DOKUMENTACJA TECHNICZNA PROJEKTU

16 stycznia 2025

Spis treści

Rozdział 1

Struktura i komponenty projektu

1.1 Cel projektu

Celem projektu jest stworzenie symulatora pożarów, który pozwala na analizę ryzyka wystąpienia pożaru na danym obszarze dla podanych ubezpieczycieli. Symulator wykorzystuje dane dotyczące położenia budynków, ryzyka pożarowego oraz parametrów reasekuracyjnych do przeprowadzenia symulacji, które umożliwiają ocenę potencjalnych strat.

1.2 Opis ogólny

Projekt składa się z kilku kluczowych komponentów, w tym klas do przechowywania danych, funkcji do przetwarzania danych wejściowych, funkcji symulacyjnych oraz interfejsu użytkownika opartego na bibliotece **ImGui**. Klasy takie jak `VectorSim`, `VectorPozarPierwotny`, `VectorPozarRozprzestrzeniony` oraz `Ubezpieczyciel` są odpowiedzialne za organizację i przechowywanie danych związanych z symulacjami pożarów oraz informacjami o ubezpieczycielach. Program zawiera również funkcje, które wczytują i przetwarzają dane wejściowe z plików CSV, co umożliwia załadowanie istotnych informacji o pożarach i ubezpieczeniach. Zestaw funkcji symulacyjnych przeprowadza obliczenia dotyczące ryzyka pożarowego oraz rozprzestrzenienia pożarów, pozwalając na symulację różnych scenariuszy i analizę wyników. Interfejs użytkownika, stworzony z wykorzystaniem biblioteki ImGui, zapewnia łatwy dostęp do funkcji programu, umożliwiając konfigurację parametrów wejściowych, wczytywanie danych, uruchamianie symulacji oraz zapis wyników. Dodatkowo, program został zaprojektowany jako aplikacja wielowątkowa, co pozwala na równoległe przetwarzanie danych, a zastosowanie odpowiednich mechanizmów zapewnia synchronizację wątków, gwarantując bezpieczeństwo danych i efektyw-

ność wykonywanych operacji. Całość projektu tworzy złożony system, który umożliwia analizę ryzyka pożarowego dla ubezpieczeń, wspierając użytkowników w podejmowaniu lepszych decyzji na podstawie symulacji i analiz danych.

1.3 Środowisko i wymagania

1) Wymagania sprzętowe

- Współczesny komputer z systemem operacyjnym Windows 7
- Procesor wielordzeniowy
- Minimum 8 GB RAM
- Dysk SSD zalecany

2) Wymagania programowe

- Kompilator C++ (MSVC zawarty w Visual Studio 2022)
- Biblioteki: Boost, GLFW, ImGui, BS_thread_pool (domyślnie są załączone do projektu)

1.4 Instrukcja obsługi

1.4.1 Instalacja i uruchomienie programu

1. Skopiuj wszystkie pliki projektu, w tym plik rozwiązania `.sln`, na swój komputer. Upewnij się, że struktura katalogów jest zachowana.
2. Upewnij się, że masz zainstalowane *Visual Studio 2022 Community*. Podczas instalacji wybierz opcję *„Desktop development with C++”*, aby zainstalować wszystkie niezbędne narzędzia i biblioteki.
3. Otwórz plik `.sln` projektu w *Visual Studio*.
4. Wybierz odpowiednią konfigurację (`Release`) oraz platformę docelową (`x64`) z paska narzędzi.
5. Kliknij *„Build”* → *„Build Solution”* lub użyj skrótu klawiszowego `Ctrl + Shift + B`, aby skompilować projekt.

1.4.2 Konfiguracja parametrów

Uruchom program i skonfiguruj parametry symulacji za pomocą interfejsu użytkownika. W interfejsie użytkownika dostępne są wszystkie opcje pozwalające dostosować przebieg symulacji do indywidualnych potrzeb.

1.4.3 Uruchomienie symulacji

Aby rozpocząć proces symulacji, wykonaj następujące kroki:

1. Kliknij przycisk *"Wczytaj listę ubezpieczycieli"*.
2. Następnie wybierz opcję *„Wczytaj dane"*.
3. Na koniec kliknij *„Włącz symulację"*, aby uruchomić symulację.

1.4.4 Monitorowanie postępu

Podczas symulacji program wyświetla pasek postępu, który wizualizuje, jak daleko jest w procesie symulacji. Dzięki temu użytkownik może na bieżąco monitorować stan wykonywanych obliczeń.

1.4.5 Zapis wyników

Po zakończeniu symulacji wyniki są automatycznie zapisywane w wybranym katalogu w formie plików `CSV`. Pliki te można następnie otworzyć w arkuszu kalkulacyjnym lub innym programie do analizy danych.

1.5 Struktura projektu

1.5.1 Struktura katalogów

Projekt posiada następującą strukturę katalogów:

- `src/` - Katalog zawierający pliki źródłowe programu.
- `include/` - Katalog zawierający pliki nagłówkowe.
- `data/` - Katalog zawierający dane wejściowe w formacie `CSV`.
- `output/` - Katalog, w którym zapisywane są wyniki symulacji.

1.5.2 Pliki źródłowe i ich funkcje

Projekt wykorzystuje kilka plików źródłowych, które pełnią różne role w implementacji programu. Poniżej znajduje się ich opis:

- `main.cpp` - Główny plik programu, zawierający:
  - Funkcje do obsługi interfejsu użytkownika.
  - Funkcje do przetwarzania danych wejściowych.
  - Funkcje symulacyjne.
  - Algorytmy rozprzestrzeniania się pożarów.
  - Obliczenia związane z wpływem pożarów na ubezpieczenia.
  - Obsługę parametrów symulacji.
  - Wczytywaniem danych wejściowych z plików `CSV`.
  - Przetwarzaniem danych na potrzeby symulacji.
  - Zapisywaniem wyników symulacji w plikach wyjściowych.
  - Obsługa elementów graficznych z wykorzystaniem biblioteki `ImGui`.
  - Reakcje na akcje użytkownika, takie jak wczytywanie danych, konfiguracja symulacji czy uruchamianie procesu symulacyjnego.
  - Deklaracje i inicjalizacje zmiennych, m.in. takich jak:
    * `std::atomic<double> stanSymulacji = 0.0;`
    * `std::atomic<double> stanSymulacjiZapisu = 0.0;`
    * `std::atomic<double> stanSymulacjiZapisuSzkod = 0.0;`
    * `std::atomic<int> licznik_sym = 0;`
    * ...
  - Obsługę bibliotek, w tym:
    * **Standardowe biblioteki C++:**
      · `<atomic>` - Używana do obsługi zmiennych atomowych, co zapewnia bezpieczną współpracę wątków w środowisku wielowątkowym.
      · `<chrono>` - Umożliwia precyzyjne zarządzanie czasem i pomiarami czasu, np. do mierzenia czasu trwania symulacji.
      · `<iostream>` - Służy do obsługi wejścia i wyjścia, np. wypisywania komunikatów na konsolę.

        - `<vector>` - Wykorzystywana do przechowywania i manipulacji dynamicznymi tablicami danych.
        - `<string>` - Służy do obsługi ciągów znaków.
        - `<thread>` - Umożliwia tworzenie i zarządzanie wątkami w programie.
        - `<future>` - Obsługuje obiekty (`std::future`), które są wykorzystywane do zarządzania wynikami asynchronicznych obliczeń.
        - `<random>` - Używana do generowania liczb losowych w procesie symulacji.
        - `<filesystem>` - Obsługuje operacje na systemie plików, takie jak wczytywanie i zapisywanie danych.
        - `<mutex>` - Zapewnia mechanizmy synchronizacji w środowisku wielowątkowym.
        - `<deque>` - Umożliwia efektywne przechowywanie i manipulację danymi w dwukierunkowej kolejce.
        - `<condition_variable>` - Służy do zarządzania współbieżnym dostępem do zasobów przez wątki.
    - **Zewnętrzne biblioteki:**
        - `csvstream.hpp` - Biblioteka do wygodnego odczytu i zapisu danych w formacie `CSV`.
        - `BS_thread_pool.hpp` - Biblioteka implementująca pulę wątków (*thread pool*), co pozwala na efektywne zarządzanie zadaniami w środowisku wielowątkowym. W projekcie jest wykorzystywana do równoległego przetwarzania zadań symulacyjnych.
        - `boost/random/beta_distribution.hpp` - Moduł z biblioteki Boost, który implementuje rozkład beta. W projekcie służy do generowania danych losowych o określonych parametrach statystycznych.
    - **Biblioteki graficzne:**
        - `imgui.h`, `imgui_impl_glfw.h`, `imgui_impl_opengl2.h` - Biblioteki wykorzystywane do tworzenia interfejsu graficznego użytkownika (GUI). `ImGui` umożliwia tworzenie nowoczesnych i interaktywnych elementów interfejsu.
        - `GLFW/glfw3.h` - Biblioteka do obsługi okien i kontekstu OpenGL, używana w połączeniu z `ImGui`.

- Pliki związane z biblioteką `ImGui`:
    - `imgui.cpp`, `imgui_draw.cpp`, `imgui_tables.cpp`, `imgui_widgets.cpp` - Pliki implementujące podstawowe funkcje graficzne interfejsu użytkownika.

– `imgui_impl_glfw.cpp`, `imgui_impl_opengl2.cpp`, `imgui_impl_opengl3.cpp` - Pliki implementujące integrację `ImGui` z bibliotekami `GLFW` oraz `OpenGL`.

1.6 Architektura programu

Program został zaprojektowany w oparciu o trzy kluczowe komponenty:

- **Zarządzanie danymi** - odpowiedzialne za wczytywanie, przetwarzanie i zapisywanie danych.
- **Symulacja** - wykonuje obliczenia związane z rozprzestrzenianiem się pożarów i ich wpływem na ubezpieczenia.
- **Interfejs użytkownika** - umożliwia użytkownikowi interakcję z programem i konfigurację symulacji.

Program został zaprojektowany w sposób modularny, co ułatwia jego rozwój i modyfikacje.

Rozdział 2

Szczegóły techniczne implementacji

2.1 Graficzne przedstawienie przepływu danych w aplikacji

Proces działania aplikacji został przedstawiony na ogólnym diagramie, który ilustruje główne etapy pracy programu oraz przepływ danych. Diagram ten obrazuje logiczną sekwencję działań, od konfiguracji danych wejściowych, przez przygotowanie i przetwarzanie danych, aż po symulację oraz zapis wyników.

2.1.1 Opis etapów działania aplikacji

1. **Konfiguracja parametrów danych wejściowych**
   Pierwszy etap działania aplikacji obejmuje ustawienia początkowe, które użytkownik musi zdefiniować przed przystąpieniem do symulacji. W ramach tego etapu użytkownik:

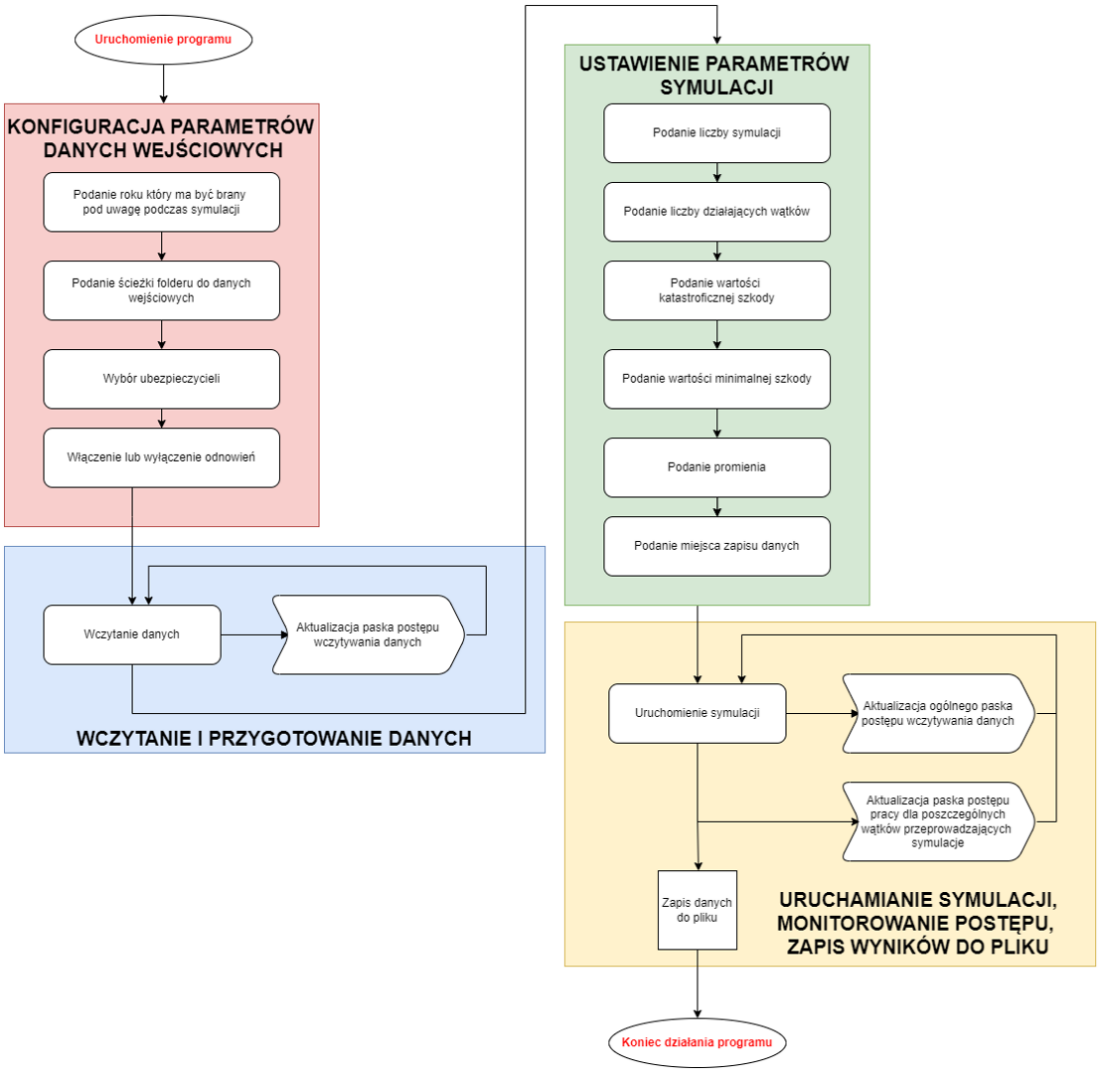
   - Podaje rok, który ma być brany pod uwagę podczas symulacji.
   - Wskazuje ścieżkę do folderu zawierającego dane wejściowe.
   - Wybiera ubezpieczycieli, których dane zostaną uwzględnione w symulacji.
   - Włącza lub wyłącza możliwość odniesień do dodatkowych odnowień.

