蒙特卡洛定位

蒙特卡洛定位是基于已经建好的地图进行定位，该算法输入为：轮式里程计数据，激光数据和地图；输出为：机器人当前位姿。在机器人被绑架后，可进行重定位。

6.1.粒子滤波

蒙特卡洛定位算法的主体部分依旧是例子滤波，不过和 gmapping 不同的是，蒙特卡洛定位中使用粒子滤波只需要估计机器人位置，不需要建图。粒子滤波主要流程仍和 gmapping 中类似，主要流程可写成如下 3 步：

(1)生成粒子，通过预测方程预测位姿。

(2)通过测量方程计算粒子的权重。

(3)重采样，计算估计出来的位姿。

6.2KD 树

KD 树可看作 BST 树扩展，BST 树可以提升一维数据的检索效率，左子树上的节点存储的值全部小于根节点的值，右子树上的节点存储的值全部大于根节点的值。与 BST 类似，KD 树上左子树上节点的关键值小于根节点的关键值，右子树的关键值大于根节点的关键值。

KD 树数据结构如下：

```
// A kd tree
typedef struct
{
  // Cell size
  double size[3];

  // The root node of the tree
  pf_kdtree_node_t *root;
  // The number of nodes in the tree
  int node_count, node_max_count;
  pf_kdtree_node_t *nodes;
  // The number of leaf nodes in the tree
  int leaf_count;
} pf_kdtree_t;
```

6.2.1 节点插入

kd 树节点的数据结构如下：

```
typedef struct pf_kdtree_node { // Depth in the tree
int leaf, depth; // Pivot dimension and value
int pivot_dim;
double pivot_value; // The key for this node
int key[3]; // The value for this node
double value; // The cluster label (leaf nodes)
int cluster; // Child nodes
struct pf_kdtree_node *children[2];
} pf_kdtree_node_t;
```

以下函数将一个位姿插入到 Kd 树中，需要创建一个节点存储这个位姿，位姿 (x,y,θ) 和节点键值的转换关系为：

$$key=\begin{bmatrix} 2x \\ 2y \\ \frac{180}{\pi}*\frac{\theta}{10} \end{bmatrix}$$

```
void pf_kdtree_insert(pf_kdtree_t *self, pf_vector_t pose, double value)
{
  int key[3];
//将位姿转换成 kd 树键值
  key[0] = floor(pose.v[0] / self->size[0]);
  key[1] = floor(pose.v[1] / self->size[1]);
  key[2] = floor(pose.v[2] / self->size[2]);
//将键值插入到 kd 树中
  self->root = pf_kdtree_insert_node(self, NULL, self->root, key, value);
  return;
}
```

以下函数将 kd 树节点插入到 Kd 树中：

```
pf_kdtree_node_t *pf_kdtree_insert_node(pf_kdtree_t *self, pf_kdtree_node_t *parent,
                                        pf_kdtree_node_t *node, int key[], double value)
{
  int i;
  int split, max_split;

  // If the node doesnt exist yet...
  if (node == NULL)
  {//当前节点为空，则创建新节点存储键值
    assert(self->node_count < self->node_max_count);
    //获取一个新节点的地址
```

```
  node = self->nodes + self->node_count++;
  //初始化新节点
  memset(node, 0, sizeof(pf_kdtree_node_t));
  //新创建的节点为叶子节点(因为 kd 树是从上往下更新的)
   node->leaf = 1;
   //如果新创建的节点父节点不为空，则其深度更新为其父节点的深度加 1
  if (parent == NULL)
    node->depth = 0;
  else
    node->depth = parent->depth + 1;
  //复制键值
  for (i = 0; i < 3; i++)
    node->key[i] = key[i];
 //更新与位姿对应的值(其实是粒子权重)
  node->value = value;
  //新增加的节点为叶子节点，当前 Kd 树的叶子节点总数加 1
  self->leaf_count += 1;
}
// If the node exists, and it is a leaf node...
else if (node->leaf)
{//当前节点不为空，并且是叶子节点
  // If the keys are equal, increment the value
  if (pf_kdtree_equal(self, key, node->key))
  {//如果键值相等，则累加权重(发现两个位姿一样的粒子)
    node->value += value;
  }
  // The keys are not equal, so split this node
  else
  {
    // Find the dimension with the largest variance and do a mean
    // split
   //键值不相等，计算 split 值和索引
    max_split = 0;
    node->pivot_dim = -1;
    for (i = 0; i < 3; i++)
    {
      split = abs(key[i] - node->key[i]);
      if (split > max_split)
      {
        max_split = split;
        node->pivot_dim = i;
      }
    }
    assert(node->pivot_dim >= 0);
```

```
      node->pivot_value = (key[node->pivot_dim] + node->key[node->pivot_dim]) / 2.0;
    //以当前节点为父节点插入新节点
      if (key[node->pivot_dim] < node->pivot_value)
      {
        node->children[0] = pf_kdtree_insert_node(self, node, NULL, key, value);
        node->children[1] = pf_kdtree_insert_node(self, node, NULL, node->key, node->value);
      }
      else
      {
        node->children[0] = pf_kdtree_insert_node(self, node, NULL, node->key, node->value);
        node->children[1] = pf_kdtree_insert_node(self, node, NULL, key, value);
      }
      node->leaf = 0;
      self->leaf_count -= 1;
    }
  }
  // If the node exists, and it has children...
  else
  {//如果节点存在，将新键值插入到其左右子树中
    assert(node->children[0] != NULL);
    assert(node->children[1] != NULL);
    if (key[node->pivot_dim] < node->pivot_value)
      pf_kdtree_insert_node(self, node, node->children[0], key, value);
    else
      pf_kdtree_insert_node(self, node, node->children[1], key, value);
  }
  return node;
}
```

kd 树键值比较函数，如果所有键值都相等，则键值相等。

```
int pf_kdtree_equal(pf_kdtree_t *self, int key_a[], int key_b[])
{
  //double a, b;

  if (key_a[0] != key_b[0])
    return 0;
  if (key_a[1] != key_b[1])
    return 0;
  if (key_a[2] != key_b[2])
    return 0;
  return 1;
}
```

通过源码可以看出，上面的节点插入操作与 BST 树非常类似。

6.2.3 节点搜索

```
pf_kdtree_node_t *pf_kdtree_find_node(pf_kdtree_t *self, pf_kdtree_node_t *node, int key[])
{

  if (node->leaf)
  {//当前节点为叶子节点，直接比较键值
    //printf("find  : leaf %p %d %d %d\n", node, node->key[0], node->key[1], node->key[2]);
    // If the keys are the same...
    if (pf_kdtree_equal(self, key, node->key))
      return node;
    else
      return NULL;
  }
  else
  {
    //printf("find  : brch %p %d %f\n", node, node->pivot_dim, node->pivot_value);
    assert(node->children[0] != NULL);
    assert(node->children[1] != NULL);
    // If the keys are different...
    //搜索键值小于当前节点键值，往其左子树搜索，大于则往其右子树搜索
    if (key[node->pivot_dim] < node->pivot_value)
      return pf_kdtree_find_node(self, node->children[0], key);
    else
      return pf_kdtree_find_node(self, node->children[1], key);
  }
  return NULL;
}
```

