# CERES SOLVER: TUTORIAL & REFERENCE

SAMEER AGARWAL
sameeragarwal@google.com

KEIR MIERLE
keir@google.com

November 17, 2012

CONTENTS

## A NOTE TO THE READER

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

Despite our best efforts, this manual remains a work in progress and the source code for Ceres Solver remains the ultimate reference.

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

## VERSION HISTORY

1.4.0

*API Changes*

The new ordering API breaks existing code. Here the common case fixes.

**Before**

```
options.linear_solver_type = ceres::DENSE_SCHUR
options.ordering_type = ceres::SCHUR
```

**After**

```
options.linear_solver_type = ceres::DENSE_SCHUR
```

**Before**

```
options.linear_solver_type = ceres::DENSE_SCHUR;
options.ordering_type = ceres::USER;
for (int i = 0; i < num_points; ++i) {
  options.ordering.push_back(my_points[i])
}
for (int i = 0; i < num_cameras; ++i) {
  options.ordering.push_back(my_cameras[i])
}
options.num_eliminate_blocks = num_points;
```

**After**

```
options.linear_solver_type = ceres::DENSE_SCHUR;
options.ordering = new ceres::ParameterBlockOrdering;
for (int i = 0; i < num_points; ++i) {
  options.linear_solver_ordering->AddElementToGroup(my_points[i], 0);
}
for (int i = 0; i < num_cameras; ++i) {
  options.linear_solver_ordering->AddElementToGroup(my_cameras[i], 1);
}
```

*New Features*

- A new richer, more expressive and consistent API for ordering parameter blocks.
- A non-linear generalization of Ruhe & Wedin's Algorithm II. This allows the user to use variable projection on separable and non-separable non-linear least squares problems. With multithreading, this results in significant improvements to the convergence behavior of the solver at a small increase in run time.
- An image denoising example using fields of experts. (Petter Strandmark)
- Defines for Ceres version and ABI version.
- Higher precision timer code where available. (Petter Strandmark)
- Example Makefile for users of Ceres.
- IterationSummary now informs the user when the step is a non-monotonic step.
- Fewer memory allocations when using `DenseQRSolver`.
- GradientChecker for testing CostFunctions (William Rucklidge)
- Add support for cost functions with 10 parameter blocks in Problem. (Fisher)
- Add support for 10 parameter blocks in AutoDiffCostFunction.

*Bug Fixes*

- static cast to force Eigen::Index to long conversion
- Change LOG(ERROR) to LOG(WARNING) in `schur_complement_solver.cc`.
- Remove verbose logging from `DenseQRSolve`.
- Fix the Android NDK build.
- Better handling of empty and constant Problems.
- Remove an internal header that was leaking into the public API.
- Memory leak in `trust_region_minimizer.cc`
- Schur ordering was operating on the wrong object (Ricardo Martin)

- MSVC fixes (Petter Strandmark)
- Various fixes to `nist.cc` (Markus Moll)
- Fixed a jacobian scaling bug.
- Numerically robust computation of `model_cost_change`.
- Signed comparison compiler warning fixes (Ricardo Martin)
- Various compiler warning fixes all over.
- Inclusion guard fixes (Petter Strandmark)
- Segfault in test code (Sergey Popov)
- Replaced EXPECT/ASSERT_DEATH with the more portable EXPECT_DEATH_IF_SUPPORTED macros.
- Fixed the camera projection model in Ceres' implementation of Snavely's camera model. (Ricardo Martin)

1.3.0

*New Features*

- Android Port (Scott Ettinger also contributed to the port)
- Windows port. (Changchang Wu and Pierre Moulon also contributed to the port)
- New subspace Dogleg Solver. (Markus Moll)
- Trust region algorithm now supports the option of non-monotonic steps.
- New loss functions `ArcTanLossFunction, TolerantLossFunction` and `ComposedLossFunction`. (James Roseborough).
- New `DENSE_NORMAL_CHOLESKY` linear solver, which uses Eigen's LDLT factorization on the normal equations.
- Cached symbolic factorization when using `CXSparse`. (Petter Strandark)
- New example `nist.cc` and data from the NIST non-linear regression test suite. (Thanks to Douglas Bates for suggesting this.)

- The traditional Dogleg solver now uses an elliptical trust region (Markus Moll)
- Support for returning initial and final gradients & Jacobians.
- Gradient computation support in the evaluators, with an eye towards developing first order/gradient based solvers.
- A better way to compute `Solver::Summary::fixed_cost`. (Markus Moll)
- `CMake` support for building documentation, separate examples, installing and uninstalling the library and Gerrit hooks (Arnaud Gelas)
- `SuiteSparse4` support (Markus Moll)
- Support for building Ceres without `TR1` (This leads to slightly slower `DENSE_SCHUR` and `SPARSE_SCHUR` solvers).
- `BALProblem` can now write a problem back to disk.
- `bundle_adjuster` now allows the user to normalize and perturb the problem before solving.
- Solver progress logging to file.
- Added `Program::ToString` and `ParameterBlock::ToString` to help with debugging.
- Ability to build Ceres as a shared library (MacOS and Linux only), associated versioning and build release script changes.
- Portable floating point classification API.

*Bug Fixes*

- Fix how invalid step evaluations are handled.
- Change the slop handling around zero for model cost changes to use relative tolerances rather than absolute tolerances.
- Fix an inadvertant integer to bool conversion. (Petter Strandmark)
- Do not link to `libgomp` when building on windows. (Petter Strandmark)
- Include `gflags.h` in `test_utils.cc`. (Petter Strandmark)
- Use standard random number generation routines. (Petter Strandmark)
- `TrustRegionMinimizer` does not implicitly negate the steps that it takes. (Markus Moll)

- Diagonal scaling allows for equal upper and lower bounds. (Markus Moll)
- TrustRegionStrategy does not misuse LinearSolver:Summary anymore.
- Fix Eigen3 Row/Column Major storage issue. (Lena Gieseke)
- QuaternionToAngleAxis now guarantees an angle in $[-\pi, \pi]$. (Guoxuan Zhang)
- Added a workaround for a compiler bug in the Android NDK to the Schur eliminator.
- The sparse linear algebra library is only logged in Summary::FullReport if it is used.
- Rename the macro `CERES_DONT_HAVE_PROTOCOL_BUFFERS` to `CERES_NO_PROTOCOL_BUFFERS` for consistency.
- Fix how static structure detection for the Schur eliminator logs its results.
- Correct example code in the documentation. (Petter Strandmark)
- Fix `fpclassify.h` to work with the Android NDK and STLport.
- Fix a memory leak in the `levenber_marquardt_strategy_test.cc`
- Fix an early return bug in the Dogleg solver. (Markus Moll)
- Zero initialize Jets.
- Moved `internal/ceres/mock_log.h` to `internal/ceres/gmock/mock-log.h`
- Unified file path handling in tests.
- `data_fitting.cc` includes `gflags`
- Renamed Ceres' Mutex class and associated macros to avoid namespace conflicts.
- Close the BAL problem file after reading it (Markus Moll)
- Fix IsInfinite on Jets.
- Drop alignment requirements for Jets.
- Fixed Jet to integer comparison. (Keith Leung)
- Fix use of uninitialized arrays. (Sebastian Koch & Markus Moll)
- Conditionally compile gflag dependencies.(Casey Goodlett)
- Add `data_fitting.cc` to the examples `CMake` file.

1.2.3

*Bug Fixes*

- `suitesparse_test` is enabled even when `-DSUITESPARSE=OFF`.
- `FixedArray` internal struct did not respect `Eigen` alignment requirements (Koichi Akabe & Stephan Kassemeyer).
- Fixed `quadratic.cc` documentation and code mismatch (Nick Lewycky).

1.2.2

*Bug Fixes*

- Fix constant parameter blocks, and other minor fixes (Markus Moll)
- Fix alignment issues when combining `Jet` and `FixedArray` in automatic differeniation.
- Remove obsolete `build_defs` file.

