Chapter 10

Optimization

10.1 Introduction

Optimization problems occur a lot in finance. We're going to consider two basic types of optimization problem in this chapter.

1. Portfolio optimization
2. Model calibration

In most of this course, we have written our own algorithms and not made much use of Matlab's built in functions. This chapter is different, we will use the built in optimization functions and not worry too much about how they are implemented.

This means that the main challenge will be to write our optimization problems clearly so that we can then use the built on solvers.

The Matlab functions we will use are:

1. `quadprog` to solve quadratic optimization problems. `quadprog` is short for "quadratic programming". For historical reasons, optimization problems are often called "programming" problems. This is a different use of the word "programming" to the more familiar use in the phrase "computer programming". I will use the word "optimization" myself, but it is helpful to know when you read the literature that "linear programming" and "quadratic programming" simply mean solving linear optimization problems and solving quadratic optimization problems.
2. `fminunc` to solve smooth unconstrained optimization problems. That is to minimize $f(x)$ where there are no restrictions on x. The name of the function comes from "**Min**imize the function $f(x)$ with x **unc**onstrained".
3. `fmincon` to solve smooth constrained optimization problems. That is to minimize $f(x)$ where x is restricted to lying in a particular set. The name of the function comes from "**Min**imize the function $f(x)$ with x **con**straints".

All of these functions solve some variant of the problem "Find x to minimize $f(x)$ subject to the constraint $x \in X$ for a given set X." All of the functions above assume that the function f is smooth, other algorithms are available if f is not smooth but we won't need them in this lecture.

All of these functions look for a local minimum only. That is they look for x where ∇f is zero and the matrix of second derivatives is positive semi-definite. This is just like looking for the minimum of a function of one variable by solving for $f'(x) = 0$ and where checking $f''(x) > 0$.

If the function f is convex, then a local minimum will always be a global minimum, so all of the functions above will find the true minimum if you give them a convex problem to solve. If you give them a non-convex problem, you can't guarantee that you will get a global minimum. This is one of the advantages of solving convex optimization problems rather than general optimization problems. Convex problems are much easier to solve than general optimization problems. MATLAB does contain functions that try to find global solutions of general optimization problems, but they don't always work very well in practice.

One interesting point is that it normally doesn't matter very much if we find more than one solution x_1, x_2 to our minimization problem. So long as $f(x_1) = f(x_2)$ they can both be perfectly valid solutions. Similarly, if we are looking for approximate solutions then we can often find x_1 and x_2 with $f(x_1) \approx f(x_2)$ even though x_1 and x_2 might be quite different. This means that when testing if your optimization has been successful by comparing it with another optimization routine, you should check the value of $f(x)$ rather than x itself as two algorithms might return quite different values for x.

10.2 Application: Modern Portfolio Theory

In 1950's, Markowitz developed a very well-known theory which is called "Modern Portfolio Theory" even though by now it is not very modern.

The aim of the theory is to work out how to choose "optimal" investments in various assets, say a collection of n different stocks.

The model that Markowitz used is that there are n assets whose returns over some fixed time period T are random variables R_i (where the index $i \in \{1, 2, \ldots, n\}$ runs over the different assets). The return of an asset is defined by the following equation:

$$R_i = \frac{\text{price of asset } i \text{ at time } T \text{ - current price of asset } i}{\text{current price of asset } i}$$

So the return of an asset is simply a scaled version of the profit and loss that takes into account how much you had to pay for the asset.

The random variables R_i are assumed to have a known mean μ_i. So we have a vector μ of length n containing the means of all the assets. We also know the $n \times n$ covariance matrix Σ for all the assets.

We wish to invest a fixed amount P over the time period T in a fixed portfolio of the assets. The portfolio is determined by choosing the weights w_i to assign

to each asset with $\sum_{i=1}^{n} w_i = 1$.

We will assume that we are allowed to short sell so the w_i may take any real values.

Markowitz's idea was that that optimal portfolio for a given expected return is a portfolio that minimizes the risk for that level of expected return. Markowitz suggested that the risk could be measured using the standard deviation of the portfolio.

If we assume that returns are normally distributed, then if the expected return of the portfolio is fixed, the distribution of the returns is determined entirely by knowing the standard deviation. This means that standard deviation is a good measure of risk for normally distributed assets, indeed it is the only possible measure of risk. However, for more general distributions one can argue that standard deviation is not a particularly good measure of risk. For this reason, it is often said that Markowitz's theory assumes that the returns on assets are normally distributed.

Suppose that we choose a vector $\mathbf{w}$ of weights, then the mean return of our portfolio will be:

$$\mu^{\mathbf{T}}\mathbf{w}$$

and the variance of the return will be:

$$\mathbf{w}^{\mathbf{T}}\mathbf{\Sigma}\mathbf{w}$$

So if we require that the expected return of our portfolio r, the optimal portfolio w can be found by solving the following optimization problem:

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & \mathbf{w}^{\mathbf{T}}\mathbf{\Sigma}\mathbf{w} \\ \text{subject to} \quad & \sum_i w_i = 1, \\ & \mu^{\mathbf{T}}\mathbf{w} = \mathbf{r}. \end{aligned} \tag{10.1}$$

Here Σ is a positive definite matrix containing the covariance matrix of the returns of the assets, μ is a vector containing the mean return of each asset.

This is an example of what is called a quadratic programming problem. Matlab has a built in function `quadprog` that can solve such problems for you. The general quadratic programming problem can be written in the following form:

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & \frac{1}{2}\mathbf{x}^{\mathbf{T}}\mathbf{H}\mathbf{x} + \mathbf{f}^{\mathbf{T}}\mathbf{x} \\ \text{Subject to} \quad & \\ & A\mathbf{x} \leq \mathbf{b} \\ & A'\mathbf{x} = \mathbf{b}' \\ & \mathbf{l} \leq \mathbf{x} \leq \mathbf{u} \end{aligned} \tag{10.2}$$

Here A and A' are matrices, $vecb$, $\mathbf{b}'$, $\mathbf{l}$ and $\mathbf{u}$ are vectors. We are writing $\mathbf{x} \leq \mathbf{y}$ if every entry of the vector x is less than or equal to the corresponding entry in

y. MATLAB allows you to use the values $-\infty$ and $+\infty$ in the lower and upper bounds l and u if you wish.

