1 Monte Carlo integration

In general total crosssections are not what we are after. There are very few experiments that can measure total crosssections. Usually the part of the output space that is covered by a detector is incomplete. If a detector covers only 80% of the phase space one cannot just multiply the observed crosssections by 5/4 to obtain the total crosssection. The distribution of the matrix element over the phase space may not be uniform. In general it is not and also the phase space integral itself may have some Jacobians when we transform it into something that we can compute. Hence we have to emulate the restrictions of the detector into our integrals. This can be extremely complicated, because the detectors may have lots of little dead zones where cables have to find their way to the outside or support beams have to keep all the equipment in place. As theorists we usually leave the finer details of those dead zones to the experimentalists. We will just concentrate on the broad features of the detectors. An example could be that we assume that a photon can be observed if $E_\gamma > 4GeV$ and $|\cos\theta_\gamma| < 0.8$.

The only general way we know to do the integration over the phase space under the above restrictions is by Monte Carlo techniques. In principle this works as follows:

Suppose we want to integrate a function $f(x)$ over the region between a and b. This integral is the average value times the size of the interval. To obtain the average value we take a number of random values between a and b, evaluate the function in these points and then take the average. Of course in a single dimension there are usually much better techniques, but in our case such techniques may not be so efficient after all. We will come to that. In general the error that we make in the integral is related to the number of sample points N by $\sigma/\sqrt{N-1}$ in which σ is the standard deviation in the values of the function in the evaluation points[1].

Let us have a look at a simple example. We want to integrate the function $f(x) = 3x^2$ over the region from 0 to 1. Because we know the answer we can see clearly what is happening.

First we need a random number generator. A rather good one is given in the file ranf.c. It is based on the research results of various people. The program is given in the file test1.c and we translate the whole program with the linux command

```
cc ranf.c test1.c -o test1 -lm
```

After the compilation we execute the program with the command

```
test1
```

and we get on the screen the answer

```
For 1000000 points:   1.00083843 +/- 0.00089423
```

[1] If one would like to be precise, one also has to worry about how accurate the estimate of the standard deviation is. We will not do that here.

Let us see whether this makes sense.

If we want to know the standard deviation we need $\sigma^2 =< f^2 > - < f >^2$ which we can compute to be 4/5. The error is then $\sigma/\sqrt{N-1}$ in which N is the number of points. The value of this quantity is printed in the second output line. It is 0.00089443. Very close! We also see that the answer of the calculation is just within the error.

One of the advantages of the Monte Carlo integration is that this $1/\sqrt{N}$ rule survives when the integral is over more than one dimension. This is not the case for quadrature rules which in one dimension may have a much better behavior, but as soon as the number of dimensions increases it becomes worse. Another advantage is the behavior with respect to experimental cuts. Suppose that in our one dimensional integral the area between c and d should be ignored. In that case we can either split the integral into two regions: a to c and d to b, or we can just put the function to zero when the value of a random point is between c and d (verify this!). In principle d could be larger than b etc, in which case we have to consider lots of special cases if we would just like to manipulate the integration boundaries. In a multi-dimensional integral this setting to zero can be done similarly, provided we know how the 'dead zones' can be described in terms of kinematical variables. Assume for instance that the energy of one of the outcoming photons in a given reaction is not equal to one of our integration variables, but instead it is a complicated function of them. In that case we calculate for each Monte Carlo point what this energy is. If we don't see photons with an energy less than 1 GeV, we can put the integrant to zero in the case that the computed value of the photon energy is less than 1 GeV. This introduces discontinuities in our integrant, but the Monte Carlo integration is relatively insensitive to this. Other methods however usually assume that the integrant is continuous and also continuous in one or more derivatives.

Another advantage (not to be underestimated) is that while integrating a function by means of Monte Carlo techniques, one can make distributions of any number of variables that are functions of the integration variables. These distributions can be presented as histograms and if done properly, they are approximations to differential crosssections.

The great sport in doing Monte Carlo integration is to select the proper integration variables. And we define proper integration variables as variables in which the standard deviation is as small as possible. As an example we take the function

$$f(x) = 1/(x+e) \tag{1}$$

and our integration domain is from zero to one. We assume that $e > 0$. For this function we can compute

$$\begin{aligned} \sigma^2 &= < f^2 > - < f >^2 \\ &= \frac{1}{e(1+e)} - (\log(\frac{1+e}{e}))^2 \end{aligned} \tag{2}$$

We see that if e becomes very small, the standard deviation increases like $1/\sqrt{e}$.

