Chapter 4

Integration

4.1 Integration and risk-neutral pricing

Sections A1-A4 (p363-p367) of the appendix of "C++ for Financial Mathematics" by John Armstrong describe the theory of risk-neutral pricing in elementary terms. Don't worry you don't need to know any C++ to read this. I'll assume that you have read this in what follows as I can't express things better than I already have there.

If we have a European derivative whose payoff is determined by the price of the option at a final time T we can compute the expected payoff by the formula:

$$E_{\mathbb{Q}}(\text{payoff}) = \int_{\mathbb{R}} \text{payoff}(S_T) \times \mathbb{Q}\text{-probability-density}(S)\,\mathrm{d}S$$

Let's introduce some symbols so we can write this more compactly. We will write q for the probability density function of S at time T. We will write $P(S)$ for the payoff given the final stock price.

$$\begin{aligned} \text{price} &= e^{-rT} E_{\mathbb{Q}}(\text{payoff}) \\ &= e^{-rT} \int_{\mathbb{R}} P(S) q(S)\,\mathrm{d}S \end{aligned} \tag{4.1}$$

We conclude that we can price these kinds of simple derivative by integration once we know the p.d.f. q.

It is up to the modeller to decide what p.d.f. to use for q. A real trader would calibrate some model to market prices for puts and calls and use this to compute the q-pdf. One obvious choice of model would be that the stock price follows geometric Brownian motion.

Let us derive the probability density at time T from this assumption. We'll try to do this quickly and painlessly as revision of what you have learned in your stochastics courses.

We begin by assuming that the stock price satisfies:

$$\mathrm{d}S_t = S_t(\mu \mathrm{d}t + \sigma \mathrm{d}W_t) \tag{4.2}$$

for some μ and σ. In general this will not be a valid $\mathbb{Q}$-measure model because the discounted stock price will not be a Martingale.

Given the model (4.2), what is the pdf of S at time T? Well by Itô's lemma, the log of the stock price z_t satisfies:

$$\mathrm{d}z_t = (\mu - \frac{1}{2}\sigma^2)\,\mathrm{d}t + \sigma\,\mathrm{d}W_t.$$

This is means that z_t follows Brownian motion. By the very definition of a stochastic differential equation we can immediately deduce that

$$z_t = z_0 + (\mu - \frac{1}{2}\sigma^2)t + \sigma(W_t - W_0).$$

Hence the distribution of z_T is a normal distribution.

$$z_T \tilde{\mathcal{N}}(z_0 + (\mu - \frac{1}{2}\sigma^2)T, \sigma\sqrt{T})$$

So by the very definition of the log normal distribution, S_T is log normally distributed.

$$S_T \tilde{\ln \mathcal{N}}(z_0 + (\mu - \frac{1}{2}\sigma^2)T, \sigma\sqrt{T}) \tag{4.3}$$

We would like to compute the p.d.f of S_t.

We need to know the p.d.f. of the log normal distribution. Suppose that $x \sim \mathcal{N}(\alpha, \beta)$. Then

$$\begin{aligned} P(x \leq t) &= \int_{-\infty}^{t} \frac{1}{\beta\sqrt{2\pi}} \exp\left(-\frac{(x-\alpha)^2}{2\beta^2}\right) \mathrm{d}x \\ &= \int_{0}^{e^t} \frac{1}{X\beta\sqrt{2\pi}} \exp\left(-\frac{(\ln X-\alpha)^2}{2\beta^2}\right) \mathrm{d}X \end{aligned}$$

Here we've made the substitution $x = \ln(X)$. We also have:

$$P(x \leq t) = P(\ln(X) \leq t) = P(X \leq e^t)$$

So the p.d.f. of the log normal distribution is:

$$\frac{1}{X\beta\sqrt{2\pi}} \exp\left(-\frac{(\ln X-\alpha)^2}{2\beta^2}\right)$$

We conclude from (4.3) that the p.d.f. of S_T is

$$\frac{1}{S\sigma\sqrt{2\pi T}} \exp\left(-\frac{(\ln(S/S_0) - (\mu - \frac{1}{2}\sigma^2)T)^2}{2T\sigma^2}\right).$$

Integrating this we find that the mean of S_T is:

$$\exp(z_0 + \mu T) = S_0 \exp(\mu T).$$

We conclude that our model (4.2) can only possibly be a $\mathbb{Q}$-measure model if we take $\mu = r$ where r is the risk free rate.

Theorem 1. *Geometric Brownian motion gives a valid $\mathbb{Q}$-measure model if and only if $\mu = r$. In this case the $\mathbb{Q}$-p.d.f. of S_T is:*

$$q_T(S) = \frac{1}{S\sigma\sqrt{2\pi T}} \exp\left(- \frac{(\ln(S/S_0) - (r - \frac{1}{2}\sigma^2)T)^2}{2T\sigma^2} \right) \tag{4.4}$$

Given a European derivative that pays off $P(S)$ if the stock price at time T is S, the risk neutral price of this derivative is:

$$\textit{price} = e^{-rT} \int_0^\infty P(S) q_T(S) \, \mathrm{d}S. \tag{4.5}$$

(Note that some derivatives are path dependent and so cannot be priced using this formula. We will cover these derivatives later in the course.)

