

# Дипломна работа на тема

Разработване на виртуална среда за актюерски  
пенсионни изчисления

Георги Веселинов Бурнаски  
Катедра Магистърска програма- Вероятности актюерство и статистика  
Факултет по Математика и Информатика  
Софийски Университет "Св. Климент Охридски"

2025-09-29

# 1 Увод

Актюерската математика заема все по-значимо място в съвременното общество, където управлението на финансовите рискове и дългосрочната устойчивост на социалните системи са от първостепенно значение.

В условията на застаряващо население, динамични финансови пазари и нарастващи изисквания към пенсионните системи, надеждните и прецизни актюерски изчисления се превръщат в ключов инструмент за вземане на информирани решения. Те подпомагат както държавните институции при моделиране и реформиране на пенсионните схеми, така и частните компании при разработване на застрахователни и инвестиционни продукти [1].

Актюерските методи намират широко приложение в различни сфери: пенсионното осигуряване, животозастраховането здравното застраховане, управлението на фондове и финансовото планиране. Общото между всички тези области е необходимостта от точни прогнози, основани на статистически модели и вероятностни методи, които да оценяват бъдещи парични потоци, продължителност на живота и свързаните с тях рискове [2].

В наши дни пенсионните системи се изправят пред редица сериозни предизвикателства. Демографските промени, и по-специално увеличаването на средната продължителност на живота и намаляването на раждаемостта, водят до нарастващо съотношение между пенсионери и активно работещо население [3]. Това поставя под натиск публичните пенсионни фондове и създава необходимост от по-прецизни модели за оценка на бъдещите задължения. Паралелно с това колебанията на финансовите пазари и инфлационните процеси влияят върху доходността на пенсионните активи и изискват по-гъвкави подходи при управлението на риска [4].

Съвременните технологии и програмни езици като Python предлагат нови възможности за прилагане на актюерската математика в практиката. Разработването на специализирана библиотека за пенсионни актюерски изчисления има двойна значимост: от една страна, улеснява изследователите и практиците при прилагането на сложни модели, а от друга – допринася за повишаване на прозрачността, възпроизводимостта и достъпността на тези изчисления [5, 6].

Въпреки наличието на различни софтуерни решения за актюерски изчисления, много от тях са или твърде специализирани, или не предлагат необходимата гъвкавост и интеграция с други аналитични инструменти. Python, с богатия си екосистем от библиотеки за научни изчисления и машинно обучение, се явява като идеална платформа за разработване на такава библиотека. Тя би могла да обедини теоретичните основи

на актюерската математика с практическите нужди на потребителите, предоставяйки лесен за използване и разширяем инструмент [12, 13].

Разработването на библиотека за пенсионни актюерски изчисления в Python би могло да включва функции за изчисляване на настояща стойност на бъдещи парични потоци, моделиране на демографски и финансови рискове, симулации на сценарии и оптимизация на пенсионни стратегии. Освен това, интеграцията с други библиотеки за визуализация и анализ би позволила по-добро представяне и интерпретация на резултатите [14, 15].

В заключение, създаването на специализирана библиотека за пенсионни актюерски изчисления в Python представлява важна стъпка към модернизацията и усъвършенстването на актюерската практика. Тя би могла да подпомогне както академичните изследвания, така и практическите приложения, като предостави мощен и достъпен инструмент за анализ и управление на пенсионните системи в съвременния свят.

Пенсионната система в България следва модела на много развити страни и се състои от три компонента, известни като "трите стълба":

- Първи стълб: Държавно задължително пенсионно осигуряване (солидарност между поколенията). Това е държавната пенсия, която се финансира от текущите осигурителни вноски на работещите. Тя е задължителна за всички работещи.
- Втори стълб: Допълнително задължително пенсионно осигуряване (ДЗПО). Това е индивидуално натрупване на средства в избрани от самия осигурено лице пенсионни фондове. Той също е задължителен за хората, родени след 31.12.1959 г., които не са избрали да останат само в Първи стълб.
- Трети стълб: Допълнително доброволно пенсионно осигуряване (ДДПО). Това е напълно доброволно допълнително пенсионно осигуряване, при което хората сами решават да спестяват допълнително за своята пенсия.

В този труд ще се фокусираме върху изчисленията, свързани със втори и трети стълб от пенсионноосигурителната система, тъй като те са поети от частните пенсионноосигурителни дружества. В България 20% от доходите на работещите се отделят за пенсионно осигуряване, като 12.8% отиват за първи стълб, а 5% - за втори стълб. Важно е да се отбележи, че тези проценти могат да варират в зависимост от законодателството и икономическите условия. Останалите 2.2% са за здравно осигуряване и трудови злополуки.[8, 9]

Условията за пенсионните изчисления са регламентирани от различни нормативни актове, като основните от тях са:

- Кодекс за социално осигуряване (КСО)
- Закон за допълнително задължително пенсионно осигуряване (ЗДЗПО)
- Закон за допълнително доброволно пенсионно осигуряване (ЗДОО)
- Наредба № Н-8 от 2004 г. за определяне на методиката за изчисляване на пенсиите
- Наредба № 3 от 2003 г. за условията и реда за изплащане на пенсиите от частните пенсионни фондове

Като изчисленията трябва да отговарят на изискванията, заложи наредба 69 от 2003 г. за определяне на методиката за изчисляване на пенсиите от частните пенсионни фондове. [8, 9, 10, 11]

Целта на настоящата дисертация е именно изграждането на такава библиотека, която да обедини теоретичните основи на пенсионната математика с предимствата на съвременните програмни среди. Чрез нея се цели създаването на гъвкав инструмент, който да подкрепя както научната работа, така и практическите решения в областта на пенсионното осигуряване.

## 2 Математика

### 2.1 Вероятност за смърт

Вероятността за смърт  $q_x$  е вероятността човек на възраст  $x$  да умре преди да достигне възраст  $x+1$ . Тази вероятност се взема от статистическата статистика на държавната осигурителна институция в случая Националният осигурителен институт(НОИ)

### 2.2 Вероятност за преживяване

Вероятността за преживяване  $p_x = 1 - q_x$  е вероятността човек на възраст  $x$  да оцелее до следващата година, т.е. до възраст  $x + 1$ .

