Queens College, CUNY, Department of Computer Science
Computational Finance
CSCI 365 / 765
Fall 2018
Instructor: Dr. Sateesh Mane

2 Homework 2

- Please email your solution, as a file attachment, to `Sateesh.Mane@qc.cuny.edu`.
- Please submit one zip archive with all your files in it.

1. The zip archive should have either of the names (CS365 or CS765):

   `StudentId_first_last_CS365_hw2.zip`

   `StudentId_first_last_CS765_hw2.zip`

2. The archive should contain one "text file" named "hw2.[txt/docx/pdf]" and one cpp file per question named "Q1.cpp" and "Q2.cpp" etc.
3. Note that not all homework assignments may require a text file.
4. Note that not all questions may require a cpp file.

2.1 Bond class, fair value, duration & yield

Let us write a "bond class" to perform various calculations relevant for bonds. A real bond class has a lot of data and class methods. We shall write a simplified bond class to calculate the fair value, Macaulay duration and modified duration, given an input yield. We shall also calculate the yield, given an input target price for the bond. Although it is a simple model, our bond class will have some of the core features of a real bond class in a financial software library.

2.2 Class declaration

- Our class has the following private data members: (i) `double Face`, (ii) `double T_maturity`, (iii) `int cpn_freq`, (iv) `int num_coupon_periods`, (v) `std::vector<double> coupons`.
- We want our `Bond` class to have methods to perform the following functions:
  1. Set the coupons (for variable rate coupons).
  2. Calculate the fair value, given an input yield.
  3. Calculate the Macaulay duration and modified duration, given an input yield.
- The calculations are all "`const`" methods. ***Explain why.***
- Write a `Bond` class with the following signature.

```
class Bond
{
public:
  Bond(double T, double F, const std::vector<double> & c, int freq);

  // public methods
  void set_coupons(const std::vector<double> & c);
  int FV_duration(double t0, double y,
                  double &B,
                  double &Macaulay_duration,
                  double &modified_duration) const;

  double FairValue(double t0, double y) const;
  double maturity() const { return T_maturity; }
  int num_periods() const { return num_coupon_periods; }
private:
  // data
  double Face;
  double T_maturity;
  int cpn_freq;
  int num_coupon_periods;
  std::vector<double> coupons;
};
```

2.3 Constructor

- Set `Face = F`. **Impose the condition `Face >= 0` in the constructor.**
- Set the coupon frequency `cpn_freq = freq`.
  Impose the condition `cpn_freq >= 1` (at least one per year) in the constructor.
- Now things get a bit tricky. The naïve thing to do is to set `T_maturity = T`.
  1. However we want the number of coupons to be an integer, and the maturity must match.
  2. To avoid roundoff error, define "`const double tol = 1.0e-6;`" and compute

     ```
     num_coupon_periods = int(cpn_freq*T + tol);
     ```

  3. **Impose the condition `num_coupon_periods >= 0` in the constructor.**
  4. **Make sure you understand the above lines of code for the value of `num_coupon_periods`.**
  5. Then set `T_maturity = (double)num_coupon_periods / (double)cpn_freq`.
  6. **Hence `T_maturity`, `num_coupon_periods` and `cpn_freq` have compatible values.**
- **This means the value of `T_maturity` may not be exactly equal to T.**
  1. Do you understand the reason for the "`double maturity()`" method in the class?
  2. **Explain why we need "`double maturity()`" (also why public and `const`).**
- We also have an input vector c to set the values of the coupons.
  1. If `num_coupon_periods == 0`, do nothing.
  2. If `num_coupon_periods > 0`, resize the coupon vector and call `set_coupons`.

     ```
     if (num_coupon_periods > 0) {
       coupons.resize(num_coupon_periods);
       set_coupons(c);                        // will be written below
     }
     ```

  3. **Do you understand why we need a data member `num_coupon_periods`?**
  4. We compute `num_coupon_periods` first so we know how to allocate the coupon vector.
- There is no dynamically allocated memory, hence we do not need to write a destructor, copy constructor or assignment operator.
- In fact, it may be better to disable copies of ojects.

2.4 set_coupons()

- The function signature is

```
void set_coupons(std::vector<double> & c)
```

- We do not allow negative coupons, so set `coupons[i] = std::max(c[i], 0.0)`, if i is less than the length of the input vector.

