Automatic Differentiation in Ceres Solver

Erik Lindén

KTH / Tobii

March 29, 2018

What is Ceres?

- What is Ceres?

What is Ceres?

- What is Ceres?
- Ceres is an non-linear least squares optimizer.

What is Ceres?

- What is Ceres?
- Ceres is an non-linear least squares optimizer.
- Mostly for bundle adjustment.

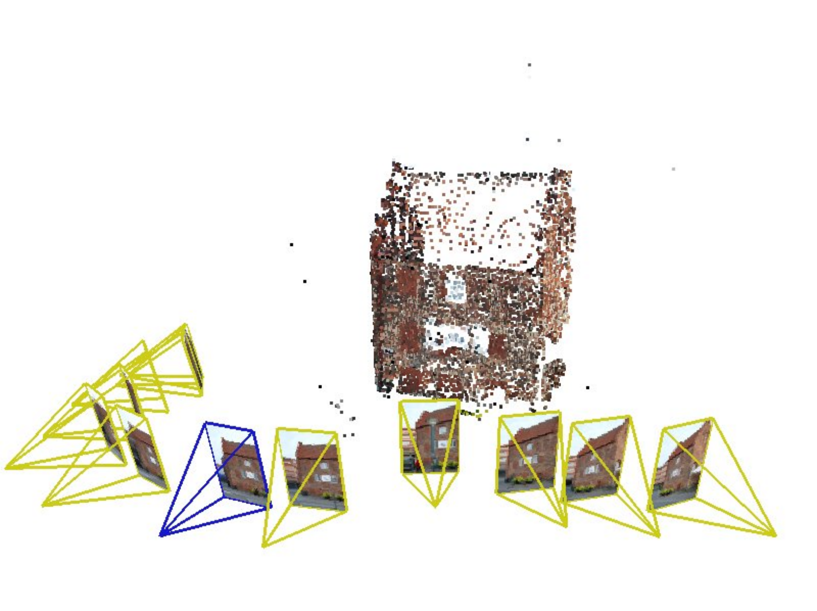
What is Ceres?

- What is Ceres?
- Ceres is an non-linear least squares optimizer.
- Mostly for bundle adjustment.
- What is bundle adjustment?

What is Ceres?

- What is Ceres?
- Ceres is an non-linear least squares optimizer.
- Mostly for bundle adjustment.
- What is bundle adjustment?
- Given N points observed by C cameras, find the positions of the points and the poses of the cameras that minimized the mean squared back-projection error.

Bundle adjustment example



Optimization problem

- Minimize

$$L(p,\theta) = \sum_{c=1}^{C} \sum_{i \in P_c} r(p_i, \theta_c, o_{c,i})^2$$

where we have camera c and point i. P_c is the points seen by the camera, p_i is the position of the point, θ_c is the pose of the camera, $o_{c,i}$ is where the camera saw the point and $r(\cdot)$ is the back-projection error.

Optimization problem

- Minimize
$$L(p,\theta) = \sum_{c=1}^{C} \sum_{i \in P_c} r(p_i, \theta_c, o_{c,i})^2$$
where we have camera c and point i. P_c is the points seen by the camera, p_i is the position of the point, θ_c is the pose of the camera, $o_{c,i}$ is where the camera saw the point and $r(\cdot)$ is the back-projection error.
- Minimized with respect to all point positions p and camera poses θ.

Optimization problem

- Minimize
$$L(p,\theta) = \sum_{c=1}^{C} \sum_{i \in P_c} r(p_i, \theta_c, o_{c,i})^2$$
where we have camera c and point i. P_c is the points seen by the camera, p_i is the position of the point, θ_c is the pose of the camera, $o_{c,i}$ is where the camera saw the point and $r(\cdot)$ is the back-projection error.
- Minimized with respect to all point positions p and camera poses θ.
- How do we do that?

Solvers

- Some may know Matlab's `lsqnonlin` (or scipy equivalent).

Solvers

- Some may know Matlab's `lsqnonlin` (or scipy equivalent).
- `lsqnonlin` uses the Jacobian of the residuals w.r.t. the parameters. For least-squares problems, this gives us the Hessian of the cost function for free

$$L = \frac{1}{2} r^T r, \quad \frac{\delta L}{\delta x} = J^T r, \quad \frac{\delta^2 L}{\delta x^2} = J^T J$$

Solvers

- Some may know Matlab's `lsqnonlin` (or scipy equivalent).
- `lsqnonlin` uses the Jacobian of the residuals w.r.t. the parameters. For least-squares problems, this gives us the Hessian of the cost function for free

$$L = \frac{1}{2} r^T r, \quad \frac{\delta L}{\delta x} = J^T r, \quad \frac{\delta^2 L}{\delta x^2} = J^T J$$

- A second-order approximation of L is

$$L(x + \triangle x) \approx L + J^T r \triangle x + \frac{1}{2!} \triangle x^T J^T J \triangle x$$