### 2.3 Среден брой живи хора

Средният брой живи хора на възраст  $x$  се обозначава с  $l_x$  и представлява броя на хората, които са живи на тази възраст в дадена популация. Тази стойност се използва за изчисляване на вероятността за смърт и други актюерски изчисления. Изчислява се по формулата:

$$l_x = l_0 \cdot \prod_{i=0}^{x-1} p_i$$

където  $l_0$  е началният брой живи хора (на възраст 0), а  $p_i$  е вероятността за преживяване на възраст  $i$  и  $l_x$  е броят на живите хора на възраст  $x$ .

### 2.4 Дисконтирац фактор

Дисконтиращият фактор  $v$  се използва за изчисляване на настоящата стойност на бъдещи парични потоци. Той се изчислява по формулата:

$$v = \frac{1}{1 + i}$$

където  $i$  е годишният лихвен процент зададен от пенсионноосигурителната компания. В нашият случай ще използваме техническия лихвен процент на пенсионна компания "Съгласие" който е 0.15% [10]

### 2.5 Настояща стойност

Настоящата стойност ( $D_x$ ) на бъдещ паричен поток се изчислява чрез дисконтиране на този поток към настоящия момент. Формулата за изчисляване

на настоящата стойност е:

$$D_x = l_x \cdot v^x$$

## 2.6 Кумулативна настояща стойност

Кумулативната настояща стойност ( $N_x$ ) представлява сумата от настоящите стойности на всички бъдещи плащания за период от  $n$  години. Тя се изчислява по формулата:

$$N_x = \sum_{i=x}^{max} D_x$$

## 2.7 Актюерски фактори и изчисления на пенсии

Актюерските фактори се използват за изчисляване на различни видове пенсии и други финансови продукти. Те включват:

- **Проста пожизнена пенсия (фактор  $k_1$ )** - изчислява се като:

$$k_1 = 12 \cdot \left( \frac{N_x}{D_x} - \frac{11}{24} \right)$$

където  $N_x$  е кумулативната настояща стойност на бъдещите плащания, а  $D_x$  е настоящата стойност на бъдещия паричен поток. Това представлява фактор, който се използва за изчисляване на месечната сума на Пожизнена пенсия. Която се изчислява като:

$$P = \frac{S}{k_1}$$

където  $P$  е месечната сума на пенсията, а  $S$  е натрупаната сума в пенсионния фонд към момента на пенсиониране.

- **Пожизнена пенсия с период на гарантирано изплащане (фактор  $k_2$ )** - изчислява се като:

$$k_2 = 12 \cdot \left( \frac{N_{x+d}}{D_x} - \frac{11}{24} \cdot \frac{D_{x+d}}{D_x} \right) + \frac{1 - v^n}{1 - \sqrt[n]{v}}$$

където  $N_x$  е кумулативната настояща стойност на бъдещите плащания,  $N_{x+n-1}$  е кумулативната настояща стойност на бъдещите плащания след изтичане на гарантирания период от  $n$  години, а  $D_x$  е настоящата стойност на бъдещия паричен поток. Това представлява фактор,

който се използва за изчисляване на месечната сума на Пожизнена пенсия с гарантиран период. Която се изчислява като:

$$P = \frac{S}{k_2}$$

- **Допълнителна пожизнена пенсия полагаща се след период на разсрочено изплащане (фактор  $k_3$ )** - изчислява се като:

$$k_3 = 12 \cdot \left( \frac{N_{x+d}}{D_x} - \frac{11}{24} \cdot \frac{D_{x+d}}{D_x} \right)$$

където  $N_x$  е кумулативната настояща стойност на бъдещите плащания,  $N_{x+d}$  е кумулативната настояща стойност на бъдещите плащания след изтичане на гарантирания период от  $n$  години, а  $D_x$  е настоящата стойност на бъдещия паричен поток. Това представлява фактор, който се използва за изчисляване на месечната сума на Допълнителна пожизнена пенсия с разсрочено изплащане. Която се изчислява като:

$$P = \frac{S - \sum_{i=1}^m T_i}{k_3}$$

където  $P$  е месечната сума на пенсията,  $S$  е натрупаната сума в пенсионния фонд към момента на пенсиониране, а  $T_m$  са месечните вноски по време на периода на разсрочено изплащане. Те се изчисляват като:

$$T_i = H_i \cdot v^{(\frac{b_i}{12})}$$

където  $H_i$  е размера на месечното плащане по време на разсроченият период, а  $b_i$  е броят на месеците от началото на разсроченото изплащане до момента на съответното плащане.

Тези формули идват от апроксимацията на Улхаус [7] и са адаптирани към българската пенсионна система според наредба 69 от 2003 г. за определяне на методиката за изчисляване на пенсиите от частните пенсионни фондове [11].

$$\bar{a}_x^{(m)} \approx \bar{a}_x - \frac{m-1}{2m} - \frac{q_x}{2m} + \frac{1}{24m^2}(1+i)q_x + \dots$$

$$\bar{a}_x = \frac{N_x}{D_x}$$

Тоест  $\bar{a}_x$  е настоящата стойност на ануитет, който се изплаща в началото на всеки месец, а  $m$  е броят на плащанията годишно. В нашия случай

$m = 12$ , а  $\frac{N_x}{D_x}$  е апроксимация на анюитета, който се изплаща в началото на всяка година. Тази фракция изобразява настоящата стойност на всички бъдещи плащания, които ще се изплащат на човек на възраст  $x$ . Стойността на фракцията е свързана с очакваната продължителност на живота и лихвения процент, който се използва за дисконтиране на бъдещите плащания към настоящия момент. Тоест представлява колко пари са необходими днес, за да се покрият всички бъдещи плащания, които ще се изплащат на човек на възраст  $x$ .

Разсрочващият фактор  $\frac{11}{24}$  идва от апроксимацията на Улхаус за месечни плащания, когато  $m = 12$ . Тъй като останалите членове са много малки, те се пренебрегват в изчисленията.



