Queens College, CUNY, Department of Computer Science
Computational Finance
CSCI 365 / 765
Fall 2017
Instructor: Dr. Sateesh Mane

2 Homework set 2

Previously we calculated the bond price from the yield using bisection. Here we shall calculate the bond price from the yield using the Newton-Raphson and secant methods. All three methods should yield the same results, up to tolerance. We shall still consider only a newly issued bond.

2.1 Bond price from yield

- First we revisit this calculation.
- Recall the formula for the bond price is

$$B = \frac{\frac{1}{2}c}{1+\frac{1}{2}y} + \frac{\frac{1}{2}c}{(1+\frac{1}{2}y)^2} + \frac{\frac{1}{2}c}{(1+\frac{1}{2}y)^3} + \cdots + \frac{F+\frac{1}{2}c}{(1+\frac{1}{2}y)^n}\,.$$

- For Newton-Raphson we also require the derivative with respect to the yield. Hence let us calculate the Macaulay duration and modified duration. The Macaulay duration is given by

$$D_{\text{Mac}} = \frac{1}{B}\left[\frac{1}{2}\frac{\frac{1}{2}c}{1+\frac{1}{2}y} + \frac{2}{2}\frac{\frac{1}{2}c}{(1+\frac{1}{2}y)^2} + \frac{3}{2}\frac{\frac{1}{2}c}{(1+\frac{1}{2}y)^{n-1}} + \cdots + \frac{n}{2}\frac{F+\frac{1}{2}c}{(1+\frac{1}{2}y)^n}\right.\,.$$

  The modified duration is given by

$$D_{\text{mod}} = \frac{D_{\text{Mac}}}{1+\frac{1}{2}y}\,.$$

- **Write a new function to calculate the above values as outputs.**
- The new function signature is

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

- The inputs are (i) double F, (ii) double c, (iii) double y, (iv) int n.
  The outputs are (v) double & B, (vi) double & D_{mac}, (vi) double & D_{mod}.
- 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 $y_{\text{decimal}} = 0.01 * y$ in your code, to avoid "factor of 100" errors.

- When summing over the cashflows to calculate the duration, be careful in your loop. Write the loop from 1 to n. Compute the time of the cashflow as $(0.5 * i)$ not $(i/2)$ else you will obtain "integer division" errors:

```
for (i = 1; i <= n; ++i) {
   double time_i = 0.5*i;
   ...
}
```

- The modified duration can be obtained from the Macaulay duration via the formula

$$D_{\text{mod}} = \frac{D_{\text{mac}}}{1 + 0.5 * y_{\text{decimal}}} .$$

  You do not need to compute separate sums for D_{mac} and D_{mod}.

- **Do not delete your old function. It is still useful.**

- Test by calling both your old and new functions with the same inputs. (You will have to write a main program.) Both functions should output the same value for B. Remember to use $F = 100$ in all your tests.

- For a zero coupon bond, the Macaulay duration equals the maturity. In our example the maturity is $0.5 * n$. Hence if you input a coupon $c = 0$, the output Macaulay duration should equal $0.5 * n$.

2.2 Yield from bond price

2.2.1 Bisection

- This calculation was assigned in the previous homework assignment.
- The previous function signature was

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

- Change the function name to "`yield_from_price_bisection`" because we are going to write additional functions to implement other algorithms.
- (I should have said this in the previous assignment.) Add an extra output `num_iter` to return the number of iterations performed:

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

- It should be reasonably obvious how to compute `num_iter`:
  (i) Initialize `num_iter = 0` at the start of the function.
  (ii) If the iteration converges, set the value of `num_iter` to the loop count.
  (iii) If the maximum number of iterations is exceeeded, set `num_iter = max_iter`.
- Otherwise the internal details of the calculation are the same as in your old function.

2.2.2 Newton-Raphson

- To implement Newton-Raphson, we require not only the bond value but also its (partial) derivative woth respect the yield.
- Hence we must call the new function
`price_from_yield(..., double & B, double & D_mac, double & D_mod)`.
- The function signature is

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

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

1. It is helpful to allocate internal arrays "`f_iter`" and "`y_iter`" of type `double` and length at least `(max_iter + 1)`. Remember to release any allocated memory on exit.
2. For Newton-Raphson, we require only one initial iterate. Since the bond price is typically not too far from par, and for a par bond the yield equals the coupon, let us set
`y_iter[0] = c`.
3. Write a loop "`for (i = 0; i < max_iter; ++i)`" as before.
4. In the loop, call the price function using `y_iter[i]`

   ```
   price_from_yield(F, c, y_iter[i], n, B, D_mac, D_mod);
   ```

5. The function value is `f_iter[i] = B - B_market`. Test if $|$`f_iter[i]`$| \leq$ `tol`. If yes, set `y = y_iter[i]` and `num_iter` to the loop count and a function return value of 0 (success). Remember to release any allocated memory.

6. We have to be careful about the derivative. The formula for the modified duration is

$$D_{\text{mod}} = -\frac{1}{B}\frac{\partial B}{\partial y}\,.$$

From this we obtain

$$\frac{\partial B}{\partial y} = -B\,D_{\text{mod}}\,.$$

However, in the above formula, the yield y is a decimal number. In our function, the value of y is a percentage, i.e. 100 times larger. Hence to obtain correct results, we must pay attention to the factor of 100:

```
double fprime = -0.01*B*D_mod;
```

Recall that we need to calculate f/f' but this requires a division so we must test to avoid division by zero. (In a real job we might use a small tolerance but here we shall just test for zero.) Write a test `if (fprime == 0.0)` and if true, then set `y = 0.0` and `num_iter = 0` and a function return value of 1 (fail). Remember to release any allocated memory.

