Solving Non-linear Least Squares

Introduction

Effective use of Ceres requires some familiarity with the basic components of a non-linear least squares solver, so before we describe how to configure and use the solver, we will take a brief look at how some of the core optimization algorithms in Ceres work.

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

(1)¶

$$\arg \min_x \frac{1}{2}\|F(x)\|^2\ . \\ L \le x \le U$$

Where, L and U are lower and upper bounds on the parameter vector x.

Since the efficient global minimization of (1) for general $F(x)$ is an intractable problem, we will have to settle for finding a local minimum.

In the following, 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 is $g(x) = \nabla \frac{1}{2}\|F(x)\|^2 = J(x)^\top F(x)$.

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

(2)¶

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

Unfortunately, naively solving a sequence of these problems and updating $x \leftarrow x+ \Delta x$ leads to an algorithm that may not converge. To get a convergent algorithm, we need to control the size of the step Δx. Depending on how the size of the step Δx is controlled, non-linear optimization algorithms can be divided into two major categories [NocedalWright].

1. **Trust Region** The trust region approach approximates the objective function using using a model function (often a quadratic) over a subset of the search space known as the trust region. If the model function succeeds in minimizing the true objective function the trust region is expanded; conversely, otherwise it is contracted and the model optimization problem is solved again.
2. **Line Search** The line search approach first finds a descent direction along which the objective function will be reduced and then computes a step size that decides how far should move along that direction. The descent direction can be computed by various methods, such as gradient descent, Newton's method and Quasi-Newton method. The step size can be determined either exactly or inexactly.

Trust region methods are in some sense dual to line search methods: trust region methods first choose a step size (the size of the trust region) and then a step direction while line search methods first choose a step direction and then a step size. Ceres implements multiple algorithms in both categories.

Trust Region Methods

The basic trust region algorithm looks something like this.

1. Given an initial point x and a trust region radius μ.
2. Solve

$$\begin{aligned}\arg\min_{\Delta x} & \frac{1}{2}\|J(x)\Delta x + F(x)\|^2 \\ \text{such that} & \|D(x)\Delta x\|^2 \le \mu \\ & L \le x + \Delta x \le U.\end{aligned}$$

3. $\rho = \dfrac{\|F(x+\Delta x)\|^2 - \|F(x)\|^2}{\|J(x)\Delta x + F(x)\|^2 - \|F(x)\|^2}$
4. if $\rho > \epsilon$ then $x = x + \Delta x$.
5. if $\rho > \eta_1$ then $\mu = 2\mu$
6. else if $\rho < \eta_2$ then $\mu = 0.5 * \mu$
7. Go to 2.

Here, μ is the trust region radius, $D(x)$ is some matrix used to define a metric on the domain of $F(x)$ and ρ measures the quality of the step Δx, 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 ρ.

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

(3)¶

$$\begin{aligned} \arg\min_{\Delta x} &\quad \frac{1}{2}\|J(x)\Delta x + F(x)\|^2 \\ \text{such that} &\quad \|D(x)\Delta x\|^2 \le \mu \\ &\quad L \le x + \Delta x \le U. \end{aligned}$$

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, each of which is augmented with a line search if bounds constraints are present [Kanzow]. The user can choose between them by setting `Solver::Options::trust_region_strategy_type`.

Footnotes

[1] At the level of the non-linear solver, the block structure is not relevant, therefore our discussion here is in terms of an optimization problem defined over a state vector of size n. Similarly the presence of loss functions is also ignored as the problem is internally converted into a pure non-linear least squares problem.

Levenberg-Marquardt

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

It can be shown, that the solution to (3) can be obtained by solving an unconstrained optimization of the form

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

Where, λ is a Lagrange multiplier that is inverse related to μ. In Ceres, we solve for

(4)¶

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

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 $\frac{1}{\sqrt{\mu}}D$ 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.

(5)¶

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

For all but the smallest problems the solution of (5) in each iteration of the Levenberg-Marquardt algorithm is the dominant computational cost in Ceres. Ceres provides a number of different options for solving (5). There are two major classes of methods - factorization and iterative.

The factorization methods are based on computing an exact solution of (4) 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 (4) is necessary at each step of the LM algorithm to solve (1). In fact, we have already seen evidence that this may not be the case, as (4) is itself a regularized version of (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 [NocedalWright].

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

(6)¶

$$\|H(x)\Delta x + g(x)\| \leq \eta_k \|g(x)\|.$$

Here, k indicates the Levenberg-Marquardt iteration number and $0 < \eta_k < 1$ is known as the forcing sequence. [WrightHolt] prove that a truncated Levenberg-Marquardt algorithm that uses an inexact Newton step based on (6) converges for any sequence $\eta_k \leq \eta_0 < 1$ and the rate of convergence depends on the choice of the forcing sequence η_k.

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 (3) was introduced by M. J. D. Powell. The key idea there is to compute two vectors

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

Note that the vector $\Delta x^{\text{Gauss-Newton}}$ is the solution to (2) and Δx^{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 Δx defined by $\Delta x^{\text{Gauss-Newton}}$ and Δx^{Cauchy} that solves the trust region problem. Ceres supports two variants that can be chose by setting `Solver::Options::dogleg_type`.

`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 [Madsen].

`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 [ByrdSchnabel].

The key advantage of the Dogleg over Levenberg-Marquardt is that if the step computation for a particular choice of μ 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 μ. 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 μ.

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. For example, consider the following regression problem

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

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 [GolubPereyra].

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

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 [RuheWedin].

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.,

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

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 Trust Region Methods is a descent algorithm in that it 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 principled 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 [Conn].

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 steps is available for all trust region strategies.

Line Search Methods

The line search method in Ceres Solver cannot handle bounds constraints right now, so it can only be used for solving unconstrained problems.

Line search algorithms

1. Given an initial point x
2. $\Delta x = -H^{-1}(x)g(x)$
3. $\arg\min_\mu \frac{1}{2}\|F(x+\mu\Delta x)\|^2$
4. $x = x + \mu\Delta x$
5. Goto 2.

Here $H(x)$ is some approximation to the Hessian of the objective function, and $g(x)$ is the gradient at x. Depending on the choice of $H(x)$ we get a variety of different search directions Δx.

Step 4, which is a one dimensional optimization or *Line Search* along Δx is what gives this class of methods its name.

Different line search algorithms differ in their choice of the search direction Δx and the method used for one dimensional optimization along Δx. The choice of $H(x)$ is the primary source of computational complexity in these methods. Currently, Ceres Solver supports three choices of search directions, all aimed at large scale problems.

1. `STEEPEST_DESCENT` This corresponds to choosing $H(x)$ to be the identity matrix. This is not a good search direction for anything but the simplest of the problems. It is only included here for completeness.
2. `NONLINEAR_CONJUGATE_GRADIENT` A generalization of the Conjugate Gradient method to non-linear functions. The generalization can be performed in a number of different ways, resulting in a variety of search directions. Ceres Solver currently supports `FLETCHER_REEVES`, `POLAK_RIBIERE` and `HESTENES_STIEFEL` directions.
3. `BFGS` A generalization of the Secant method to multiple dimensions in which a full, dense approximation to the inverse Hessian is maintained and used to compute a quasi-Newton step [NocedalWright]. BFGS is currently the best known general quasi-Newton algorithm.
4. `LBFGS` A limited memory approximation to the full `BFGS` method in which the last *M* iterations are used to approximate the inverse Hessian used to compute a quasi-Newton step [Nocedal], [ByrdNocedal].

