Imports

In [31]:

```
:dep polars = { version = "0.46.0", features = ["full"] }
:dep chrono
:dep statrs
```

In [33]:

```
use polars::prelude::*;
use polars::df;
use statrs::distribution::{ContinuousCDF, Normal};
use std::f64::consts::E;
use std::path::PathBuf;
```

Get Data from CSV

I used Python to download Yahoo Finance data, because the Rust libraries to do this are... not great. I dumped them into CSVs so I could access them from Rust.

In [57]:

```
let df_hist = CsvReadOptions::default()
    .with_has_header(true)
    .try_into_reader_with_file_path(Some(PathBuf::from("historical_data.csv"
    .finish()?;
let df_opt = CsvReadOptions::default()
    .with_has_header(true)
    .try_into_reader_with_file_path(Some(PathBuf::from("options_data.csv")))
    .finish()?
    .lazy()
    .with_column(
        col("lastTradeDate")
            .str()
            .to_date(StrptimeOptions {
                format: Some("%Y-%m-%d %H:%M:%S%z".into()),
                strict: false,
                exact: true,
                ..Default::default()
            })
            .alias("lastTradeDate"),
    )
    .with_column(
        col("expirationDate")
            .str()
            .to_date(StrptimeOptions {
                format: Some("%Y-%m-%d".into()),
                strict: false,
                exact: true,
                ..Default::default()
            })
            .alias("expirationDate"),
    )
    .with_column(
        (col("expirationDate").dt().date() - col("lastTradeDate").dt().date(
            .alias("timeToMaturity"),
    )
    .collect()?;
df_hist.head(None)
```

Out[57]:

```
shape: (10, 4)
┌───────────────────────────┬────────────┬────────────┬───────┐
│ Datetime                  ┆ NVDA       ┆ SPY        ┆ ^VIX  │
│ ---                       ┆ ---        ┆ ---        ┆ ---   │
│ str                       ┆ f64        ┆ f64        ┆ f64   │
╞═══════════════════════════╪════════════╪════════════╪═══════╡
│ 2025-02-13 14:31:00+00:00 ┆ 132.070007 ┆ 604.789978 ┆ 15.65 │
│ 2025-02-13 14:32:00+00:00 ┆ 132.439896 ┆ 604.825012 ┆ 15.66 │
│ 2025-02-13 14:34:00+00:00 ┆ 133.059296 ┆ 604.554993 ┆ 15.67 │
│ 2025-02-13 14:35:00+00:00 ┆ 132.990005 ┆ 604.52002  ┆ 15.64 │
│ 2025-02-13 14:36:00+00:00 ┆ 132.977905 ┆ 604.469971 ┆ 15.67 │
│ 2025-02-13 14:37:00+00:00 ┆ 132.729996 ┆ 604.049988 ┆ 15.73 │
│ 2025-02-13 14:38:00+00:00 ┆ 132.069901 ┆ 603.304993 ┆ 15.9  │
│ 2025-02-13 14:39:00+00:00 ┆ 132.265793 ┆ 603.75     ┆ 15.88 │
│ 2025-02-13 14:40:00+00:00 ┆ 132.550003 ┆ 603.849976 ┆ 15.87 │
│ 2025-02-13 14:41:00+00:00 ┆ 132.289993 ┆ 603.679993 ┆ 15.87 │
└───────────────────────────┴────────────┴────────────┴───────┘
```

