COM2004/3004 Lab Week 3
Computing Expectations by using Numerical Integration

Objectives

- **To gain experience with some *advanced* MATLAB programming features: passing functions as parameters; defining `anonymous' functions.**
- **To see how numerical integration can be used to approximate the mean and variance of a distribution.**

1. Background

In the lecture notes we have seen that expectations of continuous random variables (e.g. means and variances) can be computed from the probability density function by integration. In the lectures we have used 'analytic' integration to derive exact expressions for the mean and variance of the Gaussian distribution and the uniform distribution. Sometimes integration can be hard to perform analytically. In such cases we can resort to approximate numerical techniques to compute the mean and variance. There are two main approaches we can take, i/ monte-carlo sampling: (i.e. generating samples from the distribution and computing their statistics directly). ii/ integration by numerical quadrature, (i.e. approximating the area under a curve by cutting it into a finite number of slithers that we then sum up). Quadrature works very well for 1-D problem and will be the focus of this lab class.

PART 1 – Numerical quadrature

2. Introduction

Numerical quadrature is a form of numerical integration and can be used to integrate functions between definite limits, i.e. finding the area under a curve between a lower and upper limit on the x-axis. There are several different versions of the technique but they all work in the same way. The area under the curve is approximated by summing a series of segments whose area is easy to compute. In the simplest case the segments are very thin rectangles, e.g. in the figure overleaf, the area under the curve can be approximated by summing the area of each of the rectangles shown. Note that the width of each rectangle is some fixed step size, and the heights are calculated by evaluating the function f(x) at the x position where the rectangle is centered. The area of each is simply the width times the height.

The more rectangles we use, the thinner they will be and the closer the sum of their areas will be to the true area under the curve. i.e. by summing more rectangles we can improve the precision of the approximation – at the expense of additional computational effort.

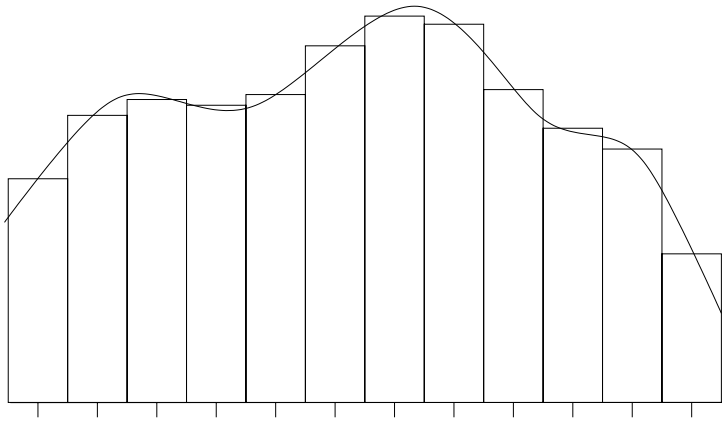

Figure 1: The area under the curve is approximated by the area of the rectangles.

3. Passing function as arguments

MATLAB already has functions for performing numeric integration but in order to get some more MATLAB programing experience we are going to write our own.

Our function needs to take four inputs:

- f() - the function we wish to integrate,
- a - the lower bound,
- b - the upper bound and
- N - the number of segments we wish to use to approximate the integral.

Note, the 1st parameter is a *function* – we are passing a function to a function. In Java you could do this by wrapping the function in a class and passing an object of that class. In MATLAB you can pass functions directly by using the '@' symbol. Let's illustrate this by passing the cos() function as a parameter to a function called squared that can compute the squared output of any function:

First write the function `squared` like this

```
function y=squared(fn, x)
y = fn(x);
y=y.^2;
```

Save it in a file called `squared.m`.

Now to compute the square of cos(pi/2) we could call our function as,

```
squared(@cos, pi/2)
```

But if we wanted to compute the square of sin(pi/2) we could write,

```
squared(@sin, pi/2)
```

What happens if you try the following,

```
squared(@sin, 0:pi/100:pi/2)
```

4. Numerical integration

We now need to write our numerical integration function.

Say we want to integrate `f()` between *a* and *b*. We are first going to evaluate *f(x)* at lots of positions between *a* and *b*, (say *N* positions).

To do this we can first generate a number line with `N` points between `a` and `b` stored in the vector `x`. We can do this using MATLAB's `linspace` function

```
x = linspace(a, b, N)
```

We then pass the vector x to our function f() in the usual way.

```
fx  =  f(x)
```

This gives us the heights of the rectangles on the previous page. There will be *N* rectangles between *a* and *b* so we can work out that their width will be *(b-a)/N*. So the area of each one is *fx(i) . (b-a)/N*. So our integral, which is the total area of all the rectangles combined, is given by

```
sum(fx)*(b-a)/N
```

The more slithers we use the better the accuracy will be.

The function signature for our integrate function should look like

```
function y=integrate(f, a, b, N)
```

The description above should give you enough detail to complete the function. If you are totally stuck put up your hand and ask for help.

5. Computing the mean of a distribution

Remember, if x is a random variable with distribution *p(x)* then its mean value, *E(x)* is given by integrating *x.p(x)* between plus and minus infinity.