   Po zakończeniu konfiguracji aplikacja wczytuje dane wejściowe. W trakcie tego procesu pasek postępu informuje użytkownika o statusie wczytywania danych.

2. **Ustawienie parametrów symulacji**
   Po wczytaniu danych użytkownik konfiguruje parametry symulacji. Na tym etapie definiowane są:
   - Liczba iteracji symulacji.
   - Liczba wątków, które mają być wykorzystywane do przetwarzania danych.
   - Wartości progowe szkód (minimalnej i katastroficznej).
   - Promień oddziaływania zdarzeń.
   - Lokalizacja, w której wyniki symulacji zostaną zapisane.
3. **Uruchamianie symulacji i zapis wyników**
   Ostatni etap to uruchomienie symulacji, monitorowanie jej postępu oraz zapis wyników. W trakcie tego procesu:
   - Aplikacja aktualizuje ogólny pasek postępu symulacji.
   - Aktualizowane są paski postępu dla poszczególnych wątków odpowiedzialnych za symulację.
   - Po zakończeniu działania wyniki są zapisywane w pliku CSV, a aplikacja kończy swoje działanie.

2.1.2 Schemat przepływu danych

Rycina 2.1 przedstawia graficzny schemat pracy aplikacji, podzielony na opisane trzy etapy.



Rysunek 2.1: Schemat przepływu danych i symulacji

2.2 Zarządzanie symulacjami przy użyciu puli wątków i bufora

W aplikacji zastosowano efektywny mechanizm zarządzania zadaniami w środowisku wielowątkowym. Mechanizm ten opiera się na wykorzystaniu puli wątków oraz bufora, co pozwala na równoległe przetwarzanie wielu symulacji oraz optymalizację zapisu wyników. Proces ten został przedstawiony na poniższym diagramie (Rycina 2.2).

Opis procesu zarządzania symulacjami

1. **Kolejka symulacji**
   Na początku każda symulacja trafia do kolejki zadań. Kolejka ta przechowuje symulacje oczekujące na przetworzenie. Jest to struktura danych typu FIFO (ang. *First In, First Out*), co oznacza, że symulacje są przetwarzane w kolejności ich dodania. Kolejka gwarantuje, że żadne zadanie nie zostanie pominięte.

2. **Pula wątków**
   Głównym elementem zarządzania symulacjami jest pula wątków. Każdy wątek w puli działa niezależnie i jest odpowiedzialny za przeprowadzenie jednej symulacji. Liczba wątków w puli jest konfigurowalna i zależy od dostępnych zasobów sprzętowych (np. liczby rdzeni procesora). Dzięki równoległemu przetwarzaniu:

   - Możliwe jest jednoczesne wykonywanie wielu symulacji.
   - Optymalizowane jest wykorzystanie mocy obliczeniowej systemu.
   - Minimalizowany jest czas przetwarzania dużych zbiorów danych.

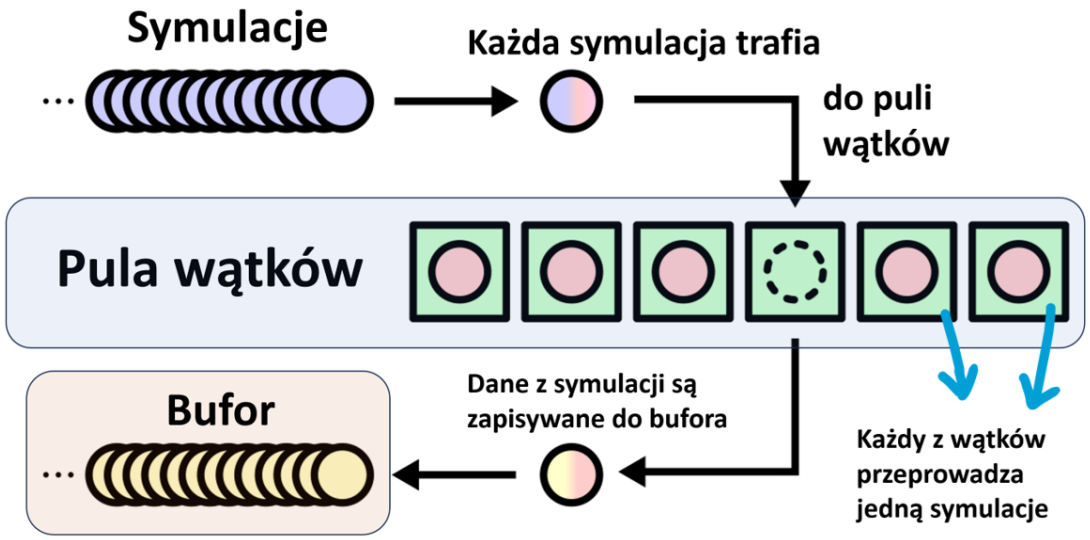
   Każda symulacja po wybraniu z kolejki jest przypisywana do jednego z wątków. Wątek przeprowadza symulację, a po jej zakończeniu zwalnia się, aby obsłużyć kolejne zadanie.

3. **Bufor wyników**
   Po zakończeniu symulacji dane są zapisywane do bufora. Bufor pełni funkcję tymczasowego magazynu wyników, które później są zapisywane w trwałej pamięci. Takie podejście pozwala na:

   - Zminimalizowanie opóźnień związanych z zapisem wyników w czasie rzeczywistym.
   - Grupowanie danych przed ich ostatecznym zapisaniem, co zwiększa efektywność operacji wejścia/wyjścia (I/O).

Schemat zarządzania symulacjami

Rycina 2.2 przedstawia graficzny schemat zarządzania symulacjami, w którym wykorzystano kolejkę zadań, pulę wątków oraz bufor wyników.



Rysunek 2.2: Schemat zarządzania symulacjami przy użyciu puli wątków i bufora

2.3 Szczegółowy opis schematu przepływu danych i symulacji w oprogramowaniu

Diagram przedstawiony na rysunku 2.3 ilustruje architekturę i przepływ danych w systemie wielowątkowym, który zarządza symulacjami w sposób równoległy. System składa się z kilku kluczowych modułów, które współpracują w celu efektywnego przetwarzania dużej liczby symulacji. Poniżej znajduje się szczegółowy opis każdego elementu.

1. Interfejs użytkownika

Interfejs użytkownika jest punktem wejścia do systemu. Umożliwia użytkownikowi dodawanie nowych symulacji do kolejki zadań. Każda nowa symulacja (np. S8) jest przekazywana do systemu za pomocą zaprojektowanego mechanizmu, który zapewnia integralność i poprawność danych wejściowych. Interfejs posiada mechanizm wizualny (pasek postępu), który informuje o stanie dodawania symulacji.

2. Kolejka zadań

Kolejka zadań pełni funkcję bufora pośredniego, w którym przechowywane są symulacje oczekujące na przetworzenie. Jak wspomniano wyżej, jest to kolejka FIFO, co oznacza, że symulacje są przetwarzane w kolejności ich dodania. Na diagramie widoczne są symulacje S7, S6, ..., S3, które czekają na przydzielenie do wątków. Mechanizm kolejki zapewnia:

- **Seryjność:** Zachowanie poprawnej kolejności przetwarzania.
- **Bezpieczeństwo wątkowe:** Synchronizacja dostępu do kolejki w środowisku wielowątkowym.

3. Pula wątków

Pula wątków jest kluczowym elementem systemu, który umożliwia równoległe przetwarzanie symulacji. Na diagramie przedstawiono dwa wątki, które równocześnie wykonują symulacje S1 i S2. Każdy wątek działa niezależnie, co pozwala na znaczną oszczędność czasu obliczeń. Wątek po zakończeniu pracy nad jedną symulacją automatycznie pobiera kolejną z kolejki. Mechanizm puli wątków zapewnia:

- **Dynamiczne zarządzanie zasobami:** Liczba wątków może być dostosowana do dostępnych zasobów sprzętowych.
- **Izolację:** Każdy wątek działa na niezależnym zestawie danych, co minimalizuje ryzyko konfliktów.

4. Symulacja

Symulacja jest procesem, który obejmuje szereg obliczeń statystycznych, geograficznych i reasekuracyjnych. Każda symulacja wykonuje następujące operacje (zgodnie z diagramem):

1. **Losowanie i próbkowanie:**
   - `randBin` - losuje liczbę pożarów na podstawie rozkładu dwumianowego, co pozwala na modelowanie zdarzeń losowych.
   - `sample_vec` - wybiera losowe próbki z wektora, co znajduje zastosowanie w analizie statystycznej.
2. **Obliczenia statystyczne:**
   - `percentage_of_loss` - oblicza procentową wielkość straty na podstawie danych wejściowych, co jest kluczowe w analizie ryzyka.

3. **Obliczenia geograficzne:**
    - `haversine_loop_cpp_vec` - oblicza odległości geograficzne między punktami, co jest istotne w modelowaniu zdarzeń przestrzennych.
4. **Reasekuracja:**
    - `reasecuration_build_fire` - oblicza kwotę reasekuracji dla budynku.
    - `reassurance_risk` - oblicza ryzyko związane z reasekuracją.
    - `calc_reas_obliga_event` - oblicza zdarzenia reasekuracyjne.
    - `calc_brutto_ring` - oblicza wartości brutto związane z ubezpieczeniami i reasekuracją.

5. Bufor wyników

Bufor wyników jest miejscem tymczasowego przechowywania danych wygenerowanych podczas działania symulacji. Buforowanie wyników zmniejsza obciążenie operacjami wejścia/wyjścia (I/O), co poprawia wydajność systemu. Dane przechowywane w buforze są następnie przekazywane do trwałej pamięci lub wykorzystywane w kolejnych etapach przetwarzania.

6. Przepływ danych

Przepływ danych w systemie można opisać w następujących krokach:

1. Użytkownik dodaje nową symulację (np. S8) za pomocą interfejsu użytkownika.
2. Symulacja trafia do kolejki zadań, gdzie oczekuje na przetworzenie.
3. Dostępne wątki w puli wątków pobierają symulacje z kolejki i rozpoczynają ich przetwarzanie.
4. Podczas działania symulacji generowane są dane, które są zapisywane w buforze.
5. Bufor przechowuje dane tymczasowo przed ich ostatecznym zapisaniem lub dalszym przetwarzaniem.

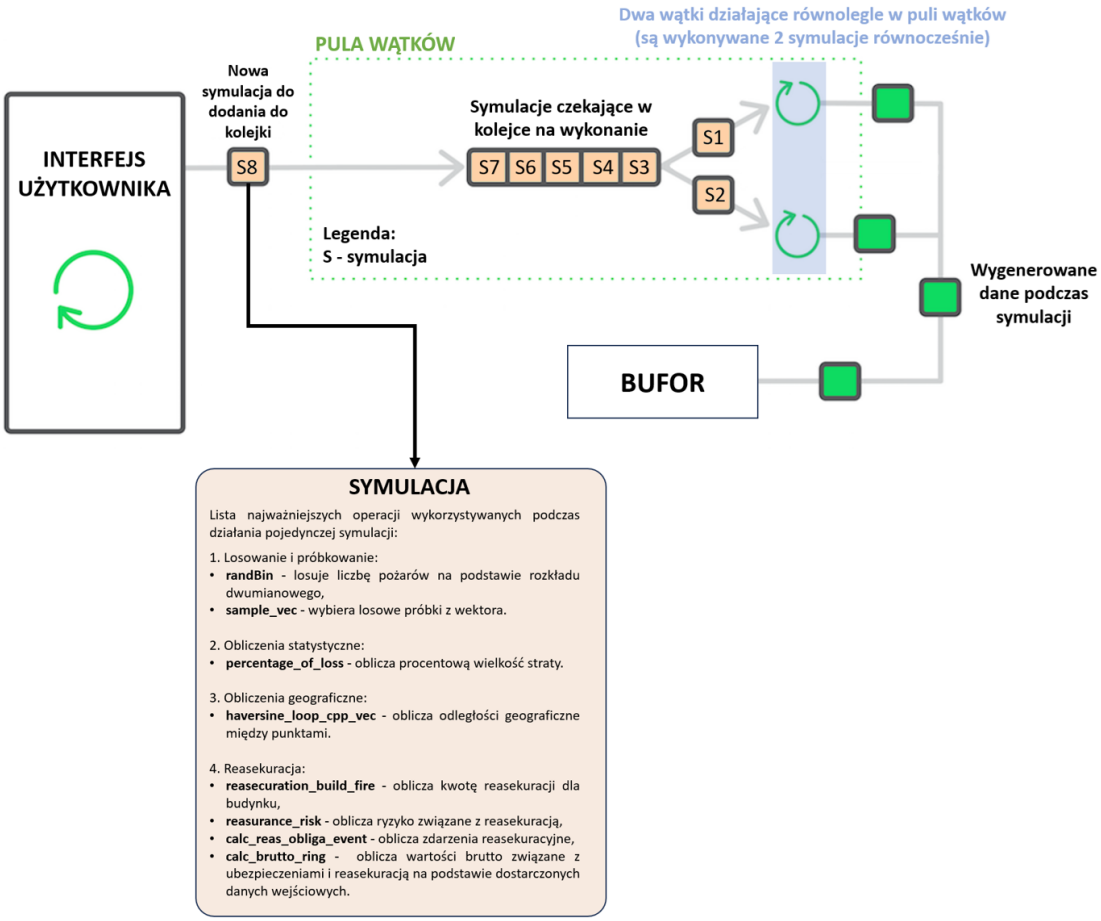
Zalety zastosowanego rozwiązania

- **Efektywność przetwarzania:** Dzięki równoległemu wykonywaniu symulacji możliwe jest znaczące skrócenie czasu obliczeń. Mechanizm puli wątków pozwala na dynamiczne przydzielanie zasobów w zależności od obciążenia.

- **Minimalizacja opóźnień:** Buforowanie wyników zmniejsza liczbę operacji zapisu na dysku, co jest szczególnie istotne w przypadku dużych symulacji generujących wiele danych.

- **Skalowalność:** Mechanizm puli wątków można łatwo skalować, dostosowując liczbę wątków do dostępnych zasobów sprzętowych.

- **Modularność:** Każdy moduł systemu działa niezależnie, co ułatwia jego rozwój i utrzymanie.



