POLSKO-JAPOŃSKA AKADEMIA TECHNIK KOMPUTEROWYCH

Wydział Informatyki

Katedra Inżynierii Oprogramowania

Inżynieria Oprogramowania i Baz Danych

Wojciech Bagiński

Nr albumu 29574

Elastyczna biblioteka do tworzenia grafów przetwarzania danych wspomagających analizę zanieczyszczeń powietrza

Praca magisterska
napisana pod kierunkiem

dra inż. Mariusza Trzaski

Warszawa, Czerwiec 2024

Streszczenie

Niniejsza praca omawia praktyczny problem, z jakim borykają się analitycy zajmujący przetwarzaniem danych związanych z modelowaniem zanieczyszczeń powietrza. W dziedzinie tej nie istnieje utrwalony standard narzędziowy, za pomocą którego można całościowo realizować analizy danych atmosferycznych. Badacze korzystają z niespójnego zbioru wielu rozdrobnionych narzędzi i skryptów w różnych językach programowania, łączonych doraźnie w większe części przy użyciu poleceń terminalowych. Niedostateczna dokumentacja, niepopularne lub przestarzałe technologie, konwencje i formaty danych powodują, że czasem uzyskanie nawet mało skomplikowanych koncepcyjnie wyników wiąże się z zaangażowaniem niewspółmiernie dużego zespołu osób o różnych kompetencjach technicznych. W pracy przedstawiono propozycję rozwiązania w postaci biblioteki w języku Julia oraz uzupełniającego pakietu dla programu Node-RED, które wspólnie umożliwiają tworzenie przejrzystych i dających się ponownie wykorzystać wizualnych skryptów do analizy danych atmosferycznych. Rozwiązanie oparte na grafowym paradygmacie przetwarzania danych łączy w sobie dużą elastyczność i zakres stosowania, a jednocześnie może być zrozumiałe także dla mniej technicznie zorientowanych osób biorących udział w procesach analiz. Autor jest przekonany, że zaproponowanie takiego rozwiązania dla danych powstających w wyniku modelowania atmosfery przyniesie korzyści w postaci łatwości tworzenia obliczeń i analiz, usprawnienia dzielenia się skryptami między członkami zespołów, oraz uzyskania przejrzystości powtarzalnych procesów.

Słowa kluczowe: przetwarzanie oparte na przepływie danych, algorytm Kahna, DAG, skierowany graf acykliczny, sortowanie topologiczne

Spis treści

1. Wstęp

Modelowanie zanieczyszczeń atmosferycznych to interdyscyplinarna dziedzina, łącząca w sobie nauki środowiskowe, meteorologię, chemię, fizykę, inżynierię, informatykę i matematykę. Leży ona na styku tych dziedzin, umożliwiając precyzyjne prognozowanie i ocenę wpływu emisji zanieczyszczeń na jakość powietrza oraz zdrowie ludzi. Ma ona w dzisiejszych czasach niebagatelny wpływ na planowanie przestrzenne, rozwój miast i ogółem politykę środowiskową państw.

Tworzenie prognoz chemii atmosfery jest skomplikowanym numerycznie procesem. Najczęściej do obliczeń wykorzystuje się farmy bardzo wydajnych komputerów, a obliczenia są wykonywane współbieżnie. Używane algorytmy są nie tylko wysoce zoptymalizowane z punktu widzenia obliczeniowego, ale też na poziomie merytorycznym wynikają ze zbiorczej wiedzy całych dekad badań naukowych, popartej oficjalnymi publikacjami w periodykach naukowych. Modelowanie chemii atmosfery symuluje procesy fizyczne i mechaniczne atmosfery oraz setki reakcji chemicznych zachodzących w różnych warstwach atmosfery.

Uzyskane prognozy muszą być poddane weryfikacji i analizie, w wyniku których można wysnuć wnioski na temat jakości powietrza w danym regionie, a także ocenić wpływ różnych czynników na jakość powietrza. Analiza taka jest procesem znacznie mniej złożonym niż modelowanie, ale wymaga przetwarzania wielu różnych danych, zarówno obserwacyjnych, jak i symulacyjnych.

Eksperymentalny charakter modelowania atmosfery sprawia, że analizy nie są jednorazowe, a często powtarzane w różnych konfiguracjach, z różnymi parametrami wejściowymi i zbiorami danych. Trudność w predefiniowaniu dokładnych procesów i duże zróżnicowanie zespołów pod względem biegłości w różnych technologiach, powoduje, że nie istnieją dedykowane zamknięte narzędzia spełniające wszystkie potrzeby tych analiz. Ten deficyt narzędzi skazuje zespoły badawcze na korzystanie z połączeń wielu rozdrobnionych programów i skryptów, z których część niejednokrotnie bywa pisana w przestarzałych technologiach, używanych z powodów historycznych, a część powstaje doraźnie bez wystarczającej dbałości o jakość kodu i jego dokumentację.

Z drugiej strony charakter środowiska, wykonywanych prac, używanych systemów operacyjnych i ogólne preferencje, nakazujące korzystanie głównie z narzędzi terminalowych, a także niewystarczające środki zarówno materialne, jak i kadrowe, sprawiają, że zespoły badawcze nie są zainteresowane ponoszeniem kosztów finansowych i czasowych na budowę zintegrowanych narzędzi o rozbudowanej funkcjonalności i opartych na najlepszych praktykach programistycznych.

Powstaje paradoks, w którym narzędzia do analizy danych, niezbędne do prowadzenia badań i tworzenia raportów, stanowią dosyć chaotyczny i trudny w obsłudze dla mniej zorientowanych technicznie osób zbiór programów i skryptów.

1.1 Cel pracy

Celem niniejszej pracy jest opracowanie uniwersalnego podejścia do przetwarzania danych związanych z analizą zanieczyszczeń atmosferycznych. Rozwiązanie oparte na paradygmacie grafowym pozwala na organizowanie i prowadzenie analiz w sposób wizualny, elastyczny i możliwie łatwy również dla osób nie będących programistami.

1.2 Rezultat pracy

Rezultatem pracy jest wskazanie i omówienie propozycji elastycznego rozwiązania pozwalającego zespołom badawczym na wizualne układanie analiz, od etapu pobrania danych wejściowych, przez dokonanie przekształceń, do zapisania wyników, w formie grafów przetwarzania, a następnie uruchamianie tych grafów w programie obliczeniowym. Zaproponowane rozwiązanie zwiększa przejrzystość procesów analitycznych, umożliwia łatwe dzielenia się pracą między członkami zespołu oraz zachęca do ponownego wykorzystywania opracowanych analiz.

1.3 Struktura pracy

Rozdział 1 zawiera wprowadzenie do tematu pracy, opisuje cele i zarys koncepcji.

Rozdział 2 omawia istniejące narzędzia, używane w analizach danych związanych z zanieczyszczeniami atmosfery.

Rozdział 3 opisuje propozycję rozwiązania.

Rozdział 4 zawiera opis technologii użytych w pracy.

Rozdział 5 omawia teoretyczne aspekty budowy biblioteki.

Rozdziały 6 i 7 zawierają szczegóły implementacyjne prototypu.

Rozdział 8 prezentuje praktyczne przykłady zastosowań.

Rozdział 9 wymienia wady i zalety rozwiązania, przedstawia potencjalne kierunki rozwoju i podsumowuje pracę.

2. Przegląd istniejących narzędzi

Środowisko narzędziowe dziedziny modelowania zanieczyszczeń atmosferycznych jest bardzo rozdrobnione. Stosunkowo zamknięty charakter prac i ograniczona współpraca między instytutami w zakresie tworzenia oprogramowania, powoduje, że właściwie nie istnieją ogólnobranżowe standardy narzędziowe. Każdy zespół badawczy tworzy własne narzędzia, które są dostosowane do specyficznych potrzeb. Poza kilkoma wyjątkami, takimi jak format NetCDF, nie ma jednolitych standardów.

W poniższych podrozdziałach opisano najpopularniejsze narzędzia wykorzystywane w dziedzinie modelowania zanieczyszczeń atmosferycznych.

2.1 NetCDF

NetCDF (ang. *Network Common Data Form*) to zestaw bibliotek programistycznych i formatów danych niezależnych od architektury komputerowej, do obsługi macierzowych danych naukowych używanych w dziedzinach takich jak meteorologia, oceanografia, geofizyka, bioinformatyka, inżynieria środowiska, itp. Projekt NetCDF jest rozwijany i utrzymywany przez jednostkę Unidata Program Center będącą częścią Zrzeszenia Uniwersyteckiego ds. Badań Atmosferycznych (UCAR), oraz przez amerykańskie Narodowe Centrum Badań Atmosferycznych (NCAR). Oficjalnie są utrzymywane interfejsy programistyczne NetCDF dla języków C, C++, Java i Fortran.

Dane w formacie NetCDF mają następujące cechy [1]:

- są samoopisujące — plik NetCDF zawiera informacje o danych, które przechowuje;
- są przenośne — plik NetCDF jest poprawnie interpretowany przez komputery o różnych systemach operacyjnych i sposobach przechowywania liczb całkowitych, znaków i liczb zmiennoprzecinkowych;
- są skalowalne — uzyskiwanie dostępu do wycinków wielkich zbiorów danych przez interfejsy NetCDF jest bardzo efektywne zarówno lokalnie, jak i zdalnie;
- są rozszerzalne — do odpowiednio skonfigurowanego pliku NetCDF można doklejać dane bez konieczności kopiowania zbiorów ani redefiniowania ich struktur;
- są dostępne — do tego samego pliku NetCDF może równocześnie uzyskiwać dostęp jeden zapisujący i wielu odczytujących;
- są archiwizowalne — gwarantuje się wsparcie dla wszystkich dotychczasowych wersji formatu we wszystkich bieżących i przyszłych wersjach oprogramowania.

Format NetCDF jest jednym z podstawowych i najpowszechniejszych formatów wymiany danych w dziedzinach nauk przyrodniczych związanych z atmosferą.

2.2 NCO

Zbiór narzędzi NCO (ang. *NetCDF Operators*) obejmuje kilkanaście samodzielnych programów wiersza poleceń. Narzędzia te przyjmują jako argumenty pliki NetCDF, HDF lub DAP, a następnie wykonują na nich wybrane przekształcenia, np. operacje macierzowe, obliczają statystyki, wyświetlają

dane, ekstrahują podzbiory, manipulują metadanymi lub wykonują interpolacje geograficzne. Następnie wyniki mogą być zapisane do plików w formatach tekstowych, binarnych lub do kolejnych plików NetCDF. Zbiór NCO wspomaga analizę zarówno danych naukowych bez struktury przestrzennej, jak i danych osadzonych na siatce geograficznej.

Biblioteka NCO składa się aktualnie z następujących narzędzi, w kolejności alfabetycznej [72]:

- `ncap2` — procesor arytmetyczny do obliczeń na plikach NetCDF;
- `ncatted` — edytor atrybutów plików NetCDF;
- `ncbo` — operator dwuargumentowy na plikach NetCDF;
- `ncea` — obliczenia statystyczne na plikach NetCDF;
- `ncchecker` — sprawdzanie zgodności plików NetCDF z konwencjami;
- `ncclimo` — generator klimatologii;
- `nces` — statystyki wiązek zbiorów danych;
- `ncecat` — łączenie wiązek zbiorów danych;
- `ncflint` — interpolacje;
- `ncks` — wielozadaniowy program do ekstrakcji i manipulacji podzbiorów zbiorów danych w plikach NetCDF;
- `ncpdq` — operacje na wymiarach zbiorów danych;
- `ncra` — agregacja zbiorów danych;
- `ncrcat` — łączenie zbiorów danych;
- `ncremap` — przekształcenia związane z siatkami geograficznymi;
- `ncrename` — zmiana nazw zmiennych, atrybutów i wymiarów;
- `ncwa` — statystyki z wagami, maskami i normalizacją.

Na listingu 2.1 znajduje się przykładowy skrypt w języku `bash`, wykorzystujący narzędzia NCO, który wybiera z 5-letniego zbioru danych miesięcznych zapisanych w plikach NetCDF uśrednione miesięczne anomalie (różnice) względem średniej pięcioletniej, dla każdego z 12 miesięcy [72].

Listing 2.1: Przykładowy skrypt w języku `bash` wykorzystujący narzędzia NCO

```
ncwa -a time 8501_8912.nc 8589.nc
ncbo 8501_8912.nc 8589.nc t_anm_8501_8912.nc
for idx in {1..12}; do
  idx=‘printf "%02d" ${idx}‘
  ncks -F -d time,${idx},,12 t_anm_8501_8912.nc foo.${idx}
  ncra foo.${idx} t_anm_8589_${idx}.nc
done
```

Zbiór narzędzi NCO daje ogromne możliwości manipulacji danymi i ich analizy, jednak wymaga od użytkownika znajomości języka powłoki `bash` i biegłości w używaniu poszczególnych komend NCO. W przypadku bardziej złożonych operacji skrypty mogą być trudne do zrozumienia i modyfikowania.

2.3 CDO

Biblioteka CDO (ang. *Climate Data Operators*) to zestaw ponad 600 operatorów do analizy danych klimatologicznych i meteorologicznych zapisanych w formacie NetCDF i GRIB. Pakiet CDO jest rozwijany przez Max Planck Institut für Meteorologie w Hamburgu w Niemczech. Narzędzia CDO, jak większość programów tego typu, mogą być używane do analizy nie tylko danych pogodowych, ale też wszelkich danych osadzonych na siatkach geograficznych, w tym danych związanych z modelowaniem zanieczyszczeń powietrza.

Listing 2.2 przedstawia kilka oryginalnych [56] przykładów użycia programu `cdo` do przekształceń na plikach NetCDF.

Listing 2.2: Wygląd przykładowych poleceń `cdo`

```
# extract the northern hemisphere
cdo -sellonlatbox,-180.0,180.0,0.0,90.0  infile outfile

# add coordinate variables if missing
cdo -f nc -setgrid,r360x180 infile outfile

# retrieve frost days
cdo -f nc -timsum -expr,
't2min = ((t2min < 0.0)) ? 1.0 : (t2min/0.0)'
infile outfile
```

2.4 CDAT

Zestaw narzędzi CDAT (ang. *Climate Data Analysis Tools*) to oprogramowanie utworzone przez laboratorium Lawrence Livermore National Laboratory (LLNL) w celu ułatwienia analizy danych klimatycznych. Nie umożliwia ono stricte analizy danych zanieczyszczeń powietrza, ale są to dziedziny pokrewne, a dane meteorologiczne są istotne z punktu widzenia modelowania zanieczyszczeń atmosfery.

Na stronie CDAT w 2023 roku została opublikowana wzmianka, że około końca roku 2023 prace zostaną ograniczone jedynie do podtrzymania istniejących funkcji i nie będą dodawane nowe funkcjonalności. Oznacza to, że oprogramowanie CDAT nie będzie już rozwijane w dotychczasowej formie [53].

Wysiłki w zakresie rozwoju oprogramowania CDAT zostały skierowane na nowy projekt pod nazwą xCDAT. Jest to rozszerzenie do biblioteki `xarray` w języku Python, które ma pełnić rolę następcy CDAT, ale być bardziej zgodne z ekosystemem narzędzi Pythona (zob. 2.10).

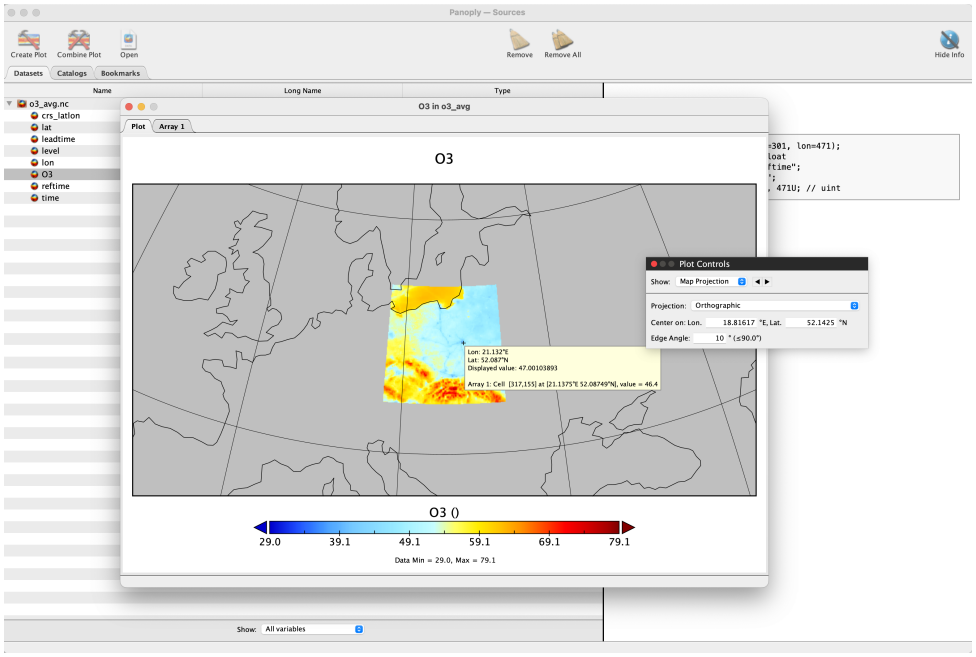
2.5 Panoply

Panoply to oprogramowanie do wizualizacji danych naukowych zapisanych w formacie NetCDF, HDF i GRIB. Jest to program rozwijany przez dra Roberta B. Schmunka z Instytutu Goddarda Studiów Kosmicznych będącego częścią Narodowej Agencji Aeronautyki i Przestrzeni Kosmicznej Stanów Zjednoczonych (NASA). Panoply jest napisany w języku Java i jest dostępny na wszystkie wiodące platformy komputerowe, jak Windows, macOS i Linux [65].

Program Panoply umożliwia wizualizację danych w formie map i wykresów, pozwala też na tworzenie animacji. Wśród funkcjonalności jest wybór różnych projekcji, skal, jednostek, kolorów, a także podstawowe operacje arytmetyczne na warstwach danych. Panoply pozwala na eksport wygenerowanych wizualizacji

do kilku popularnych formatów graficznych.

Na rysunku 2.1 przedstawiono wygląd interfejsu programu Panoply.



Rysunek 2.1: Interfejs programu Panoply, źródło: opracowanie własne

2.6 FSTD (RPN)

FSTD (RPN) jest formatem danych rozwijanym od lat 90-tych XX w. w Kanadyjskim Narodowym Centrum Meteorologicznym (CMC). Jego nazwa pochodzi od skrótu francuskiego słowa *fichier standard* i jest używana zamiennie ze skrótem RPN, od nazwy zakładu Badań Prognoz Numerycznych (fr. *Recherche en Prévision Numérique*). Jest to format plików używany do przechowywania danych meteorologicznych i atmosferycznych, w tym danych z modeli numerycznych.

Format ten jest mało popularny na świecie poza Kanadą, jednak ze względu na bogatą historię współpracy polskich naukowców z instytutami kanadyjskimi, jest on używany w Polsce, m.in. w Zakładzie Modelowania Atmosfery i Klimatu, Instytutu Ochrony Środowiska – Państwowego Instytutu Badawczego.

Obsługę tego formatu oferuje tylko oryginalne pochodzące z CMC oprogramowanie `librmn`, dostępne wyłącznie na platformy UNIX/Linux [31], z interfejsami dla języków Fortran i C.

Rysunek 2.2 przedstawia działanie jednego z narzędzi tej biblioteki (`voir`) służącego do wyświetlania zawartości plików FSTD (RPN).

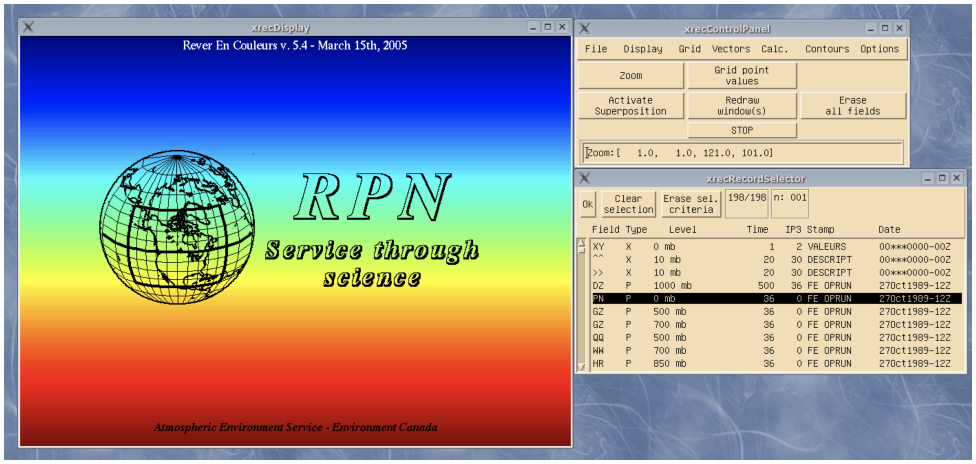
```
*****************************************************************************************
*                                                                                       *
*          VOIR                                                          V98.19         *
*                                                                                       *
*          RMNLIB  -  Release:  (Rev ) (Linux_pgi611) Fri Jan  9 15:40:53 EST 2009      *
*                                                                                       *
*                                                                                       *
*          Thu Jun 20 16:11:26 2024                                                     *
*                                                                                       *
*          BEGIN  EXECUTION                                                             *
*                                                                                       *
*****************************************************************************************

     NOMV TV ETIQUETTE       NI    NJ    NK (DATE-O  h m s)         IP1    IP2   IP3     DEET     NPAS  DTY  G   IG1   IG2    IG3   IG4

 0- >>   X  V2C_2023        536     1     1 20230325 000000          20    258     0        0        0  E32  E  1420     0  17472 17472
 1- ^^   X  V2C_2023          1   478     1 20230325 000000          20    258     0        0        0  E32  E  1420     0  17472 17472
 2- HY   X  V2C_2023          1     1     1 20230325 000000    42053040      0     0      200       18  E32  X   800  1000      0     0
 3- O3   P  V2C_2023        536   478     1 20230325 000000    26314400      1     0      200       18  E32  Z    20   258      0     0
 4- NO   P  V2C_2023        536   478     1 20230325 000000    26314400      1     0      200       18  E32  Z    20   258      0     0
 5- NO2  P  V2C_2023        536   478     1 20230325 000000    26314400      1     0      200       18  E32  Z    20   258      0     0
 6- CO   P  V2C_2023        536   478     1 20230325 000000    26314400      1     0      200       18  E32  Z    20   258      0     0
 7- SO2  P  V2C_2023        536   478     1 20230325 000000    26314400      1     0      200       18  E32  Z    20   258      0     0
 8- C3H8 P  V2C_2023        536   478     1 20230325 000000    26314400      1     0      200       18  E32  Z    20   258      0     0
 9- ALKA P  V2C_2023        536   478     1 20230325 000000    26314400      1     0      200       18  E32  Z    20   258      0     0
10- PT25 P  V2C_2023        536   478     1 20230325 000000    26314400      1     0      200       18  E32  Z    20   258      0     0
11- PTCO P  V2C_2023        536   478     1 20230325 000000    26314400      1     0      200       18  E32  Z    20   258      0     0
12- PH25 P  V2C_2023        536   478     1 20230325 000000    26314400      1     0      200       18  E32  Z    20   258      0     0
13- PHCO P  V2C_2023        536   478     1 20230325 000000    26314400      1     0      200       18  E32  Z    20   258      0     0
14- PS25 P  V2C_2023        536   478     1 20230325 000000    26314400      1     0      200       18  E32  Z    20   258      0     0
15- PSCO P  V2C_2023        536   478     1 20230325 000000    26314400      1     0      200       18  E32  Z    20   258      0     0
16- SM25 P  V2C_2023        536   478     1 20230325 000000    26314400      1     0      200       18  E32  Z    20   258      0     0
17- SMCO P  V2C_2023        536   478     1 20230325 000000    26314400      1     0      200       18  E32  Z    20   258      0     0
18- BAP1 P  V2C_2023        536   478     1 20230325 000000    26314400      1     0      200       18  E32  Z    20   258      0     0
19- BAP2 P  V2C_2023        536   478     1 20230325 000000    26314400      1     0      200       18  E32  Z    20   258      0     0
20- NH3  P  V2C_2023        536   478     1 20230325 000000    26314400      1     0      200       18  E32  Z    20   258      0     0
21- NH4  P  V2C_2023        536   478     1 20230325 000000    26314400      1     0      200       18  E32  Z    20   258      0     0
22- PMAS P  V2C_2023        536   478     1 20230325 000000    26314400      1     0      200       18  E32  Z    20   258      0     0
23- PMCD P  V2C_2023        536   478     1 20230325 000000    26314400      1     0      200       18  E32  Z    20   258      0     0
24- PMNI P  V2C_2023        536   478     1 20230325 000000    26314400      1     0      200       18  E32  Z    20   258      0     0
25- PMPB P  V2C_2023        536   478     1 20230325 000000    26314400      1     0      200       18  E32  Z    20   258      0     0
26- PL25 P  V2C_2023        536   478     1 20230325 000000    26314400      1     0      200       18  E32  Z    20   258      0     0
27- PLCO P  V2C_2023        536   478     1 20230325 000000    26314400      1     0      200       18  E32  Z    20   258      0     0
28- EU25 P  V2C_2023        536   478     1 20230325 000000    26314400      1     0      200       18  E32  Z    20   258      0     0
29- EUCO P  V2C_2023        536   478     1 20230325 000000    26314400      1     0      200       18  E32  Z    20   258      0     0
```

Rysunek 2.2: Program `voir` do wyświetlania zawartości plików FSTD (RPN), źródło: opracowanie własne

2.7 xrec

XREC to program wizualizacyjny używany do wyświetlania dwuwymiarowych pól atmosferycznych przechowywanych w plikach w formacie FSTD (RPN) (zobacz 2.6). Program ten został opracowany przez dział informatyki zakładu prognoz numerycznych w kanadyjskim Narodowym Centrum Meteorologicznym (CMC), aby zapewnić naukowcom efektywne narzędzie do przeglądania obszernych zbiorów danych produkowanych przez modele numeryczne i analizy.