Note that the derivative is assumed to be European and not path dependent. The function $q_T(S)$ is often called the pricing kernel.

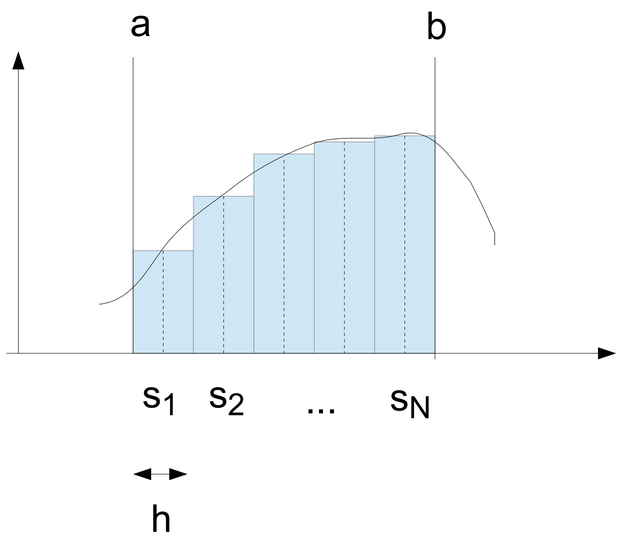
Note that by Girsanov's theorem, we can prove (as will have been done in one of your stochastics courses) that if we have a $\mathbb{P}$ measure model of the form (4.2), then the $\mathbb{Q}$ measure model obtained by setting $\mu = r$ is the unique equivalent Martingale measure. This means that if you have a $\mathbb{P}$-measure model of this form, you are forced to perform risk neutral pricing using the pricing kernel (4.4).

This gives a recap of the theory of risk-neutral pricing and explains the link between integration and mathematical finance.

4.2 Integration Methods

How can we evaluate one dimensional integrals such as (**??**) in practice? You probably already know three elementary methods of estimating integrals: the rectangle rule, the trapezium rule and Simpson's rule.

In the rectangle rule (also known as the midpoint rule) we approximate a function f that we wish to integrate over some interval $[a, b]$ using a number of strips of with h. Let the sequence of points s_i denote the midpoints of each strip. We compute the value of $f(s_i)$ at the midpoint of each strip and use this to compute the area of each strip $hf(s_i)$. We estimate the area under the graph, and hence the integral of f as the sum of all these areas.



Algorithm (Rectangle Rule)**.** *Let $f : [a,b] \longrightarrow \mathbb{R}$ be a function we wish to integrate. Choose a number of points N and define:*

$$h = \frac{b-a}{N}$$
$$s_m = a + (m - \frac{1}{2})h$$
$$R = h\sum_{i=1}^{N} f(s_m)$$

Then R is the rectangle rule estimate for $\int_a^b f(t)\,\mathrm{d}t$.

One might feel that it would be better to approximate the area under the graph by approximating the area with a series of trapeziums. In this case we let the sequence s_i denote the edges of each trapezium and evaluate the integral at these points. We calculate the area of the trapeziums to obtain.

Algorithm (Trapezium Rule)**.** *Let $f : [a,b] \longrightarrow \mathbb{R}$ be a function we wish to integrate. Choose a number of panels n and define:*

$$\begin{aligned} h &= \frac{b-a}{n} \\ s_m &= a + mh \\ T &= \frac{h}{2}(f(s_0) + 2f(s_1) + 2f(s_2) + \ldots + 2f(s_{n-1}) + f(s_n)) \end{aligned}$$

Then T is the trapezium rule estimate for $\int_a^b f(t)\,\mathrm{d}t$. The number of integration points is $N = n + 1$.

We've tried approximating the function using constant segments (midpoint rule), then linear segments (trapzium rule). What if we approximate using quadratic segments? We arrive at Simpson's rule.

We already knew the formula for the area of a rectangle and the formula for the area of a trapezium. To derive Simpson's rule we need to know the formula for the area under a quadratic curve given the values at either end and in the middle.

We state the result:

Lemma 1. *content... Let f be a quadratic function then*

$$\int_{-h}^{h} f(t)\mathrm{d}t = \frac{h}{3}(f(-h) + 4f(0) + f(h)). \tag{4.6}$$

Proof. We can prove this by noticing that:

$$\int_{-h}^{h} 1\mathrm{d}x = 2h = \frac{h}{3}(1 + 4 + 1).$$

If f is an odd function, in particular if $f(x) = x$ then

$$\int_{-h}^{h} f(x)\mathrm{d}x = 0 = \frac{h}{3}(f(-h) + 4f(0) + f(h))$$

Finally

$$\int_{-h}^{h} x^2\mathrm{d}x = \frac{2}{3}h^2 = \frac{h}{3}(1 + 0 + 1).$$

So the formula (4.6) is correct for the function $f(x) = 1.f(x) = x$ and $f(x) = x^2$. Since both sides are linear in f, it is correct for all quadratic functions. □

More surprisingly we have:

Lemma 2. *content... Let f be a* **cubic** *function then*

$$\int_{-h}^{h} f(t)\mathrm{d}t = \frac{h}{3}(f(-h) + 4f(0) + f(h)). \tag{4.7}$$

Proof. $f(x) = x^3$ is odd, so (4.6) holds for $f(x) = x^3$. So by the same argument as the previous lemma it holds for all cubic functions. □

We can add together these quadratic estimates over a number of strips to obtain a version of Simpson's rule with multiple steps.