6.2.4 聚类

kd 树聚类函数：

```
void pf_kdtree_cluster(pf_kdtree_t *self)
{
  int i;
  int queue_count, cluster_count;
  pf_kdtree_node_t **queue, *node;

  queue_count = 0;
  queue = calloc(self->node_count, sizeof(queue[0]));
  //将 kd 树中所有的叶节点全部存入 queue 中
  // Put all the leaves in a queue
  for (i = 0; i < self->node_count; i++)
  {
    node = self->nodes + i;
```

```
    if (node->leaf)
    {
      //-1 表示叶节点未被聚类
      node->cluster = -1;
      assert(queue_count < self->node_count);
      queue[queue_count++] = node;
      // TESTING; remove
      assert(node == pf_kdtree_find_node(self, self->root, node->key));
    }
  }

//聚类计数
  cluster_count = 0;
//遍历 queue 中存储的所有叶节点，并对它们进行聚类操作
  // Do connected components for each node
  while (queue_count > 0)
  {
    node = queue[--queue_count];
    // If this node has already been labelled, skip it
    if (node->cluster >= 0)
      continue;
    // Assign a label to this cluster
    node->cluster = cluster_count++;
    // Recursively label nodes in this cluster
    pf_kdtree_cluster_node(self, node, 0);
  }
  free(queue);
  return;
}
```

kd 树节点聚类函数：

```
void pf_kdtree_cluster_node(pf_kdtree_t *self, pf_kdtree_node_t *node, int depth)
{
  int i;
  int nkey[3];
  pf_kdtree_node_t *nnode;

  for (i = 0; i < 3 * 3 * 3; i++)
  {
     //对每个键值进行-1,0,+1 的处理
    nkey[0] = node->key[0] + (i / 9) - 1;
    nkey[1] = node->key[1] + ((i % 9) / 3) - 1;
    nkey[2] = node->key[2] + ((i % 9) % 3) - 1;
    //然后去 kd 树中搜索处理后的键，如果搜索到与处理后的键相等的节点
    nnode = pf_kdtree_find_node(self, self->root, nkey);
```

```
    if (nnode == NULL)
      continue;
    assert(nnode->leaf);
    // This node already has a label; skip it.  The label should be
    // consistent, however.
    if (nnode->cluster >= 0)
    {
    //该节点已经被聚类
      assert(nnode->cluster == node->cluster);
      continue;
    }
    //该节点未被聚类，更新其簇为原始节点的簇
     // Label this node and recurse
    nnode->cluster = node->cluster;
    pf_kdtree_cluster_node(self, nnode, depth + 1);
  }
  return;
}
```

一共 3*3*3=27 中情况，对每个键值进行-1,0,+1 的操作。

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
-1	-1	-1	-1	-1	-1	-1	-1	-1	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1
-1	-1	-1	0	0	0	1	1	1	-1	-1	-1	0	0	0	1	1	1	-1	-1	-1	0	0	0	1	1	1
-1	0	1	-1	0	1	-1	0	1	-1	0	1	-1	0	1	-1	0	1	-1	0	1	-1	0	1	-1	0	1

AMCL 中使用 KD 树的作用是对粒子进行聚类处理，把估计出来位姿连续(经过一系列-1,0,+1 操作位姿相等)的粒子作为一类。

6.3 里程计概率模型(预测更新)

AMCL 中有两种里程计模型(4 个模型，2 个有 bug)，这两个模型的作用都是用来计算先验估计位姿(位姿预测)。

6.3.1 omni corrected 模型

$$
\begin{aligned}
&\varphi = \left|\tan^{-1}\frac{\Delta y}{\Delta x} - \theta_k\right| + \theta_{k,i} \\
&\Delta_{xy} = \sqrt{\Delta x^2 + \Delta y^2} + \alpha * rand() \\
&\Delta_a = \Delta\theta + \beta * rand() \\
&x_{k+1,i} = x_{k,i} + \Delta_{xy}\cos(\varphi) + \gamma * rand() \\
&y_{k+1},i = y_{k,i} + \Delta_{xy}\sin(\varphi) + \gamma * rand() \\
&\theta_{k+1,i} = \theta_{k,i} + \Delta_a
\end{aligned}
$$

其中 $(\Delta x, \Delta y, \Delta\theta)$ 是里程计增量，θ_k 是里程计在 k 时刻的方向角，$(x_{k,i}, y_{k,i}, \theta_{k,i})$ 是第 i 个粒子 k 时刻的位姿，$(x_{k+1,i}, y_{k+1,i}, \theta_{k+1,i})$ 是 k+1 时刻预测出来的位姿(添加了噪声)。

6.3.2 diff corrected 模型

$$
\begin{aligned}
&\Delta_{xy} = \sqrt{\Delta x^2 + \Delta y^2} \\
&\Delta_{a1} = \begin{cases} 0, \Delta_{xy} < 0.01 \\ \left|\tan^{-1}(\frac{\Delta y}{\Delta x}) - \theta_k\right| \end{cases} \\
&\Delta_{a2} = \Delta\theta - \Delta_{a1} \\
&\Delta_{a1\_noise} = \min(|\Delta_A|, |\Delta_{a1} - \pi|) \\
&\Delta_{a2\_noise} = \min(|\Delta a_2|, |\Delta_{a2} - \pi|) \\
&\hat{\Delta}_{a1} = \Delta_{a1} - (\alpha_1 * \Delta_{a1\_noise}{}^2 + \alpha_2 * \Delta_{xy}{}^2) * rand() \\
&\hat{\Delta}_{a2} = \Delta_{a2} - (\alpha_1 * \Delta_{a2\_noise}{}^2 + \alpha_2 * \Delta_{xy}{}^2) \\
&\hat{\Delta}_{xy} = \Delta_{xy} - (\alpha_3 * \Delta_{xy}{}^2 + \alpha_4 * \Delta_{a1\_noise}{}^2 + \alpha_3 * \Delta_{a2\_noise}{}^2) * rand() \\
&x_{k+1,i} = x_{x,i} + \hat{\Delta}_{xy}\cos(\theta_{k,i} + \hat{\Delta}_{a1}) \\
&y_{k+1,i} = y_{k,i} + \hat{\Delta}_{xy}\sin(\theta_{k,i} + \hat{\Delta}_{a1}) \\
&\theta_{k+1,i} = \theta_{k,i} + \hat{\Delta}_{a1} + \hat{\Delta}_{a2}
\end{aligned}
$$

