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

3 Homework 3

- 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_hw3.zip`

     `StudentId_first_last_CS765_hw3.zip`

  2. The archive should contain one "text file" named "hw3.[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.

3.1 Homework: Yield Curves

- This homework assignment contains two parts.
- In the first part, I shall generate some yield curves for you. You task is to interpolate the spot curve using linear interpolation and constant forward rate interpolation.
- **Do the calculations "by hand" (using a calculator or spreadsheet, etc.)**
    1. **If the time t is $t \leq 0.5$, then just use the spot rate $r_{0.5}$, nothing to interpolate.**
    2. **If $t \geq$ last date in the spot curve, then use the last spot rate in the spot curve.**
- The second part of the assignment is to write a `YieldCurve` class. The material in the first part of the assignment will serve as examples to test your code.

3.2 Flat curve

- Set $y = 5\%$ for all maturities.
- I generated a yield curve with 20 points below.
- You should obtain a spot rate of $r = 4.93852\%$.
- Using either interpolation method, you should obtain $r = 4.93852\%$ for all t.
- The discount factors are simply $d(t) = e^{-0.0493852\,t}$.
- A sample table is given below.
- **You do not have to do them all.**
- **Just do enough to satisfy yourself that you know how to interpolate correctly.**

t		r_{lin} (%)		d_{lin}		r_{cfr} (%)		d_{cfr}
0.43		4.93852		0.978988		4.93852		0.978988
0.81		4.93852		0.960787		4.93852		0.960787
1.19		4.93852		0.942925		4.93852		0.942925
1.57		4.93852		0.925395		4.93852		0.925395
1.95		4.93852		0.90819		4.93852		0.90819
2.33		4.93852		0.891306		4.93852		0.891306
2.71		4.93852		0.874735		4.93852		0.874735
3.09		4.93852		0.858473		4.93852		0.858473
3.47		4.93852		0.842513		4.93852		0.842513
3.85		4.93852		0.826849		4.93852		0.826849
4.23		4.93852		0.811477		4.93852		0.811477
4.61		4.93852		0.79639		4.93852		0.79639
4.99		4.93852		0.781584		4.93852		0.781584
5.37		4.93852		0.767054		4.93852		0.767054
5.75		4.93852		0.752793		4.93852		0.752793
6.13		4.93852		0.738797		4.93852		0.738797
6.51		4.93852		0.725062		4.93852		0.725062
6.89		4.93852		0.711582		4.93852		0.711582
7.27		4.93852		0.698353		4.93852		0.698353
7.65		4.93852		0.68537		4.93852		0.68537

3.3 Upward sloping (normal) yield curve

- Set `y[i] = 1.0 + 0.9*log(i+1) (%)`, where $i = 0, 1, 2, \ldots, 19$.
- I generated a yield curve with 20 points below.
- The par yields and bootstrapped spot rates are tabulated below.
- A sample table of interpolated spot rates r_{lin} and r_{cfr} is given below.
- The discount factors are $d_{\text{lin}}(t) = e^{-r_{\text{lin}}\, t}$ and $d_{\text{cfr}}(t) = e^{-r_{\text{cfr}}\, t}$.
- **You do not have to do them all.**
- **Just do enough to satisfy yourself that you know how to interpolate correctly.**

3.3.1 Yields and spots

t		y (%)		r (%)		d
0.5		1		0.997508		0.995025
1		1.62383		1.6198		0.983933
1.5		1.98875		1.98461		0.97067
2		2.24766		2.24432		0.956106
2.5		2.44849		2.4466		0.940668
3		2.61258		2.61264		0.924614
3.5		2.75132		2.75375		0.908118
4		2.8715		2.87664		0.891308
4.5		2.9775		2.98568		0.874279
5		3.07233		3.08381		0.857109
5.5		3.15811		3.17315		0.839857
6		3.23642		3.25525		0.822575
6.5		3.30845		3.3313		0.805305
7		3.37515		3.40222		0.78808
7.5		3.43725		3.46873		0.770932
8		3.49533		3.53143		0.753886
8.5		3.54989		3.59079		0.736963
9		3.60133		3.64722		0.720183
9.5		3.65		3.70104		0.703562
10		3.69616		3.75254		0.687115

