Single-camera SLAM using the SceneLib library

Paul Smith

Robotics Research Group Talk, Nov 2005

Outline

The Camera as a Position Sensor

Aim

- Use a camera as a position sensor

Challenges

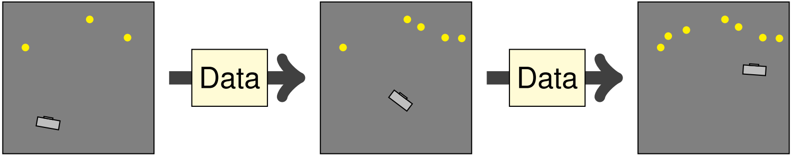
- Monocular (no depth)
- Unconstrained
- High acceleration & large rotations
- Usually want real-time localisation

Off-line Processing

Structure from motion

- Typical Computer vision approach
- Bundle adjustment over a long sequence
- Applied to post-production, 3D model reconstruction.

Sequential Real-time Processing



Simultaneous Localisation and Mapping (SLAM)

- Typical robotics approach.
- Building a long-term map by propagating and correcting uncertainty
- Mostly used in simplified 2D environments with specialised sensors such as laser range-finders.

Classic Approaches to Visual SLAM

Davison, ICCV 2003

- Traditional SLAM approach (Extended Kalman Filter)
- Maintains full camera and feature covariance
- Limited to Gaussian uncertainty only

Nistér, ICCV 2003

- Structure-from-motion approach (Preemptive RANSAC)
- Frame-to-frame motion only
- Drift: No repeatable localisation

Classic Approaches to Visual SLAM

Davison, ICCV 2003

- Traditional SLAM approach (Extended Kalman Filter)
- Maintains full camera and feature covariance
- Limited to Gaussian uncertainty only

Nistér, ICCV 2003

- Structure-from-motion approach (Preemptive RANSAC)
- Frame-to-frame motion only
- Drift: No repeatable localisation

Recent Approaches to Visual SLAM

Pupilli & Calway, BMVC 2005

- Traditional SLAM approach (Particle Filter)
- Greater robustness: handles multi-modal cases
- New features not rigorously initialised

Eade & Drummond, 2006

- FastSLAM approach (Particle Filter/Kalman Filter)
- Particle per camera hypothesis, Kalman filter for features
- Allows larger maps: update $O(M \log N)$ instead of $O(N^2)$

Recent Approaches to Visual SLAM

Pupilli & Calway, BMVC 2005

- Traditional SLAM approach (Particle Filter)
- Greater robustness: handles multi-modal cases
- New features not rigorously initialised

Eade & Drummond, 2006

- FastSLAM approach (Particle Filter/Kalman Filter)
- Particle per camera hypothesis, Kalman filter for features
- Allows larger maps: update $O(M \log N)$ instead of $O(N^2)$

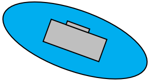
Outline

Davison's MonoSLAM: Overview

Main features

- Initialisation with known target
- Extended Kalman Filter
  - 'Constant velocity' motion model
  - Image patch features with Active Search
- Automatic Map Measurement
- Particle filter for initialisation of new features

Nomenclature



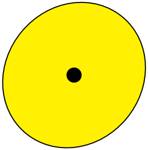
The camera

- The camera position state $\boldsymbol{x}_p$ is its 3D position and orientation

$$\boldsymbol{x}_p = \begin{pmatrix} \boldsymbol{r}^W \\ \boldsymbol{q}^{WR} \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \\ q_0 \\ q_x \\ q_y \\ q_z \end{pmatrix}$$

- The camera state $\boldsymbol{x}_v$ contains $\boldsymbol{x}_p$ plus optional additional state information (e.g. velocity and angular velocity)

Nomenclature



Features

- Each feature $\boldsymbol{y}_i$ is a 3D position vector

$$\boldsymbol{y}_i = \begin{pmatrix} x_i \\ y_i \\ z_i \end{pmatrix}$$

Nomenclature

System State

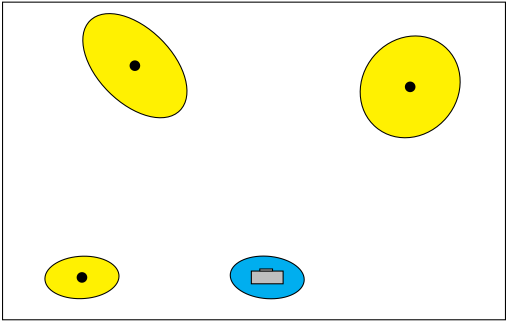
$$\boldsymbol{x} = \begin{pmatrix} \boldsymbol{x}_v \\ \boldsymbol{y}_1 \\ \boldsymbol{y}_2 \\ \vdots \end{pmatrix} \qquad \mathbf{P} = \begin{bmatrix} \mathbf{P}_{xx} & \mathbf{P}_{xy_1} & \mathbf{P}_{xy_2} & \cdots \\ \mathbf{P}_{y_1x} & \mathbf{P}_{y_1y_1} & \mathbf{P}_{y_1y_2} & \cdots \\ \mathbf{P}_{y_2x} & \mathbf{P}_{y_2y_1} & \mathbf{P}_{y_2y_2} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

- PDF over camera and feature state is modelled as a single multi-variate Gaussian and we can use the Extended Kalman Filter.