其中 $(\Delta x, \Delta y, \Delta\theta)$ 是里程计增量，θ_k 是里程计在 k 时刻的方向角，$(x_{k,i}, y_{k,i}, \theta_{k,i})$ 是第 i 个粒子 k 时刻的位姿，$(x_{k+1,i}, y_{k+1,i}, \theta_{k+1,i})$ 是 k+1 时刻预测出来的位姿(添加了噪声)。

6.3.3 时间更新

```
bool AMCLOdom::UpdateAction(pf_t *pf, AMCLSensorData *data)
{
  AMCLOdomData *ndata;
  ndata = (AMCLOdomData*) data;

  // Compute the new sample poses
  pf_sample_set_t *set;
  set = pf->sets + pf->current_set;
  pf_vector_t old_pose = pf_vector_sub(ndata->pose, ndata->delta);
  switch( this->model_type )
  {
  case ODOM_MODEL_OMNI:
  {
    double delta_trans, delta_rot, delta_bearing;
    double delta_trans_hat, delta_rot_hat, delta_strafe_hat;
    delta_trans = sqrt(ndata->delta.v[0]*ndata->delta.v[0] +
                       ndata->delta.v[1]*ndata->delta.v[1]);
    delta_rot = ndata->delta.v[2];
    // Precompute a couple of things
    double trans_hat_stddev = (alpha3 * (delta_trans*delta_trans) +
                               alpha1 * (delta_rot*delta_rot));
    double rot_hat_stddev = (alpha4 * (delta_rot*delta_rot) +
                             alpha2 * (delta_trans*delta_trans));
    double strafe_hat_stddev = (alpha1 * (delta_rot*delta_rot) +
                                alpha5 * (delta_trans*delta_trans));
    for (int i = 0; i < set->sample_count; i++)
    {
      pf_sample_t* sample = set->samples + i;
      delta_bearing = angle_diff(atan2(ndata->delta.v[1], ndata->delta.v[0]),
                                 old_pose.v[2]) + sample->pose.v[2];
      double cs_bearing = cos(delta_bearing);
      double sn_bearing = sin(delta_bearing);
      // Sample pose differences
      delta_trans_hat = delta_trans + pf_ran_gaussian(trans_hat_stddev);
      delta_rot_hat = delta_rot + pf_ran_gaussian(rot_hat_stddev);
      delta_strafe_hat = 0 + pf_ran_gaussian(strafe_hat_stddev);
      // Apply sampled update to particle pose
      sample->pose.v[0] += (delta_trans_hat * cs_bearing +
                            delta_strafe_hat * sn_bearing);
      sample->pose.v[1] += (delta_trans_hat * sn_bearing -
                            delta_strafe_hat * cs_bearing);
      sample->pose.v[2] += delta_rot_hat ;
```

```
    }
  }
  break;
  case ODOM_MODEL_DIFF:
  {
    // Implement sample_motion_odometry (Prob Rob p 136)
    double delta_rot1, delta_trans, delta_rot2;
    double delta_rot1_hat, delta_trans_hat, delta_rot2_hat;
    double delta_rot1_noise, delta_rot2_noise;
    // Avoid computing a bearing from two poses that are extremely near each
    // other (happens on in-place rotation).
    if(sqrt(ndata->delta.v[1]*ndata->delta.v[1] +
            ndata->delta.v[0]*ndata->delta.v[0]) < 0.01)
      delta_rot1 = 0.0;
    else
      delta_rot1 = angle_diff(atan2(ndata->delta.v[1], ndata->delta.v[0]),
                              old_pose.v[2]);
    delta_trans = sqrt(ndata->delta.v[0]*ndata->delta.v[0] +
                       ndata->delta.v[1]*ndata->delta.v[1]);
    delta_rot2 = angle_diff(ndata->delta.v[2], delta_rot1);
    // We want to treat backward and forward motion symmetrically for the
    // noise model to be applied below.  The standard model seems to assume
    // forward motion.
    delta_rot1_noise = std::min(fabs(angle_diff(delta_rot1,0.0)),
                                fabs(angle_diff(delta_rot1,M_PI)));
    delta_rot2_noise = std::min(fabs(angle_diff(delta_rot2,0.0)),
                                fabs(angle_diff(delta_rot2,M_PI)));
    for (int i = 0; i < set->sample_count; i++)
    {
      pf_sample_t* sample = set->samples + i;
      // Sample pose differences
      delta_rot1_hat = angle_diff(delta_rot1,
                                  pf_ran_gaussian(this->alpha1*delta_rot1_noise*delta_rot1_noi
se +
                                                  this->alpha2*delta_trans*delta_trans));
      delta_trans_hat = delta_trans -
              pf_ran_gaussian(this->alpha3*delta_trans*delta_trans +
                              this->alpha4*delta_rot1_noise*delta_rot1_noise +
                              this->alpha4*delta_rot2_noise*delta_rot2_noise);
      delta_rot2_hat = angle_diff(delta_rot2,
                                  pf_ran_gaussian(this->alpha1*delta_rot2_noise*delta_rot2_noi
se +
                                                  this->alpha2*delta_trans*delta_trans));
      // Apply sampled update to particle pose
```

```
    sample->pose.v[0] += delta_trans_hat *
            cos(sample->pose.v[2] + delta_rot1_hat);
    sample->pose.v[1] += delta_trans_hat *
            sin(sample->pose.v[2] + delta_rot1_hat);
    sample->pose.v[2] += delta_rot1_hat + delta_rot2_hat;
  }
}
break;
case ODOM_MODEL_OMNI_CORRECTED:
{
  double delta_trans, delta_rot, delta_bearing;
  double delta_trans_hat, delta_rot_hat, delta_strafe_hat;
  delta_trans = sqrt(ndata->delta.v[0]*ndata->delta.v[0] +
                     ndata->delta.v[1]*ndata->delta.v[1]);
  delta_rot = ndata->delta.v[2];
  // Precompute a couple of things
  double trans_hat_stddev = sqrt( alpha3 * (delta_trans*delta_trans) +
                                  alpha4 * (delta_rot*delta_rot) );
  double rot_hat_stddev = sqrt( alpha1 * (delta_rot*delta_rot) +
                                alpha2 * (delta_trans*delta_trans) );
  double strafe_hat_stddev = sqrt( alpha4 * (delta_rot*delta_rot) +
                                   alpha5 * (delta_trans*delta_trans) );
  for (int i = 0; i < set->sample_count; i++)
  {
    pf_sample_t* sample = set->samples + i;
    delta_bearing = angle_diff(atan2(ndata->delta.v[1], ndata->delta.v[0]),
                               old_pose.v[2]) + sample->pose.v[2];
    double cs_bearing = cos(delta_bearing);
    double sn_bearing = sin(delta_bearing);
    // Sample pose differences
    delta_trans_hat = delta_trans + pf_ran_gaussian(trans_hat_stddev);
    delta_rot_hat = delta_rot + pf_ran_gaussian(rot_hat_stddev);
    delta_strafe_hat = 0 + pf_ran_gaussian(strafe_hat_stddev);
    // Apply sampled update to particle pose
    sample->pose.v[0] += (delta_trans_hat * cs_bearing +
                          delta_strafe_hat * sn_bearing);
    sample->pose.v[1] += (delta_trans_hat * sn_bearing -
                          delta_strafe_hat * cs_bearing);
    sample->pose.v[2] += delta_rot_hat ;
  }
}
break;
case ODOM_MODEL_DIFF_CORRECTED:
{
```

```
    // Implement sample_motion_odometry (Prob Rob p 136)
    double delta_rot1, delta_trans, delta_rot2;
    double delta_rot1_hat, delta_trans_hat, delta_rot2_hat;
    double delta_rot1_noise, delta_rot2_noise;
    // Avoid computing a bearing from two poses that are extremely near each
    // other (happens on in-place rotation).
    if(sqrt(ndata->delta.v[1]*ndata->delta.v[1] +
            ndata->delta.v[0]*ndata->delta.v[0]) < 0.01)
      delta_rot1 = 0.0;
    else
      delta_rot1 = angle_diff(atan2(ndata->delta.v[1], ndata->delta.v[0]),
                              old_pose.v[2]);

    delta_trans = sqrt(ndata->delta.v[0]*ndata->delta.v[0] +
                       ndata->delta.v[1]*ndata->delta.v[1]);
    delta_rot2 = angle_diff(ndata->delta.v[2], delta_rot1);
    // We want to treat backward and forward motion symmetrically for the
    // noise model to be applied below.  The standard model seems to assume
    // forward motion.
    delta_rot1_noise = std::min(fabs(angle_diff(delta_rot1,0.0)),
                                fabs(angle_diff(delta_rot1,M_PI)));
    delta_rot2_noise = std::min(fabs(angle_diff(delta_rot2,0.0)),
                                fabs(angle_diff(delta_rot2,M_PI)));
    for (int i = 0; i < set->sample_count; i++)
    {
      pf_sample_t* sample = set->samples + i;
      // Sample pose differences
      delta_rot1_hat = angle_diff(delta_rot1,
                                  pf_ran_gaussian(sqrt(this->alpha1*delta_rot1_noise*delta_rot
1_noise +
                                                       this->alpha2*delta_trans*delta_trans)))
;
      delta_trans_hat = delta_trans -
              pf_ran_gaussian(sqrt(this->alpha3*delta_trans*delta_trans +
                                   this->alpha4*delta_rot1_noise*delta_rot1_noise +
                                   this->alpha4*delta_rot2_noise*delta_rot2_noise));
      delta_rot2_hat = angle_diff(delta_rot2,
                                  pf_ran_gaussian(sqrt(this->alpha1*delta_rot2_noise*delta_rot
2_noise +
                                                       this->alpha2*delta_trans*delta_trans)))
;
      // Apply sampled update to particle pose
      sample->pose.v[0] += delta_trans_hat *
              cos(sample->pose.v[2] + delta_rot1_hat);
```

```
      sample->pose.v[1] += delta_trans_hat *
              sin(sample->pose.v[2] + delta_rot1_hat);
      sample->pose.v[2] += delta_rot1_hat + delta_rot2_hat;
    }
  }
  break;
  }
  return true;
}
```