Algorithm (Simpson's Rule)**.** *Let $f : [a, b] \longrightarrow \mathbb{R}$ be a function we wish to integrate. Choose an* even *number of panels n and define:*

$$\begin{aligned} h &= \frac{b-a}{n} \\ s_m &= a + mh \\ S &= \frac{h}{3}\sum_{i=0}^{\frac{n}{2}} (f(s_{2i}) + 4f(s_{2i+1}) + f(s_{2i+2})) \\ &= \frac{h}{3}(f(s_0) + 4f(s_1) + 2f(s_2) + 4f(s_3) + 2f(s_4) + \dots \\ &\quad + 2f(s_{n-2}) + 4f(s_{n-1}) + f(s_n)) \end{aligned}$$

Then S is the Simpson's rule estimate for $\int_a^b f(t)\,\mathrm{d}t$. The number of integration points is $N = n + 1$.

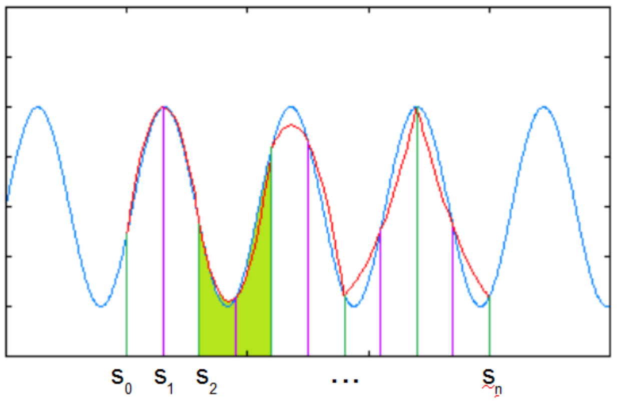


Figure 4.1: Simpson's Rule. We integrate over n strips where n is even. This gives $\frac{n}{2}$ parabolic panels such as the one shaded in green. There are $n + 1$ integration points s_i.

Theorem 2. *If f is four times differentiable with $|f^{(4)}| < K$ for some K, then the error in the Simpson's rule estimate can be bounded above by*

$$\frac{c}{N^4}$$

where c is a constant depending upon a, b and K.

Lemma 3. *If f is four times differentiable with $|f^{(4)}| < K$ for some K, then the error in the Simpson's rule estimate over an interval of length $2h$ with two steps can be bounded above by $c_1 h^5$ for some constant c_1 depending on K.*

Proof that theorem follows from lemma. The theorem follows from the lemma because we're adding up $\frac{n}{2}$ copies of the 2-step Simpson's rule. So the cumulative error is:

$$\begin{aligned}\frac{n}{2}c_1 h^5 &= \frac{n}{2}c_1\left(\frac{b-a}{n}\right)^5\\ &\leq c\frac{1}{N^4}\end{aligned}$$

for some constant c □

Proof of lemma. By Taylor's theorem:

$$\begin{aligned}f(x) &= f(x_0)+(x-x_0)f'(x_0)+\frac{1}{2}(x-x_0)^2 f^{(2)}(x_0)+\dots\\ &\quad+\frac{1}{3!}(x-x_0)^3 f^{(3)}(x_0)+\frac{1}{4!}(x-x_0)^4 f^{(4)}(\xi)\end{aligned}$$

for some $\xi(x)\in[x_0,x]$ (or $[x,x_0]$ if $x\leq x_0$).

We have already noticed that Simpson's rule is exact for cubic functions. Therefore the error from Simpson's rule over an interval $[x_0,x_0+2h]$ is equal to the integral of the non-cubic term in the above expansion.

$$\begin{aligned}\text{error} &= \left|\int_{x_0}^{x_0+h}\frac{1}{4!}(x-x_0)^4 f^{(4)}(\xi(x))\mathrm{d}x\right|\\ &\leq \int_{x_0}^{x_0+h}\frac{1}{4!}(x-x_0)^4 K\mathrm{d}x\\ &= ch^5\end{aligned}$$

for some constant c. □

It is nice to introduce a special notation for talking about the rate of convergence of a numerical method called "Big-O notation".

Definition. *Given a function $f:\mathbb{N}\to\mathbb{R}$ we write that a sequence $s_n=O(f(n))$ if $s_n\leq C|f(n)|$ for some constant.*

Theorem 3. *If f is four times differentiable with $|f^{(4)}|<K$ for some K, then the error in the Simpson's rule estimate is $O(N^{-4})$.*

Theorem 4. *If f is twice differentiable with $|f^{(2)}|<K$ for some K, then the error in the Trapezium rule estimate is $O(N^{-2})$. The same is true for the error in the Rectangle rule.*

Proof is left as an exercise. □

4.2.1 More sophisticated integration methods

It would be misleading to give the impression that Simpson's rule is the last word on numerical integration. For example, it is fairly obvious that we can develop methods with arbitrarily rapid convergence by using higher order polynomial approximations.

The state of the art of numerical integration is even more sophisticated.

One key observation is that all our integration rules are of the form:

$$\int_a^b f \approx \sum_{i=1}^{N} w_i f(x_i)$$

for some weights $w_i \in \mathbb{R}$ and evaluation points $x_i \in [a, b]$.

