`actuarydesk` (Actuarial Modelling Package) experimental ver 0.

Arief Anbiya

August 23, 2020

Contents

1 `financial_math.py`

1.1 `Contribution` and `InterestRate`

Simplest usage of this class:

```
import financial_math as fm

deposit = fm.Contribution(0, 1000)
int_rate = fm.InterestRate(0, 0.04, 'annual')
```

The `deposit` has attributes `t` (`0`), which is time of deposit, and `amount` (`1000`). Time interval of length 1 is considered as one year. `int_rate` has attributes: `t` (`0`), which is starting time of interest rate, `rate` (`0.04`), `period_desc` (`'annual'`), `period_length` (`1`), `discount` (`False`), and `compound` (`True`).

Application example: An investor deposit 1000 at time 1, 1250 at time 3, and 5000 at time 10. The interest rates is 5% compounded annualy for the first 5 years, and 3% of compound discount rate (semi annually) for years after that. What is the accumulated value at time 8 and 20? The solution can be obtained as follows:

```
import financial_math as fm

deposits = [fm.Contribution(1, 1000), \
           fm.Contribution(3, 1250), \
           fm.Contribution(10, 5000)]

int_rates = [fm.InterestRate(0, 0.05, 'annual'), \
            fm.InterestRate(5, 0.03, 'semi-annual', discount = True)]

acc_value_at_8 = [d.accumulate(8, int_rates) for d in deposits]
print(sum(acc_value_at_8))

acc_value_at_20 = [d.accumulate(20, int_rates) for d in deposits]
print(sum(acc_value_at_20))
```

The `Contribution.accumulate(t_end, interest_rates)` returns the accumulated value at time `t_end` using interest rates in the list `interest_rates`. If `t_end` is less than time of deposit, than the method returns 0. `Contribution` class also has method `Contribution.value_at`, which is more general than `Contribution.accumulate`, it honours time value of money so it can calculate both future and past value including present value of the money. Another useful method is `Contribution.filter_interest_rates(interest_rates, t_target)`, it returns the necessary interest rates for calculating the value of money at time `t_target`. See following example:

```
import financial_math as fm

deposit = fm.Contribution(10, 10000)
int_rates = [fm.InterestRate(0, 0.04, 'annual'), \
            fm.InterestRate(5, 0.03, 'semi-annual'), \
            fm.InterestRate(13, 0.04, 'annual', discount = True), \
            fm.InterestRate(20, 0.05, 'annual'), \
            fm.InterestRate(23, 0.02, 'quarter')]

filtered_1 = deposit.filter_interest_rates(int_rates, 0)

filtered_2 = deposit.filter_interest_rates(int_rates, 7)

filtered_3 = deposit.filter_interest_rates(int_rates, 11)

filtered_4 = deposit.filter_interest_rates(int_rates, 22.5)

filtered_5 = deposit.filter_interest_rates(int_rates, 30)
```

The `Contribution.value_at` and `Contribution.accumulate` use `Contribution.filter_interest_rates` in the implementation.

(*Note: if there are two or more interest rates with same time point, then interest rate with the lowest index on the list will be used. And if there is no interest rate with time point 0, then the earliest interest rate will be set to start at time 0.)

1.2 `FinancialTL` class

This class provides methods for accumulating contributions with varying interest rates and also the value of total contributions at a particular time point. See example below:

```
import financial_math as fm

deposits = [fm.Contribution(0, 100), \
           fm.Contribution(1, 100), \
           fm.Contribution(1.5, 200)]
int_rates = [fm.InterestRate(0, 0.04, 'annual'), \
            fm.InterestRate(1, 0.03, 'semi-annual')]
timeline = fm.FinancialTL(deposits, int_rates)

print(timeline.acc_value_at_point(5))
dynamics = timeline.acc_value_dynamics(5, 0.5)
dynamics_df = timeline.acc_value_dynamics_df(5, 0.5)
present_value = timeline.value_at(0)
```

The `FinancialTL(contributions, int_rates)` accepts two arguments (two lists) for initialization. The `FinancialTL.acc_value_at_point(t)` finds the accumulated value of all investments at time `t`. The `FinancialTL.acc_value_dynamics(t_end, dt)` returns two lists, the list of time points from 0 to `t_end` with difference of `dt`, and the list of the accumulated values at those time points. `FinancialTL.acc_value_dynamics_df(t_end, dt)` is similar but it returns a desired Pandas dataframe. `FinancialTL.value_at` honors the time value of money, it calculates the value of all contributions at a particular time.

