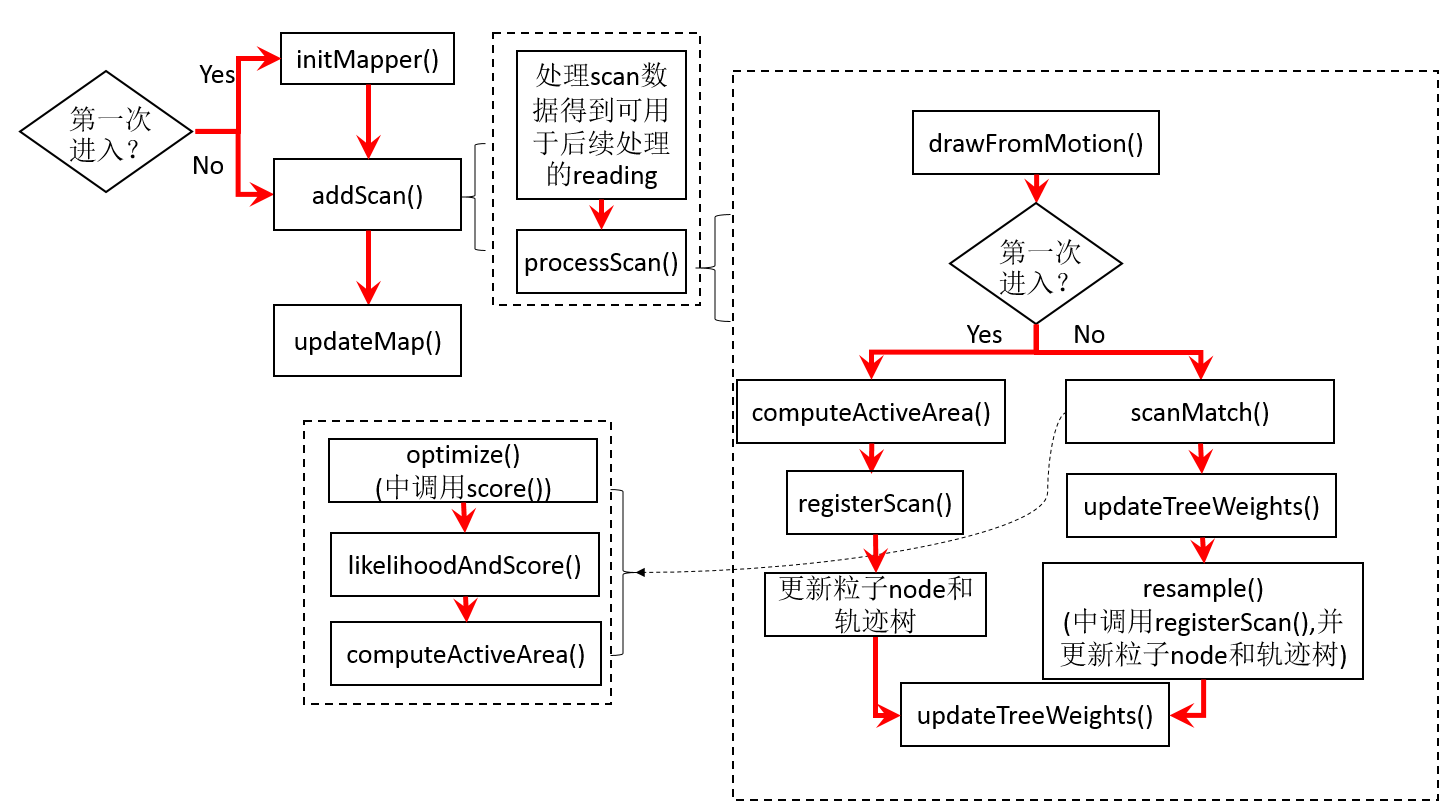
Slam\_gmaping代码阅读笔记：

# 总体流程说明

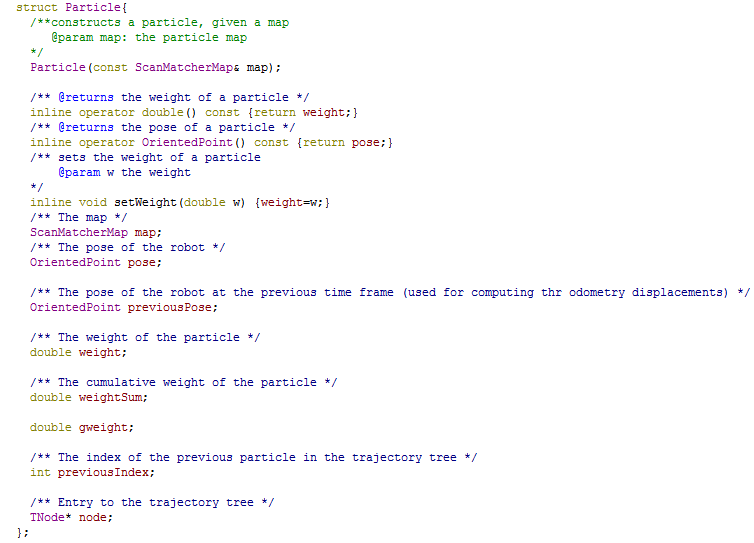
slam\_gmapping的主要代码段是“slam\_gmapping.cpp”中的laserCallback()。每收到一次扫描数据就会进入这个函数。laserCallback()代码执行流程大致如下图所示：



在第一次进入laserCallback()时，会调用initMapper()对地图进行一次初始化。之后每次进入laserCallback()，则会首先进入addScan()，再执行updateMap().