So far we have chosen to evenly distribute all the points x_i where we evaluate the function. If we use more integration points where the function is changing rapidly and less where it is roughly constant, we may be able to find good estimates with less function evaluations. By observing which parts of a numerical estimate change the most as one adds in additional integration points it is possible to decide guess where it would be most effective to add further integration points. Algorithms that use this idea are called adaptive methods.

Secondly we have noticed that our numerical schemes do better than expected. Simpson's rule is designed using quadratic approximations but works perfectly for cubics too, rather to our surprise. It turns out that if we are willing to move the integration points we use so that they are not evenly distributed we can derive "surprisingly effective" schemes in higher orders two. Specifically Gauss gave a recipe to find an integration rule which gives a perfect answer for polynomials of degree $2d - 1$ with only d points. It is called "Gaussian quadrature". Quadrature is just another word for integration. Gauss's result is mathematically very interesting. He found that if we choose the x_i to be the roots of the Legendre polynomials and the weights can be calculated in terms of the derivatives of the Legendre polynomials. The details make Gaussian quadrature a little fiddly to implement for a novice programmer (and novice mathematician) so Gaussian quadrature is not examinable. However, it is widely used in practice.

4.2.2 Integration over infinite intervals

As we have already seen, by performing a substitution one can transform an infinite integral to an integral over an open interval. Our integration rules (other than the rectangle rule) use the end points of the integral. The value may not be defined at these points. When performing the integral we just discard those values (and any other singularities) and hope that this doesn't cause a problem.

Note that the choice of substitution makes a big difference to the performance of the method. This doesn't just apply to infinite integrals. If you can make a change of variables so that your integral becomes constant, then you can estimate it perfectly with just one function evaluation!

4.2.3 Integration in practice

We will implement the rectangle rule and trapezium rule in Matlab as examples. Simpson's rule is left for you to implement. However, in practice you would never use your own integration function as Matlab has a function `integral`. This function uses an adaptive Gaussian quadrature strategy, so it is pretty sophisticated.

However, you do sometimes need to be careful when using it. Here are some pointers.

Ensure your function is well centered and scaled. If you ask Matlab to integrate the density

$$\frac{1}{\sqrt{2\pi}}\exp(-\frac{(x-\mu)^2}{2})$$

from 0 to ∞ and set $\mu = 1000$ it will, incorrectly, give the answer 0. The correct answer is, of course, 1 as this is just the p.d.f. of the normal distribution with mean μ. Here's some code for you to try to test this.

```
function ret = integrateNormal( mean )

    function r = integrand( x)
        r = exp(-(x-mean).^2 / 2 );
    end
    ret = 1/sqrt(2*pi)*integral(@integrand,-Inf,Inf);
end
```

What's going on? If $\mu = 1000$ then the p.d.f. of the normal distribution is nearly zero except near 1000. When Matlab tries to evaluate the integral it picks a few points and evaluates the integrand at these locations. Since it is unlikely to pick the point 1000, it fails to notice the places where the p.d.f. is non-zero and incorrectly deduces that the integral is zero. This is a failure of the adaptive integration strategy to find the "interesting bits" of the distribution. If we make sure the "interesting bit" is near zero and spread over a region of roughly unit length, Matlab will start looking at evaluation points around zero and so will correctly find the distribution. This is why performing the same integral with $\mu = 0$ works just fine.

Tell Matlab where the interesting points are. You can optinally pass a list of "Waypoints" to integral to tell it some points to include as evaluation points. For example you might include any discontinuities in your integral as waypoints.

Subtract off singularities. If your integrand f is singular, you can often rewrite f as $f = f_1 + f_2$ where f_1 is non-singular and f_2 is singular but can be computed analytically. Asking Matlab to integrate f_1 and then adding on your own computation for the integral of f_2 will help Matlab cope with your singular integral.

4.3 Higher dimensional integration

1-dimensional integration is easy. As we've seen an algorithm such as Simpson's rule is simple to write and has very rapid convergence ($O(N^{-4})$).

At first glance the obvious way to evaluate higher dimensional integrals is to inductively estimate d-dimensional integrals by applying one of the rules above to an $d-1$-dimensional integral

$$\int\int f(x_1, x_2, \ldots x_d)\,\mathrm{d}x_1\,\mathrm{d}x_2,\,\ldots\mathrm{d}x_d \\ = \int\left(\int f(x_2, \ldots x_d)\mathrm{d}x_2,\,\ldots\mathrm{d}x_d\right)\,\mathrm{d}x_1$$

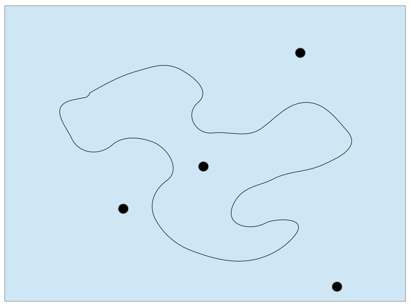
The problem with this approach is that if we use N grid points for each one dimensional integration then we will need to use N^d points in total for a d-dimensional integration.

To see this just note that if n_d be the number of function evaluations required to compute a d-dimensional integral, we have the difference equations $n_1 = N$ and $n_d = N * n_{d-1}$.

