Modeling Non-linear Least Squares

Introduction

Ceres solver consists of two distinct parts. A modeling API which provides a rich set of tools to construct an optimization problem one term at a time and a solver API that controls the minimization algorithm. This chapter is devoted to the task of modeling optimization problems using Ceres. Solving Non-linear Least Squares discusses the various ways in which an optimization problem can be solved using Ceres.

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

(1)¶

$$\begin{split}\min_{\mathbf{x}} &\quad \frac{1}{2}\sum_{i} \rho_i\left(\left\|f_i\left(x_{i_1}, ... ,x_{i_k}\right)\right\|^2\right) \\ \text{s.t.} &\quad l_j \le x_j \le u_j\end{split}$$

In Ceres parlance, the expression $\rho_i\left(\left\|f_i\left(x_{i_1},...,x_{i_k}\right)\right\|^2\right)$ is known as a **residual block**, where $f_i(\cdot)$ is a `CostFunction` that depends on the **parameter blocks** $\left\{x_{i_1},... , x_{i_k}\right\}$.

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 scalars as a **parameter block**. Of course a parameter block can be just a single scalar too.

ρ_i is a `LossFunction`. A `LossFunction` is a scalar valued function that is used to reduce the influence of outliers on the solution of non-linear least squares problems.

l_j and u_j are lower and upper bounds on the parameter block x_j.

As a special case, when $\rho_i(x) = x$, i.e., the identity function, and $l_j = -\infty$ and $u_j = \infty$ we get the more familiar unconstrained non-linear least squares problem.

(2)¶

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

`CostFunction`

For each term in the objective function, a `CostFunction` is responsible for computing a vector of residuals and Jacobian matrices. Concretely, consider a function $f\left(x_{1},...,x_{k}\right)$ that depends on parameter blocks $\left[x_{1}, ... , x_{k}\right]$.

Then, given $\left[x_{1}, ... , x_{k}\right]$, `CostFunction` is responsible for computing the vector $f\left(x_{1},...,x_{k}\right)$ and the Jacobian matrices

$$J_i = D_i f(x_1, ..., x_k) \quad \forall i \in \{1, \ldots, k\}$$

class **CostFunction**

```
class CostFunction {
 public:
  virtual bool Evaluate(double const* const* parameters,
                        double* residuals,
                        double** jacobians) = 0;
  const vector<int32>& parameter_block_sizes();
  int num_residuals() const;

 protected:
  vector<int32>* mutable_parameter_block_sizes();
  void set_num_residuals(int num_residuals);
};
```

The signature of the `CostFunction` (number and sizes of input parameter blocks and number of outputs) is stored in `CostFunction::parameter_block_sizes_` and `CostFunction::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()`.

bool CostFunction::Evaluate(double *const* **const* **parameters*, double **residuals*, double ***jacobians*)

Compute the residual vector and the Jacobian matrices.

`parameters` is an array of arrays of size `CostFunction::parameter_block_sizes_.size()` and `parameters[i]` is an array of size `parameter_block_sizes_[i]` that contains the ith parameter block that the `CostFunction` depends on.

`parameters` is never `nullptr`.

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

`residuals` is never `nullptr`.

`jacobians` is an array of arrays of size `CostFunction::parameter_block_sizes_.size()`.

If `jacobians` is `nullptr`, the user is only expected to compute the residuals.

`jacobians[i]` is a row-major array of size `num_residuals x parameter_block_sizes_[i]`.

If `jacobians[i]` is **not** `nullptr`, the user is required to compute the Jacobian of the residual vector with respect to `parameters[i]` and store it in this array, i.e.

`jacobians[i][r * parameter_block_sizes_[i] + c]` = $\frac{\partial \text{residual}[r]}{\partial \text{parameters}[i][c]}$

If `jacobians[i]` is `nullptr`, then this computation can be skipped. This is the case when the corresponding parameter block is marked constant.

The return value indicates whether the computation of the residuals and/or jacobians was successful or not. This can be used to communicate numerical failures in Jacobian computations for instance.

`SizedCostFunction`

class **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), `SizeCostFunction` can be used where these values can be specified as template parameters and the user only needs to implement `CostFunction::Evaluate()`.

```
template<int kNumResiduals, int... Ns>
class SizedCostFunction : public CostFunction {
 public:
  virtual bool Evaluate(double const* const* parameters,
                        double* residuals,
                        double** jacobians) const = 0;
};
```

`AutoDiffCostFunction`

class **AutoDiffCostFunction**

Defining a `CostFunction` or a `SizedCostFunction` can be a tedious and error prone especially when computing derivatives. To this end Ceres provides automatic differentiation.

```
template <typename CostFunctor,
       int kNumResiduals,  // Number of residuals, or ceres::DYNAMIC.
       int... Ns>          // Size of each parameter block
class AutoDiffCostFunction : public
SizedCostFunction<kNumResiduals, Ns> {
 public:
  AutoDiffCostFunction(CostFunctor* functor, ownership = TAKE_OWNERSHIP);
  // Ignore the template parameter kNumResiduals and use
  // num_residuals instead.
  AutoDiffCostFunction(CostFunctor* functor,
                       int num_residuals,
                       ownership = TAKE_OWNERSHIP);
};
```

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 MyScalarCostFunctor {
  MyScalarCostFunctor(double k): k_(k) {}

  template <typename T>
  bool operator()(const T* const x , const T* const y, T* e) const {
    e[0] = 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<MyScalarCostFunctor, 1, 2, 2>(
        new MyScalarCostFunctor(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.

By default `AutoDiffCostFunction` will take ownership of the cost functor pointer passed to it, ie. will call *delete* on the cost functor when the `AutoDiffCostFunction` itself is deleted. However, this may be undesirable in certain cases, therefore it is also possible to specify `DO_NOT_TAKE_OWNERSHIP` as a second argument in the constructor, while passing a pointer to a cost functor which does not need to be deleted by the AutoDiffCostFunction. For example:

```
MyScalarCostFunctor functor(1.0)
CostFunction* cost_function
    = new AutoDiffCostFunction<MyScalarCostFunctor, 1, 2, 2>(
        &functor, DO_NOT_TAKE_OWNERSHIP);
```

`AutoDiffCostFunction` also supports cost functions with a runtime-determined number of residuals. For example:

```
CostFunction* cost_function
    = new AutoDiffCostFunction<MyScalarCostFunctor, DYNAMIC, 2, 2>(
        new CostFunctorWithDynamicNumResiduals(1.0),   ^     ^  ^
        runtime_number_of_residuals); <----+           |     |  |
                                           |           |     |  |
                                           |           |     |  |
          Actual number of residuals ------+           |     |  |
          Indicate dynamic number of residuals --------+     |  |
          Dimension of x ------------------------------------+  |
          Dimension of y ---------------------------------------+
```

WARNING 1 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.

`DynamicAutoDiffCostFunction`

class **`DynamicAutoDiffCostFunction`**

`AutoDiffCostFunction` requires that the number of parameter blocks and their sizes be known at compile time. In a number of applications, this is not enough e.g., Bezier curve fitting, Neural Network training etc.

```
template <typename CostFunctor, int Stride = 4>
class DynamicAutoDiffCostFunction : public CostFunction {
};
```

In such cases `DynamicAutoDiffCostFunction` can be used. Like `AutoDiffCostFunction` the user must define a templated functor, but the signature of the functor differs slightly. The expected interface for the cost functors is:

```
struct MyCostFunctor {
  template<typename T>
  bool operator()(T const* const* parameters, T* residuals) const {
  }
}
```

Since the sizing of the parameters is done at runtime, you must also specify the sizes after creating the dynamic autodiff cost function. For example:

```
DynamicAutoDiffCostFunction<MyCostFunctor, 4>* cost_function =
  new DynamicAutoDiffCostFunction<MyCostFunctor, 4>(
    new MyCostFunctor());
cost_function->AddParameterBlock(5);
cost_function->AddParameterBlock(10);
cost_function->SetNumResiduals(21);
```

Under the hood, the implementation evaluates the cost function multiple times, computing a small set of the derivatives (four by default, controlled by the `Stride` template parameter) with each pass. There is a performance tradeoff with the size of the passes; Smaller sizes are more cache efficient but result in larger number of passes, and larger stride lengths can destroy cache-locality while reducing the number of passes over the cost function. The optimal value depends on the number and sizes of the various parameter blocks.

