Queens College, CUNY, Department of Computer Science

Computational Finance

CSCI 365 / 765

Fall 2017

Instructor: Dr. Sateesh Mane

1 Homework set 1

1.1 Future value

- Here is a C++ function which inputs (i) today's cashflow F_0, (ii) today's time t_0, (iii) future time t_1, (iv) continuously compounded interest rate r. The value of r is expressed as a percentage, if the interest rate is 5% then $r = 5$.

  ```
  double future_value(double F0, double t0, double t1, double r)
  {
    double r_decimal = 0.01*r;
    double F1 = F0*exp(r_decimal*(t1-t0));
    return F1;
  }
  ```

- **Compile and run this for yourself (you will need to write a main program).**
- Try a few input values. You should be able to implement a similar calculation in Excel and get the same answers.
- I say "future value" but note that the function will work even if $t_1 < t_0$.
- Sometimes when we need to baseline a set of cashflows to a common point in time, some cashflows may be in the past.

1.2 Discount factor

- **Write a function to do the inverse calculation.** (This should be easy.)
- The inputs are (i) today's cashflow F_0, (ii) future cashflow F_1, (iii) today's time t_0, (iv) future time t_1. The outputs are (v) discount factor d, (vi) continuously compounded interest rate r. As above, the value of r should be expressed as a percentage, if the interest rate is 5% then $r = 5$.
- The function signature is

```
int df_and_r(double F0, double F1, double t0, double t1, double & df, double & r);
```

- The return type is "int" because we want some validation checks.
- If $t_1 - t_0$ equals zero, then set $d = 0$ and $r = 0$ and exit with a return value -1.
- If $F_0 \leq 0$ or $F_1 \leq 0$, then set $d = 0$ and $r = 0$ and exit with a return value -2.
- If everything is fine, then exit with a return value 0.
- Hence your function should look like this

```
int df_and_r(double F0, double F1, double t0, double t1, double & df, double & r)
{
  if (t1-t0 == 0.0) {
    df = 0;
    r = 0;
    return -1;
  }
  if ((F0 < 0.0) || (F1 < 0.0)) {
    // *** you figure it out ***
  }
  // *** you have to write the rest ***

  return 0;
}
```

1.3 Bond price from yield

- We begin with the simple case where we know the bond yield and wish to calculate the bond price.
- We also begin with the simple case of a newly issued bond.
- Recall the formula is
$$B = \frac{\frac{1}{2}c}{1+\frac{1}{2}y} + \frac{\frac{1}{2}c}{(1+\frac{1}{2}y)^2} + \cdots + \frac{\frac{1}{2}c}{(1+\frac{1}{2}y)^{n-1}} + \frac{F+\frac{1}{2}c}{(1+\frac{1}{2}y)^n} \,.$$
- **Write a function to calculate the above sum.**
- The inputs are (i) double F, (ii) double c, (iii) double y, (iv) int n. The output is (v) double & B.
- The input value of the yield y is a percentage, so if the yield is 5% then $y = 5$. Hence remember to compute an internal variable $y_{\rm decimal} = 0.01 * y$ in your code, to avoid "factor of 100" errors.
- The function signature is

```
void price_from_yield(double F, double c, double y, int n, double & B);
```

- Write the function and call it with some sample inputs. (You must write a main program.)
- **Here are some tips to help you to check that your code is working correctly.**
- To keep things simple, use $F = 100$ in all your tests. There is no point in being too clever. If $F == 100$, then if the yield equals the coupon $y = c$, you should obtain $B = F$ $(= 100)$.
- Put $y = 0$. Then the value of B is a straight sum of the values of the cashflows. Since there are n cashflows of the coupons, obviously
$$B = F + \frac{nc}{2} \,.$$
Your program should give this value.
- The bond price B decreases as the yield y increases. (This is in fact a general theorem. It was proved in the 1930s, I think.)
- Put $c = 0$. This is known as a **zero coupon bond** and they do exist. A zero coupon bond pays only one cashflow, which is to pay the face value at maturity. In that case the formula is
$$B_{\text{zero coupon}} = \frac{F}{(1+\frac{1}{2}y)^n} \,.$$
This is a very simple formula and you should be able to calculate the above formula independently (use Excel, for example). Hence you should be able to validate your function, for a zero coupon bond.