Currently Ceres Solver supports both a backtracking and interpolation based Armijo line search algorithm, and a sectioning / zoom interpolation (strong) Wolfe condition line search algorithm. However, note that in order for the assumptions underlying the `BFGS` and `LBFGS` methods to be guaranteed to be satisfied the Wolfe line search algorithm should be used.

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

(7)¶

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

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 (7) is equivalent to solving the *normal equations*.

(8)¶

$$H\Delta x = g$$

Ceres provides a number of different options for solving (8).

`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 [Bjorck]. Let $J = QR$ be the QR-decomposition of J, where Q is an orthonormal matrix and R is an upper triangular matrix [TrefethenBau]. Then it can be shown that the solution to (8) is given by

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

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 (8) is given by

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

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 $J^\top J = R^\top Q^\top QR = R^\top R$. 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 [Chen] or the sparse Cholesky factorization algorithm in `Eigen` (which incidently is a port of the algorithm implemented inside `CXSparse`)

`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

$$Hx = J^\top Jx = J^\top (Jx)$$

The convergence of Conjugate Gradients depends on the conditioner number $\kappa(H)$. Usually H is poorly conditioned and a Preconditioner must be used to get reasonable performance. Currently only the `JACOBI` preconditioner is available for use with `CGNR`. It uses the block diagonal of H to precondition the normal equations.

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

`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 (8) can be constructed.

Suppose that the SfM problem consists of p cameras and q points and the variable vector x has the block structure $x = [y_1, \dots, y_p, z_1, \dots, z_q]$. 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

(9)¶

$$H = \left[\begin{matrix} B & E\\ E^\top & C \end{matrix} \right]\ ,$$

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 (8) as the block structured linear system

(10)¶

$$\left[\begin{matrix} B & E\\ E^\top & C \end{matrix}\right]\left[\begin{matrix} \Delta y \\ \Delta z \end{matrix}\right] = \left[\begin{matrix} v\\ w \end{matrix}\right]\ ,$$

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

(11)¶

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

The matrix

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

is the Schur complement of C in H. It is also known as the *reduced camera matrix*, because the only variables participating in (11) are the ones corresponding to the cameras. $S \in \mathbb{R}^{pc\times pc}$ is a block structured symmetric positive definite matrix, with blocks of size $c\times c$. The block S_{ij} corresponding to the pair of images i and j is non-zero if and only if the two images observe at least one common point.

Now, (10) can be solved by first forming S, solving for Δy, and then back-substituting Δy to obtain the value of Δz. Thus, the solution of what was an $n\times n$, $n=pc+qs$ linear system is reduced to the inversion of the block diagonal matrix C, a few matrix-matrix and matrix-vector multiplies, and the solution of block sparse $pc\times pc$ linear system (11). For almost all problems, the number of cameras is much smaller than the number of points, $p \ll q$, thus solving (11) is significantly cheaper than solving (10). This is the *Schur complement trick* [Brown].

This still leaves open the question of solving (11). The method of choice for solving symmetric positive definite systems exactly is via the Cholesky factorization [TrefethenBau] 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 [TrefethenBau]. 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 [Chen]. 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.

`ITERATIVE_SCHUR`

Another option for bundle adjustment problems is to apply Preconditioned Conjugate Gradients 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 Conjugate Gradients 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 key computational operation when using Conjuagate Gradients is the evaluation of the matrix vector product Sx for an arbitrary vector x. There are two ways in which this product can be evaluated, and this can be controlled using `Solver::Options::use_explicit_schur_complement`. Depending on the problem at hand, the performance difference between these two methods can be quite substantial.

1. **Implicit** This is default. Implicit evaluation is suitable for large problems where the cost of computing and storing the Schur Complement S is prohibitive. Because PCG only needs access to S via its product with a vector, one way to evaluate Sx is to observe that

   x_1 &= E^\top x

   x_2 &= C^{-1} x_1

   $$x_3 = Ex_2$$

   $$x_4 = Bx$$

   (12)¶

   Sx &= x_4 - x_3

   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, (12) can be implemented using just the columns of J.

   Equation (12) 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 [Saad] [Mathew].
2. **Explicit** The complexity of implicit matrix-vector product evaluation scales with the number of non-zeros in the Jacobian. For small to medium sized problems, the cost of constructing the Schur Complement is small enough that it is better to construct it explicitly in memory and use it to evaluate the product Sx.

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

> Note
>
> In exact arithmetic, the choice of implicit versus explicit Schur complement would have no impact on solution quality. However, in practice if the Jacobian is poorly conditioned, one may observe (usually small) differences in solution quality. This is a natural consequence of performing computations in finite arithmetic.

Preconditioner

The convergence rate of Conjugate Gradients for solving (8) depends on the distribution of eigenvalues of H [Saad]. A useful upper bound is $\sqrt{\kappa(H)}$, where, $\kappa(H)$ 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 (8) results in extremely poor performance.

The solution to this problem is to replace (8) 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., $M = \operatorname{diag}(A)$, which for block structured matrices like H can be generalized to the block Jacobi preconditioner. Ceres implements the block Jacobi preconditioner and refers to it as `JACOBI`. When used with CGNR it refers to the block diagonal of H and when used with ITERATIVE_SCHUR it refers to the block diagonal of B [Mandel].

Another obvious choice for ITERATIVE_SCHUR is the block diagonal of the Schur complement matrix S, i.e, the block Jacobi preconditioner for S. Ceres implements it and refers to is as the `SCHUR_JACOBI` preconditioner.

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 [KushalAgarwal]. 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.

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. For example, consider the linear system

$$\begin{aligned} x + y &= 3 \\ 2x + 3y &= 7 \end{aligned}$$

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 [LiSaad].

`Solver::Options`

class `Solver::Options`

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

bool `Solver::Options::IsValid`(string **error*) *const*

Validate the values in the options struct and returns true on success. If there is a problem, the method returns false with `error` containing a textual description of the cause.

MinimizerType `Solver::Options::minimizer_type`

Default: `TRUST_REGION`

Choose between `LINE_SEARCH` and `TRUST_REGION` algorithms. See Trust Region Methods and Line Search Methods for more details.

LineSearchDirectionType `Solver::Options::line_search_direction_type`

Default: `LBFGS`

Choices are `STEEPEST_DESCENT`, `NONLINEAR_CONJUGATE_GRADIENT`, `BFGS` and `LBFGS`.

LineSearchType `Solver::Options::line_search_type`

Default: `WOLFE`

Choices are `ARMIJO` and `WOLFE` (strong Wolfe conditions). Note that in order for the assumptions underlying the `BFGS` and `LBFGS` line search direction algorithms to be guaranteed to be satisifed, the `WOLFE` line search should be used.

