

程序相关备忘 V3.0

王仁仲 2015 年 6 月

目录

Linux 方面	- 1 -
Linux 下 NTFS 格式的硬盘挂载	- 1 -
Linux 下 shell 脚本的一些注意事项	- 1 -
合并 eps 或是 pdf 格式的文件	- 1 -
文件的压缩与解压缩	- 1 -
搜寻某个文件或文件夹	- 1 -
寻找文件夹中包含某个字符串的文件	- 2 -
关于 64 位 Linux 链接 Geant4 库失败	- 2 -
C/C++ 方面	- 3 -
格式化输出与输入	- 3 -
C++中类的静态成员，以及静态函数	- 3 -
获取环境中的环境变量	- 3 -
C++中类的相互调引	- 4 -
C++中求绝对值	- 4 -
二进制文件的读取	- 4 -
以追加的方式打开文件	- 4 -
new 方式建立的对象	- 4 -
二维数组作为函数参数	- 5 -
C++中的函数重载给数据分析带来莫大的好处	- 5 -
如何判断一个文件是否正常打开	- 5 -
判断文件是否到达末尾	- 5 -
使 Linux 系统暂停 5 秒钟	- 5 -
C++中，对主程序的输入参数的解释	- 5 -
将 string，char*类型转化为 int，double 类型	- 6 -
容器 map 的使用	- 6 -

Makefile 相关.....- 7 -

总述.....	- 7 -
注释，变量，条件语句，特殊字符，函数等.....	- 8 -
调用 ROOT 中库的 makefile 的写法.....	- 10 -
如何在 ROOT 环境中加入自己写的库.....	- 11 -
编译器链接外部库时顺序.....	- 12 -
G++编译器常用参数.....	- 13 -
Geant4 中使用 root 的库时，makefile 做如何修改.....	- 13 -

ROOT 方面.....- 14 -

Root 的简单介绍.....- 14 -

TTree 相关.....- 16 -

Tree 中的 Branch 可以使用 Draw()函数.....	- 16 -
TTree 的结构可用 Show();Print();Scan();.....	- 16 -
TTree 的使用.....	- 16 -
TChain 的使用.....	- 17 -
将 paw 下常用的 hbook 格式转换成 root 格式.....	- 17 -

数学（TMath）相关.....- 18 -

物理常数（ $e, \hbar, \pi \dots$ ）.....	- 18 -
root 中常用的随机数产生.....	- 18 -
根据直方图 TH1,TH2 得到随机数.....	- 18 -
函数参数的指定与读取.....	- 19 -
Root 里定义 TF1 的方法有很多.....	- 19 -
将两个或多个 TF1 函数合成另外一个函数（？）.....	- 20 -
画出来的函数不连续怎么办.....	- 20 -
函数的积分.....	- 20 -
TVector3 的一些简单实用.....	- 21 -

直方图（TH1，TH2）相关.....- 22 -

清空直方图.....	- 22 -
直方图的投影（ProjectionX，ProjectionY）.....	- 22 -
得到二维直方图的平均线（ProfileX,ProfileY）.....	- 22 -

设置 Histogram 的对数坐标	- 23 -
如何设置直方图的轴的性质	- 23 -
自动调整 Histogram 的可视范围	- 23 -
调整直方图的可视范围	- 23 -
获得 Histogram 的定义范围	- 23 -
Histogram 的相关操作	- 24 -
TH1D, TH2D 等类存在成员函数	- 24 -
TH1D 的 bin 的 content 可正可负	- 24 -
直方图的复制	- 25 -
直方图的加减操作	- 25 -
求 Histogram 的 FWHM(Full Width at Half Maximum)	- 25 -
对于 TH2D, 根据 X 的值找到 BINX:	- 25 -
删除直方图的统计框 (统计了 RMS, Average 等)	- 25 -
直方图的自动寻峰(TSpectrum)	- 26 -
使用 Root 的函数拟合直方图	- 26 -
TGraph 相关.....	- 27 -
TCutG 是一个做 Cut 的类	- 27 -
TGraph,TGraphErrors,TGraphAsymmErrors 相关	- 27 -
拟合完直方图或是 TGraph 之后, 希望把拟合的结果也显示出来:	- 27 -
TCanvas 和 TPad 相关.....	- 28 -
TCanvas 和 TPad 常规用法	- 28 -
TCanvas 中鼠标的操作	- 28 -
TCanvas 相关的一些函数	- 28 -
TCanvas 中调整 TPad 的大小	- 29 -
把纵坐标变为对数	- 29 -
TAxis 相关	- 30 -
改变每个小格和大格的数目	- 30 -
坐标 label 使用指数表示	- 30 -
TGraph, TH1D 在使用 SetRange()不好用,	- 30 -
于 TH1D, TGraph, 可以改变 bin 的 label。	- 30 -
TFile 相关.....	- 31 -
Root 对象的存储	- 31 -

删除 root 文件中已有的直方图	- 31 -
关于 ROOT file 的使用说明一	- 31 -
其他相关	- 33 -
TLegend 的使用	- 33 -
Root 中关于时间的读取和记录	- 33 -
Event Display 时，免不了画个小球什么的代表 Particle Hit.....	- 33 -
画 3-D line	- 34 -
TLatex 的位置	- 34 -
C++中，this 是一个关键字，值便是本对象的指针	- 35 -
在 ROOT 环境下就要遵守 ROOT 的规则。	- 35 -
实时显示处理到第几个事件.....	- 35 -
调用 Root 库画图，出来的图总是一闪而过	- 35 -
TTimer 让一个程序定期去做某件事情	- 35 -
Root 中经常遇到生成大量的谱图，希望把它们归整在一个文件中.....	- 36 -
root 进入 cint 环境后，执行带参数的 script	- 36 -
Root 中有很多全局的环境变量	- 37 -
Geant4 相关	- 38 -
Geant4 的一点认识.....	- 38 -
几何相关	- 41 -
定义 Tube, Box, 以及 logical, physical.....	- 41 -
定义材料	- 42 -
Geant4 中已经内置了很多常用的材料.....	- 42 -
定义一种分子.....	- 43 -
定义 Helium3（该方法定义的 ^3He 能和中子反应）	- 43 -
由多种材料制作混合材料.....	- 43 -
G4Material 同时包括如下函数，可用于调用：	- 44 -
粒子的发射.....	- 45 -
定义粒子	- 45 -
指定方向，初始位置，能量.....	- 46 -
指定多个粒子.....	- 46 -
初始化方向的一些函数.....	- 46 -

物理过程相关.....	- 48 -
G4 跟踪粒子的策略	- 48 -
如何定义 reference physics List	- 49 -
记录模拟信息.....	- 50 -
从 G4Step 中提取 step 的信息	- 50 -
通过 step 获取运动着的粒子的信息	- 50 -
通过 step 获取几何相关的信息	- 51 -
记录模拟信息的方法.....	- 52 -
终止次级粒子.....	- 55 -
其他相关:	- 57 -
G4ThreeVector 的一些常用函数	- 57 -
G4BestUnit 显示合适的单位	- 57 -
Geant4 中常用的粒子的名字	- 57 -
Some Tips in Geant4	- 57 -
附录.....	- 58 -
Root 的安装过程	- 58 -
Geant4 的安装过程.....	- 58 -
Color in ROOT	- 60 -
Area Fill Style.....	- 61 -
MarkerStyle.....	- 61 -
GreekLetter and Math type.....	- 62 -
Latex grammar	- 63 -
Mouse operation.....	- 65 -
生成 root 库的 makefile	- 66 -
沿着 \vec{k} 方向得到张角为 θ 的随机矢量	71
Think. Create. Share.....	72

Linux 方面

Linux 下 NTFS 格式的硬盘挂载

有一些版本的 Linux 并不能自动挂载 NTFS 格式的硬盘，这时，你必须手动挂载：.

- (1) Ntfs-3g is asked;
- (2) `mount -t ntfs-3g /dev/sdb /mnt/ntfs`
- (3) `umount /mnt/ntfs`

Linux 下 shell 脚本的一些注意事项

- 1, 在 Linux 的 shell 脚本里，`$(shell command)`可以直接调用 shell 里的命令；
- 2, Makefile 里可以直接调用 shell 环境下的环境变量；

合并 eps 或是 pdf 格式的文件

```
for pdf
pdftk a.pdf b.pdf cat output out.pdf
a.pdf b.pdf, these two files can be merged into out.pdf
for ps
gs -SDEVICE = pswrite -s OutputFile = out.ps in1.eps in2.eps...
```

文件的压缩与解压缩

```
解压缩: tar zxvf fileName.tar.tgz
压缩: tar zcvf filename.tar.tgz filename
```

搜寻某个文件或文件夹

```
find . -name *.root //在当前文件夹内寻找以.root 结尾的文件
find .//*/* -name *.root //在当前文件夹的任意二层文件夹中，寻找以.root 结尾的文件
```

寻找文件夹中包含某个字符串的文件

`grep "TCanvas" * -n` // 在当前文件夹下寻找含有字符串“TCanvas”的文件，并显示行号

关于 64 位 Linux 链接 Geant4 库失败

这里不确认下面的方法是唯一，或是真正的解决方法，但是它确实将库链接上了，先临时记在下面：

- 1, 首先确认系统的版本：`dpkg --print-architecture`
- 2, 确认打开多构架支持功能：`dpkg --print-foreign-architectures`
- 3, 如果步骤 2 报错，说明你没有打开多构架支持功能，按照如下方法打开：

```
sudo dpkg --add-architecture i386
```

```
sudo apt -get update
```

```
sudo apt-get dist-upgrade
```

- 4, 再保证 G++, Gcc 对 32 位库的识别：

```
apt-get install gcc-multilib g++-multilib
```

注 1: Ubuntu 安装后，也会出现编译错误，具体不知道什么原因，但也整理在下面了：

```
#For the libraries of Ubuntu are not in the standard place
```

```
export LIBRARY_PATH=/usr/lib/$(gcc -print-multiarch)
```

```
export C_INCLUDE_PATH=/usr/include/$(gcc -print-multiarch)
```

```
export CPLUS_INCLUDE_PATH=/usr/include/$(gcc -print-multiarch)
```

把上面的这四行 copy 到 .bashrc 文件中（第一行是注释）。

C/C++ 方面

自己对 C++ 的认识是：它将数据与函数封装在一起，使得程序在逻辑上被分为了多个功能实体；这种语言上的封装是因为，工作中的数据与操作存在对应关系，没有必要把数据公布于众，造成没有必要的混乱。封装导致了两个好处：任务中没有严重关联的部分可以并行；继承的概念允许出现。

格式化输出与输入

1,

```
#include "fstream"
#include "iomanip"

... ..
ofstream f1("... ..");
f1<<setw(5)<<setfill(" ")<<"recorded message";
```

2,

使用 `sprintf(char_tem, "good/%3d/cluster.dat", 1)`；则 `char_tem` 中将被赋值 `"good/001/cluster.dat"`；

并且，`"%.3f"` 将保留小数点后的 3 位有效数字；

C++ 中类的静态成员，以及静态函数

C++ 类的静态成员就像是有了“家”的全局变量，一个程序中该类的多个对象使用同一个全局变量。静态成员不需要对象的建立，就存在在内存中，并且多个对象共享相同的静态成员，为了调用静态成员，C++ 中引入了静态函数的概念。

```
A.h
class A
{
    ... ..
    static int p1;
    static void Func1();
};
```

```
A.cpp
#include "A.h"
int A::p1 = 1;

A::A(){}
void A::Func1(){ ;}
```

获取环境中的环境变量

```
#include "stdlib.h"
```

```
char* getenv(char* parameterName);
```

C++中类的相互调引

因为如果直接相互引用，那么编译器在编译程序的时候，就会出现死循环。

```
//class A.h
#include "B.h"
class A
{
    B b;
};
```

```
//class B.h
class A; //声明
class B
{
    A *a;
};
但在 B.cpp 中需要
#include "A.h"
```

C++中求绝对值

```
#include "math.h"
int abs(int a);
float fabs(float a);
```

二进制文件的读取

```
#include "ifstream"
ifstream fin("fileName");
int Num;
fin.read((char*) (&Num), sizeof(Num)); //像 Num 中读取 32 个 bit 的数据
```

以追加的方式打开文件

```
#include "ofstream"
ofstream outf("fileName", ios::app);
```

new 方式建立的对象

new 方式建立的对象存放在内存的动态内存区，若想删除，需要使用 delete() 函数；在 root 中，需要使用 Delete()。

注意，使用 delete()，并没有将相应的指针清零，故为避免以后的问题，一般在 delete() 之后，都将相应的指针清零；

二维数组作为函数参数

参数是二维数据需要指明第二维的维度。

```
int array[10][5];
```

函数声明: `void func(int a[][5]);`

函数调用: `func(array);` 在函数中, 使用 `array[i][j]` 的方式访问数组;

注: 对于多维数组也是一个道理: 需要指明除第一维以外的所用维度。

C++中的函数重载给数据分析带来莫大的好处

如何判断一个文件是否正常打开


```
ifstream infile( "fileName" );  
if(!infile.good()) { cout<<" Not Existed!" <<endl; }
```

判断文件是否到达末尾

有时文件的最后字符是个空格, 虽然该空格可以被流操作符>>忽略, 但是并不意味着到了文件的结尾, 所以使用

```
while(stream name)  
{  
... ..  
}会多操作一次。
```

```
fstream.ignore(999, '\n');  
char first; fstream.get(first);  
fstream.putback(first);
```



因为 `while(stream name)` 实际上是流返回了一个状态, 该流是否正常, 正常情况下当然正常, 故保险的做法是在 `while` 循环的最后添加:

使 Linux 系统暂停 5 秒钟

```
#include "unistd.h"  
sleep(5); //睡 5 秒
```

C++中, 对主程序的输入参数的解释

```
void main( int argc, char* argv[] )  
编译出来的程序需要参数的输入。
```

`int argc` 记录有多少个参数，若不加参数，`argc=1`，（命令本身也算一个参数）。然后，一个“-”代表一个字符数组的开始。`argv`是指针的指针。

比如：`./hello -r 9 6 -h good -g yes 3`

这行命令中，`argc=4`，并且：

```
argv[1][1]=r; argv[1][2]=9; argv[1][3]=6;
```

```
argv[2][1]=h; argv[2][2]=good;
```

```
argv[3][1]=g; argv[3][2]=yes; argv[3][3]=3;
```

将 `string`，`char*` 类型转化为 `int`，`double` 类型

```
#include "stdlib.h"
```

```
double d = atof(s.c_str());
```

```
int d = atoi(s.c_str()); //如果这个字符串是个浮点型，则只返回整数部分
```

容器 `map` 的使用

数组是存储有序数据的一种好手段，通过 0, 1, 2 等整数编号就可以调用这些有规律的数据，但是有些数据，他们之间是散乱的关联在一起的，不可能在逻辑上将他们联在一起。这时，容器 `map` 是一个好工具，它的索引是在你存储时来指定的，可以是 `int`，`char`，`string`。这样，在从 `map` 中调取数据时，只需提供这个索引，就可以得到值了。

我在 Geant4 的模板类中使用的 `Config` 类来存储程序的参数，就是得益于 `map` 的强大功能。关于它的使用，可直接参考这个类。另外，现在发现自己在数据处理过程中，费劲心思新的 `FissionMap` 类，实际上就是实现 `map` 的功能。

Makefile 相关

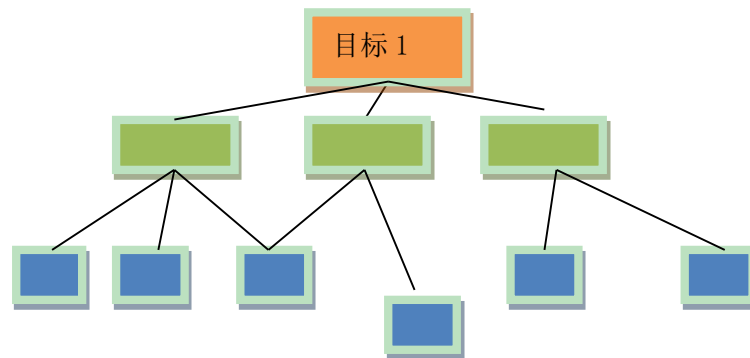
总述

在 linux 下编程，Makefile 是一个必要的工具。市面上也有很多它的说明文档，里面讲的都非常的详细，可作为教程。这里仅是我的学习总结。

首先，说明一下程序的编译过程。在 Linux 下，常用的编译器有 GCC，G++。其中，GCC 编译 c 语言，G++ 编译 c++。编译过程首先是编译器将源代码编译为目标文件(机器语言)，然后再使用 Linker 将各个目标文件链接，最后在链接好的文件前面加上操作系统启动语句。

由于源文件中存在依赖关系（如：定义在文件 File_A 中的函数 A 依赖定义在文件 File_B 中的函数 B；类之间的包含关系等等），Makefile 就是描述这些依赖关系的文件。他告诉 make 命令：要编译什么；编译的东西依赖什么；如何编译。所以，Makefile 里面逻辑上是一棵棵的“依赖树”，一个目标文件是一棵依赖树，依赖关系可以使用树形图来描述。

如下图：



注：

(1)，make 会把它看到的第一个目标作为该文件的**终极目标**，生成它，该 make 命令便结束了。

所以，如果有多个目标时，便定义一个虚主目标，如：
`all:object1 object2`
这样，生成 object1, object2 后，make 命令便会结束。

(2)，实际上，make 在给 makefile 建立依赖树时，会检查依赖树中，枝叶是否比根新，如果是，说明有必要更新根，否则 make 将什么都不干。

以上这两项是 make 的主要内容，为了完成这两个目标，make 命令本身引入了很多附加功能，如：下面会介绍的：变量，函数，条件控制等。

Makefile 相关

(3), 为了使编译代码及维护代码变得简单, make 还提供了其他功能。比如说, 在 makefile 中可以加入如下:

clean:

____ Tab @echo "clean..."

Tab ____ @rm *.so *.d ...