Since $n_d = N^d$, we see that for a 100-dimensional problem with 10 grid points for each dimension we will need 1 googol function evaluations (1 googol$=10^{100}$) to estimate the integral. This is an astronomically large number. Hence this approach is completely impractical. This is called the curse of dimensionality.

4.3.1 Estimating area using Monte Carlo

Suppose that we want to estimate the are of the blob in the picture below. It lies



in a rectangle of area $1m^2$.
Suppose that we throw 4 dots randomly (i.e. uniformly distributed over the rectangle) and one lies in the blob and three outside. We could then estimate the area of the blob to be $0.25m^2$. This may seem a crazy way to estimate the area, but it will work!

Algorithm. *To find the area of a given shape* Σ

- *Bound it by a rectangle of known area* A*.*
- *Throw* n *darts randomly (i.e. uniformly distribution) into the rectangle*
- *Count the number of darts,* n_Σ*, that land in the shape* Σ*.*
- *The area is approximately*

$$\frac{n_\Sigma}{n} A$$

The same idea can be applied to computing integrals. It is then called Monte Carlo integration.

Algorithm (Monte Carlo Integration)**.** *Let $f : [a,b] \longrightarrow \mathbb{R}$. Estimate the integral of f by generating N random numbers x_m which are uniformly distributed in the interval $[a,b]$. The Monte Carlo estimate for the integral is:*

$$\frac{1}{N}(b-a)\sum_{i=1}^{N} f(x_m)$$

Theorem 5. *If we assume that the integral of $f(x)^2$ exists). By this we mean that the standard deviation of the error is $O(N^{-\frac{1}{2}}$.*

Proof. This is an application of the central limit theorem. Each individual estimate of the integral $f(x_i)$ is a random variable with mean $f(x_i)$. They are independent and identically distributed. Our assumption on f ensures that these random variabless have a variance. So by the central limit theorem, the sample average is an unbiased estimator for the true average. The standard deviation of the error is proportional to $O(N^{-\frac{1}{2}}$. □

Without any effort this can be generalized to higher dimensions. The important point is that it does not suffer from the curse of dimensionality (though see our remarks later).

Algorithm (Monte Carlo Integration)**.** *Let $f : [0,1]^d \longrightarrow \mathbb{R}$. Estimate the integral of f by generating N random numbers x_m which are uniformly distributed in the hypercube $[0,1]^d$. The Monte Carlo estimate for the integral is:*

$$\frac{1}{N}\sum_{i=1}^{N} f(x_m)$$

Experience suggests that dimension 3 or 4 is the approximate dimension where Monte Carlo integration begins to out-perform methods based on 1 dimensional integration rules. In this course we will primarily be performing either 1 or very high dimensional integrals so this intermediate cases won't crop up.

We should also remark that you cannot generate true random numbers on a computer you have to make do with pseudo-random numbers. One has to hope that the pseudo-random numbers behave rather like real random numbers. In practice you should check how many random numbers you will need and make sure that you are using a generator that guarantees to provide high quality pseudo-randomness for that many numbers. This won't be an issue on Matlab as it uses a high quality random number generator that only repeats after an enormous number of samples have been drawn.

We should remark that Monte Carlo doesn't completely get rid of curse of dimensionality. The error in our estimate is less than $cN^{-\frac{1}{2}}$ but c can grow to be enormous for high dimensions.

The one obvious problem with the Monte Carlo method is that $O(N^{\frac{1}{2}})$ is still very slow convergence. It means we will need of the order of 1 million samples to have an accuracy of the order of 1 part in a thousand. For example you can estimate π by using the Monte Carlo method to compute the area of a circle, but it will take a very long time if you want to compute π to 10 decimal places.

Fortunately in finance we don't often need to be accurate to 10 decimal places. Stock prices themselves have a bid-ask spread which is of the order of one part in a thousand. This means that if you are using "the stock price" as an input to your calculation, you are probably kidding yourself if you think it is meaningful to calculate the answer to 10 decimal places. This means that although Monte Carlo is slow, it is still often good enough to be useful.

4.4 Implementing Integration methods

We have seen the code to integrate by the rectangle rule before. We make a slight change this time and include a `for` loop. The reason we have done this is that you might want to integrate a non-vectorized function, so it is best to iterate through the points one by one and apply f rather than assume that f can be evaluated elementwise.

```
function ret = integrateByRectangleRule( f, a, b, N )
h = (b-a)/N;
s = a + h*((1:N) - 0.5);
total = 0;
for x=s
    total = total+f(x);
end
ret = h*total;
end
```

It is easy to adapt our code to perform the trapezium rule. It should then be easy for you to adapt this to perform Simpson's rule. This is left as an exercise, as is Monte Carlo integration.

```
function ret = integrateByTrapeziumRule( f, a, b, N )
n = N-1; % N = number of grid points, n=number of panels
h = (b-a)/n;
s = a + h*(1:n-1);
total = f(a)+f(b);
for x=s
    total = total+2*f(x);
end
ret = 0.5*h*total;
end
```

We would also like to write general purpose functions to perform integrals over infinite intervals. By substitution we know that:

$$\int_x^\infty f(t)\mathrm{d}t = \int_0^1 \frac{1}{s^2} f(x-1+\frac{1}{s})\mathrm{d}s.$$

