Computer Vision
Visual Odometry and SLAM

dsai.asia

Asia Data Science and Artificial Intelligence Master's Program

DS&AI

Co-funded by the
Erasmus+ Programme
of the European Union

Outline

Tracking means doing recursive state estimation of a system over time using discrete, noisy observations.

Examples of system states we might want to track:

1. $x, y, z, \dot{x}, \dot{y}, \dot{z}$ of a hostile airplane or missile
2. $x, y, z, \phi, \rho, \theta$ of a camera on a moving robot
3. $x, y, \theta, \dot{x}, \dot{y}, \dot{\theta}$ of a robot on a football field
4. 27 degrees of freedom of a model of the human hand
5. x, y, z positions of Mercury, Venus, Earth, Mars, Jupiter, Saturn, Neptune, Pluto

Examples of noisy measurements helpful for state estimation:

1. Angular rotation of a radar rig when the missile is detected
2. Robot-relative pitch and yaw of 3 known landmarks
3. x, y, θ of the robot measured from an image
4. Center of gravity and fingertip positions in an image of the hand
5. Pitch and yaw of the planets at particular times, relative to Kepler's house in Prague

Many methods exist, but the Kalman filter (Kalman, 1960) is the most widely used.

We'll use the following notation in these lecture notes:

- Both probability densities and discrete probability distributions are written $P(\cdot)$.
- Vectors are written in bold face (e.g. $\mathbf{x}$) and scalars are written in italics (e.g. x).
- Random variables are written in uppercase (e.g. $P(\mathrm{X}) = \mathcal{N}(\boldsymbol{\mu}, \Sigma)$).
- Particular values of a random variable are written in lowercase (e.g. $P(\mathrm{X} = \mathrm{x} \mid \mathrm{Y} = \mathrm{y}) \propto e^{-\frac{1}{2}(\mathrm{x}-\mathrm{y})\Sigma^{-1}(\mathrm{x}-\mathrm{y})}$).
- $P(\mathrm{x})$ is shorthand for $P(\mathrm{X} = \mathrm{x})$.

Outline

Kalman filter assumptions

- Time is discrete.
- The state vector x_{t+1} at time $t+1$ is a linear function of the state x_t at time t, with noise added.
- We have a linear sensor producing at each time t a noisy observation of the system's state.

Then tracking means estimating, at every time t, the mean $\boldsymbol{\mu}_t$ and covariance Σ_t of a d-dimensional Gaussian distribution $P(X_t \mid z_{1:t})$.

Kalman filtering works in two steps:

1. Prediction of the state x_{t+1} using $\boldsymbol{\mu}_t$ and the transition model.
2. Updating the state estimate $\boldsymbol{\mu}_{t+1}, \Sigma_{t+1}$ using the prediction and the observation z_{t+1}.

Suppose we have a state estimate $P(X_t \mid z_{1:t})$ for time t and a transition model $P(X_{t+1} \mid x_t)$.

Conditioning on x_t, we can then predict the state at time $t+1$ as

$$P(X_{t+1} \mid z_{1:t}) = \int_{x_t} P(X_{t+1} \mid x_t) P(x_t | z_{1:t}) dx_t$$

If $P(X_{t+1} \mid x_t)$ and $P(X_t | z_{1:t})$ are both Gaussian, we have a convolution of two Gaussians, which will always be another Gaussian.

If we ignore the observations and just predict forward in time, we begin with the prior $P(X_0) = \mathcal{N}(\boldsymbol{\mu}_0, \Sigma_0)$ then predicting forward we always have a spreading Gaussian state distribution.

Consider a random walk of a continuous state variable X_t and a noisy observation Z_t.

Perhaps you track your roommate's "happiness" using a happiness survey or facial expression measurement sensor administered once per day, converted into a scalar index. Without knowing the underlying factors influencing happiness, we simply assume a random walk.

We begin with the prior distribution over X_0, i.e.,

$$P(x_0) = \alpha e^{-\frac{(x_0-\mu_0)^2}{2\sigma_0^2}}$$

The transition model just adds a random perturbance with variance σ_x^2:

$$P(x_{t+1} \mid x_t) = \alpha e^{-\frac{(x_{t+1}-x_t)^2}{2\sigma_x^2}}$$

Now consider what happens when we predict X_1 from $P(X_0)$:

$$\begin{aligned} P(x_1) &= \int_{-\infty}^{\infty} P(x_1|x_0)P(x_0)dx_0 \\ &= \alpha \int_{-\infty}^{\infty} e^{-\frac{(x_1-x_0)^2}{2\sigma_x^2}} e^{-\frac{(x_0-\mu_0)^2}{2\sigma_0^2}} dx_0 \\ &= \alpha \int_{-\infty}^{\infty} e^{-\frac{\sigma_0^2(x_1-x_0)^2+\sigma_x^2(x_0-\mu_0)^2}{2\sigma_0^2\sigma_x^2}} dx_0. \end{aligned}$$

Note that the numerator inside the exponential is quadatic in x_0. We can complete the square using the equation

$$ax_0^2 + bx_0 + c = a\left(x_0 - \frac{-b}{2a}\right)^2 + c - \frac{b^2}{4a}$$

giving

$$P(x_1) = \alpha e^{-\frac{1}{2}\left(c - \frac{b^2}{4a}\right)} \int_{-\infty}^{\infty} e^{-\frac{a}{2}\left(x_0 - \frac{-b}{2a}\right)^2} dx_0.$$

but the intergral is now over the full range of a Gaussian, which is constant, so we finally obtain, after simplification,

$$P(x_1) = \alpha e^{-\frac{1}{2}\left(\frac{(x_1 - \mu_0)^2}{\sigma_0^2 + \sigma_x^2}\right)}.$$

We have a new Gaussian with mean μ_0 and variance $\sigma_0^2 + \sigma_x^2$.

After computing the prediction $P(X_1)$, we need to update, incorporating the observation z_1.

Now we have (using Bayes' rule)

$$\begin{aligned} P(x_1 \mid z_1) &= \alpha P(z_1 \mid x_1) P(x_1) \\ &= \alpha e^{-\frac{(z_1 - x_1)^2}{2\sigma_z^2}} e^{-\frac{(x_1 - \mu_0)^2}{2(\sigma_0^2 + \sigma_x^2)}} \end{aligned}$$

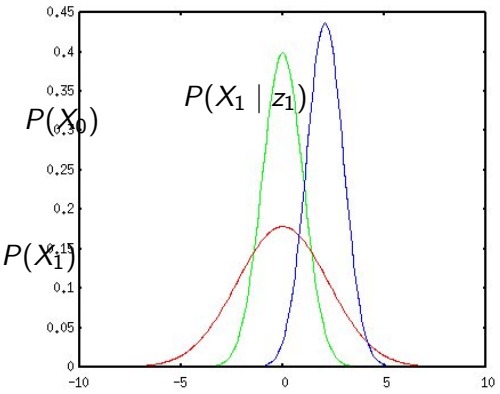
Completing the square and simplifying, we can obtain a Gaussian with mean and variance

$$\begin{aligned} \mu_{t+1} &= \frac{(\sigma_t^2 + \sigma_x^2) z_{t+1} + \sigma_z^2 \mu_t}{\sigma_t^2 + \sigma_x^2 + \sigma_z^2} \\ \sigma_{t+1}^2 &= \frac{(\sigma_t^2 + \sigma_x^2)\sigma_z^2}{\sigma_t^2 + \sigma_x^2 + \sigma_z^2} \end{aligned}$$

What happens when $\sigma_0 = 1.0$, $\sigma_x = 2.0$, $\sigma_z = 1.0$, and $z_1 = 2.5$?

$P(X_1)$ (the prediction) is a Gaussian with mean 0 and variance $\sigma_0^2 + \sigma_x^2$.

$P(X_1 \mid z_1)$ (the updated state estimate) is a Gaussian with mean $\frac{(\sigma_0^2+\sigma_x^2)z_1+\sigma_z^2\mu_0}{\sigma_0^2+\sigma_x^2+\sigma_z^2} = 2.08$ and variance $\frac{(\sigma_0^2+\sigma_x^2)\sigma_z^2}{\sigma_0^2+\sigma_x^2+\sigma_z^2} = 0.83$



Observations about the 1D Kalman update

- After predicting we are less confident in our state estimate.
- When the observation arrives, our new estimate is a weighted mean between the observation and the prediction.
- The variance σ_t does not depend on z, so if σ_z and σ_x never change, σ_t will converge to a fixed value over time.

In general we have a multivariate Gaussian

$$\mathcal{N}(\boldsymbol{\mu}, \Sigma)(\mathrm{x}) = \alpha e^{-\frac{1}{2}(\mathrm{x}-\boldsymbol{\mu})^T \Sigma^{-1}(\mathrm{x}-\boldsymbol{\mu})}$$

Transition model

$$P(\mathrm{x}_{t+1} \mid \mathrm{x}_t) = \mathcal{N}(\mathrm{F}\mathrm{x}_t, \Sigma_x)(\mathrm{x}_{t+1})$$

Here F is a matrix describing the (linear) relationship between x_t and x_{t+1}. Σ_x is the covariance of the transition noise.

Sensor model

$$P(\mathrm{z}_t \mid \mathrm{x}_t) = \mathcal{N}(\mathrm{H}\mathrm{x}_t, \Sigma_z)(\mathrm{z}_t)$$

Here H describes the (linear) relationship between the state x_t and observation z_t. Σ_z is the covariance of the observation noise.

When we obtain a new observation z_{t+1} at time $t+1$, we update our estimate of the mean and covariance of the state.

Update of the mean

$$\boldsymbol{\mu}_{t+1} = \mathrm{F}\boldsymbol{\mu}_t + \mathrm{K}_{t+1}(z_{t+1} - \mathrm{HF}\boldsymbol{\mu}_t)$$

Update of the covariance

$$\Sigma_{t+1} = (\mathrm{I} - \mathrm{K}_{t+1})(\mathrm{F}\Sigma_t\mathrm{F}^T + \Sigma_x)$$

Here K_{t+1} is the Kalman gain at time $t+1$.

$$\mathrm{K}_{t+1} = (\mathrm{F}\Sigma_t\mathrm{F}^T + \Sigma_x)\mathrm{H}^T(\mathrm{H}(\mathrm{F}\Sigma_t\mathrm{F}^T + \Sigma_x)\mathrm{H}^T + \Sigma_z)^{-1}$$

When working with Kalman filters you have to get used to casting your problem in the right form.

Consider a particle moving in one dimension with random acceleration.

$$X_{t+1} = X_t + \dot{X}_t \Delta_t$$
$$\dot{X}_{t+1} = \dot{X}_t + \ddot{X}_t \Delta_t$$
$$\ddot{X}_{t+1} = \lambda \ddot{X}_t + W_t$$

Where the particle at time t has position x_t, velocity $\dot{x}_t$, and acceleration $\ddot{x}_t$. λ is a damping factor. $W_t \sim \mathcal{N}(0, \sigma_a^2)$.