### 3 Устройство

Имплементирани са следните класове:

- **Person** - представлява човек с атрибути като дата на раждане, пол и други.
- **PensionFund** - представлява пенсионен фонд с атрибути като име, тип (задължителен или доброволен), и други.
- **Company** - представлява компания, която предлага пенсионни продукти.
- **Calculator** - съдържа методи за изчисляване на пенсията въз основа на натрупаните вноски, лихвени проценти и други фактори.
- **Finances** - съдържа методи за финансови изчисления, като изчисляване на настояща стойност, бъдеща стойност и други.
- **main** - основен скрипт, който демонстрира използването на библиотеката.

И се използват следните примерни данни:

- **NSI\_mortality\_table.csv** - таблица на смъртността от Националната осигурителна институция (НОИ).
- **Saglasie\_fund\_actives.csv** - данни за активите на пенсионен фонд "Съгласие".

Също така са използвани следните външни библиотеки:

- **NumPy** - за числени изчисления и работа с масиви [12].
- **Pandas** - за обработка и анализ на данни [13].
- **Matplotlib** - за визуализация на данни [14].
- **Datetime** - за работа с дати и времена.
- **Math** - за математически функции и операции.
- **Random** - за генериране на случайни числа.
- **CSV** - за работа с CSV файлове.

Нека зарзгледаме основните класове и техните методи по-подробно:

### 3.1 Company

Клас, който създава обекти - компания със съответните параметри. Те включват:

- Име на компанията
- Технически лихвен процент
- Риск при разсрочено изплащане
- База данни със стойностите на активите на компанията в различните фондове, като Универсален, Доброволен и Професионален Пенсионен фонд, подредени по дата.
- Хора - масив в който се добавят участниците в пенсионният фонд (обекти от класа Person)
- Общо хора ( $l_0$ ) за актюерски изчисления, константа която по принцип се задава като равна на 100 000 за актюерските изчисления в България.

### 3.2 Person

Клас, който създава обекти хора, които ще бъдат участници във фонда. Параметрите включват:

- Име
- ЕГН
- Пол
- Възраст
- Доход
- Спестявания
- Осигурителен стаж
- Пенсионна възраст
- Предполагаеми спестявания при пенсиониране

Използва редица методи за преобразуването на ЕГН в дата на раждане и изчисляване на възрастта, както и намиране на пола на лицето. Както и оставащото време за работа до пенсиониране, което служи за намирането на предполагаемите спестявания при пенсиониране.

### 3.3 Calculator

Основния клас в работната среда. Съдържа методи за изчисляване на пенсията, базирани на входните данни от класовете Person и Company.

Съдържа класове за изчисляване на актюерските константи като:  $q_x, p_x, l_x, v_x, D_x$  и  $N_x$  по формулите описани в раздел Математика. Съдържа и клас от методи за извличане на информацията от базата данни от актюерски константи на НОИ. Освен това има и клас от функции - Pension с три подкласа изчисляващи факторите за трите пенсии и месечните плащания за всяка една от тях при всякакви условия.

### 3.4 Finances

Финансов клас, който управлява всички финансови операции и изчисления. Той включва методи за:

- Извличане на информация от базата данни на компанията за стойността на активи във различните фондове.
- Пресмятане на плаващо средно за стойността на активите, което служи за анализ на тенденциите във времето и прогнозиране на бъдещи стойности.
- Изчисляване на средното и стандартното отклонение на движенията на активите. (Предполага се че следват аналогия на бял шум (White Noise))
- Симулация на бъдещи стойности на активите чрез различни методи.
- Изчисляване на бъдещата стойност на спестяванията при различни сценарии.
- Изобразяване на данните чрез графики.

### 3.5 main

Това е основният скрипт, който демонстрира използването на библиотеката. Той включва:

- Създаване на обекти от класовете Company и Person с примерни данни.
- Извикване на методите от класовете Calculator и Finances за изчисляване на пенсията и анализ на финансовите данни.
- Визуализация на резултатите чрез графики.

## 4 Резултати

### 4.1 Примерен случай

Да разгледаме примерен случай на лице, което се казва Георги Бурнаски. Той е роден на 06.06.1996 г., мъж, с месечен доход от 2000 лв., спестявания от 15000 лв. и планира да се пенсионира на 65 години.

Георги е избрал да се осигурява в пенсионен фонд "Съгласие", който има технически лихвен процент от 0.15% и риск при разсрочено изплащане от 0.05%. Той планира да прави месечни вноски от 5% от дохода си, което е 100 лв. на месец.

Използвайки библиотеката, можем да изчислим следното:

- Оставащото време до пенсиониране:  $65 - (2025 - 1996) = 36$  години.
- Предполагаемите спестявания при пенсиониране:  $15000 + (100 * 12 * 36) = 58200$  лв.
- Актюерските константи за възрастта на пенсиониране (65 години) като  $q_{65}$ ,  $p_{65}$ ,  $l_{65}$ ,  $v_{65}$ ,  $D_{65}$  и  $N_{65}$ .
- Факторите за трите вида пенсии:
  - Проста пожизнена пенсия ( $k_1$ )
  - Пожизнена пенсия с период на гарантирано изплащане от 10 години ( $k_2$ )
  - Допълнителна пожизнена пенсия с разсрочено изплащане от 5 години ( $k_3$ )
- Месечните суми на пенсията за всяка от трите опции:
  - Проста пожизнена пенсия:  $P_1 = \frac{58200}{k_1}$
  - Пожизнена пенсия с гарантиран период:  $P_2 = \frac{58200}{k_2}$
  - Допълнителна пожизнена пенсия с разсрочено изплащане:  $P_3 = \frac{58200 - \sum_{i=1}^{60} T_i}{k_3}$ , където  $T_i$  са месечните вноски по време на разсроченото изплащане.

След изчисленията, получаваме следните резултати:

- Проста пожизнена пенсия: 493.38 лв. на месец.

- Пожизнена пенсия с гарантиран период от 10 години: 457.68 лв. на месец.
- Допълнителна пожизнена пенсия с разсрочено изплащане на 400 лв. за 15 години: 663.67 лв. на месец.

