Optimization

Point triangulation

This is also discussed in ROVI:
One of the classical task of 3D inference from cameras, is making 3D measurements from two or more known cameras. Specifically we have a set of cameras, where we know the internal and external calibration of the all the cameras, and we want to find the position of an object that is identified in all cameras. This boils down to finding the coordinates of a 3D point Q from its known projections, $q_i = [1, \dots, n]$, in n known cameras, P_i.
The point is projected to the camera image as

$$p_i = P_i Q$$

Each camera viewing the point Q then puts a constraint on the position of the point in 3D space. However, due to small errors, the back-projection of the image points might not intersect. Therefore, optimization strategies need to be used. But first we look at a simple linear algorithm.

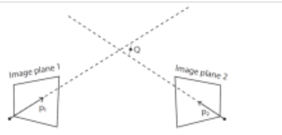


Figure 2.11: The result of point triangulation is the 3D point, Q, closest to the back projections of the observed 2D points.

Linear algorithm

To ease the notation, the rows of the P_i will here be denotes by a superscript, i.e.

$$P_i = \begin{bmatrix} P_i^1 \\ P_i^2 \\ P_i^3 \end{bmatrix}$$

and thus the pinhole camera model can be expanded as follows

$$q_i = \begin{bmatrix} s_i x_i \\ s_i y_i \\ s_i \end{bmatrix} = \begin{bmatrix} P_i^1 \\ P_i^2 \\ P_i^3 \end{bmatrix} \cdot Q \Rightarrow$$

$$s_i x_i = P_i^1 Q, \qquad s_i y_i = P_i^2 Q, \qquad s_i = P_i^3 Q \Rightarrow$$

$$x_i = \frac{s_i x_i}{s_i} = \frac{P_i^1 Q}{P_i^3 Q}, \qquad y_i = \frac{s_i y_i}{s_i} = \frac{P_i^2 Q}{P_i^3 Q}, \qquad (2.21)$$

With the help of some arithmetic, we get

$x_i = \frac{P_i^1 Q}{P_i^3 Q}$	$\Downarrow$	$y_i = \frac{P_i^2 Q}{P_i^3 Q}$	(2.22)
$P_i^3 Q x_i = P_i^1 Q$	$\Downarrow$	$P_i^3 Q y_i = P_i^2 Q$	
$P_i^3 Q x_i - P_i^1 Q = 0$	$\Downarrow$	$P_i^3 Q y_i - P_i^2 Q = 0$	
$(P_i^3 x_i - P_i^1) Q = 0$		$(P_i^3 y_i - P_i^2) Q = 0$	

(2.22) is seen to be linear constrained in Q. Since Q has three degrees of freedom we need at least three such constraint to determine Q. This corresponds to projections in at least two known cameras, since each camera poses two linear constraints in general. It is seen that the x and y parts of (2.22) corresponds to planes, ($l^T q = 0$). e.g. knowing the x-coordinate of Q's image in a given camera, defines a plane with coefficients $P_i^3 x_i - p_i^1$, which Q lies on. The inersection of the planes the x and y coordinates pose is the 3D line corresponding to the back projection of the 2D point q_i.

The linear constraints from each camera view can then be stacked into a matrix

$$B = \begin{bmatrix} P_1^3 x_1 - P_1^1 \\ P_1^3 y_1 - P_1^2 \\ \vdots \\ P_i^3 x_i - P_i^1 \\ P_i^3 y_i - P_i^2 \end{bmatrix}$$

Which can be rewritten to

$$BQ = 0$$

We add an extra constraint, $\|Q\| \neq 0$ to remove the trival solution to the estimation problem. Specifically, Q, is a homogeneous representation of a 3D point, and this **defined up to scale**, this scale should not be zero. The scale can be fixed by requring

$$\|Q\| = 1$$

However, the cameras are not perfect, so the measurement will be noisy. We therefore solve

$$\min_Q \|BQ\|_2^2, \qquad where \;\; \|Q\| = 1$$

Which is the least square problem.

Statistically issues with the linear algorithm

The pinhole model described in (2.21) and the localization of the 2D point q_i are assumed to be perfect and without noise. This is not true for a real system, therefore, a more accurate version of (2.21) is:

$$x_i = \frac{P_i^1 Q}{P_i^3 Q} + \varepsilon_i^x, \qquad y_i = \frac{P_i^2 Q}{P_i^3 Q} + \varepsilon_i^y, \qquad (2.24)$$

where $(\varepsilon_i^x, \varepsilon_i^y)$ is the noise of the 2D location (x_i, y_i). (2.22) is rewritte to:

$x_i = \frac{P_i^1 Q}{P_i^3 Q} + \varepsilon_i^x$	$\Downarrow$	$y_i = \frac{P_i^2 Q}{P_i^3 Q} + \varepsilon_i^y$	(2.25)
$P_i^3 Q x_i = P_i^1 Q + P_i^3 Q \varepsilon_i^x$	$\Downarrow$	$P_i^3 Q y_i = P_i^2 Q + P_i^3 Q \varepsilon_i^y$	
$(P_i^3 x_i - P_i^1) Q = P_i^3 Q \varepsilon_i^x$		$(P_i^3 y_i - P_i^2) Q = P_i^3 Q \varepsilon_i^y$	

Where $P_i^3 Q$ is equal to the distance from the camera to the 3D point. The minimization problem is then

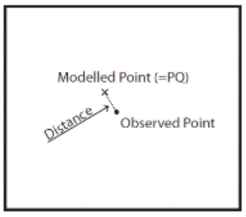
$$\min_Q \|BQ\|_2^2 = \min_Q \sum_{i=1}^{n} \left(P_i^3 Q \varepsilon_i^x\right)^2 + \left(P_i^3 Q \varepsilon_i^y\right)^2$$

$$= \min_Q \sum_{i=1}^{n} \left(P_i^3 Q \varepsilon_i^x\right)^2 \left\| \begin{bmatrix} \varepsilon_i^x \\ \varepsilon_i^y \end{bmatrix} \right\|_2^2$$

That is observations made by cameras further from the 3D point are weighted more. Given that the error model in (2.24) is the correct one, **this is not a meaningful quantity to minimize**. It is however the result a linear algorithm which is computationally feasible and in general gives decent results. Algorithms with this property are said to minimize an algebraic error measure.
The Linear method is therefore used to get an initial guess for a non-linear optimization scheme.

The error model given in (2.24) states that the error is on the image point location. This make sense, because the two main error sources are: identifying where the entity is actually seen, and unmodeled optical phenomena, such as radial distortion. Both entities are well captured by a deviation in the point location.

Error between real point and modelled point



Camera measurement setup

When doing 3D point measurements from cameras, i.e. point triangulation, the accuracy depends on the configuration or special relation of the cameras and the 3D point to be measured. Figure 2.14 show the optical rays for the real position of Q, r_1 and r_2, and a permutation of r_2 in the direction of $p_2 + \varepsilon$, which produce an error on the 3D position of Q, $Q + E$. The two ideal rays, r_1, r_2, have an angle of θ between them.

If we zoom in on figure 2.14 we get figure 1.16. Here we see that

$$E = \frac{e}{\cos\left(\frac{\pi}{2} - \theta\right)}$$

which mean that lowest error happens when $\theta = \frac{\pi}{2} = 90°$. The value of θ is related to the baseline, B, and the depth, D, of the camera set up. Thus a general rule of thumb is tthat the relationship between baseline B and depth D shoulb be

$$\frac{1}{3} < \frac{B}{D} < 3$$

There will also be an error in image one, which will have a similar effect as that of image two, also dependent on θ.
Secondly, when using more than two cameras, this rule of thumb indicates that for any 3D point there should be at least

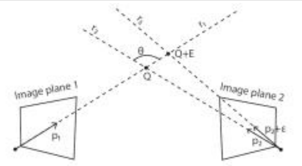
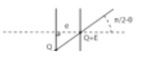
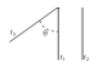
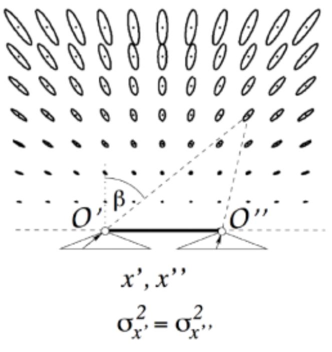


Figure 2.14: A schematic view of point triangulation of the 3D point **Q**, from the projections $\mathbf{p}_1$ and $\mathbf{p}_2$ in cameras one and two respectively. To investigate the effect of an error in an image coordinate, we assume that $\mathbf{p}_2$ is offset by ε, resulting in an error on **Q** of **E**. The optical rays resulting from three image coordinates are denoted by $\mathbf{r}_1$,$\mathbf{r}_2$ and $\mathbf{r}_3$. The angel between $\mathbf{r}_1$ and $\mathbf{r}_2$ is denoted by θ.



		a pair of cameras with a reasonable ratio between baseline and depth.	 Figure 2.16: An amplification of Figure 2.14, around Q.Here it is seen that the closer θ is to $\frac{\pi}{2}$ the smaller $\mathbb{E}$ is relative to e.
	Quality of the 3D point	Assume that we measure the image coordinates in x and y with $\sigma_{x'} = \sigma_y'$ precision. If we have a x-parallax setup, we have $X = Mx', \qquad Y = M\frac{y' + y''}{2}$ Where ' and '' refer to the first and the second camera. The uncertainty in x and y is then given by $\sigma_X = M\sigma_x' = \frac{Z}{c}\sigma_{x''}, \qquad \sigma_Y = \frac{\sqrt{2}}{2}M\sigma_y' = \frac{\sqrt{2}}{2}\frac{Z}{c}\sigma_{y'}$ We can the derive the following equations: $\sigma_Z = \frac{X}{p_x}\cdot\sigma_{p_x} = \frac{cB}{p_x^2}\cdot\sigma_{p_x} = \frac{Z^2}{cB}\cdot\sigma_{p_x} = \frac{X}{c\,B/Z}\cdot\sigma_{p_x}$ where B is he baseline, Z is the depth. We then see that the standard deviation of Z depends: - On the *x-parallax standard deviation* - Inverse quadratically on the x-parallax - Quadratically on the depth - Inversely on the base/Depth ration	 β O' O'' x', x'' $\sigma^2_{x'} = \sigma^2_{x''}$
Camera Resection (Pose estimation)		Just as a 3D point was estimated from the corresponding 2D projections in known cameras, a camera can be estimated from known 3D points and the corresponding 2D projections. Camera resection is highly related to camera calibration, because both cases are concerned with estimation the camera parameters. Like triangulation, a linear algorithm does not minimize a statistical meaningful error, by minimizes an algebraic error. Therefore, is the linear algorithm used as an initial guess and then a non-linear optimization is follow in order to heighten the quality of the solution. Again, like the triangulation point, we have two types: We can use the Epipolar geometry, or use the Homography. We have: - Epipolar Geometry ○ 8-point algorithm ○ 7-point algorithm ○ 5-point algorithm ○ Sampson - non-linear method - Homography	
	Normalization of points	When implementing algorithms on a computer numerical considerations are almost always of importance. When minimizing an algebraic error, especially in the fundamental matrix case, experience has shown that most algorithms will fail without such considerations. In the fundamental matrix case this is recommended done by changing the image coordinates, such that the mean and variance of the points in an image are zero and one respectively. This can be done via a scaling, s, and a translation, $[\Delta x, \Delta y]^T$, which in homogeneous coordinates can be implemented as $T = \begin{bmatrix} s & 0 & \Delta x \\ 0 & s & \Delta y \\ 0 & 0 & 1 \end{bmatrix}$ where T_1 and T_2 are the transformations needed in each of the two images. The mean and variance of the $\tilde{q}_{1i}$ are then zero and one respectively. The same should hold for the $\tilde{q}_{2i}$. The fundamental matrix is then estimated using the $\tilde{q}_{1i}$ and $\tilde{q}_{2i}$ giving a $\tilde{F}$, such that $0 = \tilde{q}_{2i}^T\tilde{F}\tilde{q}_{1i} = q_{2i}^T T_2^T \tilde{F} T_1 q_{1i}$ implying that the fundamental matrix sought is $G = T_2^T \tilde{F} T_1$	
	8-point	Assume that n point correspondences $q_{1i}, q_{2i}, \quad i \in [1, \dots, n]$, are given, where $q_{1i} = \begin{bmatrix} x_{1i} \\ y_{1i} \\ 1 \end{bmatrix}, \qquad q_{2i} = \begin{bmatrix} x_{2i} \\ y_{2i} \\ 1 \end{bmatrix}$ For each i we insert this into the equation for the fundamental matrix $0 = q_{2i}^T F q_{qi}$ $= [x_{1i}x_{2i} \quad x_{1i}y_{2i} \quad x_{1i} \quad y_{1i}x_{2i} \quad y_{1i}y_{2i} \quad y_{1i} \quad x_{2i} \quad y_{2i} \quad 1]\cdot vec(F)$ $= b_i T \cdot vec(F)$ with $vec(F) = [F_{11} \quad F_{21} \quad F_{31} \quad F_{12} \quad F_{22} \quad F_{32} \quad F_{13} \quad F_{23} \quad F_{33}], \qquad (Kronecker\ notation)$ We state the optimization problem as $B = \begin{bmatrix} b_1^T \\ \vdots \\ b_n^T \end{bmatrix}$ And find the solution via $\min_{vec(F)} \lVert B \cdot vec(F) \rVert_2^2, \qquad (2.35)$ Since the algorithm does not exploit the fact that the fundamental matrix has a determinant of zero, eight constraints are needed to determine F. Each point correspondence yields *one linear constraint*, so the algorithm needs 8 point correspondence, hence its name.	
	7-point	The 7-point algorithm utilize that the determinant of F is zero, thus only 7 point correspondences are needed. If only seven point are used in B in (2.35) , the result will be a linear combination $F = \alpha F' + F^\dagger,$ spanning this 2D subspace. The correct solution is found by inserting this into the constraint $\det(F) = \det(\alpha F' + F^\dagger) = 0$ This is a third order polynomial in α, which will have one or three real solutions.	
	5-point	The 5 point algorithm is used when the epipolar geometry is to be estimated from cameras with known internal parameters, because the essential matrix suffices, given the relation $F = A_2^T E A_1^{-1}$. Methods for estimating the essential matrix can be seen in the SLAM section or in Aanæs section 3.2.	
	Homography - linear algorithm	The linear algorithm for homography resemble the linear algorithm in 8-point. Assume that n 2D point correspondence, i.e. q_{1i} and $q_{2i}, \quad i \in \{1, \dots\}$ are given, where $q_{1i} = \begin{bmatrix} x_{1i} \\ y_{1i} \\ 1 \end{bmatrix}, \qquad q_{2i} = \begin{bmatrix} x_{2i} \\ y_{2i} \\ 1 \end{bmatrix}$ The linear constraint such a point pair poses on H is derived from $q_1 = Hq_2$. We have $q_{1i} = Hq_{2i} \Rightarrow$ $\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} = [q_{1i}]_x H q_{2i}$	

$$= \begin{bmatrix} 0 & -x_{2i} & x_{2i}y_{1i} & 0 & -y_{2i} & y_{2i}y_{1i} & 0 & -1 & y_{1i} \\ x_{2i} & 0 & -x_{2i}x_{1i} & y_{2i} & 0 & -y_{2i}x_{1i} & 1 & 0 & -x_{1i} \\ -x_{2i}y_{1i} & x_{2i}x_{1i} & 0 & -y_{2i}y_{1i} & y_{2i}x_{1i} & 0 & -y_{1i} & x_{1i} & 0 \end{bmatrix} \cdot vec(H)$$

$$b_i^T \cdot vec(H), \qquad (Kronecker)$$

Where

$$vec(H) = [H_{11} \quad H_{21} \quad H_{31} \quad H_{12} \quad H_{22} \quad H_{32} \quad H_{13} \quad H_{23} \quad H_{33}]^T$$

We define

$$B = \begin{bmatrix} b_1^T \\ \vdots \\ b_n^T \end{bmatrix}$$

And a solution is found via

$$\min_{vec(H)} \|B \cdot vec(H)\|_2^2 , \qquad where \ \|vec(H)\| = 1$$

The minimum number of point correspondences needed to estimate a homography is **four**. A homography has 9 parameters, subtracting one for over all scale, we need 8 constraints - i.e. H and $s \cdot H$ represent the same transformation, for any non-zero scalar s.

Problem Definition

In ROVI we learned about DLT (Direct Linear Transformation) where we find the optimal algebraic solution to an optimization problem $Ax = 0$. However, this solution is a linear solution, and does not optimize the geometrix solution.

Most optimization problem can be formulated as a nonlinear least squared problem

$$x^* = \underset{x}{\operatorname{argmin}} \frac{1}{2} \sum_{i=1}^{m} \left(f_i(x)\right)^2$$

$$x^* = \underset{x}{\operatorname{argmin}} \frac{1}{2} f(x)^T f(x)$$

, where $f_i: R^n \in R, i = 1, \dots, m$ are given functions, and m>= n. On the figure to the right, the fuction could be the residual of the fitting

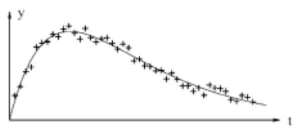


Figure 1.1. *Data points $\{(t_i, y_i)\}$ (marked by +) and model $M(\mathbf{x}, t)$ (marked by full line.)*

Line search

In line search we want to find the appropriate step. See optimization notes.

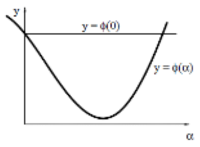


Figure 2.1. *Variation of the cost function along the search line.*

Assumptions

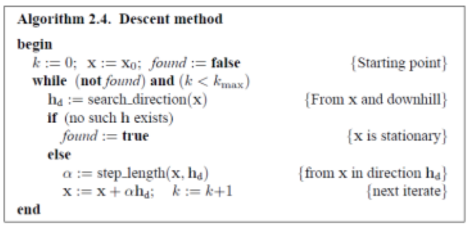
The cost function F is differentiable and so smooth that the following Taylor expansion is valid

$$F(x+h) = F(x) + h^T g + \frac{1}{2} h^T H h + O(\|h\|^3)$$

Where g is the gradient and H is the hessian

$$g \equiv F'(x) = \begin{bmatrix} \frac{\partial F}{\partial x_1}(x) \\ \vdots \\ \frac{\partial F}{\partial x_n(x)} \end{bmatrix}, \qquad H \equiv F''(x) \left[\frac{\partial^2 F}{\partial x_i \partial x_j}(x) \right]$$

For a local minima $F'(x^*) = 0$ and $F''(x^*)$ is positive definite. See optimization notes. The general decent algorithms is seen on he figure


Algorithm 2.4. Descent method

```
begin
  k := 0;  x := x_0;  found := false                     {Starting point}
  while (not found) and (k < k_max)
    h_d := search_direction(x)                           {From x and downhill}
    if (no such h exists)
      found := true                                      {x is stationary}
    else
      α := step_length(x, h_d)                           {from x in direction h_d}
      x := x + α h_d;   k := k+1                         {next iterate}
end
```

Gradient decent / Steepest Decent

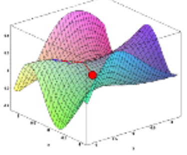
Objective function:

$$F(x, y) = \sin\left(\frac{1}{2}x^2 - \frac{1}{4}y^2 + 3\right)\cos(2x + 1 - c^y)$$

As goes as

$$\lim_{\lambda \to 0} \frac{F(x) - F(x + \lambda h)}{\lambda \|h\|} = \frac{-h^T F'(x)}{\|h\|} = -\|F'(x)\| \cos\theta$$

We can see that the steepest decent direction is for $\theta = \pi$.



To find a local minimum of a function using gradient decent, one takes steps proportional to the negative of the gradient of the function at the current point

ALG

```
    Initialize k=0, choose $x_0$
    $while\ k < k_{max}$
        $x_k = x_{k-1} - \lambda \nabla F(x_{k-1})$
```

It can be shown that this method can fail to find a local minimizer on a second degree polynomial. However, for many problems the method has a good performance in the *initial stage* of the iteration process.

Newton Method

Me make a Taylor expansion on the nonlinear system

$$F'(x+h) = F'(x) + F''(x)h + O(\|h\|^2)$$

$$\approx F'(x) + F''(x)h, \qquad for\ \|h\|\ sufficiently\ smalle$$

We derived the Newton method:
It is desired to find the step that find the stable point of the function F, that is $F' = 0$. We therefore want to solve

$$F'(x+h) = 0 = F'(x) + F''(x)h \Rightarrow$$

$$\underbrace{F''(x)}_{A}\underbrace{h}_{x} = \underbrace{-F'(x)}_{b}$$

We find the Newton step h_n, as the solution to

$$Hh_n = -F'(x), \qquad with\ H = F''(x), \qquad (*)$$

and compute the next iterate

$$x := x + h_n$$

Suppose that H is positive definite, then it is nonsingular which imply that (*) has an unique solution.

The Newton method is good in the finale stage, when x is close to x^* and the the Hessian H is **positive definite**. However, if the Hessian H happens to be **negative definite**, then the Newton method will find a global maximum. We can avoid this by imposing that the step should be in the decent direction.

A simple algorithm would be

```
If $F''(x)$ is positive definite
    $h := h_n, \qquad Netwon\_step$
Else
    $h := h_{sd}, \qquad steepest\_decent$
$x := x + \alpha h$
```

The above algorithm provide a simple mechanism to switch between the steepest decent and the Newton methods.

The newton method has Quadratic convergence - See numerical method notebook

Gauss-Newton Methods

The Guess-Newton Method is used to solve *minimization problems.* It is a modification of Newtons method for finding minimum of a function. Unlike Newton's method, the **Gauss–Newton algorithm can only be used to minimize a sum of squared function values**, but it has the advantage that *second derivatives*, Hessian, which can be challenging to compute, are not required. Instead is the **Hessian approximated.**

If $f(x^*) = 0$ then the method has quadratic convergence, if line search is used and the Jacobian has full rank, just like the Newton methods. Otherwise, we must expect linear convergence.

*Given m functions **r** = (r_1, …, r_m) (often called residuals) of n variables **β** = (β_1, …, β_n), with m ≥ n, the Gauss–Newton algorithm iteratively finds the value of the variables that minimizes the sum of squares*

The Gauss-Newton method is based on a linear approximation to the components of f (a linear model) in the neighborhood of x: for small $\|h\|$ we see from the Taylor expansion that

$$f(x+h) \approx \ell(h) \equiv f(x) + J(x)h$$

We want to minimize the cost function $f(x)$, so we put that into the minimization problem

$$x^* = \underset{x}{\operatorname{argmin}}\, F(x)$$

$$F(x) = \frac{1}{2}\ell(x)^T \ell(x)$$

We get

$$F(x+h) \approx L(h) \equiv \frac{1}{2}\ell(h)^T\ell(h) = F(x) + h^T J^T f + \frac{1}{2}h^T J^T J h$$

The Gauss-Newton step minimizes $L(h)$,

$$h_{gn} = \operatorname{argmin}_h\{L(h)\}.$$

The gradient and Hessian of L are

$$L'(h) = J^T f + J^T J h, \qquad L''(h) = J^T J$$

We notice that L'' is independent of h.

Like the Newton method, we solve $L'(h) = 0$ and find the step as

$$L'(h) = 0 = J^T f + J^T J h \Rightarrow$$
$$J^T J \cdot h_{gn} = -J^T f, \qquad (3.9)$$

and we can then use the line search to find a proper step

$$x \coloneqq x + \alpha \cdot h_{gn}$$

However, because we are approximating the Hessian, the solution is not unique, so the algorithm might fail to converge to the real solution. We then introduce a regularization term that put some constraint on the system. This is called the Levenberg-Marquardt method

Levenberg-Marquardt
The Levenberg-Marquardt is a *damped Gauss-Newton Method*, that is the term μI, and the step h_{lm} is defined by modifying (3.9):

$$\left(J^T J + \mu I\right) \cdot h_{lm} = -g, \qquad with\ g = J^T f\ and\ \mu > 0$$

And we update

$$x \coloneqq x + h_{lm}$$

The damping parameter μ has several effects

a) For all $\mu > 0$ the coefficient matrix is positive definite, and this ensures that h_{lm} is a descent direction.

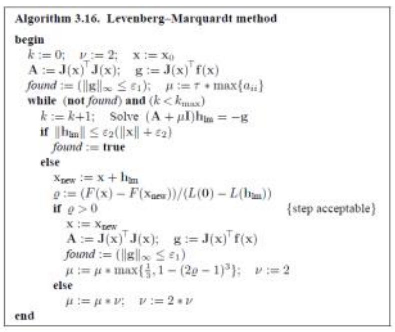
b) For large values of μ we get

$$h_{lm} \approx -\frac{1}{\mu} g = -\frac{1}{\mu} F'(x)$$

Ie. A short step in the **steepest decent** direction. This is good if the current iterate is far from the solution

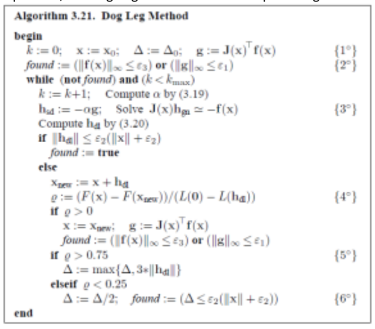
c) If μ is very small, the $h_{lm} \approx h_{gn}$, which is good step in the final stages of the iteration, when x is close to x^*. if $F(x^*) = 0$ (or very small), then we can get (almost) quadratic final convergence.

