МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
имени М. В. ЛОМОНОСОВА

НАУЧНО-ИССЛЕДОВАТЕЛЬСКИЙ ИНСТИТУТ
ЯДЕРНОЙ ФИЗИКИ имени Д. В.СКОБЕЛЬЦЫНА

В. В. Леонтьев, И. И. Белотелов

Задачи раздела «Информационные методы в физике высоких энергий»

Москва
«Университетская книга»
2011

УДК 004:[378.016:53]
ББК 74.58с51
Л47

Леонтьев В. В., Белотелов И. И.

Л47 Задачи раздела «Информационные методы в физике высоких энергий» : Описание задач практикумов / В. В. Леонтьев, И. И. Белотелов. — М. : Университетская книга, 2011. — 48 с.: табл., ил.

ISBN 978-5-91304-209-5

Практикум предназначен для студентов старших курсов и аспирантов физического факультета МГУ. Целью задач описываемого раздела является обучение навыкам работы в современной информационной среде, принятой в физике высоких энергий и адронной физике. Задачи были предложены и созданы выпускниками кафедры ФЭЧ, молодыми научными сотрудниками, работающими в лабораториях ОИЯИ, на основании собственного опыта работы.

УДК 004:[378.016:53]
ББК 74.58с51

ISBN 978-5-91304-209-5

Оглавление

Предисловие к пособию

Данное пособие содержит текстовые описания задач одного из трех разделов специального физического практикума кафедры физики элементарных частиц, предлагаемого студентам для прохождения в филиале НИИЯФ МГУ, г. Дубна. Описания должны использоваться студентами для самоподготовки перед занятиями, а также как конспект лекционных материалов, читаемых преподавателем во время занятий. Неотъемлемой частью этого курса являются ресурсы, размещенные на сайте кафедры ФЭЧ по адресу [1]. В тексте пособия повсеместно содержатся ссылки на наглядные материалы и коды программ-скриптов, необходимых для работы, без непосредственного обращения к этим *on-line* ресурсам изучение данного пособия неэффективно.

Unix-подобная операционная система

В современной экспериментальной физике высоких энергий и адронной физике новые научные результаты могут быть получены только на установках достаточно большого масштаба, на которых применяется либо коммерчески доступная либо промышленным способом изготовляемая аппаратура. Для таких установок характерна круглосуточная продолжительная работа многопользовательских ЭВМ, с большим количеством распределенных ресурсов, что предполагает широкое использование *Unix*-подобных ОС, в частности, *Linux*.

Освоение определенного минимума практических навыков работы в *Unix*-подобной среде является необходимым для начинающего научного сотрудника,

которому чаще всего предлагается принять участие в разработке и эксплуатации программной части экспериментального комплекса.

Регистрация и работа на удаленной *Linux/Unix*-машине

Понятие *xterm*. ПК-машины в компьютерном классе филиала НИИЯФ МГУ настроены таким образом, что при включении по умолчанию загружается программное обеспечение для работы в режиме удаленного терминала на сервере *sullen.msu.dubna.ru*, при этом все открывающиеся окна приложений запускаются на самом сервере, а ПК используется как консоль (устройство ввода-вывода). При этом используется стандартный эмулятор терминала для среды *X Window System* в *Unix* под названием *xterm*. Для начала работы студентам следует включить ПК, и в появившемся через некоторое время окне входа в систему ввести выданные заранее имена *login* и пароли, а после входа в систему открыть окно *xterm*, нажав значок на рабочем столе. По умолчанию у окна *xterm* отсутствует строка меню. Чтобы получить доступ к одному из трёх возможных меню настройки окна, пользователь должен, удерживая клавишу *Ctrl*, нажать левую, среднюю или правую клавишу мыши, в частности, полезно увеличить отображаемый в окне шрифт, выбрав опцию *font-Large.*

Понятие *ssh*. На данный момент, для того, чтобы получить доступ на удаленную *Unix/Linux*-машину для работы в режиме удаленного терминала (в частности, из уже открытого окна *xterm* получить доступ к другой машине в филиале НИИЯФ или во внешней сети), используется сервис под названием *ssh (**S**ecure **SH**ell)*. Этот сервис включает в себя программное обеспечение и правила

обмена информации (протоколы), которые позволяют регистрироваться на удаленной машине (*ssh* сервере), запускать на ней приложения с выведением отображения на машине пользователя (*ssh* клиенте) и т.д., причем вся обмениваемая информация зашифровывается и недоступна третьим лицам. Этим сервис ssh выгодно отличается от схожего устаревшего сервиса *telnet*, даже единичное использование которого из-за распространения программ-шпионов сейчас стало неприемлемо.

Итак, для того, чтобы зайти на другую машину по *ssh* протоколу, нужно в окне набрать команду: **`ssh -Y leon@lx.msu.dubna.ru`** , где *lx.msu.dubna.ru* имя сервера, *leon* имя пользователя, а *-Y* опция команды, которая дает разрешение запускать графические приложения на клиентской машине. На некоторых серверах опции *-Y* еще нет, в таких случаях приходится использовать предыдущую менее безопасную опцию *-X*. На сервере *lx* используются те же пароли и имена, что и на *sullen*, при работе внутри сети учреждения суффикс имени *msu.dubna.ru* можно опускать. Сервер *lx* использует то же дисковое пространство, что и *sullen*, однако, *sullen* работает под управлением ОС *FreeBSD*, а *lx* под управлением *Scientific Linux*, и у них разные ЦПУ и служебные файловые директории.

Работа с файловой системой

Для работы с файловой системой (переход в директорию, копирование, удаление и т.д.) в *Unix*-подобных ОС используются утилиты *shell*, специальной оболочки ОС. Эти утилиты можно вызвать, набрав их имена в командной строке, возможно, с нужными опциями вызова. Здесь мы лишь приведем список этих команд (**`ls, cd, mkdir, du, df, cd, pwd, mkdir, cp, mv, rm, rmdir`**), а краткое их

описание приведено на сайте [1] (Тема 2, ресурс «Подсказка по основным командам Linux»). Встроенную в ОС справку по командам (например команды *pwd*) можно также получить, набрав **`man pwd`**, или (в зависимости от настроек ОС) **`info pwd`**. Обратим внимание на несколько полезных аспектов этой темы:

`ls -lha` Если набрать не просто команду вывода списка файлов **`ls`**, а пополнить ее через пробел опцией **`-l`**, то будет выводиться еще и информация о размере файла, владельце и т.д. Если добавить опцию **`-h`** (например, можно и так: **`ls -l -h`**), то размер будет выводиться в более читаемых единицах (*k*- килобайт, *M*- мегабайт и т.п.). По умолчанию, часть файлов (например, у которых имя начинается с символа «**`.`**») командой **`ls`** не отображаются. Если же добавить опцию **`-a`** , то будут отображаться все доступные пользователю файлы и ресурсы.

Команды оценки занимаемого директорией дискового пространства **`du`** и доступного пользователю дискового пространства **`df`** также полезно дополнять опцией **`-h`**.

Если команду удаления **`rm`** дополнить опциями **`-rf`** , то эта команда будет выполняться, во-первых, рекурсивно (затрагивать все вложенные файлы и директории), во-вторых удалять без дополнительных подтверждений вообще всё, что указано, и файлы, и директории. Будьте осторожны с этой командой! *Unix*-подобные ОС изначально рассчитаны на многопользовательский режим, поэтому они сразу очищают дисковое пространство, в отличие от ОС *Windows*!

Разграничение доступа к ресурсам. Все ресурсы в *Unix*-подобной ОС (в частности, файлы) обязательно имеют владельца и список возможных разрешенных действий над ним, которые могут проделывать пользователи, разделяемые

по разрешениям на 3 категории (*user* - владелец этого файла, *group* -пользователи то же группы, что и владелец, *others* – все остальные пользователи). При помощи команды **`ls -l`** эти разрешения можно посмотреть, а при помощи команды **`chmod`** изменить. Пример использования: **`chmod g-r filename`** – удаление (символ **`-`**) у пользователей категории **`others`** права читать (**`r`**, т.е. *read*) файл **`filename`**.

Если пользователь является членом нескольких групп *group*, то он может сменить группу у своего файла или директории на другую доступную при помощи команды **`chgrp`** . Пример: **`chgrp printer PrintDirectory -R`** - рекурсивно (со всеми вложенными ресурсами) сменить для директории **`PrintDirectory`** группу с текущей на **`printer`**. После этого к данной директории можно, например, открыть доступ всем пользователям, имеющим право работать с принтером (входящим в группу **`printer`**).

