NIEZAWODNOŚĆ OPROGRAMOWANIA

Sprawozdanie



*Modele estymacji czasu wykrycia błędu*



**Imię i Nazwisko: Paweł Tarsała**

**Grupa: I8B3S4**

**Prowadzący: dr hab. inż. Kazimierz Worwa**

Spis Treści

[1. Opis problemu i sformułowanie zadania 3](#_Toc25571458)

[1.1. Model Jelińskiego-Morandy 3](#_Toc25571459)

[1.2. Model Schicka-Wolvertona 4](#_Toc25571460)

[1.3. Zbiór danych 4](#_Toc25571461)

[2. Specyfikacja wymagań 6](#_Toc25571462)

[3. Specyfikacje projektowe 6](#_Toc25571463)

[4. Opis implementacji 7](#_Toc25571464)

[5. Opis wyników testowania 10](#_Toc25571465)

[6. Opis instalacji 10](#_Toc25571466)

[7. Załącznik. Kod źródłowy aplikacji programowej 10](#_Toc25571467)

# Opis problemu i sformułowanie zadania

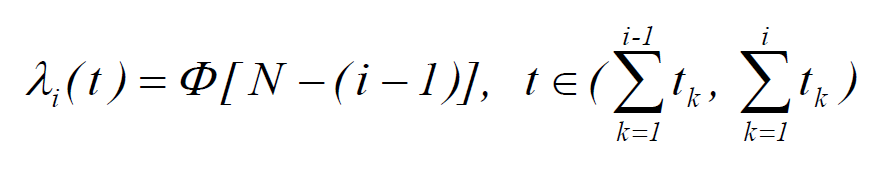
Zaprojektować i zaimplementować (w dowolnym języku i środowisku) aplikację programową, która dla zadanej dokładności obliczeń i wskazanego zbioru danych, zawierającego 240 odstępów czasowych pomiędzy wykryciem kolejnych błędów, umożliwia wyznaczenie wartości estymatorów parametrów , następujących modeli:

* Jelińskiego-Morandy,
* Schicka-Wolvertona.

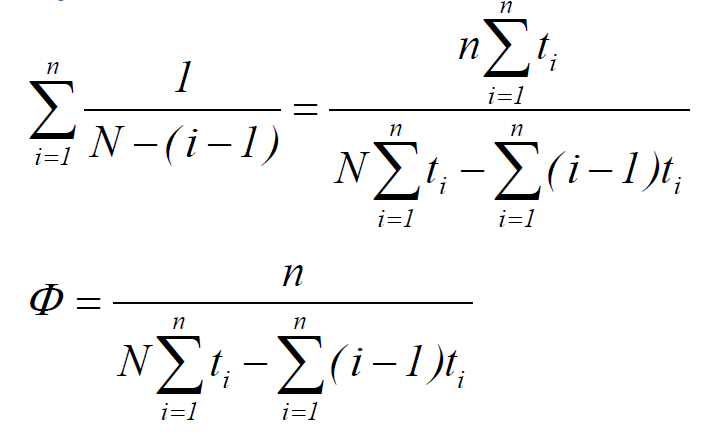
Wykorzystując wyznaczone wartości parametrów , dla każdego z ww. modeli obliczyć wartość oczekiwaną czasu, jaki upłynie do momentu wykrycia kolejnego (241.) błędu.

## Model Jelińskiego-Morandy

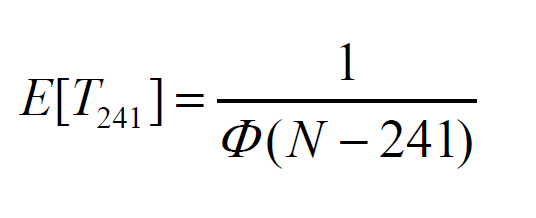
W procesie testowania programu wykryto błędów, przy czym wielkości oznaczają długości przedziałów czasu pomiędzy wykryciem kolejnych błędów. Przyjmując, że funkcja intensywności występowania błędów jest postaci:



można pokazać, że w oparciu o metodę największej wiarygodności estymatory MLE (*Maximum Likelihood Estimation*) parametrów oraz wyznacza się z zależności:

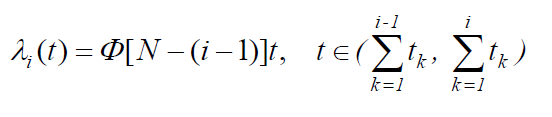


Wartość oczekiwana czasu, jaki upłynie do momentu wykrycia kolejnego (241.) błędu:



## Model Schicka-Wolvertona

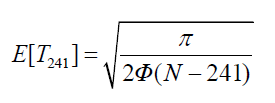
W procesie testowania programu wykryto błędów, przy czym wielkości oznaczają długości przedziałów czasu pomiędzy wykryciem kolejnych błędów. Przyjmując, że funkcja intensywności występowania błędów jest postaci:



można pokazać, że w oparciu o metodę największej wiarygodności estymatory MLE (*Maximum Likelihood Estimation)* parametrów oraz wyznacza się z zależności:

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |

Wartość oczekiwana czasu, jaki upłynie do momentu wykrycia kolejnego (241.) błędu:



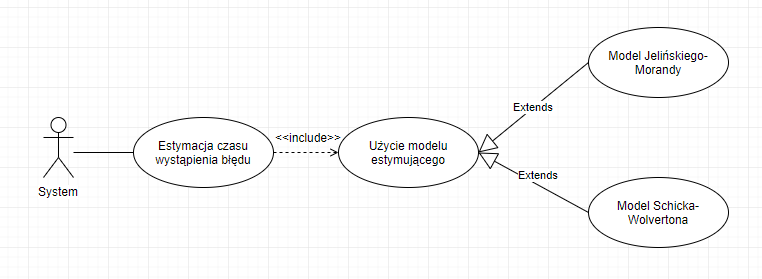
## Zbiór danych

W procesie testowania pewnego programu wykryto 240 błędów. Odcinki czasu (w minutach) pomiędzy wykryciem kolejnych błędów przedstawia poniższa tabela (czytając wierszami):

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* | *9* | *10* | *11* | *12* | *13* | *14* | *15* |
| *1* | 772 | 822 | 903 | 1003 | 1067 | 960 | 1069 | 838 | 1055 | 906 | 1061 | 1241 | 1025 | 1001 | 962 |
| *2* | 1272 | 1322 | 1403 | 1503 | 1567 | 1460 | 1569 | 1338 | 1555 | 1406 | 1561 | 1741 | 1525 | 1501 | 1462 |
| *3* | 1472 | 1522 | 1603 | 1703 | 1767 | 1660 | 1769 | 1538 | 1755 | 1606 | 1761 | 1941 | 1725 | 1701 | 1662 |
| *4* | 1672 | 1722 | 1803 | 1903 | 1967 | 1860 | 1969 | 1738 | 1955 | 1806 | 1961 | 2141 | 1925 | 1901 | 1862 |
| *5* | 1772 | 1822 | 1903 | 2003 | 2067 | 1960 | 2069 | 1838 | 2055 | 1906 | 2061 | 2241 | 2025 | 2001 | 1962 |
| *6* | 1872 | 1922 | 2003 | 2103 | 2167 | 2060 | 2169 | 1938 | 2155 | 2006 | 2161 | 2341 | 2125 | 2101 | 2062 |
| *7* | 1972 | 2022 | 2103 | 2203 | 2267 | 2160 | 2269 | 2038 | 2255 | 2106 | 2261 | 2441 | 2225 | 2201 | 2162 |
| *8* | 2072 | 2122 | 2203 | 2303 | 2367 | 2260 | 2369 | 2138 | 2355 | 2206 | 2361 | 2541 | 2325 | 2301 | 2262 |
| *9* | 2172 | 2222 | 2303 | 2403 | 2467 | 2360 | 2469 | 2238 | 2455 | 2306 | 2461 | 2641 | 2425 | 2401 | 2362 |
| *10* | 2272 | 2322 | 2403 | 2503 | 2567 | 2460 | 2569 | 2338 | 2555 | 2406 | 2561 | 2741 | 2525 | 2501 | 2462 |
| *11* | 2372 | 2422 | 2503 | 2603 | 2667 | 2560 | 2669 | 2438 | 2655 | 2506 | 2661 | 2841 | 2625 | 2601 | 2562 |
| *12* | 2472 | 2522 | 2603 | 2703 | 2767 | 2660 | 2769 | 2538 | 2755 | 2606 | 2761 | 2941 | 2725 | 2701 | 2662 |
| *13* | 2572 | 2622 | 2703 | 2803 | 2867 | 2760 | 2869 | 2638 | 2855 | 2706 | 2861 | 3041 | 2825 | 2801 | 2762 |
| *14* | 2672 | 2722 | 2803 | 2903 | 2967 | 2860 | 2969 | 2738 | 2955 | 2806 | 2961 | 3141 | 2925 | 2901 | 2862 |
| *15* | 2772 | 2822 | 2903 | 3003 | 3067 | 2960 | 3069 | 2838 | 3055 | 2906 | 3061 | 3241 | 3025 | 3001 | 2962 |
| *16* | 2972 | 3022 | 3103 | 3203 | 3267 | 3160 | 3269 | 3038 | 3255 | 3106 | 3261 | 3441 | 3225 | 3201 | 3162 |