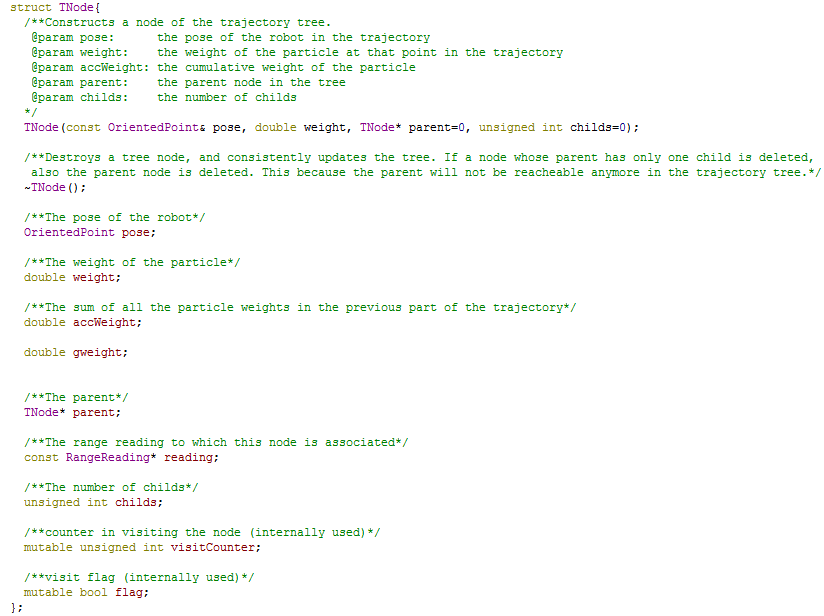
# 粒子（Particle）与轨迹树（trajectory tree）及地图的实现：

Gmapping中粒子的定义如下：

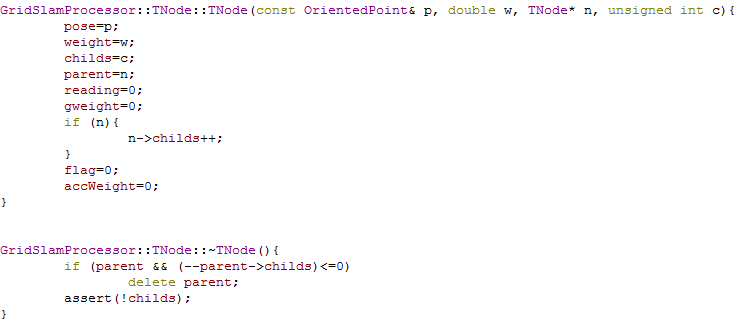


每个粒子均包含一幅地图，粒子的位姿，上一时刻的位姿，粒子的权重，粒子的累加权重，轨迹树中上个粒子的下标，以及轨迹树的入口节点等。

轨迹树的节点由TNode定义，包含该节点粒子的位姿pose，该节点粒子的权重weight，该节点所有子节点的累加权重accweight， 该节点的父节点指针parent，此时的激光数据reading，子节点个数childs。



需要注意的是TNode的构造函数和析构函数，在构造函数中还实现该节点的父节点的childs更新功能，由于每次产生新节点时，都会指定一个轨迹树中已经存在的节点作为父节点，当存在父节点时，构造函数中自动给父节点的childs加1。每次删除轨迹树中一个节点的时候，会自动进入析构函数，这是程序会将父节点的childs减一，再判断父节点是否小于等于零，是的话连父节点一起删除。因为轨迹树中的所有节点只能由子节点向父节点反向访问，如果一个节点的唯一子节点被删除，则会导致该节点无法被访问。



粒子的初始化在lasercallBack()第一次执行时调用initMapper()中实现。initMapper()中调用gsp\_->GridSlamProcessor::init()来对粒子进行初始化，粒子容器中总共30个粒子，初始位置均为（0,0,0）。m\_neff=30.0, m\_count=0, m\_readingcount=0, m\_linearDistance=m\_angularDistance=0;

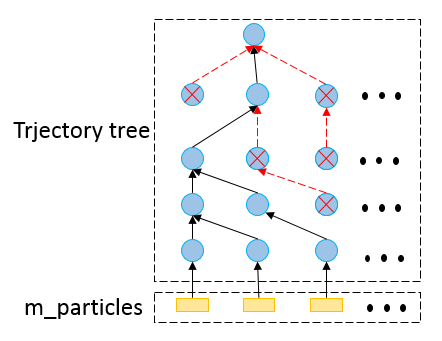
初始的轨迹树入口节点指向一个初始节点，该节点粒子位姿为（0,0,0），节点粒子权重为0，父节点为0，子节点数为零。reading、gweight、flag、accWeight均为零。



粒子的位姿在每次调用processScan()时，首先通过drawFromMotion()更新一次，并在processScan()调用结束时将上一时刻的位姿改为当前的位姿。