当里程计数据更新时，计算出里程计两个连续位姿之间的位姿变化量后，调用该函数进行位姿预测，该函数在下面的函数调用。

```
void pf_update_action(pf_t *pf, pf_action_model_fn_t action_fn, void *action_data)
{
  pf_sample_set_t *set;

  set = pf->sets + pf->current_set;
  (*action_fn) (action_data, set);

  return;
}
```

6.4 激光雷达概率模型(测量更新)

6.4.1 beam 模型

```
double AMCLLaser::BeamModel(AMCLLaserData *data, pf_sample_set_t* set)
{
  AMCLLaser *self;
  int i, j, step;
  double z, pz;
  double p;
  double map_range;
  double obs_range, obs_bearing;
  double total_weight;
  pf_sample_t *sample;
  pf_vector_t pose;

  self = (AMCLLaser*) data->sensor;
  total_weight = 0.0;
  // Compute the sample weights
  for (j = 0; j < set->sample_count; j++)
  {
```

```
    //获取当前粒子的位姿
    sample = set->samples + j;
    pose = sample->pose;
    // Take account of the laser pose relative to the robot
    pose = pf_vector_coord_add(self->laser_pose, pose);
    p = 1.0;
    step = (data->range_count - 1) / (self->max_beams - 1);




    for (i = 0; i < data->range_count; i += step)
    {
      obs_range = data->ranges[i][0];
      obs_bearing = data->ranges[i][1];

      // Compute the range according to the map
      map_range = map_calc_range(self->map, pose.v[0], pose.v[1],
                                 pose.v[2] + obs_bearing, data->range_max);
      pz = 0.0;
      // Part 1: good, but noisy, hit
      z = obs_range - map_range;
      pz += self->z_hit * exp(-(z * z) / (2 * self->sigma_hit * self->sigma_hit));
      // Part 2: short reading from unexpected obstacle (e.g., a person)
      if(z < 0)
        pz += self->z_short * self->lambda_short * exp(-self->lambda_short*obs_range);
      // Part 3: Failure to detect obstacle, reported as max-range
      if(obs_range == data->range_max)
        pz += self->z_max * 1.0;
      // Part 4: Random measurements
      if(obs_range < data->range_max)
        pz += self->z_rand * 1.0/data->range_max;
      // TODO: outlier rejection for short readings
      assert(pz <= 1.0);
      assert(pz >= 0.0);
      //      p *= pz;
      // here we have an ad-hoc weighting scheme for combining beam probs
      // works well, though...
      p += pz*pz*pz;
    }
    sample->weight *= p;
    total_weight += sample->weight;
  }
  return(total_weight);
}
```