In [48]:

```
df_opt.head(None)
```

Out[48]:

```
shape: (10, 19)
┌─────┬──────────────┬──────────────┬────────┬───┬────────────┬─────────────
─┬────────┬─────────────┐
│     ┆ contractSymb ┆ lastTradeDat ┆ strike ┆ … ┆ optionType ┆ expirationD
┆ ticker ┆ timeToMatur │
│ --- ┆ ol           ┆ e            ┆ ---    ┆   ┆ ---        ┆ ate
┆ ---    ┆ ity         │
│ i64 ┆ ---          ┆ ---          ┆ f64    ┆   ┆ str        ┆ ---
┆ str    ┆ ---         │
│     ┆ str          ┆ date         ┆        ┆   ┆            ┆ date
┆        ┆ duration[ms │
│     ┆              ┆              ┆        ┆   ┆            ┆
┆        ┆ ]           │
╞═════╪══════════════╪══════════════╪════════╪═══╪════════════╪═════════════
═╪════════╪═════════════╡
│ 0   ┆ NVDA250221C0 ┆ 2025-02-14   ┆ 0.5    ┆ … ┆ call       ┆ 2025-02-21
┆ NVDA   ┆ 7d          │
│     ┆ 0000500      ┆              ┆        ┆   ┆            ┆
┆        ┆             │
│ 1   ┆ NVDA250221C0 ┆ 2025-02-13   ┆ 1.0    ┆ … ┆ call       ┆ 2025-02-21
┆ NVDA   ┆ 8d          │
│     ┆ 0001000      ┆              ┆        ┆   ┆            ┆
┆        ┆             │
│ 2   ┆ NVDA250221C0 ┆ 2025-02-13   ┆ 1.5    ┆ … ┆ call       ┆ 2025-02-21
┆ NVDA   ┆ 8d          │
│     ┆ 0001500      ┆              ┆        ┆   ┆            ┆
┆        ┆             │
│ 3   ┆ NVDA250221C0 ┆ 2025-02-10   ┆ 2.0    ┆ … ┆ call       ┆ 2025-02-21
┆ NVDA   ┆ 11d         │
│     ┆ 0002000      ┆              ┆        ┆   ┆            ┆
┆        ┆             │
│ 4   ┆ NVDA250221C0 ┆ 2025-02-10   ┆ 2.5    ┆ … ┆ call       ┆ 2025-02-21
┆ NVDA   ┆ 11d         │
│     ┆ 0002500      ┆              ┆        ┆   ┆            ┆
┆        ┆             │
│ 5   ┆ NVDA250221C0 ┆ 2025-02-12   ┆ 3.0    ┆ … ┆ call       ┆ 2025-02-21
┆ NVDA   ┆ 9d          │
│     ┆ 0003000      ┆              ┆        ┆   ┆            ┆
┆        ┆             │
│ 6   ┆ NVDA250221C0 ┆ 2025-01-27   ┆ 3.5    ┆ … ┆ call       ┆ 2025-02-21
┆ NVDA   ┆ 25d         │
│     ┆ 0003500      ┆              ┆        ┆   ┆            ┆
┆        ┆             │
│ 7   ┆ NVDA250221C0 ┆ 2025-01-27   ┆ 4.0    ┆ … ┆ call       ┆ 2025-02-21
┆ NVDA   ┆ 25d         │
│     ┆ 0004000      ┆              ┆        ┆   ┆            ┆
┆        ┆             │
│ 8   ┆ NVDA250221C0 ┆ 2025-02-05   ┆ 4.5    ┆ … ┆ call       ┆ 2025-02-21
┆ NVDA   ┆ 16d         │
│     ┆ 0004500      ┆              ┆        ┆   ┆            ┆
┆        ┆             │
│ 9   ┆ NVDA250221C0 ┆ 2025-02-05   ┆ 5.0    ┆ … ┆ call       ┆ 2025-02-21
┆ NVDA   ┆ 16d         │
│     ┆ 0005000      ┆              ┆        ┆   ┆            ┆
┆        ┆             │
```

```
└─────┴──────────────┴──────────────┴────────┴───┴────────────┴────────────
─┴────────┴─────────────┘
```

5. Black-Scholes

In [49]:

```
#[derive(Clone, Copy)]
enum OptionType {
    Call,
    Put,
}

fn black_scholes(option_type: OptionType, s0: f64, σ: f64, τ: f64, k: f64, r
    let normal = Normal::new(0.0, 1.0).unwrap();

    let d1 = (s0.ln() - k.ln() + (r + 0.5 * σ.powi(2)) * τ) / (σ * τ.sqrt())
    let d2 = d1 - σ * τ.sqrt();

    match option_type {
        OptionType::Call => s0 * normal.cdf(d1) - k * E.powf(-r * τ) * norma
        OptionType::Put => k * E.powf(-r * τ) * normal.cdf(-d2) - s0 * norma
    }
}
```

6. Bisection

Bisection and secant are both implemented as methods on any function

$$f : \mathbb{R} \Rightarrow \mathbb{R}$$

In [50]:

```
pub trait Rootable {
    fn secant(&self, x0: f64, x1: f64, ε: f64, max_iter: usize) -> Option<f6
    fn bisect(&self, a: f64, b: f64, ε: f64, max_iter: usize) -> Option<f64>
}
impl<F> Rootable for F
where
    F: Fn(f64) -> f64,
{
    fn bisect(&self, mut a: f64, mut b: f64, ε: f64, max_iter: usize) -> Opt
        if (self(a) * self(b)).is_sign_positive() {
            eprintln!("f(a) and f(b) must have opposite signs");
            return None;
        }

        let mut mid;
        for _ in 0..max_iter {
            mid = (a + b) / 2.0;

            if self(mid).abs() < ε {
                return Some(mid);
            }
            if (self(mid) * self(a)).is_sign_negative() {
                b = mid;
            } else {
                a = mid;
            }
        }
        eprintln!("Maximum iterations reached without finding the root.");
        None
    }

    fn secant(&self, mut x0: f64, mut x1: f64, ε: f64, max_iter: usize) -> O
        for _ in 0..max_iter {
            let f0 = self(x0);
            let f1 = self(x1);

            if (f1 - f0).abs() < 1e-10 {
                eprintln!("Denominator too small, Secant method failed.");
                return None;
            }

            let x_new = x1 - f1 * (x1 - x0) / (f1 - f0);

            if (x_new - x1).abs() < ε {
                return Some(x_new);
            }

            x0 = x1;
            x1 = x_new;
        }

        eprintln!("Maximum iterations reached without finding the root.");
        None
    }
}
```

Constants for tolerance and interest rate given.