We will now make a new function called `compute_mean` that takes the distribution as an input function. The `compute_mean` will also take the upper and lower bounds and N slices parameters that will be passed on to the integrate function. So define a new function with the following signature,

```
function m =compute_mean(f, a, b, N)
```

Inside the body of this function we will use our function integrate to evaluate the integral of *x.f(x).*

But how can we pass `x.f(x)` to `integrate`? `integrate()` takes the function that we want to integrate as input, but the problem is that we haven't written a function that directly evaluates `x.f(x)`.

The solution is to use a MATLAB construct called an `anonymous function' to define a new function `on-the-fly' inside the `compute_mean` function body.

`compute_mean` has been passed a function called f and we want to write a new function that evaluates `x.*f(x)` -- which we'll call `xfx`. This can be done by defining a new function using the following anonymous function syntax

```
xfx = @(x) x.*f(x);
```

This has made a new function called `xfx` that takes `x` as its parameter and "`x.*f(x)`" as its body.

Once this function has been defined we can now integrate it using our integrate function, i.e.

```
mean = integrate(xfx, a, b, N)
```

Note that in the above you did not need to use `@xfx` when passing `xfx` to `integrate`. MATLAB knows that `xfx` is a function. However if we want to pass a normal non-anonymous function e.g. cos then we would need to use `integrate(@cos, a, b, N)`.

The details above should be sufficient for you to be able to complete the `compute_mean` function body.

6. Testing the compute_mean function

We can now test our compute mean function. MATLAB has various functions that represent common continuous distributions, e.g. `normpdf, unifpdf, betapdf` etc

`normpdf(x)` evaluates a normal pdf with 0 mean and unit variance at the point `x`.

So we can compute the mean using something like the following,

```
compute_mean(@normpdf, -100, 100, 1000)
```

Experiment with different a, b and N parameters and compare the results.

We can compute the mean of the uniform pdf using similar code,

```
compute_mean(@unifpdf, -100, 100, 1000)
```

To evaluate a normal pdf with mean 5 and variance 4 we would need to supply parameters to the MATLAB normpdf function, i.e.

```
normpdf(x, 5, 2)
```

(Note, the 3rd parameter sets the standard deviation – the square root of variance – so for a variance of 4 we need to set the standard deviation to 2).

But how can we pass the 5 and 2 parameters to our compute_mean function? The pdf function, f, that is passed to `compute_mean` is only meant to have one parameter.

The solution is to make a new one-parameter function that evaluates `normpdf(x,5,2)` and which only takes x as an input. We can do this by defining a new function, again using the anonymous function syntax, as follows,

```
mynormpdf = @(x) normpdf(x, 5, 2);
```

Then we could compute its mean using,

```
compute_mean(mynormpdf, -100, 100, 1000)
```

Experiment with different normal distributions (i.e. ones with different mean and variance parameters). Estimate the mean using different values of `N`.

7. Computing the variance

We will now extend `compute_mean` so that it also computes the variance. Make a new function called `compute_mean_and_variance` with the same parameters as computer mean but which returns both a mean and a variance

```
[mean, var]=compute_mean_and_variance(f, a, b, N)
```

Copy the code from `compute_mean` so that the mean computation is complete.

How do we compute the variance? We saw that variance is E((x-E(x))^2). This can be rearranged into a handier form (see tutorial solutions) as var(x) = E(x^2) – E(x)^2

Our function already computes E(x) i.e. the mean. We now just need to extend it to compute E(x^2). i.e. we need to integrate x^2.p(x) . Do this using the same steps described in section 3. i.e. you will need to define another anonymous function in the `compute_mean_and_variance` function body to evaluate x^2.p(x) and then pass this to the integrate function to compute E(x^2). Once you have evaluated E(x) and E(x^2) you can use them to compute var(x)

8. Evaluating the estimates

Use your mynormpdf function (i.e. the one with mean of 5 and variance of 4)

```
y = normpdf(x)
```

Set the limits of the integration to be -100 to 100. Now call `compute_mean_and_variance` with increasing values of `N`.

Given that in this case we know the true mean and variance (i.e. 5 and 4) we can compute the error in the estimates (i.e. the difference between the estimates and the true values).

Make a plot of the estimation error as a function of N.

How quickly does the error decrease with increasing N?

9. Challenge: Using your code

A variable x has a pdf defined by

p(x) = 30 * x .* ((1 - x) .^ 4) for x >=0 and x<=1

and p(x) = 0 for x<0 and x>1.

Use your code to compute the mean and variance of this distribution. Email me your answer.

PART 2 – Programming Challenge

10. Programming exercise

i/ Simulate tossing a fair coin 1,000,000 times. Count the length of each sequence of identical outcomes. Plot a histogram of the result.

For example, the following sequence, H H H T T H H H H T T T, would count as 1 sequence of 2 (T T); 2 sequences of 3 (H H H and T T T) and 1 sequence of 4 (H H H H).

Hint: the coin toss can be simulated by using the function `rand' to pick a number between 0 and 1 and outputting heads if the number is greater than 0.5 or else outputting tails.

Write your solution using a for-loop to start with.

Can you find a way of `vectorising' your algorithm – i.e. writing it without loops? Is it faster?

Time your solutions using tic and toc. If you think your code is fast then send me the solution and the time you obtained.

ii/ Bias the coin so that it comes up heads with a probability, p, greater than 0.5.

How does the shape of the histogram vary with increasing p?

11. What next – Friday Lecture

MATLAB: We'll go over solutions to this lab class.

Classification: Bayesian classifiers.