6.4.2 likehood field 模型

```
double AMCLLaser::LikelihoodFieldModel(AMCLLaserData *data, pf_sample_set_t* set)
{
  AMCLLaser *self;
  int i, j, step;
  double z, pz;
  double p;
  double obs_range, obs_bearing;
  double total_weight;
  pf_sample_t *sample;
  pf_vector_t pose;
  pf_vector_t hit;

  self = (AMCLLaser*) data->sensor;
  total_weight = 0.0;
  // Compute the sample weights
  for (j = 0; j < set->sample_count; j++)
  {
    sample = set->samples + j;
    pose = sample->pose;
    // Take account of the laser pose relative to the robot
    pose = pf_vector_coord_add(self->laser_pose, pose);
    p = 1.0;
    // Pre-compute a couple of things
    double z_hit_denom = 2 * self->sigma_hit * self->sigma_hit;
    double z_rand_mult = 1.0/data->range_max;
    step = (data->range_count - 1) / (self->max_beams - 1);
    // Step size must be at least 1
    if(step < 1)
      step = 1;
    for (i = 0; i < data->range_count; i += step)
    {
      obs_range = data->ranges[i][0];
      obs_bearing = data->ranges[i][1];
      // This model ignores max range readings
      if(obs_range >= data->range_max)
        continue;
      // Check for NaN
      if(obs_range != obs_range)
        continue;
      pz = 0.0;
      // Compute the endpoint of the beam
      hit.v[0] = pose.v[0] + obs_range * cos(pose.v[2] + obs_bearing);
```

```
      hit.v[1] = pose.v[1] + obs_range * sin(pose.v[2] + obs_bearing);
      // Convert to map grid coords.
      int mi, mj;
      mi = MAP_GXWX(self->map, hit.v[0]);
      mj = MAP_GYWY(self->map, hit.v[1]);

      // Part 1: Get distance from the hit to closest obstacle.
      // Off-map penalized as max distance
      if(!MAP_VALID(self->map, mi, mj))
        z = self->map->max_occ_dist;
      else
        z = self->map->cells[MAP_INDEX(self->map,mi,mj)].occ_dist;
      // Gaussian model
      // NOTE: this should have a normalization of 1/(sqrt(2pi)*sigma)
      pz += self->z_hit * exp(-(z * z) / z_hit_denom);
      // Part 2: random measurements
      pz += self->z_rand * z_rand_mult;
      // TODO: outlier rejection for short readings
      assert(pz <= 1.0);
      assert(pz >= 0.0);
      //      p *= pz;
      // here we have an ad-hoc weighting scheme for combining beam probs
      // works well, though...
      p += pz*pz*pz;
    }
    sample->weight *= p;
    total_weight += sample->weight;
  }
  return(total_weight);
}
```

6.4.3likehood field prob 模型

```
double AMCLLaser::LikelihoodFieldModelProb(AMCLLaserData *data, pf_sample_set_t* set)
{
  AMCLLaser *self;
  int i, j, step;
  double z, pz;
  double log_p;
  double obs_range, obs_bearing;
  double total_weight;
  pf_sample_t *sample;
  pf_vector_t pose;
  pf_vector_t hit;

  self = (AMCLLaser*) data->sensor;
  total_weight = 0.0;
  step = ceil((data->range_count) / static_cast<double>(self->max_beams));

  // Step size must be at least 1
  if(step < 1)
    step = 1;
  // Pre-compute a couple of things
  double z_hit_denom = 2 * self->sigma_hit * self->sigma_hit;
  double z_rand_mult = 1.0/data->range_max;
  double max_dist_prob = exp(-(self->map->max_occ_dist * self->map->max_occ_dist) / z_hit_deno
m);
  //Beam skipping - ignores beams for which a majoirty of particles do not agree with the map
  //prevents correct particles from getting down weighted because of unexpected obstacles
  //such as humans
  bool do_beamskip = self->do_beamskip;
  double beam_skip_distance = self->beam_skip_distance;
  double beam_skip_threshold = self->beam_skip_threshold;

  //we only do beam skipping if the filter has converged
  if(do_beamskip && !set->converged){
    do_beamskip = false;
  }
  //we need a count the no of particles for which the beam agreed with the map
  int *obs_count = new int[self->max_beams]();
  //we also need a mask of which observations to integrate (to decide which beams to integrate
 to all particles)
  bool *obs_mask = new bool[self->max_beams]();
```

```
  int beam_ind = 0;

  //realloc indicates if we need to reallocate the temp data structure needed to do beamskippi
ng
  bool realloc = false;
  if(do_beamskip){
    if(self->max_obs < self->max_beams){
      realloc = true;
    }
    if(self->max_samples < set->sample_count){
      realloc = true;
    }
    if(realloc){
      self->reallocTempData(set->sample_count, self->max_beams);
      fprintf(stderr, "Reallocing temp weights %d - %d\n", self->max_samples, self->max_obs);
    }
  }
  // Compute the sample weights
  for (j = 0; j < set->sample_count; j++)
  {
    sample = set->samples + j;
    pose = sample->pose;
    // Take account of the laser pose relative to the robot
    pose = pf_vector_coord_add(self->laser_pose, pose);
    log_p = 0;

    beam_ind = 0;

    for (i = 0; i < data->range_count; i += step, beam_ind++)
    {
      obs_range = data->ranges[i][0];
      obs_bearing = data->ranges[i][1];
      // This model ignores max range readings
      if(obs_range >= data->range_max){
        continue;
      }
      // Check for NaN
      if(obs_range != obs_range){
        continue;
      }
      pz = 0.0;
      // Compute the endpoint of the beam
      hit.v[0] = pose.v[0] + obs_range * cos(pose.v[2] + obs_bearing);
      hit.v[1] = pose.v[1] + obs_range * sin(pose.v[2] + obs_bearing);
```

```
    // Convert to map grid coords.
    int mi, mj;
    mi = MAP_GXWX(self->map, hit.v[0]);
    mj = MAP_GYWY(self->map, hit.v[1]);

    // Part 1: Get distance from the hit to closest obstacle.
    // Off-map penalized as max distance

    if(!MAP_VALID(self->map, mi, mj)){
        pz += self->z_hit * max_dist_prob;
    }
    else{
        z = self->map->cells[MAP_INDEX(self->map,mi,mj)].occ_dist;
        if(z < beam_skip_distance){
             obs_count[beam_ind] += 1;
        }
        pz += self->z_hit * exp(-(z * z) / z_hit_denom);
    }

    // Gaussian model
    // NOTE: this should have a normalization of 1/(sqrt(2pi)*sigma)

    // Part 2: random measurements
    pz += self->z_rand * z_rand_mult;
    assert(pz <= 1.0);
    assert(pz >= 0.0);
    // TODO: outlier rejection for short readings

    if(!do_beamskip){
        log_p += log(pz);
    }
    else{
        self->temp_obs[j][beam_ind] = pz;
    }
  }
  if(!do_beamskip){
    sample->weight *= exp(log_p);
    total_weight += sample->weight;
  }
}

if(do_beamskip){
  int skipped_beam_count = 0;
  for (beam_ind = 0; beam_ind < self->max_beams; beam_ind++){
```

```
      if((obs_count[beam_ind] / static_cast<double>(set->sample_count)) > beam_skip_threshold)
{
  obs_mask[beam_ind] = true;
      }
      else{
  obs_mask[beam_ind] = false;
  skipped_beam_count++;
      }
    }
    //we check if there is at least a critical number of beams that agreed with the map
    //otherwise it probably indicates that the filter converged to a wrong solution
    //if that's the case we integrate all the beams and hope the filter might converge to
    //the right solution
    bool error = false;
    if(skipped_beam_count >= (beam_ind * self->beam_skip_error_threshold)){
      fprintf(stderr, "Over %f%% of the observations were not in the map - pf may have converg
ed to wrong pose - integrating all observations\n", (100 * self->beam_skip_error_threshold));
      error = true;
    }
    for (j = 0; j < set->sample_count; j++)
    {
     sample = set->samples + j;
     pose = sample->pose;
     log_p = 0;
     for (beam_ind = 0; beam_ind < self->max_beams; beam_ind++){
       if(error || obs_mask[beam_ind]){
          log_p += log(self->temp_obs[j][beam_ind]);
        }
       }
     sample->weight *= exp(log_p);
     total_weight += sample->weight;
     }
    }
  delete [] obs_count;
  delete [] obs_mask;
  return(total_weight);
}
```

