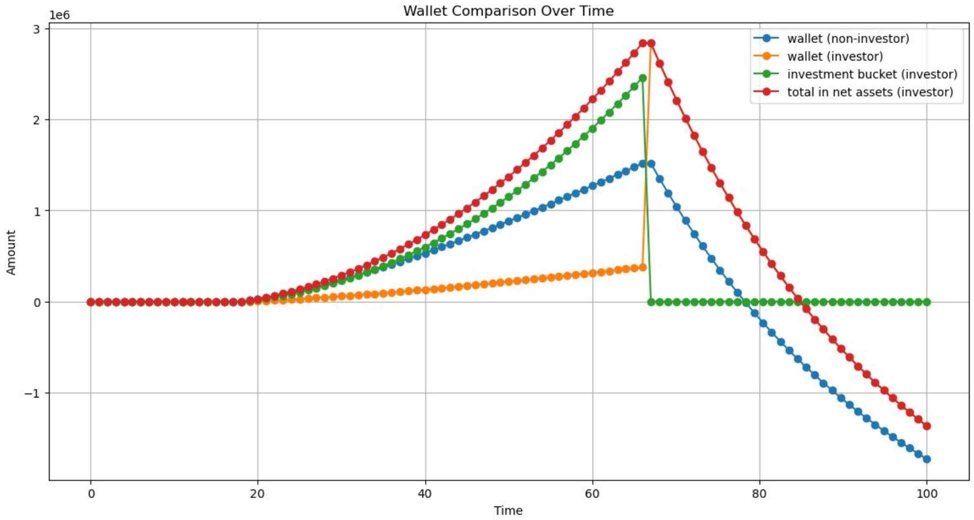
Introduction & Motivation

Members: Ruijie Jia, Pei Shi

Background: Utilize Python and the Scipy library to model **financial independence** as a result of **life cycle investment planning** using ordinary differential equations (**ODEs**).

Motivation: Employ various advanced python optimization methods based on financial independence modeling involves **enhancing the efficiency and speed** of simulations.



Methodology

Models Development

- Using a linear first-order ordinary differential equation **(ODE)** to **simulate financial growth**
- Using **numerical integration methods** to obtain a discretized approximation of wealth growth over time
- **Model Settings:**

$$\frac{dx}{dt} = \Delta(t) + x \ln\left(\frac{1+\beta R}{1+\xi}\right)$$

where:

- $x(t)$ represents the total wealth at time t,
- $\Delta(t)$ is the net yearly balance after all incomes, expenses, and taxes,
- R is the expected average interest rate on investments,
- ξ is the average yearly inflation rate,
- β is the fraction of total wealth invested, referred to as the commitment factor.

Performance Optimization

- **Advanced Python Techniques**
  - Data Structure Modification
  - Function Call Overhead Reduction
- **Cython Optimization**
  - Minimize the overhead associated with Python's dynamic nature by compiling it to C
- **Numba Optimization**
  - Numba **jit & njit** decorators
- **Mpi optimization for Monte Carlo Simulation**
  - Introduce **MPI4PY** to optimize the simulation by virtue of the parallel computing.