Thus, the damping parameter influences both the direction and the size of the step, and this leads us to make a method **without** a specific line search.



```
Algorithm 3.16. Levenberg–Marquardt method
begin
  $k := 0$;  $\nu := 2$;  $\mathbf{x} := \mathbf{x}_0$
  $\mathbf{A} := \mathbf{J}(\mathbf{x})^\top\mathbf{J}(\mathbf{x})$;  $\mathbf{g} := \mathbf{J}(\mathbf{x})^\top\mathbf{f}(\mathbf{x})$
  $found := (\|\mathbf{g}\|_\infty \le \varepsilon_1)$;  $\mu := \tau * \max\{a_{ii}\}$
  while (not found) and ($k < k_{\max}$)
    $k := k+1$;  Solve $(\mathbf{A} + \mu\mathbf{I})\mathbf{h}_{lm} = -\mathbf{g}$
    if $\|\mathbf{h}_{lm}\| \le \varepsilon_2(\|\mathbf{x}\| + \varepsilon_2)$
      $found :=$ true
    else
      $\mathbf{x}_{new} := \mathbf{x} + \mathbf{h}_{lm}$
      $\varrho := (F(\mathbf{x}) - F(\mathbf{x}_{new}))/(L(\mathbf{0}) - L(\mathbf{h}_{lm}))$
      if $\varrho > 0$                                   {step acceptable}
        $\mathbf{x} := \mathbf{x}_{new}$
        $\mathbf{A} := \mathbf{J}(\mathbf{x})^\top\mathbf{J}(\mathbf{x})$;  $\mathbf{g} := \mathbf{J}(\mathbf{x})^\top\mathbf{f}(\mathbf{x})$
        $found := (\|\mathbf{g}\|_\infty \le \varepsilon_1)$
        $\mu := \mu * \max\{\frac{1}{3}, 1 - (2\varrho - 1)^3\}$;  $\nu := 2$
      else
        $\mu := \mu * \nu$;  $\nu := 2 * \nu$
end
```

However, when the least square problem arise from a system of non-linear equations, the Dog-Leg method is the best option to go for:



```
Algorithm 3.21. Dog Leg Method
begin
  $k := 0$;  $\mathbf{x} := \mathbf{x}_0$;  $\Delta := \Delta_0$;  $\mathbf{g} := \mathbf{J}(\mathbf{x})^\top\mathbf{f}(\mathbf{x})$        {1°}
  $found := (\|\mathbf{f}(\mathbf{x})\|_\infty \le \varepsilon_3)$ or $(\|\mathbf{g}\|_\infty \le \varepsilon_1)$       {2°}
  while (not found) and ($k < k_{\max}$)
    $k := k+1$;  Compute $\alpha$ by (3.19)
    $\mathbf{h}_{sd} := -\alpha\mathbf{g}$;  Solve $\mathbf{J}(\mathbf{x})\mathbf{h}_{gn} \simeq -\mathbf{f}(\mathbf{x})$        {3°}
    Compute $\mathbf{h}_{dl}$ by (3.20)
    if $\|\mathbf{h}_{dl}\| \le \varepsilon_2(\|\mathbf{x}\| + \varepsilon_2)$
      $found :=$ true
    else
      $\mathbf{x}_{new} := \mathbf{x} + \mathbf{h}_{dl}$
      $\varrho := (F(\mathbf{x}) - F(\mathbf{x}_{new}))/(L(\mathbf{0}) - L(\mathbf{h}_{dl}))$        {4°}
      if $\varrho > 0$
        $\mathbf{x} := \mathbf{x}_{new}$;  $\mathbf{g} := \mathbf{J}(\mathbf{x})^\top\mathbf{f}(\mathbf{x})$
        $found := (\|\mathbf{f}(\mathbf{x})\|_\infty \le \varepsilon_3)$ or $(\|\mathbf{g}\|_\infty \le \varepsilon_1)$
      if $\varrho > 0.75$        {5°}
        $\Delta := \max\{\Delta, 3*\|\mathbf{h}_{dl}\|\}$
      elseif $\varrho < 0.25$
        $\Delta := \Delta/2$;  $found := (\Delta \le \varepsilon_2(\|\mathbf{x}\| + \varepsilon_2))$        {6°}
end
```

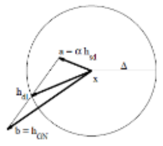
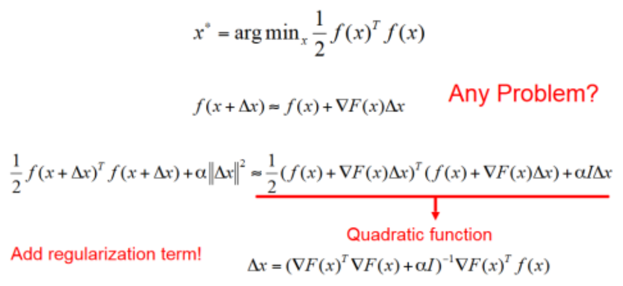


Figure 3.4. *Trust region and Dog Leg step.*[4)]

We now optimize

$$\frac{1}{2}f(x)^T f(x) \to \frac{1}{2}f(x+\Delta x)^T f(x+\Delta x) + \alpha\|\Delta x\|^2 \approx \underbrace{\frac{1}{2}\left(f(x) + \nabla F(x)\Delta x\right)^T\left(f(x) + \nabla F(x)\Delta x\right) + \alpha I \Delta x}_{Quadratic\ function}$$

$$\Delta x = \left(\nabla F(x)^T \nabla F(x) + \alpha I\right)^{-1} \nabla F(x)^T f(x)$$



$$x^* = \arg\min_x \frac{1}{2} f(x)^T f(x)$$

$$f(x+\Delta x) \approx f(x) + \nabla F(x)\Delta x$$

Any Problem?

$$\frac{1}{2} f(x+\Delta x)^T f(x+\Delta x) + \alpha\|\Delta x\|^2 \approx \frac{1}{2}(f(x) + \nabla F(x)\Delta x)^T (f(x) + \nabla F(x)\Delta x) + \alpha I \Delta x$$

Quadratic function

Add regularization term!

$$\Delta x = (\nabla F(x)^T \nabla F(x) + \alpha I)^{-1} \nabla F(x)^T f(x)$$

Bundle adjustment

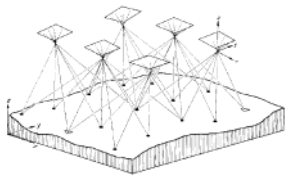
Thursday, 13 February 2020 08.00

https://www.youtube.com/watch?v=uHbRKvD8TWg

Bundle Adjustment Theory

Bundle adjustment is a process for computing from multiple camera images of a 3D scene its 3D shape and the positions and intrinsic parameters of all the cameras simultaneously so that the perspective projection relationship holds.

We want to have multiple cameras to cover whole surfaces.
It can also improve precision and increase the number of details



In order to make the bundle adjustment, we fix a world coordinate, and let t_κ be the viewpoint of the κth camea, $\kappa = 1, \dots, M$. We assume that its camera coordinate system is rotated by R_κ relative to the world coordinate system. Let f_κ be its focal length, and $(u_{0_\kappa}, v_{0_\kappa})$ its principal point.
Let $(x_{\alpha\kappa}, y_{\alpha\kappa})$ be the projection of the αth point $(X_\alpha, Y_\alpha, Z_\alpha)$, $\alpha = 1, \dots, N$, in the κth image.

The perspective projection relationships is written as

$$\begin{pmatrix} x_{\alpha\kappa}/f_0 \\ y_{\alpha\kappa}/f_0 \\ 1 \end{pmatrix} \simeq P_\kappa \begin{pmatrix} X_\alpha \\ Y_\alpha \\ Z_\alpha \\ 1 \end{pmatrix}, \qquad (11.1)$$

where we can write the camera matrix P_κ of the κth camera in the form

$$P_\kappa = \begin{pmatrix} f_\kappa & 0 & u_{0_\kappa} \\ 0 & f_\kappa & v_{0_\kappa} \\ 0 & 0 & f_0 \end{pmatrix} \left(R_\kappa^T \; - R_\kappa^T t_\kappa \right), \qquad (11.2)$$

We rewrite (11.2) as

$$P_\kappa = K_\kappa R_\kappa^T (I - t_\kappa), \qquad K_\kappa \equiv \begin{pmatrix} f_\kappa & 0 & u_{0_\kappa} \\ 0 & f_\kappa & v_{0_\kappa} \\ 0 & 0 & f_0 \end{pmatrix}$$

Where K_κ are the intrinsic parameters of the κth camera.

From (11.1) we can express $x_{\alpha\kappa}$ and $y_{\alpha\kappa}$ in the form

$$x_{\alpha\kappa} = f_0 \frac{P_{\kappa(11)}X_\alpha + P_{\kappa(12)}Y_\alpha + P_{\kappa(13)}Z_\alpha + P_{\kappa(14)}}{P_{\kappa(31)}X_\alpha + P_{\kappa(32)}Y_\alpha + P_{\kappa(33)}Z_\alpha + P_{\kappa(34)}}$$

$$y_{\alpha\kappa} = f_0 \frac{P_{\kappa(21)}X_\alpha + P_{\kappa(22)}Y_\alpha + P_{\kappa(23)}Z_\alpha + P_{\kappa(24)}}{P_{\kappa(31)}X_\alpha + P_{\kappa(32)}Y_\alpha + P_{\kappa(33)}Z_\alpha + P_{\kappa(34)}}, \qquad (11.4)$$

Here, $P_{\kappa(ij)}$ denote the (i, j) element of P_κ. Bundle adjustment is to estimate the 3D positions $(X_\alpha, Y_\alpha, Z_\alpha)$ and the camera matrices from observed $(x_{\alpha\kappa}, y_{\alpha\kappa})$, $\alpha = 1, \dots, N; \kappa = 1, \dots, M$, in such a way that (11.4) is satisfied as much as possible in the sense that - That is minimizing the residual between the given and the modelled image points:

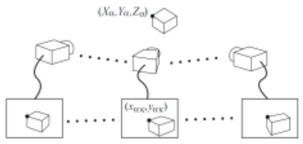
$$E = \sum_{\alpha=1}^{N}\sum_{\kappa=1}^{M} I_{\alpha\kappa} \cdot \left(\underbrace{\frac{x_{\alpha\kappa}}{f_0}}_{\substack{Given\ x \\ image \\ coordinate}} - \underbrace{\frac{P_{\kappa(11)}X_\alpha + P_{\kappa(12)}Y_\alpha + P_{\kappa(13)}Z_\alpha + P_{\kappa(14)}}{P_{\kappa(31)}X_\alpha + P_{\kappa(32)}Y_\alpha + P_{\kappa(33)}Z_\alpha + P_{\kappa(34)}}}_{\substack{Estimated\ /\ modelled \\ image \\ coordinate}} \right)^2 + \sum_{\alpha=1}^{N}\sum_{\kappa=1}^{M} I_{\alpha\kappa} \cdot \left(\underbrace{\frac{y_{\alpha\kappa}}{f_0}}_{\substack{Given\ y \\ image \\ coordinate}} - \underbrace{\frac{P_{\kappa(21)}X_\alpha + P_{\kappa(22)}Y_\alpha + P_{\kappa(23)}Z_\alpha + P_{\kappa(24)}}{P_{\kappa(31)}X_\alpha + P_{\kappa(32)}Y_\alpha + P_{\kappa(33)}Z_\alpha + P_{\kappa(34)}}}_{\substack{Estimated\ /\ modelled \\ image \\ coordinate}} \right)^2, \qquad (11.5)$$

Where $I_{\alpha\kappa}$ is the *visibility index* defined by:

$$I_{\alpha\kappa} = \begin{cases} 1, & if\ point\ \alpha\ is\ visible\ in\ image\ \kappa \\ 0, & otherwise \end{cases}$$

Because (11.5) measure the sum of the distances between the prospectively projected points and actually observed points (in unit f_0), it is called *the reprojection error.*

$\kappa = 1, \dots, M$ is the numbers of cameras and $\alpha = 1, \dots, N$ is the number of points.



$(X_\alpha, Y_\alpha, Z_\alpha)$
$(x_{\alpha\kappa}, y_{\alpha\kappa})$

Fig. 11.1 N points in the scene are viewed by M cameras. The αth point $(X_\alpha, Y_\alpha, Z_\alpha)$ is projected to a point $(x_{\alpha\kappa}, y_{\alpha\kappa})$ in the image of the κth camera

Issues

The reprojection error E in equation (11.5) is iteratively minimized by giving initial values to all unknowns and updating them so that E decrerases at each step. The unknowns are the 3D positions $X_\alpha = (X_\alpha, Y_\alpha, Z_\alpha)^T$, $\alpha = 1, \dots, N$, of all the points and the focal length f_κ, the pricipal points $(u_{0_\kappa}, v_{0_\kappa})$, translation t_κ and rotation R_κ, $\kappa = 1, \dots, M$, of all cameras. The rotation is described with Lie algebra, and the paramets for R is $\Delta\omega_\kappa$: rotation around coordinate axis.

The system consist of $3N + 9M$ terms, which makes the system very big. Futheremore, we cannot determine all of the parameters, because the camera positions are *defined relative* to the 3D shape such that their **absolute position are indeterminate**. Also, we cannot determine the **absolut scale**, because images do not have depth information, thus we cannot tell if the observed M images are taken by moving the camera by a large amount relative to a distance scene, or by moving the camera slightly relative to a nearby scene.

In order to fix the ambiguity, we adopt the following normalization

$$R_1 = I, \qquad t_1 = 0, \qquad t_{12} = 1, \qquad (11.6)$$

This means that we adopt the world coordinate system fixed to the first camera. As for scale, we take the relative displacement of the second camera in the Y-direction as the unit of length. The normalization in (11.6) reduce the number of unkowns to $3N + 9M - 7$.

For convenience, we arrange all the $3M + 9N - 7$ variations of each parameter, ΔX_α, Δf_κ, $(\Delta u_{0_\kappa}, \Delta v_{0_\kappa})$, Δt_κ, and $\Delta\omega_\kappa$, as $\Delta\xi_1, \Delta\xi_2, \dots, \Delta\xi_{3N+9M-7}$. We can then use the Levenberg-Marquardt Method to solve for the change.

Relative position
Undetermined absolute scale.

- ΔX_α: 3D position
- Δf_κ: Camera focal length
- $(\Delta u_{0_\kappa}, \Delta v_{0_\kappa})$: Principal points
- Δt_κ: translation offset
- $\Delta\omega_\kappa$: rotation around coordinate axis (Lie Algebra)
- $\alpha = 1, \dots, N;\ \kappa = 1, \dots, M$

Bundle Adjustment Algorithm

1. Assume initial values for $X_\alpha, f_\kappa, (u_{0\kappa}, v_{0\kappa}), t_\kappa$, and R_κ, and compute the reprojection error E. Let $c = 0.0001$.
2. Evaluate the first and second derivatives $\frac{\partial E}{\partial \xi_k}$ and $\frac{\partial^2 E}{\partial \xi_k \partial \xi_l}$, $k, l = 1, \dots, 3N + 9M - 7$.
3. Solve the following linear equation to determine $\Delta\xi_k$, $k = 1, \dots, 3N + 9M - 7$.

$$\underbrace{\begin{pmatrix} (1+c)\frac{\partial^2 E}{\partial\xi_1^2} & \frac{\partial^2 E}{\partial\xi_1\partial\xi_2} & \frac{\partial^2 E}{\partial\xi_1\partial\xi_3} & \cdots \\ \frac{\partial^2 E}{\partial\xi_2\partial\xi_1} & (1+c)\frac{\partial^2 E}{\partial\xi_2^2} & \frac{\partial^2 E}{\partial\xi_2\partial\xi_3} & \cdots \\ \frac{\partial^2 E}{\partial\xi_3\partial\xi_1} & \frac{\partial^2 E}{\partial\xi_3\partial\xi_2} & (1+c)\frac{\partial^2 E}{\partial\xi_3^2} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix}}_{Hessian} \cdot \underbrace{\begin{pmatrix}\Delta\xi_1\\ \Delta\xi_2\\ \Delta\xi_3\\ \vdots\end{pmatrix}}_{\Delta\xi} = -\underbrace{\begin{pmatrix}\frac{\partial E}{\partial\xi_1}\\ \frac{\partial E}{\partial\xi_2}\\ \frac{\partial E}{\partial\xi_3}\\ \vdots\end{pmatrix}}_{Gradient}, \qquad (11.7)$$

4. Update $\mathrm{X}_\alpha, f_\kappa, (u_{0\kappa}, v_{0\kappa}), t_\kappa$, and R_κ to

$$\tilde{\mathrm{X}}_\alpha \leftarrow \mathrm{X}_\alpha + \Delta\mathrm{X}_\alpha, \qquad \tilde{f}_\kappa \leftarrow f_\kappa + \Delta f_\kappa, \qquad (\tilde{u}_{0\kappa}, \tilde{v}_{0\kappa}) \leftarrow (u_{0\kappa}, v_{0\kappa}), \qquad \tilde{t}_\kappa \leftarrow t_\kappa + \Delta t_\kappa,$$
$$\tilde{R}_\kappa \leftarrow R(\Delta\omega_\kappa)R_\kappa$$

where $R(\Delta\omega_\kappa)$ denotes the matrix of rotation around $\Delta\omega_\kappa$ by $\|\Delta\omega\kappa\|$.

5. Compute the reprojection error $\tilde{E}$ for $\tilde{\mathrm{X}}_\alpha, \tilde{f}_\kappa, (\tilde{u}_{0\kappa}, \tilde{v}_{0\kappa}), \tilde{t}_\kappa$, and $\tilde{R}_\kappa$.
If $\tilde{E} > E$,
let $c \leftarrow 10c$, and go back to Step 3.

6. Update all the unknowns to

$$\mathrm{X}_\alpha \leftarrow \tilde{\mathrm{X}}_\alpha, \qquad f_\kappa \leftarrow \tilde{f}_\kappa, \qquad (u_{0\kappa}, v_{0\kappa}) \leftarrow (\tilde{u}_{0\kappa}, \tilde{v}_{0\kappa}), \qquad t_\kappa \leftarrow \tilde{t}_\kappa, \qquad R_\kappa \leftarrow \tilde{R}_\kappa$$

Stop if $|\tilde{E} - E| \le \delta$ for a pre-fixed small constant δ.
Else, let $E \leftarrow \tilde{E}$ and $c \leftarrow c/10$, and go back to Step 2.

Comments
In usual numerical iterations, the variables are successively updated until they no longer change. However, the number of unknowns for bundle adjustment is thousands or even tens of thousands, and an **impractically long computation time** would be necessary if all variables were required to converge over significant digits. However, the purpose of bundle adjustment is to find a solution that minimizes the reprojection error. Hence, it is a practical compromise to **stop if the reprojection error almost ceases to decrease**, as we described in the above procedure. If we are to stop if the update of the reprojection error is less than ε pixels per point, the constant δ above is $\frac{n\varepsilon^2}{f_0^2}$, where $n = \sum_{\alpha=1}^{N}\sum_{\kappa=1}^{M} I_{\alpha\kappa}$ (= The number of visible points). In real applications, it is reasonable to stop at around $\varepsilon = 0.01$ pixels.

Derivative

We use the optimization scheme

$$H \cdot \Delta\xi = -g$$

where H is the hessian and g is the gradient. We therefore need to find the first and second order derivative of E w.r.t. the unknown variables ξ.

We want to rewrite the error cost function E. We define

$$p_{\alpha\kappa} = P_{\kappa(11)}X_\alpha + P_{\kappa(12)}Y_\alpha + P_{\kappa(13)}Z_\alpha + P_{\kappa(14)}$$
$$q_{\alpha\kappa} = P_{\kappa(21)}X_\alpha + P_{\kappa(22)}Y_\alpha + P_{\kappa(23)}Z_\alpha + P_{\kappa(24)}$$
$$r_{\alpha\kappa} = P_{\kappa(31)}X_\alpha + P_{\kappa(32)}Y_\alpha + P_{\kappa(33)}Z_\alpha + P_{\kappa(34)}$$

So that we can express E as:

$$E = \sum_{\alpha=1}^{N}\sum_{\kappa=1}^{M} I_{\alpha\kappa} \cdot \left(\left(\frac{p_{\alpha\kappa}}{r_{\alpha\kappa}} - \frac{x_{\alpha\kappa}}{f_0}\right)^2 + \left(\frac{q_{\alpha\kappa}}{r_{\alpha\kappa}} - \frac{y_{\alpha\kappa}}{f_0}\right)^2\right), \qquad (11.13)$$

The first derivative is then

$$g = \frac{\partial E}{\partial\xi_\kappa} = 2\sum_{\alpha=1}^{N}\sum_{\kappa=1}^{M}\frac{I_{\alpha\kappa}}{r_{\alpha\kappa}^2} \cdot \begin{pmatrix} \left(\frac{p_{\alpha\kappa}}{r_{\alpha\kappa}} - \frac{x_{\alpha\kappa}}{f_0}\right)\left(r_{\alpha\kappa}\frac{\partial p_{\alpha\kappa}}{\partial\xi_\kappa} - p_{\alpha\kappa}\frac{\partial r_{\alpha\kappa}}{\partial\xi_\kappa}\right) \\ + \left(\frac{q_{\alpha\kappa}}{r_{\alpha\kappa}} - \frac{y_{\alpha\kappa}}{f_0}\right)\left(r_{\alpha\kappa}\frac{\partial q_{\alpha\kappa}}{\partial\xi_\kappa} - q_{\alpha\kappa}\frac{\partial r_{\alpha\kappa}}{\partial\xi_\kappa}\right) \end{pmatrix}, \qquad (11.14)$$

Next, we consider the second derivatives. We note that after a sufficient number of iterations Eq. (11.13) becomes very small such that $\frac{p_{\alpha\kappa}}{r_{\alpha\kappa}} - \frac{x_{\alpha\kappa}}{f_0} \approx 0$ and $\frac{q_{\alpha\kappa}}{r_{\alpha\kappa}} - y_{\alpha\kappa}/f_0 \approx 0$. Therefore we ignore terms that contain $p_{\alpha\kappa}/r_{\alpha\kappa} - x_{\alpha\kappa}/f_0$ and $q_{\alpha\kappa}/r_{\alpha\kappa} - y_{\alpha\kappa}/f_0$. This is known as **Gauss-Newton approximation**. In minimization iterations, the second derivatives affect the speed of convergence, but iterations are repeated until $\frac{\partial E}{\partial\xi_\kappa}$ is close to 0. Hence, the **accuracy of the solution is not affected by approximation** of second derivatives. The second derivatives of E with Gauss-Newton approximation have the expression:

$$H = \frac{\partial E^2}{\partial\xi_\kappa\partial\xi_l} = 2\sum_{\alpha=1}^{N}\sum_{\kappa=1}^{M}\frac{I_{\alpha\kappa}}{r_{\alpha\kappa}^4} \cdot \begin{pmatrix} \left(r_{\alpha\kappa}\frac{\partial p_{\alpha\kappa}}{\partial\xi_\kappa} - p_{\alpha\kappa}\frac{\partial r_{\alpha\kappa}}{\partial\xi_\kappa}\right)\left(r_{\alpha\kappa}\frac{\partial p_{\alpha\kappa}}{\partial\xi_l} - p_{\alpha\kappa}\frac{\partial r_{\alpha\kappa}}{\partial\xi_l}\right) \\ + \left(r_{\alpha\kappa}\frac{\partial q_{\alpha\kappa}}{\partial\xi_\kappa} - q_{\alpha\kappa}\frac{\partial r_{\alpha\kappa}}{\partial\xi_\kappa}\right)\left(r_{\alpha\kappa}\frac{\partial q_{\alpha\kappa}}{\partial\xi_l} - q_{\alpha\kappa}\frac{\partial r_{\alpha\kappa}}{\partial\xi_l}\right) \end{pmatrix}, \qquad (11.15)$$

From Eqs. (11.14) and (11.15), we see that for evaluating the first derivatives $\frac{\partial E}{\partial\xi k}$ and the second derivatives $\frac{\partial^2 E}{\partial\xi_k\partial\xi_l}$ of E, **we need to evaluate only the first derivatives** $\frac{\partial p_{\alpha\kappa}}{\partial\xi_k}, \frac{\partial q_{\alpha\kappa}}{\partial\xi_k}$, and $\frac{\partial r_{\alpha\kappa}}{\partial\xi_k}$.

The derivation w.r.t each of the variable can be seen in the subsection.