6.3.4 测量更新

```
void pf_update_sensor(pf_t *pf, pf_sensor_model_fn_t sensor_fn, void *sensor_data)
{
  int i;
  pf_sample_set_t *set;
  pf_sample_t *sample;
  double total;

  set = pf->sets + pf->current_set;
  // Compute the sample weights
  total = (*sensor_fn) (sensor_data, set);
  //更新粒子权重
  if (total > 0.0)
  {
    // Normalize weights
    double w_avg=0.0;
    for (i = 0; i < set->sample_count; i++)
    {
      sample = set->samples + i;
      w_avg += sample->weight;
      sample->weight /= total;
    }
    // Update running averages of likelihood of samples (Prob Rob p258)
    w_avg /= set->sample_count;
    if(pf->w_slow == 0.0)
      pf->w_slow = w_avg;
    else
      pf->w_slow += pf->alpha_slow * (w_avg - pf->w_slow);
    if(pf->w_fast == 0.0)
      pf->w_fast = w_avg;
    else
      pf->w_fast += pf->alpha_fast * (w_avg - pf->w_fast);
    //printf("w_avg: %e slow: %e fast: %e\n",
           //w_avg, pf->w_slow, pf->w_fast);
  }
  else
  {
    // Handle zero total
    for (i = 0; i < set->sample_count; i++)
    {
      sample = set->samples + i;
      sample->weight = 1.0 / set->sample_count;
    }
```

```
  }
  return;
}
```

6.5 重采样

```
// Resample the distribution
void pf_update_resample(pf_t *pf)
{
  int i;
  double total;
  pf_sample_set_t *set_a, *set_b;
  pf_sample_t *sample_a, *sample_b;

  //double r,c,U;
  //int m;
  //double count_inv;
  double* c;
  double w_diff;
  set_a = pf->sets + pf->current_set;
  set_b = pf->sets + (pf->current_set + 1) % 2;
  // Build up cumulative probability table for resampling.
  // TODO: Replace this with a more efficient procedure
  // (e.g., http://www.network-theory.co.uk/docs/gslref/GeneralDiscreteDistributions.html)
  c = (double*)malloc(sizeof(double)*(set_a->sample_count+1));

  c[0] = 0.0;
  for(i=0;i<set_a->sample_count;i++)
    c[i+1] = c[i]+set_a->samples[i].weight;
  // Create the kd tree for adaptive sampling
  pf_kdtree_clear(set_b->kdtree);

  // Draw samples from set a to create set b.
  total = 0;
  set_b->sample_count = 0;
  w_diff = 1.0 - pf->w_fast / pf->w_slow;
  if(w_diff < 0.0)
    w_diff = 0.0;

  while(set_b->sample_count < pf->max_samples)
  {
    sample_b = set_b->samples + set_b->sample_count++;
    if(drand48() < w_diff)
```

```
    sample_b->pose = (pf->random_pose_fn)(pf->random_pose_data);
  else
  {
    // Naive discrete event sampler
   //把选出来的粒子位姿存入到 set_b 中
    double r;
    r = drand48();
    for(i=0;i<set_a->sample_count;i++)
    {
      if((c[i] <= r) && (r < c[i+1]))
        break;
    }
    assert(i<set_a->sample_count);
    sample_a = set_a->samples + i;
    assert(sample_a->weight > 0);
    // Add sample to list
    sample_b->pose = sample_a->pose;
  }
  sample_b->weight = 1.0;
  total += sample_b->weight;

  // Add sample to histogram
  //将选出来的粒子加入到 kd 树中
  pf_kdtree_insert(set_b->kdtree, sample_b->pose, sample_b->weight);
  // See if we have enough samples yet
  //当选出来的粒子个数超过 kd 树中叶子节点个数时，停止选取粒子
  if (set_b->sample_count > pf_resample_limit(pf, set_b->kdtree->leaf_count))
    break;
}

// Reset averages, to avoid spiraling off into complete randomness.
if(w_diff > 0.0)
  pf->w_slow = pf->w_fast = 0.0;

// Normalize weights
//粒子权重归一化
for (i = 0; i < set_b->sample_count; i++)
{
  sample_b = set_b->samples + i;
  sample_b->weight /= total;
}

// Re-compute cluster statistics
//kd 树聚类
```

```
  pf_cluster_stats(pf, set_b);

  // Use the newly created sample set
  //切换当前粒子集合
  pf->current_set = (pf->current_set + 1) % 2;
  //更新粒子滤波收敛状态
  pf_update_converged(pf);
  free(c);
  return;
}
```