Solvers

- Some may know Matlab's `lsqnonlin` (or scipy equivalent).
- `lsqnonlin` uses the Jacobian of the residuals w.r.t. the parameters. For least-squares problems, this gives us the Hessian of the cost function for free

$$L = \frac{1}{2} r^T r, \quad \frac{\delta L}{\delta x} = J^T r, \quad \frac{\delta^2 L}{\delta x^2} = J^T J$$

- A second-order approximation of L is

$$L(x + \triangle x) \approx L + J^T r \triangle x + \frac{1}{2!} \triangle x^T J^T J \triangle x$$

- Taking the derivative w.r.t $\triangle x$ and setting it to zero gives

$$J^T J \triangle x = -J^T r$$

Jacobians

- How do we find the Jacobian J? `lsqnonlin` gives you two options: numeric or symbolic differentiation.

Jacobians

- How do we find the Jacobian J? `lsqnonlin` gives you two options: numeric or symbolic differentiation.
- Numeric differentiation is slow and commits both cardinal sins of numerical analysis: *“thou shalt not add small numbers to big numbers”*, and *“thou shalt not subtract numbers which are approximately equal”*.

Jacobians

- How do we find the Jacobian J? `lsqnonlin` gives you two options: numeric or symbolic differentiation.
- Numeric differentiation is slow and commits both cardinal sins of numerical analysis: *"thou shalt not add small numbers to big numbers"*, and *"thou shalt not subtract numbers which are approximately equal"*.
- Symbolic differentiation is tedious and error-prone.

Jacobians

- How do we find the Jacobian J? `lsqnonlin` gives you two options: numeric or symbolic differentiation.
- Numeric differentiation is slow and commits both cardinal sins of numerical analysis: *“thou shalt not add small numbers to big numbers”*, and *“thou shalt not subtract numbers which are approximately equal”*.
- Symbolic differentiation is tedious and error-prone.
- But there is a third way...

The chain rule

- Consider

$$y = f(g(h(x))) = f(g(w_1)) = f(w_2)$$

The chain rule

- Consider

$$y = f(g(h(x))) = f(g(w_1)) = f(w_2)$$

- The chain rule says

$$\frac{dy}{dx} = \frac{dy}{dw_2}\frac{dw_2}{dw_1}\frac{dw_1}{dx}$$

The chain rule

- Consider

$$y = f(g(h(x))) = f(g(w_1)) = f(w_2)$$

- The chain rule says

$$\frac{dy}{dx} = \frac{dy}{dw_2}\frac{dw_2}{dw_1}\frac{dw_1}{dx}$$

- In forward-mode automatic differentiation, we evaluate the chain rule from the inside out.

Dual numbers

- Forward-mode automatic differentiation can be efficiently implemented using dual numbers. We replace the real number x with $x + v\epsilon$, where ϵ is *infinitesimal* such that $\epsilon^2 = 0$.

Dual numbers

- Forward-mode automatic differentiation can be efficiently implemented using dual numbers. We replace the real number x with $x + v\epsilon$, where ϵ is *infinitesimal* such that $\epsilon^2 = 0$.
- Then we can evaluate simple expressions like this

$$\begin{aligned} f(x) &= x^2 \\ &= (x + v\epsilon)^2 \\ &= x^2 + 2xv\epsilon + \epsilon^2 \\ &= x^2 + 2xv\epsilon \end{aligned}$$

Dual numbers

- Forward-mode automatic differentiation can be efficiently implemented using dual numbers. We replace the real number x with $x + v\epsilon$, where ϵ is *infinitesimal* such that $\epsilon^2 = 0$.
- Then we can evaluate simple expressions like this

$$\begin{aligned} f(x) &= x^2 \\ &= (x + v\epsilon)^2 \\ &= x^2 + 2xv\epsilon + \epsilon^2 \\ &= x^2 + 2xv\epsilon \end{aligned}$$

- In this case, the gradient is $2xv$.

Dual numbers continued...

- We (usually) evaluate the expressions immediately, to get a numerical value. For example, let $x = 10$. Then we replace x with $10 + 1\epsilon$. We have $v = 1$ since the gradient of x with respect to x is 1. And then evaluate

$$\begin{aligned} f(10 + 1\epsilon) &= 10^2 + 2 \cdot 10 \cdot 1\epsilon \\ &= 100 + 20\epsilon \end{aligned}$$

Dual numbers continued...

- We (usually) evaluate the expressions immediately, to get a numerical value. For example, let $x = 10$. Then we replace x with $10 + 1\epsilon$. We have $v = 1$ since the gradient of x with respect to x is 1. And then evaluate

$$\begin{aligned} f(10 + 1\epsilon) &= 10^2 + 2 \cdot 10 \cdot 1\epsilon \\ &= 100 + 20\epsilon \end{aligned}$$

- So the value of $f(10)$ is 100 and the gradient is 20.

Dual numbers continued...