Efficient Computation and Memory Use

A significant part of the summed terms of $\sum_{\alpha=1}^{N}\sum_{\kappa=1}^{M}$ in gradient and hessian of the cost function, Equation (11.14) and (11.15), is 0, which mean that we do not waste computer time evaluating these.
We defined Kronecker delta.

$$\delta_{\alpha\beta} = \begin{cases} 1, & if\ \alpha = \beta \\ 0 & otherwise \end{cases}$$

Consider the gradient:
If $\Delta\xi_k$ is the update ΔX_β of the βth point, we need to evaluate in the summation $\sum_{\alpha=1}^{N}$ **only terms for which $\alpha = \beta$** due to the Kronecker delta $\delta_{\alpha\beta}$ in Eq. (11.16). If $\Delta\xi_k$ is the update of $f_\lambda, (u_{0\lambda}, v_{0\lambda}), t_\lambda$, or R_λ of the λth image, we need to evaluate in the summation $\sum_{\kappa=1}^{M}$ **only terms for which $\kappa = \lambda$** due to the Kronecker delta $\delta_{\kappa\lambda}$ in Eqs. (11.17), (11.18), (11.19), and (11.21). Thus the summation $\sum_{\alpha=1}^{N}\sum_{\kappa=1}^{M}$ in **Eq. (11.14) need to be summed over either α or κ** and not not a double loop.
Moreover, we need to consider for the αth point only those κth images in which it is seen and for the κth image only those αth points that are seen there.

Consider the Hessian:
If $\Delta\xi_k$ and $\Delta\xi_l$ both refer to points, Eq. (11.15) is **0 if they are different points**, and if they are the same point, we need to evaluate in $\sum_{\alpha=1}^{N}$ only the term corresponding to that point. If $\Delta\xi_k$ and $\Delta\xi_l$ both refer to images, Eq. (11.15) is **0 if they are different images**, and if they are the same image, we need to evaluate in $\sum_{\kappa=1}^{M}$ only the term corresponding to that image. If one of $\Delta\xi_k$ and $\Delta\xi_l$ refers to a point and the other to an image, we need to evaluate in $\sum_{\alpha=1}^{N}\sum_{\kappa=1}^{M}$ **only the term corresponding to that point and that image**, provided that point is seen in that image

So far we have been able to reduce the computational time for the gradient and Hessian. However, the Hessian matrix is of size $(3N + 9M - 7) \times (3N + 9M - 7)$ which cannot be stored in memory when N

and M is large.
We defined 3 matrices
$E \in \mathbb{R}^{3N\times 3N}$: Stores second derivatives w.r.t points, such as $\frac{\partial^2 E}{\partial X_\alpha^2}$ and $\frac{\partial^2 E}{\partial X_\alpha \partial Y_\alpha}$.
$F \in \mathbb{R}^{3N\times 9M}$: Stores second derivatives w.r.t points and images, such as $\frac{\partial^2 E}{\partial X_\alpha \partial f_\kappa}$, $\frac{\partial^2 E}{\partial X_\alpha \partial u_{0\kappa}}$.
$G \in \mathbb{R}^{9M\times 9M}$: Stores second derivatives w.r.t images, such as $\frac{\partial^2 E}{\partial f_\kappa^2}$, and $\frac{\partial^2 E}{\partial f_\kappa \partial u_{0\kappa}}$.
This reduces the required memory size to $27NM + 9N + 81M$. If we remove the redundancies in the second derivative elements, the **net array elements are $\boldsymbol{27NM + 6N + 41M}$** in number, among which those of $I_{\alpha\kappa} = 0$ are not necessary. Considering all these, it is better to define the Hessian $H(k,l)$ not as an array but as a function program that returns its (k, l) element.

Solving the linear equations

We now want to solve Equation (11.7). As stated in the above section, the Hessian matrices can be too large to be in working memory. We therefore represent the Hessian matrix as

$$\underbrace{\begin{pmatrix} E_1^{(c)} & & & F_1 \\ & \ddots & & \vdots \\ & & E_N^{(c)} & F_N \\ F_1^T & \dots & F_N^T & G^{(c)} \end{pmatrix}}_{(3N+9M-7)\times(3N+9M-7)} \begin{pmatrix} \Delta\xi_P \\ \Delta\xi_F \end{pmatrix} = -\begin{pmatrix} \boldsymbol{d}_P \\ \boldsymbol{d}_F \end{pmatrix}$$

where
- $\Delta\xi_P \in \mathbb{R}^{3N}$: is the vector for the **3D positions**
- $\Delta\xi_F \in \mathbb{R}^{9M-7}$: is the vector for the **camera parameters**
- $\boldsymbol{d}_P \in \mathbb{R}^{3N}$ and $\boldsymbol{d}_F \in \mathbb{R}^{9M-7}$ are the corresponding parts of the right side of Eq. (11.7).
- $E_\alpha^{(c)} \in \mathbb{R}^{3\times 3}$: contains the second derivatives of the reprojection error E w.r.t. the αth point.
- $F_\alpha \in \mathbb{R}^{3\times(9M-7)}$: contains the second derivatives of E w.r.t the αth point and the parameters of the cameras viewing it,
- $G^{(c)} \in \mathbb{R}^{(9M-7)\times(9M-7)}$: contains the second derivatives of E w.r.t. the camera parameters

The superscript (c) means that the diagonal elements are multiplied by $(1+c)$.

Procedure 11.2 - Efficient linear equation solving
1. For each α, let ∇_{X_α} be
$$\nabla_{X_\alpha} E = \begin{pmatrix} \partial E/\partial X_\alpha \\ \partial E/\partial Y_\alpha \\ \partial E/\partial Z_\alpha \end{pmatrix}$$
2. Solve the following $(9M-7)$D linear equations for $\Delta\xi_F$.
$$\left(G^{(c)} - \sum_{\alpha=1}^{N} F_\alpha^T \left(E_\alpha^{(c)}\right)^{-1} F_\alpha \right) \Delta\xi_F = \sum_{\alpha=1}^{N} F_\alpha^T \left(E_\alpha^{(c)}\right)^{-1} \nabla_{X_\alpha} E - \boldsymbol{d}_F, \qquad (11.24)$$
3. Compute $\Delta X_\alpha, \Delta Y_\alpha$, and ΔZ_α for the αth point by
$$\begin{pmatrix} \Delta X_\alpha \\ \Delta Y_\alpha \\ \Delta Z_\alpha \end{pmatrix} = -\left(E_\alpha^{(c)}\right)^{-1} \left(F_\alpha \Delta\xi_F + \nabla_{X_\alpha} E \right)$$

Comments:
The standard method for solving linear equations are Cholesky decomposition for symmetric coefficient matrices and LU for general coefficient matrices. They both have complexity nearly proportional to the cube of the number of unknowns ~ $O\left(3N + pM - 7\right)^3$. However, in equation (11.24) we reduce the number of unknows and thereby the complexity to $O(9M-7)$. This is a great improvement when the number of points, $\boldsymbol{N}$**, is large** and the number of images, $\boldsymbol{M}$**, is small**.

Furthermore, we greatly reduce the memory usage:
The memory space required for storing intermediate values is only N arrays $E_\alpha^{(c)}$ of size 3×3, N arrays F_α of size $3\times(9M-7)$, one array $G^{(c)}$ of size $(9M-7)\times(9M-7)$, one $3N$D vector $\boldsymbol{d}_P$, and one $(9M-7)$D vector $\boldsymbol{d}_F$.

The derivation of the equation for procedure 11.2 can be seen in the sub notes.

Bundle Adjustment (BA)
Lecture slides.

We can formulate BA though

$$\underbrace{{}^{(a)}x'_{ij}}_{\substack{ij=\text{point } i \\ \text{observed in} \\ \text{image } j}} + \underbrace{{}^{(a)}\hat{v}_{x'_{ij}}}_{\text{Correction}} = \underbrace{\hat{\lambda}_{ij}}_{\substack{\text{Scale} \\ \text{factor}}} \cdot \underbrace{{}^{(a)}\hat{P}_j\left(\overbrace{x_{ij}}^{\substack{\text{image} \\ \text{pixel}}}, \overbrace{\hat{p}}^{\substack{\text{Projection} \\ \text{params}}}, \overbrace{\hat{q}}^{\text{Distorsion}} \right)}_{\substack{\text{Projection matix} \\ (\text{w} \mid \text{non-lin.Calib.})}} \cdot \underbrace{\hat{X}_i}_{\substack{3D \\ \text{point}}},$$

$$\text{with} \quad \underbrace{\Sigma_{x_{ij}x_{ij}}}_{\substack{\text{Uncertainty} \\ \text{in the image} \\ \text{coordinates}}} \quad i = 1, \dots, \underbrace{I_j}_{\substack{\text{points in} \\ \text{image } j}} \quad ; j = 1, \dots, \underbrace{J}_{\#\text{ images}}$$

a: arbitrary frame
$\hat{v}$: correction
$\hat{\lambda}$: scale factor - because we work with homogeneous coordinates
$\hat{P}$: Projection matrix (w/ non-liniear calibration)
$\hat{X}$: 3D Point
x'_{ij}: point i observed in image j
x_{ij}: True image pixel, to adjust for distortion
$\Sigma_{x_{ij}x_{ij}}$: Image pixel measurement error. This will be large if the iamge is blured.

This function encodes the projection from the 3D world to the image coordinate system and it encodes the optimal triangulation constraint.
The optimal solution is the statistically best solution, and this is given that the **errors are normal distribution**, which is rarely the case

The unknown in the equation are
- 3D locations of new points $\hat{X}_i$. (Calculated for each point)
- 1D scale factor $\hat{\lambda}_{ij}$. (Calculate for each point in each image)
- 6D exterior orientation (Calculated for each camera)
- 5D projection parameters (interior orientation). (Calculated once)
- Non-linear distortion parameters q. (Calculated once)

An example

10k images, 1k points per image
Each point is seen 10 times on avg.

How many unknown do we have?
- Observations: $2 \times 10k \times 1k = 20M$
- Unknowns: ~1M (pts) w/ 3 DoF + 10M (scale) + 10k (orientations) w 6 DoF
- ~13 Mio unknown dimensions and ~20 Mio observations

		If we convert the homogeneous coordinates into Euclidian coordinates, we can eliminate the per-point scale factor, which result in 3 Mio unknown.	
	Setting up and solving the System of normal equations	Here we use the standard procedure. We called *u* the unknown and *x* the observations Se set up the linear equation $\underbrace{A^T\Sigma^{-1}A}_{A_-} \cdot \underbrace{\Delta u}_{x_-} = \underbrace{A^T\Sigma^{-1}\Delta x}_{b_-}$ Which yields (least square solution) $\widehat{\Delta u} = \underbrace{\left(A^T\Sigma^{-1}A\right)}_{A_-}^{-1} \underbrace{A^T\Sigma^{-1}\Delta x}_{b_-}$ And the corrections $\hat{v} = A\widehat{\Delta u} - \Delta x$	
	Issues	Because it is an nonlinear optimization problem, it needs a good initial guess before it can make a proper solution. Camera calibration is a constrained version of the bundle adjustment, where the 3D points are known	
	Gross Errors /Outliers	The definition of Gross error is an error given carelessness and mainly appear because of humans. In Bundle Adjustment gross errors happens because of: - **Wrong correspondence** (the least square solution have a high respondence for outliers) - Wrong point measurements We can filter out the wrong correspondences by having enough observations of each point. We need at **least 4 views** of a point correspondence to identify the observation with a gross error. Points that have **5 to 6 different views** typically yield good estimates. Elimination gross errors - Decompose the problem into several small block of 3-6 images first - Check for gross errors in the small blocks using statistical tests - Eliminate gross errors - Compute the full bundle adjustment - Separate check of photogrammetric observations and control points. That is, not estimating the camera parameters and all the 3D points at the same time.	
	Bundle adjustment without control points Gauge freedom	The issues with not having control points is that **the scale of all the parameters of the camera can vary**. Which mean that no unique solution can be found, and thereby give rise to a condition called the **gauge freedom** (datums defect). The gauge freedom is often showing up when a system is over parameterized. For the Bundle adjustment gauge freedom appears if we use homogeneous coordinates to represent the points, or if we do not fix the one of the camera orientation. We therefore set the y distance between camera 0 and 1 to 1 - as done in equation (11.6) form the theory part - and place the first camera in the world coordinates .	
	Control points (CP)	Sometimes we have some points we know are very **accurate**, and we therefore do not want to adjust those, and might want to reduce/**remove the correction term for these points**. We have a two-step BA methods for this: 1. First we do a Bundle Adjustment with noisy control points ○ Search for gross errors using statistical tests $T = \hat{v}_{X_i}^T \Sigma_{\hat{v}_{X_i}\hat{v}_{X_i}}^{-1} \hat{v}_{X_i} \sim \chi_3^2$ ○ And eliminate erroneous control points 2. Do BA with fixed control points ○ BA with fixed C.P. to enforce the geometry to match the C.P. ○ Used if the C.P.s cannot be changed (e.g. data from external source) ○ Suboptimal approach - considering noisy C.P.s would be optimal We use fixed control point for mapping, because we usually have the same control point for multiple maps, so we want these to be consistent.	
	Number of control points	For Direct Linear Transformation (DLT) or P3P solutions, we need 3-6 control points per image pair. However, in Bundle Adjustment only a small number of control points are needed for the whole sequence of images - an exact number cannot be defined. More control points are needed at the boundary of the covers space, because there are fewer overlapping views. The small number of Control Points are a big advantages of Bundle Adjustments.	
	BA Properties - Optimality	- BA is statistically optimal - Exploits all observations and considers the uncertainties and correlations - Exploits all available information - Computes orientations, calibration parameters, and 3D point locations with highest precision - Requires an initial estimate - Assumes Gaussian distributions	
	Initializing the bundle adjustment algorithm	Because of the huge number of parameters, there are no way direct forward calculation of the model parameters, an iterative initialization is therefore done: - We have some control points that are seen from the first two cameras, and then we use these to find estimate the camera parameters. - We then use the overlapping corresponding points from the first two cameras with the third camera as control points to estimate the camera parameters of the third camera. - Ect. For the rest N cameras. We then do bundle Adjustment.	
	Solving the least square problem	When we have initialized our least square optimization problem, we want to solve it. However, we cannot solve the problem in a straightforward manner because of the enormous amount of parameters. We therefore explore the sparse nature of the matrixes, as discussed in the theory part.	
Summary		- Bundle adjustment = least square solution to relative and absolute orientation considering uncertainties - Statistically optimal solution - BA leads to sparse matrices - Often sequential solution of orientation parameters and point coordinates	

Geometry of Calibrated Versus uncalibrated cameras		The advantages of the uncalibrated camera model is that it has 12 degree of freedom - the camera matrix is 3×4, and its easy to use standard linear algebra because no constrains are imposed on the system. The calibrated camera only has 6 degree of freedom - the postion and orientation. This result in a non-linear manifold solution. However, in practice, we like to work with the calibrated camera, because the physical mean of fewer degree of freedom in the calibrated case, is that we have a better modelled system. If we define a full rank 4×4 matric, H, we can manipulate the **uncalibrated** pinhole model in the following way: $\underbrace{q_{ij}}_{observed} = \underbrace{P_i Q_j}_{Unknown} = P_i H^{-1} H Q_j = \underbrace{\tilde{P}_i \tilde{Q}_j}_{Unknown}, \quad (3.1)$ where $\tilde{P}_i = P_i H^{-1}, \quad and, \quad \tilde{Q}_j = H Q_j$ Since we have only observed q_{ij}, we cannot know if P_i and Q_j or $\tilde{P}_i$ and $\tilde{Q}_j$ is the solution so to our equation. That is, in the uncalibrated case, we can only define the solution up to an arbitrary transformation of a full rank matric, H. For the **calibrated** pinhole model equation (3.1) still holds, but we know that the internal parameters, A, constains H in the form $H_{cal} = \begin{bmatrix} \tilde{s} \cdot \tilde{R} & \tilde{t} \\ 000 & 1 \end{bmatrix}$ where $\tilde{s}$ is an arbitrary scale, $\tilde{R}$ an arbitrary rotation and $\tilde{t}$ an arbitrary translation. This, H_{cal} is reduce to being **an arbitrary definition of the global coordinate system and an arbitrary scaling**.	

Derivatives

Saturday, 6 June 2020 14.45

11.3.2 Derivatives with Respect to 3D Positions

We write the derivation with respect to X_β, Y_β, and Z_β as the vector operator $\nabla_{\mathsf{X}_\beta} = (\partial/\partial X_\beta, \partial/\partial Y_\beta, \partial/\partial Z_\beta)^\top$. The derivatives of $p_{\alpha\kappa}$, $q_{\alpha\kappa}$, and $r_{\alpha\kappa}$ with respect to X_β, Y_β, and Z_β are written in the following form ($\hookrightarrow$ Problem 11.2(1)).

$$\nabla_{\mathsf{X}_\beta} p_{\alpha\kappa} = \delta_{\alpha\beta}\begin{pmatrix} P_{\kappa(11)} \\ P_{\kappa(12)} \\ P_{\kappa(13)} \end{pmatrix}, \quad \nabla_{\mathsf{X}_\beta} q_{\alpha\kappa} = \delta_{\alpha\beta}\begin{pmatrix} P_{\kappa(21)} \\ P_{\kappa(22)} \\ P_{\kappa(23)} \end{pmatrix}, \quad \nabla_{\mathsf{X}_\beta} r_{\alpha\kappa} = \delta_{\alpha\beta}\begin{pmatrix} P_{\kappa(31)} \\ P_{\kappa(32)} \\ P_{\kappa(33)} \end{pmatrix}. \tag{11.16}$$

Here, $\delta_{\alpha\beta}$ is the Kronecker delta, taking 1 for $\alpha = \beta$ and 0 otherwise.

11.3.3 Derivatives with Respect to Focal Lengths

The derivatives of $p_{\alpha\kappa}$, $q_{\alpha\kappa}$, and $r_{\alpha\kappa}$ with respect to f_λ are given as follows ($\hookrightarrow$ Problem 11.2(2)).

$$\frac{\partial p_{\alpha\kappa}}{\partial f_\lambda} = \frac{\delta_{\kappa\lambda}}{f_\kappa}\Big(p_{\alpha\kappa} - \frac{u_0}{f_0} r_{\alpha\kappa}\Big), \quad \frac{\partial q_{\alpha\kappa}}{\partial f_\lambda} = \frac{\delta_{\kappa\lambda}}{f_\kappa}\Big(q_{\alpha\kappa} - \frac{v_0}{f_0} r_{\alpha\kappa}\Big), \quad \frac{\partial r_{\alpha\kappa}}{\partial f_\lambda} = 0. \tag{11.17}$$

11.3.4 Derivatives with Respect to Principal Points

We write the derivation with respect to $u_{0\lambda}$ and $v_{0\lambda}$ as the vector operator $\nabla_{\mathbf{u}_{0\lambda}} = (\partial/\partial u_{0\lambda}, \partial/\partial u_{0\lambda})^\top$. The derivatives of $p_{\alpha\kappa}$, $q_{\alpha\kappa}$, and $r_{\alpha\kappa}$ with respect to $u_{0\lambda}$ and $v_{0\lambda}$ are written in the following form ($\hookrightarrow$ Problem 11.2(3)).

$$\nabla_{\mathbf{u}_{0\lambda}} p_{\alpha\kappa} = \begin{pmatrix} \delta_{\kappa\lambda} r_{\alpha\kappa}/f_0 \\ 0 \end{pmatrix}, \quad \nabla_{\mathbf{u}_{0\lambda}} q_{\alpha\kappa} = \begin{pmatrix} 0 \\ \delta_{\kappa\lambda} r_{\alpha\kappa}/f_0 \end{pmatrix}, \quad \nabla_{\mathbf{u}_{0\lambda}} r_{\alpha\kappa} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}. \tag{11.18}$$

11.3.5 Derivatives with Respect to Translations

We write the derivation with respect to $t_{\lambda 1}$, $t_{\lambda 2}$, and $t_{\lambda 3}$ as the vector operator $\nabla_{\mathbf{t}_\lambda} = (\partial/\partial t_{\lambda 1}, \partial/\partial t_{\lambda 2}, \partial/\partial t_{\lambda 3})^\top$. The derivatives of $p_{\alpha\kappa}$, $q_{\alpha\kappa}$, and $r_{\alpha\kappa}$ with respect to $t_{\lambda 1}$, $t_{\lambda 2}$, and $t_{\lambda 3}$ are written in the following form ($\hookrightarrow$ Problem 11.2(4)).

$$\nabla_{\mathbf{t}_\lambda} p_{\alpha\kappa} = -\delta_{\kappa\lambda}(f_\kappa \mathbf{r}_{\kappa 1} + u_0 \mathbf{r}_{\kappa 3}), \quad \nabla_{\mathbf{t}_\lambda} q_{\alpha\kappa} = -\delta_{\kappa\lambda}(f_\kappa \mathbf{r}_{\kappa 2} + v_0 \mathbf{r}_{\kappa 3}),$$

$$\nabla_{\mathbf{t}_\lambda} r_{\alpha\kappa} = -\delta_{\kappa\lambda} f_0 \mathbf{r}_{\kappa 3}. \tag{11.19}$$

Here, $\mathbf{r}_{\kappa 1}$, $\mathbf{r}_{\kappa 2}$, and $\mathbf{r}_{\kappa 3}$ are, respectively, the first, the second, and the third columns of the rotation R_κ; that is,

$$\mathbf{r}_{\kappa 1} = \begin{pmatrix} R_{\kappa(11)} \\ R_{\kappa(21)} \\ R_{\kappa(31)} \end{pmatrix}, \quad \mathbf{r}_{\kappa 2} = \begin{pmatrix} R_{\kappa(12)} \\ R_{\kappa(22)} \\ R_{\kappa(32)} \end{pmatrix}, \quad \mathbf{r}_{\kappa 3} = \begin{pmatrix} R_{\kappa(13)} \\ R_{\kappa(23)} \\ R_{\kappa(33)} \end{pmatrix}, \tag{11.20}$$

where $R_{\kappa(ij)}$ is the (i, j) element of R_κ.

11.3.6 Derivatives with Respect to Rotations

We write the derivations with respect to the rotation R_λ as the vector operator $\nabla_{\omega_\lambda} = (\partial/\partial\omega_{\lambda 1}, \partial/\partial\omega_{\lambda 2}, \partial/\partial\omega_{\lambda 3})^\top$. The derivatives of $p_{\alpha\kappa}$, $q_{\alpha\kappa}$, and $r_{\alpha\kappa}$ with respect to R_λ are given in the following form ($\hookrightarrow$ Problem 11.2(5)).

$$\begin{aligned} \nabla_{\omega_\lambda} p_{\alpha\kappa} &= \delta_{\kappa\lambda}(f_\kappa \mathbf{r}_{\kappa 1} + u_{0\kappa}\mathbf{r}_{\kappa 3}) \times (\mathsf{X}_\alpha - \mathbf{t}_\kappa), \\ \nabla_{\omega_\lambda} q_{\alpha\kappa} &= \delta_{\kappa\lambda}(f_\kappa \mathbf{r}_{\kappa 2} + v_{0\kappa}\mathbf{r}_{\kappa 3}) \times (\mathsf{X}_\alpha - \mathbf{t}_\kappa), \\ \nabla_{\omega_\lambda} r_{\alpha\kappa} &= \delta_{\kappa\lambda} f_0 \mathbf{r}_{\kappa 3} \times (\mathsf{X}_\alpha - \mathbf{t}_\kappa). \end{aligned} \tag{11.21}$$

Derivation of Procedure 11.2

Procedure 11.2 is obtained by the following consideration ($\hookrightarrow$ Problem 11.3). Equation (11.22) is split into two parts in the form

$$\begin{pmatrix} \mathsf{E}_1^{(c)} & & \\ & \ddots & \\ & & \mathsf{E}_N^{(c)} \end{pmatrix} \Delta\boldsymbol{\xi}_P + \begin{pmatrix} \mathsf{F}_1 \\ \vdots \\ \mathsf{F}_N \end{pmatrix} \Delta\boldsymbol{\xi}_F = -\mathbf{d}_P \tag{11.26}$$

$$\begin{pmatrix} \mathsf{F}_1^\top & \cdots & \mathsf{F}_N^\top \end{pmatrix} \Delta\boldsymbol{\xi}_P + \mathsf{G}^{(c)}\Delta\boldsymbol{\xi}_F = -\mathbf{d}_F. \tag{11.27}$$

We first solve Eq. (11.26) for $\Delta\boldsymbol{\xi}_P$. If we note that $\mathbf{d}_P$ is a vertical array of $\nabla_{\mathsf{X}_\alpha} E$, $\alpha = 1, \ldots, N$, of Eq. (11.23), we can write the solution in the form

$$\begin{aligned} \Delta\boldsymbol{\xi}_P &= -\begin{pmatrix} \mathsf{E}_1^{(c)-1} & & \\ & \ddots & \\ & & \mathsf{E}_N^{(c)-1} \end{pmatrix} \begin{pmatrix} \mathsf{F}_1 \\ \vdots \\ \mathsf{F}_N \end{pmatrix} \Delta\boldsymbol{\xi}_F - \begin{pmatrix} \mathsf{E}^{(c)-1} & & \\ & \ddots & \\ & & \mathsf{E}^{(c)-1} \end{pmatrix} \mathbf{d}_P \\ &= -\begin{pmatrix} \mathsf{E}_1^{(c)-1}\mathsf{F}_1 \\ \vdots \\ \mathsf{E}_N^{(c)-1}\mathsf{F}_N \end{pmatrix} \Delta\boldsymbol{\xi}_F - \begin{pmatrix} \mathsf{E}_1^{(c)-1}\nabla_{\mathsf{X}_1} E \\ \vdots \\ \mathsf{E}_N^{(c)-1}\nabla_{\mathsf{X}_N} E \end{pmatrix}. \end{aligned} \tag{11.28}$$

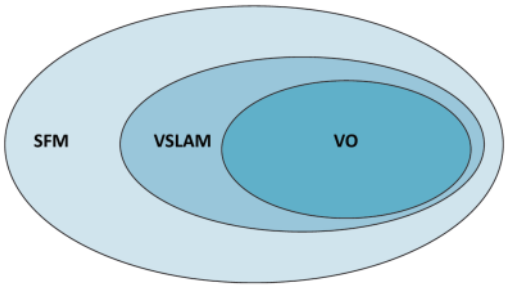
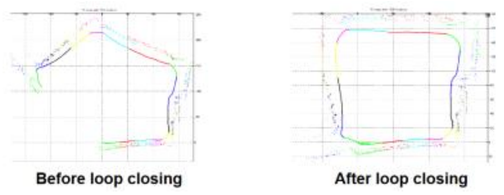
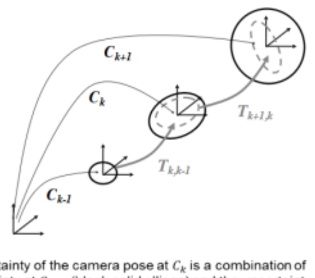
Substituting this into Eq. (11.27), we obtain

$$\begin{aligned} &-\begin{pmatrix} \mathsf{F}_1^\top & \cdots & \mathsf{F}_N^\top \end{pmatrix} \begin{pmatrix} \mathsf{E}_1^{(c)-1}\mathsf{F}_1 \\ \vdots \\ \mathsf{E}_N^{(c)-1}\mathsf{F}_N \end{pmatrix} \Delta\boldsymbol{\xi}_F - \begin{pmatrix} \mathsf{F}_1^\top & \cdots & \mathsf{F}_N^\top \end{pmatrix} \begin{pmatrix} \mathsf{E}_1^{(c)-1}\nabla_{\mathsf{X}_1} E \\ \vdots \\ \mathsf{E}_N^{(c)-1}\nabla_{\mathsf{X}_N} E \end{pmatrix} + \mathsf{G}^{(c)}\Delta\boldsymbol{\xi}_F \\ &= -\mathbf{d}_F, \end{aligned} \tag{11.29}$$

from which Eq. (11.24) is obtained. The part of Eq. (11.28) corresponding to X_α is written in the form of Eq. (11.25).