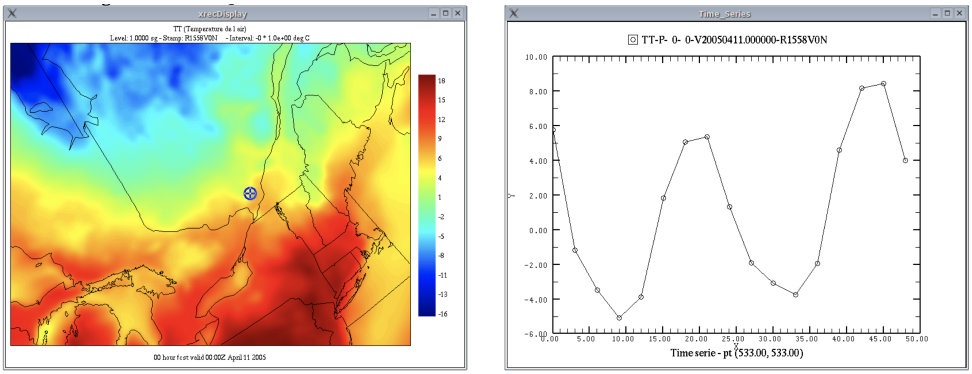
Na rysunku 2.3 przedstawiono ekran początkowy programu `xrec`.



Rysunek 2.3: Ekran początkowy programu `xrec`, źródło: opracowanie własne

Program xrec jest jedynym dostępnym programem mogącym bezpośrednio wyświetlać dane zawarte w plikach FSTD (RPN) w postaci graficznej, bez dodatkowych zabiegów związanych z konwersją formatów. Ograniczeniem jest to, że program, podobnie jak cała biblioteka do obsługi formatu, działa tylko w systemach UNIX/Linux.

Na rysunku 2.4 przedstawiono przykład wizualizacji danych meteorologicznych w programie `xrec`.



Rysunek 2.4: Przykład wizualizacji danych meteorologicznych w programie `xrec`, źródło: [7]

2.8 Fortran

W powszechnej świadomości język Fortran, którego początki sięgają roku 1957, jest przestarzałym i nieużywanym już językiem programowania. Jest to jednak mylne przeświadczenie. W wielu dziedzinach naukowych, również w obszarze modelowania zanieczyszczeń atmosferycznych, język Fortran jest właściwie do dzisiaj najważniejszym i najczęściej używanym językiem programowania. Wiele modeli numerycznych, w tym modele atmosferyczne, jest napisanych właśnie w tym języku.

Wśród zalet języka Fortran można wymienić:

- szybkość — język Fortran jest językiem niskopoziomowym, statycznie typowanym, co pozwala na budowę efektywnych kompilowanych programów;
- prostota — język Fortran jest językiem o prostej składni, szczególnie w przypadku operatorów matematycznych na macierzach i wektorach;

- wsparcie dla obliczeń numerycznych — język Fortran ma bogaty zbiór wydajnych bibliotek matematycznych, co pozwala na łatwe i szybkie pisanie skomplikowanych obliczeń numerycznych;
- wsparcie dla wielowymiarowych tablic — język Fortran ma wbudowane wsparcie dla wielowymiarowych tablic, co pozwala na łatwe operacje na danych przestrzennych;
- stabilność i niezawodność — język Fortran jest językiem o długiej historii, co sprawia, że został wnikliwie przetestowany i jest bardzo stabilny;
- wbudowane wsparcie dla równoległego programowania.

Wadami języka Fortran są:

- skomplikowana obsługa napisów i operacji na napisach;
- duże różnice między wersjami języka;
- brak wsparcia dla programowania obiektowego w starszych wersjach języka;
- skomplikowana obsługa wejścia/wyjścia;
- trudne debugowanie ze względu na nieczytelne komunikaty błędów kompilatora;
- brak wygodnych narzędzi do zarządzania zależnościami i budowania programów.

Mimo małej obecnie powszechnej popularności tego języka, nie zanosi się, aby miał on zostać zastąpiony przez inne języki programowania w dziedzinach naukowych, w których jest używany.

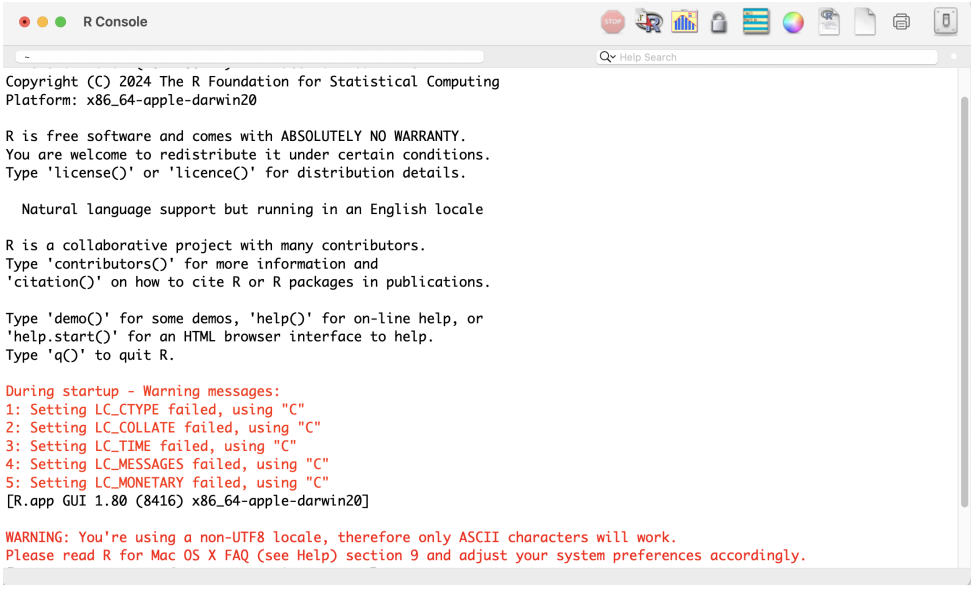
Jako jedyny wyjątek spełniający na podobnym poziomie wymogi związane z programowaniem naukowym, a jednocześnie oferującym nowoczesne rozwiązania, można wymienić język Julia (zob. 4.1).

2.9 Język R

Język R jest uznanym standardem w dziedzinie analizy danych i statystyki. Do niedawna narzędzie to dominowało na tych polach i było chętnie wykorzystywane również w niektórych analizach dotyczących modelowania zanieczyszczeń atmosferycznych. W ostatnich latach jednak, z powodu szybkiego rozwoju języka Python i jego bibliotek, R zaczął tracić na znaczeniu w tej dziedzinie.

Jego zaletą jest bogata biblioteka pakietów do analizy danych. Trudność stanowi zaś niekonwencjonalna składnia języka, która wymaga od użytkownika dużego nakładu pracy, aby móc efektywnie korzystać z narzędzi dostępnych w języku R. Język R uchodzi też za wolny i ma problemy z wydajnością w przypadku dużych zbiorów danych.

Rysunek 2.5 przedstawia interfejs programu RConsole dla języka R.



Rysunek 2.5: Program RConsole dla języka R, źródło: opracowanie własne

2.10 Język Python i jego biblioteki

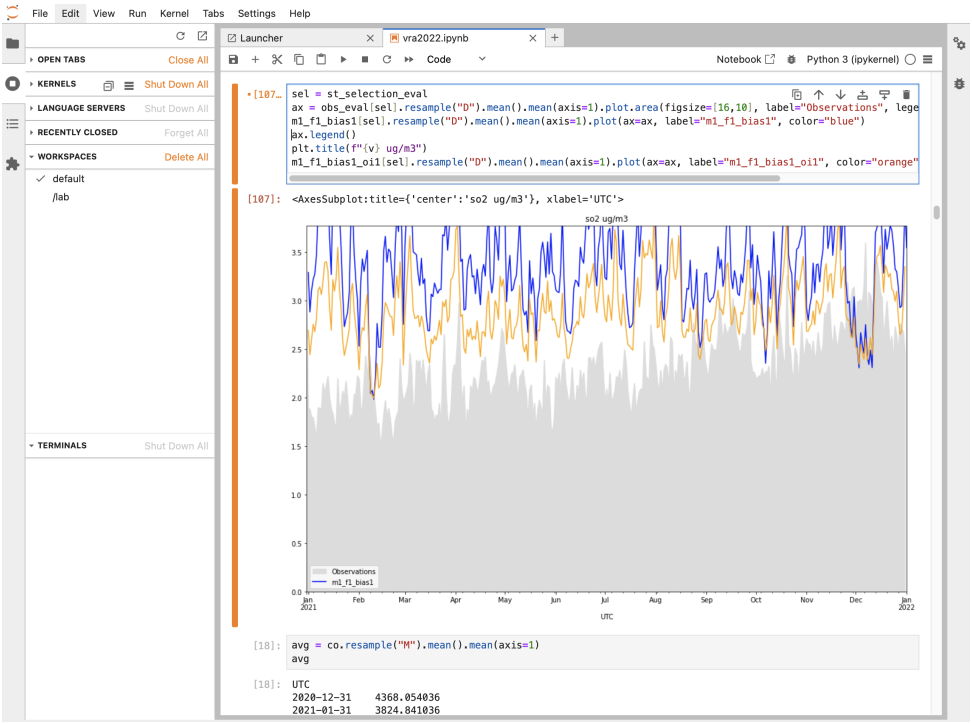
Spośród ogólnych narzędzi do analizy danych największą popularność w ostatnich latach zyskał język Python wraz z licznymi bibliotekami naukowymi. Środowisko języka jest darmowe, dojrzałe i ma duży ekosystem bibliotek oraz znaczącą bazę użytkowników.

Od czasu pojawienia się aplikacji Jupyter Notebook i Jupyter Lab, łatwiejszym stało się dzielenie swoją pracą z innymi inżynierami [59]. Katalizatorem zmian stało się też coraz większe zainteresowanie wykorzystaniem technik sztucznej inteligencji, które są w języku Python łatwo dostępne za pomocą darmowych, wysoce rozwiniętych bibliotek takich, jak TensorFlow [38], Sci-Kit Learn [28] czy PyTorch [32]. Przykładowe użycie środowiska JupyterLab do analizy danych atmosferycznych pokazano na rysunku 2.6.

Dodatkową zaletą jest możliwość wizualizacji wyników za pomocą biblioteki Matplotlib [42]. Istnieje też dobre wsparcie dla danych geograficznych poprzez biblioteki Cartopy [62], Geopandas [4] i PyProj [71]. Dane wielowymiarowe pozwala łatwo i stosunkowo wydajnie obsługiwać biblioteka Xarray [29].

Wadą Pythona w przypadku użycia do danych atmosferycznych jest to, że jest on językiem interpretowanym i w niektórych obszarach obliczeniowych działa wolno. Biblioteka numeryczna Numpy [27], będąca rozszerzeniem napisanym w języku C, jest powszechnie stosowanym rozwiązaniem dla tej bolączki i przyspiesza niektóre obliczenia, ale wymaga poznania niekonwencjonalnej, nie zawsze intuicyjnej, składni dla danych macierzowych.

Dodatkową trudnością jest nastręczający wiele problemów system pakietów języka Python. Nierzadko występują konflikty wersji pakietów używanych przez narzędzia wewnętrzne. Ponadto język Python nie oferuje, przynajmniej w czasie, kiedy powstaje ta praca, pełnego przetwarzania równoległego ze względu na mechanizm GIL (ang. *Global Interpreter Lock*) [36].



Rysunek 2.6: Przykładowe użycie środowiska JupyterLab do analizy danych atmosferycznych, źródło: opracowanie własne

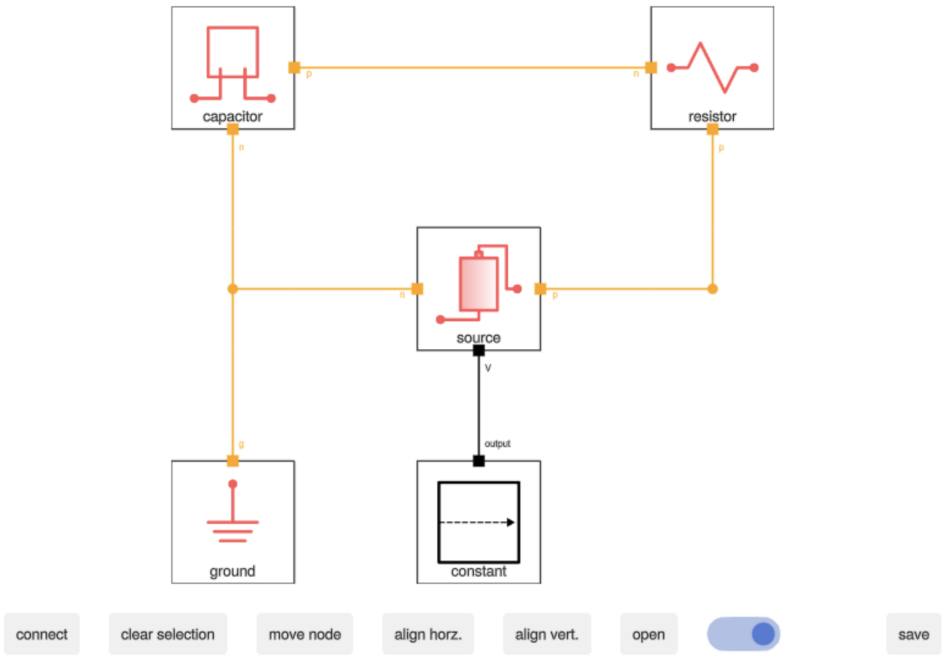
2.11 Inne rozwiązania

W ostatnich latach można zauważyć, że mimo wyjątkowej popularności języka Python w dziedzinie analizy danych, język Julia zdobywa coraz większe uznanie i zyskuje na znaczeniu w środowisku naukowym. Choć Python pozostaje nadal dominującym językiem w tej dziedzinie, to Julia zdobywa na popularności dzięki swojej wydajności, elastyczności, natywnemu wsparciu przetwarzania równoległego, niezwykle łatwej integracji z gotowymi bibliotekami w językach Fortran, C i Python i wciąż tak samo dużej przejrzystości i prostocie kodu.

Jest to obiecująca alternatywa, która przyciąga uwagę badaczy i profesjonalistów zajmujących się analizą danych, a także naukowców z innych dziedzin, poszukujących efektywnego narzędzia do swoich obliczeń i eksperymentów. Więcej informacji na temat języka Julia zawiera podrozdział 4.1.

W ramach przeglądu dostępnych narzędzi, spośród bibliotek oferujących grafowe podejście do przetwarzania danych, natrafiono jedynie na pakiet ModelingToolkitDesigner [6]. Program ten jest we wczesnej fazie rozwoju, a jego zastosowanie jest zawężone do współpracy z konkretną biblioteką — jest to program pomocniczy do wizualizowania połączeń w modelach utworzonych w pakiecie ModelingToolkit.jl [63]. Interfejs pakietu ModelingToolkitDesigner został przedstawiony na rysunku 2.7.

Pozostałe pakiety Julii, które mają związek z grafami, służą do ogólnych rozważań i badania własności grafów oraz analizowania sieci. Takimi pakietami są np. Graphs.jl [48], LightGraphs.jl [5].



Rysunek 2.7: Interfejs programu ModelingToolkitDesigner, źródło: [6]

3. Propozycja rozwiązania

Proponowane rozwiązanie problemu związanego z analizą danych atmosferycznych polega nie tyle na zastąpieniu całego, budowanego przyrostowo, środowiska narzędziowego opisanego w rozdziale 2, co zaproponowaniu biblioteki, za pomocą której można nie tylko wykonywać równoważne analizy, ale też w razie potrzeby korzystać z dotychczasowych skryptów i programów obudowanych i dodanych do biblioteki w postaci operatorów.

W tym celu utworzono następujący zbiór narzędzi:

- pakiet do programu Node-RED umożliwiający wizualne komponowanie procesu analizy danych związanych z zanieczyszczeniami powietrza w postaci grafu i zapisywanie go w formie pliku JSON;
- biblioteka w języku Julia pozwalająca na odczytywanie pliku powstałego w programie Node-RED i budowanie wewnętrznej grafowej reprezentacji procesu oraz przetwarzanie tej reprezentacji w celu uzyskania wyniku końcowego;
- pomocniczy program uruchomieniowy do wsadowego uruchamiania grafów.

3.1 Założenia funkcjonalności

Z punktu widzenia potrzeb zespołów analizujących dane atmosferyczne rozwiązanie powinno spełniać następujące warunki:

- dawać duże możliwości tworzenia grafów dla inżynierów i jednocześnie umożliwiać modyfikacje grafów przez osoby nie będące programistami;
- dawać możliwość budowania w jednym miejscu całych analiz obejmujących etap wczytywania danych, przetwarzania i generowania wyników, włącznie z wizualizacjami;
- zapewniać możliwość wyświetlania i zapisywania skryptów;
- oferować możliwość łatwego dzielenia się skryptami w zespole.

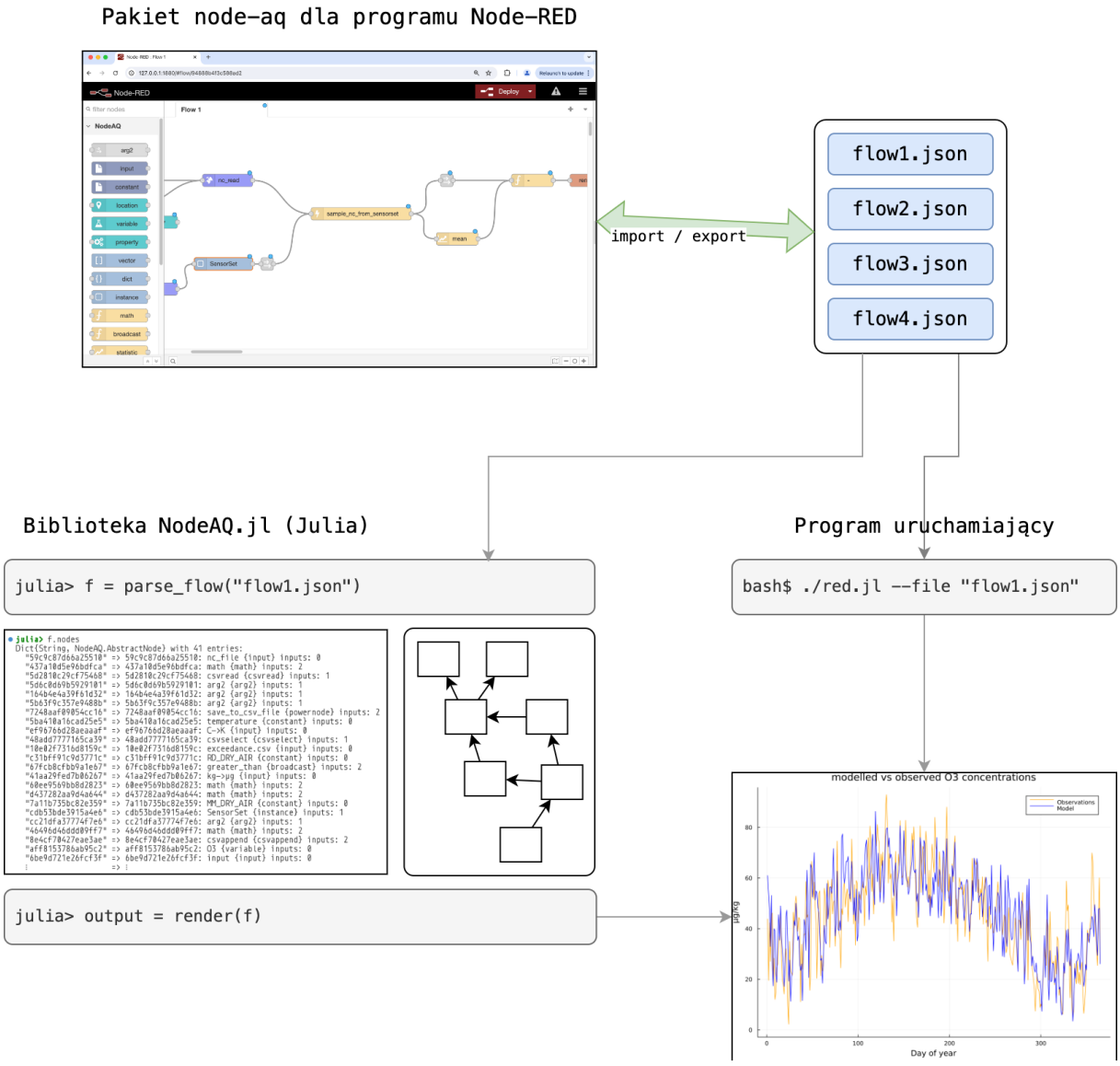
3.2 Założenia techniczne

Z punktu widzenia technicznego, rozwiązanie powinno spełniać następujące założenia:

- być napisane w języku Julia, który jest językiem wysokiego poziomu, ale jednocześnie bardzo wydajnym, co pozwala na szybkie prototypowanie i łatwe optymalizowanie kodu;
- aspekt wizualny powinien być zrealizowany w powszechnie używanym i darmowym programie, takim jak Node-RED, który jest dostępny na wszystkich popularnych systemach operacyjnych;
- umożliwiać uruchamianie w środowisku terminalowym, głównie na systemach UNIX-owych;
- nie ograniczać operacji, jakie można przeprowadzić na danych, i umożliwiać dodawanie nowych operacji w postaci funkcji języka Julia lub opakowanych w nie funkcji zewnętrznych.

3.3 Struktura rozwiązania

Rysunek 3.1 przedstawia schematycznie strukturę rozwiązania. Wszystkie elementy są ze sobą połączone w taki sposób, żeby możliwe było tworzenie grafów w programie Node-RED, zapisywanie ich w formie pliku JSON, a następnie odczytywanie tego pliku w bibliotece w języku Julia, która przetwarza graf na wynik końcowy. Uruchamianie może się odbywać albo w interakcyjnym środowisku REPL Julii albo za pomocą programu uruchomieniowego.



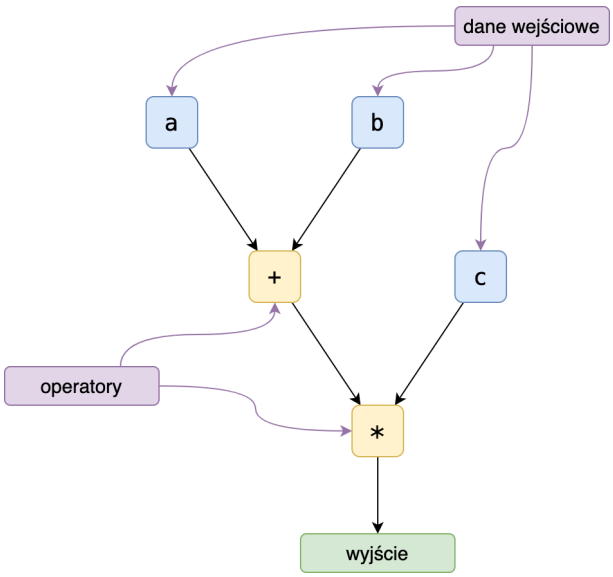
Rysunek 3.1: Struktura prototypu, źródło: opracowanie własne

3.4 Architektura rozwiązania

Rozwiązanie jest przykładem zastosowania paradygmatu obliczeń opartych na przepływie danych (ang. *Data Flow Computing*).

Kluczową ideą stojącą za tym podejściem jest to, że program jest reprezentowany jako graf, w którym wierzchołki odpowiadają operacjom, a krawędzie ustanawiają zależności danych.

Weźmy na przykład funkcję obliczającą wartość wyrażenia $(a+b)*c$, dla zmiennych a, b i c. Możemy reprezentować to obliczenie jako graf, jak na rysunku 3.2.



