CERES SOLVER: TUTORIAL & REFERENCE

SAMEER AGARWAL
sameeragarwal@google.com

KEIR MIERLE
keir@google.com

May 14, 2012

A NOTE ABOUT THIS DOCUMENT

Building this pdf from source requires a relatively recent installation of `LaTeX` [1], `minted.sty`[2] and `pygments`[3].

This document is incomplete and we are working to fix this. In the meanwhile please refer to the source code.

[1]http://www.tug.org/texlive/
[2]http://code.google.com/p/minted/
[3]http://pygments.org/

INTRODUCTION

Ceres Solver[1] is a non-linear least squares solver developed at Google. It is designed to solve small and large sparse problems accurately and efficiently [2]. Amongst its various features is a simple but expressive API with support for automatic differentiation, robust norms, local parameterizations, automatic gradient checking, multithreading and automatic problem structure detection.

The key computational cost when solving a non-linear least squares problem is the solution of a linear least squares problem in each iteration. To this end Ceres supports a number of different linear solvers suited for different needs. This includes dense QR factorization (using `Eigen3`) for small scale problems, sparse Cholesky factorization (using `CHOLMOD`) for general sparse problems and specialized Schur complement based solvers for problems that arise in multi-view geometry [5].

Ceres has been used for solving a variety of problems in computer vision and machine learning at Google with sizes that range from a tens of variables and objective functions with a few hundred terms to problems with millions of variables and objective functions with tens of millions of terms.

2.1 WHAT'S IN A NAME?

While there is some debate as to who invented of the method of Least Squares [16]. There is no debate that it was Carl Friedrich Gauss's prediction of the orbit of the newly discovered asteroid Ceres based on just 41 days of observations that brought it to the attention of the world [17]. We named our solver after Ceres to celebrate this seminal event in the history of astronomy, statistics and optimization.

2.2 CONTRIBUTING TO CERES SOLVER

We welcome contributions to Ceres, whether they are new features, bug fixes or tests. If you have ideas on how you would like to contribute to Ceres, please join the Ceres mailing list (`ceres-solver@googlegroups.com`) or if you are looking for ideas, please let us know about your interest and skills and we will be happy to make a suggestion or three.

We follow Google's C++ Style Guide [3].

[1]For brevity, in the rest of this document we will just use the term Ceres.

[2]For a gentle but brief introduction to non-liner least squares problems, please start by reading the Tutorial

[3]`http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml`

2.3 CITING CERES SOLVER

If you use Ceres for an academic publication, please cite this manual. e.g.,

```
@manual{ceres-manual,
Author = {Sameer Agarwal and Keir Mierle},
Title = {Ceres Solver: Tutorial \& Reference},
Organization = {Google Inc.}
}
```

2.4 ACKNOWLEDGEMENTS

A number of people have helped with the development and open sourcing of Ceres.

Fredrik Schaffalitzky when he was at Google started the development of Ceres, and even though much has changed since then, many of the ideas from his original design are still present in the current code.

Amongst Ceres' users at Google two deserve special mention: William Rucklidge and James Roseborough. William was the first user of Ceres. He bravely took on the task of porting production code to an as-yet unproven optimization library, reporting bugs and helping fix them along the way. James is perhaps the most sophisticated user of Ceres at Google. He has reported and fixed bugs and helped evolve the API for the better.

Nathan Wiegand contributed the MacOS port.

2.5 LICENSE

Ceres Solver is licensed under the New BSD license, whose terms are as follows.

BUILDING CERES

Ceres source code and documentation is hosted at `http://code.google.com/p/ceres-solver/`.

3.1 DEPENDENCIES

Ceres relies on a number of open source libraries, some of which are optional. However, we recommend that you start out by building Ceres with all its dependencies. For details on customizing the build process, please see Section 3.4.

1. `cmake` [1] is the cross-platform build system used by Ceres.
2. `Eigen3` [2] is used for doing all the low level matrix and linear algebra operations.
3. `google-glog` [3] is used for error checking and logging.

   Note: Ceres requires `glog` version 0.3.1 or later. Version 0.3 (which ships with Fedora 16) has a namespace bug which prevents Ceres from building.
4. `gflags` [4] is used by the code in `examples`. It is also used by some of the tests. Strictly speaking it is not required to build the core library, we do not recommend building Ceres without it.
5. `SuiteSparse` [5] is used for sparse matrix analysis, ordering and factorization. In particular Ceres uses the `AMD`, `COLAMD` and `CHOLMOD` libraries. This is an optional dependency.
6. `BLAS` and `LAPACK` are needed by `SuiteSparse`. We recommend either `GotoBlas2` [6] or `ATLAS` [7], both of which ship with `BLAS` and `LAPACK` routines.
7. `protobuf` [8] is an optional dependency that is used for serializing and deserializing linear least squares problems to disk. This is useful for debugging and testing. Without it, some of the tests will be disabled.

[1] `http://www.cmake.org/`
[2] `http://eigen.tuxfamily.org`
[3] `http://code.google.com/p/google-glog`
[4] `http://code.google.com/p/gflags`
[5] `http://www.cise.ufl.edu/research/sparse/suitesparse/`
[6] `http://www.tacc.utexas.edu/tacc-projects/gotoblas2`
[7] `http://math-atlas.sourceforge.net/`
[8] `http://code.google.com/p/protobuf/`

Currently we support building on Linux and MacOS X. Support for other platforms is forthcoming.

3.2 BUILDING ON LINUX

We will use Ubuntu as our example platform.

1. `cmake`

   ```
   sudo apt-get install cmake
   ```

2. `gflags` can either be installed from source via the `autoconf` invocation

   ```
   tar -xvzf gflags-2.0.tar.gz
   cd gflags-2.0
   ./configure --prefix=/usr/local
   make
   sudo make install.
   ```

   or via the `deb` or `rpm` packages available on the `gflags` website.

3. `google-glog` must be configured to use the previously installed `gflags`, rather than the stripped down version that is bundled with `google-glog`. Assuming you have it installed in `/usr/local` the following `autoconf` invocation installs it.

   ```
   tar -xvzf glog-0.3.2.tar.gz
   cd glog-0.3.2
   ./configure --with-gflags=/usr/local/
   make
   sudo make install
   ```

4. `Eigen3`

   ```
   sudo apt-get install libeigen3-dev
   ```

5. `SuiteSparse`

   ```
   sudo apt-get install libsuitesparse-dev
   ```

   This should automatically bring in the necessary `BLAS` and `LAPACK` dependencies.

6. `protobuf`

   ```
   sudo apt-get install libprotobuf-dev
   ```

We are now ready to build and test Ceres. Note that `cmake` requires the exact path to the `libglog.a` and `libgflag.a`

```
tar zxf ceres-solver-1.0.tar.gz
mkdir ceres-bin
cd ceres-bin
cmake ../ceres-solver-1.0
make -j3
make test
```

You can also try running the command line bundling application with one of the included problems, which comes from the University of Washington's BAL dataset [1]:

```
examples/simple_bundle_adjuster \
  ../ceres-solver-1.0/data/problem-16-22106-pre.txt \
```

This runs Ceres for a maximum of 10 iterations using the `DENSE_SCHUR` linear solver. The output should look something like this.

```
0: f: 1.598216e+06 d: 0.00e+00 g: 5.67e+18 h: 0.00e+00 rho: 0.00e+00 mu: 1.00e-04 li:  0
1: f: 1.116401e+05 d: 1.49e+06 g: 1.42e+18 h: 5.48e+02 rho: 9.50e-01 mu: 3.33e-05 li:  1
2: f: 4.923547e+04 d: 6.24e+04 g: 8.57e+17 h: 3.21e+02 rho: 6.79e-01 mu: 3.18e-05 li:  1
3: f: 1.884538e+04 d: 3.04e+04 g: 1.45e+17 h: 1.25e+02 rho: 9.81e-01 mu: 1.06e-05 li:  1
4: f: 1.807384e+04 d: 7.72e+02 g: 3.88e+16 h: 6.23e+01 rho: 9.57e-01 mu: 3.53e-06 li:  1
5: f: 1.803397e+04 d: 3.99e+01 g: 1.35e+15 h: 1.16e+01 rho: 9.99e-01 mu: 1.18e-06 li:  1
6: f: 1.803390e+04 d: 6.16e-02 g: 6.69e+12 h: 7.31e-01 rho: 1.00e+00 mu: 3.93e-07 li:  1

Ceres Solver Report
-------------------
                                     Original                  Reduced
Parameter blocks                        22122                    22122
Parameters                              66462                    66462
Residual blocks                         83718                    83718
Residual                               167436                   167436

                                        Given                     Used
Linear solver                     DENSE_SCHUR              DENSE_SCHUR
Preconditioner                            N/A                      N/A
Ordering                                SCHUR                    SCHUR
num_eliminate_blocks                      N/A                    22106
Threads:                                    1                        1
Linear Solver Threads:                      1                        1

Cost:
Initial                          1.598216e+06
Final                            1.803390e+04
Change                           1.580182e+06

Number of iterations:
Successful                                  6
Unsuccessful                                0
Total                                       6

Time (in seconds):
Preprocessor                     0.000000e+00
Minimizer                        2.000000e+00
Total                            2.000000e+00
Termination:                 FUNCTION_TOLERANCE
```

3.3 BUILDING ON OS X

On OS X, we recommend using the `homebrew` [9] package manager.

1. `cmake`

   ```
   brew install cmake
   ```

2. `glog` and `gflags`

   Installing `google-glog` takes also brings in `gflags` as a dependency.

   ```
   brew install glog
   ```

3. `Eigen3`

   ```
   brew install eigen
   ```

4. `SuiteSparse`

   ```
   brew install suite-sparse
   ```

5. `protobuf`

   ```
   brew install protobuf
   ```

We are now ready to build and test Ceres.

```
tar zxf ceres-solver-1.0.tar.gz
mkdir ceres-bin
cd ceres-bin
cmake ../ceres-solver-1.0
make -j3
make test
```

Like the Linux build, you should now be able to run `examples/simple_bundle_adjuster`.

[9] http://mxcl.github.com/homebrew/

3.4 CUSTOMIZING THE BUILD PROCESS

It is possible to reduce the libraries needed to build Ceres and customize the build process by passing appropriate flags to `cmake`. But unless you really know what you are doing, we recommend against disabling any of the following flags.

1. `protobuf`

   Protocol Buffers is a big dependency and if you do not care for the tests that depend on it and the logging support it enables, you can turn it off by using

   `-DPROTOBUF=OFF.`

2. `SuiteSparse`

   By default, Ceres will only link to SuiteSparse if all its dependencies are present. To build Ceres without `SuiteSparse` use

   `-DSUITESPARSE=OFF.`

   This will also disable dependency checking for `LAPACK` and `BLAS`. This saves on binary size, but the resulting version of Ceres is not suited to large scale problems due to the lack of a sparse Cholesky solver. This will reduce Ceres' dependencies down to `Eigen3`, `gflags` and `google-glog`.

3. `gflags`

   To build Ceres without `gflags`, use

   `-DGFLAGS=OFF.`

   Disabling this flag will prevent some of the example code from building.

