lsd-slam

`lsd-slam`

Sophus/sophus

这几行内容只是一些简单的类型定义或者声明，其中使用了trait技法，不明白trait技法的好处的朋友可以去翻看stl源码解析书上的相关内容

```
    namespace Sophus {
    template<typename _Scalar, int _Options=0> class SO3Group;
    typedef EIGEN_DEPRECATED SO3Group<double> SO3;
    typedef SO3Group<double> SO3d;
    typedef SO3Group<float> SO3f;
    }

    template<typename _Scalar, int _Options>
    struct traits<Map<Sophus::SO3Group<_Scalar>, _Options> >
        : traits<Sophus::SO3Group<_Scalar, _Options> > {
      typedef _Scalar Scalar;
      typedef Map<Quaternion<Scalar>,_Options> QuaternionType;
    };

    template<typename _Scalar, int _Options>
    struct traits<Map<const Sophus::SO3Group<_Scalar>, _Options> >
        : traits<const Sophus::SO3Group<_Scalar, _Options> > {
      typedef _Scalar Scalar;
      typedef Map<const Quaternion<Scalar>,_Options> QuaternionType;
    };
    }
    }
```

SO3GroupBase

SO3是一个群，所谓群就是数学意义上的那个群，SO3正好是一个乘法群而已，满足封闭性，单位元，结合律等，实际上学会这个类主要是要明白它和旋转是如何联系的，exp映射和log映射是什么意思，以及邻接关系是什么，在这之前，让我们先来看下这样一段函数类型定义

```
template<typename Derived>
class SO3GroupBase {
public:
  typedef typename internal::traits<Derived>::QuaternionType &
  QuaternionReference;
  typedef const typename internal::traits<Derived>::QuaternionType &
  ConstQuaternionReference;

  static const int DoF = 3;
  static const int num_parameters = 4;
  static const int N = 3;
```

```
typedef Matrix<Scalar,N,N> Transformation;
typedef Matrix<Scalar,3,1> Point;
typedef Matrix<Scalar,DoF,1> Tangent;
typedef Matrix<Scalar,DoF,DoF> Adjoint;
```

这个部分的程序，无非是定义了一些数据了类型以及三个常数，前面的数据类型是两个四元数的引用类型(注意记住这里很重要，待会儿明白它是如何实现的，这里是关键)
接下来定义自由度，由于是三维空间的旋转所以只有3个自由度，四元数有4个参数，以及变换矩阵是3*3矩阵
最后定义的是要用到的数据类型，分别是3*3的旋转矩阵，3*1的空间点，3*1的角速度，还有3*3的邻接矩阵

下面这个函数是得到邻接关系的函数，请看代码

```
1.        inline  const Adjoint Adj() const {
2.          return matrix();
3.          }
```

追踪进去之后得到的是

```
1.        inline
2.        const Transformation matrix() const {
3.          return unit_quaternion().toRotationMatrix();
4.        }
```

这个代码在文件的211行，表达的是一个单位四元数到一个旋转矩阵的变换，再追踪进去，在文件的291行

```
1.        ConstQuaternionReference unit_quaternion() const {
2.      return static_cast<const Derived*>(this)->unit_quaternion();
3.    }
```

然后你会发现，这里看起来做了一个超级危险的操作，之后就不知道做了啥了
实际上，这个地方是在做一件事，即调用Eigen库里面封装得四元数
咋调用的呢？很简单，如果认真浏览一下这个类，会发现，这个类里面没有实实在在的真正使用的数据，仅仅只是有一些static函数和一些内联函数，而后面的SO3Group是继承了这个类型
在SO3Group中，实实在在定义了一个四元数的成员(在代码的677行)，换句话说，我们真正使用的类型是SO3Group,这个类型可以调用基类的Ad()这个函数，调用之后，它只会调用基类的unit_quaternion()函数，但是这个函数会把指针转化为派生类的指针，然后调用派生类的函数(注意到继承类的继承方式是*class SO3Group : public SO3GroupBase >*)，所以实际上调用的还是派生类自身的unit_quaternion()在(665行)，这个函数返回了成员变量unit_quaternion_然后通过这个成员变量，调用了Eigen库内部的四元数算法，这就是基类调用子类函数的经典实现方式

```
1.        QuaternionBase<Derived>::toRotationMatrix(void) const {
2.        Matrix3 res;
3.
4.        const Scalar tx  = Scalar(2)*this->x();
5.        const Scalar ty  = Scalar(2)*this->y();
6.        const Scalar tz  = Scalar(2)*this->z();
7.        const Scalar twx = tx*this->w();
8.        const Scalar twy = ty*this->w();
9.        const Scalar twz = tz*this->w();
```

```
        const Scalar txx = tx*this->x();
        const Scalar txy = ty*this->x();
        const Scalar txz = tz*this->x();
        const Scalar tyy = ty*this->y();
        const Scalar tyz = tz*this->y();
        const Scalar tzz = tz*this->z();

        res.coeffRef(0,0) = Scalar(1)-(tyy+tzz);
        res.coeffRef(0,1) = txy-twz;
        res.coeffRef(0,2) = txz+twy;
        res.coeffRef(1,0) = txy+twz;
        res.coeffRef(1,1) = Scalar(1)-(txx+tzz);
        res.coeffRef(1,2) = tyz-twx;
        res.coeffRef(2,0) = txz-twy;
        res.coeffRef(2,1) = tyz+twx;
        res.coeffRef(2,2) = Scalar(1)-(txx+tyy);

        return res;
      }
```

也就是这个算法，这个算法本身是很简单的，只要你明白四元数是如何表示旋转的，那么写一个函数，将四元数转化为旋转矩阵，就是这个矩阵了，这是其中一个邻接关系，类中还定义了另一个邻接，看下面这段代码

```
        inline static
        const Adjoint d_lieBracketab_by_d_a(const Tangent & b) {
          return -hat(b);
        }
```

这个又是返回邻接矩阵，传入值是一个角速度，这个邻接矩阵不同于刚才(把四元数转化为旋转矩阵),这个矩阵是把一个角速度转化为角速度矩阵的函数，由于角速度和向量相乘是叉积，但有时候需要把这样的运算表示成矩阵乘法的方式，由此，需要把角速度(向量，实际上是一阶反对称张量)转化为矩阵的形式(实际上是还原成本来的张量表达形式)，这个的操作方式是很简单的，如下:

```
        inline static
        const Transformation hat(const Tangent & omega) {
          Transformation Omega;
          Omega <<  static_cast<Scalar>(0), -omega(2),  omega(1)
              ,  omega(2),     static_cast<Scalar>(0), -omega(0)
              , -omega(1),  omega(0),     static_cast<Scalar>(0);
          return Omega;
        }
```

接下来看一下四元数是如何求逆的，下面这个函数就是求逆的函数，实际上就是共轭的四元数，然后用这个四元数构造了个SO3Group的对象

```
        /**
         * \returns group inverse of instance
         */
        inline
        const SO3Group<Scalar> inverse() const {
          return SO3Group<Scalar>(unit_quaternion().conjugate());
        }
```

最后我们来看下两个重要的映射关系

```
inline
const Tangent log() const {
  return SO3Group<Scalar>::log(*this);
}
```

这个函数跟踪进去会到475行的log()函数，之后会发现这个log函数调用了logAndTheta()函数，也就是这个函数

```
inline static
const Tangent logAndTheta(const SO3Group<Scalar> & other,
                          Scalar * theta) {
  const Scalar squared_n
      = other.unit_quaternion().vec().squaredNorm();
  const Scalar n = std::sqrt(squared_n);
  const Scalar w = other.unit_quaternion().w();

  Scalar two_atan_nbyw_by_n;

  if (n < SophusConstants<Scalar>::epsilon()) {

    if (std::abs(w) < SophusConstants<Scalar>::epsilon()) {
      throw SophusException("Quaternion is not normalized!");
    }
    const Scalar squared_w = w*w;
    two_atan_nbyw_by_n = static_cast<Scalar>(2) / w
                         - static_cast<Scalar>(2)*(squared_n)/(w*squared_w);
  } else {
    if (std::abs(w)<SophusConstants<Scalar>::epsilon()) {
      if (w > static_cast<Scalar>(0)) {
        two_atan_nbyw_by_n = M_PI/n;
      } else {
        two_atan_nbyw_by_n = -M_PI/n;
      }
    }else{
      two_atan_nbyw_by_n = static_cast<Scalar>(2) * atan(n/w) / n;
    }
  }

  *theta = two_atan_nbyw_by_n*n;

  return two_atan_nbyw_by_n * other.unit_quaternion().vec();
}
```

我觉得应该明白这个函数是干啥用的，看返回值，很清楚的说明了这个函数是为了得到一个旋转的角速度或者说得到转轴(两个同向，实际上它是得到了so(3)群里面的那个对应于旋转的元素)，直观上可以认为传入值实际上是另一个四元数，另一个输入作为输出参数的值，代表了角速度的大小，具体计算方案实际上是李群里面的一组映射中的对数映射，从SO(3)向so(3)的映射

```
  inline static
  const SO3Group<Scalar> exp(const Tangent & omega) {
    Scalar theta;
    return expAndTheta(omega, &theta);
  }
```

```
6.
7.
8.
9.        inline static
10.       const SO3Group<Scalar> expAndTheta(const Tangent & omega,
11.                                          Scalar * theta) {
12.         const Scalar theta_sq = omega.squaredNorm();
13.         *theta = std::sqrt(theta_sq);
14.         const Scalar half_theta = static_cast<Scalar>(0.5)*(*theta);
15.
16.     Scalar imag_factor;
17.     Scalar real_factor;;
18.     if((*theta)<SophusConstants<Scalar>::epsilon()) {
19.       const Scalar theta_po4 = theta_sq*theta_sq;
20.       imag_factor = static_cast<Scalar>(0.5)
21.                     - static_cast<Scalar>(1.0/48.0)*theta_sq
22.                     + static_cast<Scalar>(1.0/3840.0)*theta_po4;
23.       real_factor = static_cast<Scalar>(1)
24.                     - static_cast<Scalar>(0.5)*theta_sq +
25.                     static_cast<Scalar>(1.0/384.0)*theta_po4;
26.     } else {
27.       const Scalar sin_half_theta = std::sin(half_theta);
28.       imag_factor = sin_half_theta/(*theta);
29.       real_factor = std::cos(half_theta);
30.     }
31.
32.     return SO3Group<Scalar>(Quaternion<Scalar>(real_factor,
33.                                                imag_factor*omega.x(),
34.                                                imag_factor*omega.y(),
35.                                                imag_factor*omega.z()));
36.       }
```

这两个函数或许又会让你痛苦很长一段时间，因为它又是李群的映射运算，而且这个代码还有一点小bug
这两个函数的核心其实就是下面那个函数，它其实是遵从以下公式得到的

$$G_1 = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{pmatrix}, G_2 = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ -1 & 0 & 0 \end{pmatrix}, G_3 = \begin{pmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

$$\begin{aligned} \boldsymbol{\omega} &\in \mathbb{R}^3 \\ \boldsymbol{\omega}_1 G_1 + \boldsymbol{\omega}_2 G_2 + \boldsymbol{\omega}_3 G_3 &\in \mathrm{so}(3) \end{aligned}$$

$$\exp\left(\boldsymbol{\omega}_\times\right) = \mathbf{I} + \sum_{i=0}^{\infty} \left[\frac{\boldsymbol{\omega}_\times^{2i+1}}{(2i+1)!} + \frac{\boldsymbol{\omega}_\times^{2i+2}}{(2i+2)!} \right]$$

$$
\begin{aligned}
\exp(\boldsymbol{\omega}_\times) &= \mathbf{I} + \left(\sum_{i=0}^{\infty} \frac{(-1)^i \theta^{2i}}{(2i+1)!}\right) \boldsymbol{\omega}_\times + \left(\sum_{i=0}^{\infty} \frac{(-1)^i \theta^{2i}}{(2i+2)!}\right) \boldsymbol{\omega}_\times^2 \\
&= \mathbf{I} + \left(1 - \frac{\theta^2}{3!} + \frac{\theta^4}{5!} + \cdots\right) \boldsymbol{\omega}_\times + \left(\frac{1}{2!} - \frac{\theta^2}{4!} + \frac{\theta^4}{6!} + \cdots\right) \boldsymbol{\omega}_\times^2 \\
&= \mathbf{I} + \left(\frac{\sin\theta}{\theta}\right) \boldsymbol{\omega}_\times + \left(\frac{1 - \cos\theta}{\theta^2}\right) \boldsymbol{\omega}_\times^2
\end{aligned}
$$

人生苦短，证明都见鬼去吧，阅读代码的时候应该注意代码有点小bug，在这里:

```
if((*theta)<SophusConstants<Scalar>::epsilon()) {
  const Scalar theta_po4 = theta_sq*theta_sq;
  imag_factor = static_cast<Scalar>(0.5)
                - static_cast<Scalar>(1.0/48.0)*theta_sq
                + static_cast<Scalar>(1.0/3840.0)*theta_po4;
  real_factor = static_cast<Scalar>(1)
                - static_cast<Scalar>(0.5)*theta_sq +
                static_cast<Scalar>(1.0/384.0)*theta_po4;
}
```

其实if这里面这一大块是Taylor公式，上面是sin(x)/x的，下面是cos(x)的，但值得注意的是，这里的x都是取的半角，也就是x/2，所以指数部分还要多乘以一个1/2^n,但是**static_cast(0.5)*theta_sq**这个地方作者忘了乘以这个系数，其他地方都是乘了的

DataStructures

最科学的欣赏源码方式，必然是先看内存管理，即进入文件夹下的第二个文件FrameMemory.h
这个文件只有一个类FrameMemory,接口也不多,这个类偷偷地管理了每一帧的所有内存，第一个函数是getINstance();
注意到构造函数被私有化，显然这是初始化对象的函数，这个函数实现就两行，你会发现内部维护的是一个static的对象，然后返回这个对象，也就是说，一个进程里面，有且仅有一个FrameMemory的对象,也就是所谓的单例(这个是弱化的单例)

```
    FrameMemory& FrameMemory::getInstance()
    {
        static FrameMemory theOneAndOnly;
        return theOneAndOnly;
    }
```

接下来我们来看真正做内存管理一组函数

FrameMemory::getBuffer

首先进入这个函数的时候，直接调用boost里面的互斥锁，把这段函数里面的这段内存锁上了
这里会用到两个成员变量：

```
    std::unordered_map< void*, unsigned int > bufferSizes;
    std::unordered_map< unsigned int, std::vector< void* > > availableBuffers;
```

从字面上来说，这是两个无序映射
我上c++官网上查了下，实际上就是hash映射
(http://www.cplusplus.com/referece/unordered_map/unordered_map/)

之后判断可用buffer中是否有sizeInByte这个大小的内存，如果有，那么返回1，没有返回0，所以，搜索到会进入if内部，否则进入else内部
进入if内部:首先是获取sizeInByte所对应的value的引用，也就是需要的内存的首地址，之后会判断

- 如果是空的，那么会调用allocateBUffer申请一段内存，注意在allocateBuffer内部调用了Eigen的内存管理函数(底层实际上还是malloc，如果失败会抛出一个throw_std_bad_alloc)，之后做一个映射，把buffer的首地址和尺寸映射起来，之后返回buffer的首地址，这样便可以得到一个buffer
- 如果不是空，那么直接得到一个那个尺寸的内存，然后返回

```
std::vector< void* >& availableOfSize = availableBuffers.at(sizeInByte);
if (availableOfSize.empty())
{
    void* buffer = allocateBuffer(sizeInByte);
    return buffer;
}
else
{
    void* buffer = availableOfSize.back();
    availableOfSize.pop_back();
    return buffer;
}
```

进入else : 如果没有这个尺寸的内存，就调用allocateBuffer申请一段，之后返回

```
void* buffer = allocateBuffer(sizeInByte);
assert(buffer != 0);
return buffer;
```

有申请就有释放，剩下两个同组函数，returnBuffer只是把内存还回去(放入map中),而真正释放是releaseBuffer,这里就不细讲了

还有三个函数，分别是

```
boost::shared_lock<boost::shared_mutex> activateFrame(Frame* frame);
void deactivateFrame(Frame* frame);
void pruneActiveFrames();
```

他们都是对成员***std::list activeFrames*** 操作的，这个操作也带了一个锁***boost::mutex activeFramesMutex***，请看代码

```
    boost::shared_lock<boost::shared_mutex> FrameMemory::activateFrame(Frame* frame)
    {
        boost::unique_lock<boost::mutex> lock(activeFramesMutex);
        if(frame->isActive)
            activeFrames.remove(frame);
        activeFrames.push_front(frame);
        frame->isActive = true;
        return boost::shared_lock<boost::shared_mutex>(frame->activeMutex);
    }
```

这个函数是用来激活某一帧的，如果它已经在激活表中，就把他放在前面，然后返回这一帧里面的自带互斥锁

```
void FrameMemory::deactivateFrame(Frame* frame)
{
    boost::unique_lock<boost::mutex> lock(activeFramesMutex);
    if(!frame->isActive) return;
    activeFrames.remove(frame);

    while(!frame->minimizeInMemory())
        printf("cannot deactivateFrame frame %d, as some acvite-lock is lingering. May cause deadlock!\n", frame->id());    // do it in a loop, to make shure it is really, really deactivated.

    frame->isActive = false;
}
```

停用帧，这个函数首先将要停用的帧从activeFrame中移除，然后调用frame->minimizeInMemory()，这个函数里面有个线程锁，所以要确保这个函数被真的调用，因而循环调用

```
void FrameMemory::pruneActiveFrames()
{
    boost::unique_lock<boost::mutex> lock(activeFramesMutex);

    while((int)activeFrames.size() > maxLoopClosureCandidates + 20)
    {
        if(!activeFrames.back()->minimizeInMemory())
        {
            if(!activeFrames.back()->minimizeInMemory())
            {
                printf("failed to minimize frame %d twice. maybe some active-lock is lingering?\n",activeFrames.back()->id());
                return;  // pre-emptive return if could not deactivate.
            }
        }
        activeFrames.back()->isActive = false;
        activeFrames.pop_back();
    }
}
```

修剪帧，假如activeFrame的参数已经比了loopClosure的候选的最大帧数(loopClosure是slam里面的重要组成之一)还要多20，那么进入修剪，如果修剪两次失败，直接返回

Frame

帧这玩意儿贯穿始终，是slam中最基本的数据结构，我觉得想要理解这个类，应该从类中的结构体Data开始,从程序上可以看出，定义了一个金字塔，PYRAMID_LEVELS这个宏在setting.h头文件中

```
#define SE3TRACKING_MIN_LEVEL 1
#define SE3TRACKING_MAX_LEVEL 5

#define SIM3TRACKING_MIN_LEVEL 1
#define SIM3TRACKING_MAX_LEVEL 5

#define QUICK_KF_CHECK_LVL 4

#define PYRAMID_LEVELS (SE3TRACKING_MAX_LEVEL > SIM3TRACKING_MAX_LEVEL ? SE3TRACKING_MAX_LEVEL :
SIM3TRACKING_MAX_LEVEL)
```