We will call the constraint $A\mathbf{x} \leq \mathbf{b}$ the "inequality constraint". We will call the constraint $A'\mathbf{x} = \mathbf{b}'$ the "equality constraint" and we will call the constraints $\mathbf{l} \leq \mathbf{x} \leq \mathbf{u}$ upper and lower bounds on x.

Because we are using matrix equations for the constraints, we can combine multiple constraints together into a single matrix equation. For example in the problem (10.1) we have two equality constraints, but they can be combined into the single constraint:

$$\begin{aligned} &A'\mathbf{w} = \mathbf{b}' \text{ where} \\ &A' := \begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ \mu_1 & \mu_2 & \mu_3 & \dots & \mu_n \end{pmatrix}, \\ &b' := \begin{pmatrix} 1 \\ r \end{pmatrix} \end{aligned} \tag{10.3}$$

Thus the problem (10.1) is a special case of the quadratic programming problem with $H = 2\Sigma$, $f = 0$ and with A' and b' as above. There are no inequality constraints or upper or lower bounds.

We can now call `quadprog` with these parameters to compute the optimal portfolio. To solve the problem (10.2), one writes $A' =$ `Aeq` and $b' =$ `beq` then makes the MATLAB call:

```
[x,fVal,exitFlag]=quadprog(H,f,A,b,Aeq,beq,lb,ub,x0,options);
```

When this function returns x will contain the optimal value for x and `fVal` will contain the corresponding mimimum of $\frac{1}{2}\mathbf{x^T H x} + \mathbf{f^T}$. The return value `exitFlag` indicates whether the optimization worked. You must check this is greater than zero whenever you call `quadprog` to be sure that the optimization actually worked.

As you can see `quadprog` takes a lot of parameters. We won't always have inequality constraints, equality constraints or upper or lower bounds. If this is the case you can use the empty array `[]` in place of the constraint in your call to `quadprog`.

The final `options` parameter allows you to fine tune the optimization if you wish. It can be omitted if you are happy with the default options. We will see how to configure options shortly.

But first let us see how to write a function to find the solution of (10.1).

```
function [ ret, variance, w ] = markowitzOptimizeRet(...
     r, mu, sigma )

H = sigma;
n = size(sigma,1);
f = zeros(n,1);
```

```
Aeq = [ ones(1,n); mu];
beq = [1; r];

[w,~,exitFlag] = quadprog(H,f,[],[],Aeq,beq,[],[],[]);
assert(exitFlag>0);
ret = mu * w;
variance = w' * sigma * w;

end
```

The code simply translates the problem (10.1) into the precise format (10.2) that Matlab understands. We compute the required values for H, f, $A' =$ `Aeq` and $b' =$ `beq`. All the other constraint matrices and vectors are empty.

The computation of $A' =$ `Aeq` and $b' =$ `beq` corresponds exactly to our computation (10.3).

If you run the code on the previous slide, MATLAB prints out a lot of messages and (for some versions of MATLAB) a warning. This is rather annoying if you plan to call this function often. To switch off the messages you can customize the options and to switch off the warning you use the `warning` command to indicate that you aren't interested in the specific warning message. These changes are made in the code below.

```
function [ ret, variance, w ] = markowitzOptimizeRet( ...
    r, mu, sigma )
warning('off','optim:quadprog:WillRunDiffAlg');


H = sigma;
n = size(sigma,1);
f = zeros(n,1);
Aeq = [ ones(1,n); mu];
beq = [1; r];

options = optimset('quadprog');
options = optimset(options,'Display','off');
[w,~,exitFlag]=...
    quadprog(H,f,[],[],Aeq,beq,[],[],[],options);
assert(exitFlag>0);
ret = mu * w;
variance = w' * sigma * w;

end
```

It is worth examining this code to see how one creates options for `quadprog`. First we find the default options with a call to `optimset` passing in a single parameter, the name of the optimization routine you will call. This returns an `options` object.

```
options = optimset('quadprog');
```

We then call `optimset` a second time to modify the options. This time we pass in our `options` object and a set of key/value pairs that indicate what options you would like to set. In this example, we're setting the option `Display` to `off`.

```
options = optimset(options,'Display','off');
```

The function `optimset` returns the new modified options object which is equal to the default options modified to switch off the display of information about whether the optimization succeeded.

Having configured the options object, we pass it to the optimization function as the final parameter.

```
[w,~,exitFlag]=...
    quadprog(H,f,[],[],Aeq,beq,[],[],[],options);
```

The reason for this rather complicated procedure is to make sure that you choose the default values for all options except the ones you wish to customize.

All optimization functions take similar options that are configured in this way. You can use options to set things like:

- How much information to print during the calculation (`Display`)
- The actual algorithm to use (`Algorithm`)
- How accurate the answer needs to be
- The maximum number of steps to perform of the algorithm before giving in
- etc.

You should consult the MATLAB documentation for details of the available options and their settings.

10.2.1 An application with real data

Historic stock prices

Let us try to use Markowitz's theory to select an optimal portfolio of FTSE 100 stocks—i.e. to find the portfolio that minimizes standard deviation for a given expected return.

To do this, we will first need to estimate the mean vector μ and the covariance matrix Σ. We can do this using historic data. The file `ukx.xslx` on my website contains data downloaded from Bloomberg for the FTSE 100 index (and then

modified as one shouldn't publish Bloomberg data). It contains weekly prices for each of entry of the FTSE 100 since the year 2000

As is often the case when working with real data, there is some missing data. For a couple of companies there are missing entries. For simplicity we will exclude those from our Portfolio.