As a rule of thumb, try using `AutoDiffCostFunction` before you use `DynamicAutoDiffCostFunction`.

`NumericDiffCostFunction`

class **`NumericDiffCostFunction`**

In some cases, its not possible to define a templated cost functor, for example when the evaluation of the residual involves a call to a library function that you do not have control over. In such a situation, numerical differentiation can be used.

Note

TODO(sameeragarwal): Add documentation for the constructor and for NumericDiffOptions. Update DynamicNumericDiffOptions in a similar manner.

```
template <typename CostFunctor,
          NumericDiffMethodType method = CENTRAL,
          int kNumResiduals,  // Number of residuals, or ceres::DYNAMIC.
          int... Ns>          // Size of each parameter block.
class NumericDiffCostFunction : public
SizedCostFunction<kNumResiduals, Ns> {
};
```

To get a numerically differentiated `CostFunction`, you must define a class with a `operator()` (a functor) that computes the residuals. The functor must write the computed value in the last argument (the only non-`const` one) and return `true` to indicate success. Please see `CostFunction` for details on how the return value may be used to impose simple constraints on the parameter block. e.g., an object of the form

```
struct ScalarFunctor {
 public:
  bool operator()(const double* const x1,
                  const double* const x2,
                  double* residuals) const;
}
```

For example, consider a scalar error $e = k - x'y$, where both x and y are two-dimensional column vector parameters, the prime sign indicates transposition, 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 $x'y$ might be the model expectation for a series of measurements, where there is an instance of the cost function for each measurement k.

To write an numerically-differentiable class:*CostFunction* for the above model, first define the object

```
class MyScalarCostFunctor {
  MyScalarCostFunctor(double k): k_(k) {}

  bool operator()(const double* const x,
                  const double* const y,
                  double* residuals) const {
    residuals[0] = 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 `double` s. 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, the residual is a scalar, so only `residuals[0]` is set.

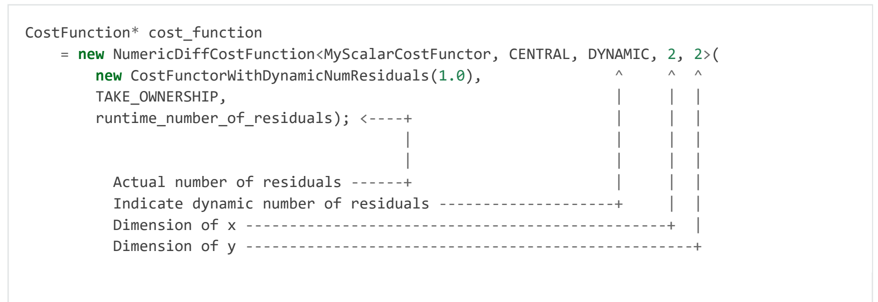
Then given this class definition, the numerically differentiated `CostFunction` with central differences used for computing the derivative can be constructed as follows.

```
CostFunction* cost_function
    = new NumericDiffCostFunction<MyScalarCostFunctor, CENTRAL, 1, 2, 2>(
        new MyScalarCostFunctor(1.0));                    ^     ^  ^  ^
                                                          |     |  |  |
                              Finite Differencing Scheme -+     |  |  |
                              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 `MyScalarCostFunctor`, `1, 2, 2`, describe the functor as computing a 1-dimensional output from two arguments, both 2-dimensional.

NumericDiffCostFunction also supports cost functions with a runtime-determined number of residuals. For example:



```
CostFunction* cost_function
    = new NumericDiffCostFunction<MyScalarCostFunctor, CENTRAL, DYNAMIC, 2, 2>(
        new CostFunctorWithDynamicNumResiduals(1.0),               ^     ^  ^
        TAKE_OWNERSHIP,                                            |     |  |
        runtime_number_of_residuals); <----+                       |     |  |
                                           |                       |     |  |
                                           |                       |     |  |
          Actual number of residuals ------+                       |     |  |
          Indicate dynamic number of residuals --------------------+     |  |
          Dimension of x ------------------------------------------------+  |
          Dimension of y ---------------------------------------------------+
```

There are three available numeric differentiation schemes in ceres-solver:

The `FORWARD` difference method, which approximates $f'(x)$ by computing $\frac{f(x+h)-f(x)}{h}$, computes the cost function one additional time at $x + h$. It is the fastest but least accurate method.

The `CENTRAL` difference method is more accurate at the cost of twice as many function evaluations than forward difference, estimating $f'(x)$ by computing $\frac{f(x+h)-f(x-h)}{2h}$.

The `RIDDERS` difference method[Ridders]_ is an adaptive scheme that estimates derivatives by performing multiple central differences at varying scales. Specifically, the algorithm starts at a certain h and as the derivative is estimated, this step size decreases. To conserve function evaluations and estimate the derivative error, the method performs Richardson extrapolations between the tested step sizes. The algorithm exhibits considerably higher accuracy, but does so by additional evaluations of the cost function.

Consider using `CENTRAL` differences to begin with. Based on the results, either try forward difference to improve performance or Ridders' method to improve accuracy.

WARNING A common beginner's error when first using `NumericDiffCostFunction` 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*. In the example above, that would be `<MyScalarCostFunctor, 1, 2>`, which is missing the last `2` argument. Please be careful when setting the size parameters.

Numeric Differentiation & LocalParameterization

If your cost function depends on a parameter block that must lie on a manifold and the functor cannot be evaluated for values of that parameter block not on the manifold then you may have problems numerically differentiating such functors.

This is because numeric differentiation in Ceres is performed by perturbing the individual coordinates of the parameter blocks that a cost functor depends on. In doing so, we assume that the parameter blocks live in an Euclidean space and ignore the structure of manifold that they live As a result some of the perturbations may not lie on the manifold corresponding to the parameter block.

For example consider a four dimensional parameter block that is interpreted as a unit Quaternion. Perturbing the coordinates of this parameter block will violate the unit norm property of the parameter block.

Fixing this problem requires that `NumericDiffCostFunction` be aware of the `LocalParameterization` associated with each parameter block and only generate perturbations in the local tangent space of each parameter block.

For now this is not considered to be a serious enough problem to warrant changing the `NumericDiffCostFunction` API. Further, in most cases it is relatively straightforward to project a point off the manifold back onto the manifold before using it in the functor. For example in case of the Quaternion, normalizing the 4-vector before using it does the trick.

Alternate Interface

For a variety of reasons, including compatibility with legacy code, `NumericDiffCostFunction` can also take `CostFunction` objects as input. The following describes how.

To get a numerically differentiated cost function, define a subclass of `CostFunction` such that the `CostFunction::Evaluate()` function ignores the `jacobians` parameter. The numeric differentiation wrapper will fill in the jacobian parameter if necessary by repeatedly calling the `CostFunction::Evaluate()` 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 `CostFunction` interface.

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.

`DynamicNumericDiffCostFunction`

class **`DynamicNumericDiffCostFunction`**

Like `AutoDiffCostFunction` `NumericDiffCostFunction` requires that the number of parameter blocks and their sizes be known at compile time. In a number of applications, this is not enough.

```
template <typename CostFunctor, NumericDiffMethodType method = CENTRAL>
class DynamicNumericDiffCostFunction : public CostFunction {
};
```

In such cases when numeric differentiation is desired, `DynamicNumericDiffCostFunction` can be used.

Like `NumericDiffCostFunction` the user must define a functor, but the signature of the functor differs slightly. The expected interface for the cost functors is:

```
struct MyCostFunctor {
  bool operator()(double const* const* parameters, double* residuals) const {
  }
}
```

Since the sizing of the parameters is done at runtime, you must also specify the sizes after creating the dynamic numeric diff cost function. For example:

```
DynamicNumericDiffCostFunction<MyCostFunctor>* cost_function =
  new DynamicNumericDiffCostFunction<MyCostFunctor>(new MyCostFunctor);
cost_function->AddParameterBlock(5);
cost_function->AddParameterBlock(10);
cost_function->SetNumResiduals(21);
```

As a rule of thumb, try using `NumericDiffCostFunction` before you use `DynamicNumericDiffCostFunction`.

WARNING The same caution about mixing local parameterizations with numeric differentiation applies as is the case with `NumericDiffCostFunction`.

`CostFunctionToFunctor`

class **`CostFunctionToFunctor`**

`CostFunctionToFunctor` is an adapter class that allows users to use `CostFunction` objects in templated functors which are to be used for automatic differentiation. This allows the user to seamlessly mix analytic, numeric and automatic differentiation.