# Specyfikacja wymagań

System ma na celu określenie estymowanego czasu wystąpienia następnego błędu. Do estymacji czasu użyłem modeli Jelińskiego-Morandy oraz Schicka-Wolvertona.



## Specyfikacja przypadków użycia

## Estymacja czasu wystąpienia błędu

## Przepływ zasadniczy

## Przepływy alternatywne

## Użycie modelu estymującego

## Przepływ zasadniczy

## Przepływy alternatywne

# Specyfikacje projektowe

Ze względu na charakter realizacji zadania, projekt nie został podzielony na oddzielne byty aplikacji. Dzięki mechanizmowi tworzenia „komórek” (eng. c*ells*) w środowisku Jupyter Notebook oraz gotowych bibliotekach wspomagających matematyczne wyliczenia, nie miałem potrzeby tworzenia oddzielnych klas, ponieważ kod stanowi spójną i czytelną całość.

Program podzieliłem na 4 komórki w celu separacji realizowanych funkcji programu:

1. Załadowanie bibliotek
2. Załadowanie danych oraz realizacja modelu Jelińskiego-Morandy
3. Realizacja modelu Schicka-Wolvertona
4. Wypisanie wyników dla obydwu modeli

Każda komórka wykonuje się oddzielnie przez interpreter Pythona. Po uruchomieniu komórki, wszystkie zmienne oraz metody są załadowywane do pamięci RAM oraz dostępne podczas uruchamiania kolejnej komórki. Kolejność wykonywania instrukcji ma kluczowe znaczenie. Poniżej znajduje się zrzut ekranu obrazujący komórkę w środowisku Jupyter. Zmienne przy uruchomieniu komórki są dostępne przez wykonanie wcześniejszych komórek.

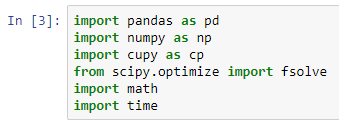


# Opis implementacji

Model Jelińskiego-Morandy oraz model Schicka-Wolvertona zostały zaimplementowane w języku Python przy użyciu środowiska Jupyter Notebook. Środowisko Jupyter jest bardzo wygodne podczas implementacji skomplikowanych matematycznych równań, ponieważ pozwala ono na wywoływanie kodu zawierającego się w zaznaczonej komórce. W ramach testowania oszczędza to wiele czasu ze względu na brak potrzeby budowania projektu, a podczas implementacji pozwala na szybką weryfikację obecnego statusu napisanego skryptu. Główną przyczyna wybrania języka Python była ogromna część wkładu społeczności internetowej w gotowe biblioteki pozwalające na wykonywanie skomplikowanych wyliczeń matematycznych. Python jest aktualnie najbardziej rozpoznawalnym językiem oraz dostarcza wiele bibliotek związanych z wnioskowaniem z danych (Data Science, Data Engineering), których twórcy należą do komercyjnych gigantów takich jak Google czy Facebook. Każda popularna biblioteka związana z obliczeniami tensorowymi jest optymalizowana pod kątem minimalizacji czasu realizacji obliczeń. Ze względu na sprzyjający format danych wejściowych (możliwość wykorzystania macierzy) wystąpiła możliwość przyspieszenia obliczeń przy pomocy użycia jednostki karty graficznej (GPU). Do zrównoleglenia obliczeń użyłem platformy obliczeniowej CUDA (produkt Nvidia), a same obliczenia wykonywane były na karcie NVIDIA Quadro M1000M, która posiada 512 wątków oraz 2GB pamięci.