Initial Code & Python Optimization

Initial Code

- Introduces a **Python class** “Life” simulating financial trajectories with or without investment over a person's lifetime. Simulates financial outcomes with functions.
- Initial time for Simulating 1000 times: **12.04s**

```python
import numpy as np
import pandas as pd
from scipy.integrate import odeint
import matplotlib.pyplot as plt

class Life_6:
    def __init__(self, investment_fraction, interest_rate_proc=5, income=10000, spending=7000, tax_rate=0.19,
                 pension=4000, starting_age=18, retirement_age=67, pay_raise=250, life_inflation=50,
                 inflation_proc=4):
        self.investment_fraction = investment_fraction  # beta
        self.interest_rate_proc = interest_rate_proc  # 5%
        self.income = income  # gold pcs. per month
        self.spending = spending  # gold pcs. per month
        self.tax_rate = tax_rate  # example
        self.pension = pension
        self.starting_age = starting_age
        self.retirement_age = retirement_age
        self.pay_raise = pay_raise
        self.life_inflation = life_inflation
        self.inflation_proc = inflation_proc

    def earn(self, t):
        if t < self.starting_age:
            return 0
        elif self.starting_age <= t < self.retirement_age:
            return 12 * (self.income + self.pay_raise \
                    * (t - self.starting_age))
        else:
            return 12 * self.pension

    def spend(self, t):
        return 12 * (self.spending + self.life_inflation \
                * (t - self.starting_age))

    def pay_taxes(self, t):
        return self.earn(t) * self.tax_rate
```

```python
def live_with_investing(x, t, you):
    balance = you.earn(t) - you.spend(t) - you.pay_taxes(t)
    x1 = balance * (1 - you.investment_fraction) - np.log(1 + 0.01 * you.inflation_proc) * x[0]
    if t < you.retirement_age:
        x2 = np.log(1 + 0.01 * you.interest_rate_proc) * x[1] + you.investment_fraction * balance
    else:
        x2 = np.log(1 + 0.01 * you.interest_rate_proc) * x[1]  # Continue to grow the investment bucket
    x2 -= np.log(1 + 0.01 * you.inflation_proc) * x[1]

    return [x1, x2]

def live_without_investing(x, t, you):
    balance = you.earn(t) - you.spend(t) - you.pay_taxes(t)
    return balance - np.log(1 + 0.01*you.inflation_proc) * x

def simulate(you):
    ...  # t0, t1, t2 - as before
    t0 = np.linspace(0, you.starting_age - 1, num=you.starting_age)
    t1 = np.linspace(you.starting_age, you.retirement_age - 1, num=(you.retirement_age - you.starting_age))
    t2 = np.linspace(you.retirement_age, 100, num=(100 - you.retirement_age))

    # non-investor
    x1_0 = np.zeros((t0.shape[0], 1))
    x1_1 = odeint(live_without_investing, 0, t1, args=(you,))
    x1_2 = odeint(live_without_investing, x1_1[-1], t2, args=(you,))

    # investor
    x2_0 = np.zeros((t0.shape[0], 2))
    x2_1 = odeint(live_with_investing, [0, 0], t1, args=(you,))
    x2_2 = odeint(live_with_investing, x2_1[-1], t2, args=(you,))


    df0 = pd.DataFrame({'time': t0, 'wallet (non-investor)': x1_0[:, 0],
                        'wallet (investor)': x2_0[:, 0], 'investment bucket (investor)': x2_0[:, 1]})
    df1 = pd.DataFrame({'time': t1, 'wallet (non-investor)': x1_1[:, 0],
                        'wallet (investor)': x2_1[:, 0], 'investment bucket (investor)': x2_1[:, 1]})
    df2 = pd.DataFrame({'time': t2, 'wallet (non-investor)': x1_2[:, 0],
                        'wallet (investor)': x2_2[:, 0], 'investment bucket (investor)': x2_2[:, 1]})
    return pd.concat([df0, df1, df2])
```

