Using the Levenberg Marquardt Algorithm for Camera Calibration without the Analytical Jacobian

Bharath Kumar Comandur
Robot Vision Laboratory, Purdue, West Lafayette, IN, USA
bcomandu@purdue.edu

1 Introduction

This is a brief tutorial on refining the Camera Calibration parameters using the Levenberg Marquardt (LM) algorithm in C++. For readers who are new to LM and camera calibration, links to useful sites and documents are provided below.

Wikipedia

For a more elaborate discussion of the LM method, the following references may be useful

- Appendix 6 in the Hartley & Zisserman's Book
- Section 9.4 in the Chong & Zak's Book (the official textbook for ECE580)
- K.Madsen et al, "Methods for non-linear least squares problems", IMM Lecture Note 2004, available on the web here

Pradit's memo on implementing camera calibration in MATLAB is a very good way to review the key concepts. As the memo describes, one key issue in implementing an LM optimization is finding an analytical expression for the Jacobian which can be quite complex for the case of camera calibration. For C/C++ users, especially, it involves using MATLAB to find the Jacobian and then using the MATLAB function *ccode* to get the corresponding C code which is cumbersome.

There is an alternative and less complex method that can be used with good results for normal camera calibration and the Homeworks. We use the **levmar** package for this. It can be downloaded **here** Note that you can use levmar from Matlab and Python as well. To make maximum use of the various functions in levmar, you need to install the LAPACK or CLAPACK libraries. Instructions to install and use the levmar package in Windows can be found here

The levmar package offers the option to implement the LM algorithm without evaluating the analytic expression for the Jacobian. As mentioned in Levmar FAQ levmar includes routines that approximate the Jacobian using a combination of finite differences and secant updates. If it is easy to evaluate the analytical expression for the Jacobian, then it is advisable to do so as the algorithm converges more quickly. However in cases where the Jacobian is very expensive to compute, better performance can be achieved by using the finite difference approximation.

The rest of this tutorial is set out as follows. In Section 2, a simple example on how to use the levmar package is presented. Section 3 describes using the LM algorithm to refine the camera calibration parameters. In Section 4, the method to use the levmar package for Camera Calibration is discussed.

2 Simple Curve Fitting Example

We consider a very simple case of curve fitting to an exponential function. Note that this example has been chosen, because it is one of the examples that is provided along with the levmar package to understand how to use LM optimization. Refer expfit.c in the levmar package. Assume that we have data samples of a theoretical curve $y = f(t)$ given as

$$y = p_0 e^{(-p_1 * t)} + p_2$$

We need to determine p_0,$p_1$and p_2. Obviously the data samples we have are corrupted either by noise or poor measurement equipment. Hence we can only estimate the parameters depending upon the level of accuracy needed for the application. For this, there exist a variety of linear and non-linear optimization tools. Dr. Avinash Kak's scroll is a very good read to understand the common concepts and differences between some of the popular optimization algorithms.

For the specific case of Levenberg Marquardt, we require the following, an initial estimate of the parameters, the set of data samples $y_{measured}$ and t .

2.1 Syntax of function

The levmar package includes a function called $dlevmar\_dif$ and a function called $dlevmar\_der$. Use $dlevmar\_der$ if you want to use the analytical Jacobian. Else use the $dlevmar\_dif$ function to approximate the Jacobian. The $dlevmar\_dif$ function has the following syntax :-

/* * Returns the number of iterations (>=0) if successful, -1 if failed * */

int dlevmar_dif(void (*func)(double *p, double *hx, int m, int n, void *adata),double *p, double *x, int m,int n,int itmax, double opts[5],double info[LM_INFO_SZ],double *work,double *covar,void *adata**)**

where

void (*func) /*used to call the specific function to be optimized*/

double *p, /* initial parameter estimates. On output contains estimated solution */

double *x, /* I: measurement vector. NULL implies a zero vector */

int m, /* I: parameter vector dimension (i.e. #unknowns) */

int n, /* I: measurement vector dimension */

int itmax, /* I: maximum number of iterations */

double opts[5], /* minimization options. Refer levmar documentation for details */

double info[LM_INFO_SZ], /*information regarding the minimization */

double *work, /* I: working memory */

double *covar, /* O: Covariance matrix corresponding to LS solution*/

void *adata) /* I: pointer to possibly needed additional data, passed uninterpreted to func */