根据上面那一段，目测是等于5了，data里面，定义了帧的id,宽度高度等各种各样的参数，这些参数在函数中都有所访问，浏览完了这些参数后，实际上你会看到帧里面还有两个互斥锁，是否活的flag，以及大量公有变量，这些变量之后我都会有所介绍

构造函数有两个，主要是最后一个参数给的有所不同，实际上它代表的是两种不同格式的图片数据
进入构造函数之后，会调用initialize函数

```
initialize(id, width, height, K, timestamp);
```

在这个initialize函数中，对图像内存，位姿参数，相机参数，相机逆参等进行了初始化
相机参数

```
data.K[0] = K;
data.fx[0] = K(0,0);
data.fy[0] = K(1,1);
data.cx[0] = K(0,2);
data.cy[0] = K(1,2);
```

相机逆参

```
data.KInv[0] = K.inverse();
data.fxInv[0] = data.KInv[0](0,0);
data.fyInv[0] = data.KInv[0](1,1);
data.cxInv[0] = data.KInv[0](0,2);
data.cyInv[0] = data.KInv[0](1,2);
```

初始化金字塔

```
data.width[level] = width >> level;
data.height[level] = height >> level;

```

```
data.imageValid[level] = false;
data.gradientsValid[level] = false;
data.maxGradientsValid[level] = false;
data.idepthValid[level] = false;
data.idepthVarValid[level] = false;

data.image[level] = 0;
data.gradients[level] = 0;
data.maxGradients[level] = 0;
data.idepth[level] = 0;
data.idepthVar[level] = 0;
data.reActivationDataValid = false;
```

初始化相机金字塔

```
if (level > 0)
{
    data.fx[level] = data.fx[level-1] * 0.5;
    data.fy[level] = data.fy[level-1] * 0.5;
    data.cx[level] = (data.cx[0] + 0.5) / ((int)1<<level) - 0.5;
    data.cy[level] = (data.cy[0] + 0.5) / ((int)1<<level) - 0.5;

    data.K[level]  << data.fx[level], 0.0, data.cx[level], 0.0, data.fy[level], data.cy[level], 0.0, 0.0, 1.0;  // synthetic
    data.KInv[level] = (data.K[level]).inverse();

    data.fxInv[level] = data.KInv[level](0,0);
    data.fyInv[level] = data.KInv[level](1,1);
    data.cxInv[level] = data.KInv[level](0,2);
    data.cyInv[level] = data.KInv[level](1,2);
}
```

初始化函数结束之后，自然会在构造函数中将图像拷贝到Frame中来，注意存储格式为float

```
data.image[0] = FrameMemory::getInstance().getFloatBuffer(data.width[0]*data.height[0]);
float* maxPt = data.image[0] + data.width[0]*data.height[0];
for(float* pt = data.image[0]; pt < maxPt; pt++)
{
    *pt = *image;
    image++;
}

data.imageValid[0] = true;

privateFrameAllocCount++;
```

值得注意的是，这里的内存管理是之前讲过的FrameMemory的对象管理的，在这里就调用了getFloatBuffer函数，同样的，析构函数也是如此，内存管理是靠FrameMemory，这里只是回收内存，以后不特别说明，都默认为回收内存

```
FrameMemory::getInstance().deactivateFrame(this);

if(!pose->isRegisteredToGraph)
    delete pose;
else
```

```
    pose->frame = 0;

for (int level = 0; level < PYRAMID_LEVELS; ++ level)
{
    FrameMemory::getInstance().returnBuffer(data.image[level]);
    FrameMemory::getInstance().returnBuffer(reinterpret_cast<float*>(data.gradients[level]));
    FrameMemory::getInstance().returnBuffer(data.maxGradients[level]);
    FrameMemory::getInstance().returnBuffer(data.idepth[level]);
    FrameMemory::getInstance().returnBuffer(data.idepthVar[level]);
}

FrameMemory::getInstance().returnBuffer((float*)data.validity_reAct);
FrameMemory::getInstance().returnBuffer(data.idepth_reAct);
FrameMemory::getInstance().returnBuffer(data.idepthVar_reAct);
```

由于有5层金字塔，所以需要循环回收，最后再释放permaRef_colorAndVarData和permaRef_posData,这两个参数是位置，颜色和方差的引用，用于重定位，注意：这个参数只是在initialize中初始化为空指针

在介绍其他函数之前，我觉得应该县说明一下几个辅助函数，因为这些函数总是在很多函数中被调用,他们是:

```
 void Frame::require(int dataFlags, int level)
 void Frame::release(int dataFlags, bool pyramidsOnly, bool invalidateOnly)
 bool Frame::minimizeInMemory()
```

第一个是某种请求函数，第二个是某种释放函数，最后一个是最小化储存函数

首先是require(int, int),这个函数实际上在判断需要怎样的数据，然后调用相应的build函数

Frame::buildImage

首先是递归构建底层金字塔，因为上层金字塔是以底层为基础的

```
if (level == 0)
{
    printf("Frame::buildImage(0): Loading image from disk is not implemented yet! No-op.\n");
    return;
}

require(IMAGE, level - 1);
```

递归到底层后，调用buildMutex互斥锁，之后才是判断这个等级的金字塔是否已经构建，然后向内存管理的对象申请内存，之后构建整个金字塔

```
float* dest = data.image[level];
    int wh = width*height;
    const float* s;
    for(int y=0;y<wh;y+=width*2)
    {
        for(int x=0;x<width;x+=2)
        {
            s = source + x + y;
```

```
            *dest = (s[0] +
                    s[1] +
                    s[width] +
                    s[1+width]) * 0.25f;
            dest++;
        }
    }

    data.imageValid[level] = true;
```

方案是下层金字塔的四个像素的值的平均值合并成一个(同样的，实际上releaseImage也是相应的写法)，我想另外说明的一个构建函数是buildGradients(int)函数
这个函数前面部分和所有build函数是一样的，值得注意的是梯度的计算

```
const float* img_pt = data.image[level] + width;
const float* img_pt_max = data.image[level] + width*(height-1);
Eigen::Vector4f* gradxyii_pt = data.gradients[level] + width;

// in each iteration i need -1,0,p1,mw,pw
float val_m1 = *(img_pt-1);
float val_00 = *img_pt;
float val_p1;

for(; img_pt < img_pt_max; img_pt++, gradxyii_pt++)
{
    val_p1 = *(img_pt+1);

    *((float*)gradxyii_pt) = 0.5f*(val_p1 - val_m1);
    *(((float*)gradxyii_pt)+1) = 0.5f*(*(img_pt+width) - *(img_pt-width));
    *(((float*)gradxyii_pt)+2) = val_00;

    val_m1 = val_00;
    val_00 = val_p1;
}
```

第一个维度存储的是左右两个像素点的梯度，是中心差分，第二个维度储存了上下两个像素点的梯度，也是中心差分，第三个维度存储了当前图像的值，最后一个维度没有存储数据

最后一个介绍一个build函数应当好好介绍buildIDepthAndIDepthVar(int level)
这个函数一次性build了深度的均值和方差(高斯分布)

```
int sw = data.width[level - 1];

const float* idepthSource = data.idepth[level - 1];
const float* idepthVarSource = data.idepthVar[level - 1];
float* idepthDest = data.idepth[level];
float* idepthVarDest = data.idepthVar[level];

for(int y=0;y<height;y++)
{
    for(int x=0;x<width;x++)
    {
        int idx = 2*(x+y*sw);
        int idxDest = (x+y*width);
```

```
14.
15.         float idepthSumsSum = 0;
16.         float ivarSumsSum = 0;
17.         int num=0;
18.
19.         // build sums
20.         float ivar;
21.         float var = idepthVarSource[idx];
22.         if(var > 0)
23.         {
24.             ivar = 1.0f / var;
25.             ivarSumsSum += ivar;
26.             idepthSumsSum += ivar * idepthSource[idx];
27.             num++;
28.         }
29.
30.         var = idepthVarSource[idx+1];
31.         if(var > 0)
32.         {
33.             ivar = 1.0f / var;
34.             ivarSumsSum += ivar;
35.             idepthSumsSum += ivar * idepthSource[idx+1];
36.             num++;
37.         }
38.
39.         var = idepthVarSource[idx+sw];
40.         if(var > 0)
41.         {
42.             ivar = 1.0f / var;
43.             ivarSumsSum += ivar;
44.             idepthSumsSum += ivar * idepthSource[idx+sw];
45.             num++;
46.         }
47.
48.         var = idepthVarSource[idx+sw+1];
49.         if(var > 0)
50.         {
51.             ivar = 1.0f / var;
52.             ivarSumsSum += ivar;
53.             idepthSumsSum += ivar * idepthSource[idx+sw+1];
54.             num++;
55.         }
56.
57.         if(num > 0)
58.         {
59.             float depth = ivarSumsSum / idepthSumsSum;
60.             idepthDest[idxDest] = 1.0f / depth;
61.             idepthVarDest[idxDest] = num / ivarSumsSum;
62.         }
63.         else
64.         {
65.             idepthDest[idxDest] = -1;
66.             idepthVarDest[idxDest] = -1;
67.         }
68.     }
69. }
```

这一整块就是构建过程的计算，你会发现各种倒数还有各种加权平均数，看起来特别繁琐，实际上这里的深度是"逆深度"(论文上翻译而来)，简单说就是深度的倒数，为啥用"逆深度"，这个实际上在论文里面有详细说明，我就不想多做解释(反正对阅读代码没有太多障碍)，简单说他这是怎么算的呢？
首先计算所有的方差的倒数，并把他们加起来，同时用这个倒数乘以逆深度得到一种加权数据，也把这些数据加起来 也就是上面的4个if部分，然后如果这个四个点有可用的num>0，
那么深度就等于float depth = ivarSumsSum / idepthSumsSum,取倒数得到逆深度
最后的不确定度(方差)为idepthVarDest[idxDest] = num / ivarSumsSum

接下来介绍一下设置深度这个函数Frame::setDepth
进入这个函数之后，首先调用了锁，把数据都锁了起来，然后申请金字塔第一层的内存，之后对金字塔第一层进行拷贝，注意在lsd-slam中，深度是在用高斯分布进行逼近的，所以拷贝的时候，自然会拷贝一个均值和一个方差，值得注意的是，他把平滑后的版本拷贝到了未平滑版本中,如果某点是缺失的(也就是说没有估计值)，那么那一点的值填为-1

```
float* pyrIDepth = data.idepth[0];
float* pyrIDepthVar = data.idepthVar[0];
float* pyrIDepthMax = pyrIDepth + (data.width[0]*data.height[0]);

float sumIdepth=0;
int numIdepth=0;

for (; pyrIDepth < pyrIDepthMax; ++ pyrIDepth, ++ pyrIDepthVar, ++ newDepth) //, ++ pyrRefID)
{
    if (newDepth->isValid && newDepth->idepth_smoothed >= -0.05)
    {
        *pyrIDepth = newDepth->idepth_smoothed;
        *pyrIDepthVar = newDepth->idepth_var_smoothed;

        numIdepth++;
        sumIdepth += newDepth->idepth_smoothed;
    }
    else
    {
        *pyrIDepth = -1;
        *pyrIDepthVar = -1;
    }
}
```

最后计算这一帧的平均深度和总共有多少个点被估计出了深度

```
meanIdepth = sumIdepth / numIdepth;
numPoints = numIdepth;
```

下面这个函数比较重要，是计算的准备操作，主要是用来设置相机坐标变换的一个函数，传入参数分别是：哪一帧，这一帧到参考帧的相似变换矩阵，相机参数，以及金字塔等级

首先通过另外那一帧储存的变换矩阵计算出了乘以相机内参的投影矩阵以及平移向量

```
Sim3 otherToThis = thisToOther.inverse();

//otherToThis = data.worldToCam * other->data.camToWorld;
K_otherToThis_R = K * otherToThis.rotationMatrix().cast<float>() * otherToThis.scale();
otherToThis_t = otherToThis.translation().cast<float>();
K_otherToThis_t = K * otherToThis_t;
```

然后记录通过传入变换矩阵计算得到的这一帧到参考帧的乘以相机内参后的投影矩阵和旋转矩阵,并计算距离，最后记录金字塔等级和参考帧ID

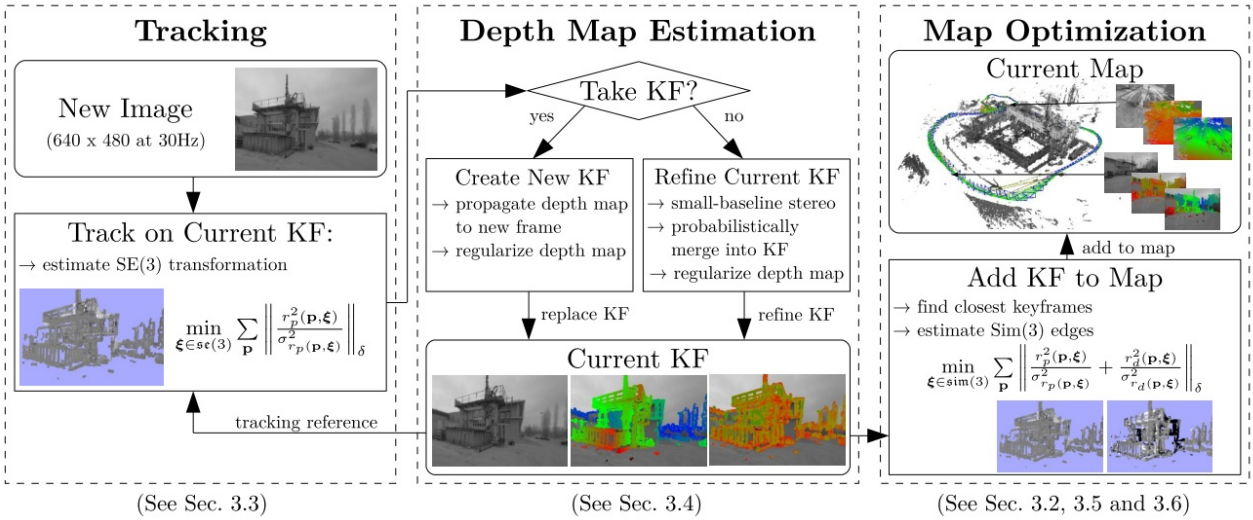
```
thisToOther_t = thisToOther.translation().cast<float>();
K_thisToOther_t = K * thisToOther_t;
thisToOther_R = thisToOther.rotationMatrix().cast<float>() * thisToOther.scale();
otherToThis_R_row0 = thisToOther_R.col(0);
otherToThis_R_row1 = thisToOther_R.col(1);
otherToThis_R_row2 = thisToOther_R.col(2);

distSquared = otherToThis.translation().dot(otherToThis.translation());

referenceID = other->id();
referenceLevel = level;
```

FramePoseStruct

FramePoseStruct.hpp和.cpp文件相对来说是最简单的两个文件了
这里面主要定义的是和优化相关的一些数据，比如说和tracking相关的trackingParent,和g2o相关的bool isRegisteredToGrap，bool isOptimizedi，bool isInGraph以及VertexSim3* graphVertex，还有一些设置变换的方法，希望读者自行解读

算法解析

当你对Frame有一定的了解后，自然希望知道这个算法是如何构建的(所谓程序=数据结构+算法)
需要先了解整个slam是如何运作的



我直接把论文中的图片截了过来，实际上整个过程分为三大模块，第一个模块是Tracking,第二个模块是Depth Map Estimation, 第三个模块是Map Optimization，这三个过程分别位于论文的3.3，3.4和3下的其他部分(实际上Optimization可以说是整个算法的核心部分)

Tracking

回想之前写过的数据结构Frame，里面包含了图像，深度和深度方差，即

$$\mathcal{K}_i = (I_i, D_i, V_i),$$

式中下标i表示关键帧的id,I表示从图像向一个实数的映射，D表示深度图到正实数的映射，V表示深度的方差到正实数的映射(注意这里的深度都是逆深度，简单称呼成深度)

当一个新的图像(或者说新的观测)输入时，认为它与关键帧之间有个se(3)的变换关系，这个变换关系我们需要根据关键帧和当前图像的数据优化得到，优化方程为

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

$$\text{with} \quad r_p(\mathbf{p}, \boldsymbol{\xi}_{ji}) := I_i(\mathbf{p}) - I_j(\omega(\mathbf{p}, D_i(\mathbf{p}), \boldsymbol{\xi}_{ji}))$$

$$\sigma^2_{r_p(\mathbf{p},\boldsymbol{\xi}_{ji})} := 2\sigma_I^2 + \left(\frac{\partial r_p(\mathbf{p}, \boldsymbol{\xi}_{ji})}{\partial D_i(\mathbf{p})} \right)^2 V_i(\mathbf{p})$$

$$\omega(\mathbf{p}, d, \boldsymbol{\xi}) := \begin{pmatrix} x'/z' \\ y'/z' \\ 1/z' \end{pmatrix} \quad \text{with} \quad \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} := \exp_{\mathfrak{se}(3)}(\boldsymbol{\xi}) \begin{pmatrix} \mathbf{p}_x/d \\ \mathbf{p}_y/d \\ 1/d \\ 1 \end{pmatrix}$$

$$\|r^2\|_\delta := \begin{cases} \frac{r^2}{2\delta} & \text{if } |r| \leq \delta \\ |r| - \frac{\delta}{2} & \text{otherwise.} \end{cases}$$

这似乎看起来是比较抽象的几个方程，实际上可以简单地理解把r理解为图像误差，sigma理解为不确定度，w表示从图像向keyFrame的一个映射(先映射之后，发现可能对应点没有重合，出现误差),最后是误差的衡量边准，E是总的误差。
最后实际上我们就通过优化E，算出当前观测到的图像到关键帧的变换，从而得到当前的位置姿态信息

Depth Map Estimation

tracking成功之后，就进入深度估计过程，首先是判断，是不是新建关键帧，论文中如此写道：

> If the camera moves too far away from the existing map, a new keyframe is created from the most recent tracked image.

就是说关键帧的创建是根据相机移动来判断的，如果相机移动了足够远，那么就重新创建一个关键帧，否则就refine当前的关键帧
事实上，这个过程有一整套很复杂的过程，在论文Semi-Dense Visual Odometry for a Monocular Camera中，且让我慢慢道来

深度估计主要有三个过程，分别是：
1. 用立体视觉方法来从先前的帧得到新的深度估计
2. 深度图帧与帧之间的传播
3. 部分规范化已知深度

基于立体视觉对深度的更新

首先，跟新深度我们需要选择参考帧，让我们的深度估计尽可能精确，但是与此用时，希望差异搜索范围和观测角度尽可能小(效率高)，这个算法中使用了一种自适应的方法:

> We use the oldest frame the pixel was observed in, where the disparity search range and the observation angle do not exceed a certain threshold (see Fig. 4). If a disparity search is unsuccessful (i.e., no good match is found), the pixel's "age" is increased, such that subsequent disparity searches use newer frames where the pixel is likely to be still visible.

