Jonas R. Glesaaen: glesaaen@th.physik.uni-frankfurt.de

C library for Numerical Integration

1 Introduction

In this project we will write multiple C-functions to calculate numerical integrals of one-dimensional functions. We will start with three basic closed region algorithms. After having used these to write constant stepsize numerical integration functions, we will move on to semi-adaptive[1] stepsize algorithms. We will then tackle integrals with improper integration ranges (such as infinity), and finally we will implement the simplest Monte Carlo integration algorithm.

2 Constant stepsize algorithms

You have already worked with the simplest of the integration algorithms in tutorial VI, the Riemann sum[2]. In the tutorial the midpoint rule was used, which we will come back to later, but first let us look at the left- and right Riemann sums.

$$\int_{x_0}^{x_1} f(x)\,\mathrm{d}x = hf(x_0) + \mathcal{O}(h^2 f'), \qquad (\textit{left}) \tag{1}$$

$$\int_{x_0}^{x_1} f(x)\,\mathrm{d}x = hf(x_1) + \mathcal{O}(h^2 f'), \qquad (\textit{right}) \tag{2}$$

where h denotes the stepsize, which is $(x_1 - x_0)$ in the above equations. Graphical representations of eqs. (1,2) are shown in figures 1 and 2 respectively.

1. Write out the analytic formulae approximating the integral using the left- and right Riemann sums with N steps

2.1 Function signature

To write a routine which can integrate any function (or at least try to integrate any function), you will need to use functional pointers. To begin with, a function which takes in one variable and returns the result of the evaluation:

```
double function(double x)
```

will always be sufficient. However, at some point one might also want to pass some number of parameters, so that one doesn't have to write a different C-function for every interesting parameter choice. To solve this, you could have one integration function for a number of parameter set-ups:

```
double integral(...,double(*func)(double),...)
double integral(...,double(*func)(double,double),...)
double integral(...,double(*func)(double,double,int),...)
```

but this solution obviously creates as much double work as the previous one. To bypass this, we can make use of the fact that any pointer type in C can be typecast to a `void` pointer. So, by using the following function signature

```
double function(double x, void *params)
```

[1] Only semi-adaptive as the stepsize is not fully flexible, but can only be divided by a constant, 2 and 3 in our case
[2] http://en.wikipedia.org/wiki/Riemann_sum

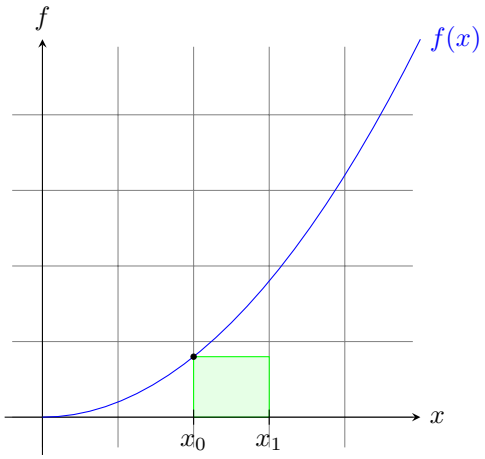

Figure 1: Left Riemann Sum

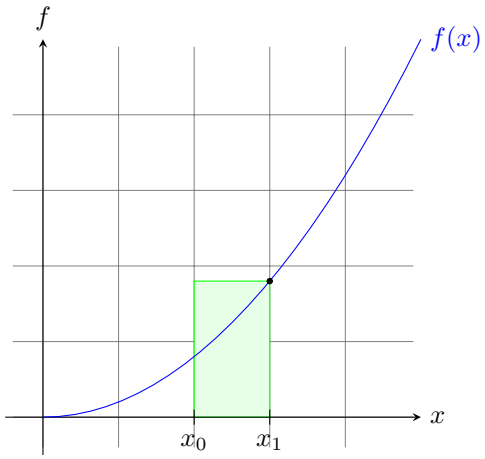

Figure 2: Right Riemann Sum

we can pass any number of parameters we wish. One does this by re-casting it back to its initial form inside of the function call, so e.g. if you want to pass two `double`'s, it can be written like this:

```
//In the main()-function
double p[2] = {2., 4.};
function(x,p);

//...

double function(double x, void *params){

  double *arr = (double *)params;
  double p1 = arr[0];
  double p2 = arr[1];

  //...
}
```