In [51]:

```
const TOL: f64 = 1e-5;
const R: f64 = 0.0433;
```

Had some problems formatting dates

In [52]:

```
use chrono::{NaiveDate, Duration};

fn days_to_date(days: i32) -> NaiveDate {
    let epoch = NaiveDate::from_ymd_opt(1970, 1, 1).unwrap();
    epoch + Duration::days(days.into())
}
```

Giant kludge function for implied volatility calculation. If I had finished this, I would definitely have refactored it so I could reuse parts of it for the rest of the problems.

In [53]:

```
fn process_options(df: DataFrame, s0: f64, ticker: &str) -> PolarsResult<Dat
    let df = df.lazy().filter(col("ticker").eq(lit(ticker))).collect()?;
    let expiration_dates = df.column("expirationDate")?.date()?.unique()?;
    let v: Vec<_> = expiration_dates
        .into_iter()
        .map(|expiration_date| -> PolarsResult<LazyFrame> {
            let options_for_expiry = df
                .clone()
                .lazy()
                .filter(col("expirationDate").eq(lit(expiration_date.unwrap(
                .collect()?;
            let atm_call_option = options_for_expiry
                .clone()
                .lazy()
                .filter(
                    col("inTheMoney")
                        .eq(lit(true))
                        .and(col("optionType").eq(lit("call"))),
                )
                .sort(["strike"], Default::default())
                .tail(1)
                .collect()?;
            let atm_put_option = options_for_expiry
                .clone()
                .lazy()
                .filter(
                    col("inTheMoney")
                        .eq(lit(true))
                        .and(col("optionType").eq(lit("put"))),
                )
                .sort(
                    ["strike"],
                    SortMultipleOptions::default().with_order_descending(tru
                )
                .tail(1)
                .collect()?;

            let opt_calc = |opt_type: OptionType| -> PolarsResult<f64> {
                let atm_option = match opt_type {
                    OptionType::Call => atm_call_option.clone(),
                    OptionType::Put => atm_put_option.clone(),
                };
                let k = atm_option.column("strike")?.f64()?.get(0).unwrap();
                let bid = atm_option.column("bid")?.f64()?.get(0).unwrap();
                let ask = atm_option.column("ask")?.f64()?.get(0).unwrap();

                let last_trade_date = atm_option.column("lastTradeDate")?.da
                let expiration_date = atm_option.column("expirationDate")?.d

                let market_price = (bid + ask) / 2.0;
                let tau = (expiration_date - last_trade_date) as f64 / 252.0
                let implied_volatility = |sigma: f64| {
                    let bs_price = black_scholes(opt_type, s0, sigma, tau, k
                    bs_price - market_price
                };
                let iv = implied_volatility.bisect(0.0001, 2.0, TOL, 100);

                Ok(iv.unwrap())
            };
            let calc_call_iv = opt_calc(OptionType::Call)?;
            let calc_put_iv = opt_calc(OptionType::Put)?;

            // Get all options between in the money and out of the money, us
            // Moneyness is defined here by the ratio of S0 to the strike pr
            let between_call_options = options_for_expiry
                .clone()
                .lazy()
                .filter(
                    lit(s0)
                        .gt(lit(0.95) * col("strike"))
                        .and(lit(s0).lt(lit(1.05) * col("strike")))
                        .and(col("optionType").eq(lit("call"))),
                )
                .collect()?;
            let between_put_options = options_for_expiry
                .clone()
                .lazy()
                .filter(
                    lit(s0)
                        .gt(lit(0.95) * col("strike"))
                        .and(lit(s0).lt(lit(1.05) * col("strike")))
                        .and(col("optionType").eq(lit("put"))),
                )
                .collect()?;
            // Average the implied volatilities of these options
            let avg_call_iv = between_call_options
                .column("impliedVolatility")?
                .f64()?
                .mean()
                .unwrap();
            let avg_put_iv = between_put_options
                .column("impliedVolatility")?
                .f64()?
                .mean()
                .unwrap();

            Ok(df!(
                "expirationDate" => [days_to_date(expiration_date.unwrap())]
                "ticker" => [ticker],
                "callAvgIV" => [avg_call_iv],
                "callCalcIV" => [calc_call_iv],
                "putAvgIV" => [avg_put_iv],
                "putCalcIV" => [calc_put_iv]
            )?
            .lazy())
        })
        .filter_map(Result::ok)
        .collect();
    let full = concat(v, Default::default())?
        //.with_column(col("expirationDate").dt().date().alias("expirationDa
        ;
    full.collect()
}
```

Get the latest prices for NVDA and SPY.