NonlinearConjugateGradientType `Solver::Options::nonlinear_conjugate_gradient_type`

Default: `FLETCHER_REEVES`

Choices are `FLETCHER_REEVES`, `POLAK_RIBIERE` and `HESTENES_STIEFEL`.

int `Solver::Options::max_lbfgs_rank`

Default: 20

The L-BFGS hessian approximation is a low rank approximation to the inverse of the Hessian matrix. The rank of the approximation determines (linearly) the space and time complexity of using the approximation. Higher the rank, the better is the quality of the approximation. The increase in quality is however is bounded for a number of reasons.

1. The method only uses secant information and not actual derivatives.
2. The Hessian approximation is constrained to be positive definite.

So increasing this rank to a large number will cost time and space complexity without the corresponding increase in solution quality. There are no hard and fast rules for choosing the maximum rank. The best choice usually requires some problem specific experimentation.

bool `Solver::Options::use_approximate_eigenvalue_bfgs_scaling`

Default: `false`

As part of the `BFGS` update step / `LBFGS` right-multiply step, the initial inverse Hessian approximation is taken to be the Identity. However, [Oren] showed that using instead $I * \gamma$, where γ is a scalar chosen to approximate an eigenvalue of the true inverse Hessian can result in improved convergence in a wide variety of cases. Setting `use_approximate_eigenvalue_bfgs_scaling` to true enables this scaling in `BFGS` (before first iteration) and `LBFGS` (at each iteration).

Precisely, approximate eigenvalue scaling equates to

$$\gamma = \frac{y_k' s_k}{y_k' y_k}$$

With:

$$y_k = \nabla f_{k+1} - \nabla f_k$$

$$s_k = x_{k+1} - x_k$$

Where $f()$ is the line search objective and x the vector of parameter values [NocedalWright].

It is important to note that approximate eigenvalue scaling does **not** *always* improve convergence, and that it can in fact *significantly* degrade performance for certain classes of problem, which is why it is disabled by default. In particular it can degrade performance when the sensitivity of the problem to different parameters varies significantly, as in this case a single scalar factor fails to capture this variation and detrimentally downscales parts of the Jacobian approximation which correspond to low-sensitivity parameters. It can also reduce the robustness of the solution to errors in the Jacobians.

LineSearchIterpolationType `Solver::Options::line_search_interpolation_type`

Default: `CUBIC`

Degree of the polynomial used to approximate the objective function. Valid values are `BISECTION`, `QUADRATIC` and `CUBIC`.

double `Solver::Options::min_line_search_step_size`

The line search terminates if:

$$\|\Delta x_k\|_\infty < \text{min\_line\_search\_step\_size}$$

where $\|\cdot\|_\infty$ refers to the max norm, and Δx_k is the step change in the parameter values at the k-th iteration.

double `Solver::Options::line_search_sufficient_function_decrease`

Default: `1e-4`

Solving the line search problem exactly is computationally prohibitive. Fortunately, line search based optimization algorithms can still guarantee convergence if instead of an exact solution, the line search algorithm returns a solution which decreases the value of the objective function sufficiently. More precisely, we are looking for a step size s.t.

$$f(\text{step\_size}) \le f(0) + \text{sufficient\_decrease} * [f'(0) * \text{step\_size}]$$

This condition is known as the Armijo condition.

double `Solver::Options::max_line_search_step_contraction`

Default: `1e-3`

In each iteration of the line search,

$$\text{new\_step\_size} >= \text{max\_line\_search\_step\_contraction} * \text{step\_size}$$

Note that by definition, for contraction:

$$0 < \text{max\_step\_contraction} < \text{min\_step\_contraction} < 1$$

double `Solver::Options::min_line_search_step_contraction`

Default: `0.6`

In each iteration of the line search,

$$\text{new\_step\_size} <= \text{min\_line\_search\_step\_contraction} * \text{step\_size}$$

Note that by definition, for contraction:

$$0 < \text{max\_step\_contraction} < \text{min\_step\_contraction} < 1$$

int `Solver::Options::max_num_line_search_step_size_iterations`

Default: `20`

Maximum number of trial step size iterations during each line search, if a step size satisfying the search conditions cannot be found within this number of trials, the line search will stop.

As this is an ‘artificial’ constraint (one imposed by the user, not the underlying math), if `WOLFE` line search is being used, *and* points satisfying the Armijo sufficient (function) decrease condition have been found during the current search (in $<=$ `max_num_line_search_step_size_iterations`). Then, the step size with the lowest function value which satisfies the Armijo condition will be returned as the new valid step, even though it does *not* satisfy the strong Wolfe conditions. This behaviour protects against early termination of the optimizer at a sub-optimal point.

int `Solver::Options::max_num_line_search_direction_restarts`

Default: `5`

Maximum number of restarts of the line search direction algorithm before terminating the optimization. Restarts of the line search direction algorithm occur when the current algorithm fails to produce a new descent direction. This typically indicates a numerical failure, or a breakdown in the validity of the approximations used.

double `Solver::Options::line_search_sufficient_curvature_decrease`

Default: `0.9`

The strong Wolfe conditions consist of the Armijo sufficient decrease condition, and an additional requirement that the step size be chosen s.t. the *magnitude* (‘strong’ Wolfe conditions) of the gradient along the search direction decreases sufficiently. Precisely, this second condition is that we seek a step size s.t.

$$\|f'(\text{step\_size})\| <= \text{sufficient\_curvature\_decrease} * \|f'(0)\|$$

Where $f()$ is the line search objective and $f'()$ is the derivative of f with respect to the step size: $\frac{df}{d\,\text{step size}}$.

double `Solver::Options::max_line_search_step_expansion`

Default: `10.0`

During the bracketing phase of a Wolfe line search, the step size is increased until either a point satisfying the Wolfe conditions is found, or an upper bound for a bracket containing a point satisfying the conditions is found. Precisely, at each iteration of the expansion:

$$\text{new\_step\_size} <= \text{max\_step\_expansion} * \text{step\_size}$$

By definition for expansion

$$\text{max\_step\_expansion} > 1.0$$

TrustRegionStrategyType `Solver::Options::trust_region_strategy_type`

Default: `LEVENBERG_MARQUARDT`

The trust region step computation algorithm used by Ceres. Currently `LEVENBERG_MARQUARDT` and `DOGLEG` are the two valid choices. See Levenberg-Marquardt and Dogleg for more details.