2.2 Example using Exponential function

Going back to the example of curve fitting to the function

$$y = 5e^{(-0.1*t)} + 1$$

We first write the function to be passed as an argument to *dlevmar_dif*. In this particular example, we have a simple parametric form for y in terms of t and the parameters p_i .

```
void expfunc(double *p, double *x, int m, int n, void *data)
{
int i;
        for(i=0; i<n; ++i)
        {
                x[i]=p[0]*exp(-p[1]*i) + p[2];
        }
}
```

A simple code to call *dlevmar_dif* is shown below. We take 40 measurements to estimate 3 parameters. For this simple example, we simulate noisy measurements by adding noise generated using a noise function.

```
double noise(mean,variance) /*function to generate noise */
int main()
{
        int n=40, m=3; // 40 measurements, 3 parameters
        double p[m], x[n], opts[LM_OPTS_SZ], info[LM_INFO_SZ];
        int i; int ret;
        for(i=0; i<n; ++i)
        {
                /*generate measurement data*/
                x[i]=(5.0*exp(-0.1*i) + 1.0) + noise(0.0, 0.1);
        }
        /* initial parameters estimate: (1.0, 0.0, 0.0) */
        p[0]=1.0; p[1]=0.0; p[2]=0.0;
        /*optimization control parameters*/
        opts[0]=LM_INIT_MU; opts[1]=1E-15;
        opts[2]=1E-15; opts[3]=1E-20;
        opts[4]=LM_DIFF_DELTA; //for finite difference Jacobian
        /* invoke the optimization function */
        ret=dlevmar_dif(expfunc, p, x, m, n, 1000, opts, info,
           NULL, NULL, NULL); // without Jacobian
        /*ret=dlevmar_der(expfunc, jacexpfunc, p, x, m, n, 1000,
            opts, info, NULL, NULL, NULL); with Jacobian */
}
```

The refined parameters are stored in p. On printing the results we get $p_0 = 4.999$, $p_1 = 0.1$ and $p_2 = 0.9999$.

Note the jacexpfunc used in the call to *dlevmar_der* . If the Jacobian is fairly easy to calculate, then jacexpfunc should be defined like expfunc , according to the analytical form of the Jacobian.

3 Camera Calibration

We now consider Zhengyou Zhang's camera calibration procedure (here) . The reader is advised to go through Zhang's report which is quite comprehensive.

Equation 10 of Zhang's report is

$$\sum\sum \|\mathbf{x_j} - \hat{\mathbf{x}}_\mathbf{j}(\mathbf{K}, \mathbf{R}, \mathbf{t}, \mathbf{x_{M,j}})\|^\mathbf{2}$$

where

$\mathbf{x_j}$ is the j^{th} point in the image captured from the camera

$\mathbf{x_{M,j}}$ is the j^{th} point in the model plane

$\hat{\mathbf{x}}_\mathbf{j}$is the projection of the j^{th} model point into the camera image.

$\mathbf{K}$ is the intrinsic camera parameter.

$\mathbf{\{R,\ t\}}$ are the rotation and translation of the camera position that is used to define the extrinsic parameters.

We use nearly 20 images in the camera calibration procedure. It is important to note that the summation in the above equation should be done for all the points in each image.

It is also important to understand the difference between constrained and unconstrained optimization. The $\mathbf{R}$ matrix is a rotation matrix. Therefore it has only 3 DOF. Hence unconstrained optimization assuming the elements of $\mathbf{R}$ are independent will yield a refined $\mathbf{R}$ matrix that does not make any physical sense. Hence it is necessary to use the Rodrigues formula to represent $\mathbf{R}$ as a 3-vector $\mathbf{w}$. (Refer Wikipedia and Zhang's report)

