September, 1998

PHY307F/407F - Computational Physics
Background Material for Expt. 4 - Monte Carlo Simulation

David Harrison

INTRODUCTION

In Experiment 1 we investigated techniques to compare theoretical predictions with experimental data. This experiment extends that study to cases in which least-squares fits are not possible and/or appropriate. It concentrates on a method of generating synthetic data sets called *Monte Carlo simulation* (the name is after the casino).

This document is organised as follows:

I. Physics Background: needed background for the experiment.

  A. Review of special relativity.

  B. High energy physics.

II. Motivation: circumstances in which Monte Carlo techniques are called for.

  A. Quantum mechanics.

  B. Confidence limits.

III. Implementation: details of how to implement a Monte Carlo calculation.

  A. The Monte Carlo algorithm.

  B. Pseudo-random number generators.

IV. References.

Since you will be modifying the code in the package for the experiment, the code listing for that package is in the notebook for the experiment.

I. PHYSICS BACKGROUND

I.A Review of Special Relativity

Some students have not done any special relativity in a while, and this section is intended as a review for those students. If you have not done *any* special relativity you should consult a textbook.

The rest mass m_0 of a particle is related to its energy E and momentum $\vec{p}$ according to:

$$m_0^2 = E^2 - \vec{p}^2 \qquad \text{(I.1)}$$

Here $\vec{p}$ is the three dimensional vector momentum and $\vec{p}^2$ means the dot product of $\vec{p}$ with itself: $\vec{p}\cdot\vec{p}$. Also, we have adopted units where c is equal to one. Of course:

$$E = \frac{m_o}{\sqrt{1-v^2}} = m$$

$$\vec{p} = \frac{m_o\vec{v}}{\sqrt{1-v^2}} = m\vec{v}$$

where $\vec{v}$ is the three dimensional velocity vector.

The rest mass m_0 is also called the *invariant mass* because it has the same value in all reference frames; this is another way of saying that Equation (I.1) is invariant under the Lorentz transformation. Thus, E and $\vec{p}$ can be measured in any reference frame to determine m_0.

Say the particle described by Equation (I.1) decays into two other particles, 1 and 2. The invariant mass of the parent particle is then given by:

$$m_0^2 = (E_1 + E_2)^2 - (\vec{p}_1 + \vec{p}_2)^2 \qquad \text{(I.2)}$$

Thus, the mass of an elementary particle can be determined by measuring the energy and momenta of the decay products; since the mass is Lorentz invariant the measurement can be made in any reference frame. For a parent particle decaying into three or more daughters, Equation (I.2) generalises to:

$$m_o^2 = (\sum E)^2 - (\sum \vec{p})^2 \qquad \text{(I.3)}$$

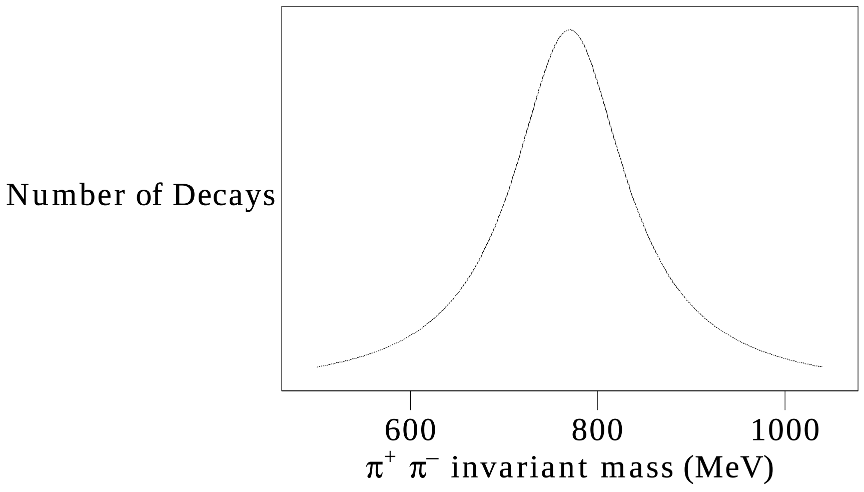
In practice, in high energy physics we usually measure the momenta of the decay products by measuring their radius of curvature in a magnetic field. If we know the rest mass of a decay product, we can then use Equation (I.1) to determine its energy. Then, we can use Equation (I.3) to determine the mass of the parent particle.