3.3.2 Interpolation

t		r_{lin} (%)		d_{lin}		r_{cfr} (%)		d_{cfr}
0.43		0.997508		0.99572		0.997508		0.99572
0.81		1.38333		0.988858		1.47383		0.988133
1.19		1.75842		0.979292		1.79454		0.978871
1.57		2.02097		0.968769		2.03093		0.968617
1.95		2.21835		0.957664		2.22434		0.957553
2.33		2.37782		0.946104		2.38756		0.945889
2.71		2.51634		0.93408		2.5238		0.933892
3.09		2.63804		0.921718		2.64141		0.921622
3.47		2.74528		0.909135		2.74643		0.909099
3.85		2.83978		0.896433		2.84313		0.896318
4.23		2.9268		0.883553		2.93		0.883434
4.61		3.00727		0.870546		3.00909		0.870472
4.99		3.08185		0.857457		3.08204		0.857449
5.37		3.14992		0.844382		3.15152		0.844309
5.75		3.2142		0.831257		3.21598		0.831172
6.13		3.27502		0.81811		3.27622		0.81805
6.51		3.33272		0.804962		3.33283		0.804956
6.89		3.38662		0.791886		3.3875		0.791838
7.27		3.43814		0.778838		3.43927		0.778773
7.65		3.48754		0.765828		3.4884		0.765778

3.4 Downward sloping (inverted) yield curve

- Set `y[i] = 8.0/(1.0 + 0.03*i) (%)`, where $i = 0, 1, 2, \ldots$.
- I generated a yield curve with 20 points below.
- The par yields and bootstrapped spot rates are tabulated below.
- A sample table of interpolated spot rates r_{lin} and r_{cfr} is given below.
- The discount factors are $d_{\text{lin}}(t) = e^{-r_{\text{lin}}\, t}$ and $d_{\text{cfr}}(t) = e^{-r_{\text{cfr}}\, t}$.
- **You do not have to do them all.**
- **Just do enough to satisfy yourself that you know how to interpolate correctly.**

3.4.1 Yields and spots

t		y (%)		r (%)		d
0.5		8		7.84414		0.961538
1		7.76699		7.61562		0.926671
1.5		7.54717		7.39736		0.894974
2		7.33945		7.18879		0.866082
2.5		7.14286		6.98935		0.83968
3		6.95652		6.79856		0.815498
3.5		6.77966		6.61593		0.793297
4		6.61157		6.44104		0.772872
4.5		6.45161		6.27345		0.754043
5		6.29921		6.1128		0.736652
5.5		6.15385		5.95872		0.720558
6		6.01504		5.81086		0.705639
6.5		5.88235		5.66891		0.691786
7		5.7554		5.53257		0.678901
7.5		5.6338		5.40156		0.666899
8		5.51724		5.27561		0.655702
8.5		5.40541		5.15446		0.645242
9		5.29801		5.0379		0.635457
9.5		5.19481		4.92568		0.626291
10		5.09554		4.81761		0.617695

3.4.2 Interpolation

t		r_{lin} (%)		d_{lin}		r_{cfr} (%)		d_{cfr}
0.43		7.84414		0.966833		7.84414		0.966833
0.81		7.70246		0.939516		7.66922		0.939769
1.19		7.53268		0.914261		7.51107		0.914496
1.57		7.36816		0.89076		7.36016		0.890872
1.95		7.20964		0.868847		7.20483		0.868929
2.33		7.05716		0.848376		7.04756		0.848565
2.71		6.90922		0.829245		6.90065		0.829438
3.09		6.76569		0.811347		6.76133		0.811456
3.47		6.62689		0.794571		6.62541		0.794612
3.85		6.49351		0.778801		6.48874		0.778944
4.23		6.36395		0.763994		6.35903		0.764153
4.61		6.23811		0.750079		6.23512		0.750182
4.99		6.11601		0.736984		6.1157		0.736996
5.37		5.99878		0.724601		5.99602		0.724708
5.75		5.88479		0.712928		5.88157		0.713059
6.13		5.77395		0.701915		5.77173		0.702011
6.51		5.66618		0.691516		5.66598		0.691525
6.89		5.56257		0.681635		5.56087		0.681715
7.27		5.46182		0.672284		5.45959		0.672393
7.65		5.36377		0.663432		5.36204		0.66352

