Nonlinear optimization

camera calibration and triangulation

Morten R. Hannemose, mohan@dtu.dk

March 4, 2022

02504 Computer vision course lectures,
DTU Compute, Kgs. Lyngby 2800, Denmark

Learning objectives

After this lecture you should be able to:

- perform non-linear optimization within computer vision
- explain the principle behind Levenberg-Marquardt
- compute the Jacobian of a function
- reason about different parameterizations of rotations in optimization

Presentation topics

- Non-linear least-squares optimization
- Gradients
    - Analytical gradients
    - Finite differences approximation
    - Rotations in optimization
- Camera calibration

Non-linear least-squares optimization

Least-squares problems

Problems of the form

$$\min_{\boldsymbol{x}} \|g(\boldsymbol{x}) - \boldsymbol{y}\|_2^2 \,.$$

Make all parameters into a vector $\boldsymbol{x}$ and optimize everything!

Least-squares problems

Problems of the form

$$\min_{\boldsymbol{x}} \|g(\boldsymbol{x}) - \boldsymbol{y}\|_2^2 \, .$$

Make all parameters into a vector $\boldsymbol{x}$ and optimize everything!

- Homography estimation
- Pose estimation
- Bundle adjustment
- Camera calibration with lens distortion
- Triangulation
- ...

Reformulate a bit

$$
\begin{aligned}
e(\boldsymbol{x}) &= ||\underbrace{g(\boldsymbol{x}) - \boldsymbol{y}}_{f(\boldsymbol{x})}||_2^2 \\
&= \|f(\boldsymbol{x})\|_2^2 \\
&= f(\boldsymbol{x})^{\mathsf{T}} f(\boldsymbol{x})
\end{aligned}
$$

Levenberg-Marquardt

We solve the problem in an iterative fashion. At the k^{th} iteration:

Replace with f with first order approximation around $\boldsymbol{x}_k$

$$f(\boldsymbol{x}_k + \delta) = f(\boldsymbol{x}_k) + \boldsymbol{J}\delta$$

$\boldsymbol{J}$ is the Jacobian that contains all first order derivatives of f at $\boldsymbol{x}_k$.

Note that the Jacobian of f and g are identical.

Levenberg-Marquardt

The sum of squared errors is then:

$$\begin{aligned} e(\boldsymbol{x}_k + \delta)) &= \|f(\boldsymbol{x}_k + \delta)\|_2^2 \\ &= (f(\boldsymbol{x}_k) + \boldsymbol{J}\delta)^{\mathsf{T}}(f(\boldsymbol{x}_k) + \boldsymbol{J}\delta) \end{aligned}$$

Levenberg-Marquardt

The sum of squared errors is then:

$$\begin{aligned} e(\boldsymbol{x}_k + \delta)) &= \|f(\boldsymbol{x}_k + \delta)\|_2^2 \\ &= (f(\boldsymbol{x}_k) + \boldsymbol{J}\delta)^\mathsf{T}(f(\boldsymbol{x}_k) + \boldsymbol{J}\delta) \\ &= f(\boldsymbol{x}_k)^\mathsf{T} f(\boldsymbol{x}_k) + 2(\boldsymbol{J}\delta)^\mathsf{T} f(\boldsymbol{x}_k) + (\boldsymbol{J}\delta)^\mathsf{T}\boldsymbol{J}\delta^\mathsf{T} \\ &= f(\boldsymbol{x}_k)^\mathsf{T} f(\boldsymbol{x}_k) + 2\delta^\mathsf{T}\boldsymbol{J}^\mathsf{T} f(\boldsymbol{x}_k) + \delta^\mathsf{T}\boldsymbol{J}^\mathsf{T}\boldsymbol{J}\delta^\mathsf{T} \end{aligned}$$

which is a second order approximation of e using only first order derivatives of f 🙂

Levenberg-Marquardt

Take the derivative of $e(\boldsymbol{x}_k + \delta))$

$$\frac{\partial e(\boldsymbol{x}_k + \delta))}{\partial \delta} = 2\boldsymbol{J}^\top f(\boldsymbol{x}_k) + 2\boldsymbol{J}^\top \boldsymbol{J}\delta.$$