DoglegType `Solver::Options::dogleg_type`

Default: `TRADITIONAL_DOGLEG`

Ceres supports two different dogleg strategies. `TRADITIONAL_DOGLEG` method by Powell and the `SUBSPACE_DOGLEG` method described by [ByrdSchnabel] . See Dogleg for more details.

bool `Solver::Options::use_nonmonotonic_steps`

Default: `false`

Relax the requirement that the trust-region algorithm take strictly decreasing steps. See Non-monotonic Steps for more details.

int `Solver::Options::max_consecutive_nonmonotonic_steps`

Default: `5`

The window size used by the step selection algorithm to accept non-monotonic steps.

int `Solver::Options::max_num_iterations`

Default: `50`

Maximum number of iterations for which the solver should run.

double `Solver::Options::max_solver_time_in_seconds`

Default: `1e6` Maximum amount of time for which the solver should run.

int `Solver::Options::num_threads`

Default: `1`

Number of threads used by Ceres to evaluate the Jacobian.

double `Solver::Options::initial_trust_region_radius`

Default: `1e4`

The size of the initial trust region. When the `LEVENBERG_MARQUARDT` strategy is used, the reciprocal of this number is the initial regularization parameter.

double `Solver::Options::max_trust_region_radius`

Default: `1e16`

The trust region radius is not allowed to grow beyond this value.

double `Solver::Options::min_trust_region_radius`

Default: `1e-32`

The solver terminates, when the trust region becomes smaller than this value.

double `Solver::Options::min_relative_decrease`

Default: `1e-3`

Lower threshold for relative decrease before a trust-region step is accepted.

double `Solver::Options::min_lm_diagonal`

Default: `1e6`

The `LEVENBERG_MARQUARDT` strategy, uses a diagonal matrix to regularize the trust region step. This is the lower bound on the values of this diagonal matrix.

double `Solver::Options::max_lm_diagonal`

Default: `1e32`

The `LEVENBERG_MARQUARDT` strategy, uses a diagonal matrix to regularize the trust region step. This is the upper bound on the values of this diagonal matrix.

int `Solver::Options::max_num_consecutive_invalid_steps`

Default: `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.

double `Solver::Options::function_tolerance`

Default: `1e-6`

Solver terminates if

$$\frac{|\Delta \text{cost}|}{\text{cost}} <= \text{function\_tolerance}$$

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

double `Solver::Options::gradient_tolerance`

Default: `1e-10`

Solver terminates if

$$\|x - \Pi \boxplus (x, -g(x))\|_\infty <= \text{gradient\_tolerance}$$

where $\|\cdot\|_\infty$ refers to the max norm, Π is projection onto the bounds constraints and $\boxplus$ is Plus operation for the overall local parameterization associated with the parameter vector.

double `Solver::Options::parameter_tolerance`

Default: `1e-8`

Solver terminates if

$$\|\Delta x\| <= (\|x\| + \text{parameter\_tolerance}) * \text{parameter\_tolerance}$$

where Δx is the step computed by the linear solver in the current iteration.

LinearSolverType `Solver::Options::linear_solver_type`

Default: `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 built with support for `SuiteSparse` or `CXSparse` or `Eigen`'s sparse Cholesky factorization, the default is `SPARSE_NORMAL_CHOLESKY`, it is `DENSE_QR` otherwise.

PreconditionerType `Solver::Options::preconditioner_type`

Default: `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`. See Preconditioner for more details.

VisibilityClusteringType `Solver::Options::visibility_clustering_type`

Default: `CANONICAL_VIEWS`

Type of clustering algorithm to use when constructing a visibility based preconditioner. The original visibility based preconditioning paper and implementation only used the canonical views algorithm.

This algorithm gives high quality results but for large dense graphs can be particularly expensive. As its worst case complexity is cubic in size of the graph.

Another option is to use `SINGLE_LINKAGE` which is a simple thresholded single linkage clustering algorithm that only pays attention to tightly coupled blocks in the Schur complement. This is a fast algorithm that works well.

The optimal choice of the clustering algorithm depends on the sparsity structure of the problem, but generally speaking we recommend that you try `CANONICAL_VIEWS` first and if it is too expensive try `SINGLE_LINKAGE`.

DenseLinearAlgebraLibrary `Solver::Options::dense_linear_algebra_library_type`

Default: `EIGEN`

Ceres supports using multiple dense linear algebra libraries for dense matrix factorizations. Currently `EIGEN` and `LAPACK` are the valid choices. `EIGEN` is always available, `LAPACK` refers to the system `BLAS + LAPACK` library which may or may not be available.

This setting affects the `DENSE_QR`, `DENSE_NORMAL_CHOLESKY` and `DENSE_SCHUR` solvers. For small to moderate sized probem `EIGEN` is a fine choice but for large problems, an optimized `LAPACK + BLAS` implementation can make a substantial difference in performance.

SparseLinearAlgebraLibrary `Solver::Options::sparse_linear_algebra_library_type`

Default: The highest available according to: `SUITE_SPARSE` > `CX_SPARSE` > `EIGEN_SPARSE` > `NO_SPARSE`

Ceres supports the use of three sparse linear algebra libraries, `SuiteSparse`, which is enabled by setting this parameter to `SUITE_SPARSE`, `CXSparse`, which can be selected by setting this parameter to `CX_SPARSE` and `Eigen` which is enabled by setting this parameter to `EIGEN_SPARSE`. Lastly, `NO_SPARSE` means that no sparse linear solver should be used; note that this is irrespective of whether Ceres was compiled with support for one.

`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`.

Last but not the least you can use the sparse linear algebra routines in `Eigen`. Currently the performance of this library is the poorest of the three. But this should change in the near future.

Another thing to consider here is that the sparse Cholesky factorization libraries in Eigen are licensed under `LGPL` and building Ceres with support for `EIGEN_SPARSE` will result in an LGPL licensed library (since the corresponding code from Eigen is compiled into the library).

The upside is that you do not need to build and link to an external library to use `EIGEN_SPARSE`.

int `Solver::Options::num_linear_solver_threads`

Default: `-1`

This field is deprecated, and is ignored by Ceres. Solver::Options::num_threads controls threading for all of Ceres Solver. This setting is scheduled to be removed in 1.15.0.

Number of threads used by the linear solver.

shared_ptr<ParameterBlockOrdering> `Solver::Options::linear_solver_ordering`

Default: `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~ref{sec:ordering`` for more details.

If `NULL`, the solver is free to choose an ordering that it thinks is best.

See Ordering for more details.

bool `Solver::Options::use_explicit_schur_complement`

Default: `false`

Use an explicitly computed Schur complement matrix with `ITERATIVE_SCHUR`.

By default this option is disabled and `ITERATIVE_SCHUR` evaluates evaluates matrix-vector products between the Schur complement and a vector implicitly by exploiting the algebraic expression for the Schur complement.

The cost of this evaluation scales with the number of non-zeros in the Jacobian.

For small to medium sized problems there is a sweet spot where computing the Schur complement is cheap enough that it is much more efficient to explicitly compute it and use it for evaluating the matrix-vector products.

Enabling this option tells `ITERATIVE_SCHUR` to use an explicitly computed Schur complement. This can improve the performance of the `ITERATIVE_SCHUR` solver significantly.

bool `Solver::Options::use_post_ordering`

Default: `false`

Sparse Cholesky factorization algorithms use a fill-reducing ordering to permute the columns of the Jacobian matrix. There are two ways of doing this.

1. Compute the Jacobian matrix in some order and then have the factorization algorithm permute the columns of the Jacobian.
2. Compute the Jacobian with its columns already permuted.

The first option incurs a significant memory penalty. The factorization algorithm has to make a copy of the permuted Jacobian matrix, thus Ceres pre-permutes the columns of the Jacobian matrix and generally speaking, there is no performance penalty for doing so.