Just imagine however that we change to another variable: $u = (\log(x+e) - \log(e))/(\log(1+e) - \log(e))$. This variable will run from zero to one as well.

Homework: You are to verify that in terms of this variable the standard deviation is actually zero!

Of course we are usually not in such a good situation. What would be more realistic is that we integrate a function

$$f(x) \quad = \quad g(x)/(x+e) \tag{3}$$

in which $g(x)$ depends very weakly on x. Now, if we change to the variable u, we have $g(x(u))$ which supposedly depends rather weakly on u as well and the standard deviation has been improved greatly. We call this technique mapping and the art of phase space integration is to 'map away' the inverse power peaks in the integrant as much as possible[2].

There is a general principle about how to make such mappings. Assume that $f(x)$ is the function that we want to 'compensate' for. This means that we want to distribute our points according to the surface under the function. This is done by defining

$$F(x) \quad = \quad \int dx\ f(x) \tag{4}$$

$$u \quad = \quad \frac{F(x) - F(x_-)}{F(x_+) - F(x_-)} \tag{5}$$

We assume here that f is nonnegative over the integration region and that we integrate from x_- to x_+. This makes that u is a number between 0 and 1. In that case we have

$$\int_{x_-}^{x_+} dx\ g(x)f(x) \quad = \quad (F(x_+) - F(x_-)) \int_0^1 du\ g(x(u)) \tag{6}$$

This can always be done provided we know what the primitive function F looks like and that we can invert it. This inversion can in principle also be done numerically.

Another way to improve the standard deviation is by means of stratified sampling. In stratified sampling we try to make the distribution of the random points a bit more uniform than they would be if they were completely random. This can be done for instance by dividing the integration area in 10 equal regions and give each region 10% of the points, randomly distributed through the region. This is like a cross-breed between a Simpson rule and a Monte Carlo. In a D-dimensional cube one would make subdivisions in D dimensions. In a 7 dimensional space with 3 subdivisions in each direction this would give 2187 little hypercubes, which is already quite a few. It shows that accurate multi-dimensional integration isn't easy.

We can illustrate this with the function of our first example. The program is in the file test2.c and it is compiled like before with now test2 instead of test1. The answer is

[2] A few years ago a rather systematic way for doing this has been developed which is called sector decomposition. If you ever get into serious complications with peaks, look up papers on this subject.

```
For one range of 1000000 points: 1.00083843 +/- 0.00089423
For 1000 ranges of each 1000 points: 1.00000098 +/- 0.00000100
```

As you can see the stratification gives a considerably smaller error. The improvement is of the order of the number of strata. This is not always the case. We just had a rather benign function. Functions like $f(x) = 2\sin^2(1000\pi x)$ would give much less of an improvement.

Homework: Playing with the values of the variables $np1$ and $np2$ in the program test2, what would be their optimal values to obtain an error of 10^{-6}? With optimal we mean that the total number of function evaluations is minimal.

The program we use for the Monte Carlo integration is called VEGAS. It is a somewhat older program, but basically it has not been possible to improve on it in a significant way. Other general programs are variations on it with slight improvements. There do exist dedicated programs for special cases that may be better, but here we want a general program that is moreover easy to understand.

VEGAS does Monte Carlo integration using two types of improvement techniques to reduce the value of the error. Its main technique is based on mapping. For this VEGAS works in iterations. It evaluates the function in a number of points, after which it studies the results. Based on these results it adapts the distribution of its points in the next iteration. It stops when either a maximum number of iterations is reached or a required precision has been achieved. When VEGAS is called we have to specify how many points we want to use in each iteration. When VEGAS starts it begins with setting up the stratification. It divides the unit cube (it always integrates over a unit cube) into a maximal number of subcubes, in such a way that each subcube has at least two random points in it. The two points guarantee that for each subcube an error can be computed and afterwards all errors can be combined. At the same time it sets up a system of D linear mappings. In each dimension it defines an array that starts out evenly spaced. Hence if we assume that there are 7 dimensions, there will be 7 such arrays. For each Monte Carlo point there will be a function value and this value is added to the appropriate bins in the 7 arrays. This will leave us with 7 distributions in which the function has been integrated over the other 6 dimensions. Based on these distributions VEGAS will redistribute the points in the next iterations in such a way that it attempts the new bins to have equal contents. This is not quite as good as the mapping we used before (that was continuous, while this method makes the new effective function rather discontinuous), but it is quite an improvement. By making some bins smaller (if they are very full) and other bins larger (if they have very little content) and throwing roughly equal numbers of points in each bin, one can see that in each iteration VEGAS manages to get a smaller standard deviation and hence a smaller error. This continues untill the grid cannot be improved. One cannot do a perfect job this way. This would only be possible if the integrant would be a product of one dimensional functions. Unfortunately, multi-dimensional structures cannot be dealt with this way. Programs that do attempt to resolve multi-dimensional structures are usually much more complicated and rely on a very large number of integration points. Assymptotically they may win,