Levenberg-Marquardt

Take the derivative of $e(\boldsymbol{x}_k + \delta))$

$$\frac{\partial e(\boldsymbol{x}_k + \delta))}{\partial \delta} = 2\boldsymbol{J}^\top f(\boldsymbol{x}_k) + 2\boldsymbol{J}^\top \boldsymbol{J}\delta.$$

Find the optimum by setting the derivative equal to zero

$$\begin{aligned} \mathbf{0} &= 2\boldsymbol{J}^\top f(\boldsymbol{x}_k) + 2\boldsymbol{J}^\top \boldsymbol{J}\delta \Leftrightarrow \\ \boldsymbol{J}^\top \boldsymbol{J}\delta &= -\boldsymbol{J}^\top f(\boldsymbol{x}_k). \end{aligned}$$

Solve for δ, and then set $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \delta$.

Rinse and repeat!

The λ parameter

The second order approximation of e is better the closer we are to the minimum.

When far away, it can be quite bad, in which case we want to take a small step in the gradient direction.

$$(\boldsymbol{J}^\top \boldsymbol{J} + \lambda \boldsymbol{I})\delta = -\boldsymbol{J}^\top f(\boldsymbol{x}_k).$$

Decrease λ when $f(\boldsymbol{x}_{k+1}) < f(\boldsymbol{x}_k)$ and increase when not.

Levenberg-Marquardt summary

Levenberg-Marquardt is not the only algorithm for nonlinear optimization, but it is often well suited to the types of problems we are likely to encounter in computer vision.

The nature of the least-squares problem is exploited to get a second order method, using only first order derivatives.

Gradients

Gradients

How do we compute $\boldsymbol{J}$?

Gradients

How do we compute $\boldsymbol{J}$?

It's just a lot of derivatives!

- Analytical gradients
- Automatic differentiation
  - Reverse mode automatic differentiation (backpropagation)
  - Forward mode automatic differentiation (dual numbers)
- Finite differences approximation

Gradients

How do we compute $\boldsymbol{J}$?

It's just a lot of derivatives!

- Analytical gradients
- Automatic differentiation
  - Reverse mode automatic differentiation (backpropagation)
  - Forward mode automatic differentiation (dual numbers)
- Finite differences approximation

Analytical gradients - how to find them?

Analytical gradients - how to find them?

Get out pen and paper (or Maple) and find the exact derivative.

Analytical gradients - triangulation example

The reprojection error of a point in 3D in multiple cameras

$$f(\boldsymbol{Q}) = \begin{bmatrix} \Pi\left(\boldsymbol{\mathcal{P}}_1 \Pi^{-1}(\boldsymbol{Q})\right) - \tilde{\boldsymbol{q}}_1 \\ \vdots \\ \Pi\left(\boldsymbol{\mathcal{P}}_n \Pi^{-1}(\boldsymbol{Q})\right) - \tilde{\boldsymbol{q}}_n \end{bmatrix}$$

- f returns a vector of length $2n$
- $\boldsymbol{Q}$ has three parameters
- thus $\boldsymbol{J}$ is $2n \times 3$.

Analytical gradients - triangulation example

The projected point in homogeneous coordinates:

$$\boldsymbol{q} = \boldsymbol{\mathcal{P}} \underbrace{\Pi^{-1}(\boldsymbol{Q})}_{\boldsymbol{Q}_h}$$

$$= \begin{bmatrix} sx \\ sy \\ s \end{bmatrix} = \begin{bmatrix} \boldsymbol{p}^{(1)} \\ \boldsymbol{p}^{(2)} \\ \boldsymbol{p}^{(3)} \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

Analytical gradients - triangulation example

Let's focus on x

$$x = \frac{\boldsymbol{p}^{(1)}\boldsymbol{Q}_h}{\boldsymbol{p}^{(3)}\boldsymbol{Q}_h} = \frac{\mathscr{P}_{11}X + \mathscr{P}_{12}Y + \mathscr{P}_{13}Z + \mathscr{P}_{14}}{\mathscr{P}_{31}X + \mathscr{P}_{32}Y + \mathscr{P}_{33}Z + \mathscr{P}_{34}}$$