Rysunek 2.3: Schemat przepływu danych i symulacji w systemie wielowątkowym

2.4 Omówienie interfejsu graficznego aplikacji

Interfejs graficzny aplikacji został zaprojektowany w sposób intuicyjny i zorganizowany, co ułatwia użytkownikom konfigurację i monitorowanie symulacji. Składa się z kilku sekcji, które odpowiadają różnym etapom pracy z aplikacją. Poniżej opisano szczegółowo każdą sekcję interfejsu.

2.4.1 Przegląd interfejsu użytkownika

1. Sekcja przygotowania danych

Sekcja ta umożliwia wprowadzenie podstawowych informacji niezbędnych do przeprowadzenia symulacji:

- **Pole tekstowe "Rok"** – pozwala użytkownikowi wprowadzić rok, który ma być uwzględniony w analizie.
- **Pole tekstowe "Ścieżka do folderu input"** – umożliwia podanie ścieżki do folderu zawierającego dane wejściowe.
- **Przycisk "Wczytaj listę ubezpieczycieli"** – służy do załadowania listy ubezpieczycieli, którzy będą brali udział w symulacji.
- **Tabela wyboru ubezpieczycieli** – wyświetla listę ubezpieczycieli, z której użytkownik może dokonać wyboru.
- **Przycisk "Wybierz wszystkich"** – umożliwia zaznaczenie wszystkich ubezpieczycieli na liście.

2. Sekcja odnowienia

Ta sekcja pozwala na konfigurację opcji odnowień:

- **Przełącznik** – umożliwia włączenie lub wyłączenie odnowień w symulacji.

3. Sekcja uruchamiania i śledzenia wczytywania danych

Sekcja ta służy do inicjowania procesu wczytywania danych oraz monitorowania jego postępu:

- **Przycisk "Wczytaj dane"** – rozpoczyna proces wczytywania danych wejściowych.
- **Pasek postępu** – wizualizuje aktualny postęp procesu wczytywania danych.

4. Sekcja parametrów symulacji

W tej sekcji użytkownik może skonfigurować parametry symulacji:

- **Pola liczby** – pozwalają ustawić takie parametry jak:
  - liczba symulacji,
  - liczba wątków do obliczeń i zapisu,
  - wartości szkód (katastroficznych i minimalnych),
  - liczba budynków do zapisania,
  - promień analizy.
- **Pole tekstowe "Ścieżka zapisu"** – umożliwia podanie ścieżki do miejsca, w którym zostaną zapisane wyniki symulacji.

5. Sekcja wyboru zapisu budynków

W tej sekcji użytkownik może zdecydować, które dane dotyczące budynków mają zostać zapisane:

- **Opcje:**
  - **"Wszystkie budynki"** – zapisuje dane wszystkich budynków.
  - **"Wybrane budynki"** – zapisuje dane tylko wybranych budynków.

6. Sekcja uruchamiania i śledzenia symulacji

Sekcja ta pozwala na rozpoczęcie symulacji oraz monitorowanie jej postępu:

- **Przycisk "Włącz symulację"** – inicjuje proces symulacji.
- **Paski postępu:**
  - **Ogólny pasek postępu** – pokazuje ogólny postęp symulacji.
  - **Pasek postępu zapisu** – wizualizuje postęp zapisywania wyników.

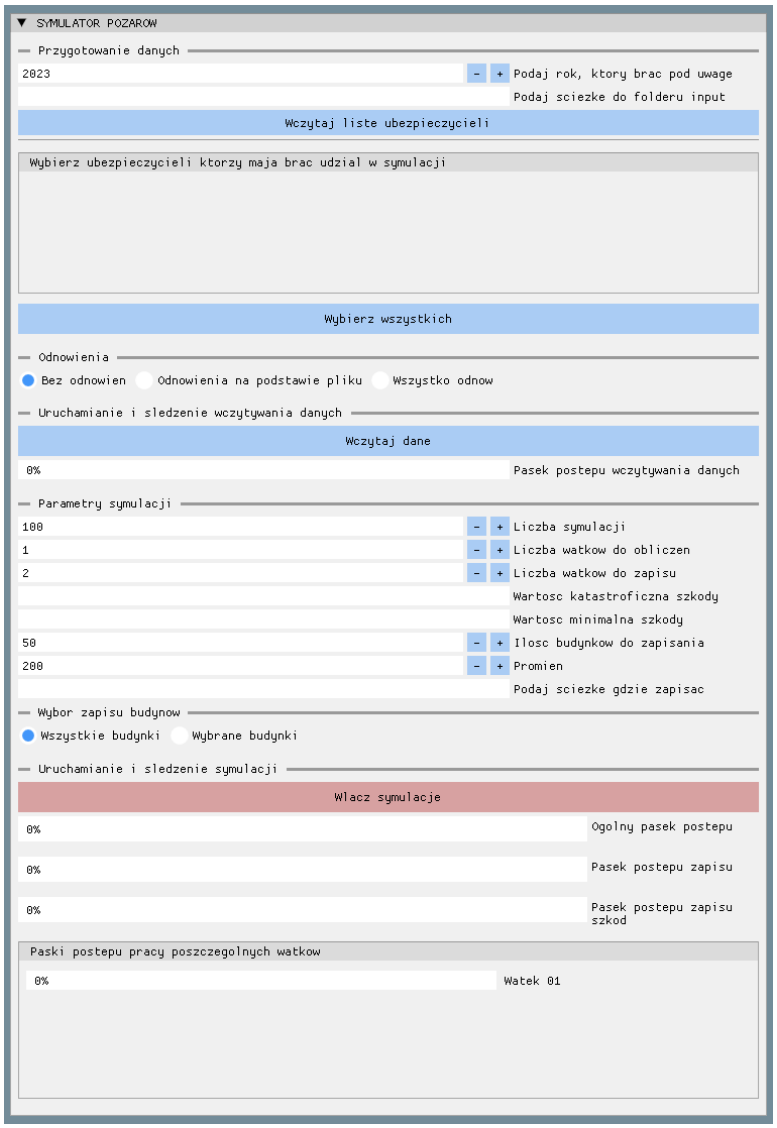
7. Sekcja pasków postępu pracy poszczególnych wątków

W tej sekcji można monitorować postęp pracy każdego wątku indywidualnie:

- **Paski postępu** – wyświetlają postęp generowania symulacji dla każdego wątku.

2.4.2 Podsumowanie

Interfejs graficzny aplikacji został zaprojektowany w sposób intuicyjny i przejrzysty. Podział na sekcje ułatwia użytkownikowi konfigurację symulacji, monitorowanie procesów oraz dostosowanie parametrów do indywidualnych potrzeb. Dzięki zastosowaniu wizualnych elementów, takich jak paski postępu, użytkownik może na bieżąco śledzić przebieg symulacji i procesów wczytywania danych.



Rysunek 2.4: Widok interfejsu graficznego aplikacji

2.5 Opis kontenerów danych

Kod implementuje symulację strat ubezpieczeniowych związanych z pożarami. Dane są przechowywane i przetwarzane za pomocą klas takich jak `Ubezpieczyciel`, `VectorSim`, `VectorPozarPierwotny` i `VectorPozarRozprzestrzeniony`. Kluczowym elementem jest klasa `Ubezpieczyciel`, która zarządza szczegółowymi danymi dotyczącymi strat ubezpieczycieli, zarówno brutto, netto, jak i katastroficznych.

2.5.1 Klasa `Ubezpieczyciel`

Klasa `Ubezpieczyciel` reprezentuje pojedynczego ubezpieczyciela i przechowuje dane dotyczące strat w różnych kategoriach.

Składowe klasy

- **Pożary pierwotne:**
  - `buildPierwotny_brutto_vec`: Wektor obiektów klasy `VectorPozarPierwotny`, zawierający dane o stratach brutto.
  - `buildPierwotny_netto_vec`: Wektor obiektów klasy `VectorPozarPierwotny`, zawierający dane o stratach netto.
- **Pożary rozprzestrzenione:**
  - `buildRozprzestrzeniony_brutto_vec`: Wektor obiektów klasy `VectorPozarRozprzestrzeniony`, zawierający dane o stratach brutto.
  - `buildRozprzestrzeniony_netto_vec`: Wektor obiektów klasy `VectorPozarRozprzestrzeniony`, zawierający dane o stratach netto.
- **Sumaryczne wartości strat:**
  - `sum_vec_out_vec`: Wektor wartości sumarycznych strat brutto.
  - `sum_vec_netto_out_vec`: Wektor wartości sumarycznych strat netto.
- **Straty katastroficzne:**
  - `buildPierwotny_brutto_kat_vec`: Wektor danych o stratach katastroficznych brutto (pożary pierwotne).
  - `buildPierwotny_netto_kat_vec`: Wektor danych o stratach katastroficznych netto (pożary pierwotne).

- `buildRozprzestrzeniony_brutto_kat_vec`: Wektor danych o stratach katastroficznych brutto (pożary rozprzestrzenione).
- `buildRozprzestrzeniony_netto_kat_vec`: Wektor danych o stratach katastroficznych netto (pożary rozprzestrzenione).
- `sum_vec_kat_out_vec`: Wektor sumarycznych wartości strat katastroficznych brutto.
- `sum_vec_netto_kat_out_vec`: Wektor sumarycznych wartości strat katastroficznych netto.

Opis przechowywanych danych

Każdy ubezpieczyciel przechowuje dane w postaci wektorów obiektów klas `VectorPozarPierwotny` i `VectorPozarRozprzestrzeniony`. Dane te są podzielone na:

- Straty brutto (przed uwzględnieniem reasekuracji).
- Straty netto (po uwzględnieniu reasekuracji).
- Straty katastroficzne (przekraczające określony próg wartości).

2.5.2 Klasa `VectorSim`

Klasa `VectorSim` jest odpowiedzialna za przechowywanie i zarządzanie danymi liczbowymi w postaci wektorów dwuwymiarowych:

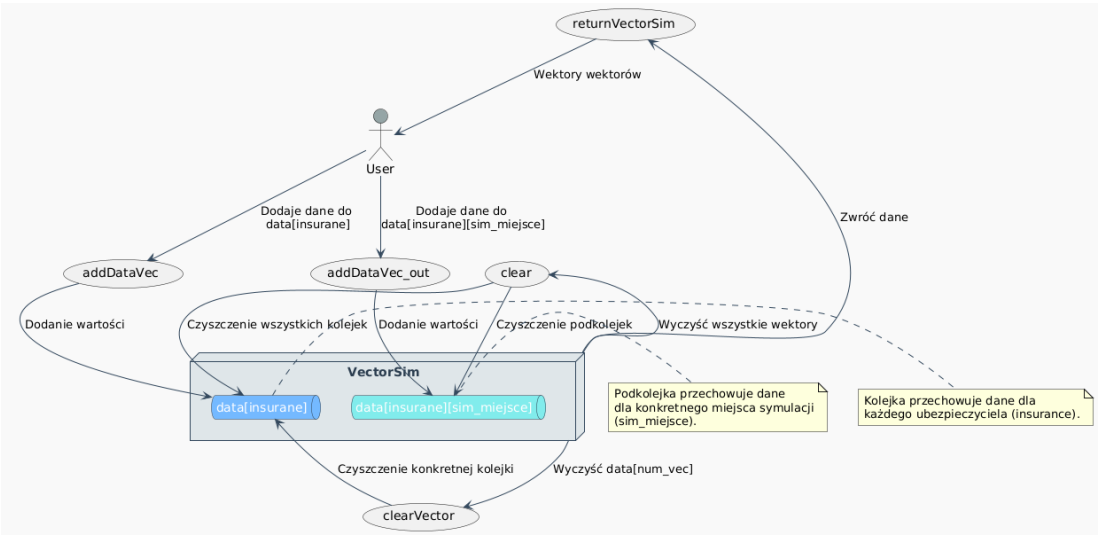
```
std::vector<std::vector<double>>
```

Funkcjonalność klasy

- **Dodawanie danych:**
  - `addDataVec(int insurance, double value)`: Dodaje wartość do określonego ubezpieczyciela.
  - `addDataVec_out(int insurance, double value, int sim_miejsce)`: Dodaje wartość do określonej symulacji.
- **Czyszczenie danych:**
  - `clearVector(int num_vec)`: Czyści dane dla określonego ubezpieczyciela.
  - `clear()`: Czyści wszystkie dane.
- **Zwracanie danych:**
  - `returnVectorSim()`: Zwraca wektor danych.

Opis diagramu kolejek zadań dla klasy `VectorSim`

Diagram przedstawia przepływ zadań i operacji wykonywanych w klasie `VectorSim`. Klasa ta zarządza hierarchiczną strukturą danych w postaci wektorów wektorów (`std::vector<std::vector<double>>`). Główne elementy diagramu to:



Rysunek 2.5: Diagram kolejek zadań dla klasy `VectorSim`.

Elementy diagramu

- **Aktor `User`**: Reprezentuje system, który korzysta z klasy `VectorSim`. Aktor inicjuje operacje na danych.
- **Kolejki danych**:
  - `data[insurane]`: Główna kolejka przechowująca dane dla poszczególnych ubezpieczycieli (`insurance`). Każdy ubezpieczyciel ma przypisaną swoją kolejkę.
  - `data[insurane][sim_miejsce]`: Podkolejka przechowująca dane dla konkretnego miejsca symulacji (`sim_miejsce`). Jest to struktura zagnieżdżona w głównej kolejce.
- **Klasa `VectorSim`**: Zarządza kolejkami danych, implementując operacje dodawania, czyszczenia i zwracania danych.

Operacje na danych

Diagram przedstawia następujące operacje wykonywane na kolejkach:

- `addDataVec(int insurane, double value)`: Dodaje wartość `value` do głównej kolejki `data[insurane]` dla danego ubezpieczyciela (`insurance`).
- `addDataVec_out(int insurane, double value, int sim_miejsce)`: Dodaje wartość `value` do podkolejki `data[insurane][sim_miejsce]` dla konkretnego miejsca symulacji.
- `clearVector(int num_vec)`: Czyści dane w konkretnej kolejce `data[num_vec]`.
- `clear()`: Czyści wszystkie kolejki i podkolejki, usuwając wszystkie dane.
- `returnVectorSim()`: Zwraca pełną strukturę danych w postaci wektora wektorów.

Podsumowanie

Diagram wizualizuje sposób, w jaki klasa `VectorSim` zarządza strukturą danych i operacjami na kolejkach. Dzięki hierarchicznej organizacji danych i metodom umożliwiającym ich modyfikację, klasa może być używana w systemach wymagających zarządzania dużymi zbiorami danych w symulacji lub innych złożonych operacji.