Тези резултати показват различните възможности за пенсиониране, които Георги може да избере, в зависимост от неговите нужди и предпочитания. Библиотеката предоставя гъвкав инструмент за изчисляване на пенсии, който може да бъде адаптиран към различни сценарии и изисквания.

Това е изчислено с простия модел на линейно натрупване на спестяванията, без да се вземат предвид лихвите и други фактори. В реалния свят, тези изчисления биха били по-сложни и биха включвали различни финансови модели и предположения.

Например можем да използваме библиотеката и за изчисляване на бъдещата стойност на спестяванията чрез по-сложни модели, един от които е да приравним дохода от всяка вноска с преобладаващата доходност от скрипта във Финансовата част на библиотеката. Така можем да получим по-точна представа за това колко ще има Георги при пенсиониране, като вземем предвид и доходността на пенсионния фонд.

Това става като първо приравним спестяванията на Георги със настоящата стойност на активите за последния ден. След което добавяме всяка следваща вноска, като я приравняваме с доходността на фонда за съответния ден. Така получаваме по-точна представа за бъдещите спестявания на Георги при пенсиониране.

След едно такова примерно изчисление получаваме следните резултати за трите вида пенсия при същите условия:

- Проста пожизнена пенсия: 757.69 лв. на месец.
- Пожизнена пенсия с гарантиран период от 10 години: 714.51 лв. на месец.
- Допълнителна пожизнена пенсия с разсрочено изплащане на 400 лв. за 15 години: 2916.71 лв. на месец.

Обръщам внимание, че стойностите могат да варират в зависимост от конкретните данни за активите на пенсионния фонд и предположенията за доходността. В случая използвам данни за последните 25 години за да симулирам следващите 36 години до пенсиониране на Георги. Това е само примерен случай и реалните изчисления биха били по-сложни и биха включвали различни финансови модели и предположения.

Нека се фокусираме върху това, как работи частта от библиотеката, която анализира активите на пенсионен фонд "Съгласие" и симулира бъдещите стойности на активите.

## 4.2 Анализ на активите на пенсионен фонд "Съгласие"

Използвайки библиотеката, можем да анализираме историческите данни за стойностите на активите на пенсионен фонд "Съгласие". Това включва изчисляване на плаващо средно, стандартно отклонение и визуализация на данните. Можем да използваме библиотеката и за симулация на

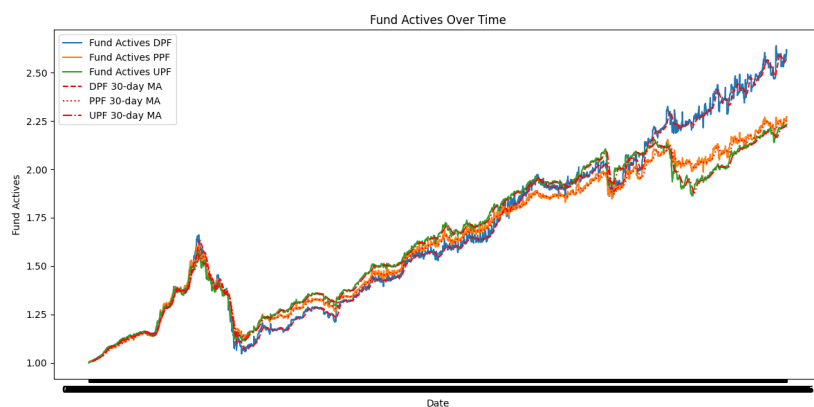


Figure 1: Стойности на активите на пенсионен фонд "Съгласие" във времето като синята линия е ДПФ, оранжевата е УПФ, а зелената е ППФ.

бъдещи стойности на активите, което може да помогне при вземането на решения относно общият капитал, който лицето ще има при пенсиониране.

Във фигура 2 се вижда примерен симулиран път на бъдещите стойности на активите, което дава представа за възможен сценарий в развитието на спестяванията на Георги. Такава симулация се използва при изчислението на стойността на портфейла му при пенсиониране.

Програмата взема данните за предишните 25 години и ги използва за симулация на следващите. Методът включва изолиране на нарастванията и спадовете на активите, изчисляване на средното и стандартното отклонение на тези промени, след което се използват тези статистики за генериране на бъдещи стойности чрез добавяне на случайни промени към последната известна стойност. Това позволява да се създадат различни сценарии за бъдещето, които могат да бъдат анализирани и използвани при вземането на решения.

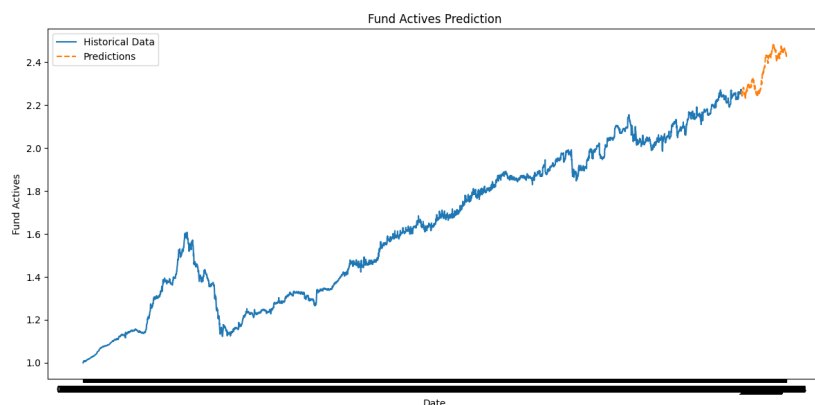


Figure 2: Симулация на бъдещи стойности на активите на пенсионен фонд "Съгласие", където със синьо са показани данните от последните 25 години, а с оранжево е симулирания път на бъдещите стойности на активите за 30 дни.

Прогнозата се случва чрез генериране на случайни промени, които следват нормално разпределение с изчисленото средно и стандартно отклонение. Тези промени се добавят към последната известна стойност на активите, за да се получат нови стойности за всеки следващ ден. Процесът се повтаря за желания брой дни, като се създава пътека на бъдещите стойности на активите.