1.4 Yield from bond price

- *This is the hard calculation. But it is important.*
- In practice, we observe the market prices of bonds and we have to calculate their yields.
- Let us confine our attention to the simple case of a newly issued bond.
- We shall employ a bisection algorithm to calculate the yield, given an input market price for a bond.
- From the previous problem, you have a function to calculate the bond price given the yield.
- We shall employ this function in an iteration loop.
- The inputs are (i) double F, (ii) double c, (iii) int n, (iv) double $B_{\rm market}$, (v) double `tol`, (vi) int `max_iter`. The output is (vii) double & y.
- The market bond price be $B_{\rm market}$. This is the input target value for our bisection algorithm.
- We also need an input tolerance parameter `tol`. Remember that a bisection algorithm needs a cutoff parameter to stop iteration.
- As a safety check, let us also input an upper limit `max_iter` on the number of iterations, in case the computations take too long.
- The function signature is

  ```
  int yield_from_price(double F, double c, int n, double B_market,
                       double tol, int max_iter, double & y);
  ```

  The return type is "`int`" not `void`, because the bisection algorithm might not converge. If it succeeds, we return 0. It it fails, we return 1.

The following pseudocode describes the steps. You can use it as the basis to write a working function.

1. We know that the bond price decreases as the yield increases. We also know that the bond price is a continuous function of the yield. These are both important useful pieces of information.
2. We select an initial "low yield" of $y_{\rm low} = 0.0$ and an initial "high yield" of $y_{\rm high} = 100.0$, which we hope will bracket the true yield.
3. Set $y_{\rm low} = 0.0$ and calculate an output $B_{\rm y\_low}$ using `price_from_yield(F, c, y_low, n, B_y_low)`.
4. If $|B_{\rm y\_low} - B_{\rm market}| \leq$ `tol`, *then we are done.* Set $y = y_{\rm low}$ and a function return value of 0 (success).
5. We expect $B_{\rm y\_low}$ to be larger than the target value $B_{\rm market}$, so if $B_{\rm y\_low} < B_{\rm market}$, it means even a yield of zero is too high. Set $y = 0$ and a function return value of 1 (fail).

6. Set $y_{\text{high}} = 100.0$ and calculate an output $B_{\text{y\_high}}$ using `price_from_yield(F, c, y_high, n, B_y_high)`.

7. If $|B_{\text{y\_high}} - B_{\text{market}}| \le$ `tol`, *then we are done.* Set $y = y_{\text{high}}$ and a function return value of 0 (success).

8. We expect $B_{\text{y\_high}}$ to be smaller than the target value B_{market}, so if $B_{\text{y\_high}} > B_{\text{market}}$, it means even a yield of 100% is too low. Set $y = 0$ and a function return value of 1 (fail).

9. If we have made it this far, then we know that we have bracketed the answer, because $B_{\text{y\_low}} > B_{\text{market}} > B_{\text{y\_high}}$. Hence we know that the true yield y lies somewhere between y_{low} and y_{high}.

10. Now we begin the bisection loop. We know the bond price is a continuous function of the yield, so it will not blow up to infinity or other nasty behavior.

11. Write a loop "`for (i = 0; i < max_iter; ++i)`" to avoid looping to infinity.

12. In the loop, set $y = (y_{\text{low}} + y_{\text{high}})/2.0$ and calculate an output B using `price_from_yield(F, c, y, n, B)`.

13. If $|B - B_{\text{market}}| \le$ `tol`, *then we are done.* We have found a "good enough" value for y. Set a function return value of 0 (success).

14. Else if $B > B_{\text{market}}$, then the value of y is too small. Hence set $y_{\text{low}} = y$.

15. Else obviously $B < B_{\text{market}}$, so the value of y is too large. Set $y_{\text{high}} = y$.

16. *Don't be in rush to iterate!* If $y_{\text{high}} - y_{\text{low}} \le$ `tol`, *then this is good enough.* The algorithm has converged. Set a function return value of 0 (success).

17. If we have come this far, continue with the iteration loop.

18. If we exit the iteration loop after `max_iter` steps and the calculation still has not converged, then set $y = 0$ and set a function return value of 1 (fail). This can happen if the tolerace parameter `tol` is too small. We want to avoid looping too many times.

19. We have reached the end of the function. By now either we have a "good enough" answer (return value = 0 = success) or not (return value = 1 = fail).

20. Test your function with a few simple cases. (You will have to write a main program to call your function.) Remember always use $F = 100$ to keep things simple.

21. Try an input $B_{\text{market}} = 100$. If your function works correctly, it should output $y = c$ (up to the tolerance), for any value of c. Remember that if $F == 100$ and $y == c$ (the yield equals the coupon), then $B == 100$. The converse also holds true.

22. If $B_{\text{market}} < 100$ then your output should be $y > c$. If $B_{\text{market}} > 100$ then your output should be $y < c$.

23. Put $c = 0$. Recall that for a zero coupon bond, we have

$$B_{\text{zero coupon}} = \frac{F}{(1 + \frac{1}{2}y)^n} \,.$$

Hence if $y > 0$ then we must have $B < F$. Hence if you input $B_{\text{market}} > 100$, your function should exit with $y = 0$ and a return value of 1 (fail).