Rysunek 3.2: Graf przepływu danych dla wyrażenia $(a+b)*c$, źródło: opracowanie własne

W grafie tym niebieskie prostokąty reprezentują dane wejściowe, zaś żółte — funkcje (dodawanie i mnożenie). Strzałki oznaczają zależności danych — argumenty funkcji. Struktura grafu wyraża, że obliczenie dodawania zostanie wykonane wówczas, gdy będą dostępne wartości a i b, a mnożenie wtedy, kiedy zostanie uzyskany wynik dodawania oraz wartość b.

Ta prosta abstrakcja pozwala na wygodne reprezentowanie obliczeń. W przypadku bardziej skomplikowanych procesów, grafy mogą być znacznie bardziej złożone, ale zasada pozostaje ta sama.

4. Narzędzia i technologie

Niniejszy rozdział przedstawia technologie i narzędzia użyte podczas tworzenia pracy.

4.1 Julia

Julia to nowoczesny język programowania, stworzony w 2012 roku przez grupę naukowców z Massachusetts Institute of Technology (MIT). Prace nad językiem programowania Julia rozpoczęły się w 2009 roku — jego pomysłodawcami byli Jeff Bezanson, Stefan Karpinski, Viral B. Shah i Alan Edelman. 12 lutego 2012 r. zespół utworzył i opublikował stronę internetową, na której zaprezentował swoją misję dotyczącą tego języka [39].

Piszą na swoim blogu między innymi tak: „*Chcemy języka, który ma otwarty kod, z liberalną licencją. Chcemy szybkości C z dynamiką Ruby. Chcemy języka, który jest homoikoniczny, ma prawdziwe makra jak Lisp, a jednocześnie oczywistą, znajomą notacją matematyczną jak Matlab. Chcemy czegoś, co jest tak użyteczne do ogólnego programowania jak Python, tak łatwe do statystyki jak R, tak naturalne do przetwarzania napisów jak Perl, tak potężne w algebrze liniowej jak Matlab i tak dobre do łączenia programów jak skrypty w terminalu. Coś, co jest łatwe w nauce, a jednocześnie satysfakcjonujące dla najpoważniejszych hakerów. Chcemy, aby był to język interaktywny i kompilowany*.” [3]

Obecnie po ponad 12 latach od wydania pierwszej wersji Julii, zarówno język, jak i środowisko jego użytkowników, są już rozwinięte i dojrzałe. Nie można powiedzieć, że język Julia jest językiem popularnym w tym samym sensie, co języki ogólnego zastosowania, jak Python czy JavaScript. Jest on popularny w środowiskach badawczych zajmujących się analizą danych, obliczeniami naukowymi, symulacjami, modelowaniem, itp. Stanowi też bardzo dobrą alternatywę dla języków takich, jak Python i R, wykorzystywanych w analizie danych i obliczeniach naukowych [54].

Język Julia zaczyna stanowić bardzo solidną konkurencję dla języka Fortran w zastosowaniach inżynieryjnych. Dotychczas zdecydowana większość modeli klimatycznych i atmosferycznych była tworzona w języku Fortran. Wspólna inicjatywa instytutu MIT (Massachusetts Institute of Technology) i laboratorium JPL (Jet Propulsion Laboratory) agencji NASA, dowodzi, że język Julia zdobywa bardzo silną pozycję w dziedzinie symulacji atmosferycznych [60].

4.1.1 Zalety języka Julia

Wśród zalet języka Julia można wymienić [21]:

- wielometodowość, umożliwiająca zdefiniowanie zachowań funkcji dla różnych kombinacji typów argumentów [16];
- możliwość eleganckiego i wydajnego zastosowania dowolnej funkcji element po elemencie na tablicach, macierzach i innych kolekcjach danych, za pomocą operatorów z kropką lub równoważnej im funkcji `broadcast` [8];
- dynamiczny system typów; typy zdefiniowane przez użytkownika są tak szybkie, jak typy wbudowane [22];

- wydajność na poziomie języków o typach statycznych, jak C czy Fortran; Julia jest kompilowana do kodu maszynowego, a nie interpretowana [12];
- wbudowany menedżer pakietów [18];
- makra i metaprogramowanie o charakterze znanym z języków lispowych [13];
- wywoływanie funkcji C bezpośrednio, bez konieczności stosowania kodu pomocniczego (*wrapperów*) czy specjalnych API [9]; na przykład w języku Python pisanie rozszerzeń jest zadaniem wymagającym, a jego trudność znacząco rośnie, jeśli chcemy tworzyć rozszerzenia zgodne z biblioteką Numpy — jedną z kluczowych bibliotek numerycznych [26];
- łatwa integracja z innymi językami — istnieje możliwość uruchamiania z poziomu Julii funkcji w innych językach, jak np. Python [49], Java [43], R [45] czy Mathematica [44];
- możliwości uruchamiania programów powłoki do zarządzania innymi procesami [20];
- wysoce rozwinięte funkcjonalności dla obliczeń równoległych i rozproszonych [17];
- automatyczne generowanie kodu dla różnych typów argumentów [10];
- konwersja i promowanie dla typów numerycznych (i innych) [11];
- wsparcie kodu Unicode; można pisać programy np. w językach narodowych albo używać symboli matematycznych w kodzie, co pozwala na większą zgodność kodu z równaniami matematycznymi lub pracami naukowymi, do których się on odwołuje [23].

Na rysunku 4.1 przedstawiono przykładowy kod w języku Julia, z użyciem symboli Unicode.

```
using Statistics, LinearAlgebra

function gradient_descent(𝒟train, ϕ, ∇loss; η=0.1, T=100)
    w = zeros(length(ϕ(𝒟train[1][1])))
    for t in 1:T
        w = w .- η*mean(∇loss(x, y, w, ϕ) for (x,y) ∈ 𝒟train)
    end
    return w
end
```

Rysunek 4.1: Przykładowy kod w języku Julia, źródło: [58]

4.1.2 Wielometodowość

Język Julia nie jest językiem obiektowym i nie ma w nim obiektów zaopatrzonych w metody. Zamiast tego mamy typy oraz osobno funkcje. **Metodami** (a także **specjalizacjami**) w języku Julia nazywamy konkretne implementacje danej funkcji, konkretne jej zachowania, zdefiniowane dla określonej kombinacji typów argumentów [15]. Decyzja o tym, która dokładnie wersja funkcji (czyli która metoda) zostanie wywołana, zostaje podjęta w trakcie działania programu dynamicznie na podstawie typów wartości. Ten mechanizm jest nazywany **wielometodowością**, lub **wielokrotną dyspozycją** (*multiple dispatch*). Taki rodzaj polimorfizmu jest szczególnie użyteczny w przypadku kodu matematycznego, w którym często nie da się w naturalny sposób wyróżnić argumentu, do którego metoda przynależy bardziej niż do pozostałych. W praktyce okazuje się to bardzo wygodną cechą pozwalającą na pisanie uogólnionych algorytmów. Więcej na ten temat można przeczytać w [2]. Sam mechanizm wielometodowości nie jest powszechny, ale dostępny natywnie w niektórych językach, takich jak Common Lisp [66] czy Groovy [33].

Zalety wielometodowości (zob. [2] oraz [16]):

- możliwość dodawania specjalizacji danej funkcji dla nowych typów, bez modyfikowania istniejących już metod tej funkcji opracowanych dla innych typów, co ułatwia rozbudowywanie i konserwację kodu;
- kod jest przejrzysty, ponieważ można wyraźnie rozdzielić logikę specyficzną dla różnych typów argumentów;
- nowe zachowania funkcji mogą być dodawane niezależnie od hierarchii typów;
- jest wyjątkowo przydatna w zastosowaniach matematycznych i naukowych, gdzie funkcje mogą mieć różne zachowania w zależności od typów argumentów; umożliwia to naturalne i efektywne wyrażanie operacji matematycznych.

Wady wielometodowości:

- wymaga rozstrzygania niejednoznaczności w przypadku funkcji zawierających argumenty dowolnego typu (typu `Any`) [14];
- powoduje trudność w debugowaniu, kiedy specjalizacje funkcji dla różnych typów są rozproszone w wielu pakietach, często różnych od pakietu, w którym jest definiowana funkcja bazowa [55].

Z punktu widzenia tworzonej biblioteki mechanizm wielometodowości idealnie wpisuje się w realizację grafowego paradygmatu przetwarzania, gdzie argumenty wejściowe połączone krawędziami grafu mogą determinować wybór najbardziej odpowiedniej specjalizacji metody funkcji zawartej w węźle operacyjnym. Mimo że jest to działanie dynamiczne, to nie dzieje się kosztem wydajności.

4.1.3 Rozpowszechnianie operacji (ang. *broadcast*)

W języku Julia istnieje wygodny mechanizm wywoływania funkcji na kolekcjach danych, element po elemencie. Każda funkcja, która jest zdefiniowana dla pojedynczego elementu, może być wywołana na całym wektorze, macierzy, czy na innej kolekcji danych, za pomocą operatora z kropką. Na przykład `sin.(A)`, gdzie `A` jest macierzą dwuwymiarową, zwróci macierz powstałą z obliczenia funkcji sinus dla każdego elementu macierzy. Mechanizm ten jest bardzo wygodny i pozwala na zwiększenie czytelności kodu, a także na poprawę wydajności, ponieważ operacje na tablicach są optymalizowane na etapie kompilacji programu. Warto zaznaczyć, że rozgłaszanie jest mechanizmem ogólnym, wersja funkcji z kropką jest skrótem dla wywołania funkcji `broadcast` [8].

Mechanizm rozpowszechniania operacji w przypadku tworzonej biblioteki pozwala na wygodne stosowanie funkcji do całych wektorów i macierzy danych, bez konieczności budowy osobnych funkcji, co może przyczynić się do zwiększenia przejrzystości i czytelności grafów.

4.2 Pakiety języka Julia

Poniżej wymieniono pakiety języka Julia, które zostały bezpośrednio użyte przy tworzeniu aplikacji.

4.2.1 Statistics

Standardowa biblioteka podstawowych funkcji statystycznych. Jest używana bezpośrednio i jako pomoc w budowie dodatkowych operatorów (zob. [50]).

4.2.2 NetCDF

Biblioteka funkcji do obsługi formatu zapisu danych NetCDF. Zawiera interfejs wysokiego i niskiego poziomu do zapisu i odczytu (zob. [47]).

4.2.3 CSV

Biblioteka do obsługi rodziny formatów csv i danych tabelarycznych w plikach tekstowych (zob. [61]).

4.2.4 DataFrames

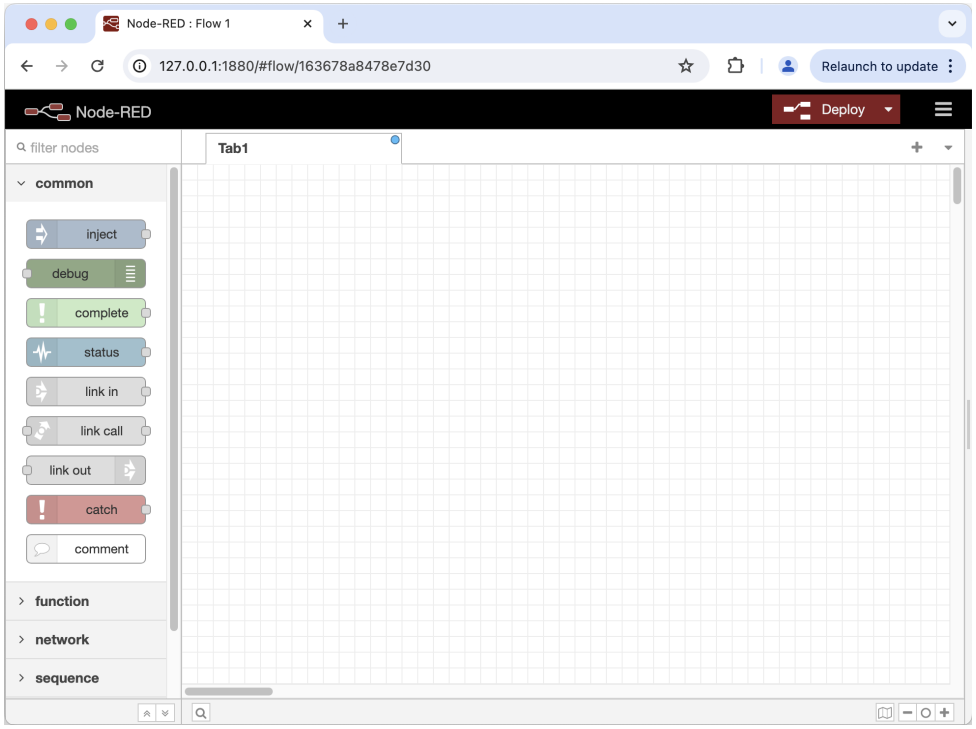
Zbiór narzędzi do pracy z danymi tabelarycznymi w Julii. Funkcjonalność jest zbliżona do popularnych bibliotek: pandas dla języka Python oraz dplyr dla języka R. Pakiet DataFrames zajmuje centralne miejsce w ekosystemie inżynierii danych języka Julia i oferuje liczne integracje z wieloma innymi bibliotekami (zob. [52]).

4.3 Node-RED

Node-RED to narzędzie do programowania wizualnego, które jest przeznaczone do tworzenia aplikacji IoT (ang. *Internet of Things*) w przeglądarce internetowej. Program powstał w 2013 roku jako eksperymentalny projekt w grupie Emerging Technology Services w IBM. Był przeznaczony do wizualizowania i modyfikowania odwzorowań między tematami MQTT (ang. *Message Queuing Telemetry Transport* — protokół komunikacji IoT). Okazał się bardzo skuteczny i stał się szybko centralnym projektem powstałej organizacji JS Foundation [35].

Jest to darmowy i otwarty projekt. Aplikację można uruchomić na dowolnym serwerze lub lokalnie na komputerze. Ten fakt, oraz to, że jest to program stosunkowo prosty w obsłudze, zdecydowało o użyciu go w niniejszej pracy.

Na rysunku 4.2 przedstawiono wygląd interfejsu programu Node-RED przy pierwszym uruchomieniu.



Rysunek 4.2: Interfejs programu Node-RED, źródło: opracowanie własne

4.4 Pozostałe narzędzia

W tym podrozdziale zostały opisane pozostałe narzędzia użyte podczas tworzenia pracy.

4.4.1 Visual Studio Code

Bezpłatny edytor kodów źródłowych udostępniany przez firmę Microsoft [57].

Podczas tworzenia niniejszej pracy ważną cechą programu była oficjalna wtyczka dla języka Julia oraz wysoce zintegrowana z edytorem obsługa pakietu Revise.jl [41], umożliwiającego automatyczne rekompilowanie kodu bez ponownego uruchamiania interakcyjnego środowiska REPL.

4.4.2 neovim

Terminalowy edytor tekstowy, następca programu vi/vim [24]. Stanowił lekkie uzupełnienie dla VS Code.

4.4.3 draw.io

Program do tworzenia diagramów, dostępny w przeglądarce internetowej [46]. Umożliwia tworzenie wizualizacji wektorowych i eksportowanie ich do różnych formatów graficznych.

4.4.4 npm

Menedżer pakietów dla języka JavaScript [34]. Umożliwia instalację i zarządzanie zależnościami dla projektów w JavaScript. Za jego pomocą instaluje się program Node-RED 4.3 oraz jego pakiety rozszerzeń.

4.4.5 git

System kontroli wersji używany do zarządzania kodem źródłowym projektu [69].

4.4.6 lazygit

Szybki terminalowy program do obsługi systemu kontroli wersji git [30].

4.4.7 github

Popularny serwis internetowy obsługujący system kontroli wersji git [37]. Umożliwia przechowywanie kodu źródłowego.

5. Teoretyczne aspekty budowy biblioteki

Rozwiązanie zaproponowane w niniejszej pracy korzysta z elementarnych zagadnień dotyczących grafów skierowanych.

5.1 Grafy skierowane

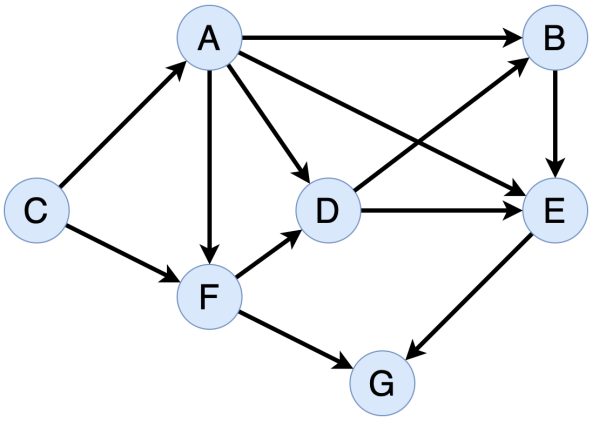
Graf skierowany $G = (V, E)$ to uporządkowana para zbiorów, gdzie V to zbiór wierzchołków, a E to zbiór krawędzi — uporządkowanych par wierzchołków [40].

Krawędź $(u, v) \in E$ nazywamy **krawędzią skierowaną**, lub **łukiem**, gdzie u to wierzchołek początkowy, a v to wierzchołek końcowy.

5.2 Reprezentacja grafu

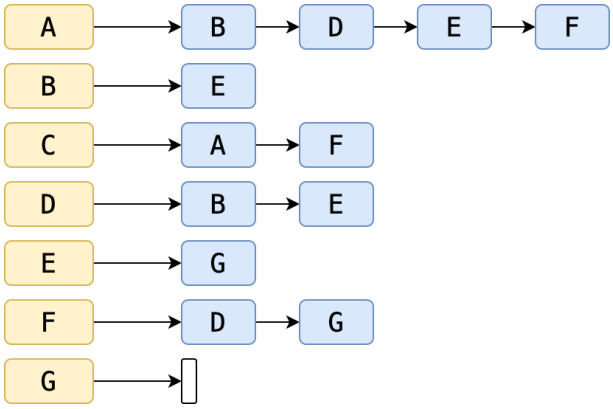
Standardowym sposobem reprezentowania grafu $G = (V, E)$ w przypadku grafów *rzadkich*, czyli takich, dla których $|E| \ll |V|^2$, są **listy sąsiedztwa** [25].

Przykładowo, graf przedstawiony na rysunku 5.1 może być reprezentowany za pomocą list sąsiedztwa pokazanych na rysunku 5.2.



Rysunek 5.1: Przykładowy graf skierowany, źródło: opracowanie własne

Reprezentacja w postaci list sąsiedztwa polega na przechowywaniu tablicy lub słownika zawierającego każdy wierzchołek grafu, z kolei dla każdego z tych wierzchołków — listy wierzchołków sąsiednich. W przypadku grafu skierowanego, lista sąsiedztwa zawiera wierzchołki, do których prowadzą krawędzie z danego wierzchołka.



Rysunek 5.2: Reprezentacja grafu 5.1 za pomocą list sąsiedztwa, źródło: opracowanie własne

5.3 Acykliczne grafy skierowane

W wielu zagadnieniach praktycznych, takich jak planowanie zadań, analiza zależności, czy przetwarzanie danych, występują grafy skierowane, które nie zawierają cykli. Takie grafy nazywamy **acyklicznymi grafami skierowanymi** (ang. *Directed Acyclic Graph*, DAG).

5.4 Sortowanie topologiczne

Sortowanie topologiczne jest najważniejszą operacją wykonywaną na acyklicznych grafach skierowanych. Można o nim myśleć jak o takim uszeregowaniu wierzchołków w jednej linii, że dla każdej krawędzi (u, v) wierzchołek u znajduje się przed wierzchołkiem v, po jego lewej stronie. Takie uporządkowanie nie jest możliwe, jeśli graf zawiera cykl skierowany, ponieważ nie da się odwiedzać wierzchołków po łukach w jednym kierunku i powrócić do wierzchołka, który już został odwiedzony [67].

Każdy acykliczny graf skierowany ma co najmniej jeden porządek topologiczny. Znalezienie takiego porządku może być o tyle użyteczne, że daje on możliwość wykonania działań na każdym wierzchołku przed jego następnikami w grafie. Biblioteka opisywana w niniejszej pracy wykorzystuje tę własność porządku topologicznego.

5.5 Algorytm Kahna

Koncepcyjnie najprostszym algorytmem sortowania topologicznego jest algorytm Kahna [51], inaczej nazywany sortowaniem z usuwaniem wierzchołków niezależnych (mających zerowy stopień wejściowy). Algorytm ten polega na identyfikowaniu wierzchołków, które nie mają krawędzi wejściowych, a następnie usuwaniu ich z grafu wraz z ich krawędziami. Kolejność usunięcia wierzchołków wyznacza porządek topologiczny.

Algorytm ten można opisać w następujący sposób [68]:

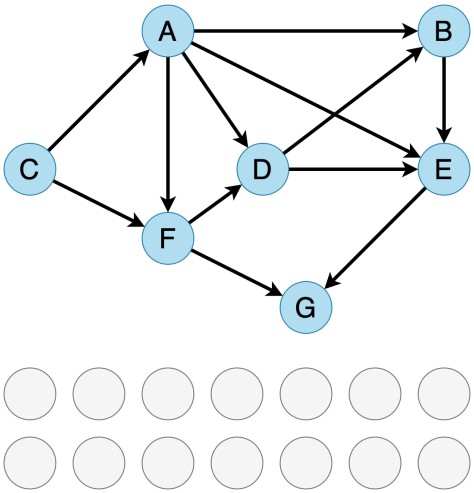
Dane wejściowe: skierowany acykliczny graf $G = (V, E)$, V — niepusty zbiór wierzchołków, E — niepusty zbiór łuków,

1. Utwórz listę L, do której będą wstawiane wierzchołki w porządku topologicznym
2. Znajdź w grafie G wierzchołek niezależny, $u \in V$, tj. taki który ma stopień wejściowy zero.
3. Dołącz wierzchołek u do końca listy L

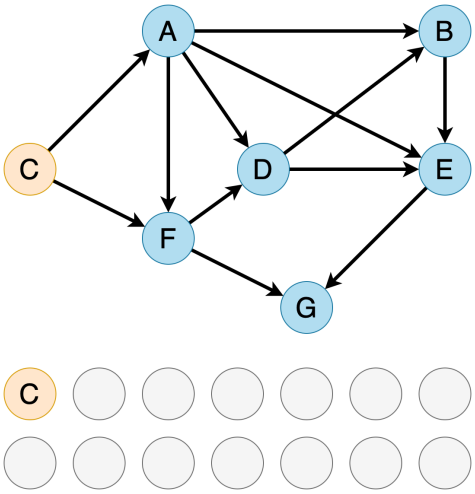
4. Usuń wierzchołek u z grafu G.
5. Jeśli graf G zawiera jeszcze jakieś niezależne wierzchołki, to idź do punktu 2. W przeciwnym razie koniec.

Dane wyjściowe: lista L zawierająca wierzchołki grafu G w porządku topologicznym.

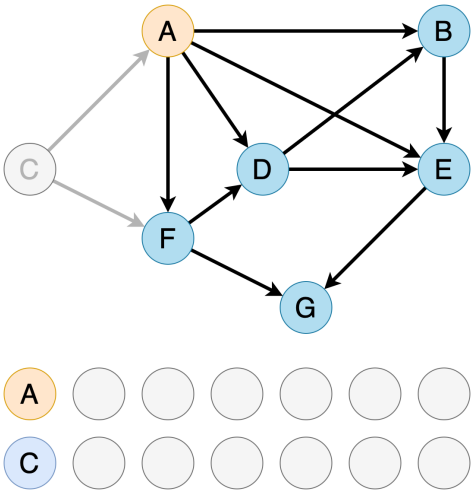
Rysunki 5.3a–5.3i przedstawiają działanie algorytmu Kahna.