2.5.3 Klasa `VectorPozarPierwotny`

Klasa `VectorPozarPierwotny` przechowuje dane dotyczące pożarów pierwotnych w postaci macierzy:

std::vector<std::vector<long double>>.

Struktura danych

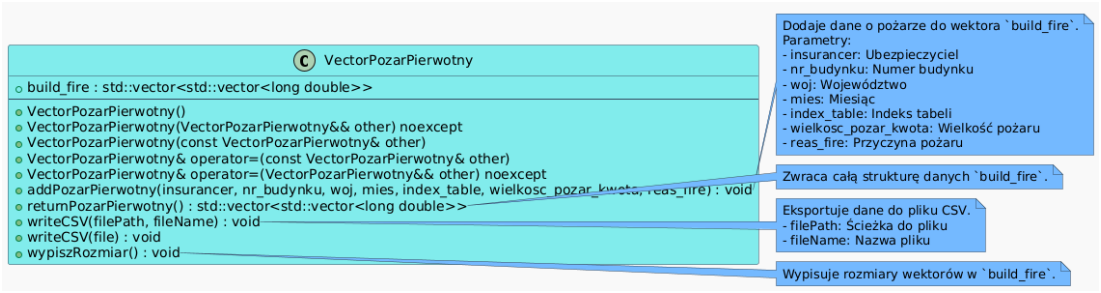
Każda kolumna macierzy reprezentuje różne informacje o pożarze pierwotnym:

- `[0]`: Numer ubezpieczyciela.
- `[1]`: Długość geograficzna budynku.
- `[2]`: Szerokość geograficzna budynku.
- `[3]`: Województwo.
- `[4]`: Miesiąc wystąpienia pożaru.
- `[5]`: Wartość sumaryczna budynku.
- `[6]`: Indeks tabeli.
- `[7]`: Wielkość strat w złotówkach.
- `[8]`: Powód pożaru.

Funkcjonalność klasy

- **Dodawanie danych:**
  - `addPozarPierwotny(...)`: Dodaje dane o nowym pożarze pierwotnym.
- **Zwracanie danych:**
  - `returnPozarPierwotny()`: Zwraca przechowywaną macierz danych.
- **Zapis do pliku CSV:**
  - `writeCSV(...)`: Zapisuje dane do pliku CSV.
- **Debugowanie:**
  - `wypiszRozmiar()`: Wyświetla rozmiary przechowywanych danych.

W poniższym diagramie przedstawiono strukturę klasy `VectorPozarPierwotny`, która zarządza danymi o pożarach.



Rysunek 2.6: Diagram klasy `VectorPozarPierwotny`

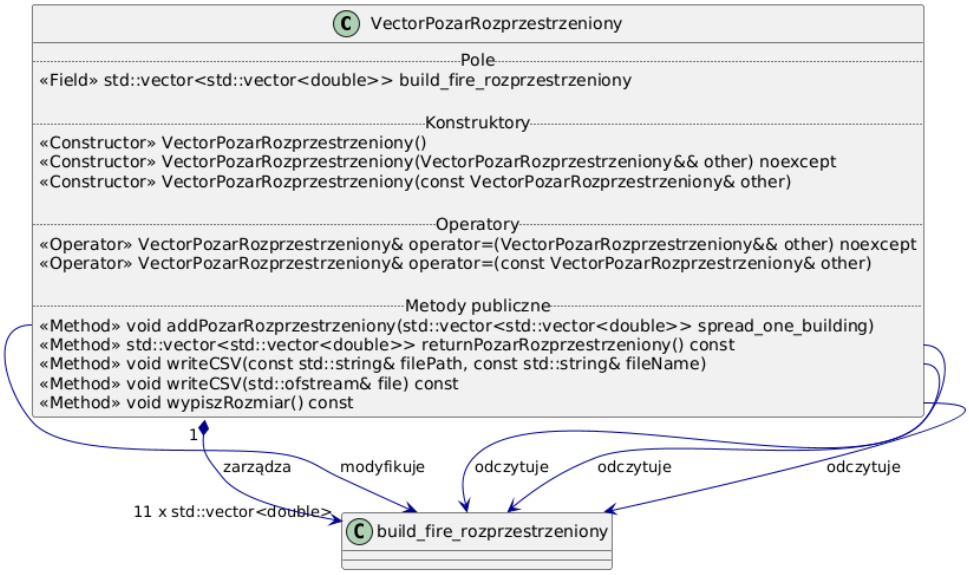
2.5.4 Klasa `VectorPozarRozprzestrzeniony`

Klasa `VectorPozarRozprzestrzeniony` przechowuje dane dotyczące pożarów rozprzestrzenionych. Struktura i funkcjonalność są podobne do klasy `VectorPozarPierwotny`, z dodatkowymi polami dotyczącymi promienia rozprzestrzenienia pożaru oraz danych o reasekuracji.

Diagram poniżej przedstawia strukturę klasy `VectorPozarRozprzestrzeniony` w języku C++. Klasa zarządza dwuwymiarowym wektorem `build_fire_rozprzestrzeniony`, który przechowuje dane o rozprzestrzenianiu się pożarów. Diagram uwzględnia:

- Pole `build_fire_rozprzestrzeniony`, które jest głównym elementem danych klasy.

- Konstruktory (domyślny, kopiujący, przenoszący) oraz operatory przypisania.
- Metody publiczne, takie jak:
  - `addPozarRozprzestrzeniony` – dodaje dane o rozprzestrzenianiu się pożaru do istniejącego wektora.
  - `returnPozarRozprzestrzeniony` – zwraca kopię danych.
  - `writeCSV` – zapisuje dane do pliku CSV.
  - `wypiszRozmiar` – wypisuje rozmiary poszczególnych wierszy wektora.



Rysunek 2.7: Diagram klasy `VectorPozarRozprzestrzeniony`.

Diagram pokazuje również przepływ danych między metodami a polem `build_fire_rozprzestrzeniony`, wskazując, które metody modyfikują, a które odczytują dane.

2.6 Opis buforów danych i mechanizmów synchronizacji

Kod implementuje wielowątkowe przetwarzanie danych z użyciem buforów i mechanizmów synchronizacji. Jego głównym celem jest zarządzanie zapisami danych o pożarach pierwotnych i rozprzestrzenionych do plików CSV, przy jednoczesnym zapewnieniu bezpieczeństwa wątków i synchronizacji.

2.6.1 Struktury danych: `BuforPierwotny` i `BuforRozprz`

Struktury te przechowują dane związane z pożarami pierwotnymi i rozprzestrzenionymi. Każda struktura zawiera:

- Obiekt klasy przechowującej dane (`VectorPozarPierwotny` lub `VectorPozarRozprzestrzeniony`).
- Ścieżkę do pliku, gdzie dane zostaną zapisane.
- Nazwę pliku.

Kod struktur:

Listing 2.1: Struktury danych

```
struct BuforPierwotny
{
    VectorPozarPierwotny vpp;
    std::string filePath;
    std::string fileName;
    BuforPierwotny(VectorPozarPierwotny&& vpp, std::string filePath, std::string
        ↪  fileName)
        : vpp(std::move(vpp)), filePath(filePath), fileName(fileName) {}
};

struct BuforRozprz
{
    VectorPozarRozprzestrzeniony vpr;
    std::string filePath;
    std::string fileName;
    BuforRozprz(VectorPozarRozprzestrzeniony&& vpr, std::string filePath, std::
        ↪ string fileName)
        : vpr(std::move(vpr)), filePath(filePath), fileName(fileName) {}
};
```

2.6.2 Globalne bufory i mechanizmy synchronizacji

Kod wykorzystuje globalne bufory i mutexy do synchronizacji:

- Bufory:
  - `global_buffer_pierwotny`: Przechowuje dane o pożarach pierwotnych.
  - `global_buffer_rozprz`: Przechowuje dane o pożarach rozprzestrzenionych.
- Mutexy:

  - `mtx_pierwotny` i `mtx_rozprz`: Chronią dostęp do buforów.
  - `mtxx` i `mtxy`: Chronią dostęp do warunkowych zmiennych.

- Zmienne warunkowe:
  - `cv` i `cy`: Powiadamiają wątki o dostępności nowych elementów w buforach.

Kod globalnych zmiennych:

Listing 2.2: Globalne bufory i mutexy

```
std::deque<BuforPierwotny> global_buffer_pierwotny;
std::deque<BuforRozprz> global_buffer_rozprz;

std::mutex mtx_pierwotny;
std::mutex mtx_rozprz;

std::mutex mtxx;
std::mutex mtxy;
std::condition_variable cv;
std::condition_variable cy;
```

2.6.3 Funkcje dodające dane do buforów

Funkcje `przeniesDoBuforPierwotny` i `przeniesDoBuforRozprz`:

- Dodają dane do odpowiednich buforów.
- Używają mutexów, aby zapewnić bezpieczeństwo dostępu.
- Powiadamiają wątki konsumentów o nowym elemencie za pomocą zmiennych warunkowych.

Kod funkcji:

Listing 2.3: Funkcje dodające dane do buforów

```
void przeniesDoBuforPierwotny(VectorPozarPierwotny&& vpp, std::string filePath,
    ↪ std::string fileName)
{
    {
        std::lock_guard<std::mutex> lock(mtxx);
        global_buffer_pierwotny.emplace_back(std::move(vpp), std::move(filePath)
            ↪ , std::move(fileName));
    }
}
```

```
void przeniesDoBuforRozprz(VectorPozarRozprzestrzeniony&& vpr, std::string
    ↪ filePath, std::string fileName)
{
    {
        std::lock_guard<std::mutex> lock(mtxy);
        global_buffer_rozprz.emplace_back(std::move(vpr), std::move(filePath),
            ↪ std::move(fileName));
    }
}
```

2.6.4 Wątki zapisujące dane

Wątki `watekZapisPierwotny` i `watekZapisRozprz`:

- Pobierają dane z buforów.
- Zapisują dane do plików CSV.
- Synchronizują dostęp do buforów za pomocą mutexów i zmiennych warunkowych.
- Kończą swoją pracę, gdy licznik symulacji (`licznik_sym`) osiągnie 0 i bufory są puste.

Kod wątków:

Listing 2.4: Wątki zapisujące dane

```
void watekZapisPierwotny() {
    while (true) {
        std::unique_lock<std::mutex> lock(mtxx);
        cv.wait(lock, []() { return !global_buffer_pierwotny.empty() ||
            ↪ licznik_sym == 0; });

        if (licznik_sym == 0 && global_buffer_pierwotny.empty()) {
            break;
        }

        if (!global_buffer_pierwotny.empty()) {
            BuforPierwotny data = std::move(global_buffer_pierwotny.front());
            global_buffer_pierwotny.pop_front();
            licznik_sym--;
            lock.unlock();
            data.vpp.writeCSV(data.filePath, data.fileName);
        }
    }
}

void watekZapisRozprz() {
```

```
    while (true) {
        std::unique_lock<std::mutex> lock(mtxy);
        cy.wait(lock, []() { return !global_buffer_rozprz.empty() || licznik_sym
            ↪  == 0; });

        if (licznik_sym == 0 && global_buffer_rozprz.empty()) {
            break;
        }

        if (!global_buffer_rozprz.empty()) {
            BuforRozprz data = std::move(global_buffer_rozprz.front());
            global_buffer_rozprz.pop_front();
            licznik_sym--;
            lock.unlock();
            data.vpr.writeCSV(data.filePath, data.fileName);
        }
    }
}
```

Opis Diagramu Systemu Buforów

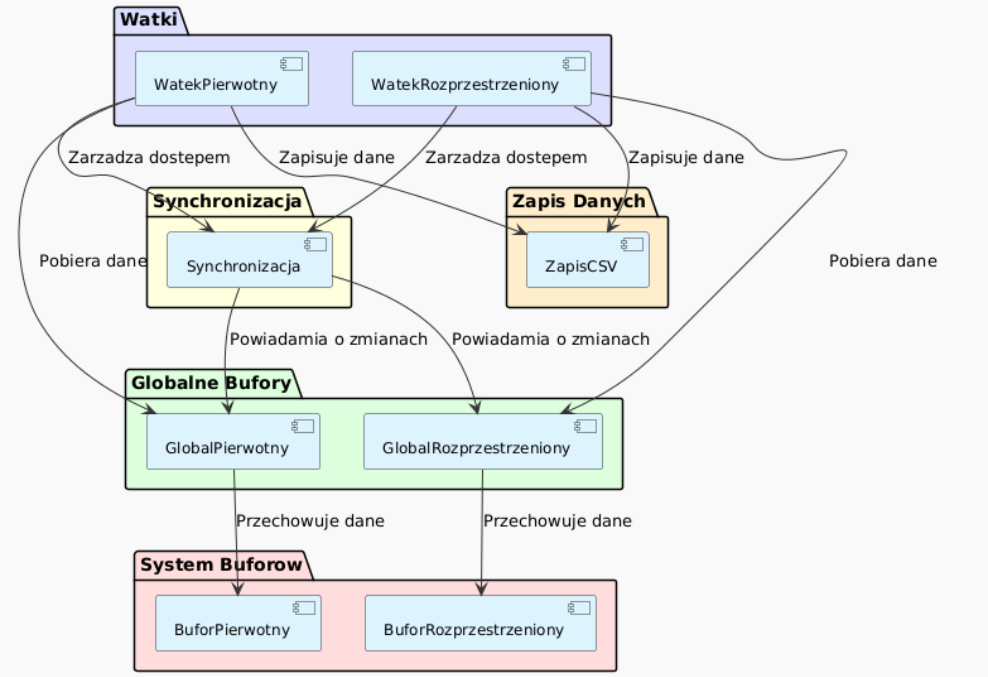
Diagram przedstawia architekturę systemu zarządzania danymi pożarowymi w środowisku wielowątkowym. W systemie wyróżniono następujące komponenty:

- **Bufory (System Buforów)**:
  - **BuforPierwotny (BP)**: Przechowuje dane dotyczące pierwotnych informacji o pożarach.
  - **BuforRozprzestrzeniony (BR)**: Przechowuje dane dotyczące rozprzestrzeniania się pożarów.
- **Globalne Bufory**:
  - **GlobalPierwotny (GP)**: Kolejka danych pierwotnych, z której korzystają wątki.
  - **GlobalRozprzestrzeniony (GR)**: Kolejka danych dotyczących rozprzestrzeniania się pożarów.
- **Wątki (Watki)**:
  - **WatekPierwotny (WP)**: Pobiera dane z kolejki **GP**, przetwarza je i zapisuje do pliku CSV.