**轨迹树更新：**

每个粒子中均有一个节点指针指向轨迹树中的一个节点，轨迹树初始节点在初始化时给定，该节点粒子位姿为（0,0,0），节点粒子权重为0，父节点为0，所有粒子最开始均指向这个节点。随着程序运行，轨迹树按如下方式更新，在第一次进入processScan()时，由运动模型随机出每个粒子的位置，然后根据每个粒子的位置建立30个（共30个粒子）新的节点，每个新节点的父节点为初始节点。之后每次再进入processScan()时，轨迹树的更新发生在resample()函数中。在重采样所需要的粒子时，同时会产生新节点,节点的位姿是被选中的粒子，父节点是m\_particle中被选中粒子所指向的上一代节点，未被选中的粒子所指向的节点会被删除。所有的粒子更新后均指向一个最新一代的节点，每个节点通过parent能访问上一代的父节点，最终形成的结构如下图：



# 具体函数说明

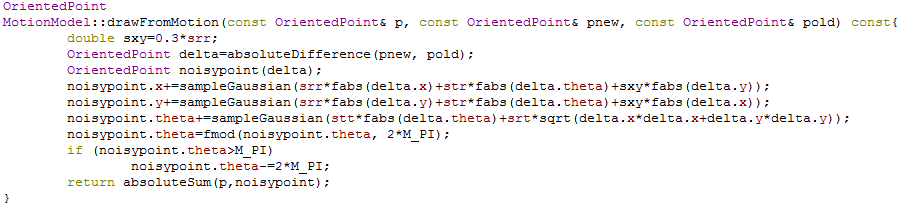
## addScan()---->processScan()

addscan()执行时，首先会处理扫描到的激光数据，得到一组叫reading的数据供下一步调用。然后函数processScan()会调用该数据进行进一步处理。

### processScan() ----> drawFromMotion()

processScan()处理的第一步是通过运动模型来更新每个粒子的位姿。通过函数drawFromMotion()来实现。

源代码如下：



**算法注释：**

该算法需读取***上一时刻当前粒子的位姿p***，***该时刻里程计显示位姿pnew***以及***上一时刻里程计所读位姿pold*。**

(1)该函数调用absoluteDifference()算出根据里程计显示的绝对偏移量，该偏移量被转化到pold坐标系。

(2)对该偏移量各个分量（x,y,theta）融入高斯白噪声；注：theta融入噪声后进行了fmod对2\*PI取余操作，并在取余操作后将theta转化到（-PI,PI）

(3)将融合噪声的偏移量叠加到粒子的位姿作为粒子的新位姿。

**疑问**：该算法认为在pold坐标系中的沿x,y的平移与旋转量theta分别会对在世界坐标系下的x,y,theta方向的运动以固定系数产生影响。即噪声与pold的位姿无关，只与pnew与pold的相对位姿有关。然而，当pold位姿取不同值时，对噪声的影响理应不同。《概率机器人中》5.4.1节所提供的另一种算法（ sample\_motion\_model\_odometry）更为准确，之后可尝试用该方法进行实现。不过最终粒子位姿都会通过激光雷达扫描数据进行校准，该算法的准确性应该对结果影响不大。

### processScan() ----> computeActiveArea()

通过运动模型更新每个粒子的位姿之后，接下来需要判断程序是否是第一次进入processScan()。如果是第一次进去，则只要进行**计算有效区域computeActiveArea()**和**计算占用概率栅格地图registerScan()**以及**更新粒子node和轨迹树**几个步骤。如果不是第一次进入，则需要进行**扫描匹配scanMatch()**,**权重计算updateTreeWeights()**以及**重采样resample()**三个步骤。

computeActiveArea()通过如下方式实现：

源代码如下(使用的是scanmacher.new.cpp下的函数)：

****

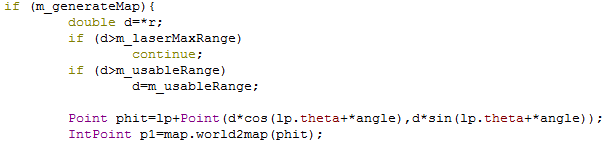
**算法注释：**

该算法需读取上一时刻**粒子现有地map**，**粒子位姿p**以及**当前激光雷达数据reading**。

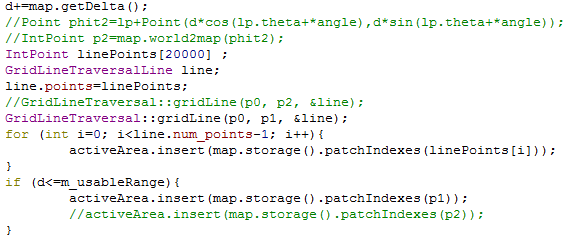
1. 在每次调用computeActiveArea()之前，都会先调用invalidateActiveArea()，在其中将m\_activeareaComputed赋值为false。computeActiveArea()首先计算出激光雷达在世界坐标系中的位姿，并转化到地图坐标系，记为p0。m\_laserpose应该是指激光雷达安装位置相对于机器人坐标系的位姿。



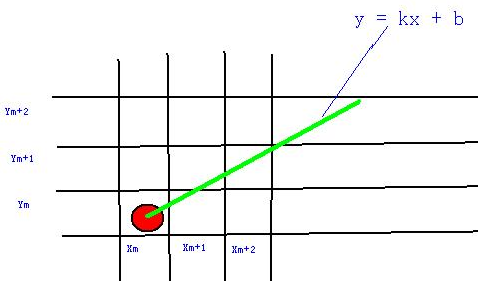
1. 对于每一条激光数据，首先判断m\_generateMap 是否为真，为真，则执行如下操作：
   * 1. 计算出激光探测点在地图中的位置，记为p1。