Visual SLAM

Thursday, 20 February 2020 08.42

Visual Odometry (VO)		**VO is the process of estimating the egomotion of an agent using only the input of a single or multiple cameras**. VO works by incrementally estimating the pose of the vehicle through examination of the changes that motion induces on the image of its onboard cameras. VO work most efficiently if it is in a **sufficient illuminated environment** and have a **static scene with enough texture** to allow apparent motion to be extracted. Furthermore, consecutive frames should have a **sufficient scene overlap**. VO is a particular case of Structure from Motion (SFM). SFM is general and tackle the problem of 3D reconstruction of both the **structure and camera poses** from sequentially ordered or unordered image sets. Where the final structure and camera pose typically are optimized offline afterwards. Whereas, **VO focuses on estimating the 3D motion** of the camera sequentially - as new frames arrives - and **in real time**. The real time version of bundle adjustment can then be used to refine the local estimate of the trajectory.	 
	VO vs V-SLAM	The compromise between **V-SLAM** and **VO** is, VO only concerned with the local consistency of the trajectory, whereas SLAM is concerned with the global consistency. This result that SLAM can correct its map when a loop is found. The sliding window optimization can be considered equivalent to building a local map in SLAM; however, the philosophy is different: in VO, we only care about **local consistency of the trajectory** and the local map is used to obtain a more accurate estimate of the local trajectory (for example, in bundle adjustment), whereas SLAM is concerned with the **global map consistency**. VO can be used as a building block for a complete SLAM algorithm to recover the incremental motion of the camera; however, to make a complete SLAM method, one must also add some way to **detect loop closing** and possibly a **global optimization step** to obtain a metrically consistent map (without this step, the map is still topologically consistent). A V-SLAM method is **potentially much more precise**, because it enforces many more constraints on the path, but **not necessarily more robust** (e.g., outliers in loop closing can severely affect the map consistency). In addition, it is more complex and computationally expensive. In the end, the choice between VO and V-SLAM depends on the **tradeoff between performance and consistency**, and simplicity in implementation. Although the global consistency of the camera path is sometimes desirable, VO trades off consistency for real-time performance, without the need to keep track of all the previous history of the camera.	
Stereo vs monocular VO		Stereo VO means that the relative 3D position of the measured features are directly measured by triangulation at every robot location and are used to derive the relative motion. A single camera can be used for this purpose. The other case is monocular VO, where the features can only be determined up to scale, and an environment measurement need to be done in order to determine the scale of the VO	
	Stereo VO	Different approached has been used including using RANSAC for outlier removal and ICP. A good approach turned out to be to, Instead of using 3-D-to-3-D point registration or 3-D-to-2-D camera-pose estimation techniques, they relied on the quadrifocal tensor, which allows motion to be computed from 2-D-to-2-D image matches without having to triangulate 3-D points in any of the stereo pairs. The benefit of using directly raw 2-D points in lieu of triangulated 3-D points lays in a more accurate motion computation.	
	Monocular VO	Monocular VO needs to determine both the relative motion and 3D structure from the 2D bearing data. As new images arrives, the relative scale and pose w.r.t. the first two frames are determine using either the knowledge of 3D structures or the trifocal tensor. Monocular VO can be divided into three categories: Feature-based methods, appearance-based methods, and hybrid methods **Feature-based methods**: - Are based on salient (noticeable) and repeatable features that are tracked over the frames - Features based on corner detection Appearance-based methods: - Used the intensity information of all the pixels in the image or subregions of it. - Not robust against occlusions Hybrid methods - A combination The appearance-base is not well statistical described, so we use feature base The above methods	
Reducing the drift		Because the camera path is calculated by an incrementally pose estimation (a pose is based on the previous pose), the error introduced by each new frame-to-frame motion accumulate over time. This generate a drift of the estimated trajectory form the real path. This drift can be reduced through **local optimization** over the last *m* camera poses. This approach is called *sliding window bundle adjustment* or *windowed bundle adjustment*. If we distribute our error as a normal distribution, it will be an ellipsoid that grows - given the covariance matrix. We can describe the covariance for each individual transformation. If we have $C_k = f(C_{k-1}, T_k)$ The combined covariance Σ_k is $\Sigma_k = J \begin{bmatrix} \Sigma_{k-1} & 0 \\ 0 & \Sigma_{k,k-1} \end{bmatrix} J^T$ Where J is the jacobian of f.	 C_{k+1} C_k $T_{k+1,k}$ $T_{k,k-1}$ C_{k-1} The uncertainty of the camera pose at C_k is a combination of

previous pose), the error introduced by each new frame-to-frame motion accumulate over time. This generate a drift of the estimated trajectory form the real path. This drift can be reduced through **local optimization** over the last *m* camera poses. This approach is called *sliding window bundle adjustment* or *windowed bundle adjustment*.

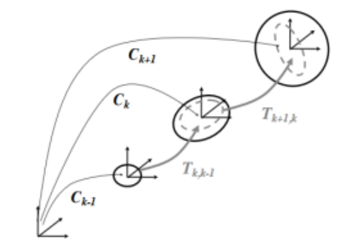
If we distribute our error as a normal distribution, it will be an ellipsoid that grows - given the covariance matrix. We can describe the covariance for each individual transformation.
If we have

$$C_k = f(C_{k-1}, T_k)$$

The combined covariance Σ_k is

$$\Sigma_k = J \begin{bmatrix} \Sigma_{k-1} & 0 \\ 0 & \Sigma_{k,k-1} \end{bmatrix} J^T$$

Where J is the jacobian of f.



The uncertainty of the camera pose at C_k is a combination of the uncertainty at C_{k-1} (black solid ellipse) and the uncertainty of the transformation $T_{k,k-1}$ (gray dashed ellipse)

VO problem formulation

An agent is moving through an environment and are taking images with a rigidly attached camera system at discrete time instances *k*. For the **stereo camera** case the set of images taking at *time k* is denoted $I_{l,0:n} = \{I_{l,0}, \dots, I_{l,n}\}$ by and $I_{r,0:n} = \{I_{r,0}, \dots, I_{r,n}\}$. The setting is show on the figure to the right.

Two camera positions at adjacent time instants $k-1$ and k are related by the rigid body transformation $T_{k,k-1} \in \mathbb{R}^{4\times4}$ of the following form:

$$T_{k,k-1} = \begin{bmatrix} R_{k,k-1} & t_{k,k-1} \\ 0 & 1 \end{bmatrix}$$

where $R_{k,k-1} \in SO(3)$ is the rotation matrix, and $t_{k,k-1} \in \mathbb{R}^3$ the translation vector. The set $T_{1:n} = \{T_{1,0}, \dots, T_{m,m-1}\}$ contains all subsequent motions. To simplify the notation, from now on, T_k will be used instead of $T_{k,k-1}$. Finally, the set of camera poses $C_{0:n} = \{C_0, \dots, C_n\}$ contains the transformations of the camera with respect to the initial coordinate frame at $k = 0$. The current pose C_n can be computed by concatenating all the transformations T_k $(k = 1 \dots n)$, and, therefore, $C_n = C_{n-1}Tn$, with C_0 being the camera pose at the instant $k = 0$, which can be set arbitrarily by the user.

In the VO we want to compute the relative transformations T_k from the image pairs I_{k-1} and I_k and then concatenate the transformations to recover the full trajectory $C_{0:n}$ of the camera. The pipeline can be seen on figure 2. That is, the VO recovers the path increamentally, **pose after pose**. A refinement can be done over the last m poses with e.g. **bundle adjustment.**

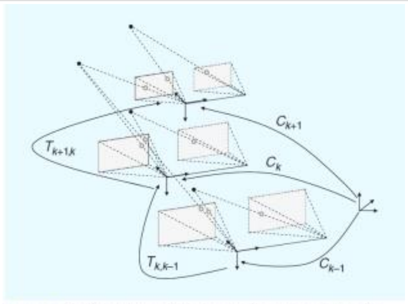


Figure 1. An illustration of the visual odometry problem. The relative poses $T_{k,k-1}$ of adjacent camera positions (or positions of a camera system) are computed from visual features and concatenated to get the absolute poses C_k with respect to the initial coordinate frame at $k = 0$.

Pipeline

The VO pipeline is summarized in Figure 2.

- For every new image I_k (or image pair in the case of a stereo camera), detect and match the 2-D features with those from the previous frames. Two-dimensional features that are the reprojection of the same 3-D feature across different frames are called image correspondences.
- We distinguish between **feature matching** and **feature tracking**.
  - **Feature matching** consists of detecting features independently in all the images and then matching them based on some similarity metrics.
  - **Feature tracking** consists of finding features in one image and then tracking them in the next images using a local search technique, such as correlation.
  - Feature tracking is more suitable when the images are taken from nearby viewpoints, whereas the latter is more suitable when a large motion or viewpoint change is expected
- The third step consists of computing the **relative motion $\boldsymbol{T_k}$** between the time instants $k-1$ and k. Depending on whether the correspondences are specified in three or two dimensions, there are **three distinct approaches** to tackle this problem.
- The camera pose C_k is then computed by concatenation of T_k with the previous pose.
- Finally, an iterative refinement (bundle adjustment) can be done over the last m frames to obtain a more accurate estimate of the local trajectory.

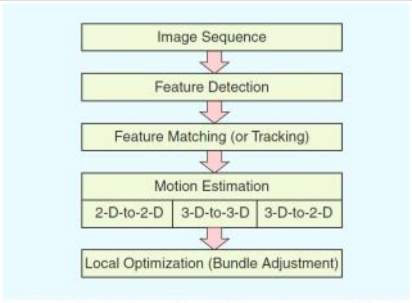


Figure 2. A block diagram showing the main components of a VO system.

Motion estimation using VO

The transformation T_k between two images I_{k-1} and I_k can be computed from two **sets of corresponding features** f_{k-1}, f_k at time instants $k-1$ and k, respectively. We have three methods to calculate the relative motion:

- 2D-to-2D
  - Both f_{k-1} and f_k are specified in 2D **image coordinates**
  - Requires a minimum of 5 correspondences (five-point algorithm)
- 3D-to-3D
  - Both f_{k-1} and f_k are specified in 3D. To do this, it is necessary to **triangulate 3D points** at each time instant, for instance, **by a stereo camera system**
- 3D-to-2D
  - In this case, f_{k-1} are specified in 3-D and f_k are their corresponding 2-D reprojections on the image I_k.
  - Requires 3 correspondences (P3P)
  - In the monocular case, the 3-D structure needs to be triangulated from two adjacent camera views (e.g., I_{k-2} and I_{k-1}) and then matched to 2-D image features in a third view (e.g., I_k). In the monocular scheme, **matches over at least three views are necessary.**

Type of correspondences	Monocular	Stereo
2D-2D	X	X
3D-3D		X
3D-2D	X	X

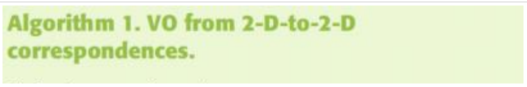
Notice that features can be both points and lines.

Comparisons
For a stereo vision setup **3D-to-2D methods perform better** than 3D-to-3D due to the triangulated 3-D points being much more uncertain in the depth direction. When 3-D-to-3-D feature correspondences are used in motion computation, their uncertainty may have a devastating effect on the motion estimate. In fact, in the 3-D-to-3-D case, the 3-D position error is minimized whereas in the 3-D-to-2-D case the image reprojection error.

A stereo camera scheme perform better than a monocular one because the 3D features are calculated directly in the absolute scale - given the fixed baseline - and because the matches only need to be computed between two views instead of three views as in the monocular scheme. Additionally, since the 3-D structure is computed directly from a single stereo pair rather than from adjacent frames as in the monocular case, the **stereo scheme exhibits less drift** than the monocular one in case of small motions.
However, Monocular methods are interesting because stereo VO degenerates into the monocular case when the distance to the scene is much larger than the stereo baseline (i.e., the distance between the two cameras). In this case, stereo vision becomes ineffective and monocular methods must be used.

2D to 2D correspondence

Estimating the Essential Matrix
The geometric relation between two images I_k and I_{k-1} of a calibrated camera are described by the essential matrix *E*. *E* contain the camera motion parameters up to an unknown scale factor for the translation in the following form:



Algorithm 1. VO from 2-D-to-2-D correspondences.

$$E_k \cong [t_k]_x R_k$$

, where $[t_k]_x$ is the cross product matrix of $t_k = [t_x, t_y, t_z]^T$. $\cong$ is used to endicate it is only equal up to an scale.
The essential matrix can be computed from 2D to 2D feature correspondence, and the **rotation and translation can directly be extracted** from E. Here we **exploit the epipolar constraint**:

$$\tilde{p}'^{\mathrm{T}} \mathrm{E} \tilde{\mathrm{p}} = 0$$

which determine where the line on which the corresponding feature point $\tilde{p}'$ of $\tilde{p}$ lies in the other image (e.g. I_{k-1}). $\tilde{p}$ and $\tilde{p}'$ are **normalizesed image coordinates** of the form $\tilde{p} = [\tilde{u}, \tilde{v}, 1]^T$**. Nisters five-point algorithm** is the standard for 2D-to-2D motion estimation in presence of outliers.
A simple and straightforward solution for $n \geq 8$ noncoplanar points is the **Longuet-Higgins'** eight-point algorithm, which I summarized here.

Longuet-Higgins eight point algorithm:
Each feature match gives a constrain in the following form

$$[\tilde{u}'\tilde{u} \quad \tilde{u}'\tilde{v} \quad \tilde{u}' \quad \tilde{u}\tilde{v}' \quad \tilde{v}\tilde{v}' \quad \tilde{v}' \quad \tilde{u} \quad \tilde{v} \quad 1] \cdot E = 0, \qquad E = [e_1 \quad \dots \quad e_9]^T$$

Stacking the constraints from the eight point algorithm fives the linear equation system $AE = 0$, and by solving the system, the parameters of E can be computed. The system is homogeneous and can be solved using SVD
The SVD of A: $A = USV^T$, and the least square estimate of E with $\|E\| = 1$ can be found as the last column of V. However, this solution **does not fulfill the inner constraints of an essential matrix**, which is that the singular components should be $E = USV^T, \quad diag(S) = \{s, s, 0\}$ - that is the first and second singular value are equal and the third is zero. We reproject the Essential matrix with

$$\boxed{\bar{E} = U \cdot diag\{1,1,0\} \cdot V^T}$$

Extracting R and t from $\bar{E}$
From the estimated $\bar{E}$, the rotation and translation parts can be extracted. There are in general four different solutions for R,t for one essential matrix

$$R = U(\pm W^T)V^T, \qquad \hat{t} = U(\pm W)SU^T$$

, where

$$W^T = \begin{bmatrix} 0 & \pm 1 & 0 \\ \mp 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

However, there is only one correct R, t pair, and that is the one resulting the point to be in front of both cameras.
We can check this with

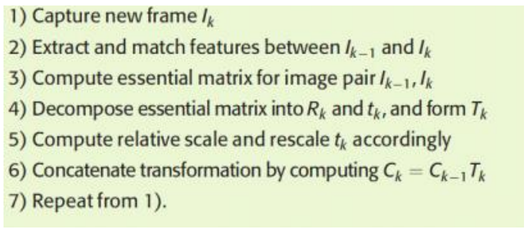
$$P = A[R\ t]$$
$$[0 \quad 0 \quad 1][R \quad t]Q > 0$$

Compute the relative Scale
To recover the trajectory of an image sequence, the different transformations $T_{0:n}$ have to be concatenated. To do this, the relative scale need to be computed, because the absolute cannot be determined from two images.
One way to do this is to triangulate 3D points X_{k-1} and X_k from two subsequent image pairs. From the corresponding 3D points, the relative distance between any combination of two 3D points can be computed. The proper scale can then be determine from the distance ratio r between a point pair in X_{k-1} and a poir in X_k

$$r = \frac{\|X_{k-1,i} - X_{k-1,j}\|}{\|X_{k,i} - X_{k,i}\|}$$

For robustness, the scale ratios for many point pairs are computed and the mean (or median in presence of outliers) is used. The translation vector t is then scaled with this distance ration.

NB: this requires features to be matched/tracked over multiple frames (at least 3,), instead of preforming explicit triangulation of the 3D points, the scale could then be recovered by exploiting the trifocal constraint


1) Capture new frame I_k
2) Extract and match features between I_{k-1} and I_k
3) Compute essential matrix for image pair I_{k-1}, I_k
4) Decompose essential matrix into R_k and t_k, and form T_k
5) Compute relative scale and rescale t_k accordingly
6) Concatenate transformation by computing $C_k = C_{k-1}T_k$
7) Repeat from 1).

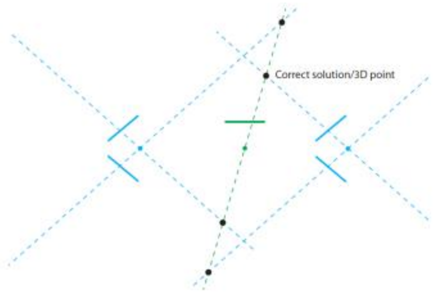


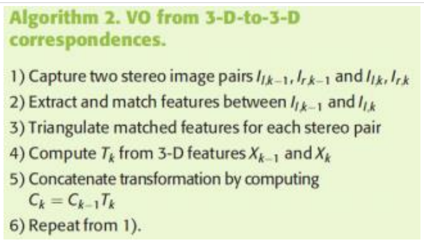
Figure 3.2: An abstract illustration of the four possible camera configurations for a given essential matrix $\mathbf{E}$, where the first camera $\mathbf{P}_1$ (in green) is kept fixed. The four possible solutions for camera two, $\mathbf{P}_2$, are depicted as blue camera centers and image plane. For a given point correspondence, denoted by the dashed lines (a green and a blue for each of the four possible camera configurations) it is possible to estimate a 3D point, denote by black dots, for each camera configuration. It is noted, that only one of these four 3D points is located *in front* of both cameras. This fact can be used to disambiguate (i.e. chose the right) the camera configuration. The four solutions are shown separately in Figure 3.3

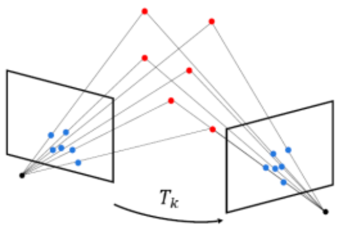
2D to 2D recap from lecture slide

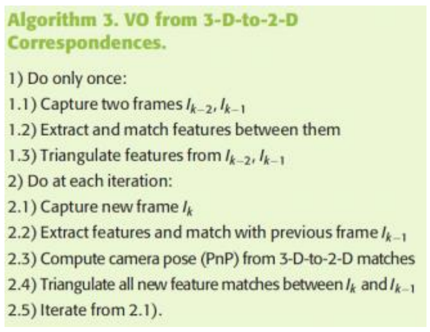
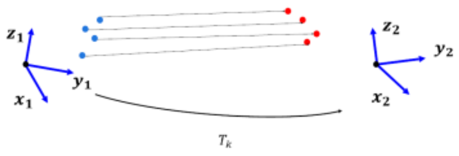
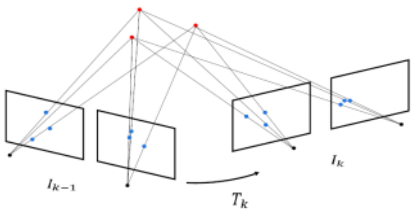
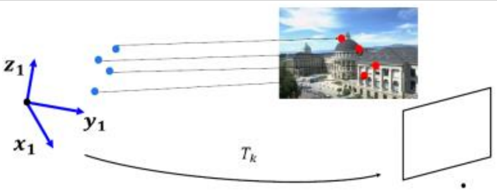
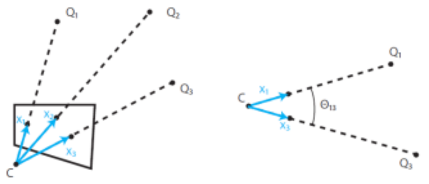
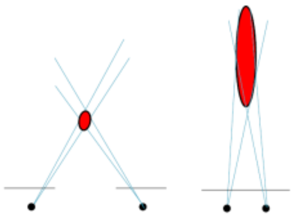
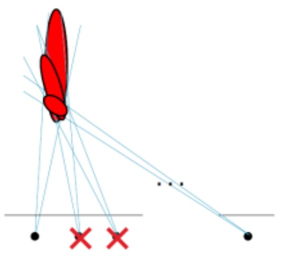
- Both f_{k-1} and f_k are spcified in 2D.
- The minimal-case solution involves 5-point correspondences
- The solution is found by determining the transformation that **minimizes the reprojection error** of the triangulated points in each image

$$T_k = \begin{bmatrix} R_{k,k-1} & t_{k,k-1} \\ 0 & 1 \end{bmatrix} = \underset{X^i, C_k}{\operatorname{argmin}} \sum_{i,k} \|p_k^i - \hat{p}_{k-1}^i\|^2$$

where $\hat{p}_{k-1}^i$ is the reprojection of the 3D point X_{k-1}^i into image I_k according to transfromation T_k.


Algorithm 2. VO from 3-D-to-3-D correspondences.
1) Capture two stereo image pairs $I_{l,k-1}, I_{r,k-1}$ and $I_{l,k}, I_{r,k}$
2) Extract and match features between $I_{l,k-1}$ and $I_{l,k}$
3) Triangulate matched features for each stereo pair
4) Compute T_k from 3-D features X_{k-1} and X_k
5) Concatenate transformation by computing $C_k = C_{k-1}T_k$
6) Repeat from 1).



