assn2

February 27, 2025

1 Environment

1.1 Imports

[126]:
```
:dep itertools
:dep polars = { version = "0.46", features = [ "lazy", "list_arithmetic",␣
↪"round_series", "log", "range"] }
:dep statrs
```

[127]:
```
use polars::prelude::*;
use itertools::izip;
use std::f64::consts::E;
```

1.2 Utilities

Polars does not natively provide the ability to get the normal cdf, so we have to use an alternative library and use arbitrary mapping in order to use the function conveniently.

[128]:
```
use statrs::distribution::ContinuousCDF;

pub fn N(e: Expr) -> Expr {
    e.map(
        |s| {
            Ok(Some(Column::new(
                "".into(),
                s.f64()
                    .unwrap()
                    .into_iter()
                    .map(|ca| {
                        ca.map(|f| {
                            statrs::distribution::Normal::standard()
                                .cdf(f)
                        })
                    })
                    .collect::<Vec<_>>(),
            )))
        },
        GetOutput::from_type(DataType::Float64),
```

```
    )
}
```

2 Problem 1

Initialize data. I used the example in the slides as well in order to verify calculations.

[129]:
```
let mut df = df!(
    "name" => ["example", "problem 1"],
    "T" => [10.0/12.0, 0.5],
    "B_0" => [960.0, 904.0],
    "K" => [1000.0, 900.0],
    "sigma" => [0.09, 0.05],
    "r_m" => [0.10, 0.025],
    "payment frequency" => [0.5, 0.5],
    "coupon payment" => [50.0, 10.0],
    "time to next payment" => [0.25, 0.25],
    "option type" => ["call", "put"]
)?;
df
```

[129]: shape: (2, 10)

name	T	B_0	K	…	payment frequency	coupon payment	time to next payment	option type
---	---	---	---		---	---	---	---
str	f64	f64	f64		f64	f64	f64	str
example	0.833333	960.0	1000.0	…	0.5	50.0	0.25	call
problem 1	0.5	904.0	900.0	…	0.5	10.0	0.25	put

Add the interest rate curve separately, as the `df!` macro does not support lists.

[130]:
```
df.with_column(Column::new(
    "r curve".into(),
    &[
        Series::new(
```

```
            "".into(),
            &[0.09, 0.095],
        ),
        Series::new("".into(), &[0.025]),
    ],
))?;

df
```

[130]: shape: (2, 11)

name	T	B_0	K	…	coupon payment	time to next payment	option type	r curve
---	---	---	---		---	---	---	---
str	f64	f64	f64		f64	f64	str	list[f64]
example	0.833333	960.0	1000.0	…	50.0	0.25	call	[0.09, 0.095]
problem 1	0.5	904.0	900.0	…	10.0	0.25	put	[0.025]

Declare expressions for each column.

[131]:
```
let name = col("name");

let T = col("T");

let B_0 = col("B_0");

let K = col("K");

let sigma = col("sigma");

let r_m = col("r_m");

let payment_frequency =
    col("payment frequency");
```

```
let coupon_payment = col("coupon payment");

let time_to_next_payment =
    col("time to next payment");

let option_type = col("option type");

let r_curve = col("r curve");
```

We will need the time of last payment to get the range of coupon payments.

[132]:
```
let time_of_last_payment =
    (time_to_next_payment.clone()
        + (((T.clone()
            - time_to_next_payment
                .clone())
            / payment_frequency.clone())
        .floor()
        .cast(DataType::Float64)
            * payment_frequency.clone()))
    .alias("time of last payment");

df.clone()
    .lazy()
    .select([
        name.clone(),
        time_of_last_payment.clone(),
    ])
    .collect()?
```

[132]: shape: (2, 2)

name	time of last payment
---	---
str	f64
example	0.75
problem 1	0.25

We will precompute the “dirty” strike price K here as well.

