当前姿态变换
```

```
  vector<double> tr_delta = estimateMotion(p_matched);

  // on failure
  if (tr_delta.size()!=6)
    return false;

  // set transformation matrix (previous to current frame)
  Tr_delta = transformationVectorToMatrix(tr_delta);
  Tr_valid = true;

  // success
  return true;
}
```

```
vector<double> VisualOdometryStereo::estimateMotion
↪ (vector<Matcher::p_match> p_matched) {

    // compute minimum distance for RANSAC samples
    double width=0,height=0;
    for (vector<Matcher::p_match>::iterator
    ↪ it=p_matched.begin(); it!=p_matched.end(); it++) {
        if (it->u1c>width)  width  = it->u1c;
        if (it->v1c>height) height = it->v1c;
    }
    double min_dist = min(width,height)/3.0;

    // get number of matches
    int32_t N  = p_matched.size();
    if (N<6)
        return vector<double>();

    // allocate dynamic memory
    ...

```

```
// project matches of previous image into 3d
// 把之前一帧的匹配点投影在3D坐标内
/*
d代表水平视差，B代表baseline(水平基线)
d = max(u_l - u_r, 0.0001)
Z = (f×B)/d
X = (u - c_u)B/d
Y = (v - c_v)B/d
*/
for (int32_t i=0; i<N; i++) {
    double d = max(p_matched[i].u1p -
    ↪  p_matched[i].u2p,0.0001f);
    X[i] = (p_matched[i].u1p-param.calib.cu)*param.base/d;
    Y[i] = (p_matched[i].v1p-param.calib.cv)*param.base/d;
    Z[i] = param.calib.f*param.base/d;
}

// loop variables
vector<double> tr_delta;
vector<double> tr_delta_curr;
tr_delta_curr.resize(6);

// clear parameter vector
inliers.clear();

// 进行ransac_iters轮RANSAC算法匹配
for (int32_t k=0;k<param.ransac_iters;k++) {

    // 随机选择3个观测点作为初始inlier
    vector<int32_t> active = getRandomSample(N,3);

    // clear parameter vector
    for (int32_t i=0; i<6; i++)
```

```
            tr_delta_curr[i] = 0;

        // minimize reprojection errors
        VisualOdometryStereo::result result = UPDATED;
        int32_t iter=0;
        // 迭代寻找到最优的投影参数
        while (result==UPDATED) {
            result = updateParameters(p_matched,active,tr_delta_curr,1,1e-6);
            if (iter++ > 20 || result==CONVERGED)
                break; //迭代20次或者结果收敛就停止迭代
        }

        // overwrite best parameters if we have more inliers
        if (result!=FAILED) {
            vector<int32_t> inliers_curr = getInlier(p_matched,tr_delta_curr);
            // 获取符合当前模型的内点 (inlier)
            if (inliers_curr.size()>inliers.size()) {
                inliers = inliers_curr;
                tr_delta = tr_delta_curr;
                // 模型的优劣以能匹配到的inlier数量决定
                // 保留能匹配到更多inlier的模型
            }
        }
    }

    // 最后根据匹配到的inlier修正模型
    if (inliers.size()>=6) {
        int32_t iter=0;
        VisualOdometryStereo::result result = UPDATED;
        while (result==UPDATED) {
```

```
        result = updateParameters(p_matched,inliers,tr_delta,1,1e-8);
        if (iter++ > 100 || result==CONVERGED)
            break;
        }

        // not converged
        if (result!=CONVERGED)
            success = false;

    // not enough inliers
    } else {
        success = false;
    }

    ...

    // parameter estimate succeeded?
    if (success) return tr_delta;
    else         return vector<double>();
}
```

```
VisualOdometryStereo::result VisualOdometryStereo::updateParameters(vector<Matcher::p_match> &p_matched,vector<int32_t> &active,vector<double> &tr,double step_size,double eps) {

    // we need at least 3 observations
    if (active.size()<3)
        return FAILED;

    // extract observations and compute predictions
    computeObservations(p_matched,active);
    //将active点的“当前帧”坐标放入p_observation中
```

```
    computeResidualsAndJacobian(tr,active);
    // 计算残差与雅克比行列式

    // init
    Matrix A(6,6);
    Matrix B(6,1);

    // fill matrices A and B
    for (int32_t m=0; m<6; m++) {
        for (int32_t n=0; n<6; n++) {
            double a = 0;
            for (int32_t i=0; i<4*(int32_t)active.size(); i++) {
                a += J[i*6+m]*J[i*6+n];
            }
            A.val[m][n] = a;
        }
        double b = 0;
        for (int32_t i=0; i<4*(int32_t)active.size(); i++) {
            b += J[i*6+m]*(p_residual[i]);
        }
        B.val[m][0] = b;
    }

    // perform elimination
    if (B.solve(A)) { //如果模型合理
        bool converged = true;
        for (int32_t m=0; m<6; m++) {
            tr[m] += step_size*B.val[m][0];
            if (fabs(B.val[m][0])>eps)
                converged = false;
                // 如果某项参数更优
        }
```

```
            if (converged)
                return CONVERGED;
            else
                return UPDATED;
        } else {
            return FAILED;
        }
}
```

3 实验运行结果

```
vector<int32_t> VisualOdometryStereo::getInlier(vector<Matcher::p_match> &p_matched,vector<double> &tr) {

    // 把所有观测值标记为active
    vector<int32_t> active;
    for (int32_t i=0; i<(int32_t)p_matched.size(); i++)
        active.push_back(i);

    computeObservations(p_matched,active);
    computeResidualsAndJacobian(tr,active);

    vector<int32_t> inliers;
    for (int32_t i=0; i<(int32_t)p_matched.size(); i++)
        if (pow(p_observe[4*i+0]-p_predict[4*i+0],2)+pow(p_observe[4*i+1]-p_predict[4*i+1],2) +
            pow(p_observe[4*i+2]-p_predict[4*i+2],2)+pow(p_observe[4*i+3]-p_predict[4*i+3],2) < param.inlier_threshold*param.inlier_threshold)
        inliers.push_back(i);
```

```
        // 如果某个点观测值与预测值的偏差小于阈值，就将其标记为inlier
    return inliers;
}
```

```
Matrix VisualOdometry::transformationVectorToMatrix (vector<double> tr) {

    ...

    // 计算姿态变换矩阵
    Matrix Tr(4,4);
    Tr.val[0][0] = +cy*cz;          Tr.val[0][1] = -cy*sz;          Tr.val[0][2] = +sy;    Tr.val[0][3] = tx;
    Tr.val[1][0] = +sx*sy*cz+cx*sz; Tr.val[1][1] = -sx*sy*sz+cx*cz; Tr.val[1][2] = -sx*cy; Tr.val[1][3] = ty;
    Tr.val[2][0] = -cx*sy*cz+sx*sz; Tr.val[2][1] = +cx*sy*sz+sx*cz; Tr.val[2][2] = +cx*cy; Tr.val[2][3] = tz;
    Tr.val[3][0] = 0;               Tr.val[3][1] = 0;               Tr.val[3][2] = 0;      Tr.val[3][3] = 1;
    return Tr;
}
```