3.5 Humped yield curve

- Set `y[i] = 1.0 + 0.16*i - 0.007*i*i (%)`, where $i = 0, 1, 2, \ldots$.
- I generated a yield curve with 20 points below.
- The par yields and bootstrapped spot rates are tabulated below.
- A sample table of interpolated spot rates r_{lin} and r_{cfr} is given below.
- The discount factors are $d_{\text{lin}}(t) = e^{-r_{\text{lin}}\, t}$ and $d_{\text{cfr}}(t) = e^{-r_{\text{cfr}}\, t}$.
- **You do not have to do them all.**
- **Just do enough to satisfy yourself that you know how to interpolate correctly.**

3.5.1 Yields and spots

t		y (%)		r (%)		d
0.5		1		0.997508		0.995025
1		1.153		1.15013		0.988565
1.5		1.292		1.28907		0.98085
2		1.417		1.41434		0.97211
2.5		1.528		1.52591		0.962571
3		1.625		1.62371		0.952456
3.5		1.708		1.70766		0.941983
4		1.777		1.77763		0.931364
4.5		1.832		1.83349		0.920805
5		1.873		1.8751		0.910506
5.5		1.9		1.90233		0.90066
6		1.913		1.91504		0.891453
6.5		1.912		1.91313		0.883068
7		1.897		1.89651		0.875679
7.5		1.868		1.86512		0.869459
8		1.825		1.81897		0.864575
8.5		1.768		1.7581		0.861192
9		1.697		1.68262		0.859473
9.5		1.612		1.59271		0.859584
10		1.513		1.48861		0.861689

3.5.2 Interpolation

t		r_{lin} (%)		d_{lin}		r_{cfr} (%)		d_{cfr}
0.43		0.997508		0.99572		0.997508		0.99572
0.81		1.09213		0.991193		1.11433		0.991015
1.19		1.20293		0.985787		1.21668		0.985626
1.57		1.30661		0.979695		1.31141		0.979621
1.95		1.40181		0.973035		1.4047		0.97298
2.33		1.48798		0.965924		1.49335		0.965803
2.71		1.56699		0.958424		1.57138		0.95831
3.09		1.63882		0.950621		1.64083		0.950562
3.47		1.70262		0.94263		1.7033		0.942608
3.85		1.75664		0.934606		1.75854		0.934537
4.23		1.80332		0.926556		1.80496		0.926492
4.61		1.84264		0.918562		1.84341		0.918529
4.99		1.87427		0.910714		1.87435		0.910711
5.37		1.89525		0.903233		1.89574		0.903209
5.75		1.90868		0.896059		1.90896		0.896044
6.13		1.91454		0.889264		1.91451		0.889265
6.51		1.9128		0.882918		1.91277		0.882919
6.89		1.90016		0.877287		1.89996		0.877299
7.27		1.87956		0.872281		1.87902		0.872315
7.65		1.85128		0.867949		1.85064		0.867991

Part 2

3.6 `YieldCurve` class

- We shall implement a simplified `YieldCurve` class.
- Our input will be a set of tenors and par yields ("information from database").
    1. **In general, we do not use the term "maturities" for dates in a yield curve.**
    2. We use the term "tenors" instead.
    3. Hence we shall speak of the "6 month tenor" and "12 month tenor" etc.
    4. Hence our `YieldCurve` class will input a set of tenors and par yields.
- We shall only support a semiannual frequency.
- Our class will provide the following functionality:
    1. Bootstrap the par yields to obtain the discount factors and spot rates.
    2. *The bootstrap may fail. We shall discuss this.*
    3. Accessor methods to get the discount factors, spot rates and par yields.
    4. Interpolate the spot curve. There is some black magic here.

3.7 Class signature

```
class YieldCurve
{
public:
  YieldCurve();
  ~YieldCurve();

  int create_curves(int n, const std::vector<double> & dates,
                           const std::vector<double> & yields);

  // const methods
  int length_par_curve() const { return par_yields.size(); }
  int length_spot_curve() const { return spot_rates.size(); }

  const double* df(int n) const;
  const double* spot(int n) const;
  const double* tenor(int n) const;
  const double* yield(int n) const;

  int interpolate(int algorithm, double t, double & df, double & r) const;

private:
  void reset();
  int set_yields(int n, const std::vector<double> & dates,
                        const std::vector<double> & yields);
  void bootstrap();

  // data
  std::vector<double> tenors;
  std::vector<double> par_yields;
  std::vector<double> discount_factors;
  std::vector<double> spot_rates;

  // disable copy
  YieldCurve(const YieldCurve& );
  YieldCurve& operator=(const YieldCurve& );
};
```