Initial Code & Python Optimization

Python Optimization (1): Data Structure Modification

- **Remove the class-based structure** and instead use standalone functions with explicit parameters for all variables. Function calls can be slightly faster than method calls.
- Optimized time for Simulating 1000 times: **4.95s**

```
import numpy as np
import pandas as pd
from scipy.integrate import odeint
import matplotlib.pyplot as plt

def earn(t, starting_age, retirement_age, income, pay_raise, pension):
    if t < starting_age:
        return 0
    elif starting_age <= t < retirement_age:
        return 12 * (income + pay_raise * (t - starting_age))
    else:
        return 12 * pension

def spend(t, starting_age, spending, life_inflation):
    return 12 * (spending + life_inflation * (t - starting_age))

def pay_taxes(t, tax_rate, starting_age, retirement_age, income, pay_raise, pension):
    earned = earn(t, starting_age, retirement_age, income, pay_raise, pension)
    return earned * tax_rate

def live_with_investing(x, t, investment_fraction, interest_rate_proc, inflation_proc, starting_age, retirement_age,
    balance = earn(t, starting_age, retirement_age, income, pay_raise, pension) - spend(t, starting_age, spending, l
    x1 = balance * (1 - investment_fraction) - np.log(1 + 0.01 * inflation_proc) * x[0]
    if t < retirement_age:
        x2 = np.log(1 + 0.01 * interest_rate_proc) * x[1] + investment_fraction * balance
    else:
        x2 = np.log(1 + 0.01 * interest_rate_proc) * x[1]  # Continue to grow the investment bucket
    x2 -= np.log(1 + 0.01 * inflation_proc) * x[1]

    return [x1, x2]

def live_without_investing(x, t, inflation_proc, starting_age, retirement_age, income, pay_raise, pension, spending,
    balance = earn(t, starting_age, retirement_age, income, pay_raise, pension) - spend(t, starting_age, spending, l
    return balance - np.log(1 + 0.01 * inflation_proc) * x
```

```
def simulate(investment_fraction, interest_rate_proc, inflation_proc, starting_age, retirement_age, income, pay_rais
    t0 = np.linspace(0, starting_age - 1, num=starting_age)
    t1 = np.linspace(starting_age, retirement_age - 1, num=(retirement_age - starting_age))
    t2 = np.linspace(retirement_age, 100, num=(100 - retirement_age))

    # non-investor
    x1_0 = np.zeros((t0.shape[0], 1))
    x1_1 = odeint(live_without_investing, 0, t1, args=(inflation_proc, starting_age, retirement_age, income, pay_rai
    x1_2 = odeint(live_without_investing, x1_1[-1], t2, args=(inflation_proc, starting_age, retirement_age, income,

    # investor
    x2_0 = np.zeros((t0.shape[0], 2))
    x2_1 = odeint(live_with_investing, [0, 0], t1, args=(investment_fraction, interest_rate_proc, inflation_proc, st
    x2_2 = odeint(live_with_investing, x2_1[-1], t2, args=(investment_fraction, interest_rate_proc, inflation_proc,

    df0 = pd.DataFrame({'time': t0, 'wallet (non-investor)': x1_0[:, 0], 'wallet (investor)': x2_0[:, 0], 'investmen
    df1 = pd.DataFrame({'time': t1, 'wallet (non-investor)': x1_1[:, 0], 'wallet (investor)': x2_1[:, 0], 'investmen
    df2 = pd.DataFrame({'time': t2, 'wallet (non-investor)': x1_2[:, 0], 'wallet (investor)': x2_2[:, 0], 'investmen
    return pd.concat([df0, df1, df2])
```

Initial Code & Python Optimization

Python Optimization (2): Function Call Overhead Reduction

- **Integrate subsidiary functions** like earn, spend, and pay_taxes directly into the primary simulation functions *at a cost of modularity*.
- Optimized time for Simulating 1000 times: **4.31s**

```python
def live_with_investing(x, t, investment_fraction, interest_rate_proc, inflation_proc, starting_age, retirement_age,

    # calculate earning
    if t < starting_age:
        earned = 0
    elif starting_age <= t < retirement_age:
        earned = 12 * (income + pay_raise * (t - starting_age))
    else:
        earned = 12 * pension

    balance = (1 - tax_rate) * earned - 12 * (spending + life_inflation * (t - starting_age))

    x1 = balance * (1 - investment_fraction) - np.log(1 + 0.01 * inflation_proc) * x[0]
    if t < retirement_age:
        x2 = np.log(1 + 0.01 * interest_rate_proc) * x[1] + investment_fraction * balance
    else:
        x2 = np.log(1 + 0.01 * interest_rate_proc) * x[1]  # Continue to grow the investment bucket
    x2 -= np.log(1 + 0.01 * inflation_proc) * x[1]

    return [x1, x2]

def live_without_investing(x, t, inflation_proc, starting_age, retirement_age, income, pay_raise, pension, spending,

    # calculate earning
    if t < starting_age:
        earned = 0
    elif starting_age <= t < retirement_age:
        earned = 12 * (income + pay_raise * (t - starting_age))
    else:
        earned = 12 * pension

    balance = (1 - tax_rate) * earned - 12 * (spending + life_inflation * (t - starting_age))
    return balance - np.log(1 + 0.01 * inflation_proc) * x
```