  - **WatekRozprzestrzeniony (WR)**: Pobiera dane z kolejki **GR**, przetwarza je i zapisuje do pliku CSV.

- **Synchronizacja (Sync)**: Zarządza dostępem do globalnych buforów, zapewniając bezpieczeństwo w środowisku wielowątkowym.

- **Zapis Danych (ZapisCSV)**: Odpowiada za zapis przetworzonych danych do plików CSV.



Rysunek 2.8: Diagram architektury systemu buforów

Strzałki na diagramie przedstawiają przepływ danych oraz interakcje między komponentami:

- Wątki (**WP** i **WR**) pobierają dane z globalnych kolejek (**GP** i **GR**).

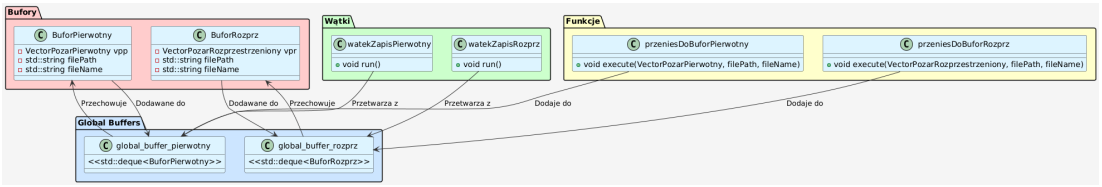
- Dane są przechowywane w odpowiednich buforach (**BP** i **BR**).

- Synchronizacja (**Sync**) zapewnia kolejność przetwarzania i dostęp do zasobów.

- Dane są ostatecznie zapisywane do plików CSV przez komponent **ZapisCSV**.

Diagram systemu zarządzania buforami

Poniżej przedstawiono diagram ilustrujący relacje między komponentami systemu zarządzania buforami:



Opis Diagramu

Diagram przedstawia system zarządzania buforami dla danych pożarów pierwotnych i rozprzestrzenionych. Składa się z następujących komponentów:

- **Bufory**:
  - *BuforPierwotny* — przechowuje dane pożarów pierwotnych.
  - *BuforRozprz* — przechowuje dane pożarów rozprzestrzenionych.
- **Globalne Bufory**:
  - *global_buffer_pierwotny* — kolejka przechowująca obiekty *BuforPierwotny*.
  - *global_buffer_rozprz* — kolejka przechowująca obiekty *BuforRozprz*.
- **Funkcje**:
  - *przeniesDoBuforPierwotny* — dodaje dane do kolejki *global_buffer_pierwotny*.
  - *przeniesDoBuforRozprz* — dodaje dane do kolejki *global_buffer_rozprz*.
- **Wątki**:
  - *watekZapisPierwotny* — przetwarza dane z kolejki *global_buffer_pierwotny*.
  - *watekZapisRozprz* — przetwarza dane z kolejki *global_buffer_rozprz*.

2.7 Opis działania funkcji

2.7.1 Funkcja `testAll`

Funkcja `testALL` jest rozbudowaną procedurą w języku C++, której celem jest zarządzanie procesem:

- wczytywania i przetwarzania danych wejściowych,
- uruchamiania symulacji,
- zapisywania wyników do odpowiednich struktur i plików.

Funkcja działa w dwóch trybach, zależnie od wartości parametru wejściowego `choice`:

- `choice = 1`: Wczytywanie danych wejściowych i ich przetwarzanie.
- `choice = 2`: Uruchamianie symulacji oraz zapisywanie wyników.

Opis szczegółowy funkcji

Parametr wejściowy

- `int choice` - parametr określający tryb działania funkcji:
  - `choice = 1` - wczytywanie i przygotowanie danych wejściowych,
  - `choice = 2` - uruchamianie symulacji i zapis wyników.

Główne etapy działania funkcji

1. Resetowanie pul wątków

```
pool.reset(liczba_dzialajacych_watkow);
poolFiles.reset(4);
```

Pule wątków są resetowane:

- `pool` - pula wątków odpowiedzialnych za symulacje. Liczba wątków jest określona przez zmienną `liczba_dzialajacych_watkow`.
- `poolFiles` - pula wątków do obsługi operacji na plikach. Liczba wątków w tej puli jest ustawiona na 4.

2. Filtrowanie nazw ubezpieczycieli

```
for (int i = 0; i < ubezp_nazwy.size(); i++) {
    if (flagi[i] != 0) {
        testVec.push_back(ubezp_nazwy[i]);
    }
}
```

Z listy nazw ubezpieczycieli (`ubezp_nazwy`) wybierane są te, które mają ustawiony odpowiedni znacznik w tablicy `flagi`. Wybrane nazwy są zapisywane w wektorze `testVec`.

3. Wyświetlanie wybranych ubezpieczycieli

```
for (int i = 0; i < testVec.size(); i++) {
    std::cout << testVec[i] << std::endl;
}
```

Wszystkie nazwy ubezpieczycieli, które zostały zapisane w `testVec`, są wypisywane w konsoli.

4. Ustawienia regionalne i kopiowanie nazw

```
std::setlocale(LC_ALL, "nb_NO.UTF-8");
std::vector<std::string> fileNames = testVec;
```

- Ustawiana jest lokalizacja na `nb_NO.UTF-8` (norweska), co jest istotne dla przetwarzania danych w lokalnych formatach liczbowych i tekstowych.
- Kopiowana jest zawartość `testVec` do wektora `fileNames`, który będzie używany w dalszej części funkcji.

Tryb `choice = 1`: Wczytywanie danych wejściowych

1. Inicjalizacja struktur danych

```
for (int woj = 0; woj < 17; ++woj) {
    exponsure_longitude[woj].resize(12);
    exponsure_latitude[woj].resize(12);
    exponsure_insurance[woj].resize(12);
    exponsure_reassurance[woj].resize(12);
    exponsure_sum_value[woj].resize(12);
}
```

Przygotowywane są struktury danych do przechowywania informacji o ekspozycji w 17 województwach. Każda z nich ma 12 elementów, co może odpowiadać np. miesiącom.

2. Ścieżki i parametry wejściowe

```
std::string dane_wejsciowe = std::string(sciezka_input);
std::string odnowienia = (czy_wlaczyc_odnowienia == 1) ? "tak" : "nie";
std::string year = std::to_string(wybrany_rok);
```

- `dane_wejsciowe` - ścieżka do katalogu z danymi wejściowymi.
- `odnowienia` - informacja, czy włączyć odnowienia (`"tak"` lub `"nie"`).
- `year` - rok wybrany do analizy.

3. Liczenie wierszy w plikach CSV

```
count_rows += count_csv_rows_1(...);
count_rows += count_csv_rows_2(...);
count_rows += count_csv_rows_3(...);
count_rows += count_csv_rows_4(...);
```

Funkcje `count_csv_rows_X` obliczają liczbę wierszy w różnych plikach CSV. Wynik jest wykorzystywany do ustalenia postępu wczytywania danych.

4. Przetwarzanie danych wejściowych

```
processReas(...);
processOblig(...);
processBudynki(...);
```

Dane wejściowe są przetwarzane przez różne funkcje:

- `processReas` - przetwarza dane reasekuracji,
- `processOblig` - przetwarza dane dotyczące zobowiązań,
- `processBudynki` - przetwarza dane dotyczące budynków.

Tryb `choice = 2`: Uruchamianie symulacji

1. Wczytywanie wartości katastroficznych i minimalnych szkód

```
std::stringstream ss(inputString);
while (std::getline(ss, token, ',')) {
    double value = std::stod(token);
    wartosci_katastrof_szk.push_back(value);
}
```

Z ciągu znaków rozdzielonego przecinkami wczytywane są wartości katastroficznych szkód (`wartosci_katastrof_szk`) oraz minimalnych szkód (`wartosci_minimal_szk`).

2. Uruchamianie symulacji

```
for (int sim_num = 0; sim_num < sim; sim_num++) {
    pool.detach_task([nazwakatalogu, sim, ...](BS::concurrency_t idx) {
        simulateExponsureTEST(...);
    });
}
pool.wait();
```

Symulacje są uruchamiane w puli wątków `pool`. Każda symulacja jest realizowana jako osobne zadanie.

3. Zapisywanie wyników

```
zapiszDoCSV(...);
create_custom_directory(full_path);
create_csv_files(full_path, subfolders[0], insurerIndex, ...);
```

Wyniki symulacji są zapisywane do plików CSV w odpowiednich katalogach.

4. Czyszczenie danych

```
out_brutto_final.clear();
out_brutto_kat_final.clear();
out_netto_final.clear();
out_netto_kat_final.clear();
```

Po zapisaniu wyników wektory wynikowe są czyszczone, aby przygotować je na kolejne symulacje.

Opis diagramu sekwencji funkcji `testALL`

Diagram sekwencji przedstawia szczegółowy przebieg działania funkcji `testALL(int choice)`, która realizuje różne zadania w zależności od wartości parametru `choice`. Funkcja ta obsługuje dwa główne scenariusze: wczytywanie i przetwarzanie danych ubezpieczeniowych (`choice == 1`) oraz wykonywanie symulacji szkód katastroficznych (`choice == 2`). Poniżej opisano szczegółowo poszczególne etapy działania funkcji.

1. Inicjalizacja

Na początku funkcja wykonuje operacje inicjalizacyjne:

- Resetuje dwie pule wątków:
  - `pool` – główna pula wątków odpowiedzialna za przetwarzanie równoległe,
  - `poolFiles` – osobna pula wątków przeznaczona do operacji na plikach.
- Tworzy pusty wektor `testVec`, który będzie przechowywał nazwy ubezpieczycieli spełniających określone kryteria.

2. Filtrowanie nazw ubezpieczycieli

W tej części funkcja iteruje po wektorze `ubezp_nazwy`, zawierającym nazwy ubezpieczycieli, i sprawdza, które elementy spełniają warunek `flagi[i] != 0`. Nazwy spełniające ten warunek są dodawane do wektora `testVec`. Po zakończeniu filtrowania zawartość `testVec` jest wyświetlana w konsoli.

3. Decyzja na podstawie parametru `choice`

Funkcja podejmuje różne działania w zależności od wartości parametru `choice`:

- **Gdy `choice == 1` (wczytywanie danych ubezpieczeniowych):**
  - Kopiuje zawartość `testVec` do wektora `fileNames`.
  - Inicjalizuje wektory ekspozycji (`exponsure_longitude`, `exponsure_latitude`, itp.) dla 17 województw.
  - Wczytuje dane wejściowe, takie jak parametry reasekuracji, dane o budynkach, prawdopodobieństwa pożaru, itp.
  - Wyświetla pasek postępu wczytywania danych, obliczając krok postępu na podstawie liczby wierszy w przetwarzanych plikach CSV.

  - Na końcu tej sekcji wyświetla czas wczytywania danych oraz informację o poprawnym zakończeniu wczytywania.

- **Gdy `choice == 2` (symulacja szkód katastroficznych):**
  - Funkcja parsuje wartości katastroficznych i minimalnych szkód dla ubezpieczycieli, przekształcając ciągi znaków na liczby zmiennoprzecinkowe.
  - Wyświetla te wartości w konsoli, w zależności od liczby ubezpieczycieli.
  - Inicjalizuje parametry symulacji, takie jak liczba iteracji, krok zapisu wyników oraz liczba ubezpieczycieli.
  - Tworzy folder wyjściowy, w którym będą zapisywane wyniki symulacji.
  - Uruchamia wątki odpowiedzialne za zapis danych w tle (`watekZapisPierwotny` i `watekZapisRozprz`).
  - Dla każdej iteracji symulacji uruchamia zadania równoległe w puli wątków `pool`, które wykonują symulację ekspozycji.
  - Po zakończeniu symulacji czeka na zakończenie wszystkich zadań w puli wątków i zapisuje wyniki do plików CSV.
  - Wyświetla czas trwania symulacji oraz informację o poprawnym zapisaniu wyników.

4. Zakończenie

Na końcu funkcja wyświetla status końcowy, informując o poprawnym zakończeniu wczytywania danych lub zapisaniu wyników symulacji do plików. W przypadku `choice == 2`, funkcja dodatkowo tworzy strukturę katalogów i zapisuje wyniki w odpowiednich podfolderach, takich jak `Brutto`, `Brutto_Kat`, `Netto` i `Netto_Kat`.

Kluczowe elementy diagramu

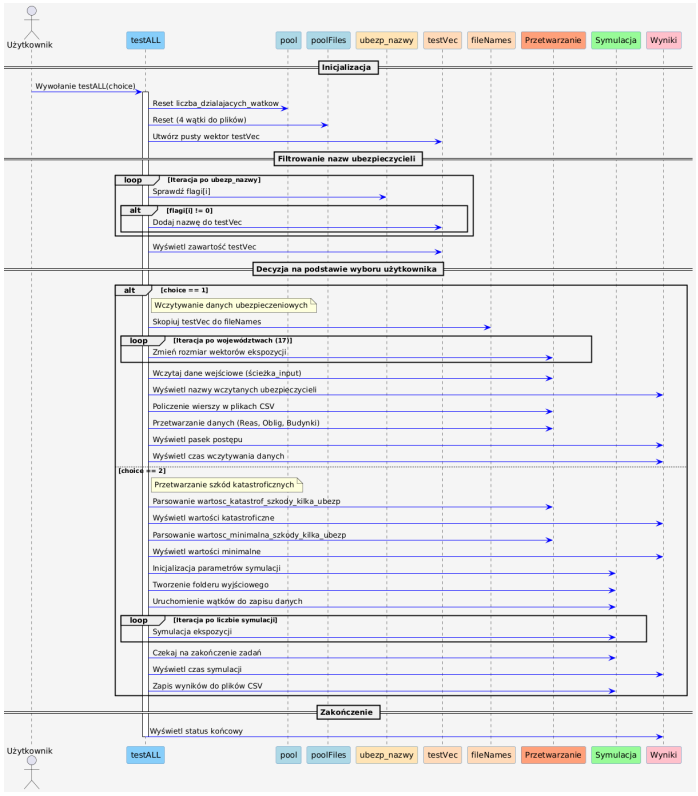
Diagram sekwencji przedstawia interakcje pomiędzy następującymi elementami:

- **Użytkownik** – inicjuje funkcję `testALL`, przekazując jej parametr `choice`.
- **Funkcja `testALL`** – główny element, który zarządza całym procesem wczytywania danych lub wykonywania symulacji.
- **Pule wątków (`pool` i `poolFiles`)** – umożliwiają równoległe przetwarzanie danych oraz operacje na plikach.