but we don't always have the computer resources to evaluate the function for assymptoticly many times.

Homework: Imagine the function $f(x) = x$ in the region 0 to 1. a: What is the standard deviation? What error should we get with 100 random points? b: Imagine we do stratified sampling and divide the region into two equal parts. What should the error become now if we use 50 points in each region? c: Using the mapping technique of VEGAS with two bins in which we try to make the bins such that they give equal contributions to the integral, what will the error be now (with 50 points in each bin)? d: Another option is to adjust the bins such that they all contribute equally to the error. What is the result now?

Next we do the same for the function $f(x) = 3x^2$. Assume we divide the integration region in two parts: 0 to a and a to 1. What do you notice now? What is the best value of a?

Finally we have another example of a two dimensional function in which we have the program produce distributions. These distributions are actually histograms. For each Monte Carlo point we compute the relevant variables and add the value of the integral to the proper bin. Note that one has to take into account that if we use mappings in whatever form, there will be jacobians involved.

In principle one can just calculate the standard deviation and hence the error of all points that end up in the bin, regardless of during which iteration. This can cause problems with iterations in which there are one or two really bad points. Another approach is to only take the points of the last iteration. There exist programs that do this. The approach we take here is to weight the results in each bin for each iteration with the inverse of the relative error obtained in that iteration. Note: one should not use the absolute error, because if in an early iteration Vegas gets a small result with a small error because it didn't see the most important part of the function yet, we would put a very heavy weight on this unreliable iteration.

When this system of making histograms is used it produces an output file which can directly be offered to the LaTeXsystem for further processing. It uses axodraw2.sty as a style file and one has to translate it eventually into a postscript file or a pdf file. The relevant part of the example is in the file main2.c. The way you use this histogramming is by running the program 2 with the command

```
prog2 plot2.tex
```

This creates the LaTeXfile plot2.tex with the relevant information. Its information can be made visible with the commands

```
latex plot2
dvips plot2 -o
okular plot2.ps
```

or

```
pdflatex plot2
axohelp plot2 -o
```

```
pdflatex plot2
okular plot2.ps
```

It needs the file axodraw2.sty which you should have installed in your current directory. You can make the .ps file into a .pdf file with

```
ps2pdf plot2.ps
```

On older systems the program okular may not exist. In that case you may try the call `kghostview plot2.ps`. In the case you prefer to use pdflatex you also need the axohelp program. The source of it is available in the axodraw distribution and you can compile it with

```
cc -o axohelp -O3 axohelp.c -lm
```

This program is only needed when you have either new graphics or changed existing graphics in your LaTeXfile. It is needed because neither LaTeXnor the pdf language offer facilities for nontrivial calculations. Hence axohelp is needed to translate the axodraw primitives into the simple pdf commands. In the case of the use of the latex/dvips commands the axodraw primitives are transferred to postscript which is a language that can do nontrivial mathematics.

There is one word of caution with respect to the use of vegas. When vegas has trouble converging on the proper answer this manifests itself in a rather poor value of the variable named 'chi-squared per iteration'. This variable is defined as

$$\chi^2 \;=\; \frac{1}{n-1}\sum_i \frac{(< f >_i - < f >)^2}{e_i^2} \tag{7}$$

in which the index i refers to the different iterations, n is the number of iterations, $< f >_i$ is the answer in the i-th iteration, e_i is the error in that iteration and $< f >$ is the combined answer of all iterations. If everything goes according to statistics this variable should have a value of about one. If it is significantly larger usually it means that vegas may have missed a significant part of the function during one or more iterations. There are various corrective actions:

- Find a mapping that makes the function smoother.
- Give vegas more points in each iteration.
- Run for more iterations.

The first solution, if it exists, is the best.