Таким образом, развитая система разграничения и делегирования прав доступа в *Unix*-подобных ОС серьезно снижает вирусную уязвимость ОС.

Работа с текстовой информацией

В этом разделе практикума мы будем плотно работать с *emacs*, популярным в *Unix*-подобных ОС многооконным редактором текстовых файлов, его вызов: **`emacs`** или **`xemacs.`** Помимо редакторов, полезно использовать следующие утилиты, работающие с текстовой информацией:

`cat` - утилита, выводящая последовательно указанные файлы, объединяя их в единый поток. Если вместо имени файла указывается символ «-», то читается стандартный ввод (клавиатура), например, команда **`cat a.txt - b.txt > abc.txt`** помещает в файл **`abc.txt`**

(при необходимости создает его) содержимое сначала файла **a.txt**, потом ту информацию, которую пользователь наберет на клавиатуре (и завершит ввод нажатием клавиш *Ctrl+C*), а затем содержимое файла **b.txt**. Эта несложная, легко формализуемая утилита часто используется сотрудниками для автоматизации текстового вывода информации - например, для записи логов измерений.

less - консольная утилита для просмотра (но не изменения) содержимого текстовых файлов на экране, с возможностью прокрутки. В отличие от редакторов, которые также можно использовать для просмотра файлов, *less* не нуждается в чтении всего файла перед стартом и поэтому гораздо быстрее работает с большими файлами. Пример: **less abc.txt**

tail - выводит несколько (по умолчанию 10) последних строк из файла. Пример: **tail -n 4 abc.txt** .Ключ **-n <количество строк>** (или просто **-<количество строк>**) позволяет изменить количество выводимых строк. При использовании специального ключа **-f** утилита следит за файлом в режиме реального времени: новые строки, добавляемые в конец файла другим процессом (например, стоит потренироваться с *emacs*) автоматически выводятся на экран, или куда указано. Это особенно удобно для слежения за логами эксперимента.

cut – утилита для вывода выбранной части каждой строки файла. Пример: **cut -b 1-3 abc.txt** (вывод с первого по третий байт).

grep – поиск строки с регулярным выражением, не только в файле, но, и например, в стандартном текстовом выводе. Примеры: **grep -n expression filename** (поиск в файле **filename** и вывод с номером строки), **grep -n**

expression path -R (рекурсивный поиск в указанном месте *path* дискового пространства)

sed – этот редактор, в отличие от *emacs* или *vi*, не является интерактивным, не требует непосредственного участия пользователя, поэтому полезен для автоматизации текстовой обработки файлов (для больших файлов, или для часто повторяющихся операций). Пример: **sed -e 's/oldstuff/newstuff/g' inputFile > outputFile** ищет в файле **inputFile** все (опция **g**) выражения **oldstuff**, заменяет их (опция **s**) на выражения **newstuff**, а затем полученный текст сохраняет в файл **outputFile** .

Поиск файлов

Для поиска поиска ресурсов дискового пространства ОС можно использовать несколько разных команд:

find - утилита непосредственного поиска. Может производить поиск в одной или нескольких директориях с использованием критериев, заданных пользователем. По умолчанию, **find** возвращает все файлы после текущего в рабочей директории. Например, **find path -name file** производит поиск ресурса по его имени, при опции **-type** по его типу (файл, директория, линк) **-newer** по дате модификации).

Замечание. Для этой и для других команд поддерживаются т.н. «регулярные выражения» (например, выражение **abc*** означает, что команда касается всех файлов, имена которых начинаются с **abc**). Описание регулярных выражений также есть на сайте [1](*Тема2, ресурс «Подсказка по регулярным выражениям»*), однако, оно действительно не для всех ОС.

locate - утилита быстрого поиска в дисковом

пространстве по его индексированной базе данных (БД). Утилита выводит имя файла или содержимое папки с именем, включающем выражения для поиска. Следует учесть, что использование готовой БД сильно ускоряет поиск, однако, есть риск того, что эта БД несколько устарела, поэтому часть ресурсов может оказаться не найденной. Пример использования: **`locate filename | grep -n leon`** — выводит все ресурсы, содержащие в имени выражение «**`filename`**» , однако, не сразу, а после обработки результата поиска утилитой *grep*, которая оставляет только те строки, в которых содержится еще и выражение «**`leon`**» (фактически, только те ресурсы, которые принадлежат пользователю **`leon`**). Здесь уместно упомянуть о синтаксисе управления потоками данных при помощи объекта *pipes*.

Полезные детали

Pipes — управление потоками. При помощи нескольких лаконичных выражений пользователь в *shell* имеет широкие возможности получать информацию для используемых программ в автоматическом режиме (с других серверов, из распределенного дискового пространства) и передавать ее (на принтер, на специализированные носители и т.д.):

`|` - пример приведен только что выше, означает последовательную передачу от одной утилиты к другой.

`>` - пример использования этого символа приводился в описании утилиты *cat*, он предназначен для обозначения намерения пользователя отправлять полученную в результате работы программы информацию в выбранный текстовый файл.

< - то же самое, только теперь определяется файл, откуда информация забирается.

Приемы, ускоряющие ввод с клавиатуры. Последовательность команд, введенная пользователем с клавиатуры хранится ОС в служебной файле *.history* . Часто бывает полезно вызвать из памяти команду в командную строку, нажав нужное количество раз клавишу «стрелка вверх». Кроме того, текстовая информация в ОС на *sullen* копируется в буфер обмена просто при выделении текста мышью. Для того, чтобы затем вставить текст из буфера в необходимое место, нужно нажать среднюю кнопку (колесико) мыши.

Управление процессами в ОС Unix

Интерактивный режим. Для того, чтобы запустить исполняемый файл в *shell*, достаточно набрать в командной строке его имя, например: **`mozilla`** . Правда, из соображений безопасности во многих ОС сейчас требуется добавить еще два символа: **`./mozilla`** . При исполнении этого файла командная строка блокируется для других программ, и происходит взаимодействие только с работающей программой (процессом). Пользователь может, например, посылать текстовую информацию для процесса, или управлять им, например, нажатием сочетания клавиш *Ctrl+C* прекратить выполнение, или сочетанием *Ctrl+Z* (введя затем **`bg`**) отправить его в фоновый режим, когда командная строка освобождается.

Пакетный режим, или фоновый, можно выбрать также и сразу при старте программы, добавив символ *&* к команде запуска: **`mozilla & .`**

Команды управления процессами. Для практики

управления нам потребуется небольшая подготовка, нужно будет скачать в свою рабочую директорию на сервере *lx.msu.dubna.ru* простую *C++* программу бесконечного цикла **testUNLIM.cpp**, скомпилировать ее там (**g++ testUNLIM.cpp**), и полученный исполняемый файл запустить в фоновом режиме: **./a.out &** . Теперь можно попробовать выполнить следующие команды:

top — утилита для просмотра всех запущенных процессов, сортирует процессы по степени загрузки ЦПУ, запустить можно только интерактивно, прерывается нажатием *Ctrl+C* . Если хочется увидеть процессы только одного пользователя, можно запустить команду в связке с *grep*: **top | grep -n username**

Свои ошибочные, «зависшие» или просто ненужные процессы (например, запущенный **a.out**) пользователь может отследить при помощи команды **ps -ux**, выяснить его номер-идентификатор *uid*, а затем прекратить командой **kill** (пример: **kill 59011** , где 59011 - конкретный *uid*).

Как правило, процессы привязаны к тому окну *xterm*, из которого они были запущены (код можно посмотреть утилитой *ps*), и прекращают свою работу при закрытии этого окна. Если нужно, чтобы процесс не прекращался, например, при выходе пользователя, то запускать процесс нужно с опцией *nohup*. Пример: **nohup ./a.out &** .

Если пользователь запускает процесс надолго (часы, сутки), то признаком хорошего тона является заблаговременное снижение приоритета такого процесса, чтобы не создавать помех другим пользователям и процессам. Для этого используется опция *nice*. Пример: **nice nohup ./a.out &** При помощи утилит *top* или *ps* рекомендуется посмотреть влияние этих опций.

Оболочка shell, переменные shell

Оболочка *shell*,, является мощным, гибким и поистине необъятным инструментом для взаимодействия с *Unix*-подобными ОС. Существует несколько популярных версий *shell* (*cshell, tcshell, bash*), которые могут выбраны при создании записи каждого нового пользователя на *Unix*-машине. Помимо того, что *shell* включает в себя все утилиты, она хранит много служебной информации, в частности, настройки глобальных переменных пользователя. В этом разделе практикума еще будет практическая работа с *shell.* На данный момент уместно упомянуть, что, например, для того, чтобы посмотреть, что указано как переменная *$PATH*, содержащая имена директорий, в которых ОС по умолчанию будет искать для текущего пользователя исполняемые файлы (программы), нужно набрать в командной строке **`echo $PATH`** .