* + 1. 然后调用gridLine()计算激光束通过的地图上的点，将算出的点标记为activeArea。



* + 1. gridLine()中核心代码在gridLineCore()中实现。gridLineCore()中核心算法注释如下。



对每一束激光束，设start为该激光束起点，end为激光束端点（障碍物位置）算法原理如下：

设start、end所在直线方程为y = kx + b，我们以0<k<1的情况为例进行推导。记dy=|end.y-start.y|,dx=|end.x-start.x|。由于斜率小于1，选择在x方向进行步进。我们的目的就是要计算该束激光所经过的所有网格的坐标。假设当前x = 时，y = 。那么当x 执行一个单位步进时（即x = +1时），y等于还是等于+1更符合这个直线方程呢？我们可以通过比较y方向上两个网格与直线上真实坐标的差值来确定哪个更符合。

Dupper = + 1 - Yreal = ; 表示+1和y真实值的差

Ddown = Yreal -  = ; 表示和y真实值的差

记Diff = Ddown - Dupper

= 

我们就是要判断Diff的正负，如果Diff大于零，说明 + 1更接近真实值，反之则更接近。由于k=dy/dx，在方程两边同乘以dx，得到



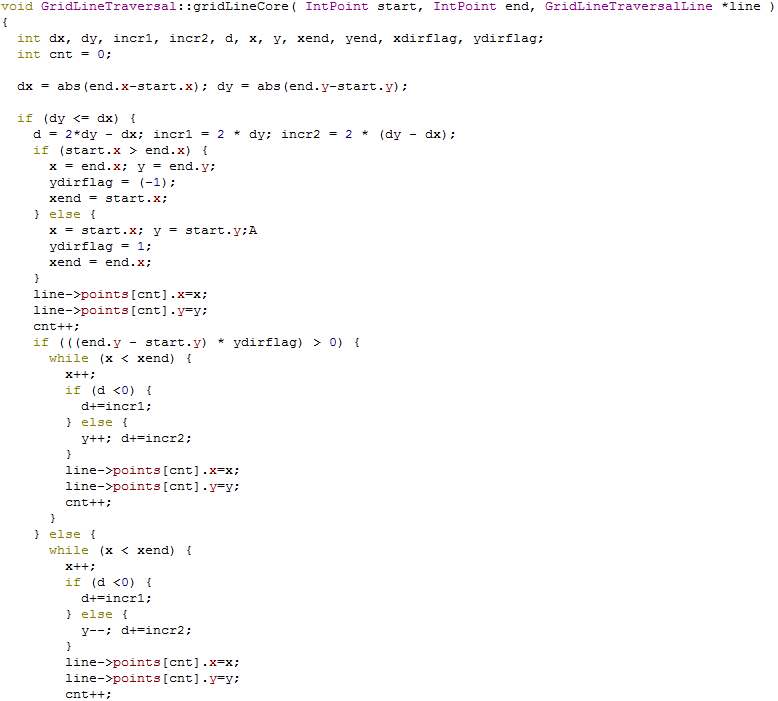
那么

算法循环判断，如果大于0，则下一个点的y坐标加1。在第一个点，带入初始点，及，可得



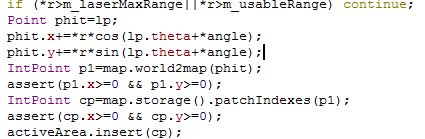
K取其他范围值时，计算思路与上述过程类似。在|k|>1时，程序需选取y方向执行单位步进。k<0时，非步进方向每次改变时应该是减一而非加一。

源代码如下：





如果m\_generateMap非真，则只将探测点标记为activeArea.



m\_generateMap在gsp\_地图初始化时被设为false,扫描匹配时一直是false。在updatemap()函数中新定义一个matcher,其中的m\_generateMap被置为true。所以粒子中保存的地图和updateMap()中输出的地图中包含信息不同。

### processScan() ----> registerScan()

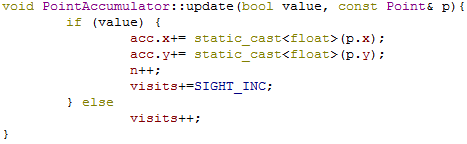
registerScan源码如下：



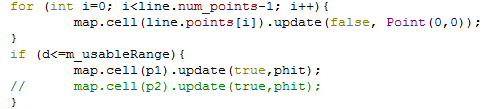
**算法注释：**

该算法需读取上一时刻**粒子现有地map**，**粒子位姿p**以及**当前激光雷达数据reading**。

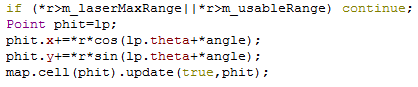
1. registerScan()函数进入后会首先调用computeActiveArea()。然后计算出激光雷达在世界坐标系中的位姿，并转化到地图坐标系，算法处理和computeActiveArea()基本类似。通过调用update()来对相应的网格特性进行更新。



1. 首先判断m\_generateMap 是否为真，为真，则执行如下操作，将光束经过的网格visits全部加一，如果量到的距离在有效量程之内，还需将障碍物所在网格p1的性质acc.x,acc.y,n,visits做相应变化。每扫到一次障碍物，障碍物在地图上的网格cell里的acc会将障碍物在世界坐标系中的位置累加进来，并且n加一，后面用到位置的时候取均值mean() const {return 1./n\*Point(acc.x, acc.y); p1处的visits会被加两次，n加一次.