	3D to 3D correspondence	- Both f_{k-1} and f_k are spcified in 3D. To do this, it is necessary to triangulate 3D points (e.g. use stereo camera) - The minimal-case involves 3 non-collinear correspondences - The solution is found by determining the aligning transformation that **minimizes the 3D-3D distance** $T_k = \begin{bmatrix} R_{k,k-1} & t_{k,k-1} \\ 0 & 1 \end{bmatrix} = \underset{T_k}{\operatorname{argmin}} \sum_i \lVert \tilde{X}_k^i - T_k \tilde{X}_{k-1}^i \rVert$ where the superscript denotes the ith feature, and $\tilde{X}_k$ and $\tilde{X}_{k-1}$are the homogeneous coordinates of the 3D points.	 **Algorithm 3. VO from 3-D-to-2-D Correspondences.** 1) Do only once: 1.1) Capture two frames I_{k-2}, I_{k-1} 1.2) Extract and match features between them 1.3) Triangulate features from I_{k-2}, I_{k-1} 2) Do at each iteration: 2.1) Capture new frame I_k 2.2) Extract features and match with previous frame I_{k-1} 2.3) Compute camera pose (PnP) from 3-D-to-2-D matches 2.4) Triangulate all new feature matches between I_k and I_{k-1} 2.5) Iterate from 2.1).  
	3D to 2D correspondence	- f_{k-1} is specified in 3D and f_k in 2D. - This problem is known as camera resection or PnP - The minimal-case solution involves 3-point correspondences - The solution is found by determining the transformation that **minimizes the reprojection error** $T_k = \begin{bmatrix} R_{k,k-1} & t_{k,k-1} \\ 0 & 1 \end{bmatrix} = \underset{X^i, C_k}{\operatorname{argmin}} \sum_{i,k} \lVert p_k^i - \hat{p}_{k-1}^i \rVert^2$ where $\hat{p}_{k-1}^i$ is the reprojection of the 3D point X_{k-1}^i into image I_k according to transfromation T_k. - In the monocular case, the 3D structure needs to be triangulated from two adjacent camera views (e.g., and) and then matched to 2D image features in a third view (e.g., I_k).	  Figure 3.4: The nature of the constraints for calibrated camera resectioning are that the 3d to 2d point correspondences define rays, x_i, and that the angle between any ray pair, Θ_{ij}, can thus be computed. Only three 3D to 2D point correspondences are need to calculate the essential matrix. AS shown above, this give 4 solutions, so a fourth is often used for disambiguation.
Triangulation and Key frames selection		Because a small baseline will give a big error (rule of thumb $\frac{Baseline}{distance} = \frac{1}{3}$)- see image to the right. To conquere this, we might triangulate between key frames, that is not sequential frames but some with a given distance. We do this by defining an uncertainty threshold that need obeyed before we accept a new fram.	 
Summary consideration		In the Stereo vision case, 3D-2D method exhibits less drift than 3D-3D method ➢ Stereo vision has the advantage over monocular vision that both motion and structure are computed in the absolute scale. It also exhibits less drift. ➢ When the distance to the scene is much larger than the stereo baseline, stereo VO degenerates into monocular VO ➢ Keyframes should be selected carefully to reduce drift ➢ Regardless of the chosen motion computation method, local bundle adjustment (over the last m frames) should be always performed to compute a more accurate estimate of the trajectory.	
Robust keypoint estimation. RANSAC		Matched point are usually contaminated by outliers, that is, wrong data associations. We can see on the image to the right that a small outlier can have a tremendous effect on the result. We use RANSAC to remove outliers (RANSAC has become the standard for model estimation in	

present of outliers). In this case we use the essential matrix to map between the point in at two time instances and use the motion model as the model.

We now want to figure out how many iterations of RANSAC we need

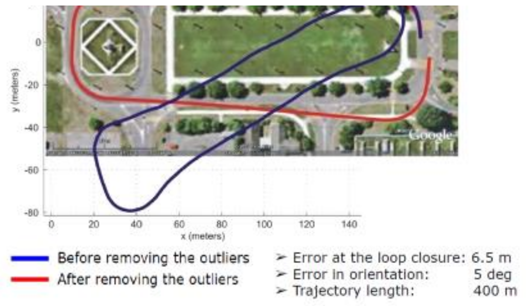
- The number of iterations N that is necessary to guarantee that a correct solution is found can be computed by (binominal distribution)

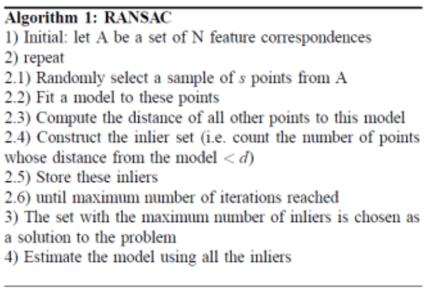
$$N = \frac{\log\left(1-p\right)}{\log(1-(1-\epsilon)^{S})}$$

- S is the number of points from which the model can be instantiated
- ε is the percentage of outliers in the data
- p is the requested probability of success

• RANSAC is an iterative method and is non deterministic in that it exhibits a different solution on different run.; however, the solution tends to be stable when the number of iterations grows
• For the sake of robustness, in many practical implementations N is usually multiplied by a factor of 10
• More advanced implementations fo RANSAC estimate the fraction of inliers adaptively, iteration after iteration

If we have a non-holonomic constrain on our model, we can use this to reduce the number of iterations




Algorithm 1: RANSAC

```
1) Initial: let A be a set of N feature correspondences
2) repeat
2.1) Randomly select a sample of $s$ points from A
2.2) Fit a model to these points
2.3) Compute the distance of all other points to this model
2.4) Construct the inlier set (i.e. count the number of points whose distance from the model $< d$)
2.5) Store these inliers
2.6) until maximum number of iterations reached
3) The set with the maximum number of inliers is chosen as a solution to the problem
4) Estimate the model using all the inliers
```

Visual SLAM 2

Thursday, 5 March 2020 09.09

SLAM pipeline

There are no universal solution for the SLAM algorithm. You therefore have to pick and choose from a different part of different SLAM algorithm.

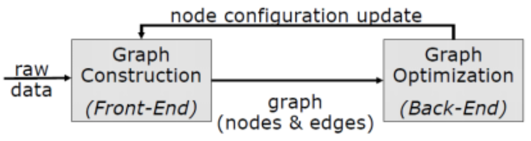
Graph based SLAM

GRAPH: Representation of a set of objects where pairs of objects are connected by links encoding relations between the objects.

We use graphs because it is a good at representing pose and how to get to that pose. When we drive around in an environment, we connect the successive nodes. We often revisit the same area, and can therefore generates constraints between non-successive poses.
The graph can handle constraints and connect the poses while the robot is moving and the graph-based SLAM algorithm constructs the graph out of the **raw sensor measurements**.

Basic idea:
- Use a graph to represent the problem
- Each node in the graph represents a robot position and a measurement acquired at that position
- An edge between two nodes represents a spatial constraint relating the two robot poses.
    - A constraint consists in a probability distribution over the relative transformations between the two poses. These transformations are either odometry measurements between sequential robot positions or are determined by aligning the observations acquired at the two robot locations.

Once the graph is constructed one seeks to find the configuration of the robot poses that best satisfies the constraints. Thus, in graph-based SLAM the problem is decoupled in two tasks:
- **Front-end** (graph construction)
    - Construct the graph from the raw measurements, determining the most likely configuration of the poses given the edges of the graph (graph optimization).
- **Back-end** (graph optimization/refinement)
    - Determining the most likely configuration of the poses given the edges of the graph.
    - Least squares optimization


node configuration update
raw data
Graph Construction (Front-End)
graph (nodes & edges)
Graph Optimization (Back-End)

General problem

Solving the SLAM problem consists of estimating the robot trajectory and the map of the environment as the robot moves in it. Due to the inherent noise in the sensor measurements, a SLAM problem is usually described by means of probabilistic tools. The robot is assumed to move in an unknown environment, along a trajectory described by the **sequence of random variables $\boldsymbol{x_{1:T}} = \{\boldsymbol{x_1}, \ldots, \boldsymbol{x_T}\}$**. While moving, it acquires a **sequence of odometry measurements $\boldsymbol{u_{1:T}} = \{\boldsymbol{u_1}, \ldots, \boldsymbol{u_T}\}$** and **perceptions of the environment $\boldsymbol{z_{1:T}} = \{\boldsymbol{z_1}, \ldots, \boldsymbol{z_T}\}$**. Solving the full SLAM problem consists of estimating the posterior probability of the robot's trajectory $x_{1:T}$ and the map m of the environment given all the measurements plus an initial position $x0$:

$$p(x_{1:T}, m \mid z_{1:T}, u_{1:T}, x_0)$$

The poses $x_{1:T}$ and the odometry $u_{1:T}$ are usually represented as 2D or 3D transformations in SE(2) or in SE(3), m is the model of the robot's trajectory and the map of the environment

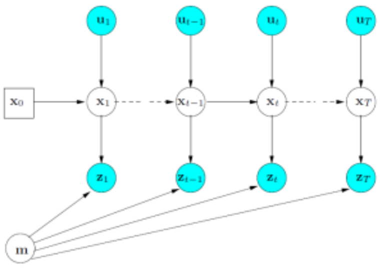


Fig. 4. Dynamic Bayesian Network of the SLAM process.

Problem

Let $x = (x_1, \ldots, x_T)T$ be a vector of parameters, where x_i describes the pose of node i. Let z_{ij} and Ω_{ij} be respectively *the mean* and *the information matrix* of a virtual measurement between the node i and the node j. This virtual measurement is a transformation that makes the observations acquired from i maximally overlap with the observation acquired from j. Let $\hat{z}_{ij}(x_i, x_j)$ be *the prediction* of a virtual measurement given a configuration of the nodes x_i and x_j. Usually this prediction is the relative transformation between the two nodes. The log-likelihood l_{ij} of a measurement z_{ij} is therefore

$$l_{ij} \propto \underbrace{\left[z_{ij} - \hat{z}_{ij}(x_i, x_j)\right]^T \cdot \Omega_{ij} \cdot \left[z_{ij} - \hat{z}_{ij}(x_i, x_j)\right]}_{\substack{negative\ \log\ likeligood \\ F_{ij}}}$$

Let $e(x_i, x_j, z_{ij})$ be a function that computes a difference between the expected observation $\hat{z}_{ij}$ and the real observation z_{ij} gathered by the robot. For simplicity of notation, we will encode the indices of the measurement in the indices of the error function

$$\boxed{e_{ij}(x_i, x_j) = z_{ij} - \hat{z}_{ij}(x_i, x_j)}$$

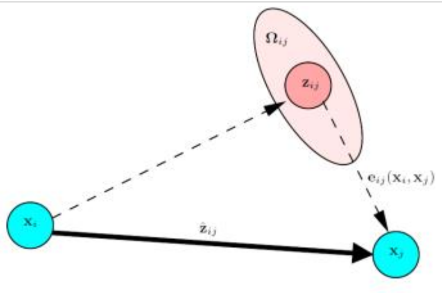


Fig. 7. Aspects of an edge connecting the vertex $\mathbf{x}_i$ and the vertex $\mathbf{x}_j$. This edge originates from the measurement $\mathbf{z}_{ij}$. From the relative position of the two nodes, it is possible to compute the expected measurement $\hat{\mathbf{z}}_{ij}$ that represents $\mathbf{x}_j$ seen in the frame of $\mathbf{x}_i$. The error $\mathbf{e}_{ij}(\mathbf{x}_i, \mathbf{x}_j)$ depends on the displacement between the expected and the real measurement. An edge is fully characterized by its error function $\mathbf{e}_{ij}(\mathbf{x}_i, \mathbf{x}_j)$ and by the information matrix $\mathbf{\Omega}_{ij}$ of the measurement that accounts for its uncertainty.

Error function and minimization

The error function is defined above, but it contains multiple variable. We therefore redefine the two state, x_i and x_j to a single varaible x, so we can define an optimization problem

We can describe the problem as

$$\underbrace{x}_{\substack{States \\ (unkonwn)}} \rightarrow \begin{matrix} f_1(x) = \hat{z}_1 \\ f_2(x) = \hat{z}_2 \\ \vdots \\ \underbrace{f_n(x) = \hat{z}_n}_{\substack{predicted \\ measurements}} \end{matrix} \qquad \begin{matrix} z_1 \\ z_2 \\ \vdots \\ \underbrace{z_n}_{\substack{real \\ measurements}} \end{matrix}$$

- x: positions of 3D feature
- z_i: coordinates of the 3D features projected on camera images
- Estimate the most likely 3D position of the features based on the image projections (given the camera poses)

The error as the different between the predicted and the actual measurement

$$e_i(x) = z_i - f_i(x) = z_i - \hat{z}_i$$

- We assume that the measurement error has zero mean and is normally distributed
- Gaussian error with information matrix Ω_i
- The squared error of a measurement depends only on the state and is a scalar

$$e_i(x) = \boldsymbol{e}_i(x)^T \Omega_i \boldsymbol{e}_i(x)$$

We can now state the minimization problem

- Find the state x^* which minimizes the the negative log likelihood given all measurements
$$x^* = \underset{x}{\operatorname{argmin}} F(x), \qquad global\ error\ (scalar)$$
$$= \underset{x}{\operatorname{argmin}} \sum_i e_i(x), \qquad squared\ error\ term\ (Scalar)$$
$$= \underset{x}{\operatorname{argmin}} \sum_i \boldsymbol{e}_i^T(x) \Omega_i \boldsymbol{e}_i(x)$$

- A general solution is to derive the global error function and find its zeros
- In general complex and no closed from solution can be found. We therefore uses numerical approaches

We have to linearize the error function

- We approximate the error functions around an initial guess x via Taylor expansion
$$e_i(x + \Delta x) \cong \underbrace{e_i(x)}_{e_i} + J_i(x)\Delta x$$
- If we fix x and carry out the Taylor expansion in the increments Δx
- We replace the Taylor expansion in the squared error terms:
$$e_i(x+\Delta x) = e_i^T(x+\Delta x)\Omega_i e_i(x+\Delta x)$$
$$\cong (\boldsymbol{e}_i + J_i\Delta x)^T \Omega_i (\boldsymbol{e}_i + +J_i \Delta x)$$
$$= \boldsymbol{e}_i^T \Omega_i \boldsymbol{e}_i + \boldsymbol{e}_i^T \Omega_i J_i \Delta x + \Delta x^T J_i^T \Omega_i \boldsymbol{e}_i + \Delta x^t J_i \Omega_i J_i \Delta x$$

NB: Ω is the inverse of the covariance matrix for the given measurement

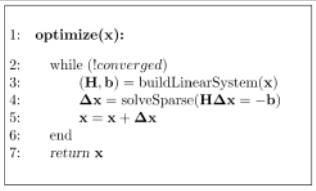
- By grouping similar terms, we obtain:
$$e_i(x+\Delta x) \cong \underbrace{\boldsymbol{e}_i^T \Omega_i \boldsymbol{e}_i}_{c_i} + 2 \cdot \underbrace{\boldsymbol{e}_i^T \Omega_i J_i}_{b_i^T} \Delta x + \Delta x^T \underbrace{J_i \Omega_i J_i}_{H_i} \Delta x$$
$$= c_i + 2b_i^T \Delta x + \Delta x^T H_i \Delta x$$
- The global error is the sum of the squared errors terms corresponding to the individual measurements
$$F(x+\Delta x) \cong \sum_i \left(c_i + 2b_i^T\Delta x + \Delta x^T H_i \Delta x\right)$$
$$= \underbrace{\sum_i c_i}_{c} + 2\underbrace{\left(\sum_i b_i^T\right)}_{b^T} \Delta x + \Delta x^T \underbrace{\left(\sum_i H_i\right)}_{H} \Delta \mathrm{x}$$
$$= \boxed{c + 2b^T\Delta x + \Delta x^T H \Delta x}, \qquad b^T = \sum_i \boldsymbol{e}_i^T \Omega_i J_i, \qquad H = \sum_i J_i^T \Omega_i J_i$$

We can now write the global error terms as a quadratic from in Δx
$$\boxed{F(x+\Delta x) \cong c + 2b^T \Delta x + \Delta x^T H \Delta x}$$
Our goal is to minimize this function. The derivative is given by
$$\frac{\partial F(x+\Delta x)}{\partial \Delta x} \cong 2b + 2H\Delta x$$
Setting the derivative to zero leads to the linear system
$$H\Delta x = -b$$
The solution for the increment Δx^* is
$$\Delta x^* = -H^{-1} b$$

Overall procedure

Iterate the following steps:

- Linearize around x and compute for each measurement
$$e_i(x+\Delta x) \cong e_i(x) + J_i \Delta x$$
- Compute the terms for the linear sustem
- Solve $b^T = \sum_i \boldsymbol{e}_i^T \Omega_i J_i$, $H = \sum_i J_i^T \Omega_i J_i$
- Updating state
$$\Delta x^* = -H^{-1} b$$
$$x \leftarrow x + \Delta x^*$$



```
1: optimize(x):
2:     while (!converged)
3:         (H, b) = buildLinearSystem(x)
4:         Δx = solveSparse(HΔx = −b)
5:         x = x + Δx
6:     end
7:     return x
```

Trivial 1D Example

Two nodes are on observation
$$x = (x_1\ x_2)^T = (0\ 0)\text{^}T$$
$$z_{12} = 1$$
$$\Omega = 2$$
$$e_{12} = z_{12} - (x_2 - x_1) = 1 - (0-0) = 1$$
$$J_{12} = (1\ -1)$$
$$b_{12}^T = e_{12}^T \Omega_{12} J_{12} = (2\ -2)$$
$$H_{12} = J_{12}^T \Omega J_{12} = \begin{pmatrix} 2 & -2 \\ -2 & 2 \end{pmatrix}$$
$$\Delta x = -H_{12}^{-1} b_{12}$$

OBS: This cannot be solved because $\det(H) = 0$

We can therefore not uniquely determine the solution.

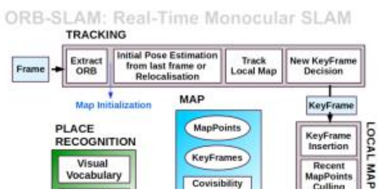
The issues is that the constraint specifies a relative constraint between both nodes. Therefore any solution would fit that satisfy the constraint. **We therefore need to fix one of the notes**. This is normally set to the origin

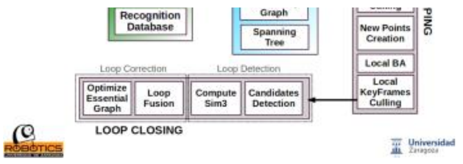
$$H = \begin{pmatrix} 2 & -2 \\ -2 & 2 \end{pmatrix} + \underbrace{\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}}_{\substack{constraint \\ that\ sets \\ dx_1 = 0}}$$
$$\Delta x = -H^{-1} b_{12} = (0\ 1)^T$$

ORB-SLAM

ORB stands for "Oriented FAST and Rotated BRIEF)

- We start from receiving a frame. We calculate the initial pose estimation based on the previous pose.
- The KeyFrame, is as define in the previous notebook.
- Local BA: Bundle adjustment
- The blue MAP in the middle is the global map.
- The Covisibility encode from which position it can see which specific nodes.
- We need some way of recognizing loops. That is the Sim3, that is a look-alike



		measurement. Limitations - Monocular: the absolute scale is unknown ○ Include in the map a known- size object - Requires a reasonably lit area - Needs texture: will fail with large plain walls - Map is too sparse for interaction with the environment · Extensions - Improve agility using IMU - Stereo camera: real scale and robustness to quick motions - RGB- D camera: track with features + dense map - Semi- dense or dense mapping	 Recognition Database Graph Spanning Tree New Points Creation PING Local BA Local KeyFrames Culling Loop Correction Loop Detection Optimize Essential Graph Loop Fusion Compute Sim3 Candidates Detection LOOP CLOSING ROBOTICS Universidad Zaragoza

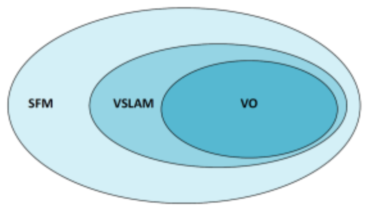
Structure From Motion

Friday, 6 March 2020 11.06

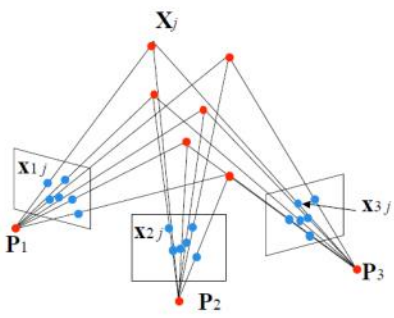
Introduction

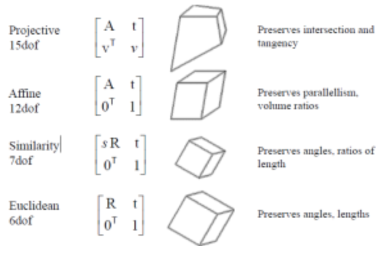
In structure from motion we do not have calibrated cameras. The issues is therefore a bit broader than the VSLAM. Because of the added complexity of the calibration, we cannot handle distortion
Our problem is now:
$\kappa = 1, \dots, M$ is the numbers of cameras and $\alpha = 1, \dots, N$ is the number of points.

- Given: $\kappa = 1, \dots, M$ images of $\alpha = 1, \dots, N$ fixed 3D points
  $x_{\kappa\alpha} = P_\kappa X_\alpha, \quad \kappa = 1, \dots, M \;\; \alpha = 1, \dots, N$
- Problem: estimate M projection matrices P_κ and N 3D points X_α from the MN correspondenecs $x_{\kappa\alpha}$
  - That is, we have a bunch of correspondence in image coordinates, $x_{\kappa\alpha}$, and we then want to estimate both the camera matrices and the 3D points. We do this by calibrating the camera, and then solve for the Camera matrix and the 3D points.



Ambiguity
- If w scale the entire scene by some factor k and, at the same time, scale the camera matrices by the factor of $1/k$, the projections of the scene points in the image remain exactly the same:
  $$x = PX = \left(\frac{1}{k}P\right)(kX)$$
- It is **impossible to recover the absolute scale of the scene**
- More generally: if we transform the scene using a transformation Q and apply the inverse transformation to the camera matrices, then the images do not change
  $$x = PX = \left(PQ^{-1}\right)(QX)$$
- We try to reduce the problem to an affine ambiguity





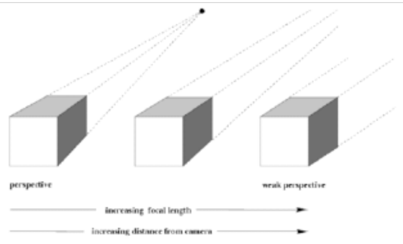
Projective 15dof	$\begin{bmatrix} A & t \\ v^T & v \end{bmatrix}$	Preserves intersection and tangency
Affine 12dof	$\begin{bmatrix} A & t \\ 0^T & 1 \end{bmatrix}$	Preserves parallellism, volume ratios
Similarity 7dof	$\begin{bmatrix} sR & t \\ 0^T & 1 \end{bmatrix}$	Preserves angles, ratios of length
Euclidean 6dof	$\begin{bmatrix} R & t \\ 0^T & 1 \end{bmatrix}$	Preserves angles, lengths

Affine cameras

Affine cameras does not use the pinhole model, but use the weak perspective model. This assumption only holds when the image we are looking at is only in the plane, or we look at a small neighborhood where this assumption hold.

A general affine camera combines the effect of an affine transformation of the 3D space, orthographic projection, and an affine transformation of the image:

$$P = [3 \times 3\ affine] \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} [4 \times 4\ affine] = \begin{bmatrix} a_{11} & a_{12} & a_{13} & b_1 \\ a_{21} & a_{22} & a_{23} & b_2 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} A & b \\ 0 & 1 \end{bmatrix}$$



$i = \kappa = 1, \dots, M$ is the numbers of cameras and
$j = \alpha = 1, \dots, N$ is the number of points.

Affine projection is a linear mapping + translation in inhomogeneous coordinate
- Given: M image of N fixed 3D points
  $x_{\kappa\alpha} = A_\kappa X_\alpha + b_\kappa, \quad \kappa = 1, \dots, M \;\; \alpha = 1, \dots, N$
- Problem : use the MN correspondences $x_{\kappa\alpha}$ to estimate M projection matrices A_κ and translation vectors b_κ, and N points X_α
- The reconstruction is defined up to an arbitrary affine transformation Q (12 DoF)
  $$\begin{bmatrix} A & b \\ 0 & 1 \end{bmatrix} \to \begin{bmatrix} A & b \\ 0 & 1 \end{bmatrix} Q^{-1}, \qquad \begin{pmatrix} X \\ 1 \end{pmatrix} \to Q \begin{pmatrix} X \\ 1 \end{pmatrix}$$
- We have $2MN$ knowns and $8M + 3N$ unknowns (minus 12 DoF for affine ambiguity)
- Thus, we must have $2MN \geq 8M + 3N - 12$
- For two views, we need four point correspondences

For simplicity we want to center the points in the world system coordinates
- Centering: subtract the centroid of the image points
  $$\hat{x}_{\kappa\alpha} = x_{\kappa\alpha} - \frac{1}{N}\sum_{j=1}^{N} x_{\kappa j} = A_\kappa X_\alpha + b_\kappa - \frac{1}{N}\sum_{j=1}^{N}\left(A_\kappa X_j + b_\kappa\right) = A_\kappa\left(X_\alpha - \frac{1}{N}\sum_{j=1}^{N} X_j\right) = A_i \hat{X}_\alpha$$
- For simplicity, assume that the origin of the world coordinates system is at the centroid of the 3D points
- After centering, each normalized point $x_{\kappa\alpha}$ is related to the 3D point X_κ by
  $$\hat{x}_{\kappa\alpha} = \underbrace{A_\kappa}_{\substack{(=\Pi_\kappa) \\ Book\ notation}} X_\alpha$$
  where Π_k is the **camera matrix of the affine camera.**

Factorization and Affine Reconstruction

We arrange all the camera matrices Π_κ and all the 3D positions $(X_\alpha, Y_\alpha, Z_\alpha)$ in the matrix form:

$$\boldsymbol{M} = \begin{pmatrix} \Pi_1 \\ \vdots \\ \Pi_M \end{pmatrix}, \qquad \boldsymbol{S} = \begin{pmatrix} X_1 \dots X_N \\ Y_1 \dots Y_N \\ Z_1 \dots Z_N \end{pmatrix}$$

The $2M \times 3$ matrix M is called the **motion matrix**, and the $3 \times N$ matrix S the **shape matrix**. On the other hand, we arrange all the observed coordinates $(x_{\alpha\kappa}, y_{\alpha\kappa})$ in the matrix form:

$$\boldsymbol{W} = \begin{pmatrix} x_{11} & x_{21} & \cdots & x_{N1} \\ y_{11} & y_{21} & \cdots & y_{N1} \\ x_{12} & x_{22} & \cdots & x_{N2} \\ y_{12} & y_{22} & \cdots & y_{N2} \\ \vdots & \vdots & \ddots & \vdots \\ x_{1M} & x_{2M} & \cdots & x_{NM} \\ y_{1M} & y_{2M} & \cdots & y_{NM} \end{pmatrix}$$

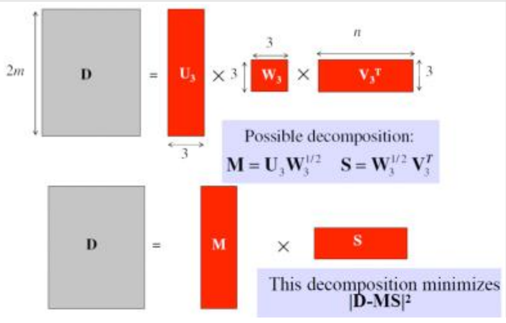
This $2M \times N$ matrix is called the **observation matrix**. We can write the following equality
$$\boldsymbol{W} = \boldsymbol{MS}$$
However, the actual images we observe are taken by perspective cameras, therefore the matrix $\boldsymbol{W}$ cannot be factored into matrices $\boldsymbol{S}$ and $\boldsymbol{M}$ in this form. Below, we introduce the following approximation.