- If the length of the input vector is too short, set the remaining coupon values to the last value in the input vector `coupons[i] = std::max(c.back(), 0.0)`.

- Example #1:

  1. Suppose the value of `num_coupon_periods` is 4.
  2. Suppose the input vector is $\{3.5, -0.2, 4.0, 4.4\}$.
  3. Then the coupon vector is set to the following $\{3.5, 0, 4.0, 4.4\}$.

- Example #2:

  1. Suppose the value of `num_coupon_periods` is 4.
  2. Suppose the input vector is $\{3.5, 1.2\}$.
  3. Then the coupon vector is set to the following $\{3.5, 1.2, 1.2, 1.2\}$.

2.5 FairValue()

- Notice there is a function called "`FairValue()`" with return type `double`.
- It is a wrapper. Many times we want only the fair value and not the duration.
- Do this:

```
double Bond::FairValue(double t0, double y) const
{
  double B = 0;
  double Macaulay_duration = 0;
  double modified_duration = 0;
  FV_duration(t0, y, B, Macaulay_duration, modified_duration);
  return B;
}
```

- We shall write the `FV_duration()` function next.

2.6 Fair value & duration: `FV_duration()`

- The inputs are (i) t_0, (ii) y (both `double`). The outputs are (iii) `double &B`, (iv) `double &Macaulay_duration`, (v) `double &modified_duration`.
- Initialize `B = 0`, `Macaulay_duration = 0` and `modified_duration = 0`.
- **Validation tests:**
  1. If `num_coupon_periods <= 0` or $t_0 \ge T_{\text{maturity}}$, then return 1 (fail) and exit.
  2. This is why the function return type is "`int`" not `void`.
- The mathematical formula for the bond fair value B was given in the lectures.
  1. Note that in the mathematical formula, the indexing runs from $i = 1$ through n.
  2. The coupons are indexed as $c_1, \dots, c_n$ and the yield y is a decimal number.
  3. We only include terms in the sum where $t_i - t_0 > 0$.

$$\begin{aligned} B &= \biggl[\frac{c_1/f}{(1+y/f)^{f(t_1-t_0)}} + \frac{c_2/f}{(1+y/f)^{f(t_2-t_0)}} + \dots \\ &\qquad \dots + \frac{c_{n-1}/f}{(1+y/f)^{f(t_{n-1}-t_0)}} + \frac{F+(c_n/f)}{(1+y/f)^{f(t_n-t_0)}} \biggr]_{t_i>t_0} \\ &= \sum_{i=1}^{n} \biggl[\frac{(\text{numerator})_i}{(1+y/f)^{f(t_i-t_0)}} \biggr]_{t_i>t_0} . \end{aligned} \tag{2.6.1}$$

  4. The definition of "`numerator_i`" in eq. (2.6.1) is obvious.
  5. The formula for the Macaulay duration is

$$D_{\text{Macaulay}} = \frac{1}{B} \sum_{i=1}^{n} \biggl[(t_i - t_0)\, \frac{(\text{numerator})_i}{(1+y/f)^{f(t_i-t_0)}} \biggr]_{t_i>t_0} . \tag{2.6.2}$$

  6. The formula for the modified duration is

$$D_{\text{mod}} = \frac{D_{\text{Macaulay}}}{1+y/f} . \tag{2.6.3}$$

- The input value of the yield y is a percentage, so if the yield is 5% then $y = 5$.
  1. **Hence employ an internal variable $y_{\text{decimal}} = 0.01 * y$, to avoid "factor of 100" errors in your code.**
- The coupon dates are `t_i = double(i)/double(cpn_freq)`, for $i = 1, \dots, n$.
  1. However, we have to guard against floating–point roundoff error.
  2. Define a tolerance parameter "`const double tol = 1.0e-6`" in your function.
  3. **Only include terms in the sum such that $t_i \ge t_0 + \texttt{tol}$.**
- **Write a loop(s) to compute the sums in eqs. (2.6.1) and (2.6.2).**
- The modified duration is easy to obtain from the Macaulay duration (see eq. (2.6.3)).
- Return 0 (success) and exit.