The code below reads the excel file and returns a matrix data containing a column of prices for each stock in the FTSE 100 for which we have a full set of prices. To understand the code you will need to look at the spreadsheet of data. Note that only every third column contains a stock price.

```
function data = ukxData( nSecurities )

% Read the raw data from the excel file
bloombergData = xlsread( 'ukx.xlsx', 'A3:KP736' );

% n is the number of securities that we've read
n = min( nSecurities, floor(size(bloombergData,2)/3));

% Now eliminate the empty columns and any data
% for a security where
% we don't have a full history of returns
index = 1;
for i=1:n
    col = bloombergData(:,(i-1)*3+1);
    if (~isnan(col(end)))
        data(:,index)=col;
        index = index+1;
    end
end

end
```

We're using the `isnan` function to detect missing data. If there is no number at the end of the column, some data is missing.

It's quite normal when working with data to have to write a bit of code to read the file. As we've done here, it is best to put this in a function of it's own so it is easy to understand and test.

Estimating μ and σ

To estimate μ and σ we simply compute the weekly returns and hence estimate the sample covariance matrix Σ and the mean weekly vector μ for the FTSE 100. We will then assume that over the next week, these stocks will be distributed with the same return and variance.

```
data = ukxData( nSecurities );
returns = (data(2:end,:) - data(1:end-1,:))./data(1:end-1,:);
```

```
% We assume that historical returns allow us to estimate
% expected return (mu) and covariance (sigma)
sigma = cov(returns);
mu = mean( returns );
```

It is interesting to plot the mean return and standard deviation for each stock for which we have data. This can be done with the following code.

```
% Plot a scatter plot of the return for each consitutent
sds = sqrt( diag( sigma ));
scatter( sds, mu );
xlabel('Standard deviation');
ylabel('Expected return');
```

10.2.2 The efficient frontier

Finally, we would like to compute the solution to (10.1) for various values of r. We will then plot the standard deviation of the optimal portfolio, the result is shown in Figure 10.1. The small circles show the return and standard deviation for each individual stock. The curve shows the optimal standard deviation for each level of return. Each point representing a stock is on the right of the efficient frontier. This is simply because investing in one stock is riskier than investing in the optimal portfolio with the same return. Figure 10.1 gives a visual illustration of the benefits of diversification.

The code used to plot the efficient frontier is shown below. It simply calls our `markowitzOptimizeRet` function repeatedly.

```
% Now compute the efficient frontier and plot it
r = -0.01:0.0005:0.02;
frontierX = zeros(length(r),1);
frontierY = zeros(length(r),1);
for i=1:length(r);
    [ret,var]=markowitzOptimizeRet( r(i), mu, sigma );
    frontierX(i)=sqrt(var);
    frontierY(i)=ret;
end
hold on;
plot( frontierX, frontierY );
title('The Markowitz efficient frontier');
hold off;
```

One interesting point is that we use the functions `hold on` and `hold off`. Sometimes you want to superimpose charts in Matlab. For example, our plot Figure 10.1 shows a scatter plot and a line plot simultaneously. By calling `hold on` you tell Matlab that you want all plotting commands to appear on the same chart. You then cancel this behaviour with a call to `hold off`.

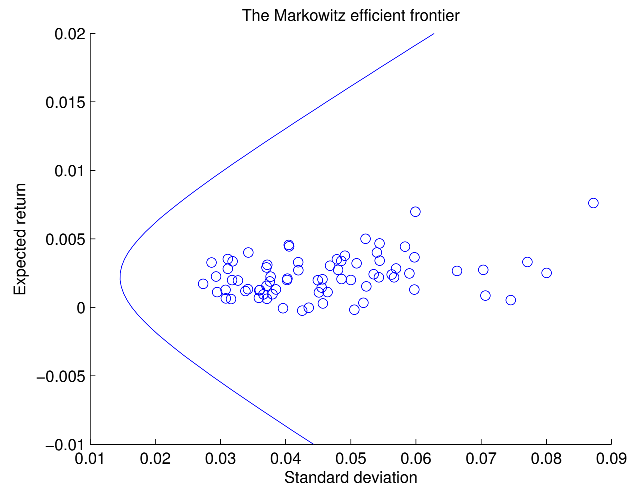


Figure 10.1: The efficient frontier

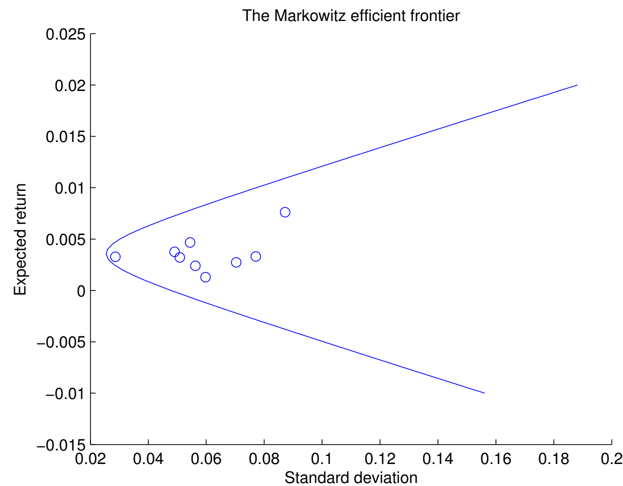
.

Discussion

The Markowitz model is very influential, but you shouldn't use it uncritically.

Firstly, standard deviation is not always a good measure of risk. Using standard deviation to measure risk can be justified for normally distributed assets, but not in general. For more general return distributions it makes more sense to perform utility optimization. We will show how this is done later in the chapter.

In addition, the results we obtained for the FTSE 100 are too good to be true. We've calibrated a 100×100 matrix Σ using historic data and this means that we've over-fit our model. It would be more sensible to use just 10 stocks as illustrated in the figure below.



As you can see, with ten stocks, diversification improves our portfolio but not too such a radical extent.

