Podstawy informatyki

dr inż. Sławomir Koczubiej

Politechnika Świętokrzyska
Wydział Zarządzania i Modelowania Komputerowego
Katedra Technologii Informatycznych

(3 marca 2024)

Kontakt

Budynek C, pokój 3.26
sk@tu.kielce.pl

Materiały do pobrania, aktualności, terminy zaliczeń

http://staff.tu.kielce.pl/sk

Organizacja wykładów

- Wykłady są nieobowiązkowe (zgodnie z postanowieniami regulaminu), ale...
- Na wykłady czasem warto zajrzeć.

Warunki zaliczenia wykładu

Test zaliczeniowy po zakończeniu wykładów.

Organizacja laboratoriów

- Zajęcia laboratoryjne są obowiązkowe.
- Dopuszcza się jedną nieobecność.
- Większa liczba nieobecności powoduje zmniejszenie oceny do niedostatecznej włącznie (3 lub więcej nieobecności).
- W przypadku usprawiedliwionej nieobecności zajęcia można odrobić z inną grupą (jeśli istnieje taka możliwość).

Warunki zaliczenia laboratoriów

Wykonanie ćwiczeń i zaliczenie sprawdzianów kontrolnych.

Treść wykładów

- Wprowadzenie i istota informatyki. Elementy kodowania informacji, systemy liczbowe, zapis liczb w systemie komputerowym.
- Pojęcie algorytmu. Algorytmy sumowania i sortowania danych. Wprowadzenie do metod numerycznych.
- Semantyka i syntaktyka języka programowania. Podstawowe typy danych. Wyrażenia algebraicznych i logiczne. Instrukcje wejścia/wyjścia, instrukcje sterujące. Podprogramy, przekazywanie parametrów do podprogramów. Zasięg zmiennych. Typ plikowy, obsługa plików.
- Komputerowe wspomaganie obliczeń inżynierskich – oprogramowanie typu CAS (*Computer Algebra System*). Obliczenia numeryczne i symboliczne z zakresu analizy matematycznej, algebry i statystyki.

Treść laboratoriów

- Oprogramowanie użytkowe typu CAS – środowisko, zapis wyrażeń arytmetycznych i podstawowych funkcji. Generowanie wykresów funkcji. Operacje na wektorach i macierzach. Rozwiązywanie równań, układów równań, nierówności. Analiza statystyczna.
- Definiowanie prostych algorytmów. Zmienne, operatory i wyrażenia. Instrukcja przypisania.
- Komunikacja z użytkownikiem: instrukcje wejścia/wyjścia. Instrukcje sterujące: instrukcja warunkowa i wyboru. Iteracyjne instrukcje sterujące – pętle.
- Programowanie z wykorzystaniem typu tablicowego i typów pochodnych. Definiowanie własnych funkcji i procedur. Parametry procedur i funkcji, sposoby ich przekazywania. Programowanie z wykorzystaniem plików tekstowych i binarnych.
- Przykłady obliczeń symbolicznych w systemach CAS.

Literatura

- Karpisz D., Wojnar L. *Podstawy informatyki.* Wydawnictwo Politechniki Krakowskiej, Kraków 2005.
- Null L., Lobur J. *Struktura organizacyjna i architektura systemów komputerowych.* Helion, Gliwice 2004.
- Alagic S., Arbib M. *Projektowanie programów poprawnych i dobrze zbudowanych.* WNT, Warszawa 1982.
- Wirth N. *Wstęp do programowania systematycznego.* WNT, Warszawa 1999.
- Wirth N. *Algorytmy + struktury danych = programy.* WNT, Warszawa 2001.
- Cormen T. H., Leiserson C. E., Rivest R. L., Stein C. *Wprowadzenie do algorytmów.* WNT, Warszawa 2004.
- Wróblewski P. *Algorytmy, struktury danych i techniki programowania.* Helion, Gliwice 2015.
- Dawson M. *Python dla każdego. Podstawy programowania.* Helion, Gliwice 2021.
- Lutz M. *Python. Wprowadzenie.* Helion, Gliwice 2020.
- Saha A. *Matematyka w Pythonie.* Helion, Gliwice 2021.

Informacja i środki jej przetwarzania

Informacja

Informacja (łac. *informatio*) – przedstawienie, wizerunek; termin interdyscyplinarny, definiowany różnie w różnych dziedzinach nauki. Najogólniej — właściwość pewnych obiektów, relacja między elementami zbiorów pewnych obiektów.

Cechy informacji:

- wielkość abstrakcyjna,
- może być przechowywana,
- może być przesyłana między obiektami,
- może być przetwarzana przez pewne obiekty.

Informacje mogą stanowić różnorodne dane, takie jak liczby, tekst, obiekty multimedialne (grafika, dźwięki).

Informatyka

Dyscyplina nauki zaliczana do nauk ścisłych oraz techniki zajmująca się przetwarzaniem informacji, w tym również technologiami przetwarzania informacji oraz technologiami wytwarzania systemów przetwarzających informację.

Początkowo stanowiła część matematyki, później rozwinęła się do odrębnej dyscypliny – pozostaje jednak nadal w ścisłej relacji z matematyką, która dostarcza informatyce podstaw teoretycznych.

W języku polskim termin **informatyka** zaproponował w październiku 1968 Romuald Marczyński w Zakopanem na ogólnopolskiej konferencji poświęconej *maszynom matematycznym* na wzór (fr.) *informatique* i (niem.) *informatik*.

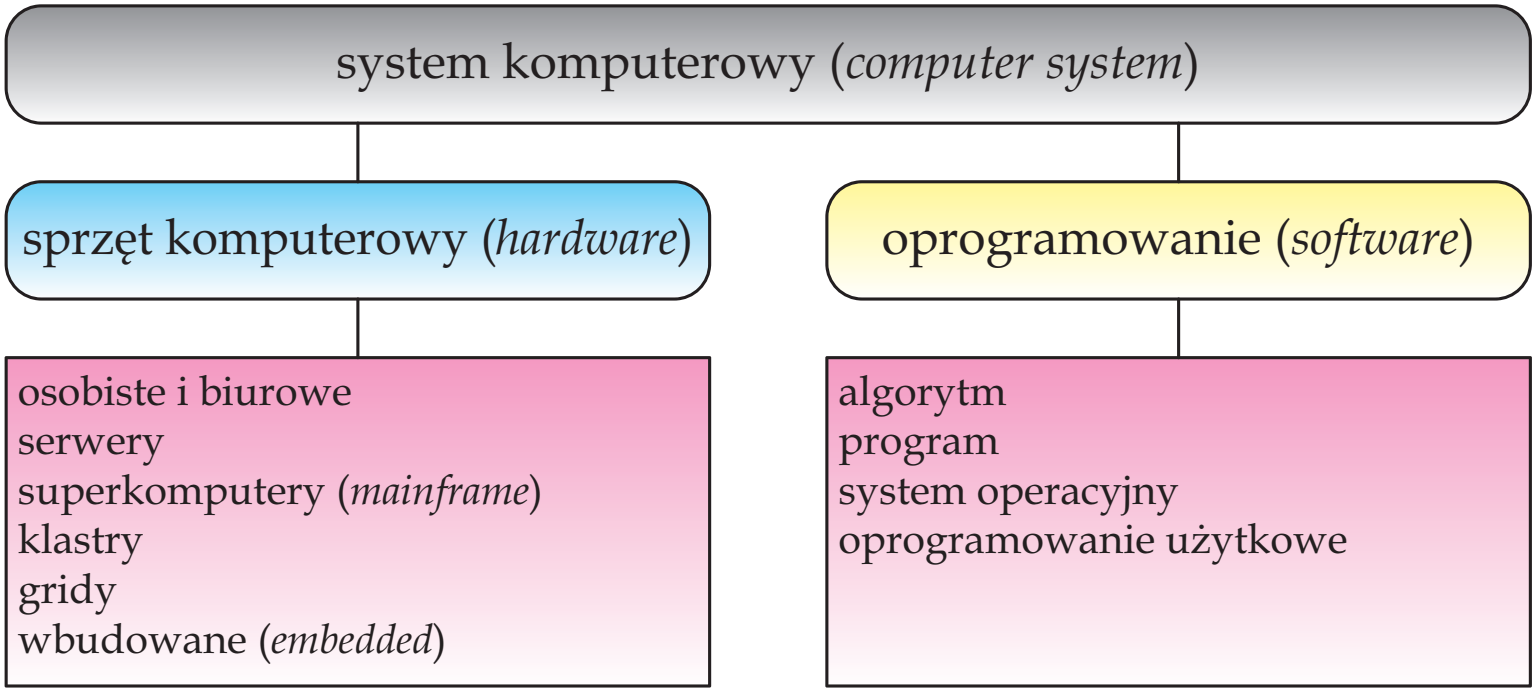
Bardziej znane działy informatyki to przede wszystkim:

- **administracja sieciowa** – zarządzanie siecią komputerową,
- **administracja systemem** – zarządzanie systemem informatycznym,
- **algorytmika** – tworzenie i analizowanie algorytmów,
- **architektura procesorów** – projektowanie procesorów,
- **grafika komputerowa** – wykorzystuje technikę komputerową w celu wizualizacji rzeczywistości,
- **inżynieria oprogramowania** – produkcja oprogramowania,
- **języki programowania** – tworzenie języków programowania,
- **programowanie komputerów** – czyli tworzenie kodu źródłowego programów komputerowych,
- **sprzęt komputerowy** – komputery i ich urządzenia peryferyjne,
- **symulacja komputerowa** – komputerowa symulacja z wykorzystaniem modelowania matematycznego,
- **bazy danych** – tworzenie, zarządzanie systemami baz danych,
- **sztuczna inteligencja** – komputerowe symulowanie inteligencji,
- **teoria informacji** – dyscyplina zajmująca się problematyką przetwarzania i przesyłania informacji,
- **webmastering** – projektowanie, programowanie serwisów internetowych.

System komputerowy

System komputerowy

Układ dwóch składowych: **sprzętu komputerowego** oraz **oprogramowania**. System komputerowy coraz częściej działa w sieci komputerowej. Można mówić o różnych poziomach takiego systemu: sprzęt komputerowy, system operacyjny (oprogramowanie systemowe), oprogramowanie użytkowe (aplikacje).



Sprzęt komputerowy – materialna część komputera. Ogólnie hardware'em nazywa się sprzęt komputerowy jako taki i odróżnia się go od software'u – czyli oprogramowania.

Podział ten jest nieostry, gdyż współcześnie wiele elementów sprzętu komputerowego posiada *wszyte* weń na stałe oprogramowanie, stanowiące jego integralną część, bez którego elementy te nie mogłyby funkcjonować. Np. większość drukarek komputerowych posiada w swojej pamięci zestaw komend, przy pomocy których realizuje proces drukowania i których odpowiednik znajduje się w pamięci komputera stanowiąc programowy sterownik tego urządzenia. Wiele urządzeń – typu karty graficzne, płyty główne posiada własne oprogramowanie nazywane **BIOS**-em. W stosunku do oprogramowania niektórych urządzeń używa się słowa **firmware**.

Oprogramowanie (*software*) – całość informacji w postaci zestawu instrukcji, zaimplementowanych interfejsów i zintegrowanych danych przeznaczonych dla komputera do realizacji wyznaczonych celów. Celem oprogramowania jest przetwarzanie danych w określonym przez twórcę zakresie.

Oprogramowanie tworzą programiści w procesie programowania. Oprogramowanie jako przejaw twórczości jest chronione prawem autorskim.

Oprogramowanie pisane jest zazwyczaj przy użyciu różnych języków programowania z wykorzystaniem algorytmów.

Często składowe systemu komputerowego (sprzęt i oprogramowanie) przedstawia się w postaci kilku warstw:

- komputer,
- oprogramowanie systemowe,
- oprogramowanie narzędziowe,
- oprogramowanie użytkowe,
- użytkownicy.

Komputer – zapewnia podstawowe możliwości obliczeniowe (procesor, pamięć, urządzenia wejścia/wyjścia), czyli są to podstawowe zasoby systemu komputerowego.

Oprogramowanie systemowe – kontroluje i koordynuje działanie zasobów sprzętowych przez zastosowanie różnych programów użytkowych dla różnych użytkowników.

Oprogramowanie narzędziowe – dogodne interfejsy użytkowe wspomagające zarządzanie zasobami sprzętowymi oraz usprawniające, modyfikujące oprogramowanie systemowe.

Oprogramowanie użytkowe – określają sposoby użycia zasobów systemowych do rozwiązywania problemów obliczeniowych zadanych przez użytkownika (kompilatory, systemy baz danych, gry, oprogramowanie biurowe), tworzone przez programistów.

Użytkownicy – ludzie, maszyny, inne komputery, mający bezpośredni kontakt z oprogramowaniem użytkowym.

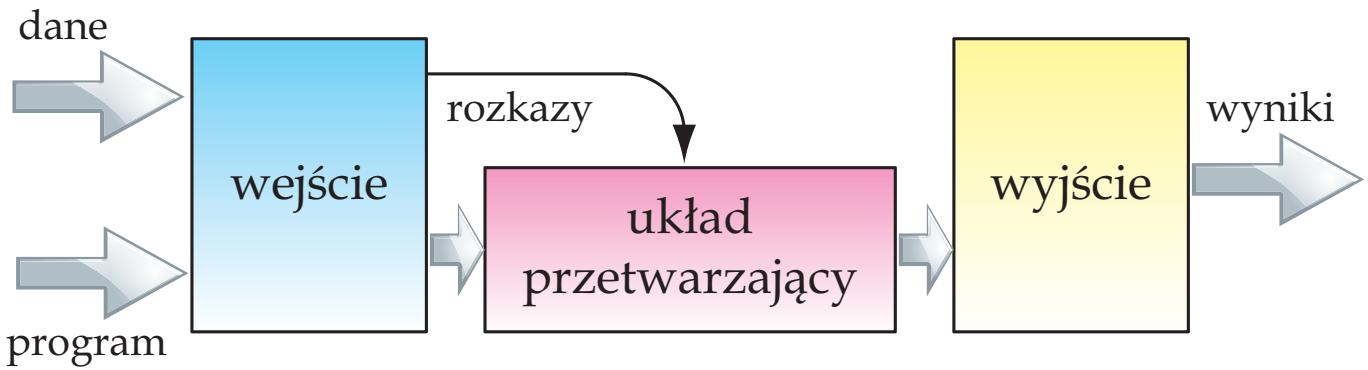
Komputer

Komputer

Urządzenie do przetwarzanie danych, umożliwiające wprowadzanie, przechowywanie i wyprowadzanie danych. Charakteryzuje je zdolność wykonywania wielokrotnie, automatycznie powtarzanych obliczeń, według algorytmicznego wzorca zwanego programem.

Granica jest tu umowna, ponieważ taką definicję komputera spełniają też kalkulatory programowalne (naukowe, inżynierskie), jednak kalkulatory służą tylko do obliczeń numerycznych, podczas gdy nazwa komputer najczęściej dotyczy **urządzeń wielofunkcyjnych**.

Jakkolwiek istnieją mechaniczne urządzenia liczące, które potrafią realizować całkiem złożone programy, zazwyczaj nie zalicza się ich do komputerów. Warto jednak pamiętać, że prawzorem komputera jest abstrakcyjny model zwany **maszyną Turinga**, a pierwsze urządzenia ułatwiające obliczenia były znane w starożytności, np. **abakus** z 440 p.n.e.



Architektura komputera

Sposób organizacji elementów tworzących komputer. Pojęcie to używane jest dosyć luźno. Może ono dzielić systemy komputerowe ze względu na wiele czynników, zazwyczaj jednak pod pojęciem architektury komputera rozumie się organizację połączeń pomiędzy pamięcią, procesorem i urządzeniami wejścia-wyjścia.

Innym, stosowanym potocznie znaczeniem terminu *architektura komputera* jest typ procesora wraz z zestawem jego instrukcji. Właściwszym określeniem w tym przypadku jest **model programowy procesora** (*ISA*, *Instruction Set Architecture*).

Model programowy procesora to określenie dotyczące organizacji, funkcjonalności i zasad działania procesora, widoczne z punktu widzenia programisty jako dostępne mechanizmy programowania.

Na model programowy procesora składają się m.in.:

- lista rozkazów procesora,
- obsługiwane typy danych,
- dostępne tryby adresowania,
- zestaw rejestrów dostępnych dla programisty,
- zasady obsługi wyjątków i przerwań.

Procesory posiadające ten sam model programowy są ze sobą **kompatybilne**, co oznacza, że mogą wykonywać te same programy i generować te same rezultaty. Przykładami modeli programowych:

- **IA-32**: i386, Pentium, K6, Athlon (RISC),
- **SPARC**: UltraSPARC, SPARC64,
- **AMD64**: Athlon 64 i wzwyż, Pentium 4 Prescott i wzwyż.

Taksonomie architektur komputerowych

Jednym z rodzajów klasyfikacji jest **Taksonomia Skillicorna**, zaproponowana ok. 1988 roku. Zakłada ona, że każda architektura stanowi połączenie pewnej liczby abstrakcyjnych składników. W efekcie, taksonomia syntetyzuje architekturę, zamiast ją klasyfikować.

Klasyfikacja Skillicorna posługuje się elementami architektury:

- procesory instrukcji (sterujące) – **IP** (*Instruction Processor*),
- procesory danych – **DP** (*Data Processor*),
- hierarchia pamięci instrukcji – **IM** (*Instruction Memory Hierarchy*),
- hierarchia pamięci danych – **DM** (*Data Memory Hierarchy*),
- układy łączące powyższe elementy.

Zadania **procesora instrukcji**:

- wyznaczenie adresu następnej instrukcji do wykonania,
- sterowanie adresacją pamięci instrukcji,
- odczyt i dekodowanie instrukcji,
- sterowanie procesorem danych – wysłanie mu instrukcji do wykonania,
- wyznaczanie adresów operandów,
- odbieranie informacji stanu od procesorów danych.

Zadania **procesora danych**:

- odbiór od układu sterującego instrukcji i adresów operandów,
- odczyt operandów z pamięci danych,
- wykonanie instrukcji operandach,
- wyznaczenie informacji stanu dla układu sterującego,
- zapamietanie wyników obliczeń w pamięci danych.

Hierarchiczna pamięć instrukcji jest zbudowana podobnie jak **hierarchiczna** pamięć danych.

Hierarchiczna pamięć instrukcji lub danych zawiera:

- pamięć podręczną instrukcji/danych,
- pamięć operacyjną,
- pamięć dodatkową (np. dyskową).

Pamięć operacyjna oraz pamięć dodatkowa może być wspólna lub oddzielna dla danych i instrukcji. Tego problemu klasyfikacja Skillicorna nie rozstrzyga.

Układy łączące obejmują następujące typy:

- połączenie 1 do 1 – pojedyncze połączenie, (1 - 1),
- połączenie n do n – *n* połączeń pojedynczych równoległych (n - n),
- połączenie 1 do n – rozesłanie informacji do *n* odbiorców (1 - n),
- połączenie n na n – przełącznik krzyżowy łączący *n* wejść z *n* wyjściami (n$\times$n).

Same procesory danych nie zawierają żadnych elementów pamiętających.

W modelach architektur przyjmuje się, że liczba hierarchii pamięci jest równa liczbie procesorów danego typu. Oznacza to, że model architektury ze wspólną hierarchią pamięci dla kilku procesorów jest przedstawiany jako model z kilkoma hierarchiami pamięci i możliwością dostępu każdego procesora do każdej hierarchii pamięci.

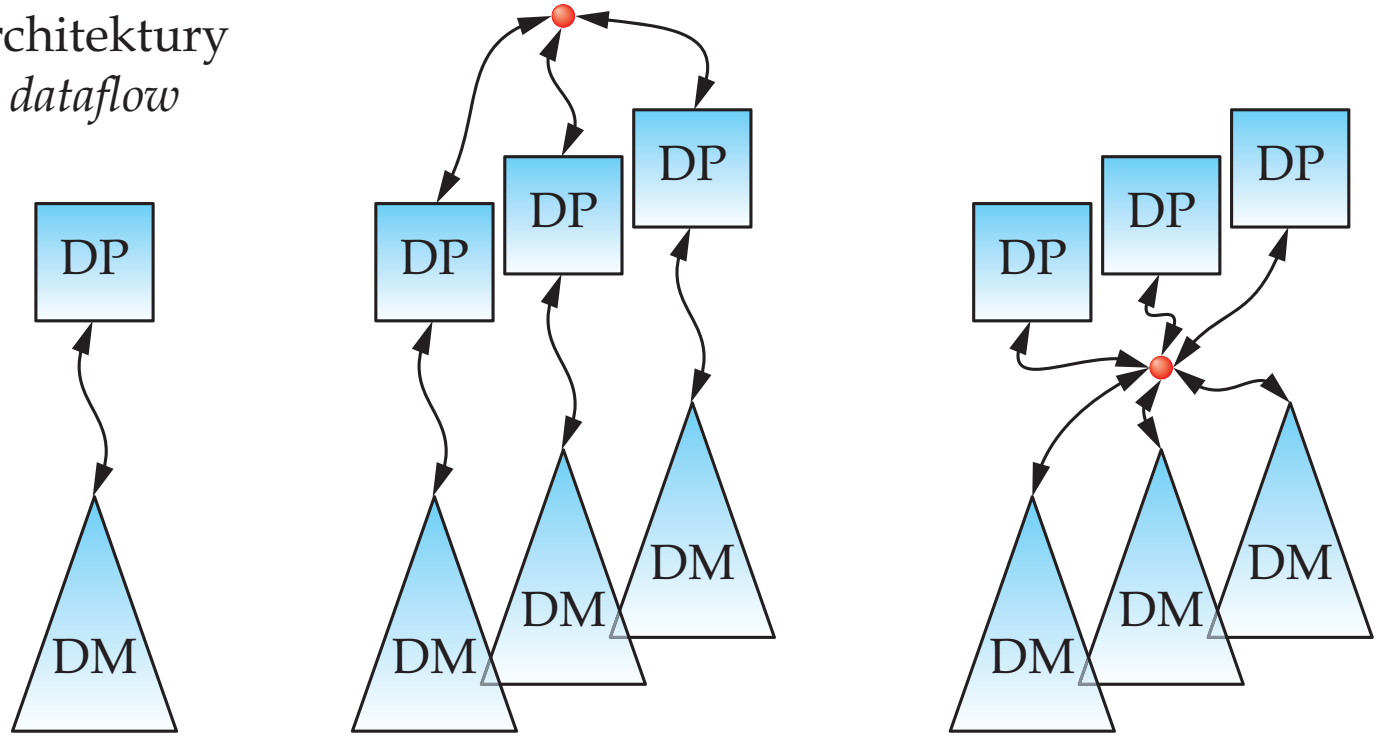
Komputer musi zawierać przynajmniej jeden procesor danych. Dozwolone są połączenia pomiędzy procesorami i hierarchiami pamięci tego samego rodzaju (kodu albo danych) oraz połączenia pomiędzy procesorami (tego samego lub różnych typów).

Używając taksonomii Skillcorna można zbudować około 30 różnych modeli architektur. Sześć z tych modeli ma sensowne znaczenie:

- uniprocesor dataflow (w taksonomii Flynna będzie to model bez strumienia instrukcji i jednym strumieniem danych, NISD),
- wieloprocesor dataflow (w taksonomii Flynna będzie to model bez strumienia instrukcji i N strumieniami danych, NIMD),
- uniprocesor von Neumanna (SISD),
- procesor wektorowy (SIMD),
- wieloprocesor słabo sprzężony (MIMD),
- wieloprocrsor silnie sprzężony (MIMD).

Sprzężenie pomiędzy procesorami tego samego typu ma sens tylko dla procesorów danych. Sprzężenie słabe będzie realizowane przez kanał komunikacyjny a silne przez wzajemny dostęp do hierarchii pamięci (praktycznie wspólna hierarchia pamięci).

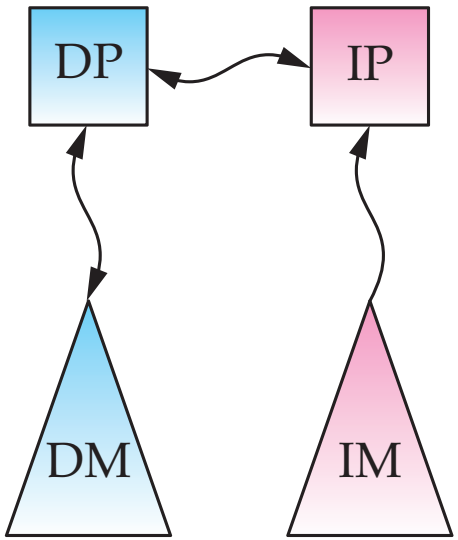
architektury
dataflow



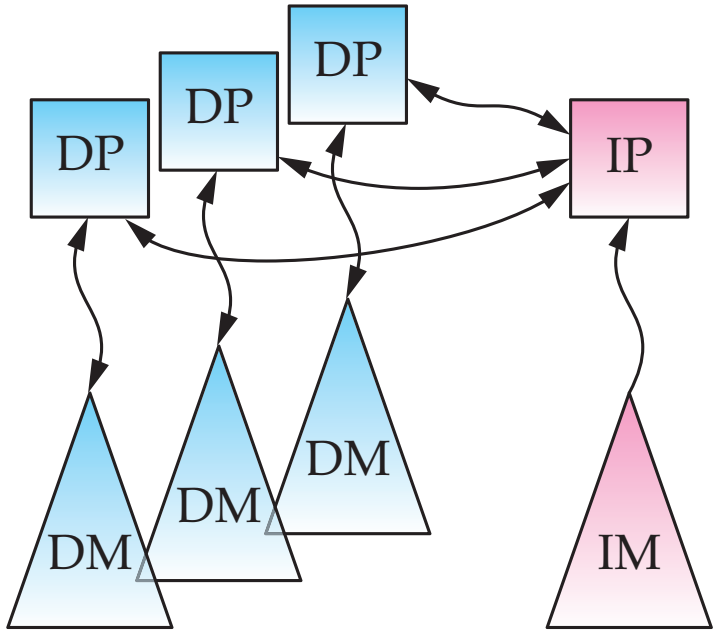
uniprocesor

wieloprocesor
słabo sprzężony

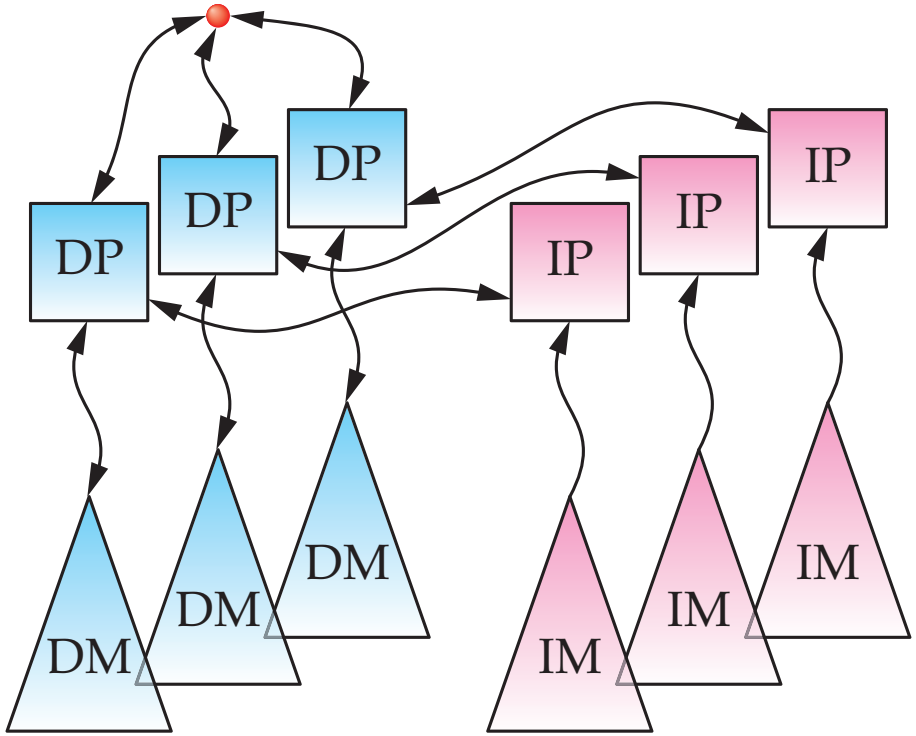
wieloprocesor
silnie sprzężony



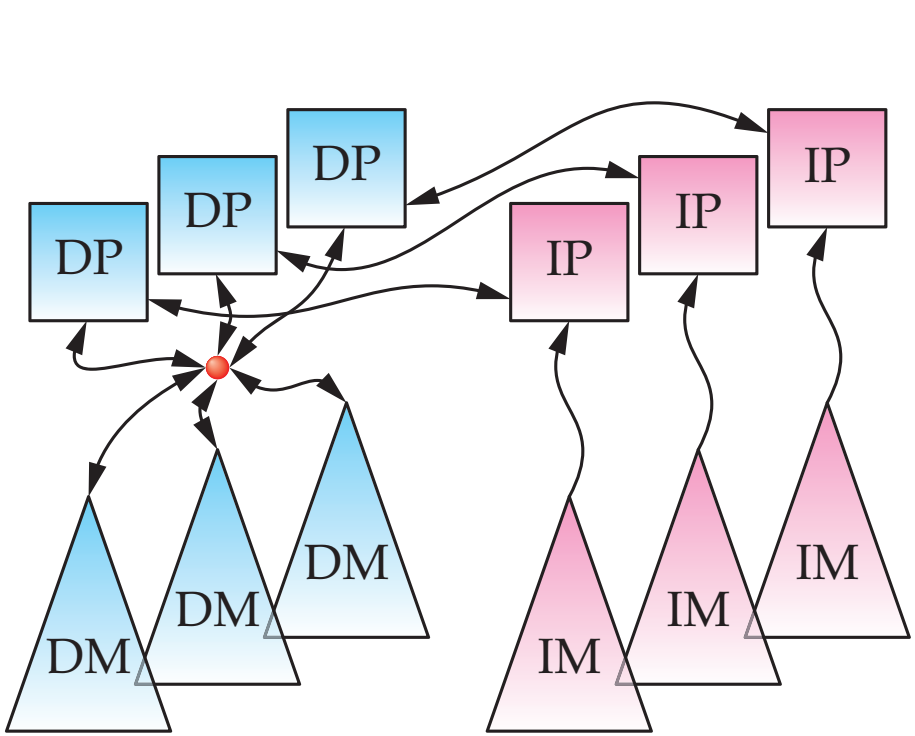
uniprocesor
von Neumanna



procesor
wektorowy



wieloprocesor
słabo sprzężony



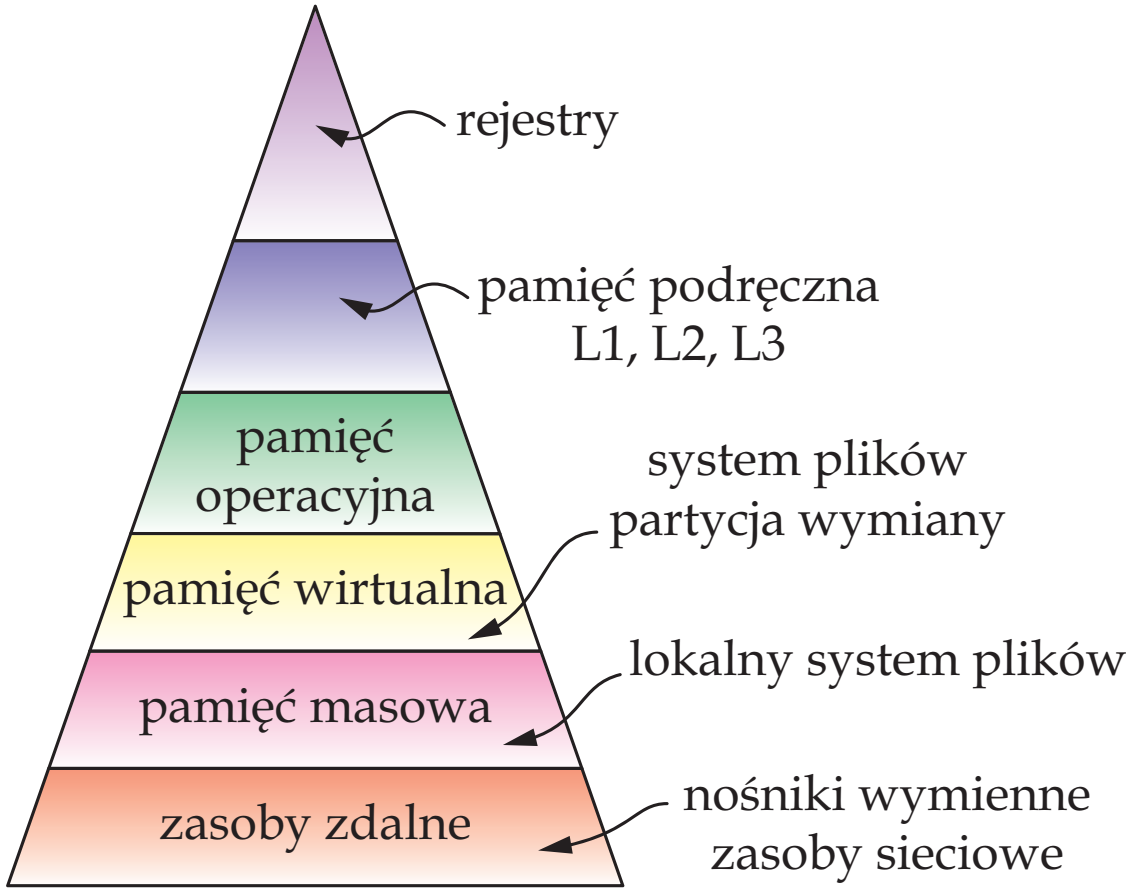
wieloprocesor
silnie sprzężony

Hierarchia pamięci

W taksonomia Skillicorna występuje pojęcie hierarchii pamięci (a nie pamięć). Słowo *hierarchia* dobrze oddaje budowę pamięci współczesnego komputera, w którym znajduje się kilka bloków funkcjonalnych służących do przechowywania programów i danych.

Idealny komputer powinien mieć jak największą i jak najszybszą pamięć. Pojemność pamięci wpływa na jej fizyczne rozmiary, a te – na czas dostępu. Nie można więc zbudować dowolnie dużej i jednocześnie szybkiej pamięci.

Problem ten rozwiązuje się przez wyodrębnienie wielu warstw o zróżnicowanej pojemności i szybkości, tworzących razem hierarchię pamięci. Kolejne warstwy w miarę oddalania się od procesora mają coraz większe pojemności i coraz dłuższe czasy dostępu.


rejestry
pamięć podręczna
L1, L2, L3
pamięć
operacyjna
system plików
partycja wymiany
pamięć wirtualna
lokalny system plików
pamięć masowa
nośniki wymienne
zasoby sieciowe
zasoby zdalne

Hierarchia pamięci współczesnego komputera, z punktu widzenia konstrukcji komputera, składa się z czterech warstw.

Rejestry fizycznie znajdują się wewnątrz procesora, dzięki czemu dostęp do nich jest bardzo szybki.

Pamięć podręczna (kieszeń, *cache*), wprowadzone po raz pierwszy około 1968 roku, zapewniają buforowanie danych pomiędzy procesorem i pamięcią operacyjną w celu przyspieszenie dostępu do pamięci.

Warstwa pamięci **wirtualnej**, powstała również około 1968 roku, zapewnia rozszerzenie pamięci operacyjnej.

Z punktu widzenia użytkownika do hierarchii pamięci należy zaliczyć wszelkie zasoby służące przechowywaniu danych. Logiczne staje się więc uzupełnienie rysunku o **lokalny system plików** komputera oraz o **zasoby zdalne**, w postaci nośników wymiennych i serwerów sieciowych.

Mechanizmy sterujące przemieszczaniem danych pomiędzy poszczególnymi warstwami są różne.

O umieszczeniu danych w rejestrach decyduje **programista** piszący program w asemblerze lub **kompilator** języka wysokiego poziomu.

Styk warstwy pamięci podręcznej i pamięci operacyjnej jest sterowany na **poziomie sprzętu**.

Stykiem pamięci operacyjnej i wirtualnej steruje **system operacyjny** przy użyciu jednostki zarządzania pamięcią.