[133]:
```
let dirty_K = (K.clone()
    + (coupon_payment.clone()
        / payment_frequency.clone())
        * (T.clone()
            - time_of_last_payment.clone()))
.alias("dirty K");
```

```
df.clone()
    .lazy()
    .select([name.clone(), dirty_K.clone()])
    .collect()?
```

[133]:
```
shape: (2, 2)

 name        dirty K
 ---         ---
 str         f64

 example     1008.333333
 problem 1   905.0
```

Calculate sum of coupon payments I.

[134]:
```
let I = map_multiple(
    move |cols| match cols {
        [b, c, d, e, f] => {
            let (b, c, d, e, f) = (b.clone(), c.clone(), d.clone(), e.clone(),
 ↪f.clone());

            let time_to_next_payment = b.f64()?;
            let time_to_maturity = c.f64()?;
            let coupon_payment = d.f64()?;
            let payment_frequency = e.f64()?;
            let r_curve = f.list()?;

            let res: Float64Chunked = izip!(
                time_to_next_payment,
                time_to_maturity,
                coupon_payment,
                payment_frequency,
                r_curve
            )

            .map(|(t1, tm, c, dt, r)| match (t1, tm, c, dt, r) {
                (Some(t1), Some(tm), Some(c), Some(dt), Some(r)) => {
                    let r_vec: Vec<f64> = r.f64().unwrap().to_vec_null_aware().
 ↪left().unwrap();
                    let num_payments = ((tm - t1) / dt).ceil() as usize;
                    Some(
                        (0..num_payments)
                            .map(|i| {
                                let t = t1 + (i as f64) * dt;
```

```
                                (-r_vec[i % r_vec.len()] * t).exp() * c
                            })
                            .sum::<f64>(),
                    )
                }
                _ => None,

            })
            .collect();
            Ok(Some(res.into_column()))
        }
        _ => Err(PolarsError::ComputeError(
            "Expected exactly 5 columns".into(),
        )),
    },

    &[
        time_to_next_payment.clone(),
        T.clone(),
        coupon_payment.clone(),
        payment_frequency.clone(),
        r_curve.clone(),
    ],
    GetOutput::from_type(DataType::Float64),
)
.alias("I");

df.clone()
    .lazy()
    .select([name.clone(), I.clone()])
    .collect()?
```

[134]: shape: (2, 2)

name	I
---	---
str	f64
example	95.449015
problem 1	9.937695

The discount rate function $P(0, T)$.

[135]:
```
let P = ((-r_m.clone() * T.clone()).exp())
    .alias("P");

df.clone()
```

```
    .lazy()
    .select([name.clone(), P.clone()])
    .collect()?
```

[135]:
```
shape: (2, 2)

  name        P
  ---         ---
  str         f64

  example     0.920044
  problem 1   0.987578
```

The forward bond price F_B.

[136]:
```
let F_B = ((B_0.clone() - I.clone())
    / P.clone())
.alias("F_B");

df.clone()
    .lazy()
    .select([name.clone(), F_B.clone()])
    .collect()?
```

[136]:
```
shape: (2, 2)

  name        F_B
  ---         ---
  str         f64

  example     939.683967
  problem 1   905.308224
```

We'll need d_1 and d_2 for our Black-Scholes calculation.

[137]:
```
let d_1 = (((F_B.clone() / K.clone()
    - lit(1.0))
.log1p()
    + sigma.clone().pow(2) * T.clone()
        / lit(2.0))
    / (sigma.clone() * T.clone().sqrt()))
.alias("d_1");

let d_2 = (d_1.clone()
    - sigma.clone() * T.clone().sqrt())
.alias("d_2");
```

```
df.clone()
    .lazy()
    .select([
        name.clone(),
        d_1.clone(),
        d_2.clone(),
    ])
    .collect()?
```

[137]: shape: (2, 3)

name	d_1	d_2
---	---	---
str	f64	f64
example	-0.716137	-0.798295
problem 1	0.184009	0.148654

These are the Black-Scholes formulas for call and put options c and p.