Cython Optimization

Cython Modifications

- **Static Type Definitions:** Declare variables with C data types by cdef
- **Using C Functions:** Replace Python's math.log with libc.math cimport log
- **Efficient Array Operations:** Use typed memory views for function parameters
- Optimized time for Simulating 1000 times: **3.93s**

```
%%cython
import numpy as np
cimport numpy as np
from libc.math cimport log
from scipy.integrate import odeint
import pandas as pd
import time

cdef double live_without_investing(double x, double t, double inflation_proc, int starting_age, int retirement_age,
    cdef double earned, balance
    if t < starting_age:
        earned = 0
    elif starting_age <= t < retirement_age:
        earned = 12 * (income + pay_raise * (t - starting_age))
    else:
        earned = 12 * pension

    balance = (1 - tax_rate) * earned - 12 * (spending + life_inflation * (t - starting_age))
    return balance - log(1 + 0.01 * inflation_proc) * x

def live_with_investing(double[:] x, double t, double investment_fraction, double interest_rate_proc, double inflati
    cdef double earned, balance
    cdef double x1, x2
    if t < starting_age:
        earned = 0
    elif starting_age <= t < retirement_age:
        earned = 12 * (income + pay_raise * (t - starting_age))
    else:
        earned = 12 * pension

    balance = (1 - tax_rate) * earned - 12 * (spending + life_inflation * (t - starting_age))

    x1 = balance * (1 - investment_fraction) - log(1 + 0.01 * inflation_proc) * x[0]
    x2 = log(1 + 0.01 * interest_rate_proc) * x[1] + investment_fraction * balance
    x2 -= log(1 + 0.01 * inflation_proc) * x[1]

    return np.array([x1, x2], dtype=np.float64)
```

```
def simulate_cython(double investment_fraction, double interest_rate_proc, double inflation_proc, int starting_age,
    cdef int num_years1 = retirement_age - starting_age
    cdef int num_years2 = 100 - retirement_age
    cdef np.ndarray[np.float64_t, ndim=1] t0 = np.linspace(0, starting_age - 1, num=starting_age, dtype=np.float64)
    cdef np.ndarray[np.float64_t, ndim=1] t1 = np.linspace(starting_age, retirement_age - 1, num=num_years1, dtype=n
    cdef np.ndarray[np.float64_t, ndim=1] t2 = np.linspace(retirement_age, 100, num=num_years2, dtype=np.float64)

    # non-investor
    cdef np.ndarray[np.float64_t, ndim=1] x1_0 = np.zeros(t0.shape[0], dtype=np.float64)
    cdef np.ndarray x1_1 = odeint(live_without_investing, 0, t1, args=(inflation_proc, starting_age, retirement_age,
    cdef np.ndarray x1_2 = odeint(live_without_investing, x1_1[-1], t2, args=(inflation_proc, starting_age, retireme

    # investor
    cdef np.ndarray[np.float64_t, ndim=2] x2_0 = np.zeros((t0.shape[0], 2), dtype=np.float64)
    cdef np.ndarray x2_1 = odeint(live_with_investing, np.array([0, 0], dtype=np.float64), t1, args=(investment_frac
    cdef np.ndarray x2_2 = odeint(live_with_investing, x2_1[-1], t2, args=(investment_fraction, interest_rate_proc,

    df0 = pd.DataFrame({'time': t0, 'wallet (non-investor)': x1_0, 'wallet (investor)': x2_0[:, 0], 'investment buck
    df1 = pd.DataFrame({'time': t1, 'wallet (non-investor)': x1_1[:, 0], 'wallet (investor)': x2_1[:, 0], 'investmen
    df2 = pd.DataFrame({'time': t2, 'wallet (non-investor)': x1_2[:, 0], 'wallet (investor)': x2_2[:, 0], 'investmen
    return pd.concat([df0, df1, df2])

cdef double interest_rate_proc = 5  # 5%
cdef double income = 10000  # gold pcs. per month
cdef double spending = 7000  # gold pcs. per month
cdef double tax_rate = 0.19  # example
cdef double pension = 4000
cdef int starting_age = 18
cdef int retirement_age = 67
cdef double pay_raise = 250
cdef double life_inflation = 50
cdef double inflation_proc = 4
```

Numba Optimization

Numba jit&njit Decorators

- Use Numba to generate optimized machine code from Python code using the LLVM compiler infrastructure. NJIT for balance calculation functions and JIT for simulate.
- Optimized time for Simulating 1000 times: **3.58s**

```
@njit
def live_with_investing(x, t, investment_fraction, interest_rate_proc, inflation_proc, starting_age, retirement_age,
```