Extended Kalman Filter: Prediction Step

Time Update

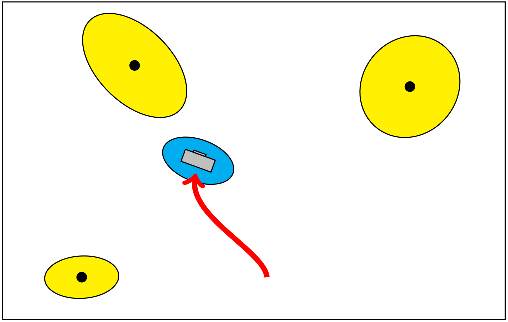
1. Estimate new location
2. Add process noise



Extended Kalman Filter: Prediction Step

Time Update

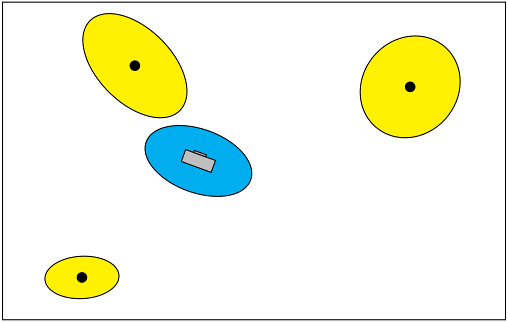
1. Estimate new location
2. Add process noise



Extended Kalman Filter: Prediction Step

Time Update

1. Estimate new location
2. Add process noise



Extended Kalman Filter: Prediction Step

Time Update

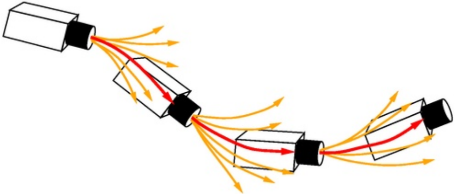
1. Project the state ahead

$$\hat{\boldsymbol{x}}_{\text{new}} = \boldsymbol{f}(\hat{\boldsymbol{x}}, \boldsymbol{u})$$

2. Project the error covariance ahead

$$\mathbf{P}_{\text{new}} = \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}} \mathbf{P} \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}^{T} + \mathbf{Q}$$

Extended Kalman Filter: Motion models



Constant velocity

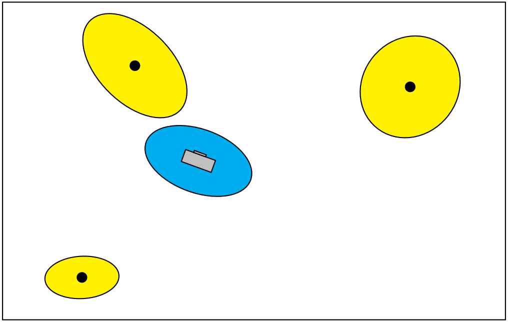
Assume bounded, Gaussian-distributed linear and angular acceleration

$$\boldsymbol{x}_{\text{new}} = \begin{pmatrix} \boldsymbol{r}^W_{\text{new}} \\ \boldsymbol{q}^{WR}_{\text{new}} \\ \boldsymbol{v}^W_{\text{new}} \\ \omega^R_{\text{new}} \end{pmatrix} = \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{u}) = \begin{pmatrix} \boldsymbol{r}^W + \left(\boldsymbol{v}^W + \boldsymbol{V}^W\right)\Delta t \\ \boldsymbol{q}^{WR} \times \boldsymbol{q}\left(\left(\omega^R + \Omega^R\right)\Delta t\right) \\ \boldsymbol{v}^W + \boldsymbol{V}^W \\ \omega^R + \Omega^R \end{pmatrix}$$

Extended Kalman Filter: Update Step

Measurement Update

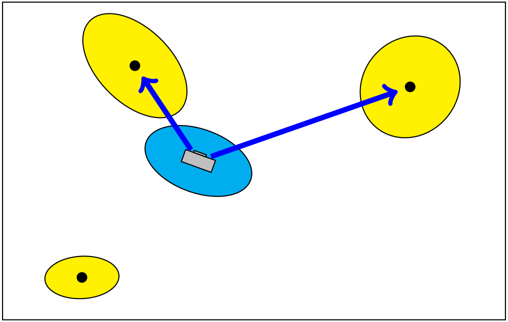
1. Measure feature(s)
2. Update positions and uncertainties



Extended Kalman Filter: Update Step

Measurement Update

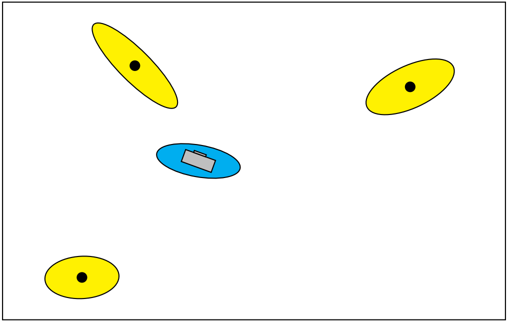
1. Measure feature(s)
2. Update positions and uncertainties