[138]:
```
let c = (P.clone()
    * (F_B.clone() * N(d_1.clone())
        - K.clone() * N(d_2.clone())))
.alias("c");

let p = (P.clone()
    * (K.clone() * N(-d_2.clone())
        - F_B.clone() * N(-d_1.clone())))
.alias("p");

df.clone()
    .lazy()
    .select([
        name.clone(),
        c.clone(),
        p.clone(),
    ])
    .collect()?
```

[138]: shape: (2, 3)

name	c	p
---	---	---
str	f64	f64
example	9.487262	64.980691

```
problem 1   15.367536   10.125251
```

This is a single expression that will give the correct option price based on the direction of the option. **This is the answer to Problem 1**.

[139]:
```
let option_price = (when(
    option_type.clone().eq(lit("call")),
)
.then(c.clone())
.otherwise(p.clone()))
.alias("clean option price");

df.clone()
    .lazy()
    .select([
        name.clone(),
        option_price.clone(),
    ])
    .collect()?
```

[139]:
```
shape: (2, 2)

  name        clean option price
  ---         ---
  str         f64

  example     9.487262
  problem 1   10.125251
```

We will now use the dirty K computed earlier in order to get the option price assuming that the strike price is dirty. **This is the other answer to Problem 1**.

[140]:
```
let dirty_d_1 = ((F_B.clone()
    / dirty_K.clone()
    - lit(1.0))
.log1p()
    + sigma.clone().pow(2) * T.clone()
        / lit(2.0))
    / (sigma.clone() * T.clone().sqrt());

let dirty_d_2 = dirty_d_1.clone()
    - sigma.clone() * T.clone().sqrt();

let dirty_c = P.clone()
    * (F_B.clone() * N(dirty_d_1.clone())
        - dirty_K.clone()
```

```
            * N(dirty_d_2.clone()));

let dirty_p = P.clone()
    * (dirty_K.clone()
        * N(-dirty_d_2.clone())
        - F_B.clone()
            * N(-dirty_d_1.clone()));

let dirty_option_price = (when(
    option_type.clone().eq(lit("call")),
)
.then(dirty_c.clone())
.otherwise(dirty_p.clone()))
.alias("dirty option price");

df.clone()
    .lazy()
    .select([
        name.clone(),
        dirty_option_price.clone(),
    ])
    .collect()?
```

[140]: shape: (2, 2)

name	dirty option price
---	---
str	f64
example	7.968597
problem 1	12.4561

3 Problem 2

Initialize data.

[141]:
```
let df = df!(
    "name" => ["example", "problem 2"],
    // assumes that yield curve is flat with continuous compounding
    "yield curve" => [0.06, 0.0405],
    "L" => [100_000_000.0, 5_000_000.0],
    "n" => [3.0, 5.0],
    "sigma" => [0.20, 0.15],
    "m" => [0.5, 1.0],
    "T" => [5.0, 2.0],
    "s_k" => [0.062, 0.0415],
```

```
    "yield curve compounding" => ["continuous", "annual"],
)?;
```

Declare expressions for each column.

[142]:
```
let name = col("name");

let yield_curve = col("yield curve");

let L = col("L");

let n = col("n");

let sigma = col("sigma");

let m = col("m");

let T = col("T");

let s_k = col("s_k");

let yield_curve_compounding =
    col("yield curve compounding");
```

In the case where the yield curve is continuous, the value for s_0 is discounted, as in the example. Problem 2 is annual compounding, so it ignores this discount factor.

[143]:
```
let s_0 = (when(
    yield_curve_compounding
        .clone()
        .eq(lit("continuous")),
)
.then(
    ((yield_curve.clone() * m.clone()).exp()
        - lit(1.0))
        / m.clone(),
)
.otherwise(yield_curve.clone()))
.alias("s_0");

df.clone()
    .lazy()
    .select([name.clone(), s_0.clone()])
    .collect()?
```

[143]:
```
shape: (2, 2)

  name        s_0
```

```
---         ---
str         f64

example     0.060909
problem 2   0.0405
```

The sum of payoffs A is given thusly. The discount rate is appropriately changed based on whether the yield curve is continuous.