1.2.1

*New Features*

- Powell's Dogleg solver
- Documentation now has a brief overview of Trust Region methods and how the Levenberg-Marquardt and Dogleg methods work.

*Bug Fixes*

- Destructor for `TrustRegionStrategy` was not virtual (Markus Moll)
- Invalid `DCHECK` in `suitesparse.cc` (Markus Moll)
- Iteration callbacks were not properly invoked (Luis Alberto Zarrabeiti)
- Logging level changes in ConjugateGradientsSolver

- VisibilityBasedPreconditioner setup does not account for skipped camera pairs. This was debugging code.
- Enable SSE support on MacOS
- `system_test` was taking too long and too much memory (Koichi Akabe)

1.2.0

*New Features*

- `CXSparse` support.
- Block oriented fill reducing orderings. This reduces the factorization time for sparse `CHOLMOD` significantly.
- New Trust region loop with support for multiple trust region step strategies. Currently only Levenberg-Marquardt is supported, but this refactoring opens the door for Dog-leg, Stiehaug and others.
- `CMake` file restructuring. Builds in `Release` mode by default, and now has platform specific tuning flags.
- Re-organized documentation. No new content, but better organization.

*Bug Fixes*

- Fixed integer overflow bug in `block_random_access_sparse_matrix.cc`.
- Renamed some macros to prevent name conflicts.
- Fixed incorrent input to `StateUpdatingCallback`.
- Fixes to AutoDiff tests.
- Various internal cleanups.

1.1.1

*Bug Fixes*

- Fix a bug in the handling of constant blocks. (Louis Simard)

- Add an optional lower bound to the Levenberg-Marquardt regularizer to prevent oscillating between well and ill posed linear problems.
- Some internal refactoring and test fixes.

## 2.1 1.1.0

*New Features*

- New iterative linear solver for general sparse problems - `CGNR` and a block Jacobi preconditioner for it.
- Changed the semantics of how `SuiteSparse` dependencies are checked and used. Now `SuiteSparse` is built by default, only if all of its dependencies are present.
- Automatic differentiation now supports dynamic number of residuals.
- Support for writing the linear least squares problems to disk in text format so that they can loaded into `MATLAB`.
- Linear solver results are now checked for nan and infinities.
- Added `.gitignore` file.
- A better more robust build system.

*Bug Fixes*

- Fixed a strict weak ordering bug in the schur ordering.
- Grammar and typos in the documents and code comments.
- Fixed tests which depended on exact equality between floating point values.

## 1.0.0

Initial Release.

# 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 `SuiteSparse`) for general sparse problems and specialized Schur complement based solvers for problems that arise in multi-view geometry [8].

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.

## 3.1 WHAT'S IN A NAME?

While there is some debate as to who invented of the method of Least Squares [20]. 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 [21]. We named our solver after Ceres to celebrate this seminal event in the history of astronomy, statistics and optimization.

## 3.2 CONTRIBUTING TO CERES SOLVER

We welcome contributions to Ceres, whether they are new features, bug fixes or tests. The Ceres mailing list[3] is the best place for all development related discussions. Please consider joining it. If you have ideas on how you would like to contribute to Ceres, it is a good idea to let us know on the mailinglist before you start development. We may have suggestions that will save effort when trying to merge your work into the main branch. 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 [4] and use `git` for version control.

[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://groups.google.com/group/ceres-solver
[4] http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml

## 3.3 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.}
}
```

## 3.4 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.

## LICENSE

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

# BUILDING CERES

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

## 5.1 DEPENDENCIES

Ceres relies on a number of open source libraries, some of which are optional. For details on customizing the build process, please see Section 5.7.

1. `cmake` [1] is the cross-platform build system used by Ceres. We require that you have a relative recent install of `cmake` (version 2.8.0 or better).
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** `gflags`.
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. `CXSparse` [6] is used for sparse matrix analysis, ordering and factorization. While it is similar to `SuiteSparse` in scope, its performance is a bit worse but is a much simpler library to build and does not have any other dependencies. This is an optional dependency.
7. `BLAS` and `LAPACK` are needed by `SuiteSparse`. We recommend either `GotoBlas2` [7] or `ATLAS` [8], both of which ship with `BLAS` and `LAPACK` routines.

[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.cise.ufl.edu/research/sparse/CXSparse/
[7] http://www.tacc.utexas.edu/tacc-projects/gotoblas2
[8] http://math-atlas.sourceforge.net/

8. protobuf [9] 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.

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

## 5.2 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 and CXSparse

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

[9] http://code.google.com/p/protobuf/

This should automatically bring in the necessary `BLAS` and `LAPACK` dependencies. By co-incidence on Ubuntu, this also installs `CXSparse`.

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-1.2.1.tar.gz
mkdir ceres-bin
cd ceres-bin
cmake ../ceres-solver-1.2.1
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]:

```
bin/simple_bundle_adjuster \
  ../ceres-solver-1.2.1/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
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
```

## 5.3 BUILDING ON OS X

On OS X, we recommend using the homebrew [10] 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 and CXSparse

   ```
   brew install suite-sparse
   ```

5. protobuf

   ```
   brew install protobuf
   ```

We are now ready to build and test Ceres.

```
tar zxf ceres-solver-1.2.1.tar.gz
mkdir ceres-bin
cd ceres-bin
cmake ../ceres-solver-1.2.1
make -j3
make test
```

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

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

## 5.4 BUILDING ON WINDOWS WITH VISUAL STUDIO

On Windows, we support building with Visual Studio 2010 or newer. Note that the Windows port is less featureful and less tested than the Linux or Mac OS X versions due to the unavaliability of SuiteSparse and CXSparse. Building is also more involved since there is no automated way to install the dependencies.

1. Make a toplevel directory for deps & build & src somewhere: `ceres/`
2. Get dependencies; unpack them as subdirectories in `ceres/` (`ceres/eigen`, `ceres/glog`, etc)
   - Eigen 3.1 from eigen.tuxfamily.org (needed on Windows; 3.0.x will not work). There is no need to build anything; just unpack the source tarball.
   - Goolge Log. Open up the Visual Studio solution and build it.
   - Goolge Flags. Open up the Visual Studio solution and build it.
3. Unpack the Ceres tarball into `ceres`. For the tarball, you should get a directory inside `ceres` similar to `ceres-solver-1.3.0`. Alternately, checkout Ceres via git to get `ceres-solver.git` inside `ceres`.
4. Install CMake.
5. Make a dir `ceres/ceres-bin` (for an out-of-tree build)
6. Run CMake; select the `ceres-solver-X.Y.Z` or `ceres-solver.git` directory for the CMake file. Then select the `ceres-bin` for the build dir.
7. Try running "Configure". It won't work. It'll show a bunch of options. You'll need to set:
   - `GLOG_INCLUDE`
   - `GLOG_LIB`
   - `GFLAGS_LIB`
   - `GFLAGS_INCLUDE`

   to the appropriate place where you unpacked/built them.
8. You may have to tweak some more settings to generate a MSVC project. After each adjustment, try pressing Configure & Generate until it generates successfully.
9. Open the solution and build it in MSVC

To run the tests, select the RUN_TESTS target and hit "Build RUN_TESTS" from the build menu.

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

Notes:

- The default build is Debug; consider switching it to release mode.
- Currently system_test is not working properly.
- Building Ceres as a DLL is not supported; patches welcome.
- CMake puts the resulting test binaries in ceres-bin/examples/Debug by default.
- The solvers supported on Windows are DENSE_QR, DENSE_SCHUR, CGNR, and ITERATIVE_SCHUR.
- We're looking for someone to work with upstream SuiteSparse to port their build system to something sane like CMake, and get a supported Windows port.

## 5.5 BUILDING ON ANDROID

Download the Android NDK. Run ndk-build from inside the jni directory. Use the libceres.a that gets created.

TODO(keir): Expand this section further.

## 5.6 COMPILER FLAGS TO USE WHEN BUILDING YOUR OWN APPLICATIONS

TBD

## 5.7 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. CXSparse

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

   -DCXSPARSE=OFF.

   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.

4. gflags To build Ceres without gflags, use

   -DGFLAGS=OFF.

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

5. 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.

6. OpenMP

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

   -DOPENMP=OFF.

# Part I

# Tutorial

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. Then,

$$\arg\min_x \frac{1}{2} \sum_{i=1}^{k} \|f_i(x)\|^2. \tag{6.1}$$

is a Non-linear least squares problem [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

$$\arg\min_{m,c} \sum_{i=1}^{k} (y_i - mx_i - c)^2. \tag{6.2}$$

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

$$\arg\min_{m,c} \sum_{i=1}^{k} \left(y_i - e^{mx_i+c}\right)^2. \tag{6.3}$$

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.

[1] Ceres can solve a more general version of this problem, but for pedagogical reasons, we will restrict ourselves to this class of problems for now. See section 11 for a full description of the problems that Ceres can solve

## HELLO WORLD!

To get started, let us consider the problem of finding the minimum of the function

$$\frac{1}{2}(10-x)^2. \tag{7.1}$$

This is a trivial problem, whose minimum is easy to see is located at $x = 10$, but it is a good place to start to illustrate the basics of solving a problem with Ceres[1].

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 subclass of `CostFunction`. It is responsible for computing the value of the residual function and its derivative (also known as the Jacobian) with respect to $x$.