Extended Kalman Filter: Update Step

Measurement Update

1. Measure feature(s)
2. Update positions and uncertainties



Extended Kalman Filter: Update Step

Measurement Update

1. Make measurement $\boldsymbol{z}$ to give the innovation $\boldsymbol{\nu}$

$$\boldsymbol{\nu} = \boldsymbol{z} - \boldsymbol{h}(\hat{\boldsymbol{x}})$$

2. Calculate innovation covariance **S** and Kalman gain **W**

$$\mathtt{S} = \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{x}} \mathtt{P} \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{x}}^T + \mathtt{R}$$

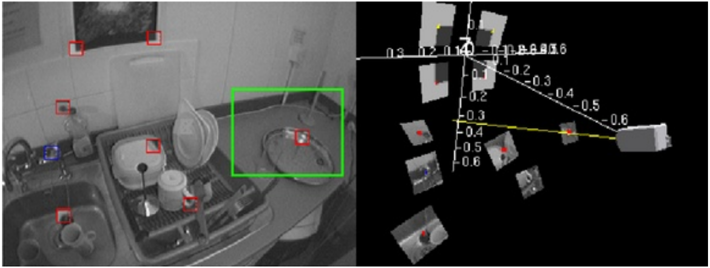
$$\mathtt{W} = \mathtt{P} \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{x}}^T \mathtt{S}^{-1}$$

3. Update estimate and error covariance

$$\hat{\boldsymbol{x}}_{\text{new}} = \hat{\boldsymbol{x}} + \mathtt{W}\boldsymbol{\nu}$$

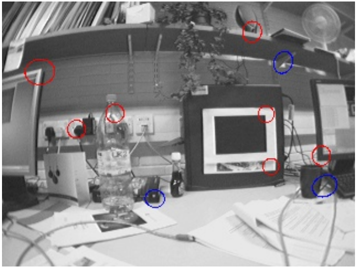
$$\mathtt{P}_{\text{new}} = \mathtt{P} - \mathtt{W}\mathtt{S}\mathtt{W}^T$$

Measurement Step: Image Features and the Map



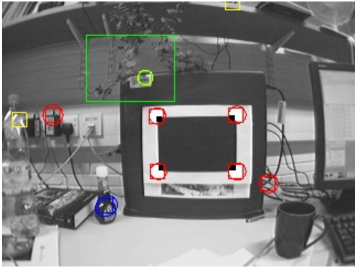
- Feature measurements are the locations of salient image patches.
- Patches are detected once to serve as long-term visual landmarks.
- Sparse set of landmarks gradually accumulated and stored indefinitely.

Measurement Step: Active Search



- Active search within elliptical search regions defined by the feature innovation covariance.
- Template matching via exhaustive correlation search.

Automatic Map Management

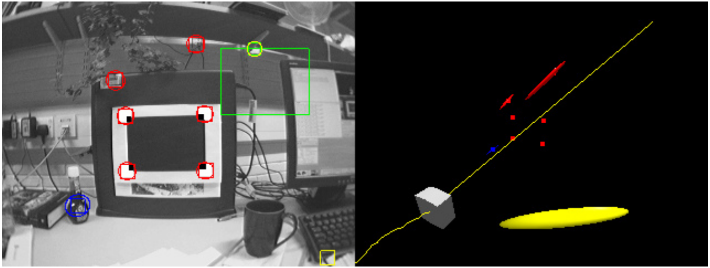


- Initialise system from a few known features.
- Add a new feature if number of visible features drops below a threshold (e.g. 12).
- Choose salient image patch from a search box in an underpopulated part of the image.

Monocular Feature Initialisation with Depth Particles

A new feature has unknown depth

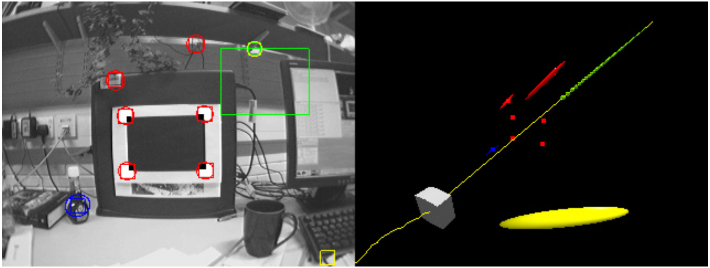
1. Populate the line with 100 particles, spaced uniformly between 0.5m and 5m from the camera.
2. Match each particle in successive frames to find probability of that depth.
3. When depth covariance is small,



Monocular Feature Initialisation with Depth Particles

A new feature has unknown depth

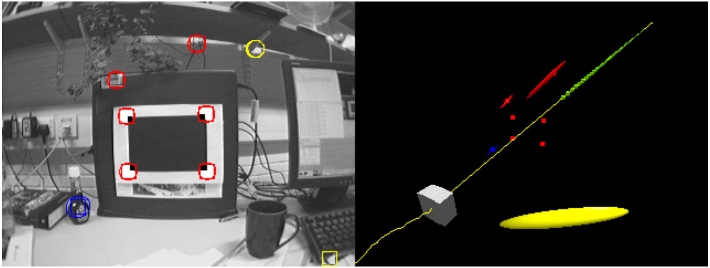
1. Populate the line with 100 particles, spaced uniformly between 0.5m and 5m from the camera.
2. Match each particle in successive frames to find probability of that depth.
3. When depth covariance is small,