- If we have more then one number, we introduce more infinitesimals

$$x = x + v_1\epsilon + v_2\delta$$
$$y = y + u_1\epsilon + u_2\delta$$

Dual numbers continued...

- If we have more then one number, we introduce more infinitesimals

$$x = x + v_1\epsilon + v_2\delta$$
$$y = y + u_1\epsilon + u_2\delta$$

- By initializing, seeding, $v_1 = 1$, $v_2 = 0$ for x and $u_1 = 0$, $u_2 = 1$ for y, we can compute gradients for both variables at the same time.

Dual numbers continued...

- If we have more then one number, we introduce more infinitesimals

$$x = x + v_1\epsilon + v_2\delta$$
$$y = y + u_1\epsilon + u_2\delta$$

- By initializing, seeding, $v_1 = 1$, $v_2 = 0$ for x and $u_1 = 0$, $u_2 = 1$ for y, we can compute gradients for both variables at the same time.
- In general, each dual number needs to hold a vector of N gradients, where N is the number of parameters.

Dual numbers continued...

- If we have more then one number, we introduce more infinitesimals

$$x = x + v_1\epsilon + v_2\delta$$
$$y = y + u_1\epsilon + u_2\delta$$

- By initializing, seeding, $v_1 = 1$, $v_2 = 0$ for x and $u_1 = 0$, $u_2 = 1$ for y, we can compute gradients for both variables at the same time.
- In general, each dual number needs to hold a vector of N gradients, where N is the number of parameters.
- Though bundle adjustment is a special case...

Sparsity in bundle adjustment

- In bundle adjustment, the Jacobian gets a very special sparsity pattern, since any residual only depends on one point and one camera.

Sparsity in bundle adjustment

- In bundle adjustment, the Jacobian gets a very special sparsity pattern, since any residual only depends on one point and one camera.
- The dual numbers only need to track $3 + 3 + 3 = 9$ gradients when computing a residual.

Sparsity in bundle adjustment

- In bundle adjustment, the Jacobian gets a very special sparsity pattern, since any residual only depends on one point and one camera.
- The dual numbers only need to track $3 + 3 + 3 = 9$ gradients when computing a residual.
- Ceres does a lot of algorithmic magic to leverage this special sparsity structure.

Implementation?

- How does Ceres implement dual numbers?

Implementation?

- How does Ceres implement dual numbers?
- ```
  template<int N>
  class Jet{
    ...
    double a;
    Eigen::Vector<double, N> v;
  };
  ```
  ```

Implementation?

- How does Ceres implement dual numbers?
- ```
  template<int N>
  class Jet{
    ...
    double a;
    Eigen::Vector<double, N> v;
  };
  ```
- Almost all math operations are overloaded for `Jet`.
  ```

Implementation?

- How does Ceres implement dual numbers?
- ```
  template<int N>
  class Jet{
     ...
     double a;
     Eigen::Vector<double, N> v;
  };
  ```
- Almost all math operations are overloaded for `Jet`.
- Your cost function must be templated on the value type.
  ```

Forward vs. reverse mode AD

- Forward mode automatic differentiation walks the chain rule inside out.

Forward vs. reverse mode AD

- Forward mode automatic differentiation walks the chain rule inside out.
- Most efficient for functions $f : \mathbb{R}^n \to \mathbb{R}^m$ where $m \gg n$.

Forward vs. reverse mode AD

- Forward mode automatic differentiation walks the chain rule inside out.
- Most efficient for functions $f : \mathbb{R}^n \to \mathbb{R}^m$ where $m \gg n$.
- Reverse mode automatic differentiation walks the chain rule from the outside in.

Forward vs. reverse mode AD

- Forward mode automatic differentiation walks the chain rule inside out.
- Most efficient for functions $f : \mathbb{R}^n \to \mathbb{R}^m$ where $m \gg n$.
- Reverse mode automatic differentiation walks the chain rule from the outside in.
- Most efficient for functions $f : \mathbb{R}^n \to \mathbb{R}^m$ where $m \ll n$.

Forward vs. reverse mode AD

- Forward mode automatic differentiation walks the chain rule inside out.
- Most efficient for functions $f : \mathbb{R}^n \to \mathbb{R}^m$ where $m \gg n$.
- Reverse mode automatic differentiation walks the chain rule from the outside in.
- Most efficient for functions $f : \mathbb{R}^n \to \mathbb{R}^m$ where $m \ll n$.
- When training neural networks, reverse mode automatic differentiation is called “backprop”.

Summary

- There is a method to quickly and easily compute accurate gradients.

Summary

- There is a method to quickly and easily compute accurate gradients.
- It is implemented in most languages.

Further reading

- The Wikipedia page.

Further reading

- The Wikipedia page.
- Ceres' documentation.

Further reading

- The Wikipedia page.
- Ceres' documentation.
- *Automatic Differentiation in Machine Learning: a Survey*.

Questions?