O umieszczeniu danych w pamięci wirtualnej decyduje **użytkownik** – otwierając plik danych lub uruchamiając program.

Przemieszczaniem danych pomiędzy lokalnym systemem plików i nośnikami wymiennymi lub zasobami sieciowymi steruje **użytkownik**.

Tabela przedstawia orientacyjne parametry poszczególnych warstw hierarchii pamięci. Należy zwrócić uwagę na dużą różnicę czasów dostępu pamięci podręcznej i pamięci operacyjnej lub wirtualnej – czas podany dla pamięci dotyczy pojedynczego, losowego dostępu do pamięci dynamicznej typu DDR.

warstwa, pojemność, czas dostępu		
rejestry	< 1 kB	< 1 ns
pamięć podręczna L1	$\leqslant 128$ kB	≈ 1 ns
pamięć podręczna L2	$\leqslant 12$ MB	$1 \ldots 2$ ns
pamięć podręczna L3	$\leqslant 256$ MB	$2 \ldots 5$ ns
pamięć operacyjna	$\leqslant 64$ GB	$10 \ldots 50$ ns
pamięć wirtualna, system plików	$\geqslant 128$ GB	$\leqslant 10$ ms
nośniki wymienne, sieć komputerowa	∞	s, min.

Maszyna von Neumanna

Maszyna von Neumanna – model architektury komputera, opracowany przez Johna von Neumanna, Johna W. Mauchly'ego oraz Johna P. Eckerta w 1945 roku. Jej cechą charakterystyczną jest taki sam sposób przechowywania instrukcji i danych w hierarchii pamięci. Model maszyny von Neumanna wprowadza specyficzny mechanizm dostępu do pamięci – poprzez adres.

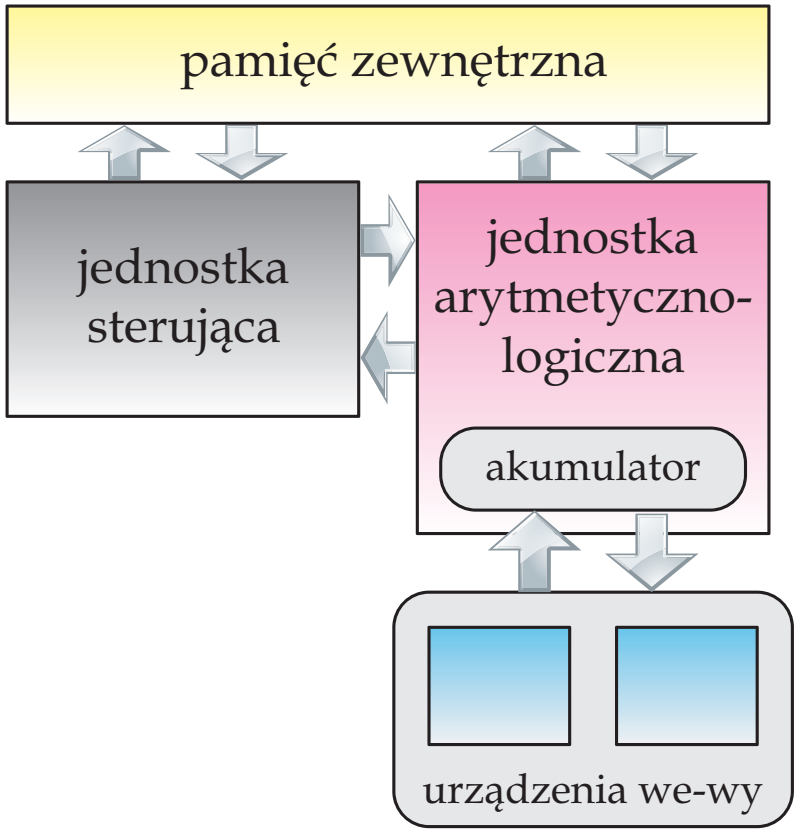
Maszyna von Neumanna następujące cechy:

- wykorzystuje model obliczeń zaproponowany przez Turinga, wykonuje obliczenia zgodnie z programem,
- program jest przechowywany w pamięci razem z danymi,
- pamięć składa się z pewnej liczby ponumerowanych komórek,
- dostęp do pamięci następuje poprzez podanie numeru komórki, czyli **adresu**,
- adres jest przechowywany i inkrementowany w specjalnym rejestrze procesora, zwanym **licznikiem instrukcji** (PC, *Program Counter*).

W architekturze tej komputer składa się z czterech głównych komponentów:

- pamięci komputerowej przechowującej dane programu oraz instrukcje programu, każda komórka pamięci ma unikatowy adres,
- jednostki sterującej odpowiedzialnej za pobieranie danych i instrukcji, pamięci oraz ich sekwencyjne przetwarzanie,
- jednostki arytmetyczno-logicznej odpowiedzialnej za wykonywanie podstawowych operacji arytmetycznych,
- rządzeń wejścia-wyjścia służących do interakcji z operatorem.

Jednostka sterująca wraz z jednostką arytmetyczno-logiczną tworzą procesor.


pamięć zewnętrzna
jednostka sterująca
jednostka arytmetyczno-logiczna
akumulator
urządzenia we-wy

System komputerowy zbudowany w oparciu o architekturę von Neumanna powinien:

- mieć skończoną i funkcjonalnie pełną listę rozkazów,
- mieć możliwość wprowadzenia programu do systemu komputerowego poprzez urządzenia zewnętrzne i jego przechowywanie w pamięci w sposób identyczny jak danych,
- dane i instrukcje w takim systemie powinny być jednakowo dostępne dla procesora,
- informacja jest tam przetwarzana dzięki sekwencyjnemu odczytywaniu instrukcji z pamięci komputera i wykonywaniu tych instrukcji w procesorze.

Podane warunki pozwalają przełączać system komputerowy z wykonania jednego zadania (programu) na inne bez fizycznej ingerencji w strukturę systemu, a tym samym gwarantują jego uniwersalność.

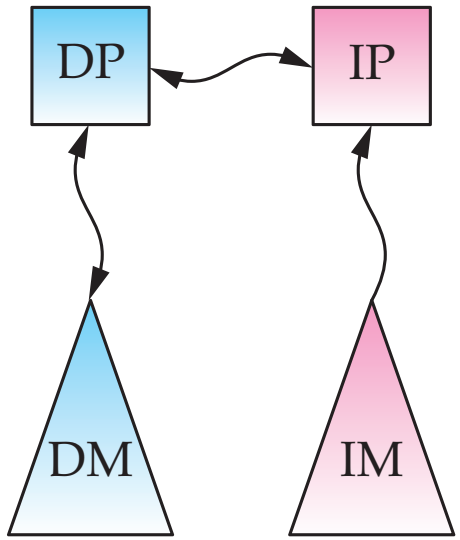
Dwa warianty architektury von Neumanna różnią, się sposobem przechowywania instrukcji i danych:

- architektura **Harvard** – oddzielne hierarchie pamięci danych i rozkazów,
- architektura **Princeton** – wspólna hierarchia pamięci danych i rozkazów.

Architektura Harvard jest niekiedy uważana za architekturę nie spełniającą postulatów von Neumanna wobec faktu oddzielnego przechowywania instrukcji i danych.

Cechy charakterystyczne architektury Harvard:

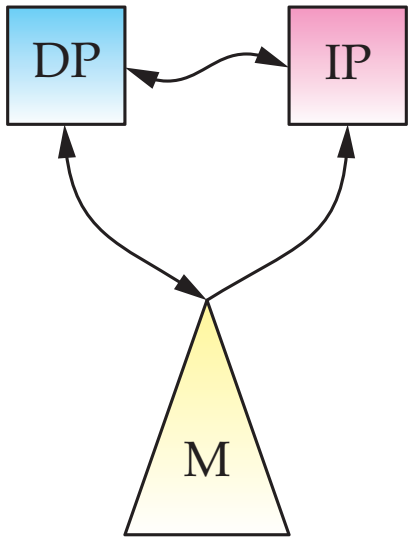
- oddzielne hierarchie pamięci danych i instrukcji,
- wysoka wydajność dzięki możliwości równoczesnego pobierania instrukcji i wykonywania operacji na hierarchii pamięci,
- brak możliwości programowania – nie ma możliwości zapisu instrukcji do pamięci instrukcji,
- komputer jest dostarczany ze stałym programem,
- jest wykorzystywana w procesorach sygnałowych oraz mikrokomputerach jednoukładowych (zastosowania wbudowane).



Uwaga! Rysunek posługuje się symbolami zapożyczonymi z taksonomii Skillicorna w sposób sprzeczny z zasadami budowy modeli wprowadzonymi przez tę taksonomię.

Cechy charakterystyczne architektury Princeton:

- *wzorcowa* realizacja maszyny von Neumanna ze wspólną hierarchią pamięci instrukcji i danych,
- nie można równocześnie pobierać danych i rozkazów (*von Neumann bottleneck*),
- nieograniczone możliwości modyfikacji programu,
- obiekt zapisany jako dana może być pobrany jako instrukcja,
- wykorzystywana w komputerach uniwersalnych.

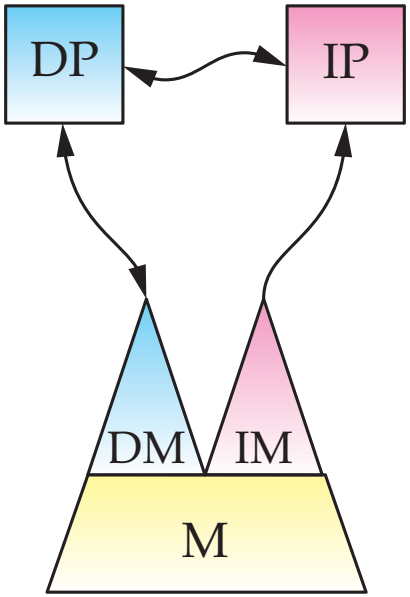


Uwaga! Rysunek posługuje się symbolami zapożyczonymi z taksonomii Skillicorna w sposób sprzeczny z zasadami budowy modeli wprowadzonymi przez tę taksonomię.

Zmodyfikowana architektura harwardzka – architektura mieszana, łączy w sobie cechy architektury harwardzkiej i architektury von Neumanna. Oddzielone zostały częściowo hierarchie pamięci na dane i instrukcje.

Cechy charakterystyczne architektury Harvard-Princeton:

- realizacja maszyny von Neumanna z oddzielonymi *górnymi* warstwami hierarchii pamięci i wspólnymi *dolnymi*,
- przynajmniej jeden poziom pamięci podręcznej jest oddzielny dla procesorów instrukcji i danych,
- szybkie działanie dzięki równoległości dostępów jak w architekturze Harvard,
- możliwość programowania niezbędna w komputerach uniwersalnych, jak w architekturze Princeton,
- program użytkowy nie ma pełnej kontroli nad położeniem obiektów w hierarchii pamięci, brak możliwości modyfikacji,
- w/w kontrolę może mieć wyróżniony program (proces) – system operacyjny, większość współczesnych komputerów uniwersalnych, w tym komputery PC ma architekturę Harvard-Princeton.

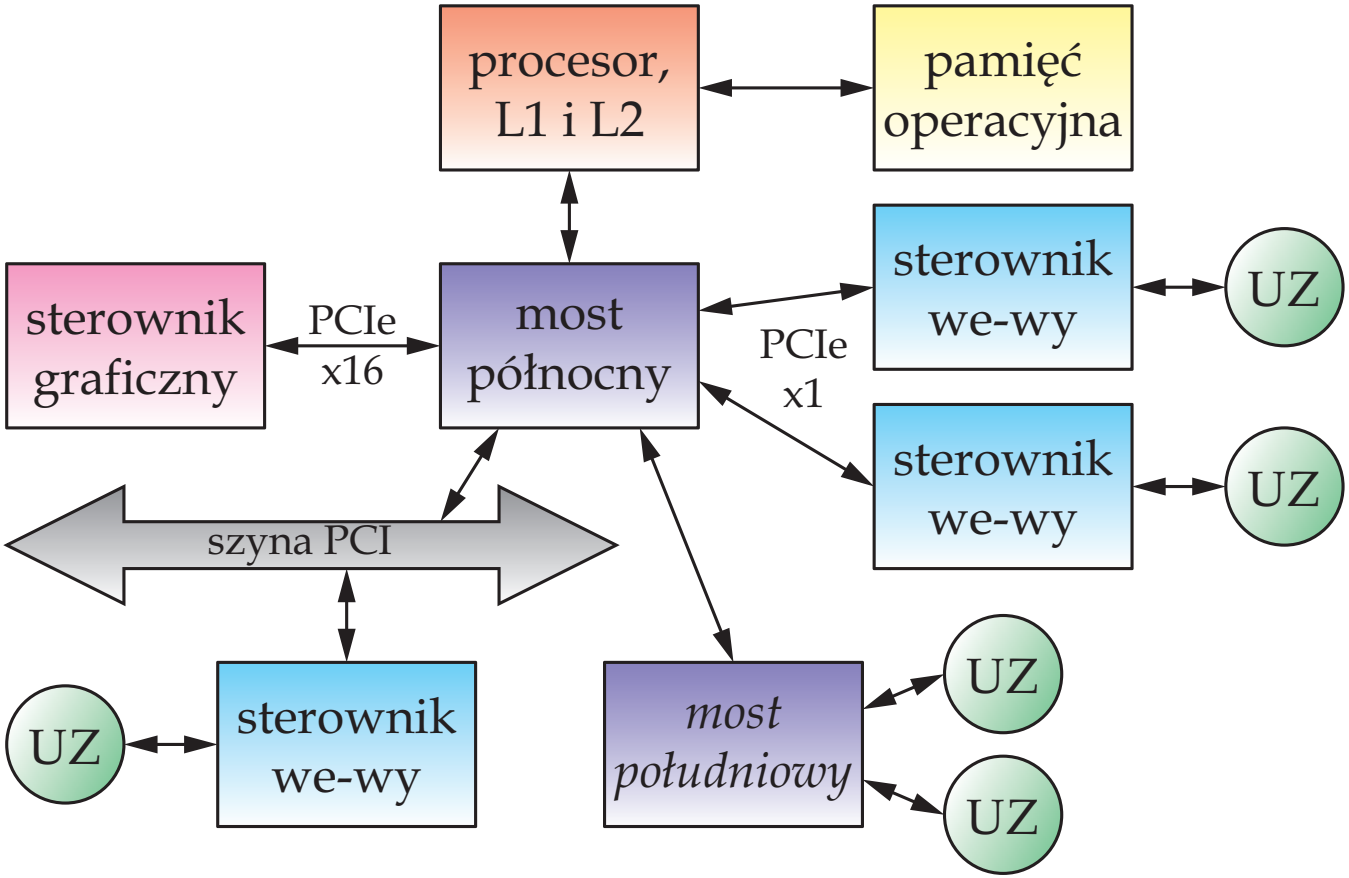


Uwaga! Rysunek posługuje się symbolami zapożyczonymi z taksonomii Skillicorna w sposób sprzeczny z zasadami budowy modeli wprowadzonymi przez tę taksonomię.

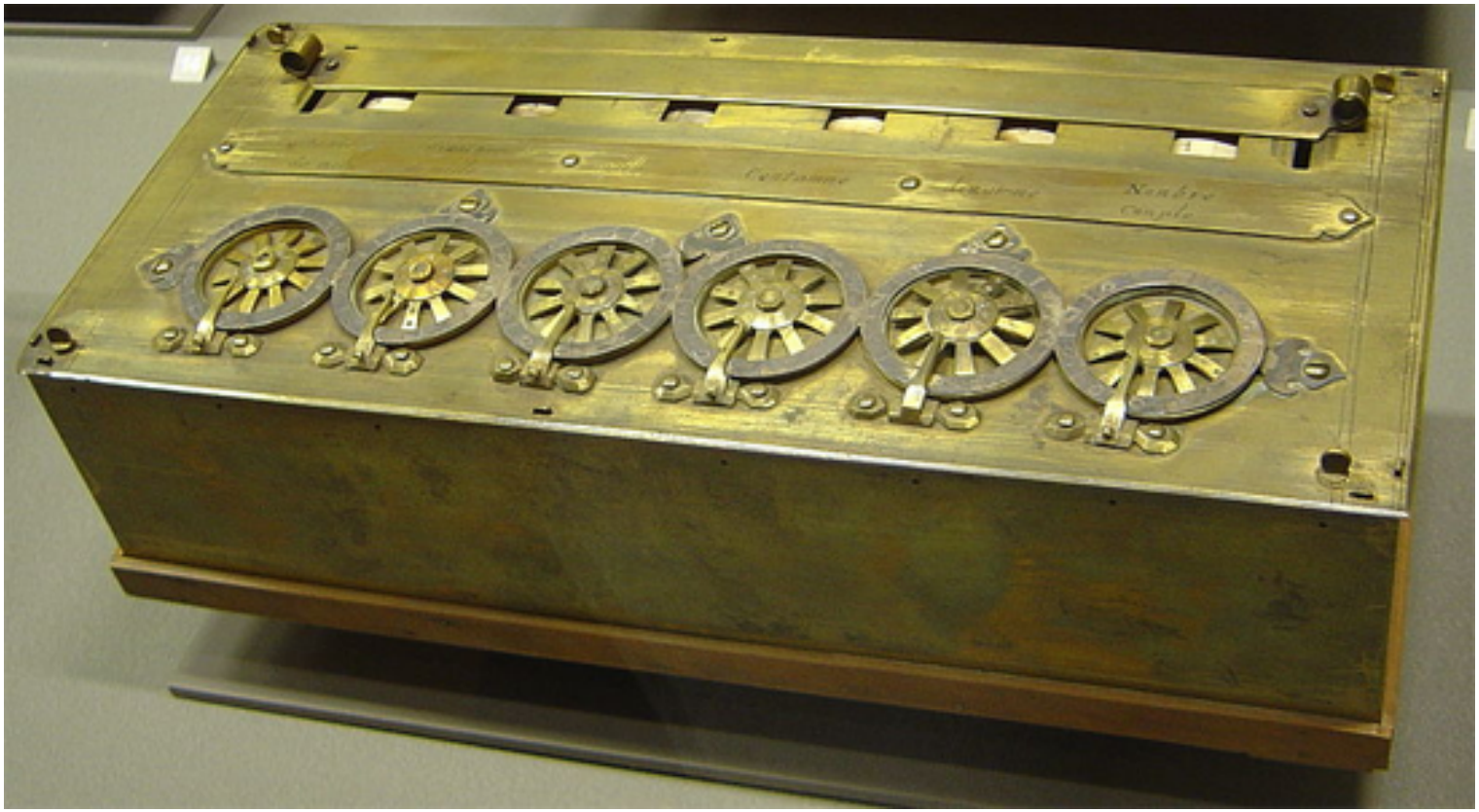
Architektura z połączeniami punkt-punkt

Architektura obecnie konstruowanych komputerów (klasy PC) jest bardziej złożona. We współczesnych komputerach **sterownik pamięci** umieszczony jest w **procesorze**. **Most północny** jest wyposażony w indywidualne łącza dla sterowników urządzeń zewnętrznych, zrealizowane w standardzie **PCI express**.

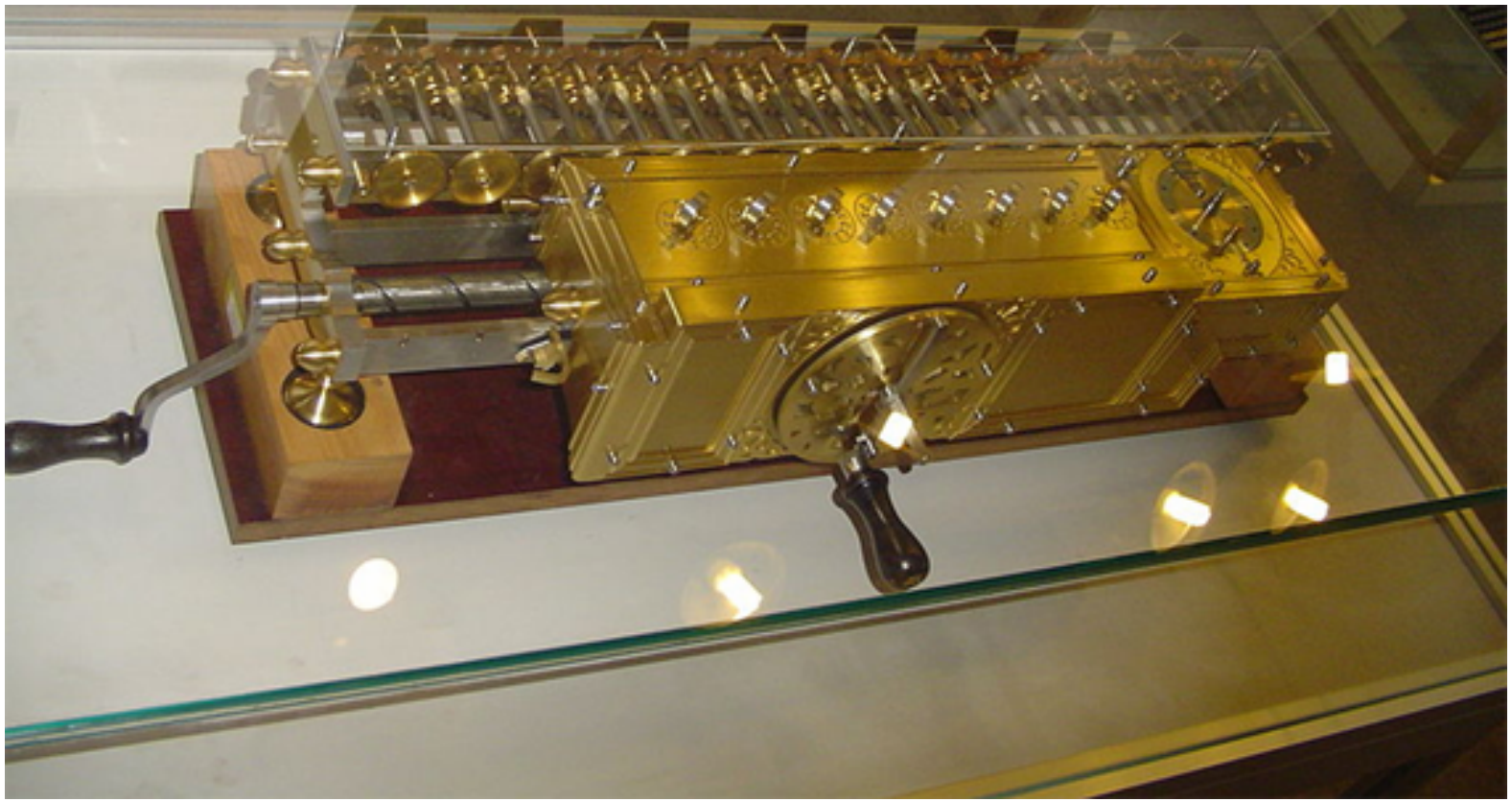
Most południowy jest zintegrowanym **sterownikiem urządzeń zewnętrznych**. Szyna **PCI** została zachowana w celu umożliwienia podłączenia starszych sterowników urządzeń. Jest ona przeznaczona do usunięcia.


procesor,
L1 i L2
pamięć
operacyjna
sterownik
graficzny
PCIe
x16
most
północny
PCIe
x1
sterownik
we-wy
UZ
sterownik
we-wy
UZ
szyna PCI
UZ
sterownik
we-wy
most
południowy
UZ
UZ

Pascalina – maszyna licząca zaprojektowana przez Blaise Pascala około 1645 roku. Pascalina umożliwiała jedynie dodawanie i odejmowanie liczb – była więc mechanicznym sumatorem.

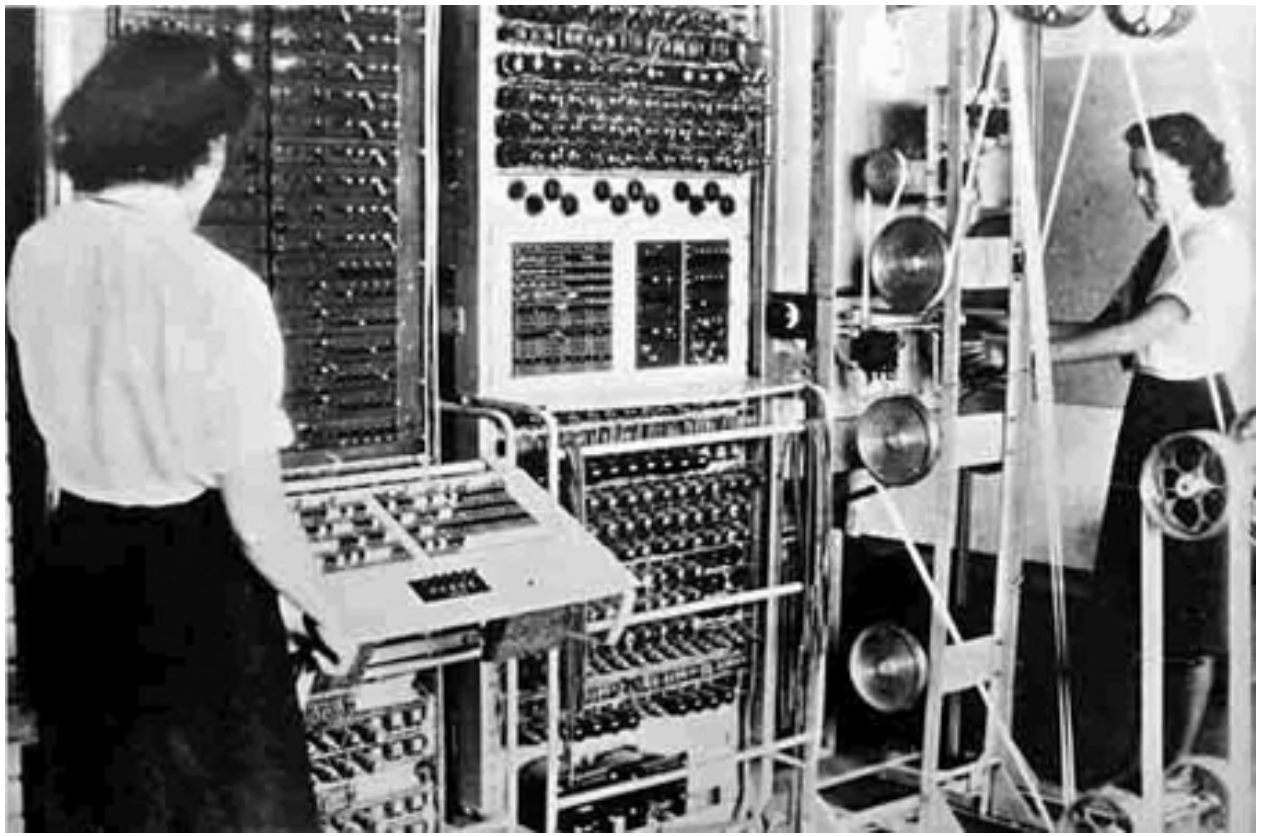


Lebendige Rechenbank, dosł. *żywa ława do obliczeń* – maszyna licząca zaprojektowana i wykonana przez Gottfrieda Wilhelma Leibniza w 1671. Maszyna ta była formą kalkulatora mechanicznego. Była zdolna do odejmowania, mnożenia, dzielenia i wyprowadzania pierwiastków kwadratowych.

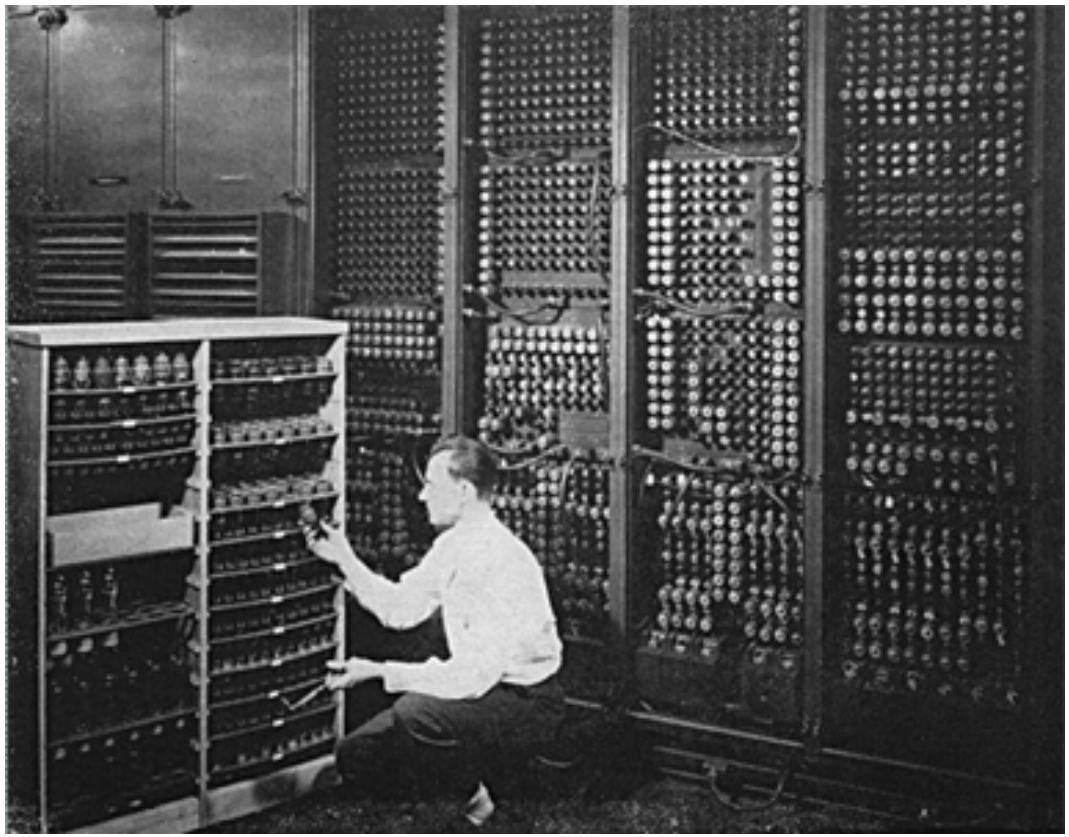


Z3 – to pierwszy działający, w pełni automatyczny komputer o zmiennym programie zbudowany przez niemieckiego inżyniera Konrada Zuse w 1941 roku na bazie jego wcześniejszej, mechanicznej konstrukcji, Z1. Maszyna była wykorzystywana w czasie wojny do obliczeń niezbędnych przy projektowaniu skrzydeł.

Colossus – seria programowalnych maszyn cyfrowych oparta na teoretycznych podstawach prac Alana Turinga. Projektem Colossus kierowali Max Newman i Tommy Flowers, uczestniczył w nim również Alan Turing. Colossus został zbudowany 14 kwietnia w 1943 roku w brytyjskim ośrodku kryptograficznym Bletchley Park i przeznaczony był do zastosowań wojskowych. Służył do rozpracowania sposobu działania niemieckiej Maszyny Lorenza i łamania jej szyfrów.



ENIAC, ang. *Electronic Numerical Integrator And Computer* – komputer skonstruowany w latach 1943-1945 przez J. P. Eckerta i J. W. Mauchly'ego na Uniwersytecie Pensylwanii w USA. Zaprzestano jego używania w 1955. Do roku 1975 powszechnie uważany był za pierwszy komputer na świecie, jednak teraz o miano to ubiegają się również – po odtajnieniu danych brytyjskich maszyny Colossus oraz niemieckie Konrada Zuse.



Informacja a komputer

Wewnętrzna budowa komputera wymusza używanie ściśle określonych sposobów kodowania (systemów liczbowych) i przetwarzania informacji.

Komputer ENIAC pracował w **systemie dziesiętnym**! Przełączniki elektryczne musiały rozpoznawać 10 stanów napięcia – jedna z przyczyn wolnej pracy i małej odporności na zniekształcenia elektryczne.

System dziesiętny nie jest jedynym sposobem przedstawiania liczb.

System binarny (dwójkowy) – urządzenia korzystające z niego muszą rozpoznawać tylko dwa stany, czyli **1** (jest sygnał) lub **0** (nie ma sygnału) a nie dziesięć stanów jak to ma miejsce w systemie dziesiętnym.

Zaletami systemu binarnego, w kontekście budowy komputera sa:

- znacznie prostszy układ elektroniczny,
- poziom sygnałów elektrycznych, rozpatrywanych jako odrębne stany mogą należeć do szerszego przedziału napięć,
- większa odporność na zniekształcenia.

Dla dowolnego systemu liczenia istnieje zbiór cyfr, z których tworzone są liczby.

Systemy liczbowe dzieli się na:

- pozycyjne,
- niepozycyjne.

Systemy niepozycyjne

W systemach **niepozycyjnych** poszczególne cyfry zachowują swą wartość liczbową bez względu na miejsce jakie zajmują w liczbie (np. system rzymski – siedem znaków: I, V, X, L, C, D, M).

XII = 10 (X) + 1 (I) + 1 (I) = 12
IX = 10 (X) - 1 (I) = 9
MCDX = 1000 (M) + 500 (D) - 100 (C) + 10 (X) = 1410

Systemy pozycyjne

W systemach **pozycyjnych** wartość liczbowa cyfry zależy od umiejscowienia (pozycji) w liczbie (np. system dziesiętny, dwójkowy).

Liczba różnych cyfr systemu nazywa się jego **podstawą** P.

Wartość liczbowa cyfry określona jest przez **wagę**. Waga na pozycji n równa jest podstawie P podniesionej do potęgi n.

Sposób zapisu liczby L w dowolnym systemie pozycyjnym jest bardzo prosty:

$$L = a_{n-1} \cdot P^{n-1} + a_{n-2} \cdot P^{n-2} + \ldots + a_0 \cdot P^0 = \sum_{i=0}^{n-1} a_i \cdot P^i.$$

Dla systemu **dziesiętnego** (decymalnego):

$$P = 10, \quad a_n = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\},$$

$$142 = 1 \cdot 10^2 + 4 \cdot 10^1 + 2 \cdot 10^0 = 1 \cdot 100 + 4 \cdot 10 + 2 \cdot 1.$$

Dla systemu **dwójkowego** (binarnego):

$$P = 2, \quad a_n = \{0, 1\},$$

$$10010_2 = 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 1 \cdot 16 + 0 \cdot 8 + 0 \cdot 4 + 1 \cdot 2 + 0 \cdot 1 = 18.$$

Dla systemu **szesnastkowego** (heksadecymalnego):

$$P = 16, \quad a_n = \{0, 1, 3, 4, 5, 6, 7, 8, 9, \mathrm{a}, \mathrm{b}, \mathrm{c}, \mathrm{d}, \mathrm{e}, \mathrm{f}\},$$

$$3e8_{16} = 3 \cdot 16^2 + 14 \cdot 16^1 + 8 \cdot 16^0 = 3 \cdot 256 + 14 \cdot 16 + 8 \cdot 1 = 1000.$$

Konwersja liczb naturalnych

$$190 = 1 \cdot 10^2 + 9 \cdot 10^1 + 0 \cdot 10^0,$$

$$190 = ?_2$$

dzielenie, wynik		
$190 / 2$	$= 95$	reszta $= 0$
$95 / 2$	$= 47$	reszta $= 1$
$47 / 2$	$= 23$	reszta $= 1$
$23 / 2$	$= 11$	reszta $= 1$
$11 / 2$	$= 5$	reszta $= 1$
$5 / 2$	$= 2$	reszta $= 1$
$2 / 2$	$= 1$	reszta $= 0$
$1 / 2$	$= 0$	reszta $= 1$

$$190 = 10111110_2.$$

$$10111110_2 = 1 \cdot 2^7 + 0 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 =$$

$$= 1 \cdot 128 + 0 \cdot 64 + 1 \cdot 32 + 1 \cdot 16 + 1 \cdot 8 + 1 \cdot 4 + 1 \cdot 2 + 0 \cdot 1.$$

Używając jednego bajtu, można zapisać liczę z zakresu $0 \ldots 255$, czyli $00000000_2 \ldots 11111111_2$ i $00_{16} \ldots ff_{16}$.

Działania arytmetyczne na liczbach w systemie dwójkowym i szesnastkowym są odpowiednikiem działań w systemie dziesiętnym i opierają się na elementarnych działaniach, np.:

- $1_2 + 0_2 = 1_2$,
- $1_2 + 1_2 = 10_2$,
- $1_2 \cdot 0_2 = 0_2$,
- $1_2 \cdot 1_2 = 1_2$,
- $10_2 - 1_2 = 1_2$.