Monocular Feature Initialisation with Depth Particles

A new feature has unknown depth

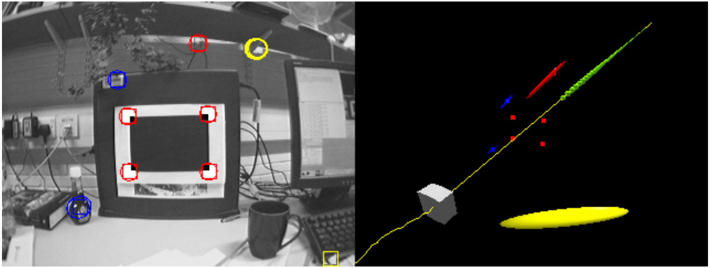
1. Populate the line with 100 particles, spaced uniformly between 0.5m and 5m from the camera.
2. Match each particle in successive frames to find probability of that depth.
3. When depth covariance is small,



Monocular Feature Initialisation with Depth Particles

A new feature has unknown depth

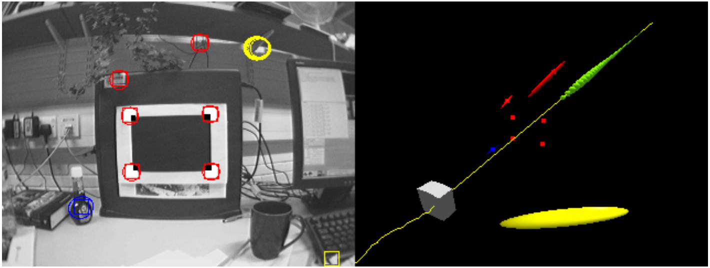
1. Populate the line with 100 particles, spaced uniformly between 0.5m and 5m from the camera.
2. Match each particle in successive frames to find probability of that depth.
3. When depth covariance is small,



Monocular Feature Initialisation with Depth Particles

A new feature has unknown depth

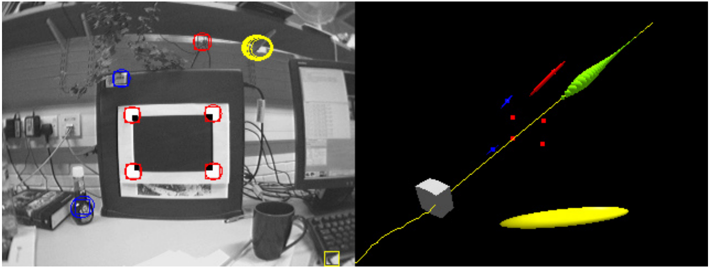
1. Populate the line with 100 particles, spaced uniformly between 0.5m and 5m from the camera.
2. Match each particle in successive frames to find probability of that depth.
3. When depth covariance is small,



Monocular Feature Initialisation with Depth Particles

A new feature has unknown depth

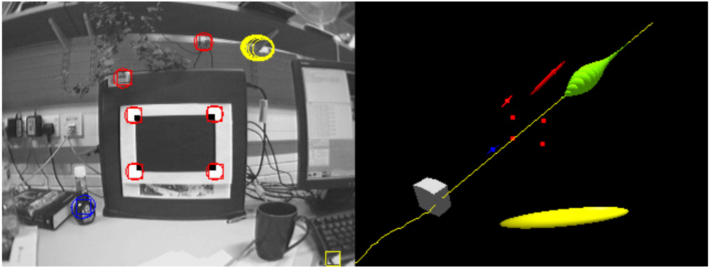
1. Populate the line with 100 particles, spaced uniformly between 0.5m and 5m from the camera.
2. Match each particle in successive frames to find probability of that depth.
3. When depth covariance is small,



Monocular Feature Initialisation with Depth Particles

A new feature has unknown depth

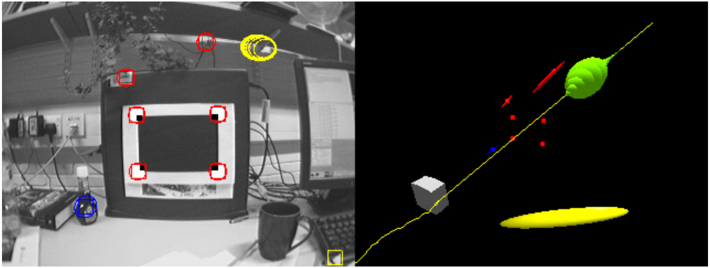
1. Populate the line with 100 particles, spaced uniformly between 0.5m and 5m from the camera.
2. Match each particle in successive frames to find probability of that depth.
3. When depth covariance is small,



