# 第一阶段

**\* 首先通过碱基质量分数进行重矫正。基于不同的协变量生成一个重矫正的table。默认的协变量是read group，reported quality score，machine cycle，和nucleotide context。**

**\* 该walker产生基于特定协变量的tables。仅仅在已知的变异集合VCF中通过基因座（轨迹）执行遍历. ExAc,genomAD,或者dbsnp资源可以被用作已知变异位点集合。我们假设我们看到的所有reference mismatch都是错误，并表示为低质量碱基。因为那里有一大堆数据被用来计算empirical error的可能性，在该位点看到指定的协变量，P(error)=num mismatches/num observations。输出的是表格文件，（一些协变量的值，num observations，num mismatches，empirical quality score）**

**\* 输入的read数据，需要评估其碱基质量分数**

**\* 可跳过的已知多态站点的数据库**

**\* 有很多表格的GATK报告文件：**

**参数列表；量化的质量表；通过read group生成的重矫正表；通过质量分数生成的重矫正表；针对所有的可选协变量生成重矫正表；**

**\*GATK报告要要趋向于易于读取和计算，查看GATKReport的文档，了解如何操作这个表**

**// 生成BQSR重矫正表格**

**// BQSR第一步：基于用户指定的协变量生成重矫正表格；协变量有read group，reported quality score，machine cycle，和nucleotide context**

**public final class** BaseRecalibrator **extends** ReadWalker

1 表示针对BQSR所有的命令行参数以及其协变量

@ArgumentCollection(doc=**"all the command line arguments for BQSR and its covariates"**)  
**private final** RecalibrationArgumentCollection **recalArgs** = **new** RecalibrationArgumentCollection();

2 本算法是将每一个reference mismatch 表示为一个error，然而真正的基因突变，我们所期望的是mismatch 到reference。因此给定本工具一个真正的多态变异位点的数据集（dbsnp）显得尤为重要，同时也会将会跳过这些点。本工具接收包含VCF，BCF，BED等任意类型的文件作为数据集。对于用户期望排除这些区间内的已知变异仅仅使用-XL my.interval.list 从而跳过处理这些位点。然而请注意这些通过工具的-XL参数反映并过滤掉的位点，并不是真正的变异位点。

@Argument(fullName = ***KNOWN\_SITES\_ARG\_FULL\_NAME***, doc = **"One or more databases of known polymorphic sites used to exclude regions around known polymorphisms from analysis."**, optional = **false**)  
**private** List<FeatureInput<Feature>> **knownSites**;

3 header之后，data统计了每一行的一个（read），直到文件末尾。一行的前几个项目是各自独立的协变量值，在运行时将会根据那个协变量发生变化。Data的最后三个项目，是对于组合协变量的观察数，reference的mismatch数和通过phred-scaling the mismatch rate计算的原始empirical quality score。使用/dev/stdout打印到标准输出

@Argument(shortName = StandardArgumentDefinitions.***OUTPUT\_SHORT\_NAME***, fullName = StandardArgumentDefinitions.***OUTPUT\_LONG\_NAME***, doc = **"The output recalibration table file to create"**, optional = **false**)  
**private** File **recalTableFile** = **null**;

**private** BaseRecalibrationEngine **recalibrationEngine**;

4 保存用户传递的reference的数据资源，我们正在使用一个来自不同于engine本身的缓存数据，以避免干扰它的缓存

**private** ReferenceDataSource **referenceDataSource**;

5 an object用来对quality score quantization追踪有必要的信息。

**private** QuantizationInfo **quantizationInfo** = **null**;

6 解析-cov 协变量参数并建立协变量列表，基于协变量，评估并分配初始化data hashmap的容量。（第一阶段）

**public void** onTraversalStart()

7 获取默认的过滤read(第二阶段)

**public** List<ReadFilter> getDefaultReadFilters(){

7.1 返回原始read filters全列表，用于BQSR contexts，包含WellFormed

**public static** List<ReadFilter> getStandardBQSRReadFilterList()

7.2 返回基本元素的列表，原始read filters用于BQSR contexts，不包含WellFormed

**public static** List<ReadFilter> getBQSRSpecificReadFilterList()

}

8

// 父类ReadWalker.java方法说明{

// 处理一个一个单独的read（有可选的Contextual信息）必须被工具使用者实现

// 通常，工具的使用者应该简单从apply()输出流，并尽可能保留一点内在的状态

//Todo: 确定GATK引擎是否以及在何种程度上应该提供减少操作

//Todo: 补充这个操作。至少，我们应该让apply()返回一个值

// Todo: 在walkers不鼓励有状态，但是如何处理这个值是TBD

}

**public final class** BaseRecalibrator **extends** ReadWalker

**public abstract class** ReadWalker **extends** GATKTool

针对每条read在该位点（基因座）获取不同的协变量值，并基于该碱基是‑否匹配到reference特定的位置。在map中增加该位置。

@Override  
**public void** apply( GATKRead read, ReferenceContext ref, FeatureContext featureContext ) {

// 针对每条read在该位点（基因座）获取不同的协变量值，并基于该碱基是‑否匹配到reference特定的位置。在map中增加该位置。（第三阶段）  
 **recalibrationEngine**.processRead(read, **referenceDataSource**, featureContext.getValues(**knownSites**));  
}