[144]:
```
let A = map_multiple(
    move |cols| match cols {
        [b, c, d, e, f] => {
            let (b, c, d, e, f) = (
                b.clone(),
                c.clone(),
                d.clone(),
                e.clone(),
                f.clone(),
            );

            let T = b.f64()?;
            let n = c.f64()?;
            let m =
                d.f64()?;
            let yield_curve = e.f64()?;
            let yield_curve_compounding = f.str()?;

            let res: Float64Chunked = izip!(
                T,
                n,
                m,
                yield_curve,
                yield_curve_compounding,
            )
            .map(|(T, n, m, r, s)| {
                match (T, n, m, r, s) {
                    (
                        Some(T),
                        Some(n),
                        Some(m),
                        Some(r),
                        Some(s)
                    ) => {
                        let num_payments =
                            (n / m).floor() as usize;
                        Some(m *
```

```
                            (1..=num_payments)
                                .map(|i| {
                                    let t = T + m * (i as f64);
                                    if s == "continuous" {
                                        (-r * t)
                                        .exp()
                                    } else {
                                        1.0 / (1.0 + r).powf(t)
                                    }
                                })
                                .sum::<f64>(),
                        )
                    }
                    _ => None,
                }
            })
            .collect();

            Ok(Some(res.into_column()))
        }
        _ => Err(PolarsError::ComputeError(
            "Expected exactly 5 columns"
                .into(),
        )),
    },
    &[
        T.clone(),
        n.clone(),
        m.clone(),
        yield_curve.clone(),
        yield_curve_compounding.clone(),
    ],
    GetOutput::from_type(DataType::Float64),
)
.alias("A");

df.clone()
    .lazy()
    .select([name.clone(), A.clone()])
    .collect()?
```

[144]: shape: (2, 2)

```
name        A
---         ---
str         f64
```

```
example     2.003558
problem 2   4.106236
```

The same values for Black-Scholes as before, just with different columns in the context of a swaption.

[145]:
```
let d_1 = (((s_0.clone() / s_k.clone()
    - lit(1.0))
.log1p()
    + sigma.clone().pow(2) * T.clone()
        / lit(2.0))
    / (sigma.clone() * T.clone().sqrt()))
.alias("d_1");

let d_2 = (d_1.clone()
    - sigma.clone() * T.clone().sqrt())
.alias("d_2");

df.clone()
    .lazy()
    .select([
        name.clone(),
        d_1.clone(),
        d_2.clone(),
    ])
    .collect()?
```

[145]: shape: (2, 3)

name	d_1	d_2
---	---	---
str	f64	f64
example	0.183911	-0.263302
problem 2	-0.008916	-0.221048

This is the swaption price. **This is the answer to Problem 2**.

[146]:
```
let swaption_price = (L.clone()
    * A.clone()
    * (s_0.clone() * N(d_1.clone())
        - s_k.clone() * N(d_2.clone())))
.alias("swaption price");

df.clone()
    .lazy()
    .select([
```

```
        name.clone(),
        swaption_price.clone(),
    ])
    .collect()?
```

[146]: shape: (2, 2)

name	swaption price
---	---
str	f64
example	2.0710e6
problem 2	61307.182857

4 Problem 3

Initialize data.

[147]:
```
let df = df!(
    "start of collar" => [0.0],
    "collar length" => [5.0],
    // assumed flat
    "r" => [0.035],
    "m" => [E],
    // assumes constant tenor
    "tenor" => [0.25],
    "R_F" => [0.031],
    "R_C" => [0.038],
    "L" => [5_000_000.0],
    "sigma" => [0.12],
)?;
```

Declare expressions for each column.