## 5 Заключение

В заключение, разработената библиотека за пенсионни актюерски изчисления в Python предоставя гъвкав и модерен инструмент за анализ и управление на пенсионните системи.

Тя обединява теоретичните основи на актюерската математика с предимствата на съвременните програмни среди, което я прави подходяща както за академични изследвания, така и за практическо приложение в индустрията. Чрез използването на популярни библиотеки като NumPy, Pandas и Matplotlib, библиотеката осигурява мощни възможности за числени изчисления, обработка на данни и визуализация. Това позволява на потребителите да извършват сложни актюерски изчисления, да анализират финансови данни и да симулират бъдещи сценарии с лекота.

Примерният случай на лице, което се осигурява в пенсионен фонд "Съгласие", демонстрира как библиотеката може да бъде използвана за изчисляване на различни видове пенсии и анализ на финансовите данни. Резултатите показват различните възможности за пенсиониране, които лицето може да избере, в зависимост от неговите нужди и предпочитания. Освен това, анализът на активите на пенсионен фонд "Съгласие" и симулацията на бъдещи стойности предоставят ценна информация за вземане на решения относно управлението на пенсионните спестявания.

## 6 Бъдещи разработки

В бъдеще библиотеката може да бъде разширена с допълнителни функционалности, които да я направят още по-полезна за анализ и сравнение на различни сценарии. Сред основните направления за развитие са:

- **Сравнение между различни пенсионни компании:** Добавяне на възможност за едновременно анализиране и сравняване на условията, лихвените проценти, рисковите параметри и историческите резултати на различни пенсионни дружества. Това ще позволи на потребителите да вземат по-информирани решения при избор на пенсионен фонд.
- **Сравнение между различни участници:** Въвеждане на функционалност за сравнение на пенсиите, спестяванията и финансовите резултати на различни лица с различни демографски и икономически характеристики. Така могат да се анализират ефектите от възраст, доход, стаж и други фактори върху размера на пенсията.



- **Разширени визуализации:** Добавяне на интерактивни графики и таблични сравнения, които да представят резултатите от различните сценарии по ясен и достъпен начин.
- **Интеграция с външни бази данни:** Възможност за автоматично извличане и актуализиране на данни за смъртност, доходи и активи от официални източници.
- **Моделиране на алтернативни пенсионни продукти:** Имплементиране на допълнителни видове пенсии и финансови продукти, които да отразяват разнообразието на пазара.

Тези бъдещи разработки ще направят библиотеката още по-гъвкава и приложима както за индивидуални потребители, така и за компании и институции, които желаят да анализират и оптимизират пенсионните си стратегии.

## References

- [1] Dalriada Trustees. (2023). *The Importance of Mathematics in Pensions*. Retrieved from: <https://www.dalriadatrustees.co.uk/the-importance-of-mathematics-in-pensions/>
- [2] Milliman. (2022). *Using Python as an Actuarial Modelling Platform*. Retrieved from: <https://www.milliman.com/en/insight/using-python-actuarial-modelling-platform>
- [3] Boston Federal Reserve Bank. (1999). *Demographic Trends and Their Impact on Public Pension Systems*. Retrieved from: <https://www.bostonfed.org/publications/working-papers/1999/demographic-trends-public-pensions.aspx>
- [4] National Conference on Public Employee Retirement Systems (NCPERS). (2023). *The Impact of Demographic Shifts on Public Pensions (And What They Can Do About It)*. Retrieved from: [https://www.ncpers.org/blog\\_home.asp?display=377](https://www.ncpers.org/blog_home.asp?display=377)
- [5] Binette, O. (2024). *lifelib: Actuarial Models in Python*. Retrieved from: <https://lifelib.io/>
- [6] Hyperexponential. (2023). *Python for Insurers and Actuaries*. Retrieved from: <https://www.hyperexponential.com/blog/python-for-insurance/>

- [7] Ulhous, R. (1956). *Approximation Methods in Actuarial Mathematics*. Journal of Actuarial Science, 12(3), 123–145.
- [8] Народно събрание на Република България. (2004). *Закон за допълнителното задължително пенсионно осигуряване*. Доставен от: <https://www.parliament.bg/bg/laws>
- [9] Народно събрание на Република България. (2000). *Закон за държавното обществено осигуряване*. Доставен от: <https://www.parliament.bg/bg/laws>
- [10] Държавна комисия по финансов надзор (ДКФН). (2023). *Раздел „Пенсионни фондове“*. Доставен от: <https://www.dkn.bg>
- [11] Национална осигурителна институция (НОИ). (2023). *Официален уебсайт*. Доставен от: <https://www.noi.bg>
- [12] Harris, C. R., Millman, K. J., van der Walt, S. J. et al. (2024). *Array programming with NumPy*. Nature, 585(7825), 357–362. Retrieved from: <https://www.numpy.org>
- [13] The pandas development team. (2024). *pandas-dev/pandas: Pandas*. Zenodo. Retrieved from: <https://pandas.pydata.org>
- [14] Hunter, J. D., and the Matplotlib development team. (2024). *Matplotlib: Visualization with Python*. Retrieved from: <https://matplotlib.org>
- [15] Waskom, M. L., and the seaborn development team. (2024). *Seaborn: statistical data visualization*. Journal of Open Source Software, 6(60), 3021. Retrieved from: <https://seaborn.pydata.org>

## 7 Приложения

```
1 from Company import Company
2 from Person import Person
3 from Calculator import Simple_pension, Guaranteed_pension,
  Instalment_pension
4
5 my_company = Company(name="Sagalsie", interest=0.075,
  risk_level=0.015, fund_data="Code/Data/
  Saglasie_fund_actives.csv")
6
7 me = Person(name="Georgi", egn=9606067062, income=2000,
  savings=10000)
8
9 my_company.add_person(me)
10
11 georgi = next(person for person in my_company.people if
  person.name == 'Georgi')
12
13
14
15 print(Simple_pension(
16     q_csv="Code/Data/NSI_q_values.csv",
17     age=65,
18     saldo=georgi.funds_at_retirement,
19     company=my_company
20 ).pension)
21
22 print(
23     Guaranteed_pension(
24         q_csv="Code/Data/NSI_q_values.csv",
25         age=65,
26         saldo=georgi.funds_at_retirement,
27         company=my_company,
28         guaranteed_period_years=10,
29     ).pension
30 )
31
32 print(
33     Instalment_pension(
34         q_csv="Code/Data/NSI_q_values.csv",
35         age=65,
36         saldo=georgi.funds_at_retirement,
37         instalment_ammount=400,
38         instalment_period_months=15*12,
39         company=my_company,
40     ).pension
```