For example, let us assume that

```
class IntrinsicProjection : public SizedCostFunction<2, 5, 3> {
  public:
    IntrinsicProjection(const double* observation);
    virtual bool Evaluate(double const* const* parameters,
                          double* residuals,
                          double** jacobians) const;
};
```

is a `CostFunction` that implements the projection of a point in its local coordinate system onto its image plane and subtracts it from the observed point projection. It can compute its residual and either via analytic or numerical differentiation can compute its jacobians.

Now we would like to compose the action of this `CostFunction` with the action of camera extrinsics, i.e., rotation and translation. Say we have a templated function

```
template<typename T>
void RotateAndTranslatePoint(const T* rotation,
                             const T* translation,
                             const T* point,
                             T* result);
```

Then we can now do the following,

```
struct CameraProjection {
  CameraProjection(double* observation)
  : intrinsic_projection_(new IntrinsicProjection(observation)) {
  }

  template <typename T>
  bool operator()(const T* rotation,
                  const T* translation,
                  const T* intrinsics,
                  const T* point,
                  T* residual) const {
    T transformed_point[3];
    RotateAndTranslatePoint(rotation, translation, point, transformed_point);

    // Note that we call intrinsic_projection_, just like it was
    // any other templated functor.
    return intrinsic_projection_(intrinsics, transformed_point, residual);
  }

 private:
  CostFunctionToFunctor<2,5,3> intrinsic_projection_;
};
```

Note that `CostFunctionToFunctor` takes ownership of the `CostFunction` that was passed in to the constructor.

In the above example, we assumed that `IntrinsicProjection` is a `CostFunction` capable of evaluating its value and its derivatives. Suppose, if that were not the case and `IntrinsicProjection` was defined as follows:

```
struct IntrinsicProjection {
  IntrinsicProjection(const double* observation) {
    observation_[0] = observation[0];
    observation_[1] = observation[1];
  }

  bool operator()(const double* calibration,
                  const double* point,
                  double* residuals) const {
    double projection[2];
    ThirdPartyProjectionFunction(calibration, point, projection);
    residuals[0] = observation_[0] - projection[0];
    residuals[1] = observation_[1] - projection[1];
    return true;
  }
  double observation_[2];
};
```

Here `ThirdPartyProjectionFunction` is some third party library function that we have no control over. So this function can compute its value and we would like to use numeric differentiation to compute its derivatives. In this case we can use a combination of `NumericDiffCostFunction` and `CostFunctionToFunctor` to get the job done.

```
struct CameraProjection {
  CameraProjection(double* observation)
      : intrinsic_projection_(
            new NumericDiffCostFunction<IntrinsicProjection, CENTRAL, 2, 5, 3>(
                new IntrinsicProjection(observation))) {}

  template <typename T>
  bool operator()(const T* rotation,
                  const T* translation,
                  const T* intrinsics,
                  const T* point,
                  T* residuals) const {
    T transformed_point[3];
    RotateAndTranslatePoint(rotation, translation, point, transformed_point);
    return intrinsic_projection_(intrinsics, transformed_point, residuals);
  }

 private:
  CostFunctionToFunctor<2, 5, 3> intrinsic_projection_;
};
```

DynamicCostFunctionToFunctor

class **DynamicCostFunctionToFunctor**

`DynamicCostFunctionToFunctor` provides the same functionality as `CostFunctionToFunctor` for cases where the number and size of the parameter vectors and residuals are not known at compile-time. The API provided by `DynamicCostFunctionToFunctor` matches what would be expected by `DynamicAutoDiffCostFunction`, i.e. it provides a templated functor of this form:

```
template<typename T>
bool operator()(T const* const* parameters, T* residuals) const;
```

Similar to the example given for `CostFunctionToFunctor`, let us assume that

```
class IntrinsicProjection : public CostFunction {
  public:
    IntrinsicProjection(const double* observation);
    virtual bool Evaluate(double const* const* parameters,
                          double* residuals,
                          double** jacobians) const;
};
```

is a `CostFunction` that projects a point in its local coordinate system onto its image plane and subtracts it from the observed point projection.

Using this `CostFunction` in a templated functor would then look like this:

```
struct CameraProjection {
  CameraProjection(double* observation)
      : intrinsic_projection_(new IntrinsicProjection(observation)) {
  }

  template <typename T>
  bool operator()(T const* const* parameters,
                  T* residual) const {
    const T* rotation = parameters[0];
    const T* translation = parameters[1];
    const T* intrinsics = parameters[2];
    const T* point = parameters[3];

    T transformed_point[3];
    RotateAndTranslatePoint(rotation, translation, point, transformed_point);

    const T* projection_parameters[2];
    projection_parameters[0] = intrinsics;
    projection_parameters[1] = transformed_point;
    return intrinsic_projection_(projection_parameters, residual);
  }

 private:
  DynamicCostFunctionToFunctor intrinsic_projection_;
};
```

Like `CostFunctionToFunctor`, `DynamicCostFunctionToFunctor` takes ownership of the `CostFunction` that was passed in to the constructor.

`ConditionedCostFunction`

class **`ConditionedCostFunction`**

This class allows you to apply different conditioning to the residual values of a wrapped cost function. An example where this is useful is where you have an existing cost function that produces N values, but you want the total cost to be something other than just the sum of these squared values - maybe you want to apply a different scaling to some values, to change their contribution to the cost.

Usage:

```
//  my_cost_function produces N residuals
CostFunction* my_cost_function = ...
CHECK_EQ(N, my_cost_function->num_residuals());
vector<CostFunction*> conditioners;

//  Make N 1x1 cost functions (1 parameter, 1 residual)
CostFunction* f_1 = ...
conditioners.push_back(f_1);

CostFunction* f_N = ...
conditioners.push_back(f_N);
ConditionedCostFunction* ccf =
  new ConditionedCostFunction(my_cost_function, conditioners);
```

Now `ccf` 's `residual[i]` (i=0..N-1) will be passed though the ith conditioner.

```
ccf_residual[i] = f_i(my_cost_function_residual[i])
```

and the Jacobian will be affected appropriately.

`GradientChecker`

class **`GradientChecker`**

This class compares the Jacobians returned by a cost function against derivatives estimated using finite differencing. It is meant as a tool for unit testing, giving you more fine-grained control than the check_gradients option in the solver options.

The condition enforced is that

$$\forall i, j : \frac{J_{ij} - J'_{ij}}{max_{ij}(J_{ij} - J'_{ij})} < r$$

where J_{ij} is the jacobian as computed by the supplied cost function (by the user) multiplied by the local parameterization Jacobian, J'_{ij} is the jacobian as computed by finite differences, multiplied by the local parameterization Jacobian as well, and r is the relative precision.

Usage:

```
//  my_cost_function takes two parameter blocks. The first has a local
//  parameterization associated with it.
CostFunction* my_cost_function = ...
LocalParameterization* my_parameterization = ...
NumericDiffOptions numeric_diff_options;

std::vector<LocalParameterization*> local_parameterizations;
local_parameterizations.push_back(my_parameterization);
local_parameterizations.push_back(nullptr);

std::vector parameter1;
std::vector parameter2;
// Fill parameter 1 & 2 with test data...

std::vector<double*> parameter_blocks;
parameter_blocks.push_back(parameter1.data());
parameter_blocks.push_back(parameter2.data());

GradientChecker gradient_checker(my_cost_function,
    local_parameterizations, numeric_diff_options);
GradientCheckResults results;
if (!gradient_checker.Probe(parameter_blocks.data(), 1e-9, &results) {
  LOG(ERROR) << "An error has occurred:\n" << results.error_log;
}
```

`NormalPrior`

class **NormalPrior**

```
class NormalPrior: public CostFunction {
 public:
  // Check that the number of rows in the vector b are the same as the
  // number of columns in the matrix A, crash otherwise.
  NormalPrior(const Matrix& A, const Vector& b);

  virtual bool Evaluate(double const* const* parameters,
                        double* residuals,
                        double** jacobians) const;
 };
```

Implements a cost function of the form

$$cost(x) = ||A(x - b)||^2$$

where, the matrix A and the vector b are fixed and x is the variable. In case the user is interested in implementing a cost function of the form

$$cost(x) = (x - \mu)^T S^{-1} (x - \mu)$$

where, μ is a vector and S is a covariance matrix, then, $A = S^{-1/2}$, i.e the matrix A is the square root of the inverse of the covariance, also known as the stiffness matrix. There are however no restrictions on the shape of A. It is free to be rectangular, which would be the case if the covariance matrix S is rank deficient.