1.3 `Bond` class

The `Bond(t_start, term, face_value, nominal_coupon_rate, coupon_period, redemption_value, interest_rates)` object initiation requires 7 arguments. We can calculate the price of a bond by just applying `Bond.price` (see example below).

```
import financial_math as fm

int_rate = [fm.InterestRate(0, 0.03, 'annual')]
bond = fm.Bond(2, 10, 10000, 0.1, 'semi-annual', 5000, int_rate)
print(bond.price)
```

2 `actuarial_tables.py`

This module provides ready to use actuarial tables (only mortality for the current version). One way to access a table is by using the `table_dictionary` like the following example.

```
import actuarial_tables as at

table = at.table_dictionary['UK_ONS_2016_to_2018_male']
```

By the above example, `table` is a dictionary and `table['x']` will return a list of ages. The `table['qx']` and `table['px']` will return a list of probabilities q_x and p_x respectively, with the age being the age in the corresponding same position/index in `table['x']`. Another useful tool to use the table is by using the `MortalityTable` class:

```
import actuarial_tables as at
```

```
table = at.MortalityTable('UK_ONS_2016_to_2018_male')
```

The `table` has useful methods such as `table.increase_qx`, `table.increase_px`, `table.multiple_px`, and `table.multiple.qx`. They return modified table in the form of new dictionary, without changing the original table. See example below, the `tbl.table` is the original mortality table in the form of dictionary, and `new_table` is a new dictionary of the modified original table by adding 0.2 to all q_x with value inside $[0, 0.1]$.

```
import actuarial_tables as at

tbl = at.MortalityTable('UK_ONS_2016_to_2018_male')
tbl_dictionary = tbl.table
new_table = tbl.increase_qx(0.2, value_interval=[0, 0.1])
```

3 actuarial_math.py

3.1 ${}_tp_x$ and ${}_tq_x$

The `t_p_x(x, t, table)` returns the probability of an insured aged `x` will survive in the interval $[x, x+t]$, according to mortality table `table`. The `t_q_x(x,t,table)` returns `1-t_p_x(x,t,table)`. Both functions also accept fractional `x` and `t` (assumed to be uniform inside each year).

3.2 `Benefit` class

This class is a subclass of `financial_math.Contribution`, it accepts two required arguments: `t` and `amount`. It also accepts one keyword argument: `endowment`, because the probability of an endowment benefit paid is different (this argument default to `False`). The class creates an object of insurance benefit that is paid at time `t` with amount of `amount`. This class is useful because it can calculate the probability of this benefit is paid for an insured (aged `x` at policy start), by `Benefit.claim_prob(x, table)` and also its actuarial present value by `Benefit.actuarial_pv(x, table, interest_rates)`.

3.3 `Premium` class

This is also a subclass of `financial_math.Contribution`. This class is used to create a premium object, paid at time `t` of amount `amount`. We can calculate the probability of payment and its actuarial present value using `Premium.payment_prob` and `Premium.actuarial_pv` respectively. We can also set some portion of the premium for investment in a collective fund (see example below).

```
import financial_math as fm
import actuarial_math as am

int_rate = [fm.InterestRate(0, 0.03, 'annual')]
premium = am.Premium(0, 100)
print(premium.amount)
premium.partial_transform_to_collective_fund(0.4)
print(premium.amount)
collective_fund_contribution = premium.collective_fund_premium()
print(collective_fund_contribution.amount)
```

By default, `Premium.collective_fund_premium` will return a `Premium` object of amount 0 with the same time as the main premium. The collective fund premium is used in cashflow calculation (cashflow can be calculated using `ActuarialModel` class).

3.4 `TermLifeInsurance` class, `TermLifeAnnuity` class, and `find_premium` function

This class is used to create a term life insurance model. The code below demonstrates a case where an insured aged 30 starts a policy of a simple 10 year term life insurance product with benefit of 8000 payable at end year of death. Assuming an interest rate of 3% compounded annually.

```
import financial_math as fm
import actuarial_tables as at
import actuarial_math as am

table = at.UK_ONS_2016_to_2018_male
int_rate = [fm.InterestRate(0, 0.03, 'annual')]
x = 30; term = 10;
benefits = [am.Benefit(i, 8000) for i in range(x+1, x+term+1)]
ins_model = am.TermLifeInsurance(x, term, benefits, int_rate, table)
level_premium = ins_model.minimum_level_premium(annuity='due')
```

The `am.TermLifeInsurance(x, term, benefits, int_rate, table)` creates an object of the insurance model using the mortaility table `table`. The method `TermLifeInsurance.minimum_level_premium(annuity)` returns the amount of yearly level premium such that the expected loss is 0. In this particular case, `ins_model.minimum_level_premium(annuity='due')` returns P such that

$$P\ddot{a}_{30:\overline{10}|} = 8000A_{30:\overline{10}|}$$

We can also see the expected expense from the insurer's perspective, using the `TermLifeInsurance.expected_expense_df(t_end)` until some `t_end`. Also, the `TermLifeInsurance.acc_expected_expense(t_end, interest_rates)` returns the total accumulated expense at `t_end`. If at some point we change the benefits or interest rates used in the model, we should also run `TermLifeInsurance.adjustments()` after the benefits and/or interest rates are edited by the user (this will sort the list of interest rates and benefits from earlier time to older time, and deletes time-duplicate interest rates), this method is applied in every initialization of a `TermLifeInsurance` object.