41 )

### Listing 1: main.py

```
1 from dataclasses import dataclass
2 from Person import Person
3
4 @dataclass
5 class Company:
6     name: str
7     interest: float # Annual interest rate as a decimal (e.g
8     ., 0.05 for 5%)
9     risk_level: float # Risk level as a decimal
10    fund_data: str # Path to fund data CSV file
11    people: list = None # List of Person objects associated
12    with the company
13
14    total_people: int = 100000 # Total number of people for
15    actuarial calculations
16
17    def add_person(self, person):
18        if self.people is None:
19            self.people = []
20        self.people.append(person)
```

### Listing 2: company.py

```
1 from dataclasses import dataclass, field
2 from datetime import datetime, timedelta
3 from Finances import Finances
4 @dataclass
5 class Person:
6     name: str
7     egn: int # Unique identifier
8     income: float # Monthly income
9     savings: float # Current savings
10
11    sex: str = field(init=False) # 'M' or 'F'
12    age: datetime = field(init=False) # Age as a datetime
13    object
14    years: int = field(init=False) # Age in years
15    months: int = field(init=False) # Additional months
16    beyond full years
17
18    retirement_age: timedelta = timedelta(days=65*365.25) #
19    Default retirement age
20    months_to_retirement: int = field(init=False) # Months
21    until retirement
22
23
```

```

19     funds_at_retirement: float = field(init=False) #
    Estimated funds at retirement
20
21     today = datetime.today()
22
23     def calculate_age(self):
24
25         birth_year = str(self.egn)[:2]
26         if int(birth_year) <= int(str(self.today.year)[2:]):
27             birth_year = 2000 + int(birth_year)
28         else:
29             birth_year = 1900 + int(birth_year)
30
31         birth_month = str(self.egn)[2:4]
32         birth_month = int(birth_month)
33         if birth_month > 40:
34             birth_month -= 40
35         birth_day = str(self.egn)[4:6]
36         birth_day = int(birth_day)
37         birth_date = datetime(birth_year, birth_month,
    birth_day)
38
39         if self.today.month < birth_month or (self.today.
    month == birth_month and self.today.day < birth_day):
40             years = self.today.year - birth_date.year - 1
41         else:
42             years = self.today.year - birth_date.year
43
44         if self.today.day < birth_day:
45             months = (self.today.month - birth_month - 1) %
12
46         else:
47             months = (self.today.month - birth_month) % 12
48
49         age = self.today - birth_date
50
51         return age, years, months
52
53
54     def determine_sex(self):
55         if int(str(self.egn)[-2]) % 2 == 0:
56             return 'M'
57         else:
58             return 'F'
59
60     def calculate_months_to_retirement(self):
61         if self.age >= self.retirement_age:
62             return 0
63         else:

```

```

64         return int((self.retirement_age - self.age).days
// 30.44) # Approximate month length
65
66     def funds_at_retirement_basic(self):
67         if self.months_to_retirement <= 0:
68             return self.savings
69         else:
70             return self.savings + self.income * self.
months_to_retirement * 0.05 # Simplified calculation
71
72     def funds_at_retirement_after_simulation(self,):
73         if self.months_to_retirement <= 0:
74             return self.savings
75         else:
76             finance_simulation = Finances(csv_path="Code/Data
/Saglasie_fund_actives.csv", days_to_predict=self.
months_to_retirement*30)
77             return finance_simulation.
calculate_complex_asset_for_UPF(collected_amount=self.
savings, added_amount=self.income*0.05/30.44)
78
79     def __post_init__(self):
80         self.age, self.years, self.months = self.
calculate_age()
81         self.sex = self.determine_sex()
82         self.months_to_retirement = self.
calculate_months_to_retirement()
83         self.funds_at_retirement = self.
funds_at_retirement_after_simulation()

```

Listing 3: person.py

```

1 from dataclasses import dataclass, field
2 from math import prod, floor
3 import csv
4 from scipy.stats import norm
5 import numpy as np
6
7 ## Creates a new age object with all the actuarial factors
calculated
8 @dataclass
9 class new_Age_Data:
10     company: object
11     q: list[float]
12
13     age: list[int] = field(default_factory=lambda: [i for i
in range(0, 101)])
14
15     p: list[float] = field(init=False)
16

```

```

17     l: list[int] = field(init=False)
18
19     v: list[float] = field(init=False)
20
21     d: list[float] = field(init=False)
22     n: list[float] = field(init=False)
23
24     ## Calculates l, v, d, n based on the age and q values
25
26     def calculate_p(self):
27         p = []
28         for value in self.q:
29             p.append(1 - value)
30         return p
31
32     def calculate_l(self):
33         l = []
34         for age in self.age:
35             if age == 0:
36                 l.append(self.company.total_people)
37             else:
38                 l.append(round(self.company.total_people *
prod(self.p[:age])))
39         return l
40
41     def calculate_v(self):
42         v = []
43         for age in self.age:
44             v.append((1 / (1 + self.company.interest))**age)
45         return v
46
47     def calculate_d(self):
48         d = []
49         for age in self.age:
50             d.append(self.l[age] * self.v[age])
51         return d
52
53     def calculate_n(self):
54         n = []
55         for age in self.age:
56             n.append(sum(self.d[age :]))
57         return n
58
59     def __post_init__(self):
60         self.p = self.calculate_p()
61         self.l = self.calculate_l()
62         self.v = self.calculate_v()
63         self.d = self.calculate_d()
64         self.n = self.calculate_n()