I.B High Energy Physics

The study of sub-nuclear structure has discovered many many so-called "elementary particles." Some of these are the photon, neutrinos, electron e, the μ, the τ, plus the e, μ and τ anti-particles; all of these are believed to not have any sub-structure. All the other elementary particles, the *hadrons*, are believed to be made of *quarks*.

With the exception of the photon and electron and the probable exception of the proton, all of the particles are unstable, decaying into other elementary particles. For example, the electrically neutral ρ meson decays into a positively charged π^+ meson and a negatively charged π^- meson.

The mass of the ρ is 770 MeV. If we have a large number of ρ mesons decaying, measure the $\pi^+ \pi^-$ invariant mass from each decay, and form a histogram of the result it will look something like the following:



The shape of the above curve is called a *Breit-Wigner* or *Lorentzian*; you looked at this shape in the Fitting Techniques experiment.

The mass is not exactly 770 MeV for each decay, but is spread out in a bell-shaped curve. The reason is a consequence of the Heisenberg Uncertainty Principle $\Delta E \Delta t \sim \hbar$. Here we think of Δt as the lifetime of the ρ, and it is so small that the measured values of the energy are smeared out.[1] Thus, a measurement of the width of the Breit-Wigner allows us to determine the lifetime of the particle that has decayed.

In the experiment that you will be studying, we have a stationary liquid Hydrogen target; this target can be considered as a collection of stationary protons. An 11 Gev/c K^- beam is incident on the target. A number of different reactions can occur, and 100,000 events were collected corresponding to the final state:

1. Recall that energy and mass are synonyms.

$$K^- \, p \rightarrow K_S \, \pi^+ \, \pi^- \, n$$

There is a meson called a $K^*(1780)$ which decays into a K_S, a π^+, and a π^-. Thus, we view the reaction in which this meson is produced as:

$$K^- \, p \rightarrow K^*(1780) \, n$$
$$K^*(1780) \rightarrow K_S \, \pi^+ \, \pi^-$$

A fairly clean sample of events corresponding to this intermediate state was selected by choosing events with a K_S, π^+, π^- invariant mass between 1700 and 1900 MeV.

There are at least three possible decay mechanisms for the $K^*(1780)$:

1. $K^*(1780) \rightarrow \pi^+ \, \pi^- \, K_S$, i.e. the meson just "falls apart" into a positive π meson, a negative π meson, and a neutral *K-short* meson.
2. $K^*(1780) \rightarrow K^*(890)^- \, \pi^+$, and the $K^*(890)$ then decays into a negative π meson and a K-short. Note that we end up with the same particles in the final state as we did in the first mechanism.
3. $K^*(1780) \rightarrow \rho(770) \, K_S$, and the ρ then decays into a positive and negative π meson. Again the final state is the same.

If we think in terms of quantum mechanics, each of these "decay modes" is described in terms of a wave function Ψ, and the total wave function is:

$$\Psi_{tot} = \Psi_{falls\ apart} + \Psi_{K^*(890)^- \, \pi^+} + \Psi_{\rho(770) \, K_S}$$

The probabilty is then:

$$\Psi^*\Psi = \Psi^*_{falls\ apart}\Psi_{falls\ apart} + \Psi^*_{K^*(890)^- \, \pi^+}\Psi_{K^*(890)^- \, \pi^+} + \Psi^*_{\rho(770) \, K_S}\Psi_{\rho(770) \, K_S} + cross\ terms$$

There are strong theoretical and experimental indications that $\Psi_{falls\ apart}$ is zero. Fortunately, you may also ignore the cross terms. Finally, although other decay modes probably exist for the $K^*(1780)$, in the context of this experiment they are negligible and may be ignored.

II. MOTIVATION

This section discusses the situations in which physicists commonly use Monte Carlo simulations in their work.

II.A Quantum Mechanics

Consider any experimental situation in which quantum mechanics forms the theoretical underpinning. As we know, quantum mechanics only gives us a wave function Ψ, whose usual interpretation is that its square is the probability of some outcome. Thus, when we are attempting to compare an experimental result to a theoretical prediction, the latter is only a probability.

This means that the comparison between theory and experiment is quite different from the sorts of comparison that we looked at in Experiment 1 - *Fitting Techniques*.