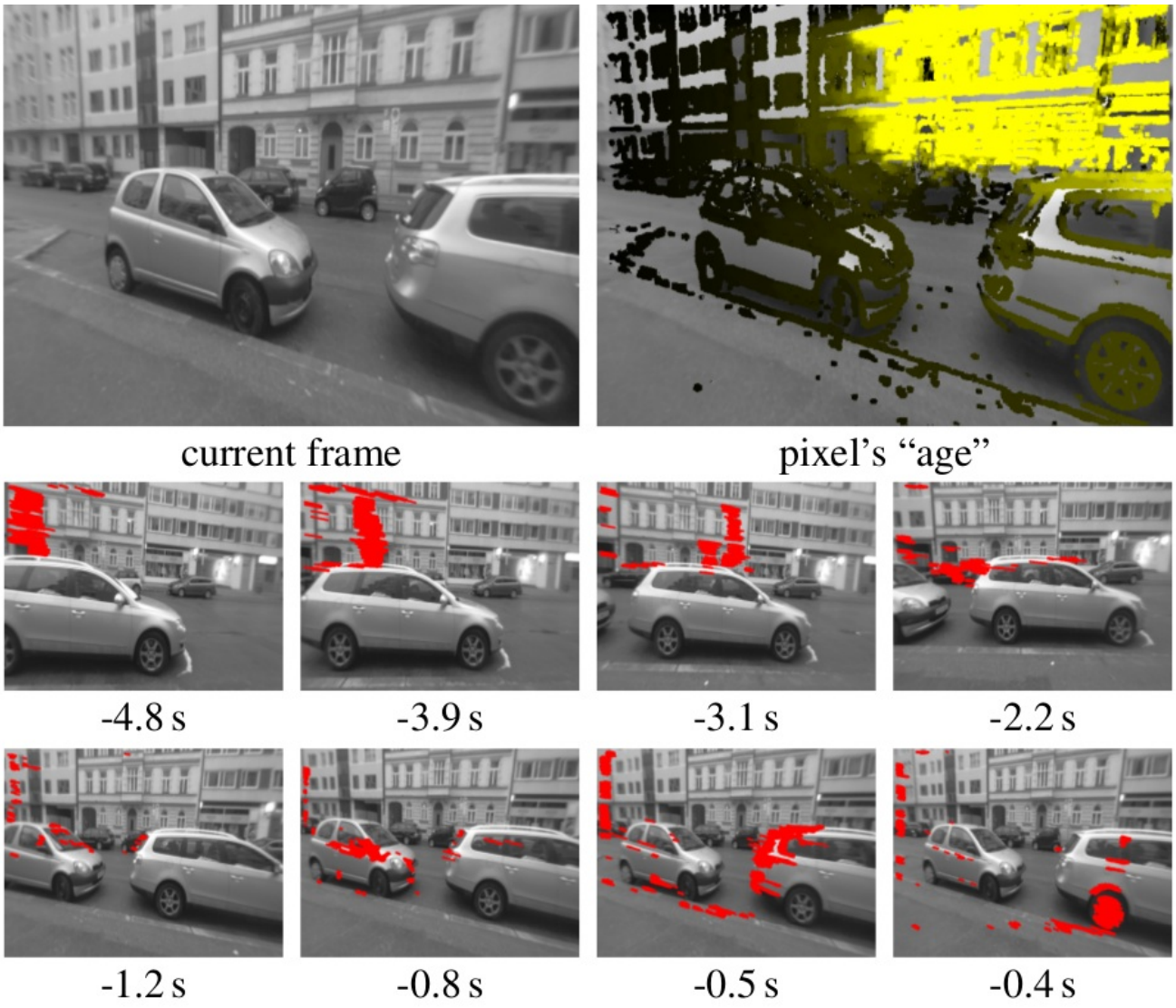


Figure 4. Adaptive Baseline Selection: For each pixel in the new frame (top left), a different stereo-reference frame is selected, based on how long the pixel was visible (top right: the more yellow, the older the pixel.). Some of the reference frames are displayed below, the red regions were used for stereo comparisons.

实际上就是搜索到最先看到这些像素的帧，一直到当前帧的前一帧作为参考帧，如果搜索失败，说明匹配很差，那么就增大像素的"年龄"，让它在新的能够看到这些像素的帧里面能够被搜索到

确定了搜索帧的范围之后，知道如何对每一帧进行操作。
其实方法很简单，对于每个像素（我估计是有效的像素），沿着极线方向搜索匹配，论文上有讲到，如果深度估计是可用的，那么搜索范围限制到均值加减2倍方差的区间，如果不可用，那么所有的差异性都需要被搜索，并使用，最后使用SSD误差作匹配算法

说明了以上两点之后，我们需要说明深度的到底是如何计算的。

就实际上而言，我们需要考虑从两幅图片I0,I1以及他们相对的朝向kesi还有投影矩阵pi中得到一个最好的深度估计值，在得到这个估计值的同时，计算出它的可靠度，即

$$d^* = d(I_0, I_1, \xi, \pi)$$
$$\sigma_d^2 = J_d \Sigma J_d^T$$

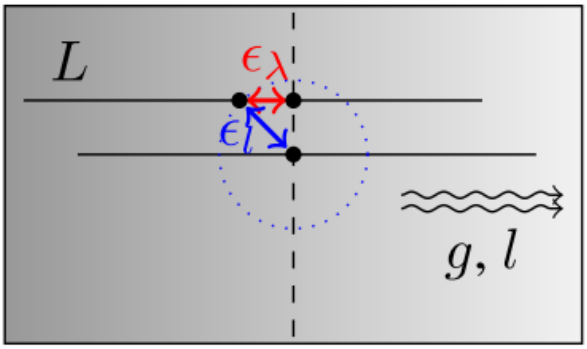
就实际运作而言，也是分为三大步：
1. 参考帧上极线的计算
2. 极线上得到最好的匹配位置
3. 通过匹配位置计算出最佳的深度

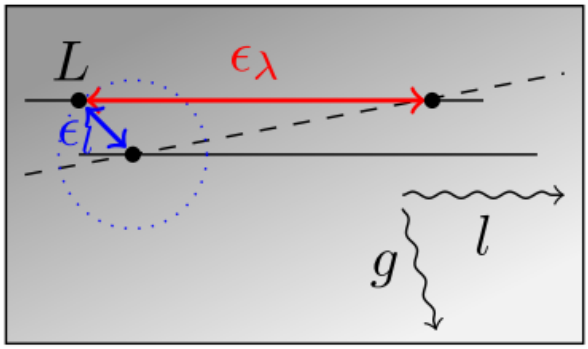
在这个三大步中，第一步需要得到几何差异误差，第二步需要得到图像差异误差，最后一步需要根据基线量化误差

几何误差 几何误差来源于相对朝向以及投影矩阵，假设极线L属于R2空间，可以用下面的表达式定义：

$$L := \{l_0 + \lambda \begin{pmatrix} l_x \\ l_y \end{pmatrix} | \lambda \in S\}$$

其中S是差异性区间，(lx,ly)^T是极线的单位化后的方向向量，l0表示一个对应无限深度的点。我们现在假设线单元上只有直线位置，l0受到高斯噪声，在实际应用中，我们需要极线尽可能短，旋转的影响尽可能小，使得这个估计尽可能好





如图所示，虚线是图像的等值线，等值线上的两个点，是同样深度的点，L是极线，e_l极线的高斯误差，e_λ是几何误差，明显的是，如果L平行与梯度方向，实际上误差会比较小，如果几乎垂直了，误差会很大
因此论文上有这样一个"定义为"的公式:

$$l_0 + \lambda^* \begin{pmatrix} l_x \\ l_y \end{pmatrix} \stackrel{!}{=} g_0 + \gamma \begin{pmatrix} -g_y \\ g_x \end{pmatrix}$$

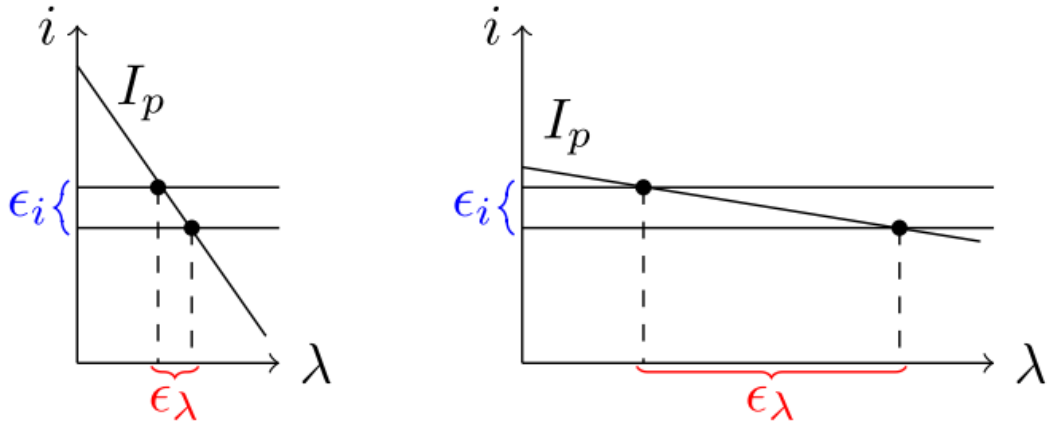
g_0表示等值线上的某个点，$g = (g_x, g_y)^T$表示图像的梯度方向
将上面式子l_0移项，然后两边同时点乘g,之后将分母移到等号右边可以得到最后的λ的优化表达式：

$$\lambda^*(l_0) = \frac{\langle g, g_0 - l_0 \rangle}{\langle g, l \rangle}$$

然后根据协方差传递的公式，最终可以得到：

$$\sigma^2_{\lambda(\xi,\pi)} = J_{\lambda^*(l_0)} \begin{pmatrix} \sigma_1^2 & 0 \\ 0 & \sigma_2^2 \end{pmatrix} J^T_{\lambda^*(l_0)}$$

图像误差



如图所示，如果梯度灰度梯度绝对值越大，对差异影响越小，反过来，如果绝对值越小，影响反而会很大，把这个公式化，我们实际上是要找到一个lambda,使得他们的差异最小，即

$$\lambda^* = \min_{\lambda} \left(i_{ref} - I_p(\lambda)\right)^2$$

其中i_{ref}是参考的灰度，$I_p(\lambda)$是当前图像在极线方向变化λ时的灰度，g_p是梯度

直观理解

我试图说明这个想法，让这个理解起来比较直观，简单说，测量都是不精确的(如果你能有完全精确的测量方法，请告诉我，让我膜拜几十年)，因此我们希望的是能够在有扰动的情况下，让这个扰动对深度的影响足够小，也就是让λ变化足够小(深度预测总是和极线相关的)，因此我们试图寻找下降的最快的方向(即梯度方向)，或者让梯度足够大
可以发现，第一个方案就是寻找了梯度的方向，第二个方案是让梯度足够大

像素到深度的转换

通常极线λ 与深度d是成比例的，我们计算总的观测不确定度为：

$$\sigma^2_{d,\,obs} = \alpha^2 \left(\sigma^2_{\lambda(\xi,\pi)} + \sigma^2_{\lambda(I)}\right)$$

其中：$\alpha := \frac{\delta_d}{\delta_\lambda}$

我会告诉你，这个实际上就是有限增量公式吗？

深度观测融合

我们使用贝叶斯迭代算法(对于高斯分布实际上就是拓展卡尔曼滤波)，来更新深度的估计，最终得到分布为：

$$\mathcal{N}\left(\frac{\sigma_p^2 d_o + \sigma_o^2 d_p}{\sigma_p^2 + \sigma_o^2}, \frac{\sigma_p^2 \sigma_o^2}{\sigma_p^2 + \sigma_o^2}\right)$$

深度的前向传播

我们假设两帧之间相机的旋转很小，那么新的深度d1可以用下面公式计算

$$d_1(d_0) = (d_0^{-1} - t_z)^{-1}$$

$$\sigma_{d_1}^2 = J_{d_1} \sigma_{d_0}^2 J_{d_1}^T + \sigma_P^2 = \left(\frac{d_1}{d_0}\right)^4 \sigma_{d_0}^2 + \sigma_P^2$$

t_z表示相机的平移，σ_p表示预测的不确定度

深度图的正则化

对于每一帧，我们使用一个正则迭代，把每个像素与其周边的加权深度作为改点的深度值，假如两个邻接深度之间的差值远大于2sigmod，他们便不做这个处理。但我们需要记住，处理之后，各自的方差不变

创建KeyFrame

现在回到主流程Tracking后的处理，如果是要创建关键帧，那么这个是很容易的，论文中是这么写的

> Once a new frame is chosen to become a keyframe, its depth map is initialized by projecting points from the previous keyframe into it, followed by one iteration of spatial regularization and outlier removal as proposed in 9. Afterwards, the depth map is scaled to have a mean inverse depth of one - this scaling factor is directly incorporated into the sim(3) camera pose. Finally, it replaces the previous keyframe and is used for tracking subsequent new frames

就是说，新的关键帧需要之前的关键帧将点投影过来(投影方案已经在深度前传介绍过)，得到这一帧的有效点，深度通过sim(3)变换投影均值和缩放因子，最后用这个关键帧替换掉之前的关键帧

refine Depth Map

如果不创建关键帧，那么就用当前的观测对之前的深度进行修正，论文原文

> A high number of very efficient small- baseline stereo comparisons is performed for image regions where the expected stereo accuracy is sufficiently large, as described in 9. The result is incorporated into the existing depth map, thereby refining it and potentially adding new pixels – this is done using the filtering approach proposed in 9.

实际上修正方法就是之前介绍的深度估计部分，使用贝叶斯后验概率算法，对深度进行修正，在这里也不再复述

Map Optimization

我估计看到这里，可能刚刚松了一口气，但是不得不说的是，我们进入了最后一个环节，全局地图优化
它的背景是这样的，单目slam由于它的绝对尺度信息是不能直接得到的(一只眼睛很难确定远近)，导致长距离运动之后，会产生巨大的尺度漂移。为了解决这个问题，我们需要对地图进行全局优化

首先我们需要插图关键帧到地图当中，要插入关键帧，自然需要知道什么时候需要插入，那么我们需要定义一个距离

> $$dist(\xi_{ji}) := \xi_{ji}^T W \xi_{ji},\ \xi \in se(3)$$

这里的W是一个对角阵，表示每个维度的权重，我们设定一个阈值，加入这个距离大于设定的阈值，那么就需要插入一帧，这个阈值实际上和当前场景有关，同时我们需要保证它足以满足小基线立体相机的要求

在插入帧的同时，我们还需要知道两帧之间是如何变换的，由于我们是单目slam，尺度漂移几乎是不可避免的，因此如果在这么"大"的一个尺度上，还是用se(3)，可能会导致两帧之间的变换不那么准确，因此我们放出一个尺度的自由度，使用sim(3)来衡量两帧之间的变换，也就是说，我们需要找到一个sim(3)群中的变换$\boldsymbol{\xi}$，使得下面定义这个误差最小：

$$E(\boldsymbol{\xi}_{ji}) := \sum_{\mathbf{p} \in \Omega_{D_i}} \left\| \frac{r_p^2(\mathbf{p}, \boldsymbol{\xi}_{ji})}{\sigma^2_{r_p(\mathbf{p}, \boldsymbol{\xi}_{ji})}} + \frac{r_d^2(\mathbf{p}, \boldsymbol{\xi}_{ji})}{\sigma^2_{r_d(\mathbf{p}, \boldsymbol{\xi}_{ji})}} \right\|_\delta$$

$$r_d(\mathbf{p}, \boldsymbol{\xi}_{ji}) := [\mathbf{p}']_3 - D_j([\mathbf{p}']_{1,2})$$

$$\sigma^2_{r_d(\mathbf{p}, \boldsymbol{\xi}_{ji})} := V_j([\mathbf{p}']_{1,2}) \left(\frac{\partial r_d(\mathbf{p}, \boldsymbol{\xi}_{ji})}{\partial D_j([\mathbf{p}']_{1,2})} \right)^2 + V_i(\mathbf{p}) \left(\frac{\partial r_d(\mathbf{p}, \boldsymbol{\xi}_{ji})}{\partial D_i(\mathbf{p})} \right)^2$$

那么如何去寻找要插入的位置呢？方法很简单，首先去寻找所有可能相似的关键帧，并计算视觉意义上的相似度，之后对这些帧进行排序，得到最相似的那几帧，然后根据上面那个方程算出sim(3),相似度衡量公式:

$$e(\boldsymbol{\xi}_{j_k i}, \boldsymbol{\xi}_{i j_k}) := (\boldsymbol{\xi}_{j_k i} \circ \boldsymbol{\xi}_{i j_k})^T \left(\boldsymbol{\Sigma}_{j_k i} + \mathrm{Adj}_{j_k i} \boldsymbol{\Sigma}_{i j_k} \mathrm{Adj}_{j_k i}^T \right)^{-1} (\boldsymbol{\xi}_{j_k i} \circ \boldsymbol{\xi}_{i j_k})$$

如果这个数值足够小(这个相似度足够高)，那么这一帧便插入map中，最后执行图优化(g2o)边为连接关系，节点为关键帧，即优化：

$$E(\boldsymbol{\xi}_{W1} \dots \boldsymbol{\xi}_{Wn}) := \sum_{(\boldsymbol{\xi}_{ji}, \boldsymbol{\Sigma}_{ji}) \in \mathcal{E}} (\boldsymbol{\xi}_{ji} \circ \boldsymbol{\xi}_{Wi}^{-1} \circ \boldsymbol{\xi}_{Wj})^T \boldsymbol{\Sigma}_{ji}^{-1} (\boldsymbol{\xi}_{ji} \circ \boldsymbol{\xi}_{Wi}^{-1} \circ \boldsymbol{\xi}_{Wj}).$$

Tracking

LGSX

于是就进入了Tracking文件夹里的第一个类型LSGX，这里面有3个类，分别是LSG4,LSG6和LSG7，他们定义了4个参数6个参数以及7个参数的最小二乘法
我想值得注意的是这段代码：

```
inline void update(const Vector6& J, const float& res, const float& weight)
 {
   A.noalias() += J * J.transpose() * weight;
   b.noalias() -= J * (res * weight);
   error += res * res * weight;
   num_constraints += 1;
 }
```

这里使用了Eigen库里面的noalias机制，为了讲解这个用法，请看下面这段代码

```
#include <iostream>
 #include <stdio.h>

 #include <eigen3/Eigen/Dense>

 using namespace std;

 template <typename T>
 class A {
 public:
     A() {printf("A()\n");}
     ~A() {printf("~A()\n");}
     template <typename U>
     A operator*(A<U>&){
         A a;
         return a;
     }

 };

 int main()
 {
     A<int> a1,b1;
     A<int> c1;
     c1 = a1*b1;
     Eigen::Matrix2f a, b, c;

     c.noalias() = a * b;
     a(0, 0) = 0;
     a(0, 1) = 1;
     a(1, 0) = 1;
     a(1, 1) = 0;
     b(0, 0) = 1;
     b(0, 1) = 2;
     b(1, 0) = 3;
     b(1, 1) = 4;
     c.noalias()  = a * b;
     std::cout << c << std::endl;
```

```
39.
40.         return 0;
41.     }
```