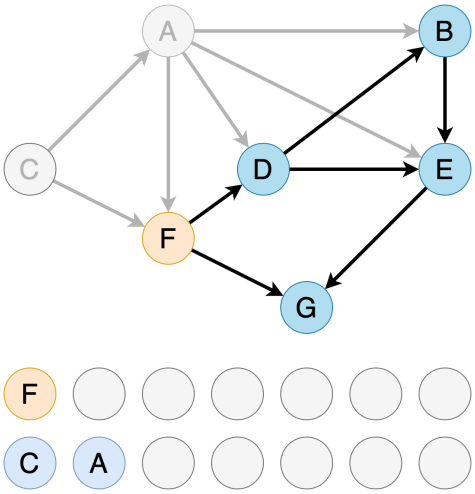
(a) Krok 1



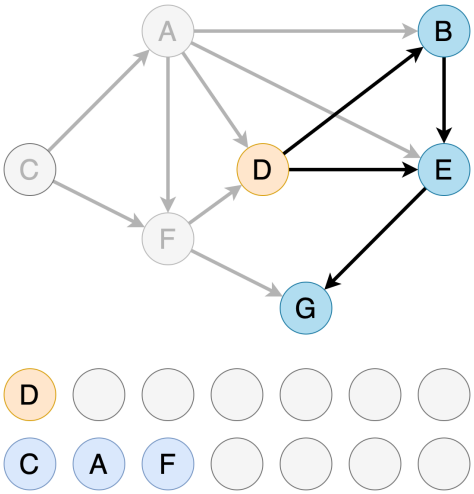
(b) Krok 2



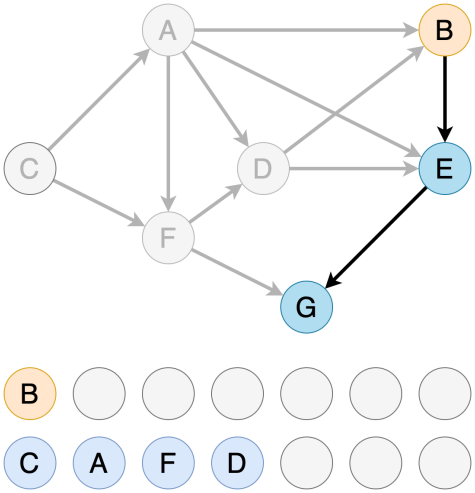
(c) Krok 3



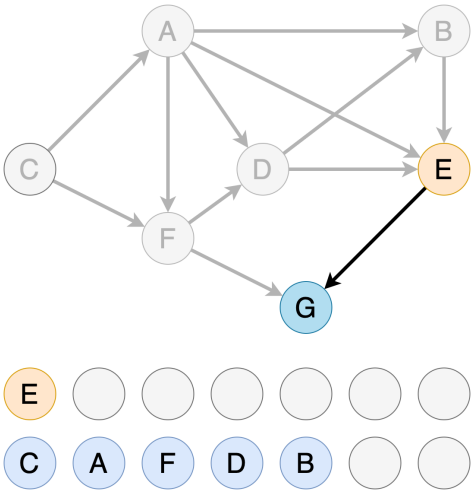
(d) Krok 4



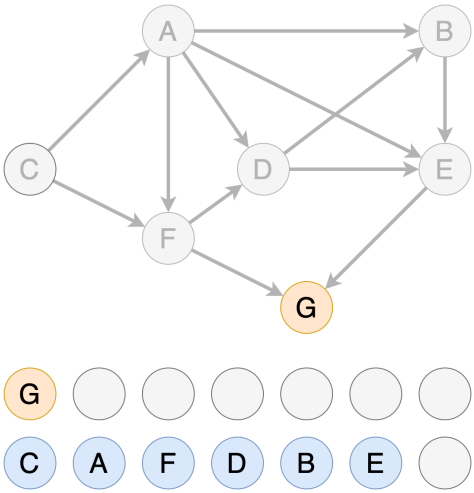
(e) Krok 5



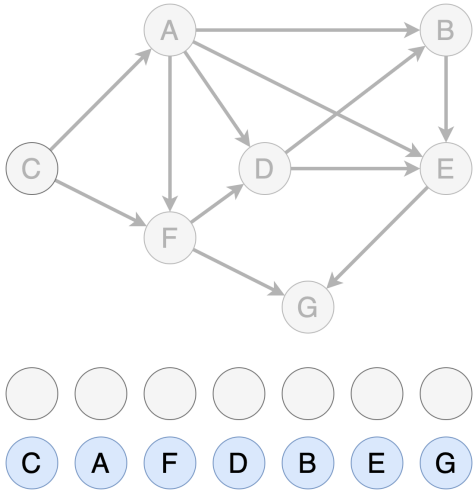
(f) Krok 6



(g) Krok 7



(h) Krok 8



(i) Krok 9 — lista wynikowa

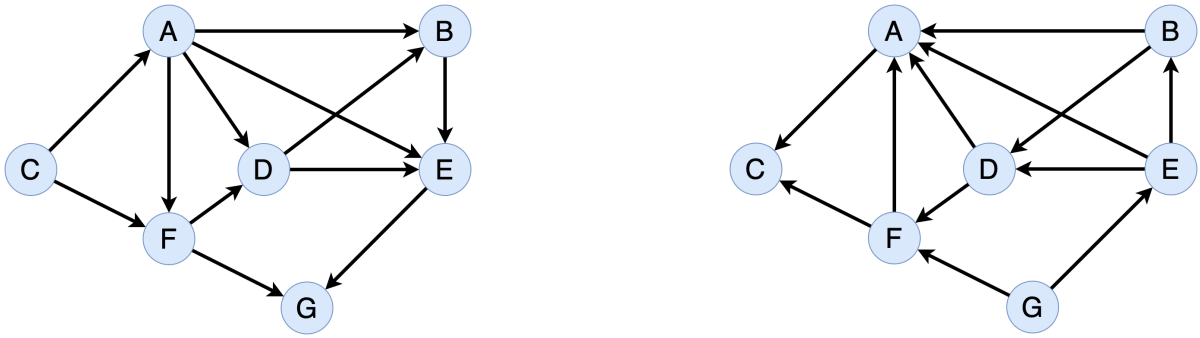
Rysunek 5.3: Ilustracja algorytmu Kahna, źródło: opracowanie własne

5.6 Transpozycja grafu skierowanego

Niech $G = (V, E)$ będzie grafem skierowanym. Wówczas grafem transponowanym grafu G nazywamy graf $G^T = (V, E^T)$, gdzie $(x, y) \in E^T \iff (y, x) \in E$ [25]. Nazwa odzwierciedla fakt, że macierz sąsiedztwa grafu G^T jest transpozycją macierzy sąsiedztwa grafu G. Graf transponowany jest też nazywany grafem odwrotnym, a działanie transponowania — odwracaniem grafu.

Jak zauważono powyżej, jeśli graf jest zadany macierzą sąsiedztwa, to transponowanie jest operacją równoważną transpozycji tej macierzy. W niniejszym oprogramowaniu graf przetwarzania jest reprezentowany za pomocą list sąsiedztwa, realizowanych z użyciem słownika.

Na rysunku 5.4 przedstawiono wizualną interpretację transponowania grafu skierowanego.



(a) Graf G

(b) Graf G^T

Rysunek 5.4: Transponowanie grafu, źródło: opracowanie własne

Algorytm transponowania grafu zadanego listami sąsiedztwa [70]:

Dane wejściowe: n — liczba wierzchołków w grafie; A — n-elementowa tablica list sąsiedztwa grafu wyjściowego.

Dane wyjściowe: A^T — n-elementowa tablica list sąsiedztwa grafu transponowanego.

Zmienne pomocnicze: v — wierzchołek; p, r — wskaźniki elementu listy.

1. Utworzenie n elementowej tablicy list sąsiedztwa A^T;
2. Wypełnienie tablicy A^T pustymi listami;
3. Dla $v = 0, 1, \ldots, n-1$ wykonuj kroki 4–11;
4. $p \leftarrow A[v]$ (ustawienie w p pozycji początku listy sąsiedztwa dla wierzchołka v);
5. Dopóki $p \neq \emptyset$, wykonuj kroki 6–11;
6. Utworzenie nowego elementu listy;
7. $r \leftarrow$ pozycja nowego elementu;
8. $(r \rightarrow v) \leftarrow v$ — umieszczenie numeru v w elemencie;
9. $(r \rightarrow \mathrm{next}) \leftarrow A^T[p \rightarrow v]$ — dodanie elementu r do listy A^T sąsiada;
10. $A^T[p \rightarrow v] \leftarrow r$;
11. $p \leftarrow (p \rightarrow next)$ — przejście do następnego sąsiada na liście A;
12. Koniec.

W tworzonej bibliotece transpozycja grafu jest wykorzystywana do odwrócenia grafu zapisanego przez program Node-RED (zob. rozdział 6). Program Node-RED zapisuje krawędzie jako wskazujące na następne węzły. Każdy węzeł zawiera atrybut `wires` będący listą identyfikatorów węzłów, do których trafia przepływ. Z kolei w tworzonej bibliotece w języku Julia (zob. rozdział 7) relacja sąsiedztwa wskazuje na węzeł, z którego pochodzi argument do funkcji danego węzła. Dlatego odczytanie konfiguracji z programu Node-RED wymaga odwrócenia grafu.

6. Prototyp — pakiet do programu Node-RED

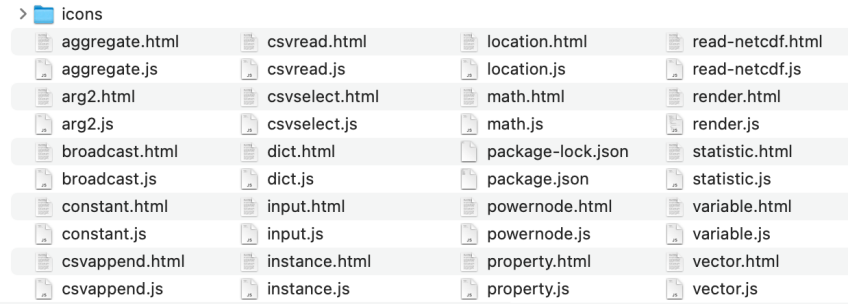
Podrozdziały 6.2–6.4 opisują budowę i działanie pakietu dla programu Node-RED, służącego do budowy węzłów możliwych do odczytania następnie przez bibliotekę w języku Julia (zob. rozdział 7).

6.1 Wstęp

W pracy wykorzystano program Node-RED (zob. podrozdział 4.3) w roli prototypowego edytora wizualnego. Nie są używane jego węzły funkcjonalne i środowisko uruchomieniowe. Ze względu na możliwość samodzielnej budowy pakietów do Node-RED, opisaną szczegółowo w dokumentacji, zdecydowano się na zbudowanie własnego pełnego zestawu węzłów, które nie mają żadnego funkcjonalnego zastosowania w samym programie Node-RED i nie zmieniają przepływów bezpośrednio w programie. Służą one jedynie realizacji zamierzeń niniejszego projektu, czyli umożliwienia wizualnego budowania grafów następnie odczytywanych w bibliotece w Julii (zob. rozdział 7). Konieczne było w tym celu przeanalizowanie i rozpracowanie, w jaki sposób program zapisuje wizualne grafy do plików zewnętrznych. Ten aspekt nie jest opisany w dokumentacji i wymagał eksperymentów oraz pewnej dozy inżynierii odwrotnej.

6.2 Struktura pakietu

Na rysunku 6.1 przedstawiono zawartość utworzonego pakietu węzłów dla programu Node-RED.



Rysunek 6.1: Zawartość dyskowa pakietu węzłów dla programu Node-RED, źródło: opracowanie własne

6.3 Podstawowa budowa elementów

Wszystkie węzły pakietu dla programu Node-RED mają podobną budowę, za wyjątkiem kilku wyróżnionych węzłów specjalnego przeznaczenia.

Pod względem budowy można je podzielić na następujące kategorie:

- zwykłe węzły funkcyjne odpowiadające jakiemuś operatorowi w bibliotece Julia, wyróżnione ze względu na częstość użycia, jako zamknięta całość
- węzły funkcyjne z wbudowanym ograniczeniem wyboru operatora lub jakiegoś argumentu, ułatwia-

jące użytkownikowi podjęcie decyzji, np. wybór parametru statystycznego

- węzeł specjalny Arg2 — do oznaczania kolejności argumentów
- węzeł specjalny render — oznaczający koniec grafu przetwarzania

Definicja każdego węzła (bloczka) dla programu Node-RED składa się z dwóch plików:

- plik JavaScript rejestrujący dany węzeł w programie Node-RED (6.3.1)
- plik HTML definiujący strukturę, wygląd węzła, argumenty i wartości specjalne (6.3.2)

6.3.1 Przykładowy plik JavaScript

Listing 6.1 zawiera kod przykładowego pliku JavaScript dla węzła *aggregate*. Wygląd tego węzła w przestrzeni roboczej programu Node-RED przedstawiono na rysunku 6.2.

Rysunek 6.2: Węzeł `aggregate`, źródło: opracowanie własne

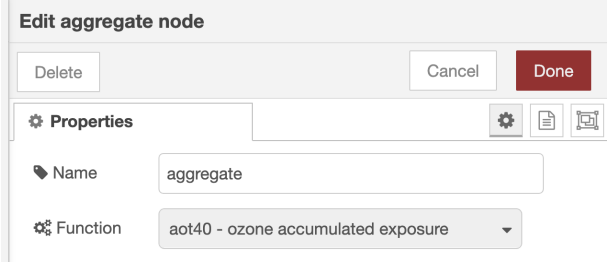
Listing 6.1: Przykładowy plik JavaScript dla węzła *aggregate*

```
module.exports = function(RED) {
    function Aggregate(config) {
        RED.nodes.createNode(this,config);
        var node = this;
        node.on('input', function(msg) {
            msg.payload = { "source_id": this.id };
            msg.topic = "arg1";
            node.send(msg);
        });
    }
    RED.nodes.registerType("aggregate", Aggregate);
}
```

Zostaje wyeksportowana funkcja `Aggregate`, która tworzy nowy węzeł o nazwie `aggregate`.

6.3.2 Przykładowy plik HTML

Listing 6.2 zawiera kod przykładowego pliku HTML dla węzła *aggregate*. Oprócz wyglądu samego węzła, plik ten definiuje również strukturę okna konfiguracyjnego, w którym użytkownik może zmieniać wartości argumentów węzła. Na rysunku 6.3 przedstawiono panel konfiguracyjny węzła `aggregate`.



Rysunek 6.3: Panel konfiguracyjny węzła `aggregate`, źródło: opracowanie własne

Listing 6.2: Przykładowy plik HTML dla węzła *aggregate*

```
<script type="text/javascript">
RED.nodes.registerType('aggregate',{
    category: 'NodeAQ',
    color: '#ffdd99',
    defaults: {
        name: {value: "aggregate"},
        argument: { value: 1 },
        operator: { value: "yavg"}
    },
    inputs: 1,
    outputs: 1,
    icon: "font-awesome/fa-line-chart",
    label: function() {
        return this.operator;
    },
    name: function() {
        return this.operator;
    },
    oneditprepare: function() {
    $("#node-input-operator").typedInput({
        types: [
        {
            value: "operator",
            options: [
                { value: "yavg", label: "yavg - yearly average"},
                { value: "wavg", label: "wavg - winter average"},
                { value: "aot40",
                  label: "aot40 - ozone accumulated exposure"
                },
                { value: "p8d932",
                  label: "p8d932 - p93.2 from rolling 8-hour avg
    maximum"},
                { value: "pd904",
                  label: "pd904 - p90.4 of daily concentrations"
                },
                { value: "pd992",
                  label: "pd992 - p99.2 of daily concentrations"
                },
                { value: "nh350", label: "nh350 - count hours >350
    ug/m3"},
                { value: "nh200", label: "nh200 - count hours >200
    ug/m3"}
            ]
        }
            ],
        // value: node.operator
        })
    },
    oneditsave: function() {
        var node = this;
        node.operator = $("#node-input-operator").val();
    }
});
</script>
```

```
52
53 <script type="text/html" data-template-name="aggregate">
54     <div class="form-row">
55         <label for="node-input-name"><i class="fa fa-tag"></i>
    Name</label>
56         <input type="text" id="node-input-name" placeholder="Name">
57     </div>
58     <div class="form-row">
59         <label for="node-input-operator"><i class="fa fa-cogs"></i>
    Function</label>
60         <input type="text" id="node-input-operator">
61     </div>
62 </script>
63
64 <script type="text/html" data-help-name="aggregate">
65     <p>Aggregate air quality statistics on timeseries.</p>
66     <p>Arg1::DataFrame with yearly timeseries at stations</p>
67 </script>
```

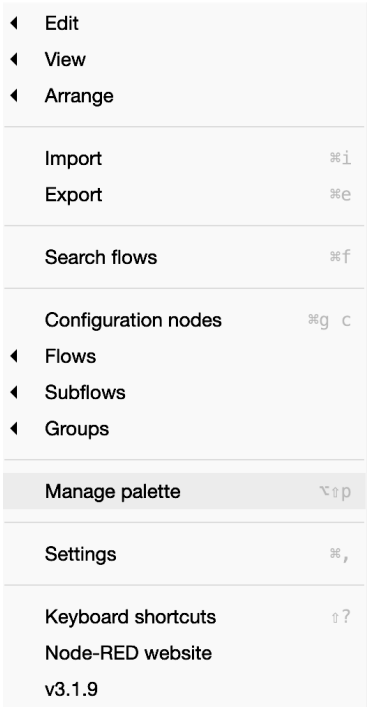
6.4 Instalowanie pakietu

Wszystkie węzły w katalogu pakietu są zgłaszane do rejestracji przez program Node-RED, za pomocą pliku konfiguracyjnego `package.json`. Cały katalog wraz z tym plikiem i zdefiniowanymi w nim węzłami, ustanawia pakiet, który może być zainstalowany w palecie węzłów programu Node-RED za pomocą narzędzia `npm` i polecenia z listingu 6.3.

Listing 6.3: Instalacja pakietu w programie Node-RED

```
1 npm install <katalog-nodeaq>
```

Aby włączyć wyświetlanie zainstalowanych węzłów w programie Node-RED, należy z menu w prawym górnym rogu programu wybrać polecenie **Manage palette**, jak pokazano na rysunku 6.4.



Rysunek 6.4: Menu Manage palette, źródło: opracowanie własne

Następnie w oknie instalacji węzłów nacisnąć przycisk **Enable All** odnoszący się do nowo zainstalowanego pakietu.

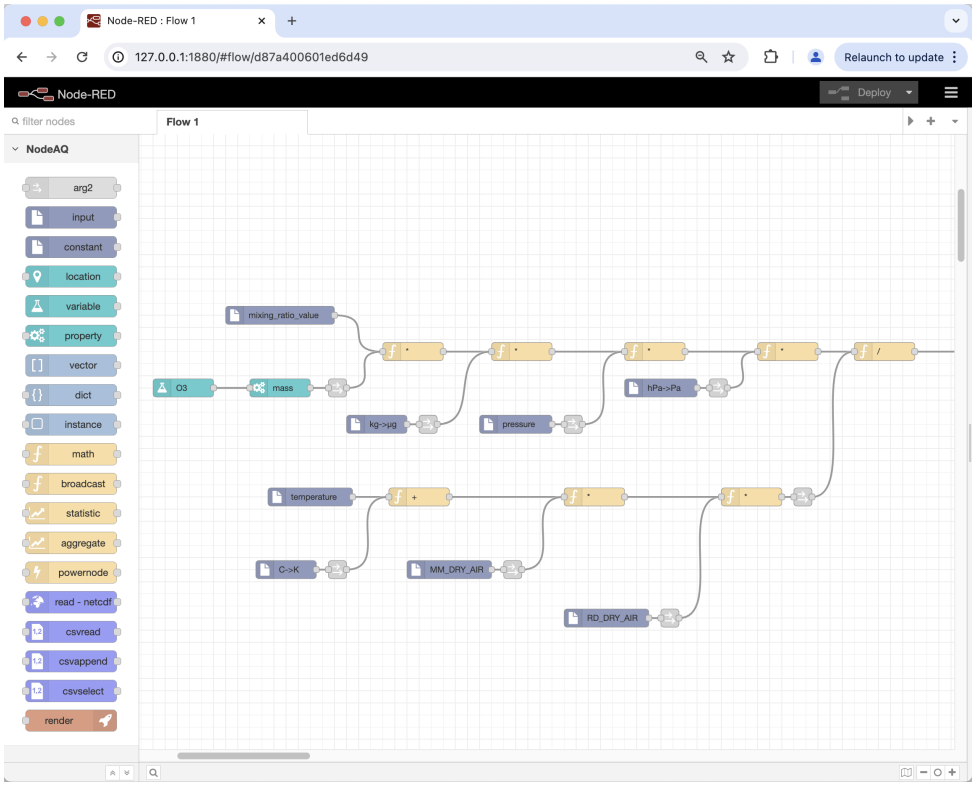
Najlepiej nacisnąć też przycisk **Disable All** dla oryginalnych węzłów programu Node-RED, aby je ukryć. W przypadku tworzenia grafów przepływów dla biblioteki w Julii, oryginalne węzły nie mają żadnego zastosowania.

Biblioteka w Julii traktuje je jak węzły identycznościowe. I na odwrót, przepływy budowane do innych zastosowań i uruchamiane bezpośrednio w programie Node-RED, potraktują węzły pakietu node-aq jako „przezroczyste". Jednak z punktu widzenia zaśmiecenia przestrzeni roboczej, najlepiej nie łączyć tych palet ze sobą.

6.5 Użycie pakietu

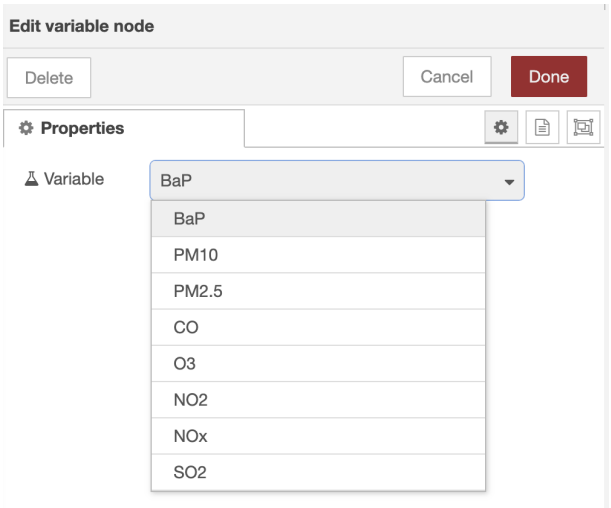
Budowanie grafów polega na przeciąganiu wybranych węzłów z palety po lewej stronie na przestrzeń roboczą, a następnie łączeniu ich krawędziami. Aby połączyć krawędzią wyjście jednego węzła z wejściem następnego, należy kliknąć na porcie wyjściowym (małym kwadracie po prawej stronie) i przeciągnąć krawędź na kolejny węzeł.

Na rysunku 6.5 przedstawiono przykładowy graf zbudowany w programie Node-RED za pomocą węzłów z utworzonego pakietu.



Rysunek 6.5: Przykładowy graf zbudowany w programie Node-RED, źródło: opracowanie własne

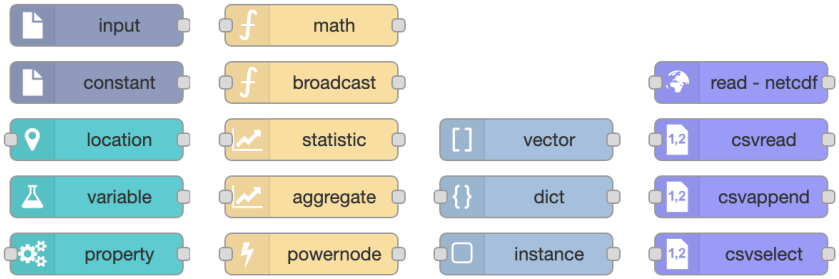
Węzły można konfigurować, klikając na nie dwukrotnie. Wówczas otwiera się okno konfiguracyjne, w którym w zależności od użytego węzła można zmienić podaną wartość, wybierać predefiniowane wartości z listy, itp. Na rysunku 6.6 przedstawiono panel konfiguracyjny węzła `Variable`, w którym można ustalić wybrany związek chemiczny.



Rysunek 6.6: Panel konfiguracyjny węzła `Variable`, źródło: opracowanie własne

6.6 Węzły operacyjne

Na rysunku 6.7 przedstawiono zestaw dostępnych w pakiecie węzłów operacyjnych.



Rysunek 6.7: Węzły operacyjne, źródło: opracowanie własne

Zbiór węzłów operacyjnych na obecnym etapie rozwoju pakietu jest eksperymentalny. Dobór funkcji i operatorów jest subiektywny i podyktowany prowadzonymi testami.

Paleta zawiera następujące węzły:

`Input`

Definiuje wartość stałą podaną przez użytkownika, o typie podstawowym wybranym spośród: `String`, `Int`, `Float32`, `Float64`, `Bool`.

`Constant`

Węzeł, w którym użytkownik może wybrać jedną z predefiniowanych stałych fizycznych, takich jak uniwersalna stała gazowa, masa molowa suchego powietrza, stała Avogadro, itp.

`Location`

Tworzy typ `Location` (zob. podrozdział 7.7.1) z podanego słownika o kluczach `lat` i `lon` zawierającego współrzędne geograficzne (odpowiednio szerokość i długość geograficzną).

`Variable`

Węzeł służący do wybrania predefiniowanego związku chemicznego — wartości typu `Variable` (zob. podrozdział 7.7.9).

`Property`

Węzeł służący do pobrania wskazanej cechy poprzedzającego obiektu, odpowiada funkcji `getproperty` w języku Julia.

`Math`

Zawiera zbiór standardowych funkcji matematycznych.

`Broadcast`

Węzeł zawierający te same funkcje, co węzeł `Math`, ale w wersji wektorowej (zob. podrozdział 4.1.3).

`Statistic`

Zawiera funkcje statystyczne, takie jak średnia, odchylenie standardowe, kwantyle, itp. Mogą one być zastosowane zarówno do wektorów i macierzy, jak i do kolumn z typów tabelarycznych.