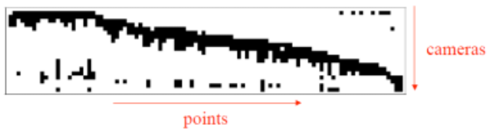
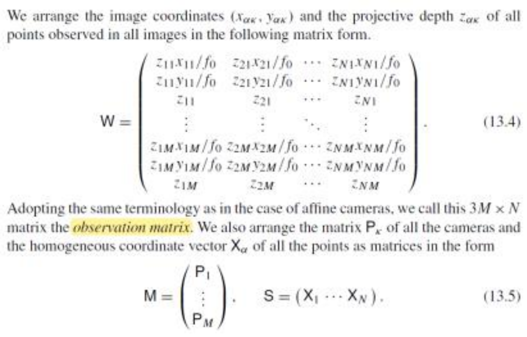
Affine construction

The algorithm for a affine structure from motion

- Given: $\kappa = 1, \dots, M$ images of $\alpha = 1, \dots, N$ fixed 3D points
  $x_{\kappa\alpha} = P_\kappa X_\alpha, \quad \kappa = 1, \dots, M \;\; \alpha = 1, \dots, N$

Given: M images and N features $x_{\kappa\alpha}$
- For each image κ, center the feature coordinates
- Construct a $2M \times N$ measurement matrix D:
  - Column α contains the projection of point α in all views
  - Row κ contains one coordinate of the projections of all the N points in image κ
- Factorize $\boldsymbol{W}$:
  - Compute $SVD: \boldsymbol{W} = UWV^T$
  - Create U_3 by taking the first 3 columns of U
  - Create V_3 by taking the first 3 columns of V
  - Create W_3 by taking the upper left 3×3 block of W
- Create the motion and shape matrices:

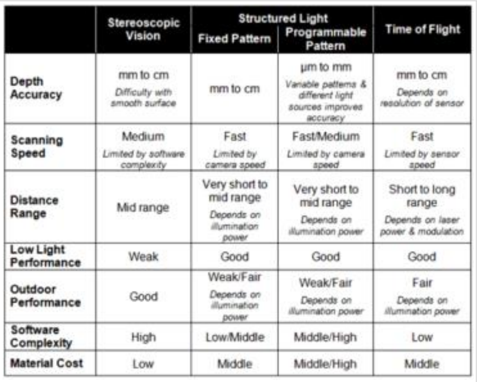

2m
D
=
U3
× 3
W3
×
V3T
3
n
3
3
Possible decomposition:
$\mathbf{M} = \mathbf{U}_3\mathbf{W}_3^{1/2} \quad \mathbf{S} = \mathbf{W}_3^{1/2}\mathbf{V}_3^T$
D
=
M
×
S
This decomposition minimizes |D-MS|²

		○ $\boldsymbol{M} = U_3 W_3^{½}$ and $\boldsymbol{S} = W_3^{½} V_3^T$ ○ **(or $\boldsymbol{M} = \boldsymbol{U_3}$ *and* $\boldsymbol{S} = \boldsymbol{W_3} \boldsymbol{V_3^T}$)** - Eliminate affine ambiguity The decomposition is not unique. We get the same D by using any 3×3 matrix C and applying the transformations $M \rightarrow MC$, $S \rightarrow C^{-1}S$ That is because we have only an affine transformation and we have not enforced any Euclidean constraints (like forcing the image axes to be perpendicular, for example) There exist two approaches for resolving this indeterminacy of affine transformations. - **Use of knowledge about the scene** - If we know the lengths and angles (e.g., being orthogonal) of some parts, we multiply the shape matrix S by some matrix A^{-1} from the left so that the lengths and angles become correct. - **Use of knowledge about the cameras** - If we know some properties about the cameras, we multiply the motion matrix M by some matrix A from the right such that the expected properties hold.	
	Issues	So far we have assumed that all points are visible in all views. In reality, the measurement matrix typically looks some like this: 	
Perspective camera		We arrange the image coordinates $(x_{\alpha\kappa}, y_{\alpha\kappa})$ and the projective depth $z_{\alpha\kappa}$ of all points observed in all images in the following matrix form. $W = \begin{pmatrix} z_{11}x_{11}/f_0 & z_{21}x_{21}/f_0 & \cdots & z_{N1}x_{N1}/f_0 \\ z_{11}y_{11}/f_0 & z_{21}y_{21}/f_0 & \cdots & z_{N1}y_{N1}/f_0 \\ z_{11} & z_{21} & \cdots & z_{N1} \\ \vdots & \vdots & \ddots & \vdots \\ z_{1M}x_{1M}/f_0 & z_{2M}x_{2M}/f_0 & \cdots & z_{NM}x_{NM}/f_0 \\ z_{1M}y_{1M}/f_0 & z_{2M}y_{2M}/f_0 & \cdots & z_{NM}y_{NM}/f_0 \\ z_{1M} & z_{2M} & \cdots & z_{NM} \end{pmatrix}. \quad (13.4)$ Adopting the same terminology as in the case of affine cameras, we call this $3M \times N$ matrix the *observation matrix*. We also arrange the matrix P_κ of all the cameras and the homogeneous coordinate vector X_α of all the points as matrices in the form $M = \begin{pmatrix} P_1 \\ \vdots \\ P_M \end{pmatrix}, \quad S = (X_1 \cdots X_N). \quad (13.5)$	

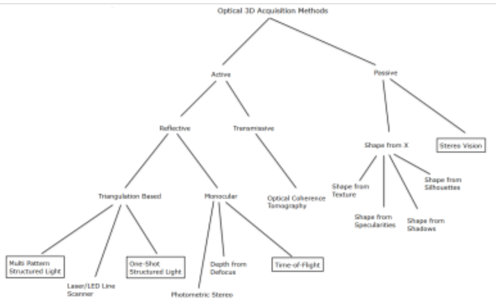
3D sensors

Thursday, 12 March 2020 15.27

Overview



	Stereoscopic Vision	Structured Light		Time of Flight
		Fixed Pattern	Programmable Pattern	
Depth Accuracy	mm to cm *Difficulty with smooth surface*	mm to cm	µm to mm *Variable patterns & different light sources improves accuracy*	mm to cm *Depends on resolution of sensor*
Scanning Speed	Medium *Limited by software complexity*	Fast *Limited by camera speed*	Fast/Medium *Limited by camera speed*	Fast *Limited by sensor speed*
Distance Range	Mid range	Very short to mid range *Depends on illumination power*	Very short to mid range *Depends on illumination power*	Short to long range *Depends on laser power & modulation*
Low Light Performance	Weak	Good	Good	Good
Outdoor Performance	Good	Weak/Fair *Depends on illumination power*	Weak/Fair *Depends on illumination power*	Fair *Depends on illumination power*
Software Complexity	High	Low/Middle	Middle/High	Low
Material Cost	Low	Middle	Middle/High	Middle



Geometric principle

Image rectification

We use Image rectification to align the epipolar lines, such that we can speed up that search speed (see chapter 9)

Given camera matrices

$$P_1 = A_1[R_1\ t_1], \qquad P_2 = A_2[R_2\ \ t_2]$$

Compute the camera centers

$$Q_{c1} = R_1^T t_1, \qquad Q_{c2} = R_2^T t_2$$

Find the basis A, B, C for the plane to project the image onto

$$Q = aA + bB + C = [A \quad B \quad C]\begin{bmatrix} a \\ b \\ 1 \end{bmatrix}$$

$$A = \frac{Q_{c2} - Q_{c1}}{\left\| Q_{c2} - Q_{c1} \right\|_2}$$

Furthermore B is found via crossing the average of the image plane normal with A

$$B' = \frac{1}{2}\left(R_1^T \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} + R_2^T \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \right) \times A,$$

$$B = \frac{B'}{\|B'\|}$$

And C can be found by point triangulation. Then the initial rectifying homotropies are given by

$$H_1' = \left(P_1 \begin{bmatrix} A & B & C \\ 0 & 0 & 1 \end{bmatrix} \right)^{-1}, \qquad H_2' = \left(P_2 \begin{bmatrix} A & B & C \\ 0 & 0 & 1 \end{bmatrix} \right)^{-1}$$

The scale s should t hen be determined and can be applied with

Type equation here.

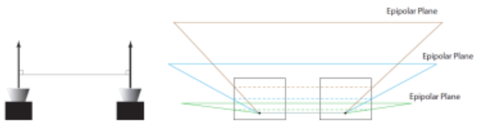
$$H_1 = \begin{bmatrix} s & 0 & 0 \\ 0 & s & 0 \\ 0 & 0 & 1 \end{bmatrix} H_1', \qquad H_2 = \begin{bmatrix} s & 0 & 0 \\ 0 & s & 0 \\ 0 & 0 & 1 \end{bmatrix} H_2'$$

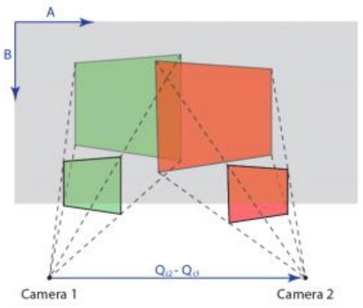
$$m = \det\left(\begin{bmatrix} \frac{\partial a}{\partial x} & \frac{\partial a}{\partial y} \\ \frac{\partial b}{\partial x} & \frac{\partial b}{\partial y} \end{bmatrix} \right)$$

$$s = \frac{\alpha}{\min(|m_1|, |m_2|)}, \qquad (gfood\ heuristic)$$

Given the final rectifying homographies. Then the associated camera matrices are given by

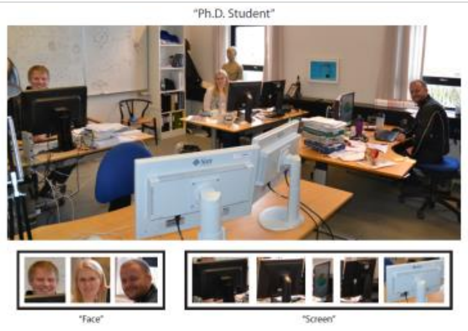
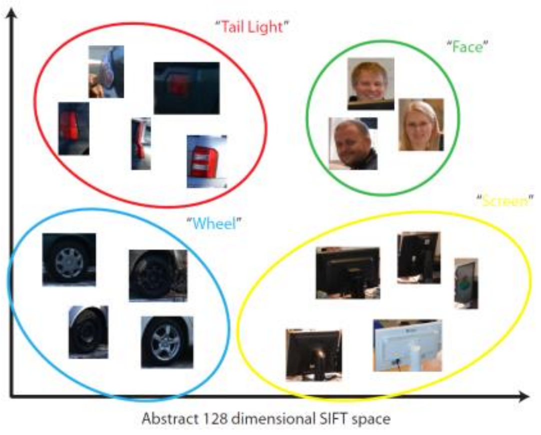
$$\tilde{P}_1 = H_1 P_1, \quad \tilde{P}_2 = H_2 P_2$$





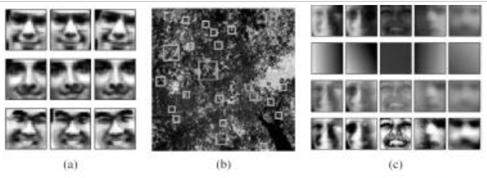
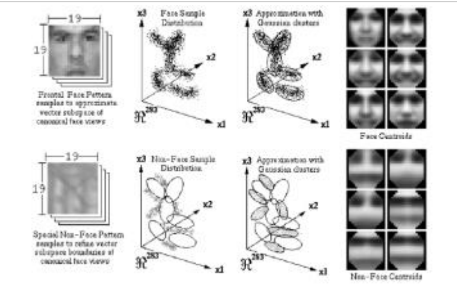
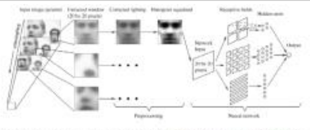
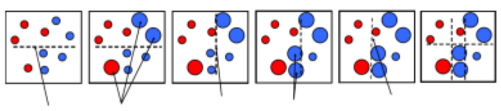
Bag of Words - Image Description and Matching

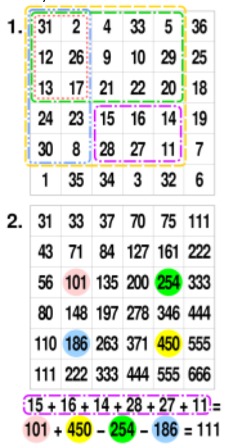
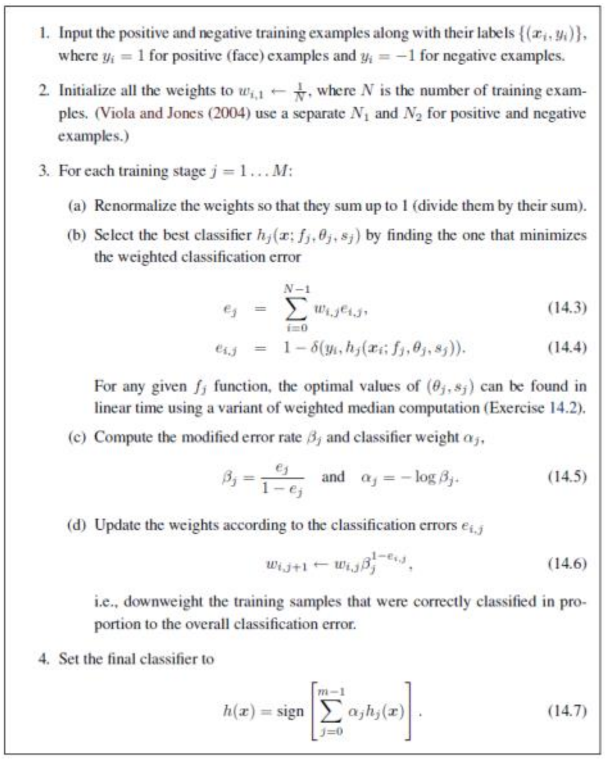
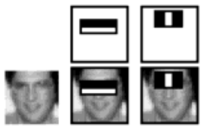
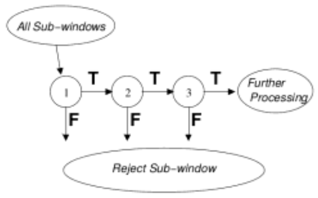
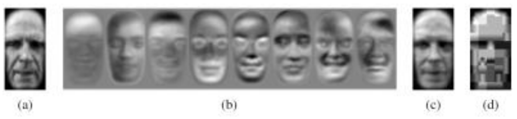
Wednesday, 18 March 2020 09.15

Object recognition and segmentation		Object recognition and segmentation can roughly be divided into 3 categories: - Image Search ○ e.g. Search through a database and find all images with faces - Localization ○ e.g. Locate where in the image the face is - Segmentation ○ e.g. classify which pixel are part of the face	 Figure III.1: An illustration of the loos taxonomy of object recognition and segmentation. **a)** Image search, i.e. from a collection if images find the one containing a given object. **b)** Localization, i.e. set a bounding box around instances of the given object. **c)** Segmentation, i.e. find the pixels that are part of the given object. The face image and it's segmentation is supplied by Jens Fagertun.
Bag of Words		The Bag of Word is an image descriptor. That is, a descriptor that is used to describe the whole image, as oppose to an feature descriptor, which is use to describe specific features in an image. The bag of word descriptor can be used to search through an image database and find the images that best resemble "closets to" the input search image. It is therefore important to formulate the term "closets to" The bag of word descriptor is inspired of the algorithm used to search though text and classify documents. The algorithm works by creating a histogram by counting how many time each individual word occurs, that is how many times "jump", "hello", "the", etc.. Occurs. However, some words are more discriminating than others. For example is the word "jump" less frequently use than "hello", and will therefore describe the test better. The histogram is therefore weighted, so the rare words has a higher weight than the common words. This is translated into images with the concept of *visual words*, where an image part is labeled with a word such as the one seen on the right.	 Figure 7.9: The visual words of "Face" and (computer) "screen" are good cues that an image is of a Ph.D. student.
	Estimation of visual words	The visual words can be estimated from a training set of images, this could be the database itself. 1. Extract sift features and descriptors from all images, and store them as a combined data set. This are interpreted as a lot of data points in the 128 dimensional space of the SIFT descriptor. 2. Perform K-means clustering on this data set of SIFT descriptors from item one. Here the number of clusters k is chosen to being equal to the number of visual words (defined by user). 3. Associate each cluster with a visual word. However, an issue can be that boats are often in lakes. Which mean that a lake without a boat might be scored as a boat. We can then **query from** the database by describing the by the same visual words used to form the database. 1. Extract SIFT features and SIFT descriptors. 2. For each SIFT descriptor, determine which cluster and thus visual word it belongs to. 3. Compute the histogram of visual words for the given image. The histogram is then the image descriptor, and the distance between two descriptors can be calculated $$dist(a,b) = 2\sum_{i=!}\frac{w_i\left(a_i-b_i\right)^2}{a_i+b_i}$$ Where a and b are two descriptors and the w_i are the importance weight, given by the inverse frequency of the visual words occurrence. In practice however, the 2-norm is used instead, althrough still weightet by the w_i. w_i could be calclated as $$w_i = \ln\left(\frac{N}{N_i}\right)$$ Where N is the number of images in the database and N_i is the number of images with at least one ocurrence of the visual word *I*,	 "Tail Light" "Face" "Screen" "Wheel" Abstract 128 dimensional SIFT space
	Hierarchy clustering	A major computational burden by doing query is that an image need to be compared to all images, so we do a pyramid-hierarchy where we start by a very course sorted images and slow increase the complexity.	

Face Recognition

Tuesday, 9 June 2020 16.08

Face recognition		If we have an image as shown on the right, we want to find the faces. One approach could be to apply a recognition algorithm to every possible sub-window in this image. Such algorithms are likely to be both slow and error-prone. Instead, it is more effective to construct special-purpose detectors, whose job is to rapidly find likely regions where particular objects might occur. Face detection is a successful example of recognition. The recognition is then divided into two steps, a face detection phase, and a face classification phase. - The face detection phase perform two-way classification - face or no face - in order to identify if a given patch has a face on it. - The face classification phase performs multiclass classification, because it need to identify the person	 **Figure 14.2** Face detection results produced by Rowley, Baluja, and Kanade (1998a) © 1998 IEEE. Can you find the one false positive (a box around a non-face) among the 57 true positive results?
Face detection		Before face recognition can be applied to a general image, the location and sizes of any faces must be found. According to Yang, face detection techniques can be classified as feature based, template-based or appearance-based. **Feature based** techniques attempt to find the locations of distinctive image features such as the eyes, nose and mouth, and then verify whether these features are in a plausible geometrical arrangement. **Template based** approaches, such as active appearance models (AAMs), can deal with a wide range of pose and expression variability. Typically, they require good initialization near a real face and are therefore not suitable fast face detectors. **Appearance based** approaches scan over small overlapping rectangular patches of the image searching for likely face candidates, which can then be refined using cascade of more expensive but selective detection algorithms. In order to deal with scale variation, the image is usually **converted into a sub-octave pyramid** and a separate scan is performed on each level. Most appearance-based approached rely heavily on training classifiers using sets of labeled face and non-face patches.	
	Basic Concept for Face detection. (Preprocessing)	Sung and Poggio (1998) and Rowley, Baluja, and Kanade (1998a) present two of the earliest appearance-based face detectors and introduce a number of innovations that are widely used in later work by others. To start with, both systems **collect a set of labeled face patches** (Figure 14.2) as well as a set of patches taken from **images that are known not to contain faces**, such as aerial images or vegetation (Figure 14.3b). The collected face images are augmented by **artificially mirroring, rotating, scaling, and translating** the images by small amounts to make the face detectors less sensitive to such effects (Figure 14.3a). After an initial set of training images has been collected, some optional preprocessing can be performed, such as subtracting an average gradient (linear function) from the image to compensate for global shading effects and using histogram equalization to compensate for varying camera contrast (Figure 14.3c). When the preprocessing of the face detection has been done, the final detection can be carried out by various algorithms. The methods include **Clustering and PCA**, **Neural Networks**, **Supported Vector Machines** and **Boosting**.	 **Figure 14.3** Pre-processing stages for face detector training (Rowley, Baluja, and Kanade 1998a) © 1998 IEEE: (a) artificially mirroring, rotating, scaling, and translating training images for greater variability; (b) using images without faces (looking up at a tree) to generate non-face examples; (c) pre-processing the patches by subtracting a best fit linear function (constant gradient) and histogram equalizing.
	Clustering and PCA (fundamental early work)	Once the face and non-face patterns have been pre-processed, Sung and Poggio (1998) cluster each of these datasets into six separate clusters using k-means and then fit PCA subspaces to each of the resulting 12 clusters (Figure 14.4). At detection time, the DIFS and DFFS metrics are used to produce 24 Mahalanobis distance measurements (two per cluster). The resulting 24 measurements are input to a multi-layer perceptron (MLP).	 19 19 Frontal Face Pattern samples to approximate vector subspace of canonical face views x3 Face Sample Distribution x2 x1 x3 Approximation with Gaussian clusters Face Centroids Special Non-Face Pattern samples to refine vector subspace boundaries of canonical face views x3 Non-Face Sample Distribution x3 Approximation with Gaussian clusters Non-Face Centroids **Figure 14.4** Learning a mixture of Gaussians model for face detection (Sung and Poggio 1998) © 1998 IEEE. The face and non-face images (19^2-long vectors) are first clustered into six separate clusters (each) using k-means and then analyzed using PCA. The cluster centers are shown in the right-hand columns.
	Boosting Viola and Jones.	**This method is the best for detection** Boosting is a widely used technique and involves training a series of increasingly discriminating simple classifiers and then blending their outputs. In more details, boosting involves constructing a *classifier* $h(x)$ as a sum of simple *weak learners*, $\underset{Classifier}{h}\left(\underset{\substack{Input\\RoI}}{x}\right) = sign\left[\sum_{j=0}^{m-1} \alpha_j \underset{\substack{Weak\\learner}}{h_j}(x)\right]$ Where each of the weak learners $h_j(x)$ is an extremly simple function of the input, and hence is not expected to contribute much (in isolation) to the classification performance. In most variants of boosting, the weak learners are threshold functions, $h_j(x) = a_j\left[f_j < \theta_j\right] + b_j\left[f_j \geq \theta_j\right] = \begin{cases} a_j, & if\ f_j < \theta_j \\ b_j, & otherwise \end{cases}$	 **Figure 14.5** A neural network for face detection (Rowley, Baluja, and Kanade 1998a) © 1998 IEEE. Overlapping patches are extracted from different levels of a pyramid and then pre-processed as shown in Figure 14.3b. A three-layer neural network is then used to detect likely face locations. 