9 GATKTool.java{

// 在成功遍历之后立即执行的操作，（等价于在遍历期间没有抛出未捕获到的异常）

// 在遍历之后，应该被工具使用者覆盖，并需要关闭本地资源等。

// 而且允许工具返回一个值表示遍历的结果，应该被engine输出

// 默认实现是什么都不做，返回null

// 返回一个表示遍历结果的目标，或者假如工具没有返回值就是null

}

**public** Object onTraversalSuccess(){

// 如果合适的话，在重新校准表中得到所有数据。

// 所有的read调用完processRead之后，被调用一次

// 假设所有主要的表格（通过质量分数）已经被完全更新了

// 遍历此数据以创建汇总数据表，如通过 read group表。

**recalibrationEngine**.finalizeData();{

// 解释同上

*finalizeRecalibrationTables*(**recalTables**);（第四阶段）

finalized = true

}

// 计算量化的质量分数

quantizeQualityScores();（10）

// 输出重矫正的报告

generateReport();(11)

}

10 //通过quality score表和使用observations和empirical quality score针对quantization去建立quality score 直方图。然后利用量化的直方图去生成量化的map<recalibrated\_qual -> quantized\_qual>

// recalArgs.QUANTIZING\_LEVELS=16：BQSR生成一个量化的表格尊对快速量化，通过随后的工具，BQSR不会量化碱基质量，该操作是利用-qq 或-bqsr选项被engine所执行， 该参数告诉BQSR建立量化表格时，量化的水平数量

// 在执行finalizeData()之后，获取最终的重矫正表格

// 通过该engine收集并返回最终的最终的重矫正表

// 在finalizeData被调用前，执行该方法会产生错误

// 通过该engine收集得到，并返回最后的重矫正表格，

**private void** quantizeQualityScores() {

// 该类封装BQSR质量分数量化所必需的信息  
 **quantizationInfo** = **new** QuantizationInfo(**recalibrationEngine**.getFinalRecalibrationTables(), **recalArgs**.**QUANTIZING\_LEVELS**);(第五阶段)  
}

11 //生成报告

//

RecalUtils.*outputRecalibrationReport*(recalTableStream, **recalArgs**, **quantizationInfo**, **recalibrationEngine**.getFinalRecalibrationTables(), **recalibrationEngine**.getCovariates());（**第六阶段**）

# 第二阶段

第一阶段：

**public void** apply( GATKRead read, ReferenceContext ref, FeatureContext featureContext ) {  
 **recalibrationEngine**.processRead(read, **referenceDataSource**, featureContext.getValues(**knownSites**));  
}

**public final class** BaseRecalibrationEngine **implements** Serializable

1 每个调用的EventType.values(),或者任何枚举类型，建立一个新的数组实例，但是他们都是相等的（等价于包含完全相同的元素。）当这个数据在BQSR案例中被建立上10亿次，这将是昂贵的并且有巨大浪费的。解决办法就是缓存该数组

2 对于该位点为每条read获取不同的协变量值，并基于该碱基是否匹配到reference特定的位置。将该位置（位点）添加到map中。（第二阶段）

3 处理Read，

3.1 首先进行Read转化，

consolidateCigar合并Cigar String；

设置碱基质量的默认值；

重置原始碱基质量；

硬跳过接头序列（切掉接头序列）；

切掉软跳过的碱基）

3.2 解析read的platform

3.3 修正SNP，Insertion和Deletion参数，不能被跳过，BAQ或非BAQ（碱基比对质量）

计算Errors（定位所有的SNP和INDEL事件，将他们保存在提供的snp，isIns和isDel的数组中，并返回总的SNP/INDEL事件总数）

获取refBases

获取read的CigarElements

M;EQ,X:记录snp的数量

D: 记录Deletion的索引位置及变异长度

N： 跳过参考序列的长度

I： 记录InSertion的变异

S： 软跳过的长度：ReferenceContext没有软跳过的碱基

H:P 不操作。

否则： 就是GATK不识别的cigar操作符

返回SNP和INDEL的事件总数，GATK不会把他们相加起来，因为有可能会将相同的地方设置为1为两次（不懂）

3.4 注意：鉴于效率考虑，我们没有计算BAQ的数组，除非我们真的需要。忽略了一些error。对于ILMN85%的数据reads没有error

3.5 Errors（SNP和INDEL总数等于0）或者不进行BAQ的话：flatBAQArray（创建一个BAQ类型的数组，意味着没有比对的不确定性。做flatBAQArray可以被优化，完全可以避免这种函数，通过使用内联，假如前面有计算的代码）。或者：calculateBAQArray(针对read计算真实的BAQ数组，基于read的质量和reference序列，)

3.6

**public void** processRead( **final** GATKRead originalRead, **final** ReferenceDataSource refDS, **final** Iterable<? **extends** Locatable> knownSites )

**public interface** ReadTransformer **extends** UnaryOperator<GATKRead>, SerializableFunction<GATKRead, GATKRead> {

**private static** GATKRead consolidateCigar( **final** GATKRead read )

**public interface** GATKRead **extends** Locatable

**public interface** Locatable

// 任何有一个单独逻辑mapping到基因上的class应该实现Localtable接口；记录位点应该从1开始（1-based），并终止于两端

**private static** GATKRead consolidateCigar( **final** GATKRead read )

**public static** Cigar consolidateCigar( **final** Cigar c )

**public class** Cigar **implements** Serializable, Iterable<CigarElement>

# 第三阶段：

1 利用empirical 质量分布创建直方图（QualityUtils.***MAX\_SAM\_QUAL\_SCORE***

=93）

2 获得质量分数表

3 转换empirical质量为整数（它已经被MAX\_QUAL所capped）

4 给每一个key添加观察的碱基数量

5 直方图拥有每个观察到的碱基empirical 质量值

6 针对期望的水平数量量化质量值

7 利用原始的map到量化的qual map(在RAC中使用标准的水平分数)