4. Template Specializations

   If you are concerned about binary size/compilation time over some small (10-20%) performance gains in the `SPARSE_SCHUR` solver, you can disable some of the template specializations by using

   `-DSCHUR_SPECIALIZATIONS=OFF.`

5. `OpenMP`

   On certain platforms like Android, multithreading with OpenMP is not supported. OpenMP support can be disabled by using

   `-DOPENMP=OFF.`

TUTORIAL

4.1 NON-LINEAR LEAST SQUARES

Let $x \in \mathbb{R}^n$ be an n-dimensional vector of variables, and $F(x) = [f_1(x); \ldots; f_k(x)]$ be a vector of residuals $f_i(x)$. The function $f_i(x)$ can be a scalar or a vector valued function. We are interested in finding the solution to the following optimization problem

$$\arg\min_x \frac{1}{2} \sum_{i=1}^{k} \|f_i(x)\|^2. \tag{4.1}$$

Here $\|\cdot\|$ denotes the Euclidean norm of a vector.

Such optimization problems arise in almost every area of science and engineering. Whenever there is data to be analyzed, curves to be fitted, there is usually a linear or a non-linear least squares problem lurking in there somewhere.

Perhaps the simplest example of such a problem is the problem of Ordinary Linear Regression, where given observations $(x_1, y_1), \ldots, (x_k, y_k)$, we wish to find the line $y = mx + c$, that best explains y as a function of x. One way to solve this problem is to find the solution to the following optimization problem

$$\arg\min_{m,c} \sum_{i=1}^{k} (y_i - mx_i - c)^2. \tag{4.2}$$

With a little bit of calculus, this problem can be solved easily by hand. But what if, instead of a line we were interested in a more complicated relationship between x and y, say for example $y = e^{mx+c}$. Then the optimization problem becomes

$$\arg\min_{m,c} \sum_{i=1}^{k} \left(y_i - e^{mx_i+c}\right)^2. \tag{4.3}$$

This is a non-linear regression problem and solving it by hand is much more tedious. Ceres is designed to help you model and solve problems like this easily and efficiently.

4.2 HELLO WORLD!

Let us consider the problem of finding the minimum of the function

$$g(x) = \frac{1}{2}(10 - x)^2. \tag{4.4}$$

This is a trivial problem, whose minimum is easy to see is located at 10, but it is a good place to start to illustrate the basics of solving a problem with Ceres. Let us write this problem as a non-linear least squares problem by defining the scalar residual function $f_1(x) = 10 - x$. Then $F(x) = [f_1(x)]$ is a residual vector with exactly one component.

When solving a problem with Ceres, the first thing to do is to define a `CostFunction` object[1]. It is responsible for computing the value of the residual function and its derivative (also known as the Jacobian) with respect to x. Listing 4.2 has the code.

```
class SimpleCostFunction
  : public ceres::SizedCostFunction<1 /* number of residuals */,
                                    1 /* size of first parameter */> {
 public:
  virtual ~SimpleCostFunction() {}
  virtual bool Evaluate(double const* const* parameters,
                        double* residuals,
                        double** jacobians) const {
    double x = parameters[0][0];
    // f(x) = 10 - x.
    residuals[0] = 10 - x;
    if (jacobians != NULL) {
          // If the jacobians are asked for,
          // then compute the derivative.
      jacobians[0][0] = -1;
    }
    return true;
  }
};
```

A `CostFunction` for $f = 10 - x$

`SimpleCostFunction` is provided with an input array of parameters, an output array for residuals and an optimal output array for Jacobians. In our example, there is just one parameter and one residual and this is known at compile time, therefore we inherit from templated `SizedCostFunction` class. The `jacobians` array is optional, `Evaluate` is expected to check

[1]Full working code for this and other examples in this manual can be found in the `examples` directory. Code for this example can be found in `examples/quadratic.cc`

when it is non-null, and if it is the case then fill it with the values of the derivative of the residual function. In this case since the residual function is linear, the Jacobian is constant.

Let us now look at the construction and solution of the problem using this `CostFunction`.

```
int main(int argc, char** argv) {
  double x = 5.0;
  ceres::Problem problem;

  // The problem object takes ownership of the newly allocated
  // SimpleCostFunction and uses it to optimize the value of x.
  problem.AddResidualBlock(new SimpleCostFunction, NULL, &x);

  // Configure the solver.
  ceres::Solver::Options options;
  options.max_num_iterations = 2;
  // Small, simple problem so we will use the Dense QR
  // factorization based solver.
  options.linear_solver_type = ceres::DENSE_QR;
  options.minimizer_progress_to_stdout = true;

  ceres::Solver::Summary summary;
  ceres::Solve(options, &problem, &summary);
  std::cout << summary.BriefReport() << "\n";
  std::cout << "x : 5.0 -> " << x << "\n";
  return 0;
}
```

Problem construction and solution for $F(x) = \frac{1}{2}(x-10)^2$

Compiling and running this program gives us

```
0: f: 1.250000e+01 d: 0.00e+00 g: 5.00e+00 h: 0.00e+00 rho: 0.00e+00 mu: 1.00e-04 li:  0
1: f: 1.249750e-07 d: 1.25e+01 g: 5.00e-04 h: 5.00e+00 rho: 1.00e+00 mu: 3.33e-05 li:  1
2: f: 1.388518e-16 d: 1.25e-07 g: 1.67e-08 h: 5.00e-04 rho: 1.00e+00 mu: 1.11e-05 li:  1
Ceres Solver Report: Iterations: 2, Initial cost: 1.250000e+01,  \
Final cost: 1.388518e-16, Termination: PARAMETER_TOLERANCE.
x : 5 -> 10
```

Starting from a $x = 5$, the solver in two iterations goes to 10. The careful reader will note that this is a linear problem and one linear solve should be enough to get the optimal value. The default configuration of the solver is aimed at non-linear problems, and for reasons of simplicity we did not change it in this example. It is indeed possible to obtain the solution to this problem using Ceres in one iteration. Also note that the solver did get very close to the optimal function value of 0 in the very first iteration. We will discuss these issues in greater detail when we talk about convergence and initial parameter setting for Ceres.

4.3 A NON-LINEAR EXAMPLE

Consider now a slightly more complicated example – the minimization of Powell's function. Let $x = [x_1, x_2, x_3, x_4]$ and

$$f_1(x) = x_1 + 10 * x_2 \tag{4.5}$$

$$f_2(x) = \sqrt{5} * (x_3 - x_4) \tag{4.6}$$

$$f_3(x) = (x_2 - 2 * x_3)^2 \tag{4.7}$$

$$f_4(x) = \sqrt{10} * (x_1 - x_4)^2 \tag{4.8}$$

$$F(x) = \frac{1}{2}\left(f_1^2(x) + f_2^2(x) + f_3^2(x) + f_4^2(x)\right) \tag{4.9}$$

$F(x)$ is a function of four parameters, and has four residuals. Now, one way to solve this problem would be to define four `CostFunctions` that computes the residual and Jacobian. *e.g.* Listing 4.3 shows the implementation for $f_4(x)$.

```
class F4 : public ceres::SizedCostFunction<1, 4> {
 public:
  virtual ~F4() {}
  virtual bool Evaluate(double const* const* parameters,
                        double* residuals,
                        double** jacobians) const {
    double x1 = parameters[0][0];
    double x4 = parameters[1][0];

    // f4 = sqrt(10) * (x1 - x4)^2
    residuals[0] = sqrt(10.0) * (x1 - x4) * (x1 - x4)
    if (jacobians != NULL) {
      // ∂_{x1} f1(x)
      jacobians[0][0] = 2.0 * sqrt(10.0) * (x1 - x4);
      // ∂_{x2} f1(x)
      jacobians[0][1] = 0.0;
      // ∂_{x3} f1(x)
      jacobians[0][2] = 0.0;
      // ∂_{x4} f1(x)
      jacobians[0][3] = -2.0 * sqrt(10.0) * (x1 - x4);
    }
    return true;
  }
};
```

A full `CostFunction` implementation of $f_4(x) = \sqrt{10} * (x_1 - x_4)^2$.

But this can get painful very quickly, especially for residuals involving complicated multivariate terms. Ceres provides two ways around this problem. Numeric and automatic symbolic differentiation.

Automatic Differentiation

With its automatic differentiation support, Ceres allows you to define templated objects that will compute the residual and it takes care of computing the Jacobians as needed and filling the `jacobians` arrays with them. For example, for $f_4(x)$ we define

```
class F4 {
 public:
  template <typename T> bool operator()(const T* const x1,
                                        const T* const x4,
                                        T* residual) const {
    // f4 = √10 * (x1 - x4)^2
    residual[0] = T(sqrt(10.0)) * (x1[0] - x4[0]) * (x1[0] - x4[0]);
    return true;
  }
};
```

Templated functor implementing $f_4(x) = \sqrt{10} * (x_1 - x_4)^2$ for use in automatic differentiation.

The important thing to note here is that the `operator()` is a templated method, which assumes that all its inputs and outputs are of some type `T`. Note also that the parameters are not packed into a single array, they are instead passed as separate arguments to `operator()`. Similarly we can define classes `F1`,`F2` and `F4`. Then let us consider the construction and solution of the problem. For brevity we only describe the relevant bits of code. The full source code for this example can be found in `examples/powell.cc`.

```
double x1 =  3.0; double x2 = -1.0; double x3 =  0.0; double x4 =  1.0;

// Add residual terms to the problem using the using the autodiff
// wrapper to get the derivatives automatically. The parameters, x1 through
// x4, are modified in place.
problem.AddResidualBlock(
  new ceres::AutoDiffCostFunction<F1, 1, 1, 1>(new F1), NULL, &x1, &x2);
problem.AddResidualBlock(
  new ceres::AutoDiffCostFunction<F2, 1, 1, 1>(new F2), NULL, &x3, &x4);
problem.AddResidualBlock(
  new ceres::AutoDiffCostFunction<F3, 1, 1, 1>(new F3), NULL, &x2, &x3)
problem.AddResidualBlock(
  new ceres::AutoDiffCostFunction<F4, 1, 1, 1>(new F4), NULL, &x1, &x4);
```

Problem construction using `AutoDiffCostFunction` for Powell's function.

A few things are worth noting in the code above. First, the object being added to the `Problem` is an `AutoDiffCostFunction` with `F1`, `F2`, `F3` and `F4` as template parameters. Second, each `ResidualBlock` only depends on the two parameters that the corresponding residual object depends on and not on all four parameters.