As was discussed in §I of this document, we will be looking at the decay of the $K^*(1780)$ meson via three possible decay modes. We will be generating some number of random events corresponding to each of these using a Monte Carlo, and combining them and comparing the generated events with the experimental data.

When the total generated events are closest to the actual data, their relative numbers will allow us to estimate the "branching ratio", which is the fraction of $K^*(1780)$ mesons that decay via each decay mode.

II.B Confidence Limits

You may recall that we stated about non-linear fits that if the data has noise, which is almost a certainty for real experimental data, then there is a difficulty. We can take two sets of data from the same apparatus using the same sample, fit each dataset to a nonlinear model using identical initial values for the fit parameters, and get very different final fits. This situation leads to ambiguity about which fit results are "correct."

A moment's reflection may convince you that the an extension of this problem can arise in any experimental situation in which there are random statistical errors in one or both coordinates of the data. In the supplementary notes for the Fitting Techniques experiment, I called this possibility a case of when the "statistics conspire." A random statistical error means that it is, for example, a standard deviation and we expect that repeated measurements of the quantity will yield something like a Gaussian or Poisson distribution. We expect 68% of the measurements to lie with one standard deviation of the mean, 95% within two standard deviations, 99% within three standard deviations, but since the Gaussian only asymptotically approaches zero the probability of getting a totally wild result from a measurement is never zero.

Of course, actual experimental errors are seldom truly random and statistical in this sense. But they often are approximately random and statistical.

We do a fit of a dataset to some model and get back a set of parameters a_{expt}. In a classical situation there is a set of true values for the parameters, a_{true}, that are known only to Mother Nature. Clever fitters, such as the ones we have used, attempt to estimate the errors in the exerimental parameters, but again the statistics can conspire so that the true values are not within errors of the experimental ones. Further, these estimates of errors in the values of the parameters are based on a series of sometimes dubious statistical assumptions.

Given the data and its experimental errors, we can use a Monte Carlo technique to generate a synthetic data set, say $data_i$. We can fit the synthetic data to the same model we used with the real data to get a set of parameters a_i. Calculate the difference between a_{expt} and a_i. Simulate enough data sets and you will know the distribution of $a_{expt} - a_i$. In this way you can quote a *confidence limit* in which you state that, say, there is a 68% chance that the true parameter values fall within this region around the measured value. Other values for confidence limits are commonly 95% and 99%.

The above technique has revolutionised many fields of experimental science.

III. IMPLEMENTATION

III.A The Monte Carlo Algorithm

Recall that in nuclear decays, a histogram of the energy of one of the decay products will be a Gaussian; you studied one such decay in the Fitting Techniques experiment.

Imagine that we want to generate a number of events whose histogram will be Gaussian. The Gaussian shape is:

$$Gaussian[x, ampl, x_{0,}\sigma] := ampl\ e^{-\frac{(x - x_0)^2}{2\sigma^2}} \tag{III.1}$$

where:

$ampl = the\ maximum\ amplitude$
$x_0 = the\ value$ for $which\ the\ Gaussian\ is\ a\ maximum$
$\sigma = a\ measure\ of\ the\ width\ of\ the\ curve$

Note that we have written the left hand side of Equation (III.1) in a *Mathematica*-esque fashion, and in fact *EDA* supplies a Gaussian routine implementing this definition.

We have values for x_0 and σ, and will set the maximum amplitude *ampl* to 1. Now choose a random value for x. It's value can be anything between zero and infinity, although in practice you will see that choosing a value much outside of the range from $x_0 - 3\sigma$ and $x_0 + 3\sigma$ or so is a waste of time.

Next calculate the value of the Gaussian for that value of x; it will be a number between zero and one, with the higher values indicating that x is closer to x_0.

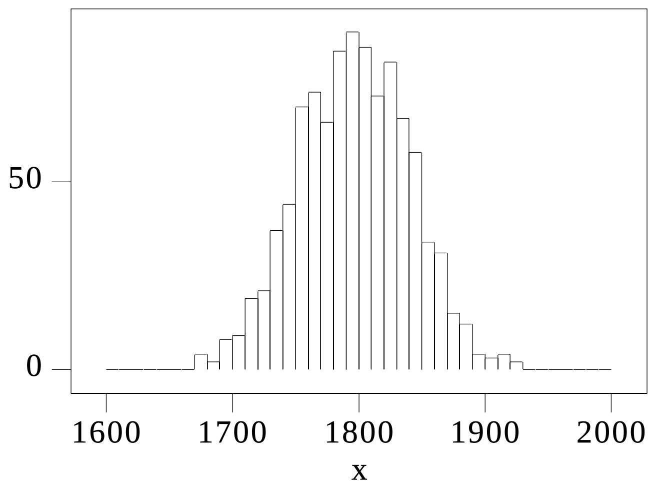
Generate a random number between zero and one. If the value of the Gaussian is greater than the random number, keep the value of *x*; otherwise try again. This is the heart of the **Monte Carlo** technique.