7. Next compute

```
double delta_y = f_iter[i] / fprime;
```

Test if |`delta_y`| $\leq$ `tol`. If yes, set `y = y_iter[i]` and `num_iter` to the loop count and a function return value of 0 (success). Remember to release any allocated memory.

8. Otherwise set `y_iter[i+1] = y_iter[i] - delta_y` (*note the minus sign*) and continue with the loop.

9. If we exit the iteration loop after `max_iter` steps and the calculation still has not converged, then set $y = 0$ and `num_iter = max_iter` and set a function return value of 1 (fail). Remember to release any allocated memory.

10. 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).

11. Use your main function to call both the bisection and Newton-Raphson functions with the same inputs. They should converge to the same answer, up to tolerance. Also print the number of iterations. Newton-Raphson should converge in fewer steps.

2.2.3 Secant method

- The details of the function are almost the same as for Newton-Raphson, but we require two initial iterates, and we do not need to compute the function derivative.
- Hence we can call the old function
`price_from_yield(..., double & B)`.
- The function signature is

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

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

1. For the secant method, we require two initial iterates. We can leverage from the Newton-Raphson function

   ```
   y_iter[0] = c;
   y_iter[0] = c + 0.001;
   ```

2. We also need to compute the value of `f_iter[0]` before we begin the iteration loop

   ```
   price_from_yield(F, c, y_iter[0], n, B);
   f_iter[0] = B - B_market;
   ```

3. The loop must begin with $i = 1$ "`for (i = 1; i < max_iter; ++i)`" for obvious reasons.
4. In the loop, call the old price function using `y_iter[i]`

   ```
   price_from_yield(F, c, y_iter[i], n, B);
   ```

5. The computation of the derivative does not have factor of 100 problems:

   ```
   double fprime = (f_iter[i] - f_iter[i-1]) / (y_iter[i] - y_iter[i-1]);
   ```

   Remember to test "`if (fprime == 0.0)`" to guard against division by zero.
6. The rest is the same as in the Newton-Raphson function.
7. Use your main function to call all three functions with the same inputs (i) bisection, (ii) Newton-Raphson, (iii) secant. They should all converge to the same answer, up to tolerance. Also print the number of iterations. Newton-Raphson should converge in the fewest steps, then secant, then bisection.

2.2.4 Fixed point iteration

- This time we require one initial iterate and we do not need to compute the function derivative.
- Hence we can call the old function
  `price_from_yield(..., double & B)`.
- The function signature is

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

- For fixed point iteration, instead of solving for $f(x) = 0$, we solve for $g(x) = x$. It was stated in the lectures that we could define $g(x) = x + kf(x)$, where k is any constant $k \neq 0$. (The lectures used c not k, but we are using c to denote the bond coupon.)
- We need to be careful here. Our function is $f(y) = B(y) - B_{\text{market}}$ and we wish to solve for $f(y) = 0$. However, we cannot simply add and say $g(y) = y + k(B(y) - B_{\text{market}})$. The yield is an interest rate and the bond price has units of dollars. We can instead write

$$g(y) = y + \frac{k}{t_{\text{year}}} \frac{B(y) - B_{\text{market}}}{B_{\text{market}}}$$

  We measure time in years so $t_{\text{year}} = 1$, so overall

$$g(y) = y + k \frac{B(y) - B_{\text{market}}}{B_{\text{market}}}$$

- ***What value of k to use?*** To save you grief, I found that $k = 10$ works.

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

1. For fixed point iteration, we require one initial iterate. Set `y_iter[0] = c`.
2. Define a constant

```
const double scale = 10.0;
```

3. The loop begins with $i = 0$, so `for (i = 0; i < max_iter; ++i)`.
4. We compute as follows in the loop

```
price_from_yield(F, c, y_iter[i], n, B);
y_iter[i+1] = y_iter[i] + scale*(B - B_market)/B_market;
```

5. Test if $|\texttt{B} - \texttt{B\_market}| \leq \texttt{tol}$. If yes, set `y = y_iter[i]` and `num_iter` to the loop count and a function return value of 0 (success). Remember to release any allocated memory.

6. Test if $|\texttt{y\_iter}[\texttt{i}+1] - \texttt{y\_iter}[\texttt{i}]| \leq \texttt{tol}$. If yes, set `y = y_iter[i]` and `num_iter` to the loop count and a function return value of 0 (success). Remember to release any allocated memory.

7. Iterate until the algorithm converges or the maximum loop count is exceeded.

8. Use your main function to call all four functions with the same inputs (i) bisection, (ii) Newton-Raphson, (iii) secant, (iv) fixed point. They should all converge to the same answer, up to tolerance. Also print the number of iterations. Newton-Raphson should converge in the fewest steps, then secant. Depending on the inputs, the fixed point iteration might be faster or slower than bisection. In some cases the fixed point iteration can be almost as fast as Newton-Raphson, in other cases it can be much slower than bisection.