We have shown how to optimize portfolios in the Markowitz model with no trading constraints. For such problems, the efficient frontier will always be a hyperbola. It is actually quite easy to prove analytic results in this special case. However, for more general problems with constraints using `quadprog` is essential.

10.3 Calibrating pricing models

As is well known, the Black–Scholes model does not fit market data precisely. Recall that the implied volatility is the value of σ you must plug into the Black–Scholes formula to obtain the market price of an option. If the Black–Scholes model was correct, all European options would have the same implied volatility, but in reality, if you plot the implied volatility against the strike of a call option you get a curve called "the volatility smile". This is shown in figure **??**. Note that this picture shows the curve for one maturity, you will get different smiles for different maturities.

There are many factors which explain the inaccuracy of the Black–Scholes model. For example stock returns have fat tails, time series analysis suggests volatility is not constant, market prices sometimes jump etc..

For example, there are stochastic volatility models where the volatility itself follows a stochastic process. One example is the Heston model which we have already met once in this course.

In this section we will consider another type of model, called jump diffusion models. In these models, in addition to the usual white noise that changes the stock price continuously, there is also the possibility of instantaneous jumps in the stock price.

These models are both examples of incomplete market models. In both stochastic volatility models, there is more than one source of randomness effecting the stock price. However, you can only hedge one source of risk by trading in the underlying. As a result, for these models there is more than one equivalent martingale measure compatible with a $\mathbb{P}$-measure model of jump diffusion or stochastic volatility type.

In practice, when working with an incomplete market model one normally chooses a $\mathbb{Q}$-measure model directly without attempting to derive it from a $\mathbb{P}$ measure model. One then uses this to model both the underlying and vanilla option prices as discounted expectations for this measure.

Since we are using a $\mathbb{Q}$-measure model, prices obtained in this way will be arbitrage free. Moreover if option prices and the stock follow the model, it will be possible to find dynamic replication strategies for exotics by trading in both the underlying and in vanilla options.

Thus, in practice one chooses a general form for a $\mathbb{Q}$-measure model and then calibrates this model to match market price reasonably closely. The hope is that one can then offer to buy and sell derivatives and hedge them using both the stock and exchanged traded options.

The process of calibrating is an optimization problem. We are seeking a $\mathbb{Q}$-measure model that gives the best fit to market prices. The word best is the clue that optimization will be involved in calibration.

Although we are giving the example of calibrating a $\mathbb{Q}$-measure model to prices, it is also very normal to calibrate a $\mathbb{P}$-measure model to historic data. This process normally called fitting a statistical model. It too is an optimization problem. For example, one well known example is finding the best linear model to data which is carried out using least squared regression. This is in fact a quadratic optimization problem which could be solved with `quadprog`, although it is simple enough to solve by hand.

10.3.1 A Jump Diffusion model

We will consider the simplest possible jump diffusion model. We will simply write down a Q-measure model and the associated pricing formula for European call options.

Our model will consist of a diffusion process for the stock price with additional jumps that occur occasionally. The jumps will be of a fixed size and occur with a Poisson distribution. This is a very simple jump diffusion model, it is easy to elaborate on this basic model if desired.

We let N_t be an integer valued random process, representing the number of events that have occurred for a Poisson process with intensity λ. We then suppose that when a jump occurs, the stock price jumps from S_t to JS_t for some $J > 0$. So $J < 1$ represents a fixed size downward jump and $J > 1$ represents a fixed upward jump. Between jumps the stock price obeys the standard geometric Brownian motion

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

For this to be a valid $\mathbb{Q}$ measure model, we will need to make sure that $\exp^{-rT} S_t$ is a martingale.

Fortunately it is easy to calculate expectations of payoffs at time T. Let N_T be the number of jumps that have occurred up to time T. Let $\overline{S}_T$ follow the process:

$$\mathrm{d}\overline{S}_t = \overline{S}_t(\mu \mathrm{d} + \sigma \mathrm{d}W_t).$$

without any jumps, then $S_T = J^{N_T}\overline{S}_T$. So the expectation conditioned on the number of jumps being N_T can be calculated using the expectation in the Black-Scholes model but with a starting price of $J^{N_T} S_0$.

As an example, we compute the expectation of the stock price at time T:

$$\begin{aligned} E_{\mathbb{Q}}(S) &= \sum_{i=0}^{\infty} P_{\mathbb{Q}}(N_T = i) J^i e^{\mu T} S_0 \\ &= \sum_{i=0}^{\infty} e^{-\lambda} \frac{\lambda^i}{i!} e^{\mu T} J^i S_0 \\ &= e^{J\lambda} e^{-J} e^{\mu T} S_0 \\ &= e^{(J-1)\lambda + \mu T} S_0 \end{aligned}$$

So this is a Martingale if and only if

$$\mu = -(J-1)\lambda + r.$$

We can also write down the price of a European call in our jump diffusion model it is:

$$\sum_{j=0}^{\infty} e^{-\lambda T} \frac{(\lambda T)^j}{j!} \mathrm{BS}(K, T, S_0 e^{(\mu - r)T} J^j, r, \sigma) \tag{10.4}$$

Here $\mathrm{BS}(K, T, S_0, r, \sigma)$ is the usual Black Scholes pricing formula.

Note that we are not claiming that fixed size jumps and a constant intensity is a particularly plausible model for jumps in a stock price, we've chosen this model simply because it gives an example of an incomplete market model where we can easily price call options.

Let us write the MATLAB code to price an option in our jump diffusion model. Since (10.4) is an infinite sum, we can't evaluate every term in the sum. We will use a `while` loop to evaluate terms in the sum until they contribute less than one part in a million to the sum. This gives rise to the following code:

```
function total = jumpDiffusionPrice( ...
    K, T, S0, r, sigma, lambda, J )
total = 0.0;
coefficient = exp( - lambda*T );
j = 0;
mu = -(J-1)*lambda + r;
while true
    term = coefficient * blackScholesCallPrice(...
```

```
        K, T, S0*exp((mu-r)*T)*J^j, r, sigma );
    total = total + term;
    if (abs(term)/abs(total)<1e-6)
        return;
    end
    j = j+1;
    coefficient = coefficient * lambda*T / j;
end
end
```

10.3.2 Calibrating a Jump Diffusion model

Having written down our jump diffusion model, we now consider how to choose the parameters σ, λ and J. We wish to calibrate the model to market option prices. In other words we wish to find the values of the parameters that give the "best fit" to market prices.