Monocular Feature Initialisation with Depth Particles

A new feature has unknown depth

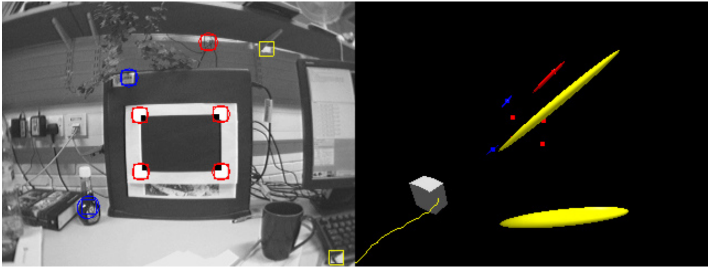
1. Populate the line with 100 particles, spaced uniformly between 0.5m and 5m from the camera.
2. Match each particle in successive frames to find probability of that depth.
3. When depth covariance is small,



Monocular Feature Initialisation with Depth Particles

A new feature has unknown depth

1. Populate the line with 100 particles, spaced uniformly between 0.5m and 5m from the camera.
2. Match each particle in successive frames to find probability of that depth.
3. When depth covariance is small, convert to Gaussian.



Feature initialisation

- New features need adding to the state vector and covariance matrix.

Increasing the state size dynamically

$$\mathbf{x}_{\text{new}} = \begin{pmatrix} \mathbf{x}_v \\ \mathbf{y}_1 \\ \mathbf{y}_2 \\ \mathbf{y}_i \end{pmatrix}$$

$$\mathbf{P} = \begin{bmatrix} \mathbf{P}_{xx} & \mathbf{P}_{xy_1} & \mathbf{P}_{xy_2} & \mathbf{P}_{xx}\frac{\partial \mathbf{y}_i}{\partial \mathbf{x}_v}^T \\ \mathbf{P}_{y_1x} & \mathbf{P}_{y_1y_1} & \mathbf{P}_{y_1y_2} & \mathbf{P}_{y_1x}\frac{\partial \mathbf{y}_i}{\partial \mathbf{x}_v}^T \\ \mathbf{P}_{y_2x} & \mathbf{P}_{y_2y_1} & \mathbf{P}_{y_2y_2} & \mathbf{P}_{y_2x}\frac{\partial \mathbf{y}_i}{\partial \mathbf{x}_v}^T \\ \frac{\partial \mathbf{y}_i}{\partial \mathbf{x}_v}\mathbf{P}_{xx} & \frac{\partial \mathbf{y}_i}{\partial \mathbf{x}_v}\mathbf{P}_{xy_1} & \frac{\partial \mathbf{y}_i}{\partial \mathbf{x}_v}\mathbf{P}_{xy_2} & \frac{\partial \mathbf{y}_i}{\partial \mathbf{x}_v}\mathbf{P}_{xx}\frac{\partial \mathbf{y}_i}{\partial \mathbf{x}_v}^T + \frac{\partial \mathbf{y}_i}{\partial \mathbf{h}_G}\mathbf{R}_L\frac{\partial \mathbf{y}_i}{\partial \mathbf{h}_G}^T \end{bmatrix}$$

Feature deletion

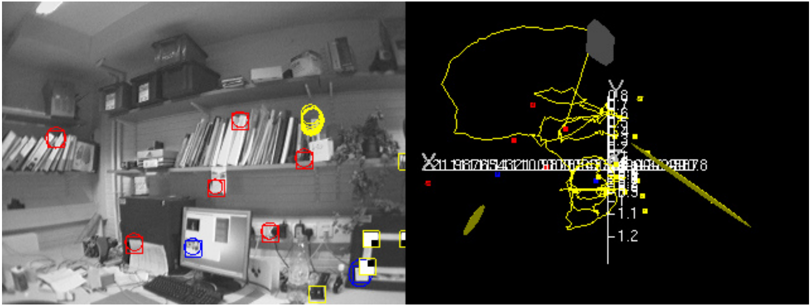
- Delete a feature if more than half of attempted measurements fail.

Reducing the state size dynamically

$$\begin{pmatrix} \mathbf{x}_v \\ \mathbf{y}_1 \\ \mathbf{y}_2 \\ \mathbf{y}_3 \end{pmatrix} \rightarrow \begin{pmatrix} \mathbf{x}_v \\ \mathbf{y}_1 \\ \mathbf{y}_3 \end{pmatrix}$$

$$\begin{bmatrix} \mathbf{P}_{xx} & \mathbf{P}_{xy_1} & \mathbf{P}_{xy_2} & \mathbf{P}_{xy_3} \\ \mathbf{P}_{y_1x} & \mathbf{P}_{y_1y_1} & \mathbf{P}_{y_1y_2} & \mathbf{P}_{y_1y_3} \\ \mathbf{P}_{y_2x} & \mathbf{P}_{y_2y_1} & \mathbf{P}_{y_2y_2} & \mathbf{P}_{y_2y_3} \\ \mathbf{P}_{y_3x} & \mathbf{P}_{y_3y_1} & \mathbf{P}_{y_3y_2} & \mathbf{P}_{y_3y_3} \end{bmatrix} \rightarrow \begin{bmatrix} \mathbf{P}_{xx} & \mathbf{P}_{xy_1} & \mathbf{P}_{xy_3} \\ \mathbf{P}_{y_1x} & \mathbf{P}_{y_1y_1} & \mathbf{P}_{y_1y_3} \\ \mathbf{P}_{y_3x} & \mathbf{P}_{y_3y_1} & \mathbf{P}_{y_3y_3} \end{bmatrix}$$

Example Sequence



What Enables It To Run In Real-time?