At each time t, we obtain a noisy observation of the particle's position.

$$Z_t = X_t + V_t$$

Where $V_t \sim \mathcal{N}(0, \sigma_z^2)$. Try to write the transition and sensor models.

How to know if your system or sensor is linear or nonlinear?

Well, write them down as mathematical functions.

Consider a particle moving in the plane with linear velocity that is fluctuating randomly:

$$\mathbf{x}_t = \begin{bmatrix} x \\ y \\ \dot{x} \\ \dot{y} \end{bmatrix} \quad \mathbf{x}_{t+1} = \begin{bmatrix} x + \dot{x}\Delta_t \\ y + \dot{y}\Delta_t \\ \dot{x} + w_x \\ \dot{y} + w_y \end{bmatrix} \quad \begin{matrix} w_x \sim \mathcal{N}(0, \sigma_x^2) \\ w_y \sim \mathcal{N}(0, \sigma_y^2) \end{matrix}$$

Try converting this to the form $\mathbf{x}_{t+1} = \mathbf{F}\mathbf{x}_t + \mathbf{w}$.

Compare the previous system to a non-holonomic vehicle moving in the plane with a turning angle and speed that are fluctuating randomly:

$$\mathrm{x}_t = \begin{bmatrix} x \\ y \\ \theta \\ s \\ \dot{\theta} \end{bmatrix} \quad \mathrm{x}_{t+1} = \begin{bmatrix} x + cos(\theta)d_x - sin(\theta)d_y \\ y + sin(\theta)d_x + cos(\theta)d_y \\ \theta + \dot{\theta}\Delta_t \\ s + w_s \\ \dot{\theta} + w_\theta \end{bmatrix}$$

$$d_x = \frac{s}{\dot{\theta}}(\cos\dot{\theta}\Delta_t - 1)$$
$$d_y = \frac{s}{\dot{\theta}}\sin\dot{\theta}\Delta_t$$
$$w_s \sim \mathcal{N}(0, \sigma_s^2)$$
$$w_\theta \sim \mathcal{N}(0, \sigma_\theta^2)$$

Try converting this one to the form $\mathrm{x}_{t+1} = \mathrm{F}\mathrm{x}_t + \mathrm{w}$!

The same analysis holds for the sensor. If we have a GPS device:

$$\mathbf{x}_t = \begin{bmatrix} x \\ y \\ \dot{x} \\ \dot{y} \end{bmatrix} \quad \mathbf{z}_t = \begin{bmatrix} x \\ y \end{bmatrix} \quad \text{vs.} \quad \mathbf{x}_t = \begin{bmatrix} x \\ y \\ \theta \\ s \\ \dot{\theta} \end{bmatrix} \quad \mathbf{z}_t = \begin{bmatrix} x \\ y \end{bmatrix}$$

But if we're measuring bearing and distance to a landmark:

$$\mathbf{x}_t = \begin{bmatrix} x \\ y \\ \dot{x} \\ \dot{y} \end{bmatrix} \quad \mathbf{z}_t = \begin{bmatrix} \tan^{-1} \frac{y_l - y}{x_l - x} \\ \sqrt{(x_l - x)^2 + (y_l - y)^2} \end{bmatrix} \quad \text{vs.} \quad \mathbf{x}_t = \begin{bmatrix} x \\ y \\ \theta \\ s \\ \dot{\theta} \end{bmatrix} \quad \mathbf{z}_t = \begin{bmatrix} ? \\ ? \end{bmatrix}$$

Outline

In vision and robotics, transition models and sensor models are usually nonlinear. What to do?

The extended Kalman filter (EKF) approximates the nonlinear sensor and transition models with linear functions.

Example EKF application

We consider a point moving in 3D with some velocity observed by two cameras with known calibration.

Let the system state be $(x, y, z, \phi, \theta, v)^T$:

- (x, y, z) is the position of the object in $\mathbb{R}^3$.
- ϕ is the pitch (inclination from the $x - z$ plane) of the object. We assume $-\frac{\pi}{2} \leq \phi \leq \frac{\pi}{2}$.
- θ is the yaw (rotation in the $x - z$ plane) of the object. We assume $0 \leq \theta \leq 2\pi$.
- v is the object's velocity. We assume $0 \leq v \leq v_{max}$.

Transition model

$$\mathrm{f}(\mathrm{x}) = \begin{pmatrix} x + \Delta_t \cdot v \cdot \cos(\phi)\cos(\theta) \\ y + \Delta_t \cdot v \cdot \sin(\phi) \\ z + \Delta_t \cdot v \cdot \cos(\phi)\sin(\theta) \\ \phi \\ \theta \\ v \end{pmatrix} \text{ plus Gaussian noise}$$

The sensor is two cameras with known calibration observing the position $(x, y, z)^T$ of the object. If image coordinates are rectified and centered so that

$$K = \begin{bmatrix} -f & 0 & 0 \\ 0 & -f & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

we have image points $K(R_1(x, y, z)^T + T_1)$ for camera 1 and $K(R_2(x, y, z)^T + T_2)$ for camera 2.

EKF example: sensor model

Now let $T_i = (t_{ix}, t_{iy}, t_{iz})^T$ and $R_i = (\mathrm{r}_{i1}, \mathrm{r}_{i2}, \mathrm{r}_{i3})^T$. We can write

Sensor model

$$\mathrm{h}(\mathrm{x}) = \begin{pmatrix} -f(\mathrm{r}_{11} \cdot (x,y,z)^T + t_{1x})/(\mathrm{r}_{13} \cdot (x,y,z)^T + t_{1z}) \\ -f(\mathrm{r}_{12} \cdot (x,y,z)^T + t_{1y})/(\mathrm{r}_{13} \cdot (x,y,z)^T + t_{1z}) \\ -f(\mathrm{r}_{21} \cdot (x,y,z)^T + t_{2x})/(\mathrm{r}_{23} \cdot (x,y,z)^T + t_{2z}) \\ -f(\mathrm{r}_{22} \cdot (x,y,z)^T + t_{2y})/(\mathrm{r}_{23} \cdot (x,y,z)^T + t_{2z}) \end{pmatrix}$$

plus Gaussian noise

The system noise needs to account for unknown changes in the velocity, pitch, and yaw of our object in a given period of time.
For simplicity, let's assume

$$\Sigma_x = \Delta_t^2 \begin{bmatrix} \sigma_x^2 & 0 & 0 & 0 & 0 & 0 \\ 0 & \sigma_x^2 & 0 & 0 & 0 & 0 \\ 0 & 0 & \sigma_x^2 & 0 & 0 & 0 \\ 0 & 0 & 0 & \sigma_\phi^2 & 0 & 0 \\ 0 & 0 & 0 & 0 & \sigma_\theta^2 & 0 \\ 0 & 0 & 0 & 0 & 0 & \sigma_v^2 \end{bmatrix}$$

with, say, $\sigma_x = 0.1, \sigma_\phi = \pi, \sigma_\theta = \pi, \sigma_v = 0.5, \Delta_t = 1/30$.

We could be more accurate by calculating Σ_x on each iteration, taking into account the current estimated state of the system and the expected acceleration.

For the sensor noise, let's assume an isotropic Gaussian:

$$\Sigma_z = \begin{bmatrix} \sigma_z^2 & 0 & 0 & 0 \\ 0 & \sigma_z^2 & 0 & 0 \\ 0 & 0 & \sigma_z^2 & 0 \\ 0 & 0 & 0 & \sigma_z^2 \end{bmatrix}$$

with, say, $\sigma_z = 2$.

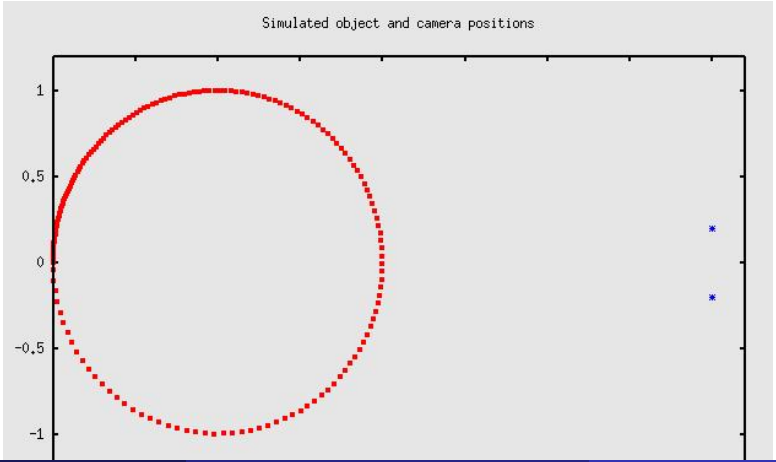
We could be more accurate here by taking the effects of radial distortion, the size of the object, or other factors into account.

Now our system is fully specified except for initial conditions. Let's assume for simplicity $\boldsymbol{\mu}_0 = (0, 0, 0, 0, 0, 0)^T$ and $\Sigma_0 = \text{diag}(0.1^2, 0.1^2, 0.1^2, (10/180 \cdot \pi)^2, (10/180 \cdot \pi)^2, 0.1^2)$

EKF pseudocode

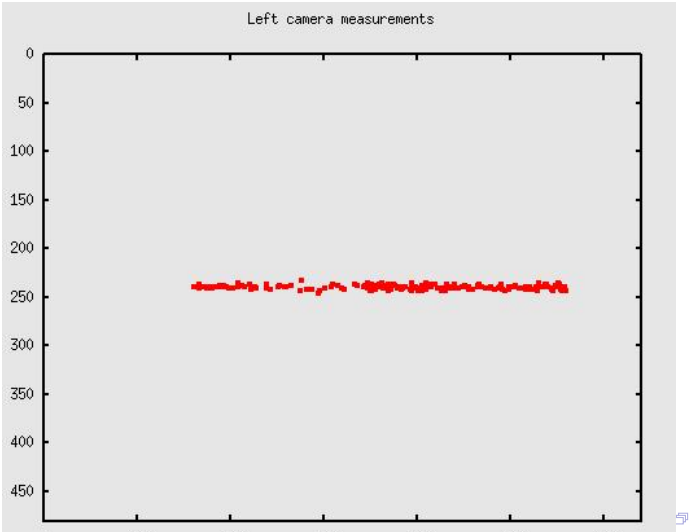
$\boldsymbol{\mu}_t = \boldsymbol{\mu}_0, \Sigma_t = \Sigma_0, t = 1$
While $t \leq N$
$\quad \mathrm{x}_{pred} = \mathrm{f}(\boldsymbol{\mu}_t)$
$\quad \mathrm{z}_{pred} = \mathrm{h}(\mathrm{x}_{pred})$
$\quad J_f =$ the Jacobian of $\mathrm{f}(\cdot)$ evaluated at x_{pred}
$\quad J_h =$ the Jacobian of $\mathrm{h}(\cdot)$ evaluated at x_{pred}
$\quad \Sigma_{pred} = J_f \Sigma_t J_f^T + \Sigma_x$
$\quad \boldsymbol{\delta}_z = \mathrm{z}_t - \mathrm{z}_{pred}$
$\quad K = \Sigma_{pred} J_h^T (J_h \Sigma_{pred} J_h^T + \Sigma_z)^{-1}$
$\quad \Sigma_t = (I - K J_h^T) \Sigma_{pred}$
$\quad \mu_t = \mathrm{x}_{pred} + K \boldsymbol{\delta}_z$
$\quad t = t + 1.$

Suppose our object actually travels in a circle of radius 1m in the $x - z$ plane with constant acceleration.