[148]:
```
let start_of_collar = col("start of collar");

let collar_length = col("collar length");

let r = col("r");

let m = col("m");

let tenor = col("tenor");

let R_F = col("R_F");
```

```
let R_C = col("R_C");

let L = col("L");

let sigma = col("sigma");
```

The reset times T_i.

[149]:
```
let T = linear_space(
    start_of_collar.clone(),
    start_of_collar.clone()
        + collar_length.clone(),
    collar_length.clone() / tenor.clone(),
    ClosedInterval::Right,
)
.alias("T");

df.clone()
    .lazy()
    .select([T.clone()])
    .collect()?
```

[149]:
```
shape: (20, 1)

 T
 ---
 f64

 0.25
 0.5
 0.75
 1.0
 1.25
 …
 4.0
 4.25
 4.5
 4.75
 5.0
```

And the payments time one tenor later, T_{i+1}

[150]:
```
let T1 =
    (T.clone() + tenor.clone()).alias("T1");

df.clone()
    .lazy()
```

```
    .select([T1.clone()])
    .collect()?
```

[150]: shape: (20, 1)

```
T1
---
f64

0.5
0.75
1.0
1.25
1.5
…
4.25
4.5
4.75
5.0
5.25
```

A dynamically adjusted $P(0, t)$ based on the compounding frequency.

[151]:
```
let P = (when(m.clone().eq(E))
    .then(
        // use continuous compounding
        (-r.clone() * T1.clone()).exp(),
    )
    .otherwise(
        // else discrete compounding
        lit(1.0)
            / (lit(1.0)
                + r.clone() * m.clone())
            .pow(T1.clone() / m.clone()),
    ))
.alias("P");

df.clone()
    .lazy()
    .select([P.clone()])
    .collect()?
```

[151]: shape: (20, 1)

```
P
---
f64
```

```
0.982652
0.974092
0.965605
0.957193
0.948854
…
0.861785
0.854277
0.846834
0.839457
0.832144
```

The forward F.

[152]:
```
let F = ((r.clone() * tenor.clone()).exp()
    - lit(1.0))
    / tenor.clone();

df.clone()
    .lazy()
    .select([F.clone()])
    .collect()?
```

[152]:
```
shape: (1, 1)

r
---
f64

0.035154
```

Calculating the cap as portfolio of put options.

[153]:
```
let d_1_C = ((F.clone() / R_C.clone()
    - lit(1.0))
.log1p()
    + sigma.clone().pow(2) * T.clone()
        / lit(2.0))
    / (sigma.clone() * T.clone().sqrt());

let d_2_C = d_1_C.clone()
    - sigma.clone() * T.clone().sqrt();

let cap = (P.clone()
    * tenor.clone()
```

```
    * L.clone()
    * (F.clone() * N(d_1_C.clone())
        - R_C.clone() * N(d_2_C.clone())))
.alias("cap");

df.clone()
    .lazy()
    .select([cap.clone()])
    .collect()?
```

[153]:

```
shape: (20, 1)

 cap
 ---
 f64

 123.207149
 366.972376
 601.697944
 816.114641
 1011.527468
 …
 2425.655753
 2512.998653
 2595.774989
 2674.2916
 2748.822292
```

And the floor with a similar method.

[154]:

```
let d_1_F = ((F.clone() / R_F.clone()
    - lit(1.0))
.log1p()
    + sigma.clone().pow(2) * T.clone()
        / lit(2.0))
    / (sigma.clone() * T.clone().sqrt());

let d_2_F = d_1_F.clone()
    - sigma.clone() * T.clone().sqrt();

let floor = P.clone()
    * tenor.clone()
    * L.clone()
    * (R_F.clone() * N(-d_2_F.clone())
        - F.clone() * N(-d_1_F.clone()));

df.clone()
```

```
    .lazy()
    .select([floor.clone()])
    .collect()?
```

[154]:
```
shape: (20, 1)

 P
 ---
 f64

 15.921184
 104.098043
 227.413657
 359.804346
 491.866816
 …
 1619.972645
 1696.113706
 1768.83483
 1838.319023
 1904.735197
```

The collar is defined by the long cap and the short floor. **This is the answer to Problem 3**.