`Aggregate`

Węzeł służący do wybierania statystyk agregacyjnych, które można zastosować do serii czasowych, tzn. typu tabelarycznego z indeksem czasowym.

`Powernode`

Węzeł służący do wywołania dowolnej funkcji będącej w przestrzeni nazw Julii. Węzeł ten istnieje na potrzeby testów i eksperymentów, nie jest zalecany do użytku w praktyce, a na pewno nie powinien być wyeksponowany w interfejsie użytkownika, ponieważ pozwala na skonstruowanie potencjalnie niebezpiecznego kodu.

`Vector`

Węzeł służący do tworzenia pustego wektora, do którego następnie można dodawać elementy.

`Dict`

Węzeł służący do tworzenia słownika z pary klucz-wartość.

`Instance`

Węzeł służący do wywołania konstruktora wskazanego typu, z argumentami będącymi wynikami poprzednich obliczeń.

`read-netcdf`

Służy do odczytu pliku w formacie NetCDF (zob. podrozdział 2.1), zwraca obiekt typu macierzowego zawierający pole wartości dla wskazanej zmiennej (najczęściej nazwy związku chemicznego).

`csvread`

Służy do odczytu pliku w formacie CSV, zwraca obiekt typu DataFrame.

`csvappend`

Służy do dodania kolumny do obiektu typu DataFrame.

`csvselect`

Służy do wybrania kolumn z obiektu typu DataFrame.

6.7 Węzeł `Arg2`

Węzeł `Arg2` jest węzłem specjalnym, który służy do oznaczenia kolejności, w której wystąpi ten węzeł na liście argumentów swojego następnika w grafie.

Na rysunku 6.8 przedstawiono węzeł `Arg2` w obu wariantach (pełnym, tak jak występuje na palecie węzłów — 6.8a, i w formie zredukowanej — 6.8b).

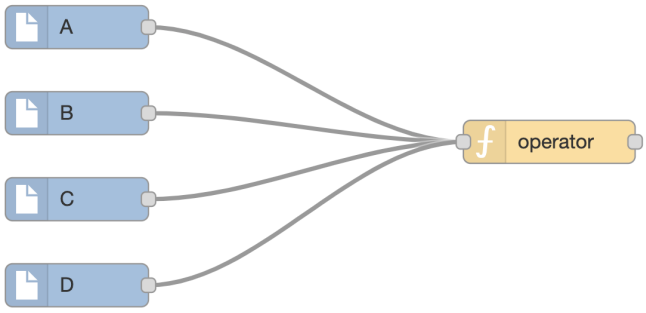
(a) Wersja pełna

(b) Wersja zredukowana

Rysunek 6.8: Węzeł `Arg2`, źródło: opracowanie własne

W programie Node-RED liczba portów wejściowych każdego węzła jest ograniczona do co najwyżej jednego. Wprawdzie można łączyć węzły w sposób dowolny i do wejścia danego węzła można podłączyć dowolną liczbę poprzedników, to węzeł sam z siebie nie ma możliwości określenia kolejności, w jakiej otrzymał dane od swoich poprzedników. Kolejność ta nie jest też ani wymuszana, ani zapisywana w pliku konfiguracyjnym JSON.

Na rysunku 6.9 pokazano przykład połączenia z wieloma krawędziami wejściowymi.



Rysunek 6.9: Wiele węzłów wejściowych, źródło: opracowanie własne

W bibliotece w języku Julia, w której są przetwarzane wynikowe grafy, kolejność ta ma niekiedy kluczowe znaczenie. Poprawną kolejność argumentów można czasem odgadnąć na podstawie typów tych argumentów, jednak nie zawsze jest to możliwe. Np. w przypadku operatorów, takich jak dzielenie dwóch liczb zmiennoprzecinkowych, kolejności argumentów nie da się wywnioskować bez informacji pomocniczej.

Zdecydowano się więc na wprowadzenie węzła specjalnego `Arg2`, niosącego w sobie stosowną informację za pomocą atrybutu `argument = { value : 2 }`, który nie ma żadnego działania w programie Node-RED, ale jest odpowiednio interpretowany w bibliotece w języku Julia.

Aby oznaczyć argument wejściowy jako drugi, należy węzeł `Arg2` umieścić na krawędzi łączącej argument z operatorem, jak pokazano na rysunku 6.10.

(a) Dowolna kolejność

(b) Ustalona kolejność

Rysunek 6.10: Węzeł `Arg2`, źródło: opracowanie własne

Na palecie węzłów programu Node-RED węzeł `Arg2` jest wymieniony z nazwy, jednak zadbano o to, aby podczas przeciągania na obszar roboczy, był on automatycznie redukowany do samej ikony, co pozwala na zachowanie przejrzystości grafu.

Różnicowanie dwóch argumentów wystarcza do tego, aby ustalać kolejność argumentów w funkcjach przyjmujących większą ich liczbę, przez stosowanie funkcji częściowych. Jednak z punktu widzenia wygody użycia, w takich sytuacjach powinno się rozważyć udostępnienie nowych operatorów, aby użytkownicy nie mieli wątpliwości, jak należy łączyć węzły.

6.8 Węzeł `render`

Na rysunku 6.11 przedstawiono wygląd węzła `render`.

Rysunek 6.11: Węzeł `render`, źródło: opracowanie własne

Węzeł `render` jest węzłem specjalnym, który oznacza koniec grafu przetwarzania. W bibliotece w języku Julia i pomocniczym programie uruchomieniowym jest on interpretowany jako oznaczenie wyjścia. Krawędź wchodząca do węzła `render` wskazuje dane, które mają być zwrócone jako wynik obliczeń.

We wszystkich grafach, które są w danym momencie obecne w interfejsie programu (wliczając wszystkie zakładki), a co za tym idzie — wszystkich grafach zapisywanych przez program Node-RED do pliku konfiguracyjnego JSON, musi występować dokładnie jeden węzeł `render`. W przeciwnym razie biblioteka w języku Julia zgłosi błąd i będzie się domagać, aby pozostał tylko jeden taki węzeł.

Węzeł `render` jest jedynym węzłem pakietu, który nie ma portów wyjściowych.

Można używać tego węzła w celach testowych, przestawiając go we wcześniejsze miejsca w grafie, aby zobaczyć, jakie dane powstają na innych etapach przetwarzania. Ta funkcjonalność przynosi też duże korzyści praktyczne, ponieważ pozwala na częściowe uruchamianie przetwarzania, bez konieczności przebudowywania całego grafu. Jeśli np. wynikiem działania grafu jest jakiś zbiór plików, a poprawka dotyczy tylko jednego z nich, to można przestawić węzeł `render` na odpowiednie miejsce i uruchomić graf tylko dla danego podgrafu.

6.9 Wdrożenie grafu

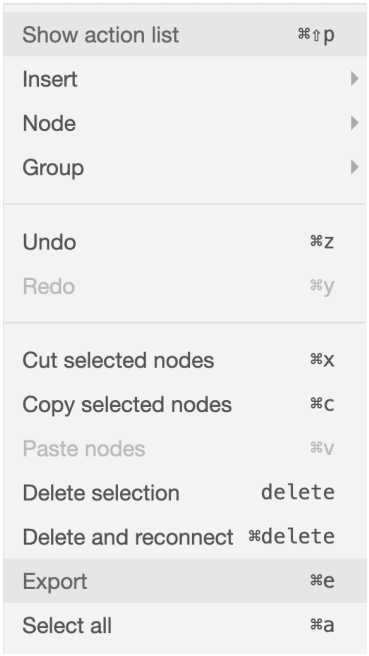
Po zbudowaniu grafu przetwarzania, należy go wdrożyć lub wyeksportować, aby mógł być uruchomiony. W celu wdrożenia grafu, należy nacisnąć przycisk `Deploy` w prawym górnym rogu okna aplikacji. Graf zostanie wówczas zapisany w domyślnym katalogu użytkownika, w pliku o nazwie `flows.json`.

Wygląd przycisku `Deploy` przedstawiono na rysunku 6.12.

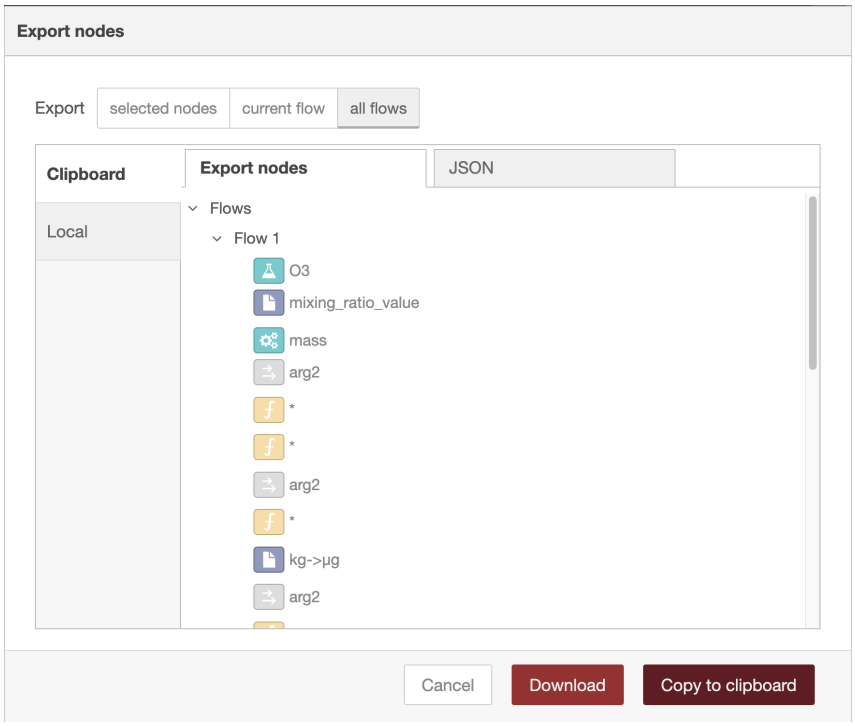
Rysunek 6.12: Przycisk `Deploy`, źródło: opracowanie własne

Alternatywnie, można wyeksportować graf do pliku. W tym celu należy kliknąć prawym przyciskiem myszy na grafie i z wyświetlonego menu wybrać polecenie `Export`. Następnie w wyświetlonym oknie wybrać albo pobranie pliku na dysk albo skopiowanie jego zawartości do schowka. Uzyskanymi w ten sposób plikami można się dzielić z innymi użytkownikami lub przechowywać je na dysku w celu późniejszego uruchomienia.

Na rysunku 6.13 przedstawiono menu kontekstowe grafu, zaś na rysunku 6.14 okno eksportu.



Rysunek 6.13: Menu kontekstowe grafu, źródło: opracowanie własne



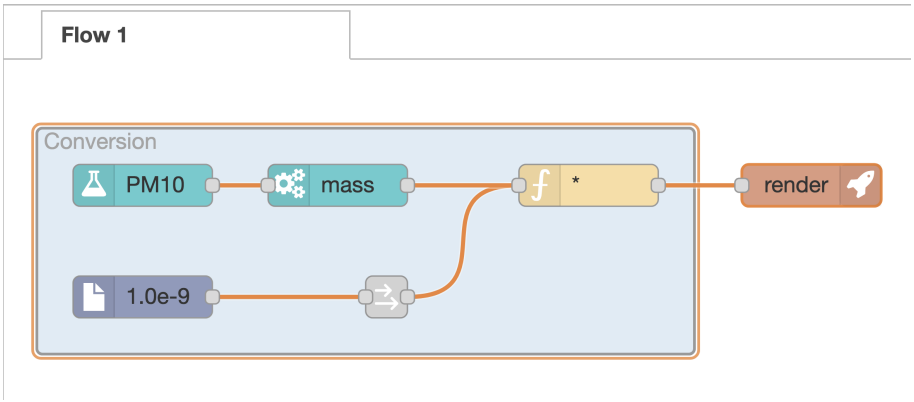
Rysunek 6.14: Okno eksportu, źródło: opracowanie własne

6.10 Format zapisu

Program Node-RED zapisuje grafy przetwarzania w plikach konfiguracyjnych w formacie JSON. Plik taki zawiera listę węzłów, z których każdy jest opisany za pomocą atrybutów, takich jak:

- `id` — unikatowy identyfikator węzła;
- `type` — typ węzła, jeden spośród `tab`, `subflow`, `group` lub jednego z typów zdefiniowanych w pakiecie;
- `name` — nazwa węzła;
- `label` — etykieta w przypadku typu `tab`;
- `x` — współrzędna x węzła na płaszczyźnie;
- `y` — współrzędna y węzła na płaszczyźnie;
- `z` — identyfikator struktury nadrzędnej (zakładki);
- `g` — identyfikator grupy, do której należy węzeł;
- `wires` — lista identyfikatorów węzłów, do których wychodzą krawędzie z węzła;
- `in` — wskazują wejście w węźle typu subflow;
- `out` — wskazują wyjścia w węźle typu subflow;
- `argument` — numer argumentu, który stanowi dany węzeł;
- `arg` — wartość argumentu dla konstruktora typu, w przypadku niektórych węzłów;
- i inne pomocnicze atrybuty dotyczące wyglądu węzła.

Listing 6.4 zawiera przykładowy plik konfiguracyjny JSON zapisany przez program Node-RED, składający się z kilku węzłów z niniejszego pakietu. Odpowiada mu graf przedstawiony na rysunku 6.15. Taki wynikowy plik JSON jest bezpośrednim wejściem dla biblioteki w języku Julia (zob. rozdział 7), która przetwarza go na własną reprezentację grafową i wykonuje obliczenia.



Rysunek 6.15: Przykładowy graf odpowiadający konfiguracji na listingu 6.4, źródło: opracowanie własne

Listing 6.4: Fragment pliku konfiguracyjnego JSON

```
[
    {
        "id": "1759d877ddbfe9c2",
        "type": "tab",
        "label": "Flow 1",
        "disabled": false,
        "info": "",
```

```
        "env": []
    },
    {
        "id": "8f5ff790715d5ddb",
        "type": "group",
        "z": "1759d877ddbfe9c2",
        "name": "Conversion",
        "style": {
            "label": true,
            "fill": "#bfdbef",
            "fill-opacity": "0.54"
        },
        "nodes": [
            "dee590077c5a8f03",
            "6c80f3cae4e52e54",
            "ac72a5a9368b875d",
            "84fae30b157a7b9c",
            "4fb7b177f273e018"
        ],
        "x": 774,
        "y": 519,
        "w": 472,
        "h": 162
    },
    {
        "id": "dee590077c5a8f03",
        "type": "math",
        "z": "1759d877ddbfe9c2",
        "g": "8f5ff790715d5ddb",
        "name": "math",
        "argument": 1,
        "operator": "*",
        "x": 1170,
        "y": 560,
        "wires": [
            [
                "cce3801af93d079b"
            ]
        ]
    },
    {
        "id": "6c80f3cae4e52e54",
        "type": "arg2",
        "z": "1759d877ddbfe9c2",
        "g": "8f5ff790715d5ddb",
        "name": "arg2",
        "l": false,
        "argument": 2,
        "operator": "",
        "x": 1025,
        "y": 640,
        "wires": [
            [
                "dee590077c5a8f03"
            ]
        ]
```

```
        },
        {
            "id": "ac72a5a9368b875d",
            "type": "variable",
            "z": "1759d877ddbfe9c2",
            "g": "8f5ff790715d5ddb",
            "name": "PM10",
            "argument": 1,
            "arg": "PM10",
            "x": 850,
            "y": 560,
            "wires": [
                [
                    "84fae30b157a7b9c"
                ]
            ]
        },
        {
            "id": "cce3801af93d079b",
            "type": "render",
            "z": "1759d877ddbfe9c2",
            "name": "render",
            "argument": 1,
            "x": 1330,
            "y": 560,
            "wires": []
        },
        {
            "id": "84fae30b157a7b9c",
            "type": "property",
            "z": "1759d877ddbfe9c2",
            "g": "8f5ff790715d5ddb",
            "name": "property",
            "argument": 1,
            "arg": "mass",
            "x": 990,
            "y": 560,
            "wires": [
                [
                    "dee590077c5a8f03"
                ]
            ]
        },
        {
            "id": "4fb7b177f273e018",
            "type": "input",
            "z": "1759d877ddbfe9c2",
            "g": "8f5ff790715d5ddb",
            "name": "input",
            "argument": 1,
            "operator": "input",
            "arg": "1.0e-9",
            "argtype": "Float64",
            "x": 850,
            "y": 640,
            "wires": [
```

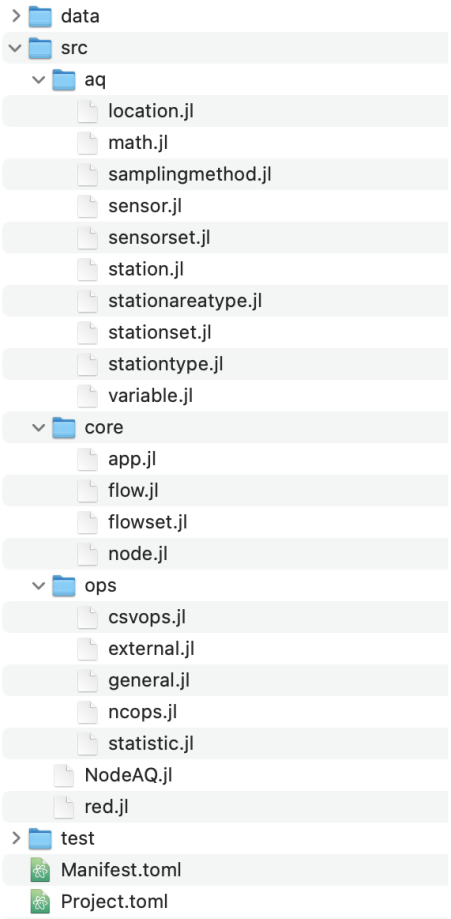
```
                    [
                        "6c80f3cae4e52e54"
                    ]
                ]
            }
    ]
```

7. Prototyp — biblioteka w języku Julia

Biblioteka w języku Julia, o roboczej nazwie NodeAQ, jest częścia odpowiedzialną za odczytywanie plików konfiguracyjnych w formacie JSON utworzonych za pośrednictwem programu Node-RED (zob. rozdział 6) i budowanie z nich abstrakcyjnej reprezentacji grafów przetwarzania oraz uruchamianie tych grafów. Funkcjonalność ta może być używana zarówno w trybie interakcyjnym w środowisku REPL Julii, jak też wsadowo, za pomocą dodatkowego programu uruchamiającego.

7.1 Organizacja biblioteki w języku Julia

Na rysunku 7.1 przedstawiono strukturę organizacji kodu źródłowego biblioteki NodeAQ w języku Julia.



Rysunek 7.1: Struktura katalogów biblioteki NodeAQ, źródło: opracowanie własne

Główny kod programu znajduje się w katalogu `src`. Pozostałe katalogi `data` i `test` zawierają odpowiednio dane testowe i kod testowy.

Podrozdziały 7.2–7.9 opisują poszczególne składniki biblioteki, począwszy od najważniejszych.

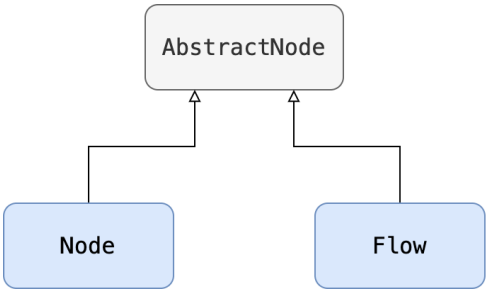
7.2 Katalog `core`

Katalog zawiera typy i funkcje stanowiące trzon biblioteki do przetwarzania danych.

7.3 Typ `node`

Definiuje typy i funkcje związane z węzłami w grafie przetwarzania danych.

Podstawowym typem abstrakcyjnym jest `AbstractNode`. Reprezentuje on obiekty w grafie przypływu danych. Są dwa typy konkretne podrzędne względem `AbstractNode`: `Node` oraz `Flow`, z których pierwszy jest pojedynczym węzłem z operatorem, a drugi jest węzłem złożonym, zawierającym w sobie podgraf. Relację między nimi przedstawia rysunek 7.2.



Rysunek 7.2: Hierarchia typów węzłów w bibliotece NodeAQ, źródło: opracowanie własne

Budulcem i kluczowym składnikiem wszystkich grafów przetwarzania jest `Node`. Typ `Node` reprezentuje w grafie przepływu danych węzeł zawierający pojedynczą nazwaną lub anonimową funkcję. Definicja typu `Node` została przedstawiona na listingu 7.1.

Listing 7.1: Definicja typu `Node`

```
mutable struct Node <: AbstractNode
    id :: String
    label :: String
    inputs :: Vector
    op :: Function
    attributes :: Dict
end
```

Składowymi typu `Node` są identyfikator (`id`), nazwa wyświetlana (`label`), wektor węzłów wejściowych (`inputs`), operator (`op`) oraz słownik atrybutów (`attributes`), przeznaczony na dodatkowe informacje, cechy istotne w interfejsie graficznym lub przyszłe funkcjonalności. W przypadku niektórych węzłów atrybuty te biorą udział w wyborze funkcji operatora podczas budowania grafu.

Typ `Node` jest wyposażony w konstruktory pozwalające zarówno na ręczne definiowanie węzłów, jak też budowę automatyczną na podstawie zapisanego na dysku pliku konfiguracyjnego. Konstruktory, do których nie przekazuje się funkcji operatora (argumentu `op`), tworzą same anonimową funkcję tożsamościową `(x, args...) -> x::Any`, zwracającą pierwszy argument, aby każdy nowo utworzony węzeł bez konkretnej funkcji mógł figurować w drzewie przetwarzania danych, nie mając wpływu na wynik obliczeń.

Listing 7.2 przedstawia dostępne konstruktory typu `Node`.

Listing 7.2: Konstruktory typu `Node`

```
Node(id::String, label:: String, inputs :: Vector, op :: Function) =
    Node(id, label, inputs, op, Dict())
Node(id::String; attributes::Dict=Dict()) =
    Node(id, "", [], (x, args...) -> x, attributes)
Node() = Node("")
Node(attr::Dict, op::Function) =
    Node(attr["id"], attr["name"], [], op, attr)
```

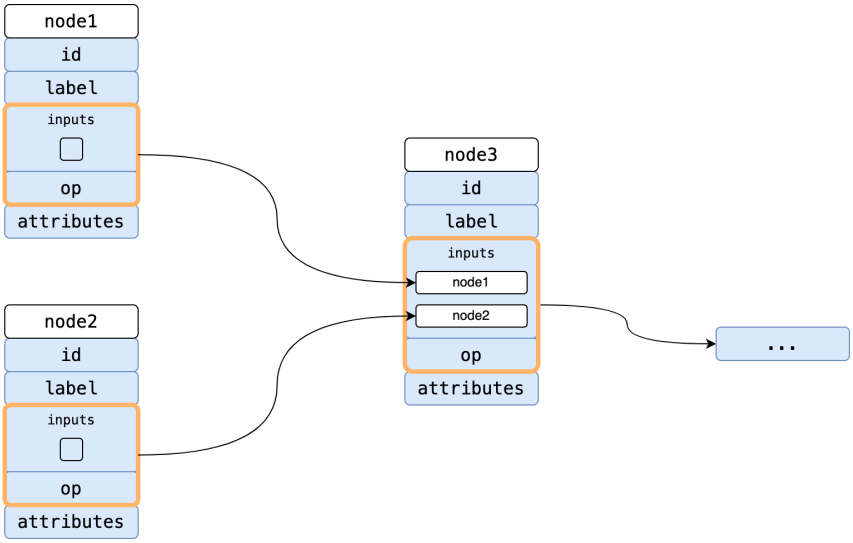
Ewaluacja funkcji w węźle jest inicjowana przez wywołanie metody `output(n::AbstractNode)`. Metoda ta wywołuje funkcję operatora `n.op`, z argumentami będącymi wynikiem zmapowania metody output na wektorze wejściowym. Kod funkcji `output` pokazano na listingu 7.3.

Listing 7.3: Funkcja `output(n::AbstractNode)`

```
function output(node :: AbstractNode)
    if isempty(node.inputs)
        return node.op()
    else
        return node.op(map(x -> output(x), node.inputs)...)
    end
end
```

W efekcie funkcja `output` wywołuje `output` na wszystkich węzłach-poprzednikach w grafie, dochodząc do węzłów niezależnych, dla których to wywołuje samą funkcję bezargumentową w węźle i zwraca wynik. Na obecnym etapie rozwoju biblioteki węzły niezależne zawierają w sobie funkcję zwracającą jakąś wartość stałą lub instancję typu dostępnego w standardowej przestrzeni nazw. Potencjalnie mogą to być też węzły z funkcjami nie wymagającymi żadnych danych wejściowych, jak np. węzeł z funkcją generującą losowe liczby, albo węzeł z funkcją pobierającą dane z zewnętrznego źródła.

Na rysunku 7.3 pokazano schematycznie działanie funkcji `output` używanej do ewaluacji węzłów w grafie przetwarzania danych.