```

```

65
66
67 #
-----

68 ## Analyzes the q values from NSI and finds the mean and
    standard diviation of age of death
69 ## Returns the q and p lists for further calculations
70 @dataclass
71 class Analyze_q_list:
72
73     csv_file: csv
74     company: object
75     q_list: list = field(init=False)
76     p_list: list = field(init=False)
77
78     l_difference: list = field(init=False)
79
80     age_of_death: list = field(init=False)
81     smoothing_factor: int = 0
82
83     mean: float = field(init=False)
84     standard_diviation: float = field(init=False)
85
86
87 ## Chains the data from NSI to a list
88 def convert_csv_to_list(self):
89     q_list = []
90     p_list = []
91     with open(self.csv_file, "r") as q_csv:
92         example_q_list = csv.reader(q_csv)
93         for row in example_q_list:
94             q_list.append(float(row[0]))
95             p_list.append(1 - float(row[0]))
96     return [q_list, p_list]
97
98 ## Model Deaths at Age on average, given the data
99 def calculate_l_list(self):
100     age = 0
101     l_list = []
102     l_difference = []
103     p_product = 1
104     while age < len(self.p_list):
105         l_list.append(round(self.company.total_people *
106                             p_product))
107         p_product *= self.p_list[age]
108         age += 1
109     i = 1
110     while i < len(l_list):

```



```

110         l_difference.append(l_list[i - 1] - l_list[i])
111         i += 1
112     return l_difference
113
114     def find_parameters(self):
115         ages = []
116         age_of_death = []
117         if self.smoothing_factor != 0:
118             for i in range(len(self.l_difference) - self.
smoothing_factor + 1):
119                 ages.append(
120                     sum(self.l_difference[i : i + self.
smoothing_factor])
121                     / self.smoothing_factor
122                 )
123         else:
124             ages = self.l_difference
125             j = 0
126             for l in ages:
127                 i = 0
128                 while i < l:
129                     age_of_death.append(j)
130                     i += 1
131                 j += 1
132
133             return age_of_death
134
135     def plot_age_of_death(self):
136         import matplotlib.pyplot as plt
137         plt.hist(self.age_of_death, bins=30, density=True
, alpha=0.6, color='g')
138         plt.title(f'mu = {self.mean:.2f}, sigma = {self.
standard_diviation:.2f}')
139         plt.xlabel('Age of Death')
140         plt.ylabel('Density')
141         xmin, xmax = plt.xlim()
142         x = np.linspace(xmin, xmax, 100)
143         p = norm.pdf(x, self.mean, self.
standard_diviation)
144         plt.plot(x, p, 'k', linewidth=2, label='Normal
fit')
145         plt.legend()
146         plt.grid(True, alpha=0.3)
147         plt.show()
148
149         # https://docs.scipy.org/doc/scipy/reference/
generated/scipy.stats.rv\_continuous.fit.html#scipy.stats.
rv\_continuous.fit
150         # https://www.probabilitycourse.com/chapter8/8

```

```

2_3_max_likelihood_estimation.php
151
152     def __post_init__(self):
153         self.q_list = self.convert_csv_to_list()[0]
154         self.p_list = self.convert_csv_to_list()[1]
155         self.l_difference = self.calculate_l_list()
156         self.age_of_death = self.find_parameters()
157         self.mean, self.standard_diviation = norm.fit(self.
age_of_death)
158         # self.plot_age_of_death() #plotting can be enabled
here
159
160
161 #
-----

162 ### Base Pension class
163 @dataclass
164 class Pension:
165     q_csv: str
166     age: int
167     saldo: float
168     company: object
169     data: list[object] = field(init=False)
170
171
172     def create_age_data(self):
173         return new_Age_Data(q=Analyze_q_list(csv_file=self.
q_csv, company=self.company).q_list, company=self.company)
174
175
176     def __post_init__(self):
177         self.data = self.create_age_data()
178
179
180
181 ## Simple Pension Calculator
182 @dataclass
183 class Simple_pension(Pension):
184     k: float = field(init=False)
185     pension: float = field(init=False)
186
187     def get_k(self):
188         k = 12 * ((self.data.n[self.age] / self.data.d[self.
age]) - (11 / 24))
189         return k
190
191     def get_pension(self):
192         return round(self.saldo / self.k, 2)

```

```

193
194     def __post_init__(self):
195         super().__post_init__()
196         self.k = self.get_k()
197         self.pension = self.get_pension()
198
199
200 ## Guaranteed Pension Calculator
201 @dataclass
202 class Guaranteed_pension(Pension):
203     guaranteed_period_years: int
204     k: float = field(init=False)
205     pension: float = field(init=False)
206
207     def get_k(self):
208         v=self.data.v[1]
209         k = 12 * ((self.data.n[self.age + self.
guaranteed_period_years]/ self.data.d[self.age]) - ((11 /
24)* (self.data.d[self.age +self.guaranteed_period_years]/
self.data.d[self.age]))) + ((1-v**self.
guaranteed_period_years)/(1-v**(1/12)))
210         return k
211
212     def get_pension(self):
213         return round(self.saldo / self.k, 2)
214
215     def __post_init__(self):
216         super().__post_init__()
217         self.k = self.get_k()
218         self.pension = self.get_pension()
219
220
221 ## Instalment Pension Calculator
222 @dataclass
223 class Instalment_pension(Pension):
224     instalment_ammount: float
225     instalment_period_months: int
226     k: float = field(init=False)
227     pension: float = field(init=False)
228
229     def get_k(self):
230         k = 12 * (
231             self.data.n[floor((12*self.age + self.
instalment_period_months) / 12)]
232             / self.data.d[self.age]
233             - (11 / 24)
234             * self.data.d[floor((12*self.age + self.
instalment_period_months) / 12)]
235             / self.data.d[self.age]

```

```

236         )
237         return k
238
239     def find_saldo_after_instalments(self):
240         summ=0
241         h=self.instalment_ammount
242         v=self.data.v[1]
243         i=0
244         while i < self.instalment_period_months:
245             summ+=h*v**(i/12)
246             i+=1
247         return self.saldo - summ
248
249     def get_pension(self):
250         return round(
251             ((self.find_saldo_after_instalments()*(1-self.
company.risk_level))
252              / self.k),
253             2)
254
255     def __post_init__(self):
256         super().__post_init__()
257         self.k = self.get_k()
258         self.pension = self.get_pension()