		Which are also known as decision stumps. - The feature value f_j is the window response. - The threshold θ_j is learned by the algorithm - The weak learner outputs a_j if it believes it's a face, otherwise b_j The number of weak classifiers in a system is in the order of thousands. The key to the success of boosting is the method for incrementally selecting the weak learners and for re-weighting the training examples after each stage (Figure 14.7). The AdaBoost (Adaptive Boosting) algorithm does this by re-weighting each sample as a function of whether it is correctly classified at each stage, and using the stage-wise average classification error to determine the final weightings α_j among the weak classifiers, as described in Algorithm 14.1. While the resulting classifier is extremely fast in practice, the training time can be quite slow (in the order of weeks), because of the large number of feature (difference of rectangle) hypotheses that need to be examined at each stage.	Weak classifier 1 Weights increased Weak classifier 2 Weights increased Weak classifier 3 Final classifier **Figure 14.7** Schematic illustration of boosting, courtesy of Svetlana Lazebnik, after original illustrations from Paul Viola and David Lowe. After each weak classifier (decision stump or hyperplane) is selected, data points that are erroneously classified have their weights increased. The final classifier is a linear combination of the simple weak classifiers. We use the integral image to speed up the weak classifier response (Viola Jones)  1. 31 2 4 33 5 36 12 26 9 10 29 25 13 17 21 22 20 18 24 23 15 16 14 19 30 8 28 27 11 7 1 35 34 3 32 6 2. 31 33 37 70 75 111 43 71 84 127 161 222 56 101 135 200 254 333 80 148 197 278 346 444 110 186 263 371 450 555 111 222 333 444 555 666 15 + 16 + 14 + 28 + 27 + 11 = 101 + 450 – 254 – 186 = 111
	AdaBoost algorithm	 1. Input the positive and negative training examples along with their labels $\{(x_i, y_i)\}$, where $y_i = 1$ for positive (face) examples and $y_i = -1$ for negative examples. 2. Initialize all the weights to $w_{i,1} \leftarrow \frac{1}{N}$, where N is the number of training examples. (Viola and Jones (2004) use a separate N_1 and N_2 for positive and negative examples.) 3. For each training stage $j = 1 \dots M$: (a) Renormalize the weights so that they sum up to 1 (divide them by their sum). (b) Select the best classifier $h_j(x; f_j, \theta_j, s_j)$ by finding the one that minimizes the weighted classification error $e_j = \sum_{i=0}^{N-1} w_{i,j} e_{i,j}$, (14.3) $e_{i,j} = 1 - \delta(y_i, h_j(x_i; f_j, \theta_j, s_j))$. (14.4) For any given f_j function, the optimal values of (θ_j, s_j) can be found in linear time using a variant of weighted median computation (Exercise 14.2). (c) Compute the modified error rate β_j and classifier weight α_j, $\beta_j = \frac{e_j}{1 - e_j}$ and $\alpha_j = -\log \beta_j$. (14.5) (d) Update the weights according to the classification errors $e_{i,j}$ $w_{i,j+1} \leftarrow w_{i,j} \beta_j^{1-e_{i,j}}$, (14.6) i.e., downweight the training samples that were correctly classified in proportion to the overall classification error. 4. Set the final classifier to $h(x) = \text{sign}\left[\sum_{j=0}^{m-1} \alpha_j h_j(x)\right]$. (14.7)	
	Optimization by cascade	The figure to the right show the two first weak classifiers for a trained set, and you can see that they represent the eyes and the nose. Someone found out that they could optimize the algorithm by using the fact that the first weak classifiers will have a strong impact on if it is a face or not. They therefore changed the algorithm such that if the first couple of classifiers classified the patch a not a face, then they should terminate. In the given paper, this reduce the runtime for the detector to go from 6000 weak learner evaluations down to an average of 15 weak learner evaluations per window NB: The reason that we don't just terminate whenever a weak classifier reject the patch is - as the name suggest - that it is only a weak classifier, and will therefore make many misclassifications	  All Sub-windows T T T 1 2 3 Further Processing F F F Reject Sub-window
Face Classification	Eigenfaces (Training)	An Eigenface is a gray-level image projected onto lower dimensional subspaces. It rely on the observation that an arbitrary face image x can be compressed an reconstructed by starting with a mean image m and adding a small number of scaled signed images u_i, $\tilde{x} = m + \sum_{i=0}^{M-1} a_i u_i$, $(*)$ Where the signed basic images can be derived from an ensemble of training images using PCA (eigenvalue analysis). In more detail, we start with a collection of training images $\{x_j\}$, which we have flattened, such that each image patch is a $1D$ vector denoted x, and the dimension D pixels.	That is, an image can be represented as the mean of the image + the variation image for each of its eigenvector axis.  **Figure 14.13** Face modeling and compression using eigenfaces (Moghaddam and Pentland 1997) © 1997 IEEE: (a) input image; (b) the first eight eigenfaces; (c) image reconstructed by projecting onto this basis and compressing the image to 85 bytes; (d) image reconstructed using JPEG (530 bytes).

We can compute the mean image *m* and a covariance matrix

$$m = \frac{1}{N}\sum_{i=1}^{N} x_i$$

$$C = \frac{1}{N}\sum_{j=0}^{N-1}\left(x_j - m\right)\left(x_j - m\right)^T$$

We can apply the eigenvalue decomposition to represent this matrix as

$$\boxed{C = U\Lambda U^T = \sum_{i=0}^{N-1} \lambda_i u_i u_i^T}$$

Where $\lambda_i, i = 1, \dots, D$ are the eigenvalues of C and u_i are the eigenvectors,. (These can be called eigenpictures or eigenfaces).

Two important properties of the eigenvalue decomposition are that *the optimal (approximation) coefficients a_i for* ***any new image x*** *ca*n be computed as

$$a_i = (x - m) \cdot u_i, \qquad (14.11)$$

And, assuming the eigenvalues $\{\lambda_i\}$ are *sorted in decreasing order*, truncaing the approximation given in (*) at any point *M* gives the best possible approximation (least error) between $\tilde{x}$ and x.

Eigenface (Classification)

Truncating the eigenface decomposition of a face image (*) after *M* components is equivalent to projecting the image onto a linear subspace *F*, which we call the ***face space*.** Because the eigenvectors (eigenfaces) are orthogonal and of unit norm, the <u>distance of a **projected face** $\tilde{x}$ to the mean face *m*</u> can be written as

$$DIFS = \|\tilde{x} - m\| = \sqrt{\sum_{i=0}^{M-1} a_i^2}$$

where DIFS stands for *distance in face space*. The remaining distance between the original image *x* and its projection onto face space $\tilde{x}$, i.e. the *distance from face space* (DFFS), can be computed directly in pixel space and represents the "faceness" of a particular image.

$$DFFS = x - \tilde{x}$$

This measure can then be used to check if the input patch actually is a face - if DFFS is big, it is unlikely a face.

The **distance between two different faces** in face space can be measured as:

$$DIFS(x, y) = \|\tilde{x} - \tilde{y}\| = \sqrt{\sum_{i=0}^{M-1}\left(a_i - b_i\right)^2},$$

where the $b_i = \left(y - m\right) \cdot u_i$ are the eigenfaces coefficients corresponding to y. Which can be used to classify the input face patch to a known face patch by thresholding the distance measurement. If the DFFS is small but the smallest DIFS is big, then it means that the face patch is a new person. We can then add it to the data, such that we expand the number of people in the classifier

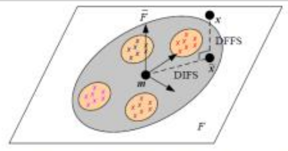


Figure 14.14 Projection onto the linear subspace spanned by the eigenface images (Moghaddam and Pentland 1997) © 1997 IEEE. The distance from face space (DFFS) is the orthogonal distance to the plane, while the distance in face space (DIFS) is the distance along the plane from the mean image. Both distances can be turned into Mahalanobis distances and given probabilistic interpretations.

Eigenface (limitation)

In many real-life applications, we can't be sure that same-person face cluster nicely together in face space. This is because the *"intra-class"* difference can be larger than *"inter-class"* variations, i.e. difference between persons!
In face space, this can be manifest as overlapping clusters as seen on the figure.

We can tackle this issue by noting that the Eigenface DIFS does not take the covariance matrix C, calculated by the EVD, into account.
If we interpret the covariance matrix C as the covariance of a *multi-variate Gaussian* we can turn the *DIFS* into a log likelihood by computing the Mahalanobis distance

$$DIFS' = \|\tilde{x} - m\|_{C^{-1}} = \sqrt{\sum_{i=0}^{M-1} a_i^2/\lambda_i^2}$$

Instead of measuring the squared distance along each principal component in face space F,the Mahalanobis distance measures the **ratio between the squared distance** and the corresponding variance $\sigma_i^2 = \lambda_i$ and then sums these squared ratios (per-component log-likelihoods).
Alternative we can pre-scale each eigenvector by the inverse square root of its corresponding eigenvalue,

$$\hat{U} = U\Lambda^{-1/2}, \qquad (14.15)$$

Equation (14.15) mean that the Euclidean distance in feature space now correspond directly to log likelihoods.

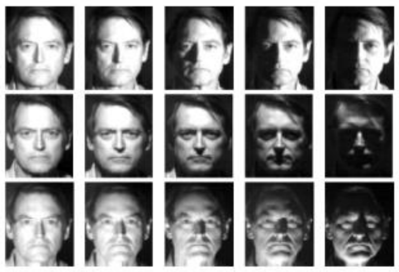


Figure 14.15 Images from the Harvard database used by Belhumeur, Hespanha, and Kriegman (1997) © 1997 IEEE. Note the wide range of illumination variation, which can be more dramatic than inter-personal variations.

Fisherfaces

One of the biggest advantages of Eigenface is that they reduce the comparison between faces from a P-dimensional difference in pixel space to an M-dimensional difference in face space,

$$\|x - x_k\| = \|a - a_k\|$$

where $a = U^T(x - m)$, equation (14.11), involves computing a dot product between the signed difference-from-mean image $(x - m)$ and each of the eigenfaces u_i. As note above, the Euclidean does not account for the information about covariance.
If we look at figure 14.15 we note that the intra class is big. So we want to approximate faces by a **linear subspace** that weights discriminates between different classes (people) is **less sensitive to within-class variations**.

If we take the three classes on Figure (14.16) as an example. We can see, the ***distributions within a class*** *(indicated by the tilted colored axes)* ***are elongated*** and **tilted with respect to the main face space PCA**, which is aligned with the black x and y axes. We compute the total within-class scatter matrix as

$$S_W = \sum_{k=0}^{K-1} S_k = \sum_{k=0}^{K-1}\sum_{i \in X_k}\left(x_i - m_k\right)\left(x_i - m_k\right)^T$$

where m_k is the mean of class k and S_k is the within class scatter matrix. Similarly, we compute the between-class scatter as

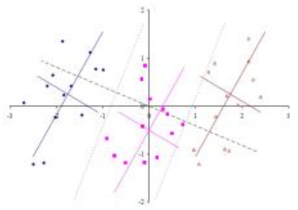
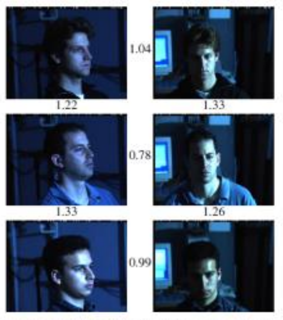
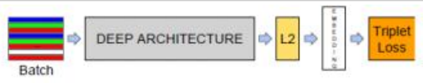
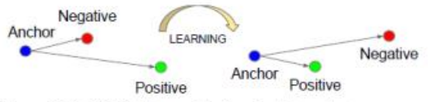
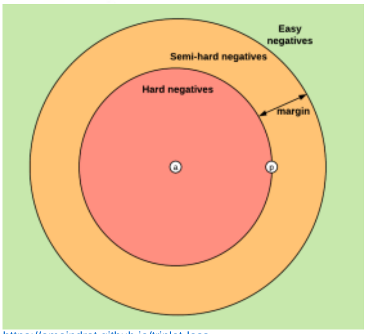


Figure 14.16 Simple example of Fisher linear discriminant analysis. The samples come from three different classes, shown in different colors along with their principal axes, which are scaled to $2\sigma_i$. (The intersections of the tilted axes are the class means m_k.) The dashed line is the (dominant) Fisher linear discriminant direction and the dotted lines are the linear discriminants between the classes. Note how the discriminant direction is a blend between the principal directions of the between-class and within-class scatter matrices.

		$S_B = \sum_{k=0}^{K-1} N_k (m_k - m)(m_k - m)^T$ where N_k are the number of exemplars in each class and m is the overall mean. To compute the most discriminating direction, *Fisher's linear discriminant (FLD), (A.k.a. Linear discriminant analysis (LDA))*, select the direction u that result in the largest ratio between the projected between-class and within-class cariation $u^* = \underset{u}{\operatorname{argmax}} \frac{u^T S_B u}{u^T S_W u}$ which is equivalent to finding the eigenvector corresponding to the largest eigenvalue of the generalized eigenvalue problem $S_B u = \lambda S_W u, \qquad or, \qquad \lambda u = S_W^{-1} S_B\, u$ As we can see on figure (14.16), **using the direction $\boldsymbol{u}^*$ results in a better seperation between the classes** than the using the dominant PCA direction, which is the horixontal axis.	
FaceNet		FaceNet is another approach to classify faces, and they have managed to solve the limitation of illumination. They have done this by introducing an error term called *Tripled Loss*, which better ignore external effects such as illumination, head pose, ect. The figure to the right show three different people (vertical) in different light settings (horizontal). Unlike parameter-free methods such as eigen decomposition, FaceNet is build using deep nets which use thousands of learnable parameters to find f	 Figure 1. **Illumination and Pose invariance.** Pose and illumination have been a long standing problem in face recognition. This figure shows the output distances of FaceNet between pairs of faces of the same and a different person in different pose and illumination combinations. A distance of 0.0 means the faces are identical, 4.0 corresponds to the opposite spectrum, two different identities. You can see that a threshold of 1.1 would classify every pair correctly.
	Loss function	The embedding is represented by $f(x) \in \mathbb{R}^d$. It embeds an image *x* into *d*-dimensional Euclidean space. Additional, the embedding is constrained to live on the *d*-dimensional hypersphere $\lVert f(x) \rVert_2 = 1$. The lost functions is designed as a nearest-neightbor classification. Here the goal is to ensure that an image x_i^a (anchor) of a specific person is closer to all other images x_i^p (positive) of the same person, than it is to any image x_i^n (negative) of any other person. See figure 3. Thus we want, $\lVert f(x_i^a) - f(x_i^p) \rVert_2^2 + \alpha < \lVert f(x_i^a) - f(x_i^n) \rVert_2^2, \qquad (1)$ $\forall \left(f(x_i^a), f(x_i^p), f(x_i^n) \right) \in \mathcal{T}, \qquad (2)$ where α is a margin that is enforced between positive and negative pairs. $\mathcal{T}$ is the set of all possible triples in the training set and has cardinality *N*. The loss being minimized is the $L = \sum_i^N \left[\lVert f(x_i^a) - f(x_i^p) \rVert_2^2 - \lVert f(x_i^a) - f(x_i^n) \rVert_2^2 + \alpha \right]_+$ In order to ensure fast convergence it is crucial to select triplets that violate the triplet constraint in Eq. (1). This means that, given x_i^a, we want to select an x_i^p (hard positive) such that $\operatorname{argmax}_{x_i^p} \lVert f(x_i^a) - f(x_i^p) \rVert_2^2$ and similarly x_i^n (hard negative) such that $\operatorname{argmin}_{x_i^n} \lVert f(x_i^a) - f(x_i^n) \rVert_2^2$. It is infeasible to compute the $argmin$ and $argmax$ across the whole training set. Additionally, it might lead to poor training, as mislabelled and poorly imaged faces would dominate the hard positives and negatives. We have two choices that avoid this issue: - Generate triplets offline every n steps, using the most recent network checkpoint and computing the $argmin$ and $argmax$ on a subset of the data. - Generate triplets online. This can be done by selecting the hard positive/negative exemplars from within a mini-batch. In the given paper they use the min-batch method.	 Batch DEEP ARCHITECTURE L2 EMBEDDING Triplet Loss Figure 2. **Model structure.** Our network consists of a batch input layer and a deep CNN followed by L_2 normalization, which results in the face embedding. This is followed by the triplet loss during training.  Anchor Negative Positive LEARNING Anchor Negative Positive Figure 3. The **Triplet Loss** minimizes the distance between an *anchor* and a *positive*, both of which have the same identity, and maximizes the distance between the *anchor* and a *negative* of a different identity.  Easy negatives Semi-hard negatives Hard negatives margin a p https://omoindrot.github.io/triplet-loss

Object Detection

Thursday, 26 March 2020 08.04

Terminology

In object detection we have two task
- Instance-level detection
  - Detect a specific instance of an object
- Category-level
  - Detect an instance of a certain object type

Instance level detection is most often used in robotic to detect very specific objects, whereas category detection is more general.

Template matching vs statistically learning

For time-critical applications, template matching is an attractive solution because new objects can be easily learned online, in contrast to statistical-learning techniques that require many training samples .

Linemod

Linemod is a instance-level detection.
Linemod is a **template matching model**. Template matching is easy and intuitive, but has its limitation:
- All variations of the object needs to be modeled
  - Scale changes
  - Viewpoint changes
  - Illumination changes
  - Etc

This requires a lot of templates, which makes them slow. However, LineMod addresses some of these issues. Furthermore, often 3D CAD models of the desired objects needs to be available.

LineMod is a multimodal template matching which uses gradient/normal features for description in color and depth images, that does not depends on 3D CAD models. These features are somewhat robust to certain transformations and illumination changes. Furthermore, their matching is fast - that is near real time for thousands of templates.

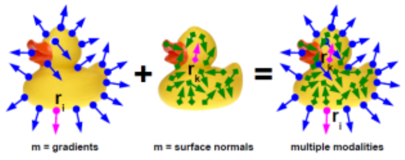


Figure 2. A toy duck with different modalities. **Left:** Image gradients are mainly found on the contour. The gradient location r_i is displayed in pink. **Middle:** Surface normals are found on the body of the duck. The normal location r_k is displayed in pink. **Right:** our approach combines multiple cues which are complementary: gradients are usually found on the object contour while surface normals are found on the object interior

Color and depth features

Instead of using grayscale images - as classical template matching algorithms - LineMod calculate the normalized gradient, since considering only the normalized gradients and not their magnitudes makes the measure robust to contrast changes.
To increase robustness, we **compute the normalized gradients on each color channel** of our input image separately and for each image location use the normalized **gradient of the channel whose magnitude is largest**.
The effect can be seen on the right figure, where the edge of the duck is clearly visible.

Given an RGB color image $\mathcal{I}$, compute the normalized gradient map $\mathcal{I}_g(x)$ at location x with

$$\mathcal{I}_g(x) = \frac{\partial \hat{\mathcal{C}}}{\partial x}$$

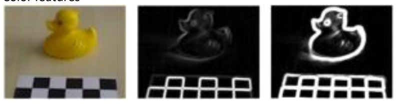
where

$$\hat{\mathcal{C}}(x) = \underset{\mathcal{C} \in \{R,G,,B\}}{\operatorname{argmax}} \left\| \frac{\partial \mathcal{C}}{\partial x} \right\|$$

and R,G,B are RGB color channels of the corresponding color image.

Furthermode, LineMode used depth features to find cue about the object. And given that there is no concept of "depth gradient" they used local surface normal as their depth gradient. This is valid, because the gradient given some kind of differentiable value of the a function, and the same does the surface normal.
However, calculating surface normal in point clouds are expensive, so they approximated the surface normal by Taylor expansions.
However, the approximation does not hold at occluded areas, so the contribution of pixels, whose depth difference with the central pixel are higher than a threshold, are ignored.

Color features



Depth features

Quantization of features

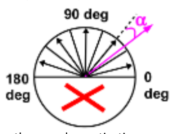
From the above features (gradient and surface normal), we have the computed image/depth gradients which are normalized. We therefore only have an direction and no magnitude. The normalization is done to make the features more robust against noise and illumination change in the image.

Color Gradient Quantization
In order to quantize the gradient map we omit the gradient direction, consider only the gradient orientation and divide the orientation space into n_0 equal spacings as shown in the top figure on the right. To make the quantization robust to noise, we assign to each location the gradient whose quantized orientation occurs most often in a 3×3 neighborhood. We also keep only the gradients whose norms are larger than a small threshold.

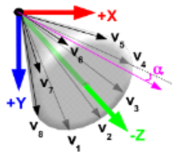
Depth Surface Normal Quantization:
As seen on the bottom Figure, we measure the angles between the computed normal and a set of precomputed vectors, v_i, to quantize the normal directions into n_0 bins. These vectors are arranged in a right circular cone shape originating from the peak of the cone **pointing towards the camera**. To make the quantization robust to noise, we assign to each location the quantized value that occurs most often in a 5 × 5 neighborhood.

Summary
The resulting directions are quantized into predefined orientation bins:
- Color gradient orientations are binned into a small number of bins, corresponding to a number of directions between 0 and 180 degree.
- The normal vectors are a bit more difficult, because they are 3D - however, the direction of all the normal vectors will be the same. A cone is defined and the distance to the different normal vectors in the database are then found.

Color gradient quantization



Depth normal quantization



Matching

The template matching is preformed similarly as normal template matching, that is, sliding a window over the whole image. This have to be done for both the depth and color Image. The matching score is then simply the cross-correlation in the two domains at each pixel location.

In the color domain, each object template gradient (at location r) gets multiplied by the dot product with each image gradient (at location t):

$$f_{\mathcal{g}}\left(\mathcal{O}_{\mathcal{g}}(r), \mathcal{I}_{\mathcal{g}}(t)\right) = \left| \underbrace{\mathcal{O}_{\mathcal{g}}(r)^T}_{\substack{Normalized \\ gradient\ map \\ \boldsymbol{reference}}} \cdot \underbrace{\mathcal{I}_{\mathcal{g}}(t)}_{\substack{Normalized \\ gradient\ map \\ \boldsymbol{current\ image}}} \right|$$

The same is true for the normal vectors, but the absolute values is unnecessary, since they are never anti-parallel:

$$f_{\mathcal{D}}\left(\mathcal{O}_{\mathcal{D}}(r), \mathcal{I}_{\mathcal{D}}(t)\right) = \underbrace{\mathcal{O}_{\mathcal{D}}(r)^T}_{\substack{Normalized \\ surface\ map \\ \boldsymbol{Referance}}} \cdot \underbrace{\mathcal{I}_{\mathcal{D}}(t)}_{\substack{Normalized \\ surface\ map \\ \boldsymbol{Current\ image}}}$$

The final matching function between a single object template and an image window centered at pixel position c looks like this:

$$\mathcal{E}\left(\{\mathcal{I}_m\}_{m\in\mathcal{M}}, \mathcal{T}, c\right) = \sum_{(r,m)\in\mathcal{P}} \left(\max_{t\in\mathcal{R}(c+r)} f_m(\mathcal{O}_m(r), \mathcal{I}_m(t))\right)$$

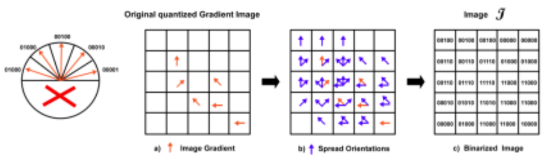
m denotes the two modalities (color($m = \mathcal{g}$)/depth($m = \mathcal{D}$)) and the sum runs over all pixels r in the template

Binarization

In the matching process above, we have not utilized the advantage of quantized features in the matching. LineMod used some clever trick to speeding up the processing such that the dot product does not need to be calculated for every pixel.

What we do is we take the image gradient and then we spread them out to neighboring pixel, as show on the figure, which mean that the matching can be run with a non-overlapping windows (or at least with a stride). Furthermore, each orientation can be represented in a binary string which allows for binary matching - which is fast. (Note the values of the direction vector.)

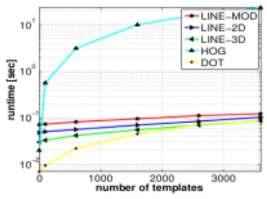
Matching is done with Hamming distances



Results

The figure show the LineMod matches for ~3k templates with a VGA image at 10Hz, which runs on a CPU. We can see that the matching speed is very fast.

Furthermore,
- LineMod has quite impressive tolerance towards clutter and occlusions
- LineMod matches non-textured objects, where e.g. SIFT would likely fail
- However, the tested objects are fairly distinct in the modalities used (geometry and edge profile)



Objects as Points (CenterNet)

We look at a neural-based detector which the author called CenterNet
CenterNet is - like many other neural detectors - a category-level detector
However, this does not mean that it cannot be retrained into an instance detector, category-level detection is just usually just more general/hard/interesting.

Unlike other neural detectors, that randomly generate bounding boxes in an image and then tries to rescale them to fit an object, the CenterNet represent objects by a single point at the bounding box center. Other properties, such as object size, dimension, 3D extent, orientation, and pose are then regressed directly from image features at the center location. **Object detection is then a standard keypoint estimation problem**

NB: Another detector is also called CenterNet

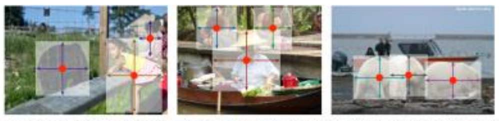
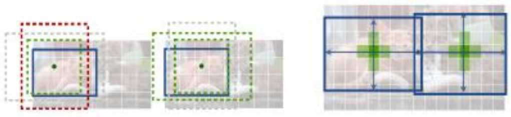


Figure 2: We model an object as the center point of its bounding box. The bounding box size and other object properties are inferred from the keypoint feature at the center. Best viewed in color.



(a) Standard anchor based detection. Anchors count as positive with an overlap $IoU > 0.7$ to any object, negative with an overlap $IoU < 0.3$, or are ignored otherwise.

(b) Center point based detection. The center pixel is assigned to the object. Nearby points have a reduced negative loss. Object size is regressed.

Figure 3: Different between anchor-based detectors (a) and our center point detector (b). Best viewed on screen.