Here is a fragment of *Mathematica* code implementing the above.

```
(*
 * We assume here that EDA is loaded so we can use the Gaussian
 * procedure, and that values for "x0" and "sigma" have already been
 * established. We will also assume a variable "number" has been
 * set to determine the number of events to be generated.  We also
 * assume a variable "maxiters" has been set which is the maximum
 * number of iterations that will be performed; this is so if
 * we give poor values for "x0", "sigma" or "number" the program
 * doesn’t run forever.
 *
 * First establish an empty list of the result.
 *)

result = {};

(* Set the iteration counter to 1. *)
iter = 1;

(* Now loop through until we get enough events. *)
While[ Length[result] < number && iter <= maxiters,

      (* Increment the iteration counter *)
      iter++;

      (*
       * Choose a value for x that is a random number between
       * x0 minus 3 sigma and x0 plus 3 sigma
       *)
      x = Random[ Real, { x0 - 3 * sigma , x0 + 3 * sigma } ];

      (* Find the value of the Gaussian. Note that ampl = 1. *)
      value = Gaussian[x, 1, x0, sigma];

      (*
       * Test the Gaussian value against a random number. Random[]
       * returns a value between 0 and 1 by default.
```

```
         *)
        If [  value > Random[],
              (* Keep this one *)
              result = Append[ result, x ];
        ]

  ] (* end of While[] loop *)

  If[ iter == maxiters,
        Print["Warning: maximum iterations reached."]
  ]
```

A histogram of the output from the above code in one trial run is:[2]



You will note that although the above shape is Gaussian, it is not perfect. Random variations occur, as they should for all Monte Carlo simulations; these random variations also occur in real data from nuclear decays.

III.B Pseudo-Random Number Generators

Consider flipping an honest coin. We *say* that it is random whether we get a heads or a tails, but it isn't: in principle if we knew the initial conditions, speed and rotation of the coin, and the density and viscosity of the air and the various

2. You may wish to know that there are 1000 events, with $x_0 = 1800$ and $\sigma = 44$.

moduli and coefficients of friction of the coin and the surface on which it lands etc. etc. we can calculate whether it will come up heads or tails. We shall call such an event *pseudo-random*.

Consider a single radioactive atom. The moment when it decays is usually considered to be a truly random event.[3]

Computers are, of course, totally deterministic (unless they are broken). This section investigates techniques to generate pseudo-random numbers using a computer. This is a huge and subtle topic, and we will only scratch the surface.

One early suggestion is due to von Neumann, one of the early giants of computer theory. In 1946 he proposed that we start with, say, a four digit number which we will call a *seed*.

```
In[1]:=
      seed = 5772;
```

Square the number and take the middle four digits: this should be a pseudo-random number:

```
In[2]:=
      seed * seed
Out[2]=
      33315984
```

So the pseudo-random number is 3159. Square this and take the middle four digits to generate the next pseudo-random number:

```
In[3]:=
      3159ˆ2
Out[3]=
      9979281
```

So the next pseudo-random number is 9792.

The following code duplicates the above calculations in a more elegant fashion:

```
In[4]:=
```

3. Most conventional interpretations of quantum mechanics say it is random, that "God plays dice with the universe." Recently Bohm and his school developed an interpretation of quantum mechanics which says that radioactive decay is pseudo-random.

```
      ran = 5772;
In[5]:=
      ran = N[Mod[Floor[ran^2/10^2],10^4], 4]
Out[5]=
      3159.
In[6]:=
      ran = N[Mod[Floor[ran^2/10^2],10^4], 4]
Out[6]=
      9792.
```

`Floor[]` rounds its argument down to an integer; `Mod[]` divides its first argument by the second and returns the remainder.