Rysunek 7.3: Działanie funkcji `output(n::AbstractNode)`, źródło: opracowanie własne

Warto podkreślić, że taki proces zachodzi dopiero podczas ewaluacji (przy wywołaniu funkcji `output`). Wektory `inputs` nie przechowują kopii ani referencji obliczonych wartości, a jedynie wskazują na węzły grafu.

7.4 Typ `flow`

Zawiera definicję typu `Flow` i metod na nim operujących.

Typ `Flow` reprezentuje węzeł, który zawiera w sobie podgraf złożony z innych węzłów typu `AbstractNode`. Uczynienie węzła `Flow` podtypem typu abstrakcyjnego `AbstractNode` jest podyktowane ideą, aby można było tworzyć funkcje złożone z innych funkcji, a jednocześnie traktować je w całości jako odpowiedniki pojedynczych węzłów w grafie przepływu danych.

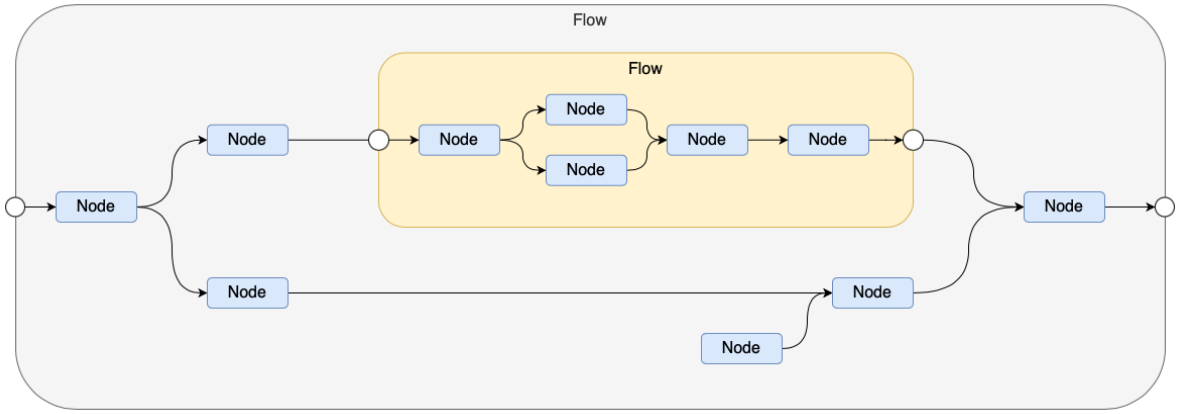
Alternatywnie zamiast użycia węzła `Flow` w charakterze węzła z funkcją złożoną, można utworzyć węzeł `Node` zawierający jako operator wyrażenie funkcji złożonej, korzystając z wygodnej składni języka Julia naśladującej konwencjonalny matematyczny zapis $(f \circ g)$. Wadą tego podejścia jest to, że struktura funkcji złożonej nie jest widoczna w grafie, co utrudnia zrozumienie przepływu danych. Dodatkowo funkcja złożona zapisana w formie wyrażenia jest mniej elastyczna w kwestii dostępu do dodatkowych argumentów jej funkcji składowych.

Drugie przeznaczenie węzłów typu `Flow` ma związek z tym, jak są reprezentowane i zapisywane grupy węzłów w języku Node-RED. W programie Node-RED można bowiem zapisywać podgrafy jako pojedyncze węzły wielokrotnego użytku. Mechanizm ten pozwala użytkownikom na tworzenie skomplikowanych funkcji złożonych z prostszych funkcji, a następnie używanie ich jako pojedynczych węzłów w grafach przepływu bez konieczności kopiowania całych podgrafów.

Odczytywanie takich grupowych węzłów zapisanych przez program Node-RED nie jest obecnie zaimplementowane w bibliotece w Julii ze względu na nie rozpoznany jeszcze dokładny sposób ich zapisywania w pliku JSON, w przypadku wielokrotnych zagnieżdżeń. Funkcjonalność ta stanowi jednak oczekiwany kierunek rozwoju biblioteki wpływający na jej elastyczność i użyteczność.

Z uwagi na powyższe argumenty, uzasadnione jest to, aby cały graf przetwarzania danych również był reprezentowany jako pojedynczy węzeł typu `Flow`.

Ideę tę można zilustrować rysunkiem 7.4.



Rysunek 7.4: Zastosowanie typów `Node` i `Flow`, źródło: opracowanie własne

7.5 Typ `flowset`

Zawiera definicję typu `FlowSet`, którego głównym zadaniem jest reprezentowanie całego zestawu grafów, zakładek z grafami, oraz definicji grafów grupowych z plików konfiguracyjnych JSON programu Node-RED. Funkcje skojarzone z typem `FlowSet` w pliku służą do parsowania plików JSON i budowania na ich podstawie grafów przetwarzania.

Zdefiniowana tutaj funkcja `parse_flow` pobiera plik JSON z dysku lub pod wskazanym adresem url i po przetworzeniu zapisanej w nim konfiguracji grafu przetwarzania, zwraca obiekt typu `FlowSet`.

Funkcja parsująca tworzy najpierw słownik wszystkich identyfikatorów węzłów występujących w pliku konfiguracyjnym, oraz wszystkich struktur nadrzędnych węzłów, takich jak zakładki i węzły grupowe. Jest to potrzebne do odczytania relacji sąsiedztwa węzłów, w których to relacjach identyfikatory są użyte jako wskaźniki do węzłów-sąsiadów. Wymaga to dodatkowej iteracji po wszystkich węzłach, ponieważ zdecydowano się nie polegać na kolejności w jakiej węzły są zapisane w pliku. Jest wielce prawdopodobne, że plik JSON z programu Node-RED zawsze zawiera poszczególne obiekty już w porządku topologicznym (zob. podrozdział 5.4, ale biblioteka w Julii nie powinna tego zakładać z góry.

Następnie jest tworzona tymczasowa struktura grafu transponowanego względem struktury zawartej w pliku JSON. Jest ona budowana w formie list sąsiedztwa (zob. 5.2) opartych na słowniku. Lista `wires` (zob. podrozdział 6.10) zawiera węzły docelowe. Każdy z tych węzłów zostaje wstawiony do słownika węzłów źródłowych, jeśli go tam jeszcze nie ma, i wskazany jako węzeł źródłowy. Na listingu 7.4 przedstawiono fragment odpowiedzialny za umieszczanie kolejnych parsowanych węzłów w grafie transponowanym.

Listing 7.4: Fragment funkcji `parse_flow` odpowiedzialny za tworzenie grafu transponowanego

```
# wire_sources :: Dict
if P_WIRES in keys(node)
    for wire in node[P_WIRES]
        for target_id in wire
            if target_id in keys(wire_sources)
                push!(wire_sources[target_id], node[P_ID])
            else
                wire_sources[target_id] = [node[P_ID]]
            end
        end
    end
end
```

Każdy przetworzony węzeł jest od razu konkretyzowany jako typ `Node` (zob. podrozdział 7.3). Na listingu 7.5 przedstawiono fragment funkcji `parse_flow` odpowiedzialny za konkretyzację węzłów.

Listing 7.5: Fragment funkcji `parse_flow` odpowiedzialny za konkretyzację węzłów

```
if node[P_TYPE] !in keys(NODE_RED_TYPES)
    if occursin(":", node[P_TYPE])
        type, subtype = split(node[P_TYPE], ":")
        if type == "subflow"
            n = NODE_RED_TYPES["subflow"](node)
        end
    else
        n = NODE_RED_TYPES["pass"](node)
    end
else
    if (P_DISABLE in keys(node))
```

```
        && (node[P_DISABLE] == true)
        n = NODE_RED_TYPES["pass"](node)
    else
        n = NODE_RED_TYPES[node[P_TYPE]](node)
    end
end
nodes[node[P_ID]] = n
```

Funkcja ta korzysta ze słownika odwzorowań między typami węzłów w programie Node-RED a funkcjami konstruującymi węzły w bibliotece w Julii. W przypadku, gdy typ węzła nie jest rozpoznany, jest on traktowany jako węzeł przekazujący dane bez zmian. Na listingu 7.6 przedstawiono wybrane odwzorowania typów węzłów programu Node-RED na operacje zapisywane w węzłach `Node`. Każda para klucz-wartość oznacza nazwę typu węzła oraz funkcję konstruującą węzeł w Julii na podstawie obiektu JSON z pliku konfiguracyjnego.

Listing 7.6: Wybrane odwzorowania typów węzłów w programie Node-RED

```
"arg2" => n -> Node(n, x -> x),
"constant" => n -> Node(n, constant(astype(Float64, n[P_ARG]))),
"csvread" => n -> Node(n, x -> csvread(x, n)),
"dict" => n -> Node(n, (x, y) -> Dict(x => y)),
"input" => n -> Node(n, constant(astype(INPUT_TYPES[n[P_ARGTYPE]],
   n[P_ARG]))),
"instance" => n -> Node(n, create_instance(n[P_OP])),
"location" => n -> Node(n, (x, y) -> Location(x, y)),
"math" => n -> Node(n, eval(Meta.parse(n[P_OP]))),
"pass" => n -> Node(n, pass),
"property" => n -> Node(n, x -> getproperty(x, Symbol(n[P_ARG]))),
"variable" => n -> Node(n, constant(string_to_variable(n[P_ARG]))),
"vector" => n -> Node(n, constant([])),
"subflow" => n -> Flow(n[P_ID], n[P_NAME]; attributes=n),
```

Następnie zostają zmapowane relacje sąsiedztwa węzłów z tymczasowego słownika, na relacje sąsiedztwa węzłów w grafie przetwarzania danych. Ten dodatkowy krok wynika z faktu, że definicje funkcji w Julii mają swój wiek (*world age*) i nie mogą być użyte w tym samym „świecie”, w którym zostały zdefiniowane. O ile w przypadku użycia operatorów zdefiniowanych w całości w bibliotece Julii ten problem nie występuje, o tyle w przypadku funkcji, które przy tworzeniu używają jakichś atrybutów węzłów ze środowiska graficznego, trzeba wziąć pod uwagę ten aspekt. Więcej na ten temat można przeczytać w [19].

Na listingu 7.7 przedstawiono fragment funkcji `parse_flow` odpowiedzialny za przeniesienie relacji sąsiedztwa węzłów z tymczasowego słownika, na obiekty typu `Node` w grafie przetwarzania danych.

Listing 7.7: Fragment funkcji `parse_flow` przenoszący relację sąsiedztwa węzłów

```
for (target, sources) in wire_sources
    if target in keys(nodes)
        nodes[target].inputs = map(x -> nodes[x], sources)
    end
end
```

Ostatnim etapem jest sprawdzenie kolejności argumentów wejściowych każdego węzła na podstawie obecności węzłów typu `Arg2` (zob. podrozdział 6.7). Po zmianie kolejności argumentów wejściowych

węzłów, zwracany jest obiekt typu `FlowSet` zawierający wszystkie węzły przetwarzania danych.

Na utworzonym w ten sposób obiekcie `FlowSet` wywołuje się funkcję `render`, która inicjuje ewaluację grafu i zwraca wynik obliczeń (zob. też podrozdział 7.10).

Typ `FlowSet` przechowuje informacje o tym, w którym z utworzonych grafów w całym zestawie, znajdują się węzły typu `render` i ile ich jest. Jeśli jest dokładnie jeden, to zostaje on uruchomiony, zaś w pozostałych przypadkach jest zwracany błąd.

7.6 Moduł `app`

Moduł oddzielający logicznie funkcje uruchomieniowe najwyższego rzędu, pozwalające na parsowanie i uruchamianie domyślnego pliku konfiguracyjnego.

Program Node-RED zapisuje pliki konfiguracyjne i plik JSON z bieżącym grafem w domyślnym katalogu użytkownika. W przypadku braku podania ścieżki do pliku, biblioteka automatycznie odczytuje plik z domyślnego katalogu.

7.7 Katalog `aq`

Katalog `aq` zawiera typy związane z substancjami chemicznymi, lokalizacjami geograficznymi, stacjami i stanowiskami pomiarowymi dotyczącymi zanieczyszczeń powietrza.

7.7.1 `Location`

Typ `Location` określa lokalizację geograficzną we współrzędnych biegunowych oraz wysokość nad poziomem morza. Z typem tym jest związana funkcja `distance`, która oblicza odległość geograficzną w kilometrach, między dwoma lokalizacjami. Funkcja ta może być wykorzystana do obliczania korelacji przestrzennych stężeń substancji w atmosferze. Obecnie jedynym dostępnym algorytmem obliczania odległości jest długość łuku ortodromy między dwoma lokalizacjami, z użyciem funkcji sinus versus.

7.7.2 `SamplingMethod`

Typ wyliczeniowy oznaczający sposób pozyskiwania pomiarów na stanowisku pomiarowym.

7.7.3 `Sensor`

Typ odpowiadający stanowisku pomiarowemu na stacji pomiarowej. Stężenia substancji na stacjach pomiarowych są mierzone różnymi metodami za pomocą zróżnicowanych aparatów pomiarowych. Stanowisko odpowiada każdemu takiemu urządzeniu. Podczas analizy danych czasem ma znaczenie możliwość odfiltrowania niektórych pomiarów na podstawie użytej metody pozyskiwania danych.

7.7.4 `SensorSet`

Typ będący aliasem dla zbioru sensorów `::Set{Sensor}` (zobacz 7.7.3).

7.7.5 `Station`

Typ oznaczający stację pomiarową. Każda stacja może zawierać wiele stanowisk pomiarowych (zobacz 7.7.3). Przeznaczeniem tego typu jest głównie filtrowanie pomiarów na podstawie cech lokalizacji, takich jak rodzaj stacji: (komunikacyjna — do mierzenia zanieczyszczeń w arteriach komunikacyjnych, albo

przemysłowa — związana z zanieczyszczeniami przemysłowymi). Innym kryterium wyboru jest rodzaj obszaru geograficznego, miejski, pozamiejski.

7.7.6 `StationAreaType`

Typ wyliczeniowy obszaru, z którym jest związana stacja pomiarowa.

7.7.7 `StationSet`

Typ synonimiczny dla zbioru stacji `::Set{Station}`.

7.7.8 `StationType`

Typ wyliczeniowy rodzaju stacji.

7.7.9 `Variable`

Typ oznaczający substancję chemiczną, na potrzeby usystematyzowania i ustandaryzowania nazw i symboli związków chemicznych pobieranych z plików zewnętrznych i na potrzeby raportowania.

Typ ten zawiera też informację o wartościach mas cząsteczkowych danych związków chemicznych, co jest wykorzystywane w wielu obliczeniach matematyczno-fizycznych.

Wśród atrybutów tego typu są też standardowe nazwy tych związków używane w przypadku pobierania danych z powszechnie stosowanych zbiorów i baz danych.

7.7.10 `math`

Zbiór stałych i funkcji ściśle powiązanych z powyższymi typami, takich jak funkcje odległości, konwersja jednostek i inne funkcje pomocnicze.

7.8 Katalog `ops`

Katalog zawiera operatory, których można używać w węzłach budowanych grafów. Rozszerzają zbiór funkcji ogólnodostępnych w bibliotece standardowej o operacje bardziej wyspecjalizowane, skrojone na miarę potrzeb użytkowników.

7.8.1 `csvops`

Funkcje do obsługi plików csv i tabel typu DataFrame w uzgodnionych formatach dla danych serii czasowych i danych o stacjach pomiarowych.

7.8.2 `external`

Miejsce dla instrukcji dołączania zewnętrznych pakietów funkcji. Cechą typową dla języka Julia jest podział na wiele małych pakietów o zawężonej funkcjonalności, które można ze sobą dowolnie składać. Koszt, który się z tym wiąże, polega na wydłużeniu czasu kompilacji bibliotek. Moduł ten służy ułatwieniu importowania pakietów.

7.8.3 `general`

Miejsce na operatory ogólne, ułatwiające budowanie grafów, ale nie niezbędne do ich działania czy składania. Niektóre kombinacje węzłów lub funkcji mogą być trudne do komponowania bez szczegółowej wiedzy o wewnętrznych funkcjach w Julii.

7.8.4 `ncops`

Funkcje służące do zapisu i odczytu danych z plików NetCDF, rozszerzone o obsługę typów z folderu `aq`.

7.9 Katalog `statistic`

Katalog ten zawiera funkcje statystyczne związane z analizą i raportowaniem danych dotyczących jakości powietrza, według rozporządzeń Ministerstwa Klimatu i Środowiska.

Poniższa lista opisuje, które funkcje zawarte w katalogu `statistic` odpowiadają parametrom statystycznym używanym do analizy wyników matematycznego modelowania transportu i przemian substancji w powietrzu, w zależności od modelowanej substancji [64]. Schemat nazw funkcji jest zgodny z konwencją stosowaną w Generalnym Inspektoracie Ochrony Środowiska i dla osób związanych z analizą jakości powietrza nie powinien być obcy, mimo że nazwy są skrótowe.

Dwutlenek siarki (SO2)

- `ph997` —– percentyl 99,7 z rocznej serii stężeń jednogodzinnych;
- `nh350` — liczba przekroczeń wartości jednogodzinnej $350\,\mu g/m^3$ w roku kalendarzowym;
- `pd992` — percentyl 99,2 z rocznej serii stężeń dobowych;
- `nd125` — liczba dni z przekroczeniami wartości dobowej $125\,\mu g/m^3$ w roku kalendarzowym;
- `wavg` — średnie stężenie w okresie zimowym (01.10–31.03);
- `yavg` — średnie stężenie roczne.

Dwutlenek azotu (NO2)

- `ph998` — percentyl 99,8 z rocznej serii stężeń jednogodzinnych;
- `nh200` — liczba godzin z przekroczeniami wartości jednogodzinnej $200\,\mu g/m^3$ w roku kalendarzowym;
- `yavg` — stężenie średnie roczne.

Tlenki azotu (NOx)

- `yavg` — stężenie średnie roczne.

Pył zawieszony PM10

- `pd904` — percentyl 90,4 z rocznej serii stężeń dobowych;
- `nd50` — liczba dni z przekroczeniami wartości dobowej $50\,\mu g/m^3$ w roku kalendarzowym;
- `yavg` — stężenie średnie roczne;
- `max36` — trzydzieste szóste maksimum ze średnich dobowych.

Zgodnie z rozporządzeniem [64], trzydzieste szóste maksimum ze średnich dobowych jest to: „*36. wartość w uporządkowanym nierosnąco ciągu wartości stężeń 24-godzinnych z roku kalendarzowego. Jest to wartość powiązana z definicją poziomu dopuszczalnego dla pyłu zawieszonego PM10. Jeżeli wartość 36.*

maksimum jest większa od 50 µg/m³, oznacza to, że w roku wystąpiło więcej niż dozwolone 35 przypadków przekroczeń poziomu dopuszczalnego (D24=50 µg/m³), czyli że stężenie dopuszczalne 24-godzinne dla pyłu zawieszonego PM10 zostało przekroczone."

Pył zawieszony PM2,5

- `yavg` — stężenie średnie roczne.

Ozon (O3)

- `p8d932_3y` — percentyl 93,2 z trzyletniej serii maksimów dobowych stężenia ośmiogodzinnego kroczącego;
- `n8d120_3y` — liczba dni z przekroczeniami wartości $120\,\mu g/m^3$ przez stężenia ośmiogodzinne kroczące w roku uśrednione dla trzech lat;
- `p8d932` — percentyl 93,2 w rocznej serii maksimów dobowych stężenia ośmiogodzinnego kroczącego;
- `n8d120` — liczba dni z przekroczeniami wartości $120\,\mu g/m^3$ przez stężenia ośmiogodzinne kroczące w roku oceny;
- `aot40_5y` — parametr AOT40 liczony w godzinach między 8.00–20.00 czasu środkowoeuropejskiego w okresie 01.05–31.07 uśredniony dla pięciu lat;
- `aot40` — parametr AOT40 liczony w godzinach między 8.00–20.00 czasu środkowoeuropejskiego w okresie 01.05–31.07 w roku oceny;
- `nd180` — liczba dni w ciągu roku, w których jednogodzinne stężenie ozonu przekroczyło wartość $180\,\mu g/m^3$ i $240\,\mu g/m^3$.

AOT40 jest to wskaźnik określający zanieczyszczenie powietrza ozonem w okresie maj–lipiec, będący sumą różnic między stężeniem średnim godzinowym wyrażonym w $\mu g/m^3$ a wartością $80\,\mu g/m^3$, dla każdej godziny z zakresu 8.00–20.00 czasu środkowoeuropejskiego CET, dla której stężenie jest większe niż $80\,\mu g/m^3$ [64].

7.9.1 Benzo(a)piren w pyle zawieszonym PM10 (B(a)P)

- `yavg` — stężenie średnie roczne.

7.9.2 Arsen w pyle zawieszonym PM10

- `yavg` — stężenie średnie roczne.

7.10 Uruchamianie grafów

Są dwa sposoby uruchamiania grafów w bibliotece NodeAQ. Pierwszym z nich jest wywołanie funkcji uruchamiającej graf w trybie interaktywnym w REPL Julii, drugi zaś to uruchomienie grafu w trybie wsadowym za pomocą dodatkowego programu uruchamiającego `red.jl` zawartego w bibliotece.

7.10.1 Uruchamianie w trybie interaktywnym

W trybie interaktywnym w REPL Julii można uruchomić graf przetwarzania danych, wczytując go za pomocą funkcji `parse_flow` a następnie uruchamiając funkcją `render`, jak pokazano na listingu 7.8.

Listing 7.8: Uruchamianie grafu w trybie interaktywnym

```
julia> using NodeAQ
julia> flowset = parse_flow("plik.json")
julia> render(flowset)
```

Zaletą pracy z grafami przetwarzania danych w trybie interaktywnym jest możliwość eksperymentowania na żywo z jego wynikami. Wynik działania grafu można przypisać do zmiennej i dalej przetwarzać, korzystając z całego bogactwa funkcji i udogodnień języka Julia.

Dzięki temu niekoniecznie trzeba używać grafów jako zamkniętych całości. Można czerpać z nich dane częściowe. Jeśli jest np. jakiś trudny do zapamiętania proces, albo zawiera zbiory danych, których skompletowanie zajmuje dużo czasu, to można te procedury zawrzeć w grafie. Następnie można po nie sięgnąć w dowolnym momencie, nawet jeśli celem nie jest uzyskanie pełnego wyniku, a jedynie przetworzenie danych pośrednich.

Na rysunku 7.5 przedstawiono przykład dalszego przetwarzania danych uzyskanych z grafu w trybie interaktywnym. W przykładzie znaleziono stację pomiarową znajdującą się najbliżej Polsko-Japońskiej Akademii Technik Komputerowych.

```
julia> sensors = render()
Set{Sensor} with 34 elements:
  Sensor("OpKKozBSmial-O3-1g", Station("OpKKozBSmial", "OpKKozBSmial", NodeAQ.Location(50.349608, 18.236575, 181.0), "P…
  Sensor("DsJelGorOgin-O3-1g", Station("DsJelGorOgin", "DsJelGorOgin", NodeAQ.Location(50.913433, 15.765608, 345.0), "P…
  Sensor("MzWarKondrat-O3-1g", Station("MzWarKondrat", "MzWarKondrat", NodeAQ.Location(52.290864, 21.042458, 85.0), "PL…
  Sensor("SkGoluUjWody-O3-1g", Station("SkGoluUjWody", "SkGoluUjWody", NodeAQ.Location(50.621482, 20.614057, 270.0), "P…
  Sensor("ZpSzczAndrze-O3-1g", Station("ZpSzczAndrze", "ZpSzczAndrze", NodeAQ.Location(53.380975, 14.663347, 10.0), "PL…
  Sensor("MzWarWokalna-O3-1g", Station("MzWarWokalna", "MzWarWokalna", NodeAQ.Location(52.160772, 21.033819, 102.0), "P…
  Sensor("WpKaliSawick-O3-1g", Station("WpKaliSawick", "WpKaliSawick", NodeAQ.Location(51.74795, 18.049063, 140.0), "PL…
  Sensor("MzWarMeteo-O3-1g", Station("MzWarMeteo", "MzWarMeteo", NodeAQ.Location(52.281304, 20.963383, 95.0), "PL1401",…
  Sensor("KpBydWarszaw-O3-1g", Station("KpBydWarszaw", "KpBydWarszaw", NodeAQ.Location(53.134083, 17.995708, 49.0), "PL…
  ⋮

julia> PJATK = NodeAQ.Location(52.2238, 20.99346)
NodeAQ.Location(52.2238, 20.99346, 0.0)

julia> nearest_sensor = sort(collect(sensors), by = (x -> distance(PJATK, x.station.location)))[1];

julia> println("$(nearest_sensor.id): ($(nearest_sensor.station.location.lat), $(nearest_sensor.station.location.lon))")

MzWarChrosci-O3-1g: (52.207742, 20.906073)

julia>
```

Rysunek 7.5: Przykład dalszego interaktywnego przetwarzania danych uzyskanych z grafu, źródło: opracowanie własne

7.10.2 Uruchamianie w trybie wsadowym

Uruchamianie w trybie wsadowym polega na zapisaniu konfiguracji grafu w formacie `JSON`, tak jak bezpośrednio eksportuje grafy program Node-RED, a następnie podaniu go wśród argumentów wywołania pomocniczego programu uruchamiającego `red.jl` dostarczonego wraz z biblioteką w Julii. Program ten jest *wrapperem* dla funkcji opisanych w podrozdziale 7.10.1. Wczytuje on plik konfiguracyjny, buduje na jego podstawie graf przetwarzania danych i go uruchamia. Wynik działania grafu jest wypisywany na standardowe wyjście, ale jeśli w grafie są węzły o działaniach ubocznych, takich jak zapisanie wyników do plików, to skutkuje to zapisaniem danych na dysku.