In some rare cases, it is worth using a more complicated reordering algorithm which has slightly better runtime performance at the expense of an extra copy of the Jacobian matrix. Setting `use_postordering` to `true` enables this tradeoff.

bool `Solver::Options::dynamic_sparsity`

Some non-linear least squares problems are symbolically dense but numerically sparse. i.e. at any given state only a small number of Jacobian entries are non-zero, but the position and number of non-zeros is different depending on the state. For these problems it can be useful to factorize the sparse jacobian at each solver iteration instead of including all of the zero entries in a single general factorization.

If your problem does not have this property (or you do not know), then it is probably best to keep this false, otherwise it will likely lead to worse performance.

This setting only affects the *SPARSE_NORMAL_CHOLESKY* solver.

int `Solver::Options::min_linear_solver_iterations`

Default: `0`

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` or `CGNR`.

int `Solver::Options::max_linear_solver_iterations`

Default: `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` or `CGNR`.

double `Solver::Options::eta`

Default: `1e-1`

Forcing sequence parameter. The truncated Newton solver uses this number to control the relative accuracy with which the Newton step is computed. This constant is passed to `ConjugateGradientsSolver` which uses it to terminate the iterations when

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

bool `Solver::Options::jacobi_scaling`

Default: `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.

bool `Solver::Options::use_inner_iterations`

Default: `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 Inner Iterations.

double `Solver::Options::inner_iteration_tolerance`

Default: `1e-3`

Generally speaking, inner iterations make significant progress in the early stages of the solve and then their contribution drops down sharply, at which point the time spent doing inner iterations is not worth it.

Once the relative decrease in the objective function due to inner iterations drops below `inner_iteration_tolerance`, the use of inner iterations in subsequent trust region minimizer iterations is disabled.

shared_ptr<ParameterBlockOrdering> `Solver::Options::inner_iteration_ordering`

Default: `NULL`

If `Solver::Options::use_inner_iterations` true, then the user has two choices.

1. Let the solver heuristically decide which parameter blocks to optimize in each inner iteration. To do this, set `Solver::Options::inner_iteration_ordering` to `NULL`.
2. 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. Not all parameter blocks need to be included in the ordering.

See Ordering for more details.

LoggingType `Solver::Options::logging_type`

Default: `PER_MINIMIZER_ITERATION`

bool `Solver::Options::minimizer_progress_to_stdout`

Default: `false`

By default the `Minimizer` progress is logged to `STDERR` depending on the `vlog` level. If this flag is set to true, and `Solver::Options::logging_type` is not `SILENT`, the logging output is sent to `STDOUT`.

For `TRUST_REGION_MINIMIZER` the progress display looks like

```
iter      cost      cost_change  |gradient|   |step|    tr_ratio  tr_radius
ls_iter  iter_time  total_time
   0  4.185660e+06    0.00e+00    1.09e+08   0.00e+00   0.00e+00  1.00e+04       0
7.59e-02    3.37e-01
   1  1.062590e+05    4.08e+06    8.99e+06   5.36e+02   9.82e-01  3.00e+04       1
1.65e-01    5.03e-01
   2  4.992817e+04    5.63e+04    8.32e+06   3.19e+02   6.52e-01  3.09e+04       1
1.45e-01    6.48e-01
```

Here

1. `cost` is the value of the objective function.
2. `cost_change` is the change in the value of the objective function if the step computed in this iteration is accepted.
3. `|gradient|` is the max norm of the gradient.
4. `|step|` is the change in the parameter vector.
5. `tr_ratio` is the ratio of the actual change in the objective function value to the change in the value of the trust region model.
6. `tr_radius` is the size of the trust region radius.
7. `ls_iter` is the number of linear solver iterations used to compute the trust region step. For direct/factorization based solvers it is always 1, for iterative solvers like `ITERATIVE_SCHUR` it is the number of iterations of the Conjugate Gradients algorithm.
8. `iter_time` is the time take by the current iteration.
9. `total_time` is the total time taken by the minimizer.

For `LINE_SEARCH_MINIMIZER` the progress display looks like

```
0: f: 2.317806e+05 d: 0.00e+00 g: 3.19e-01 h: 0.00e+00 s: 0.00e+00 e:  0 it: 2.98e-
02 tt: 8.50e-02
1: f: 2.312019e+05 d: 5.79e+02 g: 3.18e-01 h: 2.41e+01 s: 1.00e+00 e:  1 it: 4.54e-
02 tt: 1.31e-01
2: f: 2.300462e+05 d: 1.16e+03 g: 3.17e-01 h: 4.90e+01 s: 2.54e-03 e:  1 it: 4.96e-
02 tt: 1.81e-01
```

Here

1. `f` is the value of the objective function.
2. `d` is the change in the value of the objective function if the step computed in this iteration is accepted.
3. `g` is the max norm of the gradient.

4. `h` is the change in the parameter vector.
5. `s` is the optimal step length computed by the line search.
6. `it` is the time take by the current iteration.
7. `tt` is the total time taken by the minimizer.

vector<int> `Solver::Options::trust_region_minimizer_iterations_to_dump`

Default: `empty`

List of iterations at which the trust region minimizer should dump the trust region problem. Useful for testing and benchmarking. If `empty`, no problems are dumped.

string `Solver::Options::trust_region_problem_dump_directory`

Default: `/tmp`

> Directory to which the problems should be written to. Should be non-empty if `Solver::Options::trust_region_minimizer_iterations_to_dump` is non-empty and `Solver::Options::trust_region_problem_dump_format_type` is not `CONSOLE`.

DumpFormatType `Solver::Options::trust_region_problem_dump_format`

Default: `TEXTFILE`

The format in which trust region problems should be logged when `Solver::Options::trust_region_minimizer_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.

- `TEXTFILE` Write out the linear least squares problem to the directory pointed to by `Solver::Options::trust_region_problem_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 `ceres_solver_iteration_???.m` is also output, which can be used to parse and load the problem into memory.

bool `Solver::Options::check_gradients`

Default: `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, the optimization fails and the details are stored in the solver summary.

double `Solver::Options::gradient_check_relative_precision`

Default: `1e08`

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.

double
`Solver::Options::gradient_check_numeric_derivative_relative_step_size`

Default: `1e-6`

Note

This option only applies to the numeric differentiation used for checking the user provided derivatives when when *Solver::Options::check_gradients* is true. If you are using `NumericDiffCostFunction` and are interested in changing the step size for numeric differentiation in your cost function, please have a look at `NumericDiffOptions`.

Relative shift used for taking numeric derivatives when *Solver::Options::check_gradients* is *true*.

For finite differencing, each dimension is evaluated at slightly shifted values, e.g., for forward differences, the numerical derivative is

$$\delta = gradient\_check\_numeric\_derivative\_relative\_step\_size$$
$$\Delta f = \frac{f((1+\delta)x) - f(x)}{\delta x}$$

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.

vector<IterationCallback> `Solver::Options::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 `Solver::Options::update_state_every_iteration`. If the user wishes to have access to the updated parameter blocks when his/her callbacks are executed, then set `Solver::Options::update_state_every_iteration` to true.