Przykłady:

$$\begin{array}{r} \ 1\ 1\ 1\ 1\ 1\ 1 \\ \ 1\ 1\ 1\ 1\ 1\ 1\ 1 \\ +\quad\ \ 1\ 0\ 0\ 1\ 1 \\ \hline 1\ 0\ 0\ 1\ 0\ 0\ 1\ 0 \end{array}$$

$$\begin{array}{r} \ 1\ 1\ 1\ 1\ 1\ 1\ 1 \\ -\quad\ \ 1\ 0\ 0\ 1\ 1 \\ \hline 1\ 1\ 0\ 1\ 1\ 0\ 0 \end{array}$$

$$\begin{array}{r} \ \ 1\ 1\ 0\ 1 \\ \times\quad\ \ 1\ 0\ 1\ 1 \\ \hline \ \ 1\ 1\ 0\ 1 \\ \ 1\ 1\ 0\ 1 \\ \ 0\ 0\ 0\ 0 \\ +\ 1\ 1\ 0\ 1 \\ \hline 1\ 0\ 0\ 0\ 1\ 1\ 1\ 1 \end{array}$$

$$\begin{array}{r} \ \ 1\ 1\ 0 \\ \hline \ 1\ 1\ 0\ 1 \ /\ 1\ 0 \\ -\ 1\ 0 \\ \hline \ 0\ 1\ 0\ 1 \\ -\quad 1\ 0 \\ \hline \ 0\ 0\ 0\ 1 \\ -\quad\ \ 1\ 0 \\ \hline \ 0\ 0\ 0\ 1 \end{array}$$

Jednostki informacji

Najmniejszą jednostką informacji potrzebna do określenia, który z dwóch równie prawdopodobnych stanów przyjął układ jest **bit**, **b** (*BInary digiT*). Bit odpowiada informacji Tak – Nie, Prawda – Fałsz, 1 – 0.

Jest to również najmniejsza jednostka informacji używana w odniesieniu do sprzętu komputerowego. Bit przyjmuje wartości wtedy **0** lub **1**.

Wyższe jednostki to informacji:

- półbajt (*nybble*) – 4 bity,
- **oktet** (*octet*) – 8 bitów,
- **bajt** (*byte*) **B** – pierwotnie liczba bitów przetwarzana jednocześnie przez komputer lub adresowana przez procesor, obecnie używany wyłącznie do oznaczania 8 bitów (czyli oktetu).

Często używa się też:

- **słowo** (*word*) to zwykle 16 bitów (2 bajty) – jednostka informacji, na której operuje komputer, mogą zdarzać się słowa o innej długości, np.: 4, 8, 16 bajtów,
- **słowo procesora** – jednostka informacji o długości naturalnej dla procesora,
- **słowo pamięci** – jednostka informacji możliwa do przesłania w jednym cyklu transmisji do/z pamięci, zwykle 64 b, czasem 128 b.

Do określenia większej liczby bajtów stosuje się często (**niepoprawnie**) przedrostki **kilo**, **mega**, **giga** itd. Są to przedrostki dziesiętne układu SI, będące wielokrotnościami liczby 10 ($10^{3 \cdot n}$) a nie liczby 2:

- **kilobajt** (*kilobyte*), **kB** – $10^3 = 1000$ bajtów,
- **megabajt** (*megabyte*), **MB** – $10^6 = 1000^2 = 1$ milion bajtów,
- **gigabajt** (*gigabyte*), **GB** – $10^9 = 1000^3 = 1$ miliard bajtów,
- **terabajt** (*terabyte*), **TB** – $10^{12} = 1000^4 = 1$ bilion bajtów.

W celu odróżnienia przedrostków o mnożniku 1000 od przedrostków o mnożniku 1024, pojawiła się propozycja ujednoznacznienia, opracowana przez IEC, polegająca na dodawaniu litery **i** po symbolu przedrostka dwójkowego oraz **bi** po jego nazwie.

Nowe przedrostki nazywane zostały przedrostkami **dwójkowymi** (binarnymi). Jednak ta propozycja rozwiązania problemu niejednoznaczności przedrostków nie została przyjęta przez wszystkie środowiska. Przedrostki dwójkowe są wielokrotnościami liczby 2 ($2^{10 \cdot n}$):

- **kibibajt** (*kibibyte*), **KiB** – $2^{10} = 1024$ bajty,
- **mebibajt** (*mebibyte*), **MiB** – $2^{20} = 1024^2 = 1$ milion 48 tysięcy 576 bajtów,
- **gibibajt** (*gibibyte*), **GiB** – $2^{30} = 1024^3 = 1$ miliard 73 miliony 741 tysięcy 824 bajtów,
- **tebibajt** (*tebibyte*), **TiB** – $2^{40} = 1024^4 = 1$ bilion 99 miliardów 511 milionów 627 tysięcy 776 bajtów.

Współczesne komputery są używane do przetwarzania danych różnych typów – liczbowych, logicznych, tekstowych, a także obrazów i dźwięków. Działają w oparciu o system binarny.

- Wartości logiczne (prawda/fałsz).
- Znaki pisarskie.
- Liczby
  - całkowite
    - nieujemne
    - ze znakiem
  - niecałkowite
    - stałopozycyjne
    - zmiennopozycyjne
- Dźwięki, sygnały jednowymiarowe.
- Obrazy rastrowe.

Wszystkie dane, na których operuje komputer, są zapisane w postaci tbciągów cyfr binarnych – bitów, interpretowanych najczęściej jako liczby binarne.

Wszelkie dane o charakterze innym niż liczby, muszą być zapisane (zakodowane) w postaci liczb lub grup liczb.

Komputer przetwarza wyłącznie grupy bitów, tworząc liczby binarne, których długość wynosi $8 \cdot 2^n$ bitów (np. 8, 16, 32, 64).

Dane alfanumeryczne

Dane alfanumeryczne, tekstowe – mają postać znaków pisarskich – liter, cyfr, znaków przestankowych i innych symboli. W komputerze są one reprezentowane przez liczby, określające pozycję danego symbolu w tablicy kodowej.

We współczesnych komputerach używa się kilku standardów kodowania znaków pisarskich, najpopularniejsze to:

- ASCII,
- UNICODE.

ASCII

ASCII – (*American Standard Code for Information Interchange*), 7-bitowy kod przyporządkowujący liczby z zakresu 0 . . . 127 literom (alfabetu angielskiego), cyfrom, znakom przestankowym i innym symbolom oraz poleceniom sterującym. Został opracowany dla urządzeń dalekopisowych.

Na przykład litera **a** jest kodowana liczbą 97, a znak spacji jest kodowany liczbą 32.

Litery, cyfry oraz inne znaki drukowane tworzą zbiór znaków ASCII. Jest to 95 znaków o kodach 32 . . . 126. Pozostałe 33 kody (0 . . . 31 i 127) to tzw. kody sterujące służące do sterowania urządzeniem odbierającym komunikat, np. drukarką lub terminalem.

- Kody sterujące zajmują pozycje: $0 \ldots 31$, w tym:
  - **CR** - powrót na początek wiersza: kod 13,
  - **LF** – przejście do następnego wiersza: kod 10,
  - inne ważne: **HT** (tabulacja pozioma), **FF** (rozpoczęcie nowej strony), **BEL** (sygnał dźwiękowy), **VT** (tabulacja pionowa), **BSP** (cofnięcie o jeden znak).
- **Spacja**: kod 32 ($0x20$).
- **Cyfry** $0 \ldots 9$: kody $48 \ldots 57$ ($0x30 \ldots 0x39$).
- **Litery** w kolejności alfabetycznej:
  - wielkie: kody $65 \ldots 90$ ($0x41 \ldots 0x5a$),
  - małe: kody $97 \ldots 122$ ($0x61 \ldots 0x7a$).
- **Kasowanie znaku**: kod 127 ($0x7f$).

Ponieważ kod ASCII jest 7-bitowy, a większość komputerów operuje na 8-bitowych bajtach, dodatkowy bit został wykorzystany na powiększenie zbioru kodowanych znaków do 256 symboli.

Powstało wiele różnych rozszerzeń ASCII wykorzystujących ósmy bit. W kodach tych pierwsze 128 pozycji jest identyczne, jak w kodzie ASCII, a następne 128 pozycji zawiera znaki dodatkowe, np. litery akcentowane, rozszerzony zestaw symboli matematycznych, litery alfabetów narodowych (słowiańskie, skandynawskie, cyrylica, etc.).

Istnieje wiele rozszerzeń tej rodziny, używanych w różnych częściach świata. W Polsce najpowszechniej używa się kodów **ISO8859-2** oraz Microsoft **CP1250**, nazywanych stronami kodowymi.

BIN	DEC	HEX	znak	BIN	DEC	HEX	znak
00000000	0	00	NUL (*Null*)	01000001	65	41	A
00000001	1	01	SOH (*Start Of Heading*)	01000010	66	42	B
00000010	2	02	STX (*Start of Text*)	01000011	67	43	C
00000011	3	03	ETX (*End of Text*)	01000100	68	44	D
...				...			
00110000	48	30	0	01110111	119	77	w
00110001	49	31	1	01111000	120	78	x
00110010	50	32	2	01111001	121	79	y
00110011	51	33	3	011 1010	122	7A	z
...				...			
00111101	61	3D	=	01111100	124	7C	|
00111110	62	3E	>	01111101	125	7D	}
00111111	63	3F	?	01111110	126	7E	~
01000000	64	40	@	01111111	127	7F	DEL (*Delete*)
...				...			

Unicode

Unicode – uniwersalny komputerowy zestaw znaków mający w zamierzeniu obejmować wszystkie znakówi pisarskie zapisu fonetycznego (głoskowego) używanych na całym świecie. Liczba pozycji kodowych jest praktycznie nieograniczona, obecnie jest zdefiniowanych kilkadziesiąt tysięcy znaków.

Definiują go dwa standardy – Unicode oraz **ISO10646**. Znaki obu standardów są identyczne. Standardy te różnią się w drobnych kwestiach, m.in. Unicode określa sposób składu.

Unicode jest rozwijany jest przez konsorcjum, w którego skład wchodzą firmy komputerowe, producenci oprogramowania, instytuty naukowe, agencje międzynarodowe oraz grupy zainteresowanych użytkowników. Konsorcjum współpracuje z organizacją ISO.

Standard Unicode obejmuje przydział przestrzeni numeracyjnej poszczególnym grupom znaków, nie obejmuje zaś sposobów bajtowego kodowania znaków.

Jest kilka metod kodowania, oznaczanych skrótowcami **UCS** (*Universal Character Set*) i **UTF** (*Unicode Transformation Format*). Do najważniejszych należą:

- UTF-32/UCS-4,
- UTF-16,
- UTF-8.

Kody pierwszych 256 znaków Unicode pokrywają się z kodami ISO Latin 1 (czyli ISO8859-1), przez co kody pierwszych 128 znaków pokrywają się z kodami ASCII.

Należy jednak pamiętać, że jest to zbieżność wyłącznie numerów przyporządkowanych konkretnym znakom, wartości bajtów użytych do ich zapisania mogą się różnić od tych, które uzyska się stosując Latin 1 lub ASCII.

UTF-8 – zmiennej długości system kodowania znaków dla Unicode. Może reprezentować każdy znak w systemie Unicode. Kodowanie zostało zaprojektowane dla zapewnienia zgodności z ASCII.

Zalety kodowania UTF-8:

- każdy tekst w ASCII jest tekstem w UTF-8,
- żaden znak spoza ASCII nie zawiera bajtu z ASCII,
- typowy tekst ISO Latin-x rozrasta się w bardzo niewielkim stopniu po przekonwertowaniu do UTF-8,
- znaki o kodzie różnym od 0 nie zawierają bajtu 0, co pozwala stosować UTF-8 w ciągach zakończonych zerem,
- o każdym bajcie wiadomo, czy jest początkiem znaku, czy też leży w jego środku,
- nie zawiera bajtów 0$x$ff i 0$x$fe, więc łatwo można go odróżnić od tekstu UTF-16.
- brak problemów z uporządkowaniem bajtów (*endianness*),
- jest domyślnym kodowaniem w ML (również w jego aplikacjach: HTML, SVG, XSL, CML, MathML).

Dźwięk i obraz

Dźwięk jest zapamiętywany w postaci wartości napięcia reprezentującego chwilowe ciśnienie akustyczne. Wartość ta jest kwantowana z użyciem rozpiętości zwykle od 8 bitów do 32 bitów z częstotliwością zależną od potrzeb, zwykle od 8 kHz do 48 kHz (próbkowanie).

Obraz rastrowy jest zapisany w postaci prostokątnej macierzy punktów (**pikseli**). Każdemu pikselowi odpowiada jeden kolor, zapisany w postaci składowych – jasności świateł (barw) podstawowych. Wartości jasności zapisane są w postaci liczb.

Dane logiczne

Najprostszy typ danych stanowią dane **logiczne**. Mogą one przyjmować dwie wartości. Bajtowe adresowanie danych używane w komputerach oraz fakt, że wiele komputerów traktuje jako podstawowy format danych słowo 32-bitowe powodują, że dane logiczne są zwykle zapisywane w postaci bajtów lub słów, pomimo, że do ich zapisu wystarczyłby pojedynczy bit.

Należy zwrócić uwagę na reprezentację wartości *prawda* i *fałsz* w różnych językach programowania. Wartość *fałsz* jest zwykle reprezentowana przez słowo o wszystkich bitach równych 0.

Wartość *prawda* może być reprezentowana przez liczbę całkowitą równą 1, liczbę całkowitą o dowolnej wartości różnej od zera (również ujemną), słowo o wszystkich bitach równych 1.

Liczby całkowite nieujemne

W dalszej będziemy posługiwali się założeniem, że dane reprezentowane są przez słowo komputera, w którym poszczególne bity zostały ponumerowane od prawej do lewej strony.

NBC

Kod **NBC** (*Natural Binary Code*, NKB, naturalny kod binarny) jest w zasadzie tym samym co pozycyjny system dwójkowy. Numer bitu jest równy wykładnikowi jego wagi binarnej.

$$L_{NCB} = \sum_{i=0}^{n-1} b_i \cdot 2^i.$$

Wartości wag w kodzie NBC (dla bajtu)

waga	$2^7 = 128$	$2^6 = 64$	$2^5 = 32$	$2^4 = 16$	$2^3 = 8$	$2^2 = 4$	$2^1 = 2$	$2^0 = 1$
cyfra	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0

Liczby całkowite ze znakiem

2C

Zapis **2C** (*Two's Complement*, U2, uzupełnień do 2) jest najczęściej stosowanym zapisem liczb całkowitych. Jest on podobny do NKB, z tą różnicą, że najbardziej znaczący bit ma wagę ujemną. Typ `int` jest we współczesnych komputerach implementowany jako zapis 2C. Negacja liczby w tym systemie polega na negacji bitowej i inkrementacji.

$$L_{2C} = -b_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} b_i \cdot 2^i.$$

Wartości wag w kodzie 2C

waga	$-(2^8) = -128$	$2^6 = 64$	$2^5 = 32$	$2^4 = 16$	$2^3 = 8$	$2^2 = 4$	$2^1 = 2$	$2^0 = 1$
cyfra	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0

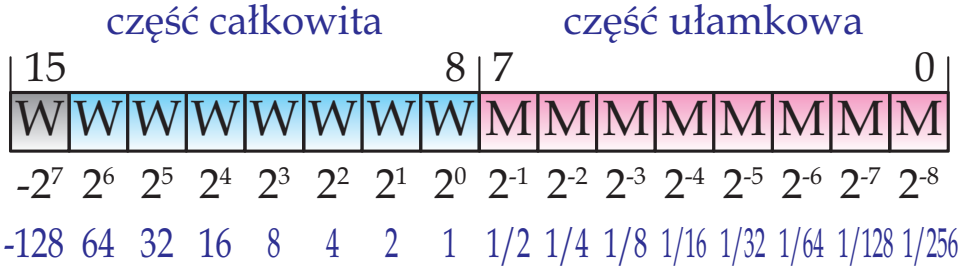
Liczby niecałkowite

System stałopozycyjny

Do zapisywania liczb ułamkowych i mieszanych można użyć zapisu **stałopozycyjnego**. W zapisie tym liczba jest reprezentowana przez słowo binarne, w którym pewne, z góry określone liczby bitów reprezentują część całkowitą i część ułamkową liczby.

Odpowiada to interpretacji zapisu całkowitoliczbowego, pomnożonej przez wartość będącą ujemną potęgą liczby 2.

Do zapisu liczb bez znaku używa się jako bazowej postaci NKB, a do zapisu liczb ze znakiem kodu U2 (zwykle).



Komputery zazwyczaj nie obsługują w szczególny sposób zapisów stałopozycyjnych. Podstawowe operacje są wykonywane tak samo, jak na liczbach całkowitych, odmienna jest jedynie interpretacja zapisu, za którą jest odpowiedzialny wyłącznie programista.

Zapis zmiennopozycyjny dla systemu dziesiętnego

Zapis **zmiennopozycyjny** umożliwia zapisywanie liczb całkowitych i ułamkowych o bardzo dużym zakresie dynamiki wartości bezwzględnych.

Każda liczba może być zapisana na kilka sposobów, różniących się położeniem przecinka oddzielającego część całkowitą od ułamkowej i wartością wykładnika:

$$-1,2345 \cdot 10^5 \qquad -0,12345 \cdot 10^6 \qquad -12,345 \cdot 10^4.$$

Wartość liczby zmiennoprzecinkowej jest obliczana według wzoru:

$$L = Z \cdot M \cdot B^W,$$

gdzie:

- Z (*sign*) – znak liczby, 1 lub -1,
- M (*mantissa*) – mantysa, liczba ułamkowa,
- B (*base*) – podstawa systemu liczbowego,
- W (*exponent*) – wykładnik (cecha), liczba całkowita.

Postacią **znormalizowaną** nazwiemy liczbę, która ma część całkowitą części znaczącej wyrażoną przez pojedynczą cyfrę różną od zera.

Aby zapisać (przechować) liczbę, musimy zapisać jej znak, część znaczącą oraz wykładnik, który jest liczbą całkowitą ze znakiem. Podstawa systemu jest ustalona i jej zapis jest zbędny.

$$-1,2345e5$$

W postaci znormalizowanej nie da się zapisać zera, ponieważ zero nie ma żadnej cyfry znaczącej różnej od 0.

Binarny zapis zmiennopozycyjny

Obecnie niemal wszystkie komputery posługują się binarnym zapisem zmiennopozycyjnym zgodnym ze standardem IEEE754.

Standard zakłada, że, o ile tylko jest to możliwe, liczby zapisuje się w postaci znormalizowanej. **Bazą** systemu jest liczba 2, **wykładnik** określa potęgę liczby 2.

Ponieważ w systemie binarnym jedyną cyfrą różną od zera jest jedynka, każda liczba w postaci znormalizowanej ma część całkowitą równą 1, nie ma więc potrzeby zapisywania jej, zapisuje się tylko część ułamkową.

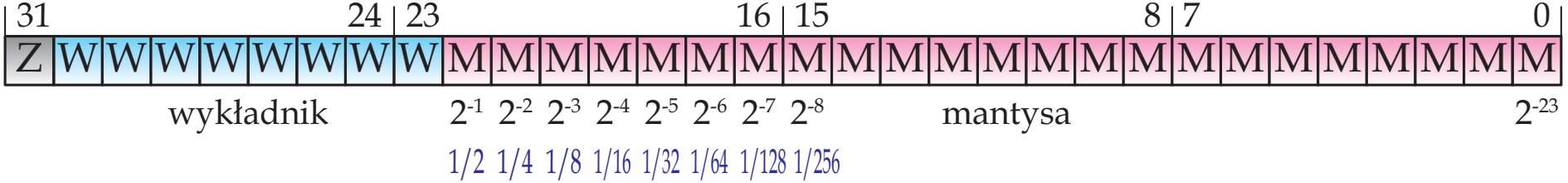
Wykładnik jest liczbą całkowitą ze znakiem. W IEEE754 wykładnik jest zapisywany w kodzie **spolaryzowanym** (biased), w którym wartość podkładu jest określona wzorcem bitowym $01\ldots11$, o liczbie bitów równej szerokości pola bitowego wykładnika. Dwie wartości pola wykładnika są zarezerwowane i oznaczają, że zapis nie reprezentuje postaci znormalizowanej.

Bity wykładnika o wzorcu $00\ldots00$ oznaczają zapis zdenormalizowany. Wartość wykładnika jest w tym przypadku taka sama, jak przy zapisie znormalizowanym z wzorcem wykładnika $00\ldots01$, a część całkowita części znaczącej ma wartość 0 (a nie 1 jak w postaci znormalizowanej).

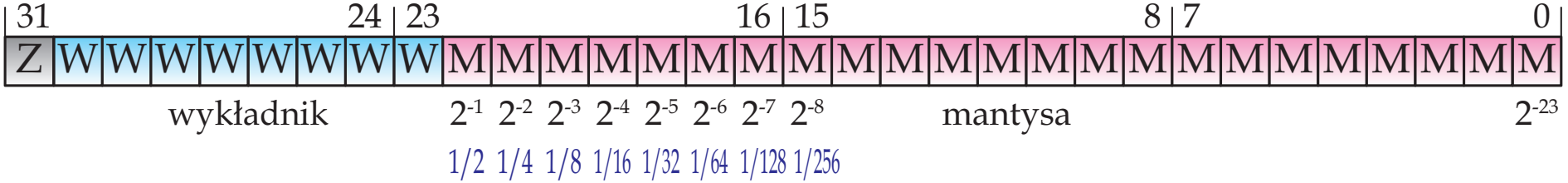
Pole wykładnika o wzorcu $11\ldots11$ oznacza nie-liczby: wartości **nieskończone** i wartości **błędne**.

Liczba pojedynczej precyzji zajmuje 4 bajty (32 bity).

Mantysa ma 23 bity, wykładnik ma 8 bitów, znak 1 bit, a nadmiar wynosi 127.



Mantysa jest **znormalizowana**, tj. należy do przedziału $[1, 2)$ (przedział prawostronnie otwarty). Jeżeli M jest stałe, a W zmienia się, wówczas przesunięciu ulega przecinek – stąd właśnie pochodzi nazwa tej reprezentacji.



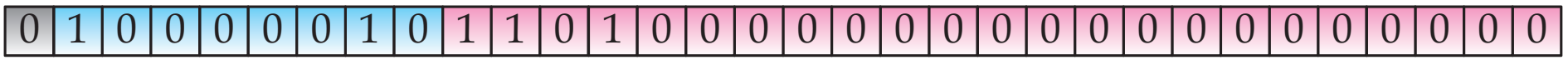
Procedura zapisu liczby L:

- określenie znaku, bit 31 $= 1$ jeżeli L jest ujemna, 0 jeśli dodatnia,
- znalezienie największej liczby postaci 2^W, mniejszej niż L,
- zapisanie wykładnika z nadmiarem $W + 127$,
- dzielenie liczby $L/2^W$ (wynik 1.xxxx),
- odjęcie 1 i szukanie mantysy,
- zaznaczenie bitu jako 1 jeśli po odjęciu $1/2$ (bit $+ 1$) mamy wartość nieujemną, jeśli wartość jest ujemna, bit oznaczany jest jako 0, i ignorowana operacja odejmowania,
- procedura jest powtarzana aż w wyniku odejmowania otrzymamy 0 lub dojdzie do bitu nr 0.

Przykład dla liczby 14,5:

- liczba jest dodatnia $\Rightarrow$ bit 31 $= 0$,
- największa liczba 2^W mniejsza niż 14,5 to $3^3 = 8 \Rightarrow W = 3$,
- zapisanie wykładnika $127 + W = 130 \Rightarrow 10000010$,
- dzielenie $14,5/2^3 = 1,8125$,
- odejmowanie $1,8125 - 1 = 0,8125$,
  - $0,8125 - 1/2 = 0.3125 \Rightarrow$ bit 23 $= 1$,
  - $0,3125 - 1/4 = 0.0625 \Rightarrow$ bit 22 $= 1$,
  - $0,0625 - 1/8 = -0.0625 \Rightarrow$ bit 21 $= 0$ (ignorowanie operacji),
  - $0,0625 - 1/16 = 0 \Rightarrow$ bit 20 $= 1$,
- pozostałe bity mantysy $= 0$.

14,5



Niektóre wartości specjalne

wartość,	wykładnik,	mantysa,	uwagi
NaN	11...11	xx...xx	symbol nie reprezentuje wartości liczbowej, powstaje zazwyczaj w wyniku niedozwolonej operacji (np. pierwiastkowanie liczby ujemnej)
INF	11...11	00...00	bit znaku decyduje o znaku wartości, rozróżnia się $+\infty$ i $-\infty$
0	00...00	00...00	bit znaku decyduje o znaku wartości, jest wynikiem operacji w przypadku wystąpienia nadmiaru (przepełnienia), przy dzieleniu przez 0, itp., może być dodatnia lub ujemna

Standard IEEE754, definiuje dwie klasy liczb:

- pojedynczej precyzji (`single`),
- podwójnej precyzji (`double`).

Podstawowym formatem jest format typu `double` – 64-bitowy (podwójnej precyzji).

Format 32-bitowy jest używany w zastosowaniach, gdzie wymagana precyzja jest niewielka, ma on tylko 23 bity znaczące (typ `single`).

Liczba **pojedynczej** precyzji: mantysa 23 bity, wykładnik 8 bitów, znak 1 bit, nadmiar $2^8 - 1 = 127$.

Liczba **podwójnej** precyzji: mantysa 52 bity, wykładnik 11 bitów, znak 1 bit, nadmiar $2^{11} - 1 = 1023$.

Format 80-bitowy (`extended`) był używany w starszych jednostkach zmiennopozycyjnych procesorów rodziny x86. Obecnie wychodzi z użycia.

Format 128-bitowy jest formatem *przyszłościowym* dla liczb o dużej precyzji.

Precyzja – liczba miejsc po przecinku jest określona przez mantysę.

Pojedyncza precyzja
Mantysa ma 23 bity, $1/2^{23} \approx 1,2 \cdot 10^{-7} \Rightarrow$ 7 cyfr po przecinku.

Podwójna precyzja
Mantysa ma 52 bity, $1/2^{52} \approx 2,2 \cdot 10^{-16} \Rightarrow$ 15 - 16 cyfr po przecinku.

Rozszerzona precyzja
Mantysa ma 64 bity, $1/2^{64} \approx 5,4 \cdot 10^{-20} \Rightarrow$ 19 cyfr po przecinku.

Operacje arytmetyczne

Mamy dwie liczby zmiennoprzecinkowe: $L_1 = M_1 \cdot B^{W_1}$ i $L_2 = M_2 \cdot B^{W_2}$, przy czym $L_1 > L_2$, wówczas:

- dodawanie
$$L_1 + L_2 = \left(M_1 + M_2 \cdot B^{W_2 - W_1}\right) \cdot B^{E_1},$$
- odejmowanie
$$L_1 - L_2 = \left(M_1 - M_2 \cdot B^{W_2 - W_1}\right) \cdot B^{E_1}.$$

Mamy dwie liczby zmiennoprzecinkowe: $L_1 = Z_1 \cdot M_1 \cdot B_1^W$ i $L_2 = Z_2 \cdot M_2 \cdot B_2^W$, wówczas:

- mnożenie
$$L_1 \cdot L_2 = (Z_1 \cdot Z_2) \cdot (M_1 \cdot M_2) \cdot B^{W_1 + W_2},$$
- dzielenie
$$L_1 / L_2 = (Z_1 \cdot Z_2) \cdot (M_1 / M_2) \cdot B^{W_1 - W_2}.$$

Jeśli liczby mają różne wykładniki, to podczas dodawania mantysa liczby o mniejszym wykładniku musi zostać **zdenormalizowana** – we wzorze jest to przemnożenie M_2 przez czynnik $B^{W_2-W_1}$.

W szczególnym przypadku, jeśli $W_1 - W_2$ jest większe niż liczba cyfr mantysy, to po denormalizacji mantysa będzie miała wartość 0, a liczba o mniejszym wykładniku nie wpłynie na wynik dodawania bądź odejmowania.

Liczby o skończonej reprezentacji dziesiętnej mogą mieć nieskończoną reprezentację binarną (np.: 0,1; 0,3).

Reprezentacja zmiennopozycyjna jest reprezentacją przybliżoną, a wyniki operacji są w rzeczywistości przybliżeniami. Oznacza to, że w praktyce nie można stosować relacji równości w odniesieniu do liczb zmiennopozycyjnych ($|a - b| < \epsilon$).

Arytmetyka liczb zmiennopozycyjnych **nie jest łączna**, dla x i y może zachodzić:

$$(x + y) + z \neq x + (y + z),$$
$$(x \cdot y) \cdot z \neq x \cdot (y \cdot z),$$

nie jest też rozdzielna, czyli:

$$x \cdot (y + z) \neq (x \cdot y) + (x \cdot z).$$

Innymi słowy, kolejność wykonywania operacji arytmetycznych ma znaczenie i wpływa na końcowy wynik.

Potrzeby:

- projektowanie,
- modelowanie,
- symulacja,
- analiza wyników.

Narzędzia:

- obliczenia algebraiczne,
- rozwiązywanie równań i układów równań,
- interpolacja i aproksymacja,
- symulacja,
- prezentacja wyników.

System algebry komputerowej

System algebry komputerowej lub komputerowy system obliczeń symbolicznych (*computer algebra system*, *CAS*) – program komputerowy wspomagający obliczenia symboliczne w matematyce, fizyce i dyscyplinach technicznych.

Typowe wyrażenia z jakimi operują tego rodzaju programy zbudowane są z wielomianów jednej lub wielu zmiennych, funkcji elementarnych, macierzy lub macierzy całek i pochodnych takich wyrażeń.

Typowe operacje wykonywane na wyrażeniach:

- upraszczanie wyrażeń,
- podstawianie wyrażeń symbolicznych za zmienne i redukcja wyrazów podobnych,
- rozwijanie iloczynów,
- rozkład wyrażeń na czynniki,
- różniczkowanie symboliczne,
- całkowanie symboliczne – całki oznaczone i nieoznaczone,
- symboliczne rozwiązywanie niektórych typów równań i ich układów,
- rozwiązywanie równań różniczkowych określonych typów,

- obliczanie granic funkcji i ciągów,
- obliczanie sum szeregów,
- rozwijanie funkcji w szereg,
- operacje na macierzach – mnożenie, odwracanie, obliczanie wyznacznika,
- obliczenia związane z teorią grup,
- obliczenia związane ze statystyką matematyczną,
- operacje na listach i zbiorach elementów,
- eksport wyników obliczeń do formatu TeX-a i EPS.

Większość programów CAS umożliwia rysowanie wykresów funkcji (jednej i dwu zmiennych oraz zmiennej zespolonej) i przeprowadzanie obliczeń z praktycznie dowolną dokładnością.

Wiele z nich ma wbudowane języki programowania, dzięki czemu użytkownik może wykorzystywać do rozwiązywania zadań własne algorytmy i zwiększać w ten sposób funkcjonalność programu.

Systemy algebry komputerowej należy właściwie odróżnić od programów przeznaczonych do obliczeń **numerycznych i inżynierskich** lub języków programowania wyposażonych w dodatkowe biblioteki do obliczeń i symulacji komputerowych.

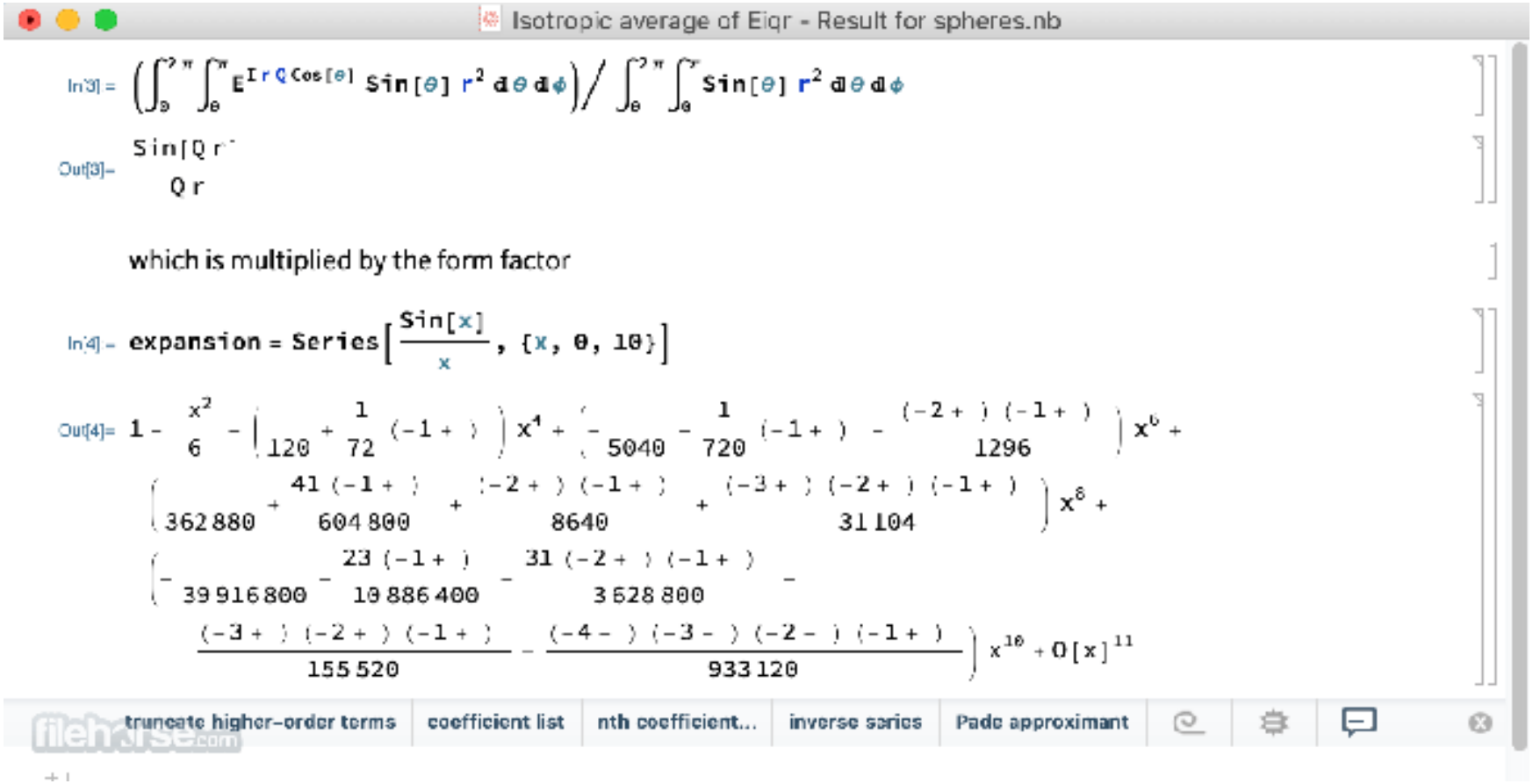
Rozgraniczenie nie jest jednakże jednoznaczne, gdyż niektóre systemy algebraiczne zawierają moduły umożliwiające szybkie obliczenia numeryczne i na odwrót.