`LossFunction`

class **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 `LossFunction::Evaluate()`, which given a non-negative scalar `s`, computes

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

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

Most sane choices of ρ satisfy:

$$\rho(0) = 0$$
$$\rho'(0) = 1$$
$$\rho'(s) < 1 \text{ in the outlier region}$$
$$\rho''(s) < 0 \text{ in the outlier region}$$

so that they mimic the squared cost for small residuals.

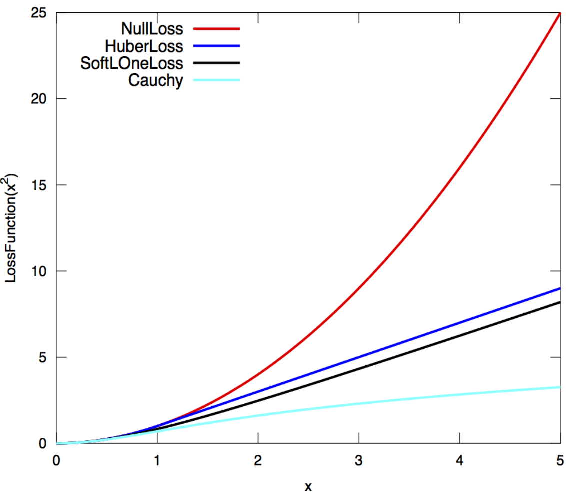
Scaling

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.

Instances

Ceres includes a number of predefined loss functions. For simplicity we described their unscaled versions. The figure below illustrates their shape graphically. More details can be found in `include/ceres/loss_function.h`.



Shape of the various common loss functions.

class TrivialLoss

$$\rho(s) = s$$

class HuberLoss

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

class SoftLOneLoss

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

class CauchyLoss

$$\rho(s) = \log(1 + s)$$

class ArctanLoss

$$\rho(s) = \arctan(s)$$

class TolerantLoss

$$\rho(s, a, b) = b\log(1 + e^{(s-a)/b}) - b\log(1 + e^{-a/b})$$

class ComposedLoss

Given two loss functions `f` and `g`, implements the loss function `h(s) = f(g(s))`.

```
class ComposedLoss : public LossFunction {
 public:
  explicit ComposedLoss(const LossFunction* f,
                        Ownership ownership_f,
                        const LossFunction* g,
                        Ownership ownership_g);
};
```

class **ScaledLoss**

Sometimes you want to simply scale the output value of the robustifier. For example, you might want to weight different error terms differently (e.g., weight pixel reprojection errors differently from terrain errors).

Given a loss function $\rho(s)$ and a scalar a, `ScaledLoss` implements the function $a\rho(s)$.

Since we treat a `nullptr` Loss function as the Identity loss function, rho = `nullptr`: is a valid input and will result in the input being scaled by a. This provides a simple way of implementing a scaled ResidualBlock.

class **LossFunctionWrapper**

Sometimes after the optimization problem has been constructed, we wish to mutate the scale of the loss function. For example, when performing estimation from data which has substantial outliers, convergence can be improved by starting out with a large scale, optimizing the problem and then reducing the scale. This can have better convergence behavior than just using a loss function with a small scale.

This templated class allows the user to implement a loss function whose scale can be mutated after an optimization problem has been constructed, e.g,

```
Problem problem;

// Add parameter blocks

CostFunction* cost_function =
    new AutoDiffCostFunction < UW_Camera_Mapper, 2, 9, 3>(
        new UW_Camera_Mapper(feature_x, feature_y));

LossFunctionWrapper* loss_function(new HuberLoss(1.0), TAKE_OWNERSHIP);
problem.AddResidualBlock(cost_function, loss_function, parameters);

Solver::Options options;
Solver::Summary summary;
Solve(options, &problem, &summary);

loss_function->Reset(new HuberLoss(1.0), TAKE_OWNERSHIP);
Solve(options, &problem, &summary);
```

Theory

Let us consider a problem with a single parameter block.

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

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

$$\begin{aligned} g(x) &= \rho' J^\top(x) f(x) \\ H(x) &= J^\top(x)\left(\rho' + 2\rho'' f(x) f^\top(x)\right) J(x) \end{aligned}$$

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 robustified Gauss-Newton step corresponds to an ordinary linear least squares problem.

Let α be a root of

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

Then, define the rescaled residual and Jacobian as

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

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

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

`LocalParameterization`

class **`LocalParameterization`**

In many optimization problems, especially sensor fusion problems, one has to model quantities that live in spaces known as Manifolds , for example the rotation/orientation of a sensor that is represented by a Quaternion.

Manifolds are spaces, which locally look like Euclidean spaces. More precisely, at each point on the manifold there is a linear space that is tangent to the manifold. It has dimension equal to the intrinsic dimension of the manifold itself, which is less than or equal to the ambient space in which the manifold is embedded.

For example, the tangent space to a point on a sphere in three dimensions is the two dimensional plane that is tangent to the sphere at that point. There are two reasons tangent spaces are interesting:

1. They are Euclidean spaces, so the usual vector space operations apply there, which makes numerical operations easy.
2. Movement in the tangent space translate into movements along the manifold. Movements perpendicular to the tangent space do not translate into movements on the manifold.

   Returning to our sphere example, moving in the 2 dimensional plane tangent to the sphere and projecting back onto the sphere will move you away from the point you started from but moving along the normal at the same point and the projecting back onto the sphere brings you back to the point.

Besides the mathematical niceness, modeling manifold valued quantities correctly and paying attention to their geometry has practical benefits too:

1. It naturally constrains the quantity to the manifold through out the optimization. Freeing the user from hacks like *quaternion normalization*.
2. It reduces the dimension of the optimization problem to its *natural* size. For example, a quantity restricted to a line, is a one dimensional object regardless of the dimension of the ambient space in which this line lives.

   Working in the tangent space reduces not just the computational complexity of the optimization algorithm, but also improves the numerical behaviour of the algorithm.

A basic operation one can perform on a manifold is the $\boxplus$ operation that computes the result of moving along delta in the tangent space at x, and then projecting back onto the manifold that x belongs to. Also known as a *Retraction*, $\boxplus$ is a generalization of vector addition in Euclidean spaces. Formally, $\boxplus$ is a smooth map from a manifold $\mathcal{M}$ and its tangent space $T_\mathcal{M}$ to the manifold $\mathcal{M}$ that obeys the identity

$$\boxplus(x, 0) = x, \quad \forall x.$$

That is, it ensures that the tangent space is *centered* at x and the zero vector is the identity element. For more see [Hertzberg] and section A.6.9 of [HartleyZisserman].

Let us consider two examples:

The Euclidean space R^n is the simplest example of a manifold. It has dimension n (and so does its tangent space) and $\boxplus$ is the familiar vector sum operation.

$$\boxplus(x, \Delta) = x + \Delta$$

A more interesting case is $SO(3)$, the special orthogonal group in three dimensions - the space of 3x3 rotation matrices. $SO(3)$ is a three dimensional manifold embedded in R^9 or $R^{3\times 3}$.