It is up to use to define what "best fit" actually means. We need to choose some metric to measure the error of the fit. We can then use an optimization algorithm to minimize the error.

We will define the error to be the sum of the squared errors when comparing the predicted price of an option with the market price. Using squared errors rather than absolute values ensures our error function is smooth. It is perfectly possible, and legitimate, to use other measures of "best fit", we have simply made a simple and convenient choice.

So let us suppose that we have n exchange traded call options all with maturity T. Option i has strike K_i. Let C_i denote the market price of the option i. We will then let $C'_i(\sigma, \lambda, J)$ denote the jump-diffusion predicted price with the given parameters. We define:

$$\text{error}(\sigma, \lambda, J) = \sum_i (C_i - C'_i(\sigma, \lambda, J))^2 \tag{10.5}$$

The calibration problem is to minimize error subject to the constraints

$$\sigma > 0, \lambda > 0, J > 0.$$

To eliminate the constraints we write $\sigma = e^{x_1}$, $\lambda = e^{x_2}$, $J = e^{x_3}$ we can convert this to an unconstrained optimization problem of choosing (x_1, x_2, x_3) to minimize the error.

Matlab provides a function `fminunc` for unconstrained optimization. To use `fminunc` you must first write your objective function, that is the function to minimize. This function just should take a vector of parameters and return a single real number.

In our case we will want to write a function to compute "error" given the 3-vector of parameters $x = (x_1, x_2, x_3)$.

Assuming that you have written a function f that computes your objective, you call `fminunc` as follows:

```
[x,fVal,exitFlag]=fminunc( @f, x0, options );
```

In this call `x0` is an initial guess at the solution. `x`, `fVal` and `exitFlag` and `options` have the same meaning as for `quadprog`. Just as for `quadprog` it is important to check the exit flag to make sure the optimization has succeeded. Just as for `quadprog` you can provide options if you want to tune the optimization, but you don't have to.

One important ddifference is that the quadratic program we solved was convex. That is we wished to minimize a convex function defined over a convex domain. Convexity ensures that the optimization problem is "nice". For example a local minimum of the function is an absolute minimum.

The calibration problem we have written down is smooth, but non-convex. As a result `fminunc` will only find a local minimum of the problem. We'll just have to hope that the error achieve for our fit is small, we can't guarantee that it is the best possible fit.

10.3.3 Implementation

To calibrate a model, we need the market price of the options. The file `goog-options.xslx` contains option data for Google taken from Bloomberg on 20 March 2014 (and then modified slightly as one shouldn't publish actual Bloomberg data).

This file contains strikes and mid prices for numerous options all with the same maturity date of 19 April. On that day the stock price was 1205.415. We will also assume a risk free rate of 0.16. The time to maturity is 31/365.

The code below reads in this market data and returns the various values as MATLAB arrays and variables.

```
function [strike,T,S0,r,mid] = googOptionData()

% Read the raw data from the excel file
T = 31/365;
S0 = 1205.4;
r = 0.16/100;

bloombergData = xlsread( 'goog-options.xlsx', 'C5:D54' );
mid = bloombergData(:,1);
strike = bloombergData(:,2);

end
```

The main task in performing the optimization is to implement the objective function. We write this as a nested function called `errorFunction` inside of our main function `calibrateJumpDiffusion`.

The implementation of the objective function follows from our definition of error given in (10.5). The code simply sums the squares of each individual error in a `for` loop.

```
function [ sigma, lambda, J ] = calibrateJumpDiffusion()
[strike,T,S0,r,mid] = googOptionData();

function e = errorFunction( x )
    sigma = exp(x(1));
    lambda = exp(x(2));
    J = exp(x(3));
    fprintf( 'Sigma=%d, lambda=%d, J=%d\n',sigma,lambda,J );
    e = 0;
    for i=1:length(strike)
        K = strike(i);
        p1 = jumpDiffusionPrice(K,T,S0,r,sigma,lambda,J);
        p2 = mid(i);
        e = e + (p1-p2)^2;
    end
end
```

Now that we have written the objective function, we simply need to call `fminunc`.

```
% Note that we may only find a local minimum, so
% a good choice of x0 is important
x0 = [ log(0.25), 0.3,0.9 ];

[xOpt,~,exitFlag] = fminunc( @errorFunction, x0 );
assert(exitFlag>0);

sigma = exp( xOpt(1) );
lambda = exp( xOpt(2) );
J = exp( xOpt(3) );

end
```

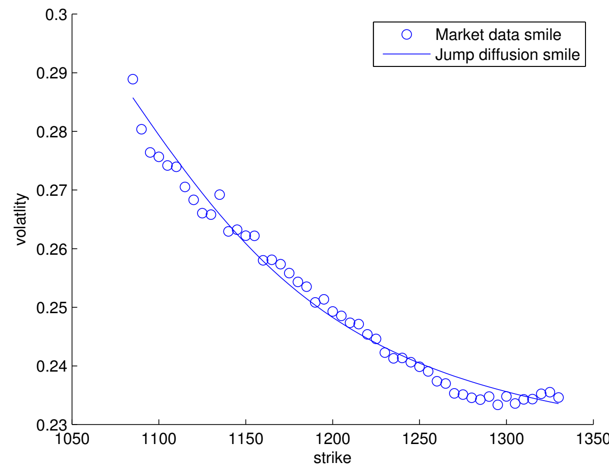
Note that we check the exit flag to ensure that the optimization was successful.

Having calibrated the model, we can plot a graph of the prices predicted by the model against actual market prices. The code to plot the graph and to compute the implied volatilities from market data is left for you to implement yourself. See Section 10.3.3 for the final result.