```
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 {
    const double x = parameters[0][0];
    residuals[0] = 10 - x;  // f(x) = 10 - x
    // Compute the Jacobian if asked for.
    if (jacobians != NULL && jacobians[0] != NULL) {
      jacobians[0][0] = -1;
    }
    return true;
  }
};
```

`SimpleCostFunction` is provided with an input array of parameters, an output array for residuals and an optional output array for Jacobians. In our example, there is just one parameter and one residual

[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`

and this is known at compile time, therefore we can save some code and instead of inheriting from `CostFunction`, we can instaed inherit from the templated `SizedCostFunction` class.

The `jacobians` array is optional, `Evaluate` is expected to check 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.

Once we have a way of computing the residual vector, it is now time to construct a Non-linear least squares problem using it and have Ceres solve it.

```
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);

  // Run the solver!
  Solver::Options options;
  options.max_num_iterations = 10;
  options.linear_solver_type = ceres::DENSE_QR;
  options.minimizer_progress_to_stdout = true;
  Solver::Summary summary;
  Solve(options, &problem, &summary);
  std::cout << summary.BriefReport() << "\n";
  std::cout << "x : 5.0 -> " << x << "\n";
  return 0;
}
```

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 [2]. 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 parameter settings for Ceres.

[2]Actually the solver ran for three iterations, and it was by looking at the value returned by the linear solver in the third iteration, it observed that the update to the parameter block was too small and declared convergence. Ceres only prints out the display at the end of an iteration, and terminates as soon as it detects convergence, which is why you only see two iterations here and not three.

## POWELL'S FUNCTION

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

$$f_1(x) = x_1 + 10 * x_2 \tag{8.1}$$
$$f_2(x) = \sqrt{5} * (x_3 - x_4) \tag{8.2}$$
$$f_3(x) = (x_2 - 2 * x_3)^2 \tag{8.3}$$
$$f_4(x) = \sqrt{10} * (x_1 - x_4)^2 \tag{8.4}$$
$$F(x) = [f_1(x),\ f_2(x),\ f_3(x),\ f_4(x)] \tag{8.5}$$

$F(x)$ is a function of four parameters, and has four residuals. Now, one way to solve this problem would be to define four `CostFunction` objects that compute the residual and Jacobians. *e.g.* the following code 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[0][3];
    // f4 = sqrt(10) * (x1 - x4)^2
    residuals[0] = sqrt(10.0) * (x1 - x4) * (x1 - x4)
    if (jacobians != NULL) {
      jacobians[0][0] = 2.0 * sqrt(10.0) * (x1 - x4);   // ∂x1 f4(x)
      jacobians[0][1] = 0.0;                            // ∂x2 f4(x)
      jacobians[0][2] = 0.0;                            // ∂x3 f4(x)
      jacobians[0][3] = -2.0 * sqrt(10.0) * (x1 - x4);  // ∂x4 f4(x)
    }
    return true;
  }
};
```

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

## 8.1 AUTOMATIC DIFFERENTIATION

With its automatic differentiation support, Ceres allows you to define templated objects/functors 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 = sqrt(10) * (x1 - x4)^2
    residual[0] = T(sqrt(10.0)) * (x1[0] - x4[0]) * (x1[0] - x4[0]);
    return true;
  }
};
```

The important thing to note here is that `operator()` is a templated method, which assumes that all its inputs and outputs are of some type `T`. The reason for using templates here is because Ceres will call `F4::operator<T>()`, with `T=double` when just the residual is needed, and with a special type $T$ = `Jet` when the Jacobians are needed.

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 [1]

```
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.
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);
```

[1] The full source code for this example can be found in `examples/powell.cc`.

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 $4 \times 10^{-12}$.

## 8.2 NUMERIC DIFFERENTIATION

If a templated implementation is not possible then a `NumericDiffCostFunction` object can be used. The user defines a `CostFunction` object whose `Evaluate` method is only computes the residuals. A wrapper object `NumericDiffCostFunction` then uses it to compute the residuals and the Jacobian using finite differencing. `examples/quadratic_numeric_diff.cc` shows a numerically differentiated implementation of `examples/quadratic.cc`.

We recommend that if 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.

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[1]. Let us fit some data to the curve

$$y = e^{mx+c}. \tag{9.1}$$

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 {
    residual[0] = T(y_) - exp(m[0] * T(x_) + c[0]);  // y - e^(mx + c)
    return true;
  }

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

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.

[1]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 $\sigma = 0.2$.

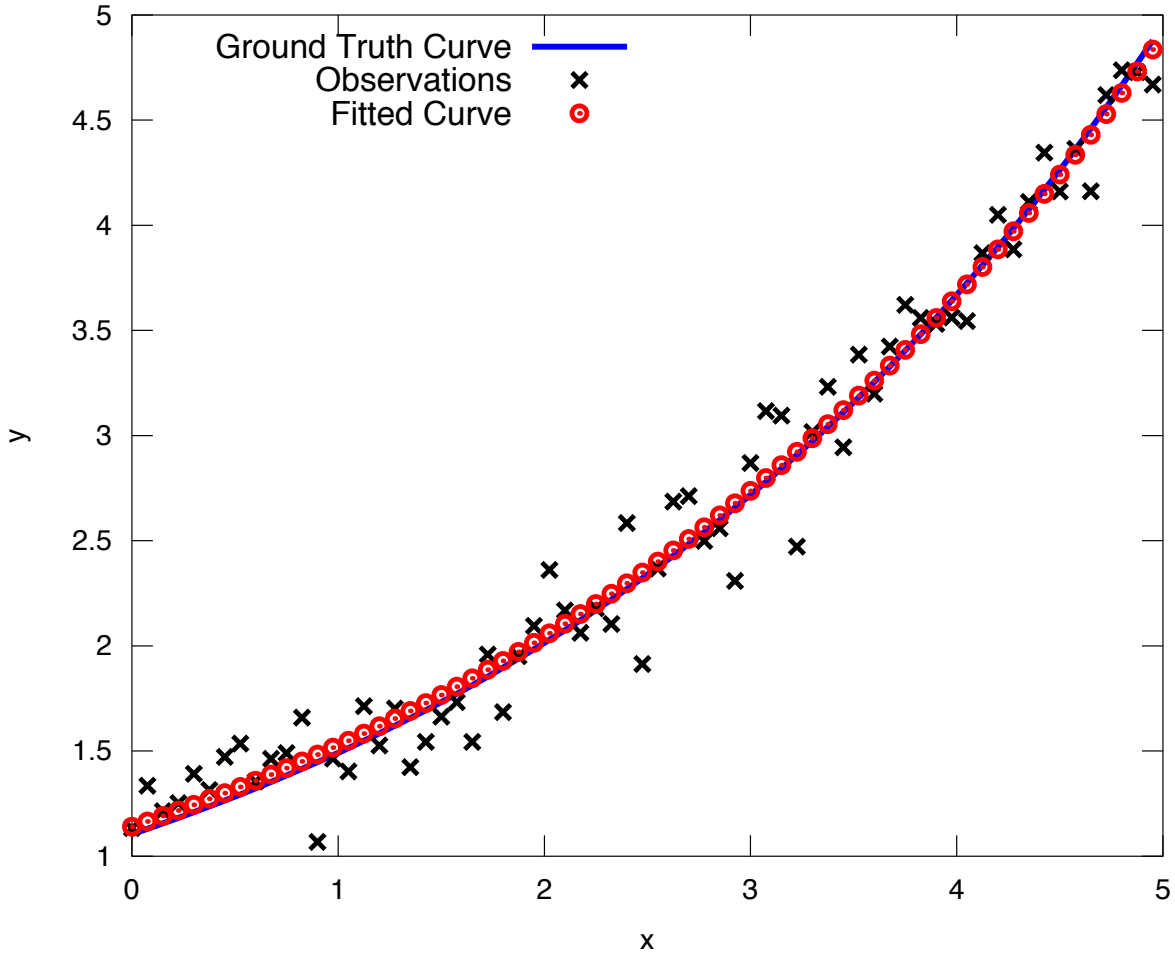
```
double m = 0.0;
double c = 0.0;