Задание

Доступ ко всем нужным материалам и к веб-форме для автоматизированной пересылки результатов преподавателю студент может получить, зарегистрировавшись в данном курсе на сайте [1] и зайдя в подраздел «*Задание: Unix-OC, Linux Beginner Quest*».

1.Выполните вход на *Unix/Linux*-машину (используя локальный терминал, команду *ssh* или программу *putty*).

2.Создайте директорию для выполнения теста и перейдите в нее (используйте *mkdir, cd*).

3.Загрузите с сайта архив с файлами задания *LBQ4.tgz* и распакуйте его в текущей директории (с помощью утилиты *tar,* **`tar -xzf LBQ4.tgz`**).

4.Запустите на исполнение скрипт *generateTree.sh* (команда **`./generateTree.sh`**). Скрипт выполняется в течение примерно минуты. В результате создается директория *LinuxBeginnerQuest*. Перейдите в нее и прочитайте файл *README*. Действуйте по инструкциям в этом и последующих найденных файлах.

5.На последнем шаге Вы получите строку, которую нужно будет прислать в качестве ответа, используя веб-форму.

Пакет ROOT, часть 1

Значительную часть работы научного сотрудника составляет преставление своих результатов в графическом виде. Поэтому мощный, специально приспособленный к особенностям физики высоких энергий, пользующийся преимуществами языка *C++* графический пакет *ROOT*[2] является востребованным инструментом и теоретиков, и экспериментаторов, несмотря на сохраняющиеся в нем недоделки и ошибки кода.

Настройка переменных окружения

Мы предполагаем, что *ROOT* уже установлен и Вы знаете его расположение. Например, на вычислительной *Linux*-ферме ОИЯИ (*lxpubXX.jinr.ru*) версии *ROOT* расположены в директории **`/usr/local/root/pro/`** , на ферме в ЦЕРНе (*lxplusXXX.ЦЕРН.ch*) *ROOT* в директории **`/afs/ЦЕРН.ch/cms/external/lcg/external/root/`** , на *Linux*-сервере филиала МГУ (*lx.msu.dubna.ru*) текущая версия находится в директории **`/opt/sw/root/pro`**

Сначала проверьте, может быть, ваша среда уже настроена для использования *ROOT* , просто попробуйте в командной строке ввести команду: **`root .`**

Если ваша система не находит *ROOT*, значит, нужно установить некоторые переменные окружения. Как это делается, зависит от версии *shell* , которую Вы используете. Определить версию можно, набрав команду **finger** (ниже пример использования):

```
host>finger belotel
Login: belotel              Name: Ivan Belotelov
Directory: /home/belotel    Shell: /bin/tcsh
host>
```

Для варианта **tcsh(csh)** отредактируйте ваш файл **.cshrc** , находящийся в вашей домашней директории (или создайте его, если он отсутствует). Добавьте туда следующие строки (приведён пример для *lx.msu.dubna.ru*):

```
setenv ROOTSYS /opt/sw/root/pro
setenv PATH ${ROOTSYS}/bin:${PATH}
setenv LD_LIBRARY_PATH ${ROOTSYS}/lib:$
{LD_LIBRARY_PATH}
```

Для **bash(sh)** отредактируйте (или создайте) файл **.bash_profile**, добавив туда следующие строки:

```
export ROOTSYS=/opt/sw/root/pro
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:
$ROOTSYS/lib
export PATH=$PATH:$ROOTSYS/bin
```

При следующем входе в систему только что отредактированные начальные скрипты выполнятся и переменные среды установятся нужным образом. Чтобы начать работать прямо сейчас, выполните команду **source .cshrc** либо **source .bash_profile .** Теперь попробуйте ещё раз запустить *ROOT*. Если всё в порядке, вы должны увидеть рисунок-заставку и строку приглашения. Теперь мы можем начать пользоваться пакетом *ROOT* .

Гистограммы

Гистограммы — это основной графический объект, возможности работы с которым предоставляет *ROOT*.

Гистограмма является графиком, предназначенным для отображения распределений количества физических событий по интересующим параметрам этих событий. Экспериментальные данные отображаются на гистограмме следующим образом. Допустим, было набрано 1000 событий и необходимо посмотреть, как распределились эти события по некоему одному параметру (например, по суммарной энергии зарегистрированных в единичном событии частиц). Тогда нужно создать гистограмму, в которой по оси *X* будет располагаться величина этой суммарной энергии, а то оси *Y* количество распределенных событий, такая гистограмма называется одномерной. При создании гистограммы необходимо определить, какова будет область определения этой энергии (например, от *0* до *5 ТэВ*) и на какое количество частей эта область будет разделена. Такие части называются бинами гистограммы; если, например, определено, что бинов будет 50, это означает, что на гистограмме будет отображено 50 экспериментальных точек. Если определить, что бины должны быть равной ширины, то первый бин будет со значения *0 ТэВ* и заканчиваться значением *0.1 ТэВ*, второй бин будет *0.1-0.2 ТэВ* и т.д. При необходимости можно создавать гистограммы и с разной шириной бинов. Итак, если из 1000 событий в диапазон первого бина попало, например, 9 штук, то первая отображаемая точка гистограммы будет иметь координаты (0.0, 9), если в диапазон второго бина попало 18 событий то координата второй точки (0.1, 18) и т.д.

Таким образом, гистограмма предназначена для работы со статистическими данными, и важным отличием её от других графиков является возможность заполнять её постепенно, по мере того, как новые статистические данные становятся доступны.

Одномерные гистограммы нужны прежде всего для количественной оценки фона и искомого эффекта, математического описания формы распределения эффектов, нахождения усредненных величин (в частности, нахождения массы частиц и резонансов). Поэтому скрупулезная работа с одномерными гистограммами необходима для получения конечных результатов экспериментов. В качестве примера этой важности, в частности, тщательной оценки уровня фона можно привести произошедшее несколько лет назад открытие и «закрытие» *пентакварка* [1] (*Тема «ROOT», ресурс «Выводы из истории поиска пентакварка»*)[3].

Начало практической работы с гистограммами

В любом текстовом редакторе создаём файл с C++ кодом по имени, например, *example1.cxx*. Работая на сервере *sullen*, удобнее всего воспользоваться редактором *emacs*:

```
host>emacs example1.cxx &
```

Код для этого скрипта, который будет создавать и отображать 3 гистограммы, можно скопировать с сайта по адресу [1] (*Тема «ROOT», ресурс «Практика работы с гистограммами»*). Поясним те процедуры, которые относятся собственно к классам *ROOT*:

```
TH1F* hist1 = new TH1F("h1","hist. 1",100, 0., 10);
```

- конструктор гистограммы с именем `h1`, с отображаемым заголовком (title) «`hist. 1`», количество бинов 100, область определения 0-10.

```
hist1->SetLineColor(kRed);
```
-настройка отображения лини гистограммы красным цветом.

```
TRandom* generator = new TRandom();
hist1->Fill(generator->Gaus(5, 0.5));
```
- создание генератора псевдослучайного числа и заполнение единичным событием с распределением Гаусса.

hist3->Add(hist2); - прибавление к **hist3** содержимого гистограммы **hist2**.