这样, make clean 就会直接执行 clean 后面的命令。注意, makefile 中, 命令要以 Tab 键开头。如果在命令的前面加@表示, 在执行该命令时, 不再显示该命令本身, 否则, 在执行该命令前, 会显示该条命令。

以上便是 makefile 的主要骨架。下面介绍它的辅助内容, 利用这些东西, 你的 makefile 会简洁明了。

注释, 变量, 条件语句, 特殊字符, 函数等

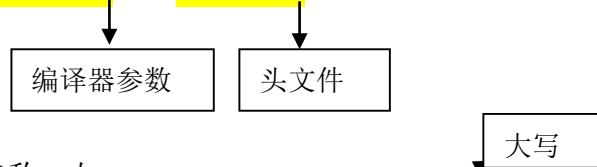
1, 注释

makefile 一般使用 “#” 作为注释符, 使用 “\” 作为下一行的继续符;

2, 变量

在 makefile 中, 虽然变量的名字是任你取的, 但大家会有不成文的习惯, 你最好采用这些, 方便很多。如:

CXXFLAGS += \$(DEFINES) \$(INCLUDES)

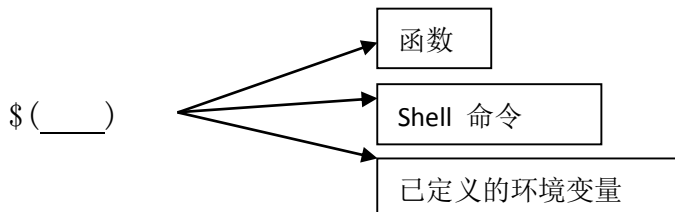


LD 指链接器的名称, 如: g++;

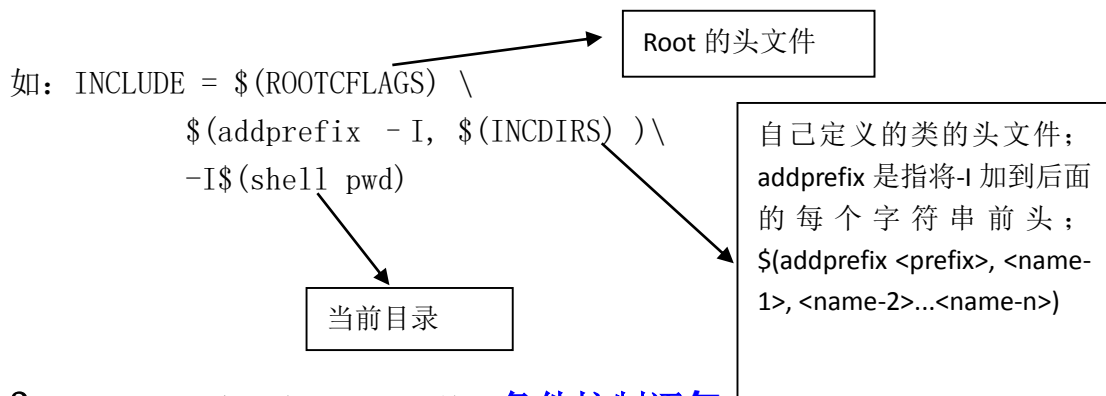
LDLFLAGS: 涉及到上面 LD 的参数, 如: -O3;

SOFLAGS: 涉及到生成动态库, 如: -shared;

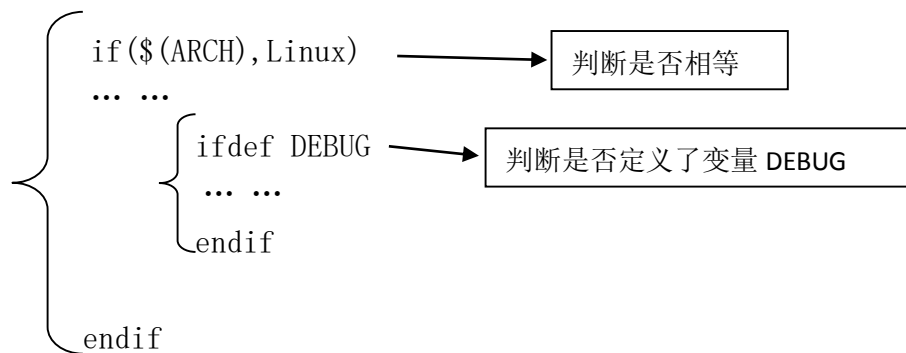
调用这些变量时, 使用\$(变量名); \$符号在 makefile 中是比较霸道的, 无需说明, 在字符串中可直接使用, 如: “\$(A) ...”; 并且, \$符号还有很多用途,



Makefile 相关



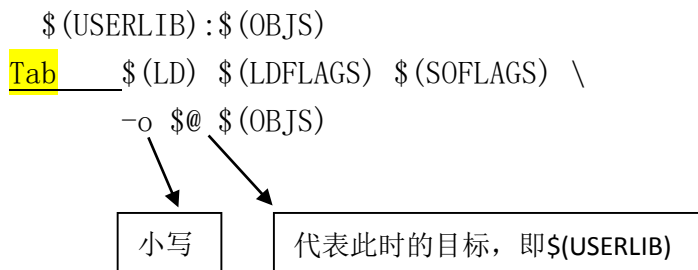
3, makefile 中也允许一些简单的**条件控制语句**，如：



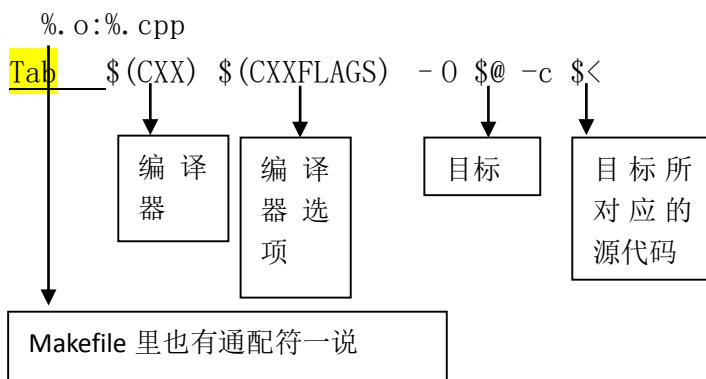
4, 特殊字符

Makefile 中可以使用一些特殊字符，使编译过程更加简洁。

(1)



(2) make 的核心工作是编译，链接。编译的时候，当然不希望写出每个源文件的依赖关系。



Makefile 相关

使用这种目标依赖关系，就是告诉编译器，如何生成以“.o”结尾的目标文件。它等于是告诉 make 一个生成.o 文件的模式。

5, 函数

(1),

```
SrcSuf:=cxx
```

```
SRC = particle.cxx source.cxx react.cxx
```

```
OBJ = $(SRC:. $(SrcSuf)=.o)
```

```
HDR = $(SRC:. $(SrcSuf)=.h)
```

```
DEP = $(SRC:. $(SrcSuf)=.d)
```

如此使用，将会替换 SRC 中以. \$(SrcSuf) 结尾的文件依次替换为以.o, .h, .d 结尾的文件。

(2), addprefix

在前面已经说过了。

(3), patsubst

```
OBJ = $(patsubst $(SrcDir)/%. $(SrcSuf), \
$(objDir)/%. $(ObjSuf), $(SRC))
```

将\$(SRC)中模式 1 转化为模式 2

是一个符合__模式的字符串集

(4), wildcard

```
HDR = $(wildcard $(IncDir)/*. $(IncSuf))
```

将符合该模式的文件列出来，并整体赋值给 HDR。

掌握了上面所提到的内容，你便可以读懂很多 Makefile，并且写一些简单实用的 makefile 了。一般，由于程序的存放结构式相同的（包括 include, src, obj, tem...），写好的 makefile 是通用的，只需更改很少的地方（如，源文件的后缀等），不用每次都重新写新的 makefile。

调用 ROOT 中库的 makefile 的写法

Root 中定义了大量的好用的工具，如：TH1, TH2, TGraph, TTree, TFile, TLegend, TLatex, TLine, TCanvas, TF1, TF2, TRandom 等等。很多时候，我们是希望在脱离了 CINT 环境下，继续使用这些工具的，那么就涉及到在 Makefile 中，说明这些类的头文件位置，库文件名称等。这些，Rooter 已经为我们写好的，只需像下面一样调用即可。

```
ROOTCFLAGS := $(shell root-config --cflags)
ROOTLIBS    := $(shell root-config --libs)
ROOTGLIBS   := $(shell root-config --glibs)
```


如何在 ROOT 环境中加入自己写的库

在自己的程序中使用，ROOT 中的类时，如果涉及到画图，则是件比较难办的事（还没搞清）。而且，使用 Root 的编译环境 CINT 是比较方便的。所以，有必要将自己写的类加到 CINT 环境中。

达到此目的，分如下步骤：

1, 类的定义

(1), 头文件中, #include "TObject.h"

```
ClassName:public TObject
{
... ..
```

public:

```
ClassDef (ClassName, 0)
```

```
};
```

(2), 类的实现

```
ClassImp (ClassName);,
```

2, LinkDef 的定义

定义文件 LinkDef.h, 内容是:

```
#ifdef __CINT__

#pragma link off all globals;
#pragma link off all classes;
#pragma link off all functions;

#pragma link C++ class ClassName+;

#endif
```

该文件说明欲生成的库文件中，包含哪些类。

3, makefile 的书写

makefile 中不同之处在于该文件中要加入:

```
$(SrcDir)/$(USERDICT).$(SrcSuf): $(HDR) $(LINKDEF)
    @echo "Generating dictionary $(USERDICT)..."
    $(ROOTSYS)/bin/rootcint -f $@ -c $(INCLUDES) $(DEFINES) $^
```

这行将生成一个. \$(SrcSuf) 在 src 文件夹下，剩下的和普通的 makefile 是一样的。该文件实际上是一个索引。

建立依赖关系:

```
$(ObjDir)/%. $(ObjSuf): $(SrcDir)/%. $(SrcSuf)
    $(CXX) $(CXXFLAGS) -o $@ -c $<
```

Makefile 相关

总目标是：

```
all:      $(USERLIB)

$(USERLIB):  $(OBSJ)
            $(LD) $(LDFLAGS) $(SOFLAGS) -o $@ $(OBSJ)
            @echo "$@ done"
```

关于这方面的 makefile，可以参考该文件的附录一；

4，执行 make 命令后，将会在 src 文件夹下生成相关的*Dict.cpp *Dict.hh 两个文件；

5，编译完的库直接 Load 是通不过的，你得告诉 cint 这个库依赖 root 里的那些库，使用命令：

```
symbols=`nm -C mylib.so | grep -E ' U T[:alnum:~]+:::' | sed 's,^.* U
\(:[:alnum:~]+\):.*$, (\1),' | sort | uniq | tr '\n' '|'`; (for so in
$ROOTSYS/lib/*.so; do nm -C -D --defined-only $so | grep -E 'T
('$symbols'):::' > /dev/null && echo $so; done ) | sort | uniq
```

这个命令后，你就会看到该库包依赖的库，然后，你直接 Load 这些库就可以了。

6，上面是一种工作方法，更加通用的方法是：利用上面的命令看依赖的库，然后利用 rlibmap 生成一个库的依赖列表，再把 LD_LIBRARY_PATH 包括该目录，这样直接打开 root，就可以直接用这个库了。

其中：

rlibmap 的用法如下：

```
rlibmap -f -o libName.rootmap -l libName.so -d dummy -c
libName_LinkDef.h
```

生成的依赖文件保存在 libName.rootmap 中；生成的库是 libName.so；libName_LinkDef.h 是那个生成库时依赖的文件；我们把上面生成的那些依赖库的名字代替 dummy 就可以了。

编译器链接外部库时顺序

在调用外部链接时，gcc 对库的调用顺序是有讲究的，否则会说，那个函数没定义等等。如，Hplus 中，开始就出现问题，解决办法如下：

Makefile 相关

```
Extralibs+= \  
-L$(extralib_path) \  
-lftd -lio -ltpc -ltof\\ \  
-lhplus -lphys
```

```
Extralibs+=\  
-Xlinker “-(” \  
$(HplusLibDir)/libftd.a \  
...  
...  
$(HplusLibDir)/libhplus.a\  
-Xlinker “-)” -lrt
```

这样将忽略这些库的先后顺序。

注：绝对路径；名称应写全

G++编译器常用参数

g++

-O -O1 -O2 编译的优化，后面的数字（0,1,2）越大，编译越优化；

-o 生成的目标

-I 头文件目录

-L 库文件目录

-l 库的名字（库文件一般命名为：libmath.a，使用时，不需要lib，.a。直接写-lmath就可以了）

Geant4 中使用 root 的库时，makefile 做如何修改

在已有的 makefile 中添加如下三行即可：

```
INCFLAGS += $(shell root-config --cflags)
```

```
LDLIBS += $(shell root-config --libs)
```

```
LDLIBS += $(shell root-config --glibs)
```

ROOT 方面

Root 的简单介绍

Root 是由 CERN 开发的数据分析工具，其针对核物理与粒子物理的数据处理特点，包括了数据处理时所用到的大量功能，掌握这个软件会给处理这个行当的数据带来很大方便。

Root 的学习过程是简单的。因为它就是你用到的直观的工具，只要你明白自己的需求，你就会发现它的使用是那样的直接。但是，它的很多功能在通过代码调用时才会很方便，甚至有些功能只有通过代码来调用，这就要求如果想要顺利的使用 Root，会点 C++ 的语法知识是最好不过的。

个人认为，对于做核物理实验的人，数据处理过程主要使用如下几个功能：数据存储为 TTree 的结构；从 TTree 的结构中读取数据；掌握 TFile 的创建，以及 Root 对象的存储与读取；TH1D, TH2D, TGraph, TF1, TVector3 这几个类的使用；会使用 TCanvas 画图，再学会调用类的函数来把画出来的图美化一下。只要能使用这些基本功能，个人觉得你已经可以使用 Root 做很多事情了。

对于一般的简单数据分析：画画图，看看能谱，拟合数据等，在会使用简单脚本后就足够了。但当你的任务比较复杂，或者说是需要多功能的时候，仅仅使用 Root 脚本会把让你手忙脚乱。这时，我采用方法有两个。一是在程序仅是针对一件事，并且仅仅是自己一个人来完成，那么我会把这件事分成几个功能上独立的几个块-类；再写一个 Root 脚本来调用这几个功能模块。这个方法有两个好处：功能模块之间的逻辑关系表现在脚本中；脚本中的类在运行前是编译为二进制代码的，它不像脚本中的每行语句一样，需要 Cint 每次执行都需要编译，所以运行效率与脚本比起来会高。二是如果程序中有一个功能是几个人分写的模块都需要的，比如数据格式，存储数据的函数，或是一些你写好的控制画图格式的类，或是你写好的一些常用数学函数等等，这时一个好用的办法是：基于 Root 的编译环境，将这些公用功能编译为 Root 可识别的库，即扩展 Root 的库，然后使用 Root 脚本或是其它库中就可以像调用 Root 中自有的那些功能类一样来执行。这里暂且称这两种方法为：编译类+脚本；扩展库+脚本。分别介绍如下。