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);
}
```

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 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 expected. 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 9 illustrates the fit.




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 $\sigma = 0.2$.

## BUNDLE ADJUSTMENT

One of the main reasons for writing Ceres was our need to solve large scale bundle adjustment problems [8, 23].

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 [1].

The first step as usual is to define a templated functor that computes the reprojection error/residual. The structure of the functor is similar to the `ExponentialResidual`, in that there is an instance of this object responsible for each image observation.

Each residual in a BAL problem depends on a three dimensional point and a nine parameter camera. The nine parameters defining the camera can are: Three for rotation as a Rodriquez axis-angle vector, three for translation, one for focal length and two for radial distortion. The details of this camera model can be found on Noah Snavely's Bundler homepage [2] and the BAL homepage [3].

[1] The code for this example can be found in `examples/simple_bundle_adjuster.cc`.
[2] http://phototour.cs.washington.edu/bundler/
[3] http://grail.cs.washington.edu/projects/bal/

```
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.
    T xp = - p[0] / p[2];
    T yp = - 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.
    const T& focal = camera[6];
    T predicted_x = focal * distortion * xp;
    T predicted_y = focal * 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;
};
```

Note that 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. The function `AngleAxisRotatePoint` and other functions for manipulating rotations can be found in `include/ceres/rotation.h`.

Given this functor, the bundle adjustment problem can be constructed as follows:

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

Again note that that the problem construction for bundle adjustment is very similar to the curve fitting example.

One way to solve this problem is 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`.

```
ceres::Solver::Options options;
options.linear_solver_type = ceres::DENSE_SCHUR;
options.minimizer_progress_to_stdout = true;
ceres::Solver::Summary summary;
ceres::Solve(options, &problem, &summary);
std::cout << summary.FullReport() << "\n";
```

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

# Part II

# Reference

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

$$\frac{1}{2}\sum_{i=1}\rho_i\left(\left\|f_i\left(x_{i_1},\ldots,x_{i_k}\right)\right\|^2\right). \tag{11.1}$$

Where $f_i(\cdot)$ is a cost function that depends on the parameter blocks $\left[x_{i_1},\ldots,x_{i_k}\right]$ and $\rho_i$ 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 $\rho_i\left(\left\|f_i\left(x_{i_1},\ldots,x_{i_k}\right)\right\|^2\right)$ 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.

Solving problems using Ceres consists of two steps.

1. Modeling Define parameter blocks and residual blocks and build a `Problem` object containing them.
2. Solving Configure and run the solver.

These two steps are mostly independent of each other. This is by design. Modeling the optimization problem should not depend on how the solver and the user should be able to switch between various solver settings and strategies without changing the way the problem is modeled. In the next two chapters we will consider each of these steps in detail.

## 12.1 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 $\left[x_{i_1},\ldots,x_{i_k}\right]$, compute the vector $f_i\left(x_{i_1},\ldots,x_{i_k}\right)$ and the matrices

$$J_{ij} = \frac{\partial}{\partial x_j} f_i\left(x_{i_1},\ldots,x_{i_k}\right), \quad \forall j \in \{i_1,\ldots,i_k\} \tag{12.1}$$

```
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. The Jacobian matrices are in the same order as parameter_block_sizes_. 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.,

$$\texttt{jacobians[i][r * parameter\_block\_size\_[i] + c]} = \frac{\partial \texttt{residual[r]}}{\partial \texttt{parameters[i][c]}} \tag{12.2}$$

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 $i^{\text{th}}$ parameter block must not be returned, this is the case when the a parameter block is marked constant.

## 12.2 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.

## 12.3 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.

*Theory & Implementation*

TBD

## 12.4 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.

## 12.5 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

$$\texttt{out} = \left[\rho(s), \quad \rho'(s), \quad \rho''(s)\right] \tag{12.3}$$

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 $\rho$ satisfy:

$$\rho(0) = 0 \tag{12.4}$$
$$\rho'(0) = 1 \tag{12.5}$$
$$\rho'(s) < 1 \text{ in the outlier region} \tag{12.6}$$
$$\rho''(s) < 0 \text{ in the outlier region} \tag{12.7}$$

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 12.5 illustrates their shape graphically.

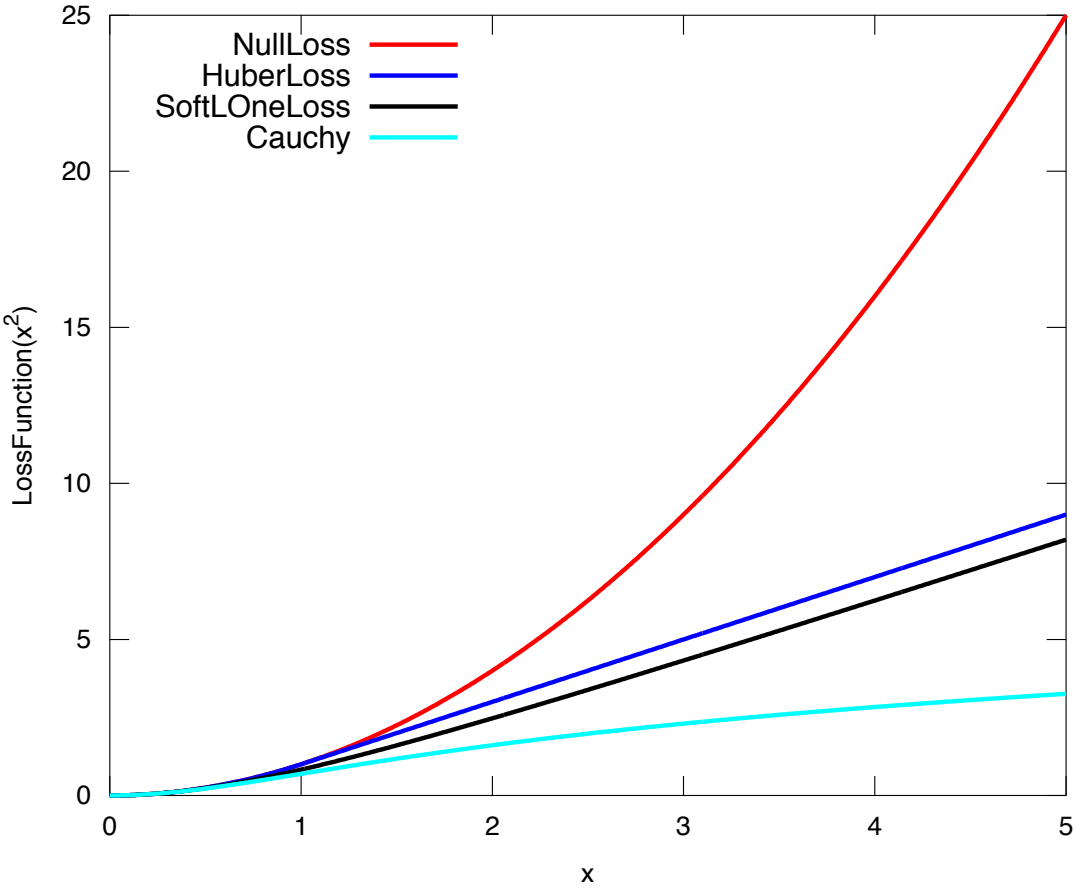
$$\rho(s) = s \tag{NullLoss}$$
$$\rho(s) = \begin{cases} s & s \le 1 \\ 2\sqrt{s} - 1 & s > 1 \end{cases} \tag{HuberLoss}$$
$$\rho(s) = 2(\sqrt{1+s} - 1) \tag{SoftLOneLoss}$$
$$\rho(s) = \log(1+s) \tag{CauchyLoss}$$