TCanvas * c = new TCanvas("c", "c", 300, 10, 400, 700); c->Divide(1,3); - создание канвы, т.е. окна, в котором графически будут отображаться гистограммы с разрешением *400x700 точек*. Разделение на три подканвы для отображения трех гистограмм.

hist1->Draw("same"); - отображение гистограммы поверх активного на данный момент графического объекта (опция **same**).

c->Print("c1.pdf"); - сохранение изображения в экспортном формате *pdf*. В зависимости от расширения имени файла, можно заказать сохранение в форматах *eps, gif, png* и т.д. Файл формата *pdf* затем можно посмотреть с помощью команды: **acroread имя_файла &**

Запускаем интерпретатор ROOT в командной строке: **root example1.cxx** . Или можно запустить *ROOT*, а уже потом из его собственной командной строки запустить и скрипт: **root [0].x example1.cxx**

Если всё сделано правильно, то отобразится окно (канва) с нашими гистограммами, которые можно сравнить с изображением в выше упомянутом ресурсе сайта. Сейчас можно попрактиковаться в настройке атрибутов гистограмм, кликая курсором на объекты (титулы гистограмм, заголовки осей) и выбирая соответствующие опции. Отметим, что наличие поясняющих подписей на осях является необходимым признаком профессионализма! Канву с настроенными объектами можно сохранить в файле формата *ROOT* или скрипта *C++*, если выбрать команду *File* в верхней строке канвы и затем соответствующую опцию. Сохраненные файлы можно загрузить и в новой сессии *ROOT* (выйти при помощи команды **.q** , запустить снова **root**). Для загрузки *ROOT* файла (*c1.root*) нужно

вызвать *ROOT* браузер командой **`TBrowser`**, а в нем найти активировать мышью сохраненный файл. *ROOT* файлы позволяют получить канву со всеми настройками. *C++* скрипт (*c1.C*) при своем запуске (команда **`.x c1.C`** в командной строке *ROOT*) отобразит канву не с предыдущими, а с текущими настройками сессии, но зато файл *C++* полезен тем, что в его коде можно посмотреть, как те настройки, что были сделаны вручную, можно сделать *C++* средствами, а затем использовать этот код в других скриптах для автоматизации настроек.

Чтобы сохранить не канву с гистограммами, а самостоятельные гистограммы, нужно открыть *ROOT* файл и записать туда гистограммы (или любые другие объекты):

```
TFile *rootFile = new Tfile("test.root",
"recreate"); hist1->Write(); hist2->Write();
rootFile->Close();
```

Чтобы уже в другом скрипте (или даже в новой сессии *ROOT*) просмотреть содержимое файла, а затем считать объекты из файла, можно применить следующие команды:

```
    TFile *rFile = new TFile("test.root");
rFile->ls(); TH1F* hst1 = rFile->Get("h1");
```

Работа с 2D гистограммами

Если одномерные гистограммы позволяют получить количественную оценку зависимости, то многомерные гистограммы применяются для качественного поиска корреляций. В качестве примера использования *2D*-гистограмм как ключевого экспериментального результата можно привести поиск 2011 года динамических эффектов кварк-глюонной плазмы коллаборацией CMS[1] (*Тема «ROOT», ресурс «Исследование корреляций на CMS, 2011»*)

[4]. Для того, чтобы нащупать корреляции, необходимо уметь настроить наиболее информативное отображение гистограммы, учитывая что двумерная гистограмма имеет уже 3 оси измерений (на осях *X* и *Y* значения параметров, по которым распределяются события, на оси *Z* количество событий). Проблема для создателя некой гистограммы даже не в том, чтобы увидеть искомую корреляцию самому, а в том, чтобы в её наличии убедились другие сотрудники, которые не просматривали эту гистограмму сотни раз в разных видах. Поэтому методика визуализации является довольно громоздким, но важным инструментарием для сотрудников в нашей области науки.

Отобразить на плоскости (на бумаге, на экране монитора) настоящее трехмерное изображение, естественно, невозможно, поэтому применяются либо проекции трех измерений наподобие изометрической (опции «**`surf`**» или «**`lego`**» метода **`Draw()`**), либо два измерения *X* и *Y* отображаются на плоскости, а отображение величин третьего измерения *Z* подменяется каким-нибудь достаточно информативным для человеческого глаза суррогатом (цветом области бина *XY*, или числом меток, помещаемых в эту область, и т.п.).

Отображение третьего измерения цветом

Для отображения шкалы *Z* цветом можно применить метод **`Draw("col")`** или **`Draw("colz")`**, во втором случае справа от гистограммы отображается столбец, информирующий, какой цвет соответствует какой величине. Это соответствие, называемое палитрой, по умолчанию настроено не самым наглядным образом в текущих версиях *ROOT*. Для отображения на экране или для цветной печати бывает полезно настроить палитру для разложения диапазона величин *Z* по цветам радуги

(**`gStyle->SetPalette(1)`**), что нагляднее, поскольку последовательность цветов в радуге многим привычна. Для подготовки гистограмм к черно-белой печати целесообразнее настроить палитру на отображение оттенков серого **`gStyle->SetPalette(8)`**, возможно число от 2 до 9.

Поскольку число отображаемых градаций оси *Z* получается ограниченным, для большей наглядности часто возникает нужда отобразить её в логарифмической шкале. В ручном режиме для этого нужно кликнуть в свободном поле нужной канвы и в новом меню выбрать опцию *SetLogz*.

После того, как корреляция найдена, её количественную оценку можно провести, сворачивая *2D* гистограмму в одномерные объекты. Для этого стоит потренироваться в работе с проекциями гистограммы вдоль осей (объекты *ProjectionX* и *ProjectionY,* [1] *Тема «ROOT», ресурс «Свертка 2D гистограмм»*). Для получения количественных характеристик эти свертки, как и другие одномерные гистограммы можно, например, *фитировать*.

Фитирование гистограмм

Для фитирования гистограмм, то есть для поиска математических функций, описывающих содержимое этих гистограмм, в *ROOT* используется метод ***TH1::Fit()***. Вот сигнатура этого метода и пояснение его параметров:

```
void Fit(const char *fname, Option_t * option,
OPtion_t * goption, Axis_t xxmin, Axis_t xxmax);
```

`fname` - имя фитирующей функции. Это должно быть имя одной из предопределённых в *ROOT* функций, или имя функции, определённой пользователем. Вот список имён предопределённых функций, которые могут использоваться в функции **`TH1::Fit() :`**

- **`gaus`** - гауссиан с 3-мя параметрами:
  *f(x)=p0*exp(-0.5*((x-p1)/p2)^2))*

- **expo** - экспонента с 2-мя параметрами: *f(x)=exp(p0+p1*x)*
- **polN** - полином степени *N*: *f(x)=p0+p1*x+p2*x^2+...*
- **landau** - функция Ландау с двумя параметрами.

option это опции настройки процесса фитирования, **goption** опции отображения, а **xxmin, xxmax** задают границы области фитирования.

Фитирование предопределённой функцией. Чтобы фитировать гистограмму предопределённой функцией, достаточно просто использовать функцию **Fit()** , передав ей в качестве первого аргумента имя функции: **hist.Fit("gaus");**. Для предопределённых функций нет необходимости устанавливать начальные значения параметров, это делается автоматически.

Фитирование пользовательской функцией. Чтобы отфитировать гистограмму своей функцией, нужно её создать (объект класса **TF1)** и вызвать **Fit()** , передав туда имя созданной функции. Есть 3 способа создать **TF1**: 1) Используя набор "C++ - подобных" выражений и операций, определённых для класса **TFormula** ; 2) То же что и п.1, но с параметрами ; 3) Используя вашу функцию (подпрограмму).

Пример создания *TF1* с помощью *TFormula*: **TF1 * f1 = new TF1("f1","sin(x)/x",0,10);** только что созданный объект **f1** класса **TF1** может быть использован для создания следующего объекта класса **TF1: TF1 * f2 = new TF1("f2","f2 * 2",0,10);**

Пример создания *TF1* с параметрами. Следующий способ - добавить параметры в выражение для функции: **TF1 * f1 = new TF1("f1","[0]*x*sin([1]*x)",0,10);** Номер параметра - в квадратных скобках. Чтобы задать начальные значения параметров явно, можно использовать функцию *SetParameter()*: **f1->SetParameter(0,10);** . Этим устанавливается начальное значение нулевого параметра

равным 10. Можно так же установить допустимый диапазон изменения параметров: **`f1->SetParLimits(0,2,10);`** . Этим задается диапазон изменения параметра номер 0 во время фитирования только в пределах от 2 до 10.

Создание *TF1* с помощью пользовательской функции.

Третий способ создать **`TF1`** - это определить функцию самостоятельно и передать её имя в конструктор **`TF1`**. Тип возвращаемого значения и список передаваемых параметров должны в точности совпадать со следующим примером: **`Double_t fitf(Double_t *x, Double_t *par);`** Здесь параметр **`Double_t *x`** это указатель на массив переменных; размерность массива равна числу переменных от которых зависит функция, для одномерной гистограммы используется только *x[0]*, для двумерной - *x[0]* и *x[1]* и так далее. А **`Double_t *par`** это указатель на массив параметров. Вот пример фитирования с использованием функции, определённой пользователем:

```
      double ourFitFunction( double *x , double *
par){
         return par[0]*exp(-par[1]*x[0])
+par[2]*exp(-pow((par[3]-x[0])/par[4],2));
       }
       TF1* ff = new TF1("ff",ourFitFunction, 1.0,
10.0, 5);
       ff->SetParameter(0,10);     // expo parameter
       ff->SetParameter(1,0.5);    // expo parameter
       ff->SetParameter(3,5);      // gaussian shift
       ff->SetParLimits(4,0.1,1); // gaussian sigma
       hist->Fit("ff","q0r");
```

Сначала определяем функцию **`ourFitFunction()`** - она зависит только от одной переменной (используется только **`x[0]`**) и 5-ти параметров (используются **`p[0]...p[4]`**). Затем создаём объект **`ff`** класса **`TF1`** с именем "**`ff`**", передаём ему указатель на **`ourFitFunction()`** , указываем диапазон функции (от 1.0 до 10.0) и число параметров (5).