这是程序运行结果：

```
A()
A()
A()
A()
~A()
3 4
1 2
~A()
~A()
~A()
```

什么意思呢？注意我自己写的模板类A，在main函数里面有一个乘法c1 = a1*b1;，这个会调用重载运算符，然后先构造一个对象，再拷贝，最后析构，这样对于大量使用继承的程序会大大影响效率(构造基类构造子类，析构基类，析构子类，而且如果有内存申请，拷贝，释放，效率会降低更多)，那怎么办呢？Eigen使用了一种**机制**，让我们**减少一次对象的构造**，这个机制就是用noalias()调用，这样相当于吧运行a*b的结果直接拷贝到c里面，不再去构造中间对象
如何实现呢，其实由于要考虑到泛化，代码内部实现比较复杂，但原理实际上是写一个算法类，算法类中储存了算法的函数对象(仿函数)或者函数指针和储存算法运算的结果容器的引用或者指针，然后调用算法的时候，不再去构造那个复杂的数据结构，转而构造一个简单的算法类，将运行结果赋值给容器的引用或指针，从而降低内存开销,程序中调用noalias()函数，实际上返回的是一个NoAlias的对象，这个对象维护了真正的乘法

TrackingReference

void importFrame(Frame* source)

让我们来看第一个函数void importFrame(Frame* source);

```
void TrackingReference::importFrame(Frame* sourceKF)
{
    boost::unique_lock<boost::mutex> lock(accessMutex);
    keyframeLock = sourceKF->getActiveLock();
    keyframe = sourceKF;
    frameID=keyframe->id();


    // reset allocation if dimensions differ (shouldnt happen usually)
    if(sourceKF->width(0) * sourceKF->height(0) != wh_allocated)
    {
        releaseAll();
        wh_allocated = sourceKF->width(0) * sourceKF->height(0);
    }
    clearAll();
    lock.unlock();
}
```

前面几行是不难理解的，后面是一个判断，就是判断是否需要重置资源，如果宽度和高度的乘积和先前的高度与宽度的乘积一样大，那么就不用重新配置了，直接用先前的资源即可，如果不同，那么就释放先前左右的资源(请自行查看releaseAll())，然后把资源配置的记录记录成当前的配置数，最后再清空点云的记录数据numData(实际上就是令它等于0，请自行查看clearAll())

---

void makePointCloud(int level);

第二个重要的函数就是这个void makePointCloud(int level)了
我想前面几行都是不难理解的，无非就是先锁，然后判断有搞没搞过这一层，如果有就返回，没有就搞一下。要搞这一层，首先吧这一层的有用数据从关键帧里面取出来，即

```
int w = keyframe->width(level);
int h = keyframe->height(level);

float fxInvLevel = keyframe->fxInv(level);
float fyInvLevel = keyframe->fyInv(level);
float cxInvLevel = keyframe->cxInv(level);
float cyInvLevel = keyframe->cyInv(level);

const float* pyrIdepthSource = keyframe->idepth(level);
const float* pyrIdepthVarSource = keyframe->idepthVar(level);
const float* pyrColorSource = keyframe->image(level);
const Eigen::Vector4f* pyrGradSource = keyframe->gradients(level);
```

由于之前import的时候，可能没有把内存释放了(配置参数一样的情况)，所以引入了4个判断，并考虑资源的分配
分配好之后，利用这些资源：

```
for(int x=1; x<w-1; x++)
    for(int y=1; y<h-1; y++)
    {
        int idx = x + y*w;

        if(pyrIdepthVarSource[idx] <= 0 || pyrIdepthSource[idx] == 0) continue;

        *posDataPT = (1.0f / pyrIdepthSource[idx]) * Eigen::Vector3f(fxInvLevel*x+cxInvLevel,fyInvLevel*y+cyInvLevel,1);
        *gradDataPT = pyrGradSource[idx].head<2>();
        *colorAndVarDataPT = Eigen::Vector2f(pyrColorSource[idx], pyrIdepthVarSource[idx]);
        *idxPT = idx;

        posDataPT++;
        gradDataPT++;
        colorAndVarDataPT++;
        idxPT++;
    }
```

首先我们需要从关键帧中记录的位置数据和深度数据恢复点云，
然后记录像素梯度
然后把深度方差和可视化的上色部分记录成一个向量，
之后再循环遍历整个过程，将数据全部记录下来，

最后计算指针跳了多少位，即有多少个点云被记录了下来

SE3Tracker

这个类是Tracking算法的核心类，里面定义了和刚体运动相关的Traqcking所需要得数据和算法

```
int width, height;

// camera matrix
Eigen::Matrix3f K, KInv;
float fx,fy,cx,cy;
float fxi,fyi,cxi,cyi;

DenseDepthTrackerSettings settings;


// debug images
cv::Mat debugImageResiduals;
cv::Mat debugImageWeights;
cv::Mat debugImageSecondFrame;
cv::Mat debugImageOldImageSource;
cv::Mat debugImageOldImageWarped;
```

这几个定义的数据都很简单，其中DenseDepthTrackerSettings定义了一些Tracking相关的设定，比如最大迭代次数，认为收敛的阈值(百分比形式，有些数据认为98%收敛，有些是99%)，还有huber距离所需参数等,它构造函数，初始化了相机，Tracking得配置参数，并给需要矫正的数据分配了内存，同时分配了深度，深度得方差，以及计算权重的内存，最后设置计数器为0，默认为没有diverged

```
SE3Tracker::SE3Tracker(int w, int h, Eigen::Matrix3f K)
{
    width = w;
    height = h;

    this->K = K;
    fx = K(0,0);
    fy = K(1,1);
    cx = K(0,2);
    cy = K(1,2);

    settings = DenseDepthTrackerSettings();
    //settings.maxItsPerLvl[0] = 2;

    KInv = K.inverse();
    fxi = KInv(0,0);
    fyi = KInv(1,1);
    cxi = KInv(0,2);
    cyi = KInv(1,2);


    buf_warped_residual = (float*)Eigen::internal::aligned_malloc(w*h*sizeof(float));
    buf_warped_dx = (float*)Eigen::internal::aligned_malloc(w*h*sizeof(float));
    buf_warped_dy = (float*)Eigen::internal::aligned_malloc(w*h*sizeof(float));
```

```
    buf_warped_x = (float*)Eigen::internal::aligned_malloc(w*h*sizeof(float));
    buf_warped_y = (float*)Eigen::internal::aligned_malloc(w*h*sizeof(float));
    buf_warped_z = (float*)Eigen::internal::aligned_malloc(w*h*sizeof(float));

    buf_d = (float*)Eigen::internal::aligned_malloc(w*h*sizeof(float));
    buf_idepthVar = (float*)Eigen::internal::aligned_malloc(w*h*sizeof(float));
    buf_weight_p = (float*)Eigen::internal::aligned_malloc(w*h*sizeof(float));

    buf_warped_size = 0;

    debugImageWeights = cv::Mat(height,width,CV_8UC3);
    debugImageResiduals = cv::Mat(height,width,CV_8UC3);
    debugImageSecondFrame = cv::Mat(height,width,CV_8UC3);
    debugImageOldImageWarped = cv::Mat(height,width,CV_8UC3);
    debugImageOldImageSource = cv::Mat(height,width,CV_8UC3);



    lastResidual = 0;
    iterationNumber = 0;
    pointUsage = 0;
    lastGoodCount = lastBadCount = 0;

    diverged = false;
}
```

---

float SE3Tracker::calcResidualAndBuffers

这个函数是优化相关的函数，参数共有8个，在后面可以看到，其他函数通过callOptimized这个宏调用了这个函数进行优化操作，例如：

```
callOptimized(calcResidualAndBuffers, (reference->posData[lvl], reference->colorAndVarData[lvl],
SE3TRACKING_MIN_LEVEL == lvl ? reference->pointPosInXYGrid[lvl] : 0, reference->numData[lvl],
frame, referenceToFrame, lvl, (plotTracking && lvl == SE3TRACKING_MIN_LEVEL)));
```

现在让我们开始来阅读这个函数的实现代码
首先，如果要可视化Tracking的迭代过程，那么第一步自然是把debug相关的参数都设置进去，否则直接进行下一步，这个操作是通过调用calcResidualAndBuffers_debugStart()实现的:

```
void SE3Tracker::calcResidualAndBuffers_debugStart()
{
    if(plotTrackingIterationInfo || saveAllTrackingStagesInternal)
    {
        int other = saveAllTrackingStagesInternal ? 255 : 0;
        fillCvMat(&debugImageResiduals,cv::Vec3b(other,other,255));
        fillCvMat(&debugImageWeights,cv::Vec3b(other,other,255));
        fillCvMat(&debugImageOldImageSource,cv::Vec3b(other,other,255));
        fillCvMat(&debugImageOldImageWarped,cv::Vec3b(other,other,255));
    }
}
```

之后是本地化操作

```
int w = frame->width(level);
int h = frame->height(level);
Eigen::Matrix3f KLvl = frame->K(level);
float fx_l = KLvl(0,0);
float fy_l = KLvl(1,1);
float cx_l = KLvl(0,2);
float cy_l = KLvl(1,2);

Eigen::Matrix3f rotMat = referenceToFrame.rotationMatrix();
Eigen::Vector3f transVec = referenceToFrame.translation();

const Eigen::Vector3f* refPoint_max = refPoint + refNum;

const Eigen::Vector4f* frame_gradients = frame->gradients(level);
```

然后是对所有的参考点进行操作：
首先是计算参参考点的空间位置相对于新的坐标系的空间位置的三维坐标，然后投影到图像上

```
Eigen::Vector3f Wxp = rotMat * (*refPoint) + transVec;
float u_new = (Wxp[0]/Wxp[2])*fx_l + cx_l;
float v_new = (Wxp[1]/Wxp[2])*fy_l + cy_l;
```

判断当前点是否投影到图像中，并标记

```
if(!(u_new > 1 && v_new > 1 && u_new < w-2 && v_new < h-2))
{
    if(isGoodOutBuffer != 0)
        isGoodOutBuffer[*idxBuf] = false;
    continue;
}
```

然后插值得到亚像素精度级别的深度(注意深度的第三个维度是图像数据)

```
Eigen::Vector3f resInterp = getInterpolatedElement43(frame_gradients, u_new, v_new, w);
```

插值函数如下

```
inline Eigen::Vector3f getInterpolatedElement43(const Eigen::Vector4f* const mat, const float
x, const float y, const int width)
{
    int ix = (int)x;
    int iy = (int)y;
    float dx = x - ix;
    float dy = y - iy;
    float dxdy = dx*dy;
    const Eigen::Vector4f* bp = mat +ix+iy*width;


    return dxdy * *(const Eigen::Vector3f*)(bp+1+width)
            + (dy-dxdy) * *(const Eigen::Vector3f*)(bp+width)
            + (dx-dxdy) * *(const Eigen::Vector3f*)(bp+1)
            + (1-dx-dy+dxdy) * *(const Eigen::Vector3f*)(bp);
```

```
}
```

之后把图像数据做一次仿射操作，再算得亚像素坐标和当前像素坐标的差值

```
float c1 = affineEstimation_a * (*refColVar)[0] + affineEstimation_b;
float c2 = resInterp[2];
float residual = c1 - c2;
```

通过差值自适应算得weight以及相关值，它认为两个变换之间的像素值有个阈值5.0，小于5.0的时候等于1，如果大于5.0，那么依据比值5.0f / fabsf(residual)减小，也就是说，离得越远，权重越小

```
float weight = fabsf(residual) < 5.0f ? 1 : 5.0f / fabsf(residual);
sxx += c1*c1*weight;
syy += c2*c2*weight;
sx += c1*weight;
sy += c2*weight;
sw += weight;
```

然后判断这个点是好是坏，也是个自适应的阈值，这个阈值为一个最大的差异常数，加上梯度值乘以一个比例系数，即，MAX_DIFF_CONSTANT + MAX_DIFF_GRAD_MULT*(resInterp[0]*resInterp[0] + resInterp1*resInterp1)，这个和残差比较，如果残差的平方小于它，那么认为这个点的估计比较好，然后再把这个判断赋值给isGoodOutBuffer[*idxBuf]

```
bool isGood = residual*residual / (MAX_DIFF_CONSTANT + MAX_DIFF_GRAD_MULT*(resInterp[0]*resInterp[0] + resInterp[1]*resInterp[1])) < 1;

if(isGoodOutBuffer != 0)
    isGoodOutBuffer[*idxBuf] = isGood;
```

之后记录计算得到的这一帧改变的值，中间的乘法实际上是投影到图像坐标，即相机参数乘以差之后的梯度值，从而得到图像的改变值，点的第三个维度是z坐标，倒数正好是逆深度

```
*(buf_warped_x+idx) = Wxp(0);
*(buf_warped_y+idx) = Wxp(1);
*(buf_warped_z+idx) = Wxp(2);

*(buf_warped_dx+idx) = fx_l * resInterp[0];
*(buf_warped_dy+idx) = fy_l * resInterp[1];
*(buf_warped_residual+idx) = residual;

*(buf_d+idx) = 1.0f / (*refPoint)[2];
*(buf_idepthVar+idx) = (*refColVar)[1];
idx++;
```

之后再记录残差的平方和，以及误差值(带符号)

```
if(isGood)
{
    sumResUnweighted += residual*residual;
    sumSignedRes += residual;
```

```
    goodCount++;
}
else
    badCount++;
```

然后记录深度改变的比例，最后记录下来

```
float depthChange = (*refPoint)[2] / Wxp[2];
usageCount += depthChange < 1 ? depthChange : 1;
```

循环结束之后，记录所有的改变值，以及计算迭代之后得到的相似变换系数，最后返回平均残差

```
buf_warped_size = idx;

pointUsage = usageCount / (float)refNum;
lastGoodCount = goodCount;
lastBadCount = badCount;
lastMeanRes = sumSignedRes / goodCount;

affineEstimation_a_lastIt = sqrtf((syy - sy*sy/sw) / (sxx - sx*sx/sw));
affineEstimation_b_lastIt = (sy - affineEstimation_a_lastIt*sx)/sw;

calcResidualAndBuffers_debugFinish(w);

return sumResUnweighted / goodCount;
```

SE3Tracker::trackFrame

```
boost::shared_lock<boost::shared_mutex> lock = frame->getActiveLock();
diverged = false;
trackingWasGood = true;
affineEstimation_a = 1; affineEstimation_b = 0;

if(saveAllTrackingStages)
{
    saveAllTrackingStages = false;
    saveAllTrackingStagesInternal = true;
}

if (plotTrackingIterationInfo)
{
    const float* frameImage = frame->image();
    for (int row = 0; row < height; ++ row)
        for (int col = 0; col < width; ++ col)
            setPixelInCvMat(&debugImageSecondFrame,getGrayCvPixel(frameImage[col+row*width]),
col, row, 1);
}
```

以上部分是参数的一些配置和初始化，其中最后部分的setPixelInCvMat其实是为了可视化使用的，它将传入帧中的图片灰度化后设置到debugImageSecondFrame中，用于查看当前帧的数据数据图像数据

之后是Tracking部分，首先将初始估计记录下来(记录了参考帧到当前帧的刚度变换),然后定义一个6自由度矩阵的误差判

别计算对象ls,定义cell数量以及最终的残差

```
Sophus::SE3f referenceToFrame = frameToReference_initialEstimate.inverse().cast<float>();
LGS6 ls;

int numCalcResidualCalls[PYRAMID_LEVELS];
int numCalcWarpUpdateCalls[PYRAMID_LEVELS];

float last_residual = 0;
```

之后进入循环，从Tracking的最高等级开始，向下计算
首先得到点云，然后使用callOptimized函数调用calcResidualAndBuffers,参数为后面括号里面的所有数据，这是调用的宏定义

```
#define callOptimized(function, arguments) function arguments
```

在这个函数中，把buf_warped相关的参数全部更新，并且更新了上次的相似变换参数等，详见calcResidualAndBuffers函数分析
在此之后，判断这个warp的好坏(如果改变的太多，已经超过了这一层图片的1%)，那么我们认为差别太大，Tracking失败，返回一个空的SE3

```
if(buf_warped_size < MIN_GOODPERALL_PIXEL_ABSMIN * (width>>lvl)*(height>>lvl))
{
    diverged = true;
    trackingWasGood = false;
    return SE3();
}
```

首先判断是否使用简单的仿射变换，如果使用了，那么把通过calcResidualAndBuffers函数更新的affineEstimation_a_lastIt以及affineEstimation_b_lastIt，赋值给仿射变换系数

```
if(useAffineLightningEstimation)
{
    affineEstimation_a = affineEstimation_a_lastIt;
    affineEstimation_b = affineEstimation_b_lastIt;
}
```

然后调用calcWeightsAndResidual得到误差，并记录调用次数

```
float lastErr = callOptimized(calcWeightsAndResidual,(referenceToFrame));

numCalcResidualCalls[lvl]++;
```

calcWeightsAndResidual函数相对简单，我不想过多阐述，请自行研究代码

```
float SE3Tracker::calcWeightsAndResidual(
        const Sophus::SE3f& referenceToFrame)
{
    float tx = referenceToFrame.translation()[0];
```

```
    float ty = referenceToFrame.translation()[1];
    float tz = referenceToFrame.translation()[2];

    float sumRes = 0;

    for(int i=0;i<buf_warped_size;i++)
    {
        float px = *(buf_warped_x+i);    // x'
        float py = *(buf_warped_y+i);    // y'
        float pz = *(buf_warped_z+i);    // z'
        float d = *(buf_d+i);    // d
        float rp = *(buf_warped_residual+i); // r_p
        float gx = *(buf_warped_dx+i);  // \delta_x I
        float gy = *(buf_warped_dy+i);  // \delta_y I
        float s = settings.var_weight * *(buf_idepthVar+i); // \sigma_d^2


        // calc dw/dd (first 2 components):
        float g0 = (tx * pz - tz * px) / (pz*pz*d);
        float g1 = (ty * pz - tz * py) / (pz*pz*d);


        // calc w_p
        float drpdd = gx * g0 + gy * g1;    // ommitting the minus
        float w_p = 1.0f / ((cameraPixelNoise2) + s * drpdd * drpdd);

        float weighted_rp = fabs(rp*sqrtf(w_p));

        float wh = fabs(weighted_rp < (settings.huber_d/2) ? 1 : (settings.huber_d/2) /
weighted_rp);

        sumRes += wh * w_p * rp*rp;


        *(buf_weight_p+i) = wh * w_p;
    }

    return sumRes / buf_warped_size;
}
```

然后是调用LM算法更新参数，首先我们来看看SE3Tracker::calculateWarpUpdate函数
首先现将ls参数初始化成默认值

```
ls.initialize(width*height);
```

然后本地化参数

```
float px = *(buf_warped_x+i);
float py = *(buf_warped_y+i);
float pz = *(buf_warped_z+i);
float r =  *(buf_warped_residual+i);
float gx = *(buf_warped_dx+i);
float gy = *(buf_warped_dy+i);
```