The solver does NOT take ownership of these pointers.

bool `Solver::Options::update_state_every_iteration`

Default: `false`

If true, the user's parameter blocks are updated at the end of every Minimizer iteration, otherwise they are updated when the Minimizer terminates. This is useful if, for example, the user wishes to visualize the state of the optimization every iteration (in combination with an IterationCallback).

Note: If `Solver::Options::evaluation_callback` is set, then the behaviour of this flag is slightly different in each case:

1. If `Solver::Options::update_state_every_iteration` is false, then the user's state is changed at every residual and/or jacobian evaluation. Any user provided IterationCallbacks should **not** inspect and depend on the user visible state while the solver is running, since they it have undefined contents.
2. If `Solver::Options::update_state_every_iteration` is false, then the user's state is changed at every residual and/or jacobian evaluation, BUT the solver will ensure that before the user provided *IterationCallbacks* are called, the user visible state will be updated to the current best point found by the solver.

bool `Solver::Options::evaluation_callback`

Default: `NULL`

If non-`NULL`, gets notified when Ceres is about to evaluate the residuals and/or Jacobians. This enables sharing computation between residuals, which in some cases is important for efficient cost evaluation. See `EvaluationCallback` for details.

Note: Evaluation callbacks are incompatible with inner iterations.

Warning: This interacts with `Solver::Options::update_state_every_iteration`. See the documentation for that option for more details.

The solver does *not* take ownership of the pointer.

`ParameterBlockOrdering`

class `ParameterBlockOrdering`

`ParameterBlockOrdering` is a class for storing and manipulating an ordered collection of groups/sets with the following semantics:

Group IDs are non-negative integer values. Elements are any type that can serve as a key in a map or an element of a set.

An element can only belong to one group at a time. A group may contain an arbitrary number of elements.

Groups are ordered by their group id.

bool `ParameterBlockOrdering::AddElementToGroup`(*const* double **element*, *const* int *group*)

Add an element to a group. If a group with this id does not exist, one is created. This method can be called any number of times for the same element. Group ids should be non-negative numbers. Return value indicates if adding the element was a success.

void `ParameterBlockOrdering::Clear`()

Clear the ordering.

bool `ParameterBlockOrdering::Remove`(*const* double **element*)

Remove the element, no matter what group it is in. If the element is not a member of any group, calling this method will result in a crash. Return value indicates if the element was actually removed.

void `ParameterBlockOrdering::Reverse`()

Reverse the order of the groups in place.

int `ParameterBlockOrdering::GroupId`(*const* double **element*) *const*

Return the group id for the element. If the element is not a member of any group, return -1.

bool `ParameterBlockOrdering::IsMember`(*const* double **element*) *const*

True if there is a group containing the parameter block.

int `ParameterBlockOrdering::GroupSize`(*const* int *group*) *const*

This function always succeeds, i.e., implicitly there exists a group for every integer.

int `ParameterBlockOrdering::NumElements`() *const*

Number of elements in the ordering.

int `ParameterBlockOrdering::NumGroups`() *const*

Number of groups with one or more elements.

`EvaluationCallback`

class `EvaluationCallback`

Interface for receiving callbacks before Ceres evaluates residuals or Jacobians:

```
class EvaluationCallback {
 public:
  virtual ~EvaluationCallback() {}
  virtual void PrepareForEvaluation()(bool evaluate_jacobians
                                      bool new_evaluation_point) = 0;
};
```

`PrepareForEvaluation()` is called before Ceres requests residuals or jacobians for a given setting of the parameters. User parameters (the double* values provided to the cost functions) are fixed until the next call to `PrepareForEvaluation()`. If `new_evaluation_point == true`, then this is a new point that is different from the last evaluated point. Otherwise, it is the same point that was evaluated previously (either jacobian or residual) and the user can use cached results from previous evaluations. If `evaluate_jacobians` is true, then Ceres will request jacobians in the upcoming cost evaluation.

Using this callback interface, Ceres can notify you when it is about to evaluate the residuals or jacobians. With the callback, you can share computation between residual blocks by doing the shared computation in PrepareForEvaluation() before Ceres calls CostFunction::Evaluate() on all the residuals. It also enables caching results between a pure residual evaluation and a residual & jacobian evaluation, via the new_evaluation_point argument.

One use case for this callback is if the cost function compute is moved to the GPU. In that case, the prepare call does the actual cost function evaluation, and subsequent calls from Ceres to the actual cost functions merely copy the results from the GPU onto the corresponding blocks for Ceres to plug into the solver.

Note: Ceres provides no mechanism to share data other than the notification from the callback. Users must provide access to pre-computed shared data to their cost functions behind the scenes; this all happens without Ceres knowing. One approach is to put a pointer to the shared data in each cost function (recommended) or to use a global shared variable (discouraged; bug-prone). As far as Ceres is concerned, it is evaluating cost functions like any other; it just so happens that behind the scenes the cost functions reuse pre-computed data to execute faster.

See `evaluation_callback_test.cc` for code that explicitly verifies the preconditions between `PrepareForEvaluation()` and `CostFunction::Evaluate()`.

`IterationCallback`

class `IterationSummary`

`IterationSummary` describes the state of the minimizer at the end of each iteration.

int32 `IterationSummary::iteration`

Current iteration number.

bool `IterationSummary::step_is_valid`

Step was numerically valid, i.e., all values are finite and the step reduces the value of the linearized model.

Note: `IterationSummary::step_is_valid` is *false* when `IterationSummary::iteration` = 0.

bool `IterationSummary::step_is_nonmonotonic`

Step did not reduce the value of the objective function sufficiently, but it was accepted because of the relaxed acceptance criterion used by the non-monotonic trust region algorithm.

Note: `IterationSummary::step_is_nonmonotonic` is *false* when when `IterationSummary::iteration` = 0.

bool `IterationSummary::step_is_successful`

Whether or not the minimizer accepted this step or not.

If the ordinary trust region algorithm is used, this means that the relative reduction in the objective function value was greater than `Solver::Options::min_relative_decrease`. However, if the non-monotonic trust region algorithm is used (`Solver::Options::use_nonmonotonic_steps` = *true*), then even if the relative decrease is not sufficient, the algorithm may accept the step and the step is declared successful.

Note: `IterationSummary::step_is_successful` is *false* when when `IterationSummary::iteration` = 0.

double `IterationSummary::cost`

Value of the objective function.

double `IterationSummary::cost_change`

Change in the value of the objective function in this iteration. This can be positive or negative.

double `IterationSummary::gradient_max_norm`

Infinity norm of the gradient vector.

double `IterationSummary::gradient_norm`

2-norm of the gradient vector.

double `IterationSummary::step_norm`

2-norm of the size of the step computed in this iteration.

double `IterationSummary::relative_decrease`

For trust region algorithms, the ratio of the actual change in cost and the change in the cost of the linearized approximation.

This field is not used when a linear search minimizer is used.

double `IterationSummary::trust_region_radius`

Size of the trust region at the end of the current iteration. For the Levenberg-Marquardt algorithm, the regularization parameter is 1.0 / member::*IterationSummary::trust_region_radius*.

double `IterationSummary::eta`

For the inexact step Levenberg-Marquardt algorithm, this is the relative accuracy with which the step is solved. This number is only applicable to the iterative solvers capable of solving linear systems inexactly. Factorization-based exact solvers always have an eta of 0.0.

double `IterationSummary::step_size`

Step sized computed by the line search algorithm.