Wspomaganie obliczeń matematycznych

rodzaj rachunku	numeryczny	symboliczny
rozwiązywanie trudnych zadań praktycznych	zazwyczaj tak	zazwyczaj nie
dostępność metod o różnej skuteczności	tak	tak
wymaga wiedzy wykraczającej poza rozwiązywane zadanie	najczęściej tak	najczęściej nie
wynik	skończony zestaw liczb lub rysunek	wzór lub informacja o rozwiązaniu

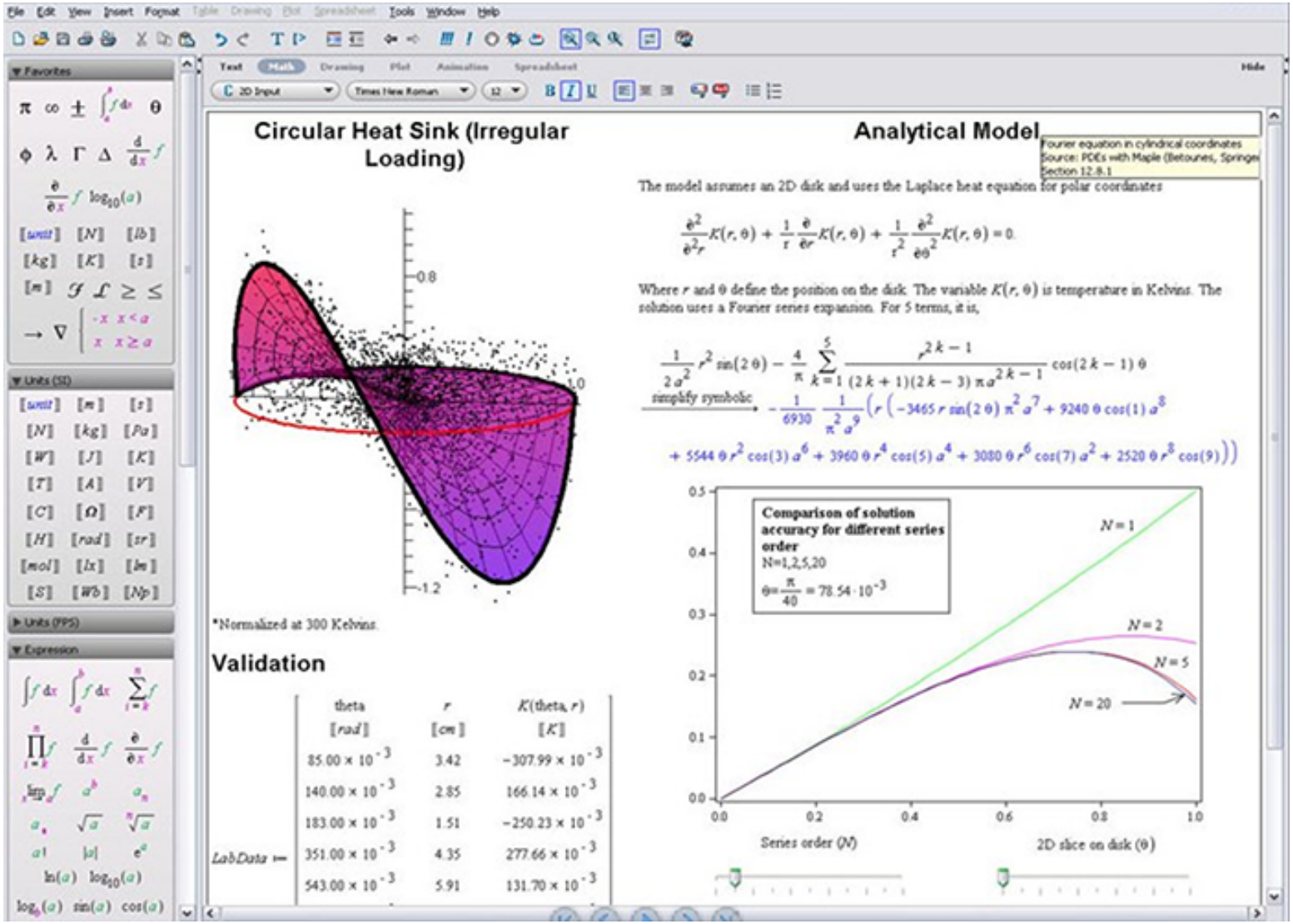
rodzaj rachunku	numeryczny	symboliczny
potrafi działać na abstrakcyjnych obiektach	nie	tak
dobrze radzi sobie z nieskończonościami	zazwyczaj nie	zazwyczaj tak
dobrze radzi sobie z mnogością parametrów	tak	nie
precyzja wyniku	ograniczona	nieskończona
ostateczna jakość wyniku	niepewna	niepewna

Mathematica



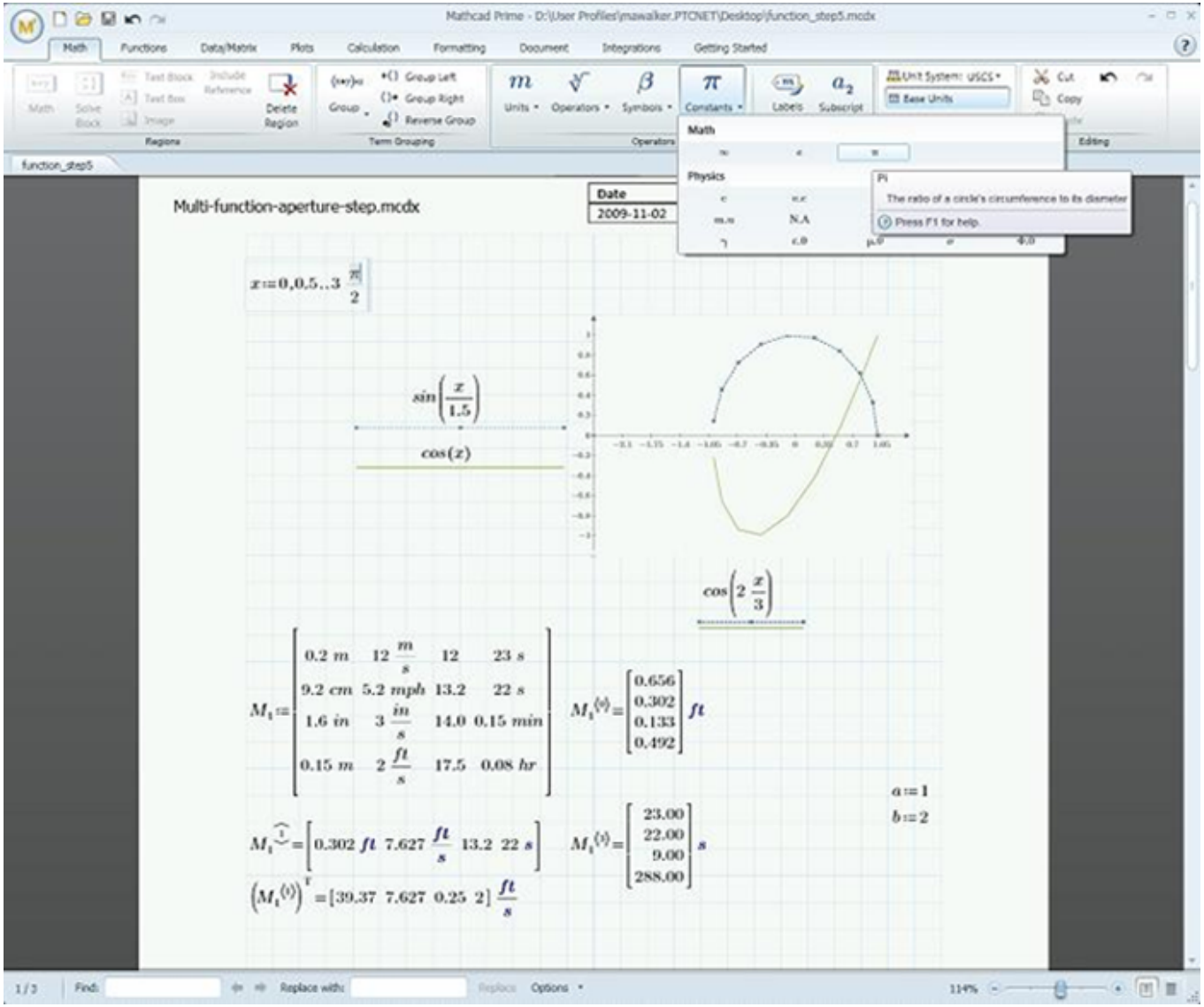
rachunek symboliczny, komercyjny

Maple



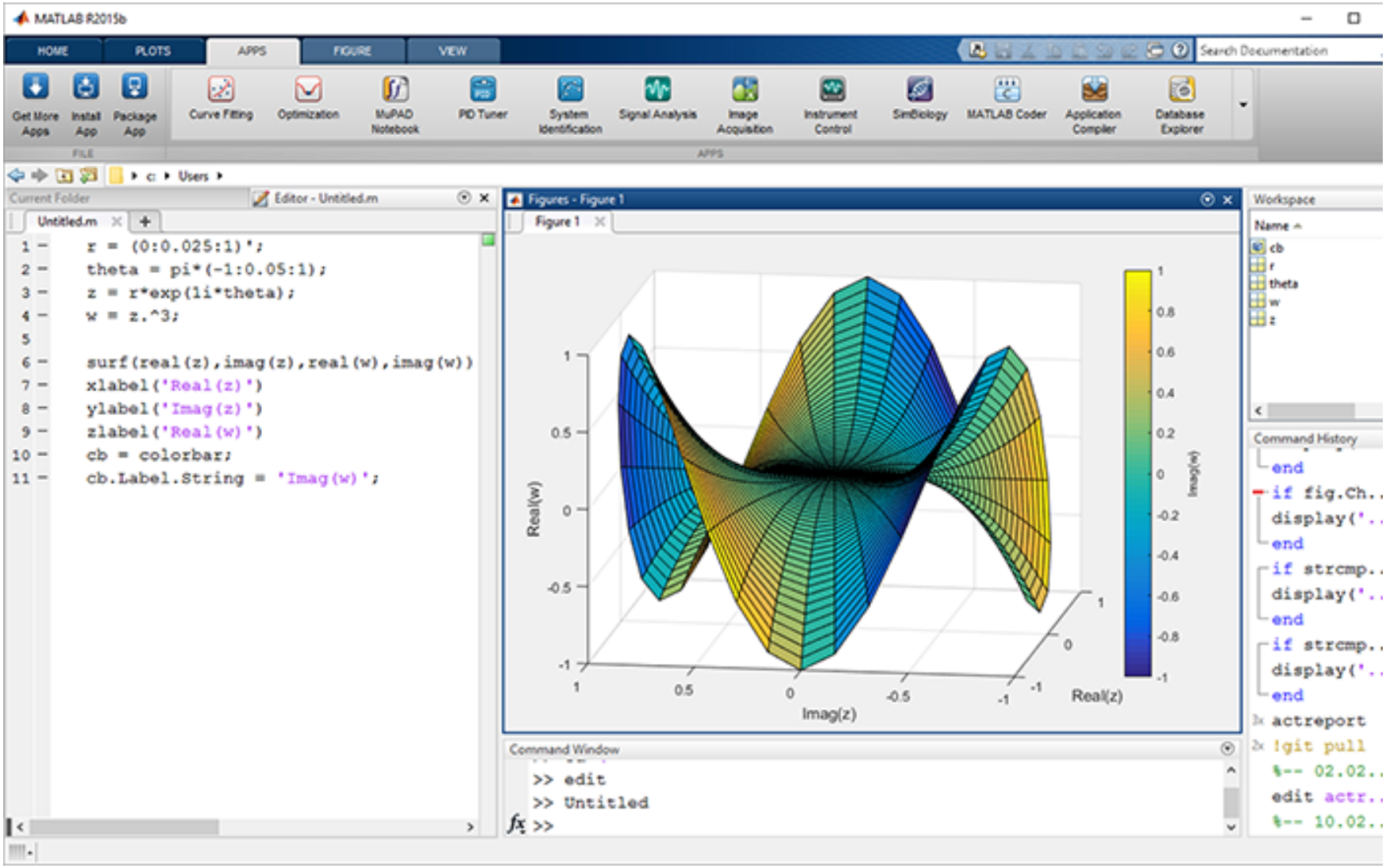
rachunek symboliczny, komercyjny

Mathcad



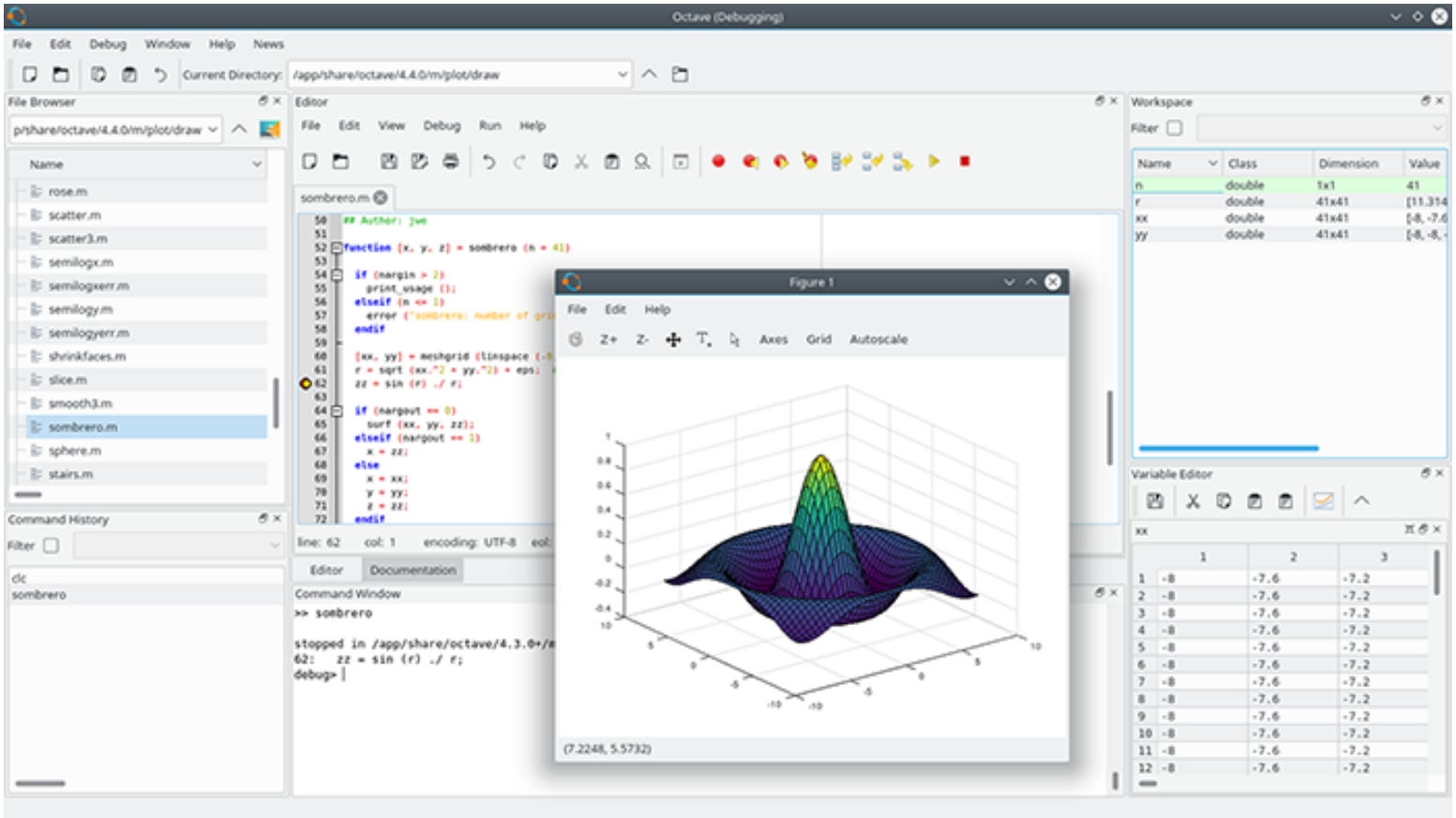
rachunek symboliczny, metody numeryczne, jednostki, komercyjny

MATLAB



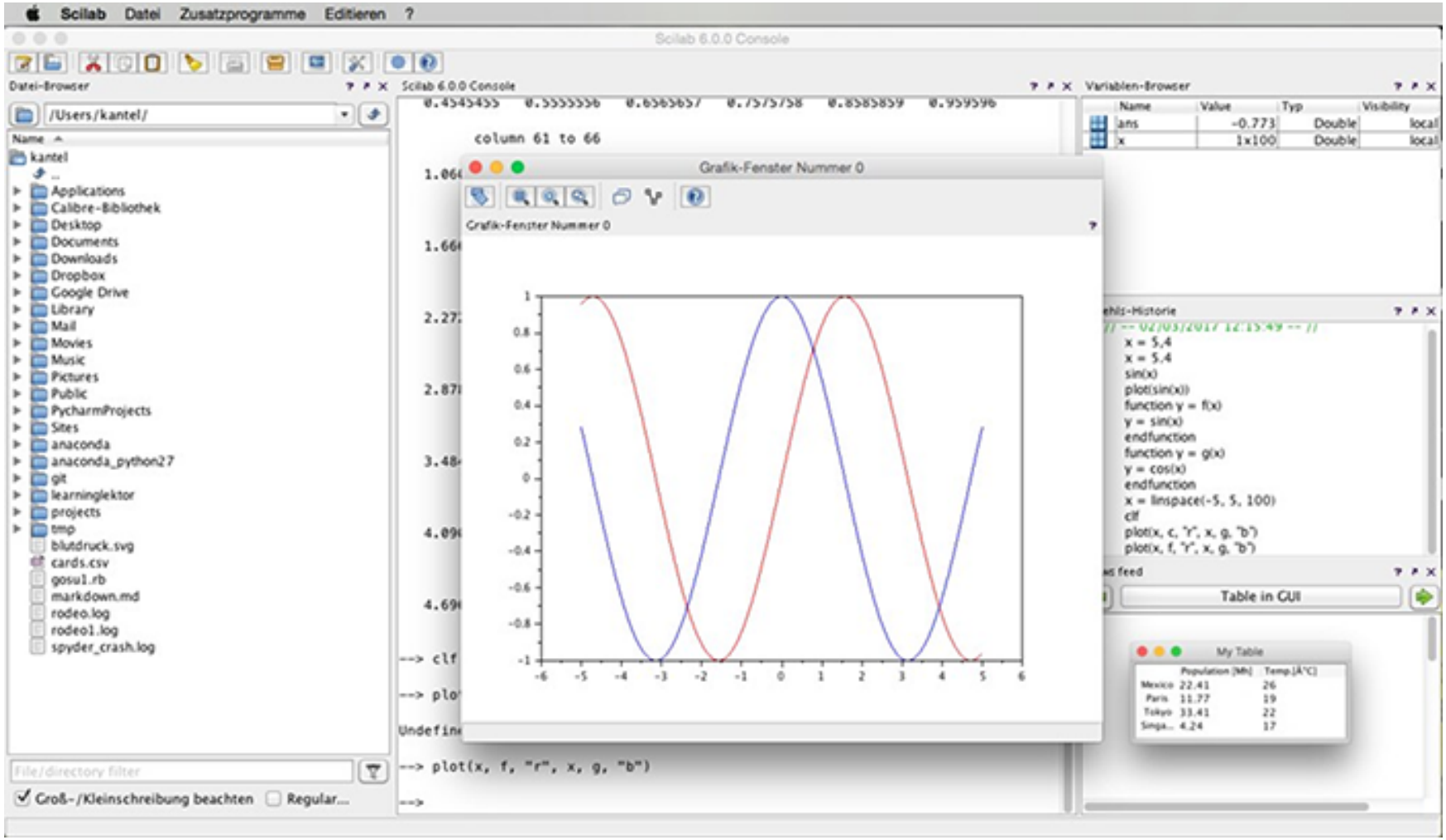
rachunek numeryczny, rachunek symboliczny, modele symulacyjne, własny język programowania, komercyjny

Octave



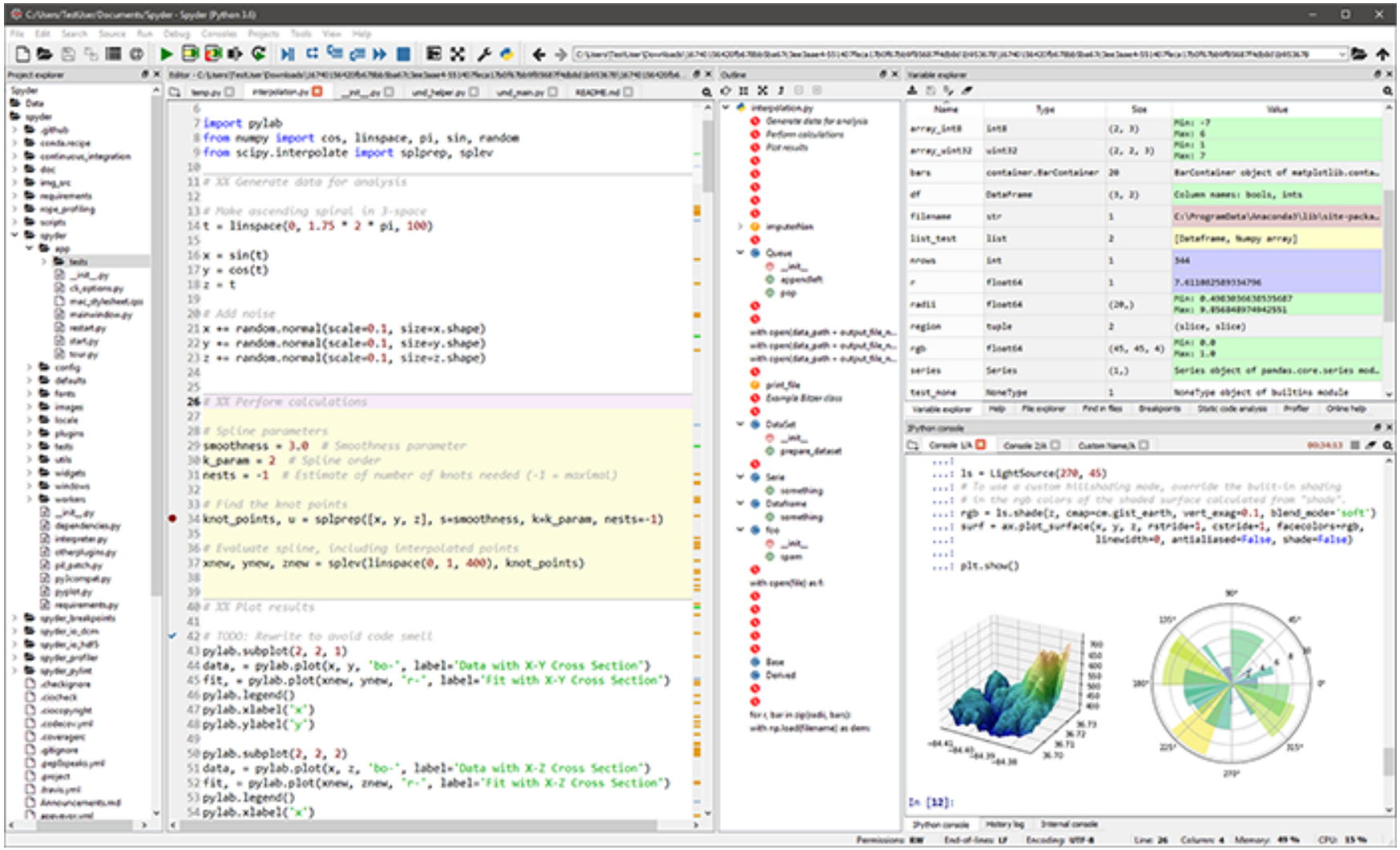
rachunek numeryczny, częściowo zgodny z systemem MATLAB, darmowy

Scilab



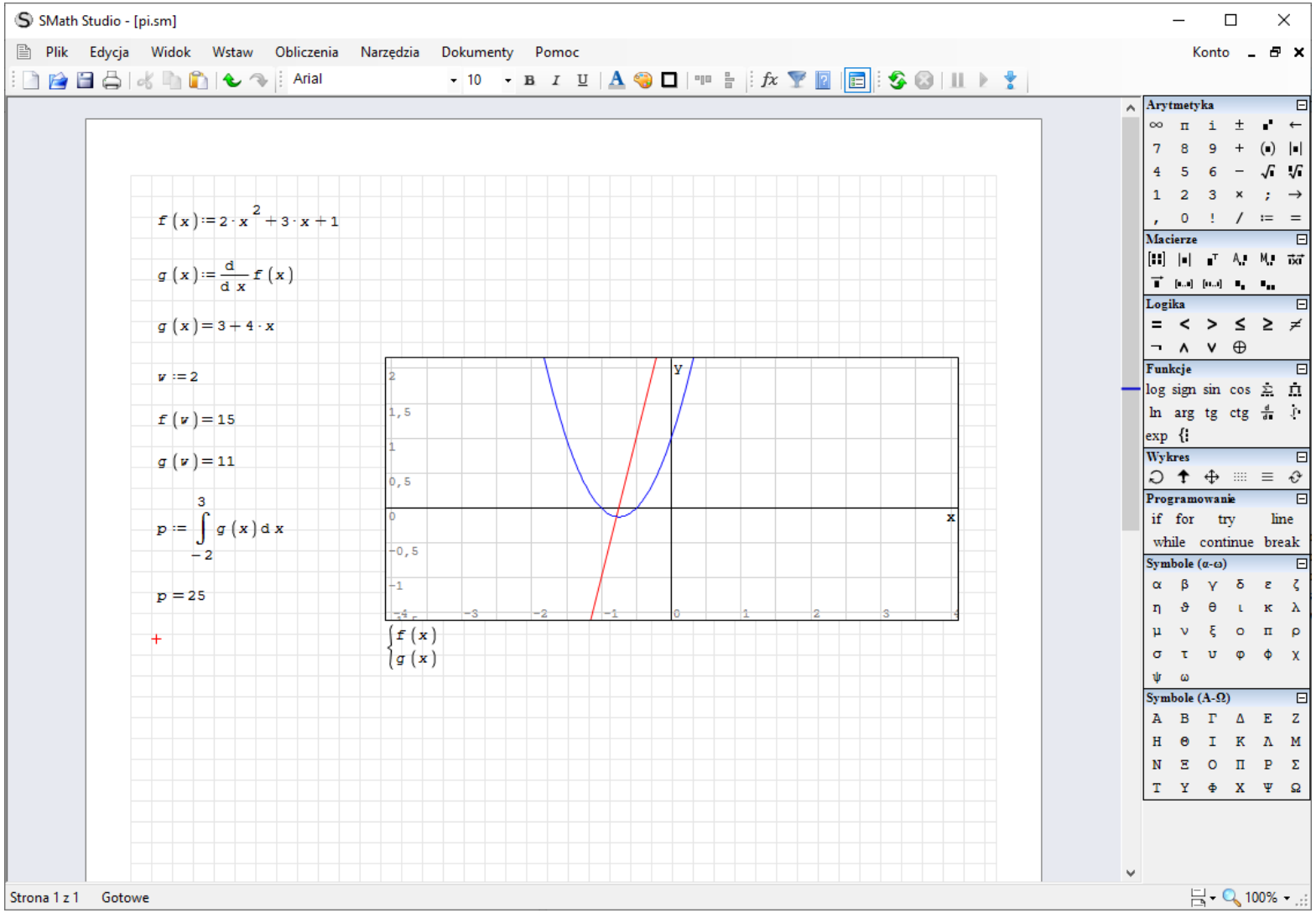
rachunek numeryczny, darmowy

Python (NumPy, SciPy, Matplotlib, Spyder)



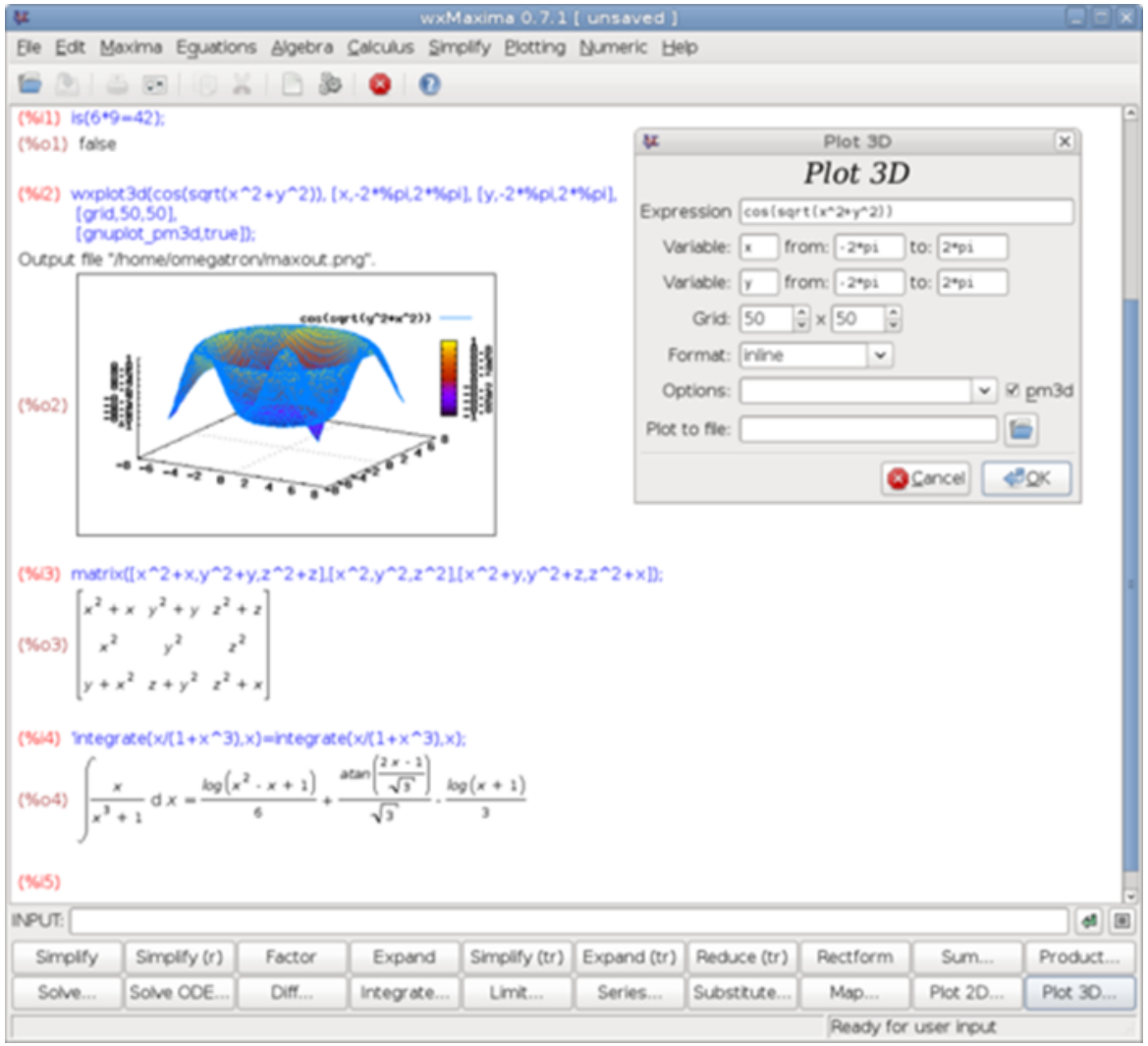
język programowania ogólnego przeznaczenia, biblioteki numeryczne, darmowy

SMath Studio



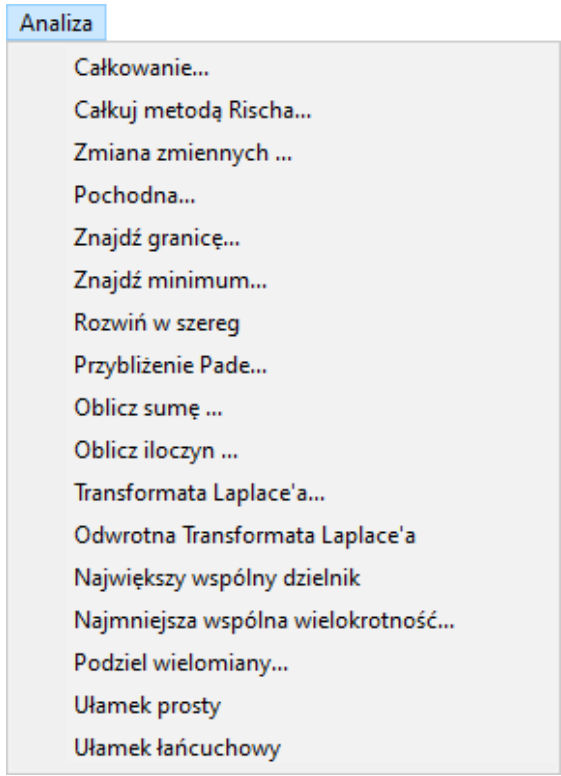
rachunek symboliczny, darmowy

Maxima



rachunek symboliczny, darmowy


Komórka
Wykonaj komórki
Evaluate All Visible Cells Ctrl+R
Evaluate All Cells Ctrl+Shift+R
Evaluate Cells Above Ctrl+Shift+P
Evaluate Cells Below
Usuń wszytskie komórki wyjściowe
Kopiuj ostatnie wejście Ctrl+I
Copy Previous Output Ctrl+U
Complete Word Ctrl+Space
Show Template Ctrl+Shift+Space
Insert Input Cell Ctrl+0
Wstaw komórkę tekstowa Ctrl+1
Wstaw komórkę tytułową Ctrl+2
Wstaw komórkę rozdziału Ctrl+3
Wstaw komórkę podrozdziału Ctrl+4
Insert Subsubsection Cell Ctrl+5
Insert heading5 Cell Ctrl+6
Insert heading6 Cell Ctrl+7
Wstaw znak końca strony
Wstaw obrazek...
Fold All Ctrl+Alt+[
Unfold All Ctrl+Alt+]
Poprzenie polecenie Alt+Up
Następne polecenie Alt+Down
Merge Cells Ctrl+M
Divide Cell Ctrl+D
Automatically answer questions

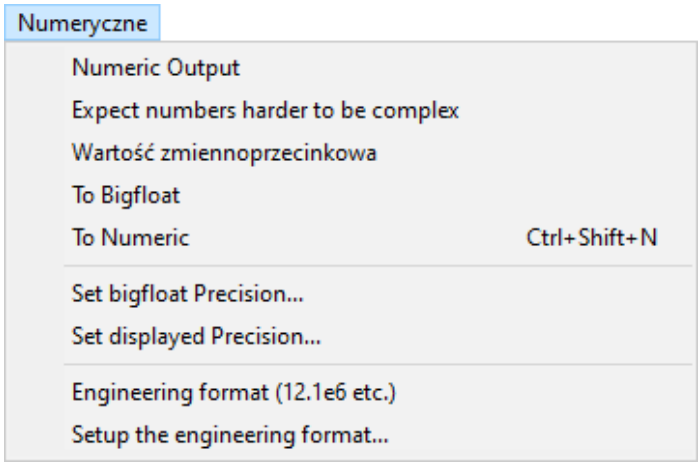

Analiza
Całkowanie...
Całkuj metodą Rischa...
Zmiana zmiennych ...
Pochodna...
Znajdź granicę...
Znajdź minimum...
Rozwiń w szereg
Przybliżenie Pade...
Oblicz sumę ...
Oblicz iloczyn ...
Transformata Laplace'a...
Odwrotna Transformata Laplace'a
Największy wspólny dzielnik
Najmniejsza wspólna wielokrotność...
Podziel wielomiany...
Ułamek prosty
Ułamek łańcuchowy

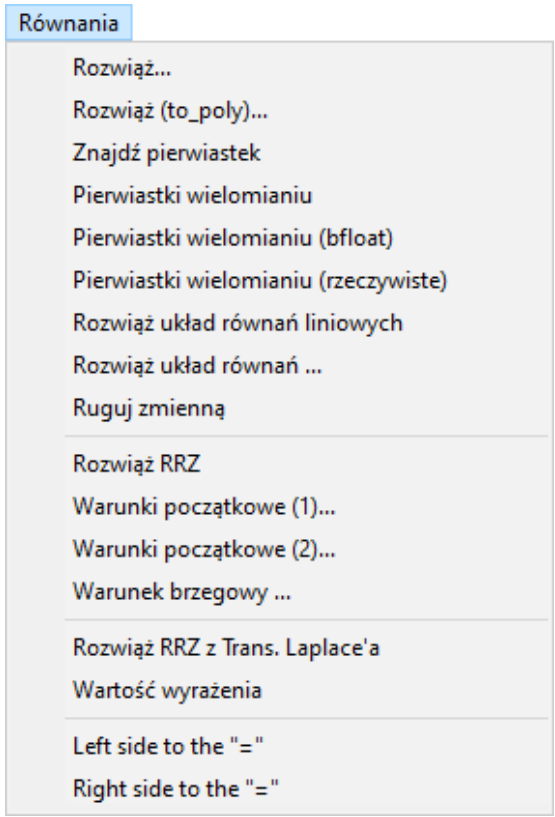
List

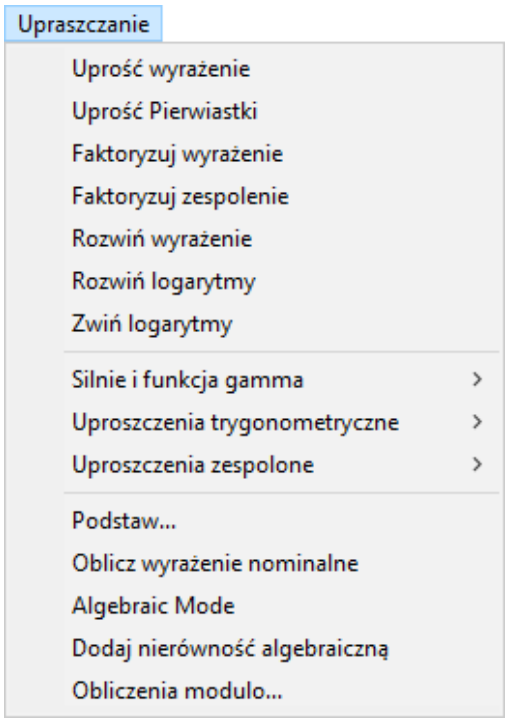
- Create list ›
- Use list ›
- Extract Elements ›
- Append ›
- Length
- Reverse
- Sort
- Remove duplicates
- Push
- Pop
- Nested list to Matrix
- Matrix to nested List

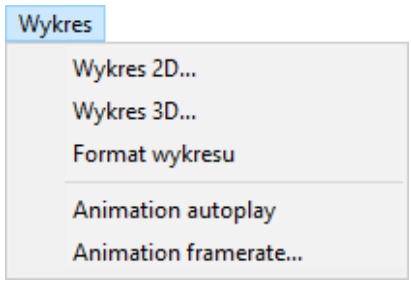
Matrix

- Generuj macierz
- Generuj macierz z wyrazu...
- Wprowadź macierz
- Nested list to Matrix
- Matrix from csv file
- Matrix to csv file
- Macierz odwrotna
- Wielomian charakterystyczny ...
- Wyznacznik
- Wartości własne
- Wektory własne
- Macierz dołączona
- Rank
- Transponuj macierz
- Extract Row
- Extract Column
- Remove Rows or Columns
- Convert Row to list
- Convert Column to list
- Multiply matrices
- Matrix exponent
- Hadamard (element-by-element) product
- Hadamard exponent
- Utwórz listę...
- Zastosuj do listy ...
- Map to List(s)...
- Mapuj macierz...