Here is the corresponding code to integrate f from x to ∞.

```
function ret = integrateToInfinity( f, x, N, integrationRule)
% Performs a substitution to change
% the infinite integral to a finite integral.
if nargin<4
   integrationRule=@integrateByRectangleRule;
end

function r = transformedFunction( s )
    r = s^(-2) * f( x - 1 + 1/s );
    if (isnan(r))
        r = 0.0;
    end
end

ret=integrationRule( @transformedFunction, 0, 1, N );
end
```

This is quite sophisticated code. It allows you to pass in an optional argument to specify the integration rule to use after we have made the change of variables. This is specified by passing in the integration function to use when estimating the integral. Our code provides a default value to use (it defaults to using the rectangle rule). If you like you can write your own functions that provide default values for arguments using `nargin`. This contains the number of parameters the user has actually passed in. So if it is lower than you expect, supply default values.

Another sophistication is that the integrand may be undefined at certain points. In particular it may not be defined at the end points. So we use the function `isnan` to see if a calculation has evaluated to “Not a number”. If this happens, we just pretend the value was zero. Hopefully if just a couple of integration points are missing, this won't affect the integral.

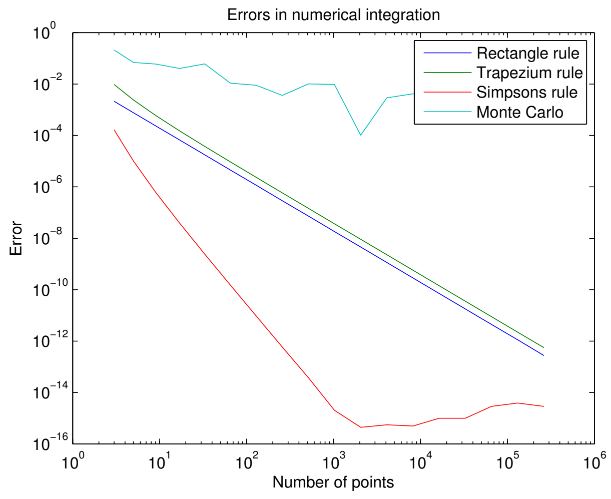
4.4.1 Analyzing our results

We have four integration methods: the rectangle, trapezium and Simpson's rule and the Monte Carlo method. Let's plot a graph of their performance. To do this we calculate

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

analytically (the answer is $-\cos(2)+\cos(0)$) and then compare this with the numerical integrals.

In **??** we plot a log-log plot of the number of evaluation points used in the integral against the error.



The gradients of the log-log plot show the rate of the convergence. Simpson's rule has gradient -4, corresponding to its $O(N^{-4})$ rate of convergence. The trapezium rule has gradient -2, and the rectangle rule has gradient -2 as expected.

The error of the Monte Carlo method is a random variable. We know its standard deviation is $O(N^{-\frac{1}{2}})$, but any particular sample will sometimes do well and sometimes do badly. We've only plotted one sample in **??**. If you squint, you may agree that the long term trend has a slope of roughly $-\frac{1}{2}$ as one might expect. We could perform repeated runs to find the average Monte Carlo error.

A surprising feature of our graph is that after a certain point the error of Simpson's rule increases. This is due to accumulating rounding errors. These grow with slope 1/2 consistent with the central limit theorem: we're adding independent errors. If you look at the chart you will see that rounding errors only start creeping in once we have an accuracy of about one part in 10^{15}. So Simpson's rule approaches machine precision accuracy rather quickly. After that it is a bad idea to use more integration points.

Here is the code that was used to generate this plot.

```
function plotErrors()
% Plots the error of computing integral of sin from 0 to 1
points=1:18;
NValues = 2.^points + 1; % Exercise: explain this line
answer = -cos(2)+cos(0);

errorR = zeros( 1, length(NValues));
errorT = zeros( 1, length(NValues));
errorS = zeros( 1, length(NValues));
errorM = zeros( 1, length(NValues));
for i=1:length(NValues)
    N = NValues(i);
    fprintf('Running calculation with %d points\n', N);
    errorR(i) = abs(integrateByRectangleRule(@sin,0,2,N) - answer);
    errorT(i) = abs(integrateByTrapeziumRule(@sin,0,2,N) - answer);
    errorS(i) = abs(integrateBySimpsonsRule(@sin,0,2,N) - answer);
    errorM(i) = abs(integrateByMonteCarlo(@sin,0,2,N) - answer);
end

figure();
loglog( NValues, errorR, NValues, errorT, NValues, errorS, NValues, errorM );
title('Errors in numerical integration');
xlabel('Number of points');
ylabel('Error');
legend('Rectangle rule', 'Trapezium rule', 'Simpsons rule', 'Monte Carlo' );

end
```

The code works by estimating the integral using $n_i = 2^i + 1$ evaluation points as i runs from 1 to 16. Given that we are producing a log-log plot, this means that our plot points will be distributed evenly in the x-direction.

The variable `points` is a vector containing all the values of i from 1 to 16. The variable `NValues` contains a vector of the different values of n_i.

We then compute a vector `errorR` containing the error in the rectangle rule for each value of n_i, a vector `errorT` for the error of Simpson's rule and so forth. These values are computed inside the `for` loop where we run through all the values of n_i.