然后计算误差向量

```
float z = 1.0f / pz;
float z_sqr = 1.0f / (pz*pz);
Vector6 v;
v[0] = z*gx + 0;
v[1] = 0 +         z*gy;
v[2] = (-px * z_sqr) * gx +
       (-py * z_sqr) * gy;
v[3] = (-px * py * z_sqr) * gx +
       (-(1.0 + py * py * z_sqr)) * gy;
v[4] = (1.0 + px * px * z_sqr) * gx +
       (px * py * z_sqr) * gy;
v[5] = (-py * z) * gx +
       (px * z) * gy;
```

最后更新到ls参数中，得到最小二乘法方程

```
ls.update(v, r, *(buf_weight_p+i));
```

回到Tracking，让我们继续看LM算法的迭代,首先更新ls中的参数，然后记录更新次数，并记录是第几次迭代

```
callOptimized(calculateWarpUpdate,(ls));

numCalcWarpUpdateCalls[lvl]++;

iterationNumber = iteration;
```

然后最小二乘法方程

```
// solve LS system with current lambda
Vector6 b = -ls.b;
Matrix6x6 A = ls.A;
for(int i=0;i<6;i++) A(i,i) *= 1+LM_lambda;
Vector6 inc = A.ldlt().solve(b);
```

得到最优解后，计算新的变换矩阵

```
Sophus::SE3f new_referenceToFrame = Sophus::SE3f::exp((inc)) * referenceToFrame;
```

$$\delta\boldsymbol{\xi}^{(n)} = -(\mathbf{J}^T\mathbf{J})^{-1}\mathbf{J}^T\mathbf{r}(\boldsymbol{\xi}^{(n)}) \quad \text{with} \quad \mathbf{J} = \left.\frac{\partial \mathbf{r}(\boldsymbol{\epsilon} \circ \boldsymbol{\xi}^{(n)})}{\partial \boldsymbol{\epsilon}}\right|_{\boldsymbol{\epsilon}=0}$$

$$\boldsymbol{\xi}^{(n+1)} = \delta\boldsymbol{\xi}^{(n)} \circ \boldsymbol{\xi}^{(n)}.$$

实际上上面两步就是在计算论文中这个过程，只不过写成了线性方程求解的形式
之后再次调用calcResidualAndBuffers计算warpedbuffer以及残差值，然后再次进行前面同样的操作

```
callOptimized(calcResidualAndBuffers, ...); //看源码的输入，太长了，我省略不写了
if(buf_warped_size < MIN_GOODPERALL_PIXEL_ABSMIN* (width>>lvl)*(height>>lvl))
{
    diverged = true;
    trackingWasGood = false;
    return SE3();
}

float error = callOptimized(calcWeightsAndResidual,(new_referenceToFrame));
numCalcResidualCalls[lvl]++;
```

之后判断误差是否变小，如果变小，那么接受刚刚的修正，否则判断是否已经小于最小步长，如果已经小于，直接跳出所有循环，否则改变LM_lambda的值，重新进行while循环

```
if(error < lastErr)
    ...
else
{
    if(enablePrintDebugInfo && printTrackingIterationInfo)
    {
        printf("(%d-%d): REJECTED increment of %f with lambda %.1f, (residual: %f -> %f)\n",
                    lvl,iteration, sqrt(inc.dot(inc)), LM_lambda, lastErr, error);
    }

    if(!(inc.dot(inc) > settings.stepSizeMin[lvl]))
    {
        if(enablePrintDebugInfo && printTrackingIterationInfo)
        {
            printf("(%d-%d): FINISHED pyramid level (stepsize too small).\n",
                    lvl,iteration);
        }
        iteration = settings.maxItsPerLvl[lvl];
        break;
    }

    if(LM_lambda == 0)
        LM_lambda = 0.2;
    else
        LM_lambda *= std::pow(settings.lambdaFailFac, incTry);
}
```

接受之后的操作：
首先是仿射系数修正，然后是判断是否收敛，如果收敛了就跳出跌打循环（把迭代次数置于最大值），之后更新误差，然后更新LM_lambda值，结束while的循环，进行下一次迭代

```
referenceToFrame = new_referenceToFrame;
if(useAffineLightningEstimation)
{
    affineEstimation_a = affineEstimation_a_lastIt;
    affineEstimation_b = affineEstimation_b_lastIt;
}

// converged?
if(error / lastErr > settings.convergenceEps[lvl])
```

```
{
    if(enablePrintDebugInfo && printTrackingIterationInfo)
    {
        printf("(%d-%d): FINISHED pyramid level (last residual reduction too small).\n",
                lvl,iteration);
    }
    iteration = settings.maxItsPerLvl[lvl];
}

last_residual = lastErr = error;

if(LM_lambda <= 0.2)
    LM_lambda = 0;
else
    LM_lambda *= settings.lambdaSuccessFac;

break;
```

迭代计算之后是保存状态

```
saveAllTrackingStagesInternal = false;

lastResidual = last_residual;

trackingWasGood = !diverged&& lastGoodCount / (frame->width(SE3TRACKING_MIN_LEVEL) *
                frame->height(SE3TRACKING_MIN_LEVEL)) > MIN_GOODPERALL_PIXEL &&
                lastGoodCount / (lastGoodCount + lastBadCount) >  MIN_GOODPERGOODBAD_PIXEL;

if(trackingWasGood)         reference->keyframe->numFramesTrackedOnThis++;
```

然后更新传入帧的属性，记录误差，到参考帧的变换，以及参考帧的信息，最后返回当前帧到参考帧的变换

```
frame->initialTrackedResidual = lastResidual / pointUsage;
frame->pose->thisToParent_raw = sim3FromSE3(toSophus(referenceToFrame.inverse()),1);
frame->pose->trackingParent = reference->keyframe->pose;
return toSophus(referenceToFrame.inverse());
```

我想在理解SE3Tracker::trackFrame函数后SE3Tracker::trackFrameOnPermaref已经很简单了他们的区别在于一个用于精确计算，一个用于快速计算，前者是在每次计算的时候，重新计算了点云，使得整个计算更加精确，但是需要付出计算点云的开销，后面那个直接使用每一帧原来存储的点云信息，效率明显更高，但是会付出精度的代价，具体程序请自行阅读

SE3Tracker::checkPermaRefOverlap

首先本地化参数

```
Sophus::SE3f referenceToFrame = referenceToFrameOrg.cast<float>();
boost::unique_lock<boost::mutex> lock2 = boost::unique_lock<boost::mutex>(reference->permaRef_mutex);

int w2 = reference->width(QUICK_KF_CHECK_LVL)-1;
int h2 = reference->height(QUICK_KF_CHECK_LVL)-1;
```

```
Eigen::Matrix3f KLvl = reference->K(QUICK_KF_CHECK_LVL);
float fx_l = KLvl(0,0);
float fy_l = KLvl(1,1);
float cx_l = KLvl(0,2);
float cy_l = KLvl(1,2);

Eigen::Matrix3f rotMat = referenceToFrame.rotationMatrix();
Eigen::Vector3f transVec = referenceToFrame.translation();

const Eigen::Vector3f* refPoint_max = reference->permaRef_posData + reference->permaRefNumPts;
const Eigen::Vector3f* refPoint = reference->permaRef_posData;
```

之后进入循环，先计算图像上的坐标

```
Eigen::Vector3f Wxp = rotMat * (*refPoint) + transVec;
float u_new = (Wxp[0]/Wxp[2])*fx_l + cx_l;
float v_new = (Wxp[1]/Wxp[2])*fy_l + cy_l;
```

判断是否在图像内部，如果在，记录到usageCount中

```
if((u_new > 0 && v_new > 0 && u_new < w2 && v_new < h2))
{
    float depthChange = (*refPoint)[2] / Wxp[2];
    usageCount += depthChange < 1 ? depthChange : 1;
}
```

最后得到pointUsage,并返回

```
pointUsage = usageCount / (float)reference->permaRefNumPts;
return pointUsage;
```

这个函数介绍完毕之后，我想SE3Tracking类就介绍的差不多了，它主要是用来计算当前帧和参考帧之间的刚体变换，以及检测Tracking是否足够好的一个很大的类，其中Tracking提供了两个方法，一个精确但速度慢一些，一个快，但精度低一些，他们内部都维护了Weighted Gauss-Newton Optimization优化方法，设置了大量检测的阈值，并且涉及到大量的坐标变换(图像之间的投影，世界坐标向图像即投影，两帧之间的坐标转换关系等)，这无疑是一个特别核心的类，理解了这个类之后，其实Sim3Tracking和这个类是几乎完全一样的，读者可以试图自己分析它的实现，来进一步掌握Tracking类的实现方法

Relocalizer

在介绍Tracking的核心类之后，我们最后介绍一下，假如Tracking失败了，那么它应该做的操作，即Relocalizer(重定位)在论文里面，并没有提及Relocalizer的事，我希望能够通过代码，介绍一下这部分是如何处理的

Relocalizer::start

首先是这个函数，进入之后先清空之前记录的假如丢失后，给重新找回机制分配的那些帧(KFForReloc，本文把他叫做找回参考帧)

```
// make KFForReloc List
KFForReloc.clear();
```

然后把传入的那些keyFrame压入KFForReloc中，并打乱顺序

```
for(unsigned int k=0;k < allKeyframesList.size(); k++)
{
    // insert
    KFForReloc.push_back(allKeyframesList[k]);

    // swap with a random element
    int ridx = rand()%(KFForReloc.size());
    Frame* tmp = KFForReloc.back();
    KFForReloc.back() = KFForReloc[ridx];
    KFForReloc[ridx] = tmp;
}
```

初始化nextRelocIDx，以及最大的找回参考帧ID，然后打开找回线程

```
nextRelocIDX=0;
maxRelocIDX=KFForReloc.size();

hasResult = false;
continueRunning = true;
isRunning = true;

// start threads
for(int i=0;i<RELOCALIZE_THREADS;i++)
{
    relocThreads[i] = boost::thread(&Relocalizer::threadLoop, this, i);
    running[i] = true;
}
```

void threadLoop(int idx)

这个函数是找回线程中运行的函数，也是这个类的核心函数
首先初始化了一个SE3Tracker的对象，调用构造函数时传入了帧的宽度，高度以及相机矩阵
之后自然是线程锁,然后判断是否是可以运行(这个成员用于stop),如果可以，那么进入循环

```
if(!multiThreading && idx != 0) return;
SE3Tracker* tracker = new SE3Tracker(w,h,K);
boost::unique_lock<boost::mutex> lock(exMutex);
while(continueRunning)
```

然后做在不越界的条件下抽取一帧，nextRelocIDX自加

```
Frame* todo = KFForReloc[nextRelocIDX%KFForReloc.size()];
nextRelocIDX++;
```

由于对关键帧都有插入操作（插入到之前的帧中），所以它自然有相邻的帧，我们把它周围的帧叫做它的邻近帧，为了重定位足够好而且足够快，我们在选择帧之后，判断它的邻近帧是不是足够多(因为实际上可能它只和一两个帧连起来，这

种帧用来重定位并不那么好)

```
if(todo->neighbors.size() <= 2) continue;
```

满足要求之后，把当前的参考帧作为需要找回帧，调用trackFrameOnPermaref函数，快速地找到这一帧到找回帧的坐标转换关系，然后计算这个tracking的分数(看他好不好，通过平均分乘以匹配上的好的匹配个数，所有的匹配个数)

```
SE3 todoToFrame = tracker->trackFrameOnPermaref(todo, myRelocFrame.get(), SE3());

// try neighbours
float todoGoodVal = tracker->pointUsage * tracker->lastGoodCount / (tracker->lastGoodCount+tracker->lastBadCount);
```

如果这个匹配比较好，那么继续后续工作
之后查看附近帧的匹配情况，首先还是初始化

```
int numGoodNeighbours = 0;
int numBadNeighbours = 0;

float bestNeightbourGoodVal = todoGoodVal;
float bestNeighbourUsage = tracker->pointUsage;
Frame* bestKF = todo;
SE3 bestKFToFrame = todoToFrame;
```

然后遍历整个附近帧

```
for(Frame* nkf : todo->neighbors)
```

得到当前帧到附近帧的变换，然后再快速Tracking，之后判断Tracking是否足够好，并且判断是否比最好的那个匹配更好，如果匹配更好，那么更新最好的匹配

```
SE3 nkfToFrame_init = se3FromSim3((nkf->getScaledCamToWorld().inverse() * todo->getScaledCamToWorld() * sim3FromSE3(todoToFrame.inverse(), 1))).inverse();
SE3 nkfToFrame = tracker->trackFrameOnPermaref(nkf, myRelocFrame.get(), nkfToFrame_init);

float goodVal = tracker->pointUsage * tracker->lastGoodCount / (tracker->lastGoodCount+tracker->lastBadCount);
if(goodVal > relocalizationTH*0.8 && (nkfToFrame * nkfToFrame_init.inverse()).log().norm() < 0.1)
    numGoodNeighbours++;
else
    numBadNeighbours++;

if(goodVal > bestNeightbourGoodVal)
{
    bestNeightbourGoodVal = goodVal;
    bestKF = nkf;
    bestKFToFrame = nkfToFrame;
    bestNeighbourUsage = tracker->pointUsage;
```

循环完毕之后，判断是否邻近帧的匹配比较好，如果比较好，那么表明已经找到，停止查找

```
if(numGoodNeighbours > numBadNeighbours || numGoodNeighbours >= 5)
{
    // set everything to stop!
    continueRunning = false;
    lock.lock();
    resultRelocFrame = myRelocFrame;
    resultFrameID = myRelocFrame->id();
    resultKF = bestKF;
    resultFrameToKeyframe = bestKFToFrame.inverse();
    resultReadySignal.notify_all();
    hasResult = true;
    lock.unlock();
}
```

最后把数据锁上，避免再次访问到这一个召回参考帧

```
lock.lock();
```

其他函数都比较简单，请读者自行查看，至此，Tracking模块的源码解析全部结束，想要真正了解代码，我觉得还需要多阅读论文，把代码和论文相互印证，理解其含义，并且程序中有很多小技巧，希望读者仔细体会

DepthEstimation

DepthMapPixelHypothesis

我想一个科学的研究方式应该是先阅读这个类，从名字上来看，这个类是对于深度图上每个像素的一个深度估计，它只有公有成员，成员变量分别表达了

- 该像素是否有效
- 该像素是否列入黑名单
- 要逃过的最小帧数
- 有效的观测次数
- 深度均值
- 深度方差
- 平滑后的深度均值
- 平滑后的深度方差

接下来是3个构造函数，最后为了可视化，设置了一个返回rgb三个值的向量

DepthMap

这是一个很核心的类，首先我们来看构造函数DepthMap::DepthMap(int w, int h, const Eigen::Matrix3f& K)

构造函数需要传入图像的宽度以及高度，还有相机的内参，首先显然是分配内存，并且本地化相机参数

```
width = w;
height = h;

activeKeyFrame = 0;
activeKeyFrameIsReactivated = false;
otherDepthMap = new DepthMapPixelHypothesis[width*height];
currentDepthMap = new DepthMapPixelHypothesis[width*height];

validityIntegralBuffer = (int*)Eigen::internal::aligned_malloc(width*height*sizeof(int));

debugImageHypothesisHandling = cv::Mat(h,w, CV_8UC3);
debugImageHypothesisPropagation = cv::Mat(h,w, CV_8UC3);
debugImageStereoLines = cv::Mat(h,w, CV_8UC3);
debugImageDepth = cv::Mat(h,w, CV_8UC3);

this->K = K;
fx = K(0,0);
fy = K(1,1);
cx = K(0,2);
cy = K(1,2);

KInv = K.inverse();
fxi = KInv(0,0);
fyi = KInv(1,1);
cxi = KInv(0,2);
cyi = KInv(1,2);
```

然后调用reset()函数，将该像素的深度估计初始化为无效

```
void DepthMap::reset()
{
    for(DepthMapPixelHypothesis* pt = otherDepthMap+width*height-1; pt >= otherDepthMap; pt--)
        pt->isValid = false;
    for(DepthMapPixelHypothesis* pt = currentDepthMap+width*height-1; pt >= currentDepthMap; pt--)
        pt->isValid = false;
}
```

最后初始化一些其他参数，比如与计时相关的，计数相关的参数等

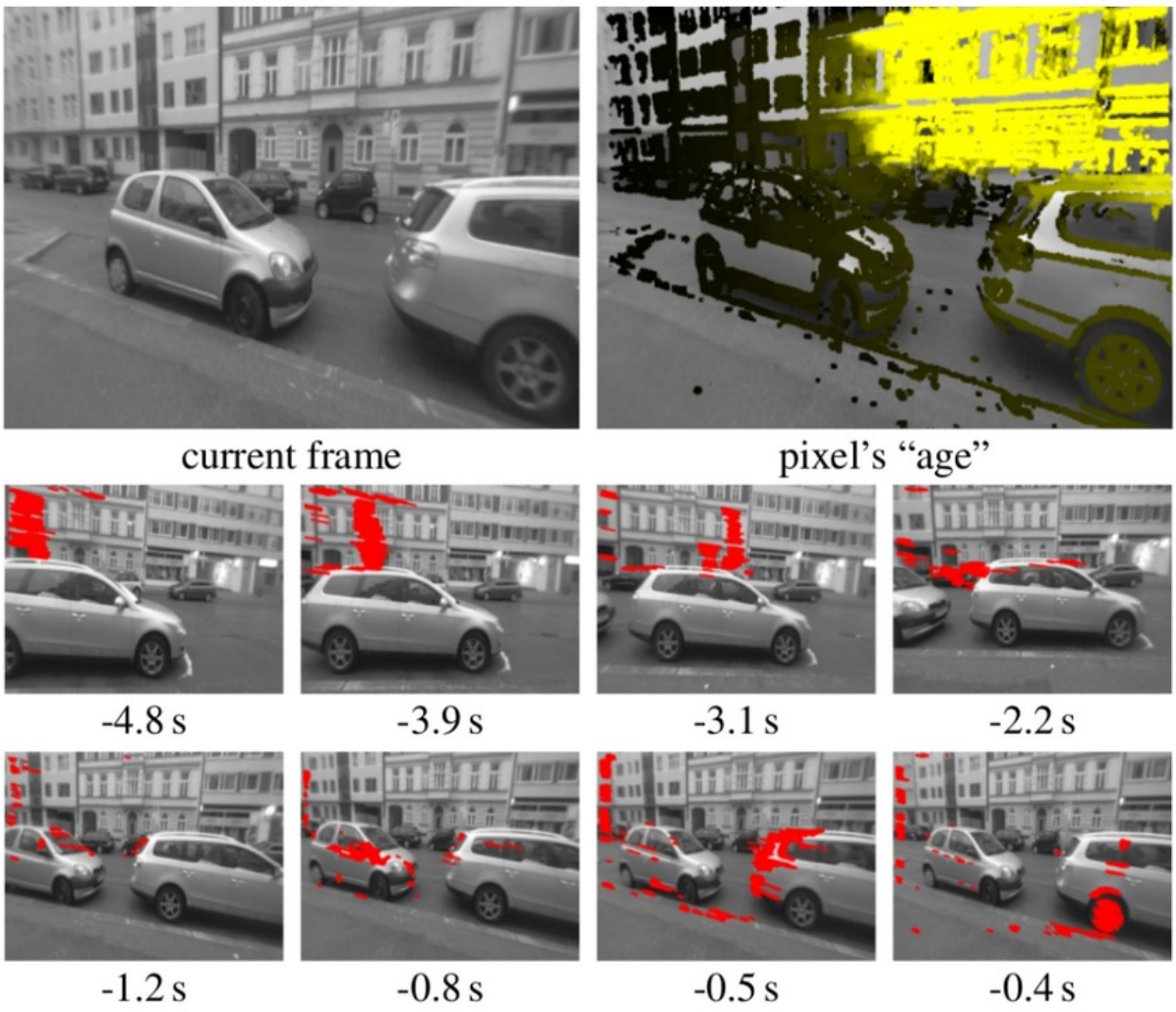
```
msUpdate =  msCreate =  msFinalize = 0;
msObserve =  msRegularize =  msPropagate =  msFillHoles =  msSetDepth = 0;
gettimeofday(&lastHzUpdate, NULL);
nUpdate = nCreate = nFinalize = 0;
nObserve = nRegularize = nPropagate = nFillHoles = nSetDepth = 0;
nAvgUpdate = nAvgCreate = nAvgFinalize = 0;
nAvgObserve = nAvgRegularize = nAvgPropagate = nAvgFillHoles = nAvgSetDepth = 0;
```