2.7 Tests

- **Here are some tests 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.
- Put $t_0 = 0$ and use a constant coupon c. Then if the yield equals the coupon $y = c$, you should obtain `FV = 100`.
- Put $t_0 = 0$ and $y = 0$. Then the fair value is a straight sum of the values of the cashflows. Your program should obtain the result

$$B = F + \sum_{i=1}^{n} \frac{c_i}{f} \,. \tag{2.7.1}$$

- 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. The formula is

$$B_{\text{zero coupon}} = \frac{F}{(1 + y/f)^{f(T_{\text{maturity}} - t_0)}} \,. \tag{2.7.2}$$

- **The Macaulay duration of a zero coupon bond equals the time to maturity:**

$$D_{\text{Macaulay}} = T_{\text{maturity}} - t_0 \,. \tag{2.7.3}$$

  This is an important fact.
- For fixed values of t_0 and y, the fair value increases if the coupons increase.
- If the coupons are positive, the value of the Macaulay duration is less than $T_{\text{maturity}} - t_0$. If you multiply all the coupons by a factor of 2 (or any number > 1), the value of the Macaulay duration will decrease.
- For a given face and coupons, etc., the fair value of a bond decreases as the yield y increases. (This is in fact a general theorem. It was proved in the 1930s, I think.)

2.8 Yield from bond price: `yield()`

- We employ the bisection algorithm.
- The fundamental idea is simple and was explained in the lecture. It goes as follows:

1. Given a target value, we wish to find the yield y such that $B(y) = B_{\text{target}}$.
2. We know from eq. (2.6.1) that $B(y)$ is a continuous function of y.
3. We also know that $B(y)$ decreases as the value of y increases.
4. Hence we find a low yield y_{low} such that $B(y_{\text{low}}) > B_{\text{target}}$ and a high yield y_{high} such that $B(y_{\text{high}}) < B_{\text{target}}$.
5. Then the solution for y lies somewhere between y_{low} and y_{high}.
6. It is possible by luck that either y_{low} and y_{high} is the solution we seek.
7. If so we exit the algorithm immediately.
8. Else we iterate as follows.
9. We use the midpoint $y_{\text{mid}} = (y_{\text{low}} + y_{\text{high}})/2$ and calculate $B(y_{\text{mid}})$.
10. If $|B(y_{\text{mid}}) - B_{\text{target}}|$ is less than a prespecified tolerance, we exit the calculation and say that y_{mid} is the solution.
11. Else we perform comparison tests (to be described below) and update either $y_{\text{low}} := y_{\text{mid}}$ or $y_{\text{high}} := y_{\text{mid}}$.
12. We repeat the iteration using the updated values of y_{low} and y_{high}.
13. Hence the interval $|y_{\text{high}} - y_{\text{low}}|$ is cut by a factor of two at each iteration step.
14. If $|y_{\text{high}} - y_{\text{low}}|$ is less than a prespecified tolerance, we exit the calculation and say that y_{mid} is the solution.

- The function signature is as follows (it is a standalone function).

```
int yield(const Bond &bond, double B_target, double t0, double & y, int & num_iter,
          double tol=1.0e-4, int max_iter=100);
```

- The inputs are `double B_target` and `double t0`, with obvious meanings.
- The outputs are: `double & y` (the yield, if the calculation converges, else 0), and `int & num_iter` (the number of iterations, if the calculation converges).
- There are two optional inputs `double tol` and `int max_iter`, with default values. Obviously `B_target` is the target bond price and `tol` is a tolerance parameter for the iteration calculation. The parameter `max_iter` is to stop the calculation after a finite number of iterations to avoid an infinite loop. The output parameter `num_iter` tells us how many iterations were performed, if the calculation converges.
- The function return type is `int` because we return 1 for failure or 0 for success.
- **Initialize `y = 0` and `num_iter = 0`.**
- **First validation test:** If $B_{\text{target}} \leq 0.0$ or `bond.num_periods() <= 0` or `t0 >= bond.maturity()`, then return 1 (fail) and exit.
- Let us keep things simple and use $y_{\text{low}} = 0.0$ and $y_{\text{high}} = 100.0$.
  (A more sophisticated algorithm would do a better job.)
- Set $y_{\text{low}} = 0.0$.
  1. Calculate the corresponding bond price `B_y_low = bond.FairValue(t0, y_low)`.
  2. This is the reason to have the function `FairValue()`.
  3. We do not need the duration in this calculation.
- Also calculate `diff_B_y_low = B_y_low - B_target` for use below.
- **Validation test:**
  1. If `std::abs(diff_B_y_low) <= tol`, ***then we are done***.
  2. The output is already within the tolerance.
  3. Set $y = y_{\text{low}}$ and "return 0" (success) and exit.
- Next set $y_{\text{high}} = 100.0$. Calculate the bond price `B_y_high = bond.FairValue(t0, y_high)`.
- Also calculate `diff_B_y_high = B_y_high - B_target` for use below.
- **Validation test:**
  1. If `std::abs(diff_B_y_high) <= tol`, ***then we are done***.
  2. The output is already within the tolerance.
  3. Set $y = y_{\text{high}}$ and "return 0" (success) and exit.
- ***Perform the above calculations and validation tests sequentially:*** if the calculation converges already for $y = y_{\text{low}}$, there is no need to waste time doing calculations with y_{high}.