A single element of $\boldsymbol{J}$ is then given by:

$$\frac{d}{dX}x = \frac{\mathscr{P}_{11}}{\boldsymbol{p}^{(3)}\boldsymbol{Q}_h} - \frac{\mathscr{P}_{31}(\boldsymbol{p}^{(1)}\boldsymbol{Q}_h)}{(\boldsymbol{p}^{(3)}\boldsymbol{Q}_h)^2}$$

Analytical gradients - summary

Analytical gradients are extremely useful

+ very fast
+ accurate
− complicated to derive and implement

Many functions in OpenCV return the Jacobian in addition the values themselves.

Finite differences approximation - forward differences

A first order Taylor expansion of f

$$f(x+h) = f(x) + \frac{d}{dx}f(x)h + O(h)$$

Finite differences approximation - forward differences

A first order Taylor expansion of f

$$f(x+h) = f(x) + \frac{d}{dx}f(x)h + O(h) \Leftrightarrow$$

can be rewritten to

$$\frac{d}{dx}f(x) = \frac{f(x+h) - f(x)}{h} + O(h).$$

This is called forward differences. (2-point)

Finite differences approximation - central differences

Using a second order Taylor expansion of f evaluated at $x - h$ and $x + h$ can be rearranged to yield

$$\frac{d}{dx}f(x) = \frac{f(x+h) - f(x-h)}{2h} + O(h^2).$$

This is called central differences, which is more accurate ($O(h^2)$), but requires two new evaluations of f.

Finite differences approximation - in practice

Principle can only be applied to one parameter at a time.

To compute $\boldsymbol{J}$ you need to evaluate f once (or twice) for each element in $\boldsymbol{J}$ (which is a lot)

Number of evaluations can be reduced if you know the sparsity structure of $\boldsymbol{J}$, i.e. which elements of $\boldsymbol{x}$ that affect which elements of $f(\boldsymbol{x})$

Finite differences approximation - summary

They only give an approximation of the gradients

+ convenient
− slow
− has parameter h
− numeric problems

Only use when speed and robustness are not your primary concerns.

Rotations in optimization

- Rotations are usually 3×3 matrices, but have just three degrees of freedom.
- How can we parametrize rotations?

Rotations in optimization

- Rotations are usually 3×3 matrices, but have just three degrees of freedom.
- How can we parametrize rotations?
    - optimizing all 9 numbers gives something that is no longer a rotation matrix

Rotations in optimization - Euler angles

Euler angles $(\theta_x, \theta_y, \theta_z)$ uses only three numbers

- Suffers from gimbal lock, i.e. that one parameter does nothing in certain configurations
    - unsuitable for optimization

Rotations in optimization - Euler angles

Euler angles $(\theta_x, \theta_y, \theta_z)$ uses only three numbers

- Suffers from gimbal lock, i.e. that one parameter does nothing in certain configurations
    - unsuitable for optimization
- If the rotation is close to identity, we are far from gimbal lock
    - no problems in this case
    - optimize over a rotation relative to the initial guess
    - $\theta_x, \theta_y, \theta_z$ will usually stay close to zero
        - only if initial guess is good
        - for finite difference start from $\theta_x = \theta_y = \theta_z = 2\pi$

Rotations in optimization - Axis-angle

Axis-angle represents a rotation as a rotation of θ around an axis $\boldsymbol{v}$, where $\|\boldsymbol{v}\|_2 = 1$.

This is then stored as a vector $\boldsymbol{v}/\theta$ (only three elements).

This is how OpenCV returns most rotations (`rvec`), and it can be converted to a full rotation matrix with `cv2.Rodrigues`.

Still has weird behaviour when interpolating between rotations, but works in most cases.

Rotations in optimization - Quaternions

Quaternions are the gold standard for representing rotations.

A quaternion uses four numbers represent a rotation (a, i, j, k) subject to $a^2 + i^2 + j^2 + k^2 = 1$.

Smooth and suffers from no problems, however we have to normalize the quaternion each optimization step, to ensure it is still a valid quaternion.

Software packages

There are many implementations out there!