编译类+脚本

这种方法是把写好的类在 Cint 中使用 .L 的方法 load 进环境中，然后再使用脚本来调用编译好的类。为了避免每次都手动的 load 类，就把待 load 的类写在 rootlogon.C 文件中，该文件默认情况是在 root 的 cint 启动时，自动执行的，这样就免去每次执行的麻烦。其中 rootlogon.C 的格式为：

```
{  
    gSystem->Load( " library name " );  
    gRoot->ProcessLine( "Root 内可被执行的命令，如：.L *.cpp;" );  
    gRoot->ProcessLine( "!shell cmd，如：rm echo···;" );  
}
```

实际上，有时即使是要使用扩展库的办法，我也是先使用这种方法来写类，并调试通过，这比每次都 make 来的快和方便，待类都写好后，再填到库中去。

扩展库+脚本

这种方法是自己在做比较复杂的事情时采用的，这种方法中，将类的头文件放在 include 中，把源文件放在 src 中，编译好的库就放在当前目录下，整个程序看起来是规整的，而且也方便管理，尤其是有很多的类时。关于这种方法的实现和 makefile 的书写，参考本文档中关于“**makefile 相关**”章中的“如何在 ROOT 环境中加入自己写的库”一节。

注意：

- 1， 在 root 中出现想要查找某个类的函数的用法，或是想要查一下某个类是否具有某个功能，最好查 root 的主页>Document>Reference Guide，它的查找功能很强大，至少对于我，基本都能解决我的问题（比一页页查说明书来的快）；
- 2， 软件的学习尽管需要自己的思考，但是，如果你已经发现某个问题自己一点头绪都没有，而且连解决这个问题的思路也没有时，我的建议是问问有经验的人，“软件技术的问题通常就是一层窗户纸”；

TTree 相关

Tree 中的 Branch 可以使用 Draw()函数

- (1) Draw("V1"); //把 V1 画出来;
- (2) Draw("V7+V8"); //把 V7+V8 画出来;
- (3) Draw("(V8-V7):(V10-V9)"); //可以画出关联谱;
- (4) Draw("(V8-V7):(V10-V9)", "V8!=0"); //还可以画满足 $V8 \neq 0$ 的关联谱;

注：待处理的电子学路数较多时（如 TPC 的成百上千的路数），可以把每一路看做一个 Event，这样使用 TTree 来存储多路电子学的结果时，便于分析。

TTree 的结构可用 Show();Print();Scan();

```
t1->Show(0) 显示第一个 Event 的内容;  
t1->Print()  把 tree 的格式打印出来;  
t1->Scan( "V2:V10:V9" ) 将列出这三个 Branches;
```

TTree 的使用

把数据存储成 TTree 的格式是方便的，尤其是对于 Event by Event 类型的数据：

(1) 存储

```
TFile* f1 = new TFile( "file_Name.root", "recreate" );  
f1->cd(); //将该 root 文件打开，就像 Linux 下的 cd 命令一样  
TTree* t1 = new TTree( "t1", "calibrated Event" ); //t1:tree 的名字，  
"calibrated Event" 用于对该 tree 的描述;
```

定义变长数组：变长数组的使用是非常有效的，因为核物理中，产物是不确定的。

```
int Num_ppac; double ppac_Index[100];  
t1->Branch( "Num_ppac", &Num_ppac, "Num_ppac/I" );  
t1->Branch( "ppac_Index", ppac_Index, "ppac_Index[Num_ppac]/D" );  
{  
    给 Num_ppac, ppac_Index 赋值; t1->Fill();  
}  
t1->Write(); //把内存中的缓存也写在内存中;
```

(2) 读取

```
TFile* f1 = new TFile( "file_Name.root" );
TTree* t1 = (TTree*) f1->Get( "t1" );
... ..
int Num_CsI;
t1->SetBranchAddresses( "Num_CsI", &Num_CsI );
... ..
int EvtNum = t1->GetEntries();
for(int i=0; i<EvtNum; i++)
{
    t1->GetEntry(i);
    ... ..//对 Num_CsI 的调用;
}
```

TChain 的使用

```
#include "TChain.h"
TChain* t1 = new TChain( "treeName" );
t1->Add( "data.root" );
t1->SetBranchAddresses( "Num_CsI", &Num_CsI );
int EvtNum = t1->GetEntries();
for(int i=0; i<EvtNum; i++)
{
    ... ..
    t1->GetEvent(i);
    ... ..
}
```

TChain 用于读取具有相同格式的 TTree。因为核物理中数据是以一个 run 一个 run 的方式来存储的，故 TChain 是这种情况下的利器。

将 paw 下常用的 hbook 格式转换成 root 格式

有压缩之说：

```
h2root xx.rzd xx.root 1 0 16384 0
```

没有压缩之说：

```
h2root xx.rzd 将会自动生成 xx.root 文件
```

数学 (TMath) 相关

物理常数 ($e, \hbar, \pi \dots$)

在代码的过程中，经常涉及到物理学中的常数，使用 TMath 中自带的函数不但满足精度的要求，并且引用方便。

```
#include "TMath.h"
using namespace TMath;
Pi()=3.1415926...;    H()=6.626...e-34;    E()=2.718.....;
Power(double x,double y)=x^y;
RadToDeg() = 180.0/Pi();
DegToRad() = Pi()/180.0;
Sqrt2() = 1.414...;
```

root 中常用的随机数产生

```
#include "TRandom.h"
gRandom->Binomial(ntot,p) 二项式分布;
gRandom->BreitWigner(mean,gamma) Breit-Wigner 分布;
gRandom->Exp(tau) 指数分布;
gRandom->Gaus(mean,sigma) 高斯分布;
gRandom->Integer(imax) 产生 (0—imax-1) 之间的整数分布;
gRandom->Landau(mean,sigma) landau 分布;
gRandom->Poisson(mean) Poisson 分布;
gRandom->PoissonD(mean) 返回 double 的 Poisson 分布;
gRandom->Rndm() 产生 (0,1] 之间的均匀分布;
gRandom->Uniform(x1,x2) 产生 (x1,x2] 之间的均匀分布;
```

根据直方图 TH1,TH2 得到随机数

```
double x = TH1D::GetRandom();
TH2D::GetRandom2(double &x, double &y);
TH3D::GetRandom3(double &x, double &y, double &z);
```

其获得随机数的思路是这样的：

首先根据各个 bin 之间的相对高度找到下一个数应该从哪个 bin 里出，然后再在这个 bin 的范围内去 uniform Random Number。

函数参数的指定与读取

一维函数最简单的定义方法: `TF1("functionName", "Function formate", double x1, double x2);`

在这种定义中, 函数的形式可以使用 Root 中识别的函数: 如 $\sin(x)$ 等。另外, 默认的变量名是 "x"; 函数中可引入参数, 使用 [0], [1], [2] 标识 (注意: 这里的参数的索引号是从 0 开始的)。

对于参数的设置, 使用如下成员函数:

```
->SetParLimits(0, -1, 1); //设置第一个参数的范围是[-1, 1]
->SetParameter(4, 10);    //指定第 5 个参数是 10;
->SetParameters(1, 2, 3); //指定前三个参数分别为: 1, 2, 3;
->FixParameter(4, 0);      //固定第五个参数为 0, 这样在使用这个函数拟合
                           //曲线时, 该参数就固定在 0;
```

拟合后的函数, 可以通过如下方法得到拟合的结果:

```
double FitResults[8];
->GetParameters(&FitResults[0]); //使用这种方法, 获得拟合后的结果;
->GetParameter(int iPara); //返回第 iPara+1 个参数
```

Root 里定义 TF1 的方法有很多

(1) 原始定义

`TF1* f1 = new TF1("f1", "gaus+(x>3 && x<90)*pol2(3)", 0, 180);` 其中, 逻辑表达式将生成一个 step function; gauss 有三个参数, pol2 的参数将从 [3] 开始;

(2) 自己定义函数的方法:

```
double CrossSection(double* x, double* paras)
{
    ... ..
    return ... ..;
}
```

`TF1* funcFit = new TF1("funcfit", CrossSection, 0, 100, 6);` //使用 CrossSection 函数, 自变量范围是 [0, 100], 并有 6 个参数;

注: 这种方法很实用, 比如自己做多源模型能谱拟合时, 就是使用这种方法。

(3) 使用类 classA 的成员函数 func1 去定义 TF1:

```
func_cn_back_fit =
    new TF1("func_cn_back_fit", this, &classA::func1, 0, 100, 3);
```

这种方法实际上用的还是 (2) 中的方法, 只不过这里使用类来管理多个函数。

将两个或多个 TF1 函数合成另外一个函数 (?)

```
TF1* f1; TF1* f2;
Double_t finter(double* x, double* par)
{
    return TMath::Abs(f1->EvalPar(x, par)+f2->EvalPar(x, par));
}
Double_t fint(double start, double end)
{
    TF1* fint = new TF1("fint", finter, start, end, 0);
    Double_t xint = fint->GetMinimumX();
    return xint;
}
Int fint3()
{
    f1 = new TF1("f1", "x^2+4", 0, 10);
    f2 = new TF1("f2", "x+3", 0, 10);
    cout<<" x_int" <<fint(0, 8)<<endl;
}
```

画出来的函数不连续怎么办

```
TF1::SetNpx(int npx=100);
```

改变画这个函数的点数, 从而得到一个分辨率比较好的图。[这样做后](#), [GetRandom\(\)](#) 的结果也会比较好?

函数的积分

```
double TF1::Integral(double x_min, double y_min);
```

将返回从 x_{\min} 到 x_{\max} 的函数面积。注意: 函数的积分与直方图的积分不同, 它不用讲究 bin 的大小, 直接得到积分的结果。

TVector3 的一些简单实用

```
TVector3::SetX(double x); SetY(double y); SetZ(double z);  
::SetXYZ(double x, double y, double z);  
::X(); Y(); Z();  
::Mag(); //得到该向量的模;  
::Angle(TVector V2); //得到与 V2 之间的夹角(0-pi);  
::Theta() // (0-pi);  
::Phi() // (-pi, pi);
```

同时，支持向量之间的加减乘除，数乘，点乘；在 Root 中，角度使用（一般）弧度制。

注意：尽管使用 TVector 可以方便的处理数据，但是如果有大量的计算，建议还是自己进行变化，这里的运行效率较低，尤其是涉及到大量的 TVector 的创建于删除。

直方图（TH1，TH2）相关

清空直方图

`TH1::Reset();` //清空直方图的 Bin 中的计数，但是其它设置保持。

直方图的投影（**ProjectionX**，**ProjectionY**）

```
TH1D* TH2::ProjectionX(Const char* Name=" ", Int_t first_bin, Int_t last_bin, Option_t* option=" ") const;
```

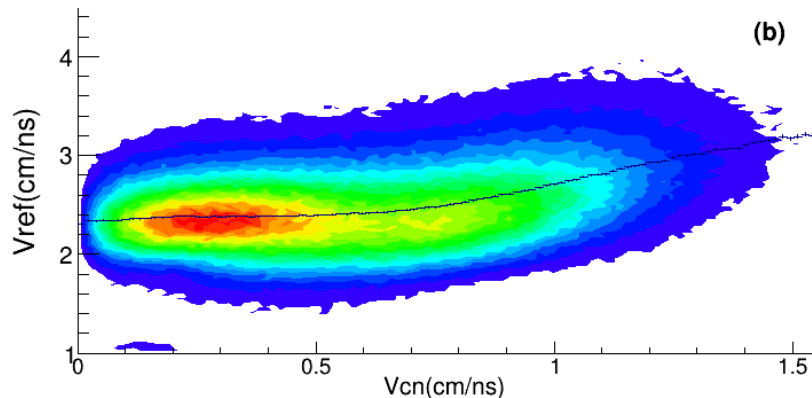
注意返回的对象是 TH1D; Name 是生成对象的名字; 返回对象的范围是从 first_bin 到 last_bin; 当然也存在函数 ProjectionY()。

得到二维直方图的平均线（**ProfileX**,**ProfileY**）

求取沿着 X 轴的在 Y 轴方向的平均值，并把他画在相同的 Canvas 内，其结果如下图所示所视：

```
{
    TProfile* h2_Vcn_Vref_profileX =
    h2_Vcn_Vref->ProfileX("AvgVref_Vcn", 0, 100); // [1]
    h2_Vcn_Vref_profileX->Draw("same");
}
```

[1] 如果没有这三个参数也是可以的，这时，Profile 后的名字会默认为二维直方图的名字+"_pfx"，默认的求平均的范围是：1, -1；其中-1 意味着一直到最后一个 bin。



自然，也存在函数 ProfileY()。

设置 Histogram 的对数坐标

对数坐标并不是 Histogram 的性质，而是所画的画布的性质：
其设置方法：

一张 TCanvas* c1 上
多个 pad: c1->cd(1)->SetLogy(1);
一个 pad: c1->SetLogy(1);

如何设置直方图的轴的性质

```
TH1::GetXaxis()->SetTitle( " ** " );  
TH1::GetYaxis()->SetTitle( " ** " ); 可以设置各轴的名字;  
也可以直接: TH1::SetTitle( " ** " ); SetYTitle( " ** " );  
SetZTitle( " ** " );  
GetXaxis()->CenterTitle();使 label 的 title 置于中间;  
GetXaxis()->SetLabelOffset(float d_default=1.0);  
GetXaxis()->SetTitleOffset(float d_default=1.0);
```

自动调整 Histogram 的可视范围

```
double x1 = FindFirstBinAbove(Double_t threshold = 0, Int_t axis = 1);  
//1-x;2-y;3-z  
double x2 = FindLastBinAbove(Double_t threshold = 0, Int_t axis = 1);  
TH1D::GetXaxis()->SetRange(x1, x2);
```

调整直方图的可视范围

有时使用 SetRange(int bin1, int bin2)时，不好用，可以试试：
Taxis::SetLimits(Double_t Xmin, Double_t Xmax);
Taxis::SetRangeUser(2, 6); 坐标是以画在 TCanvas 上的对象的单位为参考的。

获得 Histogram 的定义范围

```
(1) TH1D::GetXaxis()->GetBinUpEdge(10);  
      GetYaxis()->GetBinLowEdge(1);  
note:Root 中，histogram 的 bin 是从 1 开始的;
```

(2) 得到 Bin 的 Center 也可以通过这种方法：

```
TH1D::GetXaxis()->GetBinCenter(Int_t bin);
```

Histogram 的相关操作

```
int TH1D::GetMaximumBin() 返回值最大的 bin;  
double TH1D::GetBinCenter(int bin_Index); 返回 bin_Index 的中心； 注意  
Histogram 的 bin_Index 是从 1 开始计算的；  
double TH1D::GetMaximum(); 返回该 Histogram 的最大值；  
void TH1D::SetBins(20, 3, 5); 重新 Bins  
double GetBinContent(int binx) 返回 binx 中的值；  
int FindBin(double x, double y=0; double z=0) 返回 x 在 Histogram 中的 bin  
值；  
int GetNbinsX() 获得该直方图在 X 维度上有多少个 Bin；  
char* GetName(), 返回 Name 值；  
double GetBinWidth(int binIndex); 得到 bin 的宽度  
double GetMean(Int_t axis=1) const;  
double GetMeanError(Int_t axis=1) const;  
double GetRMS(Int_t axis=1) const;  
double GetRMSError(Int_t axis=1) const;
```

TH1D，TH2D 等类存在成员函数

SetBins(int nx, double x_min, double x_max, int by, double y_min, double y_max, int n_z, double z_min, double z_max); 来调节已经定义好的横纵坐标；

TH1D 的 bin 的 content 可正可负

可通过 TH1D::SetContent(4, -10); //这样，GetBinContent(4)时，将返回-10;

