Queens College, CUNY, Department of Computer Science

Computational Finance

CSCI 365 / 765

Spring 2018

Instructor: Dr. Sateesh Mane

due Saturday, July 7, 2018, 11.59 pm

This code will not be used for in class exams, but will be used for projects.

- Please submit your solution via email, as a file attachment, to `Sateesh.Mane@qc.cuny.edu`.
- Please submit one zip archive with all your files in it.
- The zip archive should have either of the names (CS365 or CS765):

```
StudentId_first_last_CS365_hw2.zip
StudentId_first_last_CS765_hw2.zip
```

- The archive should contain one "text file" of type "txt/docx/pdf" and code in cpp files.
- **Submit ONE text file, with the answers to different questions on separate pages.**
- Label your files clearly.

2 Homework: 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.

We shall skip "duration" in Summer 2018.

2.1 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. **We shall skip this for Summer 2018.**
  4. Calculate the yield, given an input target price for the bond.
- The fair value, duration and yield calculations are all "`const`" methods. ***Explain why.***
- Write a `Bond` class with the following signature.

```
class Bond
{
public:
  Bond(double T, double F, double c=0, int freq=2);
  ~Bond();

  // public methods
  void set_flat_coupons(double c);
  int set_coupons(std::vector<double> & c);
  int FV_duration(double t0, double y,                    // skip for Summer 2018
                  double &B,
                  double &Macaulay_duration,
                  double &modified_duration) const;

  int yield(double B_target, double tol, int max_iter, double t0,
            double & y, int & num_iter) const;

  double FairValue(double t0, double y) const;
  double maturity() const { return T_maturity; }

private:
  // data
  double Face;
  double T_maturity;
  int cpn_freq;
  int num_coupon_periods;
  std::vector<double> coupons;
};
```

2.2 Constructor & destructor

- Set `Face = F`. Impose the condition `Face >= 0` in the constructor.
- Set the coupon frequency `cpn_freq = freq`.
  This is an optional parameter with a default value of 2 (semiannual coupons).
  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 = num_coupon_periods / 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 allow an optional parameter c (default $= 0$) to set all the coupons to equal values.
  1. If some idiot inputs $c < 0$, set $c = 0$ (no negative coupons).
  2. Populate the vector. There are two ways you can do it. Choose your favorite.

     ```
     if (num_coupon_periods > 0)                          // #1
       coupons.resize(num_coupon_periods, c);

     if (num_coupon_periods > 0) {
       for (int i = 0; i < num_coupon_periods; ++i)       // #2
         coupons.push_back( c );
     }
     ```

  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.
- The destructor is easy. Just clear the coupons array and set `num_coupon_periods = 0`.

```
Bond::~Bond()
{
  coupons.clear();
  num_coupon_periods = 0;
}
```

- ***If you think about it, we do not need to do the above.***
  An empty destructor also works.

2.3 set_coupons()

- The constructor provides an easy way to set the coupons when they are all equal, in particular if they are all zero.
- However, for a variable-rate coupon bond, we need to set the coupon values individually.
- The function signature is

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

- The return type is `int` because we test for bad input and return 1 (fail) for bad input.
- **Validation check:**
  1. If `c.size() < num_coupon_periods` then "return 1" (fail) and exit.
  2. We want to learn finance, not waste time on tiresome programming details.
  3. We do not want to deal with useless complications if the input array is too short.
- **We do not allow negative coupons.**
  1. Write a loop so that if `c[i] <= 0.0` then `coupons[i] = 0.0`.
  2. Else `coupons[i] = c[i]`.
  3. However, we do not classify this as a failure.
- After the loop, return 0 (success) and exit.

2.4 Set flat coupons

- The constructor sets the coupons to equal values ("flat coupons").
- There is a class method `set_coupons()` to set variable rate coupons.
- *But what if we want to modify the coupons to a new value, all equal?*
- We have to create a vector with all equal values and call `set_coupons()`.
- This is obviously not complicated to do, but still ... *it is inconvenient.*
- Learn to design "user friendly" software.

```
void Bond::set_flat_coupons(double c)
{
  if (c < 0.0) c = 0.0;                    // no negative coupons
  // loop(?) coupons[i] = c
}
```

- The return type is `void`, why not.