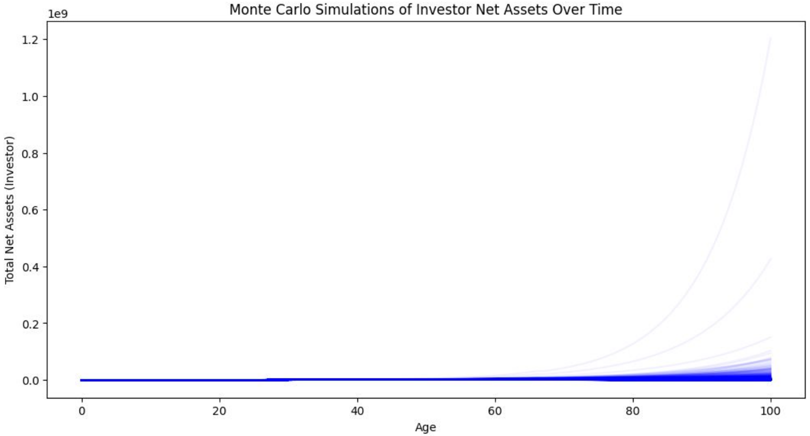
```
@njit
def live_without_investing(x, t, inflation_proc, starting_age, retirement_age, income, pay_raise, pension, spending,
```

```
@jit
def simulate(investment_fraction, interest_rate_proc, inflation_proc, starting_age, retirement_age, income, pay_rais
```

Mpi optimization for Monte Carlo Simulation

MC Simulation for Random Interest Rate, Inflation Rate, and Investment Fraction

- **Real World Life** can be very random
  - Investment fraction follows a beta distribution
  - Interest Rate and Inflation Rate follow normal distributions
- Set distribution parameters and conduct **MC simulations** to check the lifetime fortune with randomized parameters
- Optimized time for Simulating 2000 times: **4.72s**



Mpi optimization for Monte Carlo Simulation

Mpi Parallelization

- Introduce **MPI4PY** to optimize the simulation by parallel computing
- Distribute partial of Monte Carlo iterations to each MPI processes
- After all processes complete their simulations, the results are **gathered to the root process** using MPI's comm.gather
- Optimized time for Simulating 2000 times: **0.90s**

```python
from mpi4py import MPI
import numpy as np
import pandas as pd
from scipy.integrate import odeint
import matplotlib.pyplot as plt
import time

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

......

if rank == 0:
    start_time = time.time()

MC_times = 2000
local_MC_times = MC_times // size
```

```python
dfs = {}
for _ in range(local_MC_times):
    beta = np.random.beta(2, 5)
    interest_rate_proc = sig_itr * np.random.randn() + mu_itr
    inflation_proc = sig_inf * np.random.randn() + mu_inf

    instance = Life_real(investment_fraction=beta, interest_rate_proc=interest_rate_proc, inflation_proc=inflation_p
    tmp = simulate(instance)
    tmp['total net asssets (investor)'] = tmp['wallet (investor)'] + tmp['investment bucket (investor)']
    tmp = tmp.drop(columns=['wallet (non-investor)', 'wallet (investor)', 'investment bucket (investor)'])
    dfs[instance.investment_fraction] = tmp

all_dfs = comm.gather(dfs, root=0)

if rank == 0:
    combined_dfs = {}
    for d in all_dfs:
        combined_dfs.update(d)
    end_time = time.time()
    print(end_time - start_time, "s")
```

Result & Discussions

Optimization Results

Optimization Step	Execution Time (s)
Initial Code	12.04
Data Structure Modification	4.95
Function Call Overhead Reduction	4.31
Cython Implementation	3.93
Numba Implementation	3.58

Table 1: Performance Improvements in Financial Model Simulation

Monte Carlo Method	Average Execution Time (s)
Without MPI	4.72
With MPI (8 CPUs)	0.90

Table 2: Monte Carlo Simulation Performance with and without MPI

Discussions & Future Works

- We optimized and extended a basic financial model simulation to an **optimized, parallelized** Monte Carlo simulation
- The optimization techniques significantly reduced the total execution time
- **Future Works** could focus on the optimization problem under uncertain financial conditions, e.g. looking for the optimal investment fraction given some inflation rate & interest rate