如果m\_generateMap值为false，则只将障碍物所在网格phit的性质acc.x,acc.y,n,visits做相应变化。（phit是在世界坐标系，不在地图坐标系，应该需要转换吧？这个是scanmatcher.new.cpp中的实现，在另一处实现scanmatcher.cpp中的用的是p1而不是phit,此处应该是错误的。）



### processScan() ----> scanMatch()

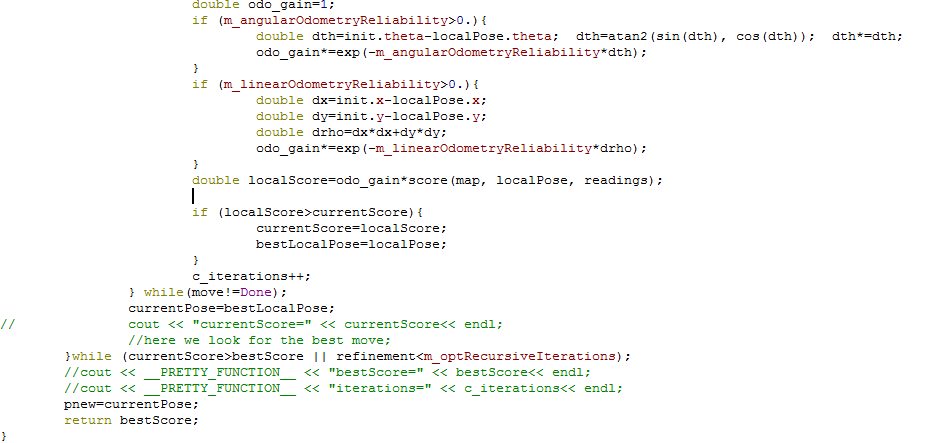
当程序不是第一次进入，则需要进行扫描匹配scanMatch(),权重计算updateTreeWeights()以及重采样resample()三个步骤。scanMach()函数则会依次调用optimize(),likelihoodAndScore(),computeActiveArea()。

对于每一个粒子scanMatch()首先定义一个位姿corrected,然后调用optimize()得到校正后的粒子位姿并返回bestscore.