另外，可以 scale 一个直方图：

```
double scale = 5/h->Integral();  
h->Scale(scale);      <- Scale 实际上就是一个乘法操作；把直方图归 5  
化；
```

使用上述方法，对直方图是归一化。而使用：TH1D::SetNormFactor(1.0)是改变了直方图的全局变量，这样当你再 Fill() 其他值时，直方图会自动归一化。

直方图的复制

```
TH1D::Clone( "newName" );
```

将返回一个 TH1D*，指向一个在动态内存区的对象，该对象的 name 就是“newName”；其中，本操作将复制该对象中的所有内容，包括其中拟合出来的函数。

直方图的加减操作

```
Add(TH1D* h1, double f1); //其中 f1 可正可负
```

求 Histogram 的 FWHM(Full Width at Half Maximum)

并没有现成的函数，但可以使用下面这个简单的办法：

```
int bin1 = h1->FindFirstBinAbove(h1->GetMaximum()*0.5);
int bin2 = h1->FindLastBinAbove(h1->GetMaximum()*0.5);
double fwhm = h1->GetBinCenter(bin1) - h1->GetBinCenter(bin2);
(这仅仅是一个简单的方法)
```

对于 TH2D，根据 x 的值找到 BINX：

(1) int FindBin(double x, double y); //对于高维的 histogram，返回值是一个 global bin;

```
(2) int binx; int biny; int binz;
void TH2D::GetBinXYZ(FindBin(x0, y0), &binx, &biny, &binz);
这样就找到了。
```

(3) Double_t GetBinContent(int Binx, int Biny); 可以返回该 Bin 的 content;

删除直方图的统计框（统计了 RMS，Average 等）

```
gStyle->SetOptStat(kFALSE);
```

直方图的自动寻峰(TSpectrum)

使用 Histogram 记录能谱时，如果有寻峰的要求，可以考虑自己写一个简单的寻峰程序：找到最大值，然后往左右两边各找 5 个 bins，如果都是依次下降的，那么就认为这是一个峰。ROOT 里也提供了更专业的寻峰类：

TSpectrum，下面是例子：

```
TSpectrum* spec = new TSpectrum(MaxPeaks); //你认为这个谱里最多有多少个峰
```

```
double NoiseLevel = 0.1; //这个意味着如果峰值小于最高峰的 0.1，那么就忽略这个峰
```

```
int nfound_Peak = spec->Search(TH1D* h1_ESpec, Double_t sigma = 2, "", NoiseLevel); //返回它找到的峰，sigma=2 是默认值，它指你要找的峰的 Sigma 在多少附近
```

```
printf("Found %d candidate peaks\n", nfound_Peak);
```

```
//Estimate background using TSpectrum::Background
```

```
TH1 *hb = s->Background(TH1D* h1_ESpec, 20, "same"); //任何一个能谱，肯定有个本底，这里将找到 h1_ESpec 的本底，并将其赋给 hb;
```

```
Float_t* xpeaks = spec->GetPositionX(); //将得到的峰位写在一个数组内，并将该数组的指针赋给 xpeaks
```

使用 Root 的函数拟合直方图

```
TF1* f1 = new TF1("f1", "[0]*TMath::Poisson(x,[1])", 1, 30);
```

```
f1->SetParameters(1, 1);
```

```
h1_Count_Static->Fit("f1", "R");
```

这个粒子是定义了一个 Poisson 分布的函数，使用它拟合直方图。

TGraph 相关

TCutG 是一个做 Cut 的类

```
#include "TCutG.h"
```

```
TCutG* g1 = new TCutG( "g1", 4); //4 是指该 cut 的点数
```

```
g1->SetPoint(0, 0.1, 0.1);
```

```
g1->SetPoint(1, 1.1, 0.2);
```

```
g1->SetPoint(2, 1.1, 1.2);
```

```
g1->SetPoint(3, 0.1, 1.2);
```

```
g1->IsInside(0.5, 0.5); 判断(0.5, 0.5)是否包括在 g1 内。
```

注：(1) 这个类在数据处理过程中，是做窗的好工具（*可别自己写函数来判断这个点是否在自己的窗里了）；

(2) 这个类会把第一个点与最后一个点连接起来，你就不需要把最后一个点定义的和第一个点相同；

(3) TCutG::Draw() 就像 TGraph 一样可以直接画在 TCanvas 上，对 LineStyle, LineColor, LineWidth 的控制与 TGraph 相同，实际上，你看一下说明书，就可以看到，这个类是从 TGraph 继承来的。

TGraph, TGraphErrors, TGraphAsymmErrors 相关

(1) TGraph* gr1 = new TGraph(int Num, double* px, double* py); //定义没有误差棒的 graph

```
gr1->Draw( "AC*" ); //A:axis; C:continue line; *:marker style;
```

(2) TGraphErrors* gel = new TGraphErrors(int Num, double* px, double* py, double* px_error, double* py_error); //定义误差棒相等的 graph

(3) TGraphAsymmErrors* gr1 = new TGraphAsymmErrors (int pointNum , double* X , double* Y, double* X_Low_Error, double* X_Up_Error, double* Y_Low_Error, double* Y_Up_Error);

拟合完直方图或是 TGraph 之后，希望把拟合的结果也显示出来：

```
#include "TStyle.h"
```

```
gStyle->SetOptFit(kTrue);
```

TCanvas 和 TPad 相关

TCanvas 和 TPad 常规用法

TCanvas::Divide(2,2)生成 2×2 个 Pad，这些 pad 可以通过 cd(1), cd(2), cd(3), cd(4) 进入；且该函数将返回相应的 TPad 的指针，通过该指针可以进一步设置该 pad 的相关属性，如大小，颜色等等；当然，返回该指针还有另外一种方法：如果 TCanvas 的名字是 “c1”，那么 Divide 出来的 pad 的名字分别是：“c1_1”，“c1_2”，“c1_3”，“c1_4”；
 TPad* subPad_3 = (TPad*) (c1->GetPrimitive(“c1_3”));
 事实上，TCanvas 是一个窗口，TPad 是一个包含在 TCanvas 中的成员。其中，TCanvas 与图形化服务器打交道，TPad 转换可 Draw() 对象成可视图元素。故，没有 TCanvas，去建立 TPad 是不行的。Divide() 的过程就是创建 TPad 的过程；

TCanvas 中鼠标的操作

使用 Root 画图时，可以使用 TCanvas 里面的 Editor，使用鼠标来调节，当然这些操作是可以使用 Draw() 里的参数调节。

->Draw(“Box”)；值散点图里面的每个点是一个” box”（大一点）；

->Draw(“ColZ”)；指有颜色的(带颜色标尺的)散点图；对于这种散点图，若使用 gStyle->SetOptFit(1)；会使出来的散点图的颜色亮一点；

注：gStyle 定义在 ” TStyle.h” 中；

TCanvas 相关的一些函数

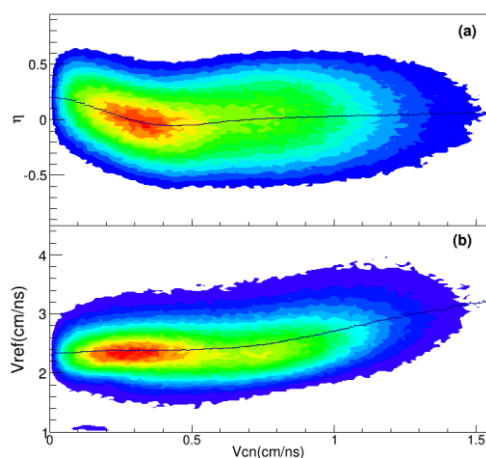
1，当一个对象画在一张 TCanvas 上，当该 TCanvas 被 Delete() 时，该对象也将被 Delete()；

2， GetName()； GetTitle()；
 SetName(char*)； SetTitle(char*)；

3， TCanvas 被 Divide() 之后，出来的 TPad 之间是有距离的，如何将该距离设置为 0；

TCanvas::Divide(2,2,0,0)；

这意味着，分成 2×2 份，同时相互之间的 X-Y 的距离为 0。如上图。



TCanvas 中调整 TPad 的大小

TCanvas::Divide(1,2); 生成的 pad 是等分的;

可以通过:

```
c1->cd(1)->SetPad(0,0.2,1,1);
```

c1->cd(2)->SetPad(0,0,1,0.25); // 其中, 0.2->0.25 可以使得 Pad 之间更紧密;

把纵坐标变为对数

TCanvas::cd()->SetLogy(1); //对于没有 Divide 的 TCanvas

TCanvas::cd(1)->SetLogy(1); //把 Divide 后的 TCanvas 的第一个 pad 设置为
//对数坐标;

TAxis 相关

改变每个小格和大格的数目

```
TAxis::SetNdivisions(ndiv,options); //default=510
```

其中, $ndiv = N1 + 100*N2 + 10000*N3$; //N1: 如同直尺上的厘米表示线;
N2: 如同直尺上的毫米表示线; N3: 毫米下一个单位的表示线, 当然, 一般的直尺是没有的 (一般在作图过程中也是不需要的, 多乱啊, 是吧!)

坐标 label 使用指数表示

TGaxis::DetMaxDigits(6); //大于 10e6 后, 才使用指数表示 (画图时, 讨厌那些 0 吧, 尤其是纵坐标); 可以根据自己的需要, 自行设置, 但建议使用 3;
注意: 该函数改变的是一个全局变量的值, 故可以直接使用, 不需要指定哪个直方图的 axis; 同样因为这是一个全局量, 在写程序时, 如果多个图对这个量的要求不一样时, 这时, 把图存在 ROOT File 中, 然后分别画出来就好。

TGraph, TH1D 在使用 SetRange()不好用,

这时, 你可以试试:

```
::GetXaxis()->SetLimits(double x1,double x2);  
::GetXaxis()->SetRangeUser(2,6); //(2,6)是以画在 TCanvas 上面的对象的  
单位为参考的;  
::GetXaxis()->SetRange(2,6); //(2,6)是 bin 的 Number
```

于 TH1D, TGraph, 可以改变 bin 的 label。

(1), 对于 TH1D, 比较直接, 以 TH1D* h1 为例;

```
h1->GetXaxis()->SetBinLabel(int Bin_Index, char* bin_Label);
```

这样, Bin_Index 的 label 就被替换为 bin_Label, 而其他的 label, 如数字什么的也会被清零;

****同时, 你可以改变 Label 的摆放方式, 如: 水平, 垂直等;**

```
h1->LabelsOption("h"); //水平
```

(2)对于 TGraph* g1; 要使用:

```
g1->GetHistogram();... ..
```

TFile 相关

TFile 实际上和我们平时使用的文件夹的概念是一样的，可以在一个 root 文件里再建立一个文件夹，在不同的文件夹内存储不同的内容。与我们平时使用的文件夹的区别在于：文件夹是操作系统使用来存储问价用的，可以使用鼠标直接访问，而 TFile 是 root 框架下的文件夹，只有 root 环境下，才能访问。

Root 对象的存储

(1) 在 TFile 打开的情况下,cd(), root 的对象可以使用 Write(“name”)存储在 root file 内。如果担心，root 中已经存储相同的对象，不想重复存储，可以使用：Write(“”,TObject::kOverwrite);

(2) Txxx::Write();是指把该对象写到现在打开的 root 文件中；

(3) Txxx::SaveAs(“XX”);是指把该对象保存为 XX 格式，默认为 XX.C;

删除 root 文件中已有的直方图

```
TH1D* h1_tem = (TH1D*) f1_storage->Get(“HistoName”);
if(h1_tem!=0)
{
    string Name_tem = “HistoName” ;
    Name_tem += “;1” ;
    f1_storage->Delete(Name_tem.c_str());
}
```

如不采用这种方法，下面 Write 进来的相同名字的对象的名字后面将依次增加为” ;2” ,” ;3”

关于 ROOT file 的使用说明一

Root file 是一个 hierarchy of directories，就如同我们平时使用的文件夹一样，允许不同级次的目录的存在。如：TFile* f1;

f1->mkdir(“ FileDirectory Name ”);

所以，要建立 a/b/c 这种级次的目录，不能直接使用这种办法，而是：

Tdirectory* cdtof = top->mkdir(“tof”);

Cdtof->mkdir(“ ”);

依次类推。

Root-TFile 相关

其中，`/workdir/root_study/`中有一个这样的例子。

其他相关

TLegend 的使用

```
TLegend(double x1,double y1,double x2,double y2,const char*
header);//x1,y1,x2,y2 是 legend 左下角和右上角的坐标;
->AddEntry(func1," ont Theory", " l" );//" l":TAttLine; " p":
TAttMarker; "F":TAttFill;
->Draw();
->SetHeader("The legent Title");
```

Root 中关于时间的读取和记录

```
{
#include "TDateTime.h"
TDateTime date1;
date1.GetHour();    date1.GetMinute();    date1.GetSecond();
}
```

Event Display 时，免不了画个小球什么的代表 Particle Hit

可以使用如下方法：

```
{
TPolyMarker3D* pm3d = new TPolyMarker3D();
for(int i=0;i<HitNum;i++)
{
... ..
pm3d->SetMarkerSize(4);
pm3d->SetMarkerStyle(8);
pm3d->SetMarkerColor(4);
... ..
pm3d->SetPoint(i,PosX,PosY,PosZ);
}
pm3d->Draw("ogl");
}
```

当然，既然有 TPolyMarker3D，自然就有 TPolyMarker，该类的函数与三维的相同，只是它是画在 2-D 的 Canvas 上的。

画 3-D line

```
// define the parameteric line equation
void line(double t, double *p, double &x, double &y, double &z)
{
    // a parameteric line is define from 6 parameters but 4 are
independent
    // x0,y0,z0,x1,y1,z1 which are the coordinates of two points on
the line
    // can choose z0 = 0 if line not parallel to x-y plane and z1 = 1;
    x = p[0] + p[1]*t;
    y = p[2] + p[3]*t;
    z = t;
}

int n = 1000;
TPolyLine3D *l0 = new TPolyLine3D(n);
for (int i = 0; i <n;++i)
{
    double t = t0+ dt*i/n;
    double x,y,z;
    line(t,p0,x,y,z);
    l0->SetPoint(i,x,y,z);
}
```

TPolyLine3D 可以表示多条相交的直线，当然使用两个点，只能确定一条直线。这个例子实际上是建立了 999 个在同一 Line 上的线段。

TLatex 的位置

TLatex 的定义：

```
TLatex* latex = new TLatex(10,20, " contex" );
Latex->Draw();
Latex->SetTextSize(0.04); //注意这里的 0.04 和 editor 里的数字定义
是不一样的
```

```
Latex->SetTextAngle(90); //将 text 逆时针旋转 90 度
```

注意这里的坐标（10，20），并不像 TLegend，而是采用了绝对坐标。

C++中，this 是一个关键字，值便是本对象的指针

在 ROOT 环境下就要遵守 ROOT 的规则。

(1), 对象不宜取同名，因为名字可以通过 ROOT 的全局指针找到相应的对象；故在使用循环创建对象时，可以使用如下的方法来避免重名：

```
char  ObjectName[200];  
sprintf(ObjectName, " histo%d", i);
```

(2), 要注意 Root 文件的打开与关闭，当一个 Root 文件打开再关闭后，在其打开的这段时间内创建的对象在文件关闭后将会自动清除。

实时显示处理到第几个事件

程序中，可以使用 `if(i%10000)==0{ cout<<" Evt: " <<i<<endl; }`达到**实时**显示的目的，同时又避免向屏幕上输出内堂而占用资源；

调用 Root 库画图，出来的图总是一闪而过

解决办法：

```
#include "TApplication.h"  
int main(int argc, char*argv[ ])  
{  
    TApplication  app( "app", &argc, argv);  
    ... ..  
    app.Run(kTrue);  
return 0;  
}
```

这样，当程序结束时，使用 root 库画出来的图就不会消失了。结束程序可以使用 `ctrl+c`；

TTimer 让一个程序定期去做某件事情

```
#include "TTimer.h"  
TTimer* timer  = new TTimer(2000);  //do one event per 2000ns;  
timer->SetCommand( "function" );  
timer->TurnOn();  
这样每隔 2s 就会执行 funtion 了；
```

注:

该类还没有在 g++ 的环境中使用, 这时可以使用

```
while(1)
{
    sleep(2);
    ... ..
}
```

Root 中经常遇到生成大量的谱图, 希望把它们归整在一个文件中

```
TCanvas* c1_TDC_Cali = new TCanvas("c1_TDC_Cali","c1_TDC_Cali",1);
c1_TDC_Cali->cd();
```

```
for(int i=0;i<5;i++)
{
    for(int j=0;j<5;j++)
    {
        h1_PPAC[i][j]->Draw();
        c1_TDC_Cali->Update();
        Name_tem = h1_PPAC[i][j]->GetName();
        c1_TDC_Cali->SaveAs((Name_tem+".pdf").c_str()); //[1]
    }
}
```

```
gROOT->ProcessLine(".! pdftk h1*.pdf cat output
Histo_OriginalSpec.pdf");
```

```
gROOT->ProcessLine(".! rm h1*.pdf");
```

[1]注意, 这里保存的是 TCanvas, 而不是 Histogram。

但是, 你的 Linux 要安装 pdftk, 使用命令: apt-get install pdftk

root 进入 cint 环境后, 执行带参数的 script

(1) .x function_file_name.C(***)

***是这个脚本执行时, 需要提供的参数。

Root 中有很多全局的环境变量

如: `gStyle`, `gSystem`, `gRandom`, `gPad` 等, 如果要调用, 应该包含相应的头文件

Geant4 相关

Geant4 的全名是 Geometry and Tracking 4，也就是说构建几何，模拟粒子在该几何中的径迹，以及在这过程中和物质发生反应的反应产物的径迹。对于 Geant4 程序，他的结构是固定的，一般情况下仅是在已有的模拟程序下根据自己的需求做些修改等，而不是从新写一个程序。但为了较好的利用这个模拟工具，建议从头学习一下 Geant4 程序，尤其是如果你可能长期和探测器，模拟打交道。下面，将简单介绍 Geant4 的一些常识，然后，就是 Geant4 中一些备忘的函数的用法。

Geant4 的一点认识

Geant4 的程序思路就是：Designer 把框架已经搭好，User 在相应的地方添加自己的要求，如：探测器几何及材料，待模拟粒子的动量及关心的物理反应，记录感兴趣的模拟信息如能损和径迹等。

我认为对于 Geant4 的认识，总共分为几步：

(1)，有基本的核物理探测器的基础知识；

(2)，安装 Geant4，并有一个可以运行的 example (Geant4 本身自带很多 example，其中 Novice/N01 是入门的，告诉你 Geant4 的程序有几个主要模块；Noice/N02 是教你上手的，告诉你，如何 code 出一个有简单实际功能的程序；所以，配合 Geant4 的 document，通读这两个 examples)；

(3)，了解如何定义材料，如何定义几何，如何定义初级粒子，如何定义粒子在物质中可以发生的物理事件；

(4)，到了这，你已经基本上了解 Geant4 的各个部件类，这时，你也知道需要修改哪些类里的哪些函数（这些函数在 geant4 document 里都有介绍）； 建议你在每个你需要修改的函数里使用：`cout << " functionName" << endl;`
`getchar();` 这样，你就知道，那个部件先运行，那个部件后运行，同时你也会清楚他们之间的关系；

(5)，最后一步将 Geant4 中模拟的结果记录在文件里，用于后续分析；这一步，你需要明白 Geant4 里的几个概念：Run, Event, Track, step, Hit；这些概念和我们实际的物理试验时所涉及的很相似。

Run: 实验上, 数据获取并不是将所得的数据记录在一个大文件里, 而是 start->(maybe 1hour)->stop->start->(maybe 1hour)->stop... 所以实验结束时, 你会得到一系列格式相同的数据文件。因为探测器或是后续电子学的性能是会漂移的, 所以, 一般假设, 在一个文件里的数据, 探测器和电子学的性能是相同的, 从而使用一套刻度系数。其中, Run 是指从一个 start -> stop, 这样也就是说, 一次实际实验, 会有很多 run, 一个 run 里, 探测器的几何和性能, trigger 是相同的;

Geant4 中, 同理, 一次 Run 意味着几何和材料参数等是相同的;

Event: 实验上, 发生了一次核反应, 并且反应产物符合 trigger 的条件, 则这些产物在探测器里留下的信息将被记录; 实验结束后, experimentalist 就根据这些信息, 重建出核反应的一些信息, 从而研究核反应;

Geant4 中, 意味着初始化了一些初始的待模拟的粒子 (你可以认为这些待模拟的粒子就是一次核反应的产物), Geant4 跟踪这些粒子, 直到他们的能量小于 threshold 或是出了模拟的边界; 所以, 一个 Run 里会有很多 Event;

Track: 实验上, Track 就是反应产物的飞行径迹;

Geant4 中, Track 是由一系列小的 Step 来近似描述的; 一个 step 包含如下信息: 粒子的 PID, 粒子的初始位置和动量信息, 粒子的结位置和动量信息, 这个 step 过程中的能损等等; 这样, 使用一系列的 step 就可以描述一个粒子在 Geometry 里面的绝大部分信息;

Hit: 实验上, Hit 一般是指, 一个粒子击中探测器就叫一个 Hit; 探测器将会根据 Hit 的位置以及能损有相应的信号输出;

Geant 中, 探测器输出什么样的信号要由模拟的人根据 step 的信息和探测器的工作原理总结出一个相应的信号, 用于记录;

探测器都会有探测区和支架保护部分。所以, 模拟时, 你就要指定, 告诉 Geant4 那部分几何是灵敏区(sensitive 探测区), 统计在这个区里的 step 信息生成相应的 hit 信息, 用于存储。存储就是在 EndofEvent 里将 hit collection 里的 hits 再整理整理 (比如做做在线显示, 能量弥散, 位置整合等等), 再将整理的数据存在文件中。

最后, 在 EndofRun 里, 关闭存储, 清理该清理的等等。

(6) 当你完成以上这几部, 你基本上就可以写出简单实用的 Geant4 模拟程序了; 但, 其中的 physicslist 部分是你要注意的, 要将你感兴趣的物理反应道都包含进去, 否则, Geant4 将什么都不干的。

(7) 剩下的就是 Geant4 和 C++ 的经验了。Geant4 难学处在于学习他的人没有 C++ 的丰富经验，对类的继承等还不适应。所以，如果你以前有 C++ 的丰富经验，并且了解探测器的基本知识，那么 Geant4 很容易就上手。

现在的一些程序都是采用 Geant4 这种思路，程序的整体结构都已由工程师厘定好，他们把接口类做好，你所需要做的就是继承这些类，在这些类相应的成员函数中，添加自己的要求。所以，学习使用 Geant4 的好处之一就是再碰到这样大程序，你会很快知道自己如何修改他。如：NSCL 的 DAQ 就是使用这样的思路。

现在，Geant4 的最新版本是 4.10.0，GUI 很友善，物理反应的 data 也扩展了很多，所以建议直接用最新版本。

几何相关

这里面应注意的是，在指定几何体的边长，给出的都是一半的长度，所以，记着*0.5.

定义 Tube，Box，以及 logical，physical

这个例子定义了一个 Box，Tube，并把 Tube 沿着 Y 轴旋转 90 度放在 Box 中。其中的 **copy number**，在有多个 sensitive 几何体时，是个利器。

```
{
    //for the Neutron Slower
    G4ThreeVector position_Slower(0,0,Slower_Position);
    G4Box* solidSlower = new
    G4Box("solidSlower",0.5*Slower_Length,0.5*Slower_Height,0.5*Slower_
    Thick);
    G4LogicalVolume* logicalSlower = new
    G4LogicalVolume(solidSlower,Water_Natural,"logicalSlower",0,0,0);

    //for the Neutron Tube
    G4ThreeVector
    position_NeutronTube(0,0,Neutron_Tube_Position_in_Slower);
    G4RotationMatrix* NeutronTube_Rot = new G4RotationMatrix();
    NeutronTube_Rot->rotateY(90*deg);

    G4Tubs* solidNeutronTube = new
    G4Tubs("solidNeutronTube",0,Neutron_Tube_rmin,0.5*Neutron_Tube_Leng
    th,0,twopi);
    G4LogicalVolume* logicalNeutronTube = new
    G4LogicalVolume(solidNeutronTube,_3He,"logicalNeutronTube",0,0,0);
    new G4PVPlacement(NeutronTube_Rot, // rotated along the Y axis
    position_NeutronTube, // at (x,y,z)
    logicalNeutronTube, // its logical volume
    "physNeutronTube", // its name
    logicalSlower, // its mother volume
    false, // no boolean operations
    2); // copy number
}
```

定义材料

Geant4 中已经内置了很多常用的材料

使用它会带来很大方便，对于气体材料，它也允许修改温度和压强。下面是从例子中总结出来的一些方法。

应包括的头文件：

```
#include "globals.hh"
#include "G4Material.hh"
#include "G4NistManager.hh"
```

从 **NistManager** 中调取材料：

```
G4NistManager* man = G4NistManager::Instance();
man->SetVerbose(1);

// Define elements
G4Element* C = man->FindOrBuildElement("C");
G4Element* Pb = man->FindOrBuildElement("Pb");

// Define Pure NIST materials
G4Material* Al = man->FindOrBuildElement("G4_Al");
G4Material* Cu = man->FindOrBuildElement("G4_Cu");

// Define NIST materials
G4Material* Water = man->FindOrBuildMaterial("G4_WATER");
G4Material* SiO2 = man->FindOrBuildMaterial("G4_SILICON_DIOXIDE");
G4Material* Air = man->FindOrBuildMaterial("G4_AIR");

// Define HEP materials
G4Material* PbWO4 = man->FindOrBuildMaterial("G4_PbWO4");
G4Material* Vacuum = man->FindOrBuildMaterial("G4_Galactic");

// 定义非标况下的气体
G4Material* ColdAr = man->ConstructNewGasMaterial("ColdAr",
"G4_Ar", 120*kelvin, 0.5*atmosphere);

// 显示已经定义的材料
G4out<<*(G4Material::GetMaterialTable())<<endl;

故在使用 Geant4 内部定义好的材料时，最主要的就是知道它在内部的名字，下面记录几个备忘：
Malar 膜 : G4_MYLAR
甲烷 (CH4): G4_MEHTANE
氩气 (Ar) : G4_Argon
```


定义一种分子

G4 中总会有没有定义的分子，这时，如果知道了分子的化学组分，可以自己定义。

```
{
  G4double z, a, density;
  G4String name, symbol;
  G4int ncomponents, natoms;
  a = 1.01*g/mole;
  G4Element* elH = new G4Element(name="Hydrogen",symbol="H", z=1.0, a);
  a = 16.00*g/mole;
  G4Element* elO = new G4Element(name="Oxygen",symbol="O", z=8.0, a);
  Density = 1.000*g/cm3;
  G4Material* H2O = new G4Material(name="Water",density, ncomponents=2);
  H2O->AddElements(elH, natoms = 2);
  H2O->AddElements(elO, natoms = 1);
}
```

定义 Helium3（该方法定义的 ^3He 能和中子反应）

```
{
  G4int protons = 2; G4int neutrons = 1; G4int nucleons = protons+neutrons;
  G4int isotopes = 1; G4int elements = 1;
  G4double atomicMass = 3.016*g/mole;
  G4Isotope* he3 = new G4Isotope("He3", protons, nucleons, atomicMass);
  G4Element* He3 = new G4Element("Helium3", "He3", isotopes=1);
  He3->AddIsotope(he3, 100*perCent);
  G4double pressure = 4*bar;

  G4double temperature = 2*kelvin;
  // G4double molar_constant = Avogadro*k_Boltzmann; \\from clhep
  G4double density = 0.5358*mg/cm3;
  _3He = new G4Material("Helium3", density, elements=1, kStateGas, temperature,
  pressure);
  _3He->AddElement(He3, 100*perCent);
}
```

由多种材料制作混合材料

（1）由元素混合材料

这与定义一种分子的材料相同，仅是在最后说明材料组分的时候不同。如下：

```
{  
    density = 1.000*g/cm3;  
    G4Material* H2O = new G4Material(name="Water",density, ncomponents=2);  
    H2O->AddElements(elH, fractionmass=(1.0/9.0)*perCent);  
    H2O->AddElements(elO, fractionmass=(8.0/9.0)*perCent);  
}
```

(2) 从已定义的材料混合成材料

```
{  
G4Material::AddMaterial(G4Material* material, G4double fraction);  
}
```

注意：这里混合材料时，使用的都是质量的比，而我们在实验过程中说的工作气体一般都是体积比，所以，如果通过这种方法来定义混合气体，应该换算过来。

G4Material 同时包括如下函数，可用于调用：

```
inline const G4String& GetName() const{return fName;}  
inline const G4String& GetChemicalFormula() { return  
fChemicalFormula; }  
inline G4double GetDensity() const{ return fDensity; }  
inline G4State GetState() const{ return fState; }  
inline G4double GetTemperature() const{ return fTemp; }  
inline G4double GetPressure() const{ return fPressure;}
```

粒子的发射

如何设置发射粒子的类型，发射的位置，方向，能量等。这些功能都是在 `G4VUserPrimaryGeneratorAction` 类中实现的。该类中包括的函数 `void GeneratePrimaries(G4Event* anEvent)`，在每个 `Event` 开始时，都会被执行一次，通过这个函数来指定本事件中的初始粒子。所以，在这个函数中，你可以按照某种分布指定粒子的类型，发射位置，发射方向，能量等。

关于产生粒子的 `ParticleGun`，它实际上就是 `Vertex` 的产生器，你把粒子的类型，运动信息，粒子的个数指定给他，它便会将生成的 `Vertex` 返回给 `EventManager`。

对于发射粒子，有些时候，你需要模拟多种放射源的，或是多个谱线。这时，我建议，最好要分开来模拟，或是在模拟过程中，也记录发射出来的粒子的**能量或是源**。在分析过程只要根据发射源各个谱线的相对强度，处理模拟出来的结果就行了。这样，你不但知道了你的探测器对不同能量的射线的响应情况。而且，当与实验对比出现问题时，这也是你可以根据这个做适当的分析。

定义粒子

定义粒子的方法根据粒子的不同分为两类：普通粒子；重离子。

`//在头文件中做如下定义`

```
G4ParticleTable* particleTable;
G4ParticleDefinition* particle;
G4ParticleGun* particleGun;
particleGun = new G4ParticleGun(G4int n_Particle=1);
//这样一个 Gun 里就只发射出一个粒子
}
```

普通粒子

如：e⁻, e⁺, proton, alpha, neutron, gamma;

`//通过粒子名称直接指定`

```
particleTable = G4ParticleTable::GetParticleTable();
particle = particleTable->FindParticle( "proton" );
particleGun->SetParticleDefinition(particle);
}
```

重离子

`//通过 N, Z 指定`

```
particleTable = G4ParticleTable::GetParticleTable();
particle = particleTable->GetIon(Particle_Z, Particle_A, 0); // [1]
```

```
particleGun->SetParticleDefinition(particle);  
}
```

[1] 0 意味着激发能

指定方向，初始位置，能量

```
{  
    particleGun->SetParticlePosition(G4ThreeVector(PosX, PosY, PosZ));  
    particleGun->SetParticleMomentumDirection(G4ThreeVector(0, 0, 1));  
    particleGun->SetParticleEnergy(ParticleEnergy);  
    particleGun->GeneratePrimaryVertex(anEvent); //将 particleGun 指定给  
event。  
}
```

注意：对于发射出来的粒子，应在 RunAction 中定义几张直方图，统计发射出来的粒子的一些动力学信息。比如：发射能量，发射角度，等等。记录这些信息有助于检查模拟结果。

指定多个粒子

这多个粒子的指定实际上就是多次运行：

```
{  
    particleGun->GeneratePrimaryVertex(anEvent);  
}
```

该语句每运行一次，就会由 particleGun 生成 nParticle 个粒子，并将其传递给 anEvent。所以，如果你想一个事件中指定多个不同粒子（一个物理事件通常都是多个产物），仅需在执行这句话之前，通过前面介绍的方法，改变 particleGun 里面粒子的定义，能量，运动方向等即可。

初始化方向的一些函数

```
#include "TRandom.h"  
#include "TF1.h"  
#include "TMath.h"  
using namespace TMath;  
{ double Dir_X, Dir_Y, Dir_Z; }
```