10.3.4 Using fminunc

Using `fminunc` seems very easy, but there are potential pitfalls to watch out for.

Firstly, as we have mentioned `fminunc` only finds local minima. Only if your problem is convex will it find global minima.



Secondly, the numerical algorithms assume that the problem is centered and scaled. You may recall that the `integral` function can give the wrong answer if you do not centre and scale the function well. For example, if you ask it to compute the integral of the pdf of the normal distribution with a mean of 1000 and a standard deviation of 0.0001, it will fail. This is because the "center" of this function is roughly at 1000 and the scale is roughly 0.0001. A function is well centered and scaled if the interesting parts of the function are near the origin and the major changes in the function's size occur in a region of roughly size 1.

The reason that this is necessary is that the numerical methods used by $fminunc$ use discrete approximations to derivatives and differential equations. They choose the step sizes in these approximations based on an assumption of good scaling.

The algorithm used by $fminunc$ is (roughly) to estimate the first and second derivatives of the objective function near x_0. By Taylor's theorem we can approximate f using a quadratic function with coefficients given by these derivatives. By solving the optimization problem for this quadratic approximation using `quadprog` we get new estimate x_1. Continuing in this way, we get a sequence of approximations. This method is called the Newton method. We terminate the search for a solution once the gradient is reasonably close to zero.

There are many options you can configure when using `fminunc`. For example:

- When estimating derivatives what value should we use for δx?
- When do we consider the gradient to be sufficiently close to zero?
- How many steps of the algorithm should we perform before giving in?

- Should we use a different algorithm entirely (e.g. for large scale problems where x is high dimensional)?

Although there are many options, typically the best approach is to ensure that your problem is centred and scaled and then use the defaults.

10.3.5 Remarks on calibration

Our choice of jump diffusion model is very crude. Although it fit the market prices rather well for a single maturity date, it probably wouldn't fit the full pricing surface at different maturities very well.

We chose this model mostly because it is easy for us to price rapidly. If you run the optimization, you will see from the `fprintf` statement that it is called extremely often. Each call to this function then prices every single derivative. This means that to calibrate a model it is essential that you can price exchange traded options quickly.

This explains why some probabilistic models are more popular than others for pricing derivatives. An arbitrary model can probably only be priced using Monte Carlo which is very slow. For example, the Heston model is popular because it can be priced reasonably rapidly using Fourier transform methods. (We have not covered these methods in this course.)

We should emphasize again that the parameters found by fitting are for a risk neutral model and not the real world model. This means that the model tells us as about market beliefs and risk preferences rather than about actual probabilities.

10.4 Utility optimization

Markowitz's model of portfolio optimization has been very influential, but it is rather limited by the assumption that we use standard deviation to measure risk.

It makes considerably more sense to solve the problem of finding a portfolio that maximizes expected utility. If assets are normally distributed, this will always give a portfolio on the Markowitz initial frontier. However, unlike Markowitz's theory, utility optimization has a sound logical justification whatever the distribution of asset returns.

To write down a utility optimization problem mathematically, we suppose that the market contains n assets X_1, X_2, $\dots X_n$ and we have a stochastic model for the price of these assets.

There may be various constraints on our trading — for example we might only be allowed to buy or sell a certain quantity of each asset. We suppose that we have a utility function $u : \mathbb{R} \longrightarrow \mathbb{R}$ which maps the final value of our portfolio to our utility. We wish to find quantities q_1, q_2, $\dots$, q_n of each asset

to buy subject to our constraints in order to maximize our expected utility.

$$\text{maximize } E(u(\sum_{i=1}^{n} q_i X_n))$$

Since MATLAB has built in functions for solving minimization problems, we rewrite this as the problem of minimizing the disutility.

$$\text{minimize } E(-u(\sum_{i=1}^{n} q_i X_n))$$

We recall that a utility function is a concave function. Popular examples are:

- power utility with risk aversion parameter η

$$u(x) = \begin{cases} \frac{x^{1-\eta}-1}{1-\eta} & \eta \neq 1, \quad x > 0 \\ \ln(x) & \eta = 1, \quad x > 0 \\ -\infty & x \leq 0 \end{cases}$$

- exponential utility with risk aversion parameter λ

$$u(x) = 1 - e^{-\lambda x}$$

Note that power utility assigns infinite negative utility to losing money. This means that only trading strategies that have 0 chance of bankruptcy will ever be considered. So using power utility implies prohibiting short selling.

MATLAB has a built in function `fmincon` for optimization with constraints. Just as with `fminunc`, the main task we have to perform is to implement the objective function. This means computing the expected utility of a portfolio given the quantities of each asset.

Since computing expected utility is just a matter of computing an expectation, we can use all the techniques we have developed for computing expectations.

The most general approach we have seen in this course for calculating expectations it the Monte Carlo method. When applied to this problem, this means we should: first simulate all our assets using Monte Carlo; then given a vector of quantities of each asset we can compute the payoff of our portfolio in each scenario; the mean payoff is an estimate for the expected utility. All the theory of Monte Carlo integration holds in this case. For example: we can estimate a confidence interval using the central limit theorem; we can attempt to improve accuracy using antithetic sampling; we can apply all the variance reduction techniques described in the next chapter.

Occasionally one might also be interested in low dimensional problems. For example you might be interested in choosing the optimal portfolio of options on a single stock S with all the options maturing at time T. In this case we only

need to consider the one underlying random variable S_T. We can then compute expectations using low-dimensional integration methods such as the rectangle rule, Simpson's rule etc.

Assuming that we decide to use the Monte Carlo approach, this shifts the main problem we must solve to be that of simulating all of the assets. We have covered this extensively in this course. In particular:

- If the assets follow multivariate Brownian motion, they can be simulated in one step using Cholesky decomposition.
- If the assets follow multivariate geometric Brownian motion, they can be simulated in one step by simulating the log process and then exponentiating.
- For general stochastic models, you can use the Euler scheme.