Ceres includes a number of other loss functions, the descriptions and documentation for which can be found in `loss_function.h`.

*Theory & Implementation*

Let us consider a problem with a single problem and a single parameter block.

$$\min_x \frac{1}{2}\rho(f^2(x)) \tag{12.8}$$




Shape of the various common loss functions.

Then, the robustified gradient and the Gauss-Newton Hessian are

$$g(x) = \rho' J^\top(x) f(x) \tag{12.9}$$

$$H(x) = J^\top(x)\left(\rho' + 2\rho'' f(x) f^\top(x)\right) J(x) \tag{12.10}$$

where the terms involving the second derivatives of $f(x)$ have been ignored. Note that $H(x)$ is indefinite if $\rho'' f(x)^\top f(x) + \frac{1}{2}\rho' < 0$. If this is not the case, then its possible to re-weight the residual and the Jacobian matrix such that the corresponding linear least squares problem for the robustified Gauss-Newton step.

Let $\alpha$ be a root of

$$\frac{1}{2}\alpha^2 - \alpha - \frac{\rho''}{\rho'}\|f(x)\|^2 = 0. \tag{12.11}$$

Then, define the rescaled residual and Jacobian as

$$\tilde{f}(x) = \frac{\sqrt{\rho'}}{1-\alpha} f(x) \tag{12.12}$$

$$\tilde{J}(x) = \sqrt{\rho'}\left(1 - \alpha\frac{f(x)f^\top(x)}{\|f(x)\|^2}\right) J(x) \tag{12.13}$$

In the case $2\rho''\|f(x)\|^2 + \rho' \lesssim 0$, we limit $\alpha \leq 1-\epsilon$ for some small $\epsilon$. For more details see Triggs et al [23].

With this simple rescaling, one can use any Jacobian based non-linear least squares algorithm to robustifed non-linear least squares problems.

## 12.6 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 $\Delta x$ 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

$$x' = \boxplus(x, \Delta x), \tag{12.14}$$

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

$$\boxplus(x, 0) = x, \quad \forall x. \tag{12.15}$$

Instances of `LocalParameterization` implement the $\boxplus$ operation and its derivative with respect to $\Delta x$ 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 $\Delta x$ lives in. `Plus` implements $\boxplus(x, \Delta x)$ and `ComputeJacobian` computes the Jacobian matrix

$$J = \frac{\partial}{\partial \Delta x} \boxplus(x, \Delta x)\Big|_{\Delta x = 0} \tag{12.16}$$

in row major form.

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

$$\boxplus(x, \Delta x) = x + \Delta x \tag{12.17}$$

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

$$\boxplus(x, \Delta x) = x + \begin{bmatrix} 0 \\ 1 \end{bmatrix} \Delta x \tag{12.18}$$

`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 $\Delta x$ be a 3 dimensional vector and define $\boxplus$ to be

$$\boxplus(x, \Delta x) = \left[\cos(|\Delta x|), \frac{\sin(|\Delta x|)}{|\Delta x|} \Delta x\right] * x \tag{12.19}$$

The multiplication between the two 4-vectors on the right hand side is the standard quaternion product. `QuaternionParameterization` is an implementation of (12.19).

## 12.7 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 (11.1). 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.

Effective use of Ceres requires some familiarity with the basic components of a nonlinear least squares solver, so before we describe how to configure the solver, we will begin by taking a brief look at how some of the core optimization algorithms in Ceres work and the various linear solvers and preconditioners that power it.

## 13.1 TRUST REGION METHODS

Let $x \in \mathbb{R}^n$ be an $n$-dimensional vector of variables, and $F(x) = [f_1(x), \ldots, f_m(x)]^\top$ be a $m$-dimensional function of $x$. We are interested in solving the following optimization problem [1],

$$\arg\min_x \frac{1}{2}\|F(x)\|^2 . \tag{13.1}$$

Here, 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 (13.1) 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 [17]. At each iteration, the approximation is solved to determine a correction $\Delta x$ 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:

$$\min_{\Delta x} \frac{1}{2}\|J(x)\Delta x + F(x)\|^2 \tag{13.2}$$

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 $\Delta x$. And this is where the idea of a trust-region comes in. Algorithm 13.1 describes the basic trust-region loop for non-linear least squares problems.

Here, $\mu$ is the trust region radius, $D(x)$ is some matrix used to define a metric on the domain of $F(x)$ and $\rho$ measures the quality of the step $\Delta x$, i.e., how well did the linear model predict the decrease in the value of the non-linear objective. The idea is to increase or decrease the radius of the trust region depending on how well the linearization predicts the behavior of the non-linear objective, which in turn is reflected in the value of $\rho$.

[1] At the level of the non-linear solver, the block and residual structure is not relevant, therefore our discussion here is in terms of an optimization problem defined over a state vector of size $n$.

The basic trust-region algorithm.

**Require:** Initial point $x$ and a trust region radius $\mu$.
  **loop**
    Solve $\arg\min_{\Delta x} \frac{1}{2}\|J(x)\Delta x + F(x)\|^2$ s.t. $\|D(x)\Delta x\|^2 \le \mu$
    $\rho = \frac{\|F(x+\Delta x)\|^2 - \|F(x)\|^2}{\|J(x)\Delta x + F(x)\|^2 - \|F(x)\|^2}$
    **if** $\rho > \epsilon$ **then**
      $x = x + \Delta x$
    **end if**
    **if** $\rho > \eta_1$ **then**
      $\rho = 2 * \rho$
    **else**
      **if** $\rho < \eta_2$ **then**
        $\rho = 0.5 * \rho$
      **end if**
    **end if**
  **end loop**

The key computational step in a trust-region algorithm is the solution of the constrained optimization problem

$$\arg\min_{\Delta x} \frac{1}{2}\|J(x)\Delta x + F(x)\|^2 \tag{13.3}$$

$$\text{such that} \quad \|D(x)\Delta x\|^2 \le \mu \tag{13.4}$$

There are a number of different ways of solving this problem, each giving rise to a different concrete trust-region algorithm. Currently Ceres, implements two trust-region algorithms - Levenberg-Marquardt and Dogleg.

*Levenberg-Marquardt*

The Levenberg-Marquardt algorithm [10, 14] is the most popular algorithm for solving non-linear least squares problems. It was also the first trust region algorithm to be developed [10, 14]. Ceres implements an exact step [12] and an inexact step variant of the Levenberg-Marquardt algorithm [16, 25].