1, 4π 同性发射

```
{  
    TF1* f1_sin_theta = new TF1("f1_sin_theta", "sin(x)", 0, pi());
```

```
double phi_tem    = gRandom->Uniform(0, 2*Pi());
double theta_tem = fl_sin_theta->GetRandom(0, Pi());

Dir_X = Sin(theta_tem)*Cos(phi_tem);
Dir_Y = Sin(theta_tem)*Sin(phi_tem);
Dir_Z = Cos(theta_tem);
}
```

2, 指定一个方向 \vec{k} , 沿着该方向张开 θ 角, 各向同性发射

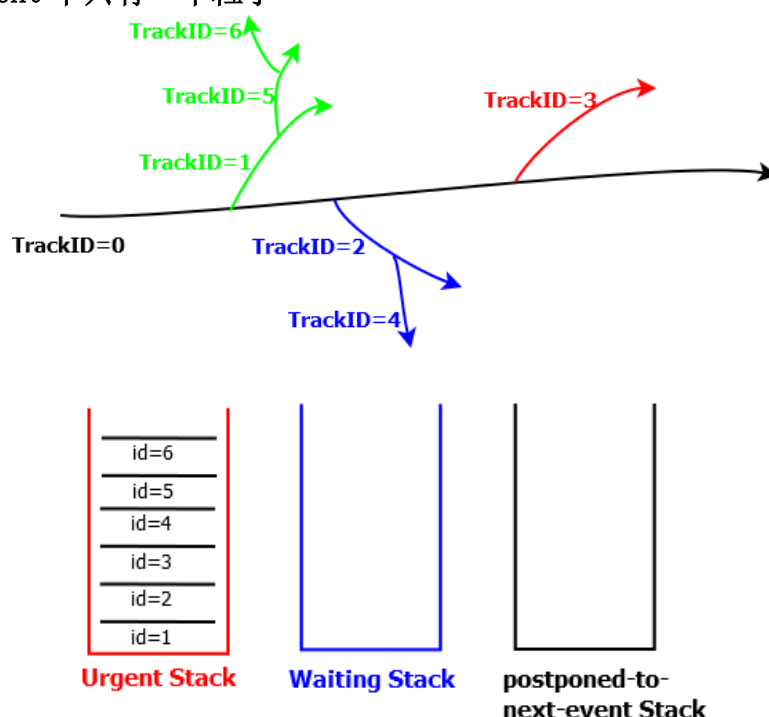
实际上, 对于 4π 同性发射的思路就是沿着 z 轴, 张开 $\theta = \pi$ 来各向同性发射。这里为了得到沿着 \vec{k} 方向的随机向量, 我的想法是: 将 z 轴通过转换矩阵 C 变换到 \vec{k} 方向; 将按 z 轴各向同性的随机向量通过变化矩阵 C 来变换, 即可。这个代码有点长, 所以, 就当附录附在后面了。

物理过程相关

G4 跟踪粒子的策略

下面是我对 Tracking 过程的理解。

1, 一个 Event 中只有一个粒子



在 G4 模拟时，有三个 stack 用于存储模拟过程中产生的粒子（使用 G4Track 代表），分别是：Urgent Stack，Waiting Stack，Postponed Stack，其用途在下面两段中解释。

首先 G4EventManager 利用粒子的 Vertex 信息生成一个 G4Track，该 Track 的 ID 是 0，并将其传递给 G4StackManager。G4StackManager 便会开始 step-by-step 的模拟该径迹。在该径迹的模拟过程中自然可能会产生次级粒子（ δ 电子等），这时会根据该粒子的信息生成 G4Track，并被保存在 Urgent Stack 中；这样如再产生次级粒子，依旧会被保存在 Urgent Stack 中，直至初始粒子的能量低于 Cut Threshold；然后再从 Urgent Stack 中抽出 G4Track 像模拟初始粒子一样，直至其能量低于 Cut Threshold；...直至 Urgent Stack 中的 G4Track 被取干净。

在模拟过程中，如果有些粒子想要最后模拟，可以把 G4Track 直接存在 Waiting Stack 中，这样当 Urgent Stack 空了后，再来访问 Waiting Stack。当然，也有可能，该 G4Track 延后到下一个事件中在模拟，这时，直接把 G4Track 放到 Postponed-to-next-event Stack 中，这样，在下一个 Event 模

拟之前，会先看看这个 Stack 中有没有 G4Track，如果有，就把其调到 Urgent Stack 中。

如何定义 reference physics List

方法一：

```
G4int verbose = 1;
FTFP_BERT* physlist = new FTFP_BERT(verbose);
runManager->SetUserInitialization(physlist);
```

方案二：

```
G4int verbose = 1;
G4PhysListFactory factory;
G4VModularPhysicsList* physlist =
factory.GetReferencePhysList("FTFP_BERT_EMV");
physlist.SetVerboseLevel(verbose);
runManager->SetUserInitialization(physlist);
```

记录模拟信息

Geant4 只负责对粒子的径迹跟踪，探测器对这些径迹穿过时的响应就要由用户根据自己探测器的性能做出相应的响应。这过程中，要通过 G4Step 得到各种信息，现整理如下。

从 G4Step 中提取 step 的信息

1, 一个 step 包括两个点，分别代表 step 的开始与结束

```
{  
    G4StepPoint* prePoint = aStep->GetPreStepPoint();  
    G4StepPoint* postPoint = aStep->GetPostStepPoint();  
}
```

2, 通过这两个点，可以得到这两个点的位置

```
{  
    G4ThreeVector Pos1 = G4Step::GetPostStepPoint()->GetPosition();  
    G4ThreeVector Pos2 = G4Step::GetPreStepPoint()->GetPosition();  
}
```

3, 通过这两个点，计算 step 的时间

```
{  
    G4double globleTime = 0.5 * ( prePoint -> GetGlobalTime() +  
    postPoint -> GetGlobalTime() );  
}
```

4, 得到能损，长度的信息

```
{  
    G4Step::GetTotalEnergyDeposit(); //得到能损信息  
    G4double GetStepLength(); //得到该 step 的长度;  
}
```

通过 step 获取运动着的粒子的信息

```
{  
    G4Track* gTrack = G4Step::GetTrack();  
    gTrack->GetDefinition()->GetPDGCharge() == 0; //判断创造该 step 的  
    track 的电荷是否等于 0;  
    gTrack->GetDefinition()->GetParticleName() == " e- " ; //判断该 track  
    的粒子的名字是否是" e- " ;
```



```
gTrack->GetTotalEnergy(); //得到该 Track 的能量;
G4int TrackID = gTrack->GetTrackID(); //每个 Track 在创建之初, 都会有个 ID, 第一个 ID 是 0;
G4ThreeVector GetMomentum() const; //获得动量
}
```

通过 step 获取几何相关的信息

```
{
    G4StepPoint* prePoint = aStep->GetPreStepPoint();
    G4StepPoint* postPoint = aStep->GetPostStepPoint();
    G4TouchableHandle* touch1 = prePoint ->GetTouchableHandle();
}
```

1, 几何重复使用, 通过 copyNum 调用

在核物理实验中, 经常会出现相同几何体的重复使用, 其不同之处就是位置以及方位。为了方便数据提取, 在摆放这些几何体时, 可以设置不同的几何体的 pCopyNumber 的不同。在 ProcessHits 中的调取 pCopyNumber 的方法如下:

```
{
    G4int copyNum = touch1->GetCopyNumber();
    G4int copyNum_mom = touch1->GetCopyNumber(1);
    G4int copyNum_GrandMom = touch1->GetCopyNumber(2);
}
```

2, 通过 step 获取几何体的名字

```
{// 通过 TouchableHandle
    G4VPhysicalVolume* Volume = touch1->GetVolume();
    G4string name = Volume->GetName();
}
{// 通过 aStep 直接返回
    G4VPhysicalVolume* Volume = aStep->GetPhysicalVolume()const;
}
```

3, 判断是否为几何的边界

```
{
    if(prePoint ->GetStepStatus()==gGeomBoundary){ //进入几何 }
    if(postPoint->GetStepStatus()==gGeomBoundary){ //出去几何 }
}
```

记录模拟信息的方法

这可能是初学者最搞不清楚的地方，这里尽我所知介绍一下。

Geant4 中材料，几何，物理过程，粒子的定义都是比较直观的，看过说明书以及相关例子后，都会快速的理解，但是模拟信息的记录一般涉及到：DetectorConstruction, SD, Hit, EventAction, RunAction, 这样 5 个类，所以都会感觉到乱。这里就是介绍我如何利用这 5 个类来存储模拟的信息的。

1, Detector Construction

建造几何在模拟程序的 coding 过程中是最先写的。建造的几何中，不同的部件起着不同的作用，如：屏蔽，支撑，探测器敏感区等。无论是何种部件，当粒子进入后，Geant4 对该粒子的跟踪策略是不会变化的，即，Geant4 本身可不关心这个部件是做什么的，只要该发生能损就能损，该发生核反应就发生核反应。我们建模的人就得根据不同部件的功能，以及我们关心的信息，对不同部件中的 step 进行信息记录，如：探测器敏感区的能损，屏蔽体的能量沉积等。

所以，为了记录信息，第一步做的是将感兴趣的部件设置为敏感区 Sensitive Area。设置敏感区实际上就是将该部件与一个 SD 的类建立关联，当粒子进入该部件后，G4 就会将部件中的每个 step 的指针发送到 SD 的一个函数中，在该函数中，你按照自己的需要留下相应的信息就可以了。

下面是我将一个几何体设置为敏感区：

```
{
// Sensitive detectors
G4SDManager* SDman = G4SDManager::GetSDMpointer();
G4String SDNaI_Name = "Sim/NaISD";
TrackerSD* aTrackerSD = new TrackerSD( SDNaI_Name );
SDman->AddNewDetector( aTrackerSD );
Crystall_Logical->SetSensitiveDetector( aTrackerSD );
}
```

这小段代码中，类 TrackerSD 就是上面我们说的 SD 类。要建立关联，要先将 aTrackerSD 在 G4SDManager 中“注册”，再将 aTrackerSD 指定给你杨瑶关联的几何的 logical 指针 Crystall_Logical。有了这一段代码，在关心的几何中有 step 时，就会发送到类 TrackerSD 中，用于信息记录。

注：一个模拟程序中可以有多个不同的敏感区，每个敏感区都可以有自己的 SD 类。当然，不同的部件可以共用一个 SD，即这几个几何体中记录的信息是相同的。

2, Sensitive Detector 与 Hit

指定好了敏感区后，我们看一下这个 SD 类是如何记录感兴趣信息的。首先我们得知道，一个事件中，在一个敏感区中，一条径迹可能会有多个 step (，

或者是多条径迹)。我们关心的信息一般都是敏感区中所有 steps 给出的效果的总和，而不是一个个的 step。所以，一个好的策略就是，对于每个 step 创建一个对象 Hit，将该 step 中提取的信息都赋值给 Hit。然后，将这些 Hits 保存在 SD 类中的一个 Collector 中，等到一个事件模拟完了，再统一的到这个 Collector 中来处理 Hits。

下面我们就来看看，如何定义 Hit，Collector，SD 类中如何提取 step 的信息等。

对于 Hit 的定义，这个很简单。从 step 中关心什么信息，在 Hit 中就定义一个变量来存储就行。比如：double Edep; //能损; double TrackID; //径迹 ID; double FlightT; //飞行时间; 等等。当然例子中的那些赋值的，取值的函数，直接 copy 就好。

我们知道要定义的 Collector 是用来存储上段中我们定义好的 Hit 的，所以，一般，我们就在 Hit 的定义文件中定义一个能够存储 Hit 的 Collector，如下面的两行语句所示：

```
{
    typedef G4THitsCollection<TrackerHit> TrackerHitsCollection;
    extern G4ThreadLocal G4Allocator<TrackerHit>* TrackerHitAllocator;
} //对于这个 Collector 的定义，我也没弄清楚，只知道这样写两行定义就行了。
```

现在我们来看一下 SD 类是如何初始化 HitsCollection，又如何从 step 中提取信息来填充到 Hit 中，又如何将 Hit 填到 HitsCollection 中。

这个 SD 类是继承 G4VSensitiveDetector 的，这个类中有三个函数，上面三个要做的事就是在前两个函数中完成的：

```
void Initialize(G4HCofThisEvent* HCE);
```

在这个函数中初始化 HitsCollection，如下语句，分别解释如下：

```
{
    trackerCollection = new TrackerHitsCollection
(SensitiveDetectorName, collectionName[0]); //[1]
    static G4int HCID = -1;
    if(HCID<0)
    { HCID = G4SDManager::GetSDMpointer() -> GetCollectionID
      ( collectionName[0]); //[2]
    }
    HCE->AddHitsCollection( HCID, trackerCollection ); //[3]
}
```

[1]这两个字符串的意义如他们的名字一样，第一个是探测器的名字，第二个是 Collection 的名字，后面，我们就是根据这个名字来调用这个用于存储 hit 的 collection。

[2]向管理 SD 的 Manager 注册一下，让他给你建好的 collection 分配个 ID。
 [3]利用这个 ID，你就可以把你的 HitsCollection 的指针保存在 HCE 中。通过 HCE，在模拟事件结束后，就可以调用 HitsCollection 了。

G4bool ProcessHits(G4Step* aStep, G4TouchableHistory*);

这个函数就是把在设置了 SD 中的 step 通过指针 aStep 传递过来。关于从 step 中能拿到那些信息，可以参考本文档中其它介绍。这里仅仅介绍如何存储信息。

```
{
    G4double edep = aStep->GetTotalEnergyDeposit();
    if(edep==0.) return false;
    TrackerHit* newHit = new TrackerHit();
    newHit->SetEdep      (edep);
    newHit->SetPos       (aStep->GetPostStepPoint()->GetPosition());
    trackerCollection->insert( newHit );
    return true;
} //这段代码就没什么好说的了，一目了然：将从 step 中拿出的信息赋给
//hit，再存储 hit 到 HitsCollection 中。
```

void EndOfEvent(G4HCofThisEvent*);

对于这个函数，我以前都是空置的，不在这做什么。但现在感觉，以前在 EventAction 中的 EndOfEvent 做的事情（即统计 Hits 中的信息，并将这些信息存储下来）实际上应该在这完成。但这方面还没有更改，所以本文档中还是照着以前的方法来弄。

3, EventAction, RunAction

如刚才所说，现在已经创建很多 hits，其中保存着粒子通过关心区域时留下的信息，现在我来介绍一下如何提取和记录这个信息。关于提取信息我是在 EventAction 中完成的，记录是在 RunAction 中完成的，下面分别概述之。

信息的提取

提取的过程在 EventAction 的 EndOfEvent() 中完成，如下面的代码所示：

```
{
    G4DigiManager* m_DM = G4DigiManager::GetDMpointer();
    void EventAction::EndOfEventAction(const G4Event* evt)
    {
        G4int HCI = -1;
        HCI = m_DM->GetHitsCollectionID("NaICollection"); //通过名字得到
        if(HCI>=0)
        {
            TrackerHitsCollection* HC = 0;
            HC = (TrackerHitsCollection*) (m_DM->GetHitsCollection(HCI));
```

```

    G4int n_hit = HC->entries();//看看这个 collection 中有多少个 hits
    if(n_hit>0)
    {
        double Edep_total = 0;//这里仅仅统计了在敏感区沉积的能量总和
        for(G4int i=0;i<n_hit;i++)
        {
            Edep_total += ((*HC)[i]->GetEdep())/MeV;
        }
        m_Run->SetEnergy_Detect(Edep_total);//[1]
        m_Run->FillData();//[1]
    }
}
}
}

```

[1]我一般的做法是在在定义 EventAction 时将 RunAction 的指针传递过来，在 RunAction 中定义好数据结构，在 EventAction 中将提取的信息通过 RunAction 的指针传递给它（m_Run->SetEnergy_Detect(Edep_total);），待所有的数据都传递过去之后，再保存即可（m_Run->FillData();）。

信息的记录

现在，我一般都会使用 TTree 来保存数据。这样的保存方法不但方便，而且便于后续处理。关于如何在 Geant4 中调用 Root 的库，可参考本文档中关于 makefile 的书写部分（这个很简单，仅仅是在原有的 makefile 后面跟上三行命令即可）。关于定义 TTree 的结构比较简单，具体参考本文档中关于 TTree 的创建即可，这里不再赘述了。