3.8 Lengths of yield curves

- There are `const` methods `length_par_curve()` and `length_spot_curve()`.
- Their definitions are obvious.
- ***But why are there two methods (not one)?*** *Don't answer this question, read below.*
- We need multiple methods because the bootstrap may fail.
- *However, the bootstrap may not fail completely.*
- We may have a partially constructed spot curve.
    1. It has discount factors and spot rates for a few tenors, but not for all of the input data.
    2. Because of the way the bootstrap proceeds, from short dates to later ones, we may still have a valid spot curve, for tenors before the bootstrap fails.
    3. It may be possible to use a partially constructed spot curve, for short time intervals.
    4. Hence we shall not clear the internal arrays if the bootstrap fails.
    5. We shall store the partially constructed spot curve.
- **However the spot curve may be shorter than the par curve.**
- **Do you understand?**

3.9 const

These questions you should be able to answer before writing any code.

3.9.1 create_curves()

- The method `create_curves(...)` is not `const`, because it changes internal data in the class.
- However, it has `const` reference input arguments:

```
const std::vector<double> & dates
const std::vector<double> & yields
```

- **Why are they `const`?**
- **Why are they references to vectors?**
- Why not write non-const non-references?

```
int create_curves(int n, std::vector<double> dates, std::vector<double> yields);
```

3.9.2 accessor methods, the "get" stuff

- **Why are all of these methods `const`? Give your answer without writing any code.**

```
const double* df(int n) const;
const double* spot(int n) const;
const double* tenor(int n) const;
const double* yield(int n) const;
```

- It will be explained later why the return type is "`const double*`" not `double`.

3.9.3 interpolate()

- We want to interpolate the spot curve.

```
int interpolate(int algorithm, double t, double & df, double & r) const;
```

- We need to calculate the discount factor and spot rate at the times specified in the input.
- Hence we obviously must perform some mathematical calculations in the function.
- Nevertheless `interpolate(...)` is `const`.
- ***Why is `interpolate(...) const`? Give your answer without writing any code.***

3.10 Constructor, destructor & reset()

- In the constructor, all the vectors are initialized to be empty. Hence a default constructor meets our needs. In the destructor, the vectors will be automatically cleared. Hence an empty destructor will work.

```
YieldCurve::YieldCurve() {}
YieldCurve::~YieldCurve() {}
```

- Note that if our class used ordinary C++ arrays, e.g. `double *tenors` and `double *par_yields`, etc., we would be responsible for the memory management using operators `new` and `delete`. The destructor would have to deallocate the memory, to avoid memory leaks.

- We also sometimes need to clear the internal data in our calculations.

  1. The `reset()` method does the job.
  2. Note that `reset()` is private.
  3. We do not want outside applications to mess with our `YieldCurve` object.

```
void YieldCurve::reset()
{
  tenors.clear();
  par_yields.clear();
  discount_factors.clear();
  spot_rates.clear();
}
```

- **We shall disable copying of our objects.**

  1. We do not want copies of our `YieldCurve` object to be made.
  2. If we update our `YieldCurve` object, we do not want out of date copies of our original object lying around.
  3. **Write declarations for the copy constructor and assignment operator.**
  4. **Make the declarations private.**
  5. **Explain how this procedure disables copies of our objects.**

```
private:
  YieldCurve(const YieldCurve& );
  YieldCurve& operator=(const YieldCurve& );
```

  6. Then we do not write function bodies for them, since copies are disabled.

3.11 create_curves()

- As the name suggests, this method creates the yield curves.
- The input data is "from a database" but we shall not use databases in this course.
- The function return type is `int` because we shall test for bad data.
- The actual code is simple. The complexity is hidden in the private class methods.

```
int YieldCurve::create_curves(int n, const std::vector<double> & dates,
                                     const std::vector<double> & yields)
{
  reset();
  int rc = set_yields(n, dates, yields);
  if (rc) return rc;
  bootstrap();
  return 0;
}
```

- First we call `reset()` to clear the internal data.
- Next we call `set_yields()` to set the par yields. This function tests for bad data and we exit (with a nonzero return cide) if the validation tests fail.
- Note that we call `reset()` *before* validating the inputs. If the validation tests fail and we exit without bootstrapping the spot curve, we do not want an old curve lying around in our object. That would be confusing to users (the calling application).
- If the validation tests pass, then `set_yields()` will populate `tenors` and `par_yields` and the bootstrap can proceed.
- The return type of `bootstrap()` is `void`. Even if the bootstrap fails and the spot curve is only partially constructed, we can live with that. It simply means the length of the spot curve is shorter than the par curve. Technically, we should set a warning message, but never mind.
- Note that `set_yields()` and `bootstrap()` are private methods. We cannot allow outside users to call them without being sure that there is valid data in the par yields.