The `TermLifeInsurance.calculate_apv(point)` returns tha actuarial present value at time point `point`, neglecting all possible expenses before that time. This is useful for calculating reserves (reserves can be calculated using `ActuarialModel` class).

The `TermLifeAnnuity` class can be used to create a model of an insurance product with custom premium amounts and timepoints. The example below calculates the expected loss of an insurance policy with benefit amount of 10000 payable at end year of death for 10 years, and with premium amount of 4000 that must be paid at beginning of policy and at the beginning of 2nd policy year.

```
import financial_math as fm
import actuarial_tables as at
import actuarial_math as am

table = at.UK_ONS_2016_to_2018_male
int_rate = [fm.InterestRate(0, 0.06, 'annual')]
x = 30; term = 10;
benefits = [am.Benefit(i, 10000) for i in range(x+1, x+term+1)]
```

```
ins_model = am.TermLifeInsurance(x, term, benefits, int_rate, table)
premiums = [am.Premium(x, 4000), \
           am.Premium(x+1, 4000)]
ann_model = am.TermLifeAnnuity(x, term, premiums, int_rate, table)
expected_loss = ins_model.calculate_apv() - ann_model.calculate_apv()
```

The expected loss above is relatively large positive value. To find the amount of premium such that we get the desired expected loss, consider another useful tool in this module, the `find_premium` function. Key inputs are the list of payable benefits, and the time points of the payable premiums. Using the same profile from code above, we can search for the desired premium amount such that the expected loss is -500 with error of 5, which means $-505 \leq E[L] \leq -495$, see following example:

```
import financial_math as fm
import actuarial_tables as at
import actuarial_math as am

table = at.UK_ONS_2016_to_2018_male
int_rate = [fm.InterestRate(0, 0.03, 'annual')]
x = 30; term = 10;
benefits = [am.Benefit(i, 10000) for i in range(x+1, x+term+1)]
ins_model = am.TermLifeInsurance(x, term, benefits, int_rate, table)

custom_premium = am.find_premium(x, term, benefits, int_rate, table, \
                              premium_time_points = [x, x+1] , loss_bound = 5,
                                  loss_target=-500, \
                              start_premium=10, dp = 0.5, max_iter=1000)
```

`find_premium` is an algorithm that tries many possible values for the premium amount, incremental, starts from `start_premium` and the increment is `dp`. `max_iter` is the maximum number of attempts after the starting premium. After reaching maximum iteration, the function will just return `None`.

3.5 `ActuarialModel` class

With this class, we can calculate reserves and cashflow of a given policy. The `ActuarialModel` requires 3 inputs, the insurance model object, the annuity model object, and the start year of the policy, the fourth optional input is the month when the policy starts (default to 1, equals January). If the second required argument is not an annuity model object, then it must be a list of two strings. The string in the first index must be either `"due"` or `"immediate"`, which means the premium payments are level at the beginning or end of each period respectively until term ends. The string in the second index must be either `"annual"`, `"semi-annual"`, `"quarter"`, or `"month"` to setup the payment period. See example below:

```
import financial_math as fm
import actuarial_tables as at
import actuarial_math as am

table = at.UK_ONS_2016_to_2018_male
int_rate = [fm.InterestRate(0, 0.05, 'annual')]
x = 50; term = 15;
benefits = [am.Benefit(i, 10000) for i in range(x+1, x+term+1)]
ins_model = am.TermLifeInsurance(x, term, benefits, int_rate, table)
model = am.ActuarialModel(ins_model, ['due', 'annual'], 2020)
```

```
reserves = model.reserves_dynamics_df(0.5, 70)
cashflow = model.cashflow_df(65)
```

`model.reserves_dynamics_df(dt=0.5, t_end=70)` returns a Pandas dataframe that shows the reserves at each time point starting from `x`, the incremental by `dt=0.5`, until maximum of `t_end=70`. The `model.cashflow_df(65)` returns a Pandas dataframe with columns:

```
Index(['t', 't_policy', 'profit_exclude_collective_fund', 'income', 'expense',
       'income_from_collective_fund',
       'accumulated_profit_exclude_collective_fund',
       'accumulated_income_collective_fund', 'total_accumulated_profit'],
      dtype='object')
```