Практический пример фитирования

Теперь нужно отфитировать все 3 гистограммы. Можно это сделать в режиме *on-line*, кликнув на линию нужной гистограммы правой кнопкой мыши и выбрав опцию *ViewFitPanel*. После настройки процесса фитирования (выбор функции, флагов настройки, области фитирования) и нажатия кнопки OK результаты фита отобразятся в виде линии на графике. Полученные параметры функции фитирования будут отображены графически в легенде гистограммы, если была включена опция *SetOptFit* (кликнуть на гистограмму правой кнопкой, выбрать опцию, в появившемся поле *0* заменить на *1*).

Можно этот процесс и автоматизировать. Для этого изменим скрипт, добавив туда код, доступный на сайте по адресу [1] (*Тема «ROOT», ресурс «Практика работы с гистограммами», параграф «Фитирование в скрипте»*). Ниже прокомментированы эти изменения:

`hist1->Fit("gaus","q0");` - фит гистограммы предопределённой функцией, причем заданные опции делают процесс фитирования проводимым без графического отображения линии функции и без текстового вывода в окно командной строки *ROOT*.

`TF1* ff = new TF1("ff",ourFitFunction, 1, 10,5); ff->SetParameter(0,10); // expo parameter` - Создание *ROOT* функции для фитирования с областью определения от 1 до 10 и пятью параметрами и задание начального значения 10 нулевому параметру.

`hist1->GetFunction("gaus")->Draw("same");`

`hist1->GetFunction("gaus")->SetLineWidth(1);` - Нахождение объекта (функции «*gaus*») по его имени, и отображение его поверх гистограммы в нужный нам

момент. Настройка толщины линии отображаемой функции.

После запуска скрипта, полученное изображение гистограмм с линиями функций и со значениями их параметров можно сравнить с образцом на сайте.

Задание

Доступ ко всем нужным материалам и к форме для автоматизированной пересылки результатов преподавателю студент может получить, зарегистрировавшись в данном курсе на сайте [1] и зайдя в подраздел «*Задание: Пакет ROOT, часть 1. Фитирование гистограмм*».

1.Загрузите к себе в рабочую директорию файл *test.root*

2.Напишите скрипт, считывающий из файла гистограмму и определите параметры распределения, отфитировов ее суммой гауссиана и полиномом 1-ой степени.

3.Создайте отображение гистограммы и функции фита.

4.Отчет по заданию пришлите в виде графического файла с гистограммой и функцией с помощью веб-формы.

Пакет ROOT, часть 2

Графы

Графы – это графические объекты *ROOT*, созданные на двух осях *X* и *Y*, и содержащие *n* точек с координатами *x,y*. Классы графов: *TGraph, TGraphErrors, TGraphAsymmErrors*, и *TMultiGraph*. Графы удобны для быстрого построения экспериментальных зависимостей в том случае, если происходит не постепенный набор статистических данных (тогда используются гистограммы), а имеется готовый

набор экспериментальных точек.

Простой пример создания и отображения графа приведен на сайте по адресу [1] (Тема «*ROOT*, часть 2», *ресурс «Практика работы с графами»*). В целом код похож на создание гистограмм, прокомментируем несколько строк:
`Tgraph * gr1 = new Tgraph(n1,x1,y1);` - создание графа с числом точек `n1`, с *X* координатами из массива `x1`, с *Y* координатами из массива `y1`.
`gr1->SetTitle("Simple Graph #phi #pm 4 #mum");` - определение титула, отображаемого заголовка графа (для гистограмм тоже работает). В титуле, в отличие от имени, могут употребляться специальные символы, так, например, символ φ кодируется как `#phi` .
`gr1->Draw("AC*");}` - Команда на отображение с опциями: `A` (нарисовать сначала оси для графа), `C` (соединить точки графа гладкой «*continuous*» линией), `*` (отобразить точки графа с помощью символов *).

Замечания:

1)При объявлении переменной необязательно указывать ее тип (например, допускается просто `c1 = new TCanvas("c1","A Simple Graph canvas", 200, 10, 700, 500);`), но лучше не портить стиль.

2)В отличие от заполнения гистограмм, порядок указания координат точек важен для построения линий графика. Так, если для графика `gr1` точки будут указаны не в порядке возрастания, то могут возникнуть петли или другие артефакты, как проиллюстрировано в упомянутом ресурсе.

3)Если координаты точек явно указывать в скрипте отображения графов, то для каждого набора данных приходится держать свой скрипт. Более надежным путем в этом случае будет создание одного универсального скрипта, который будет считывать координаты из внешнего файла.

Построение графа на тех же осях, с отображение

погрешностей. Пример приведен далее на том же ресурсе. Для задания погрешностей точек графа используются еще два массива, **`ex`** и **`ey`** . Теперь вместо знака * точки графа будут отображаться маркерами (опция **`P`**). Стиль маркера настраивается командой **`gr2->SetMarkerStyle(21)`** . Нужно учесть, что поскольку граф будет нарисован в той же канве, что и предыдущий пример, то останутся неизменными отображаемые заголовок и диапазон осей координат.

Коллекции объектов TTree, итераторы

В физике высоких энергий измеряемая или моделируемая информация часто записывается в ROOT объекты по имени *TTree* (деревья). *ROOT* деревья получили такое название из-за своей ветвящейся иерархической структуры: есть глобальные единичные параметры описания, есть большее количество ветвей (*Branch*), количество локальных параметров-листьев (*Leaf*) также значительно увеличивается. Объекты *TTree* предназначены для описания большого количества однообразных единиц информации (физических событий). Выгоды использования *TTree* следующие: 1) Компактный размер *ROOT* файла (поскольку опускается многочисленные однообразные заголовки); 2) Разработанный удобный интерфейс работы с большим количеством параметров событий (заранее определена их древообразная структура), 3) Быстрое преобразование выражений из параметров (обработка выражений ускоряется потому, что *TTree* уже представляет из себя многоразмерную гистограмму, что позволяет быстро применять новые критерии поиска корреляций и получать научную информацию).

Практический пример работы с *TTree*

Для начала работы нужно скачать к себе с сайта [1]

(*Тема«ROOT»*) в рабочую директорию *ASCII* файл по имени *cernstaff.dat*, из которого будут браться данные. Этот файл содержит в себе некоторые статистические сведения о сотрудниках *ЦЕРН* за 1988 год. Эти данные будет обрабатывать *ROOT* скрипт, пример кода которого доступен по адресу [1] (*Тема«ROOT», ресурс «Практика работы с TTree объектами»*). Скрипт создает *C++* структуру по имени *staff_t*, которая содержит несколько целочисленных и несколько текстовых членов, заполняет ее из файла *cernstaff.dat*, записывает ее в *ROOT* дерево, а само дерево сохраняет в файл *ROOT* формата для дальнейшего использования. Прокомментируем *ROOT* процедуры:

`TTree *tree = new Ttree("T","bla-bla");` - создание дерева по имени *T*

`tree-> Branch("staff", &staff.Category, "Category/I:Flag:Age:Service:Children:Grade:Step:Hr week:Cost");` - создание «ветви» дерева по имени *Category*, которая будет содержать целочисленные листки «Флаг», «Возраст», «Стаж», «Кол-во детей», «Степень», «Step», «Кол-во часов работы в неделю», «Зарплата».

`tree->Branch("Division",staff.Division, "Division/C");` - создание «ветви» дерева, которая будет содержать один текстовый листок «Подразделение».

`tree->Branch("Nation",staff.Nation,"Nation/C");` - создание «ветви», которая будет содержать единственный текстовый листочек «Гражданство»

`tree->Print();` - отображение текстовой информации о дереве в пользовательском окне *ROOT*

`tree->Write();` - запись дерева в *ROOT* файл.