Compiling and running `powell.cc` gives us:

```
Initial x1 = 3, x2 = -1, x3 = 0, x4 = 1
   0: f: 1.075000e+02 d: 0.00e+00 g: 1.55e+02 h: 0.00e+00 rho: 0.00e+00 mu: 1.00e-04 li:  0
   1: f: 5.036190e+00 d: 1.02e+02 g: 2.00e+01 h: 2.16e+00 rho: 9.53e-01 mu: 3.33e-05 li:  1
   2: f: 3.148168e-01 d: 4.72e+00 g: 2.50e+00 h: 6.23e-01 rho: 9.37e-01 mu: 1.11e-05 li:  1
   3: f: 1.967760e-02 d: 2.95e-01 g: 3.13e-01 h: 3.08e-01 rho: 9.37e-01 mu: 3.70e-06 li:  1
   4: f: 1.229900e-03 d: 1.84e-02 g: 3.91e-02 h: 1.54e-01 rho: 9.37e-01 mu: 1.23e-06 li:  1
   5: f: 7.687123e-05 d: 1.15e-03 g: 4.89e-03 h: 7.69e-02 rho: 9.37e-01 mu: 4.12e-07 li:  1
   6: f: 4.804625e-06 d: 7.21e-05 g: 6.11e-04 h: 3.85e-02 rho: 9.37e-01 mu: 1.37e-07 li:  1
   7: f: 3.003028e-07 d: 4.50e-06 g: 7.64e-05 h: 1.92e-02 rho: 9.37e-01 mu: 4.57e-08 li:  1
   8: f: 1.877006e-08 d: 2.82e-07 g: 9.54e-06 h: 9.62e-03 rho: 9.37e-01 mu: 1.52e-08 li:  1
   9: f: 1.173223e-09 d: 1.76e-08 g: 1.19e-06 h: 4.81e-03 rho: 9.37e-01 mu: 5.08e-09 li:  1
  10: f: 7.333425e-11 d: 1.10e-09 g: 1.49e-07 h: 2.40e-03 rho: 9.37e-01 mu: 1.69e-09 li:  1
  11: f: 4.584044e-12 d: 6.88e-11 g: 1.86e-08 h: 1.20e-03 rho: 9.37e-01 mu: 5.65e-10 li:  1
Ceres Solver Report: Iterations: 12, Initial cost: 1.075000e+02, \
Final cost: 2.865573e-13, Termination: GRADIENT_TOLERANCE.
Final x1 = 0.000583994, x2 = -5.83994e-05, x3 = 9.55401e-05, x4 = 9.55401e-05
```

It is easy to see that the optimal solution to this problem is at $x_1 = 0, x_2 = 0, x_3 = 0, x_4 = 0$ with an objective function value of 0. In 10 iterations, Ceres finds a solution with an objective function value of 4×10^{-12}.

If a templated implementation is not possible then a `NumericDiffCostFunction` object can be used. `examples/quadratic_numeric_diff.cc` shows a numerically differentiated implementation of `examples/quadratic.cc`.

When possible, automatic differentiation should be used. The use of C++ templates makes automatic differentiation extremely efficient, whereas numeric differentiation can be quite expensive, prone to numeric errors and leads to slower convergence.

4.4 DATA FITTING

The examples we have seen until now are simple optimization problems with no data. The original purpose of least squares and non-linear least squares analysis was fitting curves to data. It is only appropriate that we now consider an example of such a problem. Let us fit some data to the curve

$$y = e^{mx+c}. \tag{4.10}$$

The full code and data for this example can be found in `examples/data_fitting.cc`. It contains data generated by sampling the curve $y = e^{0.3x+0.1}$ and adding Gaussian noise with standard deviation $\sigma = 0.2$.

We begin by defining a templated object to evaluate the residual. There will be a residual for each observation.

```
class ExponentialResidual {
 public:
  ExponentialResidual(double x, double y)
      : x_(x), y_(y) {}

  template <typename T> bool operator()(const T* const m,
                                        const T* const c,
                                        T* residual) const {
    // y - e^{mx + c}
    residual[0] = T(y_) - exp(m[0] * T(x_) + c[0]);
    return true;
  }

 private:
  // Observations for a sample.
  const double x_;
  const double y_;
};
```

Templated functor to compute the residual for the exponential model fitting problem. Note that one instance of the functor is responsible for computing the residual for one observation.

Assuming the observations are in a $2n$ sized array called data, the problem construction is a simple matter of creating a `CostFunction` for every observation.

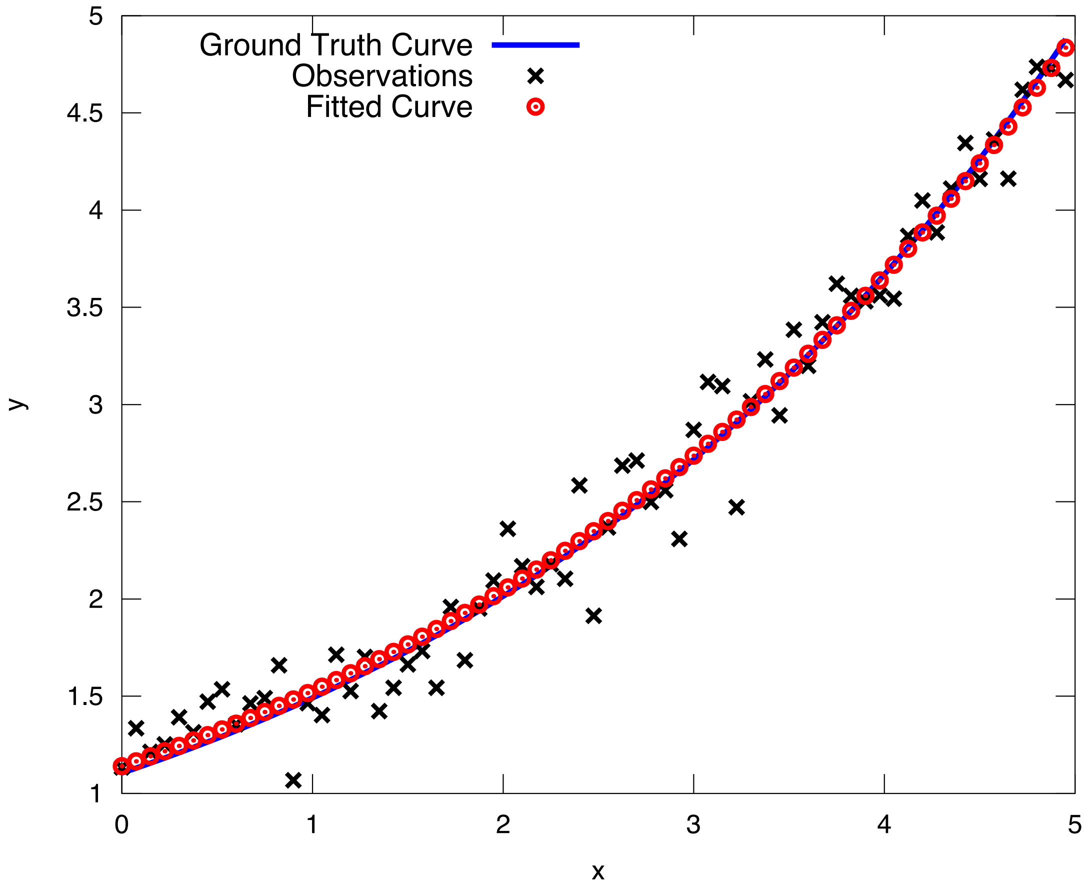
```
double m = 0.0;
double c = 0.0;

Problem problem;
for (int i = 0; i < kNumObservations; ++i) {
  problem.AddResidualBlock(
      new AutoDiffCostFunction<ExponentialResidual, 1, 1, 1>(
          new ExponentialResidual(data[2 * i], data[2 * i + 1])),
      NULL,
      &m, &c);
}
```

Problem construction for the exponential data fitting problem. A `ResidualBlock` is added for each observation.

Compiling and running `data_fitting.cc` gives us

```
 0: f: 1.211734e+02 d: 0.00e+00 g: 3.61e+02 h: 0.00e+00 rho: 0.00e+00 mu: 1.00e-04 li:  0
 1: f: 1.211734e+02 d:-2.21e+03 g: 3.61e+02 h: 7.52e-01 rho:-1.87e+01 mu: 2.00e-04 li:  1
 2: f: 1.211734e+02 d:-2.21e+03 g: 3.61e+02 h: 7.51e-01 rho:-1.86e+01 mu: 8.00e-04 li:  1
 3: f: 1.211734e+02 d:-2.19e+03 g: 3.61e+02 h: 7.48e-01 rho:-1.85e+01 mu: 6.40e-03 li:  1
 4: f: 1.211734e+02 d:-2.02e+03 g: 3.61e+02 h: 7.22e-01 rho:-1.70e+01 mu: 1.02e-01 li:  1
 5: f: 1.211734e+02 d:-7.34e+02 g: 3.61e+02 h: 5.78e-01 rho:-6.32e+00 mu: 3.28e+00 li:  1
 6: f: 3.306595e+01 d: 8.81e+01 g: 4.10e+02 h: 3.18e-01 rho: 1.37e+00 mu: 1.09e+00 li:  1
 7: f: 6.426770e+00 d: 2.66e+01 g: 1.81e+02 h: 1.29e-01 rho: 1.10e+00 mu: 3.64e-01 li:  1
 8: f: 3.344546e+00 d: 3.08e+00 g: 5.51e+01 h: 3.05e-02 rho: 1.03e+00 mu: 1.21e-01 li:  1
 9: f: 1.987485e+00 d: 1.36e+00 g: 2.33e+01 h: 8.87e-02 rho: 9.94e-01 mu: 4.05e-02 li:  1
10: f: 1.211585e+00 d: 7.76e-01 g: 8.22e+00 h: 1.05e-01 rho: 9.89e-01 mu: 1.35e-02 li:  1
11: f: 1.063265e+00 d: 1.48e-01 g: 1.44e+00 h: 6.06e-02 rho: 9.97e-01 mu: 4.49e-03 li:  1
12: f: 1.056795e+00 d: 6.47e-03 g: 1.18e-01 h: 1.47e-02 rho: 1.00e+00 mu: 1.50e-03 li:  1
13: f: 1.056751e+00 d: 4.39e-05 g: 3.79e-03 h: 1.28e-03 rho: 1.00e+00 mu: 4.99e-04 li:  1
Ceres Solver Report: Iterations: 13, Initial cost: 1.211734e+02, \
Final cost: 1.056751e+00, Termination: FUNCTION_TOLERANCE.
Initial m: 0 c: 0
Final   m: 0.291861 c: 0.131439
```

Starting from parameter values $m = 0, c = 0$ with an initial objective function value of 121.173



Least squares data fitting to the curve $y = e^{0.3x+0.1}$. Observations were generated by sampling this curve uniformly in the interval $x = (0,5)$ and adding Gaussian noise with $\sigma = 0.2$.

Ceres finds a solution $m = 0.291861, c = 0.131439$ with an objective function value of 1.05675. These values are a a bit different than the parameters of the original model $m = 0.3, c = 0.1$, but this is normal. When reconstructing a curve from noisy data, we expect to see such deviations. Indeed, if you were to evaluate the objective function for $m = 0.3, c = 0.1$, the fit is worse with an objective function value of 1.082425. Figure 4.4 illustrates the fit.

4.5 BUNDLE ADJUSTMENT

One of the main reasons for writing Ceres was our desire to solve large scale bundle adjustment problems [5, 19].

Given a set of measured image feature locations and correspondences, the goal of bundle adjustment is to find 3D point positions and camera parameters that minimize the reprojection error. This optimization problem is usually formulated as a non-linear least squares problem, where the error is the squared L_2 norm of the difference between the observed feature location and the projection of the corresponding 3D point on the image plane of the camera.

Ceres has extensive support for solving bundle adjustment problems. Let us consider the solution of a problem from the BAL [1] dataset. The code for this example can be found in `examples/simple_bundle_adjuster.cc`.