Having computed the vectors of errors, we can simply use the built-in `loglog` function to draw a `loglog` plot. We then set the title with the `title` function. We use similar functions to label the axes and provide a legend. Once you have generated your plot you can easily save it to a file to include in a LaTeXdocument as we have done.

In MATLAB you should use the `figure` command to open a new drawing window. Then commands like `plot`, `loglog`, `title`, `xlabel` and so forth can be called. They will all affect the current drawing window so you can gradually add features to your plot.

4.5 Pricing using integration

Let us put everything together and price an option using integration.

Example 1: A dividend free stock follows the Black Scholes process with unknown drift and 10% volatility. The current stock price is \$100. You should assume that the risk free rate is 5% APR. Use numerical integration to find the risk neutral price of a 3 month European call option on the stock with strike \$105?

The first thing we need to do is translate the figures from the market into familiar mathematical notation.

In finance time is measured in years, so $T = 0.25\,\text{years}$. Volatility is quoted as a percentage change over a year, so $\sigma = 0.1\,\text{years}^{-\frac{1}{2}}$. The risk free rate, r, used in the Black Scholes Formula is a continuously compounded rate. So we have $e^r = 1.05$ if the annual percentage rate (APR) is 5%. Thus $r = \ln(1.05)$.

How will we price this derivative. Learning from our experience we will write several functions.

- A function to compute the payoff of a call option.
- A function to compute the pricing kernel.
- A function to integrate their product and hence price the derivative.

The payoff of a European call option can be computed using the code below. We have vectorized the code so it will work equally well with a vector of strikes and a vector of stock prices.

```
function p=callPayoff(K, S)
inMoney = S > K;
p = inMoney.*(S-K);
end
```

You could rewrite this using the maximum function if you wished.

Recall that we have computed the pricing kernel mathematically and found that it is:

$$q_T(S) = \frac{1}{S\sigma\sqrt{2\pi T}} \exp\left(-\frac{(\ln(S/S_0) - (r - \frac{1}{2}\sigma^2)T)^2}{2T\sigma^2} \right)$$

We can translate this into MATLAB with ease.

```
function q=pricingKernel( S, T, S0, r, sigma )
coefficient = 1./(S * sigma * sqrt( 2 * pi * T));
numerator=-(log(S/S0)-(r-0.5*sigma^2)*T).^2;
denominator = 2*T*sigma^2 ;
q =  coefficient .* exp( numerator/denominator);
end
```

We can now use the theory of risk neutral pricing to compute the payoff.

$$\text{price} = e^{-rT} \int_0^\infty P(S) q_T(S)\, \mathrm{d}S.$$

```
function p=priceCallByIntegration( K, T, S0, r, sigma )
function ret=integrand( S )
   ret = callPayoff( K, S ) .*...
           pricingKernel( S, T, S0, r, sigma );
end
p = exp(-r*T) ...
    * integrateToInfinity( @integrand, 0, 10001);
end
```

We again remember that we should always test our functions. How can we test our call price? The most obvious approach is to cheat and compare it to the values from the Black-Scholes formula.

```
function testBlackScholesCallPrice()
%Compares the black scholes price against
%that obtained by integration
r = log(1.05);
S0 = 100;
sigma = 0.1;
T = 0.25;
K = 110;
actual = priceCallByIntegration(K,T, S0, r, sigma );
expected = blackScholesCallPrice(...
           K,T, S0, r, sigma );
assertApproxEqual( actual, expected, 0.001 );
end
```

If we don't want to cheat, a good test is to notice that a call option with a strike of 0 always pays off the same as the stock. So, in effect, a call option with a strike of 0 is the same as the stock and so must have the same price. If we confirm that this is the case, our code will still be doing quite a sophisticated computation involving integrating pricing kernels, so this will be quite a good test in the sense that if we've made any errors it probably will fail.

```
function testPriceCallByIntegration()
% A call option with strike 0 is the same thing
% as buying the stock. Check we get the correct answer.
% This is effectively a test that our pricing
% kernel is risk neutral
r = log(1.05);
S0 = 100;
sigma = 0.1;
T = 0.25;
actual = priceCallByIntegration(0,T, S0, r, sigma );
assertApproxEqual( actual, S0, 0.001 );
end
```

Now we know our code works. Let's answer the specific financial question.

```
function answerQuestion()
%Let's answer the question finally
T = 0.25;
r = log(1.05);
S0 = 100;
K = 105;
vol = 0.1;
answer = priceCallByIntegration(K,T,S0,r,vol);
fprintf('The answer is %f dollars\n', answer );
fprintf('The answer to two d.p. is %.2f dollars\n', answer );
end
```

This demonstrates the MATLAB code required to print things more attractively. The `fprintf` function will automatically replace the `%f` symbol with the value of answer. This allows you to mix English sentences with numerical results in a more sophisticated way than is possible just using `disp`. Notice that the `\\n` means "start a newline". The `disp` command always starts a new line, but the `fprintf` command is a bit more flexible and lets you decide if you want a new line.

We won't put a lot of effort into printing things out prettily in Matlab, but it can be useful to use the `fprintf` function if you want to generate more readable output than is possible with `disp`.