So far we have been discussing the problem of finding optimal quantities for a portfolio when one is not allowed to rebalance your portfolio at intermediate times. We have been insisting that one follows a buy-and-hold strategy.

The general problem of finding the optimal trading strategy when one is allowed to rebalance the portfolio at each time step is an example of a "dynamic programming problem". We have been considering "static programming problems" where one does not dynamically change the portfolio.

Solving dynamic programming problems is very tricky. Although some numerical methods exist, they are not always very effective, especially for large problems. This is beyond the scope of the course. However, we will see that one can use `fmincon` to help us find good solutions to dynamic programming problems, so long as we don't insist on finding genuinely optimal solutions.

Suppose that one has a fixed number of trading strategies: S_1, S_2, $\ldots$, S_n. One could then ask what would happen if one followed a linear combination of strategies $\alpha_1 S_1 + \alpha_2 S_2 + \ldots \alpha_n S_n$. For example: strategy S_1 might be "buy one Google stock". Then strategy $\alpha_1 S_1$ would be "buy α_1 Google stocks". A more sophisticated strategy S_2 might be "sell one call option at \$100 and delta hedge it". Then $\alpha_2 S_2$ would require selling α_2 call options for the same price and delta hedging them.

In effect each investment strategy S_i can be thought of as an asset in it's own right, so there isn't any mathematical difference between optimizing a portfolio over a set of strategies and optimizing over a set of assets.

We can solve for the optimal choices of α_i using Matlab's built-in function `fmincon` for constrained optimization.

To do this we will calculate N possible scenarios for the asset prices. We will then compute the profit or loss of each strategy in each scenario. This will give a vector $x^{(i)}$ of profit and loss for each strategy with rows corresponding to each scenarios.

Note that it is crucial to use the same scenarios for each strategy. This is because we want to compute the profit and loss of our entire portfolio for each

scenario, so we clearly need to use the same scenarios for each strategy. The profit and loss of the combined strategy in each scenario is

$$\sum_i \alpha_i x^{(i)}$$

Using the Monte Carlo approximation for the expected utility, we can approximate our optimization problem as

$$\begin{aligned} &\underset{\alpha}{\text{minimize}} && -\frac{1}{N}\sum_{j=1}^{N} u(\sum_{i=1}^{n} \alpha_i x_j^{(i)}) \\ &\text{(subject to any constraints)} \end{aligned}$$

In this formula, the j is running over N scenarios. i is running over the n strategies.

To be concrete, let us assume that we have some constraints of the form

$$A\alpha \leq b$$

and

$$l \leq \alpha \leq b$$

and that our utility function is exponential utility with parameter λ.

Since the variable name x is rather over-used we will write `pnlArray` for the corresponding variable in our MATLAB code. The rows of j will correspond to scenarios, the columns to strategies. Thus the problem we wish to solve is:

$$\begin{aligned} &\underset{\alpha}{\text{minimize}} && -\frac{1}{N}\sum_{j=1}^{N} u\left(\sum_{i=1}^{n} \alpha_i \texttt{pnlArray}_{ji}\right) \\ &\text{subject to} && A\alpha \leq \mathbf{b} \\ & && l \leq \alpha \leq \mathbf{u} \\ &\text{where} && u(x) = 1 - \exp^{-\lambda x} \end{aligned} \tag{10.6}$$

So long as the utility function is smooth and concave, this will be a smooth convex optimization problem.

With the problem written in this form, we can perform the optimization using `fmincon`. This function behaves like `fminunc` in that you must provide an objective function as the first parameter. However, it also allows you to specify additional constraints in much the same way as we did with `quadprog`. The resulting code to solve (10.6) shown below.

```
function [ alpha ] = optimizeUtility( ...
    lambda, ...
    pnlArray, ...
    A, ...
    b, ...
```

```
    lb, ...
    ub)

function d = expectedDisutility( alpha )
    pnl = pnlArray*alpha;
    u = mean( 1 - exp(-lambda*pnl));
    d = -u;
end

nStrategies = size(pnlArray,2);
alpha0 = zeros(nStrategies,1);
alpha0(1)=1;
options = optimset('fmincon');
options = optimset(options,...
    'Display','off', 'Algorithm', 'active-set');
[alpha,~,exitFlag] = fmincon( ...
    @expectedDisutility, alpha0, A,b,[],[],lb,ub,[],options );
assert( exitFlag>0 );

end
```

Thus the problem of finding the optimal proportions to invest in assets, or of finding the optimal linear combination of strategies can be reduced to the problem of simulating `pnlArray`.

We summarize the key features of `fmincon` below:

- `fmincon` is MATLAB's function for constrained nonlinear optimization
- It takes similar parameters to both `quadprog` and `fminunc`.
- Just like `fminunc` you specify the objective function by creating an appropriate MATLAB function. In this case we use the exponential utility function to compute the objective.
- Just like `quadprog` you specify constraints using various matrices and vectors such as `A` and `b`.
- `fminunc` also allows you to specify a general non linear constraint function if necessary.
- The same considerations of centering, scaling, smoothness and non-uniqueness apply to `fmincon` as apply to `fminunc`.

10.4.1 Application to hedging

As a concrete example, we will consider a market which a single underlying assets X_1. X_1 is a stock and it follows geometric Brownian motion. X_2 is a call option on the stock. We know the payoff at maturity and we know that we

have a customer who is willing to buy one call option today at the Black Scholes price plus a small commission.

We have the following constraints on our trading:

(i) We can sell up to one unit of the call option at time 0. We can't buy the call option. We can't trade in the call option at other times.

(ii) We can only trade in the stock at 20 evenly spaced time points. i.e. continuous time trading is not possible.

This is a dynamic programming problem and calculating the optimal trading strategy would be rather difficult. However we can consider the following strategies:

(i) Strategy S_1: delta hedging. Sell the customer the option, delta hedge at each time time point (and then pay out as required option).

(ii) Strategy S_2: no hedging. Sell the customer the option and don't bother hedging.