- **Validation test:**
  1. We must verify that $B(y_{\rm low})$ and $B(y_{\rm high})$ are on opposite sides of the $B_{\rm target}$.
  2. This will be the case if `diff_B_y_low` and `diff_B_y_high` have opposite signs.
  3. **Test if `diff_B_y_low * diff_B_y_high > 0`.**
  4. If yes, then the values of $B(y_{\rm low})$ and $B(y_{\rm high})$ do *not* bracket $B_{\rm target}$ and we have **failed.**
  5. Set $y = 0$ and "return 1" (fail) and exit.
- If we have made it this far, then we know that we have bracketed the answer, and the true yield y lies between $y_{\rm low}$ and $y_{\rm high}$.
- Hence we now begin the main bisection iteration loop.

```
for (num_iter = 1; num_iter < max_iter; ++num_iter)
```

- In the loop, set $y = (y_{\rm low} + y_{\rm high})/2.0$ and calculate `B = bond.FairValue(t0, y)`.
- Also calculate `diff_B = B - B_target` for use below.
- If `std::abs(diff_B) <= tol`, ***then we are done***.
  1. We have found a "good enough" value for the yield y.
  2. Hence "return 0" (success) and exit.
- Check if B and $B(y_{\rm low})$ are on the same side of $B_{\rm target}$.
  1. Test if `diff_B * diff_B_y_low > 0.0`.
  2. If yes, then update $y_{\rm low} = y$.
- Else obviously B and $B(y_{\rm high})$ are on the same side of $B_{\rm target}$, so update $y_{\rm high} = y$.
- *Don't be in a rush to iterate!*
  1. If $|y_{\rm high} - y_{\rm low}| \le$ `tol`, ***then this is good enough***.
  2. The algorithm has converged.
  3. Hence "return 0" (success) and exit.
- If we have come this far, continue with the iteration loop.
- If we exit the iteration loop after `max_iter` steps and the calculation still has not converged, then set $y = 0$ and "return 1" (fail).
  1. This can happen if the tolerance `tol` is too small or the value of `max_iter` is too small.
  2. Reasonable values to use are `tol = 1.0e-4` and `max_iter = 100`.
- 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).

2.9 Tests

- **Here are some tests to help you to check that your code is working correctly.**
- Remember to use $F = 100$ in all your tests. There is no point in being too clever.
- Set $t_0 = 0$ (newly issued bond). Use a constant coupon c (set it using the constructor).

  1. Try an input $B_{\text{target}} = 100$. If your function works correctly, it should output $y = c$ (up to the tolerance), for any value of c and any value of `freq` and any value of T (provided `freq * T >= 1`. Remember that if $F = 100$ and $y = c$ (the yield equals the coupon) for a newly issued bond, then $B = 100$. The converse also holds true.
  2. If $B_{\text{target}} < 100$ then your output should be $y > c$.
  3. If $B_{\text{target}} > 100$ then your output should be $y < c$.

- Use any value of `freq` and any value of T, provided `freq * T >= 1`. Use any value $t_0 \geq 0$ and $t_0 < T$. Use a constant coupon c or input a vector of coupons using `set_coupons`.

  1. Choose some value fir the yield, say y_1, and calculate the fair value, say B_1.
  2. Set a target $B_{\text{target}} = B_1 + 1$.
  3. Calculate the yield, say the output is y_2.
  4. Calculate the fair value using y_2, say the answer is B_2.
  5. If you have done your work correctly, you should obtain $B_2 - B_1 \simeq 1$ (up to tolerance).