3.12 `set_yields()`

- This is our first major function.
- We validate the input data, and if the tests pass, we populate some internal class data.
- What are the validation tests?
  1. First the obvious basic checks of length. Return 1 (fail) and exit if $n \leq 0$ or `dates.size() < n` or `yields.size() < n`.
  2. **No negative yields.** If `yields[i] < 0` for any i then return 1 (fail) and exit.
  3. **We demand that the dates are sorted, in increasing order. We also demand that all the dates are distinct (no duplicates).**
     (a) That is to say, if $i < j$ then `dates[i] < dates[j]`.
     (b) **Check that `dates` is sorted and there are no duplicates.**
     (c) Return 1 (fail) and exit if the test fails.
  4. **We also demand that all the dates are in the future, i.e. `dates[i] > 0`.**
     (a) It does not make sense to have input dates before the curve start date ($t = 0$).
     (b) We only need to verify `dates[0]>0` because the dates are sorted in increasing order.
     (c) **Test if `dates[0]<=0`.** If yes, return 1 (fail) and exit.
- We require sorted dates because we cannot perform a bootstrap otherwise.
- Similarly, we canot allow duplicate dates.
  1. **Do not attempt to sort `dates` yourself, or to remove duplicates.**
  2. There is a reason why I want `dates` to be `const`, and it's not just finance.
  3. In our simple model, we only support a semiannual frequency, so `dates[0] = 0.5`, `dates[1] = 1.0`, etc. Hence we do not really need `dates` as an input. However, a real yield curve has additional points, for example at one week or one month, etc. Also, for long dated bonds, we do not really have par bonds every six months out to 30 years. So we need a set of input dates. There is nothing to gain by oversimplifying.
- If the tests pass, we populate the class with data.
- **We insert a "dummy" entry at $t = 0$.**

```
tenors.push_back( 0.0 );
par_yields.push_back( 0.0 );
```

  1. I suppose it is not necessary to do this.
  2. However, we would like to have some record of the start date of our yield curve.
  3. It is nice to start from "zero" with a discount factor $= 1$ at $t = 0$.
- **Finish the job. Use `dates` to populate `tenors` and `yields` to populate `par_yields`.**
- Return 0 (success) and exit.

3.13 Bootstrap of par yield curve

- We now perform the bootstrap to obtain the spot rates.
- There are no validation tests.
  1. This is a private method and the inputs have already been validated by now.
  2. We have a valid set of tenors and par yields.
- Insert a dummy "zero point" at time zero. The discount factor equals 1.

```
discount_factors.push_back( 1.0 );
spot_rates.push_back( 0.0 );
```

- Compute the 6 month point.
  There are no negative yields, hence this calculation is guaranteed to succeed.
  Remember to use `par_yields[1]` because of the "zero point" at $t = 0$.

```
double y_dec = 0.01*par_yields[1];
double df = 1.0/(1.0 + 0.5*y_dec);
discount_factors.push_back( df );
double r = -100.0 * log(df) / (tenors[1] - tenors[0]);
spot_rates.push_back( r );
```

- Then set the spot rate at $t = 0$ to the value at the 6 month point.

```
spot_rates[0] = spot_rates[1];
```

- Now begin the main bootstrap loop.
  1. We begin the loop at $i = 2$ (because of the above two steps).

     ```
     for (int i = 2; i < par_yields.size(); ++i)
     ```

  2. **What if `par_yields.size() = 2`,** i.e. the input data to `create_curves()` had only one date and yield (and we added a zero point)?
     (a) Nothing will happen. We simply won't enter the loop. So what?
     (b) ***Don't laugh.*** I recall there was an example a few years ago (one of the Persian Gulf countries), and they had a brand new yield curve and it only had one point. (Maybe it had three points.)
     (c) Our software will deal with it.
  3. **Calculate the discount factor `df` using the formulas from lectures.**

     ```
     double df = (you calculate it)
     ```

  4. **Don't rush.** ***The bootstrap may fail.***

5. Remember `df` is just a calculated number. If `df <= 0.0` break out of the loop.

   ```
   if (df <= 0.0) break;
   ```

   (a) **We have a partially constructed spot curve.**

   (b) But it still has some useful content. It is not a complete failure.