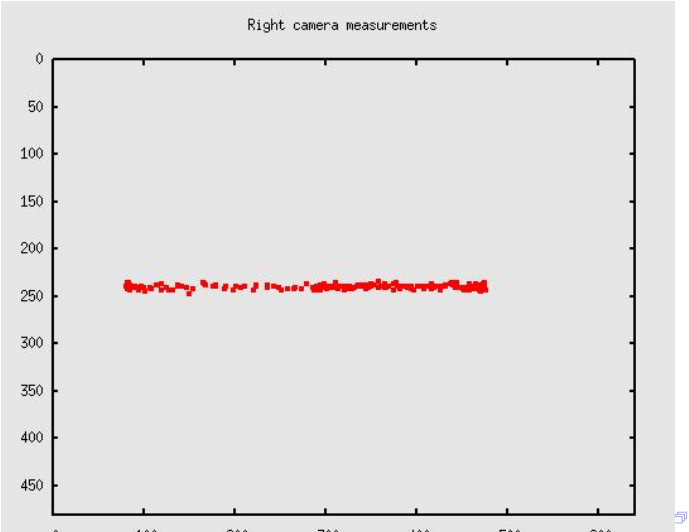
Extended Kalman filter

EKF example: simulation



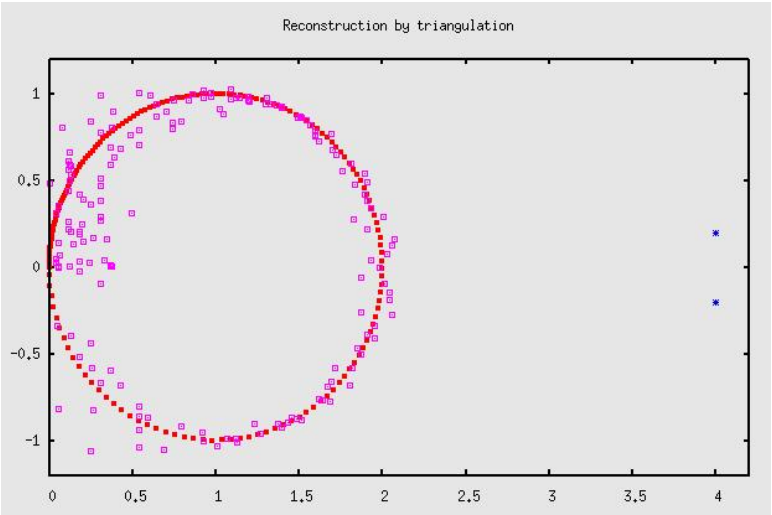
Extended Kalman filter

EKF example: simulation



Extended Kalman filter

EKF example: simulation



Let $\mathrm{f}(\mathrm{x}) = (f_1(\mathrm{x}), f_2(\mathrm{x}), \ldots, f_p(\mathrm{x}))^T$.

$$\mathrm{J}_f = \begin{bmatrix} \nabla f_1(\mathrm{x})^T \\ \nabla f_2(\mathrm{x})^T \\ \ldots \\ \nabla f_p(\mathrm{x})^T \end{bmatrix}$$

$$\nabla f_1(\mathrm{x}) = \begin{pmatrix} 1 \\ 0 \\ 0 \\ -\Delta_t v \cos(\theta) \sin(\phi) \\ -\Delta_t v \cos(\phi) \sin(\theta) \\ \Delta_t \cos(\phi) \cos(\theta) \end{pmatrix}$$

$\nabla f_2(\mathrm{x}) = ??$

$\ldots$

$\nabla h_1(\mathrm{x}) = ??$

$\ldots$

[SHOW DEMO]

Outline

As sensors for mobile robot perception, cameras are

- cheap
- small
- lightweight
- rich in information

http://www.psychol.cam.ac.uk

Vision facilitates interaction with a world of objects built for things that have eyes.

Even the most trivial intelligent behavior often requires visual perception.

When Stanford AI researchers began developing robots, planning was thought to be the exciting problem and perception was at first thought to be necessary, but easy and uninteresting.

Vision is effortless for us but planning sequential actions requires conscious effort. Shouldn't it be the same for machines?

In fact, AI made fast progress in planning but excruciatingly slow progress in machine vision.

[Show Shakey video]

In this talk we consider two problems for mobile robots:

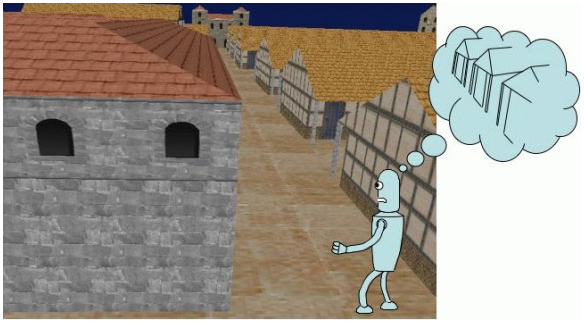
- Localization: knowing where you are
- Mapping: building a model of what (obstacles, objects to be manipulated) is around you

Localization is (relatively) easy when we have an a priori map (build a Kalman filter or particle filter over $SE(3)$).

Mapping is (relatively) easy when we have perfect knowledge of the robot's locations.

If we don't have an a priori map, we have to build the map as we navigate. This is called SLAM...

In simultaneous localization and mapping (SLAM), a robot must map its environment while navigating through that environment.



Noise, in both the perceptual sensors and the positioning sensors, conspires to make the task doubly difficult.

Some important applications requiring mapping and navigation:

- Intelligent vehicles
- In-home assistance for the elderly and disabled
- Medical tele-presence

Google driverless car (2012)

Satoh and Sakaue

More applications:

- Agricultural automation
- Search and rescue
- Surveillance and security
- Home automation

Husqvarna automower (2007)

Vision Robotics Corp. (2007)

Rescue Robot, NIST (2007)

Outline

Brief history of the most important inference tools:

- 1990: Stochastic map (Smith, Self, and Cheeseman)
- 1997: Pose graphs (Lu and Milios)
- 1998: Condensation (Isard and Blake)
- 1999: Rao-Blackwellized particle filter (RBPF, Murphy)

First, let's look at SLAM from a formal point of view. We have a sequence

$$z_0, u_1, z_1, \ldots, u_t, z_t, \ldots, u_T, z_T,$$

where z_t are observations from our sensors and u_t represent controls we send to the robot (wheel encoder or accelerometer integration from $t-1$ to t).

Our task is to estimate, at each point t in time, a posterior distribution

$$p(s_t, \Theta \mid z_{0:t}, u_{1:t})$$

over the current robot location s_t and map Θ.

Note that the best estimates of the robot's position and map as of time t would simply be the s_t and Θ that maximize $p(s_t, \Theta \mid z_{0:t}, u_{1:t})$.

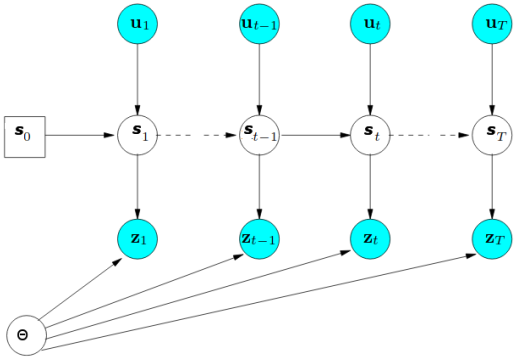
Note also that if the map is given and we only need to localize, we simply have the special case of estimating/maximizing

$$p(s_t \mid \Theta, z_{0:t}, u_{1:t}),$$

and if the positions are given and we only need to map, we simply have the special case of estimating/maximizing

$$p(\Theta \mid s_{1:t} z_{0:t}).$$

When the dynamical system's state dynamics are Markovian and the observations are conditionally independent given the state, we obtain the dynamic Bayesian network



How does this help? Try factoring and simplifying the joint distribution $p(\mathsf{s}_0, \ldots, \mathsf{s}_T)$.

In most cases, we want an up-to-date estimate of the posterior at each point t in time.

We can then recursively estimate the state using the Bayes filter equation

$$p(\mathsf{s}_t, \Theta \mid \mathsf{z}_{0:t}, \mathsf{u}_{1:t}) = \eta\, p(\mathsf{z}_t \mid \mathsf{s}_t, \boldsymbol{\theta}_{n_t}) \int p(\mathsf{s}_t \mid \mathsf{s}_{t-1}, \mathsf{u}_t)\, p(\mathsf{s}_{t-1}, \Theta \mid \mathsf{z}_{0:t-1}, \mathsf{u}_{1:t-1})\, d\mathsf{s}_{t-1}.$$

The three densities on the right hand side of the equation are the sensor model, the motion model, and the posterior estimate from time $t-1$.

For clarity, suppose there is only one landmark observed at each point in time (the generalization to multiple observations is straightforward).

Then the (usually nonlinear) dynamical system can be written

$$\begin{aligned} z_t &= g(\theta_{n_t}, s_t) + \epsilon_t \text{ (the sensor model)} \\ s_t &= h(s_{t-1}, u_t) + \delta_t \text{ (the motion model)}, \end{aligned}$$

where n_t represents the index of the element θ_{n_t} of Θ observed at time t, and ϵ_t and δ_t represent the sensor noise and odometry noise at time t.

If the sensor noise and motion noise are assumed Gaussian and we linearize g() and h(), we obtain the extended Kalman filter or stochastic map approach proposed by Smith, Self, and Cheeseman (1990).

But consider how the state vector grows over time.

When the noise is not well-approximated by Gaussians: we can represent the posterior by a discrete set of samples or particles.

Particle filter approaches to SLAM differ on how the map Θ is estimated:

- Some methods simply maintain a single maximum likelihood map estimate;
- Others use the Rao-Blackewellised particle filter (RBPF, Murphy, 1999), which maintains a different map estimate for each sampled robot path.

The RBPF is optimal, but maintaining many separate estimates of Θ is expensive.

The map Θ is normally represented as a 2D or 3D occupancy grid (Elfes, 1987) or a set of landmarks (geometric primitives such as points, lines, or surfaces, possibly annotated with appearance information).

"Smarter" maps containing surface models or pose and identities of relevant objects are also possible but less explored.