粒子聚类：

```
void pf_cluster_stats(pf_t *pf, pf_sample_set_t *set)
{
  int i, j, k, cidx;
  pf_sample_t *sample;
  pf_cluster_t *cluster;

  // Workspace
  double m[4], c[2][2];
  size_t count;
  double weight;

  // Cluster the samples
  //kd 树聚类
  pf_kdtree_cluster(set->kdtree);

  // Initialize cluster stats
  //初始化聚类簇的状态
  set->cluster_count = 0;
  for (i = 0; i < set->cluster_max_count; i++)
  {
    cluster = set->clusters + i;
    cluster->count = 0;
    cluster->weight = 0;
    cluster->mean = pf_vector_zero();
    cluster->cov = pf_matrix_zero();
    for (j = 0; j < 4; j++)
      cluster->m[j] = 0.0;
    for (j = 0; j < 2; j++)
      for (k = 0; k < 2; k++)
        cluster->c[j][k] = 0.0;
  }
  // Initialize overall filter stats
  count = 0;
```

```
weight = 0.0;
set->mean = pf_vector_zero();
set->cov = pf_matrix_zero();
for (j = 0; j < 4; j++)
  m[j] = 0.0;
for (j = 0; j < 2; j++)
  for (k = 0; k < 2; k++)
    c[j][k] = 0.0;

// Compute cluster stats
for (i = 0; i < set->sample_count; i++)
{
  sample = set->samples + i;
  //printf("%d %f %f %f\n", i, sample->pose.v[0], sample->pose.v[1], sample->pose.v[2]);

   // Get the cluster label for this sample
  //获取当前粒子的簇编号
  cidx = pf_kdtree_get_cluster(set->kdtree, sample->pose);
  assert(cidx >= 0);
  if (cidx >= set->cluster_max_count)
    continue;
  if (cidx + 1 > set->cluster_count)
    set->cluster_count = cidx + 1;
  //获取当前粒子的簇
  cluster = set->clusters + cidx;
  //更新簇信息
  cluster->count += 1;
  cluster->weight += sample->weight;
  //计算粒子个数和粒子权重和
  count += 1;
  weight += sample->weight;

  // Compute mean
  //计算粒子簇的带权重的平均位姿
  cluster->m[0] += sample->weight * sample->pose.v[0];
  cluster->m[1] += sample->weight * sample->pose.v[1];
  cluster->m[2] += sample->weight * cos(sample->pose.v[2]);
  cluster->m[3] += sample->weight * sin(sample->pose.v[2]);

  //计算带权重的粒子平均位姿
  m[0] += sample->weight * sample->pose.v[0];
  m[1] += sample->weight * sample->pose.v[1];
  m[2] += sample->weight * cos(sample->pose.v[2]);
  m[3] += sample->weight * sin(sample->pose.v[2]);
```

```
  // Compute covariance in linear components
  for (j = 0; j < 2; j++)
    for (k = 0; k < 2; k++)
    {
      cluster->c[j][k] += sample->weight * sample->pose.v[j] * sample->pose.v[k];
      c[j][k] += sample->weight * sample->pose.v[j] * sample->pose.v[k];
    }
}
// Normalize
for (i = 0; i < set->cluster_count; i++)
{
  cluster = set->clusters + i;
  //计算簇位姿平均值
  cluster->mean.v[0] = cluster->m[0] / cluster->weight;
  cluster->mean.v[1] = cluster->m[1] / cluster->weight;
  cluster->mean.v[2] = atan2(cluster->m[3], cluster->m[2]);
  cluster->cov = pf_matrix_zero();

  // Covariance in linear components
  //计算协方差矩阵
  for (j = 0; j < 2; j++)
    for (k = 0; k < 2; k++)
      cluster->cov.m[j][k] = cluster->c[j][k] / cluster->weight -
        cluster->mean.v[j] * cluster->mean.v[k];
  // Covariance in angular components; I think this is the correct
  // formula for circular statistics.
  cluster->cov.m[2][2] = -2 * log(sqrt(cluster->m[2] * cluster->m[2] +
                                       cluster->m[3] * cluster->m[3]));
  //printf("cluster %d %d %f (%f %f %f)\n", i, cluster->count, cluster->weight,
         //cluster->mean.v[0], cluster->mean.v[1], cluster->mean.v[2]);
  //pf_matrix_fprintf(cluster->cov, stdout, "%e");
}



// Compute overall filter stats
//计算粒子位姿平均值
//D(x) = E[(x-x_)^2]=E(x^2)-E(x_)^2
set->mean.v[0] = m[0] / weight;
set->mean.v[1] = m[1] / weight;
set->mean.v[2] = atan2(m[3], m[2]);
// Covariance in linear components
//计算协方差矩阵
```

```
  for (j = 0; j < 2; j++)
    for (k = 0; k < 2; k++)
      set->cov.m[j][k] = c[j][k] / weight - set->mean.v[j] * set->mean.v[k];
  // Covariance in angular components; I think this is the correct
  // formula for circular statistics.
  set->cov.m[2][2] = -2 * log(sqrt(m[2] * m[2] + m[3] * m[3]));
  return;
}
```

根据 Kd 树中叶子节点个数计算重采样生成的粒子数下界。

```
int pf_resample_limit(pf_t *pf, int k)
{
  double a, b, c, x;
  int n;

  if (k <= 1)
    return pf->max_samples;
  a = 1;
  b = 2 / (9 * ((double) k - 1));
  c = sqrt(2 / (9 * ((double) k - 1))) * pf->pop_z;
  x = a - b + c;
  n = (int) ceil((k - 1) / (2 * pf->pop_err) * x * x * x);
  if (n < pf->min_samples)
    return pf->min_samples;
  if (n > pf->max_samples)
    return pf->max_samples;

  return n;
}
```

更新粒子滤波姿态估计收敛情况，当粒子的位姿与粒子位姿平均值的距离在一个阈值内就认为粒子滤波收敛。

```
int pf_update_converged(pf_t *pf)
{
  int i;
  pf_sample_set_t *set;
  pf_sample_t *sample;
  double total;

  set = pf->sets + pf->current_set;
  double mean_x = 0, mean_y = 0;
  for (i = 0; i < set->sample_count; i++){
    sample = set->samples + i;
    mean_x += sample->pose.v[0];
    mean_y += sample->pose.v[1];
  }
```

```
  mean_x /= set->sample_count;
  mean_y /= set->sample_count;

  for (i = 0; i < set->sample_count; i++){
    sample = set->samples + i;
    if(fabs(sample->pose.v[0] - mean_x) > pf->dist_threshold ||
       fabs(sample->pose.v[1] - mean_y) > pf->dist_threshold){
      set->converged = 0;
      pf->converged = 0;
      return 0;
    }
  }
  set->converged = 1;
  pf->converged = 1;
  return 1;
}
```

6.5 重定位

收到重定位服务请求时，调用其回调函数，如下：

```
bool
AmclNode::globalLocalizationCallback(std_srvs::Empty::Request& req,
                                     std_srvs::Empty::Response& res)
{
  if( map_ == NULL ) {
    return true;
  }
  boost::recursive_mutex::scoped_lock gl(configuration_mutex_);
  ROS_INFO("Initializing with uniform distribution");
  pf_init_model(pf_, (pf_init_model_fn_t)AmclNode::uniformPoseGenerator,
                (void *)map_);
  ROS_INFO("Global initialisation done!");
  pf_init_ = false;
  return true;
}
```

