Ceres Solver

Convenient Fast Non-linear Optimzation

Mikael Persson

Department of Electrical Engineering
Computer Vision Laboratory
Linköping University

May 18, 2015

Introduction

Non-linear optimization library

- Convenient - good api, automatic symbolic differentiation
- Fast, in particular if used with Suitesparse and the correct blas
- Powerful/semi generic
- Well documented and straightforward installation in Linux.

```
// the mutable variable
double* x={1, 2, 3, 4}; //, initial values

ceres::Problem problem; // problem container
ceres::Solver::Options options; // solver configuration
ceres::LossFunction* loss=nullptr;  // iid gauss
ceres::Solver::Summary summary;

// build the problem/ cost
for(auto data:datas)
    problem.AddResidualBlock( \\
    Error::Create(data),Loss,x));

ceres::Solve(options, &problem, &summary);

cout<<summary.FullReport()<<endl;
```

Lossfunctions

Several types of loss functions are available.

- What noise is assumed
- Consider the usecases
- Influence on convergence?

Most twice derivable functions with a continuous derivative are easy to implement.

Solver Options

The most useful options are:

- Solver type
- convergence/exit criteria

Parameter Block Order:

- Sparsity and elimination order information
- Guessed if not specified
- Significant performance gains possible

cvl & mlib

CVL has a linear algebra lib

- Developed by Hedborg
- Similar to eigen
- Vector2,3,4
- Matrix3x3 to Matrix 4x4
- Common operations available

mlib extras

- quaternions
- poses(rigid transforms)
- dynamic states(cv,ca,cea, osv)
- uniformly sampled random rotations

cvl & mlib

CVL has a linear algebra lib

Why?

- Simpler to modify cvl than eigen
- $operator * (X < double >, Y < ceres :: jet >)$
- ceres::jet is the ceres differentiation wrapper

Triangulation

Model

- Let x be a $3D$ point feature.
- Let $\wp : \wp(x) = \frac{1}{x_2}\binom{x_0}{x_1}$.
- Let P_t transform from world to camera at time t.
- Let y_t be a pinhole normalized measurement of x at time t.
- Let e_t be IID Gaussian noise.
- Measurement model: $y_t = \wp(P_t x) + e_t$.

Minimize: $\sum(y_t - \wp(P_t x))^2$ over x.

```
class TrigError{
public:
  Matrix4x4<double> P;
  Vector2<double> yn;

  TrigError(Matrix4x4 P_,Vector2 yn_){P=P_; yn=yn_;}

  template<class T>
  bool operator()(const Vector3<T>* const x,
                  T* residuals) const {

    Vector3<T> xr=P*x;

    T xp=(xr[0] / xr[2]);
    T yp=(xr[1] / xr[2]);

    residuals[0] = xp - T(yn.x); // implicit squaring
    residuals[1] = yp - T(yn.y);
    return true; }

  static ceres::CostFunction*
  Create(Matrix4x4<double> P, Vector2<double> yn);};
```

```
// Cost Factory
    static ceres::CostFunction*
    TrigError::Create(Matrix4x4<double> P, Vector2<
        double> yn) {
        // residuals,parameter count
        return (new ceres::AutoDiffCostFunction<
            TrigError, 2, 3>(
                  new TrigError(P,yn)));
    }
```

```
Vector3 triangulate( vector<pair<Matrix4x4,Vector2>>
    datas){

 Vector3 x(0,0,1);
 ceres::Problem problem; // problem container
 ceres::Solver::Options options; // solver
     configuration
 ceres::LossFunction* loss=nullptr;  // iid gauss
 ceres::Solver::Summary summary;

 for(auto data:datas)
   problem.AddResidualBlock( \\
   TrigError::Create(data.first ,data.second),
   Loss,&x[0]));

 ceres::Solve(options, &problem, &summary);
 return x;
}
```

Bundle Adjustment

Model

- Let x be a $3D$ point feature.
- Let $\wp$: $\wp(x) = \frac{1}{x_2}\binom{x_0}{x_1}$.
- Let P_t transform from world to camera at time t.
- Let y_t be a pinhole normalized measurement of x at time t.
- Let e_t be IID Gaussian noise.
- Measurement model: $y_t = \wp(P_t x) + e_t$.

Minimize: $\sum(y_t - \wp(P_t x))^2$ over x and P_t.

```
class ReError{
public:
  Vector2<double> yn;

  ReError(Vector2 yn_){ yn=yn_;}

  template<class T>
  bool operator()(const Vector3<T>* const x,
      const Vector4<T>* const q,
      const Vector3<T>* const t,
                          T* residuals) const {

    Vector3<T> xr=quaternion_rotate(*q,*x) + *t;

    T xp=(xr[0] / xr[2]);
    T yp=(xr[1] / xr[2]);

    residuals[0] = xp - T(yn.x); // implicit squaring
    residuals[1] = yp - T(yn.y);
    return true; }
```

```
class Obs{vector2 yn; Vector3* x;Pose* p;};

void ba( vector<Obs> datas){
...

 for(auto data:datas)
    problem.AddResidualBlock( \\
    ReError::Create(data),
    Loss,&(data->p.q),&(data->p.t),&x[0]));

    // equivalent to mlib::unit<4> parametrization
 ceres::LocalParameterization* qp = new ceres::
      QuaternionParameterization;
  for(auto data:datas)
      problem.SetParameterization(&data->p.q, qp);

 ceres::Solve(options, &problem, &summary);

}
```

- This is how to get you started.
- There are excellent guides available online
- Great performance gains by manual specification of sparsity and elimination order. Dont try that first though!
- Questions?

Linköping University

expanding reality