#### scanMatch()---->optimize()

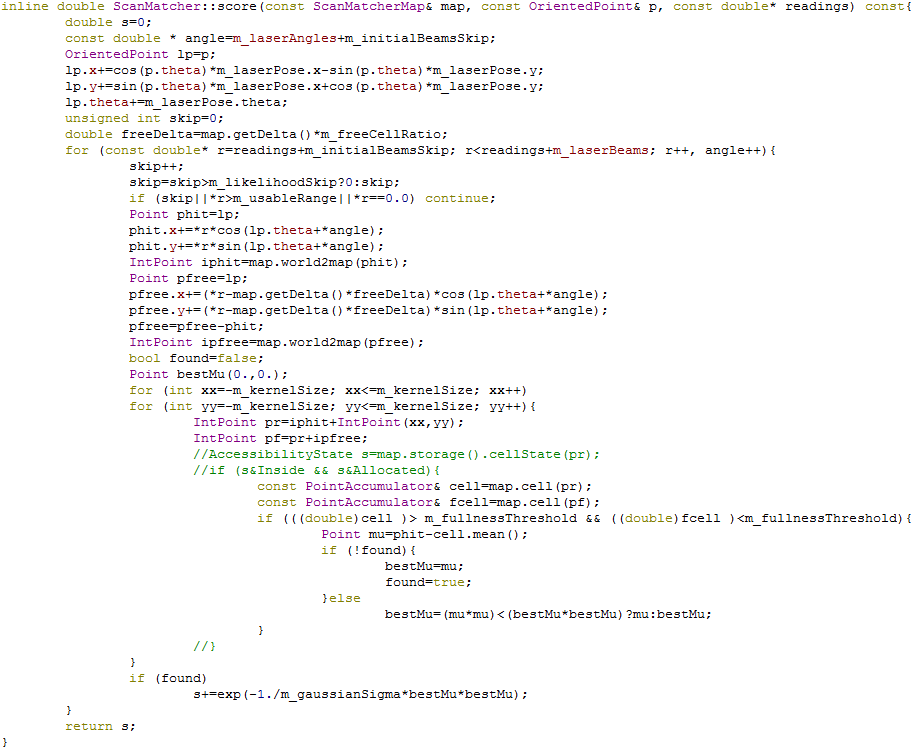
optimize()源码如下：





optimize()代码注释：

optimize()首先调用函数score()计算当前激光雷达采集数据的“概率”。score()实现方式如下：



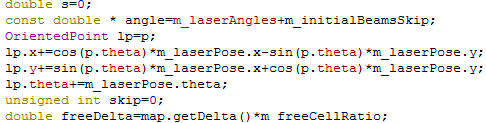
score()首先计算出雷达在世界坐标系下的位姿。

//m\_laserAngles是每一个波束对应角度的数组

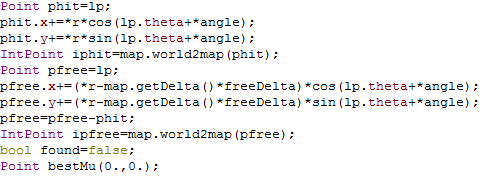
//p是机器人当前位姿（x, y, theta）

//lp代表雷达位姿，以下计算雷达在世界坐标系下的位姿

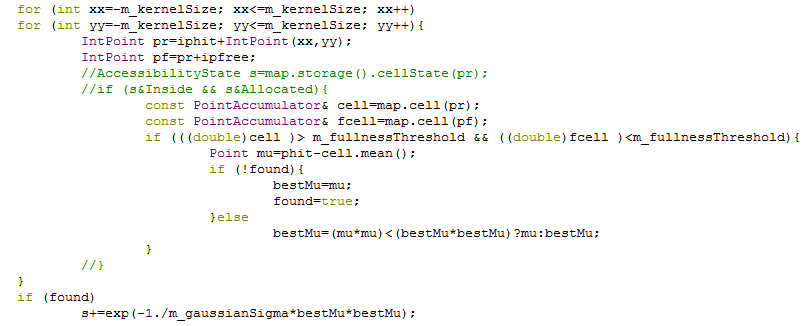
//mfreeCellRatio=sqrt(2.)



对于每一束激光，要计算出波束的探测点位iphit置和ipfree。（ipfree目的应该是要将激光探测点沿当前方向向里挪一定距离，但目前的这个取值为）不知道代表什么含义，如果是则比较好理解，是向里挪一个网格。以下是计算iphit和ipfree的代码。



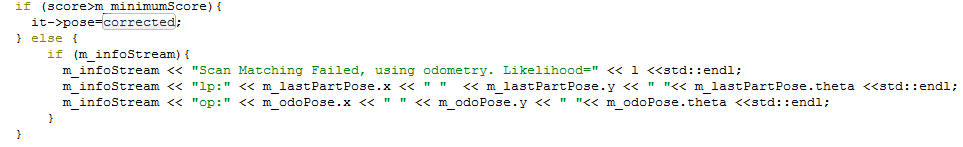
然后在以iphit为中心的九宫格中遍历，当一个网格满足以下条件：该网格被标记为障碍物（ ((double)cell）>m\_fullnessThreshold ）且该网格沿激光方向 （ ((double)fcell）< m\_fullnessThreshold ）往回走一段距离的网格没有被标记为障碍物时，满足上述条件并且离phit最近的点则被认为是真实障碍物的坐标点。（注：和当前障碍物坐标距离最近的以前(double)cell 在被访问过的网格返回值是n/visits,未访问过的网格返回值是-1.）然后计算出该真实点和phit的距离，“概率”计算类似于高斯分布公式为。（注：m\_gausianSigma=0.05）然后将每束激光的概率叠加得到该扫描的“概率”。当在九宫格中没有找到符合条件的点时，该束激光被略过，其“概率”不被叠加到总的概率中。在该算法中,只有当扫描到的点附近有之前被标记的障碍物时才会计算其概率，在激光扫描到未探索区域时，概率就会一直为零。



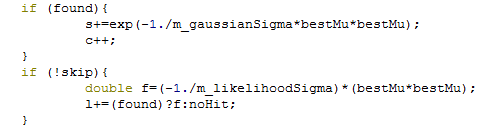
在通过score()计算完当前采集数据的“概率”后，optimize()通过如下方式确定该粒子的最优位置。通过让粒子前后左右平移以及左转右转来寻找附近使采集数据概率最大化的位置。最终得到粒子的最优位置pnew并返回采集数据的最优(大)“概率”bestscore.

#### scanMatch()---->likelihoodAndScore ()

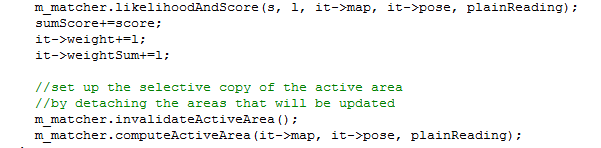
调用optimize()可得到粒子通过扫描数据校正后最优位置corrected以及该位置扫描数据的“概率”打分score。当score大于所规定的最小可信概率时，便将粒子的位置更新为corrected，否则仍采用原来里程计所给出的位置。实现如下：



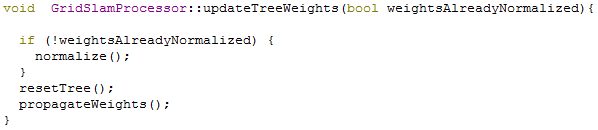
然后程序调用likelihoodAndScore ()，likelihoodAndScore实现原理和score基本一致，后面有些许不同。在score()中，当在九宫格中没有找到符合条件的点时，该束激光被略过，其“概率”不被叠加到总的概率中。而在likelihoodAndScore()中，定义了两个变量s和l，s与score()中的s相同，l的计算中则是在九宫格中没有找到符合条件的点时，该束激光的概率被认为是一个常值noHit。



调用完likelihoodAndScore()后，l被赋给粒子的weight。scanMatch()最终调用computeActiveArea()更新地图有效区域。