$\boxplus$ on $SO(3)$ is defined using the *Exponential* map, from the tangent space (R^3) to the manifold. The Exponential map Exp is defined as:

$$\text{Exp}([p,q,r]) = \begin{bmatrix} \cos\theta + cp^2 & -sr + cpq & sq + cpr \\ sr + cpq & \cos\theta + cq^2 & -sp + cqr \\ -sq + cpr & sp + cqr & \cos\theta + cr^2 \end{bmatrix}$$

where,

$$\theta = \sqrt{p^2 + q^2 + r^2}, s = \frac{\sin\theta}{\theta}, c = \frac{1-\cos\theta}{\theta^2}.$$

Then,

$$\boxplus(x, \Delta) = x\,\text{Exp}(\Delta)$$

The `LocalParameterization` interface allows the user to define and associate with parameter blocks the manifold that they belong to. It does so by defining the `Plus` ($\boxplus$) operation and its derivative with respect to Δ at $\Delta = 0$.

```
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 bool MultiplyByJacobian(const double* x,
                                  const int num_rows,
                                  const double* global_matrix,
                                  double* local_matrix) const;
  virtual int GlobalSize() const = 0;
  virtual int LocalSize() const = 0;
};
```

int `LocalParameterization::GlobalSize()`

The dimension of the ambient space in which the parameter block x lives.

int `LocalParameterization::LocalSize()`

The size of the tangent space that Δ lives in.

bool `LocalParameterization::Plus`(*const double *x, const double *delta, double *x_plus_delta)const*

`LocalParameterization::Plus()` implements $\boxplus(x, \Delta)$.

bool `LocalParameterization::ComputeJacobian`(*const double *x, double *jacobian)const*

Computes the Jacobian matrix

$$J = D_2 \boxplus (x, 0)$$

in row major form.

bool `MultiplyByJacobian`(*const double *x, const int num_rows, const double *global_matrix, double *local_matrix)const*

`local_matrix = global_matrix * jacobian`

`global_matrix` is a `num_rows x GlobalSize` row major matrix. `local_matrix` is a `num_rows x LocalSize` row major matrix. `jacobian` is the matrix returned by `LocalParameterization::ComputeJacobian()` at x.

This is only used by `GradientProblem`. For most normal uses, it is okay to use the default implementation.

Ceres Solver ships with a number of commonly used instances of `LocalParameterization`. Another great place to find high quality implementations of $\boxplus$ operations on a variety of manifolds is the Sophus library developed by Hauke Strasdat and his collaborators.

`IdentityParameterization`

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

$$\boxplus(x, \Delta) = x + \Delta$$

This is the same as x living in a Euclidean manifold.

`QuaternionParameterization`

Another example that occurs commonly in Structure from Motion problems is when camera rotations are parameterized using a quaternion. This is a 3-dimensional manifold that lives in 4-dimensional space.

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

The multiplication $*$ between the two 4-vectors on the right hand side is the standard quaternion product.

`EigenQuaternionParameterization`

Eigen uses a different internal memory layout for the elements of the quaternion than what is commonly used. Specifically, Eigen stores the elements in memory as (x, y, z, w), i.e., the *real* part (w) is stored as the last element. Note, when creating an Eigen quaternion through the constructor the elements are accepted in w, x, y, z order.

Since Ceres operates on parameter blocks which are raw `double` pointers this difference is important and requires a different parameterization. `EigenQuaternionParameterization` uses the same `Plus` operation as `QuaternionParameterization` but takes into account Eigen's internal memory element ordering.

`SubsetParameterization`

Suppose x is a two dimensional vector, and the user wishes to hold the first coordinate constant. Then, Δ is a scalar and $\boxplus$ is defined as

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

`SubsetParameterization` generalizes this construction to hold any part of a parameter block constant by specifying the set of coordinates that are held constant.

Note

It is legal to hold all coordinates of a parameter block to constant using a `SubsetParameterization`. It is the same as calling `Problem::SetParameterBlockConstant()` on that parameter block.

`HomogeneousVectorParameterization`

In computer vision, homogeneous vectors are commonly used to represent objects in projective geometry such as points in projective space. One example where it is useful to use this over-parameterization is in representing points whose triangulation is ill-conditioned. Here it is advantageous to use homogeneous vectors, instead of an Euclidean vector, because it can represent points at and near infinity.

`HomogeneousVectorParameterization` defines a `LocalParameterization` for an $n-1$ dimensional manifold that embedded in n dimensional space where the scale of the vector does not matter, i.e., elements of the projective space $\mathbb{P}^{n-1}$. It assumes that the last coordinate of the n-vector is the *scalar* component of the homogenous vector, i.e., *finite* points in this representation are those for which the *scalar* component is non-zero.

Further, `HomogeneousVectorParameterization::Plus` preserves the scale of x.

`LineParameterization`

This class provides a parameterization for lines, where the line is defined using an origin point and a direction vector. So the parameter vector size needs to be two times the ambient space dimension, where the first half is interpreted as the origin point and the second half as the direction. This local parameterization is a special case of the Affine Grassmannian manifold for the case $\operatorname{Graff}_1(R^n)$.

Note that this is a parameterization for a line, rather than a point constrained to lie on a line. It is useful when one wants to optimize over the space of lines. For example, n distinct points in 3D (measurements) we want to find the line that minimizes the sum of squared distances to all the points.

`ProductParameterization`

Consider an optimization problem over the space of rigid transformations $SE(3)$, which is the Cartesian product of $SO(3)$ and $\mathbb{R}^3$. Suppose you are using Quaternions to represent the rotation, Ceres ships with a local parameterization for that and $\mathbb{R}^3$ requires no, or `IdentityParameterization` parameterization. So how do we construct a local parameterization for a parameter block a rigid transformation?

In cases, where a parameter block is the Cartesian product of a number of manifolds and you have the local parameterization of the individual manifolds available, `ProductParameterization` can be used to construct a local parameterization of the cartesian product. For the case of the rigid transformation, where say you have a parameter block of size 7, where the first four entries represent the rotation as a quaternion, a local parameterization can be constructed as

```
ProductParameterization se3_param(new QuaternionParameterization(),
                                  new IdentityParameterization(3));
```

`AutoDiffLocalParameterization`

class **`AutoDiffLocalParameterization`**

`AutoDiffLocalParameterization` does for `LocalParameterization` what `AutoDiffCostFunction` does for `CostFunction`. It allows the user to define a templated functor that implements the `LocalParameterization::Plus()` operation and it uses automatic differentiation to implement the computation of the Jacobian.

To get an auto differentiated local parameterization, you must define a class with a templated operator() (a functor) that computes

$$x' = \boxplus(x, \Delta x),$$

For example, Quaternions have a three dimensional local parameterization. Its plus operation can be implemented as (taken from internal/ceres/autodiff_local_parameterization_test.cc)

```
struct QuaternionPlus {
  template<typename T>
  bool operator()(const T* x, const T* delta, T* x_plus_delta) const {
    const T squared_norm_delta =
        delta[0] * delta[0] + delta[1] * delta[1] + delta[2] * delta[2];

    T q_delta[4];
    if (squared_norm_delta > 0.0) {
      T norm_delta = sqrt(squared_norm_delta);
      const T sin_delta_by_delta = sin(norm_delta) / norm_delta;
      q_delta[0] = cos(norm_delta);
      q_delta[1] = sin_delta_by_delta * delta[0];
      q_delta[2] = sin_delta_by_delta * delta[1];
      q_delta[3] = sin_delta_by_delta * delta[2];
    } else {
      // We do not just use q_delta = [1,0,0,0] here because that is a
      // constant and when used for automatic differentiation will
      // lead to a zero derivative. Instead we take a first order
      // approximation and evaluate it at zero.
      q_delta[0] = T(1.0);
      q_delta[1] = delta[0];
      q_delta[2] = delta[1];
      q_delta[3] = delta[2];
    }

    Quaternionproduct(q_delta, x, x_plus_delta);
    return true;
  }
};
```

Given this struct, the auto differentiated local parameterization can now be constructed as

```
LocalParameterization* local_parameterization =
    new AutoDiffLocalParameterization<QuaternionPlus, 4, 3>;
                                                      |  |
                           Global Size ---------------+  |
                           Local Size -------------------+
```