What have we gained by pricing the derivative numerically? Well firstly we can easily adapt the code to puts and digital options. More interestingly we can price derivatives with arbitrary payoffs. Even more interestingly given a pricing kernel $q_T(S)$ we can price derivatives using that pricing kernel. For example, you could take a pricing kernel with fat tails and our code would still work.

4.5.1 Monte Carlo integration as simulation

We have the following formula for the price:

$$e^{-rT}\int_0^\infty \text{payoff}(S)q(S)\,\mathrm{d}S$$

where $q(S)$ is the $\mathbb{Q}$-p.d.f. We can make a substitution by letting $Q(S)$ be the $\mathbb{Q}$ cumulative density function and make the substitution $U = Q(S)$.

$$\mathrm{d}U = q(S)\,\mathrm{d}S$$

by definition of the probability density function as the derivative of the c.d.f.. So the price is given by:

$$e^{-rT}\int_0^1 \text{payoff}(Q^{-1}(U))\mathrm{d}U.$$

So the Monte Carlo estimate for the price is:

$$e^{-rT}\frac{1}{N}\sum_{i=1}^{n}\text{payoff}(Q^{-1}(u_i)) \tag{4.8}$$

where the u_n are uniformly distributed random variables on $[0,1]$. This has a very simple interpretation in terms of simulation.

Given any random variable with cdf F, we can simulate that random variable by simulating uniform random variables and then applying F^{-1}. See **??** to see why—it is true simply by definition of the CDF.

This means that we can interpret equation (4.8) as saying that to estimate the risk-neutral price we can:

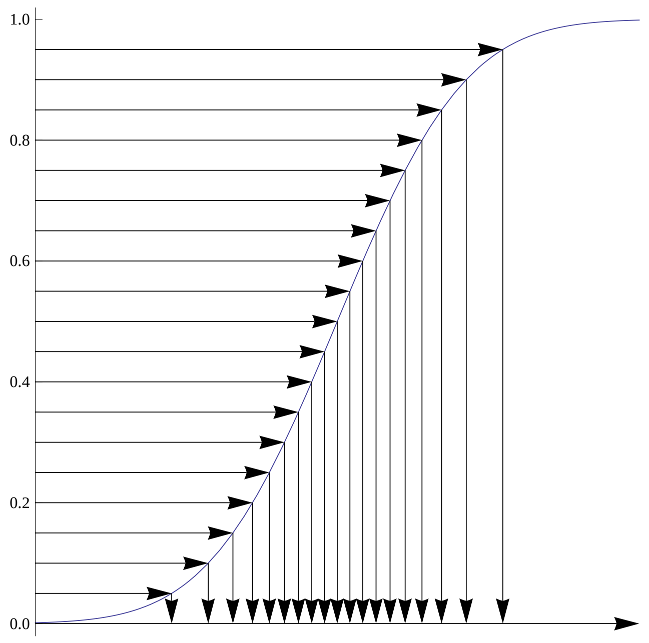


Figure 4.2: Applying the inverse cumulative distribution function F^{-1} to uniform random variables gives a distribution with c.d.f. F.

(i) Simulate stock prices in the Q measure ($Q^{-1}(u_i)$ is a simulated stock price)

(ii) Compute the payoff of the derivative for each scenario (payoff($Q^{-1}(u_i)$) is the payoff in each scenario.

(iii) Take the average payoff and discount.

Thus pricing by Monte Carlo integration is essentially equivalent to simulating in the $\mathbb{Q}$-measure and taking expectations. An advantage of this approach is that computing the q-pdf earlier was a little tedious as we had to compute how the pdf of the normal distribution changed when we changed variable from z to $S = \exp(z)$. It is easier to compute how cdfs transform under a change of variables.

Recall that we showed that for geometric Brownian motion $\mathbb{Q}$ measure

$$S \sim \log \mathcal{N}\left(z_0 + \left(r - \frac{\sigma^2}{2}\right)T, \sigma\sqrt{T}\right)$$

So by definition of the log normal distribution

$$Q(S) = N\left(\frac{1}{\sigma\sqrt{T}}\left(\ln(S/S_0) - \left(r - \frac{\sigma^2}{2}\right)T\right)\right)$$

where N is the cumulative distribution function of the normal distribution with mean 0 and standard deviation 1. Solving the equation $Q(S) = U$ we find that:

$$Q^{-1}(U) = S_0 \exp\left(\left(r - \frac{\sigma^2}{2}\right) T + \sigma\sqrt{T} N^{-1}(U)\right)$$

We can compute N^{-1} using MATLAB's `norminv` function. We can now compute prices by simply using (4.5.1).

Note that although the derivation of the explicit formula for Q is a bit easier than that for q it does have the disadvantage that it contains the function N^{-1}. We're cheating a bit by using Matlab's built-in `norminv` function to compute this. For more general models beyond a log-normal model we might well struggle to compute Q^{-1} in practice. This means that formula (4.1) is still useful in practice.

We will pursue the simulation approach further in the next chapter.

4.6 Further Reading

[1] discusses numerical differentiation and integration in chapters 6 and 7.

If you find the proof using Taylor's theorem difficult consult any real analysis textbook.

Bibliography

[1] George Lindfield and John Penny. *Numerical methods: using MATLAB.* Academic Press, 2012.