The first step is to define the CostFunction. Each residual in a BAL problem depends on a three dimensional point and a nine parameter camera. The details of this camera model can be found on Noah Snavely's Bundler homepage [2] and the BAL homepage [3]. Listing 4.5 describes the templated functor for computing the residual. Unlike the examples before this is a non-trivial function and computing its analytic Jacobian is a bit of a pain. Automatic differentiation makes our life very simple here. Given this functor, let us look at the problem construction.

```
// Create residuals for each observation in the bundle adjustment problem. The
// parameters for cameras and points are added automatically.
ceres::Problem problem;
for (int i = 0; i < bal_problem.num_observations(); ++i) {
  // Each Residual block takes a point and a camera as input and outputs a 2
  // dimensional residual. Internally, the cost function stores the observed
  // image location and compares the reprojection against the observation.
  ceres::CostFunction* cost_function =
      new ceres::AutoDiffCostFunction<SnavelyReprojectionError, 2, 9, 3>(
          new SnavelyReprojectionError(
              bal_problem.observations()[2 * i + 0],
              bal_problem.observations()[2 * i + 1]));

  problem.AddResidualBlock(cost_function,
                           NULL /* squared loss */,
                           bal_problem.mutable_camera_for_observation(i),
```

[2] http://phototour.cs.washington.edu/bundler/

[3] http://grail.cs.washington.edu/projects/bal/

```
                             bal_problem.mutable_point_for_observation(i));
  }
}
```

Note that the problem construction for bundle adjustment is not very different from the data fitting example. One extra feature here is the optional use of a robust loss function. If the user wants, instead of just using a squared reprojection error as the objective function we robustify it using Huber's loss. More details of the various loss functions available in Ceres and their characteristics can be found in `loss_function.h`.

One way to solve this problem would be to set `Solver::Options::linear_solver_type` to `SPARSE_NORMAL_CHOLESKY` and call `Solve`. And while this is a reasonable thing to do, bundle adjustment problems have a special sparsity structure that can be exploited to solve them much more efficiently. Ceres provides three specialized solvers (collectively known as Schur based solvers) for this task. The example code uses the simplest of them `DENSE_SCHUR`. For more details on the available solvers and how they work see Chapter 6.

For a more sophisticated example of bundle adjustment which demonstrates the use of the various linear solvers, robust loss functions and local parameterizations see `examples/bundle_adjuster.cc`.

```
struct SnavelyReprojectionError {
  SnavelyReprojectionError(double observed_x, double observed_y)
      : observed_x(observed_x), observed_y(observed_y) {}

  template <typename T>
  bool operator()(const T* const camera,
                  const T* const point,
                  T* residuals) const {
    // camera[0,1,2] are the angle-axis rotation.
    T p[3];
    ceres::AngleAxisRotatePoint(camera, point, p);
    // camera[3,4,5] are the translation.
    p[0] += camera[3]; p[1] += camera[4]; p[2] += camera[5];

    // Compute the center of distortion. The sign change comes from
    // the camera model that Noah Snavely's Bundler assumes, whereby
    // the camera coordinate system has a negative z axis.
    const T& focal = camera[6];
    T xp = - focal * p[0] / p[2];
    T yp = - focal * p[1] / p[2];

    // Apply second and fourth order radial distortion.
    const T& l1 = camera[7];
    const T& l2 = camera[8];
    T r2 = xp*xp + yp*yp;
    T distortion = T(1.0) + r2  * (l1 + l2  * r2);

    // Compute final projected point position.
    T predicted_x = distortion * xp;
    T predicted_y = distortion * yp;

    // The error is the difference between the predicted and observed position.
    residuals[0] = predicted_x - T(observed_x);
    residuals[1] = predicted_y - T(observed_y);
    return true;
  }

  double observed_x;
  double observed_y;
};
```

Templated functor to compute the residual using the Bundler camera. Note that the structure of this functor is similar to the `ExponentialResidual` 4.4, in that there is an instance of this object responsible for each image observation. The camera has nine parameters. Three for rotation as a Rodriquez axis-angle vector, three for translation, one for focal length and two for radial distortion. `AngleAxisRotatePoint` can be found in `rotation.h`.

Ceres solves robustified non-linear least squares problems of the form

$$\frac{1}{2}\sum_{i=1}^{k}\rho_i\left(\left\|f_i\left(x_{i_1},\ldots,x_{k_i}\right)\right\|^2\right). \tag{5.1}$$

Where $f_i()$ is a cost function that depends on the parameter blocks $\left[x_{i_1},\ldots,x_{i_k}\right]$ and ρ_i is a loss function. In most optimization problems small groups of scalars occur together. For example the three components of a translation vector and the four components of the quaternion that define the pose of a camera. We refer to such a group of small scalars as a *Parameter Block*. Of course a parameter block can just have a single parameter. The term $\rho_i\left(\left\|f_i\left(x_{i_1},\ldots,x_{k_i}\right)\right\|^2\right)$ is known as a residual block. A Ceres problem is a collection of residual blocks, each of which depends on a subset of the parameter blocks.

5.1 COSTFUNCTION

Given parameter blocks $\left[x_{i_1},\ldots,x_{i_k}\right]$, a CostFunction is responsible for computing a vector of residuals and if asked a vector of Jacobian matrices, i.e., given $\left[x_{i1},\ldots,x_{i_k}\right]$, compute the vector $f_i\left(x_{i_1},\ldots,x_{k_i}\right)$ and the matrices

$$J_{ij} = \frac{\partial}{\partial x_{i_j}} f_i\left(x_{i_1},\ldots,x_{k_i}\right), \quad \forall j = i_1,\ldots,i_k \tag{5.2}$$

```
class CostFunction {
 public:
  virtual bool Evaluate(double const* const* parameters,
                        double* residuals,
                        double** jacobians) = 0;
  const vector<int16>& parameter_block_sizes();
  int num_residuals() const;

 protected:
  vector<int16>* mutable_parameter_block_sizes();
  void set_num_residuals(int num_residuals);
};
```

The signature of the function (number and sizes of input parameter blocks and number of outputs) is stored in parameter_block_sizes_ and num_residuals_ respectively. User code inheriting from this class is expected to set these two members with the corresponding accessors. This information will be verified by the Problem when added with `Problem::AddResidualBlock`.

The most important method here is `Evaluate`. It implements the residual and Jacobian computation.

`parameters` is an array of pointers to arrays containing the various parameter blocks. parameters has the same number of elements as parameter_block_sizes_. Parameter blocks are in the same order as parameter_block_sizes_.

`residuals` is an array of size `num_residuals_`.

`jacobians` is an array of size `parameter_block_sizes_` containing pointers to storage for Jacobian matrices corresponding to each parameter block. Jacobian matrices are in the same order as `parameter_block_sizes_`, i.e., `jacobians[i]`, is an array that contains `num_residuals_` * `parameter_block_sizes_[i]` elements. Each Jacobian matrix is stored in row-major order, i.e.,

$$\texttt{jacobians[i][r*parameter\_block\_size\_[i] + c]} = \frac{\partial \texttt{residual[r]}}{\partial \texttt{parameters[i][c]}} \tag{5.3}$$

If `jacobians` is `NULL`, then no derivatives are returned; this is the case when computing cost only. If `jacobians[i]` is `NULL`, then the Jacobian matrix corresponding to the i^{th} parameter block must not be returned, this is the case when the a parameter block is marked constant.

5.2 SIZEDCOSTFUNCTION

If the size of the parameter blocks and the size of the residual vector is known at compile time (this is the common case), Ceres provides `SizedCostFunction`, where these values can be specified as template parameters.

```
template<int kNumResiduals,
         int N0 = 0, int N1 = 0, int N2 = 0, int N3 = 0, int N4 = 0, int N5 = 0>
class SizedCostFunction : public CostFunction {
 public:
  virtual bool Evaluate(double const* const* parameters,
                        double* residuals,
                        double** jacobians) = 0;
};
```

In this case the user only needs to implement the `Evaluate` method.

5.3 AUTODIFFCOSTFUNCTION

But even defining the `SizedCostFunction` can be a tedious affair if complicated derivative computations are involved. To this end Ceres provides automatic differentiation.

To get an auto differentiated cost function, you must define a class with a templated `operator()` (a functor) that computes the cost function in terms of the template parameter `T`. The autodiff framework substitutes appropriate `Jet` objects for T in order to compute the derivative when necessary, but this is hidden, and you should write the function as if T were a scalar type (e.g. a double-precision floating point number).

The function must write the computed value in the last argument (the only non-`const` one) and return true to indicate success.

For example, consider a scalar error $e = k - x^\top y$, where both x and y are two-dimensional vector parameters and k is a constant. The form of this error, which is the difference between a constant and an expression, is a common pattern in least squares problems. For example, the value $x^\top y$ might be the model expectation for a series of measurements, where there is an instance of the cost function for each measurement k.

The actual cost added to the total problem is e^2, or $(k - x^\top y)^2$; however, the squaring is implicitly done by the optimization framework.

To write an auto-differentiable cost function for the above model, first define the object

```
class MyScalarCostFunction {
  MyScalarCostFunction(double k): k_(k) {}
  template <typename T>
  bool operator()(const T* const x , const T* const y, T* e) const {
    e[0] = T(k_) - x[0] * y[0] + x[1] * y[1]
     return true;
  }

 private:
  double k_;
};
```

Note that in the declaration of `operator()` the input parameters `x` and `y` come first, and are passed as const pointers to arrays of `T`. If there were three input parameters, then the third input parameter would come after `y`. The output is always the last parameter, and is also a pointer to an array. In the example above, `e` is a scalar, so only `e[0]` is set.

Then given this class definition, the auto differentiated cost function for it can be constructed as follows.

```
CostFunction* cost_function
    = new AutoDiffCostFunction<MyScalarCostFunction, 1, 2, 2>(
        new MyScalarCostFunction(1.0));              ^  ^  ^
                                                     |  |  |
                         Dimension of residual ------+  |  |
                         Dimension of x ----------------+  |
                         Dimension of y -------------------+
```

In this example, there is usually an instance for each measurement of k.

In the instantiation above, the template parameters following `MyScalarCostFunction`, `<1, 2, 2>` describe the functor as computing a 1-dimensional output from two arguments, both 2-dimensional.

The framework can currently accommodate cost functions of up to 6 independent variables, and there is no limit on the dimensionality of each of them.

WARNING 1 Since the functor will get instantiated with different types for `T`, you must convert from other numeric types to `T` before mixing computations with other variables of type `T`. In the example above, this is seen where instead of using `k_` directly, `k_` is wrapped with `T(k_)`.

WARNING 2 A common beginner's error when first using `AutoDiffCostFunction` is to get the sizing wrong. In particular, there is a tendency to set the template parameters to (dimension of residual, number of parameters) instead of passing a dimension parameter for *every parameter block*. In the example above, that would be `<MyScalarCostFunction, 1, 2>`, which is missing the 2 as the last template argument.

5.4 NUMERICDIFFCOSTFUNCTION

To get a numerically differentiated cost function, define a subclass of `CostFunction` such that the `Evaluate` function ignores the jacobian parameter. The numeric differentiation wrapper will fill in the jacobians array if necessary by repeatedly calling the `Evaluate` method with small changes to the appropriate parameters, and computing the slope. For performance, the numeric differentiation wrapper class is templated on the concrete cost function, even though it could be implemented only in terms of the virtual `CostFunction` interface.

```
template <typename CostFunctionNoJacobian,
          NumericDiffMethod method = CENTRAL, int M = 0,
          int N0 = 0, int N1 = 0, int N2 = 0, int N3 = 0, int N4 = 0, int N5 = 0>
class NumericDiffCostFunction
    : public SizedCostFunction<M, N0, N1, N2, N3, N4, N5> {
};
```

The numerically differentiated version of a cost function for a cost function can be constructed as follows:

```
CostFunction* cost_function
    = new NumericDiffCostFunction<MyCostFunction, CENTRAL, 1, 4, 8>(
        new MyCostFunction(...), TAKE_OWNERSHIP);
```

where `MyCostFunction` has 1 residual and 2 parameter blocks with sizes 4 and 8 respectively. Look at the tests for a more detailed example.