`Problem`

class **`Problem`**

`Problem` holds the robustified bounds constrained non-linear least squares problem (1). To create a least squares problem, use the `Problem::AddResidalBlock()` 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);
```

`Problem::AddResidualBlock()` as the name implies, adds a residual block to the problem. It adds a `CostFunction`, an optional `LossFunction` and connects the `CostFunction` to a set of parameter block.

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 `nullptr`, 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 `Problem::AddParameterBlock()`. This causes additional correctness checking; however, `Problem::AddResidualBlock()` implicitly adds the parameter blocks if they are not present, so calling `Problem::AddParameterBlock()` explicitly is not required.

`Problem::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 behavior.

You can set any parameter block to be constant using `Problem::SetParameterBlockConstant()` and undo this using `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.

Ownership

`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`. 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 `Problem::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.

class **Problem::Options**

Options struct that is used to control `Problem`.

Ownership Problem::Options::cost_function_ownership

Default: `TAKE_OWNERSHIP`

This option controls whether the Problem object owns the cost functions.

If set to TAKE_OWNERSHIP, then the problem object will delete the cost functions on destruction. The destructor is careful to delete the pointers only once, since sharing cost functions is allowed.

Ownership Problem::Options::loss_function_ownership

Default: `TAKE_OWNERSHIP`

This option controls whether the Problem object owns the loss functions.

If set to TAKE_OWNERSHIP, then the problem object will delete the loss functions on destruction. The destructor is careful to delete the pointers only once, since sharing loss functions is allowed.

Ownership `Problem::Options::local_parameterization_ownership`

Default: `TAKE_OWNERSHIP`

This option controls whether the Problem object owns the local parameterizations.

If set to TAKE_OWNERSHIP, then the problem object will delete the local parameterizations on destruction. The destructor is careful to delete the pointers only once, since sharing local parameterizations is allowed.

bool `Problem::Options::enable_fast_removal`

Default: `false`

If true, trades memory for faster `Problem::RemoveResidualBlock()` and `Problem::RemoveParameterBlock()` operations.

By default, `Problem::RemoveParameterBlock()` and `Problem::RemoveResidualBlock()` take time proportional to the size of the entire problem. If you only ever remove parameters or residuals from the problem occasionally, this might be acceptable. However, if you have memory to spare, enable this option to make `Problem::RemoveParameterBlock()` take time proportional to the number of residual blocks that depend on it, and `Problem::RemoveResidualBlock()` take (on average) constant time.

The increase in memory usage is twofold: an additional hash set per parameter block containing all the residuals that depend on the parameter block; and a hash set in the problem containing all residuals.

bool `Problem::Options::disable_all_safety_checks`

Default: *false*

By default, Ceres performs a variety of safety checks when constructing the problem. There is a small but measurable performance penalty to these checks, typically around 5% of construction time. If you are sure your problem construction is correct, and 5% of the problem construction time is truly an overhead you want to avoid, then you can set disable_all_safety_checks to true.

WARNING Do not set this to true, unless you are absolutely sure of what you are doing.

Context *`Problem::Options::context`

Default: *nullptr*

A Ceres global context to use for solving this problem. This may help to reduce computation time as Ceres can reuse expensive objects to create. The context object can be *nullptr*, in which case Ceres may create one.

Ceres does NOT take ownership of the pointer.

EvaluationCallback *`Problem::Options::evaluation_callback`

Default: *nullptr*

Using this callback interface, Ceres will notify you when it is about to evaluate the residuals or Jacobians.

If an `evaluation_callback` is present, Ceres will update the user's parameter blocks to the values that will be used when calling `CostFunction::Evaluate()` before calling `EvaluationCallback::PrepareForEvaluation()`. One can then use this callback to share (or cache) computation between cost functions by doing the shared computation in `EvaluationCallback::PrepareForEvaluation()` before Ceres calls `CostFunction::Evaluate()`.

Problem does NOT take ownership of the callback.

> **Note**
>
> Evaluation callbacks are incompatible with inner iterations. So calling Solve with `Solver::Options::use_inner_iterations` set to *true* on a `Problem` with a non-null evaluation callback is an error.

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

template <typename Ts...> ResidualBlockId Problem::AddResidualBlock(CostFunction* cost_function, LossFunction* loss

Add a residual block to the overall cost function. 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 *nullptr*, in which case the cost of the term is just the squared norm of the residuals.

The parameter blocks may be passed together as a `vector<double*>`, or `double*` pointers.

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.

The Problem object 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: 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.

Example usage:

```
double x1[] = {1.0, 2.0, 3.0};
double x2[] = {1.0, 2.0, 5.0, 6.0};
double x3[] = {3.0, 6.0, 2.0, 5.0, 1.0};
vector<double*> v1;
v1.push_back(x1);
vector<double*> v2;
v2.push_back(x2);
v2.push_back(x1);

Problem problem;

problem.AddResidualBlock(new MyUnaryCostFunction(...), nullptr, x1);
problem.AddResidualBlock(new MyBinaryCostFunction(...), nullptr, x2, x1);
problem.AddResidualBlock(new MyUnaryCostFunction(...), nullptr, v1);
problem.AddResidualBlock(new MyBinaryCostFunction(...), nullptr, v2);
```

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

Add a parameter block with appropriate size to the problem. Repeated calls with the same arguments are ignored. Repeated calls with the same double pointer but a different size results in undefined behavior.

void Problem::AddParameterBlock(double **values*, int *size*)

Add a parameter block with appropriate size and parameterization to the problem. Repeated calls with the same arguments are ignored. Repeated calls with the same double pointer but a different size results in undefined behavior.

void Problem::RemoveResidualBlock(ResidualBlockId *residual_block*)

Remove a residual block from the problem. Any parameters that the residual block depends on are not removed. The cost and loss functions for the residual block will not get deleted immediately; won't happen until the problem itself is deleted. If Problem::Options::enable_fast_removal is true, then the removal is fast (almost constant time). Otherwise, removing a residual block will incur a scan of the entire Problem object to verify that the residual_block represents a valid residual in the problem.

WARNING: Removing a residual or parameter block will destroy the implicit ordering, rendering the jacobian or residuals returned from the solver uninterpretable. If you depend on the evaluated jacobian, do not use remove! This may change in a future release. Hold the indicated parameter block constant during optimization.

void Problem::RemoveParameterBlock(*const* double **values*)

Remove a parameter block from the problem. The parameterization of the parameter block, if it exists, will persist until the deletion of the problem (similar to cost/loss functions in residual block removal). Any residual blocks that depend on the parameter are also removed, as described above in RemoveResidualBlock(). If Problem::Options::enable_fast_removal is true, then the removal is fast (almost constant time). Otherwise, removing a parameter block will incur a scan of the entire Problem object.

WARNING: Removing a residual or parameter block will destroy the implicit ordering, rendering the jacobian or residuals returned from the solver uninterpretable. If you depend on the evaluated jacobian, do not use remove! This may change in a future release.

void Problem::**SetParameterBlockConstant**(*const* double **values*)

Hold the indicated parameter block constant during optimization.

void Problem::**SetParameterBlockVariable**(double **values*)

Allow the indicated parameter to vary during optimization.

bool Problem::**IsParameterBlockConstant**(*const* double **values*)*const*

Returns `true` if a parameter block is set constant, and false otherwise. A parameter block may be set constant in two ways: either by calling `SetParameterBlockConstant` or by associating a `LocalParameterization` with a zero dimensional tangent space with it.

void Problem::**SetParameterization**(double **values*, LocalParameterization **local_parameterization*)

Set the local parameterization for one of the parameter blocks. The local_parameterization is owned by the Problem by default. It is acceptable to set the same parameterization for multiple parameters; the destructor is careful to delete local parameterizations only once. Calling *SetParameterization* with *nullptr* will clear any previously set parameterization.

LocalParameterization *Problem::**GetParameterization**(*const* double **values*)*const*

Get the local parameterization object associated with this parameter block. If there is no parameterization object associated then *nullptr* is returned

void Problem::**SetParameterLowerBound**(double **values*, int *index*, double *lower_bound*)

Set the lower bound for the parameter at position *index* in the parameter block corresponding to *values*. By default the lower bound is `-std::numeric_limits<double>::max()`, which is treated by the solver as the same as $-\infty$.

void Problem::**SetParameterUpperBound**(double **values*, int *index*, double *upper_bound*)

Set the upper bound for the parameter at position *index* in the parameter block corresponding to *values*. By default the value is `std::numeric_limits<double>::max()`, which is treated by the solver as the same as ∞.

double Problem::**GetParameterLowerBound**(*const* double **values*, int *index*)

Get the lower bound for the parameter with position *index*. If the parameter is not bounded by the user, then its lower bound is `-std::numeric_limits<double>::max()`.

double Problem::**GetParameterUpperBound**(*const* double **values*, int *index*)

Get the upper bound for the parameter with position *index*. If the parameter is not bounded by the user, then its upper bound is `std::numeric_limits<double>::max()`.

int Problem::**NumParameterBlocks**()*const*

Number of parameter blocks in the problem. Always equals parameter_blocks().size() and parameter_block_sizes().size().

int Problem::**NumParameters**()*const*

The size of the parameter vector obtained by summing over the sizes of all the parameter blocks.

int Problem::**NumResidualBlocks**()*const*

Number of residual blocks in the problem. Always equals residual_blocks().size().

int Problem::**NumResiduals**()*const*

The size of the residual vector obtained by summing over the sizes of all of the residual blocks.

int Problem::**ParameterBlockSize**(*const* double **values*)*const*

The size of the parameter block.

int Problem::**ParameterBlockLocalSize**(*const* double **values*)*const*

The size of local parameterization for the parameter block. If there is no local parameterization associated with this parameter block, then `ParameterBlockLocalSize` = `ParameterBlockSize`.

bool Problem::**HasParameterBlock**(*const* double **values*)*const*

Is the given parameter block present in the problem or not?

void Problem::**GetParameterBlocks**(vector<double*> **parameter_blocks*)*const*

Fills the passed `parameter_blocks` vector with pointers to the parameter blocks currently in the problem. After this call, `parameter_block.size() == NumParameterBlocks`.

void Problem::**GetResidualBlocks**(vector<ResidualBlockId> **residual_blocks*)*const*

Fills the passed *residual_blocks* vector with pointers to the residual blocks currently in the problem. After this call, *residual_blocks.size() == NumResidualBlocks*.

void Problem::**GetParameterBlocksForResidualBlock**(*const* ResidualBlockId *residual_block*, vector<double*> **parameter_blocks*)*const*

Get all the parameter blocks that depend on the given residual block.

void Problem::**GetResidualBlocksForParameterBlock**(*const* double **values*, vector<ResidualBlockId> **residual_blocks*)*const*

Get all the residual blocks that depend on the given parameter block.

If *Problem::Options::enable_fast_removal* is *true*, then getting the residual blocks is fast and depends only on the number of residual blocks. Otherwise, getting the residual blocks for a parameter block will incur a scan of the entire `Problem` object.

const CostFunction *Problem::**GetCostFunctionForResidualBlock**(*const* ResidualBlockId *residual_block*)*const*

Get the `CostFunction` for the given residual block.

const LossFunction *Problem::**GetLossFunctionForResidualBlock**(*const* ResidualBlockId *residual_block*)*const*

Get the `LossFunction` for the given residual block.

bool **EvaluateResidualBlock**(ResidualBlockId *residual_block_id*, bool *apply_loss_function*, double **cost*, double **residuals*, double ***jacobians*)*const*

Evaluates the residual block, storing the scalar cost in `cost`, the residual components in `residuals`, and the jacobians between the parameters and residuals in `jacobians[i]`, in row-major order.

If `residuals` is `nullptr`, the residuals are not computed.

If `jacobians` is `nullptr`, no Jacobians are computed. If `jacobians[i]` is `nullptr`, then the Jacobian for that parameter block is not computed.

It is not okay to request the Jacobian w.r.t a parameter block that is constant.

The return value indicates the success or failure. Even if the function returns false, the caller should expect the output memory locations to have been modified.

The returned cost and jacobians have had robustification and local parameterizations applied already; for example, the jacobian for a 4-dimensional quaternion parameter using the `QuaternionParameterization` is `num_residuals x 3` instead of `num_residuals x 4`.

`apply_loss_function` as the name implies allows the user to switch the application of the loss function on and off.

Note

If an `EvaluationCallback` is associated with the problem, then its `EvaluationCallback::PrepareForEvaluation()` method will be called every time this method is called with *new_point = true*. This conservatively assumes that the user may have changed the parameter values since the previous call to evaluate / solve. For improved efficiency, and only if you know that the parameter values have not changed between calls, see `Problem::EvaluateResidualBlockAssumingParametersUnchanged()`.

bool **`EvaluateResidualBlockAssumingParametersUnchanged`**(ResidualBlockId *residual_block_id*, bool *apply_loss_function*, double **cost*, double **residuals*, double ***jacobians*)*const*

Same as `Problem::EvaluateResidualBlock()` except that if an `EvaluationCallback` is associated with the problem, then its `EvaluationCallback::PrepareForEvaluation()` method will be called every time this method is called with new_point = false.

This means, if an `EvaluationCallback` is associated with the problem then it is the user's responsibility to call `EvaluationCallback::PrepareForEvaluation()` before calling this method if necessary, i.e. iff the parameter values have been changed since the last call to evaluate / solve.'

This is because, as the name implies, we assume that the parameter blocks did not change since the last time `EvaluationCallback::PrepareForEvaluation()` was called (via `Solve()`, `Problem::Evaluate()` or `Problem::EvaluateResidualBlock()`).

bool `Problem`::**`Evaluate`**(*const* Problem::EvaluateOptions &*options*, double **cost*, vector<double> **residuals*, vector<double> **gradient*, CRSMatrix **jacobian*)

Evaluate a `Problem`. Any of the output pointers can be *nullptr*. Which residual blocks and parameter blocks are used is controlled by the `Problem::EvaluateOptions` struct below.

Note

The evaluation will use the values stored in the memory locations pointed to by the parameter block pointers used at the time of the construction of the problem, for example in the following code:

```
Problem problem;
double x = 1;
problem.Add(new MyCostFunction, nullptr, &x);

double cost = 0.0;
problem.Evaluate(Problem::EvaluateOptions(), &cost, nullptr, nullptr, nullptr);
```

The cost is evaluated at *x = 1*. If you wish to evaluate the problem at *x = 2*, then

```
x = 2;
problem.Evaluate(Problem::EvaluateOptions(), &cost, nullptr, nullptr, nullptr);
```

is the way to do so.

Note

If no local parameterizations are used, then the size of the gradient vector is the sum of the sizes of all the parameter blocks. If a parameter block has a local parameterization, then it contributes “LocalSize” entries to the gradient vector.

Note

This function cannot be called while the problem is being solved, for example it cannot be called from an `IterationCallback` at the end of an iteration during a solve.

Note

If an EvaluationCallback is associated with the problem, then its PrepareForEvaluation method will be called everytime this method is called with `new_point = true`.

class `Problem`::**`EvaluateOptions`**

Options struct that is used to control `Problem::Evaluate()`.

vector<double*> `Problem::EvaluateOptions::parameter_blocks`

The set of parameter blocks for which evaluation should be performed. This vector determines the order in which parameter blocks occur in the gradient vector and in the columns of the jacobian matrix. If parameter_blocks is empty, then it is assumed to be equal to a vector containing ALL the parameter blocks. Generally speaking the ordering of the parameter blocks in this case depends on the order in which they were added to the problem and whether or not the user removed any parameter blocks.

NOTE This vector should contain the same pointers as the ones used to add parameter blocks to the Problem. These parameter block should NOT point to new memory locations. Bad things will happen if you do.

vector<ResidualBlockId> `Problem::EvaluateOptions::residual_blocks`

The set of residual blocks for which evaluation should be performed. This vector determines the order in which the residuals occur, and how the rows of the jacobian are ordered. If residual_blocks is empty, then it is assumed to be equal to the vector containing all the residual blocks.

bool `Problem::EvaluateOptions::apply_loss_function`

Even though the residual blocks in the problem may contain loss functions, setting apply_loss_function to false will turn off the application of the loss function to the output of the cost function. This is of use for example if the user wishes to analyse the solution quality by studying the distribution of residuals before and after the solve.

int `Problem::EvaluateOptions::num_threads`

Number of threads to use. (Requires OpenMP).

`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;
};
```

void `EvaluationCallback::PrepareForEvaluation`(bool *evaluate_jacobians*, bool *new_evaluation_point*)

Ceres will call `EvaluationCallback::PrepareForEvaluation()` every time, and once before it computes the residuals and/or the Jacobians.

User parameters (the double* values provided by the us) are fixed until the next call to `EvaluationCallback::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 `EvaluationCallback::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 `EvaluationCallback::PrepareForEvaluation()` and `CostFunction::Evaluate()`.

`rotation.h`

Many applications of Ceres Solver involve optimization problems where some of the variables correspond to rotations. To ease the pain of work with the various representations of rotations (angle-axis, quaternion and matrix) we provide a handy set of templated functions. These functions are templated so that the user can use them within Ceres Solver's automatic differentiation framework.

template<typename **T**>
void **AngleAxisToQuaternion**(T*const* **angle_axis*, T **quaternion*)

Convert a value in combined axis-angle representation to a quaternion.

The value `angle_axis` is a triple whose norm is an angle in radians, and whose direction is aligned with the axis of rotation, and `quaternion` is a 4-tuple that will contain the resulting quaternion.

template<typename **T**>
void **QuaternionToAngleAxis**(T*const* **quaternion*, T **angle_axis*)

Convert a quaternion to the equivalent combined axis-angle representation.