The Rodrigues form represents a rotation by a vector whose direction stands for the axis of rotation and whose magnitude is the angle through which the object is rotated clockwise around the axis. Let the vector $\mathbf{w} = \begin{bmatrix} w_x & w_y & w_z \end{bmatrix}^\mathbf{T}$ represent an object rotation. The axis of rotation is given by

$$\frac{\mathbf{w}}{\|\mathbf{w}\|}$$

and the angle

$$\phi = \|\mathbf{w}\|$$

Convert the vector $\mathbf{w}$ into its 3 x 3 matrix form

$$[\mathbf{w}]_\mathbf{X} = \begin{bmatrix} 0 & -\mathbf{w_z} & \mathbf{w_y} \\ \mathbf{w_z} & 0 & -\mathbf{w_x} \\ -\mathbf{w_y} & \mathbf{w_x} & 0 \end{bmatrix}$$

$$\mathbf{R} = \mathbf{e}^{[\mathbf{w}]_\mathbf{X}} = \mathbf{1} + \frac{\mathbf{sin}(||\mathbf{w}||)}{||\mathbf{w}||}[\mathbf{w}]_\mathbf{X} + \frac{(\mathbf{1} - \mathbf{cos}(||\mathbf{w}||))}{||\mathbf{w}||^\mathbf{2}}([\mathbf{w}]_\mathbf{X})^2$$

Given $\mathbf{R}$ as

$$\mathbf{R} = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix}$$

w is given as

$$||\mathbf{w}|| = \mathbf{cos^{-1}}(\frac{\mathbf{trace(R) - 1}}{\mathbf{2}}) \ , \ \ \frac{\mathbf{w}}{||\mathbf{w}||} = \frac{1}{2sin(||\mathbf{w}||)} \begin{pmatrix} \mathbf{r_{32}} & \mathbf{-r_{23}} \\ \mathbf{r_{13}} & \mathbf{-r_{31}} \\ \mathbf{r_{21}} & \mathbf{-r_{12}} \end{pmatrix}$$

4 Applying LM for Camera Calibration in C++

We now apply LM optimization to refine the initial estimates of the camera parameters. There are a few small differences between the previous example of an exponential function and our current Camera Calibration procedure. For one, we do not have a simple parametric equation to compute $\hat{\mathbf{x}}_\mathbf{j}$.

To minimize the expression in Equation 10 of Zhang's report, we need to compute $\hat{\mathbf{x}}_\mathbf{j}$ for each model point, for each of the 20 different extrinsic locations of the camera, a.k.a 20 different images. For this example, we use 80 points from each image. Thus in total we have 1600 points.

In the code snippet below, **para** is a pointer to an array of the initial estimates of the Camera parameters. The first five elements are the elements of the **K** matrix. The sixth and seventh elements are the radial distortion parameters. The remaining elements are the extrinsic camera parameters estimated for each of the 20 different images, stacked into one array. This is done in the **Initialize** function. The **adata** pointer is used to pass the coordinates of the points in the model plane $\mathbf{x_M}$ as additional data to the optimization function. **tran_x** is the pointer to the array of the projected $\hat{\mathbf{x}}_\mathbf{j}$ for each of the 1600 points. The function Camera given below is the function to be passed as an argument to *dlevmar_dif* .

```
static void Camera(double *para, double *tran_x, int m, int n,
   void *adata)
{
        int i;
        Mat *pair;
        pair = (Mat *)adata; /*points in the model plane*/
        for(i=0; i<80*20; i++) /*iterate over all points from 20
            images*/
        {
                Mat point(3,1,CV_64FC1,Scalar(1));
                //pair is a 3x80 matrix storing the model points
```

```
                point.at<double>(0,0)=pair->at<double>(0,i%80);
                point.at<double>(1,0)=pair->at<double>(1,i%80);
                /*call function to compute projected points*/
                FunctionCam(point,para, tran_x+i*2, i,1);
        }
}
```