После того, как созданный скрипт создал *TTree*, можно, например, посмотреть текстовую информацию

одной записи дерева, например, десятую: **`tree->Show(10)`** или **`T->Show(10)`** . Для работы с деревьями в *ROOT* есть специальное средство визуализации (*TreeViewer*). Пример его запуска в командной строке *ROOT,* при этом выполняется поиск объекта с именем *«T»*:

```
root[] TFile f("staff.root")
root[] T->StartViewer()
```

Замечание: Каждый создаваемый *ROOT* объект должен иметь уникальное имя. Это очень удобно, поскольку в *ROOT* разработан развитый механизм поиска и обращения к объектам по имени, что позволяет использовать объекты в разное время и разными программами, не утруждаясь созданием указателей и т.д.

Возможен запуск просмотрщика и без загрузки дерева, например, из командной строки при помощи двух команд **`gSystem->Load("libTreeViewer.so")`** , а затем **`new TtreeViewer()`** , или из окна *TBrowser* (нужно в рабочей директории найти *ROOT* файл, зайти в него, кликнуть на дерево *T* правой кнопкой мыши и в появившемся меню выбрать опцию *StartViewer*).

При помощи просмотрщика можно быстро построить одномерную гистограмму по интересующему параметру (перетащить в окне мышью, например, листок *stuff.Service* на ось *X* и нажать крайнюю левую кнопку «*Draw current selection*»). Также просто построить *2D* и *3D* гистограммы (перетащить на ось *Y* листок *stuff.Age*, в окне *Option* написать **`box`**). Также быстро можно отображать не просто параметры, а выражения из них (например, для того, чтобы построить гистрамму распределения «Годы, проведенные вне ЦЕРНа», нужно кликнуть на ось *X*, выбрать опцию *EditExression*, в поле *Expression* вместо **`stuff.Service`** написать **`stuff.Service-stuff.Service`**, а в поле *Alias*

какое-нибудь новое имя, например, **`~staff.outCERN`**).

Кроме того, можно наложить какое-нибудь условие отбора, например, можно отобразить предыдущее распределение только для людей, которые получают более 10000 франков в месяц, если перетащить листок *stuff.Cost* в пункт «ножницы», а там отредактировать выражение на **`stuff.Cost>10000`** (и сменить *Alias*). Как уже упоминалось, опции отображения можно задать в окне *Option* (**`lego`**).

Задание

Доступ ко всем нужным материалам и к форме для автоматизированной пересылки результатов преподавателю студент может получить, зарегистрировавшись в данном курсе на сайте [1] и зайдя в подраздел «*Задание: Пакет ROOT, часть 2. Построение графов и деревьев*».

1.Построить график из упражнения 5.2 задачи «Эффект Мёссбауэра». (Зависимость положения линий спектра скорости V_i в *мм/с* от эффективного g-фактора g_i.) Если задача «Эффект Мёссбауэра» ещё не выполнена, то вариант набора экспериментальных точек можно взять в файле *Moss_var.txt*. Профитировать зависимость линией (*pol1*). Линейность графика свидетельствует о равноускоренном движении вибратора.

2.Построить гистограмму из *TTree cernstaff.dat* "Зависимость зарплаты сотрудников (в тысячах франков) от их возраста при условии, что они работают 40 часов в неделю".

3.Прислать графический файл, содержащий канву с этими изображениями, используя веб- форму задания на сайте. Все обозначения на осях и заголовки гистограмм должны

быть оформлены надлежащим образом.

Пакет PYTHIA

PYTHIA это программа для генерации событий физики высоких энергий, т.е., для описания столкновений таких высокоэнергетических элементарных частиц, как электрон, позитрон, протон и антипротон в различных комбинациях. Информация для моделирования взята по большей части из собственных исследований в *ЦЕРН*, однако, много формул и другой информации почерпнуто из литературы.

С 1997 года по нынешнее время использовалась версия этого Монте-Карло генератора, написанная в *FORTRAN77* (текущая версия 6.4). Сейчас программа переписана в *C++* (версия 8.1), однако, до тех пор, пока не осуществлен перевод всех возможностей, обе версии используются и поддерживаются одновременно.

Назначение генераторов физических событий

- Дают физикам представление о типе событий, которые они надеются увидеть, и об их скорости набора.
- Помогают в планировании новых детекторных установок, то есть, оптимизировать их характеристики для изучения интересующих сценариев физических событий в рамках существующих ограничений.
- Являются инструментом для проработки стратегии анализа данных (оптимизации отношения “сигнал/шум”).
- Используются в качестве метода оценки коррекций на геометрические и кинематические ограничения области чувствительности (*acceptance*) детекторов.
- Используются в качестве удобной рабочей оболочки для интерпретации наблюдаемых феноменов в терминах Стандартной Модели.

Философия применения генераторов

Квантовая механика вносит концепцию случайности в поведение физических процессов. Достоинством генераторов событий (*event generators*) является то, что эта случайность может быть смоделирована при помощи метода Монте-Карло. Сущность метода заключается в том, что, во-первых подразумевается наличие генератора псевдослучайных чисел, т.е. функции, которая при вызове возвращает число R в пределах от 0 до 1, при этом распределение R является плоским, и с достаточной точностью значения R являются нескоррелированными. Затем эти значения R используется для розыгрыша сценария конкретного цикла события (выбор конкретного значения для различных известных распределений величин, выбор времени распада и т.п.) Разыграв статистически достаточное количество событий, мы можем построить интересующие нас распределения (например, диапазон энергий для продуктов интересующего механизма реакции).

Что касается упомянутых известных распределений, то, например, дифференциальное сечение реакции рассчитывается из кинематических соотношений при введении матричного элемента (известного, либо предложенного теоретиками исходя из перспективных моделей), для учета высших порядков КХД вводится значение дополнительного параметра (K-множителя).

Следует заметить, что при генерации значительного числа событий (миллионов), становится актуальной проблема скоррелированости псевдослучайных чисел, и вместо встроенных в *C++ Random* генераторов приходится применять специально созданные программы.

Структура программы Pythia

С точки зрения описания физики событий полная процедура генерации события разделяется на 3 стадии:

1.Генерация «процесса», который определяет природу события. Зачастую это могут быть «жесткие процессы», такие как $gg->h^0 \rightarrow ZZ \rightarrow m^+ m^- qq_{bar}$ (а также другие процессы), которые могут быть просчитаны в рамках теории малых возмущений.

2.Генерация всех подчиненных процессов на партонном уровне, включая гамма-излучение, многократное партонные взаимодействия и структуру непровзаимодействовавшего пучка. Такие феномены приблизительно описываются теорией малых возмущений, однако непертурбативные поправки уже существенны.

3.Адронизация этой партонной конфигурации (фрагментация струй, распады нестабильных частиц). Только феноменологическое описание.

Этим стадиям отвечают три класса *ProcessLevel*, *PartonLevel* и *HaronLevel*, соответственно. Классы: *Event* (члены класса *process* и *event*), *BeamParticles*, база данных *Settings*.

С точки зрения технического устройства, взаимодействия пользователя и генератора проявляется в трех фазах:

1.Инициализация, когда формулируется задание.

2.Генерация индивидуального события (цикл события).

3.Вывод окончательной статистики.

Программа содержит теорию и модели для ряда аспектов физики, включая так называемые мягкие и жесткие взаимодействия, распределения партонов, партонные струи начального и конечного состояний, многократные партонные взаимодействия, фрагментацию и распады.

Встроенные *C++* методы программы обеспечивают доступ к информации как об отдельной частице либо процессе на любом этапе розыгрыша события, так и о событии в целом. Встроенные средства вывода позволяют получить статистическую информацию и гистограммы в виде *ASCII* кода (который можно сохранить в файл для дальнейшего использования).

Первая программа Pythia 8

Пакет Pythia является компактной (~5 Мб) независимой программой, поэтому несложно скачать и установить себе собственную локальную версию. Для этого архив установочный код (в 2011 году был **`8145.tgz`**) можно скачать с сайта разработчика [5] или с кафедрального сайта. Затем его нужно распаковать (команда **`tar -xzf 8145.tgz`**) и скомпилировать (команда **`make`** в распакованной корневой директории, занимает ~2 минут на *lx*).