The central difference method is considerably more accurate at the cost of twice as many function evaluations than forward difference. Consider using central differences begin with, and only after that works, trying forward difference to improve performance.

5.5 LOSSFUNCTION

For least squares problems where the minimization may encounter input terms that contain outliers, that is, completely bogus measurements, it is important to use a loss function that reduces their influence.

Consider a structure from motion problem. The unknowns are 3D points and camera parameters, and the measurements are image coordinates describing the expected reprojected position for a point in a camera. For example, we want to model the geometry of a street scene with fire hydrants and cars, observed by a moving camera with unknown parameters, and the only 3D points we care about are the pointy tippy-tops of the fire hydrants. Our magic image processing algorithm, which is responsible for producing the measurements that are input to Ceres, has found and matched all such tippy-tops in all image frames, except that in one of the frame it mistook a car's headlight for a hydrant. If we didn't do anything special the residual for the erroneous measurement will result in the entire solution getting pulled away from the optimum to reduce the large error that would otherwise be attributed to the wrong measurement.

Using a robust loss function, the cost for large residuals is reduced. In the example above, this leads to outlier terms getting down-weighted so they do not overly influence the final solution.

```
class LossFunction {
 public:
  virtual void Evaluate(double s, double out[3]) const = 0;
};
```

The key method is `Evaluate`, which given a non-negative scalar `s`, computes

$$\texttt{out} = \begin{bmatrix}\rho(s), & \rho'(s), & \rho''(s)\end{bmatrix} \tag{5.4}$$

Here the convention is that the contribution of a term to the cost function is given by $\frac{1}{2}\rho(s)$, where $s = \|f_i\|^2$. Calling the method with a negative value of s is an error and the implementations are not required to handle that case.

Most sane choices of ρ satisfy:

$$\rho(0) = 0 \tag{5.5}$$
$$\rho'(0) = 1 \tag{5.6}$$
$$\rho'(s) < 1 \text{ in the outlier region} \tag{5.7}$$
$$\rho''(s) < 0 \text{ in the outlier region} \tag{5.8}$$

so that they mimic the squared cost for small residuals.

Scaling

Given one robustifier $\rho(s)$ one can change the length scale at which robustification takes place, by adding a scale factor $a > 0$ which gives us $\rho(s,a) = a^2\rho(s/a^2)$ and the first and second derivatives as $\rho'(s/a^2)$ and $(1/a^2)\rho''(s/a^2)$ respectively.

The reason for the appearance of squaring is that a is in the units of the residual vector norm whereas s is a squared norm. For applications it is more convenient to specify a than its square.

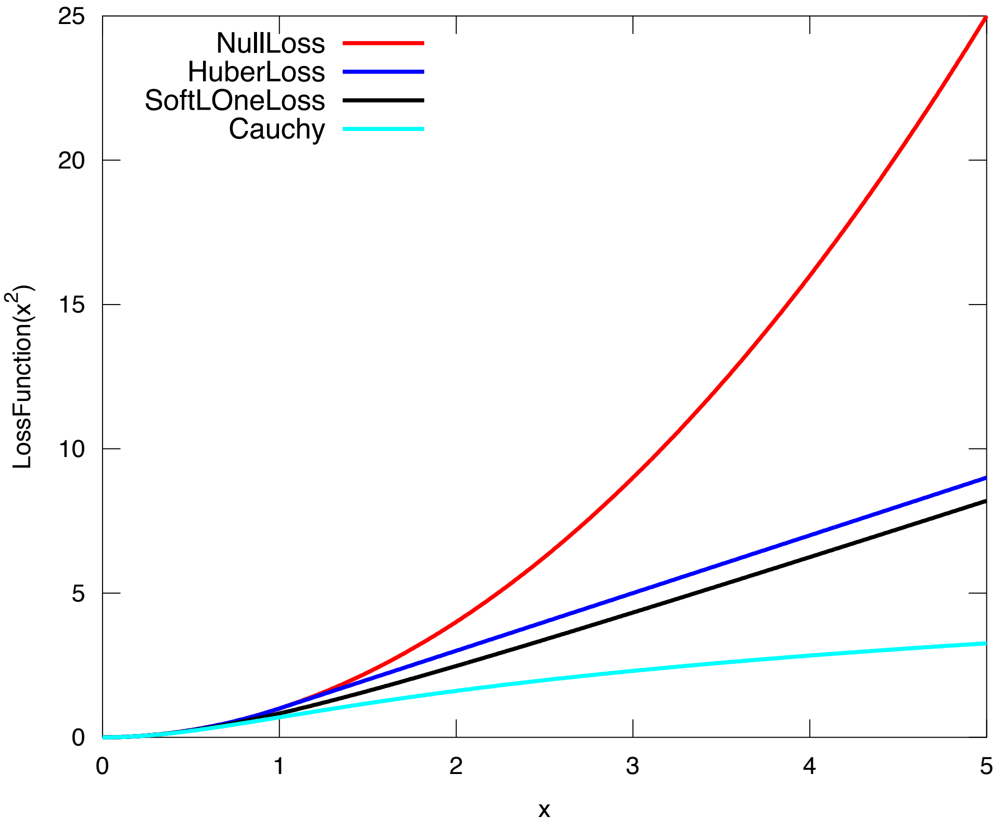
Here are some common loss functions implemented in Ceres. For simplicity we described their unscaled versions. Figure 5.5 illustrates their shape graphically.

$$\rho(s) = s \tag{NullLoss}$$

$$\rho(s) = \begin{cases} s & s \le 1 \\ 2\sqrt{s} - 1 & s > 1 \end{cases} \tag{HuberLoss}$$

$$\rho(s) = 2(\sqrt{1+s} - 1) \tag{SoftLOneLoss}$$

$$\rho(s) = \log(1+s) \tag{CauchyLoss}$$



Shape of the various common loss functions.

5.6 LOCALPARAMETERIZATION

Sometimes the parameters x can overparameterize a problem. In that case it is desirable to choose a parameterization to remove the null directions of the cost. More generally, if x lies on a manifold of a smaller dimension than the ambient space that it is embedded in, then it is numerically and computationally more effective to optimize it using a parameterization that lives in the tangent space of that manifold at each point.

For example, a sphere in three dimensions is a two dimensional manifold, embedded in a three dimensional space. At each point on the sphere, the plane tangent to it defines a two dimensional tangent space. For a cost function defined on this sphere, given a point x, moving in the direction normal to the sphere at that point is not useful. Thus a better way to parameterize a point on a sphere is to optimize over two dimensional vector Δx in the tangent space at the point on the sphere point and then "move" to the point $x + \Delta x$, where the move operation involves projecting back onto the sphere. Doing so removes a redundant dimension from the optimization, making it numerically more robust and efficient.

More generally we can define a function

$$x' = \boxplus(x, \Delta x), \tag{5.9}$$

where x' has the same size as x, and Δx is of size less than or equal to x. The function $\boxplus$, generalizes the definition of vector addition. Thus it satisfies the identity

$$\boxplus(x, 0) = x, \quad \forall x. \tag{5.10}$$

Instances of `LocalParameterization` implement the $\boxplus$ operation and its derivative with respect to Δx at $\Delta x = 0$.

```
class LocalParameterization {
 public:
  virtual ~LocalParameterization() {}
  virtual bool Plus(const double* x,
                    const double* delta,
                    double* x_plus_delta) const = 0;
  virtual bool ComputeJacobian(const double* x, double* jacobian) const = 0;
  virtual int GlobalSize() const = 0;
  virtual int LocalSize() const = 0;
};
```

`GlobalSize` is the dimension of the ambient space in which the parameter block x lives. `LocalSize` is the size of the tangent space that Δx lives in. `Plus` implements $\boxplus(x, \Delta x)$ and `ComputeJacobian` computes the Jacobian matrix

$$J = \frac{\partial}{\partial \Delta x} \boxplus(x, \Delta x)\bigg|_{\Delta x = 0} \tag{5.11}$$

in row major form.

A trivial version of $\boxplus$ is when delta is of the same size as x and

$$\boxplus(x, \Delta x) = x + \Delta x \tag{5.12}$$

A more interesting case if x is a two dimensional vector, and the user wishes to hold the first coordinate constant. Then, Δx is a scalar and $\boxplus$ is defined as

$$\boxplus(x, \Delta x) = x + \begin{bmatrix} 0 \\ 1 \end{bmatrix} \Delta x \tag{5.13}$$

`SubsetParameterization` generalizes this construction to hold any part of a parameter block constant.

Another example that occurs commonly in Structure from Motion problems is when camera rotations are parameterized using a quaternion. There, it is useful only to make updates orthogonal to that 4-vector defining the quaternion. One way to do this is to let Δx be a 3 dimensional vector and define $\boxplus$ to be

$$\boxplus(x, \Delta x) = \left[\cos(|\Delta x|), \frac{\sin(|\Delta x|)}{|\Delta x|}\Delta x\right] * x \tag{5.14}$$

The multiplication between the two 4-vectors on the right hand side is the standard quaternion product. `QuaternionParameterization` is an implementation of (5.14).

5.7 PROBLEM

```
class Problem {
 public:
  struct Options {
    Options();
    Ownership cost_function_ownership;
    Ownership loss_function_ownership;
    Ownership local_parameterization_ownership;
  };

  Problem();
  explicit Problem(const Options& options);
  ~Problem();

  ResidualBlockId AddResidualBlock(CostFunction* cost_function,
                                   LossFunction* loss_function,
                                   const vector<double*>& parameter_blocks);

  void AddParameterBlock(double* values, int size);
  void AddParameterBlock(double* values,
                         int size,
                         LocalParameterization* local_parameterization);

  void SetParameterBlockConstant(double* values);
  void SetParameterBlockVariable(double* values);
  void SetParameterization(double* values,
                           LocalParameterization* local_parameterization);

  int NumParameterBlocks() const;
  int NumParameters() const;
  int NumResidualBlocks() const;
  int NumResiduals() const;
};
```

The `Problem` objects holds the robustified non-linear least squares problem (5.1). To create a least squares problem, use the `Problem::AddResidualBlock` and `Problem::AddParameterBlock` methods.

For example a problem containing 3 parameter blocks of sizes 3, 4 and 5 respectively and two residual blocks of size 2 and 6:

```
double x1[] = { 1.0, 2.0, 3.0 };
double x2[] = { 1.0, 2.0, 3.0, 5.0 };
double x3[] = { 1.0, 2.0, 3.0, 6.0, 7.0 };

Problem problem;
problem.AddResidualBlock(new MyUnaryCostFunction(...), x1);
problem.AddResidualBlock(new MyBinaryCostFunction(...), x2, x3);
```

`AddResidualBlock` as the name implies, adds a residual block to the problem. It adds a cost function, an optional loss function, and connects the cost function to a set of parameter blocks.

The cost function carries with it information about the sizes of the parameter blocks it expects. The function checks that these match the sizes of the parameter blocks listed in `parameter_blocks`. The program aborts if a mismatch is detected. `loss_function` can be `NULL`, in which case the cost of the term is just the squared norm of the residuals.

The user has the option of explicitly adding the parameter blocks using `AddParameterBlock`. This causes additional correctness checking; however, `AddResidualBlock` implicitly adds the parameter blocks if they are not present, so calling `AddParameterBlock` explicitly is not required.