Numeryczne
Numeric Output
Expect numbers harder to be complex
Wartość zmiennoprzecinkowa
To Bigfloat
To Numeric
Ctrl+Shift+N
Set bigfloat Precision...
Set displayed Precision...
Engineering format (12.1e6 etc.)
Setup the engineering format...


Równania
Rozwiąż...
Rozwiąż (to_poly)...
Znajdź pierwiastek
Pierwiastki wielomianiu
Pierwiastki wielomianiu (bfloat)
Pierwiastki wielomianiu (rzeczywiste)
Rozwiąż układ równań liniowych
Rozwiąż układ równań ...
Ruguj zmienną
Rozwiąż RRZ
Warunki początkowe (1)...
Warunki początkowe (2)...
Warunek brzegowy ...
Rozwiąż RRZ z Trans. Laplace'a
Wartość wyrażenia
Left side to the "="
Right side to the "="


Upraszczanie
Uprość wyrażenie
Uprość Pierwiastki
Faktoryzuj wyrażenie
Faktoryzuj zespolenie
Rozwiń wyrażenie
Rozwiń logarytmy
Zwiń logarytmy
Silnie i funkcja gamma
Uproszczenia trygonometryczne
Uproszczenia zespolone
Podstaw...
Oblicz wyrażenie nominalne
Algebraic Mode
Dodaj nierówność algebraiczną
Obliczenia modulo...


Wykres
Wykres 2D...
Wykres 3D...
Format wykresu
Animation autoplay
Animation framerate...

SMath Studio

To darmowy program algebry komputerowej z zamkniętym kodem.

Cechą charakterystyczną programu **SMath Studio** jest łatwość obsługi. Interfejs programu imituje notatnik i jest intuicyjny w użyciu, a wiele operacji daje się realizować za pomocą myszy. Równania i wyrażenia algebraiczne wyświetlane są w postaci graficznej, a nie tekstowej.

Niektóre możliwości programu:

- obliczenia symboliczne,
- operacje na wektorach i macierzach,
- rozwiązywanie układów równań,
- wykreślanie wykresów funkcji jednej i dwu zmiennych,
- znajdowanie pierwiastków wielomianów i innych funkcji.

SMath Studio - [pi.sm]

Plik Edycja Widok Wstaw Obliczenia Narzędzia Dokumenty Pomoc Konto

Arial 10

$f(x) := 2 \cdot x^2 + 3 \cdot x + 1$

$g(x) := \frac{d}{dx} f(x)$

$g(x) = 3 + 4 \cdot x$

$w := 2$

$f(w) = 15$

$g(w) = 11$

$p := \int_{-2}^{3} g(x)\,dx$

$p = 25$

$\begin{cases} f(x) \\ g(x) \end{cases}$

Arytmetyka
Macierze
Logika
Funkcje
log sign sin cos
ln arg tg ctg
exp
Wykres
Programowanie
if for try line
while continue break
Symbole (α-ω)
Symbole (A-Ω)

Strona 1 z 1 Gotowe 100%

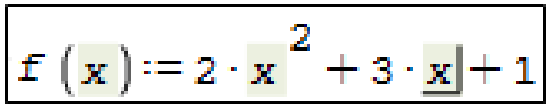
Arkusz SMath Studio

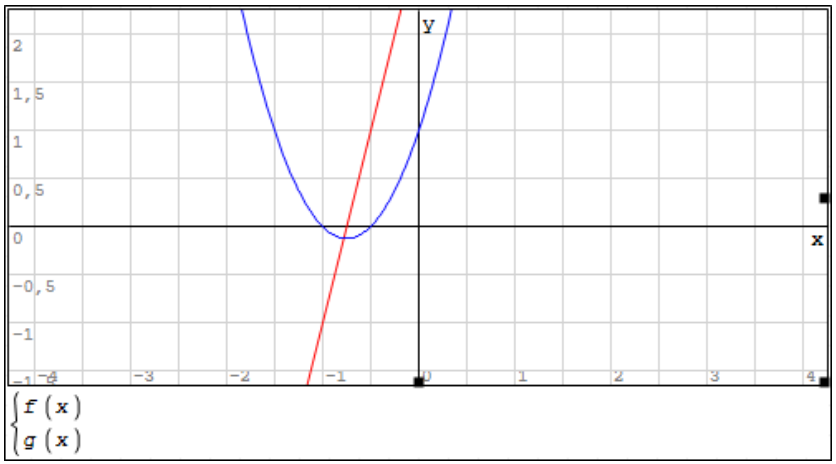
Regiony:

- region wyrażenia matematycznego,
- region tekstu,
- region wykresu.

Kursory:

- czerwony krzyżyk – miejsce wstawienia regionów,
- czerwona linia – kursor edycji tekstu,
- niebieskie linie – kursor edycji wyrażeń matematycznych.



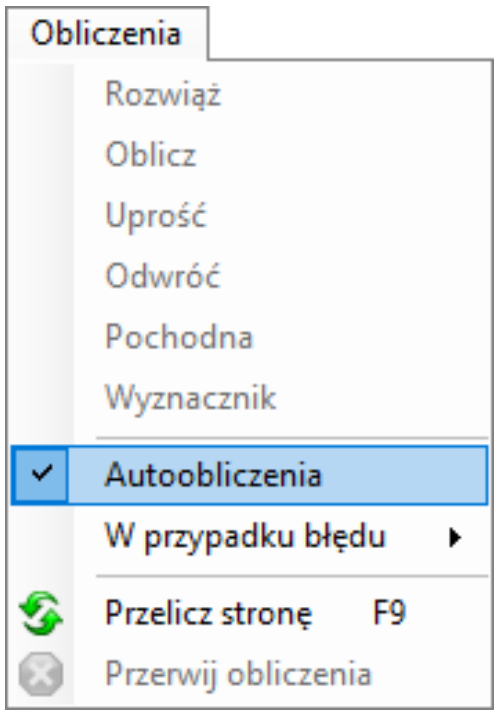


Wyrażenia

$a := 4,321$ $\quad B := \frac{1}{3}$ $\quad f(x) := x^2 - 2 \cdot x + 3$

$\frac{1}{2} + 2,34 \cdot \sqrt[3]{2} = 3,4482$ $\quad f(3) = 6$

- definicja zmiennych i funkcji,
- wyznaczanie wartości wyrażeń,
- edycja wyrażeń,
- automatyczne i ręczne wyznaczanie wartości wyrażeń,



Analiza matematyczna i algebra liniowa

- formatowanie wyników obliczeń,
- rachunek różniczkowy i całkowy,
- definicja zmiennej zakresowej,
- tworzenie wykresów funkcji,

$a := 4,321$ $\quad B := \frac{1}{3}$ $\quad f(x) := x^2 - 2 \cdot x + 3$

$\frac{1}{2} + 2,34 \cdot \sqrt[3]{2} = 3,4482$ $\quad f(3) = 6$

$f(x) := 2 \cdot x^2 + 3 \cdot x + 1$

$g(x) := \frac{d}{dx} f(x)$

$g(x) = 3 + 4 \cdot x$

$w := 2$

$f(w) = 15$

$g(w) = 11$

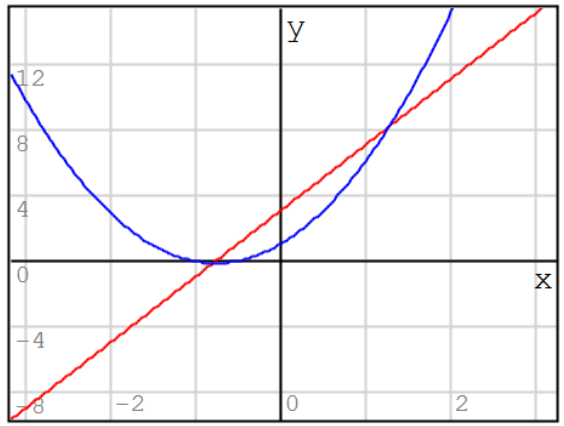
$p := \int_{-2}^{3} g(x)\,dx$

$p = 25$

$10000 \cdot \pi = 31415,9265$

$10000 \cdot \pi = 3,142 \cdot 10^4$

$10000 \cdot \pi = 31415,9265359$



$\begin{cases} f(x) \\ g(x) \end{cases}$

- transpozycja macierzy,
- mnożenie macierzy przez skalar,
- mnożenie macierzy,

$$A := \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix}$$

$$B := A^{T}$$

$$B = \begin{bmatrix} 1 & 4 & 7 & 10 \\ 2 & 5 & 8 & 11 \\ 3 & 6 & 9 & 12 \end{bmatrix}$$

$$A + B^{T} = \begin{bmatrix} 2 & 4 & 6 \\ 8 & 10 & 12 \\ 14 & 16 & 18 \\ 20 & 22 & 24 \end{bmatrix}$$

$$k := 3$$

$$k \cdot B = \begin{bmatrix} 3 & 12 & 21 & 30 \\ 6 & 15 & 24 & 33 \\ 9 & 18 & 27 & 36 \end{bmatrix}$$

- wyznacznik macierzy,
- macierz odwrotna,
- odwołanie do kolumny i wiersza,
- łączenie macierzy,

$$A \cdot B = \begin{bmatrix} 14 & 32 & 50 & 68 \\ 32 & 77 & 122 & 167 \\ 50 & 122 & 194 & 266 \\ 68 & 167 & 266 & 365 \end{bmatrix} \qquad |A \cdot B| = 0$$

$$F := \begin{bmatrix} 1 & 2 & 3 \\ -2 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \qquad A := \begin{bmatrix} 3 & 5 & 7 \\ 4 & 6 & 8 \\ 5 & 7 & 9 \end{bmatrix}$$

$$|F| = -36 \qquad |A| = 0$$

$$\mathrm{col}(F; 2) = \begin{bmatrix} 2 \\ 5 \\ 8 \end{bmatrix} \qquad \mathrm{row}(A; 3) = \begin{bmatrix} 5 & 7 & 9 \end{bmatrix}$$

$$\mathrm{stack}(A; F) = \begin{bmatrix} 3 & 5 & 7 \\ 4 & 6 & 8 \\ 5 & 7 & 9 \\ 1 & 2 & 3 \\ -2 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

$$\mathrm{augment}(A; F) = \begin{bmatrix} 3 & 5 & 7 & 1 & 2 & 3 \\ 4 & 6 & 8 & -2 & 5 & 6 \\ 5 & 7 & 9 & 7 & 8 & 9 \end{bmatrix}$$

- sortowanie według kolumny,
- sortowanie według wiersza,
- podmacierz,
- element minimalny,
- element maksymalny,
- średnia,

$$AF := \text{augment}(A; F)$$

$$AF = \begin{bmatrix} 3 & 5 & 7 & 1 & 2 & 3 \\ 4 & 6 & 8 & -2 & 5 & 6 \\ 5 & 7 & 9 & 7 & 8 & 9 \end{bmatrix}$$

$$\text{csort}(AF; 4) = \begin{bmatrix} 4 & 6 & 8 & -2 & 5 & 6 \\ 3 & 5 & 7 & 1 & 2 & 3 \\ 5 & 7 & 9 & 7 & 8 & 9 \end{bmatrix}$$

$$\text{rsort}(AF; 2) = \begin{bmatrix} 1 & 3 & 2 & 5 & 3 & 7 \\ -2 & 4 & 5 & 6 & 6 & 8 \\ 7 & 5 & 8 & 7 & 9 & 9 \end{bmatrix}$$

$$\text{submatrix}(AF; 2; 3; 3; 6) = \begin{bmatrix} 8 & -2 & 5 & 6 \\ 9 & 7 & 8 & 9 \end{bmatrix}$$

$$\min(AF) = -2$$

$$\max(AF) = 9$$

- miejsca zerowe funkcji,
- miejsca zerowe wielomianów,

$$f(x) := \mathbf{e}^{x} - x^{3}$$

$$\mathrm{roots}(f(\_x); \_x; 2) = 1,8572$$

$$\mathrm{roots}(f(\_x); \_x; 4) = 4,5364$$

$$w(x) := -3 \cdot x^{3} - 2 \cdot x^{2} + 5 \cdot x + 1$$

$$a := \begin{bmatrix} 1 \\ 5 \\ -2 \\ -3 \end{bmatrix} \qquad \mathrm{polyroots}(a) = \begin{bmatrix} -0,1897 \\ 1,1084 \\ -1,5853 \end{bmatrix}$$

- rozwiązywanie układów równań liniowych,
- rozwiązywanie układów równań nieliniowych,

$$A := \begin{bmatrix} 0 & 1 & 2 \\ 3 & 0 & 2 \\ 5 & 3 & 1 \end{bmatrix} \qquad B := \begin{bmatrix} 13 \\ -3 \\ 50 \end{bmatrix}$$

$$|A| = 25 \qquad X := A^{-1} \cdot B$$

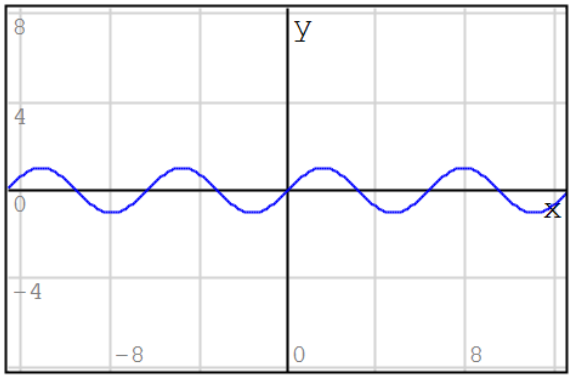
$$X = \begin{bmatrix} 0,28 \\ 16,84 \\ -1,92 \end{bmatrix} \qquad A \cdot X = \begin{bmatrix} 13 \\ -3 \\ 50 \end{bmatrix}$$

$$R(x; y; z) := \begin{bmatrix} x^3 + 3 \cdot x^2 + 9 \cdot x - 2 \cdot y = 4 \cdot z \\ e^x = y + 1 \\ z^2 = 3 - \cos(y) \end{bmatrix}$$

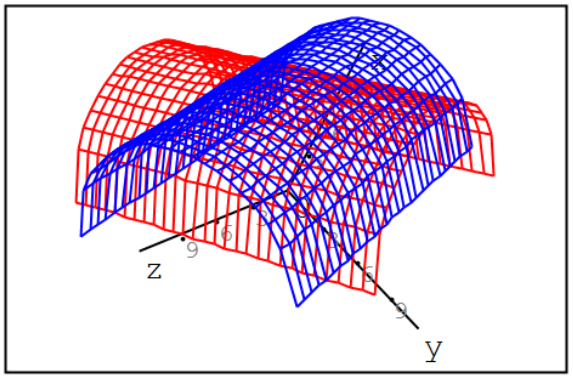
$$R_x := \text{roots}\left(R(x; y; z); \begin{bmatrix} x \\ y \\ z \end{bmatrix}; \begin{bmatrix} 1 \\ 1 \\ 2 \end{bmatrix}\right)$$

$$R_x = \begin{bmatrix} 0,7239 \\ 1,0624 \\ 1,5853 \end{bmatrix}$$

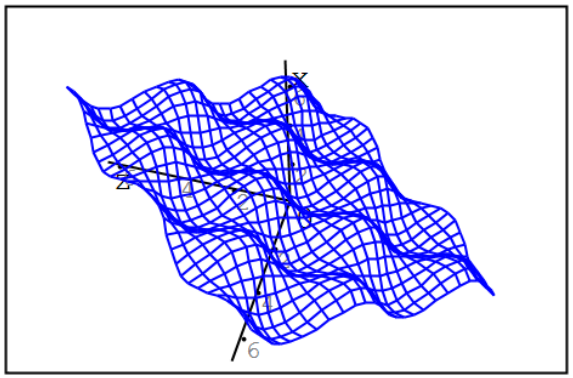
- wykresy 3D.



$\sin(x)$



$$\begin{cases} \sqrt{25 - x^2} \\ \sqrt{25 - y^2} \end{cases}$$

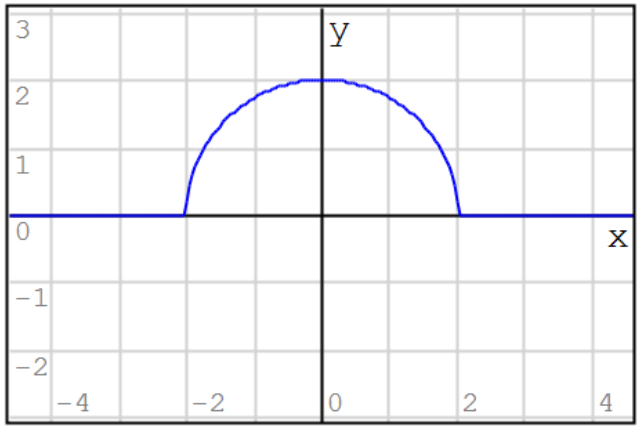


$\sin(x) \cdot \cos(y)$

Programowanie

- warunek **if**,

$$f(x) := \begin{cases} \text{if } |x| > 2 \\ 0 \\ \text{else} \\ \sqrt{4 - x^2} \end{cases}$$



$f(x)$

- pętla **for**,

$n := 5 \qquad s := 0$

for $i \in [1..n]$
  $s := s + i$

$s = 15$

$$\sum_{i=1}^{5} i = 15$$

$n := 9 \qquad s := 0$

for $i \in [1..n]$
  $s := s + i$

$s = 45$

$$\sum_{i=1}^{9} i = 45$$

- pętla **while**.

$x := 2 \qquad \varepsilon := 10^{-6}$

$r := \frac{x}{2} \qquad r_n := \frac{r}{2} + \frac{x}{2 \cdot r}$

while $|r_n - r| > \varepsilon$

$\begin{cases} r := r_n \\ r_n := \frac{r}{2} + \frac{x}{2 \cdot r} \end{cases}$

$r_n = 1,4142$

$\sqrt{2} = 1,4142$

$x := 5 \qquad \varepsilon := 10^{-6}$

$r := \frac{x}{2} \qquad r_n := \frac{r}{2} + \frac{x}{2 \cdot r}$

while $|r_n - r| > \varepsilon$

$\begin{cases} r := r_n \\ r_n := \frac{r}{2} + \frac{x}{2 \cdot r} \end{cases}$

$r_n = 2,2361$

$\sqrt{5} = 2,2361$

Po co mi programowanie?

Linus Torvalds

Computer programming is not for everyone. I think it's reasonably specialized, and nobody really expects most people to have to do it.

Steve Jobs

Everybody in this country should learn how to program a computer, because it teaches you how to think.

Celem nauki programowania jest wykształcenie umiejętności **myślenia komputacyjnego** (*computational thinking*), które obejmuje myślenie algorytmiczne w rozwiązywaniu problemów oraz umiejętność programowania rozszerzone na wszystkie obszary działalności ludzkiej. Takie podejście przede wszystkim pozwala zwiększyć efektywność i ułatwić pracę.

Nawet życiowe problemy i decyzje można potraktować jako problem algorytmiczny.

Języki programowania lub **makropolecenia** udostępniają szereg narzędzi pozwalających na przyśpieszenie i zwiększenie efektywności pracy, zrobienie czegoś w łatwiejszy sposób, a nawet utworzenie kompletnie nowych narzędzi.

Rozwój języków programowania znacznie obniżył próg umiejętności jakie należy posiadać, żeby zacząć naukę. Nie trzeba posiadać żadnych informacji na temat budowy komputera, nie trzeba mieć solidnych podstaw matematycznych (choć te są przydatne), nie trzeba mieć superkomputera ani kupować dodatkowego drogiego oprogramowania.

Przypomnienie podstawowych terminów

Program komputerowy (aplikacja) – sekwencja symboli (zrozumiałych dla komputera rozkazów) przeznaczonych do przetworzenia zgodnie z pewnymi regułami, zwanymi **językiem programowania**.

Program w postaci języka *zrozumiałego* dla człowieka nazywany jest **kodem źródłowym**, podczas gdy program wyrażony w postaci zrozumiałej dla maszyny (to jest za pomocą ciągu liczb, a bardziej precyzyjnie zer i jedynek) nazywany jest kodem maszynowym bądź postacią binarną (wykonywalną).

Program jest zazwyczaj wykonywany przez komputer, bezpośrednio – jeśli wyrażony jest w języku zrozumiałym dla danej maszyny lub pośrednio – gdy jest interpretowany przez inny program (interpreter).

Programy komputerowe można zaklasyfikować według ich zastosowań. Wyróżnia się zatem aplikacje użytkowe, systemy operacyjne, gry, kompilatory i inne. Programy wbudowane wewnątrz urządzeń określa się jako **firmware**.

W najprostszym modelu wykonanie programu (zapisanego w postaci zrozumiałej dla maszyny) polega na umieszczeniu go w pamięci operacyjnej komputera i wskazaniu procesorowi adresu pierwszej instrukcji.

Po tych czynnościach procesor będzie wykonywał kolejne instrukcje programu, aż do jego zakończenia.

Program komputerowy będący w trakcie wykonania nazywany jest **procesem** lub **zadaniem**.

Tworzenie programu komputerowego można podzielić na dwa etapy:

- Po zrodzeniu się **pomysłu** powinien powstać **algorytm**. Algorytm wymusza stosowanie podziału programu na funkcje. zmienne, obiekty, na których program będzie operował, jak również wprowadzenie procedur, które opisują wykonywane operacje.
- Algorytm należy zapisać w języku programowania, stosując dostępne struktury danych i funkcje – tworzenie **kodu źródłowego**. W trakcie tworzenia programu kod jest poddawany **debugowaniu** – wyszukiwanie błędów.

Językiem programowania nazywamy zestaw zasad tekstowego lub graficznego opisu algorytmu za pomocą przyjętych elementów języka.

Podobnie jak języki naturalne, język programowania składa się ze zbiorów reguł syntaktycznych oraz semantyki, które opisują, jak należy budować poprawne wyrażenia oraz jak komputer ma je rozumieć

Język programowania pozwala na precyzyjny zapis algorytmów oraz innych zadań, jakie komputer ma wykonać.

Podział języków programowania

Kod maszynowy (język maszynowy) – język programowania, w którym zapis programu wymaga **instrukcji** bezpośrednio jako liczb, które są rozkazami i danymi bezpośrednio pobieranymi przez procesor wykonujący ten program.

Jest dopasowany do konkretnego typu procesora i przeznaczony do bezpośredniego wykonania przez procesor. Analiza kodu maszynowego jest praktycznie niemożliwa przez człowieka.

Języki niskopoziomowe – przedstawiają one instrukcje udostępniane przez system komputerowy w postaci prostych oznaczeń (o ograniczonej liczbie, zakodowane w procesorze). Do języków niskopoziomowych należą **assemblery**.

Języki wysokopoziomowe – składnia i słowa kluczowe mają maksymalnie ułatwić rozumienie kodu programu dla człowieka, tym samym zwiększając poziom abstrakcji i dystansując się od budowy sprzętu komputerowego.

Kod napisany w języku wysokiego poziomu nie jest bezpośrednio *zrozumiały* dla komputera – większość kodu stanowią tak naprawdę normalne słowa (najczęściej w języku angielskim).

Języki wysokopoziomowe dzielimy na dwie grupy:

- interpretowane,
- kompilowane.

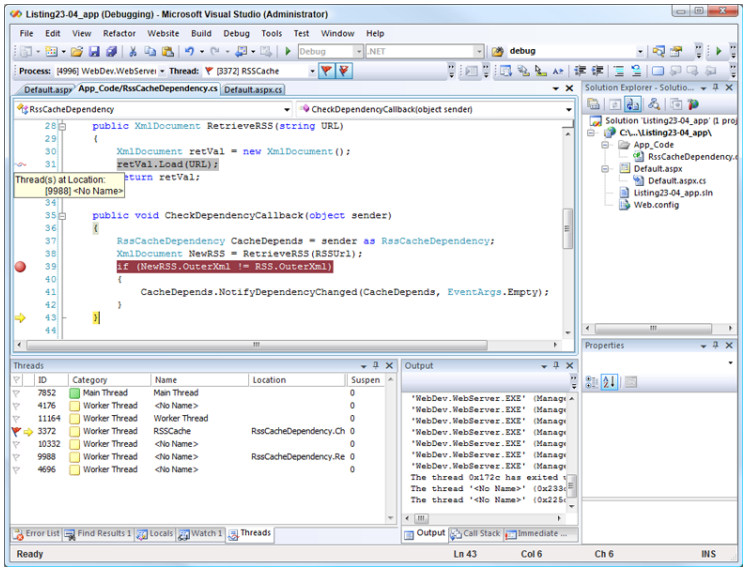
Języki interpretowane nie wymagają kompilacji tylko **interpretera**. Są przechowywane w postaci kodu źródłowego i dopiero podczas uruchomienia wczytywane, analizowane i wykonywane przez interpreter języka – **PHP**, **JavaScript**, **Python**, **PERL**. Programy przeznaczone do interpretacji często nazywane są **skryptami**.

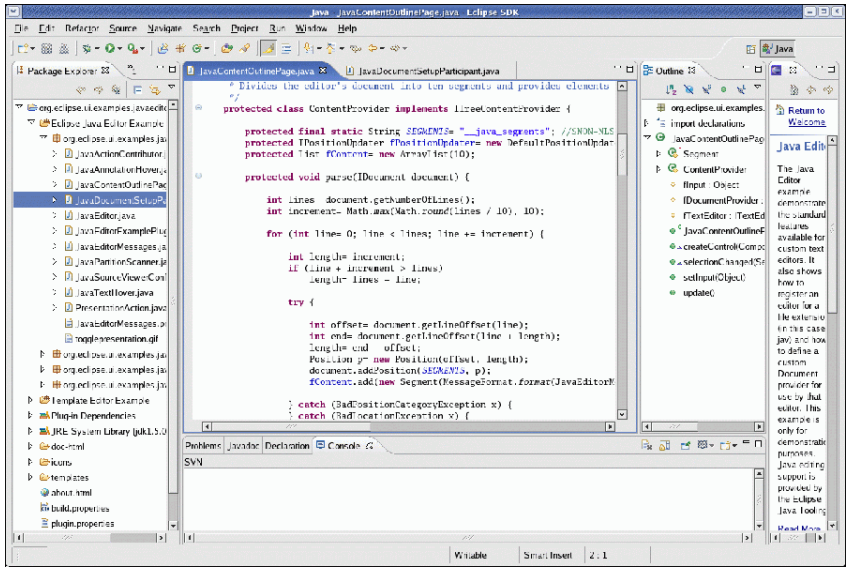
Języki kompilowane wymagają procesu kompilacji kodu źródłowego do postaci kodu maszynowego (postaci binarnej). Robi to specjalny program zwany **kompilatorem**, dzięki czemu możliwe staje się jego późniejsze uruchomienie. Języki kompilowane: **Pascal**, **C**, **C++**, **Fortran**, **Java**.

Do utworzenia programu w danym języku niezbędne są **edytor** tekstu, **debugger** i **kompilator**. Programy te mogą tworzyć integralne środowisko pracy, udostępniające kombinacje tych funkcji – mówimy wówczas o **środowisku programistycznym**.

Aby przyśpieszyć tworzenie aplikacji, szczególnie z interfejsem graficznym, tworzy się narzędzia **RAD** (*Rapid Application Development*), które umożliwiają tworzenie programów z gotowych komponentów (w sensie elementów interfejsu i funkcji z implementacją typowych algorytmów) na przykład:

- Microsoft VisualStudio,
- Eclipse IDE.





Syntaktyka i semantyka

Aby ciąg znaków mógł być rozpoznany jako program napisany w danym języku, musi spełniać reguły **syntaktyki** (składni). Składnia opisuje:

- rodzaje dostępnych symboli,
- zasady, według których symbole mogą być łączone w większe struktury.

Należy zauważyć, że na etapie przetwarzania składni w ogóle nie jest brane pod uwagę znaczenie poszczególnych symboli. W praktyce kod poprawny składniowo nie musi być poprawny semantycznie. Występuje tu analogia do języków naturalnych. Zdanie „Bźdźiągwy się muciosząl" jest poprawne pod względem gramatycznym, lecz nie posiada żadnego znaczenia, ponieważ zostały w nim użyte nieistniejące słowa.

Semantyka języka programowania definiuje precyzyjnie znaczenie poszczególnych symboli oraz ich funkcję w programie. Semantykę najczęściej definiuje się słownie, ponieważ większość z jej zagadnień jest trudna lub wręcz niemożliwa do ujęcia w jakikolwiek formalizm.

Część błędów semantycznych można wychwycić już w momencie wstępnego przetwarzania kodu programu, np. próbę odwołania się do nieistniejącej funkcji, lecz inne mogą ujawnić się dopiero w trakcie wykonywania.

Błędy

W trakcie pisania kodu nie da się uniknąć błędów. Błędy mogą wynikać z niepoprawnego wpisania instrukcji, pominięcia kropek, przecinków, nawiasów, itp. Takie błędy noszą nazwę **syntaktycznych** (składniowych).

Oprócz błędów syntaktycznych, można spotkać jeszcze błędy **semantyczne** (znaczeniowe, wykonania) i **logiczne**.

Błędy wykonania występują w chwili odtwarzania procedur (programu) i mogą wynikać z próby uruchomienia nieistniejącej procedury, otwarcia nieistniejącego pliku lub złego typowania zmiennych.

Błędy logiczne zwykle nie wywołują żadnych komunikatów błędu. Chociaż program jest poprawny pod względem syntaktycznym i semantycznym, nie powoduje żadnych problemów kompilacji i uruchamiania to sam rezultat działania może być błędny.

Błędy logiczne powodują otrzymanie wyników innych niż się spodziewano. Są to zwykle błędy bardzo trudne do odnalezienia.

Języki programowania

Dla początkujących, problemem jest mnogość dostępnych obecnie języków programowania. Najbardziej klasyczne języki programowania to **C** i **C++**, popularne są **Java** i **C#**, modny jest **Python**, **JavaScript**, **TypeScript**, **Go**.

Analitycy często używają jżyków pozwalających na szybkie pisanie aplikacji np. **Visual Basic** i eksploracji baz danych: **SQL**, czy dedykowane do analizy danych **R** lub do obliczeń numerycznych (analizy danych też) **MATLAB**, **Scilab**.

- https://www.tiobe.com/tiobe-index/ – Wskaźnik popularności języków programowania.
- https://insights.stackoverflow.com/survey/ – Ankiety serwisu społecznościowego programistów.

Paradygmat programowania

Paradygmat programowania – wzorzec programowania komputerów, który definiuje sposób patrzenia programisty na przepływ sterowania i wykonywanie programu komputerowego.

Przykładowo, w programowaniu **obiektowym** jest on traktowany jako zbiór współpracujących ze sobą obiektów, podczas gdy w programowaniu **funkcyjnym** definiujemy, co trzeba wykonać, a nie w jaki sposób.

Różne języki programowania mogą wspierać różne paradygmaty programowania. Przykładowo, **Smalltalk** i **Java** są ściśle zaprojektowane dla potrzeb programowania obiektowego, natomiast **Haskell** jest językiem funkcyjnym. Istnieją także języki wspierające kilka paradygmatów, np. **Python**.

Wiele paradygmatów jest dobrze znanych z tego, jakie praktyki są w nich zakazane, a jakie dozwolone.

Na przykład, ścisłe programowanie funkcyjne nie pozwala na tworzenie skutków ubocznych (dowolny efekt wyrażenia, lub wywołania funkcji, który wykracza poza zwrócenie wartości). W programowaniu strukturalnym nie korzysta się z instrukcji skoku.

Zależności między paradygmatami programowania mogą przybierać skomplikowane formy, ponieważ jeden język może wspierać wiele różnych paradygmatów. Na przykład, **C++** posiada elementy programowania proceduralnego, obiektowego oraz uogólnionego, co stanowi o nim, że jest hybrydowym językiem. To projektanci i programiści decydują, jak zbudować z nich w pełni działający program.

Przykłady paradygmatów programowania:

- programowanie **imperatywne**,
- programowanie **deklaratywne**,
- programowanie **proceduralne**,
- programowanie **strukturalne**,
- programowanie funkcyjne,
- programowanie **obiektowe**,
- programowanie uogólnione,
- programowanie **sterowane zdarzeniami**,
- programowanie logiczne,
- programowanie aspektowe,
- programowanie agentowe,
- programowanie modularne.

Programowanie imperatywne – paradygmat programowania, który opisuje proces wykonywania jako sekwencję instrukcji zmieniających stan programu, podobnie jak tryb rozkazujący, wyraża żądania jakichś czynności do wykonania.

Programy imperatywne składają się z ciągu komend do wykonania przez komputer. Rozszerzeniem (w sensie wbudowanych funkcji) i rodzajem (w sensie paradygmatu) programowania imperatywnego jest programowanie proceduralne.

Programowanie deklaratywne – rodzina paradygmatów programowania, które nie są z natury imperatywne. W przeciwieństwie do programów napisanych imperatywnie, programista opisuje warunki, jakie musi spełniać końcowe rozwiązanie (co chcemy osiągnąć), a nie szczegółową sekwencję kroków, które do niego prowadzą (jak to zrobić).

Przykłady języków: **XSLT**, **Prolog**.

Programowanie proceduralne to paradygmat programowania zalecający dzielenie kodu na procedury, czyli fragmenty wykonujące ściśle określone operacje.

Procedury nie powinny korzystać ze zmiennych globalnych (w miarę możliwości), lecz pobierać i przekazywać wszystkie dane (czy też wskaźniki do nich) jako parametry wywołania.

Programowanie strukturalne to paradygmat programowania zalecający hierarchiczne dzielenie kodu na bloki, z jednym punktem wejścia i jednym lub wieloma punktami wyjścia.

Chodzi przede wszystkim o nieużywanie instrukcji skoku. Dobrymi strukturami są np. instrukcje: warunkowe, pętle, wyboru.

Strukturalność zakłócają instrukcje typu: `break`, `continue`, `switch`, które jednak w niektórych przypadkach znacząco podnoszą czytelność kodu.

Praktycznie w każdym języku można programować strukturalnie, jednakże w niektórych jest to styl naturalny (np. **Pascal**).