Here are snapshots of the above example's `reserves`, `cashflow.income`, and `cashflow.expense`:

```
       t      Reserves
0   50.0  1.136868e-13
1   50.5  4.069317e+01
2   51.0  2.486006e+01
3   51.5  6.538695e+01
4   52.0  4.936419e+01
5   52.5  8.872740e+01
6   53.0  7.146854e+01
7   53.5  1.106505e+02
8   54.0  9.318802e+01
9   54.5  1.312419e+02
10  55.0  1.125823e+02
11  55.5  1.489531e+02
12  56.0  1.285155e+02
13  56.5  1.629267e+02
14  57.0  1.404230e+02
15  57.5  1.727978e+02
16  58.0  1.481499e+02
17  58.5  1.780317e+02
18  59.0  1.507629e+02
19  59.5  1.777954e+02
20  60.0  1.475350e+02
21  60.5  1.710155e+02
22  61.0  1.370296e+02
23  61.5  1.563647e+02
24  62.0  1.180351e+02
25  62.5  1.332132e+02
26  63.0  9.053292e+01
27  63.5  9.941138e+01
28  64.0  5.013702e+01
29  64.5  5.425029e+01
30  65.0  0.000000e+00
31  65.5  0.000000e+00
32  66.0  0.000000e+00
33  66.5  0.000000e+00
34  67.0  0.000000e+00
35  67.5  0.000000e+00
36  68.0  0.000000e+00
37  68.5  0.000000e+00
38  69.0  0.000000e+00
39  69.5  0.000000e+00
```

```
>>> cashflow.income
0     55.748695
1     55.560488
2     55.363526
3     55.146612
4     54.921779
5     54.678475
6     54.411316
7     54.118529
8     53.800853
9     53.454914
10    53.078912
11    52.667657
12    52.217927
13    51.733449
14    51.195525
15     0.000000
Name: income, dtype: float64
```

```
>>> cashflow.expense
0       0.000000
1      33.760000
2      35.330321
3      38.909304
4      40.329686
5      43.642901
6      47.922024
7      52.519129
8      56.983534
9      62.053378
10     67.445859
11     73.769513
12     80.670788
13     86.903905
14     96.490941
15    102.099582
Name: expense, dtype: float64
```

3.6 `combined_two_cashflows` and `combined_cashflows`

We can also generate combined cashflow from different policies:

```
import financial_math as fm
import actuarial_tables as at
import actuarial_math as am
```

```
table = at.UK_ONS_2016_to_2018_male
int_rate = [fm.InterestRate(0, 0.05, 'annual')]

x = 50; term = 15;
benefits = [am.Benefit(i, 10000) for i in range(x+1, x+term+1)]
ins_model = am.TermLifeInsurance(x, term, benefits, int_rate, table)
model = am.ActuarialModel(ins_model, ['due', 'annual'], 2020)
cashflow = model.cashflow_df(x+term)

x_2 = 30; term_2 = 20;
benefits_2 = [am.Benefit(i, 10000) for i in range(x_2+1, x_2+term_2+1)]
ins_model_2 = am.TermLifeInsurance(x_2, term_2, benefits_2, int_rate, table)
model_2 = am.ActuarialModel(ins_model_2, ['due', 'annual'], 2020)
cashflow_2 = model_2.cashflow_df(x_2+term_2)

combined_df = am.combined_two_cashflows(cashflow, cashflow_2, int_rate)
```

```
import math

import financial_math as fm
import actuarial_tables as at
import actuarial_math as am

table = at.UK_ONS_2016_to_2018_male
int_rate = [fm.InterestRate(0, 0.05, 'annual')]

xs = [20, 20.5, 35, 40, 55, 50, 43.5]
terms = [20, 20, 30, 30, 30, 20, 20]
pol_yrs = [2020, 2020, 2018, 2020, 2019, 2017, 2020]
cashflows = []
for i in range(len(xs)):
    x = math.floor(xs[i]); term = terms[i]
    t_points = [x + j for j in range(1, term+1)]
    benefits = [am.Benefit(j, 10000) for j in t_points]
    ins_model = am.TermLifeInsurance(x, term, benefits, int_rate, table)
    model = am.ActuarialModel(ins_model, ['due', 'annual']', pol_yrs[i])
    cashflow = model.cashflow_df(x+term)
    cashflows.append(cashflow)

combined_df = am.combined_cashflows(cashflows, int_rate)
```

References

[1]