- **Wektory danych (`testVec` i `fileNames`)** – przechowują przefiltrowane nazwy ubezpieczycieli.
- **Przetwarzanie danych (`Processing`)** – obejmuje wczytywanie i przetwarzanie plików wejściowych.
- **Symulacja (`Simulation`)** – odpowiada za wykonywanie równoległych symulacji szkód katastroficznych.
- **Wyniki (`Output`)** – odpowiada za wyświetlanie informacji o postępie, czasie przetwarzania oraz zapis wyników do plików.

5. Diagram sekwencji

Diagram sekwencji przedstawiający przebieg działania funkcji `testALL` znajduje się na rycinie 2.9.



Rysunek 2.9: Diagram sekwencji funkcji `testALL`

Podsumowanie

Diagram w pełni odzwierciedla logikę funkcji `testALL`, podzielonej na etapy inicjalizacji, filtrowania danych, decyzji na podstawie parametru `choice`, przetwarzania danych lub symulacji oraz zakończenia. Dzięki zastosowaniu diagramu sekwencji, procesy równoległe oraz interakcje pomiędzy poszczególnymi komponentami zostały przedstawione w sposób czytelny i uporządkowany.

2.7.2 Funkcja `simulateExponsureTEST`

Funkcja `simulateExponsureTEST` symuluje skutki pożarów w różnych lokalizacjach w zależności od parametrów takich jak:

- liczba ubezpieczycieli,
- wielkość strat,
- reasekuracja,
- rozprzestrzenianie się pożarów.

Wyniki symulacji są zapisywane w strukturach danych i mogą być eksportowane do plików CSV.

- **Rozmiar kroku symulacji**:

$$\text{step\_size} = \frac{1.0}{17 \cdot 12 + \text{ilosc\_ubezpieczycieli}}$$

  Oznacza wielkość kroku postępu dla każdego wątku. Liczba $17 \cdot 12$ odpowiada liczbie województw (17) i miesięcy (12).

- **Rozmiar kroku paska postępu**:

$$\text{bar\_step} = \frac{1.0}{(17 + \text{ilosc\_ubezpieczycieli}) \cdot \text{sim}}$$

  Oznacza wielkość kroku dla paska postępu, uwzględniając liczbę województw, ubezpieczycieli i symulacji.

- **Wielkość pożaru w procentach**:

  $\text{wielkosc\_pozar\_kwota} = \text{wielkosc\_pozar\_procent} \cdot \text{exponsure\_sum\_value}[woj][mies][nr\_budynku]$

  Oznacza wartość strat w wyniku pożaru dla konkretnego budynku.

- **Minimalna wartość strat**:

$$\text{wielkosc\_pozar\_kwota} \geqslant 500$$

  Wartość strat nie może być mniejsza niż 500.

Opis działania funkcji

- **Inicjalizacja zmiennych**:
  Funkcja definiuje zmienne takie jak rozmiar kroku, liczba budynków i wartości strat.
- **Iteracja przez województwa i miesiące**:
  Symulacja jest wykonywana dla każdego województwa i miesiąca.
- **Losowanie liczby pożarów**:
  Liczba pożarów jest losowana na podstawie rozkładu dwumianowego.
- **Obliczanie strat**:
  Dla każdego budynku obliczana jest wielkość strat w procentach i kwotach.
- **Uwzględnienie reasekuracji**:
  Straty są dzielone pomiędzy ubezpieczycieli z uwzględnieniem reasekuracji.
- **Zapisywanie wyników**:
  Wyniki są zapisywane w strukturach danych i mogą być eksportowane do plików CSV.

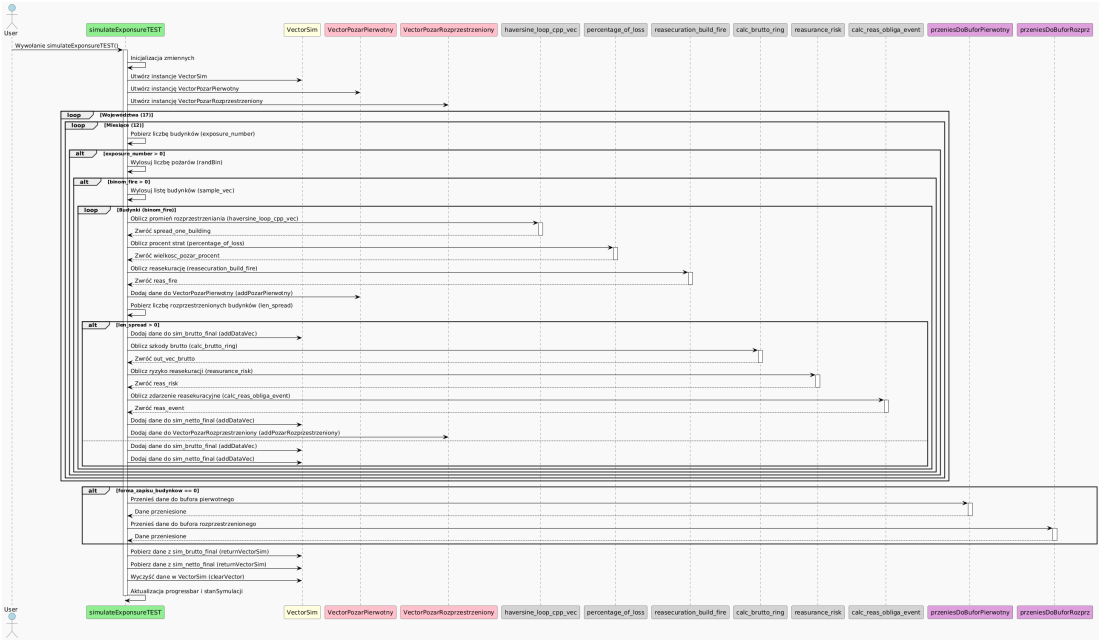
Opis diagramu sekwencji

Diagram przedstawia przepływ danych i wywołań funkcji w programie symulacyjnym 'simulateExponsureTEST'. Funkcja ta jest kluczowym elementem systemu symulacji ryzyka pożarowego i reasekuracyjnego. Główne elementy diagramu to:

- **simulateExponsureTEST** – główna funkcja odpowiedzialna za inicjalizację i zarządzanie przepływem symulacji.
- **VectorSim, VectorPozarPierwotny, VectorPozarRozprzestrzeniony** – klasy przechowujące dane dotyczące budynków, pożarów pierwotnych i rozprzestrzenionych.
- **Funkcje pomocnicze** – takie jak *haversine_loop_cpp_vec*, *percentage_of_loss*, czy *calc_brutto_ring*, które realizują obliczenia związane z promieniem rozprzestrzeniania, stratami i ryzykiem reasekuracyjnym.

- **Pętle i warunki** – diagram zawiera pętle dla województw, miesięcy i budynków oraz warunki decyzyjne, które określają, czy dane są przetwarzane.
- **Bufory** – dane są przenoszone do buforów (‘przeniesDoBuforPierwotny‘, ‘przeniesDoBuforRozprz‘) przed zapisaniem.

Poniżej znajduje się diagram sekwencji ilustrujący ten proces:



Rysunek 2.10: Diagram sekwencji funkcji `simulateExponsureTEST`.

2.7.3 Funkcja `randZeroToOne`

Funkcja `randZeroToOne` generuje losową liczbę rzeczywistą z przedziału $[a, b]$ przy użyciu rozkładu jednostajnego. Matematycznie, jeśli $X \sim U(a, b)$, to X jest zmienną losową o rozkładzie jednostajnym na przedziale $[a, b]$, gdzie wszystkie wartości z tego przedziału mają jednakowe prawdopodobieństwo.

$$P(X = x) = \begin{cases} \frac{1}{b-a}, & \text{dla } x \in [a, b], \\ 0, & \text{dla } x \notin [a, b]. \end{cases}$$

Listing 2.5: Funkcja `randZeroToOne`

```
double randZeroToOne(int a, int b)
{
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_real_distribution<double> distribution;
    distribution.param(std::uniform_real_distribution<double>::param_type(a, b))
        ↪ ;
    return distribution(gen);
}
```

2.7.4 Funkcja `sample_vec`

Funkcja `sample_vec` losowo wybiera próbkę o rozmiarze n (`sampleSize`) z populacji `population`. Wybór odbywa się poprzez losowanie indeksów elementów z populacji za pomocą funkcji `randZeroToOne`.

Dla populacji $P = \{p_1, p_2, \ldots, p_N\}$ i próbki $S = \{s_1, s_2, \ldots, s_n\}$:

$$S = \{P[i] \mid i \sim U(0, N-1)\}, \quad i = 1, \ldots, n.$$

Listing 2.6: Funkcja `sample_vec`

```
std::vector<int> sample_vec(std::vector<int>& population, int sampleSize)
{
    std::vector<int> sampleData(sampleSize);
    for (int i = 0; i < sampleSize; i++)
    {
        int randomIndex = randZeroToOne(0.0, population.size() - 1);
        sampleData[i] = population[randomIndex];
    }
    return sampleData;
}
```

2.7.5 Funkcja `randBin`

Funkcja `randBin` generuje losową liczbę całkowitą na podstawie rozkładu dwumianowego $B(n, p)$, gdzie n to liczba prób (`size_exp`), a p to prawdopodobieństwo sukcesu (`prob_size`).

$$P(X = k) = \binom{n}{k} p^k (1-p)^{n-k}, \quad k = 0, 1, \ldots, n.$$

Listing 2.7: Funkcja `randBin`

```
int randBin(int size_exp, double prob_size)
```

```
{
    std::random_device rd;
    std::mt19937 gen(rd());
    std::binomial_distribution<> distrib(size_exp, prob_size);

    return distrib(gen);
}
```

2.7.6 Funkcja `search_closest`

Funkcja `search_closest` wyszukuje indeks najbliższego elementu x w posortowanej tablicy $A = \{a_1, a_2, \ldots, a_n\}$. Używa algorytmu wyszukiwania binarnego (`std::lower_bound`) w celu optymalizacji.

$$\text{closest}(x) = \arg\min_i a_i - x, \quad i = 1, \ldots, n.$$

Listing 2.8: Funkcja `search_closest`

```
int search_closest(const std::vector<double>& sorted_array, double x)
{
    auto iter_geq = std::lower_bound(
        sorted_array.begin(),
        sorted_array.end(),
        x);

    if (iter_geq == sorted_array.begin())
    {
        return 0;
    }
    else if (iter_geq == sorted_array.end())
    {
        return sorted_array.size() - 1;
    }

    double a = *(iter_geq - 1);
    double b = *(iter_geq);

    if (fabs(x - a) < fabs(x - b))
    {
        return iter_geq - sorted_array.begin() - 1;
    }
    return iter_geq - sorted_array.begin();
}
```

2.7.7 Funkcja `percentage_of_loss`

Funkcja `percentage_of_loss` wyznacza procent strat w zależności od wielkości pożaru i jego prawdopodobieństwa. Wykorzystuje funkcję `randZeroToOne` do generowania losowego prawdopodobieństwa oraz funkcję `search_closest` do znalezienia najbliższej wartości w wektorze prawdopodobieństw.

$$\text{percentage\_of\_loss} = \text{exponsure\_sensitiv}[\text{search\_closest}(\text{probability}, \text{randZeroToOne}(0,1))].$$

Listing 2.9: Funkcja `percentage_of_loss`

```
double percentage_of_loss(std::vector<std::vector<double>> wielkosc_pozaru)
{
    int ind_prob;
    double exp_sen;
    double val_dist;
    val_dist = randZeroToOne(0, 1);
    std::vector<double> probability;
    probability = wielkosc_pozaru[1];
    std::vector<double> exponsure_sensitiv;
    exponsure_sensitiv = wielkosc_pozaru[0];
    ind_prob = search_closest(probability, val_dist);
    exp_sen = exponsure_sensitiv[ind_prob];
    return (exp_sen);
}
```

2.7.8 Funkcja `calc_reas_bligator`

Funkcja `calc_reas_bligator` oblicza wysokość ryzyka przypisanego do reasekuracji obowiązkowej w zależności od sumy składek ubezpieczeniowych (`sum_prem`) oraz parametrów ryzyka obowiązkowego (`vec_obligat_insur_risk`).

Matematycznie funkcja realizuje następujące operacje:

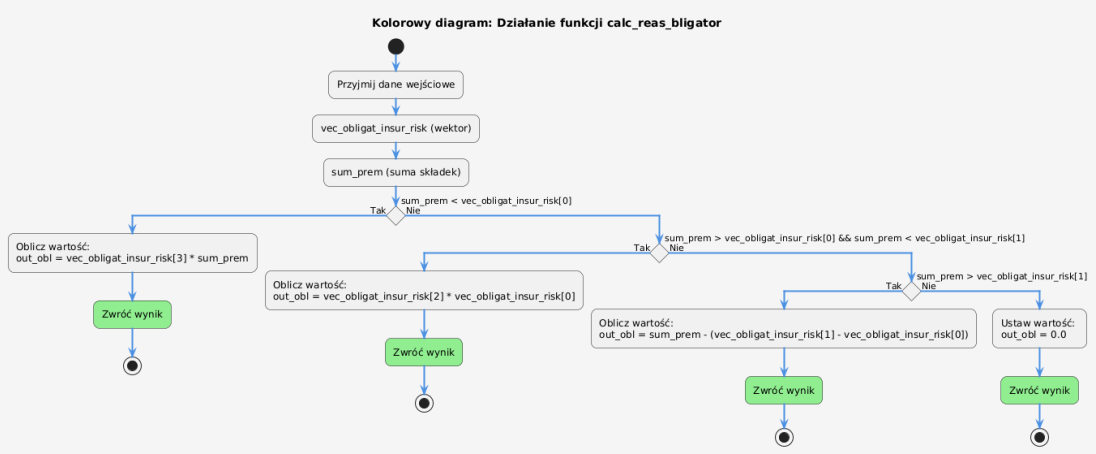
$$\texttt{out\_obl} = \begin{cases} \texttt{vec\_obligat\_insur\_risk}[3] \cdot \texttt{sum\_prem}, \\ \text{dla } \texttt{sum\_prem} < \texttt{vec\_obligat\_insur\_risk}[0], \\ \\ \texttt{vec\_obligat\_insur\_risk}[2] \cdot \texttt{vec\_obligat\_insur\_risk}[0], \\ \text{dla } \texttt{vec\_obligat\_insur\_risk}[0] \leqslant \texttt{sum\_prem} < \texttt{vec\_obligat\_insur\_risk}[1], \\ \\ \texttt{sum\_prem} - (\texttt{vec\_obligat\_insur\_risk}[1] - \texttt{vec\_obligat\_insur\_risk}[0]), \\ \text{dla } \texttt{sum\_prem} \geqslant \texttt{vec\_obligat\_insur\_risk}[1]. \end{cases}$$

- `vec_obligat_insur_risk` - wektor zawierający dane ryzyka ubezpieczeniowego,
- `sum_prem` - suma składek.