---

void DepthMap::updateKeyframe

这个函数需要传入参考帧的指针队列，根据参考帧更新当前关键帧，是尤其重要的一个函数

首先记录最"年轻"的参考帧与最"老"的参考帧，对应算法的这个部分




```
    oldest_referenceFrame = referenceFrames.front().get();
    newest_referenceFrame = referenceFrames.back().get();
    referenceFrameByID.clear();
    referenceFrameByID_offset = oldest_referenceFrame->id();
```

然后遍历所有帧，判断参考帧是不是当前关键帧，如果不是，就转换到当前关键帧，并且调用帧里面的prepareForStereoWith函数，算出需要用的投影矩阵，之后把帧这些准备好的帧压入参考帧队列,并初始化计数器

```
    for(std::shared_ptr<Frame> frame : referenceFrames)
    {
        assert(frame->hasTrackingParent());

        if(frame->getTrackingParent() != activeKeyFrame)
        {
            printf("WARNING: updating frame %d with %d, which was tracked on a different frame (%d).\nWhile this should work, it is not recommended.",
                    activeKeyFrame->id(), frame->id(),
                    frame->getTrackingParent()->id());
        }

```

```
        Sim3 refToKf;
        if(frame->pose->trackingParent->frameID == activeKeyFrame->id())
            refToKf = frame->pose->thisToParent_raw;
        else
            refToKf = activeKeyFrame->getScaledCamToWorld().inverse() *  frame->getScaledCamToWorld();

        frame->prepareForStereoWith(activeKeyFrame, refToKf, K, 0);

        while((int)referenceFrameByID.size() + referenceFrameByID_offset <= frame->id())
            referenceFrameByID.push_back(frame.get());
    }

    resetCounters();
```

之后是调用观测函数，这个函数实际上要向下调用threadReducer对象的reduce，然后通过boost的bind调用到了observeDepthRow函数，这无疑是一段很飘逸的代码，让我们来好好看下是如何实现的i
首先是记录时间并调用observeDepth()，完成之后记录消耗时间，并记录下本次观测

```
    gettimeofday(&tv_start, NULL);
    observeDepth();
    gettimeofday(&tv_end, NULL);
    msObserve = 0.9*msObserve + 0.1*((tv_end.tv_sec-tv_start.tv_sec)*1000.0f + (tv_end.tv_usec-tv_start.tv_usec)/1000.0f);
    nObserve++;
```

observeDepth()函数实际上就一行有效代码，即

```
threadReducer.reduce(boost::bind(&DepthMap::observeDepthRow, this, _1, _2, _3), 3, height-3, 10);
```

这个函数是通过IndexThreadReduce的对象threadReducer，调用方法reduce，传入了一个函数对象(或者叫做仿函数)this->observeDepthRow(int, int, RunningStats*), 以及三个参数3, height-3,10

IndexThreadReduce::reduce

这个函数要求的传入参数是一个函数对象，三个int类型的整数，这个函数对象需要三个参数，分别是int,int,RunningStats*，返回值为void，这四个参数正好对应了this->observeDepthRow(int, int, RunningStats*)，3, height-3,10,
我想你读到这里，应该能够深刻体会到bind的强大之处，函数指针void()*(int,int,RunningStats)*和函数指针void DepthMap::observeDepthRow(int yMin, int yMax, RunningStats* stats)类型是不同的，实际如果只是简单使用一下函数指针传递，编译是无法通过的，但是bind内部维护了这个转化，让你能够轻松地使用类里面的函数
由于slam肯定使用多线程，传入的stepSize==10，所以我们可以直接跳过前面几行，进入到互斥锁，之后的操作是本地化参数

```
    this->callPerIndex = callPerIndex;
    nextIndex = first;
    maxIndex = end;
    this->stepSize = stepSize;
```

之后开始工作线程

```
    // go worker threads!
    for(int i=0;i<MAPPING_THREADS;i++)
        isDone[i] = false;

    // let them start!
    todo_signal.notify_all();
```

然后运行线程，等待结束

```
    //printf("reduce waiting for threads to finish\n");
    // wait for all worker threads to signal they are done.
    while(true)
    {
        // wait for at least one to finish
        done_signal.wait(lock);
        //printf("thread finished!\n");

        // check if actually all are finished.
        bool allDone = true;
        for(int i=0;i<MAPPING_THREADS;i++)
            allDone = allDone && isDone[i];

        // all are finished! exit.
        if(allDone)
            break;
    }
```

最后再还原线程相关参数

```
    nextIndex = 0;
    maxIndex = 0;
    this->callPerIndex = boost::bind(&IndexThreadReduce::callPerIndexDefault, this, _1, _2, _
3);
```

至于这个调用是咋调用的呢，看到这里，你应该说我晕啊，这个该死的程序咋嵌套这么麻烦。。。实际上是在下面这个地方，把他做成并行计算了

```
void workerLoop(int idx)
    {
        boost::unique_lock<boost::mutex> lock(exMutex);

        while(running)
        {
            // try to get something to do.
            int todo = 0;
            bool gotSomething = false;
            if(nextIndex < maxIndex)
            {
                // got something!
                todo = nextIndex;
                nextIndex+=stepSize;
                gotSomething = true;
            }

```

```
            // if got something: do it (unlock in the meantime)
            if(gotSomething)
            {
                lock.unlock();

                assert(callPerIndex != 0);

                RunningStats s;
                callPerIndex(todo, std::min(todo+stepSize, maxIndex), &s);

                lock.lock();
                runningStats.add(&s);
            }

            // otherwise wait on signal, releasing lock in the meantime.
            else
            {
                isDone[idx] = true;
                //printf("worker %d waiting..\n", idx);
                done_signal.notify_all();
                todo_signal.wait(lock);
            }
        }
    }
```

DepthMap::observeDepthRow

现在在让我们来看线程中实际运行的函数void DepthMap::observeDepthRow(int yMin, int yMax, RunningStats* stats)，首先得到关键帧最大的梯度,然后开始循环

```
const float* keyFrameMaxGradBuf = activeKeyFrame->maxGradients(0);
```

首先还是本地化参数，之后测试是否可用，不可用(梯度太小，无效点，列入黑名单)就继续下一次

```
    int idx = x+y*width;
    DepthMapPixelHypothesis* target = currentDepthMap+idx;
    bool hasHypothesis = target->isValid;
    // ======== 1. check absolute grad =========
    if(hasHypothesis && keyFrameMaxGradBuf[idx] < MIN_ABS_GRAD_DECREASE)
    {
        target->isValid = false;
        continue;
    }

    if(keyFrameMaxGradBuf[idx] < MIN_ABS_GRAD_CREATE || target->blacklisted < MIN_BLACKLIST)
        continue;
```

然后计算深度(如果没有深度就创建，有深度就更新）

```
if(!hasHypothesis)
    success = observeDepthCreate(x, y, idx, stats);
else
    success = observeDepthUpdate(x, y, idx, keyFrameMaxGradBuf, stats);
```

bool DepthMap::observeDepthCreate

老样子，首先是本地化参数，然后检测是不是可用，我就不细说了，之后是对极线的计算
调用makeAndCheckEPL函数
首先计算epl

```
    float epx = - fx * ref->thisToOther_t[0] + ref->thisToOther_t[2]*(x - cx);
    float epy = - fy * ref->thisToOther_t[1] + ref->thisToOther_t[2]*(y - cy);
```

这里相当于是把相对于点(每个点都算了一次，可以注意到，前面分了很多线程，凑起来正好每个点都计算了一次)的平移投影做了拉伸，投影到了图像上，然后加上了点的坐标，但是都必须投影到同一个平面上，所以乘以了深度值，得到了表达那个点的"向量线段"，由于假设了旋转影响小，所以只有平移的投影，应该是对应了论文这个部分

sponding to infinite depth. We now assume that only the absolute position of this line segment, i.e., l_0 is subject to isotropic Gaussian noise ϵ_l. As in practice we keep the searched epipolar line segments short, the influence of rotational error is small, making this a good approximation.

然后是想判断这个极线的角度和梯度的角度是不是足够小（当然在这之前还检测了一下长度达不到达标），如果足够小，那么就返回极限的归一化向量

```
    // ======== check epl length ==========
    float eplLengthSquared = epx*epx+epy*epy;
    if(eplLengthSquared < MIN_EPL_LENGTH_SQUARED)
    {
        if(enablePrintDebugInfo) stats->num_observe_skipped_small_epl++;
        return false;
    }


    // ===== check epl-grad magnitude ======
    float gx = activeKeyFrameImageData[idx+1] - activeKeyFrameImageData[idx-1];
    float gy = activeKeyFrameImageData[idx+width] - activeKeyFrameImageData[idx-width];
    float eplGradSquared = gx * epx + gy * epy;
    eplGradSquared = eplGradSquared*eplGradSquared / eplLengthSquared;  // square and norm with epl-length

    if(eplGradSquared < MIN_EPL_GRAD_SQUARED)
    {
        if(enablePrintDebugInfo) stats->num_observe_skipped_small_epl_grad++;
        return false;
    }


    // ===== check epl-grad angle ======
    if(eplGradSquared / (gx*gx+gy*gy) < MIN_EPL_ANGLE_SQUARED)
    {
        if(enablePrintDebugInfo) stats->num_observe_skipped_small_epl_angle++;
        return false;
    }


    // ===== DONE - return "normalized" epl =====
```

```
32.     float fac = GRADIENT_SAMPLE_DIST / sqrt(eplLengthSquared);
33.     *pepx = epx * fac;
34.     *pepy = epy * fac;
```

之后的工作是调用了一个极度复杂的函数DepthMap::doLineStereo，为了看懂这个函数，把我折磨地死去活来，现在让我们来看下这个函数是如何工作的

float DepthMap::doLineStereo

```
1.     float error = doLineStereo(
2.             new_u,new_v,epx,epy,
3.             0.0f, 1.0f, 1.0f/MIN_DEPTH,
4.             refFrame, refFrame->image(0),
5.             result_idepth, result_var, result_eplLength, stats);
```

首先这是调用，从调用上来看，传入了当前点的坐标，极线的单位向量，最小深度(0.0)，当前深度(1.0)，最大深度(1/MIN_DEPTH,当前参考帧，当前参考帧的最底层图像，然后传入3个数据记录结果，最后是运行状态
进入函数之后，首先计算的是当前点在参考帧上的投影极线信息，先从图像投影到空间，然后从这一帧的相机坐标向参考帧转换，最后加上平移

```
1.     Eigen::Vector3f KinvP = Eigen::Vector3f(fxi*u+cxi,fyi*v+cyi,1.0f);
2.     Eigen::Vector3f pInf = referenceFrame->K_otherToThis_R * KinvP;
3.     Eigen::Vector3f pReal = pInf / prior_idepth + referenceFrame->K_otherToThis_t;
```

之后确定极线起点和终点，我估计这里计算这个的目的是想要判断最大搜索范围是不是在图像梯度可算范围内

```
1.     float rescaleFactor = pReal[2] * prior_idepth;
2.
3.     float firstX = u - 2*epxn*rescaleFactor;
4.     float firstY = v - 2*epyn*rescaleFactor;
5.     float lastX = u + 2*epxn*rescaleFactor;
6.     float lastY = v + 2*epyn*rescaleFactor;
```

后面很明显有判断范围，以及判断了缩放范围，再下来调用了getInterpolatedElement函数，计算了下图像在该点的值，实际上就是插值，函数体如下

```
1.  inline float getInterpolatedElement(const float* const mat, const float x, const float y, con
    st int width)
2.  {
3.      //stats.num_pixelInterpolations++;
4.
5.      int ix = (int)x;
6.      int iy = (int)y;
7.      float dx = x - ix;
8.      float dy = y - iy;
9.      float dxdy = dx*dy;
10.     const float* bp = mat +ix+iy*width;
11.
12.
13.     float res =   dxdy * bp[1+width]
14.                 + (dy-dxdy) * bp[width]
15.                 + (dx-dxdy) * bp[1]
```

```
                + (1-dx-dy+dxdy) * bp[0];

    return res;
}
```

然后计算最"近"的点

```
    Eigen::Vector3f pClose = pInf + referenceFrame->K_otherToThis_t*max_idepth;
    // if the assumed close-point lies behind the
    // image, have to change that.
    if(pClose[2] < 0.001f)
    {
        max_idepth = (0.001f-pInf[2]) / referenceFrame->K_otherToThis_t[2];
        pClose = pInf + referenceFrame->K_otherToThis_t*max_idepth;
    }
    pClose = pClose / pClose[2]; // pos in new image of point (xy), assuming max_idepth
```

之后计算最"远"的点

```
    Eigen::Vector3f pFar = pInf + referenceFrame->K_otherToThis_t*min_idepth;
    // if the assumed far-point lies behind the image or closter than the near-point,
    // we moved past the Point it and should stop.
    if(pFar[2] < 0.001f || max_idepth < min_idepth)
    {
        if(enablePrintDebugInfo) stats->num_stereo_inf_oob++;
        return -1;
    }
    pFar = pFar / pFar[2]; // pos in new image of point (xy), assuming min_idepth
```

得到极线线段，并归一化

```
    // calculate increments in which we will step through the epipolar line.
    // they are sampleDist (or half sample dist) long
    float incx = pClose[0] - pFar[0];
    float incy = pClose[1] - pFar[1];
    float eplLength = sqrt(incx*incx+incy*incy);
    if(!eplLength > 0 || std::isinf(eplLength)) return -4;

    incx *= GRADIENT_SAMPLE_DIST/eplLength;
    incy *= GRADIENT_SAMPLE_DIST/eplLength;
```

然后拓展长度，以及搜索宽度，确保这两个点都在图像内部

```
    // extend one sample_dist to left & right.
    pFar[0] -= incx;
    pFar[1] -= incy;
    pClose[0] += incx;
    pClose[1] += incy;


    // make epl long enough (pad a little bit).
    if(eplLength < MIN_EPL_LENGTH_CROP)
    {
```

```
        float pad = (MIN_EPL_LENGTH_CROP - (eplLength)) / 2.0f;
        pFar[0] -= incx*pad;
        pFar[1] -= incy*pad;

        pClose[0] += incx*pad;
        pClose[1] += incy*pad;
    }

    // if inf point is outside of image: skip pixel.
    if(
            pFar[0] <= SAMPLE_POINT_TO_BORDER ||
            pFar[0] >= width-SAMPLE_POINT_TO_BORDER ||
            pFar[1] <= SAMPLE_POINT_TO_BORDER ||
            pFar[1] >= height-SAMPLE_POINT_TO_BORDER)
    {
        if(enablePrintDebugInfo) stats->num_stereo_inf_oob++;
        return -1;
    }



    // if near point is outside: move inside, and test length again.
    if(
            pClose[0] <= SAMPLE_POINT_TO_BORDER ||
            pClose[0] >= width-SAMPLE_POINT_TO_BORDER ||
            pClose[1] <= SAMPLE_POINT_TO_BORDER ||
            pClose[1] >= height-SAMPLE_POINT_TO_BORDER)
    {
        if(pClose[0] <= SAMPLE_POINT_TO_BORDER)
        {
            float toAdd = (SAMPLE_POINT_TO_BORDER - pClose[0]) / incx;
            pClose[0] += toAdd * incx;
            pClose[1] += toAdd * incy;
        }
        else if(pClose[0] >= width-SAMPLE_POINT_TO_BORDER)
        {
            float toAdd = (width-SAMPLE_POINT_TO_BORDER - pClose[0]) / incx;
            pClose[0] += toAdd * incx;
            pClose[1] += toAdd * incy;
        }

        if(pClose[1] <= SAMPLE_POINT_TO_BORDER)
        {
            float toAdd = (SAMPLE_POINT_TO_BORDER - pClose[1]) / incy;
            pClose[0] += toAdd * incx;
            pClose[1] += toAdd * incy;
        }
        else if(pClose[1] >= height-SAMPLE_POINT_TO_BORDER)
        {
            float toAdd = (height-SAMPLE_POINT_TO_BORDER - pClose[1]) / incy;
            pClose[0] += toAdd * incx;
            pClose[1] += toAdd * incy;
        }

        // get new epl length
        float fincx = pClose[0] - pFar[0];
        float fincy = pClose[1] - pFar[1];
        float newEplLength = sqrt(fincx*fincx+fincy*fincy);
```

```
69.
70.         // test again
71.         if(
72.                 pClose[0] <= SAMPLE_POINT_TO_BORDER ||
73.                 pClose[0] >= width-SAMPLE_POINT_TO_BORDER ||
74.                 pClose[1] <= SAMPLE_POINT_TO_BORDER ||
75.                 pClose[1] >= height-SAMPLE_POINT_TO_BORDER ||
76.                 newEplLength < 8.0f
77.                 )
78.         {
79.             if(enablePrintDebugInfo) stats->num_stereo_near_oob++;
80.             return -1;
81.         }
82.
83.
84.     }
```

得到远点的图像信息

```
1.  float cpx = pFar[0];
2.  float cpy =  pFar[1];
3.
4.  float val_cp_m2 = getInterpolatedElement(referenceFrameImage,cpx-2.0f*incx, cpy-2.0f*incy, wi
    dth);
5.  float val_cp_m1 = getInterpolatedElement(referenceFrameImage,cpx-incx, cpy-incy, width);
6.  float val_cp = getInterpolatedElement(referenceFrameImage,cpx, cpy, width);
7.  float val_cp_p1 = getInterpolatedElement(referenceFrameImage,cpx+incx, cpy+incy, width);
8.  float val_cp_p2;
```