It can be shown, that the solution to (13.4) can be obtained by solving an unconstrained optimization

of the form

$$\arg\min_{\Delta x} \frac{1}{2}\|J(x)\Delta x + F(x)\|^2 + \lambda\|D(x)\Delta x\|^2 \tag{13.5}$$

Where, $\lambda$ is a Lagrange multiplier that is inverse related to $\mu$. In Ceres, we solve for

$$\arg\min_{\Delta x} \frac{1}{2}\|J(x)\Delta x + F(x)\|^2 + \frac{1}{\mu}\|D(x)\Delta x\|^2 \tag{13.6}$$

The matrix $D(x)$ is a non-negative diagonal matrix, typically the square root of the diagonal of the matrix $J(x)^\top J(x)$.

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.

$$\min_{\Delta x} \frac{1}{2}\|J(x)\Delta x + f(x)\|^2. \tag{13.7}$$

For all but the smallest problems the solution of (13.7) in each iteration of the Levenberg-Marquardt algorithm is the dominant computational cost in Ceres. Ceres provides a number of different options for solving (13.7). There are two major classes of methods - factorization and iterative.

The factorization methods are based on computing an exact solution of (13.6) using a Cholesky or a QR factorization and lead to an exact step Levenberg-Marquardt algorithm. But it is not clear if an exact solution of (13.6) is necessary at each step of the LM algorithm to solve (13.1). In fact, we have already seen evidence that this may not be the case, as (13.6) is itself a regularized version of (13.2). 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 [17].

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 [17]. Second, a termination rule for the iterative solver. A typical termination rule is of the form

$$\|H(x)\Delta x + g(x)\| \leq \eta_k \|g(x)\|. \tag{13.8}$$

Here, $k$ indicates the Levenberg-Marquardt iteration number and $0 < \eta_k < 1$ is known as the forcing sequence. Wright & Holt [25] prove that a truncated Levenberg-Marquardt algorithm that uses an inexact Newton step based on (13.8) converges for any sequence $\eta_k \leq \eta_0 < 1$ and the rate of convergence depends on the choice of the forcing sequence $\eta_k$.

Ceres supports both exact and inexact step solution strategies. When the user chooses a factorization based linear solver, the exact step Levenberg-Marquardt algorithm is used. When the user chooses an iterative linear solver, the inexact step Levenberg-Marquardt algorithm is used.

*Dogleg*

Another strategy for solving the trust region problem (13.4) was introduced by M. J. D. Powell. The key idea there is to compute two vectors

$$\Delta x^{\text{Gauss-Newton}} = \arg\min_{\Delta x} \frac{1}{2}\|J(x)\Delta x + f(x)\|^2. \tag{13.9}$$

$$\Delta x^{\text{Cauchy}} = -\frac{\|g(x)\|^2}{\|J(x)g(x)\|^2} g(x). \tag{13.10}$$

Note that the vector $\Delta x^{\text{Gauss-Newton}}$ is the solution to (13.2) and $\Delta x^{\text{Cauchy}}$ is the vector that minimizes the linear approximation if we restrict ourselves to moving along the direction of the gradient. Dogleg methods finds a vector $\Delta x$ defined by $\Delta x^{\text{Gauss-Newton}}$ and $\Delta x^{\text{Cauchy}}$ that solves the trust region problem. Ceres supports two variants.

`TRADITIONAL_DOGLEG` as described by Powell, constructs two line segments using the Gauss-Newton and Cauchy vectors and finds the point farthest along this line shaped like a dogleg (hence the name) that is contained in the trust-region. For more details on the exact reasoning and computations, please see Madsen et al [12].

`SUBSPACE_DOGLEG` is a more sophisticated method that considers the entire two dimensional subspace spanned by these two vectors and finds the point that minimizes the trust region problem in this subspace[4].

The key advantage of the Dogleg over Levenberg Marquardt is that if the step computation for a particular choice of $\mu$ does not result in sufficient decrease in the value of the objective function, Levenberg-Marquardt solves the linear approximation from scratch with a smaller value of $\mu$. Dogleg on the other hand, only needs to compute the interpolation between the Gauss-Newton and the Cauchy vectors, as neither of them depend on the value of $\mu$.

The Dogleg method can only be used with the exact factorization based linear solvers.

*Inner Iterations*

Some non-linear least squares problems have additional structure in the way the parameter blocks interact that it is beneficial to modify the way the trust region step is computed. e.g., consider the following regression problem

$$y = a_1 e^{b_1 x} + a_2 e^{b_3 x^2 + c_1} \tag{13.11}$$

Given a set of pairs $\{(x_i, y_i)\}$, the user wishes to estimate $a_1, a_2, b_1, b_2$, and $c_1$.

Notice that the expression on the left is linear in $a_1$ and $a_2$, and given any value for $b_1$, $b_2$ and $c_1$, it is possible to use linear regression to estimate the optimal values of $a_1$ and $a_2$. It's possible to analytically eliminate the variables $a_1$ and $a_2$ from the problem entirely. Problems like these are known as separable least squares problem and the most famous algorithm for solving them is the Variable Projection algorithm invented by Golub & Pereyra [7].

Similar structure can be found in the matrix factorization with missing data problem. There the corresponding algorithm is known as Wiberg's algorithm [24].

Ruhe & Wedin present an analysis of various algorithms for solving separable non-linear least squares problems and refer to *Variable Projection* as Algorithm I in their paper [18].

Implementing Variable Projection is tedious and expensive. Ruhe & Wedin present a simpler algorithm with comparable convergence properties, which they call Algorithm II. Algorithm II performs an additional optimization step to estimate $a_1$ and $a_2$ exactly after computing a successful Newton step.

This idea can be generalized to cases where the residual is not linear in $a_1$ and $a_2$, i.e.,

$$y = f_1(a_1, e^{b_1 x}) + f_2(a_2, e^{b_3 x^2 + c_1}) \tag{13.12}$$

In this case, we solve for the trust region step for the full problem, and then use it as the starting point to further optimize just $a_1$ and $a_2$. For the linear case, this amounts to doing a single linear least squares solve. For non-linear problems, any method for solving the $a_1$ and $a_2$ optimization problems will do. The only constraint on $a_1$ and $a_2$ (if they are two different parameter block) is that they do not co-occur in a residual block.

This idea can be further generalized, by not just optimizing $(a_1, a_2)$, but decomposing the graph corresponding to the Hessian matrix's sparsity structure into a collection of non-overlapping independent sets and optimizing each of them.

Setting `Solver::Options::use_inner_iterations` to true enables the use of this non-linear generalization of Ruhe & Wedin's Algorithm II. This version of Ceres has a higher iteration complexity, but also displays better convergence behavior per iteration.

Setting `Solver::Options::num_threads` to the maximum number possible is highly recommended.

*Non-monotonic Steps*

Note that the basic trust-region algorithm described in Algorithm 13.1 is a descent algorithm in that they only accepts a point if it strictly reduces the value of the objective function.

Relaxing this requirement allows the algorithm to be more efficient in the long term at the cost of some local increase in the value of the objective function.

This is because allowing for non-decreasing objective function values in a princpled manner allows the algorithm to "jump over boulders" as the method is not restricted to move into narrow valleys while preserving its convergence properties.

Setting `Solver::Options::use_nonmonotonic_steps` to `true` enables the non-monotonic trust region algorithm as described by Conn, Gould & Toint in [6].

Even though the value of the objective function may be larger than the minimum value encountered over the course of the optimization, the final parameters returned to the user are the ones corresponding to the minimum cost over all iterations.

The option to take non-monotonic is available for all trust region strategies.

## 13.2 LINEARSOLVER

Recall that in both of the trust-region methods described above, the key computational cost is the solution of a linear least squares problem of the form

$$\min_{\Delta x} \frac{1}{2}\|J(x)\Delta x + f(x)\|^2. \tag{13.13}$$

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 (13.13) is equivalent to solving the *normal equations*

$$H\Delta x = g \tag{13.14}$$

Ceres provides a number of different options for solving (13.14).