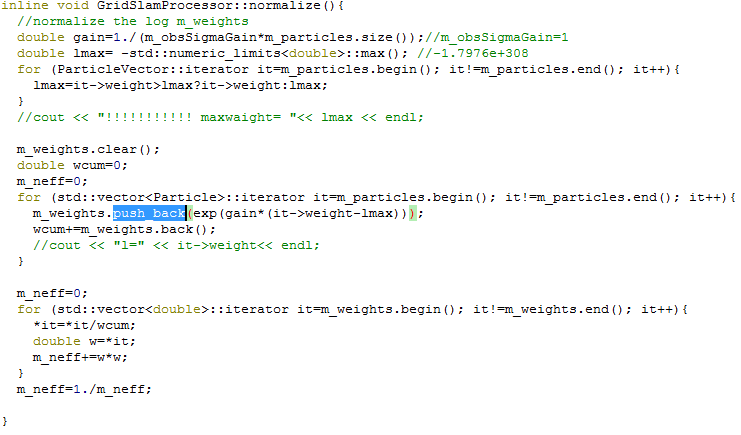
### processScan() ----> updateTreeWeights ()



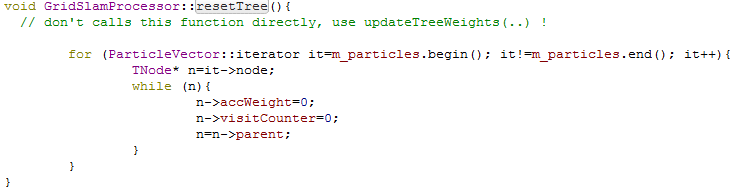
updateTreeWeights()分三个部分： normalize()， resetTree() 和propagateWeights()。

#### updateTreeWeights ()---->normalize()

normalize()首先寻找出各个粒子的最大weight,记为lmax，然后进行如下转换，将每个粒子的权重归一化。并计算出m\_neff来确定是否应该进行重采样。



resetTree()将轨迹树中所有的节点的accweight和visitCounter全部赋为零。



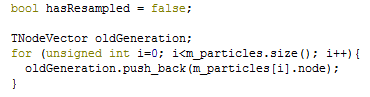
propagateWeights()中更新轨迹树中的accweight和visitCounter, accweight是指该节点所有子节点的权重的累加值，visitCounter是为了更新accweight时轨迹树向上遍历是内部使用，visitCounter等于childs是表示所有的子节点权重均已累加，就会向上遍历。



### processScan() ----> resample ()

代码说明：

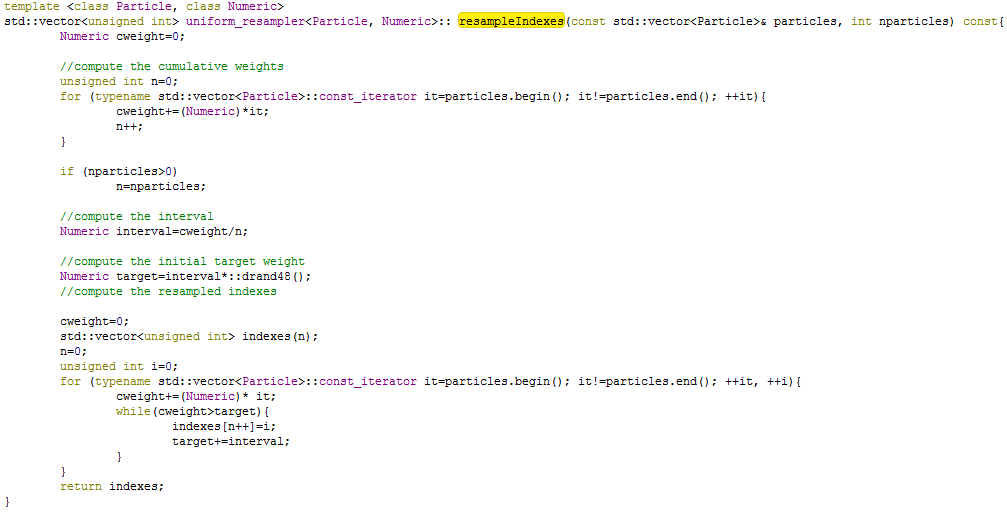
首先resample()定义变量hasResampled并定义为false，这个变量作为函数返回值，在进行重采样之后会被赋为true。然后函数将所有当前粒子存入oldGeneration中。



然后程序利用m\_neff判断是否应该进行重采样，m\_neff是在normalize()中求得的。这个判断是用来防止好的粒子被替换，具体原理参见[1,2]。在本程序实现中，当m\_neff < 0.5\*m\_particles.size()时，就进行重采样。

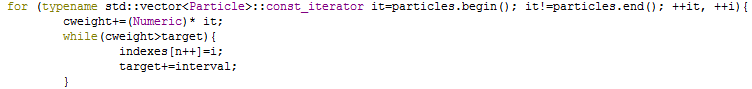


重采样的实现方法在函数resampleIndexes()中实现。



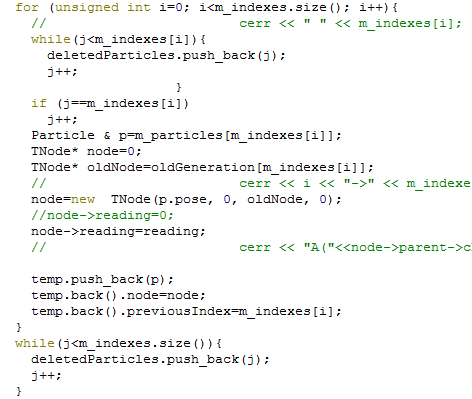
算法思路如下，先算出所有粒子权重的累加值记为cweight,再用cweight除以粒子数n得到粒子权重的平均值interval。之后随即产生一个0到1之间的随机数再乘以interval,记为target，target的取值则是在0到interval之间。进行n次抽签，定义容器indexes来存储被抽中的粒子下标。方法如下：