Timings

Image loading and administration	1ms
Image correlation searches	2ms
Kalman Filter update	1ms
Feature initialisation search	3ms
Graphical rendering	5ms
Total	12ms

Easily manages 30Hz processing on a 3.4GHz desktop PC using C++, Linux, OpenGL

Main time-saving features

- Automatic map management criteria to maintain a sufficient but sparse map
- Active search guided by uncertainty

What Enables It To Run In Real-time?

Timings

Image loading and administration	1ms
Image correlation searches	2ms
Kalman Filter update	1ms
Feature initialisation search	3ms
Graphical rendering	5ms
Total	12ms

Easily manages 30Hz processing on a 3.4GHz desktop PC using C++, Linux, OpenGL

Main time-saving features

- Automatic map management criteria to maintain a sufficient but sparse map
- Active search guided by uncertainty

Outline

The SceneLib Libraries

- A complete basic Davison MonoSLAM system
- Written in standard C++
- Three libraries and an application

SceneLib A generic SLAM library. Base classes for motion models, features, measurements and the Kalman Filter.

SceneImproc Image processing for MonoSLAM (i.e. feature detection and correlation)

MonoSLAM Specific motion and feature-measurement models for single-camera SLAM, and a control class.

MonoSLAMGlow Application based on GLOW/GLUT library which uses the libraries.

Obtaining the SceneLib libraries

- All files available from the Active Vision CVS repository
- Will also need VW34 library (soon to be VW35)

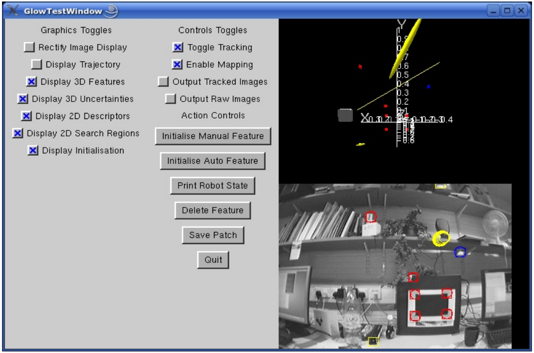
Getting and building files from CVS

```
cvs -d/data/lav-local/common/cvsroot co VW34
cvs -d/data/lav-local/common/cvsroot co SceneLib
cvs -d/data/lav-local/common/cvsroot co MonoSLAMGlow
cd VW34
./bootstrap
./configure
make
cd ../SceneLib
./configure
make
cd ../MonoSLAMGlow
make
```

Running the MonoSLAMGlow application

Running MonoSLAMGlow

```
./scenerob
```

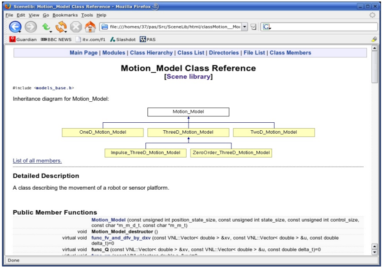


- Comment out `-D_REALTIME_` in `Makefile` to use previously saved image sequence

SceneLib Documentation

Making and viewing documentation

```
cd SceneLib
make docs
firefox html/index.html
```



- See also `SceneLib/Docs/models.tex`.

The Scene library

- Scene is a generic SLAM library.

Main Scene classes

Scene_Single Stores the full system state and manages features.

Feature Stores and manages a feature's state vector and covariances

Motion_Model Base class for all motion models.

Feature_Measurement_Model Base class for all feature and measurement models.

Sim_Or_Rob Base class for something that can make measurements.

Kalman Friend class of *Scene_Single* that implements a Kalman filter.

The Scene library

- The main Scene classes are completely generic:

  Scene_Single stores the 'robot' state vector $\boldsymbol{x}_v$, covariance $\mathbf{P}_{xx}$ and a list of Features.

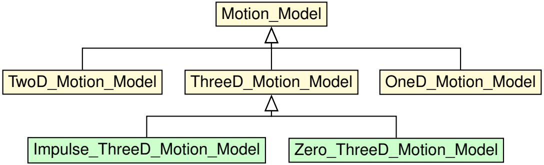
  Feature stores a feature state vector $\boldsymbol{y}_i$, it's covariances $\mathbf{P}_{x_v y_i}$, $\mathbf{P}_{y_i y_i}$ and $\mathbf{P}_{y_i y_j} (\forall j < i)$.

  Kalman updates the state given measurements via *Sim_Or_Rob*.

- The *Motion_Model* and *Feature_Measurement_Model* classes access and interpret the state.
- Classes in MonoSLAM derived from these base classes make the application specific.