If your function doesn't take any parameters, you can simply pass `NULL`. If you are having difficulties with this, you can use the simple function call mentioned at the beginning of this subsection, and implement this change at the very end.

With this in mind, a possible function signature for the integration function itself is:

```
double int_finite_c_alg(double a, double b, void *params,
                        double (*func)(double,void*), double h)
```

where `a` and `b` are the integration bounds, `params` are the parameters to be passed on, `func` the function pointer, and lastly some sort of error control. To begin with we will used a fixed stepsize, and later move on to absolute and relative errors.

(Optional)

A cleaner, more secure and more readable possibility is to define your own `struct` for a given parameter set. This way you can also pass multiple different variable types, such as two `double`'s and one `int`. For example for the Gaussian distribution

$$f(x;\mu,\sigma) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}},$$

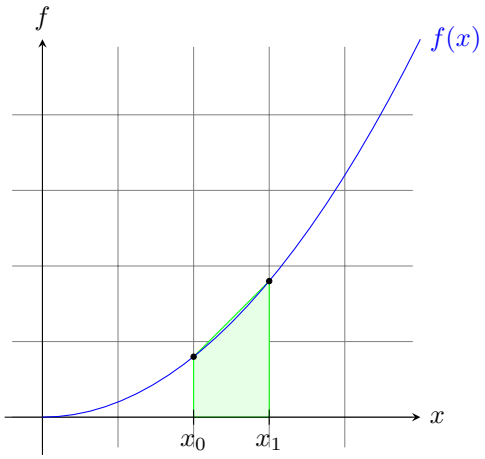

Figure 3: Trapezoidal Rule

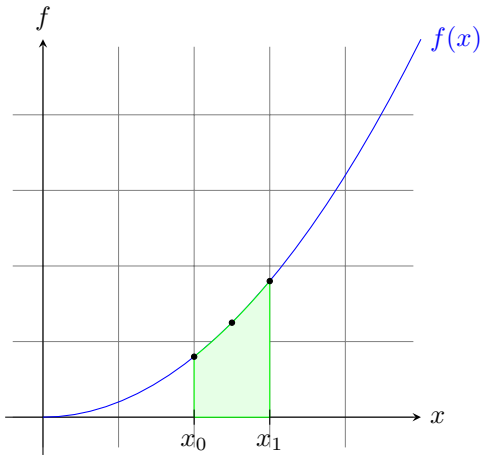

Figure 4: Simpson's rule

you could write a `struct` on the form:

```
struct gaussian_parameters{
  double mu;
  double sigma;
};
```

then pass an instance of this `struct` to your function and typecast it as shown above. `struct`'s will be covered in the lectures sometime after Christmas. If you want to read about them earlier, you can consult your favourite text book on C programming, or have a look here: `http://www.cplusplus.com/doc/tutorial/structures/`.

---

2. Implement the left and right Riemann sums and test them on the Gaussian distribution with 0 mean and 1 standard deviation in the range $[-1, 1]$. Compare the result with ones found in standard mathematical tables using varying stepsizes.

2.2 Improvements

Implementing the left and right Riemann sums, you should have noticed that depending on the form of the function you want to integrate, one of them will overshoot while the other will undershoot. We should therefore get a better result if we take the average of the two, which leads us to the trapezoidal rule:

$$\int_{x0}^{x1} f(x)\,\mathrm{d}x = \frac{h}{2}\big[f(x_0) + f(x_1)\big] + \mathcal{O}(h^3 f''). \quad (\textit{trapezoidal}) \tag{3}$$

It is already apparent from the error estimate that the trapezoidal rule is valid to one order higher than the Riemann sums, but at the cost of one more function evaluation. An graphical representation of the trapezoidal rule is shown in figure 3.