If we're talking about visual sensors, we haven't gotten far beyond the 3D point cloud representation.

The RBPF has been found to work well with 2D grids and high-level landmark lists but not with 3D grids or 3D point clouds.

We now vision-based SLAM specifically, where we need a 3D grid or 3D point cloud.

Outline

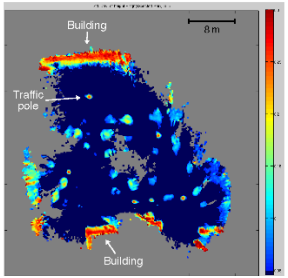
A prototype autonomous vacuum cleaner built at Vision Robotics Corp. in the early 2000s used

- Stereo vision
- A 3D occupancy grid
- A particle filter for robot position with the maximum likelihood map estimate.

[SHOW VIDEO]

Vision Robotics Corp. (2004)

There was some success in outdoor terrain mapping with 2D occupancy grids for Mars rovers with stereo vision (Se et al., 2006; Marks et al., 2008).



Marks, Howard, Bajracharya, Cottrell, and Matthies (2008)

Benefits of the occupancy grid approach:

- Sensor model is straightforward
- Data association is simple
- Robust to noisy observations
- Good for visualization

Disadvantages of occupancy grids:

- 2D grids are only sufficient for simple scenes
- Detailed 3D grids are impractical for large-scale scenes
- Absence of evidence is not always evidence of absence: 3D grids tend to provide noisy wireframe reconstructions of the scene.

In 2020, most state-of-the-art methods use the pose graph approach or a variant.

Outline

Pose graph approaches

Sparse vs. dense

We saw that occupancy grids give a dense representation of the scene but have not seen much success in 3D visual mapping.

There are alternative approaches to dense mapping, for example doing direct keyframe alignment and generating a separate point cloud for each keyframe (see, e.g., LSD-SLAM).

Sparse approaches, however, represent the map as an explicit list of discrete landmarks.

Sparsity allows us to trade off accuracy against time (number of landmarks) to obtain real-time computation on embedded systems.

Modern successful sparse approaches are all based on the pose graph approach.

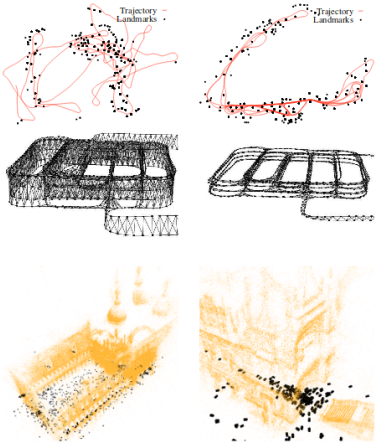
There are many ways of setting up the pose graph, depending on details of the robot, sensors, and approach to the optimization.

Basic idea:

- Each pose s_t is represented by one vertex of the pose graph.
- Each landmark X_i is represented by one vertex of the pose graph.
- Edges connecting s_{t-1} and s_t are annotated with corresponding controls u_t as constraints.
- Edges connecting s_t and X_i represent observations z_{tj}.

Pose graph approaches

Pose graphs and graph-based optimization



Kümmerle et al. (2011), Fig. 1

Victoria Park dataset with unoptimized trajectory and optimized trajectory.

Multi-level parking garage pose graph before and after optimization.

Two views of Venice dataset after optimization.

To be estimated:

- For pose vertices, we want to estimate the homogeneous transformation from the world frame to the body frame at time t

$$\mathtt{T}_t = \begin{bmatrix} \mathtt{R}_t & \mathtt{t}_t \\ 0^\top & 1 \end{bmatrix} \in SE(3)$$

  or, equivalently, $\mathtt{s}_t = \exp(\mathtt{T}_t) \in \mathfrak{se}(3)$.
- For landmark vertices, assuming a point cloud, we want to estimate $\mathtt{X}_i \in \mathbb{R}^3$.

A Lie group G comprises a smooth differentiable manifold and a group.

In computer vision, for the group, we only care about real matrix groups represented in $\mathbb{R}_{n \times n}$.

The multiplication and inversion operations are the usual matrix multiplication and matrix inversion operations.

The group is represented by a specific sublass of $n \times n$ matrices, so there are fewer than n^2 degrees of freedom.

Elements on the differential manifold are represented by vectors in $\mathbb{R}^k$, the number of degrees of freedom in the group.

The exp() function maps members of the group to points on the manifold, and the log() function does the reverse.

See `https://arxiv.org/pdf/1812.01537.pdf` (*A Micro Lie Theory for State Estimation in Robotics*, Sola et al., 2019) for a good summary of Lie theory for robotics.

The most difficult-to-understand Lie algebra concept used in robotics is the representatio of rotations.

Consider $\mathtt{R}(t)$ varying over time. Clearly, $\dot{\mathtt{R}} = \frac{\partial \mathtt{R}}{\partial t}$ will always lie in the space tangent to the manifold of valid rotation matrices.

Rotations:

Performing optimization over the Lie group for 3D rotations $\mathfrak{so}(3)$ or 3D rigid motion $\mathfrak{se}(3)$ gives a minimal 3-element representation of rotations without the gimbal lock problem present in Euler angle representations.

Gimbal locked airplane. When the pitch (green) and yaw (magenta) gimbals become aligned, changes to roll (blue) and yaw apply the same rotation to the airplane.

From Wikipedia, *Gimbal Lock*

Graph optimization reduces the global optimization to a series of small local optimizations.

To find a maximum likelihood estimate of s_t holding all other variables constant, we can assume Gaussian errors and find T_t minimizing

- The deviation $\|u_t - \exp(T_t T_{t-1}^{-1})\|_{\Sigma_u}$
- The deviation $\|z_t - \pi(\phi_t(X), T_t)\|_{\Sigma_z}$, where $\phi_t(\cdot)$ selects the landmarks from the global landmark database X that are observed at time t, and $\pi(\cdot, \cdot)$ projects a set of 3D points into the image plane.

We begin with the guess $T_t = \log(u_t)T_{t-1}$ then use Levenberg-Marquardt to find the local minimum.

Estimating X_j is similar, with an initial guess from triangulation from two views.

g2o (Kuemmerle et al., 2011), is probably the most famous toolbox for implementing graph optimization.

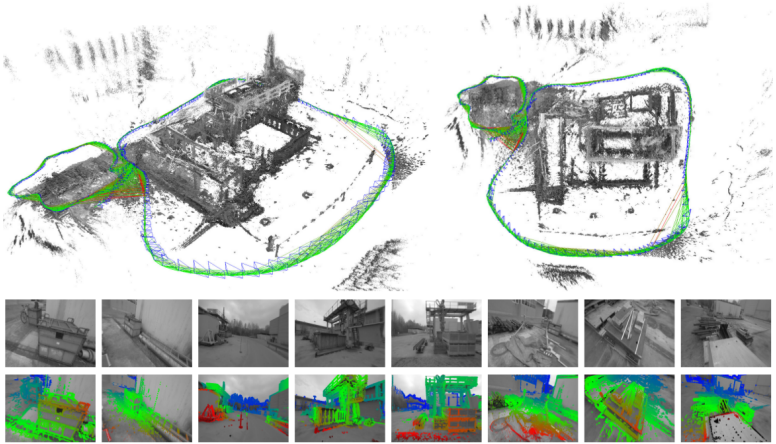
A clean API is used to add vertices and edges run the optimization, and retrieve the resulting estimates.

The optimizer is very lightweight and efficient, yet it is general, working for many different optimization problems, not only SLAM.

`https://github.com/RainerKuemmerle/g2o`

Outline

Engel, Stückler, and Cremers' (2015) method LSD SLAM (Large-Scale Direct) is among the best modern monocolar SLAM methods.



LSD-SLAM principles:

- Attempt to use all information in the image (not just features).
- Extract a series of "keyframes" from the video stream and find relationship between each pair of camera poses using graph optimization.
- Pairwise optimization in the pose graph minimizes photometric error between two keyframes.
- Each keyframes is paired with an inverse depth map containing a mean and variance (inverse confidence) in the inverse depth at each pixel.

Most SfM methods are feature based.

They first estimate camera positions and 3D point positions using sparse feature extraction, factorization of E, resectioning, and bundle adjustment.

A dense model can then be made through mesh construction and texture mapping.

Direct methods scrunch the 2-step process, finding the camera poses and depth maps that align images directly without feature extraction.

Camera must be calibrated a priori.

All image points are converted to normalized camera coordinates (undistorting and multiplying by K^{-1}).

Instead of representing camera poses by a rotation matrix and translation vector packed into a 4×4 homography matrix, we use elements of the Lie group $\mathfrak{se}(3)$, written as 6-element vectors.

Similarity transformations, normally represented by a scalar scale factor, a rotation matrix, and a translation vector, are instead represented by elements of $\mathfrak{sim}(3)$, written as 7-element vectors.

Images are aligned by minimization of photometric error

$$E(\boldsymbol{\xi}) = \sum_i (I_{\text{ref}}(\mathrm{p}_i) - I(\omega(\mathrm{p}_i, D_{\text{ref}}(\mathrm{p}_i), \boldsymbol{\xi})))^2$$

where

- $I(\mathrm{p})$ is the intensity of image I at location p,
- $\boldsymbol{\xi}$ is a possible transformation between the cameras imaging I_{ref} and I,
- $\omega(\mathrm{p}, d, \boldsymbol{\xi})$ projects point p with inverse depth d in the reference camera's frame into the transformed camera frame,
- $D_{\text{ref}}(\mathrm{p})$ is the inverse depth of 2D point p.

The parameters of $\boldsymbol{\xi}$ can be estimated using an initial guess and Gauss-Newton minimization.

Occlusions, reflections, and moving objects would introduce outliers.

Rather than remove outliers explicitly, we re-weight each point p_i on each iteration, down-weighting large residual errors:

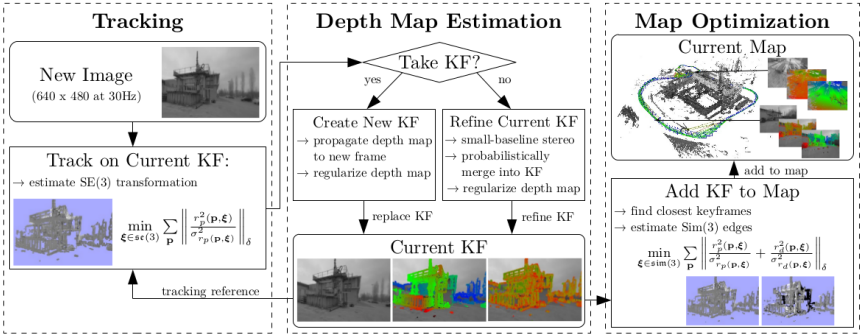
$$E(\boldsymbol{\xi}) = \sum_i w_i(\boldsymbol{\xi})(I_{\mathrm{ref}}(\mathrm{p}_i) - I(\omega(\mathrm{p}_i, D_{\mathrm{ref}}(\mathrm{p}_i), \boldsymbol{\xi})))^2.$$

Defining $r_i(\boldsymbol{\xi})$ to be the residual $I_{\mathrm{ref}}(\mathrm{p}_i) - I(\omega(\mathrm{p}_i, D_{\mathrm{ref}}(\mathrm{p}_i), \boldsymbol{\xi}))$, the Gauss-Newton update becomes

$$\delta\boldsymbol{\xi}^{(n)} = -(\mathtt{J}^T\mathtt{W}\mathtt{J})^{-1}\mathtt{J}^T\mathtt{W}r(\boldsymbol{\xi}^{(n)}),$$

where $\mathtt{J}$ is the Jacobian of the residuals r_i with respect to the changes in $\boldsymbol{\xi}$.