Motion model classes

- A *Motion_Model* knows how to perform the state update $\hat{\boldsymbol{x}}_{vnew} = \boldsymbol{f}_v(\hat{\boldsymbol{x}}_v, \boldsymbol{u})$
- It stores no state itself



Main functions

func_fv_and_dfv_by_dxv(xv,u,delta_t) Calculates the new camera state $\hat{\boldsymbol{x}}_{vnew}$ and Jacobian $\frac{\partial \boldsymbol{f}_v}{\partial \boldsymbol{x}_v}$

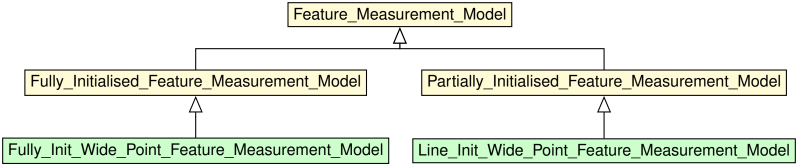
func_Qi(xv, u, delta_t) Calculates the covariance $\mathbf{Q}_i$ of the process noise.

Feature measurement classes

- A *Feature_Measurement_Model* knows how to manage a feature's state and predict a measurement $\boldsymbol{h}_i = \boldsymbol{h}(\boldsymbol{y}_i, \boldsymbol{x}_p)$
- It stores no state itself
- Subclassed into two special types:

  Fully_Initialised_Feature_Measurement_Model A complete feature in the SLAM map.

  Partially_Initialised_Feature_Measurement_Model A feature currently being initialised.



Fully-initialised Features

- A *Fully_Initialised_Feature_Measurement_Model* understands the state of a full feature and determines how it is measured.
- It stores no state itself.

Main functions

func_hi_and_dhi_by_dxp_and_dhi_by_dyi(yi, xp) Calculates the expected measurement vector $\boldsymbol{h}_i$ and its Jacobians $\frac{\partial \boldsymbol{h}_i}{\partial \boldsymbol{x}_p}$ and $\frac{\partial \boldsymbol{h}_i}{\partial \boldsymbol{y}_i}$.

func_Ri(hi) Calculates the measurement noise $\mathbf{R}_i$.

visibility_test(xp, yi, xp_orig, hi) Decides whether a feature should be measured.

func_Si(Pxx, Pxyi, Pyiyi, dhi_by_dxv, dhi_by_dyi, Ri) Calculates the innovation covariance $\mathbf{S}_i$.

Partially-initialised Features

- A *Partially_Initialised_Feature_Measurement_Model* handles a feature which still has some free parameters λ.
- Each partially-initialised feature references a *FeatureInitInfo*
- *FeatureInitInfo* stores a vector of *Particles* representing possible λs and their probabilities.
- A *Partially_Initialised_Feature_Measurement_Model* can convert its feature into a fully-initialised feature.

Partially-initialised Features

Main functions

func_hpi_and_dhpi_by_dxp_and_dhpi_by_dyi(yi, xp, lambda)
: Calculates the expected measurement vector $\boldsymbol{h}_i$ and its Jacobians, given values of the free parameters $\boldsymbol{\lambda}$.

func_Ri(hi)
: Calculates the measurement noise $\mathbf{R}_i$.

visibility_test(xp, yi, xp_orig, hi)
: Decides whether a feature should be measured.

func_Si(Pxx, Pxyi, Pyiyi, dhi_by_dxv, dhi_by_dyi, Ri)
: Calculates the innovation covariance $\mathbf{S}_i$.

func_yfi_and_dyfi_by_dypi_and_dyfi_by_dlambda(ypi, lambda)
: Converts a partially-initialised into a fully-initialised feature.

The MonoSLAM library

The MonoSLAM library provides

- Specialisations of Scene base classes:
  - *Impulse_ThreeD_Motion_Model* and *ZeroOrder_ThreeD_Motion_Model* motion models.
  - *Fully_Init_Wide_Point_Feature_Measurement_Model* and *Line_Init_Wide_Point_Feature_Measurement_Model*.
  - *Robot* (derived from *Sim_Or_Rob*) to handle image feature measurement.
- A *MonoSLAM* class to provide the main interface.
- Functions to draw the two graphical displays.

The MonoSLAM Class

- The *MonoSLAM* class provides the basic SLAM functionality

Main functions

GoOneStep(image, delta_t, currently_mapping_flag) Step the system onto the next frame. (*image* is ignored, and instead it must be set using *Scene_Single::load_new_image()*).

- The *MonoSLAMInterface* class provides full control and feedback functions.