### *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 [22]. Then it can be shown that the solution to (13.14) is given by

$$\Delta x^* = -R^{-1}Q^\top f \tag{13.15}$$

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

*DENSE_NORMAL_CHOLESKY & 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 (13.14) is given by

$$\Delta x^* = R^{-1} R^{-\top} g. \tag{13.16}$$

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.

`DENSE_NORMAL_CHOLESKY` as the name implies performs a dense Cholesky factorization of the normal equations. Ceres uses `Eigen`'s dense LDLT factorization routines.

`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. Ceres uses the sparse Cholesky factorization routines in Professor Tim Davis' `SuiteSparse` or `CXSparse` packages [5].

*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 (13.14) 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

$$H = \begin{bmatrix} B & E \\ E^\top & C \end{bmatrix}, \tag{13.17}$$

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 (13.14) as the block structured linear system

$$\begin{bmatrix} B & E \\ E^\top & C \end{bmatrix} \begin{bmatrix} \Delta y \\ \Delta z \end{bmatrix} = \begin{bmatrix} v \\ w \end{bmatrix}, \tag{13.18}$$

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 $\Delta z$ by observing that $\Delta z = C^{-1}(w - E^\top \Delta y)$, giving us

$$\left[B - EC^{-1}E^\top\right]\Delta y = v - EC^{-1}w \ . \tag{13.19}$$

The matrix

$$S = B - EC^{-1}E^\top \ , \tag{13.20}$$

is the Schur complement of $C$ in $H$. It is also known as the *reduced camera matrix*, because the only variables participating in (13.19) 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, (13.18) can be solved by first forming $S$, solving for $\Delta y$, and then back-substituting $\Delta y$ to obtain the value of $\Delta z$. 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 (13.19). For almost all problems, the number of cameras is much smaller than the number of points, $p \ll q$, thus solving (13.19) is significantly cheaper than solving (13.18). This is the *Schur complement trick* [3].

This still leaves open the question of solving (13.19). The method of choice for solving symmetric positive definite systems exactly is via the Cholesky factorization [22] 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 [22]. 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 [5]. 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.

*`CGNR`*

For general sparse problems, if the problem is too large for `CHOLMOD` or a sparse linear algebra library 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

$$Hx = J^\top Jx = J^\top (Jx) \tag{13.21}$$

When the user chooses `ITERATIVE_SCHUR` as the linear solver, Ceres automatically switches from the exact step algorithm to an inexact step algorithm.

*`ITERATIVE_SCHUR`*

Another option for bundle adjustment problems 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.

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

$$\begin{aligned} x_1 &= E^\top x \\ x_2 &= C^{-1} x_1 \\ x_3 &= E x_2 \\ x_4 &= B x \\ Sx &= x_4 - x_3 \ . \end{aligned} \tag{13.22}$$

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 $H$, (13.22) can be implemented using just the columns of $J$.

Equation (13.22) 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 Sub-structuring [15, 19].

## 13.3 PRECONDITIONER

The convergence rate of Conjugate Gradients for solving (13.14) depends on the distribution of eigenvalues of $H$ [19]. 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 (13.14) results in extremely poor performance.

The solution to this problem is to replace (13.14) 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 $\kappa(M^{-1}A)$.

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.* , $M = \text{diag}(A)$, which for block structured matrices like $H$ can be generalized to the block Jacobi preconditioner.

For `ITERATIVE_SCHUR` there are two obvious choices for block diagonal preconditioners for $S$. The block diagonal of the matrix $B$ [13] 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.

For bundle adjustment problems 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 [9]. 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.

## 13.4 ORDERING

The order in which variables are eliminated in a linear solver can have a significant of impact on the efficiency and accuracy of the method. For example when doing sparse Cholesky factorization, there are matrices for which a good ordering will give a Cholesky factor with O(n) storage, where as a bad ordering will result in an completely dense factor.

Ceres allows the user to provide varying amounts of hints to the solver about the variable elimination ordering to use. This can range from no hints, where the solver is free to decide the best ordering

based on the user's choices like the linear solver being used, to an exact order in which the variables should be eliminated, and a variety of possibilities in between.

Instances of the `ParameterBlockOrdering` class are used to communicate this information to Ceres.

Formally an ordering is an ordered partitioning of the parameter blocks. Each parameter block belongs to exactly one group, and each group has a unique integer associated with it, that determines its order in the set of groups. We call these groups *elimination groups*.

Given such an ordering, Ceres ensures that the parameter blocks in the lowest numbered elimination group are eliminated first, and then the parameter blocks in the next lowest numbered elimination group and so on. Within each elimination group, Ceres is free to order the parameter blocks as it chooses. e.g. Consider the linear system

$$x + y = 3 \tag{13.23}$$
$$2x + 3y = 7 \tag{13.24}$$

There are two ways in which it can be solved. First eliminating $x$ from the two equations, solving for y and then back substituting for $x$, or first eliminating $y$, solving for $x$ and back substituting for $y$. The user can construct three orderings here.

1. $\{0 : x\}, \{1 : y\}$: Eliminate $x$ first.
2. $\{0 : y\}, \{1 : x\}$: Eliminate $y$ first.
3. $\{0 : x, y\}$: Solver gets to decide the elimination order.

Thus, to have Ceres determine the ordering automatically using heuristics, put all the variables in the same elimination group. The identity of the group does not matter. This is the same as not specifying an ordering at all. To control the ordering for every variable, create an elimination group per variable, ordering them in the desired order.

If the user is using one of the Schur solvers (`DENSE_SCHUR`, `SPARSE_SCHUR`, `ITERATIVE_SCHUR`) and chooses to specify an ordering, it must have one important property. The lowest numbered elimination group must form an independent set in the graph corresponding to the Hessian, or in other words, no two parameter blocks in in the first elimination group should co-occur in the same residual block. For the best performance, this elimination group should be as large as possible. For standard bundle adjustment problems, this corresponds to the first elimination group containing all the 3d points, and the second containing the all the cameras parameter blocks.

If the user leaves the choice to Ceres, then the solver uses an approximate maximum independent set algorithm to identify the first elimination group [11] .

## 13.5 SOLVER::OPTIONS

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

1. trust_region_strategy_type (LEVENBERG_MARQUARDT) The trust region step computation algorithm used by Ceres. Currently LEVENBERG_MARQUARDT and DOGLEG are the two valid choices.
2. dogleg_type (TRADITIONAL_DOGLEG) Ceres supports two different dogleg strategies. TRADITIONAL_DOGLEG method by Powell and the SUBSPACE_DOGLEG method described by Byrd et al. [4]. See Section 13.1 for more details.
3. use_nonmonotoic_steps (false) Relax the requirement that the trust-region algorithm take strictly decreasing steps. See Section 13.1 for more details.
4. max_consecutive_nonmonotonic_steps (5) The window size used by the step selection algorithm to accept non-monotonic steps.
5. max_num_iterations (50) Maximum number of iterations for Levenberg-Marquardt.
6. max_solver_time_in_seconds ($10^9$) Maximum amount of time for which the solver should run.
7. num_threads (1) Number of threads used by Ceres to evaluate the Jacobian.
8. initial_trust_region_radius ($10^4$) The size of the initial trust region. When the LEVENBERG_MARQUARDT strategy is used, the reciprocal of this number is the initial regularization parameter.
9. max_trust_region_radius ($10^{16}$) The trust region radius is not allowed to grow beyond this value.
10. min_trust_region_radius ($10^{-32}$) The solver terminates, when the trust region becomes smaller than this value.
11. min_relative_decrease ($10^{-3}$) Lower threshold for relative decrease before a Levenberg-Marquardt step is acceped.
12. lm_min_diagonal ($10^6$) The LEVENBERG_MARQUARDT strategy, uses a diagonal matrix to regularize the the trust region step. This is the lower bound on the values of this diagonal matrix.