Logika działania funkcji opiera się na sprawdzeniu kilku warunków i odpowiednim obliczeniu wartości wyjściowej `out_obl`. Aby lepiej zrozumieć proces przepływu danych w funkcji, poniżej przedstawiono diagram kolejki.

Diagram kolejki funkcji

Diagram kolejki ilustruje przepływ danych i procesy zachodzące w funkcji `calc_reas_bligator`.



Rysunek 2.11: Diagram kolejki funkcji `calc_reas_bligator`

Opis działania funkcji

Funkcja działa w następujący sposób:

1. Użytkownik wywołuje funkcję `calc_reas_bligator`, przekazując dwa argumenty: `vec_obligat_insur_risk` oraz `sum_prem`.
2. Funkcja pobiera dane wejściowe:
   - `vec_obligat_insur_risk` - wektor zawierający wartości graniczne i współczynniki ryzyka,
   - `sum_prem` - suma składek.
3. Funkcja sprawdza kolejne warunki:
   - Jeśli `sum_prem < vec_obligat_insur_risk[0]`,
     obliczana jest wartość:
     `out_obl = vec_obligat_insur_risk[3] * sum_prem`.
   - Jeśli `sum_prem > vec_obligat_insur_risk[0]` oraz
     `sum_prem < vec_obligat_insur_risk[1]`,
     obliczana jest wartość:
     `out_obl = vec_obligat_insur_risk[2] * vec_obligat_insur_risk[0]`.
   - Jeśli `sum_prem > vec_obligat_insur_risk[1]`,
     obliczana jest wartość:
     `out_obl = sum_prem - (vec_obligat_insur_risk[1] - vec_obligat_insur_risk[0])`.
   - W przeciwnym przypadku `out_obl` jest ustawiane na `0.0`.
4. Po wykonaniu obliczeń funkcja zwraca wynik `out_obl`.

Wnioski

Diagram kolejki funkcji `calc_reas_bligator` pozwala lepiej zrozumieć przepływ danych oraz logikę działania funkcji. Dzięki wizualizacji można łatwiej przeanalizować warunki oraz sposób obliczania wartości wyjściowej.

2.7.9 Funkcja `reasecuration_build_fire`

Funkcja `reasecuration_build_fire` oblicza wartość ryzyka związanego z reasekuracją w przypadku pożaru dla danego budynku. Wykorzystuje parametry reasekuracji obowiązkowej i fakultatywnej oraz wcześniejsze obliczenia dotyczące ekspozycji na ryzyko pożaru.

Matematycznie:

- Jeśli budynek jest objęty reasekuracją fakultatywną:

  $$\texttt{reas\_fakultat} = \texttt{exp\_fire\_pre} \cdot b_f + \max\big(0, (1-b_f) \cdot \texttt{exp\_fire\_pre} - \texttt{vec\_fakul\_insur\_val}[1]\big).$$

- W przeciwnym przypadku:

  $$\begin{aligned}\texttt{reas\_fakultat} = {} & \min\big(\texttt{exp\_fire\_pre}, \texttt{vec\_fakul\_insur\_val}[0]\big) + \\ & \max\big(0, \texttt{exp\_fire\_pre} - \texttt{vec\_fakul\_insur\_val}[0] - \texttt{vec\_fakul\_insur\_val}[1]\big). \quad (2.1)\end{aligned}$$

- Wartość reasekuracji obowiązkowej jest obliczana za pomocą funkcji `calc_reas_bligator`.

Opis logiki funkcji

Na rycinie 2.12 przedstawiono diagram obrazujący proces obliczania wartości zmiennej `reas_oblig` na podstawie danych wejściowych dotyczących ekspozycji na ryzyko pożaru (`exp_fire_pre`), regionu (`woj`, `mies`) oraz numeru budynku (`nr_budynku`).

Logika funkcji składa się z następujących kroków:

Krok 1: Odczytanie danych wejściowych: Funkcja rozpoczyna od odczytania danych wejściowych:

- `exp_fire_pre` – wartość ekspozycji na ryzyko pożaru,
- `woj`, `mies` – identyfikatory regionu (województwo i miesiąc),
- `nr_budynku` – numer budynku.

Krok 2: Inicjalizacja zmiennych: Następnie inicjalizowane są zmienne:

- `reas` – wartość reasekuracji,
- `vec_fakul_insur_num`, `vec_fakul_insur_val` – wektory dotyczące reasekuracji fakultatywnej,
- `vec_obligat_insur_risk` – wektor dotyczący reasekuracji obligatoryjnej,
- `reas_fakultat` i `reas_oblig` – początkowo ustawione na wartość `exp_fire_pre`.

Krok 3: Obliczenie wartości `reas`: Wartość `reas` jest obliczana na podstawie tablicy `exponsure_reassurance`, z uwzględnieniem danych wejściowych (`woj`, `mies`, `nr_budynku`).

Krok 4: Sprawdzenie warunku `reas < 999`: Jeśli `reas` jest mniejsze niż 999, wykonywane są kolejne operacje:

a. **Sprawdzenie obecności `reas` w `vec_fakul_insur_num`**: Jeśli `reas` znajduje się w wektorze `vec_fakul_insur_num`, wykonywane są obliczenia reasekuracji fakultatywnej:

   - Wartość `b_f` jest pobierana z `vec_fakul_insur_val`.
   - Obliczana jest wartość `reas_fakultat` według wzoru:

$$\texttt{reas\_fakultat} = \texttt{exp\_fire\_pre} \cdot \texttt{b\_f} + \max\Big(0,\ (1 - \texttt{b\_f}) \cdot \texttt{exp\_fire\_pre} - \texttt{vec\_fakul\_insur\_val[reas][1]}\Big) \tag{2.2}$$

   - Wartość `reas_oblig` jest aktualizowana na podstawie `reas_fakultat`.

b. **Obliczenia alternatywne dla reasekuracji fakultatywnej**: Jeśli `reas` nie znajduje się w `vec_fakul_insur_num`, obliczana jest wartość `reas_fakultat` według wzoru:

$$\texttt{reas\_fakultat} = \min\Big(\texttt{exp\_fire\_pre}, \texttt{vec\_fakul\_insur\_val[reas][0]}\Big) + \max\Big(0,\ \texttt{exp\_fire\_pre} - \texttt{vec\_fakul\_insur\_val[reas][0]} - \texttt{vec\_fakul\_insur\_val[reas][1]}\Big) \tag{2.}$$

   Wartość `reas_oblig` jest również aktualizowana na podstawie `reas_fakultat`.

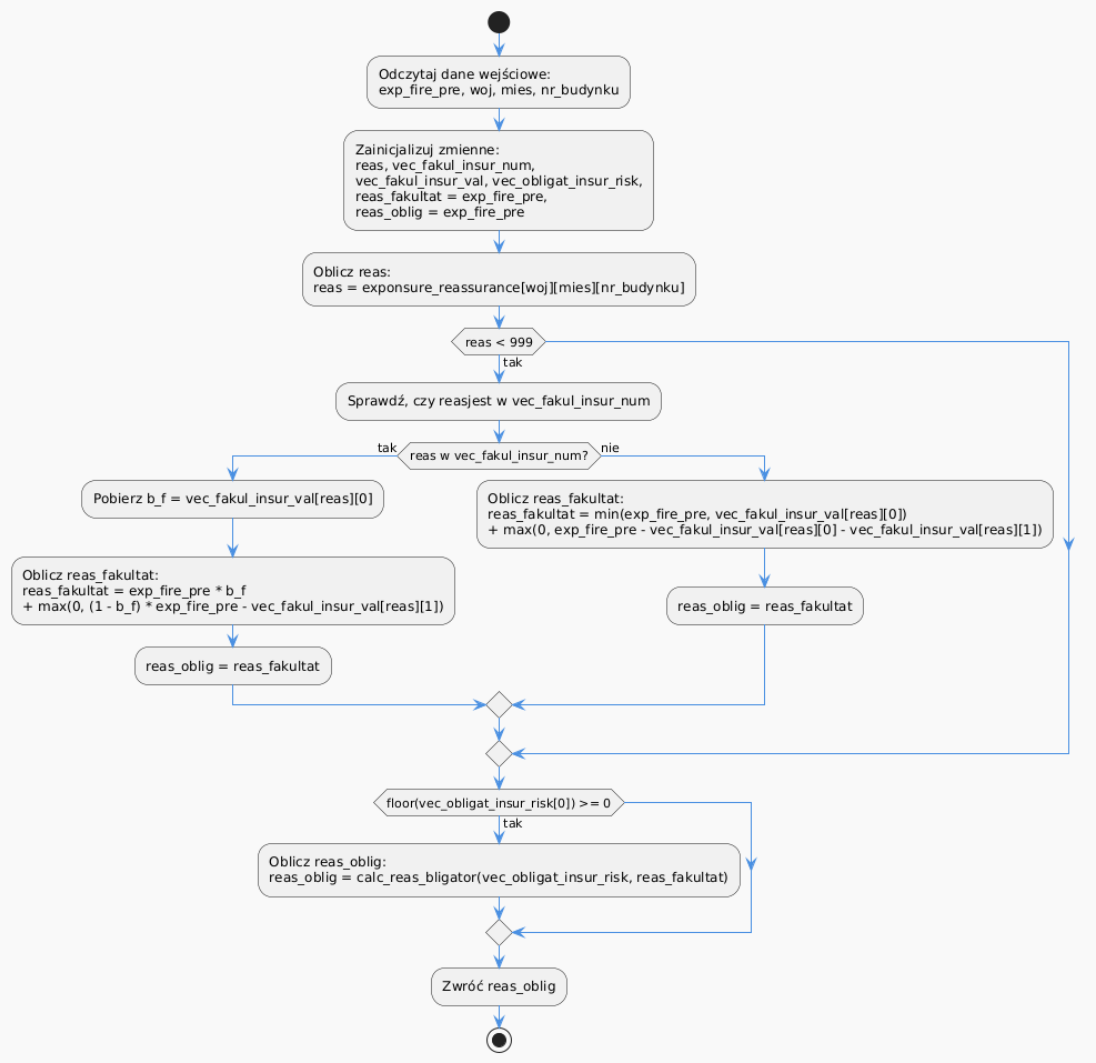
Krok 5: Sprawdzenie warunku obligatoryjnego: Jeśli wartość `floor(vec_obligat_insur_risk[0])` jest większa lub równa 0, obliczana jest wartość `reas_oblig` za pomocą funkcji `calc_reas_bligator`. Funkcja ta uwzględnia:

- wektor ryzyk obligatoryjnych (`vec_obligat_insur_risk`),
- wartość `reas_fakultat`.

Krok 6: Zwrócenie wyniku: Na końcu funkcja zwraca wartość `reas_oblig`, która reprezentuje końcową wartość zabezpieczenia reasekuracyjnego.

Uwagi

Diagram przedstawia szczegółową logikę funkcji, uwzględniając zarówno przypadki reasekuracji fakultatywnej (indywidualnej), jak i obligatoryjnej (grupowej). Każdy krok jest realizowany w sposób warunkowy, zależny od danych wejściowych i wartości pośrednich.



Rysunek 2.12: Przebieg logiki funkcji `reasecuration_build_fire`

Opis funkcji `index_spread_build`

Funkcja `index_spread_build` służy do modelowania rozprzestrzeniania się pożarów w oparciu o dane wejściowe dotyczące położenia geograficznego, odległości, ekspozycji oraz parametrów ubezpieczeniowych. Wynik działania funkcji to tablica danych (`std::vector<std::vector<double>>`), która zawiera szczegółowe informacje o rozprzestrzenionych pożarach.

Wejście funkcji

Funkcja przyjmuje następujące parametry:

- `lat_center`, `lon_center` – współrzędne geograficzne centrum obszaru.
- `distance_res` – macierz odległości dla pierścieni.
- `lat_ring`, `lon_ring` – współrzędne geograficzne dla kolejnych pierścieni.
- `insu_ring`, `reas_ring` – dane ubezpieczeniowe i przyczyny pożarów dla pierścieni.
- `exposure_sum_ring` – suma ekspozycji w pierścieniach.

Działanie funkcji

Funkcja wykonuje następujące kroki:

1. **Inicjalizacja zmiennych pomocniczych**: Tworzone są wektory przechowujące dane tymczasowe, takie jak odległości, współrzędne, ubezpieczenia, przyczyny pożarów oraz wyniki.
2. **Iteracja przez pierścienie (0–8)**: Dla każdego pierścienia funkcja:
   - Sprawdza, czy pierścień zawiera ekspozycje.
   - Oblicza prawdopodobieństwo rozprzestrzenienia pożaru w oparciu o dane wejściowe oraz parametry rozkładu beta.
   - Losuje liczbę ekspozycji, na które pożar się rozprzestrzenił.
   - Dodaje dane o pożarach do wyniku, uwzględniając minimalną wartość strat (`500`).
3. **Zwrócenie wyników**: Wyniki są zwracane w formie tablicy `out_data`.

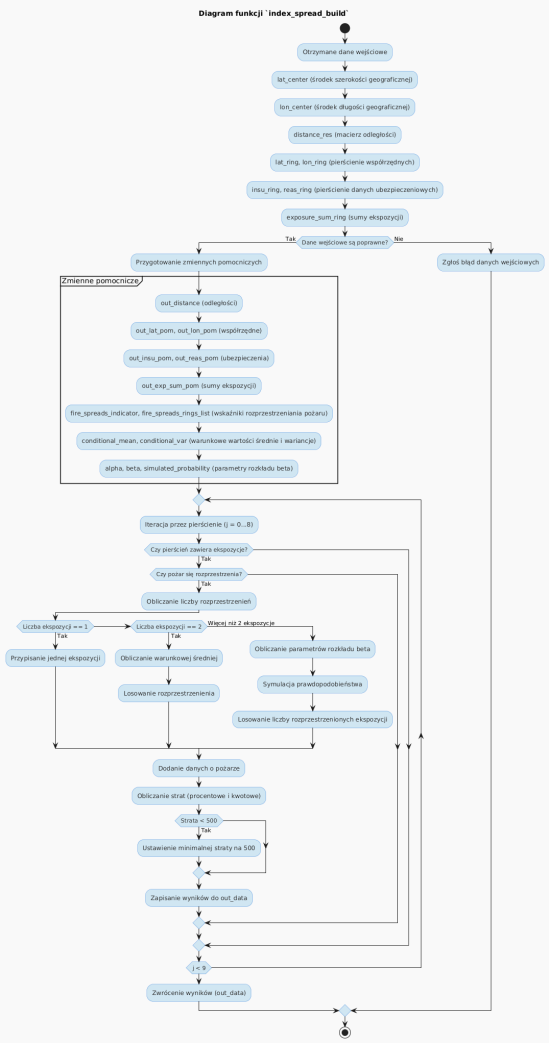
Wyjście funkcji

Funkcja zwraca tablicę `out_data`, która zawiera następujące dane:

- Odległości do ekspozycji.
- Współrzędne geograficzne ekspozycji.
- Dane ubezpieczeniowe i przyczyny pożarów.
- Suma ekspozycji.
- Wielkość strat w wyniku pożaru.

