Sapienza Università di Roma | Dipartimento di Informatica e Sistemistica Antonio Ruberti

Least Squares and SLAM
Implementing Pose-SLAM

Giorgio Grisetti

Part of the material of this course is taken from the Robotics 2 lectures given by G.Grisetti, W.Burgard, C.Stachniss, K.Arras, D. Tipaldi and M.Bennewitz

Building the Linear System

- $\mathbf{x}$ is the current linearization point
- Initialization

$$\mathbf{b} = 0 \qquad \mathbf{H} = 0$$

- For each constraint
  - Compute the error

$$\mathbf{e}_{ij} = \text{t2v}(\mathbf{Z}_{ij}^{-1}(\mathbf{X}_i^{-1} \cdot \mathbf{X}_j))$$

  - Compute the blocks of the Jacobian:

$$\mathbf{A}_{ij} = \frac{\partial \mathbf{e}(\mathbf{x}_i, \mathbf{x}_j)}{\partial \mathbf{x}_i} \qquad \mathbf{B}_{ij} = \frac{\partial \mathbf{e}(\mathbf{x}_i, \mathbf{x}_j)}{\partial \mathbf{x}_j}$$

  - Update the coefficient vector:

$$\bar{\mathbf{b}}_i^T += \mathbf{e}_{ij}^T \boldsymbol{\Omega}_{ij} \mathbf{A}_{ij} \qquad \bar{\mathbf{b}}_j^T += \mathbf{e}_{ij}^T \boldsymbol{\Omega}_{ij} \mathbf{B}_{ij}$$

  - Update the system matrix:

$$\bar{\mathbf{H}}^{ii} += \mathbf{A}_{ij}^T \boldsymbol{\Omega}_{ij} \mathbf{A}_{ij} \qquad \bar{\mathbf{H}}^{ij} += \mathbf{A}_{ij}^T \boldsymbol{\Omega}_{ij} \mathbf{B}_{ij}$$

$$\bar{\mathbf{H}}^{ji} += \mathbf{B}_{ij}^T \boldsymbol{\Omega}_{ij} \mathbf{A}_{ij} \qquad \bar{\mathbf{H}}^{jj} += \mathbf{B}_{ij}^T \boldsymbol{\Omega}_{ij} \mathbf{B}_{ij}$$

Algorithm

- ***x***: the initial guess
- While (! converged)
  - ***<H,b>*** = buildLinearSystem(***x***);
  - ***Δx*** = solveSparse(***H Δx = -b***);
  - **x** += ***Δx;***

Implementing Least Squares SLAM

- Download the tarball ***ls-slam.tgz*** from the page of the course
- It contains some test data and one possible implementation of what you have seen so far (without time/memory optimzations).
- In this lecture we will recode from scratch the functions in that file.

Loading a graph:

- A graph is stored 2 text files for the vertices and for the edges:
  - Format of the vertex file: a line a vertex
    - VERTEX2 id pose.x pose.y pose.theta
  - Format of the edge file: a line an edge
    - EDGE2 idFrom idTo mean.x mean.y mean.theta inf.xx inf.xy inf.yy inf.xt inf.yt inf.tt

- Loading the graph into matrices:
  - Vertices: 3 x N matrix, the col index is the index of the vertex
  - Edges:
    - 2 x K matrix of indices. A col of the matrix [id1 id2] indicates that the edge K connects the vertices id1 and id2.
    - 3 x K matrix containing the relative transformation encoded in the edge
    - 3 x 3 x K matrix containing the information matrices of the edges

- This is given and implemented by the function
    - function [vmeans, eids, emeans, einfs]=read_graph(vfile, efile)

- Apply this function to the data file and check that it is correct by plotting the vertices.

Error functions and Jacobans

- Write a function that,
  - given the index of an edge (k)
  - Computes the error vector $\boldsymbol{e}_k$
  - The $\mathbf{A_k}$ and $\mathbf{B_k}$ matrices.

  The function should have the following prototype:

  function [e, A, B]=linear_factors(vmeans, eids, emeans, k).

- What is the error function (see old slides)
- What are the elements of the Jacobians?