Programowanie obiektowe (*Object-Oriented Programming*) – paradygmat programowania, w którym programy definiuje się za pomocą obiektów – elementów łączących stan (czyli dane, nazywane polami lub właściwościami) i zachowanie (czyli procedury, tu: metody). Obiektowy program komputerowy wyrażony jest jako zbiór takich obiektów, komunikujących się pomiędzy sobą w celu wykonywania zadań.

Podejście to różni się od tradycyjnego programowania proceduralnego, gdzie dane i procedury nie są ze sobą bezpośrednio związane. Programowanie obiektowe ma ułatwić pisanie, konserwację i wielokrotne użycie programów lub ich fragmentów.

Największym atutem programowania, projektowania oraz analizy obiektowej jest zgodność takiego podejścia z rzeczywistością – mózg ludzki jest w naturalny sposób najlepiej przystosowany do takiego podejścia przy przetwarzaniu informacji. Przykłady języków: **C++**, **JAVA**.

Programowanie sterowane zdarzeniami – metodologia tworzenia programów komputerowych, która określa sposób ich pisania z punktu widzenia procesu przekazywania sterowania między poszczególnymi modułami tej samej aplikacji.

Programowanie sterowane zdarzeniami jest mocno powiązane ze środowiskami wieloprocesowymi, z graficznymi środowiskami systemów operacyjnych oraz z programowaniem obiektowym.

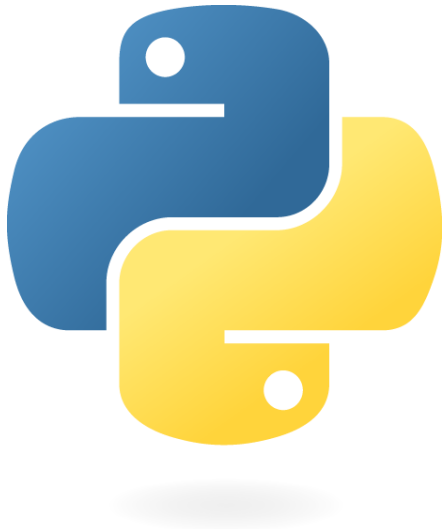
Paradygmat zakłada, że program jest cały czas bombardowany zdarzeniami, na które musi odpowiedzieć, i że przepływ sterowania w programie jest całkowicie niemożliwy do przewidzenia z góry.

Programowanie zdarzeniowe jest dominującym typem programowania związanego z graficznym interfejsem użytkownika (*Graphical User Interface*) – zdarzenia to naciśnięcia myszy, klawiszy, żądania odświeżenia przez system okienkowy, różne zdarzenia sieciowe i inne.

W programowaniu zdarzeniowym ważne jest żeby nie obsługiwać zbyt długo danego zdarzenia, bo blokuje się w ten sposób obsługę innych. W przypadku serwerów obniżyło by to znacznie wydajność, w przypadku GUI program zbyt wolno odpowiadałby na akcje użytkownika.

Można to osiągnąć za pomocą asynchronicznego I/O, wielowątkowości, rozbijania zdarzenia na podzdarzenia i wielu innych mechanizmów.

Python



Twórcą Pythona jest holender Guido van Rossum a sama nazwa pochodzi od popularnego serialu BBC **Latający Cyrk Monty Pythona**. Prace nad pierwszym interpreterem Pythona rozpoczęły się w 1989 roku jako następca języka ABC.

Od wersji 2.1 Python jest udostępniany jako projekt Open Source przez niedochodową organizację **Python Software Foundation** (PSF).

Guido van Rossum

Computer programming for everybody:

- an easy and intuitive language just as powerful as major competitors,
- open source, so anyone can contribute to its development,
- code that is as understandable as plain English,
- suitability for everyday tasks, allowing for short development times.

Rozwój języka jest prowadzony przy wykorzystaniu **PEP** (*Python Enhancement Proposal*). Dokumenty te to propozycje rozszerzeń lub zmian w języku w postaci artykułu, który jest poddawany pod dyskusję wśród programistów Pythona.

Każdy dokument zawiera opis proponowanego rozwiązania, uzasadnienie oraz aktualny status. Po osiągnięciu konsensusu propozycje są przyjmowane lub odrzucane.

Cechy Pythona:

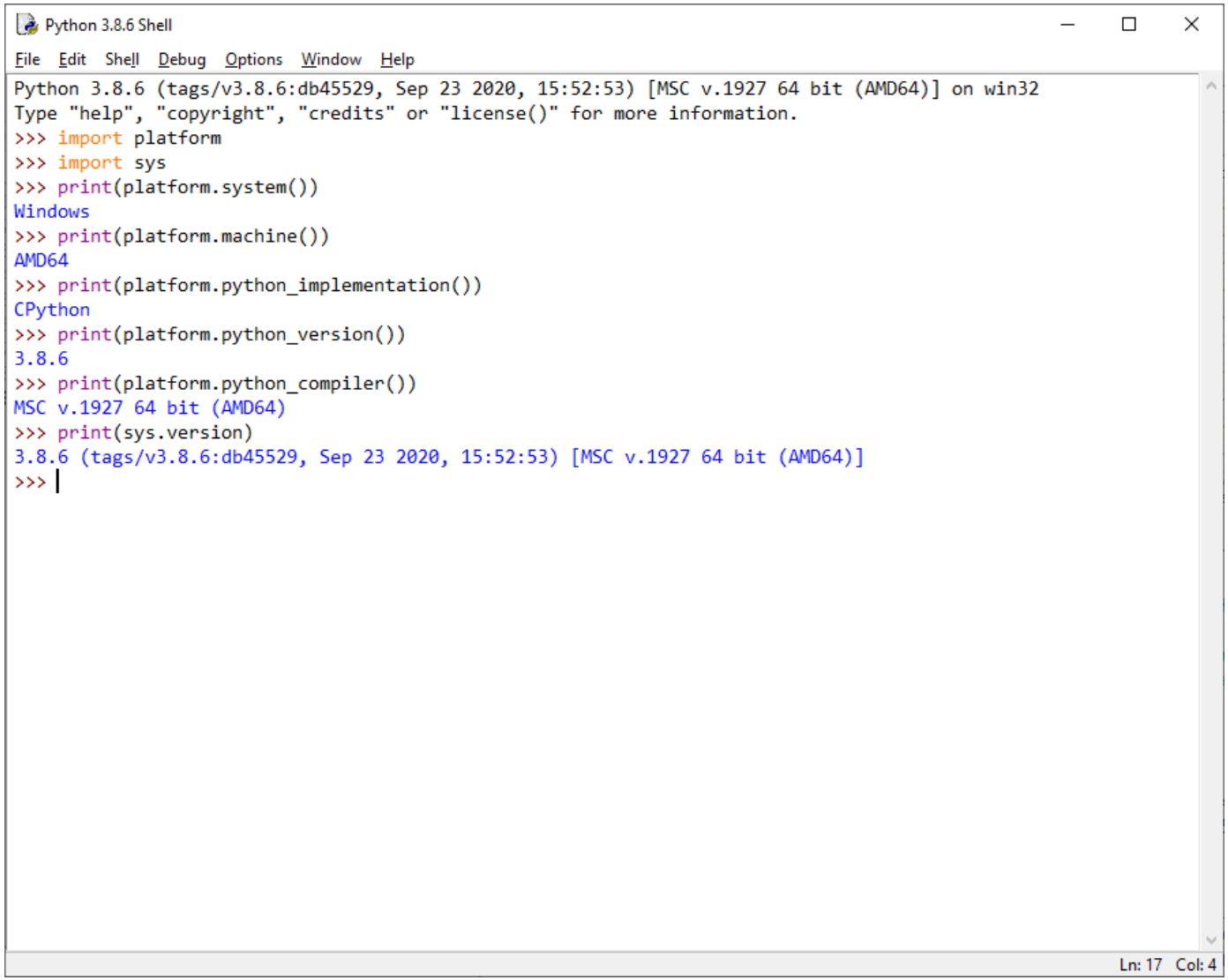
- jest językiem ogólnego przeznaczenia,
- jest w pełni obiektowy,
- umożliwia programowanie w różnych paragdygmatach (strukturalne, obiektowe, funkcyjne),
- jest dostępny dla różnych systemów operacyjnych,
- jest językiem o wzorcowej dokumentacji, dostępnej w sieci oraz lokalnie (po instalacji),
- jest językiem o dynamicznym typowaniu,
- jest językiem interpretowanym,
- chętnie jest stosowany do szybkiego prototypowania, automatyzacji i integracja komponentów, analizy danych.

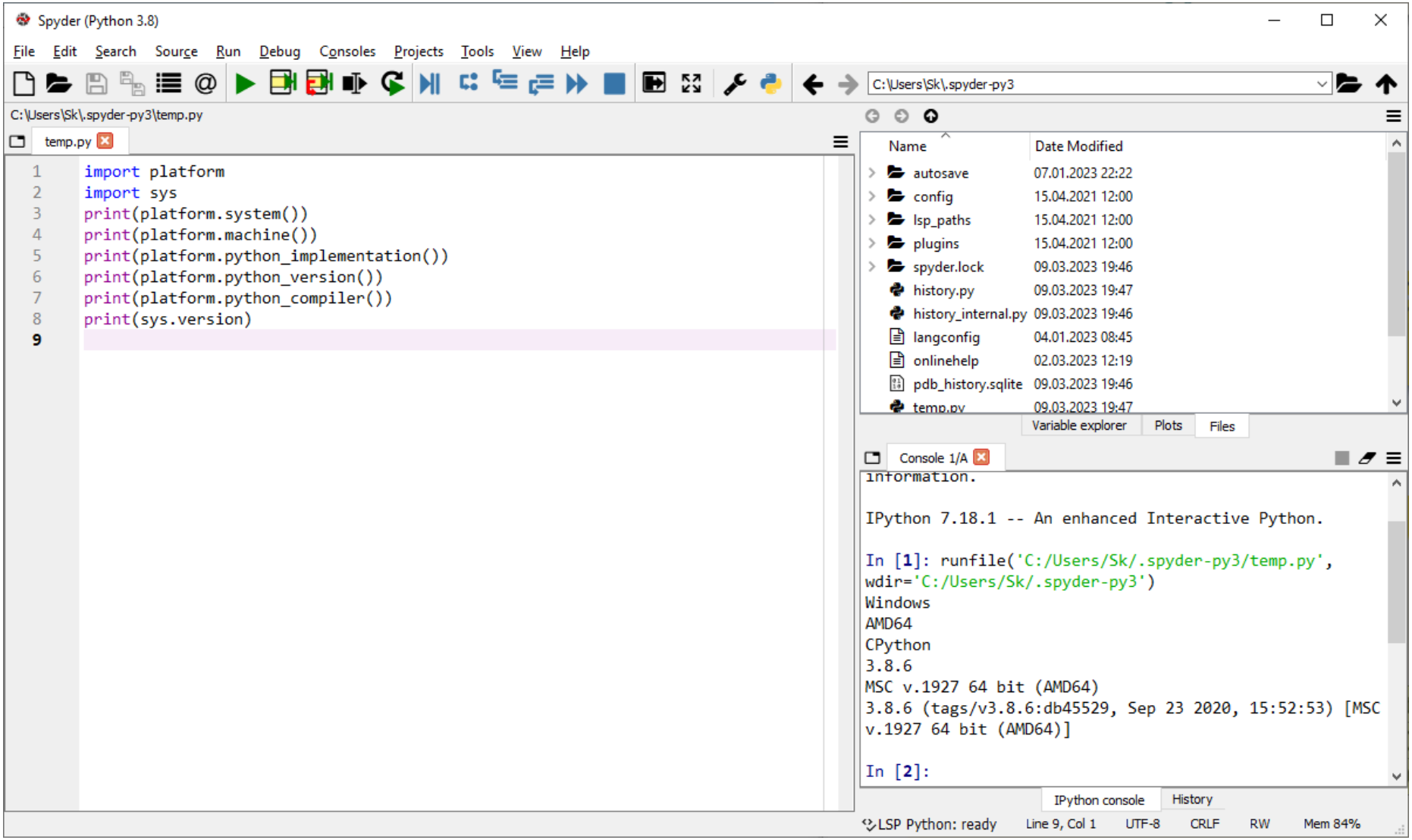
Implementacje kompilatora Pythona:

- **CPython** – implementacja referencyjna, kompilator napisany w języku C, najpopularniejszy, dystrybuowany z różnymi pakietami aplikacji i dołączany do różnych wersji SO,
- **PyPy** – kompilator napisany w Pythonie (RPythonie), skoncentrowany na wydajności, z kompilatorem JIT (*just in time*),
- **Jython** – kompilator skierowany na integrację z językiem programowania Java do, kod kompilowany do kodu bajtowego (*bytecode*) Javy, dla JVM,
- **IronPython** – kompilator zaprojektowany z myślą o umożliwieniu integracji programów napisanych w Pythonie z aplikacjami stworzonymi dla platformy .NET, kompilator do kodu bajtowego MSIL, dla .NET,
- **Numba** – kompilator do kodu maszynowego, wykorzystuje LLVM,
- **Cython** – kompilator/konwerter do C/C++.

Dwa sposoby wywołania interpretera:

- linia poleceń (konsola IDLE) – *read-eval-print loop* jak bash lub `PowerShell`,
- uruchomienie skryptu – polecenie `python foo.py`.




Spyder (Python 3.8)
File Edit Search Source Run Debug Consoles Projects Tools View Help
C:\Users\Sk\.spyder-py3
C:\Users\Sk\.spyder-py3\temp.py
temp.py
1 import platform
2 import sys
3 print(platform.system())
4 print(platform.machine())
5 print(platform.python_implementation())
6 print(platform.python_version())
7 print(platform.python_compiler())
8 print(sys.version)
9
Name
Date Modified
autosave 07.01.2023 22:22
config 15.04.2021 12:00
lsp_paths 15.04.2021 12:00
plugins 15.04.2021 12:00
spyder.lock 09.03.2023 19:46
history.py 09.03.2023 19:47
history_internal.py 09.03.2023 19:46
langconfig 04.01.2023 08:45
onlinehelp 02.03.2023 12:19
pdb_history.sqlite 09.03.2023 19:46
temp.py 09.03.2023 19:47
Variable explorer
Plots
Files
Console 1/A
information.
IPython 7.18.1 -- An enhanced Interactive Python.
In [1]: runfile('C:/Users/Sk/.spyder-py3/temp.py',
wdir='C:/Users/Sk/.spyder-py3')
Windows
AMD64
CPython
3.8.6
MSC v.1927 64 bit (AMD64)
3.8.6 (tags/v3.8.6:db45529, Sep 23 2020, 15:52:53) [MSC
v.1927 64 bit (AMD64)]
In [2]:
IPython console
History
LSP Python: ready
Line 9, Col 1
UTF-8
CRLF
RW
Mem 84%

Struktura programu

Python nie jest językiem o swobodnej składni – wcięcia w kodzie są wykorzystywane do tworzenia bloków (instrukcji złożonych).

Instrukcje jednakowo wcięte stanowią blok, nowy blok może pojawić się tylko w niektórych miejscach (zazwyczaj po dwukropku).

```
if True:
  foo = 10    # Spaces
  print(foo) # Tab
```

Linia programu

Linia programu składa się z linii logicznych, a ta z kolei z linii fizycznych.

Linia fizyczna to sekwencja znaków zakończona kodem końca linii. W plikach źródłowych i ciągach znaków można zastosować dowolną ze standardowych sekwencji zakończenia linii platformy – ASCII **LF** (Unix/Linix), ASCII **CR LF** (Windows) lub ASCII **CR**.

Zalecana długość linii fizycznej: 79 znaków.

Zalecane wcięcie: 4 spacje (niestety zwykle stosuję dwie).

Kodowanie pliku źródłowego

Kod w podstawowej dystrybucji Pythona powinien zawsze używać **UTF-8** (lub **ASCII** w Pythonie 2).

Pliki używające ASCII lub UTF-8 nie powinny mieć deklaracji kodowania.

Wcięcia

```
# Aligned with opening delimiter.
foo = long_function_name(var_one, var_two,
                         var_three, var_four)
```

```
# More indentation included to distinguish this from the rest.
def long_function_name(
        var_one, var_two, var_three,
        var_four):
    print(var_one)
```

```
# Hanging indents should add a level.
foo = long_function_name(
    var_one, var_two,
    var_three, var_four)
```

```
my_list = [
    1, 2, 3,
    4, 5, 6,
    ]
```

```
result = some_function_that_takes_arguments(
    'a', 'b', 'c',
    'd', 'e', 'f',
    )
```

```
my_list = [
    1, 2, 3,
    4, 5, 6,
]
```

```
result = some_function_that_takes_arguments(
    'a', 'b', 'c',
    'd', 'e', 'f',
)
```

Dzielenie linii z operatorami

```
# Easy to match operators with operands
income = (gross_wages
          + taxable_interest
          + (dividends - qualified_dividends)
          - ira_deduction
          - student_loan_interest)
```

Puste linie

```
# Surround top-level function definitions with two blank lines.
def function_foo():
  return "foo has been done"


def function_bar():
  return "bar has been done"
```

Import

```
# Imports should usually be on separate lines.
import os
import sys
```

```
import mypkg.sibling
from mypkg import sibling
from mypkg.sibling import example
```

Białe znaki

```
# Yes:
dct(foo[1], {bar: 2})

# No:
dct( foo[ 1 ], { bar: 2 } )
```

```
# Yes:
foo = (0,)

# No:
bar = (0, )
```

```
# Yes:
if x == 4: print x, y; x, y = y, x

# No:
if x == 4 : print x , y ; x , y = y , x
```

```
# Yes:
foo(1)

# No:
foo (1)
```

```
# Yes:
dct['key'] = lst[index]

# No:
dct ['key'] = lst [index]
```

```
# Yes:
foo[1:9], foo[1:9:3], foo[:9:3], foo[1::3], foo[1:9:]
foo[lower:upper], foo[lower:upper:], foo[lower::step]
foo[lower+offset : upper+offset]
foo[: upper_fn(x) : step_fn(x)], foo[:: step_fn(x)]
foo[lower + offset : upper + offset]

# No:
foo[lower + offset:upper + offset]
foo[1: 9], foo[1 :9], foo[1:9 :3]
foo[lower : : upper]
foo[ : upper]
```

Konwencje nazewnictwa

- nazwy pakietów i modułów: `os`, `math`,
- nazwy klas: `decimal.Decimal`, `statistics.NormalDist`,
  - nazwy funkcji i zmiennych: `time.process_time`, `itertools.zip_longest`,
  - **niekonsekwentnie** np.: `re.findall`, `random.getrandbits`,
  - z jednym podkreśleniem na początku to zmienna niepubliczna np.: `_foo` (tylko konwencja),
  - z dwoma podkreśleniami na początku *name-mangling* np.: `__bar()` dostępna jako `_Foo__bar()`,
  - z dwoma podkreśleniami na początku do specjalnego użycia np.: `__init__`, `__str__`,
- nazwy stałych: `threading.TIMEOUT_MAX` (tylko konwencja),
  - zalecane definiowanie na poziomie modułu,
  - **niekonsekwentnie** np.: `math.pi`, `string.ascii_lowercase`.

Style nazewnictwa

Powszechnie wyróżnia się następujące style nazewnictwa:

- `b` (pojedyncza mała litera),
- `B` (pojedyncza duża litera),
- `lowercase`,
- `lower_case_with_underscores`,
- `UPPERCASE`,
- `UPPER_CASE_WITH_UNDERSCORES`,
- `CapitalizedWords`,
- `mixedCaseWords`,
- `Capitalized_Words_With_Underscores`.

Używając skrótów w **CapWords**, wszystkie litery skrótu należy pisać dużą literą. Lepiej `HTTPServerError` niż `HttpServerError`.

Z czego jest zbudowany program?

W trakcie procesu interpretowania lub kompilacji kod źródłowy jest dzielony na **jednostki leksykalne**. Jednostki leksykalne w Pythonie często są nazywane **tokenami**.

Rozróżnia się następujące klasy jednostek leksykalnych:

- identyfikatory (*identifiers*),
- słowa kluczowe (*keywords*),
- literały (*literals*),
- operatory (*operators*),
- separatory, ograniczniki (*punctuators*, *delimiters*).

Identyfikatory

Identyfikatory to ciągi liter, cyfr i znaków podkreślenia, muszą zaczynać się od litery lub znaku podkreślenia. Wielkie i małe litery są rozróżniane. Długość identyfikatorów jest nieograniczona.

Polskie znaki mogą, ale nie powinny być używane.

Identyfikatory są arbitralnie wybranymi nazwami dla **zmiennych**, **stałych**, **funkcji**, **typów danych** definiowanych przez programistę. Identyfikatory nie mogą być **słowami kluczowymi**.

```
maxvalue     mindelta     _5Pi
max_value    min_delta    _5_Pi
MaxValue     MinDelta     JW23
maxValue     minDelta     JW_23
```

Słowa kluczowe

Słowa kluczowe to identyfikatory zastrzeżone i nie mogą być inaczej stosowane niż określa to standard języka.

```
import keyword
print(keyword.kwlist)
```

Słowa kluczowe powinny być pisane tak jak je podano:

```
False      None       True       and        as         assert     async
await      break      class      continue   def        del        elif
else       except     finally    for        from       global     if
import     in         is         lambda     nonlocal   not        or
pass       raise      return     try        while      with       yield
```

Separatory

Cudzysłów (*double quote*) `"  "` – wykorzystywane do tworzenia napisów (łańcuchów znaków) jednowierszowych lub wielowierszowych `"""  """`, komentarze wielowierszowe.

Apostrofy (*single quote*) `'  '` – wykorzystywane do tworzenia napisów jednowierszowych lub wielowierszowych `'''  '''`.

Znak #, płotek, kratka (*hash*, *number sign*) `#` – komentarze.

Ukośnik wsteczny (*backslash*) `\` – używany do dzielenia długich linii, preferowanym sposobem jest użycie domniemanej kontynuacji wiersza w nawiasach i nawiasach klamrowych.

Znak at, tylko nie małpa (*at sign*) `@` – dekorator służący do modyfikowania definicji funkcji, metody lub klasy.

Nawiasy kwadratowe (*brackets*) `[ ]` – wykorzystywane są do deklarowania i odwoływania się do list.

Nawiasy okrągłe (*parentheses*) ( ) wykorzystywane są do grupowania wyrażeń, izolowania wyrażeń warunkowych, wskazują wywołanie funkcji i jej parametry, deklarowanie i odwoływanie się do krotek.

Nawiasy klamrowe (*braces*) { } – wykorzystywane są do deklarowania i odwoływania się do słowników.

Przecinek (*comma*) , rozdziela zwykle elementy na liście parametrów funkcji, elementy w listach i krotkach.

Średnik (*semicolon*) ; – oddziela dwie instrukcje.

Dwukropek (*colon*) : – kończy nagłówek instrukcji złożonej.

Przypisanie (*equal sign*) = – przypisanie, stosowany pomiędzy nazwą a wartością przypisania.

Kropka (*dot*) . – daje dostęp do atrybutu obiektu.

Komentarze

Komentarze to fragmenty tekstu spełniające funkcje dowolnych objaśnień robionych przez programistów dla programistów.

Komentarze nie mogą występować w napisach i stałych znakowych. Komentarze są usuwane z tekstu źródłowego programu.

```
# To jest komentarz jednoliniowy

"""
Ten komentarz obejmuje
kilka linii
kodu
"""
```

Zmienne

Zmienna to konstrukcja programistyczna posiadająca trzy podstawowe atrybuty:

- symboliczną nazwę,
- miejsce przechowywania,
- wartość.

Zmienna pozwala w kodzie źródłowym na odwoływanie się przy pomocy nazwy do wartości lub miejsca przechowywania. Nazwa służy do identyfikowania zmiennej w związku z tym często nazywana jest **identyfikatorem**.

Miejsce przechowywania przeważnie znajduje się w pamięci komputera i określane jest przez adres i długość danych. Wartość to zawartość miejsca przechowywania.

Zmienna zazwyczaj posiada również czwarty atrybut: **typ**, określający rodzaj danych przechowywanych w zmiennej i co za tym idzie sposób reprezentacji wartości w miejscu przechowywania.

W programie wartość zmiennej może być odczytywana lub zastępowana nową wartością, tak więc wartość zmiennej może zmieniać się w trakcie wykonywania programu, natomiast dwa pierwsze atrybuty (nazwa i miejsce przechowywania) nie zmieniają się w trakcie istnienia zmiennej.

W językach ze **statycznym typowaniem** zmienna ma określony typ danych jakie może przechowywać. Jest on wykorzystywany do określenia reprezentacji wartości w pamięci, kontrolowania poprawności operacji wykonywanych na zmiennej (kontrola typów) oraz konwersji danych jednego typu na inny.

W językach z **typowaniem dynamicznym** typ nie jest atrybutem zmiennej lecz wartości w niej przechowywanej. Zmienna może wtedy w różnych momentach pracy programu przechowywać dane innego typu.

Deklaracja zmiennej to stwierdzenie, że dany identyfikator jest zmienną, przeważnie też określa typ zmiennej. W zależności od języka programowania deklaracja może być obligatoryjna, opcjonalna lub nie występować wcale.

Definicja oprócz tego, że deklaruje zmienną to przydziela jej pamięć. Podczas definiowania lub deklarowania zmiennej można określić jej dodatkowe atrybuty wpływające na sposób i miejsce alokacji, czas życia, zasięg i inne.

Zasięg zmiennej określa gdzie w treści programu zmienna może być wykorzystana, natomiast **czas życia** zmiennej to okresy w trakcie wykonywania programu gdy zmienna ma przydzieloną pamięć i posiada (niekoniecznie określoną) wartość.

Ze względu na zasięg można wyróżnić podstawowe typy zmiennych:

- **globalne** – obejmujące zasięgiem cały program,
- **lokalne** – o zasięgu obejmującym pewien blok, podprogram.

Podobnie ze zmiennymi w klasie mogą być dostępne:

- tylko dla danej klasy (zmienna **prywatna**),
- dla danej klasy i jej potomków (**zmienna chroniona**),
- w całym programie (**zmienna publiczna**).

Zmienne mogą zmieniać swój pierwotny zasięg na przykład poprzez importowanie/włącznie do zasięgu globalnego modułów, pakietów czy przestrzeni nazw.

Ze względu na czas życia i sposób alokacji zmienna może być:

- statyczna,
- automatyczna,
- dynamiczna.

Dla zmiennej **statycznej** pamięć jest rezerwowana w momencie kompilacji lub ładowania programu. Takimi zmiennymi są zmienne globalne, zmienne klasy (współdzielone przez wszystkie obiekty klasy, a nawet dostępne spoza klasy), statyczne zmienne lokalne funkcji (współdzielone pomiędzy poszczególnymi wywołaniami funkcji i zachowujące wartość po zakończeniu).

Zmiennej **automatycznej** pamięć jest automatycznie przydzielana w trakcie działania programu. Są to przeważnie zmienne lokalne podprogramów i ich parametry formalne, znikają po zakończeniu podprogramu.

Pamięć dla zmiennej **dynamicznej** alokowana jest ręcznie w trakcie wykonywania programu przy pomocy specjalnych konstrukcji lub funkcji. W zależności od języka zwalnianie pamięci może być ręczne lub automatyczne, Zazwyczaj nie posiada własnej nazwy, lecz odwoływać się do niej trzeba przy pomocy wskaźnika, referencji lub zmiennej o semantyce referencyjnej.

Typy zmiennych

Python jest językiem z dynamicznym typowaniem – przypisywanie typów do zmiennych odbywa się w trakcie wykonywania programu.

Przy samym uruchomieniu skryptu interpreter nie wie jeszcze jakiego typu będą zmienne. Określa to w momencie przypisania pierwszej wartości.

Typy w Pythonie są bardziej ogólne i mają większe możliwości, niż bywa to w innych językach. Na przykład: listy udostępniają uporządkowane zbiory różnych obiektów, słowniki przechowują obiekty według określonego klucza, obecność typu zespolonego. Zarówno listy, jak i słowniki mogą być zagnieżdżone, mogą zmieniać rozmiar. Typy są obiektowe.

Wybrane wbudowane typy zmiennych:

- logiczne `bool`,
- liczbowe:
  - całkowitoliczbowe `int`,
  - zmiennopozycyjne `float`,
  - zespolone `complex`,
- kolekcje (sekwencje):
  - listy `list`,
  - krotki `tuple`,
  - słowniki `dict`,
  - zbiory `set`,
- napisy lub łańcuchy znaków (sekwencja tekstowa) `str`.

Typ zmiennej jest ustalany na etapie wykonywania kodu i jest związany z literałem do niej przypisywanym lub z typami zmiennych biorących udział w instrukcji.

Na przykład, kiedy uruchomimy kod zawierający ciąg znaków ujęty w cudzysłów: `a = "foo"`, wykonujemy wyrażenie z literałem, które generuje i zwraca nowy obiekt typu `str`.

typ, literał	
liczby	`123`; `3.14`; `3+4j`; `0b111`; `Decinal()`; `Fraction()`
napisy	`'foo'`; `"bar"`; `b'a\x01c'`; `u'sp\xc4m'`
listy	`[1, [2, 'trzy'], 4.5]`; `list(range(10))`
słowniki	`{'fruit': 'apple', 'color': 'red'}`; `dict(hours=10)`
krotki	`(1,'foo', 4, 'U')`; `tuple('bar')`; `namedtuple`
pliki	`open('foo.txt')`; `open(r'C:\bar.bin', 'wb')`
zbiory	`set('abc')`; `{'a', 'b', 'c'}`

Sprawdzanie i porównywanie typów

```
a = c = 5
b = 5.0

print(type(a), type(b), a is b, a == b)
# <class 'int'> <class 'float'> False True

print(type(a), type(c), a is c, a == c)
# <class 'int'> <class 'int'> True True
```

```
print(type(a) is int) # niezalecane
print(isinstance(a, int))) # lepsze
```

Liczby

W Pythonie liczby nie są jednym typem obiektu, są raczej kategorią podobnych typów. Python obsługuje proste typy liczbowe (liczby całkowite oraz zmiennoprzecinkowe), a także literały służące do tworzenia liczb.

Dodatkowo Python udostępnia bardziej zaawansowaną obsługę programowania numerycznego, a także obiekty przeznaczone do bardziej zaawansowanych działań.

Możliwości języka obejmują:

- obiekty całkowite i zmiennoprzecinkowe,
- obiekty zespolone,
- obiekty dziesiętne o ustalonej precyzji,
- ułamkowe obiekty wymierne,
- zbiory: kolekcje z operacjami numerycznymi,
- wartości logiczne: `true` i `false`,
- wbudowane funkcje i moduły: `round`, `math`, `random` itp.,
- nieograniczoną precyzję liczb całkowitych,
- operacje bitowe,
- formaty szesnastkowe, ósemkowe i binarne,
- rozszerzenia tworzone przez niezależnych programistów: wektory, biblioteki, wizualizacje, tworzenie wykresów itp.

Typ logiczny

Wartości logiczne są przechowywane przez zmienne typu `bool`. Typ `bool` jest podklasą typu `int`.

Mogą one przyjmować wyłącznie wartości `True` lub `False`. Wartości logiczne są słowami kluczowymi.

Wartości logiczne prawie dokładnie odpowiadają wartościom `0` i `1`. Różnicę można zaobserwować w instrukcji `print(True)`.

Nie należy stosować porównań typu `foo == True`, ale samo `foo` lub `foo is True` (zmienna musi być typu `bool`).

Liczby całkowite

Liczby całkowite są przechowywane przez zmienne typu `int`. Typ `int`.

Rozmiar typu `int`

```
import sys
print(sys.getsizeof(int())) # 24
```

Od wersji Pythona 3.x liczby całkowite zwykłe oraz długie zostały połączone. Istnieje tylko jeden typ liczby całkowitej, który automatycznie obsługuje nieograniczoną precyzję.

Liczby całkowite mogą być w Pythonie zapisane jako **dziesiętne** (o podstawie dziesięć), **szesnastkowe** (o podstawie szesnaście), **ósemkowe** (o podstawie osiem) oraz **dwójkowe** (o podstawie dwa).

Literały szesnastkowe zaczynają się od znaków `0x` lub `0X`, po których następuje ciąg cyfr szesnastkowych od `0` do `9` i od `A` do `F`. W literałach szesnastkowych cyfry szesnastkowe mogą być zapisane małą i wielką literą.

Literały ósemkowe rozpoczynają się od znaków `0o` lub `0O` (cyfry zero i małej lub wielkiej litery **o**), po których następuje ciąg cyfr od `0` do `7`.

Literały dwójkowe rozpoczynają się od znaków `0b` lub `0B`, po których następują cyfry dwójkowe (`0` lub `1`).

Warto zauważyć, że wszystkie powyższe literały tworzą w kodzie programu obiekty liczb całkowitych, które są tylko alternatywnymi sposobami zapisu określonych wartości.

Wywołania wbudowanych funkcji `hex()`, `oct()` oraz `bin()` przekształcają liczbę całkowitą na jej **reprezentację** o podanej podstawie, natomiast wywołanie `int()` przekształca wykonywany łańcuch na liczbę całkowitą zgodnie z podaną podstawą.

Liczby rzeczywiste i zespolone

Liczby zmiennoprzecinkowe mają znak dziesiętny (w postaci kropki) lub zawierają opcjonalny wykładnik ze znakiem wprowadzony po literze `e` lub `E`, po której znajduje się opcjonalny znak.

Literał zapisany ze znakiem dziesiętnym lub wykładnikiem zostanie automatycznie zapisany jako obiekt typu `float`. Użycie liczby zmiennoprzecinkowej w wyrażeniu powoduje wykorzystanie arytmetyki liczb zmiennoprzecinkowych, a nie całkowitych.

Liczby zmiennoprzecinkowe w standardowych dystrybucjach CPython zaimplementowane są jako typ `double` z języka C, dlatego mają taką precyzję, jaką kompilator języka C wykorzystany do zbudowania interpretera Pythona przydzieli liczbom tego typu.

Literały zespolone mają postać $a + b \cdot \mathrm{j}$, gdzie a to część **rzeczywista**, b to część **urojona** liczby, a j to jednostka urojona $\mathrm{j}^2 = -1$.

Jeżeli `z` jest zmienną zespoloną, to `z.real` i `z.imag` to odpowiednio część rzeczywista i urojona tej liczby.

literał, interpretacja	
`1234`; `-24`; `0`; `99999999999999`	liczby całkowite o nieograniczonej wielkości (*integer*)
`1.23`; `1.`; `3.14e-10`; `4E21`	liczby zmiennoprzecinkowe (*floating-point*)
`0o177`; `0x9ff`; `0b101010`	literały ósemkowe, szesnastkowe i dwójkowe
`3+4j`; `3.0+4.0j`; `3J`	liczby zespolone (*complex*)
`set('foo')`; `{1, 2, 3, 4}`	zbiory
`Decimal('1.0')`; `Fraction(1,3)`	typy rozszerzające: dziesiętne i ułamkowe
`bool(foo)`; `True`; `False`	typ logiczny i stałe

Przykłady formatowania liczb

```
a = 3
b = 4
print(b / (2 + a)) # 0.8

foo = 1 / 3
bar = 1 / 6
num = 5

print(foo) # 0.3333333333333333
print("foo = %e" % foo) # 3.333333e-01
print("foo = %4.2f" % foo) # 0.33

print("foo = {}; bar = {}" . format(foo, bar))
# foo = 0.3333333333333333; bar = 0.16666666666666666

print("bar = {1}; foo = {0}" . format(foo, bar))
# bar = 0.16666666666666666; foo = 0.3333333333333333

print("foo = {0:4.2f}; bar = {1:4.4f}; num = {2:3d}"
   . format(foo, bar, num))
# foo = 0.33; bar = 0.1667; num =   5
```

Napisy (łańcuchy znaków)

Z funkcjonalnego punktu widzenia łańcuchy znaków można wykorzystać do reprezentowania wszystkiego, co można zakodować w postaci tekstowej lub bajtowej.

W dziedzinie tekstów obejmuje to między innymi symbole i słowa, pliki tekstowe załadowane do pamięci, adresy internetowe czy kod źródłowy programów.

Ciągów znaków można również używać do przechowywania surowych danych bajtowych używanych na przykład w plikach multimedialnych i transferach sieciowych, a także zakodowanych i zdekodowanych tekstów w formacie Unicode.