[155]:
```
let V = (cap.clone() - floor.clone())
    .sum()
    .alias("Value of collar");

df.clone()
    .lazy()
    .select([V.clone()])
    .collect()?
```

[155]:
```
shape: (1, 1)

 Value of collar
 ---
 f64

 12868.804054
```

5 Problem 4

Initialize data.

[156]:
```
let df = df!(
    "T" => [1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0],
    "T*" => [7.0; 7],
    "L" => [5_000_000.0; 7],
    "R_F" => [0.038; 7],
    // these are the same because the market variable is the interest rate
    "sigma_R" => [0.10; 7],
    "sigma_V" => [0.10; 7],
    // assume perfect correlation
    "rho" => [1.0; 7],
    "m" => [1.0; 7],
)?;
df
```

[156]:
```
shape: (7, 8)

  T     T*    L     R_F     sigma_R   sigma_V   rho   m
  ---   ---   ---   ---     ---       ---       ---   ---
  f64   f64   f64   f64     f64       f64       f64   f64

  1.0   7.0   5e6   0.038   0.1       0.1       1.0   1.0
  2.0   7.0   5e6   0.038   0.1       0.1       1.0   1.0
  3.0   7.0   5e6   0.038   0.1       0.1       1.0   1.0
  4.0   7.0   5e6   0.038   0.1       0.1       1.0   1.0
  5.0   7.0   5e6   0.038   0.1       0.1       1.0   1.0
  6.0   7.0   5e6   0.038   0.1       0.1       1.0   1.0
  7.0   7.0   5e6   0.038   0.1       0.1       1.0   1.0
```

Declare expressions for each column.

[157]:
```
let T = col("T");
let Tstar = col("T*");
let L = col("L");
let R_F = col("R_F");
// these are the same because the market variable is the interest rate
let sigma_R = col("sigma_R");
let sigma_V = col("sigma_V");
// assume perfect correlation
let rho = col("rho");
// annual compounding
let m = col("m");
```

We are mostly working with τ, so we will express it.

[158]:
```
let tau = Tstar.clone() - T.clone();

df.clone()
```

```
    .lazy()
    .select([tau.clone()])
    .collect()?
```

[158]: shape: (7, 1)

```
 T*
 ---
 f64

 6.0
 5.0
 4.0
 3.0
 2.0
 1.0
 0.0
```

The timing adjustment is defined as α_V here.

[159]:
```
let alpha_V = (-(rho.clone()
    * sigma_V.clone()
    * sigma_R.clone()
    * R_F.clone()
    * tau.clone())
    / (lit(1.0) + R_F.clone() / m.clone()))
.alias("alpha_V");

df.clone()
    .lazy()
    .select([alpha_V.clone()])
    .collect()?
```

[159]: shape: (7, 1)

```
 alpha_V
 ---
 f64

 -0.002197
 -0.00183
 -0.001464
 -0.001098
 -0.000732
 -0.000366
 -0.0
```

And here is each adjusted rate.

[160]:
```
let R_adjusted = (R_F.clone()
    * (alpha_V.clone() * T.clone()).exp())
.alias("R adjusted");

df.clone()
    .lazy()
    .select([R_adjusted.clone()])
    .collect()?
```

[160]:
```
shape: (7, 1)

  R adjusted
  ---
  f64

  0.037917
  0.037861
  0.037833
  0.037833
  0.037861
  0.037917
  0.038
```

The discount factor $P(0, t)$.