13. `lm_max_diagonal` ($10^{32}$) The `LEVENBERG_MARQUARDT` strategy, uses a diagonal matrix to regularize the the trust region step. This is the upper bound on the values of this diagonal matrix.

14. `max_num_consecutive_invalid_steps` (5) The step returned by a trust region strategy can sometimes be numerically invalid, usually because of conditioning issues. Instead of crashing or stopping the optimization, the optimizer can go ahead and try solving with a smaller trust region/better conditioned problem. This parameter sets the number of consecutive retries before the minimizer gives up.

15. `function_tolerance` ($10^{-6}$) Solver terminates if

$$\frac{|\Delta \text{cost}|}{\text{cost}} < \texttt{function\_tolerance} \tag{13.25}$$

where, $\Delta$cost is the change in objective function value (up or down) in the current iteration of Levenberg-Marquardt.

16. `Solver::Options::gradient_tolerance` Solver terminates if

$$\frac{\|g(x)\|_\infty}{\|g(x_0)\|_\infty} < \texttt{gradient\_tolerance} \tag{13.26}$$

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

17. `parameter_tolerance` ($10^{-8}$) Solver terminates if

$$\frac{\|\Delta x\|}{\|x\| + \texttt{parameter\_tolerance}} < \texttt{parameter\_tolerance} \tag{13.27}$$

where $\Delta x$ is the step computed by the linear solver in the current iteration of Levenberg-Marquardt.

18. `linear_solver_type` (`SPARSE_NORMAL_CHOLESKY`)

19. `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.

20. `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`.

21. `sparse_linear_algebra_library` (`SUITE_SPARSE`) Ceres supports the use of two sparse linear algebra libraries, `SuiteSparse`, which is enabled by setting this parameter to `SUITE_SPARSE` and `CXSparse`, which can be selected by setting this parameter to `CX_SPARSE`. `SuiteSparse`

is a sophisticated and complex sparse linear algebra library and should be used in general. If your needs/platforms prevent you from using `SuiteSparse`, consider using `CXSparse`, which is a much smaller, easier to build library. As can be expected, its performance on large problems is not comparable to that of `SuiteSparse`.

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

23. `use_inner_iterations` (`false`) Use a non-linear version of a simplified variable projection algorithm. Essentially this amounts to doing a further optimization on each Newton/Trust region step using a coordinate descent algorithm. For more details, see the discussion in 13.1

24. `inner_iteration_ordering` (`NULL`) If `Solver::Options::inner_iterations` is true, then the user has two choices.

    a) Let the solver heuristically decide which parameter blocks to optimize in each inner iteration. To do this, set `inner_iteration_ordering` to `NULL`.

    b) Specify a collection of of ordered independent sets. The lower numbered groups are optimized before the higher number groups during the inner optimization phase. Each group must be an independent set.

25. `linear_solver_ordering` (`NULL`) An instance of the ordering object informs the solver about the desired order in which parameter blocks should be eliminated by the linear solvers. See section 13.4 for more details.

    If `NULL`, the solver is free to choose an ordering that it thinks is best. Note: currently, this option only has an effect on the Schur type solvers, support for the `SPARSE_NORMAL_CHOLESKY` solver is forth coming.

26. `use_block_amd` (`true`) By virtue of the modeling layer in Ceres being block oriented, all the matrices used by Ceres are also block oriented. When doing sparse direct factorization of these matrices, the fill-reducing ordering algorithms can either be run on the block or the scalar form of these matrices. Running it on the block form exposes more of the super-nodal structure of the matrix to the Cholesky factorization routines. This leads to substantial gains in factorization performance. Setting this parameter to true, enables the use of a block oriented Approximate Minimum Degree ordering algorithm. Settings it to `false`, uses a scalar AMD algorithm. This option only makes sense when using `sparse_linear_algebra_library = SUITE_SPARSE` as it uses the `AMD` package that is part of `SuiteSparse`.

27. `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`.

28. `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`.

29. `eta` ($10^{-1}$) 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

$$\frac{Q_i - Q_{i-1}}{Q_i} < \frac{\eta}{i} \tag{13.28}$$

30. `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.

31. `logging_type` (`PER_MINIMIZER_ITERATION`)

32. `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`.

33. `return_initial_residuals` (`false`)

34. `return_final_residuals` (`false`) If true, the vectors `Solver::Summary::initial_residuals` and `Solver::Summary::final_residuals` are filled with the residuals before and after the optimization. The entries of these vectors are in the order in which ResidualBlocks were added to the Problem object.

35. `return_initial_gradient` (`false`)

36. `return_final_gradient` (`false`) If true, the vectors `Solver::Summary::initial_gradient` and `Solver::Summary::final_gradient` are filled with the gradient before and after the optimization. The entries of these vectors are in the order in which ParameterBlocks were added to the Problem object.

    Since `AddResidualBlock` adds ParameterBlocks to the `Problem` automatically if they do not already exist, if you wish to have explicit control over the ordering of the vectors, then use `Problem::AddParameterBlock` to explicitly add the ParameterBlocks in the order desired.

37. `return_initial_jacobian` (`false`)

38. `return_initial_jacobian` (`false`) If true, the Jacobian matrices before and after the optimization are returned in `Solver::Summary::initial_jacobian` and `Solver::Summary::final_jacobian` respectively.

    The rows of these matrices are in the same order in which the ResidualBlocks were added to the Problem object. The columns are in the same order in which the ParameterBlocks were added to the Problem object.

Since AddResidualBlock adds ParameterBlocks to the Problem automatically if they do not already exist, if you wish to have explicit control over the column ordering of the matrix, then use Problem::AddParameterBlock to explicitly add the ParameterBlocks in the order desired.

The Jacobian matrices are stored as compressed row sparse matrices. Please see ceres/crs_matrix.h for more details of the format.

39. 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.

40. 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.

41. lsqp_dump_format (TEXTFILE) 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.

42. 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.

43. gradient_check_relative_precision ($10^{-8}$) 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.

44. `numeric_derivative_relative_step_size` ($10^{-6}$) 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

$$\delta = \texttt{numeric\_derivative\_relative\_step\_size} \tag{13.29}$$

$$\Delta f = \frac{f((1+\delta)x) - f(x)}{\delta x} \tag{13.30}$$

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. $10^9$) and when the values are small (e.g. $10^{-5}$). It is possible to construct "torture cases" which break this finite difference heuristic, but they do not come up often in practice.

45. `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.

46. `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`.

47. `solver_log` If non-empty, a summary of the execution of the solver is recorded to this file. This file is used for recording and Ceres' performance. Currently, only the iteration number, total time and the objective function value are logged. The format of this file is expected to change over time as the performance evaluation framework is fleshed out.

## 13.6 `SOLVER::SUMMARY`

TBD

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.

```
   4  4  4  4
2  X  X  X  X
4  X
4     X
4        X
4           X
```

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. [12]. For a general introduction to non-linear optimization we recommend the text by Nocedal & Wright [17]. 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 [22]. Triggs et al., provide a thorough coverage of the bundle adjustment problem [23].

# 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] R.H. Byrd, R.B. Schnabel, and G.A. Shultz. Approximate solution of the trust region problem by minimization over two-dimensional subspaces. *Mathematical programming*, 40(1):247–263, 1988.

[5] 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.

[6] A.R. Conn, N.I.M. Gould, and P.L. Toint. *Trust-region methods*, volume 1. Society for Industrial Mathematics, 2000.

[7] G.H. Golub and V. Pereyra. The differentiation of pseudo-inverses and nonlinear least squares problems whose variables separate. *SIAM Journal on numerical analysis*, 10(2):413–432, 1973.

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

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

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

[11] 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.

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

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

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

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

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

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

[18] A. Ruhe and P.Å. Wedin. Algorithms for separable nonlinear least squares problems. *Siam Review*, 22(3):318–337, 1980.

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

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

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

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

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

[24] T. Wiberg. Computation of principal components when data are missing. In *Proc. Second Symp. Computational Statistics*, pages 229–236, 1976.

[25] 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.