Uruchomienie grafu jest realizowane za pomocą polecenia z listingu 7.9.

Listing 7.9: Uruchamianie grafu w trybie wsadowym

```
./red.jl --file plik-json-lub-url
```

Tryb wsadowy jest szczególnie przydatny dla procesów zamkniętych, nie wymagających dalszego interakcyjnego przetwarzania danych.

Jego wadą jest to, że uruchamianie grafu może zajmować więcej czasu, ponieważ każdorazowe wywołanie programu pomocniczego wymaga załadowania świeżego środowiska Julii i skompilowania użytych bibliotek.

Pliki konfiguracyjne w formacie JSON dobrze nadają się do przesyłania przez sieć, co pozwala na tworzenie zdalnych zbiorów dla gotowych grafów, na dedykowanym serwerze lub w popularnych serwisach takich, jak `http://github.com`.

8. Przypadki użycia

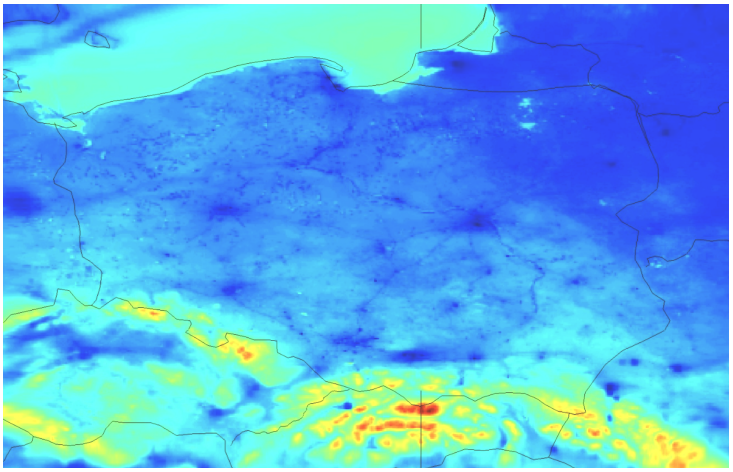
Ten rozdział zawiera przykłady użycia biblioteki NodeAQ w kontekście przetwarzania danych o zanieczyszczeniach atmosferycznych.

8.1 Przypadek użycia 1 — porównanie symulowanych i obserwowanych wartości stężenia ozonu

Niniejszy przykład ilustruje wykorzystanie biblioteki NodeAQ do porównania wartości symulowanych z obserwacjami.

Skrypt na podstawie zawartego w pliku `NetCDF` pola prognozowanego poziomu ozonu nad Polską, pobiera wartości chwilowe w punktach geograficznych stanowisk pomiarowych zdefiniowanych w odpowiednio sformatowanym pliku `csv`.

Przykładowa wizualizacja danych zawartych w pliku `NetCDF` jest przedstawiona na rysunku 8.1.

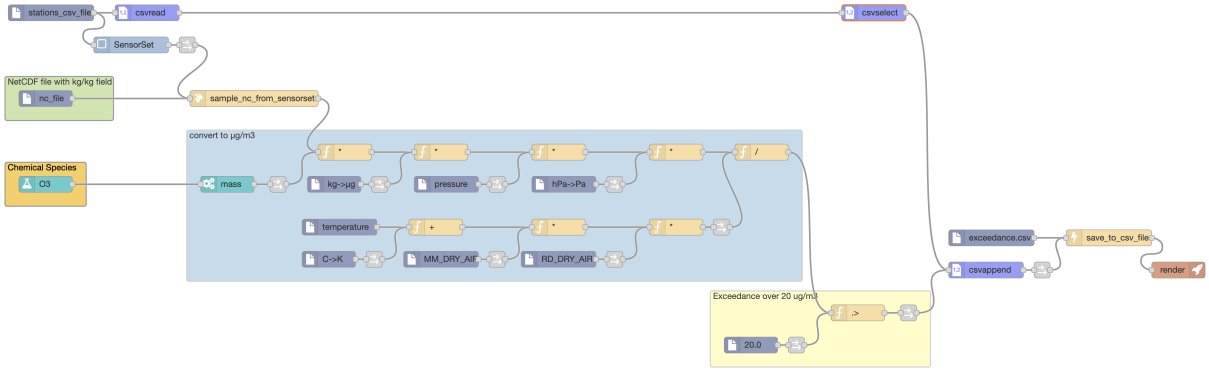


Rysunek 8.1: Przykładowe pole prognozowanego stężenia ozonu, źródło: opracowanie własne

Następnie dokonuje przeliczenia wartości wagowych stosunków zmieszania (kg/kg) do stężeń ozonu mierzonych w $\mu\mathrm{g}/\mathrm{m}^3$. W dalszym kroku skrypt znajduje te stanowiska, dla których wartość stężenia chwilowego przekracza określony próg. Końcowy wynik jest tabelą z nazwami stanowisk pomiarowych oraz nową kolumną wartości boolowskich wskazującą, czy wartość stężenia ozonu na danej stacji przekracza próg.

8.1.1 Graf przetwarzania

Rysunek 8.2 przedstawia wygląd grafu przetwarzania 1. Jego powiększona wersja znajduje się w dodatku A (rys. A.1).



Rysunek 8.2: Graf przetwarzania dla przykładu użycia 8.1, źródło: opracowanie własne

8.1.2 Składniki grafu przetwarzania

Graf przetwarzania 8.1.1 składa się z następujących węzłów:

`O3`

Węzeł z funkcją zwracającą obiekt typu `Variable` z danymi ozonu.

`stations_csv_file`

Zawiera ścieżkę do pliku `csv` z danymi stanowisk pomiarowych w konwencjonalnym formacie.

`csvread`

Służy do wczytywania wskazanego pliku `csv` i utworzenia obiektu tabelarycznego typu `DataFrame`.

`SensorSet`

Węzeł z funkcją zwracającą obiekt typu `SensorSet` (zobacz 7.7.4) z danymi stanowisk pomiarowych.

`nc_file`

Zawiera ścieżkę do pliku w formacie `NetCDF` z danymi wynikowymi symulacji.

`sample_nc_from_sensorset`

Węzeł z funkcją próbkującą pole dwuwymiarowe z pliku NetCDF w lokalizacjach geograficznych podanych stanowisk pomiarowych.

Grupa `convert_to_ugm3`

Grupa węzłów przeliczających wartości wagowych stosunków zmieszania ozonu w powietrzu (kg/kg) do wartości stężeń ($\mu\mathrm{g}/\mathrm{m}^3$):

`mass`

Węzeł typu `property` pobierający wartość atrybutu masy cząsteczkowej wybranego związku chemicznego (instancji typu `Variable` — zobacz 7.7.9), w tym wypadku ozonu.

Węzły `*`, `+`, `/`

Węzły typu `math` z operatorami arytmetycznymi dokonującymi wybranych obliczeń na wartościach liczbowych.

`kg->ug`

Mnożnik 10^9 przeliczający wartości z jednostek $\mathrm{kg/m^3}$ na $\mathrm{\mu g/m^3}$.

`pressure`

Węzeł `Constant` z wartością ciśnienia atmosferycznego normalnego, $1013.25\,\mathrm{hPa}$.

`hPa->Pa`

Mnożnik 100 przeliczający jednostki hPa (hektopaskale) na Pa (paskale).

`temperature`

Węzeł `Constant` z wartością temperatury normalnej, $20.0\,^\circ\mathrm{C}$.

`C->K`

Przeliczenie stopni Celsjusza na stopnie Kelwina (dodanie $273.15\,\mathrm{K}$).

`MM_DRY_AIR`

Węzeł `Constant` z ustawioną wartością masy molowej suchego powietrza ($28.97\,\mathrm{g/mol}$).

`RD_DRY_AIR`

Węzeł `Constant` z ustawioną wartością stałej gazowej suchego powietrza ($287.05\,\mathrm{J/(K\,kg)}$).

Grupa `exceedance_over_threshold`

Węzły obliczające, czy wartość stężenia ozonu przekracza określony próg.

threshold

Węzeł typu `Input`, wartość progu stężenia ozonu ($80.0\,\mathrm{\mu g/m^3}$).

`.>`

Operator większości w wersji rozgłaszającej oznaczający porównanie każdej wartości wektora z wartością skalarną i zwrócenie wektora wartości logicznych.

`csvselect`

Wybranie kolumny `station_id` z tabeli stanowisk pomiarowych.

`csvappend`

Dodanie kolumny `exceedance` do tabeli stanowisk pomiarowych.

`exceedance.csv`

Węzeł `Input` z nazwą pliku wyjściowego.

`save_to_csv_file`

Węzeł zapisujący wynikową tabelę do pliku `csv`.

`render`

Węzeł `render` oznaczający na koniec grafu przetwarzania (zobacz 6.8).

8.1.3 Wynik działania grafu 8.1.1

Na rysunku 8.3 przedstawiono wynik działania grafu 8.1.1. Jest to tabela z nazwami stanowisk pomiarowych i kolumną wskazującą, czy wartość stężenia ozonu przekracza ustalony próg.

station_id InlineStrings.String15	over_80_ug/m3 Bool
DsCzerStraza	false
DsJelGorOgin	false
KpBydWarszaw	false
KpCiechTezni	false
LbBiaPodOrze	true
LbFlorianRPN	true
LdGajewUjWod	false
LdLodzCzerni	false
LuGorzKosGdy	false
LuSmolBytnic	true
MpKaszowLisz	false
MpKrakBujaka	true
MzWarChrosci	false
MzWarKondrat	true
MzWarMeteo	false
MzWarWokalna	true
OpKKozBSmial	true
OpOlesSlowac	false
PdBialPPiech	true
PdBorsukowiz	true
PkJasloSikor	false
PkKrempnaMPN	false
PmGdaLeczkow	true
PmGdyPorebsk	false
SkGoluUjWody	true
SkKielTargow	false
SlBielKossak	false
SlCzestoBacz	false
WmElbBazynsk	false
WmElkStadion	true
WpBoroDrapal	false
WpKaliSawick	true
ZpKoszChopin	false
ZpSzczAndrze	false

Rysunek 8.3: Wynik działania grafu 8.1.1, źródło: opracowanie własne

8.2 Przypadek użycia 2 — obliczenie agregacji dot. stężeń zanieczyszczeń

Niniejszy przykład ilustruje wykorzystanie biblioteki NodeAQ do obliczenia statystyk agregacyjnych dla modelowanych i obserwowanych stężeń zanieczyszczeń ozonu, oraz sporządzenia tabeli z błędami względnymi dla wybranych stacji pomiarowych.

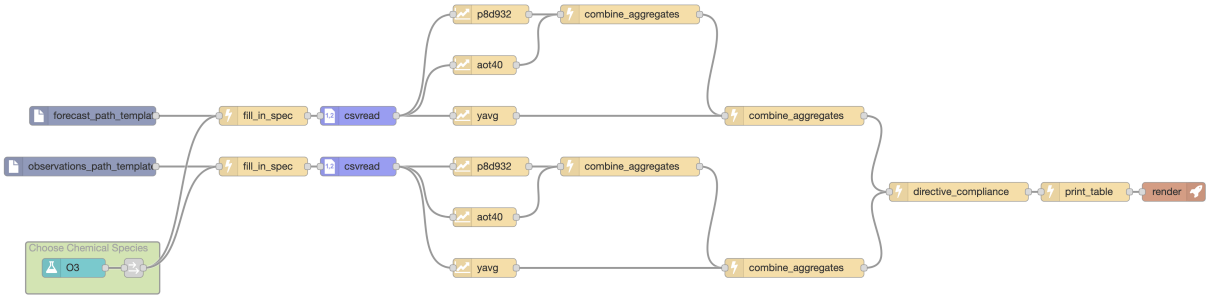
Tabela ta służy do sprawdzenia, czy modelowanie zanieczyszczeń powietrza spełnia kryteria jakościowe zawarte w rozporządzeniach Ministerstwa Klimatu i Środowiska [64].

Skrypt odczytuje z plików `csv` roczne serie czasowe stężeń ozonu dla wybranych stacji pomiarowych i dwóch rodzajów danych: symulacji i rzeczywistych pomiarów. Następnie oblicza dla każdej z tych dwóch

grup wybrane statystyki agregacyjne, takie jak średnia roczna, 93,2 percentyl z rocznej serii maksimów dobowych stężenia ośmiogodzinnego kroczącego oraz wskaźnik AOT40. Połączone tabele statystyk są ze sobą porównywane w celu obliczenia wyjściowej tabeli błędów względnych między symulacją a obserwacjami na wybranych stacjach pomiarowych. Tabela ta zostaje wyświetlona na standardowym wyjściu.

8.2.1 Graf przetwarzania

Rysunek 8.4 przedstawia wygląd grafu przetwarzania 2. Jego powiększona wersja znajduje się w dodatku A (rys. A.2).



Rysunek 8.4: Graf przetwarzania dla przykładu użycia 8.2, źródło: opracowanie własne

8.2.2 Składniki grafu przetwarzania

Graf przetwarzania 8.2.1 składa się z następujących węzłów:

`O3`

Węzeł skojarzony z obiektem typu `Variable` (zobacz 7.7.9) z danymi ozonu.

`forecast_path_template`

Węzeł z szablonem nazw dla plików serii czasowych prognozy.

`observations_path_template`

Węzeł z szablonem nazw plików serii czasowych obserwacji.

`fill_in_spec`

Oba węzły `filename` formatują szablon, wstawiając do niego odpowiedni kod nazwy związku chemicznego, który może różnić się od nazwy wyświetlanej.

`csvread`

Węzły `csvread` wczytują plik `csv` o odpowiedniej nazwie i tworzą na jego podstawie obiekt tabelaryczny typu `DataFrame`.

`p8d932`

Węzeł `aggregate` z ustawioną opcją 93,2 percentyla z rocznej serii maksimów dobowych stężenia ośmiogodzinnego kroczącego.

`aot40`

Węzeł `aggregate` z ustawioną opcją wskaźnika AOT40 z rocznej serii stężeń ozonu.

`yavg`

Węzeł `aggregate` z ustawioną opcją średniej rocznej stężenia ozonu.

`combine_aggregates`

Węzeł łączący wynikowe agregacje na podstawie wspólnego indeksu wejściowych tabel. Tabela wynikowa zawiera indeks i kolumny obu tabel wejściowych.

`directive_compliance`

Węzeł obliczający błędy względne modelu względem obserwacji dla równoważnych agregacji. Wartości błędów są wyrażone liczbami ułamkowymi.

`print_table`

Węzeł z funkcją wyświetlającą tabelę z błędami względnymi.

8.2.3 Wynik działania grafu 8.2.1

Wynik działania grafu przetwarzania jest tabelą z błędami względnymi modelu względem obserwacji:

id String	rel_aot40 Float64	rel_p8d932 Float64	rel_yavg Float64
DsCzerStraza	-0.46	-0.14	-0.18
DsJelGorOgin	-0.45	-0.1	0.089
DsKlodzSzkol	0.01	-0.034	0.23
DsLegAlRzecz	0.17	-0.046	0.11
DsOlawZolnAK	-0.016	-0.034	0.087
DsOsieczow21	-0.11	-0.014	0.12
DsSniezkaObs	0.19	-0.059	-0.082
DsStrzegomMOB	-0.17	-0.02	-0.022
DsTrzebniMaj	-0.049	-0.045	0.012
DsWalbrzWyso	-0.21	-0.042	0.17
DsWrocBartni	0.2	-0.022	0.011
DsWrocWybCon	0.29	-0.017	0.015
KpBydWarszaw	0.47	0.056	0.13
KpCiechTezni	-0.46	-0.083	0.043
KpKoniczynka	0.14	0.049	0.044
KpToruDziewu	-0.016	-0.001	0.044
KpWloclKalis	-0.31	-0.072	0.021
KpZielBoryTu	-0.36	-0.075	0.049
⋮	⋮	⋮	⋮

Rysunek 8.5: Wynik działania grafu 8.2.1, źródło: opracowanie własne

Tego typu dane tabelaryczne mogą być użyte do dalszej analizy lub jako wsad do raportów.

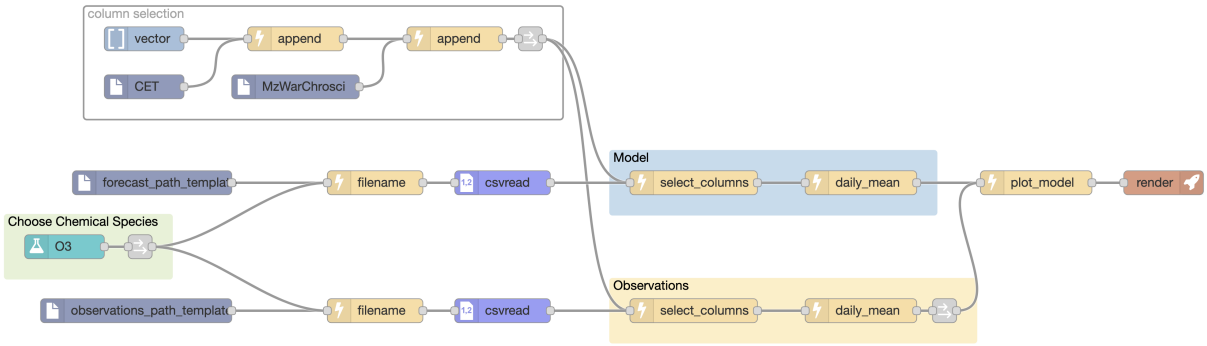
8.3 Przypadek użycia 3 — wykres porównawczy średnich dobowych stężeń ozonu

Niniejszy przykład ilustruje wykorzystanie biblioteki NodeAQ do utworzenia wykresu porównania średnich dobowych stężeń ozonu dla obserwacji i prognozy w lokalizacji konkretnej stacji pomiarowej.

Skrypt odczytuje z dwóch plików `csv` roczne serie czasowe stężeń ozonu, odpowiednio dla obserwacji i prognozy. Następnie wybiera z nich serie czasowe na wybranej stacji pomiarowej. W dalszym kroku zostaje wykonane obliczenie średnich dobowych osobno dla obserwacji i prognozy, a uzyskane wartości zostają narysowane na wykresie porównawczym.

8.3.1 Graf przetwarzania

Rysunek 8.6 przedstawia wygląd grafu przetwarzania 3. Jego powiększona wersja znajduje się w dodatku A (rys. A.3).



Rysunek 8.6: Graf przetwarzania dla przykładu użycia 8.3, źródło: opracowanie własne

8.3.2 Składniki grafu przetwarzania

Graf przetwarzania 8.3.1 składa się z następujących węzłów:

`O3`

Węzeł skojarzony z obiektem typu `Variable` (zobacz 7.7.9) z danymi ozonu.

`forecast_path_template`

Węzeł z szablonem nazw dla plików serii czasowych prognozy.

`observations_path_template`

Węzeł z szablonem nazw plików serii czasowych obserwacji.

`fill_in_spec`

Oba węzły `filename` formatują szablon, wstawiając do niego odpowiedni kod nazwy związku chemicznego, który może różnić się od nazwy wyświetlanej.

`csvread`

Węzły `csvread` wczytują plik `csv` o odpowiedniej nazwie i tworzą na jego podstawie obiekt tabelaryczny typu `DataFrame`.

Grupa `column_selection`

Grupa węzłów służących do wyboru odpowiednich kolumn z tabeli.

`vector`

Węzeł zwracający pustą instancję typu `Vector`.

`CET`

Dodaje do wektora ciąg znaków `CET`, dla nazwy kolumny indeksu czasowego serii czasowej.

`MzWarChrosci`

Węzeł typu `Input` z nazwą konkretnej stacji pomiarowej.

`append`

Węzły z funkcją dołączająca podaną wartość do wektora.

Grupy `model` **i** `observations`

Grupy węzłów wybierających odpowiednie kolumny serii czasowych i obliczające średnie dobowe.

`select_columns`

Węzeł wybierający z tabeli kolumny podane w wektorze kolumn i zwracający nową tabelę z indeksem czasowym i pojedynczą serią czasową.

`daily_mean`

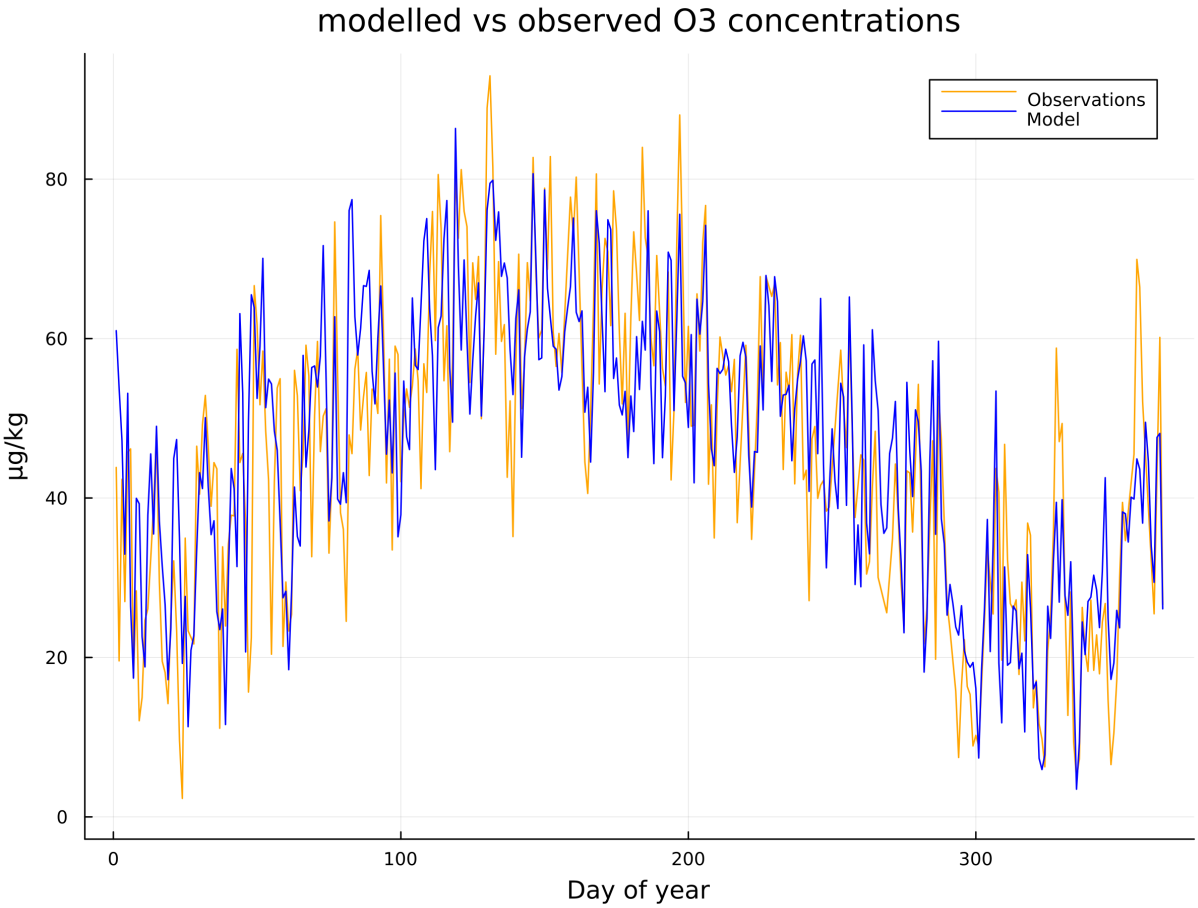
Węzeł z funkcją obliczającą średnie dobowe i zwracającą tabelę typu `DataFrame` z serią czasową wartości dobowych.

`plot_model`

Węzeł z funkcją rysującą wykres porównawczy wyników modelu do obserwacji

8.3.3 Wynik działania grafu przetwarzania 8.3.1

Wynik działania grafu przetwarzania jest wykresem porównawczym dobowych wartości stężenia ozonu dla obserwacji i prognozy w lokalizacji wybranej stacji pomiarowej.



Rysunek 8.7: Wynik działania grafu 8.3.1, źródło: opracowanie własne

9. Podsumowanie

Niniejszy rozdział stanowi podsumowanie zaprezentowanego rozwiązania do wspomagania analiz danych związanych z zanieczyszczeniami atmosfery.