- Ceres (C++)
  - Dual numbers (no implementing derivatives, 😛)
  - Quaternion parameterization
- `scipy.optimize.least_squares`
  - Easy to use
  - Finite differences or analytical gradients

Camera calibration

Outline

More tips and tricks

- Checkerboard alternatives
- Subpixel estimation
- Overfitting
- Bundle adjustment

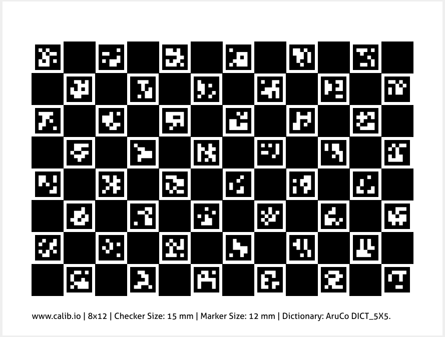
Checkerboard

OpenCV needs to see the all corners of the checkerboard in order to detect it.

Getting the entire image plane covered in detected points is hard, especially near the edges.

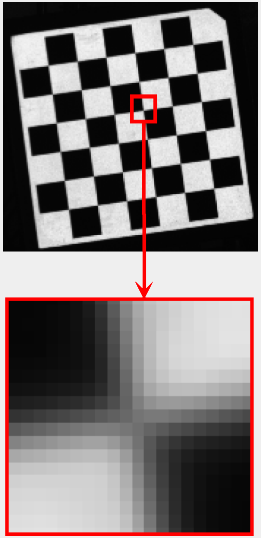
ChArUco

A ChArUco boards is a checkerboard with ArUco markers.


www.calib.io | 8x12 | Checker Size: 15 mm | Marker Size: 12 mm | Dictionary: ArUco DICT_5X5.

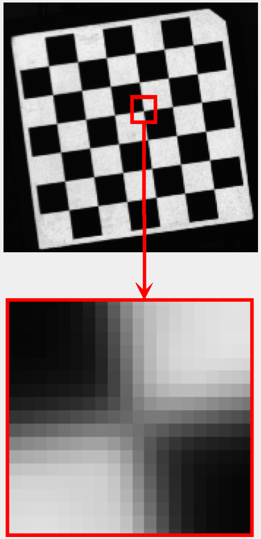
Possible to detect partial checkerboards!

Subpixel corner estimation



- Where is the corner?

Subpixel corner estimation



- Where is the corner?
- Look at a neighbourhood around a corner
- OpenCV has `cv2.cornerSubPix` but it is 💩
- What to do?

Good method for subpixel corner estimation

A good subpixel corner estimator is presented in Schops et al.[1]

Generate n random 2D vectors relative to the corner $\boldsymbol{s}_i$

$$C_{sym}(\boldsymbol{H}) = \sum_{i=1}^{n} \left(\left(I(\boldsymbol{H}(\boldsymbol{s}_i)) - I(\boldsymbol{H}(-\boldsymbol{s}_i)) \right)^2 \right)$$

$\boldsymbol{H}$ maps a point using the homography, and I interpolates the value of the image at the given point. Minimize C_{sym}.

[1]Schops, Thomas, et al. "Why having 10,000 parameters in your camera model is better than twelve." Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. 2020.

Camera calibration - A cautionary tale

- Including more distortion parameters in your model will always give a lower re-projection error

Camera calibration - A cautionary tale

- Including more distortion parameters in your model will always give a lower re-projection error
- . . . for the images used to calibration!
    - Will not necessarily generalize.
- This can be overfitting
- Use cross-validation

Camera calibration - How to do cross validation

- For each checkerboard, split your detected corners into a validation and training set
- Do the calibration using all training corners
- Use the estimated checkerboard pose to reproject the validation corners

Bundle adjustment

- The checkerboard can have imperfections, such as not being flat
    - The checkerboards you get today are not very flat
- Optimize everything again, including the 3D positions of the corners

Learning objectives

After this lecture you should be able to:

- perform non-linear optimization within computer vision
- explain the principle behind Levenberg-Marquardt
- compute the Jacobian of a function
- reason about different parameterizations of rotations in optimization

Info about exercise and exercise time!