Overall approach

CenterNet works by scanning through the image with a stride of $R = 4$.
At each (strided) location, the classifier predicts whether it is an object center
- The object size is predicted
- And finally an offset correction is made to compensate for inaccuracies caused by striding

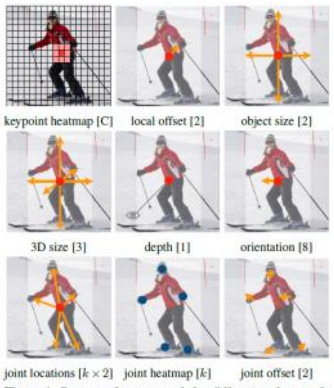


Figure 4: Outputs of our network for different tasks: *top* for object detection, *middle* for 3D object detection, *bottom:* for pose estimation. All modalities are produced from a common backbone, with a different 3×3 and 1×1 output convolutions separated by a ReLU. The number in brackets indicates the output channels. See section 4 for details.

Loss

Like other networks, the goal is to find a loss function that can be calculated and then propagated through the network. For CenterNet the overall detector loss is

$$L_{det} = L_k + \lambda_{size} L_{size} + \lambda_{off} L_{off}$$

And contain three terms
- Classification loss
- Object size loss
- Center offset loss

The λ_{size} and λ_{off} determin the scaling between the losses.

Classification loss
We train the keypoint prediction network as follows: For each ground truth keypoint $p \in \mathcal{R}$ of class c, we compute a low-resolution equivalent $\tilde{p} = \left\lfloor \frac{p}{R} \right\rfloor$. We then splat all ground truth keypoints onto a heatmap $Y \in [0,1]^{\frac{W}{R}\times\frac{H}{R}\times C}$, where C is the number for classes, using a Gaussian kernel $Y_{xyz} = \exp\left(-\frac{(x-\tilde{p}_x)^2+(y-\tilde{p}_y)^2}{2\sigma_p^2}\right)$, where σ_p is an object size-adaptive standard deviation. If two Gaussians of the same class overlap, we take the element-wise maximum. The training objective is a penalty-reduced pixelwise
logistic regression with **focal loss**:

$$L_k = -\frac{1}{N}\sum_{xyc}\begin{cases}\left(1-\hat{Y}_{xyc}\right)^{\alpha}\cdot\log\left(\hat{Y}_{xyc}\right), & if\ Y_{xyc}=1\\ \left(1-Y_{xyc}\right)^{\beta}\cdot\left(\hat{Y}_{xyc}\right)^{\alpha}\cdot\log\left(1-\hat{Y}_{xyc}\right), & otherwise\end{cases}$$

where α and β are hyper-parameters of the focal loss, N is the number of keypoints in image I, and Y is the ground truth heatmap mapped by a Gaussian kernel Y_{xyc}. $\hat{Y}_{xyc}$ is a prediction where $\hat{Y}_{xyc} = 1$ corresponds to a detected keypoint and $\hat{Y}_{xyc} = 0$ for background.

The focal loss down scale the loss for a specific classification the classifier is already very sure about his specific prediction

Offset loss
Because we downscale the image by $R = 4$ when we used the stride, the CenterNet needs to compensate for the down sampling of $R = 4$.
If the center prediction is simple upscaled by $R = 4$ to to go back to the original image dimension, there will be an offset from the ground truth center position. Therefore, the net actively tries to predict this offset:

$$L_{off} = \frac{1}{N}\sum_{p}\left|\hat{O}_{\tilde{p}} - \left(\frac{p}{R} - \tilde{p}\right)\right|$$

where $\hat{\boldsymbol{O}}$ **is the predicted offset** and the term in the parenthesis is the offset we want to predict - That is the offset between the downscaled ground truth center position p and the predicted on.

Size loss
The size loss is simple and just penalizes discrepancies between the predicted width/height and the true width/height

$$L_{size} = \frac{1}{N}\sum_{k=1}^{N}\left|\hat{S}_{pk} - s_k\right|$$

where $\hat{\boldsymbol{S}}_{\boldsymbol{pk}}$ **is the predicted size** and s_k is the truth.
In contrast with other regression methods that use L_2 loss, here L_1 loss is used, which often works better in practice.

Over all training loss function:
$L_{det} = L_k + \lambda_{size}L_{size} + \lambda_{off}L_{off}$

Inference

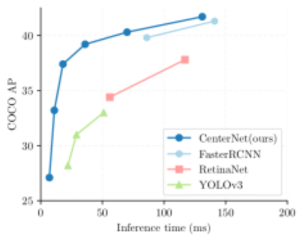
When running the net, a bounding box is trivially recovered with the predicted center (x, y), the predicted size (w, h) and the predicted offset (the deltas):

$$\left(\hat{x}_i + \delta\hat{x}_i - \frac{\hat{w}_i}{2}, \hat{y}_i + \delta\hat{y}_i - \frac{\hat{h}_i}{2}, \hat{x}_i + \delta\hat{x}_i + \frac{\hat{w}_i}{2}, \hat{y}_i + \delta\hat{y}_i + \frac{\hat{h}_i}{2},\right)$$

Since nearby pixel can get classified as objects, an 8-point non-maxima suppression is performed to remove overlapping center predictions

Results

We can see on the figure that with the same computation budget as competing methods, CenterNet achieves better results



Segmentation

Wednesday, 10 June 2020 08.24

Pixel space clustering

Mean-shift and mode finding techniques, such as k-means and mixtures of Gaussian, model the feature vectors associated with each pixel (e.g. color and position) as samples from unknown probability density function and then try to find clusters (modes) in this distribution.

If we consider the color image seen on figure (5.15a). Figure (5.16b) shows the distribution of pixels in L*u*v* space, which is equivalent to what a vision algorithm that **ignores spatial location** would see. To make the visualization simpler, let us only consider the L*u* coordinates, as shown in (Figure 5.16c). How many obvious (elongated) clusters do you see? How would you go about finding these clusters?

The k-means and mixtures of Gaussians techniques use a **parametric model** of the density function to answer this question, i.e., they assume the density is the superposition of a small number of simpler distributions (e.g., Gaussians) whose locations (centers) and shape (covariance) can be estimated. **Mean shift**, on the other hand, smoothes the distribution and finds its peaks as well as the regions of feature space that correspond to each peak. Since a complete density is being modeled, this approach is called non-parametric

Comparison
k-means and *mixtures of Gaussians* use a **parametric form** to model the **probability density function** being segmented, *mean shift* **implicitly models** this distribution using a smooth continuous **non-parametric** model.
The key to mean shift is a technique for ***efficiently finding peaks in this high-dimensional data distribution*** *without ever computing the complete function explicitly.*

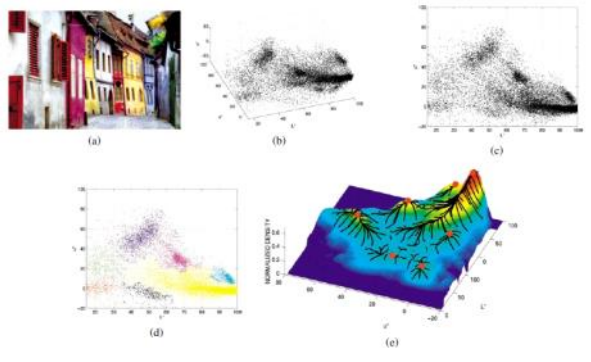


Figure 5.16 Mean-shift image segmentation (Comaniciu and Meer 2002) © 2002 IEEE: (a) input color image; (b) pixels plotted in L*u*v* space; (c) L*u* space distribution; (d) clustered results after 159 mean-shift procedures; (e) corresponding trajectories with peaks marked as red dots.

K-means and mixtures of Gaussians

K-means clustering implicitly model the probability density as a superposition of spherically symmetric distribution, without probabilistic reasoning or modelling. Instead, it find the clusters given a user defined number of clusters k, given an initial random sampling of centers.
In contrast, in Mixtures of Gaussians, each cluster center is augmented by a covariance matrix whose values are re-estimated from the corresponding samples. Further, the nearest neighbor assumption is replaced by a statistic distance, Mahalanobis distance

$$d(x_i, \mu_k; \Sigma_k) = \|x_i - \mu_k\|_{\Sigma_k^{-1}} = (x_i - \mu_k)^T \Sigma_k^{-1} (x_i - \mu_k)$$

where x_i are the input samples, μ_k are the cluster centers, and Σ_k are their covariance estimates. Samples can be associated with the nearest cluster center (a **hard assignment** of membership) or can be **softly assigned** to several nearby clusters.
The latter is the most used approach, because it corresponds to iteratively re-estimating the parameters for a mixture Gaussian density function.

$$p(x|\{\pi_k, \mu_k, \Sigma_k\}) = \sum_k \pi_k \mathcal{N}(x|\,\mu_k, \Sigma_k)$$

is the normal (Gaussian) distribution.

The Algorithm
To iteratively compute (a local) maximum likely estimate for the unknown mixture parameters $\{\pi_k, \mu_k, \Sigma_k\}$, the **expectation maximization (EM)** algorithm proceeds in two alternating stages.

1. The **expectation** stage (E step) estimates the responsibilities
$$z_{ik} = \frac{1}{Z_i}\pi_k \mathcal{N}(x|\mu_k, \Sigma_k), \qquad with \sum_k z_{ik} = 1$$
which are the estimates of how likely a sample x_i was generated from the kth Gaussian cluster.
2. **The maximization** stage (M step) updates the parameter values
$$\mu_k = \frac{1}{N_k}\sum_i z_{ik} x_i$$
$$\Sigma_k = \frac{1}{N_k}\sum_i z_{ik}(x_i - \mu_k)(x_i - \mu_k)^T$$
$$\pi_k = \frac{N_k}{N}$$
where
$$N_k = \sum_i z_{ik}$$
is an estimate of the number of sample points assigned to each cluster.

Mean Shift

If we consider the data points shown in Figure 5.16c, and if we think of them as points that have been drawn from some probability density function. Then we want to compute this density function, as visualized in Figure 5.16e, so we could find its major **peaks (modes)** and identify regions of the input space that **climb to the same peak** as being part of the same region.

We want to estimate the density function given a sparse set of samples. One of the simplest approaches it to just smooth the data, e.g. by convolving it with a fixed kernel of width h,

$$\underbrace{f(x)}_{local\ density} = \sum_i K(x - x_i) = \sum_i \underbrace{k\left(\frac{\|x - x_i\|^2}{h^2}\right)}_{kernel}, \qquad (5.34)$$

where x_i are the input samples and $k(r)$ is the kernel function. Once we have computed $f(x)$, as shown in Figures 5.16e and 5.17, we can find its **local maxima** using **gradient ascent** or some other **optimization technique**.
The kernel in mean shift plays a similar role as the distance functions in k-means and GMMs. The input to the kernel, equation (5.34), is a pairwise distance, scaled by a **bandwidth** h. The main design choices of mean shift are the kernel type and the bandwidth

For higher dimensions, the above "brute force" method becomes computationally prohibitive to evaluate $f(x)$ over the complete search space. Therefore, does **mean shift** use a variant of called ***multiple restart gradient descent***.
Starting at some guess for a local maximum, y_k, which can be a random input data point x_i, mean shift computes the gradient of the density estimate $f(x)$ at y_k and takes an uphill step in that direction (Figure (5.17)). The gradient of $f(x)$ is given by

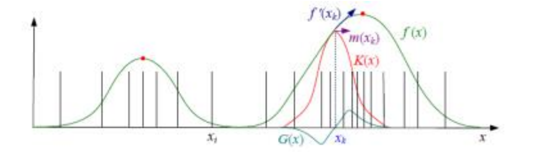


Figure 5.17 One-dimensional visualization of the kernel density estimate, its derivative, and a mean shift. The kernel density estimate $f(x)$ is obtained by convolving the sparse set of input samples x_i with the kernel function $K(x)$. The derivative of this function, $f'(x)$, can be obtained by convolving the inputs with the derivative kernel $G(x)$. Estimating the local displacement vectors around a current estimate x_k results in the mean-shift vector $m(x_k)$, which, in a multi-dimensional setting, point in the same direction as the function gradient $\nabla f(x_k)$. The red dots indicate local maxima in $f(x)$ to which the mean shifts converge.

$$\nabla f(x) = \sum_i (x_i - x) G(x - x_i) = \sum_i (x_i - x) g\left(\frac{\|x - x_i\|^2}{h^2}\right)$$

where

$$g(r) = -k'(r)$$

and $k'(r)$ is the first derivative of $k(r)$. We can re-write the gradient of the density function as

$$\nabla f(x) = \left[\sum_i F(x - x_i)\right] m(x)$$

where the vector

$$\boxed{m(x) = \frac{\sum_i x_i G(x - x_i)}{\sum_i G(x - x_i)} - x,} \qquad (mean\ shift)$$

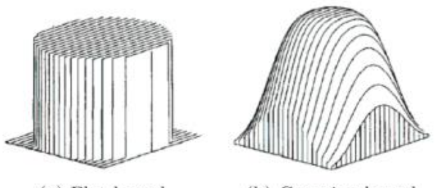
is called the mean shift, since it is the different between the weighted mean of the neighbors x_i around x and the current value of x.
In the **mean-shift procedure** (How we iterate the point), the **current estimate of the mode $\boldsymbol{y_k}$** at iteration k is replaced by its locally weighted mean

$$\boxed{y_{k+1} = y_k + m(y_k) = \frac{\sum_i x_i G(x - x_i)}{\sum_i G(x - x_i)}}$$

where the subscript k denotes the iterate index.

- Initialize each y with each data point x
- Choose a kernel type - popular choices are:
    - Epanechnikov:
      $$k_E(r) = \max(0, 1 - r)$$
    - Gaussian:
      $$k_N(r) = \exp\left(-\frac{1}{2} r\right)$$
    - where we again have:
      $$r = \frac{\|x - x_i\|^2}{h^2}$$
- Get the corresponding negative-derivatives:
    - Epanechnikov (flat):
      $$G(x - x_i) = -K'(x - x_i) = \begin{cases} 1, & \|x - x_i\| \le h \\ 0, & otherwise \end{cases}$$
    - Gaussian (radial):
      $$\mathrm{G}(\mathrm{x} - \mathrm{x_i}) = -\mathrm{K}'(\mathrm{x} - \mathrm{x_i}) = \frac{1}{2} \exp\left(\frac{\|\mathrm{x} - \mathrm{x_i}\|}{2\mathrm{h}^2}\right)$$
- This is actually enough to run the update on the y's!
  $$y_{k+1} = y_k + m(y_k) = \frac{\sum_i x_i G(x - x_i)}{\sum_i G(x - x_i)}$$

Some notes
- If you look at the kernel derivatives, you see that they decay rapidly for neighbor points longer away from than the bandwidth - for the flat kernel, the weight actually goes to zero
- For the radial kernel, the weight becomes very small
- You can therefore do with first performing a radius search (radius of h) around the midpoint and only use the neighbors for the update
- For the **flat kernel**, this is exact, since the kernel value beyond h is zero anyway
- For the **radial kernel**, it becomes and approximation, since the kernel is non-zero at infinity


(a) Flat kernel (b) Gaussian kernel

U-Net

The U-net is an alternative method to calculating segmentation.
U-net is a learning based neural segmentation algorithm that used a convolutional network. When the network is trained, the U-net predicts correct class labels for each pixel in an input image.
Like clustering methods, learning based methods also associates labels to individual pixels, but there are differences:
- A learning-based system segments based on what you teach it, which can be a bit more sophisticated than just color consistency
- The target of U-Net and many others is often called *semantic segmentation* to indicate that they can be used to segment e.g. full objects with the same label.

As seen on figure 1, the network of the U-Net resemble a U.
It consists of the repeated application of two 3x3 convolutions (unpadded convolutions), each followed by a rectied linear unit (ReLU) and a 2x2 max pooling operation with stride 2 for downsampling. At each downsampling step we double the number of feature channels. Every step in the expansive path consists of an upsampling of the feature map followed by a 2x2 convolution (\up-convolution") that halves the number of feature channels, a concatenation with the correspondingly cropped feature map from the contracting path, and two 3x3 convolutions, each followed by a ReLU.
The cropping is necessary due to the loss of border pixels in every convolution. At the final layer a 1x1 convolution is used to map each 64-component feature vector to the desired number of classes. In total the network has 23 convolutional layers.

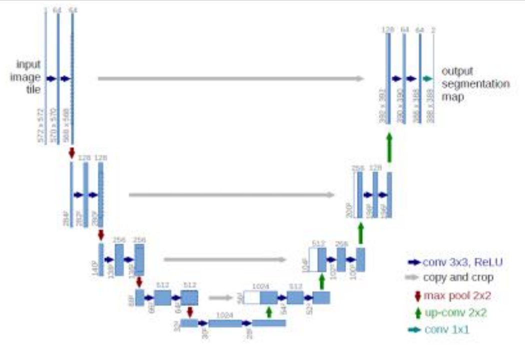


Fig. 1. U-net architecture (example for 32x32 pixels in the lowest resolution). Each blue box corresponds to a multi-channel feature map. The number of channels is denoted on top of the box. The x-y-size is provided at the lower left edge of the box. White boxes represent copied feature maps. The arrows denote the different operations.

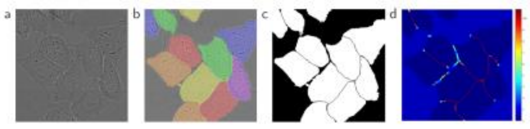


Fig. 3. HeLa cells on glass recorded with DIC (differential interference contrast) microscopy. (a) raw image. (b) overlay with ground truth segmentation. Different colors indicate different instances of the HeLa cells. (c) generated segmentation mask (white: foreground, black: background). (d) map with a pixel-wise loss weight to force the network to learn the border pixels.

Training

The energy function is computed by a pixel-wise soft-max over the final feature map combined with the cross entropy loss function. The soft-max is defined as

$$p_k(x) = \frac{\exp(a_k(x))}{\sum_{k'=1}^{K} \exp(a_{k'}(x))}$$

where $a_k(x)$ denotes the **activation in feature channel $\boldsymbol{k}$** at the pixel position $x \in \Omega \subset \mathbb{Z}^2$. K is the number of classes and **$\boldsymbol{p_k(x)}$ is the approximated maximum-function**. I.e. $p_k(x) \approx 1$ for the k that has the maximum activation $a_k(x)$ and $p_k(x) \approx 0$ for all other k. The cross entropy then penalizes at each position the deviation of $p_{\ell(x)}(x)$ from 1 using

$$E = \sum_{x \in \Omega} w(x) \cdot \log\left(p_{\ell(x)}(x)\right)$$

where $\ell: \Omega \to \{1, \dots, K\}$ is the true label of each pixel and $w: \Omega \to \mathbb{R}$ is a weight map to give some pixels more importance in the training.

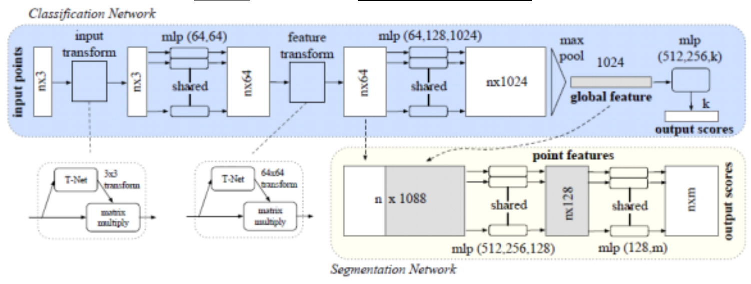
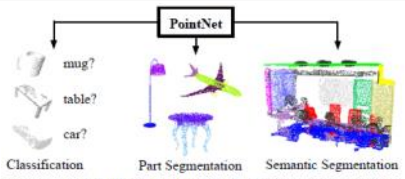
The separation border is computed using morphological operations. The weight map is then computed as

$$w(x) = w_c(x) + w_0 \cdot \exp\left(-\frac{(d_1(x) + d_2(x))^2}{2\sigma^2}\right)$$

		where $w_c: \Omega \rightarrow \mathbb{R}$ is the weight map to balance the class frequencues, w_0 is a border term between touching segments, $d_1: \Omega \rightarrow \mathbb{R}$ denotes the distance to the border of the nearest call and $d_2: \Omega \rightarrow \mathbb{R}$ the distance to the border of the second nearest call.	

PointNet

Wednesday, 10 June 2020 09.59

Problem Statement		We design a deep learning framework that directly consumes **unordered point sets** as inputs. A point cloud is represented as a set of 3D points $\{P_i \mid i = 1, \dots, n\}$, where **each point $\boldsymbol{P_i}$ is a vector of its $(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z})$ coordinate** plus extra feature channels such as color, normal etc. For the **object classification** task, the input point cloud is either directly sampled from a shape or pre-segmented from a scene point cloud. The deep network outputs k scores for all the ***k* candidate classes**. For semantic segmentation, the input can be a **single object** for part region segmentation, or a **sub-volume** from a 3D scene for object region segmentation. Our model will output $n \times m$ scores for each of the ***n* points** and each of the ***m* semantic subcategories**.  Classification Network input points input transform mlp (64,64) shared nx3 nx3 nx64 feature transform nx64 mlp (64,128,1024) shared nx1024 max pool 1024 global feature mlp (512,256,k) k output scores T-Net 3x3 transform matrix multiply T-Net 64x64 transform matrix multiply point features n x 1088 shared nx128 shared nxm output scores mlp (512,256,128) mlp (128,m) Segmentation Network Figure 2. **PointNet Architecture.** The classification network takes n points as input, applies input and feature transformations, and then aggregates point features by max pooling. The output is classification scores for k classes. The segmentation network is an extension to the classification net. It concatenates global and local features and outputs per point scores. "mlp" stands for multi-layer perceptron, numbers in bracket are layer sizes. Batchnorm is used for all layers with ReLU. Dropout layers are used for the last mlp in classification net.	 PointNet mug? table? car? Classification Part Segmentation Semantic Segmentation Figure 1. **Applications of PointNet.** We propose a novel deep net architecture that consumes raw point cloud (set of points) without voxelization or rendering. It is a unified architecture that learns both global and local point features, providing a simple, efficient and effective approach for a number of 3D recognition tasks.
Properties of the input		The network input is a subset of points from an Euclidean space. The points have three main properties that the network should be able to handle: - Unordered: ○ Unlike pixel arrays in images or voxel arrays in volumetric grids, point cloud is a set of points without specific order. ○ A network that consumes N 3D point sets needs to be invariant to $N!$ permutations of the input set in data feeding order. - Interaction among points: ○ The points are from a space with a distance metric. It means that points are not isolated, and neighboring points form a meaningful subset. ○ The model needs to be able to capture local structures from nearby points, and the combinatorial interactions among local structures. - Invariance under transformations: ○ As a geometric object, the learned representation of the point set should be invariant to certain transformations. For example, rotating and translating points all together should not modify the global point cloud category nor the segmentation of the points. The PointNet has three key modules in order to handle the above issues: - A max pooling layer as a **symmetric function** to aggregate information from all the points - A local and global information combination structure - Two joint alignment networks that align both input points and point features.	
	Symmetric function	In order to make a model invariant to input permutation, three strategies exist: 1) Sort input into a canonical order - Impossible for higher dimension 2) Treat the input as a sequence to train an RNN, but augment the training data by all kinds of permutations - Proven useless for PointClouds **3) Use a simple symmetric function to aggregate the information from each point.** - This is what PointNet uses For the given input, the symmetric function takes n vectors as input and outputs a new vector that is invariant to the input order. For example, + and * operators are symmetric binary functions. PointNet approximate a general function defined on a point set by applying a symmetric function on transformed elements in the set: $f(\{x_1, \dots, x_n\}) \approx g\left(h(x_1, \dots, h(x_n)\right)$ where $f: 2^{\mathbb{R}^N} \to \mathbb{R}$, $h: \mathbb{R}^N \to \mathbb{R}^K$ and $g: \underbrace{\mathbb{R}^K \times \cdots \times \mathbb{R}^K}_{n} \to \mathbb{R}$ is a symmestric function. h is approximated by a multi-layer perceptron network and g by a composition of a single variable function and a max pooling function. With a collection of h, the network can learn a number of f's **to capture different properties** of a set. This is called the **Global signature** of the input (point cloud descriptors).	
	Local and Global Information Aggregation	A classification of the pointCloud can be done based on the Global signature, but as the name imply, the descriptor is only global. In order to make segmentation, we also need local information about the geometry. The solution can be seen on figure 2 (segmentation network). After computing the global point cloud feature vector, it is fed back to per point features by concatenating the global feature with each of the point features. Then the network extract new per point features based on the combined point features- this time the per point feature is aware of both the local and global information.	
	Joint Alignment Network	The semantic labeling of a point cloud has to be invariant if the point cloud undergoes certain geometric transformations, such as rigid transformation. In order to handle these transformation, PointNet predict an affine transformation matrix by a mini-network (T-net in Fig 2) and directly apply this transformation to the coordinates of input points. The mini-network itself resembles the big network and is composed by basic modules of point independent feature extraction, max pooling and fully connected layers. The idea is extended to the alignment of feature space as well - as seen on the Figure. However, transformation matrix in the feature space has much higher dimension than the spatial transform matrix, which greatly increases the difficulty of optimization. A regularization term is therefore added to the Softmax training loss. The feature transformation matrix is constrained to be close to orthogonal matrix: $L_{reg} = \Vert I - AA^T \Vert_F^2$ where A is the feature alignment matrix predicted by a mini-network. An orthogonal transformation will not lose information in the input, thus is desired.	