Napisy w Pythonie należą do typu `str`. Literały tego typu można tworzyć na kilka sposobów:

- napisy mieszczące się w jednej linii można umieścić wewnątrz **cudzysłowów** (zalecane) lub **apostrofów**,
- napisy wielowierszowe umieszcza się wewnątrz potrójnych cudzysłowów lub apostrofów,
- poprzedzenie napisu literą `r` spowoduje, że znaki specjalne (takie jak `n`) staną się zwykłymi znakami.

Napisy są obiektami niemodyfikowalnymi. Każda operacja modyfikująca napis powoduje stworzenie nowego obiektu. W szczególności nie należy zapętlać operacji z wykorzystaniem `+=`.

Od Pythona 3.3 napis ma reprezentację znaków o długości 1, 2 lub 4 bajty, ustalaną w czasie uruchomienia według wartości znaków.

Aby wymusić stworzenie ciągu znaków **ASCII**, trzeba poprzedzić napis literą `b`.

Funkcje `ord()` i `chr()` służą do konwersji znaku na wartość kodu i zmiany kodu na znak.

Znaki diakrytyczne różnych języków i znaki specjalne są wspierane na wiele sposobów:

- zakodowanie znaku bezpośrednio np.: `ą`,
- zakodowanie znaku poprzez kod Unicode np.: `\u0105`,
- zakodowanie znaku poprzez nazwę Unicode np.: `\N{LATIN SMALL LETTER A WITH OGONEK}`,
- zakodowanie znaku poprzez łączenie nazw Unicode np.: `\N{LATIN SMALL LETTER A}\N{COMBINING OGONEK}`.

literały, operacje na napisach

`S = ”`	pusty łańcuch znaków
`S = "zmienna 'foo'"`	cudzysłowy
`S = 'f\no\to\x00bar'`	sekwencje znaków specjalnych
`S = """...wiele wierszy..."""`	Blok w potrójnych cudzysłowach
`S = r'\temp\foo'`	surowy łańcuch znaków (bez znaków ucieczki)
`S = b'bar'`	bajtowe łańcuchy znaków
`S = u'foo'`	łańcuch znaków Unicode
`S + S;  S * 3`	konkatenacja i powtórzenie
`S[i];  S[i:j];  len(S)`	indeksowanie, wycinek, długość
`S.find('foo')`	wyszukiwanie
`S.rstrip()`	usuwanie białych znaków
`S.replace('foo', 'bar')`	zastępowanie
`S.split(',')`	dzielenie w miejscu wystąpienia separatora
`S.lower()`	konwersja wielkości liter

Sekwencje ucieczki (sekwencje specjalne) pozwalają osadzać w łańcuchach znaki, których nie da się łatwo wpisać z klawiatury.

Znak \ i jeden lub większa liczba następujących po nim znaków w literale łańcuchowym w wynikowym obiekcie łańcucha znaków zastępowane są pojedynczym znakiem o wartości binarnej określonej przez sekwencję ucieczki.

sekwencje specjalne, znaczenie	
\\	ukośnik wsteczny (czasem lewy)
\'	apostrof
\"	cudzysłów
\a	sygnał dźwiekowy
\b	znak cofania (*backspace*)
\f	wysunięcie strony (*form feed*)
\n	nowy wiersz (*line feed*)
\r	powrót karetki
\t	tabulator poziomy
\v	tabulator pionowy

Listy

Obiekt reprezentujący listę jest ogólnym rodzajem **sekwencji**. Listy mają typ `list`.

Listy są uporządkowanymi kolekcjami dowolnych obiektów. Zachowują uporządkowanie elementów (są sekwencjami).

Dostęp do elementów list można uzyskać za pomocą pozycji przesunięcia. Element listy można z niej pobrać, indeksując listę w pozycji przesunięcia obiektu. Ponieważ elementy listy są uporządkowane pozycyjnie, możliwe jest wykonywanie wycinków czy konkatenacja.

Listy mają zmienną długość. Mogą również zawierać dowolny typ obiektów, są zatem niejednorodne (heterogeniczne). Ponieważ listy mogą zawierać inne obiekty, obsługują również dowolne zagnieżdżanie. Możliwe jest tworzenie list składających się z innych list.

Listy należą do sekwencji mutowalnych, można modyfikować je w miejscu i reagują na wszystkie operacje na sekwencjach, takie jak indeksowanie, wycinki i konkatenacja. Operacje takie po zastosowaniu do list zwracają nowe listy. Listy obsługują również operacje usuwania czy przypisania do indeksu.

Listy są tablicami referencji do obiektów, zawierają zero lub większą liczbę referencji do innych obiektów. Listy przypominają tablice wskaźników (adresów), dlatego pobranie elementu z listy jest bardzo szybkie.

Przypisanie obiektu do komponentu struktury danych czy nazwy zmiennej, spowoduje przechowanie referencji do samego obiektu, a nie jego kopię (o ile tego w jawny sposób nie zażądamy).

literały, operacje na listach

`L = []`	pusta lista
`L = [123, 'abc', 1.23, ]`	cztery elementy, indeksy od 0 do 3
`L = ['Adam', 40.0, ['prog', 'python']]`	zagnieżdżone podlisty
`L = list('mielonka')`	lista elementów obiektu iterowanego
`L = list(range(-4, 4))`	lista kolejnych liczb całkowitych
`L[i]; L[i][j]; L[i:j]; len(L)`	indeks, indeks indeksu, wycinek, długość
`L + L; L * 3`	konkatenacja, powtórzenie
`for x in L:print(x); 3 in L`	iteracja, przynależność
`L.append(4); L.extend([5, 6, 7])`	dodawanie elementów
`L.insert(i, X)`	
`L.index(X); L.count(X)`	przeszukiwanie
`L.sort(); L.reverse(); L.copy()`	sortowanie, odwracanie, kopiowanie, czyszczenie
`L.clear()`	
`L.pop(i); L.remove(X); del L[i]`	zmniejszanie listy
`del L[i:j]; L[i:j] = []`	
`L[i] = 3; L[i:j] = [4, 5, 6]`	przypisanie do indeksu, przypisanie do wycinka
`L = [x**2 for x in range(5)]`	listy składane i odwzorowania
`List(map(ord, 'foobar'))`	

Jeśli chcemy efektywnie przechować dane tylko jednego typu, możemy użyć tablicy.

```
import array

tab = array.array('d', [1, 2, 3, 4])
```

Alternatywnie można użycie klasy `numpy.array`. W praktyce dużo lepsze dla obliczeń matematycznych, zawiera wbudowaną wektoryzację wielu operacji.

Słowniki

Słowniki to struktury danych, które przechowują pary **klucz-wartość**. Klucze muszą być obiektami niemodyfikowalnymi. Słowniki należą do typu `dict`.

Słowniki nie są sekwencjami, ale są określane jako **odwzorowania** (*mappings*). Odwzorowania są również zbiorami innych obiektów, w słownikach obiekty są przechowywane według klucza, a nie ich pozycji względnej.

Słowniki, mogą rozszerzać się i kurczyć w miejscu (bez tworzenia nowych kopii) i mogą zawierać obiekty dowolnego typu. Obsługują zagnieżdżanie na dowolną głębokość (mogą zawierać listy, inne słowniki itp.).

Podobnie jak listy, są elastycznym narzędziem do reprezentowania kolekcji, ale ich bardziej mnemoniczne klucze lepiej nadają się do zastosowania, kiedy elementy kolekcji są nazwane lub oznaczone.

Każdy klucz może mieć tylko jedną powiązaną wartość, ale taka wartość może być zbiorem wielu obiektów, a każdą wartość można zapisać pod dowolną liczbą kluczy.

Słowniki dobrze zastępują rekordy, tabele wyszukiwania i wszelkie inne rodzaje agregacji, w których nazwy przedmiotów są bardziej znaczące niż ich pozycje.

Do pobrania komponentów słownika wykorzystuje się tę samą operację indeksowania co w przypadku list, jednak indeks ma tutaj postać klucza, a nie względnej wartości przesunięcia.

Python wykorzystuje zoptymalizowane algorytmy haszujące, służące do odnajdywania kluczy, dzięki czemu to pobieranie jest szybkie.

Podobnie jak listy, tak i słowniki przechowują referencje do obiektów (a nie ich kopie, o ile nie zdefiniuje się tego w sposób jawny).

literały, operacje na słownikach

`D = {}`	pusty słownik
`D = {'imie': 'Adam', 'wiek': 33}`	słownik dwuelementowy
`E = {'cto': {'imie': 'Adam', 'wiek': 33}}`	zagnieżdżanie
`D = dict(imie='Adam', wiek=33)`	inne techniki tworzenia: słowa kluczowe,
`D = dict([('imie', 'Adam'), ('wiek', 33)])`	pary klucz-wartość, spięte pary klucz-wartość,
`D = dict(zip(keyslist, valueslist))`	listy kluczy
`D = dict.fromkeys(['imie', 'wiek'])`	
`D['imie'];  D['cto']['wiek']`	indeksowanie po kluczu
`'wiek' in D`	sprawdzanie przynależności klucza
`D.keys()`	wszystkie klucze
`D.values()`	wszystkie wartości
`D.items()`	wszystkie krotki klucz+wartość
`D.copy()`	kopiowanie
`D.clear()`	czyszczenie (usuwa wszystkie elementy)

literały, operacje na słownikach

`D.update(D2)`	łączenie według kluczy
`D.get(key, default?)`	pobieranie według klucza, wartości domyślne (lub `None`), gdy brak klucza
`D.pop(key, default?)`	usuwanie według klucza, wartości domyślne (lub błąd), gdy brak klucza
`D.setdefault(key, default?)`	pobieranie według klucza, wartości domyślne (lub `None`), gdy brak klucza
`D.popitem()`	usuwanie lub zwracanie pary (klucz, wartość)
`len(D)`	długość (liczba przechowywanych wpisów)
`D[key] = 42`	dodawanie lub modyfikacja kluczy
`del D[key]`	usuwanie elementu według klucza
`list(D.keys())`	widoki słowników
`D1.keys() & D2.keys()`	
`D = {x: x*2 for x in range(10)}`	słowniki składane

Krotki

Krotki (pary, trójki) to listy o stałym rozmiarze, których nie można modyfikować. Krotki należą do typu `tuple`.

Krotki są uporządkowanymi kolekcjami obiektów — zachowują zatem kolejność elementów od lewej do prawej strony. Tak jak w przypadku list, również w krotkach możemy osadzać dowolne typy obiektów.

Dostęp do elementów krotki odbywa się za pomocą wartości przesunięcia (a nie według klucza). Krotki obsługują wszystkie operacje oparte na wartościach przesunięcia, w tym indeksowanie i wycinki.

Krotki są sekwencjami — obsługują wiele z operacji odnoszących się do typów sekwencyjnych. Są jednak niemutowalne, stąd nie obsługują żadnych operacji modyfikujących w miejscu, które mają zastosowanie do list.

Krotki mają stałą długość, są heterogeniczne i można je dowolnie zagnieżdżać. Ponieważ krotki są niemutowalne, nie można zmieniać ich rozmiaru bez tworzenia kopii.

Krotki mogą przechowywać dowolne typy obiektów, w tym inne obiekty złożone (na przykład listy, słowniki i inne krotki) i mogą być praktycznie dowolnie zagnieżdżone.

Krotki są tablicami referencji do obiektów. Krotki przechowują punkty dostępu do innych obiektów (referencje), a indeksowanie krotek jest dość szybkie.

literały, operacje na krotkach

`()`	pusta krotka
`T = (0,)`	krotka jednoelementowa (nie jest wyrażeniem)
`T = (0, 'Ni', 1.2, 3)`	krotka czteroelementowa
`T = 0, 'Ni', 1.2, 3`	inna krotka czteroelementowa (ta sama co wyżej)
`T = ('Adam', ('prog', 'pyton'))`	zagnieżdżone krotki
`T = tuple('foo')`	krotka elementów w obiekcie iterowalnym
`T[i]; T[i][j]; T[i:j]; len(T)`	indeks, indeks indeksu, wycinek, długość
`T + T; T * 3`	konkatenacja, powtórzenie
`for x in T: print(x)`	iteracja, przynależność
`'foo' in T`	
`[x ** 2 for x in T]`	
`T.index('Ni'); T.count('Ni')`	wyszukiwanie, zliczanie
`namedtuple('Emp', ['name', 'jobs'])`	krotka z nazwanymi polami (krotka typu named tuple)

Operator w programowaniu konstrukcja językowa jednoargumentowa, bądź wieloargumentowa zwracająca wartość.

Do podstawowych operatorów, będących elementem większości języków programowania, należą operatory: **przypisania**, **arytmetyczne**, **relacji** (porównania), **logicznie**.

Główne cechy opisujące operator to:

- liczba i typy argumentów,
- typ wartości zwracanej,
- wykonywane działanie,
- priorytet,
- łączność lub jej brak,
- umiejscowienie operatora względem operandów.

Przypisanie

Przypisanie (podstawienie) jest to operacja nadania, umieszczenia, wpisania do określonej **L-wartości** (jest to wartość, która istnieje dłużej niż przez jedno wyrażenie i można pobrać jej adres, jest nią też zmienna) nowej wartości.

Przypisanie może zostać dokonane:

- instrukcją przypisania,
- operatorem przypisania,
- innym operatorem,
- w inicjalizacji zmiennej,
- w wywołaniu podprogramu,
- w wyniku efektów ubocznych,
- w instrukcji wejścia.

Operator przypisania to jeden z podstawowych operatorów prostych występujących w językach programowania.

Zwykle nie jest słowem kluczowym, choć istnieją języki programowania wymagające lub zezwalające opcjonalnie na użycie słowa kluczowego.

W Pythonie operatorem przypisania jest znak równości =. Operator przypisania występuje w **instrukcji przypisania**.

Operator przypisania wartościuje listę wyrażeń (to może być pojedyncze wyrażenie lub lista wyrażeń rozdzielonych przecinkami, gdzie drugi przypadek reprezentuje krotkę) i przypisuje każdemu z elementów z listy docelowej pojedynczy obiekt wynikowy, w kolejności od lewej do prawej.

Przypisania tworzą referencje do obiektów. Zmienne tworzone są przy pierwszym przypisaniu. Przed odwołaniem się do zmiennej trzeba ją najpierw przypisać. Niektóre operacje wykonują przypisania niejawne.

Operator przypisania w Pythonie jest **lewostronnie łączny**. Umożliwia to łączenie przypisań:

```
i = 5

l = k = j = i;
```

operacja, interpretacja	
`foo = 'bar' i = 9`	forma podstawowa
`foo, bar = 'ala', 'pies'`	przypisanie krotki (pozycyjne)
`[foo, bar] = ['ala', 'pies']`	przypisanie listy (pozycyjne)
`a, b, c, d = 'foo'`	przypisanie sekwencji, uogólnione
`a, *b = 'bar'`	rozszerzone rozpakowanie sekwencji
`foo = bar = 'var'`	przypisanie do wielu celów
`foo += 42`	przypisanie rozszerzone (`foo = foo + 42`)

Operatory arytmetyczne

Operator arytmetyczny to operator, który działając na podanych argumentach reprezentujących wartości liczbowe, w wyniku zwraca również wyznaczoną wartość liczbową, realizując podstawowe operacje arytmetyki.

To jakie operatory arytmetyczne są dostępne w konkretnym języku programowania zależy od jego składni, a to jakie są zasady ich stosowania, w tym priorytet tych operatorów i kolejność opracowywania argumentów, od przyjętej implementacji języka.

Zróżnicowany jest również sposób zapisu operatorów arytmetycznych. Stosuje się zapis, bądź za pomocą symboli (znaku lub znaków nie będących literami), w konwencji zbliżonej do matematycznej, bądź zdecydowanie rzadziej w postaci słów kluczowych.

Operatory arytmetyczne realizują następujące operacje arytmetyczne (przykłady):

- jednoargumentowe:
  - zmiana znaku liczby (wyznaczenie liczby przeciwnej),
  - zachowanie znaku liczby,
  - inkrementacja,
  - dekrementacja,
- dwuargumentowe:
  - dodawanie,
  - odejmowanie,
  - mnożenie,
  - dzielenie,
  - dzielenie całkowitoliczbowe,
  - reszta z dzielenia całkowitoliczbowego,
  - potęgowanie.

operator, opis	
`x + y`	dodawanie, konkatenacja
`x - y`	odejmowanie, różnica zbiorów
`x * y`	mnożenie, powtórzenie
`x % y;`	reszta z dzielenia, format
`x / y; x // y;`	dzielenie, dzielenie bez reszty
`x ** y`	potęga
`-x; +x`	negacja, identyczność

Wyrażenie arytmetyczne jest to językach programowania dowolne wyrażenie typu liczbowego. Może być ono złożone ze zmiennych, liczb, funkcji, symboli działań (tu: operatorów), itp.

```
result =  10  * 3  # 30
result =  10  / 3  # 3
result =  10  % 3  # 1
result =  10 ** 2  # 100
result =  10  + 3  # 13
result =  10  - 3  # 7
result =  5        # 5
result = -result   # -5
```

```
result = 10.0 / 3.0 # 3.333333
result = 10 / 3     # 3 wynik moze zalezec od typow literalow
                    # i zmiennej: C, C++, Python 2.x
```

Kolejność (priorytety) wykonywania działań jest taka jak w matematyce, można ją regulować za pomocą nawiasów okrągłych. Operatory równoważne wykonywane są od strony lewej do prawej (należy wziąć pod uwagę wiązanie).

wybrane funkcje matematyczne z `math`	
`sin()`	sinus
`cos()`	cosinus
`tan()`	tangens
`asin()`	arcus sinus
`acos()`	arcus cosinus
`atan()`	arcus tangens
`exp()`	eksponent (e^x)
`log()`	logarytm naturalny
`log10()`	logarytm dziesiętny
`pow()`	potęga
`sqrt()`	pierwiastek kwadratowy
`ceil()`	zaokrąglenie do liczby całkowitej w górę
`floor()`	zaokrąglenie do liczby całkowitej w dół
`fabs()`	wartość bezwzględna

Operatory relacji i wyrażenia logiczne

Operator **relacji** jest to operator który działając na podanych argumentach, w wyniku zwraca wartość logiczną, określającą spełnienie bądź nie spełnienie reprezentowanej przez ten operator relacji zachodzącej między argumentami.

Wynikiem działania operatora relacji jest więc wartość reprezentująca zgodnie z zasadami obowiązującymi w składni języka programowania jedną z wartości logicznych: **prawdę** (*true*) lub **fałsz** (*false*).

W językach programowania dostępne są operatory, które badają relacje:

- równości,
- nierówności,
  - negacji równości,
  - nierówności ostrych,
    - mniejsze,
    - większe,
  - nierówności nieostrych,
    - mniejsze lub równe,
    - większe lub równe,
- przynależności (zawierania),
- równoważności.

operatory relacji	
==	czy równe?
!=	czy różne?
>	czy większe?
<	czy mniejsze?
>=	czy większe lub równe?
<=	czy mniejsze lub równe?

Operatory relacji mogą działać na wszystkich typach danych, jednak najczęściej odnoszą się do danych numerycznych.

Porównanie danych całkowitych nie powinno budzić zastrzeżeń, ponieważ operacje wykonywane są w sposób naturalny.

Należy zachować ostrożność przy porównywaniu wartości zmiennoprzecinkowych.

```
first = 1.00000000000001
second = 1.00000000000002

if(first == second)
  ...
```

```
first = 1.00000000000001
second = 1.00000000000002

if(fabs(first - second) < 0.000001)
  ...
```

Operator **logiczny** to operator, który działając na argumentach reprezentujących wartości logiczne, w wyniku zwraca również wartość logiczną, realizując podstawowe operacje algebry Boole'a.

Dostępność operatorów logicznych, priorytet i kolejność opracowywania argumentów zależy od przyjętej implementacji języka.

Zróżnicowany jest również sposób zapisu operatorów logicznych, stosuje się zapis, bądź w postaci słów kluczowych, bądź symboli (znaku lub znaków nie będących literami).

Operatory logiczne realizują następujące operacje logiczne:

- jednoargumentowe:
  - negacja,
- dwuargumentowe:
  - koniunkcja,
  - alternatywa,
  - alternatywa wykluczająca,
  - implikacja,
  - ekwiwalencja.

operatory logiczne	
`not`	negacja (zaprzeczenie)
`and`	koniunkcja
`or`	alternatywa

`not`, nargument, wynik	
prawda	fałsz
fałsz	prawda

and, argumenty, wynik		
prawda	prawda	prawda
prawda	fałsz	fałsz
fałsz	prawda	fałsz
fałsz	fałsz	fałsz

or, argumenty, wynik		
prawda	prawda	prawda
prawda	fałsz	prawda
fałsz	prawda	prawda
fałsz	fałsz	fałsz

Wyrażenie logiczne jest to językach programowania dowolne wyrażenie zawierające operatory relacji i operatory logiczne, stałe, zmienne logiczne, którego wynik jest typu logicznego (prawda lub fałsz).

Wyrażenia logiczne mogą zawierać wyrażenia innego typu, np. arytmetyczne.

```
result = (5 < 3 * 2) and ('L' > 'A')
result = (5 != 3 * 2) or ('L' == 'A')
result = not(2 + 2 == 5)
```

```
value = 25
even = (value % 2) == 0
extent = ((value >= 10) and (value <= 20))
result = even and extent
```

Operatory identyczności pozwalają określić, czy dwie zmienne przechowują ten sam obiekt. Występują w dwóch wersjach: `is` i `not is`.

```
x = "ala ma psa"
y = "ala nie ma psa"

if x is not y:
  print("obiekty nie sa takie same")

x = y

if x is y:
  print("obiekty sa takie same")
```

Operatorem `is` (`not is`) można także sprawdzić, czy dany obiekt jest oczekiwanym typem.

```
x = "ala ma psa"
y = 25

if type(x) is str:
  print("obiekt x jest typu str")

if type(y) is int:
  print("obiekt x jest typu int")
```

Operatory przynależności tego typu sprawdzają, czy dany element zawiera się w podzbiorze wartości danego obiektu (po których można iterować). Występują w dwóch wersjach: `in` i `not in`.

```
x = "ala ma psa"

if "ma" in x:
  print("wyraz 'ma' wystepuje w ciagu")

y = [2, 3, 4, 66]

if 4 in y:
  print("liczba 4 wystepuje w zbiorze")
else:
  print("liczba 4 nie wystepuje w zbiorze")
```

Dla większości operatorów dwuargumentowych można wykorzystać specjalne operatory przypisania (*shorthands*), pozwalające skrócić zapis, na przykład:

- mnożenie: `a *= b` zamiast `a = a * b`,
- dzielenie: `a /= b` zamiast `a = a / b`,
- reszta z dzielenia całkowitoliczbowego: `a %= b` zamiast `a = a % b`,
- dodawanie: `a += b` zamiast `a = a + b`,
- odejmowanie: `a -= b` zamiast `a = a - b`,

Operator warunkowy

Często spotyka się *symetryczne* instrukcje warunkowe:

```
if(delta > 0):
  isSolution = 1
else:
  isSolution = 0
```

```
if(a > b):
  max = a
else:
  max = b
```

Można je zapisać krócej z wykorzystaniem **operatora warunkowego** (operatora trójargumentowego):

```
isSolution = 1 if(delta > 0) else 0
```

```
max = a if(a > b) else b
```

Instrukcja warunkowa

Instrukcja warunkowa jest elementem języka programowania, który pozwala na wykonanie różnych obliczeń (zadań) w zależności od tego czy zdefiniowane przez programistę **wyrażenie logiczne** jest prawdziwe, czy fałszywe.

```
a = 1
b = 2

if(a + b == 3):

  print("suma jest rowna 3")
```

```
take = 29999.99

if(take >= 30000):

  bonus = 0.05 * take
  print("Premia w wysokosci: %f" % bonus)

else:

  bonus = 0
  print("Premii nie będzie!")
```

```
a = 1
b = 8
c = 2

delta = b * b - 4 * a * c

if(delta > 0):

  x1 = (-b - math.sqrt(delta)) / (2 * a)
  x2 = (-b + math.sqrt(delta)) / (2 * a)

  print("x1 = %f; x2 = %f" %(x1, x2))

elif(delta == 0):

  x0 = -b / (2 * a)
  print("x0 = %f" % x0)

else:

  print("Brak pierwiastkow rzeczywistych")
```

Instrukcja wyboru

Instrukcja wyboru jest instrukcją umożliwiającą wybór instrukcji do wykonania spośród wielu opcji.

Składnia instrukcji wyboru różni się w zależności od języka programowania, lecz można wyróżnić w niej charakterystyczne elementy:

- nagłówek instrukcji wyboru – słowo kluczowe rozpoczynające instrukcję, nazwa zmiennej lub wyrażenie, na podstawie którego następuje wybór,
- ciało instrukcji wyboru – kolejne instrukcje (bloki instrukcji) podlegające selekcji, poprzedzone frazami zawierającymi wartości, listy lub zakresy porównywane z wyrażeniem (zmienną) z nagłówka instrukcji,
- opcjonalnie fraza domyślna, wykonywana gdy żadna z fraz nie spełni warunku,
- koniec instrukcji wyboru.

Instrukcja wyboru (zwykle `switch(expression)` ... `case constant: statement`) nie występuje w Pythonie. Można ją zastąpić wielokrotną instrukcją wyboru.

Pętle

Pętla to jedna z podstawowych konstrukcji programowania strukturalnego (obok instrukcji warunkowej i instrukcji wyboru). Umożliwia cykliczne wykonywanie ciągu instrukcji:

- określoną liczbę razy (pętla **iteracyjna**, licznikowa),
- do momentu zajścia pewnych warunków (pętla **repetycyjna**, warunkowa),
- dla każdego elementu kolekcji (pętla **foreach**, po kolekcji),
- w nieskończoność.

Pętla iteracyjna, to rodzaj pętli, w której następuje wykonanie określonej liczby iteracji. Do kontroli przebiegu wykonania pętli iteracyjnej stosuje się specjalną zmienną, którż nazywa się **zmienną sterująca**, **kontrolną** lub **licznikową**.

W ramach pętli przejście do kolejnej iteracji wiąże się ze zmianą wartości zmiennej sterującej o określoną wielkość i sprawdzenie warunku, czy nowa wartość zmiennej sterującej znajduje się nadal w dopuszczalnym zakresie wartości, określonym dla tej zmiennej.

Kolejność wykonywania działań jest następująca:

- przypisanie wartości początkowej do zmiennej sterującej,
- sprawdzenie, czy wartość zmiennej sterującej mieści się w dopuszczalnym zakresie wartości, tzn. jej wartość jest równa lub mniejsza od wartości granicznej, albo jest równa lub większa od wartości granicznej:
  - jeżeli wartość zmiennej sterującej nie mieści się w dopuszczalnym zakresie wartości to kończy wykonywanie pętli,
  - jeżeli wartość zmiennej sterującej mieści się w dopuszczalnym zakresie wartości to nie przerywa działania,
- wykonuje iterację,
- zmienia wartość zmiennej sterującej o zadany krok, wartość ta może być zwiększana lub zmieszana.

Zmienna sterująca służy do kontroli przebiegu realizacji pętli iteracyjnej. Przyjmuje ona kolejne wartości z zadanego zakresu zmieniane o określoną wartość kroku.

W różnych językach programowania mogą być stawiane określone wymagania i ograniczenia dotyczące zmiennych sterujących. Ograniczenia te mogą dotyczyć takich atrybutów tej zmiennej jak np. dozwolony typ danych, zasięgu.

Zmiana wartości zmiennej sterującej może nastąpić:

- automatycznie, w sposób ukryty,
- jawnie, w kodzie bloku pętli, o ile dany język programowania dopuszcza taką konstrukcję.

Ogólniejszą konstrukcją jest pętla **repetycyjna** (warunkowa), która jest wykonywana, aż do odpowiedniej zmiany warunków.

Warunek zawarty w definiowanej pętli jest pewnym wyrażeniem, które najczęściej zwraca wartość typu logicznego. Istnieją języki programowania, w których składnia nie przewiduje takiego typu danych.

W językach tych stosuje się wyrażenia zwracające pewną wartość innego typu, która następnie podlega odpowiedniej interpretacji, np. wartość zero może być utożsamiana z wartością **false** typu logicznego, a pozostałe wartości z wartością **true**.

Zapis wyrażenia, oraz typ wartości wyrażenia, zależny jest od składni konkretnego języka programowania. Powszechnym jest zapis wyrażeń kontrolnych analogicznie do zapisu tych wyrażeń dla instrukcji warunkowej.

Szczególne znaczenie mają w tym przypadku operatory porównań, choć warunek może zostać wyrażony także całkiem inaczej, np. jako wywołanie funkcji zwracającej wartość logiczną, bądź jako identyfikator zmiennej logicznej, której wcześniej przypisano rezultat ewaluacji wyrażenia warunkowego.

Ponadto operator logiczny realizujący operację negacji pozwana na rekompensatę ewentualnego braku pętli powtarzanej przy spełnieniu lub niespełnieniu warunku, gdyż zastosowanie tego operatora do podanego warunku jest równoważne zastosowaniu frazy przeciwnej.

Warunki decydujące o kontynuacji lub zaprzestaniu wykonywania pętli mogą być sprawdzane:

- na początku pętli, przed wykonaniem pierwszej instrukcji zawartej w bloku definiowanej pętli,
- wewnątrz pętli, w jej bloku, po wykonaniu części instrukcji,
- na końcu pętli, po wykonaniu wszystkich instrukcji zawartych w bloku definiowanej pętli.

Jeżeli warunek jest sprawdzany na początku pętli, to może nastąpić taka sytuacja, że instrukcje zawarte w pętli nigdy nie zostaną wykonane. Będzie to miało miejsce w sytuacji, gdy przy pierwszym wykonaniu warunek nie będzie spełniony.

Inaczej jest, gdy warunek jest sprawdzany na końcu pętli. W tym przypadku instrukcje zawarte w pętli zostaną wykonane co najmniej jeden raz.

Często pożądane jest, aby instrukcje pętli zostały wykonane dla każdego elementu tablicy, kolekcji itp.

Oczywiście można to zrobić za pomocą pętli iteracyjnych lub repetycyjnych, ale często szybszym i bardziej przejrzystym sposobem jest użycie pętli typu **foreach**, która zwalnia programistę z obowiązku *ręcznego* iterowania po kolekcji.

Działanie pętli foreach, polega na powtarzaniu kolejnych iteracji dla wszystkich elementów wybranego kontenera danych, takiego jak, np. tablica, lista, kolekcja, kolejka, itp.

Pętla taka automatycznie przed przejściem do wykonania kolejnej iteracji przypisuje zadanej w nagłówku pętli zmiennej sterującej wartość kolejnego elementu.

Działanie tej pętli polega na wykonaniu następujących kroków:

- ustaleni jako bieżący, pierwszego element kontenera danych, jeżeli kontener jest pusty, zakończenie działanie pętli,
- przypisanie wartość bieżącego elementu do zmiennej sterującej,
- wykonanie iteracji,
- sprawdzenie czy istnieje kolejny element w kontenerze:
  - jeżeli istnieje, to ustala jako bieżący kolejny element kontenera i wykonuje iterację,
  - jeżeli nie istnieje to kończy działanie.

```
index = 20
counter = 0

while(index > 10):

  index -= 1
  counter += 1
  print("Indeks = %d, licznik = %d" %(index, counter))

# index = 10, counter = 10
```

Instrukcja stanowiąca ciało iteracji może **nie wykonać się ani razu**.

Wyrażenie występujące w nawiasach określa **warunek kontynuacji**.

Pętla wykonuje się dopóty, dopóki **warunek jest prawdziwy**.

```
counter = 0

for index in range(20, 10, -1):
  counter += 1
  print("Indeks = %d, licznik = %d" %(index, counter))

# index = 11, counter = 10
```

Pętla `for` w języku Python jest podobna do pętli `foreach` znanych z innych języków.

Pętla jest uniwersalnym **iteratorem** – może przechodzić przez kolejne elementy w dowolnej uporządkowanej sekwencji lub innym obiekcie iterowalnym.

Instrukcja for działa na łańcuchach znaków, listach, krotkach, innych wbudowanych obiektach, po których można iterować.

```
L = [2, 4, 6, 8]

for x in L:
  print(x, end = " ")
# 2 4 6 8
```

```
S = "python"

for x in S:
  print(x, end = " ")
# p y t h o n
```

```
T = ("ala", "ma", "psa")

for x in T:
  print(x, end = " ")
# ala ma psa
```

Instrukcje break i continue

Instrukcja break pozwala na **natychmiastowe zakończenie** nawet zagnieżdżonej, dowolnej instrukcji **pętli** lub **wyboru**.

Instrukcja continue pozwala na **przerwanie bieżącego przebiegu** dowolnej iteracji i przejście do wykonania następnego jej przebiegu.

```
print("Wpisz dowolna litere, zero konczy iteracje")

while True:

  key = input("Klawisz: ")

  if(key == '0'):
     break

  if(key >= '0' and key <= '9'):
    print("Wpisales cyfre %c o kodzie %d" % (key, ord(key)))
    continue

  print("Wpisales litere %c o kodzie %d" % (key, ord(key)))
```

Podprogramy

Podprogram – termin związany jest z programowaniem proceduralnym.
Podprogram to wydzielona część programu wykonująca jakieś operacje.

Podprogramy stosuje się, aby uprościć program główny, zwiększyć czytelność kodu.
Wartościową cechą podprogramu jest możliwość wielokrotnego jego wywołania.

W wielu językach programowania dzieli się podprogramy na funkcje i procedury.

Funkcja ma wykonywać obliczenia i zwracać jakąś wartość, nie powinna natomiast mieć żadnego innego wpływu na działanie programu.

Procedura natomiast nie zwraca żadnej wartości, zamiast tego wykonuje pewne działania.

Przez **zwracanie wartości** należy rozumieć możliwość użycia wywołania funkcji wewnątrz wyrażenia.

Oprócz powyższego podziału, wyróżnić także należy **podprogram główny**, czyli taki od którego rozpoczyna się wykonywanie programu.

Podprogram może być identyfikowany:

- przez **nazwę** – identyfikator przypisany do podprogramu w jego deklaracji, jest to najczęściej spotykany przypadek w językach wysokiego poziomu,
- przez **etykietę**,
- przez **liczbę** – literał całkowity,
- przez **referencję** (adres) – w językach wysokiego poziomu takie wskaźniki przechowywane są w zmiennych typu **proceduralnego** lub **wskaźnikowego**.

Wywołanie podprogramu może być:

- **funkcyjne** – w wyrażeniu, do którego podprogram zwraca obliczoną wartość, taka forma wywołania dotyczy tylko podprogramów mających cechy funkcji, tzn. zwracających wartość,
- **proceduralne** – czyli jako instrukcja, poprzez nazwę z listą argumentów, lub po słowie kluczowym.

Konkretne implementacje języków często dopuszczają wywołanie funkcji w postaci proceduralnej, tzn. poza wyrażeniami. W tym przypadku zwracana przez podprogram wartość jest ignorowana.

Podprogram jako samodzielna, wydzielona część algorytmu, zazwyczaj musi komunikować się z otoczeniem. Taką komunikację realizuje się za pomocą:

- zmiennych globalnych,
- argumentów (parametrów aktualnych), przypisywanych zdefiniowanym w podprogramie,
- rezultatów funkcji (wartości zwracanych do miejsca wywołania),
- pól obiektu (dla metod danego obiektu),
- innych, rzadko stosowanych lub nie zalecanych (np.: obszarów wspólnych, zmiennych nakładanych).

Funkcje w Pythonie są definiowane za pomocą nowej instrukcji `def`.
W przeciwieństwie do funkcji z języków kompilowanych (np. C) `def` jest **instrukcją wykonywalną** – funkcja w Pythonie nie istnieje, dopóki Python nie dotrze do jej kodu i nie wykona instrukcji `def`.

Poprawne (i czasami użyteczne) jest zagnieżdżanie instrukcji `def` wewnątrz innych instrukcji (`if`, `while`), a nawet innych instrukcji `def`.

Kiedy interpreter dotrze do instrukcji `def` i ją wykona, **generowany jest nowy obiekt funkcji**, który zostaje przypisany do **nazwy funkcji**.

Tak jak w przypadku wszystkich operacji przypisania, nazwa funkcji staje się **referencją do obiektu funkcji**. Do obiektów funkcji można także dołączać dowolne zdefiniowane przez użytkownika atrybuty w celu przechowywania danych.