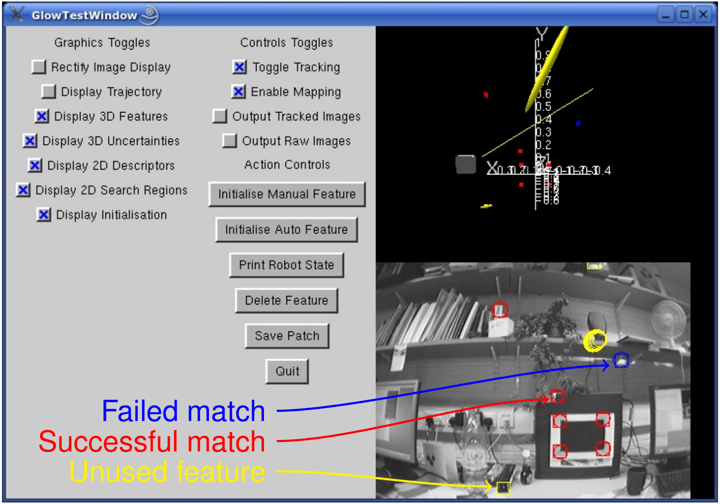
Main functions

GetScene() Get the *Scene_Single* class.

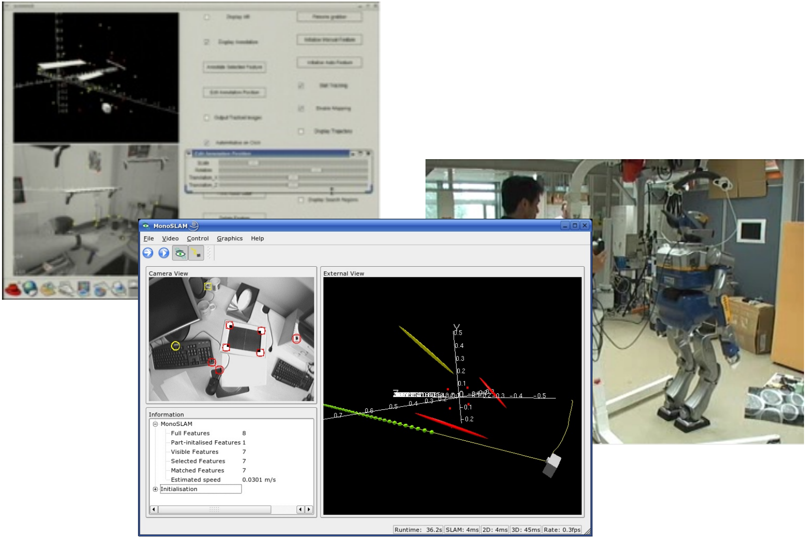
GetRobot() Get the *Robot* class.

plus >40 other *Get()* and *Set()* functions.

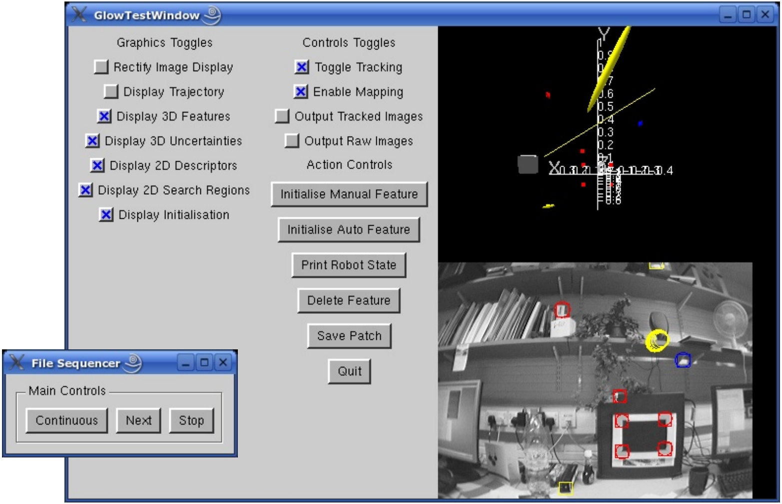
The MonoSLAMGlow Application



Other MonSLAM Applications



Live Demonstration



Outline

Final Thoughts: EKF-based Visual SLAM

Observations

- The Davison visual SLAM system works!
- It works reliably enough for a live demo.
- It needs no real hidden tricks needed to make it work.

Discussion

- Need more, better features to track
  - Faster initialisation (fewer particles?)
  - Use full-frame fast feature detection
- Better initialisation: how do we deal with points at infinity?
- Motion model: How do we get smoother, better tracks?
- Loop closing

Final Thoughts: EKF-based Visual SLAM

Observations

- The Davison visual SLAM system works!
- It works reliably enough for a live demo.
- It needs no real hidden tricks needed to make it work.

Discussion

- Need more, better features to track
  - Faster initialisation (fewer particles?)
  - Use full-frame fast feature detection
- Better initialisation: how do we deal with points at infinity?
- Motion model: How do we get smoother, better tracks?
- Loop closing

Final Thoughts: The SceneLib library

Observations

- The SceneLib libraries are (reasonably) well-designed and (reasonably) well-documented
- They make it easy to write a Davison-style visual SLAM application

Discussion

- Are they useful to the Active Vision group?
- Can we all use them and get the benefits in code sharing that that will bring?
- Can we at least all use the same nomenclature and colours?

Final Thoughts: The SceneLib library

Observations

- The SceneLib libraries are (reasonably) well-designed and (reasonably) well-documented
- They make it easy to write a Davison-style visual SLAM application

Discussion

- Are they useful to the Active Vision group?
- Can we all use them and get the benefits in code sharing that that will bring?
- Can we at least all use the same nomenclature and colours?