In [54]:

```
let data1 = df_hist
.clone()
.lazy()
.filter(
    col("Datetime")
        .str()
        .to_date(StrptimeOptions {
            format: Some("%Y-%m-%d %H:%M:%S%z".into()),
            strict: false,
            exact: true,
            ..Default::default()
        })
        .dt()
        .date()
        .eq(datetime(DatetimeArgs::new(lit(2025), lit(2), lit(13)))
            .dt()
            .date()),
)
.collect()?;
let data2 = df_hist
.lazy()
.filter(
    col("Datetime")
        .str()
        .to_date(StrptimeOptions {
            format: Some("%Y-%m-%d %H:%M:%S%z".into()),
            strict: false,
            exact: true,
            ..Default::default()
        })
        .dt()
        .date()
        .eq(datetime(DatetimeArgs::new(lit(2025), lit(2), lit(14)))
            .dt()
            .date()),
)
.collect()?;
let data1_latest = data1.sort(["Datetime"], Default::default())?.tail(Some(1
let data2_latest = data2.sort(["Datetime"], Default::default())?.tail(Some(1

let s0_nvda = data2_latest.column("NVDA")?.f64()?.get(0).unwrap();
let s0_spy = data1_latest.column("SPY")?.f64()?.get(0).unwrap();
```

In [55]:

```
process_options(df_opt, s0_nvda, "NVDA")?
```

```
Out[55]: shape: (3, 6)
┌────────────────┬────────┬──────────┬───────────┬──────────┬───────────┐
│ expirationDate ┆ ticker ┆ callAvgIV ┆ callCalcIV ┆ putAvgIV ┆ putCalcIV │
│ ---            ┆ ---    ┆ ---      ┆ ---       ┆ ---      ┆ ---       │
│ date           ┆ str    ┆ f64      ┆ f64       ┆ f64      ┆ f64       │
╞════════════════╪════════╪══════════╪═══════════╪══════════╪═══════════╡
│ 2025-02-21     ┆ NVDA   ┆ 0.513973 ┆ 0.31071   ┆ 0.513241 ┆ 0.326463  │
│ 2025-03-21     ┆ NVDA   ┆ 0.61675  ┆ 0.470043  ┆ 0.58582  ┆ 0.484683  │
│ 2025-04-17     ┆ NVDA   ┆ 0.552332 ┆ 0.420348  ┆ 0.520513 ┆ 0.443469  │
└────────────────┴────────┴──────────┴───────────┴──────────┴───────────┘
```

```
In [58]: process_options(df_opt, s0_nvda, "SPY")?
```

```
f(a) and f(b) must have opposite signs
thread '<unnamed>' panicked at src/lib.rs:60:23:
called `Option::unwrap()` on a `None` value
stack backtrace:
   0: rust_begin_unwind
             at /rustc/ed04567ba1d5956d1080fb8121caa005ce059e12/library/std/
src/panicking.rs:665:5
   1: core::panicking::panic_fmt
             at /rustc/ed04567ba1d5956d1080fb8121caa005ce059e12/library/cor
e/src/panicking.rs:74:14
   2: core::panicking::panic
             at /rustc/ed04567ba1d5956d1080fb8121caa005ce059e12/library/cor
e/src/panicking.rs:148:5
   3: core::option::unwrap_failed
             at /rustc/ed04567ba1d5956d1080fb8121caa005ce059e12/library/cor
e/src/option.rs:2004:5
   4: ctx::process_options::{{closure}}::{{closure}}
   5: <core::iter::adapters::map::Map<I,F> as core::iter::traits::iterator::
Iterator>::try_fold
   6: <alloc::vec::Vec<T> as alloc::vec::spec_from_iter::SpecFromIter<T,I>
>::from_iter
   7: run_user_code_48
   8: <unknown>
   9: <unknown>
  10: <unknown>
  11: <unknown>
  12: <unknown>
  13: <unknown>
  14: <unknown>
  15: <unknown>
  16: __libc_start_main
  17: <unknown>
note: Some details are omitted, run with `RUST_BACKTRACE=full` for a verbose
backtrace.
```