Overview of the processing pipeline:



Overall LSD-SLAM flow (Engel, Schöps, and Cremers, 2014)

We begin with an initial keyframe with a random depth map and a large variance.

Early translations of the camera enable covergence to a nearly correct depth map.

The map is a pose graph of keyframes.

Keyframe $\mathcal{K}_i$:

- Image $I_i : \Omega_i \to \mathbb{R}$
- Inverse depth map $D_i : \Omega_{D_i} \to \mathbb{R}^+$
- Variance of the inverse depth $V_i : \Omega_{D_i} \to \mathbb{R}^+$

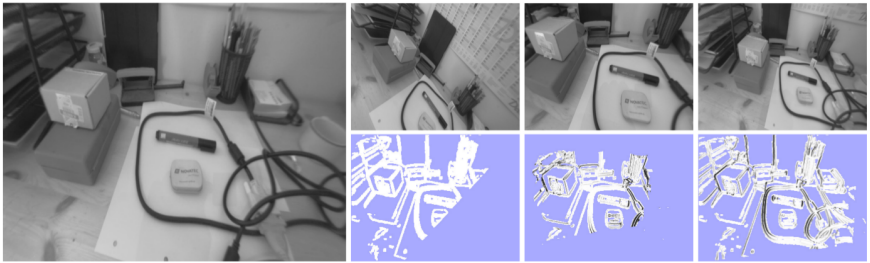
The depth map and variance are only defined for a subset of all pixels $\Omega_{D_i} \subset \Omega_i$ that have sufficient intensity gradient.

Edges ξ_{ji}:

- Similarity transform $\boldsymbol{\xi}_{ji} \in \mathfrak{sim}(3)$
- Covariance Σ_{ji} over $\boldsymbol{\xi}_{ji}$

Initial D starts out undefined for points with small gradient (purple color) and random with high variance for points with sufficient gradient.

Camera rotation gives no additional depth information; translation increases confidence/decreases variance for points with parallax under the translation.



(a) reference image (b) rotation (c) z trans. (d) x trans.

Inverse depth map over camera tranformations (Engel, Schöps, and Cremers, 2014).

From each keyframe, we perform short-term tracking, finding the relative pose $\boldsymbol{\xi}_{ji} \in \mathfrak{se}(3)$ minimizing the robust normalized photometric resiudual

$$E_p(\boldsymbol{\xi}_{ji}) = \sum_{\mathrm{p} \in \Omega_{D_i}} \left\| \frac{r_p^2(\mathrm{p}, \boldsymbol{\xi}_{ji})}{\sigma^2_{r_p(\mathrm{p}, \boldsymbol{\xi}_{ji})}} \right\|_\delta$$

where $\| \cdot \|_\delta$ is a robust norm and $\sigma^2_{r_p(\mathrm{p}, \boldsymbol{\xi}_{ji})}$ is an estimate of residual uncertainty based on depth map uncertainty $V_i(\mathrm{p})$ and assumed constant image intensity noise σ_I.

A new keyframe is created from the most recent tracked image when the local motion ξ_{ji} exceeds a threshold.

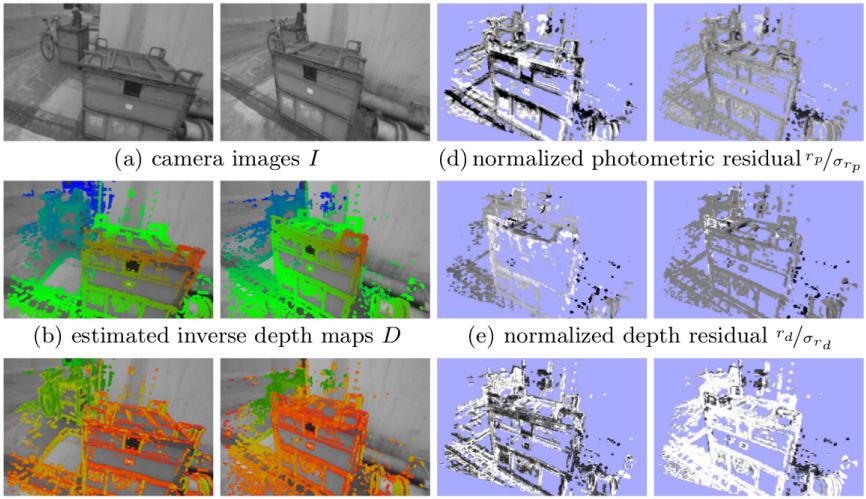
Since actual depths are unknown, the depth map for each keyframe is scaled so that the mean inverse depth is 1.

The inverse depth map for the current keyframe is updated using the motion for each new tracking frame.

Each new keyframe is aligned with the previous keyframe to minimize both photometric residual and depth residual using image alignment over $\mathfrak{sim}(3)$ — we explicitly calculate the relative depth between keyframes.

LSD SLAM

Keyframe alignment



(a) camera images I (d) normalized photometric residual r_p/σ_{r_p}

(b) estimated inverse depth maps D (e) normalized depth residual r_d/σ_{r_d}

(c) inverse depth variance V (f) robust Huber weights

Each time a new keyframe is selected, we compare with the nearest 10 keyframes looking for a loop closure: a previous keyframe close enough to the current frame to create a new edge in the pose graph.

Drift in a large loop might mean the new keyframe is too different from the best old keyframe for convergence.

There are a few tricks that help in convergence, but one of the best is a course-to-fine approach starting with downscaled 20×15 images.

Pose graph optimization runs continuously in the background.

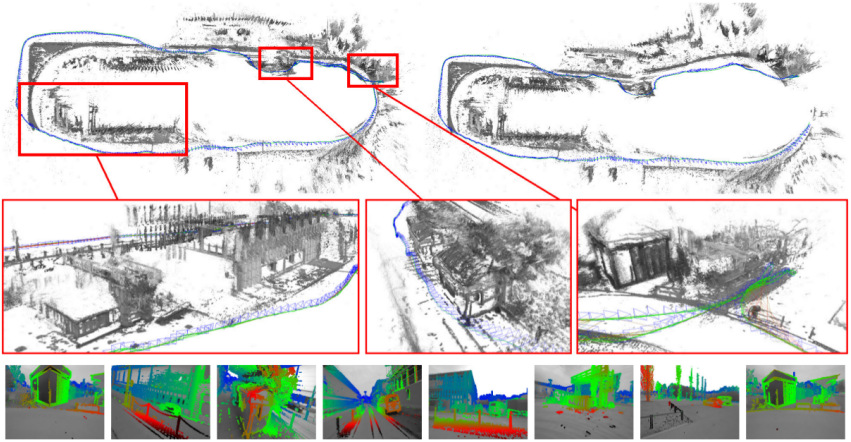


Fig. 7: Loop closure for a long and challenging outdoor trajectory (after the loop closure on the left, before on the right). Also shown are three selected close-ups of the generated pointcloud, and semi dense depth maps for selected keyframes

LSD-SLAM runs in real time on smartphones in odometry-only configurations (no large-scale map).

The full LSD-SLAM runs in real time on moderate CPUs (not requiring a GPU).

ORB-SLAM runs faster than LSD-SLAM and gives more accurate trajectories. But the resulting point clouds are sparse.

The near-term open problem is fast, accurate, dense 3D modeling from monocular cameras.

Outline

- ORB-SLAM is a visual SLAM method
- In 2016, ORB-SLAM was introduced as one of the most versatile and accurate monocular SLAM methods to date.
- ORB-SLAM is based on the main ideas of
  - Parallel tracking and mapping (PTAM) by Klein and Murray
  - Place recognition by Gálvez-López and Tardós
  - Scale-aware loop closing by Strasdat et. al
  - Covisibility information by Strasdat et. al

ORB-SLAM's aims:

- More efficient, simple and reliable system than existing visual SLAM methods.
- Real-time operation in large-environments.
- Real-time loop closing based on optimization.
- Real-time relocalization with significant invariance to viewpoint and illumination.
- Improving tracking robustness and enhance lifelong operation.

- Similar to other visual SLAM methods, ORB-SLAM needs a feature detector and descriptor to extract and match feature points from sequence of images.
- ORB-SLAM needs a feature detector and descriptor that can be used in mapping, tracking, and place recognition processes at real time.
- ORB detector and descriptor are fast enough to compute and match features while having good invariance to viewpoint.
- Therefore, ORB detector and descriptor are chosen for ORB-SLAM method.

- ORB-SLAM consists of three threads working in parallel.
  - Tracking thread
  - Mapping thread
  - Loop closing thread
- Bag of words place recognition is also an essential feature.
- The feature creates visual vocabulary from the keyframe it has been seen and stores in the database. If there are changes done to keyframes, the database is updated according to the changes.
- DBoW2 module is integrated to ORB-SLAM method for loop detection, relocalization, and keyframe culling.

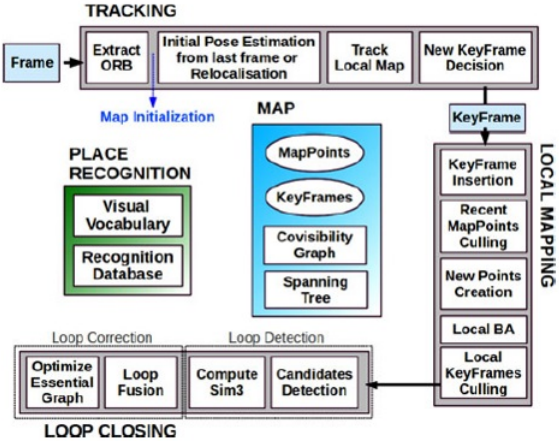


Figure: ORB-SLAM system overview

- Covisibility graph is an undirected weighted graph. Each node is a keyframe and an edge between two keyframes exists if they share observations of the same map points.
- Covisibility graph is too dense and error prone for loop closing operation. An idea of essential graph is proposed in ORB-SLAM method.
- Essential graph is a subset of covisibility graph that retains all nodes but less edges, still preserving a strong network that yields accurate results.
- Essential graph contsins spanning trees, subsets of edges from the covisibility graph with high covisibility, and the loop closure edges, resulting in a strong network of cameras.