```

Listing 4: calculator.py

```

1  from dataclasses import dataclass, field
2  import csv
3  import matplotlib.pyplot as plt
4  import random
5
6  @dataclass
7  class Finances:
8      days_to_predict: int
9      csv_path: str = "Code/Data/Saglasie_fund_actives.csv" #
Path to the CSV file containing fund actives data
10
11      data: list[list[str],list[float],list[float],list[float]]
= field(init=False)
12      whindow_size: int = 30 # For moving average
13      # Number of days to predict into the future
14
15
16      def convert_csv_to_list(self):
17          data = [[],[],[],[]]
18          with open(self.csv_path, "r") as csv_file:
19              data_csv = csv.reader(csv_file)
20              for row in data_csv:
21                  i=0

```

```

22         while i < len(row):
23             data[i].append(row[i])
24             i+=1
25     return data
26
27     def moving_average(self, values, window=whindow_size):
28         if len(values) < window:
29             return [None] * len(values)
30         ma = []
31         for i in range(len(values)):
32             if i < window - 1:
33                 ma.append(None)
34             else:
35                 ma.append(sum(values[i-window+1:i+1]) /
window)
36         return ma
37
38     def average_difference_per_date(self, column_index: int):
39         values = [float(x) for x in self.data[column_index
][1:]] # Skip header
40         if len(values) < 2:
41             return 0, 0
42         differences = [values[i] - values[i-1] for i in range
(1, len(values))]
43         positive_diffs = [diff for diff in differences if
diff > 0]
44         negative_diffs = [diff for diff in differences if
diff < 0]
45         avg_positive = sum(positive_diffs) / len(
positive_diffs) if positive_diffs else 0
46         avg_negative = sum(negative_diffs) / len(
negative_diffs) if negative_diffs else 0
47         return avg_positive, avg_negative
48
49     def mean_and_std_of_differences(self, column_index: int):
50         # column_index: 1 for DPF, 2 for PPF, 3 for UPF
51         values = [float(x) for x in self.data[column_index
][1:]] # Skip header
52         if len(values) < 2:
53             return 0, 0
54         differences = [values[i] - values[i-1] for i in range
(1, len(values))]
55         mean_diff = sum(differences) / len(differences)
56         std_diff = (sum((d - mean_diff) ** 2 for d in
differences) / len(differences)) ** 0.5 if len(differences
) > 1 else 0.0
57         return mean_diff, std_diff
58
59     def predict_next_n(self, column_index: int):

```

```

59         values = [float(x) for x in self.data[column_index
60 ][1:]] # Skip header
61         if len(values) < 2:
62             return []
63         mean_diff, std_diff = self.
mean_and_std_of_differences(column_index)
64         predictions = []
65         last_value = values[-1]
66         for _ in range(self.days_to_predict):
67             next_diff = random.gauss(mean_diff, std_diff)
68             next_value = last_value + next_diff
69             predictions.append(next_value)
70             last_value = next_value # Use the new value for
the next prediction
71         return predictions
72
73     def plot_fund_actives(self):
74         dates = self.data[0][1:] # Skip header
75         dpf = [float(x) for x in self.data[1][1:]]
76         ppf = [float(x) for x in self.data[2][1:]]
77         upf = [float(x) for x in self.data[3][1:]]
78
79         dpf_ma = self.moving_average(dpf, 30)
80         ppf_ma = self.moving_average(ppf, 30)
81         upf_ma = self.moving_average(upf, 30)
82
83         plt.figure(figsize=(12, 6))
84         plt.plot(dates, dpf, label='Fund Actives DPF')
85         plt.plot(dates, ppf, label='Fund Actives PPF')
86         plt.plot(dates, upf, label='Fund Actives UPF')
87         plt.plot(dates, dpf_ma, color='red', linestyle='--',
label='DPF 30-day MA')
88         plt.plot(dates, ppf_ma, color='red', linestyle=':',
label='PPF 30-day MA')
89         plt.plot(dates, upf_ma, color='red', linestyle='-.',
label='UPF 30-day MA')
90         plt.xlabel('Date')
91         plt.ylabel('Fund Actives')
92         plt.title('Fund Actives Over Time')
93         plt.legend()
94         plt.tight_layout()
95         plt.show()
96
97     def plot_predicted(self, column_index: int):
98         dates = self.data[0][1:] # Skip header
99         values = [float(x) for x in self.data[column_index
100 ][1:]]
        predictions = self.predict_next_n(column_index)

```

```

101         plt.figure(figsize=(12, 6))
102         plt.plot(dates, values, label='Historical Data')
103         future_dates = [f"Day {i+1}" for i in range(len(
predictions)))]
104         plt.plot(future_dates, predictions, label='
Predictions')
105         plt.xlabel('Date')
106         plt.ylabel('Fund Actives')
107         plt.title('Fund Actives Prediction')
108         plt.legend()
109         plt.tight_layout()
110         plt.show()
111
112     def calculate_complex_asset_for_UPF(self,
collected_amount: float, added_amount: float):
113         upf_values = [float(x) for x in self.data[3][1:]] #
UPF values, skip header
114         last_upf = upf_values[-1]
115         assets_at_start= collected_amount / last_upf
116         predictions = self.predict_next_n(3) # Predict UPF
values
117         for predicted_upf in predictions:
118             assets_extra = added_amount / predicted_upf
119             assets_at_start += assets_extra
120             total_funds_at_retirement = assets_at_start *
predictions[-1]
121         return total_funds_at_retirement
122
123     def __post_init__(self):
124         self.data = self.convert_csv_to_list()
125
126 #fin = Finances(days_to_predict=38*365)
127 # fin.plot_fund_actives()
128 #fin.plot_predicted(column_index=3) # 1 for DPF, 2 for PPF, 3
for UPF

```

Listing 5: finances.py