6. If `df > 0.0`, push it onto the discount factors vector.

   ```
   discount_factors.push_back( df );
   ```

7. Calculate the spot rate and push it onto the spot rates vector.

   ```
   double r = -100.0 * log(df) / (tenors[i] - tenors[0]);
   spot_rates.push_back( r );
   ```

8. Continue the loop to the next tenor.

- After the loop completes, we have a fully or partially constructed spot curve.
- Exit the function.

3.14 Accessor methods: "get" stuff

- These are all `const` methods.
- We do not want the calling application to change the data values in our object.
- This is why the return type is **const** `double*`.
- **If the value of n is out of bounds, we return a null pointer.**
  1. It is a simple way to return an error status.
  2. We could make the return type `int` and return an error status, but that just makes the interface clumsy. Why bother?
  3. Let us *not* throw an exception. *What's the point?* We are not corrupting memory or generating a NAN output or anything serious.

```
const double* YieldCurve::df(int n) const
{
  if ((n < 0) || (n >= discount_factors.size())) return 0;
  return &discount_factors[n];
}

const double* YieldCurve::spot(int n) const
{
  if ((n < 0) || (n >= spot_rates.size())) return 0;
  return &spot_rates[n];
}

const double* YieldCurve::tenor(int n) const
{
  if ((n < 0) || (n >= tenors.size())) return 0;
  return &tenors[n];
}

const double* YieldCurve::yield(int n) const
{
  if ((n < 0) || (n >= par_yields.size())) return 0;
  return &par_yields[n];
}
```

3.15 Interpolation of the spot curve

- The function signature is:

```
int interpolate(int algorithm, double t, double & df, double & r) const;
```

  1. The input parameter `algorithm` specifies the interpolation algorithm.
  2. We shall use 1 for linear and 2 for constant forward yield.
  3. The input parameter `t` is obviously the interpolation time.
  4. The output parameters are the discount factor `df` and spot rate `r`.
  5. The return type is `int` because we shall perform validation tests.
  6. The return type cannot be `const double*` with a null pointer for failure, because the function returns *two* outputs.

- **Initialize `df = 0` and `r = 0`.**
- **Validation test: if `t < tenors[0]`, return 1 (fail) and exit.**
- Next we must deal with a situation which was not discussed in the lectures.
- *What if the value of t is beyond the last tenor in the spot curve?*
- Then there is nothing we can interpolate. We simply set r to the last rate in the spot curve.
  1. **Define `int num_spots = length_spot_curve()`.**
  2. **Test if `t >= tenors[num_spots-1]`.**
  3. If yes, then set `r = spot_rates[num_spots-1]`.
  4. *You can also write* `r = spot_rates.back()`.
- *Else ...* **Find the value of i such that `tenor[i] <= t` and `t < tenor[i+1]`.**
- We know that a solution for i exists.
- **Special case $i = 0$.**
  1. In this case, by definition `spot_rates[0] = spot_rates[1]`.
  2. Therefore `r = spot_rates[0]` and we do not need to interpolate.
- Otherwise we must interpolate.
  1. **Calculate `double lambda = (t - tenors[i])/(tenors[i+1] - tenors[i])`.**
  2. Calculate `r` using the relevant interpolation formula.
  3. It is simplest to use a `switch` statement with a default *(see below)*.
- In all cases, calculate `df = exp(-0.01*r*(t - tenors[0]))`.
- Return 0 (success) and exit.