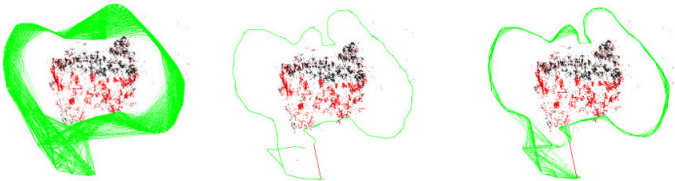


Figure: Graphs in ORB-SLAM method. a) A covisibility graph. b) A spanning tree. c) An essential graph.

- Tracking thread is responsible for the following tasks
  - Tracking
  - Map initialization
  - Relocalization when tracking is lost
  - Track local map
  - New keyframe decision

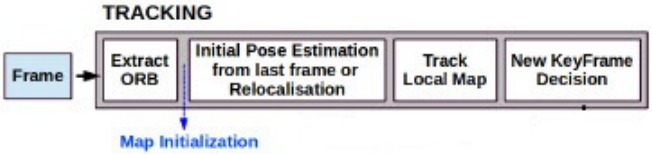


Figure: Tracking thread overview

- Tracking thread is a process that track a set of points through successive camera frames, and using these tracks to triangulate their 3D position.
- Camera pose of each frame is estimated from the relative pose of feature points between the current frame and previous frame.
- In ORB-SLAM, tracking thread extracts and uses ORB features from every images in the video sequence.

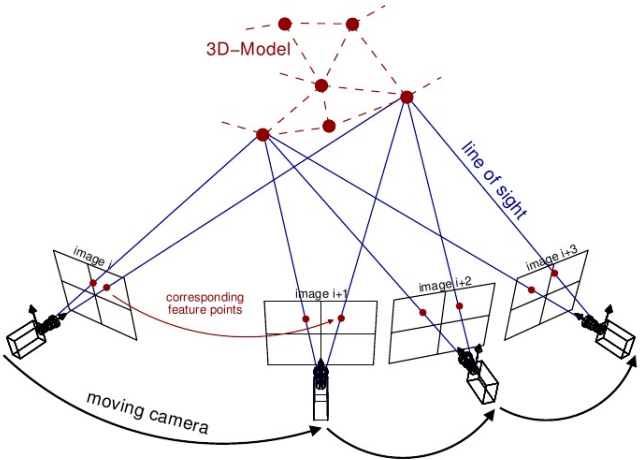


Figure: Point triangulation, from http://www.theia-sfm.org/sfm.html

- Map initialization is the first task of the tracking thread. It triangulates initial 3D correspondences with acceptable accuracy from initial image sequences.
- Matched features $x_c \leftrightarrow x_r$ between the current frame f_c and the reference frame f_r are searched to generate initial correspondences.
- Reset reference frame if not enough matches are found.

- Homography matrix H_{cr} and fundamental matrix F_{cr} are calculated. Tracking thread uses one of these matrices in map initialization process.
- Homography matrix is used if the scene is planar. Fundamental matrix is selected if the scene is nonplanar.
- The map is initialized if the selected matrix passes motion hypotheses tests and gives low reprojection error.
- Bundle adjustment is performed to refine initial correspondences and camera poses.

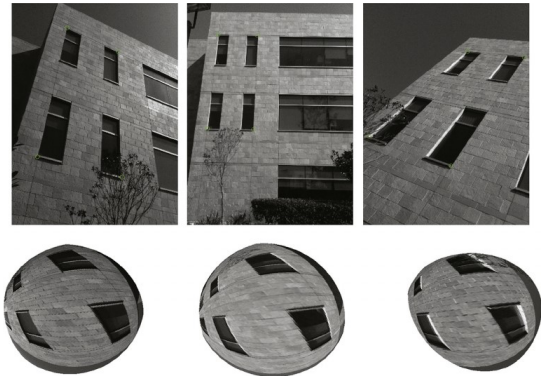


Figure: Different projections of a planar scene. from https://www.researchgate.net/figure/259128816_fig2_Fig-7-Digital-images-of-a-planar-scene-Top-prior-to-projective-frame-registration

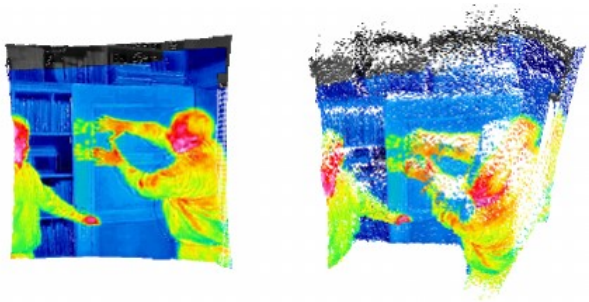


Figure: Depth maps for sample non-planar scene. from https://www.researchgate.net/figure/243459165_fig15_Figure-15-Frontal-view-onto-a-non-planar-scene-left-and-a-view-clearly-showing-the

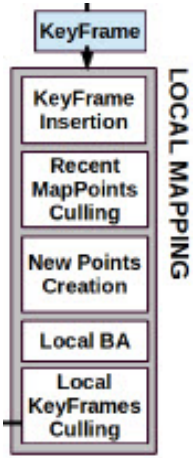
- Tracking continues the process from map initialization.
- If tracking was successful for last frame, camera pose in current frame is predicted with constant velocity motion model.
- If not enough matches were found, tracking thread performs wider search of the map points around their position in the last frame.
- The pose is optimized with the found correspondences.

- If tracking is lost, the frame is convert into bag of words and query recognition database for keyframe candidate for global relocalization.
- Correspondences with ORB associated to map points in each key frame are computed. RANSAC iterations are performed for each keyframe to find camera pose with PnP algorithm.
- If camera pose is found with enough inliers, guided search are performed to find more matches.
- The pose is then optimized and tracking procedure continues.

- Once estimated camera pose and an initial set of feature matches are obtained, the map into is projected to the frame and search more map point correspondences.
- The local map contains the set of keyframes K_1, that share map points with the current frame, and a set K_2 with neighbors to the keyframes K_1 in the covisibility graph.
- Map points seen in K_1 and K_2 are searched in the current frame. If found, these map points are added into the current frame and optimized.

- The current frame is verified as a keyframe if it meets the conditions
- The keyframe decided by the tracking tread is sent to the mapping thread to add into the map.
- Keyframes are inserted as fast as possible as it makes the tracking more robust to change in camera movements and rotations.
- Redundant keyframes will be culled in the mapping thread.

- Mapping tread is responsible for the following tasks
  - Keyframe insertion
  - Map point culling
  - New map point creation
  - Local bundle adjustment
  - Local keyframe culling



- New keyframes decided by the tracking thread are added into the map.
- Nodes and edges in the covisibiilty graph are updated.
- Spanning tree and bag of word representation are updated.

- Map points must pass a restrictive test to ensure that they are trackable and not mistakenly triangulated.
  - The tracking must find the point in more than the 25 percent of the frames in which it is predicted to be visible.
  - If more than one keyframe has passed from map point creation, it must be observed from at least three keyframes.

- For each unmatched ORB in an individual frame, unmatched correspondences are searched with other unmatched point in other keyframes.
- Newly matched map points are accepted as the new points if they pass the following tests
  - Positive depth test in both cameras.
  - Parallax test.
  - Reprojection error test.
  - Scale consistency test.

- Local BA optimizes the currently processed keyframe (K_i), all keyframes connected to K_i in the covisibility graph (K_c), and all map points seen by K_i and K_c.
- Keyframes that see map points as K_i and K_c see are included in the optimization but remain fixed.
- Outlier are discarded in this process.

- As the complexity of bundle adjustment grows with the number of keyframes, deleting redundant keyframes benefits the system in lifelong operation.
- Keyframes that has 90 percent similar map points to its three previous keyframes are removed from the map.

- Loop closing takes the last keyframe processed by the mapping thread to detect and close the loop.
- Loop closing performs the following tasks
  - Loop candidates detection.
  - Similarity transformation estimation.
  - Loop fusion.
  - Essential graph optimization.

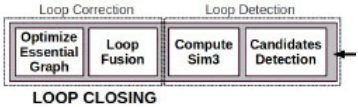


Figure: Loop closing overview.

- Similarity score is computed between the bag of word vector of the last keyframe K_i with all of its neighbors in the covisibility graph. The lowest score s_{min} is obtained.
- Recognition database is queried, keyframe whose score is lower than s_{min} are not considered as loop candidates.
- All keyframes whose are directly connected to K_i are also not considered as loop candidates.

- ORB-associated correspondences between the current keyframe and the loop candidate keyframes are calculated.
- RANSAC iterations are performed with each candidate to find a similarity transformation.
- If the similarity is found with enough inliers, the transformation is optimized and more correspondences are searched.
- After adding more correspondences, the transformation is optimized again. If the transformation is supported by enough inliers, that loop candidate is accepted.

- Once loop closing is detected and verified, two keyframes are merged together.
- The correction for this action must be done and propagated to their neighbor keyframes in covisibility graph so they can update their proterties(i.e. recalculate transformation matrix, concatenated edges in covisibility graph).

- Pose graph optimization over the Essential graph is performed to effectively close the loop.
- Scale drift is corrected, each map point is transformed according to the correction.

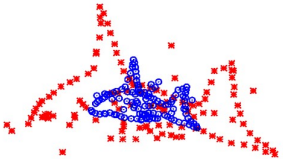


Figure: Scale drift.

Figure: ORB-SLAM example: tracking.

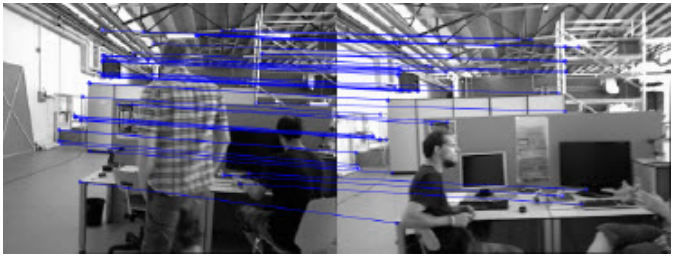


Figure: ORB-SLAM example: mapping.

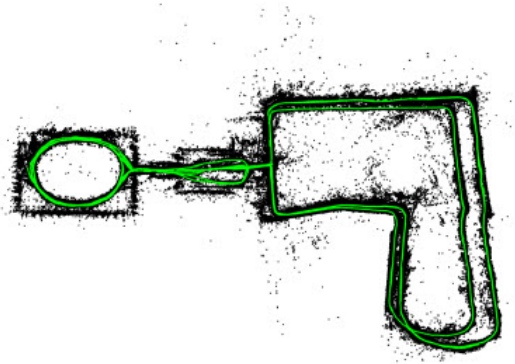


Figure: ORB-SLAM example: loop closing.