[161]:
```
let P = (lit(1.0)
    / (lit(1.0) + R_F.clone())
        .pow(Tstar.clone()))
.alias("P");

df.clone()
    .lazy()
    .select([P.clone()])
    .collect()?
```

[161]:
```
shape: (7, 1)

  P
  ---
  f64

  0.770227
  0.770227
  0.770227
  0.770227
```

```
0.770227
0.770227
0.770227
```

The payoff is defined as the average of the rates on the principal, after the timing adjustment. **This is the answer to Problem 4**.

[162]:
```
let V = (L.clone()
    * P.clone()
    * R_adjusted.clone().mean())
.mean()
.alias("Value after timing adjustment");

df.clone()
    .lazy()
    .select([V.clone()])
    .collect()?
```

[162]:
```
shape: (1, 1)

  Value after timing adjustment
  ---
  f64

  145915.245448
```

6 Problem 5

Initialize data.

[163]:
```
let df = df!(
    // yield curve is assumed flat
    "r" => [0.038],
    "m" => [E],
    // time of payoff
    "T" => [4.0],
    // time of first forward rate
    "T_1" => [4.0 + 2.0],
    // time of second forward rate
    "T_2" => [4.0 + 6.0],
    "L" => [5_000_000.0],
    "sigma" => [0.15],
)?;

df
```

[163]: shape: (1, 7)

r	m	T	T_1	T_2	L	sigma
---	---	---	---	---	---	---
f64	f64	f64	f64	f64	f64	f64
0.038	2.718282	4.0	6.0	10.0	5e6	0.15

Declare expressions for each column.

[164]:
```
let r = col("r");
let m = col("m");
let T = col("T");
let T_1 = col("T_1");
let T_2 = col("T_2");
let L = col("L");
let sigma = col("sigma");
```

We have two times in the future we're concerned with, which I've labeled T_1 and T_2. We will also want to refer to the τ of both.

[165]:
```
let tau_1 =
    (T_1.clone() - T.clone()).alias("tau_1");

let tau_2 =
    (T_2.clone() - T.clone()).alias("tau_2");

df.clone()
    .lazy()
    .select([tau_1.clone(), tau_2.clone()])
    .collect()?
```

[165]: shape: (1, 2)

tau_1	tau_2
---	---
f64	f64
2.0	6.0

Conditional discount factor $P(0, t)$.

[166]:
```
let P = when(m.clone().eq(E))
    .then(
        // use continuous compounding
        (-r.clone() * T.clone()).exp(),
    )
```

```
    .otherwise(
        // else discrete compounding
        lit(1.0)
            / (lit(1.0)
                + r.clone() * m.clone())
            .pow(T.clone() / m.clone()),
    );

df.clone()
    .lazy()
    .select([P.clone()])
    .collect()?
```

[166]:

```
shape: (1, 1)

 r
 ---
 f64

 0.858988
```

The convexity adjusted rate for each is given by the following:

$$\frac{G''(y)}{G'(y)} = -\tau$$

[167]:

```
let convexity_1 = (r.clone()
    - r.clone().pow(2)
        * sigma.clone().pow(2)
        * T.clone()
        * (-tau_1.clone())
        / lit(2.0))
.alias("convexity_1");

let convexity_2 = (r.clone()
    - r.clone().pow(2)
        * sigma.clone().pow(2)
        * T.clone()
        * (-tau_2.clone())
        / lit(2.0))
.alias("convexity_2");

df.clone()
    .lazy()
    .select([
        convexity_1.clone(),
```

```
        convexity_2.clone(),
    ])
    .collect()?
```

[167]:

```
shape: (1, 2)

convexity_1   convexity_2
---           ---
f64           f64

0.03813       0.03839
```

Then the payoff is the principal applied to the difference in the convexity adjusted forward rates. **This is the answer to Problem 5**.

[168]:

```
let V = (P.clone()
    * L.clone()
    * (convexity_2.clone()
        - convexity_1.clone()))
.alias("Value after convexity adjustment");

df.clone()
    .lazy()
    .select([V.clone()])
    .collect()?
```

[168]:

```
shape: (1, 1)

Value after convexity adjustme…
---
f64

1116.34117
```