The function Camera iteratively calls the function FunctionCam . Given the estimates of the Camera matrix and a point on the Model plane, FunctionCam computes the value of the corresponding projected coordinates, $\hat{\mathbf{x}}_\mathbf{j}$. FunctionCam is given below. It is a fairly simple piece of code, if the reader understands Zhang's Calibration procedure.

```
void FunctionCam(Mat X, double *para, double newX[2], int id,
   int Rad = 1)
{
        int idx=id/80; //image number
        /*intrinsic K*/
        double alpha = para[0], beta = para[1];
        double gamma = para[2];
        double u0 = para[3], v0 = para[4];
        /*radial distortion parameters*/
        double k1 = para[5], k2 = para[6];
        /*extrinsic parameters in Rodrigues representation*/
        double w_x = para[7+6*idx], w_y = para[7+6*idx+1];
        double w_z = para[7+6*idx+2];
        double tx = para[7+6*idx+3], ty = para[7+6*idx+4];
        double tz = para[7+6*idx+5];
        Mat R(3, 3, CV_64FC1,Scalar(0));
        Mat ptx(3, 1, CV_64FC1);
        double x,y,u,v,value;
        /*RodtoR - function to convert from Rodrigues to R
           matrix*/
        R=RodtoR(w_x,w_y,w_z);
        /* In the model plane, Z coordinate is 0. Hence third
           column is t*/
        R.at<double>(0,2)=tx;
        R.at<double>(1,2)=ty;
        R.at<double>(2,2)=tz;
        /*project point using R matrix*/
        ptx=R*X;
        x = ptx.at<double>(0,0)/ptx.at<double>(2,0);
        y = ptx.at<double>(1,0)/ptx.at<double>(2,0);
```

```
        /*project point onto image plane using elements of K*/
        u = u0 + alpha*x + gamma*y;
        v = v0 + beta*y;
        /* if flag is set for Radial distortion*/
        if(Rad)
        {
                value = pow(x,2.0)+pow(y,2.0);
                newX[0] = u+(u-u0)*(k1*value+k2*pow(value,2.0));
                newX[1] = v+(v-v0)*(k1*value+k2*pow(value,2.0));
        }
        else
        {
                newX[0] = u;
                newX[1] = v;
        }
}
```

Thus the function Camera computes $\hat{\mathbf{x}}_\mathbf{j}$ for each point. The call to *dlevmar_dif* is now made as follows.

```
        /*function as argument for dlevmar_dif*/
        void (*minfn)(double *p, double *hx, int m, int n, void
           *adata);
        double opts[LM_OPTS_SZ], info[LM_INFO_SZ];
        /*set optimization parameters*/
        opts[0]=LM_INIT_MU; opts[1]=1E-12; opts[2]=1E-12;
        opts[3]=1E-15;
        opts[4]=LM_DIFF_DELTA;
        int LM_m = (5+2+filen*6)//number of parameters
        int LM_n = 2*filen*80;//number of projected points
        double *ptx = (double *)malloc(LM_n*sizeof(double));
        double *para = (double*)malloc(LM_m*sizeof(double));
        int ind=0,jin=0;
        int npt=80; //number of model points
        Initialize(r, K, para, filen, k, rodr);//initialize para
        //all the detected corners from all images
        for(ind=0;ind<filen;ind++)
        {
        for(jin=0;jin<npt;jin++)
        {
                /*Model has 80 points in model plane*/
                /*stack repetition of the 80 points in ptx for
```

```
                each of the 20 images*/
        ptx[2*npt*ind+2*jin]=Model[ind].at<double>(0,jin);
        ptx[2*npt*ind+2*jin+1]=Model[ind].at<double>(1,jin);
    }
    }
    /*Camera is passed as argument to dlevmar_dif*/
    minfn = Camera;
    /*call dlevmar_dif*/ /*Pattern is a pointer to points in
        the model plane*/
    valuelm = dlevmar_dif(minfn, para, ptx, LM_m, LM_n, 150,
        opts, info, NULL, NULL,&Pattern); // no Jacobian
```

info[0] gives the distortion before LM refinement and **info[1]** gives the distortion after LM refinement. The number of iterations taken to converge is stored in **info[5]**. Refer levmar documentation for more details.

The refined parameters are stored in **para**. The projected points are stored in **ptx.** Thus it is possible to implement camera calibration without determining a complex analytical expression for the Jacobian.