(iii) Strategy S_3: stop-loss hedging. Sell the customer the option, perform stop-loss at each time time point.

(iv) Strategy S_4: trade in the stock alone. Borrow money to buy the stock, wait till maturity and then sell the stock.

We can then form linear combinations of strategies. You can think of the strategy $\alpha_i S_i$ as having four traders and instructing trader i to follow strategy i (scaled up or down by a factor of α_i) and then see what the net effect is.

We have already seen how to reduce the problem of finding the optimal strategies to simply calculating the profit and loss of each strategy for each scenario. For our concrete example, we did this for all the strategies except investing in the stock the chapter on delta hedging, so we can simply re-use this code. Thus computing the optimal combination of strategies can be left as an exercise. You should use the following concrete market parameters:

- S follows the Black Scholes model with $K = 100$, $S = 100$, $T = 0.5$, $r = 0.03$, $\mu = 0.2$, $\sigma = 0.2$
- The customer is willing to pay 1.1 times the Black–Scholes predicted price for the call option.

We also need to write our constraints explicitly. The customer is only willing to buy up to one call option. We cannot sell call options. This gives constraints:

$$
\begin{aligned}
\alpha_1 &\geq 0 \\
\alpha_2 &\geq 0 \\
\alpha_3 &\geq 0 \\
\alpha_1 + \alpha_2 + \alpha_3 &\leq 1
\end{aligned}
$$

We can write the last equation in matrix form as

$$\begin{pmatrix} 1 & 1 & 1 & 0 \end{pmatrix} \alpha \leq 1$$

and the first three can be written together as the bound.

$$\begin{pmatrix} 0 \\ 0 \\ 0 \\ -\infty \end{pmatrix} \leq \alpha.$$

We can now solve the optimization problem by calling `optimizeUtility` and passing in the appropriate `pnlArray` and constraint vectors and matrices. The result will depend upon the choice of risk-aversion parameter λ

In Figure 10.2 we have plotted the values of each α_i against the risk aversion parameter λ. What we see is that for low risk-aversion, we actually prefer to invest in the stock (which is a risky strategy) rather than sell the option and delta hedge. For moderate risk-aversion we enjoy the relatively low-risk option of selling an option at a mark-up and then hedging away the risk. For high risk aversion, the fact that discrete time delta-hedging is still a risky strategy means that we become reluctant to delta-hedge as λ increases.

This trick of finding the optimal linear combination of some fixed set of strategies rather than attempting to solve the full dynamic programming problem is called "the Galerkin method". The idea of using it in this way comes from the paper Koivu & Pennanen: "Galerkin methods in dynamic stochastic programming".

The combined strategy we get from this method is unlikely to be a perfectly optimal solution to the problem, but it is guaranteed to be at least as good as any individual strategy. Thus the Galerkin method allows you to improve performance by diversifying over strategies. This generalizes the notion of diversification for static trading strategies to dynamic trading strategies.

We should emphasize that this method can be applied equally well to static trading strategies. This allows us to optimize static trading strategies even when Markowitz's assumptions do not hold.

We have plotted the quantities of each strategy in our diagram. In a fuller report, it is likely that you would want to report the expected utility of the strategy. If you want to do this, note that the returned `fval` will not be an unbiased estimate of the disutility of performing our strategy. To estimate the utility correctly, you must try the portfolio on a new random sample of scenarios. In general you should always test a strategy on out-of-sample data if you have used a Monte Carlo method to find an optimal strategy.

Although we have found a static strategy, you could decide to re-run the Galerkin method once a day to find a new strategy which incorporates the information received since the previous time step. This would give an improvement on using the Galerkin method once at the beginning and never looking at how the market has changed.

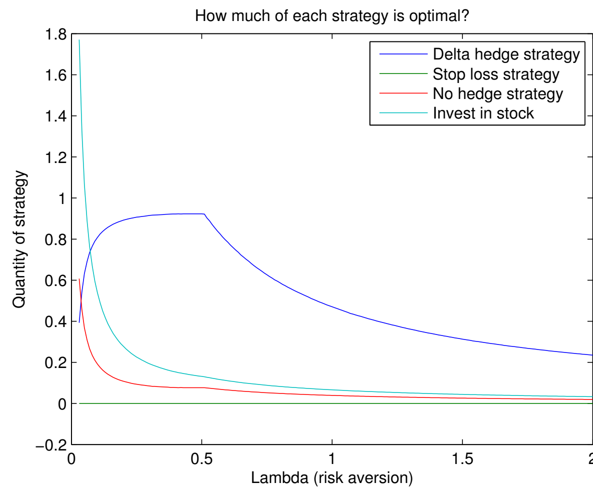


Figure 10.2: The coefficients of the optimal Galerkin strategy for our hedging problem

The power of this method is that it can be generalized easily. For example, to include transaction costs in the model you would simply need to change the computations of the profit and loss vectors to take this into account. Similarly, you can easily use a different model to generate stock paths. You can use any model at all, or historic data, to simulate price paths and hence compute profit and loss vectors. It is also simple to add in other strategies, assets etc.

In summary, although the Galerkin method does not find the true optimal solution to the problem, it does allow us to find improved solutions easily. Moreover it can be applied very generally.

10.5 Further Reading

The classic paper on Portfolio Selection is [3]. I have written my own geometric account of the same theory [1].

The Galerkin method used here is based on [2].

10.6 Summary

We have seen how `quadprog`, `fmincon` and `fminunc` can be used to perform:

- Static portfolio optimization
- Model calibration
- Dynamic portfolio optimization (via the Galerkin method)

Bibliography

[1] J. Armstrong. The Markowitz category. https://arxiv.org/abs/1611.07741, 2017.

[2] Matti Koivu and Teemu Pennanen. Galerkin methods in dynamic stochastic programming. Optimization, 59(3):339–354, 2010.

[3] Harry Markowitz. Portfolio selection. The journal of finance, 7(1):77–91, 1952.