`Problem` by default takes ownership of the `cost_function` and `loss_function pointers`. These objects remain live for the life of the `Problem` object. If the user wishes to keep control over the destruction of these objects, then they can do this by setting the corresponding enums in the `Options` struct.

Note that even though the Problem takes ownership of `cost_function` and `loss_function`, it does not preclude the user from re-using them in another residual block. The destructor takes care to call delete on each `cost_function` or `loss_function` pointer only once, regardless of how many residual blocks refer to them.

`AddParameterBlock` explicitly adds a parameter block to the `Problem`. Optionally it allows the user to associate a LocalParameterization object with the parameter block too. Repeated calls with the same arguments are ignored. Repeated calls with the same double pointer but a different size results in undefined behaviour.

You can set any parameter block to be constant using

`Problem::SetParameterBlockConstant`

and undo this using

`Problem::SetParameterBlockVariable`.

In fact you can set any number of parameter blocks to be constant, and Ceres is smart enough to figure out what part of the problem you have constructed depends on the parameter blocks that are free to change and only spends time solving it. So for example if you constructed a problem with a million parameter blocks and 2 million residual blocks, but then set all but one parameter blocks to be constant and say only 10 residual blocks depend on this one non-constant parameter block. Then the computational effort Ceres spends in solving this problem will be the same if you had defined a problem with one parameter block and 10 residual blocks.

`Problem` by default takes ownership of the `cost_function`, `loss_function` and `local_parameterization` pointers. These objects remain live for the life of the `Problem` object. If the user wishes to keep control over the destruction of these objects, then they can do this by setting the corresponding enums in the `Options` struct. Even though `Problem` takes ownership of these pointers, it does not preclude the user from re-using them in another residual or parameter block. The destructor takes care to call delete on each pointer only once.

5.8 SOLVER::OPTIONS

`Solver::Options` controls the overall behavior of the solver. We list the various settings and their default values below.

1. `minimizer_type(LEVENBERG_MARQUARDT)` The minimization algorithm used by Ceres. `LEVENBERG_MARQUARD` is currently the only valid value.
2. `max_num_iterations(50)` Maximum number of iterations for Levenberg-Marquardt.
3. `max_solver_time_sec(1e9)` Maximum amount of time (in seconds) for which the solver should run.
4. `num_threads(1)` Number of threads used by Ceres to evaluate the Jacobian.
5. `tau(1e-4)` Initial value of the regularization parameter μ used by the Levenberg-Marquardt algorithm. The size of this parameter indicate the user's guess of how far the initial solution is from the minimum. Large values indicates that the solution is far away.
6. `min_mu(1e-20)` For Levenberg-Marquardt, the minimum value of the regularizer. For well constrained problems there shold be no need to modify the default value, but in some cases, going below a certain minimum reliably triggers rank deficiency in the normal equations. In such cases, the Levenberg-Marquardt algorithm can oscillate between lowering the value of mu, seeing a numerical failure, and then increasing it making some progress and then reducing it again.

   In such cases, it is useful to set a higher value for `min_mu`.
7. `min_relative_decrease(1e-3)` Lower threshold for relative decrease before a Levenberg-Marquardt step is acceped.
8. `function_tolerance(1e-6)` Solver terminates if

   $$\frac{|\Delta \text{cost}|}{\text{cost}} < \texttt{function\_tolerance} \tag{5.15}$$

   where, Δcost is the change in objective function value (up or down) in the current iteration of Levenberg-Marquardt.
9. `Solver::Options::gradient_tolerance` Solver terminates if

   $$\frac{\|g(x)\|_\infty}{\|g(x_0)\|_\infty} < \texttt{gradient\_tolerance} \tag{5.16}$$

   where $\|\cdot\|_\infty$ refers to the max norm, and x_0 is the vector of initial parameter values.

10. `parameter_tolerance(1e-8)` Solver terminates if

$$\frac{\|\Delta x\|}{\|x\| + \texttt{parameter\_tolerance}} < \texttt{parameter\_tolerance} \tag{5.17}$$

where Δx is the step computed by the linear solver in the current iteration of Levenberg-Marquardt.

11. `linear_solver_type(SPARSE_NORMAL_CHOLESKY/DENSE_QR)` Type of linear solver used to compute the solution to the linear least squares problem in each iteration of the Levenberg-Marquardt algorithm. If Ceres is build with `SuiteSparse`linked in then the default is `SPARSE_NORMAL_CHOLESKY`, it is `DENSE_QR` otherwise.

12. `preconditioner_type(JACOBI)` The preconditioner used by the iterative linear solver. The default is the block Jacobi preconditioner. Valid values are (in increasing order of complexity) `IDENTITY`,`JACOBI`, `SCHUR_JACOBI`, `CLUSTER_JACOBI` and `CLUSTER_TRIDIAGONAL`.

13. `num_linear_solver_threads(1)` Number of threads used by the linear solver.

14. `num_eliminate_blocks(0)` For Schur reduction based methods, the first 0 to num blocks are eliminated using the Schur reduction. For example, when solving traditional structure from motion problems where the parameters are in two classes (cameras and points) then `num_eliminate_blocks` would be the number of points.

15. `ordering_type(NATURAL)` Internally Ceres reorders the parameter blocks to help the various linear solvers. This parameter allows the user to influence the re-ordering strategy used. For structure from motion problems use `SCHUR`, for other problems `NATURAL` (default) is a good choice. In case you wish to specify your own ordering scheme, for example in conjunction with `num_eliminate_blocks`, use `USER`.

16. `ordering` The ordering of the parameter blocks. The solver pays attention to it if the `ordering_type` is set to `USER` and the ordering vector is non-empty.

17. `linear_solver_min_num_iterations(1)` Minimum number of iterations used by the linear solver. This only makes sense when the linear solver is an iterative solver, e.g., `ITERATIVE_SCHUR`.

18. `linear_solver_max_num_iterations(500)` Minimum number of iterations used by the linear solver. This only makes sense when the linear solver is an iterative solver, e.g., `ITERATIVE_SCHUR`.

19. `eta(1e-1)` Forcing sequence parameter. The truncated Newton solver uses this number to control the relative accuracy with which the Newton step is computed. This constant

is passed to ConjugateGradientsSolver which uses it to terminate the iterations when

$$\frac{Q_i - Q_{i-1}}{Q_i} < \frac{\eta}{i} \tag{5.18}$$

20. `jacobi_scaling(true)` `true` means that the Jacobian is scaled by the norm of its columns before being passed to the linear solver. This improves the numerical conditioning of the normal equations.
21. `logging_type(PER_MINIMIZER_ITERATION)`
22. `minimizer_progress_to_stdout(false)` By default the Minimizer progress is logged to `STDERR` depending on the `vlog` level. If this flag is set to true, and `logging_type` is not `SILENT`, the logging output is sent to `STDOUT`.
23. `return_initial_residuals(false)`
24. `return_final_residuals(false)`
25. `lsqp_iterations_to_dump` List of iterations at which the optimizer should dump the linear least squares problem to disk. Useful for testing and benchmarking. If empty (default), no problems are dumped.
26. `lsqp_dump_directory` (`/tmp`) If `lsqp_iterations_to_dump` is non-empty, then this setting determines the directory to which the files containing the linear least squares problems are written to.
27. `lsqp_dump_formatTEXTFILE` The format in which linear least squares problems should be logged when `lsqp_iterations_to_dump` is non-empty. There are three options
    - `CONSOLE` prints the linear least squares problem in a human readable format to `stderr`. The Jacobian is printed as a dense matrix. The vectors D, x and f are printed as dense vectors. This should only be used for small problems.
    - `PROTOBUF` Write out the linear least squares problem to the directory pointed to by `lsqp_dump_directory` as a protocol buffer. `linear_least_squares_problems.h/cc` contains routines for loading these problems. For details on the on disk format used, see `matrix.proto`. The files are named `lm_iteration_???.lsqp`. This requires that `protobuf` be linked into Ceres Solver.
    - `TEXTFILE` Write out the linear least squares problem to the directory pointed to by `lsqp_dump_directory` as text files which can be read into `MATLAB/Octave`. The Jacobian is dumped as a text file containing (i,j,s) triplets, the vectors D, x and f are dumped as text files containing a list of their values.

      A `MATLAB/Octave` script called `lm_iteration_???.m` is also output, which can be used to parse and load the problem into memory.

28. `crash_and_dump_lsqp_on_failure(false)` Dump the linear least squares problem to disk if the minimizer fails due to `NUMERICAL_FAILURE` and crash the process. This flag is useful for generating debugging information. The problem is dumped in a file whose name is determined by `lsqp_dump_format`. Note that this requires a version of Ceres built with protocol buffers.

29. `check_gradients(false)` Check all Jacobians computed by each residual block with finite differences. This is expensive since it involves computing the derivative by normal means (e.g. user specified, autodiff, etc), then also computing it using finite differences. The results are compared, and if they differ substantially, details are printed to the log.

30. `gradient_check_relative_precision(1e-8)` Relative precision to check for in the gradient checker. If the relative difference between an element in a Jacobian exceeds this number, then the Jacobian for that cost term is dumped.

31. `numeric_derivative_relative_step_size(1e-6)` Relative shift used for taking numeric derivatives. For finite differencing, each dimension is evaluated at slightly shifted values, *e.g.* for forward differences, the numerical derivative is

$$\delta = \texttt{numeric\_derivative\_relative\_step\_size} \tag{5.19}$$

$$\Delta f = \frac{f((1+\delta)x) - f(x)}{\delta x} \tag{5.20}$$

The finite differencing is done along each dimension. The reason to use a relative (rather than absolute) step size is that this way, numeric differentiation works for functions where the arguments are typically large (e.g. 1e9) and when the values are small (e.g. 1e-5). It is possible to construct "torture cases" which break this finite difference heuristic, but they do not come up often in practice.

32. `callbacks` Callbacks that are executed at the end of each iteration of the `Minimizer`. They are executed in the order that they are specified in this vector. By default, parameter blocks are updated only at the end of the optimization, i.e when the `Minimizer` terminates. This behavior is controlled by `update_state_every_variable`. If the user wishes to have access to the update parameter blocks when his/her callbacks are executed, then set `update_state_every_iteration` to true.

    The solver does NOT take ownership of these pointers.

33. `update_state_every_iteration(false)` Normally the parameter blocks are only updated when the solver terminates. Setting this to true update them in every iteration. This setting is useful when building an interactive application using Ceres and using an `IterationCallback`.

5.9 SOLVER::SUMMARY

TBD

THEORY

Effective use of Ceres requires some familiarity with the underlying theory. In this chapter we will provide a brief exposition to Ceres's approach to solving non-linear least squares optimization. Much of the material in this section comes from [1, 6, 21].

6.1 THE LEVENBERG-MARQUARDT ALGORITHM

The Levenberg-Marquardt algorithm [7, 11] is the most popular algorithm for solving non-linear least squares problems. Ceres implements an exact step [9] and an inexact step variant of the Levenberg-Marquardt algorithm [13, 20]. We begin by taking a brief look at how the Levenberg-Marquardt algorithm works.