9.1 Zalety i wady rozwiązania

Biblioteka do tworzenia grafów przetwarzania danych, opisana w niniejszej pracy, posiada następujące zalety:

- wizualne przedstawienie sposobu przetwarzania danych, następstwa kroków, pochodzenia danych wejściowych, szczególnie w przypadku projektów cyklicznych powtarzanych z małą częstotliwością (np. co rok) jest bardzo korzystne;
- drzewa przetwarzania stanowią od razu wizualną dokumentację projektu, na podobnej zasadzie co samokomentujący się kod; w przypadku zmiany osoby odpowiedzialnej za projekt, nowy analityk danych może szybko zrozumieć, co się dzieje, bez konieczności czytania całego kodu;
- zwolnienie użytkownika z pamiętania licznych stałych, ścieżek do plików, wartości progowych, i zastąpienie stałych w kodzie przez węzły z predefiniowanymi wartościami możliwymi do wyboru z listy;
- możliwość dodawania wizualnych wskazówek, w którym miejscu osoby uruchamiające drzewo mogą dokonać zmian;
- możliwość przesyłania drzew przetwarzania jako zwykłego tekstu, w mailu lub przez komunikatory, metodą wytnij-wklej;
- możliwość zastosowania narzędzia diff, do sprawdzenia jak zmienił się graf przetwarzania, nawet jeśli jest to nieoczywiste na jego wizualnej reprezentacji; podobnie można przechowywać drzewa i śledzić przyrostowe zmiany za pomocą oprogramowania git;
- odseparowanie danych od kodu pozwala na zmiany algorytmów obliczeniowych bez zmiany grafu przetwarzania; nie jest to zachowanie wymuszone, ale możliwe; jeśli np. w jakimś roku zmieni się dopuszczalny poziom jakiejś substancji w powietrzu, zmiana musi być dokonana tylko w kodzie liczącym przekroczenia, zaś graf pozostaje taki sam, chyba że twórca grafu zdecyduje, że ma być inaczej;
- dowolna granulacja funkcji w grafie przetwarzania, od prostych funkcji jednoargumentowych, po złożone skomplikowane algorytmy.

Wśród wad rozwiązania można wymienić:

- w przypadku błędów w grafie przetwarzania jest potrzebna znajomość Julii, aby zrozumieć komunikaty błędów;
- nie w pełni interakcyjny interfejs graficzny, brak podpowiedzi ze strony interfejsu co do kolejności lub poprawności łączenia określonych węzłów ze sobą;
- konieczność stosowania specjalnego węzła etykietującego poprzednie dane wejściowe, jako należące do drugiego argumentu funkcji;

- konieczność pamiętania o konsekwencjach używania funkcji mutujących i struktur mutowalnych w drzewie (np. gdy wyjście funkcji mutującej jest użyte w grafie kilkukrotnie);
- tworzenie poprawnie działających grafów o dużej liczbie składników może być czasochłonne.

9.2 Plan rozwoju

Jednym z najistotniejszych kierunków rozwoju opisywanej biblioteki jest wyposażenie jej w bardziej interaktywny interfejs graficzny do tworzenia grafów. Obecne rozwiązanie do komponowania drzew przetwarzania w trybie wsadowym nie zapewnia wystarczająco dużo wskazówek na temat łączenia poszczególnych elementów. Nie informuje też o możliwych błędach tych połączeń.

Błędy uruchomieniowe są zgłaszane dopiero na etapie uruchomienia przetwarzania grafu, jednak niektóre usterki w organizacji grafu są możliwe do wychwycenia i zasygnalizowania wcześniej. Na etapie budowy biblioteki testowano użycie natywnych graficznych pakietów Julii, jednak są one jeszcze w fazie rozwoju i nie oferują jeszcze wystarczająco rozwiniętej funkcjonalności. Użycie takiej biblioteki dawałoby przewagę nad zewnętrznymi rozwiązaniami, ponieważ interfejs uzyskałby bezpośredni dostęp do przestrzeni nazw i stanu aplikacji.

Dużym ułatwieniem w budowie grafów mogą być węzły z zaznaczonymi kilkoma wejściowymi portami, osobno dla każdego argumentu wewnętrznej funkcji. Wcześniej w podrozdziale 6.7 wspomniano o tym, że wejście stanowiące drugi argument, musi przechodzić przez specjalny węzeł `Arg2`, aby przetwarzająca go funkcja zastosowała odpowiednią kolejność argumentów. Możliwość podpięcia krawędzi bezpośrednio do odpowiedniego portu mogłaby wpłynąć na przejrzystość grafu. Z drugiej strony zbyt wiele portów, z powrotem mogłoby popsuć czytelność i byłoby trudniejsze w użyciu przez mniej doświadczonych użytkowników. Kluczowe jest znalezienie balansu między jednym a drugim.

Ważnym elementem rozwoju jest współpraca z użytkownikami przy ustalaniu zestawu funkcji, które powinny być wyróżnione w bibliotece i w interfejsie. Funkcyjny charakter języka Julia pozwala na łatwe dołączanie nowych operacji, jednak taka swoboda może prowadzić do nadmiernego rozrostu funkcjonalności i chaotycznego interfejsu. Uzyskanie opinii od użytkowników pozwoli na skupienie się na najważniejszych funkcjach i uniknięcie nadmiaru opcji.

Jeśli chodzi o bibliotekę uruchomieniową, to większy nacisk powinien być położony na funkcje mutujące, które mogą potencjalnie przynosić skutki uboczne. Potrzebne jest też dogłębne zbadanie bezpieczeństwa funkcji, które mogą być wywoływane w grafie przetwarzania, i zablokowanie wywoływania funkcji umożliwiających działania na systemie plików.

Duża część pracy nad biblioteką powinna być poświęcona budowaniu kolejnych przykładów użycia wziętych bezpośrednio z praktyki analitycznej. Uzupełnianie biblioteki o kolejne operatory pozwoli na zwiększanie użyteczności narzędzia.

9.3 Wnioski końcowe

Zaproponowane podejście grafowe do tworzenia procesów analiz danych atmosferycznych wydaje się bardzo obiecujące. Pomimo wad i ograniczeń prototypu, wstępne testy w środowisku roboczym wykazały, że takie podejście jest użyteczne w praktyce. Uzyskanie wizualnej reprezentacji wykonywanych analiz przekłada się na ich rozumienie i stanowi od razu dokumentację projektu. Możliwość uruchamiania zapisanych grafów zarówno bezpośrednio w interaktywnym środowisku Julii, jak i za pomocą osobnego programu, otwiera wiele dróg wykorzystania tworzonych grafów. Rozwój biblioteki wydaje się być wartym kontynuowania i może się przyczynić do rozwiązania problemu chaotycznego środowiska narzędziowego

używanego w analizach danych atmosferycznych. Wymaga to jednak dalszych prac nad interfejsem graficznym oraz przystępnością i bezpieczeństwem biblioteki.

10. Bibliografia i spisy

Prace cytowane

[1] University Corporation for Atmospheric Research. *Network Common Data Form*. URL: https://www.unidata.ucar.edu/software/netcdf/. (dostęp: 2024-06-01).

[2] Jeff Bezanson i in. „Julia A Fresh Approach to Numerical Computing". W: *SIAM Review* 59.1 (2017), s. 65–98. DOI: 10.1137/141000671.

[3] Jeff Bezanson i in. *Why We Created Julia*. URL: https://julialang.org/blog/2012/02/why-we-created-julia/. (dostęp: 2024-06-01).

[4] Joris Van den Bossche i in. *GeoPandas*. URL: https://geopandas.org. (dostęp: 2024-06-13).

[5] Seth Bromberger. *LightGraphs.jl*. URL: https://juliagraphs.org/LightGraphs.jl/stable/. (dostęp: 2024-06-01).

[6] Bradley Carman. *Modeling Toolkit Designer*. URL: https://github.com/bradcarman/ModelingToolkitDesigner.jl. (dostęp: 2024-06-01).

[7] Yves Chartier. *xrec Documentation*. URL: https://collaboration.cmc.ec.gc.ca/science/si/eng/si/utilities/xrecpdf/index.html. (dostęp: 2024-05-15).

[8] Julia Computing. *Broadcast and vectorization*. URL: https://docs.julialang.org/en/v1/base/arrays/#Broadcast-and-vectorization. (dostęp: 2024-06-25).

[9] Julia Computing. *Calling C and Fortran Code*. URL: https://docs.julialang.org/en/v1/manual/calling-c-and-fortran-code/. (dostęp: 2024-06-25).

[10] Julia Computing. *Constructors*. URL: https://docs.julialang.org/en/v1/manual/constructors/. (dostęp: 2024-06-25).

[11] Julia Computing. *Conversion and Promotion*. URL: https://docs.julialang.org/en/v1/manual/conversion-and-promotion/. (dostęp: 2024-06-25).

[12] Julia Computing. *Julia Micro-Benchmarks*. URL: https://julialang.org/benchmarks/. (dostęp: 2024-06-01).

[13] Julia Computing. *Metaprogramming*. URL: https://docs.julialang.org/en/v1/manual/metaprogramming/. (dostęp: 2024-06-25).

[14] Julia Computing. *Method Ambiguities*. URL: https://docs.julialang.org/en/v1/manual/methods/#man-ambiguities. (dostęp: 2024-06-25).

[15] Julia Computing. *Method specializations*. URL: https://docs.julialang.org/en/v1/manual/methods/#man-method-specializations. (dostęp: 2024-06-25).

[16] Julia Computing. *Methods*. URL: https://docs.julialang.org/en/v1/manual/methods/#Methods. (dostęp: 2024-06-25).

[17] Julia Computing. *Parallel Computing*. URL: https://docs.julialang.org/en/v1/manual/parallel-computing/. (dostęp: 2024-06-25).

[18] Julia Computing. *Pkg*. URL: https://docs.julialang.org/en/v1/stdlib/Pkg/. (dostęp: 2024-06-01).

[19] Julia Computing. *Redefining Methods*. URL: https://docs.julialang.org/en/v1/manual/methods/#Redefining-Methods. (dostęp: 2024-06-01).

[20] Julia Computing. *Running External Programs*. URL: https://docs.julialang.org/en/v1/manual/running-external-programs/. (dostęp: 2024-06-25).

[21] Julia Computing. *The Julia Language*. URL: https://docs.julialang.org/en/v1/#man-introduction. (dostęp: 2024-06-01).

[22] Julia Computing. *Types*. URL: https://docs.julialang.org/en/v1/manual/types/. (dostęp: 2024-06-25).

[23] Julia Computing. *Unicode Input*. URL: https://docs.julialang.org/en/v1/manual/unicode-input/. (dostęp: 2024-06-25).

[24] Neovim Contributors. *Neovim*. URL: https://neovim.io. (dostęp: 2024-06-25).

[25] Thomas H. Cormen i in. *Introduction to Algorithms, Fourth Edition*. Cambridge, Massachusetts London, England: The MIT Press, 2022. ISBN: 9780262367509.

[26] NumPy Developers. *How to extend NumPy*. URL: https://numpy.org/doc/stable/user/c-info.how-to-extend.html. (dostęp: 2024-06-02).

[27] NumPy Developers. *NumPy*. URL: https://numpy.org. (dostęp: 2024-06-02).

[28] SciKit-Learn Developers. *scikit-learn*. URL: https://scikit-learn.org. (dostęp: 2024-06-01).

[29] xarray Developers. *xarray*. URL: http://xarray.pydata.org/en/stable/. (dostęp: 2024-06-13).

[30] Jesse Duffield. *LazyGit*. URL: https://github.com/jesseduffield/lazygit. (dostęp: 2024-06-25).

[31] ECCC-ASTD-MRD. *librmn*. URL: https://github.com/ECCC-ASTD-MRD/librmn. (dostęp: 2024-05-15).

[32] Facebook. *PyTorch*. URL: https://pytorch.org. (dostęp: 2024-06-12).

[33] Apache Software Foundation. *Differences with Java*. URL: https://groovy-lang.org/differences.html. (dostęp: 2024-06-25).

[34] OpenJS Foundation. *npm*. URL: https://www.npmjs.com. (dostęp: 2024-06-01).

[35] OpenJS Foundation i contributors. *Node-RED*. URL: https://nodered.org. (dostęp: 2024-05-31).

[36] Python Software Foundation. *Thread State and the Global Interpreter Lock*. URL: https://docs.python.org/3/c-api/init.html#thread-state-and-the-global-interpreter-lock. (dostęp: 2024-06-13).

[37] GitHub. *GitHub*. URL: https://github.com. (dostęp: 2024-06-25).

[38] Google. *TensorFlow*. URL: https://www.tensorflow.org. (dostęp: 2024-06-12).

[39] Larry Hardesty. *MIT-developed programming language Julia debuts at JuliaCon*. URL: https://news.mit.edu/2018/mit-developed-julia-programming-language-debuts-juliacon-0827. (dostęp: 2024-06-25).

[40] John Harris, Jeffry Hirst i Michael Mossinghoff. *Combinatorics and Graph Theory*. New York, NY: Springer, 2008. ISBN: 9780387797106.

[41] Tim Holy. *Revise.jl*. URL: https://timholy.github.io/Revise.jl. (dostęp: 2024-06-01).

[42] John D. Hunter. *Matplotlib*. URL: https://matplotlib.org. (dostęp: 2024-05-13).

[43] Julia Interop. *JavaCall.jl*. URL: https://github.com/JuliaInterop/JavaCall.jl. (dostęp: 2024-06-25).

[44] Julia Interop. *MathLink.jl*. URL: https://github.com/JuliaInterop/MathLink.jl. (dostęp: 2024-06-26).

[45] Julia Interop. *RCall.jl*. URL: https://github.com/JuliaInterop/RCall.jl. (dostęp: 2024-06-26).

[46] JGraph. *draw.io*. URL: https://www.diagrams.net. (dostęp: 2024-06-25).

[47] JuliaGeo. *NetCDF.jl*. URL: https://juliageo.org/NetCDF.jl/stable/. (dostęp: 2024-06-01).

[48] JuliaGraphs. *Graphs.jl*. URL: https://juliagraphs.org/Graphs.jl/stable/. (dostęp: 2024-06-01).

[49] JuliaPy. *PyCall.jl*. URL: https://github.com/JuliaPy/PyCall.jl. (dostęp: 2024-06-25).

[50] JuliaStats. *Statistics.jl*. URL: https://juliastats.org/Statistics.jl/stable/. (dostęp: 2024-06-01).

[51] Arthur B. Kahn. „Topological sorting of large networks". W: *Communications of the ACM* 5.11 (1962), s. 558–562. DOI: 10.1145/368996.369025.

[52] Bogumił Kamiński. *DataFrames.jl*. URL: https://dataframes.juliadata.org/stable/. (dostęp: 2024-06-01).

[53] Lawrence Livermore National Laboratory. *Community Data Analysis Tools*. URL: https://cdat.llnl.gov. (dostęp: 2024-05-31).

[54] MIT Lincoln Laboratory. *Wilkinson Prize Goes to Developers of Flexible Julia Programming Language*. URL: https://www.ll.mit.edu/news/wilkinson-prize-goes-developers-flexible-julia-programming-languagel. (dostęp: 2024-06-25).

[55] Andreas Lobinger. *How bad is type piracy, actually?* URL: https://discourse.julialang.org/t/how-bad-is-type-piracy-actually/37913/11. (dostęp: 2024-06-25).

[56] Max Planck Institute for Meteorology. *Climate Data Operators*. URL: https://code.mpimet.mpg.de/projects/cdo. (dostęp: 2024-05-28).

[57] Microsoft. *Visual Studio Code*. URL: https://code.visualstudio.com. (dostęp: 2024-06-01).

[58] Robert Moss. *BeautifulAlgorithms.jl*. URL: https://github.com/mossr/BeautifulAlgorithms.jl. (dostęp: 2024-06-01).

[59] Nature. *Why Jupyter is data scientists' computational notebook of choice*. URL: https://www.nature.com/articles/d41586-018-07196-1. (dostęp: 2024-06-25).

[60] MIT News. *Computing our climate future*. URL: https://news.mit.edu/2022/computing-our-climate-future-0413. (dostęp: 2024-06-01).

[61] Andreas Noack. *CSV.jl*. URL: https://csv.juliadata.org/stable/. (dostęp: 2024-06-01).

[62] Met Office. *Cartopy*. URL: https://scitools.org.uk/cartopy/docs/latest/. (dostęp: 2024-06-13).

[63] Chris Rackauckas. *ModelingToolkit.jl*. URL: https://mtk.sciml.ai/stable/. (dostęp: 2024-06-01).

[64] „Rozporządzenie Ministra Klimatu i Środowiska z dnia 15 lutego 2023 r. w sprawie zakresu i sposobu przekazywania informacji dotyczących zanieczyszczenia powietrza". W: *Dziennik Ustaw 2023 poz. 350* (2023).

[65] Robert B. Schmunk. *Panoply*. URL: https://www.giss.nasa.gov/tools/panoply/. (dostęp: 2024-05-14).

[66] Peter Seibel. *Practical Common Lisp*. New York, NY: Apress, 2005. ISBN: 9781590592397.

[67] Steven S. Skiena. *The Algorithm Design Manual*. New York, NY: Springer, 2008. ISBN: 9781848000698.

[68] Marta Szachniuk. *Algorytmy grafowe*. URL: http://www.cs.put.poznan.pl/mszachniuk/mszachniuk_files/lab_aisd/Szachniuk-ASD-t3.pdf. (dostęp: 2024-06-03).

[69] Linus Torvalds i in. *Git*. URL: https://git-scm.com. (dostęp: 2024-06-25).

[70] Grzegorz Walaszek. *Algorytmy i struktury danych*. URL: https://eduinf.waw.pl/inf/alg/001_search/index.php. (dostęp: 2024-06-03).

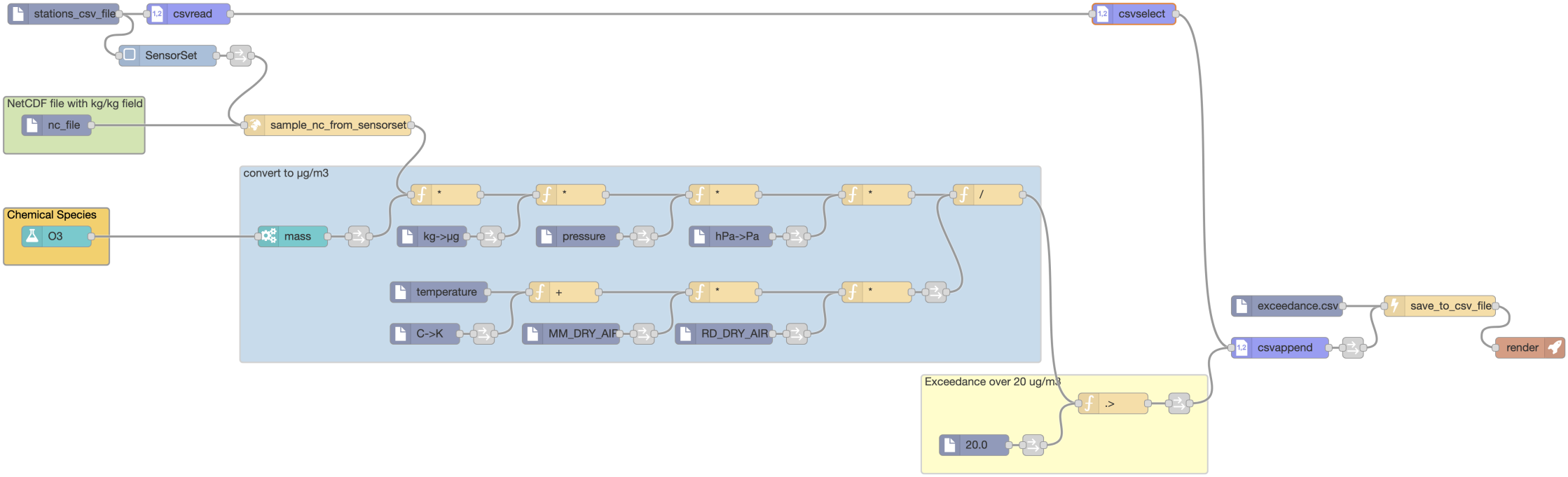
[71] Jeff Whitaker. *PyProj*. URL: https://pyproj4.github.io/pyproj/stable/. (dostęp: 2024-06-13).

[72] Charlie Zender. *NCO*. URL: http://nco.sourceforge.net. (dostęp: 2024-06-07).

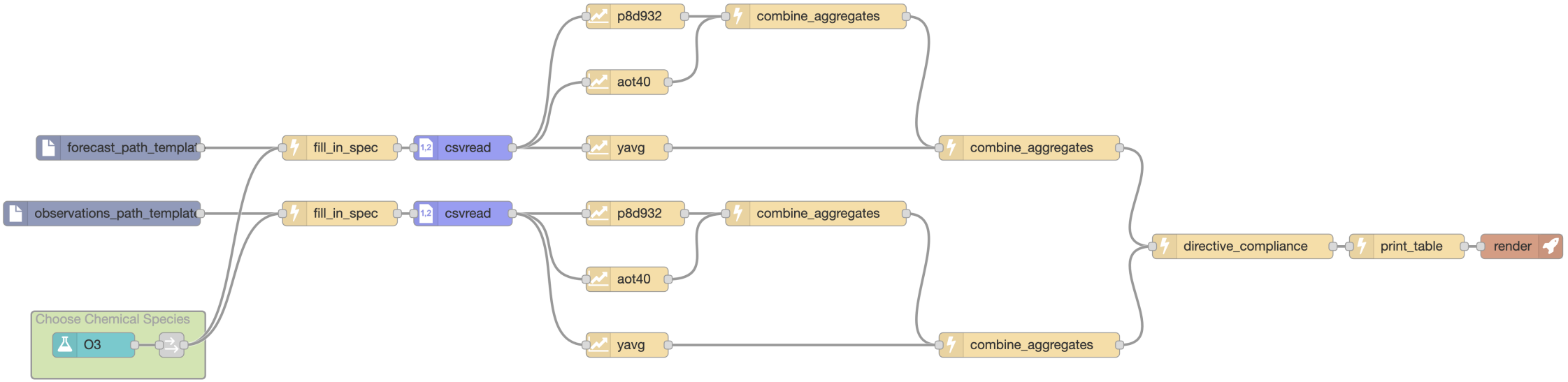
Spis rysunków

Spis listingów

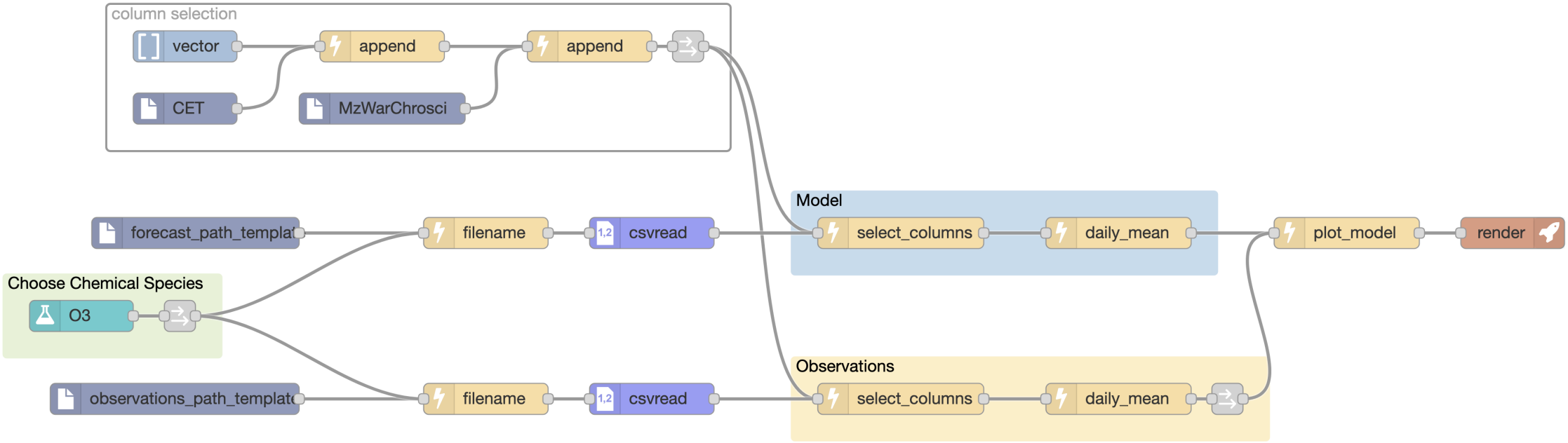
A. Dodatek



Rysunek A.1: Graf przetwarzania dla przykładu użycia 8.1, źródło: opracowanie własne



Rysunek A.2: Graf przetwarzania dla przykładu użycia 8.2, źródło: opracowanie własne



Rysunek A.3: Graf przetwarzania dla przykładu użycia 8.3, źródło: opracowanie własne