Быстрее всего начать работу с обучающими и своими первыми скриптами в поддиректории *examples*. Для этого нужно перейти туда, и добавление в служебный файл *Makefile* имени нового модуля, например, '*Pyth_hello*', который предлагается сейчас создать. Для этого нужно открыть в *emacs* файл с таким названием и скопировать туда код, выложенный на сайте [1] (*Тема «Пакет Pythia», ресурс «Практика работы с PYTHIA 8.1»*). Данный генератор моделирует столкновение барионов на *LHC* с рождением топ-кварков. Для того, чтобы детально понимать работу скрипта и подготовиться к выполнению задания рекомендуется ознакомиться с руководством разработчиков [1] (*Тема «Пакет Pythia», ресурс «A Brief Introduction to Pythia 8.1»*). Там описываются возможности по настройке свойств процессов и получения информации (например, можно сделать бозон Z^0 стабильной частицей инструкцией

pythia.readString("23:onMode=off")). В этом пособии приведем лишь комментарии тех лаконичных инструкций, которые были применены в скрипте для моделирования такого объёмного процесса:

Pythia pythia; - создание объекта класса Pythia, обращение к методам которого обеспечивает настройку и запуск генератора событий

pythia.readString("Top:gg2ttbar = on"); - включение одного процесса реакции, а именно перехода двух глюонов в *t* и анти-*t* кварки

pythia.readString("PhaseSpace:pTHatMin = 20."); - команда настройки процесса, установка минимального значения поперечного импульса в 20 *ГэВ/с*

pythia.init(2212,2212,14000.); - установка реакции как столкновение протонов с энергией 14 *ТэВ/с* в ц.м.

pythia.next(); - генерация единичного события

pythia.event.list(); - вывод текстовой информации об этом событии, в частности, о каждой рожденной частице, о её типе, энергии, о её родителе и т.д.

pythia.statistics(); - вывод текстовой информации о работе генератора (статистика ограничений , сбоев и т.д.).

Скомпилируем данный модуль (**make Pyth_hello**). Теперь его можно запустить в директории *examples* при помощи команды **./Pyth_hello.exe** или **./Pyth_hello.exe > out.txt** (с выводом текстовой информации в файл). Перед выполнением последующего задания следует изучить работу созданного генератора и получаемые результаты. Помимо вывода текстовой информации, в *Pythia* можно также создавать гистограммы, отображать их в псевдографике и экспортировать, как это предлагается в задании. Нужно при этом иметь в виду, что *Pythia 8.1* гистограммы не могут иметь более 100 бинов, поэтому

часто для большей гибкости имеет смысл экспортировать сами данные, а не готовые гистограммы.

Задание

Доступ ко всем нужным материалам и к форме для автоматизированной пересылки результатов преподавателю студент может получить, зарегистрировавшись в данном курсе на сайте [1] и зайдя в подраздел «*Задание: Пакет Pythia, Моделирование рождения Z^0 бозона на LEP*».

1.Взять за основу файл *Pyth_hello.cc* со описанными ниже последовательными изменениями.

2.Столкновение протонов (идентификатор 2212) заменить на столкновение e^+e^- при энергии (с.ц.м.) 91.2 ГэВ (посмотреть идентификаторы в приведенных на сайте кодах частиц *PDG*)

3.Изучаемый процесс заменить на **`WeakSingleBoson:ffbar2gmZ = on`** (взаимодействие двух фермионов с образованием любой квантово-механической комбинации с гамма-квантом и Z^0 обменом)
4.Вместо 20 ГэВ-ного ограничения на переданный импульс ("**`PhaseSpace:pTHatMin = 20.`**") выключить процессы с испусканием гамма квантов исходными e^+e^- (обычно экспериментальные данные корректируются на эту поправку) **`pythia.readString("PDF:lepton = off");`**
5.В теле цикла разыгрываемого события сделать вывод информации о типе каждой частицы: **`for (int i1 = 0; i1 < pythia.event.size(); ++i1) {cout << "i= " << i1 << "\tid = " << pythia.event[i1].id() << endl; }`**
6.Переместить этот вывод только для заряженных частиц конечного состояния (методы **`isCharged()`** и **`isFinal()`** для объекта **`pythia.event[i1]`**)
7.Туда же поместить созданный счетчик числа заряженных

частиц в конечном состоянии **nChg**

8.В начале программы поместить создание гистограммы множественности заряда: **Hist histoChg("charged multipl", 100, -0.5 , 99.5);**

9.Заполнить гистограмму в теле цикла: **histoChg.fill(nChg);**

10.Вывести гистограмму на стандартный вывод **cout << histoChg;**

11.Воспользовавшись стандартной библиотекой *fstream*, создать указатель **fout** на отдельный ASCII файл. (**#include <fstream> ofstream fout("out_hist.txt");**)

12.Затем вывести гистограмму в этот файл: **histoChg.table(fout);**

13.Скомпилировать программу, посмотреть результат, при необходимости провести отладку.

14.По образцу Гистограммы 1 для определенного типа частиц конечного состояния (по выбору преподавателя, K^-, или K^+, или π^-, или π^+) создать еще две гистограммы:

Гистограмма 2. распределение по энергии (метод **e()**) ;
Гистограмма 3. одной из проекций импульса (**px(), py(), pz()**) либо его модуля (**pAbs()**).

15.После того, как работа программы отлажена на малом числе событий, можно приступать к моделированию большого числа событий (набор статистики). Перед этим для ускорения процесса стоит убрать/закомментировать вывод на *cout* и листинг информации о каждом событии.

16.Сгенерировать 10000 столкновений e^+e^-, и средствами *ROOT* отобразить полученные гистограммы. Для этого нужно написать скрипт, который будет содержать создание этих гистограмм в *ROOT*, чтение из внешнего файла и заполнение гистограммы, например: **TH1F* hist1=new TH1F("hist1","energy",100,0.,100.); ifstream fin1("outEnergy.txt"); Double_t bin, binContent;**

```
while(fin1>>bin>>binContent) hist1->Fill(bin,
binContent);
```

17.Подобрать правильный диапазон отображаемых величин и ширину бина. Обратить внимание на корректное обозначение единиц, названий величин на осях и т.п.

18.Полученные 3 гистограммы экспортировать в файл графического формата и послать в качестве ответа, используя веб-форму этого задания на сайте .

Пакет PLUTO

Программный пакет *Pluto++* представляет собой коллекцию *C++* классов, созданных в рамках пакета моделирования реакций адронной физики (физики промежуточных энергий). *Pluto* интерактивно запускается в среде *ROOT* и использует ресурсы этого пакета, в совокупности с *C++* библиотекой общего пользования *CLHEP*. *Pluto* используется как стандарт для создания и эксплуатации генераторов событий путем предоставления инструментов для генерации частиц и для последующих манипуляций с ними, с каналами реакций, с составными реакциями, а также для наложения так называемых экспериментальных фильтров для регистрации продуктов реакций детекторами (*acceptance*).

Типичные случаи моделирования могут быть реализованы в несколько строк введенного кода, причем от пользователя не требуется опыта эксперта. Получаемая информация может быть проанализирована на лету, либо в дальнейшем использована в программах типа *GEANT*.

Программный пакет *Pluto++* содержит в себе модели модели резонансов и распадов Далица, спектральные функции резонансов с масс-зависимой шириной, а также анизотропные угловые распределения для некоторых

каналов реакций. Интерфейс менеджера распадов делает возможными многоступенчатые расчеты («коктейль»). В доступе находится широкоохватная база данных по частицам с поддержкой более чем 999 типов частиц. В базу включены свойства частиц и моды распадов, также пользователь может вводить определение частиц своего собственного типа. Реализовано применение термальных распределений, что делает возможным моделирование многоадронных распадов для «горячих файрболлов».

Установка пакета Pluto++

Для локальной установки пакета *Pluto*++ требуется выполнить следующую последовательность действий:

1.Найти сайт: набрать в Гугле *Pluto*++, первая же ссылка приводит на сайт разработчиков [6].

2.Скачать современный отработанный релиз (*~450 kB*)

3.Распаковать в подходящее место в домашней директории

4.Посмотреть файл *README* (**`less README`**). Главная фраза **`Just type "make" - there is no "make install". The libPluto.so stays in the main path.`**

5.Скомпилировать (**`make`**) *(занимет несколько минут)*

6.Проверить подключение библиотек *PLUTO* в командной строке *ROOT*:

```
root[] gSystem->Load("libPhysics.so")
root[] gSystem->Load("libPluto.so")
```

Замечания:

1) Если запускать *ROOT* с подключением *Pluto*++ откуда-то из другой директории (например на уровень выше), то соответственно, нужно, указывать путь(**`"../libPluto.so"`**), или копировать библиотеку в место запуска.

2) Практически, подключение библиотеки лучше проводить автоматизированно (в той же Пифии нужно подключать 3 библиотеки), через файл *rootlogon.C* (см. далее).