Outline

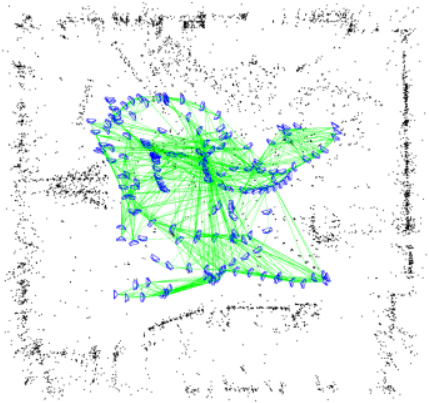
In 2017, the authors of ORB-SLAM (Raúl Mur-Artal and Juan D. Tardós) introduced "Visual-Inertial Monocular SLAM with Map Reuse."

We know that all monocular SLAM methods have the limitation of scale ambiguity.

By adding information from GPS or an IMU, we can resolve that ambiguity.

Existing work:

- IMU preintegration by Lupton and Sukkarieh (2012)
- IMU preintegration mapped to SO(3) by Forster et al. (2016)
- Factor graph representation by Indelman et al. (2013)
- ORB SLAM (2016)



Mur-Artal and Tardós (2017), Fig. 1

ORB SLAM performs tracking for the current frame using a fixed map.

On the back end, local Bundle Adjustment (BA) optimizes a local window of keyframes.

Large loops are detected using place recognition then corrected using lightweight pose graph optimization followed by full BA.

Sample result on left.

Each step (tracking and local BA) fixes some states in their optimization.

This could bias the final solution, so it is important to get a good start.

To reduce any such bias, the authors implement a VI full BA that optimizes structure, camera poses, scale, velocities, gravity, and gyroscope and accelerometer biases.

But BA always needs an initial guess. The authors propose a piece-by-piece approach to obtain this solution (next slide).

Idea of initialization:

- Process a few seconds of video with regular ORB SLAM. This gives an initial solution for structure and some keyframe poses up to unknown scale.
- Compute bias of the gyroscope based on known orientation of the keyframes.
- Solve for scale and gravity without accelerometer bias.
- Using knowledge of the magnitude of gravity, solve for accelerometer bias, refining scale and gravity direction.
- Extract velocity for each keyframe.

The main requirement is that the sensor should be moved in such a way as to make all variables observable.

The VI odometry method is based on a few key aspects.

We assume a pinhole camera. Keypoints are extracted on original image, but coordinates are undistorted before being used.

There is an IMU that works as follows:

- Acceleration a_b^k at time k in the IMU frame b.
- Angular velocity $\boldsymbol{\omega}_b^k$ at time k in the IMU frame b.
- Update frequency is on the order of 100 Hz.
- Accelerometer modeled as having slowly varying biases b_a^k and b_g^k for the accelerometer and gyroscope.
- Accelerometer is subject to gravity g_w, which must be subtracted in order to compute motion.

Goal of the system: estimate state parameters and 3D points over time.

- R_{wb}^{k} is the rotation of the IMU in the world frame at time k.
- ${}_{w}\mathrm{v}_{b}^{k}$ is the velocity in the world frame at time k.
- ${}_{w}\mathrm{p}_{b}^{k}$ is the position of the IMU in the world frame at time k.

Model is as follows:

$$\mathrm{R}_{wb}^{k+1} = \mathrm{R}_{wb}^{k} \exp\left((\boldsymbol{\omega}_{b}^{k} - \mathrm{b}_{g}^{k})\Delta_{t}\right)$$

$${}_{w}\mathrm{v}_{b}^{k+1} = {}_{w}\mathrm{v}_{b}^{k} + \mathrm{g}_{w}\Delta_{t} + \mathrm{R}_{wb}^{k}(\mathrm{a}_{b}^{k} - \mathrm{b}_{a}^{k})\Delta_{t}$$

$${}_{w}\mathrm{p}_{b}^{k+1} = {}_{w}\mathrm{p}_{b}^{k} + {}_{w}\mathrm{v}_{b}^{k}\Delta_{t} + \frac{1}{2}\mathrm{g}_{w}\Delta_{t}^{2} + \frac{1}{2}\mathrm{R}_{wb}^{k}(\mathrm{a}_{b}^{k} - \mathrm{b}_{a}^{k})\Delta_{t}^{2}$$

.

$\exp(\cdot)$ is "exponential map" of rotations, a function mapping a twist vector $\mathrm{v} \in \mathbb{R}^3$ (denoting a rotation of $\|\mathrm{v}\|$ about the unit vector $\mathrm{v}/\|\mathrm{v}\|$) to a rotation matrix.

IMU measurements between two keyframes are preintegrated based on the work of Forster et al. (2016):

$$\mathrm{R}_{wb}^{i+1} = \mathrm{R}_{wb}^{i} \Delta_{\mathrm{R}_{i,i+1}} \exp(\mathrm{J}_{\Delta R}^{g} \mathrm{b}_{g}^{i})$$

$$_{w}\mathrm{v}_{b}^{i+1} =_{w} \mathrm{v}_{b}^{i} + \mathrm{g}_{w} \Delta_{t_{i,i+1}} + \mathrm{R}_{wb}^{i} (\Delta_{\mathrm{v}_{i,i+1}} + \mathrm{J}_{\Delta_{\mathrm{v}}}^{g} \mathrm{b}_{g}^{i} + \mathrm{J}_{\Delta_{\mathrm{v}}}^{a} \mathrm{b}_{a}^{i})$$

$$_{w}\mathrm{p}_{b}^{i+1} =_{w} \mathrm{p}_{b}^{i} + \mathrm{v}_{b}^{i} \Delta_{t_{i,i+1}} + \frac{1}{2} \mathrm{g}_{w} \Delta_{t_{i,i+1}}^{2} + \mathrm{R}_{wb}^{i} \left(\Delta_{\mathrm{p}_{i,i+1}} + \mathrm{J}_{\Delta_{p}}^{g} \mathrm{b}_{g}^{i} + \mathrm{J}_{\Delta_{\mathrm{p}}}^{a} \mathrm{b}_{a}^{i} \right)$$

The idea is to separate the effect of the bias from the pre-integrated IMU measurements with a linear approximation.

The values $\Delta_{\mathbf{R}_{i,i+1}}$, $\Delta_{\mathbf{v}_{i,i+1}}$, and $\Delta_{\mathbf{p}_{i,i+1}}$ are the simple integration/concatenation of the transformations given by the IMU over the period between keyframe i and $i+1$.
The preintegrations and Jacobians are computed iteratively as the IMU measurements arrive.

Finally, the camera and IMU are considered rigidly attached with transformation

$$\mathrm{T}_{cb} = \left[\mathrm{R}_{cb} \mid {}_{c}\mathrm{p}_{b}\right]$$

between their frames.

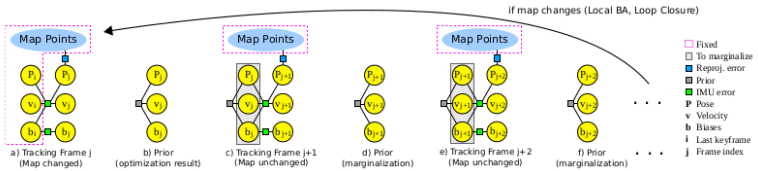
The transformation is estimated from the calibration method of Furgale et al. (2013) implemented in the open-source Kalibr tool.

As before, ORB-SLAM has 3 parallel threads: tracking, local mapping, and loop closing.

For the tracking thread:

- At each new frame, camera pose is predicted based on the IMU and the estimated pose from the previous frame.
- Visible keypoints are projected to the new frame then matched.
- Then the new frame's pose is optimized considering the matches and the IMU error.

Main idea for tracking thread:



Mur-Artal and Tardós (2017), Fig. 2.

As long as we are tracking successfully and the local mapping thread does not update 3D point locations, we continue to incrementally update the pose relative to the previous keyframe.

When the mapping thread updates the keyframe pose, we reset the current frame's pose relative to the keyframe that changed, then resume incremental updates.

[More on tracking thread]

The frame-changed optimization uses the parameter vector

$$\boldsymbol{\theta} = \left\{ \mathrm{R}^{j}_{wb}, {}^{w}\mathrm{p}^{j}_{b}, {}^{w}\mathrm{v}^{j}_{b}, \mathrm{b}^{j}_{g}, \mathrm{b}^{j}_{a} \right\}$$

$$\boldsymbol{\theta}^{*} = \underset{\boldsymbol{\theta}}{\operatorname{argmin}} \left(\sum_{k} E_{\mathrm{proj}}(k, j) + E_{\mathrm{IMU}}(i, j) \right)$$

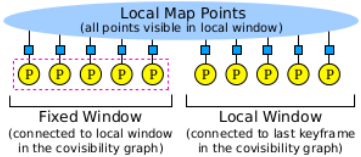
This is the sum of the reprojection error for all visible 3D points k and the IMU error from keyframe i to new frame j.

The frame-unchanged optimization uses a joint optimization of parameters for frame j and frame $j + 1$.

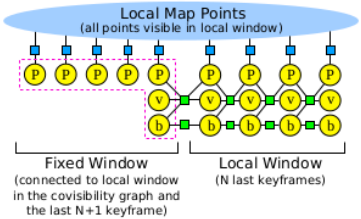
The local mapping thread performs a local BA every time a new keyframe is inserted.

An N keyframe window is optimized with all of the points observed over those keyframes (see right).

ORB-SLAM merely optimizes the poses and 3D points with reprojection error as the objective, whereas VI-ORB SLAM additionally adds IMU error terms for velocity and biases.



ORB-SLAM's Local BA



Visual-Inertial ORB-SLAM's Local BA

Mur-Artal and Tardoós (2017), Fig. 3

The loop closing thread tries to reduce drift when returning to an already-mapped area.

The place recognition module matches recent keyframes with past keyframes.

If a rigid-body transformation can be found, then an optimization is carried out over the whole trajectory.

First is a pose-graph optmization that ignores IMU data, followed by a full BA that optimizes all velocities, biases, positions, and so on.

Result: accurate, real time, sparse solution to the SLAM problem.

We learn that a monocular camera with IMU (and some sophisticated processing software) is sufficient for accurate localization and sparse point cloud mapping in real time.

No need for lasers, ultrasonic sensors, etc.!

See `https://www.youtube.com/watch?v=rdR5OR8egGI` and other examples.

See `https://github.com/jingpang/LearnVIORB` for community implementation based on the authors' ORB-SLAM 2 implementation.

Interesting project: get VI-ORB SLAM running on Android, using the smartphone IMU!

Another: compare VI-ORB-SLAM with OKVIS and SVO-2 (`https://www.youtube.com/watch?v=hR8uq1RTUfA&t=33s`).