然后开始搜索，初始化计数信息

```
1.      int loopCounter = 0;
2.      float best_match_x = -1;
3.      float best_match_y = -1;
4.      float best_match_err = 1e50;
5.      float second_best_match_err = 1e50;
6.
7.      // best pre and post errors.
8.      float best_match_errPre=NAN, best_match_errPost=NAN, best_match_DiffErrPre=NAN,
    best_match_DiffErrPost=NAN;
9.      bool bestWasLastLoop = false;
10.
11.     float eeLast = -1; // final error of last comp.
12.
13.     // alternating intermediate vars
14.     float e1A=NAN, e1B=NAN, e2A=NAN, e2B=NAN, e3A=NAN, e3B=NAN, e4A=NAN, e4B=NAN, e5A=NAN,
    e5B=NAN;
15.
16.     int loopCBest=-1, loopCSecond =-1;
```

之后开始循环搜索，只要没有走到终点pClose或者loopCounter == 0就不断循环，以搜索误差最小的那个匹配点，我们把通过几何计算然后插值得到的那个点的信息作为标准信息，进行搜索，并计算误差

```
1.  // interpolate one new point
```

```
val_cp_p2 = getInterpolatedElement(referenceFrameImage,cpx+2*incx, cpy+2*incy, width);

// hacky but fast way to get error and differential error: switch buffer variables for last loop.
float ee = 0;
if(loopCounter%2==0)
{
    // calc error and accumulate sums.
    e1A = val_cp_p2 - realVal_p2;ee += e1A*e1A;
    e2A = val_cp_p1 - realVal_p1;ee += e2A*e2A;
    e3A = val_cp - realVal;      ee += e3A*e3A;
    e4A = val_cp_m1 - realVal_m1;ee += e4A*e4A;
    e5A = val_cp_m2 - realVal_m2;ee += e5A*e5A;
}
else
{
    // calc error and accumulate sums.
    e1B = val_cp_p2 - realVal_p2;ee += e1B*e1B;
    e2B = val_cp_p1 - realVal_p1;ee += e2B*e2B;
    e3B = val_cp - realVal;      ee += e3B*e3B;
    e4B = val_cp_m1 - realVal_m1;ee += e4B*e4B;
    e5B = val_cp_m2 - realVal_m2;ee += e5B*e5B;
}
```

然后判断是否误差更小，如果是，那么设置新的信息

```
if(ee < best_match_err)
{
    // put to second-best
    second_best_match_err=best_match_err;
    loopCSecond = loopCBest;

    // set best.
    best_match_err = ee;
    loopCBest = loopCounter;

    best_match_errPre = eeLast;
    best_match_DiffErrPre = e1A*e1B + e2A*e2B + e3A*e3B + e4A*e4B + e5A*e5B;
    best_match_errPost = -1;
    best_match_DiffErrPost = -1;

    best_match_x = cpx;
    best_match_y = cpy;
    bestWasLastLoop = true;
}
```

否则，如果上次循环是"好"的匹配循环，那么设置上次匹配的"好"参数

```
if(bestWasLastLoop)
{
    best_match_errPost = ee;
    best_match_DiffErrPost = e1A*e1B + e2A*e2B + e3A*e3B + e4A*e4B + e5A*e5B;
    bestWasLastLoop = false;
}
```

如果上次匹配不好，再判断它的误差是不是比次一个等级的匹配好，如果好，就设置，不好就算了

```
if(ee < second_best_match_err)
{
    second_best_match_err=ee;
    loopCSecond = loopCounter;
}
```

然后更新参数，进行下次循环

```
eeLast = ee;
val_cp_m2 = val_cp_m1; val_cp_m1 = val_cp; val_cp = val_cp_p1; val_cp_p1 = val_cp_p2;

if(enablePrintDebugInfo) stats->num_stereo_comparisons++;

cpx += incx;
cpy += incy;

loopCounter++;
```

循环完毕之后，自然找到了极线上匹配的最好的点，然后自然要判断和不合法，不合法就别继续了
这个判断分两部分，第一部分是判断误差是不是够小了，第二部分是判断匹配上的是不是够多

```
// if error too big, will return -3, otherwise -2.
if(best_match_err > 4.0f*(float)MAX_ERROR_STEREO)
{
    if(enablePrintDebugInfo) stats->num_stereo_invalid_bigErr++;
    return -3;
}


// check if clear enough winner
if(abs(loopCBest - loopCSecond) > 1.0f && MIN_DISTANCE_ERROR_STEREO * best_match_err >
second_best_match_err)
{
    if(enablePrintDebugInfo) stats->num_stereo_invalid_unclear_winner++;
    return -2;
}
```

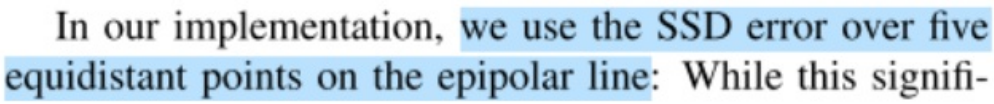
之后自然是计算准确的匹配(使用亚像素)
首先计算梯度(计算了一半)，然后判断各种各样的不合法情况
只有两种情况是可用的，即
1. gradPre_pre和gradPre_this不同号且gradPost_post和gradPost_this同号。认为前插可用
2. 上一种条件不满足情况下gradPost_post和gradPost_this不同号，认为后插值可用

然后这里就在做插值工作，得到亚像素的匹配误差

```
if(interpPre)
{
    float d = gradPre_this / (gradPre_this - gradPre_pre);
    best_match_x -= d*incx;
    best_match_y -= d*incy;
```

```
    best_match_err = best_match_err - 2*d*gradPre_this - (gradPre_pre - gradPre_this)*d*d;
    if(enablePrintDebugInfo) stats->num_stereo_interpPre++;
    didSubpixel = true;

}
else if(interpPost)
{
    float d = gradPost_this / (gradPost_this - gradPost_post);
    best_match_x += d*incx;
    best_match_y += d*incy;
    best_match_err = best_match_err + 2*d*gradPost_this + (gradPost_post - gradPost_this)*d*d;
    if(enablePrintDebugInfo) stats->num_stereo_interpPost++;
    didSubpixel = true;
}
```

于是就要做采样准备了，5个点，就论文这里


In our implementation, we use the SSD error over five equidistant points on the epipolar line: While this signifi-

```
    // sampleDist is the distance in pixel at which the realVal's were sampled
    float sampleDist = GRADIENT_SAMPLE_DIST*rescaleFactor;

    float gradAlongLine = 0;
    float tmp = realVal_p2 - realVal_p1;  gradAlongLine+=tmp*tmp;
    tmp = realVal_p1 - realVal;  gradAlongLine+=tmp*tmp;
    tmp = realVal - realVal_m1;  gradAlongLine+=tmp*tmp;
    tmp = realVal_m1 - realVal_m2;  gradAlongLine+=tmp*tmp;

    gradAlongLine /= sampleDist*sampleDist;

    // check if interpolated error is OK. use evil hack to allow more error if there is a lot of gradient.
    if(best_match_err > (float)MAX_ERROR_STEREO + sqrtf( gradAlongLine)*20)
    {
        if(enablePrintDebugInfo) stats->num_stereo_invalid_bigErr++;
        return -3;
    }
```

之后总算开始计算深度了，当然还有一会儿要用到的alpha

$$\alpha := \frac{\delta_d}{\delta_\lambda}$$

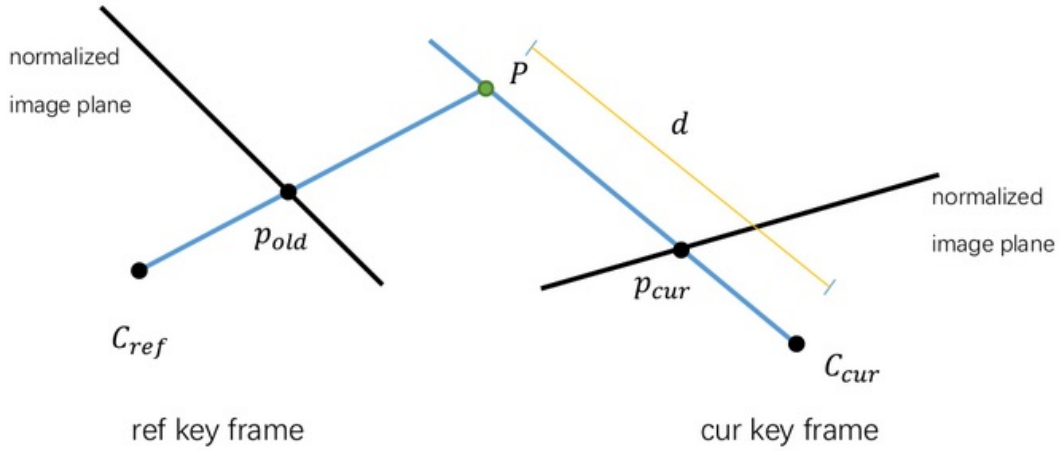
这个计算公式是用了向量的分量进行计算，首先的比较是看哪个分量比较精确用哪个（长点好，视差大）

```
float oldX = fxi*best_match_x+cxi;
float nominator = (oldX*referenceFrame->otherToThis_t[2] - referenceFrame->otherToThis_t[0]);
float dot0 = KinvP.dot(referenceFrame->otherToThis_R_row0);
float dot2 = KinvP.dot(referenceFrame->otherToThis_R_row2);

idnew_best_match = (dot0 - oldX*dot2) / nominator;
alpha = incx*fxi*(dot0*referenceFrame->otherToThis_t[2] - dot2*referenceFrame->otherToThis_t[0]) / (nominator*nominator);
```

```
float oldX = fxi*best_match_x+cxi; // fxi cxi 是 逆 内 参 数 矩 阵 中 的 元 素  K.inv().  best_match_x 逆 向 投 影 回 三 维 空 间
float nominator = (oldX*referenceFrame->otherToThis_t[2] - referenceFrame->otherToThis_t[0]);
float dot0 = KinvP.dot(referenceFrame->otherToThis_R_row0); //dot0 + t0 : KF中 三 维 空 间 坐 标 转 换 到  ref frame 中 的  x轴 坐 标
float dot2 = KinvP.dot(referenceFrame->otherToThis_R_row2);

idnew_best_match = (dot0 - oldX*dot2) / nominator;
alpha = incx*fxi*(dot0*referenceFrame->otherToThis_t[2] - dot2*referenceFrame->otherToThis_t[0]) / (nominator*nominator);
```

推导如下:

逆深度计算公式推导如下:



$$p_{cur} = K_{inv}\begin{pmatrix} x \\ y \\ 1 \end{pmatrix}, \quad p_{old} = K_{inv}\begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

$$\begin{aligned} P_{ref} &= R_{ref_{cur}} P_{cur} + t_{ref_{cur}} \\ &= R_{ref_{cur}} * d * p_{cur} + t_{ref_{cur}} \end{aligned}$$

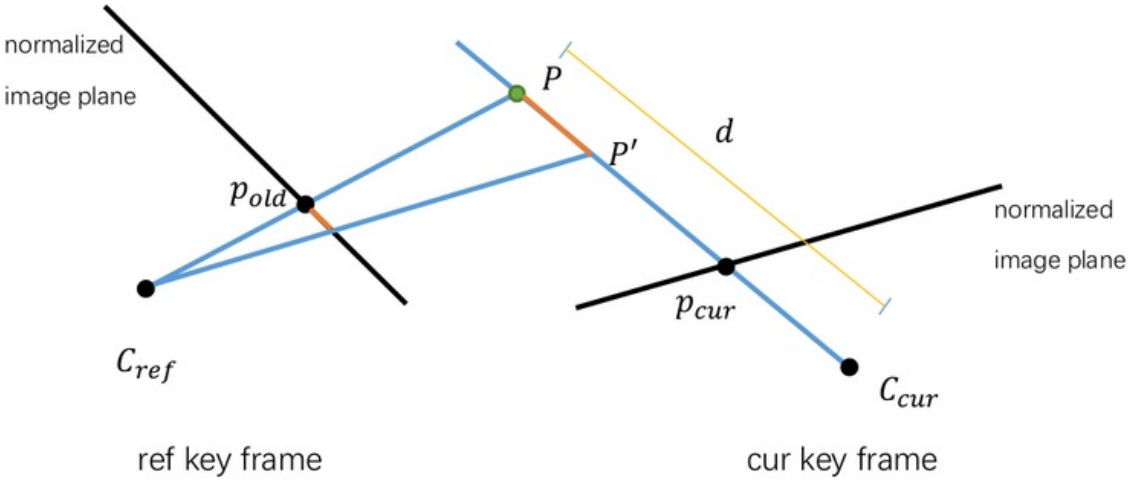
上式两边只有 d 为未知数，利用横坐标或者纵坐标相等就能计算出 d，比如用横坐标:

$$P_{ref_x} = d * R_{row0} \cdot p_{cur} + t_0 = d * dot_0 + t_0$$
$$P_{ref\_z} = dR_{row3} * p_{cur} + t_2 = d * dot_2 + t_2$$
$$p_{old\_x} = old_x = \frac{P_{ref_x}}{P_{ref\_z}} = \frac{d * dot_0 + t_0}{d * dot_2 + t_2}$$
$$d * (old_x * dot_2 - dot_0) = t_0 - old_x * t_2$$

得到逆深度的计算公式：

$$\frac{1}{d} = \frac{old_x * dot_2 - dot_0}{t_0 - old_x * t_2}$$

alpha 的推导如下



像素沿着外极线每步进一个单位时的增量为$(incx,\ incy)$：

这可以从代码中验证：

```
float incx = pClose[0] - pFar[0];
float incy = pClose[1] - pFar[1];
float eplLength = sqrt(incx*incx+incy*incy);

incx *= GRADIENT_SAMPLE_DIST/eplLength;
incy *= GRADIENT_SAMPLE_DIST/eplLength;
```

其中 GRADIENT_SAMPLE_DIST = 1，所以$1 = incx^2 + incy^2$

将步进单位从像素平面反投影到归一化图像平面得到$incx * f_{xinv}$，f_{xinv}是逆内参数矩阵对应元素。在之前的计算中，我们已经知道了

$$\rho_P = \frac{old_x * dot_2 - dot_0}{t_0 - old_x * t_2}$$

它只和变量old_x有关，所以在外极线上步进一个像素单位以后得到三维点P'在 cur frame 中的逆深度为：

$$\rho_{P'} = \frac{(old_x + incx * f_{xi}) * dot_2 - dot_0}{t_0 - (old_x + incx * f_{xi}) * t_2}$$

由α的计算公式有（Semi-Dense Visual Odometry for a Monocular Camera）：

$$\begin{aligned}
\alpha = \frac{\delta_\rho}{\delta_\lambda} &= \frac{\rho_{P'} - \rho_P}{1} \\
&= \frac{(old_x + incx * f_{xi}) * dot_2 - dot_0}{t_0 - (old_x + incx * f_{xi}) * t_2} - \frac{old_x * dot_2 - dot_0}{t_0 - old_x * t_2} \\
&= \frac{incx * fxi * dot_2 + old_x * dot_2 - dot_0}{t_0 - old_x * t_2 - (incx * fxi) * t_2} - \frac{old_x * dot_2 - dot_0}{t_0 - old_x * t_2} \\
&= \frac{incx * fxi * (dot_0 * t_2 - dot_2 * t_0)}{(t_0 - old_x * t_2 - (incx * fxi) * t_2)(t_0 - old_x * t_2)} \\
&\approx \frac{incx * fxi * (dot_0 * t_2 - dot_2 * t_0)}{(t_0 - old_x * t_2)(t_0 - old_x * t_2)}
\end{aligned}$$

后面部分在算协方差了，公式就在论文上，都比较简单，就略过了，请读者自行研究，之后就可以回到updateKeyFrame了

实际上上面那个部分结束之后，你发现整个DepthMap的函数已经差不多了。接下来我们来看下面的调用 regularizeDepthMap

```
1.      //if(rand()%10==0)
2.      {
3.          gettimeofday(&tv_start, NULL);
4.          regularizeDepthMapFillHoles();
5.          gettimeofday(&tv_end, NULL);
6.          msFillHoles = 0.9*msFillHoles + 0.1*((tv_end.tv_sec-tv_start.tv_sec)*1000.0f + (tv_end
    .tv_usec-tv_start.tv_usec)/1000.0f);
7.          nFillHoles++;
8.      }
```

```
9.
10.     gettimeofday(&tv_start, NULL);
11.     regularizeDepthMap(false, VAL_SUM_MIN_FOR_KEEP);
12.     gettimeofday(&tv_end, NULL);
13.     msRegularize = 0.9*msRegularize + 0.1*((tv_end.tv_sec-tv_start.tv_sec)*1000.0f + (tv_end.t
    v_usec-tv_start.tv_usec)/1000.0f);
14.     nRegularize++;
```

这里主要是调用了两个函数regularizeDepthMapFillHoles和regularizeDepthMap，就按照调用顺序依次介绍吧

DepthMap::regularizeDepthMapFillHoles

这个函数又是bind映射机制，前面已经讲过了如何进行的，这里我们就直接进入函数regularizeDepthMapFillHolesRow

```
1.  const float* keyFrameMaxGradBuf = activeKeyFrame->maxGradients(0);
```

本地化梯度参数，进入循环

```
1.  int idx = x+y*width;
2.  DepthMapPixelHypothesis* dest = otherDepthMap + idx;
3.  if(dest->isValid) continue;
4.  if(keyFrameMaxGradBuf[idx]<MIN_ABS_GRAD_DECREASE) continue;
```

然后判断那块区域有效点的数据是否足够多

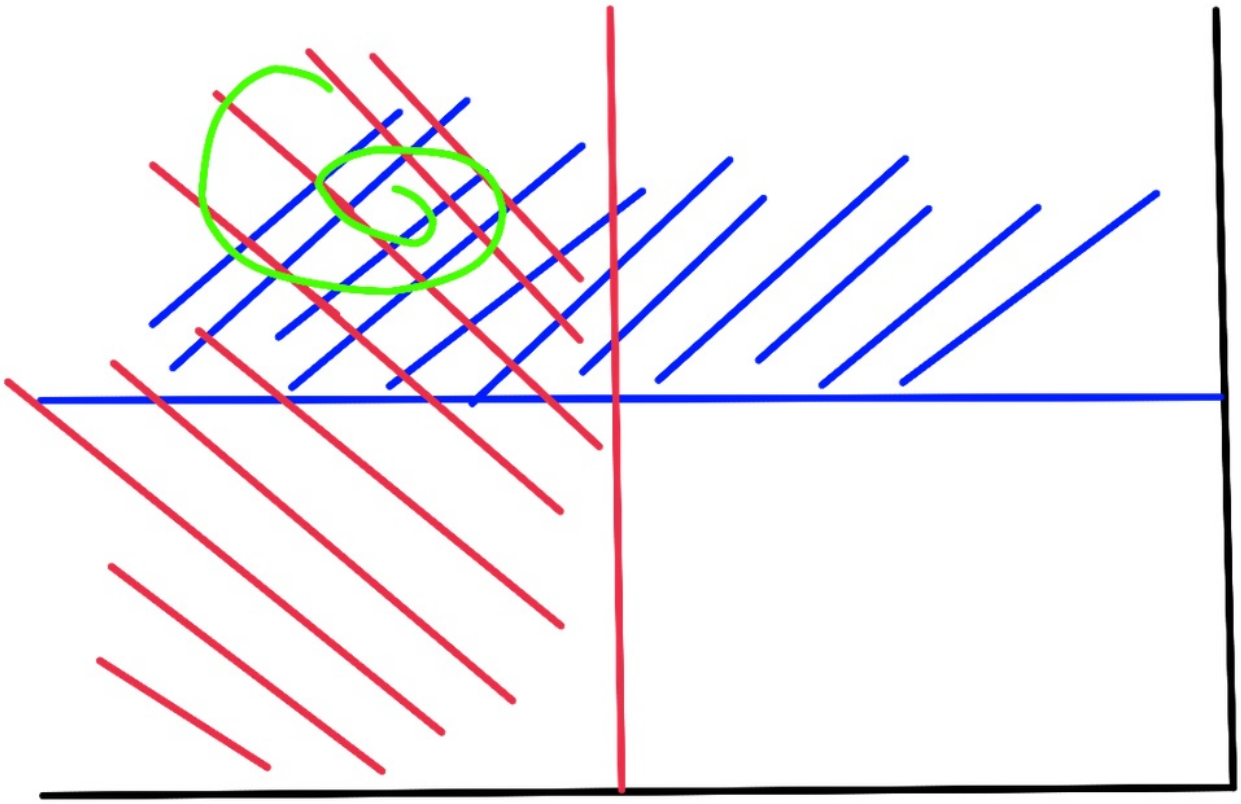
```
1.  int* io = validityIntegralBuffer + idx;
2.  int val = io[2+2*width] - io[2-3*width] - io[-3+2*width] + io[-3-3*width];
```