Полезные детали

Использование файла *rootlogon.C*

Во время своего запуска *ROOT* проверяет наличие в текущей директории файла с именем *rootlogon.C* , и при нахождении запускает его в первую очередь. Файл *rootlogoff.C* запускается, соответственно, при остановке *ROOT*. Для подключения нужных *Pluto* библиотек (вместе с диагностикой успешности подключения) достаточно в новом или уже имеющемся файле *rootlogon.C* прописать следующий код:

```
{if (gSystem->Load("libPhysics.so")==0)
printf("Shared library libPhysics.so loaded\n");
      else printf("Unable to load libPhysics.so\n");
if (gSystem->Load("libPluto.so")==0)
   printf("Shared library libPluto.so loaded\n");
      else printf("Unable to load libPluto.so\n");
}
```

Помимо этого, в этот файл полезно прописать часто встречающиеся настройки, например, внести запуск браузера *TBrowser* вместе со стартом *ROOT*:

```
      TBrowser* b = new TBrowser();
```

а также модифицировать стиль отображения графических объектов (канвы, гистограмм):

```
      gROOT->SetStyle("Plain"); // set up plain
style, useful for printouts
      gStyle->SetOptStat(1111111); // switch on all
options of histogram stat.
```

Кроме того, зачастую бывает полезно не указывать библиотеку напрямую (может смениться версия или размещение источника), а указать на библиотеку через символьную ссылку, которую можно создать в *shell*:

Использование в *shell* символических ссылок

Вместо того, чтобы постоянно копировать нужные

библиотеки (в данном случае *Pluto*) в директории, где будут размещаться скрипты для моделирования и выходные результаты, можно создать и затем копировать всего лишь ссылки на эти библиотеки. Использование символических ссылок экономит место на диске и облегчает процесс обновления программного обеспечения *Pluto*.

Так, создать ссылку на *libPluto.so* можно следующей командой:

```
ln -s ~/v_prak/2009/PLUTO09/pluto_v5.22/libPluto.so
linkPluto.so
```

После чего ссылку **`linkPluto.so`** можно копировать и использовать как обычный файл, она будет сохранять свою связь с исходным файлом.

Первая программа Pluto++

Для начала практической работы с генератором *Pluto* предлагается создать в текстовом редакторе скрипт с расширением имени *.C* (например *pluto00.C*) и скопировать в него программный код первого скрипта, размещенного на сайте [1] (*Тема «Пакет PLUTO», Ресурс «Примеры скриптов»*). Поясним кратко те процедуры, которые относятся, собственно, к классам *Pluto*:

`Putils::SetSeed(0)` — выставляет определенный параметр для генератора псевдослучайного числа. Если мы хотим получить необходимое количество событий за несколько проходов, изменение значения этого параметра позволяет избежать возникновения повторяющихся последовательностей псевдослучайных чисел.

`PReaction my_reaction("3.5","p","p","p p eta [g dilepton [e+ e-]]","eta_dalitz")` — определяет моделируемую реакцию как процесс столкновения протона пучка с кинетической энергией 3.5 *ГэВ/с* с протоном мишени, в результате чего рождается *эта*-мезон с распадом

на *гамма*-квант и пару электрон -позитрон (дилептон).

`my_reaction.Do("#filter = 1; if theta_ep<18 || theta_ep>85 || theta_em<18 || theta_em>85; #filter = 0");` - определяет кинематический фильтр, значение которого обнуляется для события, в котором полярные углы электрона и/или позитрона меньше 18^o или больше 85^o.

`my_reaction.Do("opang = [e+]->Angle([e-])");` - расчет угла раскрытия трека e^+ относительно трека e^-.

`my_reaction.Output(ntuple);` - запись рассчитанных величин (**`opang`** и проекций импульса эта-мезона **`eta_px = [eta]->Px()`** и т.д.) .

`my_reaction.Do(histo1,"if opang > (9./180.) * TMath::Pi(); _x = ([e+] + [e-])->M()");` - заполнение гистограммы **`histo1`** значением массы дилептона.

`my_reaction.Loop(100000)` — повторение петли генерации события 10^5 раз.

Данный скрипт нужно запустить в командном интерпретаторе *ROOT*: **`.x pluto00.C`** , результатом работы будет отображение контрольной гистограммы и запись дерева выходных данных в файл *ROOT*-формата, в котором они хранятся в виде дерева *ntuple*.

Таким образом, всего лишь запустив несколько строк кода, можно обеспечить набор требуемой модельной информации. Однако, при усложнении моделируемой реакции или при надобности получить более детальную информацию о процессе генерации события (о каналах распада, о промежуточных частицах), необходимо воспользоваться методами *Pluto*, которые явным образом определяют промежуточные этапы генерации события. Для этого предлагается создать и отладить связку из двух скриптов доступных на сайте по тому же адресу [1] (*Тема «Пакет PLUTO», Ресурс «Примеры скриптов»*).

При помощи первого скрипта генерируется развал

двух дейтронов на два протона и нейтрона, причем протон из дейтрона мишени (в скрипте определен, как **p2**) указан в последовательности конечных частиц последним, что по стандартам *Pluto* означает, что именно он, в соответствии со *спектаторной* моделью дейтрона, он будет «спектатором» (наблюдателем), т.е. не будет взаимодействовать с частицами пучка, и в результате, он в конечном состоянии будет иметь энергию, которая у него была внутри дейтрона (т.н. энергия Ферми), несколько *МэВ/с* и изотропно распределенный вектор импульса. Второй скрипт обеспечивает считывание нужных нам данных из общего дерева промоделированных и сохраненных в *ROOT* файле событий, а также отображение нужных нам гистограмм.

На базе освоенных скриптов можно приступать к созданию собственного кода, необходимого для выполнения описанного ниже задания.

Задание

На сайте предлагается несколько вариантов, здесь приводится вариант «Моделирование рождения Дельта-изобары, спектаторная модель»:

1.Напишите скрипт реакции $p+d \rightarrow p+n+\Delta^+$, нейтрон должен быть «спектатором» (определение «спектатора» см. чуть выше в описании). Протон пучка с моментом *1.5 ГэВ/с*, идентификатор частицы Δ+ посмотреть на сайте.

2.Смоделируйте 50 тыс. событий.

3.Постройте гистограмму распределения массы Δ^+.

4.Постройте гистограмму распределения кинетической энергии Δ^+ изобары.

5.Постройте гистограмму распределения кинетической

энергии Δ^+ изобары при условии, что полярный угол протона θ регистрируется в пределах от 20 до 40 градусов.

6.Постройте двумерную гистограмму корреляции кинетической энергии нейтрона и его угла θ.

7.Графический файл, содержащий канву с 4 гистограммами и файлы с кодами скриптов пришлите в качестве ответа, используя форму [1] (*Тема «Пакет PLUTO», Ресурс «Задание, вариант 3»).*

Благодарности

Авторы выражают благодарность сотруднику ОИЯИ, выпускнику кафедры ФЭЧ Роману Салмину за определяющий вклад в создания кодов задания *LinuxBeginnerQuest*.

Ссылки

1.*http://hep.msu.dubna.ru (4 курс,Практикум/ Раздел «Информационные методы»)*. Область на официальном сайте кафедры физики элементарных частиц (г.Дубна), где расположены необходимые ресурсы для данного практикума.

2.*http://root.cern.ch* – официальный сайт разработчиков *ROOT*.

3.Burkert V.D. "Have pentaquark states been seen?", *arXiv:hep-ph/0510309v2* , Jefferson Laboratory, **2005**.

4.CMS Collaboration, "Observation of Long-Range Near-Side Angular Correlations in Proton-Proton Collisions at the LHC", arXiv:1009.4122v1 , CERN, **2011**.

5.*http://home.thep.lu.se/~torbjorn/Pythia.html* - официальный сайт разработчика ПО *Pythia*.

6.*http://www-hades.gsi.de* - официальный сайт разработчика ПО *Pluto++*.

Для заметок

Учебное издание

Владимир Викторович Леонтьев,
Иван Иванович Белотелов

ЗАДАЧИ РАЗДЕЛА «ИНФОРМАЦИОННЫЕ МЕТОДЫ В ФИЗИКЕ ВЫСОКИХ ЭНЕРГИЙ»

Описание задач практикумов

Работа поступила в ОНТИ 23.06.2011 г.

Формат 60×84 $^{1}/_{16}$. Бумага офсетная.
Печать цифровая. Тираж 50 экз. Заказ № Т-159-11.

Отпечатано с материалов, предоставленных автором, в типографии «КДУ».
Тел./факс (495) 939-44-91; www.kdu.ru; e-mail: press@kdu.ru