This field is not used when a trust region minimizer is used.

int `IterationSummary::line_search_function_evaluations`

Number of function evaluations used by the line search algorithm.

This field is not used when a trust region minimizer is used.

int `IterationSummary::linear_solver_iterations`

Number of iterations taken by the linear solver to solve for the trust region step.

Currently this field is not used when a line search minimizer is used.

double `IterationSummary::iteration_time_in_seconds`

Time (in seconds) spent inside the minimizer loop in the current iteration.

double `IterationSummary::step_solver_time_in_seconds`

Time (in seconds) spent inside the trust region step solver.

double `IterationSummary::cumulative_time_in_seconds`

Time (in seconds) since the user called Solve().

class `IterationCallback`

Interface for specifying callbacks that are executed at the end of each iteration of the minimizer.

```
class IterationCallback {
 public:
  virtual ~IterationCallback() {}
  virtual CallbackReturnType operator()(const IterationSummary& summary) = 0;
};
```

The solver uses the return value of `operator()` to decide whether to continue solving or to terminate. The user can return three values.

1. `SOLVER_ABORT` indicates that the callback detected an abnormal situation. The solver returns without updating the parameter blocks (unless `Solver::Options::update_state_every_iteration` is set true). Solver returns with `Solver::Summary::termination_type` set to `USER_FAILURE`.
2. `SOLVER_TERMINATE_SUCCESSFULLY` indicates that there is no need to optimize anymore (some user specified termination criterion has been met). Solver returns with `` Solver::Summary::termination_type` `` set to `USER_SUCCESS`.
3. `SOLVER_CONTINUE` indicates that the solver should continue optimizing.

For example, the following `IterationCallback` is used internally by Ceres to log the progress of the optimization.

```
class LoggingCallback : public IterationCallback {
 public:
  explicit LoggingCallback(bool log_to_stdout)
      : log_to_stdout_(log_to_stdout) {}

  ~LoggingCallback() {}

  CallbackReturnType operator()(const IterationSummary& summary) {
    const char* kReportRowFormat =
        "% 4d: f:% 8e d:% 3.2e g:% 3.2e h:% 3.2e "
        "rho:% 3.2e mu:% 3.2e eta:% 3.2e li:% 3d";
    string output = StringPrintf(kReportRowFormat,
                                 summary.iteration,
                                 summary.cost,
                                 summary.cost_change,
                                 summary.gradient_max_norm,
                                 summary.step_norm,
                                 summary.relative_decrease,
                                 summary.trust_region_radius,
                                 summary.eta,
                                 summary.linear_solver_iterations);
    if (log_to_stdout_) {
      cout << output << endl;
    } else {
      VLOG(1) << output;
    }
    return SOLVER_CONTINUE;
  }

 private:
  const bool log_to_stdout_;
};
```

`CRSMatrix`

class `CRSMatrix`

A compressed row sparse matrix used primarily for communicating the Jacobian matrix to the user.

int `CRSMatrix::num_rows`

Number of rows.

int `CRSMatrix::num_cols`

Number of columns.

vector<int> `CRSMatrix::rows`

`CRSMatrix::rows` is a `CRSMatrix::num_rows` + 1 sized array that points into the `CRSMatrix::cols` and `CRSMatrix::values` array.

vector<int> `CRSMatrix::cols`

`CRSMatrix::cols` contain as many entries as there are non-zeros in the matrix.

For each row `i`, `cols[rows[i]]` ... `cols[rows[i + 1] - 1]` are the indices of the non-zero columns of row `i`.

vector<int> `CRSMatrix::values`

`CRSMatrix::values` contain as many entries as there are non-zeros in the matrix.

For each row `i`, `values[rows[i]]` ... `values[rows[i + 1] - 1]` are the values of the non-zero columns of row `i`.

e.g., consider the 3x4 sparse matrix

```
0 10  0  4
0  2 -3  2
1  2  0  0
```

The three arrays will be:

```
         -row0-  ---row1---  -row2-
rows   = [ 0,      2,          5,     7]
cols   = [ 1,  3,  1,  2,  3,  0,  1]
values = [10,  4,  2, -3,  2,  1,  2]
```

`Solver::Summary`

class `Solver::Summary`

Summary of the various stages of the solver after termination.

string `Solver::Summary::BriefReport`() *const*

A brief one line description of the state of the solver after termination.

string `Solver::Summary::FullReport`() *const*

A full multiline description of the state of the solver after termination.

bool `Solver::Summary::IsSolutionUsable`() *const*

Whether the solution returned by the optimization algorithm can be relied on to be numerically sane. This will be the case if *Solver::Summary:termination_type* is set to *CONVERGENCE*, *USER_SUCCESS* or *NO_CONVERGENCE*, i.e., either the solver converged by meeting one of the convergence tolerances or because the user indicated that it had converged or it ran to the maximum number of iterations or time.

MinimizerType `Solver::Summary::minimizer_type`

Type of minimization algorithm used.

TerminationType `Solver::Summary::termination_type`

The cause of the minimizer terminating.

string `Solver::Summary::message`

Reason why the solver terminated.

double `Solver::Summary::initial_cost`

Cost of the problem (value of the objective function) before the optimization.

double `Solver::Summary::final_cost`

Cost of the problem (value of the objective function) after the optimization.

double `Solver::Summary::fixed_cost`

The part of the total cost that comes from residual blocks that were held fixed by the preprocessor because all the parameter blocks that they depend on were fixed.

vector<IterationSummary> `Solver::Summary::iterations`

`IterationSummary` for each minimizer iteration in order.

int `Solver::Summary::num_successful_steps`

Number of minimizer iterations in which the step was accepted. Unless `Solver::Options::use_non_monotonic_steps` is *true* this is also the number of steps in which the objective function value/cost went down.

int `Solver::Summary::num_unsuccessful_steps`

Number of minimizer iterations in which the step was rejected either because it did not reduce the cost enough or the step was not numerically valid.

int `Solver::Summary::num_inner_iteration_steps`

Number of times inner iterations were performed.

int `Solver::Summary::num_line_search_steps`

Total number of iterations inside the line search algorithm across all invocations. We call these iterations "steps" to distinguish them from the outer iterations of the line search and trust region minimizer algorithms which call the line search algorithm as a subroutine.

double `Solver::Summary::preprocessor_time_in_seconds`

Time (in seconds) spent in the preprocessor.

double `Solver::Summary::minimizer_time_in_seconds`

Time (in seconds) spent in the Minimizer.

double `Solver::Summary::postprocessor_time_in_seconds`

Time (in seconds) spent in the post processor.

double `Solver::Summary::total_time_in_seconds`

Time (in seconds) spent in the solver.

double `Solver::Summary::linear_solver_time_in_seconds`

Time (in seconds) spent in the linear solver computing the trust region step.

int `Solver::Summary::num_linear_solves`

Number of times the Newton step was computed by solving a linear system. This does not include linear solves used by inner iterations.