就这样，我们把数据存储下来了。实际上不麻烦，就是对于初学者会因为在这几个类中转来转去，弄糊涂了。

终止次级粒子

在模拟过程中，有时需要删除次级粒子，这时可以考虑：在 TrackManager 将 stack 中的粒子取出，初始化新 track 时，删除该粒子。具体实现如下：
G4UserStackingAction 的 G4ClassificationOfNewTrack ClassifyNewTrack (const G4Track* fTrack)是在每次 track 初始化时调用的，TrackManager 根据该函数的返回值决定如何处理从 stack 中取出的粒子。我们就可以在该函数中选择我们想删除的粒子，然后返回 fKill（杀掉该粒子）即可，否则返回 fUrgent 继续模拟。

```

{

```

Geant4

```
G4ParticleDefinition* thePartDef = fTrack->GetDefinition();
string partName = thePartDef->GetParticleName();
if(partName==" gamma" || partName==" e-" ||partName==" e+" )
{ return fKill; }
else
{ return fUrgent; }
}
```

其他相关:

G4ThreeVector 的一些常用函数

得到 xyz, `getX(); getY(); getZ();`
设置 xyz, `setX(); setY(); setZ();`
得到幅度: `getR();`
更多的函数可以参考 `include` 中的定义

G4BestUnit 显示合适的单位

```
#include "G4BestUnit.hh"
cout<<" Steplength: " <<G4BestUnit(aStep->GetStepLength(), " Length" )
<<endl;;
cout<<" Step Energy: " <<G4BestUnit(edep, " Energy" )<<endl;
cout<<" Step Time: " <<G4BestUnit(time, "Time" )<<endl;
```

在给程序查错时, 对某些量的单位搞不清时, 最简单的方法是:

```
G4cout<<G4double time/ns<<G4endl;
```

这样在输出时, 将会自动考虑单位的换算;

Geant4 中常用的粒子的名字

```
" gamma" ; " e-" ; " e+" ; " mu+" ; " mu-" ; " proton" ; " neutron" ;
" pi-" ; " pi+" ; " alpha" ; " He3" ;
" GenericIon" ; " chargedgeantino"
```

Some Tips in Geant4

```
G4int Event_id = G4Event::GetEventID();
G4int Run_id = G4Run::GetRunID();
```

附录

Root 的安装过程

Linux 下软件安装肯定要遇到的问题：库的依赖。Root 也依赖大量的库，根据不同的 Linux 版本，Rooters 已经整理出相应的依赖库，照着安装就行了。

链接为：

<https://root.cern.ch/drupal/content/build-prerequisites>

注，这个地址可能会变化，但是在网页中的位置应该不会变：Root 主页>Download>Building Root>prerequisites page>Your Linux Version

安装这些依赖库之后，下载 ROOT 的安装包，解压缩在某个文件夹下，进入该文件夹，依次执行：

```
./configure --enable-gdml
make
make install
```

这时，编译好的 root 库已经安装在你解压缩的文件夹下了，为了运行方面，你还需要设置环境变量：

```
#####
##for ROOT
export ROOTSYS=$PACKAGES_PATH/root
source $ROOTSYS/bin/thisroot.sh
export
#the below sentence is for my root library
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$PACKAGES_PATH/myLibs_root/General
```

Geant4 的安装过程

Geant4 的升级比较频繁，但每次升级带来各方面很大的改善。Geant4.10.01 算是一次比较大的升级，和以前的版本差异很大，自然以前的程序也是不能在这个版本下做事的，需要简单的修改。

这里仅简单地介绍其安装步骤。其实，其安装步骤和 Geant4.9.6 是一样的，这里仅仅是把 Geant4.9.6 的 copy 过来。

1，安装 GDML 相关的库（如果 Geant4 中不准备使用 GDML 最为几何的导入导出，可以跳过，在后面编译 Geant4 的时候，把 gdml 的选项删掉就行了）
下载 xerces 的源文件：<http://xerces.apache.org/xerces-c/download.cgi>

下载 xerces-c-3.1.1, 将其解压缩, 然后依次执行如下命令:

- (1) ./configure
- (2) make
- (3) 在 root 权限下, make install
- (4) 设置环境变量

```
#####
```

```
##for xerces-c-3.1.1
```

```
export XERCESC_LIBRARY=/usr/local/lib
```

```
export XERCESC_INCLUDE_DIR=/usr/local/include/xercesc
```

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/lib
```

2, 安装 geant4 之前, 应该安装一些可能的依赖库, 在 ubuntu 下, 库的名称是:

```
libX11-devel mesa-libGLU-devel xorg-x11-proto-devel libXt-devel  
libXmu-devel qt-devel expat-devel
```

当然具体应该安装什么, 取决于你的系统已经安装了什么库, 最实战的方法就是在后面编译 Geant4 时, 对什么库报错, 你就再回过头来安装什么, 是最稳妥的。在安装过程中, 有时会报错, 尤其是和视图相关的部分。关于这部分报错, 显卡驱动的正确安装是关键, 实际上, 只要模拟的结果正确, 视图的不正常是没有太大影响的。

3, 下载的 geant4.10.01 源代码, 依次执行下面的步骤:

(1) 解压缩 geant4.10.01, 在同一文件夹下创建 geant4.10.01-build, geant4.10.01-install 两个文件夹;

(1.5) 如果你已经有 geant4.10.01 的 data, 为了避免再下载, 可以直接把你的 data 文件夹 copy 到: geant4.10.01-install/share/Geant4-10.1.0/

当然如果你的 geant 版本是 geant4.9.6.p03, 那么你的 data 应 copy 在: geant4.9.6.p03-install/share/Geant4-9.6.3/

(2) 进入 geant4.10.01-build, 执行:

```
cmake -
```

```
DCMAKE_INSTALL_PREFIX=/home/zhong/Packages/geant4/geant4.10.01-
```

```
install -DGEANT4_USE_GDML=ON -DGEANT4_INSTALL_DATA=ON -
```

```
DGEANT4_USE_QT=ON -DGEANT4_USE_OPENGL_X11=ON -
```

```
DGEANT4_USE_RAYTRACER_X11=ON ../geant4.10.01
```

这行命令执行完成后, 编译器将会查你的环境, 看看你是否具有相应的库, 编译器是否合格等等。

(3) 如果不出问题, 执行: make -jn (n 为你想要用的 CPU 的数目, 一般就全用上吧, 快很多)

附录

(3.5) make install

(4) 设置环境变量:

```
#####  
##for Geant4.9.6.p01  
export Geant4_9_6_path=$PACKAGES_PATH/geant4/geant4.9.6.p01-install  
source $Geant4_9_6_path/bin/geant4.sh  
source $Geant4_9_6_path/share/Geant4-9.6.1/geant4make/geant4make.sh
```

4, 测试: 找个例子, 试试。

注意:

- (1) 现在的 Geant4 安装方法简单了很多, 实际上, 如果你旁边要是已经安装好的 Geant4, 你直接将 geant4.10.01 和 geant4.10.01-build 拷贝到你的电脑下, 直接编译, 这样将会省去你下载数据的时间。我已经把这个压缩成包, 在硬盘中, 备好份了。
- (2) 因为弄 Geant4 模拟的人, 难免会有很多个任务, 这时使用一个脚本来变换 G4WORKDIR 是个好方法, 在你程序里写一个这样的脚本, 想要运行那个程序, 就执行相应的脚本, 很是方便。

```
#!/bin/bash
```

```
export G4WORKDIR=$HOME/geant4_workdir/N02
```

Color in ROOT

Figure 9-27 The basic ROOT colors

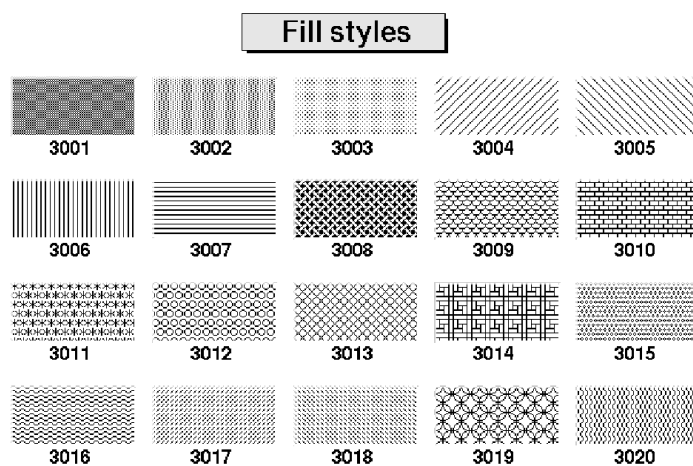


下面是使用单词表示的颜色, 想要使用这种方法调用, 只需在单词前面加 k 即可, 如 kBlack, kWhite 等

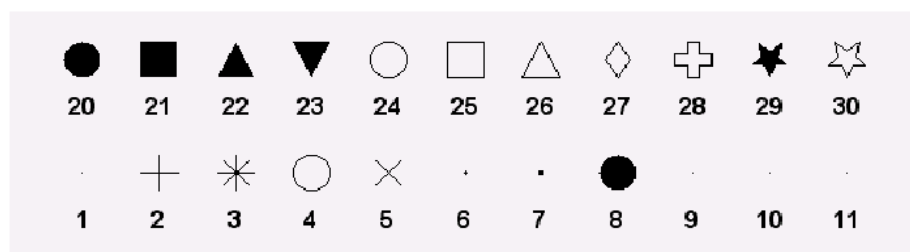
附录

颜色	颜色实效	英文名	十六进制 RGB 值
黑色		Black	#000000
银灰色		Silver	#C0C0C0
灰色		Gray	#808080
白色		White	#FFFFFF
绛紫色		Maroon	#800000
红色		Red	#FF0000
紫色		Purple	#800080
紫红色		Fuchsia	#FF00FF
绿色		Green	#008000
草绿色		Lime	#00FF00
橄榄色		Olive	#808000
黄色		Yellow	#FFFF00
海蓝色		Navy	#000080
蓝色		Blue	#0000FF
黑绿色		Teal	#008080
淡蓝色		Cyan	#00FFFF

Area Fill Style



MarkerStyle



GreekLetter and Math type

♣ #club	♦ #diamond	♥ #heart	♠ #spade
∅ #voidn	ℵ #aleph	ℵ #Jgothic	ℵ #Rgothic
≤ #leq	≥ #geq	< #LT	> #GT
≈ #approx	≠ #neq	≡ #equiv	∞ #propto
∈ #in	∉ #notin	⊂ #subset	⊄ #notsubset
⊃ #supset	⊆ #subsepeq	⊇ #supseteq	∅ #oslash
∩ #cap	∪ #cup	∧ #wedge	∨ #vee
© #ocopyright	© #copyright	® #oright	® #void1
™ #trademark	™ #void3	Å #AA	å #aa
× #times	÷ #divide	± #pm	/ #/
• #bullet	° #circ	… #3dots	· #upoint
f #voidb	∞ #infty	∇ #nabla	∂ #partial
” #doublequote	∠ #angle	↙ #downleftarrow	⋈ #corner
#lbar	#cbar	— #topbar	{ #ltbar
\ #arcbottom	(#arctop	#arcbar	#bottombar
↓ #downarrow	← #leftarrow	↑ #uparrow	→ #rightarrow
↔ #leftrightarrow	⊗ #otimes	⊕ #oplus	√ #surd
⇓ #Downarrow	⇐ #Leftarrow	⇑ #Uparrow	⇒ #Rightarrow
⇔ #Leftrightarrow	∏ #prod	∑ #sum	∫ #int
#void8	□ #Box	⊥ #perp	⊙ #odot
h̄ #hbar	#parallel		

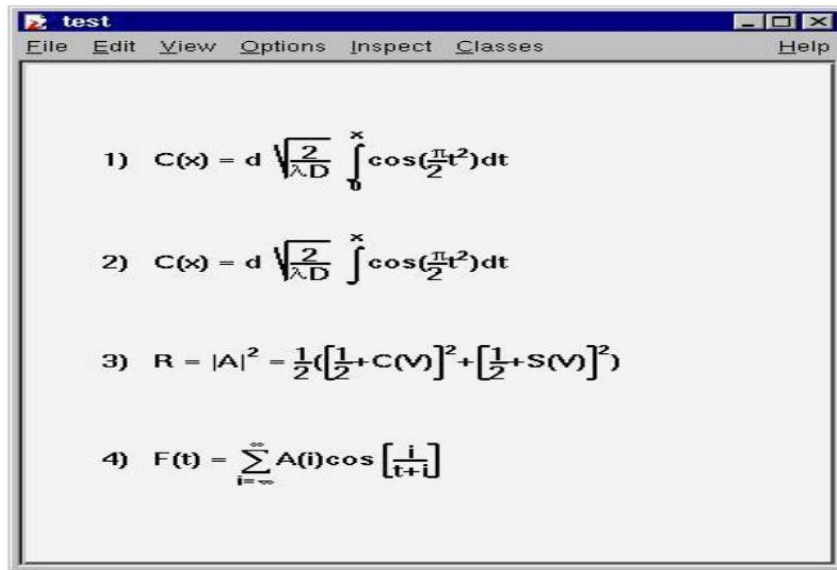
Lower case		Upper case		Variations
alpha :	α	Alpha :	A	
beta :	β	Beta :	B	
gamma :	γ	Gamma :	Γ	
delta :	δ	Delta :	Δ	
epsilon :	ϵ	Epsilon :	E	varepsilon : ε
zeta :	ζ	Zeta :	Z	
eta :	η	Eta :	H	
theta :	θ	Theta :	Θ	vartheta : ϑ
iota :	ι	Iota :	I	
kappa :	κ	Kappa :	K	
lambda :	λ	Lambda :	Λ	
mu :	μ	Mu :	M	
nu :	ν	Nu :	N	
xi :	ξ	Xi :	Ξ	
omicron :	o	Omicron :	O	
pi :	π	Pi :	Π	
rho :	ρ	Rho :	P	
sigma :	σ	Sigma :	Σ	varsigma : ς
tau :	τ	Tau :	T	
upsilon :	υ	Upsilon :	Y	varUpsilon : Υ
phi :	ϕ	Phi :	Φ	varphi : φ
chi :	χ	Chi :	X	
psi :	Ψ	Psi :	Ψ	
omega :	ω	Omega :	Ω	varomega : ϖ

Latex grammar

(1)

```
{
  gROOT->Reset();
  TCanvas c1("c1","Latex",600,700);
  TLatex l;
  l.SetTextAlign(12);
  l.SetTextSize(0.04);

  l.DrawLatex(0.1,0.8,"1) C(x) = d #sqrt{#frac{2}{#lambdaD}}
    #int^{x}_{0}cos(#frac{#pi}{2}t^{2})dt");
  l.DrawLatex(0.1,0.6,"2) C(x) = d #sqrt{#frac{2}{#lambdaD}}
    #int^{x}cos(#frac{#pi}{2}t^{2})dt");
  l.DrawLatex(0.1,0.4,"3) R = |A|^{2} =
    #frac{1}{2} (#[] {#frac{1}{2}+C(V)}^{2}+
    #[] {#frac{1}{2}+S(V)}^{2})");
  l.DrawLatex(0.1,0.2,"4) F(t) = #sum_{i=
    -#infty}^{#infty}A(i)cos#[] {#frac{i}{t+i}}");
}
```



1) $C(x) = d \sqrt{\frac{2}{\lambda D}} \int_0^x \cos(\frac{\pi t^2}{2}) dt$

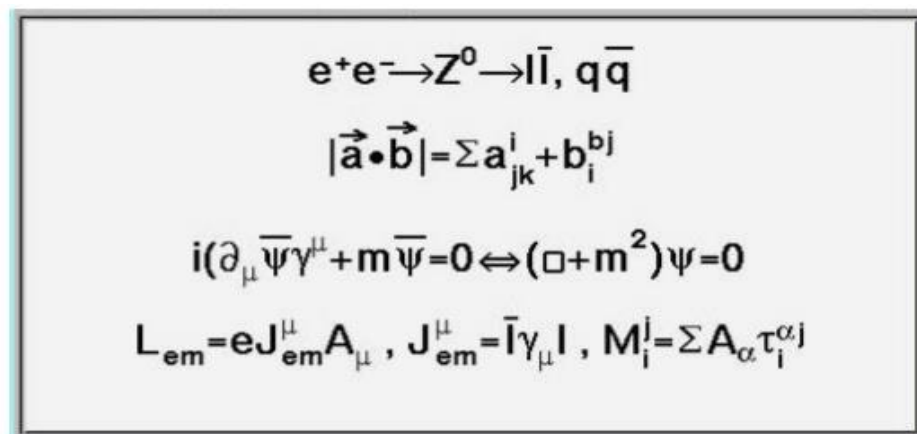
2) $C(x) = d \sqrt{\frac{2}{\lambda D}} \int_0^x \cos(\frac{\pi t^2}{2}) dt$

3) $R = |A|^2 = \frac{1}{2}([\frac{1}{2} + C(M)]^2 + [\frac{1}{2} + S(M)]^2)$

4) $F(t) = \sum_{i=-\infty}^{\infty} A(i) \cos \left[\frac{i}{t+i} \right]$

(2)