2.5 FairValue()

- The inputs are (i) t_0, (ii) y (both `double`).
- The function return value is the bond fair value `B`.
- **Validation tests:** If `num_coupon_periods <= 0` or $t_0 \geq T_{\text{maturity}}$, then return 0.
- The mathematical formula for the bond fair value B was given in the lectures.
  1. **Note: the indexing in the mathematical formula runs from $i = 1$ through n.**
  2. However C++ indexing begins at 0, not 1. **Do not make array indexing errors.**
  3. The coupons are indexed as $c_1, \ldots, c_n$ and the yield y is a decimal number.
  4. **We only include terms in the sum where $t_i - t_0 > 0$.**
  5. Coupons which have already been paid are of no use to us.

$$
\begin{aligned}
B = \Bigg[& \frac{c_1/f}{(1+y/f)^{f(t_1-t_0)}} + \frac{c_2/f}{(1+y/f)^{f(t_2-t_0)}} + \ldots \\
& \ldots + \frac{c_{n-1}/f}{(1+y/f)^{f(t_{n-1}-t_0)}} + \frac{F + (c_n/f)}{(1+y/f)^{f(t_n-t_0)}} \Bigg]_{t_i > t_0} \qquad (2.5.1) \\
= & \sum_{i=1}^{n} \left[\frac{(\text{numerator})_i}{(1+y/f)^{f(t_i-t_0)}} \right]_{t_i > t_0} .
\end{aligned}
$$

  6. The definition of "`numerator_i`" in eq. (2.5.1) is obvious.
- **Write a loop to compute the sums in eq. (2.5.1).**
- Note the following:
  1. The input value of the yield y is a percentage, so if the yield is 5% then $y = 5$.
  2. **Employ an internal variable $y_{\text{decimal}} = 0.01 * y$, to avoid "factor of 100" errors in your code.**
  3. The coupon dates are `t_i = double(i)/double(cpn_freq)`, for $i = 1, \ldots, n$.
     (a) However, we have to guard against floating–point roundoff error.
     (b) Define a tolerance parameter "`const double tol = 1.0e-6`" in your function.
     (c) **Only include terms in the sum such that $t_i \geq t_0 +$ `tol`.**
- Return the value of B and exit.

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`.
- **We shall skip this function for Summer 2018**.
- **Set `B=0`, `Macaulay_duration=0`, `modified_duration=0`.**
- **Return 0 and exit.**

2.7 Tests

- **Here are some tests to help you to check that your code is working correctly.**
- 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.
  1. Then if the yield equals the coupon $y = c$, you should obtain `FV = 100`.
  2. Pay attention to the factor of 100 in the percentage.
  3. The yield *as a percentage* equals the coupon. If $c = 7$ then $y = 7\%$.
- 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 the face value at maturity. The formula is

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

- For fixed values of t_0, Face and coupons, the fair value decreases if the yield increases. (This is in fact a general theorem. It was proved in the 1930s, I think.)
- For fixed values of t_0 and y, the fair value increases if the coupons increase.

2.8 Yield from bond price: `yield()`

- **This is an important calculation.**
- 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. (2.5.1) 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.
- **We employ only one tolerance parameter.**
  1. In reality, we should have separate tolerance parameters for $|B(y_{\text{mid}}) - B_{\text{target}}|$ and $|y_{\text{high}} - y_{\text{low}}|$.
  2. However, this just makes the code more complicated and does not teach anything about finance or programming.

- The inputs are (i) `double B_target`, (ii) `double tol`, (iii) `int max_iter`, (iv) `double t0`.
- The outputs are:
  (v) `double & y` (the yield, if the calculation converges, else 0),
  (vi) `int & num_iter` (the number of iterations, if the calculation converges).
- The function return type is "`int`" not "`void`" because we shall perform validation tests. We return 1 (failure) or 0 (success).
- 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.
- **Initialize `y = 0` and `num_iter = 0`.**
- **First validation test:**
  If $B_{\text{target}} \leq 0.0$ or `num_coupon_periods <= 0` or $t_0 \geq T_{\text{maturity}}$, 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 = 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.
- **Tolerance 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 = FairValue(t0, y_high)`.
- Also calculate `diff_B_y_high = B_y_high - B_target` for use below.
- **Tolerance 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}.

- **Second validation test:**
  1. We must verify that $B(y_{\text{low}})$ and $B(y_{\text{high}})$ are on opposite sides of the B_{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 $B(y_{\text{low}})$ and $B(y_{\text{high}})$ do *not* bracket B_{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_{low} and y_{high}.
- Hence we now begin the main bisection iteration loop.

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

- In the loop, set $y = (y_{\text{low}} + y_{\text{high}})/2.0$ and calculate `B = 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 $B(y_{\text{low}})$ are on the same side of B_{target}.
  1. Test if `diff_B * diff_B_y_low > 0.0`.
  2. If yes, then update $y_{\text{low}} = y$.
- Else obviously B and $B(y_{\text{high}})$ are on the same side of B_{target}, so update $y_{\text{high}} = y$.
- *Don't be in a rush to iterate!*
  1. If $|y_{\text{high}} - y_{\text{low}}| \leq$ `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 for 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 again but 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).