我估计这里看起来又是一段莫名其妙理解不能的代码，我在这里讲解一下
首先我们来看这个东西是在哪里初始化的，请看DepthMap::buildRegIntegralBufferRow1函数

```
1.  void DepthMap::buildRegIntegralBufferRow1(int yMin, int yMax, RunningStats* stats)
2.  {
3.      // ============ build inegral buffers
4.      int* validityIntegralBufferPT = validityIntegralBuffer+yMin*width;
5.      DepthMapPixelHypothesis* ptSrc = currentDepthMap+yMin*width;
6.      for(int y=yMin;y<yMax;y++)
7.      {
8.          int validityIntegralBufferSUM = 0;
9.
10.         for(int x=0;x<width;x++)
11.         {
12.             if(ptSrc->isValid)
13.                 validityIntegralBufferSUM += ptSrc->validity_counter;
14.
15.             *(validityIntegralBufferPT++) = validityIntegralBufferSUM;
16.             ptSrc++;
17.         }
18.     }
19. }
```

你会发现，validityIntegralBuffer实际上记录了到当前位置所有的有效点的和，那么上面那一段加加减减（ int val =

io[2+2*width] - io[2-3*width] - io[-3+2*width] + io[-3-3*width]) 是干嘛呢，请看下图



首先是那个大区域，减去红色的，减去蓝色的，再加上绿色的，是不是就只剩下的未填色部分了呢？
因此这里实际上统计了未填色部分的有效点的总数，然后下面在判断满不满足要求，满足之后进行接下来的操作
然后是这段代码：

```
DepthMapPixelHypothesis* s1max = otherDepthMap + (x-2) + (y+3)*width;
for (DepthMapPixelHypothesis* s1 = otherDepthMap + (x-2) + (y-2)*width; s1 < s1max; s1+=width)
    for(DepthMapPixelHypothesis* source = s1; source < s1+5; source++)
    {
        if(!source->isValid) continue;

        sumIdepthObs += source->idepth /source->idepth_var;
        sumIVarObs += 1.0f/source->idepth_var;
        num++;
    }

float idepthObs = sumIdepthObs / sumIVarObs;
idepthObs = UNZERO(idepthObs);

currentDepthMap[idx] =
    DepthMapPixelHypothesis(
        idepthObs,
        VAR_RANDOM_INIT_INITIAL,
        0);
```

实际上这里就是算了一下加权平均，没啥特别的，然后把估计更新了进去，实际上就是预测了一些未知区域(洞)的深度，
注意这个预测是要在周围都比较好的基础上才进行

DepthMap::regularizeDepthMap

多的废话我已经不想说了，既然能看到这个部分，足见读者编程水平已经不低，多的废话我就不说了，这个函数用和之前一样的线程机制调用了DepthMap::regularizeDepthMapRow，这个函数使用了模板技术，它有是否要移除遮挡的两个版本，在预编译期已经生成，函数本身蛮简单的，其实就是算一个加权平均，关键代码如下：

```
for(int y=yMin;y<yMax;y++)
{
    for(int x=regularize_radius;x<width-regularize_radius;x++)
    {
        DepthMapPixelHypothesis* dest = currentDepthMap + x + y*width;
        DepthMapPixelHypothesis* destRead = otherDepthMap + x + y*width;

        float sum=0, val_sum=0, sumIvar=0;//, min_varObs = 1e20;
        int numOccluding = 0, numNotOccluding = 0;

        for(int dx=-regularize_radius; dx<=regularize_radius;dx++)
            for(int dy=-regularize_radius; dy<=regularize_radius;dy++)
            {
                DepthMapPixelHypothesis* source = destRead + dx + dy*width;

                if(!source->isValid) continue;

                float diff =source->idepth - destRead->idepth;
                if(DIFF_FAC_SMOOTHING*diff*diff > source->idepth_var + destRead->idepth_var)
                    continue;

                val_sum += source->validity_counter;

                float distFac = (float)(dx*dx+dy*dy)*regDistVar;
                float ivar = 1.0f/(source->idepth_var + distFac);

                sum += source->idepth * ivar;
                sumIvar += ivar;
            }

        if(val_sum < validityTH)
        {
            dest->isValid = false;
            continue;
        }

        sum = sum / sumIvar;
        sum = UNZERO(sum);

        // update!
        dest->idepth_smoothed = sum;
        dest->idepth_var_smoothed = 1.0f/sumIvar;
    }
}
```

调用完这个函数后，实际上DepthMap::updateKeyframe中干活的部分就都结束了，剩下都是一些记录工作和调试选项的工作，请读者自行阅读

void DepthMap::propagateDepth

实际上到了这里，我应该讲清楚的是createKeyFrame函数，但是这个函数和上面那个函数重叠部分太多，之后一个关键函数，就是DepthMap::propagateDepth了，我就简单说下这个深度传播的函数，省的和前面的部分重复

```
for(DepthMapPixelHypothesis* pt = otherDepthMap+width*height-1; pt >= otherDepthMap; pt--)
{
    pt->isValid = false;
    pt->blacklisted = 0;
}
```

首先当然是初始化工作，代码如上，然后是本地化参数

```
SE3 oldToNew_SE3 = se3FromSim3(new_keyframe->pose->thisToParent_raw).inverse();
Eigen::Vector3f trafoInv_t = oldToNew_SE3.translation().cast<float>();
Eigen::Matrix3f trafoInv_R = oldToNew_SE3.rotationMatrix().matrix().cast<float>();


const bool* trackingWasGood = new_keyframe->getTrackingParent() == activeKeyFrame ?
new_keyframe->refPixelWasGoodNoCreate() : 0;


const float* activeKFImageData = activeKeyFrame->image(0);
const float* newKFMaxGrad = new_keyframe->maxGradients(0);
const float* newKFImageData = new_keyframe->image(0);
```

首先的到新坐标下的当前点，同时算出深度和图像上的位置

```
Eigen::Vector3f pn = (trafoInv_R * Eigen::Vector3f(x*fxi + cxi,y*fyi + cyi,1.0f)) / source->id
epth_smoothed + trafoInv_t;

float new_idepth = 1.0f / pn[2];

float u_new = pn[0]*new_idepth*fx + cx;
float v_new = pn[1]*new_idepth*fy + cy;
```

然后得到最大梯度，以及最好的估计目标

```
int newIDX = (int)(u_new+0.5f) + ((int)(v_new+0.5f))*width;
float destAbsGrad = newKFMaxGrad[newIDX];

...

DepthMapPixelHypothesis* targetBest = otherDepthMap +  newIDX;
```

之后计算新的方差

$$\sigma_{d_1}^2 = J_{d_1} \sigma_{d_0}^2 J_{d_1}^T + \sigma_P^2 = \left(\frac{d_1}{d_0}\right)^4 \sigma_{d_0}^2 + \sigma_P^2$$

```
float idepth_ratio_4 = new_idepth / source->idepth_smoothed;
idepth_ratio_4 *= idepth_ratio_4;
idepth_ratio_4 *= idepth_ratio_4;
```

然后去除遮挡，策略是如果之前那个bestTarget是有效的，如果一下子变化特别大，那么认为是被遮挡了

```
// check for occlusion
if(targetBest->isValid)
{
    // if they occlude one another, one gets removed.
    float diff = targetBest->idepth - new_idepth;
    if(DIFF_FAC_PROP_MERGE*diff*diff >
        new_var +
        targetBest->idepth_var)
    {
        if(new_idepth < targetBest->idepth)
        {
            if(enablePrintDebugInfo) runningStats.num_prop_occluded++;
            continue;
        }
        else
        {
            if(enablePrintDebugInfo) runningStats.num_prop_occluded++;
            targetBest->isValid = false;
        }
    }
}
```

之后是更新深度信息，如果bestTarget是无效的，那么创建一个新的估计

```
*targetBest = DepthMapPixelHypothesis(
        new_idepth,
        new_var,
        source->validity_counter);
```

如果是有效的，需要先merge，再更新，实际上就是使用了扩散卡尔曼滤波，最后再更新到当前深度,实际上就是对应论文这个部分

vation is incorporated into the prior, i.e., the two distributions are multiplied (corresponding to the update step in a

Kalman filter): Given a prior distribution $\mathcal{N}(d_p, \sigma_p^2)$ and a noisy observation $\mathcal{N}(d_o, \sigma_o^2)$, the posterior is given by

$$\mathcal{N}\left(\frac{\sigma_p^2 d_o + \sigma_o^2 d_p}{\sigma_p^2 + \sigma_o^2}, \frac{\sigma_p^2 \sigma_o^2}{\sigma_p^2 + \sigma_o^2}\right). \quad (13)$$

```
// merge idepth ekf-style
float w = new_var / (targetBest->idepth_var + new_var);
```

```
float merged_new_idepth = w*targetBest->idepth + (1.0f-w)*new_idepth;

// merge validity
int merged_validity = source->validity_counter + targetBest->validity_counter;
if(merged_validity > VALIDITY_COUNTER_MAX+(VALIDITY_COUNTER_MAX_VARIABLE))
    merged_validity = VALIDITY_COUNTER_MAX+(VALIDITY_COUNTER_MAX_VARIABLE);

*targetBest = DepthMapPixelHypothesis(
        merged_new_idepth,
        1.0f/(1.0f/targetBest->idepth_var + 1.0f/new_var),
        merged_validity);

...

// swap!
std::swap(currentDepthMap, otherDepthMap);
```

GlobalMapping

地图优化可以说是整个技术里面最简单的一部分了，核心其实就是g2o，可以看到这个里面代码也不多，总共加起来不到1000行，实际上就是这些工作，都是在为g2o服务

g2oTypeSim3Sophus

这个文件里面定义了图优化需要使用的基本定点class VertexSim3 : public g2o::BaseVertex<7, Sophus::Sim3d>和边class EdgeSim3 : public g2o::BaseBinaryEdge<7, Sophus::Sim3d, VertexSim3, VertexSim3>
在这之前，我们需要知道优化的对象是什么，请看论文中的公式

$$E(\boldsymbol{\xi}_{W1} \dots \boldsymbol{\xi}_{Wn}) := \sum_{(\boldsymbol{\xi}_{ji}, \boldsymbol{\Sigma}_{ji}) \in \mathcal{E}} (\boldsymbol{\xi}_{ji} \circ \boldsymbol{\xi}_{Wi}^{-1} \circ \boldsymbol{\xi}_{Wj})^T \boldsymbol{\Sigma}_{ji}^{-1} (\boldsymbol{\xi}_{ji} \circ \boldsymbol{\xi}_{Wi}^{-1} \circ \boldsymbol{\xi}_{Wj}).$$

实际上可以从代码和论文中明白，优化的是sim3(顶点)，代表的是相机坐标的位置，通过约束sim3((边)，代表的是两个坐标之间的变换关系，他们是g2o的基本元素，请自行研究g2o库的使用方法，即明白这里的函数都有什么作用

TrackableKeyFrameSearch

这个类主要是用于查找可用可keyFrame的，让我们直接进入函数

TrackableKeyFrameSearch::findCandidates

不得不说这是一个返回值及其扭曲的函数，下面这么长一串全部是它的返回值

```
std::unordered_set<Frame*, std::hash<Frame*>,
                std::equal_to<Frame*>,
                Eigen::aligned_allocator< Frame* > >
```

看起来比较难受，其实是返回了一个hash_set，调用的是std::hash的方法吧指针hash，应该会调用这个模板（我没仔细看，不过十有八九是这个）

```
template<typename _Tp>
  static size_t
  hash(const _Tp& __val)
  { return hash(&__val, sizeof(__val)); }
```

比较方法是std::equal_to，实际上就是调用了下面这个模板

```
template<typename _Tp>
struct equal_to : public binary_function<_Tp, _Tp, bool>
{
  bool
  operator()(const _Tp& __x, const _Tp& __y) const
  { return __x == __y; }
};
```

最后一个是空间配置器，选用了eigen的空间配置器
传入值是简单的，即一个Frame的指针，一个Frame指针的引用，判断是否使用了FABMAP(本文默认没有)，最后是个阈值，之后向下调用了函数findEuclideanOverlapFrames，计算符合要求的Frame，实际上返回了个TrackableKFStruct结构体

```
struct TrackableKFStruct
{
    Frame* ref;
    SE3 refToFrame;
    float dist;
    float angle;
};
```

findEuclideanOverlapFrames

首先计算了阈值，然后本地化了照相机到世界坐标的参数，计算尺度给后面检测使用(传入参数为true)

```
float cosAngleTH = cosf(angleTH*0.5f*(fowX + fowY));

Eigen::Vector3d pos = frame->getScaledCamToWorld().translation();
Eigen::Vector3d viewingDir = frame->getScaledCamToWorld().rotationMatrix().rightCols<1>();

float distFacReciprocal = 1;
if(checkBothScales)
    distFacReciprocal = frame->meanIdepth / frame->getScaledCamToWorld().scale();
```

之后进入循环，首先计算两帧之间的距离和角度，并进行阈值检测

```
Eigen::Vector3d otherPos = graph->keyframesAll[i]->getScaledCamToWorld().translation();

// get distance between the frames, scaled to fit the potential reference frame.
float distFac = graph->keyframesAll[i]->meanIdepth / graph->keyframesAll[i]->getScaledCamToWorld().scale();
if(checkBothScales && distFacReciprocal < distFac) distFac = distFacReciprocal;
```

```
Eigen::Vector3d dist = (pos - otherPos) * distFac;
float dNorm2 = dist.dot(dist);
if(dNorm2 > distanceTH) continue;

Eigen::Vector3d otherViewingDir = graph->keyframesAll[i]->getScaledCamToWorld().rotationMatrix().rightCols<1>();    float dirDotProd = otherViewingDir.dot(viewingDir);
if(dirDotProd < cosAngleTH) continue;
```

如果都符合了要求(没有continue)，那么压入potentialReferenceFrames

```
potentialReferenceFrames.push_back(TrackableKFStruct());
potentialReferenceFrames.back().ref = graph->keyframesAll[i];
potentialReferenceFrames.back().refToFrame = se3FromSim3(graph->keyframesAll[i]->getScaledCamToWorld().inverse() * frame->getScaledCamToWorld()).inverse();
potentialReferenceFrames.back().dist = dNorm2;
potentialReferenceFrames.back().angle = dirDotProd;
```

那么之后，就可以在TrackableKeyFrameSearch中得到潜在的帧，之后循环把参考帧都压入result,最后返回result

```
for(unsigned int i=0;i<potentialReferenceFrames.size();i++)
    results.insert(potentialReferenceFrames[i].ref);
```

TrackableKeyFrameSearch::findRePositionCandidate

这个函数也是首先调用findEuclideanOverlapFrames查找到潜在的参考帧，然后循环，进入循环之后先判断，然后调用了checkPermaRefOverlap得到可用点的分数，并做了些记录

```
if(frame->getTrackingParent() == potentialReferenceFrames[i].ref)
    continue;

if(potentialReferenceFrames[i].ref->idxInKeyframes < INITIALIZATION_PHASE_COUNT)
    continue;

struct timeval tv_start, tv_end;
gettimeofday(&tv_start, NULL);
tracker->checkPermaRefOverlap(potentialReferenceFrames[i].ref, potentialReferenceFrames[i].refToFrame);
gettimeofday(&tv_end, NULL);
msTrackPermaRef = 0.9*msTrackPermaRef + 0.1*((tv_end.tv_sec-tv_start.tv_sec)*1000.0f + (tv_end.tv_usec-tv_start.tv_usec)/1000.0f);
nTrackPermaRef++;
```

之后计算参考帧分数，调用getRefFrameScore

```
inline float getRefFrameScore(float distanceSquared, float usage)
{
    return distanceSquared*KFDistWeight*KFDistWeight
            + (1-usage)*(1-usage) * KFUsageWeight * KFUsageWeight;
}
```

然后判断这个分数够不够高，如果不够就重新算分,方法是快速tracking，然后再次计算dist，再调用getRefFrameScore，这样就可以计算到分数，之后计算差异，然后更新goodVal

```
SE3 RefToFrame_tracked = tracker->trackFrameOnPermaref(potentialReferenceFrames[i].ref, frame, potentialReferenceFrames[i].refToFrame);
Sophus::Vector3d dist = RefToFrame_tracked.translation() * potentialReferenceFrames[i].ref->meanIdepth;

float newScore = getRefFrameScore(dist.dot(dist), tracker->pointUsage);
float poseDiscrepancy = (potentialReferenceFrames[i].refToFrame * RefToFrame_tracked.inverse()).log().norm();
float goodVal = tracker->pointUsage * tracker->lastGoodCount / (tracker->lastGoodCount+tracker->lastBadCount);
checkedSecondary++;
```

之后判断这个新的tracking符不符合要求，如果符合，就更新全局的最优数据

```
bestPoseDiscrepancy = poseDiscrepancy;
bestScore = score;
bestFrame = potentialReferenceFrames[i].ref;
bestRefToFrame = potentialReferenceFrames[i].refToFrame;
bestRefToFrame_tracked = RefToFrame_tracked;
bestDist = dist.dot(dist);
bestUsage = tracker->pointUsage;
```

循环完成之后，判断找没找到，找到了就返回帧的指针，没找到返回空指针

KeyFrameGraph

这个类用于创建g2o需要使用的图，里面有的函数分别有插入帧，插入关键帧，加入约束等，这些函数都很简单，稍微长一点的就是void KeyFrameGraph::dumpMap(std::string folder),然而这个函数点进去看之后，发现竟然是保存数据用的，这个类中所有函数都巨简单无比，请读者自行研究