Let $x \in \mathbb{R}^n$ be an n-dimensional vector of variables, and $F(x) = [f_1(x), \dots, f_m(x)]^\top$ be a m-dimensional function of x. We are interested in solving the following optimization problem,

$$\arg\min_x \frac{1}{2}\|F(x)\|^2 \; . \tag{6.1}$$

The Jacobian $J(x)$ of $F(x)$ is an $m \times n$ matrix, where $J_{ij}(x) = \partial_j f_i(x)$ and the gradient vector $g(x) = \nabla \frac{1}{2}\|F(x)\|^2 = J(x)^\top F(x)$. Since the efficient global optimization of (6.1) for general $F(x)$ is an intractable problem, we will have to settle for finding a local minimum.

The general strategy when solving non-linear optimization problems is to solve a sequence of approximations to the original problem [14]. At each iteration, the approximation is solved to determine a correction Δx to the vector x. For non-linear least squares, an approximation can be constructed by using the linearization $F(x+\Delta x) \approx F(x) + J(x)\Delta x$, which leads to the following linear least squares problem:

$$\min_{\Delta x} \frac{1}{2}\|J(x)\Delta x + F(x)\|^2 \tag{6.2}$$

Unfortunately, naïvely solving a sequence of these problems and updating $x \leftarrow x + \Delta x$ leads to an algorithm that may not converge. To get a convergent algorithm, we need to control the size of the step Δx. One way to do this is to introduce a regularization term:

$$\min_{\Delta x} \frac{1}{2}\|J(x)\Delta x + F(x)\|^2 + \mu\|D(x)\Delta x\|^2 \; . \tag{6.3}$$

Here, $D(x)$ is a non-negative diagonal matrix, typically the square root of the diagonal of the matrix $J(x)^\top J(x)$ and μ is a non-negative parameter that controls the strength of regularization. It is straightforward to show that the step size $\|\Delta x\|$ is inversely related to μ. Levenberg-Marquardt updates the value of μ at each step based on how well the Jacobian $J(x)$ approximates $F(x)$. The quality of this fit is measured by the ratio of the actual decrease in the objective function to the decrease in the value of the linearized model $L(\Delta x) = \frac{1}{2}\|J(x)\Delta x + F(x)\|^2$.

$$\rho = \frac{\|F(x+\Delta x)\|^2 - \|F(x)\|^2}{\|J(x)\Delta x + F(x)\|^2 - \|F(x)\|^2} \tag{6.4}$$

If ρ is large, that means the linear model is a good approximation to the non-linear model and it is worth trusting it more in the computation of Δx, so we decrease μ. If ρ is small, the linear model is a poor approximation and μ is increased. This kind of reasoning is the basis of Trust-region methods, of which Levenberg-Marquardt is an early example [14].

Before going further, let us make some notational simplifications. We will assume that the matrix $\sqrt{\mu}D$ has been concatenated at the bottom of the matrix J and similarly a vector of zeros has been added to the bottom of the vector f and the rest of our discussion will be in terms of J and f, *i.e.* the linear least squares problem.

$$\min_{\Delta x} \frac{1}{2}\|J(x)\Delta x + f(x)\|^2. \tag{6.5}$$

Further, let $H(x) = J(x)^\top J(x)$ and $g(x) = -J(x)^\top f(x)$. For notational convenience let us also drop the dependence on x. Then it is easy to see that solving (6.5) is equivalent to solving the *normal equations*

$$H\Delta x = g \tag{6.6}$$

For all but the smallest problems the solution of (6.6) in each iteration of the Levenberg-Marquardt algorithm is the dominant computational cost in Ceres. Ceres provides a number of different options for solving (6.6).

6.2 `DENSE_QR`

For small problems (a couple of hundred parameters and a few thousand residuals) with relatively dense Jacobians, `DENSE_QR` is the method of choice [2]. Let $J = QR$ be the QR-decomposition of J, where Q is an orthonormal matrix and R is an upper triangular matrix [18]. Then it can be shown that the solution to (6.6) is given by

$$\Delta x^* = -R^{-1}Q^\top f \tag{6.7}$$

Ceres uses `Eigen`'s dense QR decomposition routines.

6.3 SPARSE_NORMAL_CHOLESKY

Large non-linear least square problems are usually sparse. In such cases, using a dense QR factorization is inefficient. Let $H = R^\top R$ be the Cholesky factorization of the normal equations, where R is an upper triangular matrix, then the solution to (6.6) is given by

$$\Delta x^* = R^{-1} R^{-\top} g. \tag{6.8}$$

The observant reader will note that the R in the Cholesky factorization of H is the same upper triangular matrix R in the QR factorization of J. Since Q is an orthonormal matrix, $J = QR$ implies that $J^\top J = R^\top Q^\top Q R = R^\top R$.

There are two variants of Cholesky factorization – sparse and dense. `SPARSE_NORMAL_CHOLESKY`, as the name implies performs a sparse Cholesky factorization of the normal equations. This leads to substantial savings in time and memory for large sparse problems. We use the Professor Tim Davis' `CHOLMOD` library (part of the `SuiteSparse` package) to perform the sparse cholesky [4].

6.4 DENSE_SCHUR & SPARSE_SCHUR

While it is possible to use `SPARSE_NORMAL_CHOLESKY` to solve bundle adjustment problems, bundle adjustment problem have a special structure, and a more efficient scheme for solving (6.6) can be constructed.

Suppose that the SfM problem consists of p cameras and q points and the variable vector x has the block structure $x = [y_1, \ldots, y_p, z_1, \ldots, z_q]$. Where, y and z correspond to camera and point parameters, respectively. Further, let the camera blocks be of size c and the point blocks be of size s (for most problems c = 6–9 and $s = 3$). Ceres does not impose any constancy requirement on these block sizes, but choosing them to be constant simplifies the exposition.

A key characteristic of the bundle adjustment problem is that there is no term f_i that includes two or more point blocks. This in turn implies that the matrix H is of the form

$$H = \begin{bmatrix} B & E \\ E^\top & C \end{bmatrix}, \tag{6.9}$$

where, $B \in \mathbb{R}^{pc \times pc}$ is a block sparse matrix with p blocks of size $c \times c$ and $C \in \mathbb{R}^{qs \times qs}$ is a block diagonal matrix with q blocks of size $s \times s$. $E \in \mathbb{R}^{pc \times qs}$ is a general block sparse matrix, with a block of size $c \times s$ for each observation. Let us now block partition $\Delta x = [\Delta y, \Delta z]$ and $g = [v, w]$ to restate (6.6) as the block structured linear system

$$\begin{bmatrix} B & E \\ E^\top & C \end{bmatrix} \begin{bmatrix} \Delta y \\ \Delta z \end{bmatrix} = \begin{bmatrix} v \\ w \end{bmatrix}, \tag{6.10}$$

and apply Gaussian elimination to it. As we noted above, C is a block diagonal matrix, with small diagonal blocks of size $s \times s$. Thus, calculating the inverse of C by inverting each of these blocks is cheap. This allows us to eliminate Δz by observing that $\Delta z = C^{-1}(w - E^\top \Delta y)$, giving us

$$\left[B - EC^{-1}E^\top\right] \Delta y = v - EC^{-1}w \ . \tag{6.11}$$

The matrix

$$S = B - EC^{-1}E^\top \ , \tag{6.12}$$

is the Schur complement of C in H. It is also known as the *reduced camera matrix*, because the only variables participating in (6.11) are the ones corresponding to the cameras. $S \in \mathbb{R}^{pc \times pc}$ is a block structured symmetric positive definite matrix, with blocks of size $c \times c$. The block S_{ij} corresponding to the pair of images i and j is non-zero if and only if the two images observe at least one common point.

Now, (6.10) can be solved by first forming S, solving for Δy, and then back-substituting Δy to obtain the value of Δz. Thus, the solution of what was an $n \times n$, $n = pc + qs$ linear system is reduced to the inversion of the block diagonal matrix C, a few matrix-matrix and matrix-vector multiplies, and the solution of block sparse $pc \times pc$ linear system (6.11). For almost all problems, the number of cameras is much smaller than the number of points, $p \ll q$, thus solving (6.11) is significantly cheaper than solving (6.10). This is the *Schur complement trick* [3].

This still leaves open the question of solving (6.11). The method of choice for solving symmetric positive definite systems exactly is via the Cholesky factorization [18] and depending upon the structure of the matrix, there are, in general, two options. The first is direct factorization, where we store and factor S as a dense matrix [18]. This method has $O(p^2)$ space complexity and $O(p^3)$ time complexity and is only practical for problems with up to a few hundred cameras. Ceres implements this strategy as the `DENSE_SCHUR` solver.

But, S is typically a fairly sparse matrix, as most images only see a small fraction of the scene. This leads us to the second option: sparse direct methods. These methods store S as a sparse matrix, use row and column re-ordering algorithms to maximize the sparsity of the Cholesky decomposition, and focus their compute effort on the non-zero part of the factorization [4]. Sparse direct methods, depending on the exact sparsity structure of the Schur complement, allow bundle adjustment algorithms to significantly scale up over those based on dense factorization. Ceres implements this strategy as the `SPARSE_SCHUR` solver.

6.5 `ITERATIVE_SCHUR`

The factorization methods described above are based on computing an exact solution of (6.3). But it is not clear if an exact solution of (6.3) is necessary at each step of the LM algorithm to solve (6.1). In fact, we have already seen evidence that this may not be the case, as (6.3) is itself a regularized version of (6.2). Indeed, it is possible to construct non-linear optimization algorithms in which the linearized problem is solved approximately. These algorithms are known as inexact Newton or truncated Newton methods [14].

An inexact Newton method requires two ingredients. First, a cheap method for approximately solving systems of linear equations. Typically an iterative linear solver like the Conjugate Gradients method is used for this purpose [14]. Second, a termination rule for the iterative solver. A typical termination rule is of the form

$$\|H(x)\Delta x + g(x)\| \leq \eta_k \|g(x)\|. \tag{6.13}$$

Here, k indicates the Levenberg-Marquardt iteration number and $0 < \eta_k < 1$ is known as the forcing sequence. Wright & Holt [20] prove that a truncated Levenberg-Marquardt algorithm that uses an inexact Newton step based on (6.13) converges for any sequence $\eta_k \leq \eta_0 < 1$ and the rate of convergence depends on the choice of the forcing sequence η_k.

The convergence rate of Conjugate Gradients for solving (6.6) depends on the distribution of eigenvalues of H [15]. A useful upper bound is $\sqrt{\kappa(H)}$, where, $\kappa(H)$f is the condition number of the matrix H. For most bundle adjustment problems, $\kappa(H)$ is high and a direct application of Conjugate Gradients to (6.6) results in extremely poor performance.

The solution to this problem is to replace (6.6) with a *preconditioned* system. Given a linear system, $Ax = b$ and a preconditioner M the preconditioned system is given by $M^{-1}Ax = M^{-1}b$. The resulting algorithm is known as Preconditioned Conjugate Gradients algorithm (PCG) and its worst case complexity now depends on the condition number of the *preconditioned* matrix $\kappa(M^{-1}A)$.

Preconditioning

The computational cost of using a preconditioner M is the cost of computing M and evaluating the product $M^{-1}y$ for arbitrary vectors y. Thus, there are two competing factors to consider: How much of H's structure is captured by M so that the condition number $\kappa(HM^{-1})$ is low, and the computational cost of constructing and using M. The ideal preconditioner would be one for which $\kappa(M^{-1}A) = 1$. $M = A$ achieves this, but it is not a practical choice, as applying this preconditioner would require solving a linear system equivalent to the unpreconditioned problem. It is usually the case that the more information M has about H, the more expensive it is use. For example, Incomplete Cholesky factorization based preconditioners have much better convergence behavior than the Jacobi preconditioner, but are also much more expensive.