Error Function Rewritten

- Highlight the rotational and the translational parts of the error vector

$$
\begin{aligned}
\Delta \mathbf{t}_{ij} &= R_z^T \left[R_i^T \left(\begin{pmatrix} x_j \\ y_j \end{pmatrix} - \begin{pmatrix} x_i \\ y_i \end{pmatrix} \right) - \begin{pmatrix} x_z \\ y_z \end{pmatrix} \right] \\
\Delta \theta_{ij} &= -\theta_z + (\theta_j - \theta_i)
\end{aligned}
$$

- Exploit the pure linear dependencies in the derivatives, to compute the Jacobian

Jacobians

$$\mathbf{A}_{ij} = \left[\begin{array}{cc} -R_z^T R_i^T & R_z^T \frac{\partial R_i^T}{\partial \theta_i} (\mathbf{t}_j - \mathbf{t}_i) \\ 0 \; 0 & -1 \end{array} \right]$$

$$\mathbf{B}_{ij} = \left[\begin{array}{cc} R_z^T R_i^T & 0 \\ 0 \; 0 & 1 \end{array} \right]$$

linear_factors(...) 1

```
function [e, A, B]=linear_factors(vmeans, eids, emeans, k)
    #extract the ids of the vertices connected by the kth edge
    id_i=eids(1,k);
    id_j=eids(2,k);
    #extract the poses of the vertices and the mean of the edge
    v_i=vmeans(:,id_i);
    v_j=vmeans(:,id_j);
    z_ij=emeans(:,k);

    #compute the homoeneous transforms of the previous solutions
    zt_ij=v2t(z_ij);
    vt_i=v2t(v_i);
    vt_j=v2t(v_j);

    #compute the displacement between x_i and x_j
    f_ij=(inverse(vt_i)*vt_j);

...
```

linear_factors(...) 2

```
    theta_i=v_i(3);
    ti=(v_i)(1:2,1);
    tj=(v_j)(1:2,1);
         dt_ij=tj-ti;

    si=sin(theta_i);
    ci=cos(theta_i);

    A= [-ci, -si, [-si, ci]*dt_ij; si, -ci, [-ci, -si]*dt_ij; 0, 0, -1 ];
    B =[  ci, si, 0           ; -si, ci, 0            ; 0, 0, 1 ];

    ztinv=inverse(zt_ij);
    e=t2v(ztinv*f_ij);
    ztinv(1:2,3) = 0;
    A=ztinv*A;
    B=ztinv*B;
end;
```

Putting Things Together…

- We now have the error function and the Jacobians.
- We can use the algorithm of the first slide to construct the linear system.
- We can solve it using the “\” operator.
- The function should have the following prototype.

```
#vmeans:   vertices positions at the linearization point
#eids:     edge ids
#emeans:   edge means
#einfs:    edge information matrices
#newmeans: new solution computed from the initial guess in vmeans
function newmeans=linearize_and_solve(vmeans, eids,
    emeans, einfs)
```

Construction of the Linear System

```
# H and b are respectively the system matrix and the system vector
H=zeros(size(vmeans,2)*3,size(vmeans,2)*3);
b=zeros(size(vmeans,2)*3,1);

# this loop constructs the global system by accumulating in H and b the contributions
# of all edges (see lecture)
for k=1:size(eids,2),
        id_i=eids(1,k);
        id_j=eids(2,k);
        [e, A, B]=linear_factors(vmeans, eids, emeans,  k);
        omega=einfs(:,:,k);

        #compute the blocks of H^k
        b_i = -A'*omega*e;
        b_j = -B'*omega*e;
        H_ii= A'*omega*A;
        H_ij= A'*omega*B;
        H_jj= B'*omega*B;

        #accumulate the blocks in H and b
        H((id_i-1)*3+1:id_i*3,(id_i-1)*3+1:id_i*3)+=H_ii;
        H((id_j-1)*3+1:id_j*3,(id_j-1)*3+1:id_j*3)+=H_jj;
        H((id_i-1)*3+1:id_i*3,(id_j-1)*3+1:id_j*3)+=H_ij;
        H((id_j-1)*3+1:id_j*3,(id_i-1)*3+1:id_i*3)+=H_ij'; #symmetric part
        b((id_i-1)*3+1:id_i*3,1)+=b_i;
        b((id_j-1)*3+1:id_j*3,1)+=b_j;
end;
```

Solution of the Linear System

```
#resolve the gauge ambiguity
H(1:3,1:3)+=eye(3);

#use a sparse solver!!!!!
SH=sparse(H);
deltax=SH\b;

#split the increments in nice 3x1 vectors and sum them
newmeans=vmeans+reshape(deltax,3,size(vmeans,2));

#normalize the angles between -PI and PI
for (i=1:size(newmeans,2))
   s=sin(newmeans(3,i));
   c=cos(newmeans(3,i));
   newmeans(3,i)=atan2(s,c);
end
```

Conclusions

- We implemented in the time of a lecture a fully working SLAM system.
- You ca think to extend it with different constraints and node types
  - Landmarks (x/y only)
  - Bearing (theta only)
- We can also determine the relative uncertainties of the nodes.