Outline

Current related research in the AI Center at AIT:

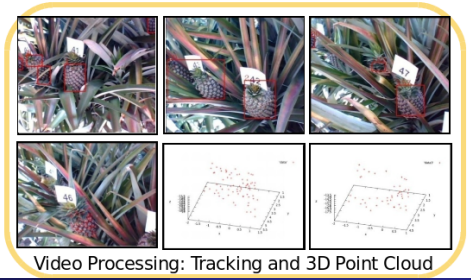
- Mobile video processing platform for agricultural crop mapping and yield prediction
- 3D virtual model construction and visualization for an agricultural field inspection robot
- Environment modeling for a security robot
- Local path planning and execution for mobile target pursuit robot
- Target tracking for mobile target pursuit
- Kinect-enabled autonomous assistant robot
- Autonomous navigation for a quad-rotor UAV with monocular camera for outdoor cluttered environments

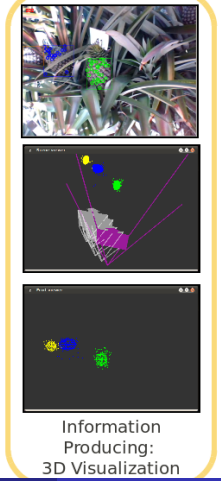
In other work, we are exploring the use of a priori maps for vision-based localization of autonomous vehicles that also have GPS and electronic compass sensors.

We said that localization from an a priori map is "easy," but the map must be in a usable form.

The idea is to combine satellite imagery with ground-based image processing for the sensor model then fuse GPS, compass, and vision data using the particle filter.

Current work in agricultural crop mapping:

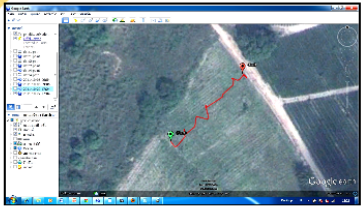




Ongoing research at AIT

Mobile video processing for agricultural crop mapping

Smartphone Platform

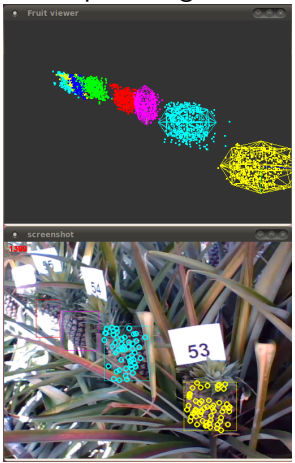


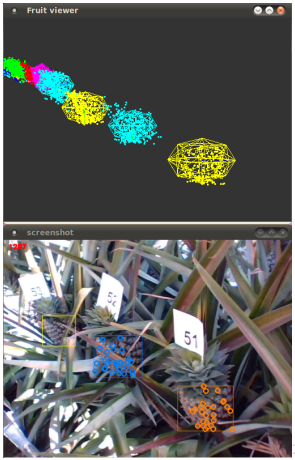
Video Processing: Raw Data

This project proposes a method for applying 3D reconstruction to an image sequence to get a 3D virtual model.

Image sequence extracted from video file taken from agricultural field

We apply keyframe selection to the video sequence, find fruit regions, and perform 3D reconstruction of the point regions.



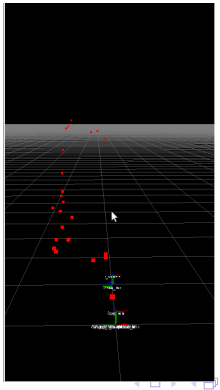


Ongoing research at AIT

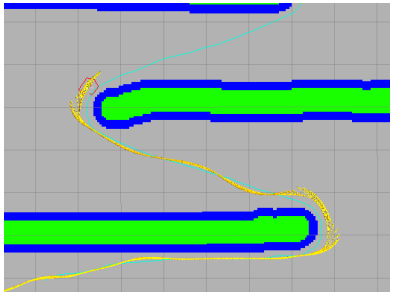
Environment modeling for security robots

We are working on 3D modeling of obstacles from a single camera for purposes of robot navigation.

Experiments on our surveillance robot use the open source ROS framework.

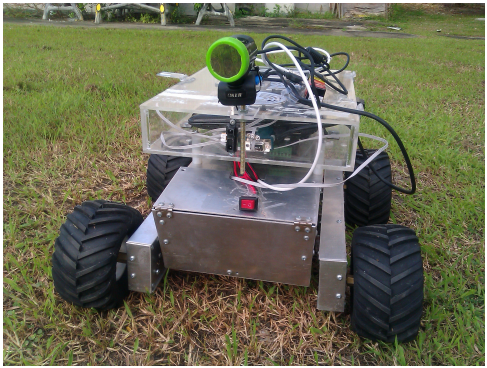


Here we are working on algorithms to plan local obstacle-avoidance trajectories in order to pursue a moving target.



Example pursuit path plan

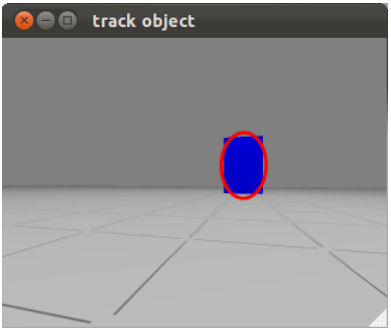
We are putting these algorithms together for pursuit of suspicious targets using a monocular camera mounted on a pursuit robot.



Pursuit robot prototype

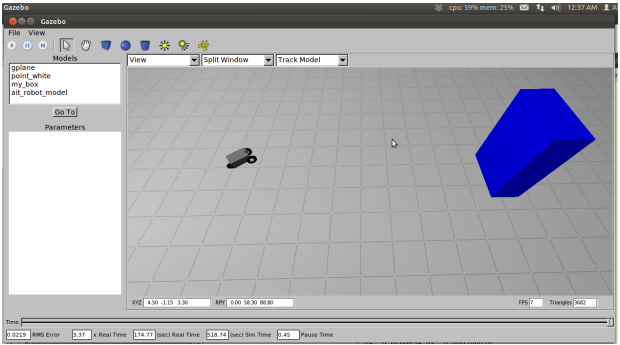
The operator manually specifies a target to track.

We track the object using color histogram tracking, a model of the robot's kinematics, a model of the target's dynamics, and an extended Kalman filter.



Tracking object from camera view in simulator

Testing is ongoing using synthetic simulations, a Gazebo simulation under ROS, and experiments on the physical pursuit robot.



Target tracking in simulator

Ongoing research at AIT

Kinect-enabled autonomous assistant robot

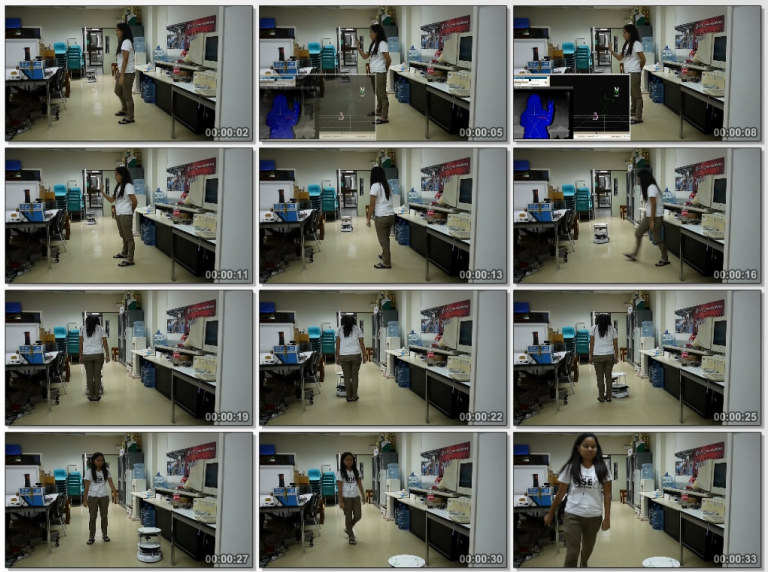
This project explores the use of the Willow Garage Turtlebot and Microsoft Kinect to carry documents in an office or orders in a restaurant.

Users can interact with the robot via speech, hand gestures, or a Web interface.

The Robot Operating System (ROS) is the main framework.

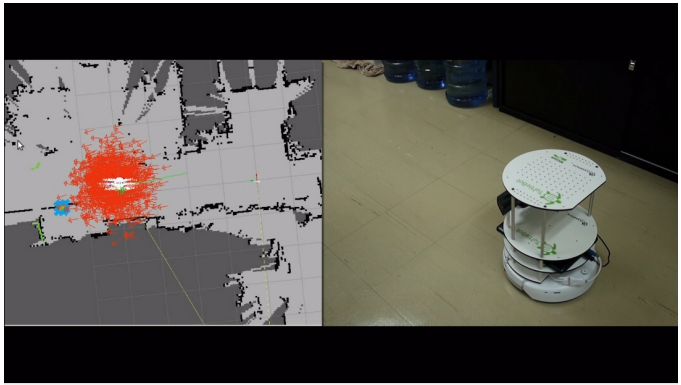
The Web interface part uses the `rosbridge` tool to handle message passing between ROS and JavaScript programs running in the Web browser over a Web socket.

Kinect-enabled autonomous assistant robot



Ongoing research at AIT

Kinect-enabled autonomous assistant robot



Localization using Kinect and a 2D obstacle map

This project uses a quad-copter with a low-resolution camera to gather videos of agricultural fields for inspection purposes.

Objectives:

- To obtain high-resolution image of a 3D scene using low quality, low-resolution monocular camera mounted on the quad-copter.
- To improve the quality of 3D-reconstruction by selecting key frames constructed using super-resolution.

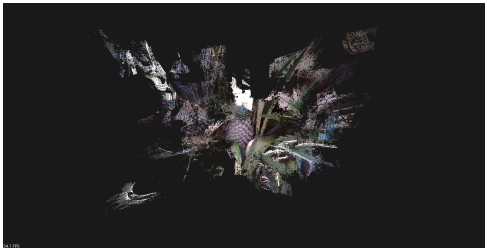
Quad-copter

Image taken from quad-copter

Ongoing research at AIT

Autonomous navigation for a quadrotor UAV

- We are developing simultaneous real-time navigation based on feature tracking and non-real-time 3D reconstruction.
- Eventually the work will enable detection of weed and pest infestation using the high-resolution frames constructed by super-resolution.



3D reconstruction

Outline

Conclusion

Vision sensors are information-rich, cheap, and lightweight.

Using vision sensors in practical mobile robot applications requires solutions to challenging AI problems.

A few of the open problems:

- Incremental localization and mapping algorithms are not yet as accurate as batch structure-from-motion algorithms in traditional computer vision.
- The higher level geometry of complex scenes is still not as accurate as we would like.
- Current methods with current sensors and embedded processors are prone to lose track when motion is too large or scene texture is too weak.
- What about control? Can we really use these tools to build obstacle maps in real time and feed the information to the control loop?

The big challenge for AI approaches to visual perception

It is difficult to impose high-level top-down constraints (e.g. that walls and floors are normally planar and orthogonal) on bottom-up statistical inference algorithms.

The field needs methods for combining high-level knowledge representation with low-level statistical inference!