```
{
gROOT->Reset();
TCanvas c1("c1","Latex",600,700);
TLatex l;
l.SetTextAlign(23);
l.SetTextSize(0.1);
l.DrawLatex(0.5,0.95,"e^{+}e^{-}#rightarrow Z^{0}
#rightarrow I#bar{I}, q#bar{q}");
l.DrawLatex(0.5,0.75,"|#vec{a}#bullet#vec{b}|=
#Sigma a^{i}_{jk}+b^{bj}_{i}");
l.DrawLatex(0.5,0.5,"i(#partial_{#mu}#bar{#psi}#gamma^{#mu}
+m#bar{#psi})=0
#Leftrightarrow(#Box+m^{2})#psi=0");
l.DrawLatex(0.5,0.3,"L_{em}=eJ^{\mu}_{em}A_{\mu}, J^{\mu}_{em}=\bar{I}\gamma_{\mu}I, M^j_i=\Sigma A_{\alpha}\tau^{\alpha}_i");
}
```



$e^+e^- \rightarrow Z^0 \rightarrow l\bar{l}, q\bar{q}$

$|\vec{a} \bullet \vec{b}| = \sum a^i_{jk} + b^{bj}_i$

$i(\partial_{\mu} \bar{\psi} \gamma^{\mu} + m \bar{\psi}) = 0 \Leftrightarrow (\square + m^2) \psi = 0$

$L_{em} = e J^{\mu}_{em} A_{\mu}, J^{\mu}_{em} = \bar{I} \gamma_{\mu} I, M^j_i = \sum A_{\alpha} \tau^{\alpha}_i$

(3)

```
{
  gROOT->Reset();
  TCanvas c1("c1");
  TPaveText pt(.1,.5,.9,.9);
  pt.AddText("#frac{2s}{#pi#alpha^{2}}
             #frac{d#sigma}{dcos#theta} (e^{+}e^{-}
             #rightarrow f#bar{f}) = ");
  pt.AddText("#left| #frac{1}{1 - #Delta#alpha} #right|^{2}
             (1+cos^{2}#theta");
  pt.AddText("+ 4 Re #left{ #frac{2}{1 - #Delta#alpha} #chi(s)
             #[]{#hat{g}_{#nu}^{e}#hat{g}_{#nu}^{f}
             (1 + cos^{2}#theta) + 2 #hat{g}_{a}^{e}
             #hat{g}_{a}^{f} cos#theta) } #right}");
  pt.SetLabel("Born equation");
  pt.Draw();
}
```

Born equation

$$\frac{2s}{\pi\alpha^2} \frac{d\sigma}{dcos\theta} (e^+e^- \rightarrow f\bar{f}) = \left| \frac{1}{1-\Delta\alpha} \right|^2 (1+\cos^2\theta)$$

$$+ 4 \operatorname{Re} \left\{ \frac{2}{1-\Delta\alpha} \chi(s) \left[\widehat{g}_v^e \widehat{g}_v^f (1 + \cos^2\theta) + 2 \widehat{g}_a^e \widehat{g}_a^f \cos\theta \right] \right\}$$

$$+ 16 |\chi(s)|^2 \left[(\widehat{g}_a^e{}^2 + \widehat{g}_v^e{}^2) (\widehat{g}_a^f{}^2 + \widehat{g}_v^f{}^2) (1 + \cos^2\theta) + 8 \widehat{g}_a^e \widehat{g}_a^f \widehat{g}_v^e \widehat{g}_v^f \cos\theta \right]$$

Mouse operation

```
enum EEventType {
    kNoEvent          = 0,
    kButton1Down      = 1, kButton2Down      = 2, kButton3Down      = 3,
    kKeyDown          = 4,
    kWheelUp          = 5, kWheelDown        = 6, kButton1Shift     = 7,
    kButton1ShiftMotion = 8,
    kButton1Up        = 11, kButton2Up        = 12, kButton3Up        = 13,
    kKeyUp            = 14,
    kButton1Motion     = 21, kButton2Motion    = 22, kButton3Motion    = 23,
    kKeyPress          = 24,
    kButton1Locate     = 41, kButton2Locate    = 42, kButton3Locate    = 43,
    kESC               = 27,
```

附录

```
kMouseMotion    = 51, kMouseEnter    = 52, kMouseLeave    = 53,  
kButton1Double = 61, kButton2Double = 62, kButton3Double = 63  
};
```

其中: kButton1Down=1 左键按下; kButton1Up=11 左键抬起;

kButton3Down=3 右键按下;

kButton2Down=2 滑轮按下; kButton2Up=12 滑轮抬起;

kWheelUp = 5, kWheelDown = 6 代表滚动滑轮;

kMouseMotion=51 鼠标在 pad 上的移动;

kMouseLeave= 53 鼠标移出 pad;

kMouseEnter= 52 鼠标移入 pad;

kButton1Double = 61, kButton2Double = 62, kButton3Double = 63 代表双击;

kKeyPress = 24, 按下键盘;

```
enum EEditMode {  
    kPolyLine    = 1, kSPolyLine    = 2, kPolyGone    = 3,  
    kSPolyGone    = 4, kBox          = 5, kDelete      = 6,  
    kPad          = 7, kText         = 8, kEditor      = 9,  
    kExit         = 10, kPave        = 11, kPaveLabel   = 12,  
    kPaveText     = 13, kPavesText   = 14, kEllipse    = 15,  
    kArc          = 16, kLine        = 17, kArrow       = 18,  
    kGraph        = 19, kMarker      = 20, kPolyMarker  = 21,  
    kPolyLine3D   = 22, kWbox        = 23, kGaxis       = 24,  
    kF1           = 25, kF2          = 26, kF3          = 27,  
    kDiamond      = 28, kPolyMarker3D = 29, kButton     = 101,  
    kCutG         = 100, kCurlyLine  = 200, kCurlyArc   = 201  
};
```

生成 root 库的 makefile

(从 Qianxin 和 ZhiGang 的径迹重建程序中合并出来的):

```
#-----  
SrcSuf := cpp  
IncSuf := hh  
ObjSuf := o  
DepSuf := d  
DllSuf := so
```


附录

```
SrcDir := src
IncDir :=include
ObjDir := obj
DepDir := $(ObjDir)

#SRC = particle.cpp source.cpp trans.cpp react.cpp
SRC = $(wildcard $(SrcDir)/*.${SrcSuf})

INCDIRS = $(shell pwd)/include
EXTRAHDR =

# Name of your package.
# The shared library that will be built will get the name lib$(PACKAGE).so
PACKAGE = calibrate

LINKDEF = $(PACKAGE)_LinkDef.h

#-----
# Compile debug version
# export DEBUG = 2
# export VERBOSE = 1
export TESTCODE = 1
export I387MATH = 1
#export EXTRAWARN = 1

# Architecture to compile for
ARCH = linux
#ARCH = solarisCC5

ROOTCFLAGS := $(shell root-config --cflags)
ROOTLIBS := $(shell root-config --libs)
ROOTGLIBS := $(shell root-config --glibs)

INCLUDES = $(ROOTCFLAGS) $(addprefix -I, $(INCDIRS) ) -I$(shell pwd)

USERLIB = lib$(PACKAGE).${DlISuf}
USERDICT = $(PACKAGE)Dict

LIBS =
GLIBS =

ifeq ($(ARCH),solarisCC5)
# Solaris CC 5.0
CXX = CC
```

```
ifdef DEBUG
    CXXFLAGS      = -g
    LDFLAGS       = -g
    DEFINES       =
else
    CXXFLAGS      = -O3
    LDFLAGS       = -O3
    DEFINES       = -DNDEBUG
#   DEFINES      = -DNDEBUG
endif

DEFINES          += -DSUNVERS -DHAS_SSTREAM
CXXFLAGS         += -KPIC
LD               = CC
SOFLAGS          = -G
endif

ifeq ($(ARCH),linux)
# Linux with gcc (RedHat)
CXX              = g++
ifdef DEBUG
    CXXFLAGS      = -g -O0
    LDFLAGS       = -g -O0
    DEFINES       =
else
    CXXFLAGS      = -O3
    LDFLAGS       = -O3
    DEFINES       =
#   DEFINES      = -DNDEBUG
endif
DEFINES          += -DLINUXVERS -DHAS_SSTREAM
CXXFLAGS         += -Wall -Woverloaded-virtual -fPIC -fno-strict-aliasing
# added -fno-strict-aliasing to avoid warning during -O3 optimization

ifdef EXTRAWARN
CXXFLAGS         += -Wextra -Wno-missing-field-initializers
endif
LD               = g++
SOFLAGS          = -shared
ifdef I387MATH
CXXFLAGS         += -mfpmath=387
else
CXXFLAGS         += -march=pentium4 -mfpmath=sse
endif
endif
```

```

ifeq ($(CXX),)
$(error $(ARCH) invalid architecture)
endif

ifdef VERBOSE
DEFINES      += -DVERBOSE
endif
ifdef TESTCODE
DEFINES      += -DTESTCODE
endif

CXXFLAGS     += $(DEFINES) $(INCLUDES)
LIBS         += $(ROOTLIBS) $(SYSLIBS)
GLIBS        += $(ROOTGLIBS) $(SYSLIBS)

MAKEDEPEND   = gcc

ifdef PROFILE
CXXFLAGS     += -pg
LDFLAGS      += -pg
endif

#-----
OBJ           = $(patsubst $(SrcDir)/%.${SrcSuf},${ObjDir}/%.${ObjSuf},${SRC})
HDR           = $(wildcard $(IncDir)/*.${IncSuf}) $(EXTRAHDR)
DEP           = $(patsubst $(SrcDir)/%.${SrcSuf},${DepDir}/%.${DepSuf},${SRC})
OBJS         = $(OBJ) $(ObjDir)/$(USERDICT).${ObjSuf}

all:          $(USERLIB)

$(USERLIB):   $(OBJS)
              $(LD) $(LDFLAGS) $(SOFLAGS) -o $@ $(OBJS)
              @echo "$@ done"

$(SrcDir)/$(USERDICT).${SrcSuf}: $(HDR) $(LINKDEF)
              @echo "Generating dictionary $(USERDICT)..."
              $(ROOTSYS)/bin/rootcint -f $@ -c $(INCLUDES) $(DEFINES) $^

clean:
              rm -f $(OBJS) $(DEP) $(USERLIB) $(SrcDir)/$(USERDICT).${SrcSuf} \
                  $(SrcDir)/$(USERDICT).${IncSuf}

.PHONY: all clean

```

```
$(ObjDir)/%.${ObjSuf}:${SrcDir}/%.${SrcSuf}  
$(CXX) $(CXXFLAGS) -o $@ -c $<
```

```
# FIXME: this only works with gcc
```

```
%.${DepSuf}: %.${SrcSuf}  
    @echo Creating dependencies for $<  
    @$ (SHELL) -ec '$(MAKEDEPEND) -MM $(INCLUDES) -c $< \  
    | sed \"s%^.*\\.o%$*\\.o%g'\" \  
    | sed \"s%\\($*\\)\\.o[ :]*%\\1.o $@ : %g'\" > $@; \  
    [ -s $@ ] || rm -f $@'
```

```
###
```

```
-include $(DEP)
```

沿着 \vec{k} 方向得到张角为 θ 的随机矢量

```
//在头文件中定义如下变量
double Const_Dir[3]; //the random vector should coming out around this vector
double Const_Theta[2]; //the default range is [0,pi], but, you can change that
double Const_Phi[2]; // default range is [0,2pi]
double zp[3]; double xp[3]; double yp[3]; // this is the transfer matrix

//计算两个向量的叉乘
void Cal_Cross_Product(double* V1_tem, double* V2_tem, double* Results)
{
    Results[0] = V1_tem[1]*V2_tem[2]-V1_tem[2]*V2_tem[1];
    Results[1] = V1_tem[2]*V2_tem[0]-V1_tem[0]*V2_tem[2];
    Results[2] = V1_tem[0]*V2_tem[1]-V1_tem[1]*V2_tem[0];
}

//计算 $\vec{k}$ 的转移矩阵C
void Cal_Trans_Matrix(double* Const_Dir_tem)
{
    for(int i=0;i<3;i++){ Const_Dir[i] = Const_Dir_tem[i]; }
    double z[3] = {0,0,1}; double x[3] = {1,0,0}; double y[3] = {0,1,0};
    for(int i=0;i<3;i++) { xp[i] = 0; yp[i] = 0; zp[i] = 0; }
    for(int i=0;i<3;i++)
    { zp[i] = Const_Dir[i]/sqrt(Const_Dir[0]*Const_Dir[0] +
Const_Dir[1]*Const_Dir[1] + Const_Dir[2]*Const_Dir[2]); }
    if(zp[2]==1) { xp[0] = 1; yp[1] = 1; }
    else if(zp[2]==-1) { xp[0] = -1; yp[1] = -1; }
    else{
        Cal_Cross_Product(zp, z, xp);
        Cal_Cross_Product(zp, xp, yp);
    }
}

//将沿着z轴得到的随机矢量，变换到沿着k的随机矢量
void Get_Const_Theta_Dir(double* Random_Dir_tem)
{
    double phi_tem = gRandom->Uniform(Const_Phi[0], Const_Phi[1]);
    double theta_tem = fl_sin_theta->GetRandom(Const_Theta[0], Const_Theta[1]);
    double Dir_tem[3] = {0};
    Dir_tem[0] = Sin(theta_tem)*Cos(phi_tem);
```

附录

```
Dir_tem[1] = Sin(theta_tem)*Sin(phi_tem);  
Dir_tem[2] = Cos(theta_tem);  
for(int i=0;i<3;i++)  
{Random_Dir_tem[i] = Dir_tem[0]*xp[i]+Dir_tem[1]*yp[i]+Dir_tem[2]*zp[i]; }  
}
```

可以看到，最后这个函数中，就是把按 z 轴得到的随机变量变换到沿着 k 轴。

Think. Create. Share.



Tracking

版本 V1.0

利用在 BNL 值班的空闲时间，将自己这几年来使用 Linux, C++, Root 的一些备忘命令整理起来，方便自己以及他人使用。这些备忘命令中，很多都是从老师的课件，论坛，自己的多次试验中总结出来的。

王仁仲 2013 年 4 月 1 日 于 BNL

版本 V2.0

基于这几年来关于 Geant4 模拟的积累，最近写了一个 Geant4 模板程序，使用的人仅需简单了解 Geant4 的工作原理，就可以直接做一些有实际意义的模拟了。故，将备忘中关于 Geant4 基础知识加入进来，同时将这一年来的一些备忘增加进来。

王仁仲 2014 年 12 月 31 日 于清华园

版本 V3.0

看着要毕业了，就把自己以前想总结的尽量写进来，应该是最后一版了。

- (1) 把以前写的 makefile 文档集成进来；(V)
- (2) 对 Root 增加介绍性的描述，及框架介绍；
- (3) 对 Geant4 增加介绍性的描述，并把自己研一的备忘笔记整理进来；
- (4) 把近期对于各方面的东西整理进这个文档。

王仁仲 2015 年 4 月 15 日 于清华园

在此说明

本文档是我这五年关于程序方面的笔记，里面的内容绝大多数都是整理的，来自于论坛，课件，讨论，试验等。因本文档中内容的出处太多，这里不像写学术论文一样一一列出。如果想修改或是补充本文档，可以直接向我要 word 版本，Email: renzhong0611@mails.jlu.edu.cn。

王仁仲 2015 年 6 月 7 日 于清华园