Diagram przepływu funkcji

Na rycinie 2.13 przedstawiono diagram przepływu funkcji `index_spread_build`, który ilustruje jej działanie krok po kroku.



Rysunek 2.13: Diagram przepływu funkcji `index_spread_build`

2.7.10 Funkcja `mean_spread_function`

Funkcja `mean_spread_function` oblicza średnią wartość rozprzestrzeniania się pożaru w zależności od pierścienia (`ring`), ekspozycji (`exposure`) oraz parametrów μ. Matematycznie jest zdefiniowana jako:

$$\mu_{\text{spread}} = \frac{\text{exposure}}{1 + \exp(-\mu_0) \cdot \text{ring}^{-\mu_1} \cdot \text{exposure}^{-\mu_2}},$$

gdzie:

- μ_0, μ_1, μ_2 są parametrami z wektora `mu_spread_parameters`,
- $\exp(-\mu_0)$ jest funkcją wykładniczą,
- `ring` to numer pierścienia (odległość od centrum),
- `exposure` to liczba obiektów w danym pierścieniu.

Funkcja `var_spread_function`

Funkcja `var_spread_function` oblicza wariancję rozprzestrzeniania się pożaru w pierścieniu, uwzględniając średnią wartość rozprzestrzeniania (μ_{spread}) oraz parametry wariancji (σ):

$$\sigma^2_{\text{spread}} = \mu_{\text{spread}} \cdot \left(1 - \frac{\mu_{\text{spread}}}{\text{exposure}}\right) \cdot \left(1 + \frac{\text{exposure} - 1}{1 + \exp(-\sigma_0) \cdot \text{ring}^{-\sigma_1} \cdot \text{exposure}^{-\sigma_2}}\right),$$

gdzie:

- μ_{spread} jest obliczane za pomocą funkcji `mean_spread_function`,
- $\sigma_0, \sigma_1, \sigma_2$ są parametrami z wektora `sigma_spread_parameters`.

Funkcja `haversine_cpp`

Funkcja `haversine_cpp` oblicza odległość między dwoma punktami na powierzchni Ziemi za pomocą wzoru:

$$d = 2r \cdot \arcsin\left(\sqrt{\sin^2\left(\frac{\Delta\phi}{2}\right) + \cos(\phi_1) \cdot \cos(\phi_2) \cdot \sin^2\left(\frac{\Delta\lambda}{2}\right)}\right),$$

gdzie:

- r to promień Ziemi (domyślnie 6378137 metrów),

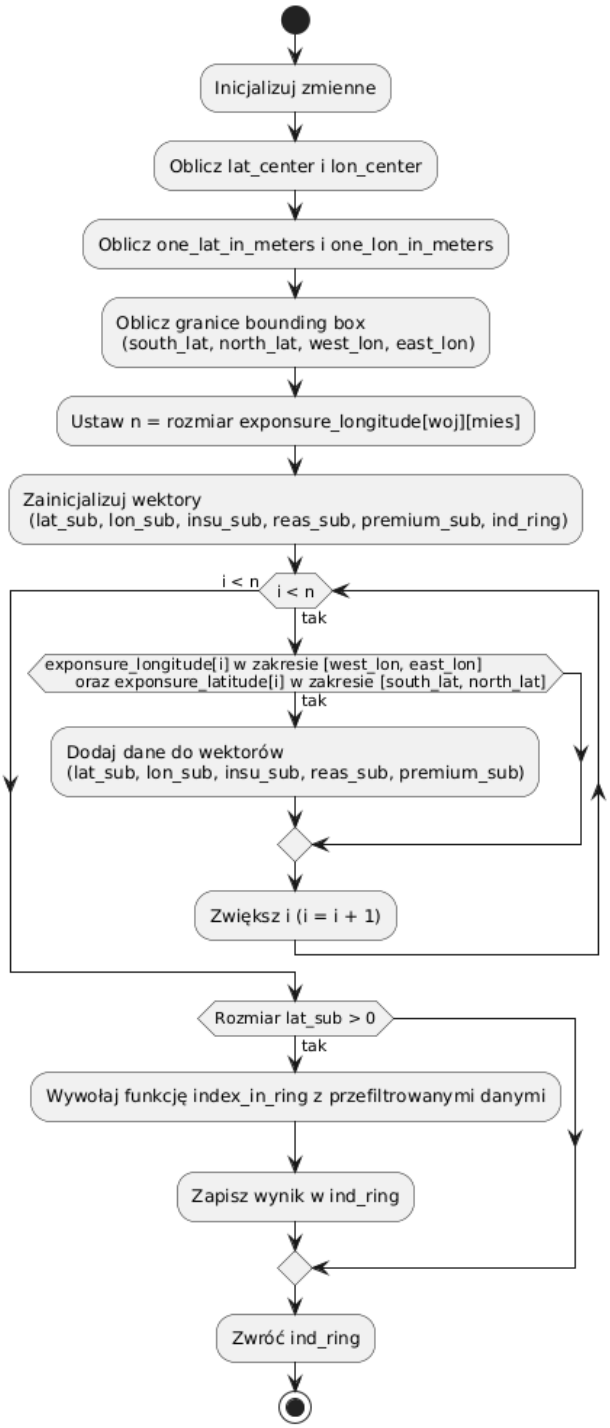
- ϕ_1, ϕ_2 to szerokości geograficzne (w radianach),
- λ_1, λ_2 to długości geograficzne (w radianach),
- $\Delta\phi = \phi_2 - \phi_1$,
- $\Delta\lambda = \lambda_2 - \lambda_1$.

Diagram przepływu funkcji `haversine_loop_cpp_vec`

Na rycinie 2.14 przedstawiono diagram przepływu funkcji `haversine_loop_cpp_vec`, który ilustruje główne kroki i logikę działania tej funkcji.

Opis diagramu

- **Inicjalizacja zmiennych**:
  Funkcja rozpoczyna się od inicjalizacji zmiennych oraz obliczenia współrzędnych centralnych (`lat_center`, `lon_center`) i granic obszaru (`south_lat`, `north_lat`, `west_lon`, `east_lon`).
- **Iteracja po danych**:
  Pętla `while` iteruje przez wszystkie dane w zadanym województwie (`woj`) i miesiącu (`mies`). Dla każdego punktu sprawdzane jest, czy znajduje się w granicach obszaru. Jeśli tak, dane są dodawane do odpowiednich wektorów.
- **Wywołanie funkcji `index_in_ring`**:
  Jeśli istnieją dane w wektorach, są one przetwarzane przez funkcję `index_in_ring`, a wynik jest zapisywany w zmiennej `ind_ring`.
- **Zwrócenie wyniku**:
  Na końcu funkcja zwraca wynik w postaci dwuwymiarowego wektora `ind_ring`.



Rysunek 2.14: Diagram przepływu funkcji `haversine_loop_cpp_vec`.

2.7.11 Funkcja `index_in_ring`

Cel funkcji

Funkcja `index_in_ring` jest kluczowym elementem modelowania przestrzennego rozprzestrzeniania się pożarów. Jej celem jest przypisanie obiektów (np. budynków, pól ubezpieczeniowych) do odpowiednich pierścieni wokół centrum pożaru, na podstawie odległości geograficznej. Pierścienie są definiowane jako zbiory obiektów znajdujących się w określonych przedziałach odległości od centrum.

Opis działania

Funkcja wykorzystuje współrzędne geograficzne (`lat_center`, `lon_center`) centrum pożaru oraz współrzędne obiektów (`lat_sub`, `lon_sub`), aby obliczyć odległość każdego obiektu od centrum. Odległość ta jest obliczana za pomocą funkcji `haversine_cpp`, która implementuje wzór haversine na sferyczną odległość między dwoma punktami na powierzchni Ziemi:

$$d = 2r \cdot \arcsin\left(\sqrt{\sin^2\left(\frac{\Delta\phi}{2}\right) + \cos(\phi_1) \cdot \cos(\phi_2) \cdot \sin^2\left(\frac{\Delta\lambda}{2}\right)}\right),$$

gdzie:

- r to promień Ziemi (domyślnie 6378137 metrów),
- ϕ_1, ϕ_2 to szerokości geograficzne (w radianach),
- λ_1, λ_2 to długości geograficzne (w radianach),
- $\Delta\phi = \phi_2 - \phi_1$,
- $\Delta\lambda = \lambda_2 - \lambda_1$.

Obiekty są następnie przypisywane do jednego z dziewięciu pierścieni, zdefiniowanych jako przedziały odległości:

- Pierścień 1: $0 \leqslant d < 0.005$ km,
- Pierścień 2: $0.005 \leqslant d < 25$ km,
- Pierścień 3: $25 \leqslant d < 50$ km,
- Pierścień 4: $50 \leqslant d < 75$ km,
- Pierścień 5: $75 \leqslant d < 100$ km,

- Pierścień 6: $100 \leqslant d < 125$ km,
- Pierścień 7: $125 \leqslant d < 150$ km,
- Pierścień 8: $150 \leqslant d < 175$ km,
- Pierścień 9: $175 \leqslant d < 200$ km.

Dla każdego pierścienia przechowywane są następujące informacje:

- Odległości obiektów od centrum (`distance_res`),
- Współrzędne geograficzne obiektów (`lat_ring`, `lon_ring`),
- Informacje o ubezpieczeniu (`insu_ring`),
- Informacje o reasekuracji (`reas_ring`),
- Suma ekspozycji (`exponsure_sum_ring`).

Na końcu funkcja wywołuje `index_spread_build`, która modeluje dalsze rozprzestrzenianie się pożaru w każdym pierścieniu.

Znaczenie funkcji

Funkcja `index_in_ring` umożliwia przestrzenną segmentację danych, co jest kluczowe w modelowaniu rozprzestrzeniania się pożarów i analizie ryzyka ubezpieczeniowego. Dzięki niej możliwe jest dokładne określenie, które obiekty znajdują się w zasięgu pożaru i w jakim stopniu mogą być zagrożone.

2.7.12 Funkcja `render_gui()`

Funkcja `render_gui()` odpowiada za rysowanie graficznego interfejsu użytkownika (GUI) w symulatorze pożarów. Wykorzystuje bibliotekę ImGui do tworzenia interaktywnych elementów, takich jak pola tekstowe, przyciski, tabele i paski postępu. Główne zadania funkcji to:

- Pobieranie danych wejściowych od użytkownika (np. rok, ścieżka do folderu z danymi, parametry symulacji).
- Wczytywanie listy ubezpieczycieli z plików w folderze.
- Umożliwienie wyboru ubezpieczycieli do symulacji.
- Konfiguracja parametrów symulacji (np. liczba wątków, liczba symulacji).
- Uruchamianie symulacji w osobnych wątkach.
- Wyświetlanie pasków postępu dla procesu wczytywania danych i symulacji.

Diagram przepływu funkcji

Na poniższym diagramie przedstawiono przepływ danych i interakcji w funkcji `render_gui()`. Diagram przedstawia główne komponenty oraz ich interakcje.

Rysunek 2.15: Diagram przepływu funkcji `render_gui()`

Szczegóły diagramu

Diagram zawiera następujące elementy:

- **User (Użytkownik)**: Reprezentuje użytkownika, który wchodzi w interakcję z interfejsem graficznym.
- **Komponenty GUI**:
  - `ImGui::InputInt/InputText`: Odpowiada za wprowadzanie danych wejściowych, takich jak liczby całkowite i tekst.
  - `ImGui::Button`: Obsługuje przyciski, które wywołują różne akcje.

  - `ImGui::ProgressBar`: Wyświetla pasek postępu podczas wczytywania danych lub symulacji.
  - `ImGui::BeginChild/BeginTable`: Wyświetla sekcje GUI, takie jak listy ubezpieczycieli.

- **Backendowe komponenty**:
  - `std::thread`: Reprezentuje wątki uruchamiane w tle (np. do wczytywania danych lub uruchamiania symulacji).
  - `getFiles()`: Funkcja odpowiedzialna za wczytywanie listy plików z folderu.
  - `std::vector`: Przechowuje dane, takie jak lista ubezpieczycieli.

Interpretacja diagramu

Przepływ działania funkcji można podzielić na następujące etapy:

1. Użytkownik wprowadza dane wejściowe (rok, ścieżka do folderu) za pomocą pól tekstowych (`InputText`).
2. Po kliknięciu przycisku `"Wczytaj listę ubezpieczycieli"` wywoływana jest funkcja `getFiles()`, która wczytuje listę plików z folderu. Dane są przechowywane w wektorze (`std::vector`).
3. Użytkownik może wybrać ubezpieczycieli lub zaznaczyć wszystkich przy użyciu przycisku `"Wybierz wszystkich"`.
4. Parametry symulacji (liczba symulacji, liczba wątków itp.) są konfigurowane za pomocą pól wejściowych.
5. Kliknięcie przycisku `"Wczytaj dane"` uruchamia wątek (`std::thread`) odpowiedzialny za wczytywanie danych.
6. Pasek postępu (`ProgressBar`) aktualizuje się w czasie rzeczywistym w trakcie wczytywania danych.
7. Kliknięcie przycisku `"Włącz symulację"` uruchamia wątek symulacji, a pasek postępu monitoruje jej postęp.

Wnioski

Diagram przepływu funkcji `render_gui()` pokazuje, jak interfejs graficzny i backend współpracują w celu realizacji zadań symulatora pożarów. Dzięki zastosowaniu wątków (`std::thread`) i dynamicznej aktualizacji GUI (np. paski postępu), aplikacja może efektywnie obsługiwać złożone procesy w tle.