3. Repeat task 1 for the trapezoidal rule. You should notice that even though eq. (3) states that we have two function evaluations rather than one at every step, the rightmost evaluation can be re-used as the leftmost evaluation at the next step. When added up, this means that we only need *one* additional evaluation of the function compared to the Riemann sums no matter the size of N. This might not seem too important for the programs we write here, but for more complicated functions, every evaluation is precious, and we want to get the best possible result with the fewest number of evaluations.

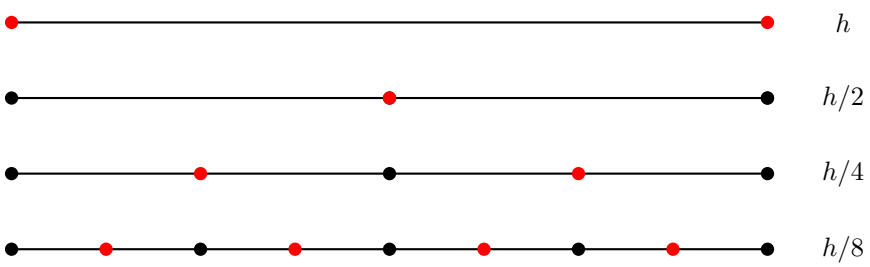

Figure 5: Stepsize halving with the trapezoidal rule

Finally, by also evaluating the function at its midpoint, we end up with Simpson's rule[3] , illustrated in figure 4:

$$\int_{x_0}^{x_1} f(x)\,\mathrm{d}x = \frac{h}{6}\left[f(x_0) + 4f(x_{1/2}) + f(x_1)\right] + \mathcal{O}(h^5 f^{(4)}). \quad (Simpson's) \tag{4}$$

Comparing the number of function evaluations to the order of the accuracy, one sees that Simpson's rule is accurate to one order higher than we have come to expect. This is due to a "lucky" cancellation owing to a left-right symmetry which makes Simpson's rule one of the best low order constant stepsize numerical integration algorithms.

4. Once again write the expression for an integral approximated by Simpson's rule with N steps. How many additional function evaluations do you need compared to the Riemann sums at a given N?

5. Implement both the trapezoidal rule and Simpson's rule to calculate integrals numerically for a given stepsize. Test the routine for the Gaussian integral ($\mu = 0$, $\sigma = 1$), and for the integral

$$\int_0^2 x\cos^2(2\pi x^2)\,\mathrm{d}x$$

which you can solve analytically. How do they compare with each other? How do they compare with the Riemann sum?

3 Semi-adaptive stepsizes

Rather than passing a stepsize you think will give an approximately close result to your function, it would be more convenient to pass an error margin we want the result to be within. One way to implement this would be to have a function which calculates the integral at a given stepsize, then at a lower stepsize (e.g. multiplied by a constant number < 1). It then compares the results, and if the difference between the two is too large, it repeats the process until the error limit is reached.

Without a proper choice of algorithm, and the constant stepsize reduction, this is quite inefficient. However, e.g. using the trapezoidal rule, and halving the stepsize at every iteration, we can reuse the result from the previous iteration when calculating the result for the current. This is because by halving the stepsize, we still visit all the points we did at full stepsize, but we add the ones in between as illustrated in figure 5.

6. Find and analytic expression for the relationship between the trapezoidal sum at stepsize h and stepsize $h/2$, alternatively with N evaluations and then with $2N$ evaluations.