The value `quaternion` must be a unit quaternion - it is not normalized first, and `angle_axis` will be filled with a value whose norm is the angle of rotation in radians, and whose direction is the axis of rotation.

template<typename **T**, int **row_stride**, int **col_stride**>
void **RotationMatrixToAngleAxis**(*const* MatrixAdapter<*const* T, row_stride, col_stride> *&R*, T **angle_axis*)

template<typename **T**, int **row_stride**, int **col_stride**>
void **AngleAxisToRotationMatrix**(T*const* **angle_axis*, *const* MatrixAdapter<T, row_stride, col_stride> *&R*)

template<typename **T**>
void **RotationMatrixToAngleAxis**(T*const* **R*, T **angle_axis*)

template<typename **T**>
void **AngleAxisToRotationMatrix**(T*const* **angle_axis*, T **R*)

Conversions between 3x3 rotation matrix with given column and row strides and axis-angle rotation representations. The functions that take a pointer to T instead of a MatrixAdapter assume a column major representation with unit row stride and a column stride of 3.

template<typename **T**, int **row_stride**, int **col_stride**>
void **EulerAnglesToRotationMatrix**(*const* T **euler*, *const* MatrixAdapter<T, row_stride, col_stride> *&R*)

template<typename **T**>
void **EulerAnglesToRotationMatrix**(*const* T **euler*, int *row_stride*, T **R*)

Conversions between 3x3 rotation matrix with given column and row strides and Euler angle (in degrees) rotation representations.

The {pitch,roll,yaw} Euler angles are rotations around the {x,y,z} axes, respectively. They are applied in that same order, so the total rotation R is Rz * Ry * Rx.

The function that takes a pointer to T as the rotation matrix assumes a row major representation with unit column stride and a row stride of 3. The additional parameter row_stride is required to be 3.

template<typename **T**, int **row_stride**, int **col_stride**>

void **QuaternionToScaledRotation**(*const* T*q*[4], *const* MatrixAdapter<T, row_stride, col_stride> &*R*)

template<typename **T**>
void **QuaternionToScaledRotation**(*const* T*q*[4], T*R*[3 * 3])

Convert a 4-vector to a 3x3 scaled rotation matrix.

The choice of rotation is such that the quaternion $\begin{bmatrix}1 & 0 & 0 & 0\end{bmatrix}$ goes to an identity matrix and for small a, b, c the quaternion $\begin{bmatrix}1 & a & b & c\end{bmatrix}$ goes to the matrix

$$I + 2\begin{bmatrix} 0 & -c & b \\ c & 0 & -a \\ -b & a & 0 \end{bmatrix} + O(q^2)$$

which corresponds to a Rodrigues approximation, the last matrix being the cross-product matrix of $\begin{bmatrix}a & b & c\end{bmatrix}$. Together with the property that $R(q1 * q2) = R(q1) * R(q2)$ this uniquely defines the mapping from q to R.

In the function that accepts a pointer to T instead of a MatrixAdapter, the rotation matrix `R` is a row-major matrix with unit column stride and a row stride of 3.

No normalization of the quaternion is performed, i.e. $R = \|q\|^2 Q$, where Q is an orthonormal matrix such that $\det(Q) = 1$ and $Q * Q' = I$.

template<typename **T**>
void **QuaternionToRotation**(*const* T*q*[4], *const* MatrixAdapter<T, row_stride, col_stride> &*R*)

template<typename **T**>
void **QuaternionToRotation**(*const* T*q*[4], T*R*[3 * 3])

Same as above except that the rotation matrix is normalized by the Frobenius norm, so that $RR' = I$ (and $\det(R) = 1$).

template<typename **T**>
void **UnitQuaternionRotatePoint**(*const* T*q*[4], *const* T*pt*[3], T*result*[3])

Rotates a point pt by a quaternion q:

$$\text{result} = R(q)\text{pt}$$

Assumes the quaternion is unit norm. If you pass in a quaternion with $|q|^2 = 2$ then you WILL NOT get back 2 times the result you get for a unit quaternion.

template<typename **T**>
void **QuaternionRotatePoint**(*const* T*q*[4], *const* T*pt*[3], T*result*[3])

With this function you do not need to assume that q has unit norm. It does assume that the norm is non-zero.

template<typename **T**>
void **QuaternionProduct**(*const* T*z*[4], *const* T*w*[4], T*zw*[4])

$$zw = z * w$$

where $*$ is the Quaternion product between 4-vectors.

template<typename **T**>
void **CrossProduct**(*const* T*x*[3], *const* T*y*[3], T*x_cross_y*[3])

$$\text{x_cross_y} = x \times y$$

template<typename **T**>
void **AngleAxisRotatePoint**(*const* T*angle_axis*[3], *const* T*pt*[3], T*result*[3])

$$y = R(\text{angle_axis})x$$

Cubic Interpolation

Optimization problems often involve functions that are given in the form of a table of values, for example an image. Evaluating these functions and their derivatives requires interpolating these values. Interpolating tabulated functions is a vast area of research and there are a lot of libraries which implement a variety of interpolation schemes. However, using them within the automatic differentiation framework in Ceres is quite painful. To this end, Ceres provides the ability to interpolate one dimensional and two dimensional tabular functions.

The one dimensional interpolation is based on the Cubic Hermite Spline, also known as the Catmull-Rom Spline. This produces a first order differentiable interpolating function. The two dimensional interpolation scheme is a generalization of the one dimensional scheme where the interpolating function is assumed to be separable in the two dimensions,

More details of the construction can be found Linear Methods for Image Interpolation by Pascal Getreuer.

class `CubicInterpolator`

Given as input an infinite one dimensional grid, which provides the following interface.

```
struct Grid1D {
  enum { DATA_DIMENSION = 2; };
  void GetValue(int n, double* f) const;
};
```

Where, `GetValue` gives us the value of a function f (possibly vector valued) for any integer n and the enum `DATA_DIMENSION` indicates the dimensionality of the function being interpolated. For example if you are interpolating rotations in axis-angle format over time, then `DATA_DIMENSION = 3`.

`CubicInterpolator` uses Cubic Hermite splines to produce a smooth approximation to it that can be used to evaluate the $f(x)$ and $f'(x)$ at any point on the real number line. For example, the following code interpolates an array of four numbers.

```
const double x[] = {1.0, 2.0, 5.0, 6.0};
Grid1D<double, 1> array(x, 0, 4);
CubicInterpolator interpolator(array);
double f, dfdx;
interpolator.Evaluate(1.5, &f, &dfdx);
```

In the above code we use `Grid1D` a templated helper class that allows easy interfacing between `C++` arrays and `CubicInterpolator`.

`Grid1D` supports vector valued functions where the various coordinates of the function can be interleaved or stacked. It also allows the use of any numeric type as input, as long as it can be safely cast to a double.

class `BiCubicInterpolator`

Given as input an infinite two dimensional grid, which provides the following interface:

```
struct Grid2D {
  enum { DATA_DIMENSION = 2 };
  void GetValue(int row, int col, double* f) const;
};
```

Where, `GetValue` gives us the value of a function f (possibly vector valued) for any pair of integers `row` and `col` and the enum `DATA_DIMENSION` indicates the dimensionality of the function being interpolated. For example if you are interpolating a color image with three channels (Red, Green & Blue), then `DATA_DIMENSION = 3`.

`BiCubicInterpolator` uses the cubic convolution interpolation algorithm of R. Keys [Keys], to produce a smooth approximation to it that can be used to evaluate the $f(r,c)$, $\frac{\partial f(r,c)}{\partial r}$ and $\frac{\partial f(r,c)}{\partial c}$ at any any point in the real plane.

For example the following code interpolates a two dimensional array.

```
const double data[] = {1.0, 3.0, -1.0, 4.0,
                       3.6, 2.1,  4.2, 2.0,
                       2.0, 1.0,  3.1, 5.2};
Grid2D<double, 1>  array(data, 0, 3, 0, 4);
BiCubicInterpolator interpolator(array);
double f, dfdr, dfdc;
interpolator.Evaluate(1.2, 2.5, &f, &dfdr, &dfdc);
```

In the above code, the templated helper class `Grid2D` is used to make a `C++` array look like a two dimensional table to `BiCubicInterpolator`.

`Grid2D` supports row or column major layouts. It also supports vector valued functions where the individual coordinates of the function may be interleaved or stacked. It also allows the use of any numeric type as input, as long as it can be safely cast to double.