该回调函数调用粒子滤波模型初始化函数，其源码如下，该函数的主要作用是调用位姿随机位姿生成函数生成的位姿重置粒子位姿。

```
void pf_init_model(pf_t *pf, pf_init_model_fn_t init_fn, void *init_data)
{
  int i;
  pf_sample_set_t *set;
  pf_sample_t *sample;

  set = pf->sets + pf->current_set;
  // Create the kd tree for adaptive sampling
```

```
  pf_kdtree_clear(set->kdtree);
  set->sample_count = pf->max_samples;
  // Compute the new sample poses
  for (i = 0; i < set->sample_count; i++)
  {
    sample = set->samples + i;
    sample->weight = 1.0 / pf->max_samples;
    sample->pose = (*init_fn) (init_data);
    // Add sample to histogram
    pf_kdtree_insert(set->kdtree, sample->pose, sample->weight);
  }
  pf->w_slow = pf->w_fast = 0.0;
  // Re-compute cluster statistics
  pf_cluster_stats(pf, set);

  //set converged to 0
  pf_init_converged(pf);
  return;
}
```

重定位时，生成粒子位姿的函数如下：

```
pf_vector_t AmclNode::uniformPoseGenerator(void* arg)
{
  map_t* map = (map_t*)arg;
#if NEW_UNIFORM_SAMPLING
  unsigned int rand_index = drand48() * free_space_indices.size();
  std::pair<int,int> free_point = free_space_indices[rand_index];
  pf_vector_t p;
  p.v[0] = MAP_WXGX(map, free_point.first);
  p.v[1] = MAP_WYGY(map, free_point.second);
  p.v[2] = drand48() * 2 * M_PI - M_PI;
#else
  double min_x, max_x, min_y, max_y;

  min_x = (map->size_x * map->scale)/2.0 - map->origin_x;
  max_x = (map->size_x * map->scale)/2.0 + map->origin_x;
  min_y = (map->size_y * map->scale)/2.0 - map->origin_y;
  max_y = (map->size_y * map->scale)/2.0 + map->origin_y;
  pf_vector_t p;
  ROS_DEBUG("Generating new uniform sample");
  for(;;)
  {
    p.v[0] = min_x + drand48() * (max_x - min_x);
    p.v[1] = min_y + drand48() * (max_y - min_y);
    p.v[2] = drand48() * 2 * M_PI - M_PI;
```

```
    // Check that it's a free cell
    int i,j;
    i = MAP_GXWX(map, p.v[0]);
    j = MAP_GYWY(map, p.v[1]);
    if(MAP_VALID(map,i,j) && (map->cells[MAP_INDEX(map,i,j)].occ_state == -1))
      break;
  }
#endif
  return p;
}
```

上述函数中 free_space_indices 数组在地图订阅的回调函数中存入数据的，如下：

```
void
AmclNode::handleMapMessage(const nav_msgs::OccupancyGrid& msg)
{
  boost::recursive_mutex::scoped_lock cfl(configuration_mutex_);

  ROS_INFO("Received a %d X %d map @ %.3f m/pix\n",
           msg.info.width,
           msg.info.height,
           msg.info.resolution);

  if(msg.header.frame_id != global_frame_id_)
    ROS_WARN("Frame_id of map received:'%s' doesn't match global_frame_id:'%s'. This could cau
se issues with reading published topics",
             msg.header.frame_id.c_str(),
             global_frame_id_.c_str());
  freeMapDependentMemory();
  // Clear queued laser objects because they hold pointers to the existing
  // map, #5202.
  lasers_.clear();
  lasers_update_.clear();
  frame_to_laser_.clear();
  map_ = convertMap(msg);
#if NEW_UNIFORM_SAMPLING
  // Index of free space
  free_space_indices.resize(0);
  for(int i = 0; i < map_->size_x; i++)
    for(int j = 0; j < map_->size_y; j++)
      if(map_->cells[MAP_INDEX(map_,i,j)].occ_state == -1)
        free_space_indices.push_back(std::make_pair(i,j));
#endif
  // Create the particle filter
  pf_ = pf_alloc(min_particles_, max_particles_,
                 alpha_slow_, alpha_fast_,
```

```
                (pf_init_model_fn_t)AmclNode::uniformPoseGenerator,
                (void *)map_);
pf_->pop_err = pf_err_;
pf_->pop_z = pf_z_;
// Initialize the filter
updatePoseFromServer();
pf_vector_t pf_init_pose_mean = pf_vector_zero();
pf_init_pose_mean.v[0] = init_pose_[0];
pf_init_pose_mean.v[1] = init_pose_[1];
pf_init_pose_mean.v[2] = init_pose_[2];
pf_matrix_t pf_init_pose_cov = pf_matrix_zero();
pf_init_pose_cov.m[0][0] = init_cov_[0];
pf_init_pose_cov.m[1][1] = init_cov_[1];
pf_init_pose_cov.m[2][2] = init_cov_[2];
pf_init(pf_, pf_init_pose_mean, pf_init_pose_cov);
pf_init_ = false;
// Instantiate the sensor objects
// Odometry
delete odom_;
odom_ = new AMCLOdom();
ROS_ASSERT(odom_);
odom_->SetModel( odom_model_type_, alpha1_, alpha2_, alpha3_, alpha4_, alpha5_ );
// Laser
delete laser_;
laser_ = new AMCLLaser(max_beams_, map_);
ROS_ASSERT(laser_);
if(laser_model_type_ == LASER_MODEL_BEAM)
  laser_->SetModelBeam(z_hit_, z_short_, z_max_, z_rand_,
                       sigma_hit_, lambda_short_, 0.0);
else if(laser_model_type_ == LASER_MODEL_LIKELIHOOD_FIELD_PROB){
  ROS_INFO("Initializing likelihood field model; this can take some time on large maps...");
  laser_->SetModelLikelihoodFieldProb(z_hit_, z_rand_, sigma_hit_,
        laser_likelihood_max_dist_,
        do_beamskip_, beam_skip_distance_,
        beam_skip_threshold_, beam_skip_error_threshold_);
  ROS_INFO("Done initializing likelihood field model.");
}
else
{
  ROS_INFO("Initializing likelihood field model; this can take some time on large maps...");
  laser_->SetModelLikelihoodField(z_hit_, z_rand_, sigma_hit_,
                                  laser_likelihood_max_dist_);
  ROS_INFO("Done initializing likelihood field model.");
}
```

```
  // In case the initial pose message arrived before the first map,
  // try to apply the initial pose now that the map has arrived.
  applyInitialPose();
}
```

由此可知，重定位时，在整个地图范围内随机生成位姿，并用这些位姿对所有粒子进行初始化，初始化后，通过激光和地图的匹配调整粒子权重，距离真实位姿近的粒子权重较大，距离真实位姿较远的粒子权重较小，通过迭代，粒子滤波输出的位姿越来越接近真实位姿。