The simplest of all preconditioners is the diagonal or Jacobi preconditioner, *i.e.* , $M = \operatorname{diag}(A)$, which for block structured matrices like H can be generalized to the block Jacobi preconditioner. Another option is to apply PCG to the reduced camera matrix S instead of H. One reason to do this is that S is a much smaller matrix than H, but more importantly, it can be shown that $\kappa(S) \leq \kappa(H)$. Ceres implements PCG on S as the `ITERATIVE_SCHUR` solver. When the user chooses `ITERATIVE_SCHUR` as the linear solver, Ceres automatically switches from the exact step algorithm to an inexact step algorithm.

There are two obvious choices for block diagonal preconditioners for S. The block diagonal of the matrix B [10] and the block diagonal S, *i.e.* the block Jacobi preconditioner for S. Ceres's implements both of these preconditioners and refers to them as `JACOBI` and `SCHUR_JACOBI` respectively.

As discussed earlier, the cost of forming and storing the Schur complement S can be prohibitive for large problems. Indeed, for an inexact Newton solver that computes S and runs PCG on it, almost all of its time is spent in constructing S; the time spent inside the PCG algorithm is negligible in comparison. Because PCG only needs access to S via its product with a vector, one way to evaluate Sx is to observe that

$$\begin{aligned} x_1 &= E^\top x \\ x_2 &= C^{-1} x_1 \\ x_3 &= E x_2 \\ x_4 &= B x \\ Sx &= x_4 - x_3 \ . \end{aligned} \tag{6.14}$$

Thus, we can run PCG on S with the same computational effort per iteration as PCG on H, while reaping the benefits of a more powerful preconditioner. In fact, we do not even need to compute H, (6.14) can be implemented using just the columns of J.

Equation (6.14) is closely related to *Domain Decomposition methods* for solving large linear systems that arise in structural engineering and partial differential equations. In the language of Domain Decomposition, each point in a bundle adjustment problem is a domain, and the cameras form the interface between these domains. The iterative solution of the Schur complement then falls within the sub-category of techniques known as Iterative Substructuring [12, 15].

For bundle adjustment problems, particularly those arising in reconstruction from community photo collections, more effective preconditioners can be constructed by analyzing and exploiting the camera-point visibility structure of the scene [6]. Ceres implements the two visibility based preconditioners described by Kushal & Agarwal as `CLUSTER_JACOBI` and `CLUSTER_TRIDIAGONAL`. These are fairly new preconditioners and Ceres' implementation of them is in its early stages and is not as mature as the other preconditioners described above.

6.6 CGNR

For general sparse problems, if the problem is too large for `CHOLMOD` or `SuiteSparse` is not linked into Ceres, another option is the `CGNR` solver. This solver uses the Conjugate Gradients solver on the *normal equations*, but without forming the normal equations explicitly. It exploits the relation

$$Hx = J^\top Jx = J^\top(Jx) \tag{6.15}$$

Currently only the `JACOBI` preconditioner is available for use with this solver. It uses the block diagonal of H as a preconditioner.

6.7 ORDERING

All three of the Schur based solvers depend on the user indicating to the solver, which of the parameter blocks correspond to the points and which correspond to the cameras. Ceres refers to them as `e_blocks` and `f_blocks`. The only constraint on `e_blocks` is that there should be no term in the objective function with two or more `e_blocks`.

As we saw in Section 4.5, there are two ways to indicate `e_blocks` to Ceres. The first is to explicitly create an ordering vector `Solver::Options::ordering` containing the parameter blocks such that all the `e_blocks`/points occur before the `f_blocks`, and setting `Solver::Options::num_eliminat` to the number `e_blocks`.

For some problems this is an easy thing to do and we recommend its use. In some problems though, this is onerous and it would be better if Ceres could automatically determine `e_blocks`. Setting `Solver::Options::ordering_type` to `SCHUR` achieves this.

The `SCHUR` ordering algorithm is based on the observation that the constraint that no two `e_block` co-occur in a residual block means that if we were to treat the sparsity structure of the block matrix H as a graph, then the set of `e_blocks` is an independent set in this graph. The larger the number of `e_block`, the smaller is the size of the Schur complement S. Indeed the reason Schur based solvers are so efficient at solving bundle adjustment problems is because the number of points in a bundle adjustment problem is usually an order of magnitude or two larger than the number of cameras.

Thus, the aim of the `SCHUR` ordering algorithm is to identify the largest independent set in the graph of H. Unfortunately this is an NP-Hard problem. But there is a greedy approximation algorithm that performs well [8] and we use it to identify `e_blocks` in Ceres.

6.8 AUTOMATIC DIFFERENTIATION

TBD

6.9 LOSS FUNCTION

TBD

6.10 LOCAL PARAMETERIZATIONS

TBD

FREQUENTLY ASKED QUESTIONS

1. Why does Ceres use blocks (ParameterBlocks and ResidualBlocks) ?

   Most non-linear solvers we are aware of, define the problem and residuals in terms of scalars and it is possible to do this with Ceres also. However, it is our experience that in most problems small groups of scalars occur together. For example the three components of a translation vector and the four components of the quaternion that define the pose of a camera. Same is true for residuals, where it is common to have small vectors of residuals rather than just scalars. There are a number of advantages of using blocks. It saves on indexing information, which for large problems can be substantial. Blocks translate into contiguous storage in memory which is more cache friendly and last but not the least, it allows us to use SIMD/SSE based BLAS routines to significantly speed up various matrix operations.

2. What is a good ParameterBlock?

   In most problems there is a natural parameter block structure, as there is a semantic meaning associated with groups of scalars – mean vector of a distribution, color of a pixel etc. To group two scalar variables, ask yourself if residual blocks will always use these two variables together. If the answer is yes, then the two variables belong to the same parameter block.

3. What is a good ResidualBlock?

   While it is often the case that problems have a natural blocking of parameters into parameter blocks, it is not always clear what a good residual block structure is. One rule of thumb for non-linear least squares problems since they often come from data fitting problems is to create one residual block per observation. So if you are solving a Structure from Motion problem, one 2 dimensional residual block per 2d image projection is a good idea.

   The flips side is that sometimes, when modeling the problem it is tempting to group a large number of residuals together into a single residual block as it reduces the number of CostFunctions you have to define.

   For example consider the following residual block of size 18 which depends on four parameter blocks of size 4 each. Shown below is the Jacobian structure of this residual block, the numbers in the columns indicate the size, and the numbers in the rows show a grouping of the matrix that best capture its sparsity structure. `X` indicates a non-zero block, the rest of the blocks are zero.

$$\begin{array}{ccccc}
 & 4 & 4 & 4 & 4 \\
2 & X & X & X & X \\
4 & X & & & \\
4 & & X & & \\
4 & & & X & \\
4 & & & & X
\end{array}$$

Notice that out of the 20 cells, only 8 are non-zero, in fact out of the 288 entries only 48 entries are non-zero, thus we are hiding substantial sparsity from the solver, and using up much more memory. It is much better to break this up into 5 residual blocks. One residual block of size 2 that depends on all four parameter block and four residual blocks of size 4 each that depend on one parameter block at a time.

4. Can I set part of a parameter block constant?

   Yes, use `SubsetParameterization` as a local parameterization for the parameter block of interest. See `local_parameterization.h` for more details.

5. Can Ceres solve constrained non-linear least squares?

   Not at this time. We have some ideas on how to do this, but we have not had very many requests to justify the effort involved. If you have a problem that requires such a functionality we would like to hear about it as it will help us decide directions for future work. In the meanwhile, if you are interested in solving bounds constrained problems, consider using some of the tricks described by John D'Errico in his fminsearchbnd toolkit [1].

6. Can Ceres solve problems that cannot be written as robustified non-linear least squares?

   No. Ceres was designed from the grounds up to be a non-linear least squares solver. Currently we have no plans of extending it into a general purpose non-linear solver.

[1] http://www.mathworks.com/matlabcentral/fileexchange/8277-fminsearchbnd

FURTHER READING

For a short but informative introduction to the subject we recommend the booklet by Madsel et al. [9]. For a general introduction to non-linear optimization we recommend the text by Nocedal & Wright [14]. Björck's book remains the seminal reference on least squares problems [2]. Trefethen & Bau's book is our favourite text on introductory numerical linear algebra [18]. Triggs et al., provide a thorough coverage of the bundle adjustment problem [19].

BIBLIOGRAPHY

[1] S. Agarwal, N. Snavely, S. M. Seitz, and R. Szeliski. Bundle adjustment in the large. In *Proceedings of the European Conference on Computer Vision*, pages 29–42, 2010.

[2] A. Björck. *Numerical methods for least squares problems*. SIAM, 1996.

[3] D. C. Brown. A solution to the general problem of multiple station analytical stereo triangulation. Technical Report 43, Patrick Airforce Base, Florida, 1958.

[4] Y. Chen, T. A. Davis, W. W. Hager, and S. Rajamanickam. Algorithm 887: CHOLMOD, Supernodal Sparse Cholesky Factorization and Update/Downdate. *TOMS*, 35(3), 2008.

[5] R. I. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, 2003.

[6] A. Kushal and S. Agarwal. Visibility based preconditioning for bundle adjustment. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2012.

[7] K. Levenberg. A method for the solution of certain nonlinear problems in least squares. *Quart. Appl. Math*, 2(2):164–168, 1944.

[8] Na Li and Y. Saad. Miqr: A multilevel incomplete qr preconditioner for large sparse least-squares problems. *SIAM Journal on Matrix Analysis and Applications*, 28(2):524–550, 2007.

[9] K. Madsen, H.B. Nielsen, and O. Tingleff. *Methods for non-linear least squares problems*. 2004.

[10] J. Mandel. On block diagonal and Schur complement preconditioning. *Numer. Math.*, 58(1):79–93, 1990.

[11] D.W. Marquardt. An algorithm for least-squares estimation of nonlinear parameters. *J. SIAM*, 11(2):431–441, 1963.

[12] T.P.A. Mathew. *Domain decomposition methods for the numerical solution of partial differential equations*. Springer Verlag, 2008.

[13] S.G. Nash and A. Sofer. Assessing a search direction within a truncated-newton method. *Operations Research Letters*, 9(4):219–221, 1990.

[14] J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer, 2000.

[15] Y. Saad. *Iterative methods for sparse linear systems*. SIAM, 2003.

[16] S.M. Stigler. Gauss and the invention of least squares. *The Annals of Statistics*, 9(3):465–474, 1981.

[17] J. Tennenbaum and B. Director. How Gauss Determined the Orbit of Ceres.

[18] L.N. Trefethen and D. Bau. *Numerical Linear Algebra*. SIAM, 1997.

[19] B. Triggs, P. F. McLauchlan, R. I. Hartley, and A. W. Fitzgibbon. Bundle Adjustment - A Modern Synthesis. In *Vision Algorithms*, pages 298–372, 1999.

[20] S. J. Wright and J. N. Holt. An Inexact Levenberg-Marquardt Method for Large Sparse Nonlinear Least Squares. *Journal of the Australian Mathematical Society Series B*, 26(4):387–403, 1985.

[21] C. Wu, S. Agarwal, B. Curless, and S.M. Seitz. Multicore bundle adjustment. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3057–3064, 2011.