1. 将cweight置零
2. 进行如下循环：则得到的indexes为被抽中的粒子下标

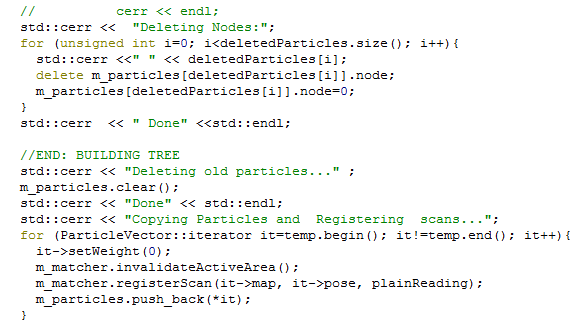


这个方法可以举例说明：假设总共三个粒子，权重分别为2.5, 0.2, 0.3。权重平均值为1，这时，我们产生的随机目标target在0到1之间随机分布。按照算法，indexes[1]=0,indexes[2]=0,若0<target<0.5,则indexes[3]=0。0.5<target<0.7时，indexes[3]=1。0.7<target<1时，indexes[3]=2。该算法保证如果一个粒子权重大于1倍粒子权重均值时，至少被抽中一次，大于2倍均值时，至少被抽中2次，以此类推。小于1倍均值，最多被抽中1次，小于2倍均值，最多被抽中2次，以此类推。

最终被抽中的粒子下标最终被存储到indexes中，通过一下算法找出需要删除的粒子的下标，存储在deletedParticles中,并将所有新选中的粒子存储在temp中。同时会产生新节点,节点的位姿是被选中的粒子，父节点是m\_particle中被选中粒子所指向的上一代节点，然后temp中的新的粒子的node指针会指向新产生的节点。



最后是更新m\_particles中存储的粒子，首先对于被淘汰的m\_particle中的粒子，删除其对应的轨迹树上的节点。然后清空m\_particle,temp中存储的新粒子存入m\_particle中，并将每个粒子的weight重置为0。

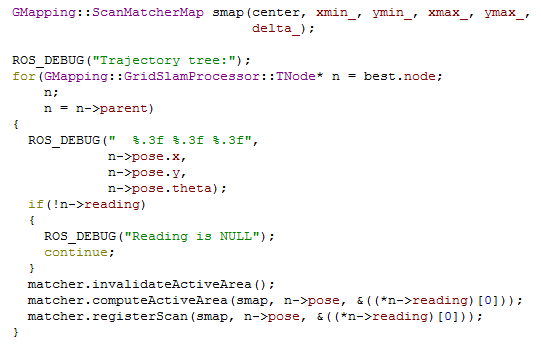


## updateMap()

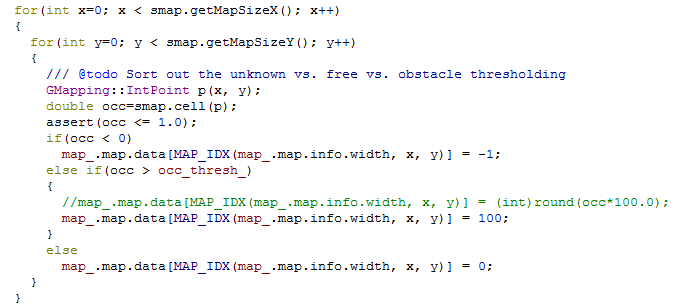
updateMap()在slam\_gmapping.cpp中实现，首先程序调用找出历史累加权重最大的粒子，记为best。



然后找到这个累加权重最大的粒子所指向的轨迹树的节点，并沿轨迹树向上遍历，并根据每个节点的位置与激光数据更新一幅新定义的地图smap。粒子中本身存储了一幅地图，但是gsp\_中的m\_generateMap是false,所以粒子更新地图调用computeActiveArea()和registerScan()时只更新障碍物所在网格。而updateMap()中重新定义了一个matcher并将其m\_generateMap置为true,所以更新smap时会将障碍物和非障碍物区域一起更新。



最终输出的地图只区分地图上每个网格是一下哪种情况：unknow vs.obstacle vs.free。对于之前定义的地图smap，遍历地图上的每一个网格，通过调用“double occ=smap.cell(p);”（返回值应该是PointAccumulator中的inline operator double() const { return visits?(double)n\*SIGHT\_INC/(double)visits:-1; }）得到每个网格的占有率，占有率小于零代表未被访问，输出的地图中相应网格值被赋为-1，占有率大于occ\_tresh小于1零代表障碍，输出的地图中相应网格值被赋为100，占有率小于occ\_tresh代表无障碍，输出的地图中相应网格值被赋为0。



# 参考文献

[1] Grisetti G, Stachniss C, Burgard W. Improved techniques for grid mapping with rao-blackwellized particle filters[J]. IEEE transactions on Robotics, 2007, 23(1): 34-46.

[2] A. Doucet, N. de Freitas, and N. Gordan, editors. Sequential Monte-Carlo Methods in Practice. Springer Verlag, 2001