[3]See e.g. Numerical Recipes, Third Edition, pages 156-160, or `http://mathworld.wolfram.com/SimpsonsRule.html`

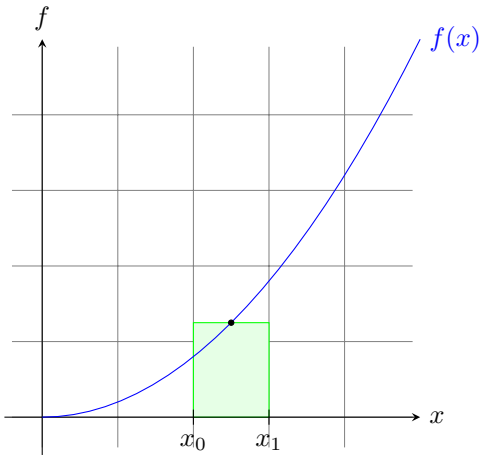

Figure 6: Midpoint rule

7. Having this relationship, try to write a function which instead of the stepsize h, takes the relative error as its final argument, and uses stepsize halving to calculate the integral within the given precision.
   (Optional) Do this for Simpson's rule as well.

4 Open boundary algorithms and infinite limits

So far we have only looked at what is called "closed algorithms" as they calculate the integral on closed intervals, $[a, b]$. Next, we need an algorithm integrate over open intervals such as $[a, b)$, which is needed if there for example is an integrable singularity at b, or if we want to integrate to infinity.

To accomplish this we turn to the midpoint rule, which is another variation of the Riemann sum. However, instead of evaluating the function at either end-point, it is evaluated at the midpoints, as shown in figure 6, giving the expression:

$$\int_{x_0}^{x_1} f(x)\,\mathrm{d}x = hf(x_{1/2}) + \mathcal{O}(h^2 f'). \qquad (midpoint) \tag{5}$$

By evaluating the function at the midpoints, we never have to evaluate the function at the boundaries, and hence we can use this to calculate open boundary integrals. With this expression, we shall go right to the semi-adaptive stepsize implementation, but there is a slight subtlety. We cannot double the number of evaluations and still re-use the result from the previous iteration, as we did with the trapezoidal rule, but we can triple it. At every point we thus add one new point to the left, and one to the right at one third stepsize. A graphical representation is given in figure 7, from which we can see that at every iteration the outer-most points we evaluate quickly approaches the end-points, but never exactly reaches them. Similar to the situations with integrals of open intervals.

8. With figure 7 in mind, write an analytic expression to show how one iteration depends on the result from the previous iteration for step tripling of the midpoint rule, similar to what you did in task 6.

9. Then write a function which takes in a relative error and then calculates the integral by utilising step tripling and the midpoint rule, again similar to what you did in task 7.

4.1 Boundaries taken to infinity

The easiest way to handle improper boundaries, be that infinite values or singular points, is by transforming the integration variable to an expression in which the integral can be evaluated by a

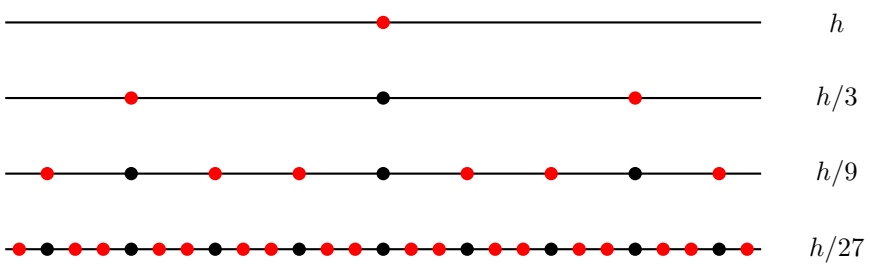

Figure 7: Evaluation point tripling with the midpoint rule

computer. For infinite boundaries, e.g. $x_1 \to \infty$, one possible transformation to use is:

$$\int_{x_0}^{x_1 \to \infty} f(x)\,\mathrm{d}x \xrightarrow{x=\frac{1}{t}} \int_{1/x_1 \to 0}^{1/x_0} \frac{1}{t^2} f\left(\frac{1}{t}\right)\mathrm{d}t \tag{6}$$

which converges if $f(t)$ declines faster than $1/t^2$. One should note that one obviously cannot both transform and integrate across zero, meaning that if we were to integrate over e.g. $[-5, \infty)$ we need to split the integral into two pieces and integrate the finite domain separately using one of our previous routines.