Next we start with a seed of 4938 and generate 20 numbers:

```
In[7]:=
      ran = 4938;
In[8]:=
      Table[ ran = N[Mod[Floor[ran^2/10^2],10^4], 4], {20}]
Out[8]=
      {3838., 7302., 3192., 1888., 5645., 8660., 9956., 1219.,
      4859., 6098., 1856., 4447., 7758., 1865., 4782., 8675.,
      2556., 5331., 4195., 5980.}
```

Repeat starting with a different seed:

```
In[9]:=
      ran = 9792;
In[10]:=
      Table[ ran = N[Mod[Floor[ran^2/10^2],10^4], 4], {20}]
Out[10]=
      {8832., 42., 17., 2., 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
      0, 0, 0, 0}
```

This doesn't look too promising. Try another seed:

```
In[11]:=
      ran = 6372;
In[12]:=
      Table[ ran = N[Mod[Floor[ran^2/10^2],10^4], 4], {20}]
Out[12]=
      {6023., 2765., 6452., 6283., 4760., 6576., 2437., 9389.,
      1533., 3500., 2500., 2500., 2500., 2500., 2500., 2500.,
      2500., 2500., 2500., 2500.}
```

We conclude that von Neumann’s idea is poor.

Mathematica’s random number generator, `Random[]`, by default generates pseudo-random numbers between zero and one. This is a common convention. Many pseudo-random number generators, including *Mathematica*’s, by default choose a seed based on the time of day. One may start with a specific seed with:

```
SeedRandom[ seed ]
```

and `Random[]` will then generate identical sequences of numbers:

```
In[13]:=
      SeedRandom[1234];
In[14]:=
      Table[Random[], {10}]
Out[14]=
      {0.681162, 0.854217, 0.432088, 0.897439, 0.890045, 0.48555,
      0.555124, 0.0837, 0.0108901, 0.309357}
In[15]:=
      Table[Random[], {10}]
Out[15]=
      {0.626772, 0.230746, 0.430667, 0.764379, 0.809171,
      0.476566, 0.643232, 0.542694, 0.565734, 0.208041}
In[16]:=
      SeedRandom[1234];
In[17]:=
      Table[Random[], {10}]
Out[17]=
      {0.681162, 0.854217, 0.432088, 0.897439, 0.890045, 0.48555,
      0.555124, 0.0837, 0.0108901, 0.309357}
In[18]:=
      Table[Random[], {10}]
Out[18]=
      {0.626772, 0.230746, 0.430667, 0.764379, 0.809171,
      0.476566, 0.643232, 0.542694, 0.565734, 0.208041}
```

All pseudo-random number generators have a non-infinite *period*: sooner or later the numbers in the sequences will repeat. A good generator will have a period that is sufficiently long that for practical purposes the numbers will never repeat. Most generators today are variations on a *linear congruential algorithm*. Calling the seed X_0, the algorithm is:

$$X_{n+1} = Mod[aX_n + c,\ m]$$

where:

$m > 0$
$0 \leq a < m$
$0 \leq c < m$

The choices for m, a and c is subtle. For example, if $m = 10$ and $X_0 = a = c = 7$, the generator has a period of only four numbers:

$$7,\ 6,\ 9,\ 0,\ 7,\ 6,\ 9,\ 0,\ \cdots$$

The UNIX random number generator `rand()` has a period of 2^{32}.

In addition to the rather arcane *theory* of random number generation, people have devised a variety of methods to test the generators. Knuth[4] lists 11 empirical tests. In the experiment you will use only one of them: the *equidistribution* test. In this test you check that the random numbers are in fact uniformly distributed between the minimum and maximum values.

IV. REFERENCES

- D.E. Knuth **The Art of Computer Programming**, 2nd ed. (Addison-Wesley, 1973), Vol 2, Chapter 3.
  This is a classic seven volume work. The cited chapter deals with computer generated random numbers.
- William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling, **Numerical Recipes: The Art of Scientific Computing** or **Numerical Recipes in C: The Art of Scientific Computing** (Cambridge Univ. Press), §14.5 (Confidence Limits) and Chapt. 7 (Random Numbers.
  A thorough discussion of confidence limits and Monte Carlo techniques.
- E.F. Taylor and J.A. Wheeler **Spacetime Physics** (W.H. Freeman, 1963).
  My personal favorite book on relativity.

4. Vol 2, §3.3.2