double `Solver::Summary::residual_evaluation_time_in_seconds`

Time (in seconds) spent evaluating the residual vector.

int `Solver::Summary::num_residual_evaluations`

Number of times only the residuals were evaluated.

double `Solver::Summary::jacobian_evaluation_time_in_seconds`

Time (in seconds) spent evaluating the Jacobian matrix.

int `Solver::Summary::num_jacobian_evaluations`

Number of times only the Jacobian and the residuals were evaluated.

double `Solver::Summary::inner_iteration_time_in_seconds`

Time (in seconds) spent doing inner iterations.

int `Solver::Summary::num_parameter_blocks`

Number of parameter blocks in the problem.

int `Solver::Summary::num_parameters`

Number of parameters in the problem.

int `Solver::Summary::num_effective_parameters`

Dimension of the tangent space of the problem (or the number of columns in the Jacobian for the problem). This is different from `Solver::Summary::num_parameters` if a parameter block is associated with a `LocalParameterization`.

int `Solver::Summary::num_residual_blocks`

Number of residual blocks in the problem.

int `Solver::Summary::num_residuals`

Number of residuals in the problem.

int `Solver::Summary::num_parameter_blocks_reduced`

Number of parameter blocks in the problem after the inactive and constant parameter blocks have been removed. A parameter block is inactive if no residual block refers to it.

int `Solver::Summary::num_parameters_reduced`

Number of parameters in the reduced problem.

int `Solver::Summary::num_effective_parameters_reduced`

Dimension of the tangent space of the reduced problem (or the number of columns in the Jacobian for the reduced problem). This is different from `Solver::Summary::num_parameters_reduced` if a parameter block in the reduced problem is associated with a `LocalParameterization`.

int `Solver::Summary::num_residual_blocks_reduced`

Number of residual blocks in the reduced problem.

int `Solver::Summary::num_residuals_reduced`

Number of residuals in the reduced problem.

int `Solver::Summary::num_threads_given`

Number of threads specified by the user for Jacobian and residual evaluation.

int `Solver::Summary::num_threads_used`

Number of threads actually used by the solver for Jacobian and residual evaluation. This number is not equal to `Solver::Summary::num_threads_given` if none of *OpenMP*, *TBB* or *CXX11_THREADS* is available.

int `Solver::Summary::num_linear_solver_threads_given`

This field is deprecated and is scheduled to be removed in 1.15.0. `Solver::Summary::num_threads_given` should be used instead. In the interim the value of this field will be the same as `Solver::Summary::num_threads_given`.

Number of threads requested by the user for solving the trust region problem.

int `Solver::Summary::num_linear_solver_threads_used`

This field is deprecated and is scheduled to be removed in 1.15.0. `Solver::Summary::num_threads_used` should be used instead. In the interim the value of this field will be the same as `Solver::Summary::num_threads_used`.

Number of threads actually used by the solver for solving the trust region problem. This number is not equal to `Solver::Summary::num_linear_solver_threads_given` if none of *OpenMP*, *TBB* or *CXX11_THREADS* is available.

LinearSolverType `Solver::Summary::linear_solver_type_given`

Type of the linear solver requested by the user.

LinearSolverType `Solver::Summary::linear_solver_type_used`

Type of the linear solver actually used. This may be different from `Solver::Summary::linear_solver_type_given` if Ceres determines that the problem structure is not compatible with the linear solver requested or if the linear solver requested by the user is not available, e.g. The user requested *SPARSE_NORMAL_CHOLESKY* but no sparse linear algebra library was available.

vector<int> `Solver::Summary::linear_solver_ordering_given`

Size of the elimination groups given by the user as hints to the linear solver.

vector<int> `Solver::Summary::linear_solver_ordering_used`

Size of the parameter groups used by the solver when ordering the columns of the Jacobian. This maybe different from `Solver::Summary::linear_solver_ordering_given` if the user left `Solver::Summary::linear_solver_ordering_given` blank and asked for an automatic ordering, or if the problem contains some constant or inactive parameter blocks.

std::string `Solver::Summary::schur_structure_given`

For Schur type linear solvers, this string describes the template specialization which was detected in the problem and should be used.

std::string `Solver::Summary::schur_structure_used`

For Schur type linear solvers, this string describes the template specialization that was actually instantiated and used. The reason this will be different from `Solver::Summary::schur_structure_given` is because the corresponding template specialization does not exist.

Template specializations can be added to ceres by editing `internal/ceres/generate_template_specializations.py`

bool `Solver::Summary::inner_iterations_given`

True if the user asked for inner iterations to be used as part of the optimization.

bool `Solver::Summary::inner_iterations_used`

True if the user asked for inner iterations to be used as part of the optimization and the problem structure was such that they were actually performed. For example, in a problem with just one parameter block, inner iterations are not performed.

vector<int> `inner_iteration_ordering_given`

Size of the parameter groups given by the user for performing inner iterations.

vector<int> `inner_iteration_ordering_used`

Size of the parameter groups given used by the solver for performing inner iterations. This maybe different from `Solver::Summary::inner_iteration_ordering_given` if the user left `Solver::Summary::inner_iteration_ordering_given` blank and asked for an automatic ordering, or if the problem contains some constant or inactive parameter blocks.

PreconditionerType `Solver::Summary::preconditioner_type_given`

Type of the preconditioner requested by the user.

PreconditionerType `Solver::Summary::preconditioner_type_used`

Type of the preconditioner actually used. This may be different from `Solver::Summary::linear_solver_type_given` if Ceres determines that the problem structure is not compatible with the linear solver requested or if the linear solver requested by the user is not available.

VisibilityClusteringType `Solver::Summary::visibility_clustering_type`

Type of clustering algorithm used for visibility based preconditioning. Only meaningful when the `Solver::Summary::preconditioner_type` is `CLUSTER_JACOBI` or `CLUSTER_TRIDIAGONAL`.

TrustRegionStrategyType `Solver::Summary::trust_region_strategy_type`

Type of trust region strategy.

DoglegType `Solver::Summary::dogleg_type`

Type of dogleg strategy used for solving the trust region problem.

DenseLinearAlgebraLibraryType `Solver::Summary::dense_linear_algebra_library_type`

Type of the dense linear algebra library used.

SparseLinearAlgebraLibraryType `Solver::Summary::sparse_linear_algebra_library_type`

Type of the sparse linear algebra library used.

LineSearchDirectionType `Solver::Summary::line_search_direction_type`

Type of line search direction used.

LineSearchType `Solver::Summary::line_search_type`

Type of the line search algorithm used.

LineSearchInterpolationType `Solver::Summary::line_search_interpolation_type`

When performing line search, the degree of the polynomial used to approximate the objective function.

NonlinearConjugateGradientType `Solver::Summary::nonlinear_conjugate_gradient_type`

If the line search direction is *NONLINEAR_CONJUGATE_GRADIENT*, then this indicates the particular variant of non-linear conjugate gradient used.

int `Solver::Summary::max_lbfgs_rank`

If the type of the line search direction is *LBFGS*, then this indicates the rank of the Hessian approximation.