10. Write a function which only takes a lower bound (a) and calculates the integral in the region $[a, \infty)$ using the semi-adaptive midpoint rule and the transformation in eq. (6). Try it out on the integral:

$$\int_0^\infty e^{-x^2}\,\mathrm{d}x,$$

and compare it with the analytic result.

4.2 Regions with integrable singularities at the boundary

A short mention is given to the fact that if one has an integrable singularity of order γ at either boundary, the semi-adaptive midpoint rule will give the correct result, but will be quite inefficient. If one instead makes use of the following transformation for singularities at the lower boundary:

$$x = t^{\frac{1}{1-\gamma}} + x_0, \tag{7}$$

giving the following equality,

$$\int_{x_0}^{x_1} f(x)\,\mathrm{d}x = \frac{1}{1-\gamma} \int_0^{(x_1-x_0)^{1-\gamma}} t^{\frac{\gamma}{1-\gamma}} f\left(t^{\frac{1}{1-\gamma}} + a\right)\mathrm{d}t, \tag{8}$$

or the transformation

$$x = x_1 - t^{\frac{1}{1-\gamma}}, \tag{9}$$

for singularities at the upper boundary, before applying the open boundary semi-adaptive midpoint algorithm.

12. *(Optional)* Integrate the function

$$f(x) = \frac{1}{\sqrt{x}}$$

in the interval $(0, 1]$ using the adaptive midpoint routine you wrote in task 9, then again using the transformation defined in eq. (7) and compare how fast they converge.

5 Monte Carlo Integration

Lastly we will look at a method of approximating integrals which is completely unrelated to the previous algorithms presented. We will approximate an integral by sampling the value of the function in the given region randomly, which is known as Monte Carlo integration.

Imagine that we take a very large number of samples N of the function $f(x)$ using a statistical variable x which is uniformly distributed in the region $[a, b]$. If we take N large enough, the samples should be smoothly distributed in the given region, and the average distance between sample points is around h/N. This would be similar to a midpoint Riemann sum with stepsize h/N, so by taking enough samples we reduce the error from the Riemann sum, as well as the error from the statistical uncertainty. The Monte Carlo integral expression is thus

$$\int_{x_0}^{x_1} f(x)\mathrm{d}x \approx h\langle f\rangle \pm h\sqrt{\frac{\langle f^2\rangle - \langle f\rangle^2}{N}}, \tag{10}$$

where

$$\langle Y\rangle = \frac{1}{N}\sum_{x=0}^{N-1} Y(x_i), \qquad x \in [x_0, x_1]. \tag{11}$$

With the sampling process being a statistical process, we cannot put as rigorous bounds on the error of the expression as we can with the previous expressions, but one uses the statistical error (given as the $\pm$ term) to get an approximate error.

One should note that the Monte Carlo integration algorithm is very inefficient compared with the other methods, as you will normally need many orders of magnitude more function evaluation to reach the same accuracy, but it is generally used for multi dimensional integrals where the boundary is hard to define analytically. Say for instance the volume inside of a Klein-bottle contained within a sphere of lower radius, or a torus cut off by a parabola. In which case one can sample the variable from an approximate volume which contains the desired volume, and simply define the function in question to be zero outside of the desired integration region.

13. The final task is to implement the simple Monte Carlo integral for an integral on a given region within a given absolute error. You can get a pseudo-random integer in the region [0,`RAND_MAX`] using the `rand()` function defined in "stdlib.h" (the macro `RAND_MAX` is also defined in "stdlib.h"). The pseudo-random number routine can be seeded using `srand()`. Test it against your other routines. How is the performance?