Do implementacji użyłem następujących bibliotek:

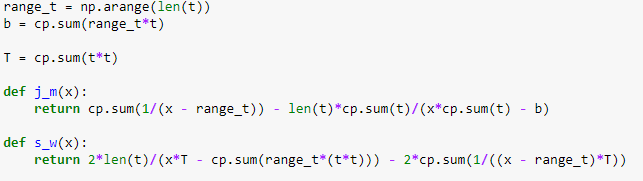
* *numpy* – pozwala na korzystanie z obliczeń macierzowych
* *cupy* – wspomaga bibliotekę *numpy* o wykorzystanie GPU
* *pandas* – pozwala na szybkie ładowanie plików oraz łatwe operowanie na danych
* *scipy* – bibliotek implementuje matematyczne równania z wykorzystaniem *numpy*
* *math –* posiada podstawowe operacje matematyczne oraz stałe*.*
* *time –* pozwala na podstawowe operacje związane z czasem



Dane testowe umieściłem w pliku *Dane.csv* w celu szybszego załadowania przez *pandas.*

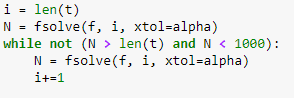


Następnie stworzyłem metody odpowiedzialne za wyliczenie w dla obydwu modeli: *j\_m(x)* dla modelu Jelińskiego-Morandy oraz *s\_w(x)* dla modelu Schicka-Walvertona.



Po przygotowaniu metod oraz danych startowych skorzystałem z funkcji optymalizacyjnej *fsolve(f, x)* zawartej w module *scipy.optimize* w celu znalezienia pierwiastków rozwiązania. Do implementacji tej szukania rozwiązania wykorzystano metodę Powella. Metoda ta minimalizuje funkcję poprzez wyszukiwanie dwukierunkowe wzdłuż każdego wektora wyszukiwania. Dwukierunkowe przeszukiwanie linii wzdłuż każdego wektora wyszukiwania przeprowadzono za pomocą przeszukiwania metodą Brenta. Metoda ta jest przydatna do obliczania lokalnego minimum funkcji ciągłej, ale złożonej, szczególnie tej bez podstawowej definicji matematycznej, ponieważ nie jest konieczne przyjmowanie pochodnych. Podstawowy algorytm jest prosty, a złożoność polega na wyszukiwaniu liniowym wzdłuż wektorów poszukiwania.

Parametr *f* funkcji *fsolve* jest funkcją jednej zmiennej, a parametr *x* jest inicjalnym estymowanym rozwiązaniem. Ze względu na to, że nie spodziewam się wartości większej od 1000 oraz mniejszej od liczby wykrytych dotychczas błędów, moja metoda rozwiązująca wygląda następująco:



, przy czym zmienna *alpha* określa precyzję szukanego rozwiązania. Następnym krokiem było wyliczenie paramtetrów dla każdego modelu:





Ostatecznym krokiem było wyliczenie wartości oczekiwanej czasu, jaki upłynie do momentu wykrycia następnego błędu:





# Opis wyników testowania

Testowałem 4 wartości parametru precyzji obliczenia *alpha.* Dla każdego testu wyliczyłem także czas wykonywania. Poniższa tabela zawiera wyniki testów.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Model Jelińskiego-Morandy | | | | Model Schicka-Wolvertona | | | |
| *alpha* |  |  |  | Czas[ms] |  |  |  | Czas[ms] |
| *0.1* | *242.31* | 4.15e-06 | 72809.42 | 9.97 | 242.10 | 6.82e-09 | 8616.63 | 9.89 |
| *0.01* | *415.09* | 1.57e-06 | 3605.32 | 20.20 | 299.60 | 2.42e-09 | 3271.24 | 29.67 |
| *0.001* | *415.12* | 1.57e-06 | 3605.08 | 19.62 | 299.79 | 2.41e-09 | 3268.63 | 39.87 |
| *0.0001* | *415.12* | 1.57e-06 | 3605.08 | 19.96 | 299.79 | 2.41e-09 | 3268.63 | 29.90 |

# Opis instalacji

W pierwszym kroku należy zainstalować CUDA, żeby użyc biblioteki *cupy*. Jeśli optymalizacja czasowa nie jest potrzebna to należy w kodzie zamienić wszystkie użycia *cp.[…]* na *np.[…]*. Następnie należy zainstalować Python w wersji 3+ oraz doinstalować wszystkie wymienione podczas importowania biblioteki komendą (w cmd):  
*pip install* [nazwa biblioteki] np. *pip install numpy*

Następnie zainstalować środowisko Jupyter Notebook „*pip install Jupyter”.* Na koniec wywołać poniższą komendę w folderze zawierającym kod „*jupyter notebook”* oraz wybrać plik z rozszerzeniem - *.ipynb.*

# Załącznik. Kod źródłowy aplikacji programowej

Kod źródłowy znajduje się w repozytorium:  
<https://github.com/BRUT4LxD/software-reliability/blob/master/Project.ipynb>