- **Hence (after the validation tests) the overall code structure looks like this:**

```
int num_spots = spot_rates.size();

// beyond last tenor of spot curve
if (t >= tenors[num_spots-1]) {
  r = spot_rates[num_spots-1];
}
else {
  // interpolate
  // find value of i such that (tenors[i] <= t) && (t < tenors[i+1])

  if (i == 0) {
    r = spot_rates[0];
  }
  else {
    double lambda = (t - tenors[i])/(tenors[i+1] - tenors[i]);

    switch (algorithm) {
    default:
    case 1:  // linear
      r = (linear interpolation)
      break;
    case 2:  // constant forward rate
      r = (constant forward rate interpolation)
      break;
    }
  }
}
df = exp(-0.01*r*(t - tenors[0]));
return 0;
```

- **Instead of "`case 1`" and "`case 2`" it would be more elegant to define an `enum`.**

```
class YieldCurve
{
public:
  enum interpolation_algorithm {DEFAULT=0, LINEAR=1, CFR=2 };
  int interpolate(interpolation_algorithm algorithm, double t,
                  double & df, double & r) const;

  // etc
```

- I decided not to do this because it just adds too much material into the class.
- Writing an `enum` has nothing to do with finance.

3.16 Tests I

- **Verify that your code matches the worked examples in the lectures.**
- **Verify that your code matches your answer in HW1 Q1.4 (if you did the homework correctly).**
- **If you did Midterm 1 correctly (Question 2), verify that your code matches the numbers you calculated in Midterm 1.**

3.17 Tests II

- Test your code using the sample test cases below. *(See Part 1 of the assignment.)*
- Set the input array length to 20.
- Initialize `date[i] = 0.5*(i+1)`, i.e. semi-annual dates.
- Try these cases for the yields:
  1. Flat: `yield[i] = 5.0;`
  2. Normal: `yield[i] = 1.0 + 0.9*log(i+1);`
  3. Inverted: `yield[i] = 8.0/(1.0 + 0.03*i);`
  4. Humped: `yield[i] = 1.0 + 0.16*i - 0.007*i*i;`
- Case 1 is a flat par yield curve. The spot rates should also be all equal (flat spot curve).
- Case 2 is an upward sloping ("normal") par yield curve. This is the most common situation in economics. The spot rates should increase with the value of i (upward sloping spot curve). If you plot a graph of the yield curves, typically the spot curve will be higher than the par yield curve. Because we use continuous compounding, this is not exactly true in my example. But for longer maturities (≥ 3 years in ny example), the spots are higher than the yields.
- Case 3 is a downward sloping ("inverted") par yield curve. The spot rates should decrease with the value of i (downward sloping spot curve). If you plot a graph of the curves, the spot curve will be lower than the par yield curve.
- Case 4 is a "humped" par yield curve. The yields initially increase, then decrease. The spot curve can be either above or below the par yield curve, depending on the maturity.
- In practice, most of the time the yield curve slopes upwards. In economics, an inverted or downward sloping yield curve is usually bad news. It typically indicates a recession is coming.
- Interpolate in steps of $\Delta t = 0.1$ (for example) and plot graphs of the spot curves.

3.18 Tests: bootstrap failure

- *Let us try a case where the bootstrap fails and the spot curve is only partially constructed.*
- Use an array of 20 items with `date[i] = 0.5*(i+1)` and `yield[i] = 1.0 + 2.0*i`.
- **In practice, there are constraints how rapidly the par yields can vary with the maturity. If the yields change too rapidly the bootstrap can fail. A complete spot curve is not always guaranteed to exist. *This has happened in real life.***
- Run this code (for example), print the output to file and plot a graph of the results.

```
void yc_bootstrap_fail()
{
  std::ofstream ofs("output.txt");

  const int n = 20;
  std::vector<double> date(n, 0.0);
  std::vector<double> yield(n, 0.0);
  for (int i = 0; i < n; ++i) {
    date[i] = 0.5*(i+1);
    yield[i] = 1.0 + 2.0*i;
  }

  YieldCurve yc;
  int rc = yc.create_curves(n, date, yield);

  std::cout << "yc_create_curves  rc = " << rc << std::endl;

  int len_par = yc.length_par_curve();
  int len_spot = yc.length_spot_curve();
  std::cout << "length par curve = " << len_par << std::endl;
  std::cout << "length spot curve = " << len_spot << std::endl;

  for (int i = 0; i < len_par; ++i) {
    const double *tenor = yc.tenor(i);
    const double *yield = yc.yield(i);
    double t = *tenor;
    double df = 0;
    double r = 0;
    yc.interpolate(t, df, r);
    ofs.width(16); ofs << *tenor << " ";
    ofs.width(16); ofs << *yield << " ";
    ofs.width(16); ofs << r << " ";
    ofs << std::endl;
  }
}
```