Definicja funkcji

```
def area(a, b):
  return a * b
```

Przykłady wywołania funkcji

```
print(area(2, 3))
```

```
x = area(2, 3)
```

```
x = 2
y = 3

z = area(x, y)
```

Przekazywanie argumentów odbywa się **zawsze przez wartość**, przy czym ta wartość jest **zawsze referencją** do obiektu, a nie wartością tego obiektu.

W praktyce oznacza to, że w przypadku **niemutowalnych** typów danych (np. liczby całkowite, łańcuchy znaków) funkcje w Pythonie działają tak, jakby ich argumenty przekazywane były **przez wartość**.

Przypisanie do nazw argumentów wewnątrz funkcji nie wpływa na wywołującego. Nazwy argumentów w nagłówku funkcji stają się w czasie wykonywania funkcji nowymi zmiennymi lokalnymi w zasięgu funkcji. Nazwy argumentów funkcji i nazwy zmiennych w zasięgu wywołującego nie są aliasami.

Natomiast w przypadku **mutowalnych** typów danych (np. listy, słowniki) na ogół jak **przez referencję**.

Modyfikacja mutowalnego argumentu w funkcji może mieć wpływ na wywołującego. Argumenty są po prostu przypisywane do przekazywanych obiektów, funkcje mogą modyfikować w miejscu przekazywane obiekty typu zmiennego, a wynik może wpłynąć na wywołującego.

Argumenty zmienne mogą być danymi wejściowymi oraz wyjściowymi funkcji.

```python
def inc(num):
  num += 1
  print("Wartosc w funkcji:", num)
  return num

num = 10

print("Wartosc przed funkcja:", num)

inc(num)

print("Wartosc po funkcji:", num)

# Wartosc przed funkcja: 10
# Wartosc w funkcji: 11
# Wartosc po funkcji: 10
```

```
def change(lst):
  lst.append([1, 2, 3])
  print("Wartosc w funkcji: ", lst)
  return lst

lst = ["ala", "ma", "psa"]

print("Wartosc przed funkcja:", lst)

change(lst)

print("Wartosc po funkcji:", lst)

# Wartosc przed funkcja: ['ala', 'ma', 'psa']
# Wartosc w funkcji:  ['ala', 'ma', 'psa', [1, 2, 3]]
# Wartosc po funkcji: ['ala', 'ma', 'psa', [1, 2, 3]]
```

Program komputerowy działa według określonego algorytmu.

Algorytm – ciąg jasno zdefiniowanych czynności, koniecznych do wykonania pewnego rodzaju zadania.

Zapis algorytmu działania w wybranym języku programowania nazywamy **implementacją algorytmu**.

Jako przykład stosowanego w życiu codziennym algorytmu podaje się często przepis kulinarny. Dla przykładu, aby ugotować bigos należy w określonej kolejności oraz odstępach czasowych (imperatyw czasowy) dodawać właściwe rodzaje kapusty i innych składników.

Przykład ten ma wyłącznie charakter poglądowy, ponieważ język przepisów kulinarnych nie został jasno zdefiniowany. Algorytmy zwykle formułowane są w sposób ścisły w oparciu o język matematyki.

Algorytm prowadzi do rozwiązania zadania w **skończonej liczbie kroków**.

Do danego celu prowadzi zwykle więcej niż jedna droga. Jak więc oceniać alternatywne sposoby rozwiązania problemu?

Podstawowe parametry algorytmu to jego **złożoność czasowa** i **złożoność pamięciowa**. Oprócz tego przy algorytmach działających na liczbach trzeba pamiętać o **stabilności numerycznej**.

Złożoność czasowa mówi, ile kroków obliczeniowych i ile czasu wymaga zakończenie algorytmu dla danej porcji danych.

Złożoność pamięciowa mówi, jaką maksymalnie część danych i wyników pośrednich trzeba w ramach danego algorytmu przechowywać w pamięci operacyjnej.

Stabilność numeryczna określa wrażliwość wyniku końcowego na błędy zaokrągleń w trakcie obliczeń oraz na dokładność danych początkowych.

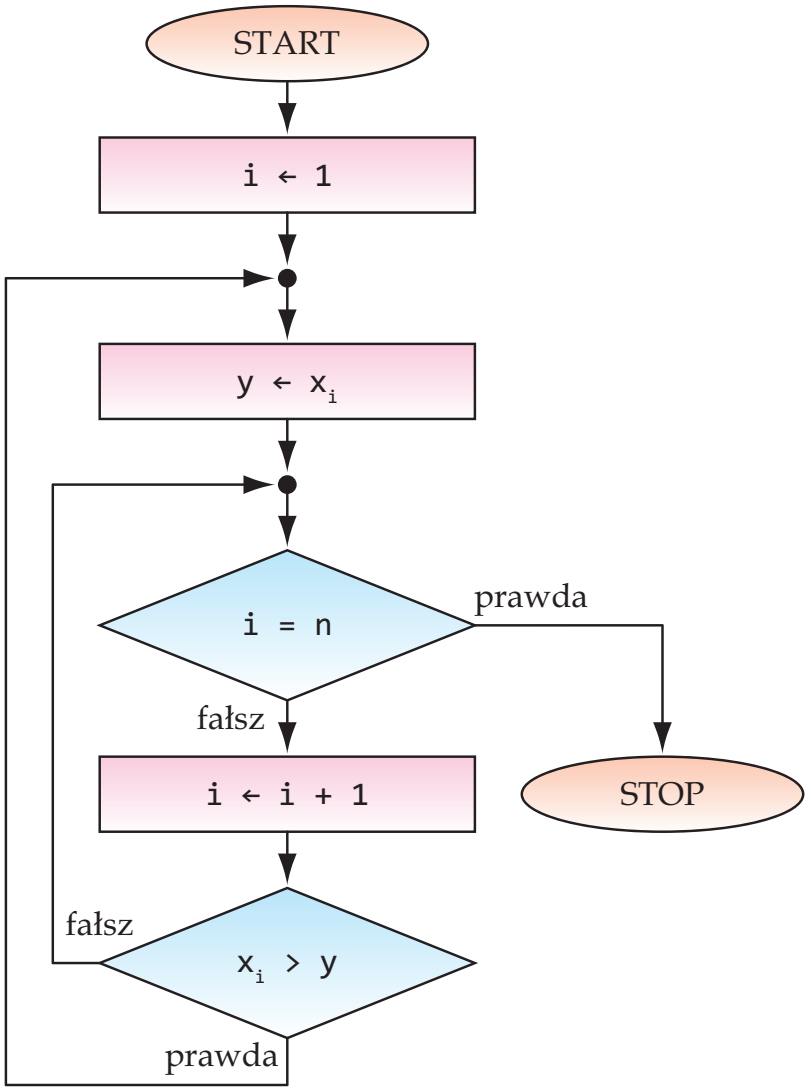
Sposoby zapisu algorytmów

Algorytm znajdowania wartości $y = \max\{x_i\}$, gdzie $1 \leqslant i \leqslant n$.

- **Język naturalny**.
- Schemat blokowy.
- Język formalny.

1. $i \leftarrow 1$, idź do 2,
2. $y \leftarrow x_i$, idź do 3,
3. Czy $i = n$? Jeśli tak – koniec, jeśli nie – idź do 4,
4. $i \leftarrow 1 + 1$, idź do 5,
5. Czy $x_i > y$? Jeśli tak – idź do 2, jeśli nie – idź do 3.

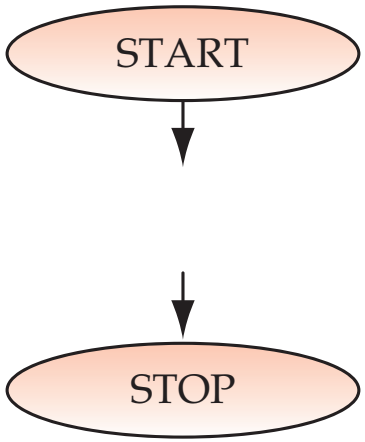
- Język naturalny.
- **Schemat blokowy**.
- Język formalny.



- Język naturalny.
- Schemat blokowy.
- **Język formalny**, C.

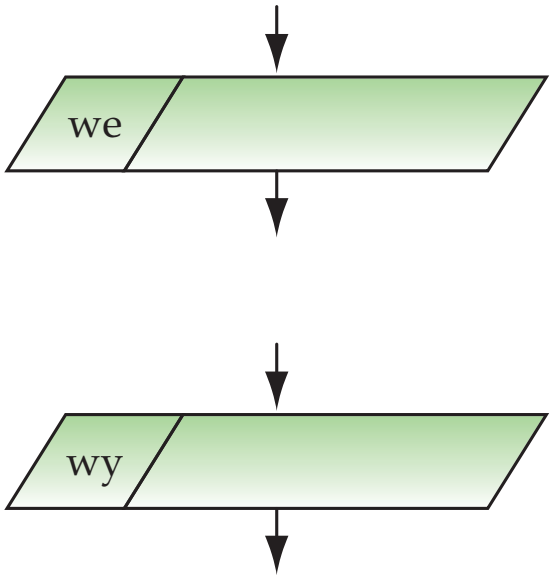
```c
int maxValue(int array[], int size)
{
  int i, max;

  max=array[0];

  for(i=1; i<size; i++)
    {
      if(array[i] > max)
        max = array[i];
    }

   return max;
}
```

Schematy blokowe



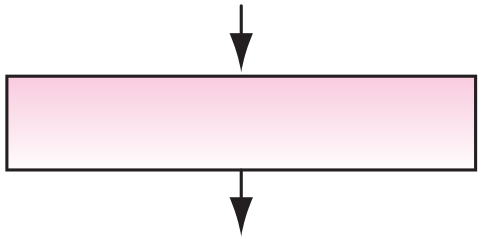
Start – początek programu

Stop – koniec programu

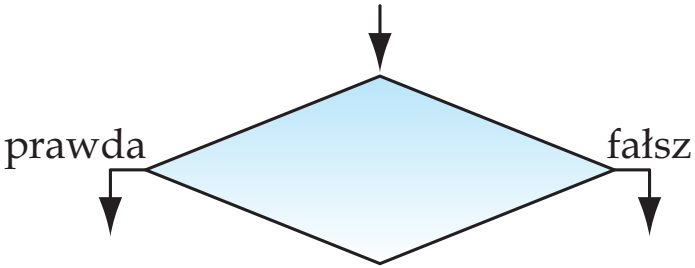


Blok wejścia – wprowadzanie danych

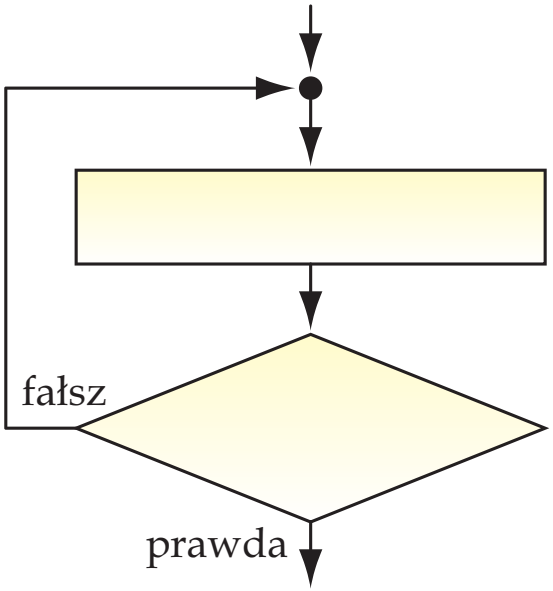
Blok wyjścia – wyprowadzanie wyników



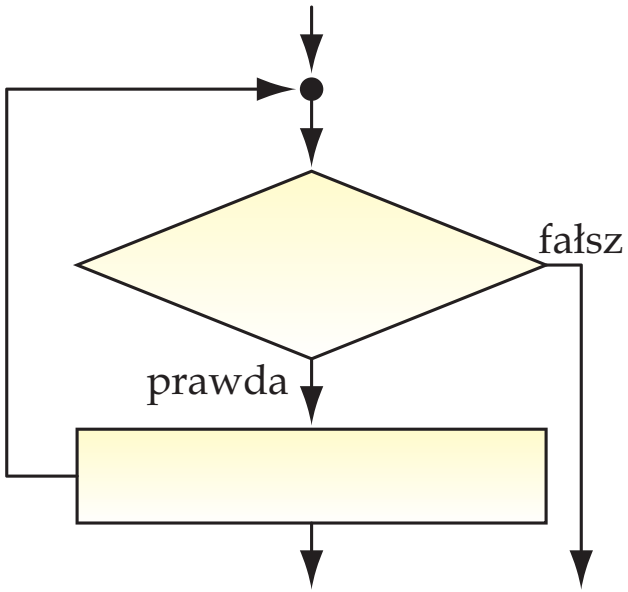
Blok operacyjny – przetwarzanie danych



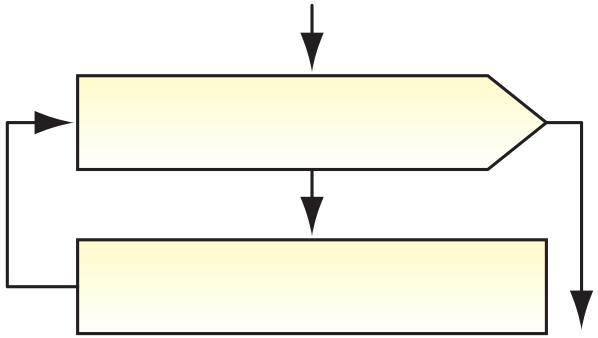
Blok decyzyjny – instrukcja warunkowa



Pętla *powtórz* – z warunkiem po bloku operacyjnym



Pętla *dopóki* – z warunkiem przed blokiem operacyjnym

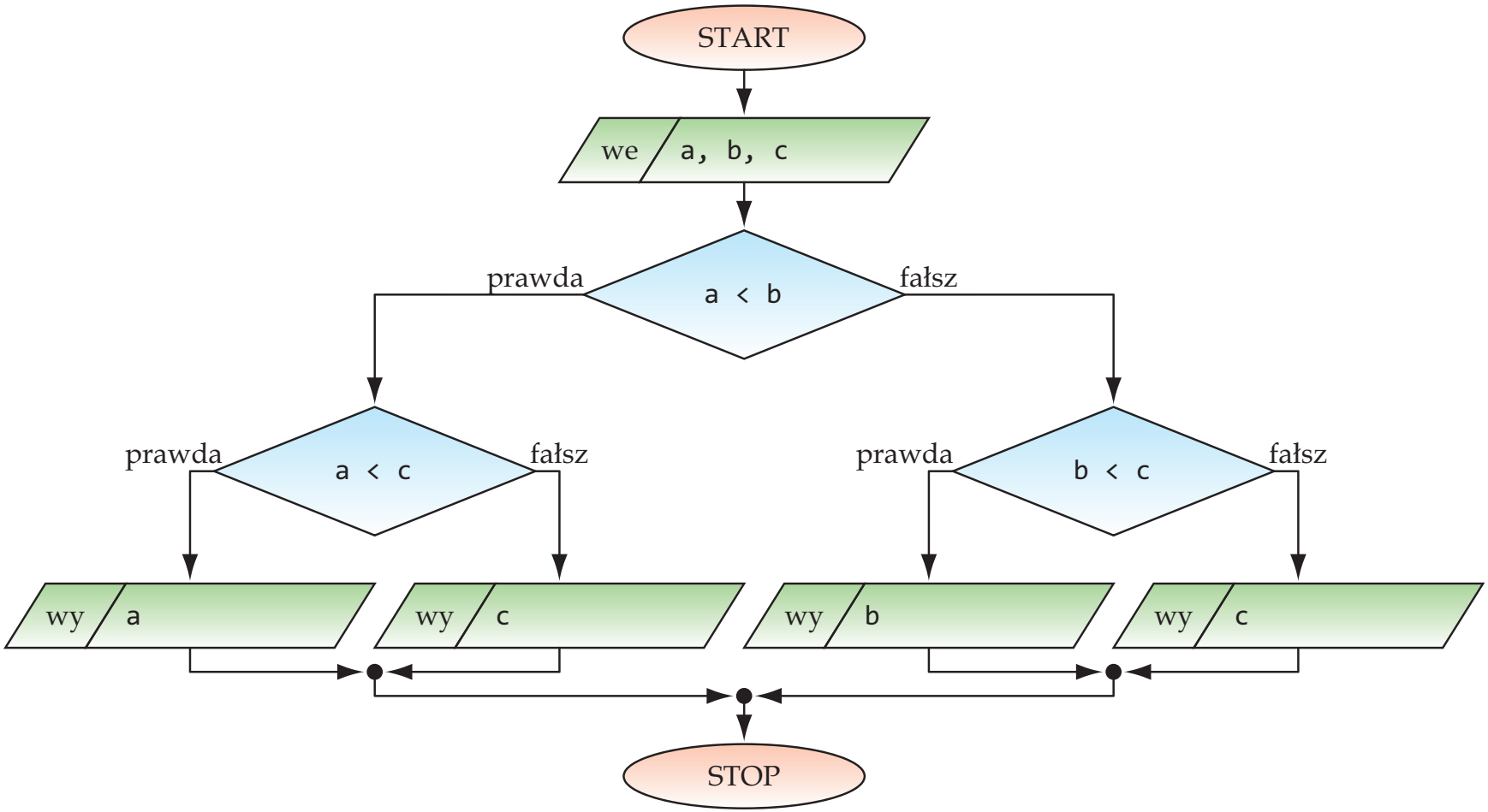


Pętla *dla* – z wiadomą liczbą iteracji

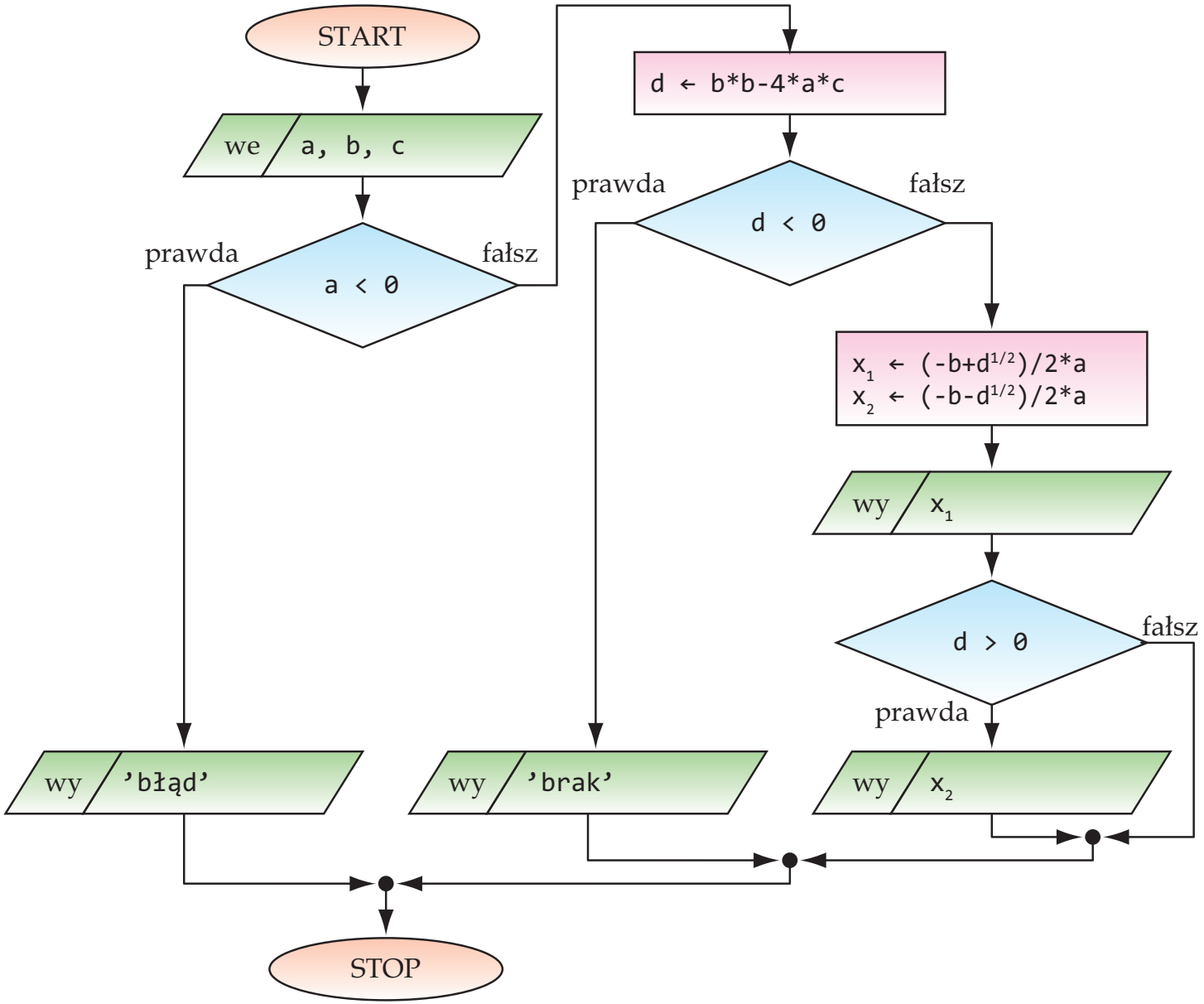
Algorytmy można podzielić na:

- algorytmy liniowe, jeśli wykorzystują operacje bezpośredniego następstwa,
- algorytmy warunkowe, jeśli wykorzystują operacje warunkowe,
- algorytmy iteracyjne, jeśli wykorzystują pętle.

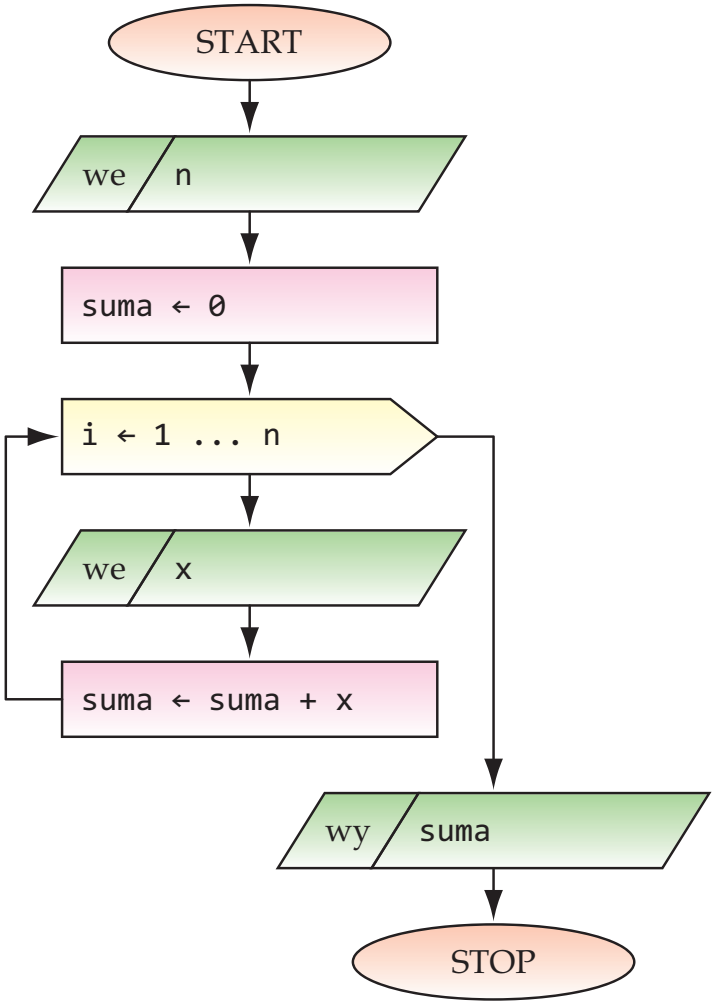
Algorytm szukania wartości minimalnej z trzech



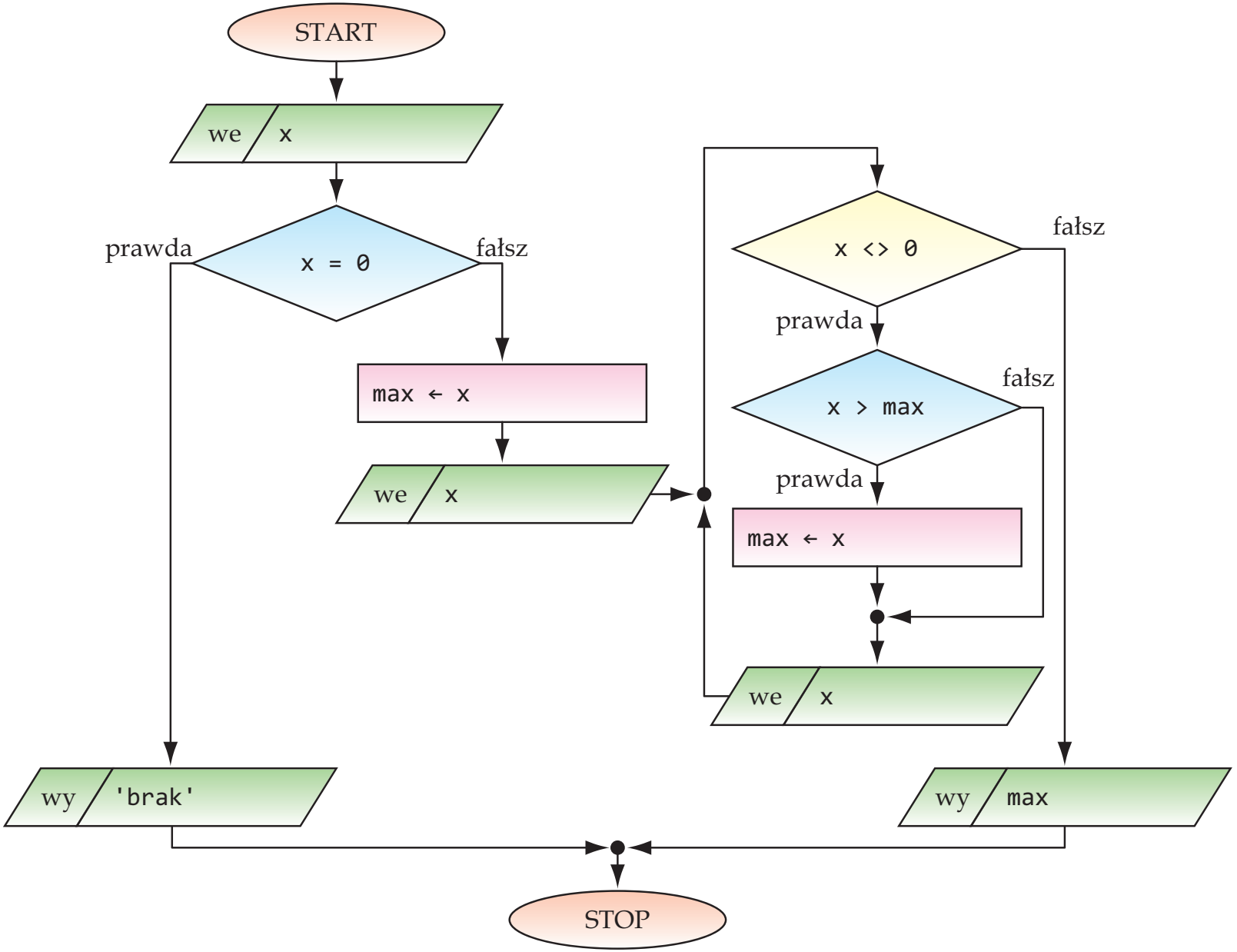
Algorytm obliczania pierwiastków równania kwadratowego



Algorytm sumowania



Algorytm szukania wartości ekstremalnej

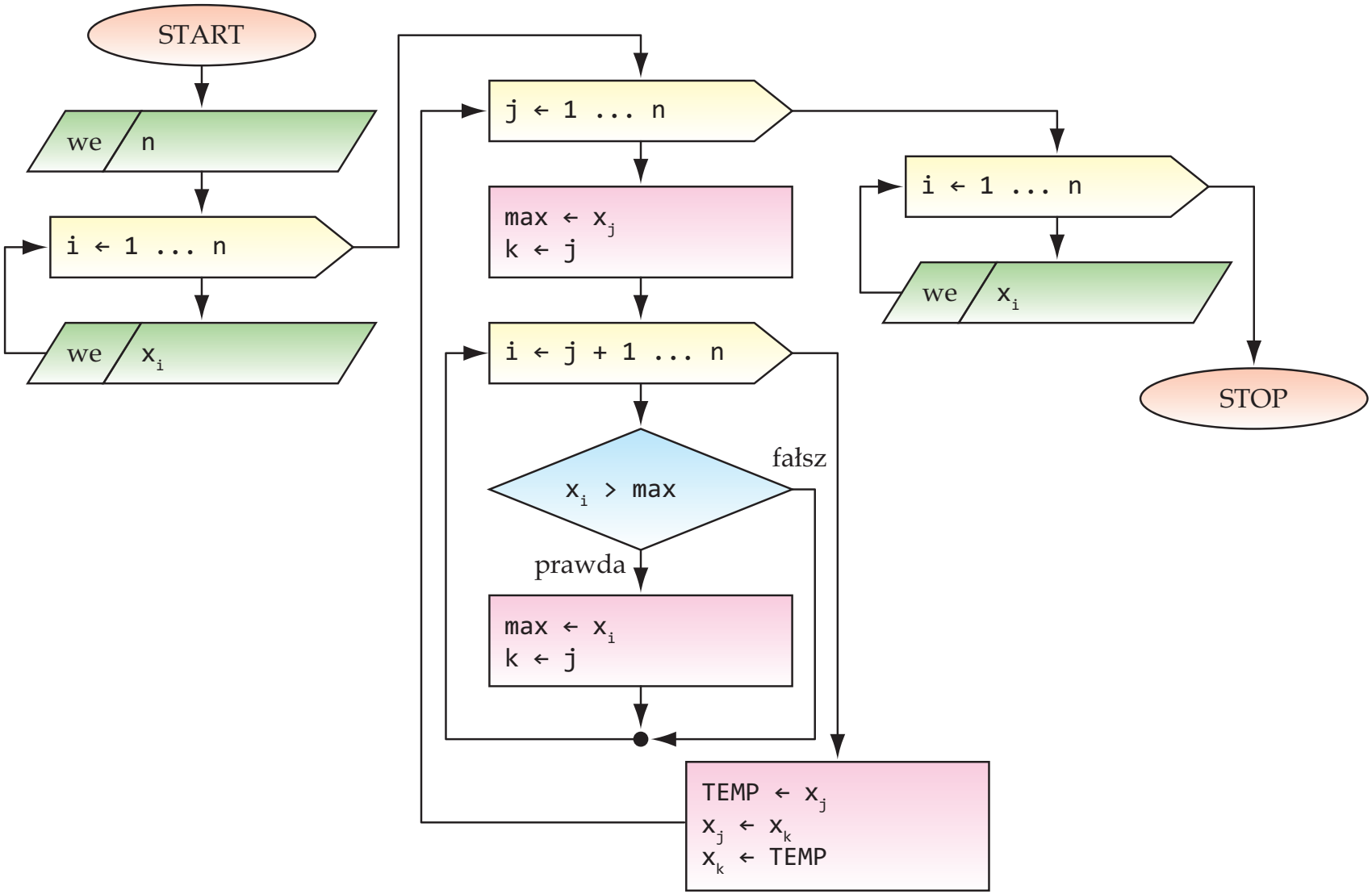


Algorytm sortowania przez wybór

Jest to chyba najbardziej intuicyjny algorytm sortowania. Polega on na wielokrotnym wyborze minimalnego elementu z coraz krótszego podciągu danych.

Przebieg:

- wybieranie minimum z ciągu elementów na pozycjach od 1 do n i zamienianie go z pierwszym elementem,
- wybieranie minimum z ciągu elementów na pozycjach od 2 do n i zamienianie go z drugim elementem (po tym kroku elementy na pozycjach od 1 do 2 są uporządkowane),
- wybieranie minimum z ciągu elementów na pozycjach $n-1$ i n i zamienianie z elementem na pozycji $n-1$ (po tej operacji elementy na pozycjach od 1 do $n-1$ są uporządkowane, a element na pozycji n jest maksymalny, czyli ciąg elementów na pozycjach od 1 do n jest uporządkowany).


START
we n
i ← 1 ... n
we x_i
j ← 1 ... n
max ← x_j
k ← j
i ← j + 1 ... n
x_i > max
fałsz
prawda
max ← x_i
k ← j
TEMP ← x_j
x_j ← x_k
x_k ← TEMP
i ← 1 ... n
we x_i
STOP

Podsumowanie

Budując algorytmy należy przestrzegać następujących zasad:

- algorytmy trzeba rozłożyć na jak najprostrze operacje podstawowe,
- każdy krok algorytmu dokładnie określić i przemyśleć,
- rozważyć wszystkie możliwe przypadki funkcjonowania algorytmu w zależności od danych wejściowych oraz operacji wewnętrznych,
- każdy algorytm powinien prowadzić do rozwiązania generując jedną lub wiele odpowiedzi na zadany problem.

Algorytm zdefiniowany z wykorzystaniem powyższych zasad powinien mieć jasno określony punkt startowy i punkt końcowy.

Metoda numeryczna

Jest to metoda, która umożliwia sprowadzenie najrozmaitszych obliczeń matematycznych, z dowolnej dziedziny ludzkiej działalności, do wykonywania skończonej liczby **najprostszych działań arytmetycznych** (czyli idealne do obliczeń komputerowych).

Skuteczne rozwiązywanie jakiegokolwiek problemu czy zadania, za pomocą odpowiedniej metody numerycznej, jest możliwe tylko wtedy, **gdy a priori wiadomo, że poszukiwane rozwiązanie istnieje i jest jedyne** (jednoznaczne).

Obliczenia numeryczne z reguły prowadzą do **wyników przybliżonych** i do nas należy ocena jakości rozwiązania, według przez nas wybranych kryteriów.

Przyczyny powstawania błędów w obliczeniach numerycznych:

- **błąd reprezentacji** – wynika ze skończonej reprezentacji liczby w komputerze,
- **błąd zaokrąglenia** – powstaje wtedy, gdy zapamiętanie dokładnego wyniku pewnego działania arytmetycznego wymaga więcej miejsca w pamięci niż jest to dla niego przeznaczone,
- **błąd redukcji** – występuje przy dodawaniu liczb różniących się znakami, o dokładnie tych samych cechach i niewiele różniących się mantysach,
- **błędy modelu** – wynikają z przybliżenia rzeczywistości przez pewne modele teoretyczne (numeryczne),
- **błędy obcięcia** – np.: obliczanego szeregu nieskończonego.

Z błędami redukcji wiąże się problem dokładności reprezentacji liczby za pomocą skończonej liczby cyfr.

Będziemy nazywać **liczbą cyfr znaczących** tę liczbę cyfr występujących z zapisie liczby, która pozostaje po odrzuceniu tzw. lewych zer (czyli występujących na lewo od pierwszej cyfry niezerowej w zapisie liczby).

$$0{,}00033333$$
$$0{,}33333$$
$$333{,}33$$
$$33333{,}$$

Każda z powyższych liczb ma liczbę cyfr znaczących równą LCZ $= 5$. LCZ decyduje o **dokładności (względnej) reprezentacji liczb**, a co za tym idzie od dokładności, z jaką wykonywane są operacje arytmetyczne przez konkretny komputer (czyli zależy od wielkości mantysy liczb zapamiętanych w komórkach jego pamięci).

Algorytmy standardowych metod numerycznych

Metody aproksymacji i interpolacji funkcji:

- metoda najmniejszych kwadratów,
- interpolacja Lagrange'a.

Całkowanie numeryczne:

- kwadratury Newtona–Cotesa (trapezów, Simpsona),
- złożone kwadratury całkowania,
- kwadratura Gaussa.

Metody rozwiązywania równań nieliniowych:

- metoda bisekcji (połowienia),
- metoda Newtona (stycznych),
- metoda siecznych.

Metody rozwiązywania układów równań liniowych:

- metody bezpośrednie (metoda eliminacji Gaussa, metoda Choleskiego),
- metody iteracyjne (metoda Jacobiego, metoda Gauss-Seidela).

Metody rozwiązywania problemów początkowych i brzegowych:

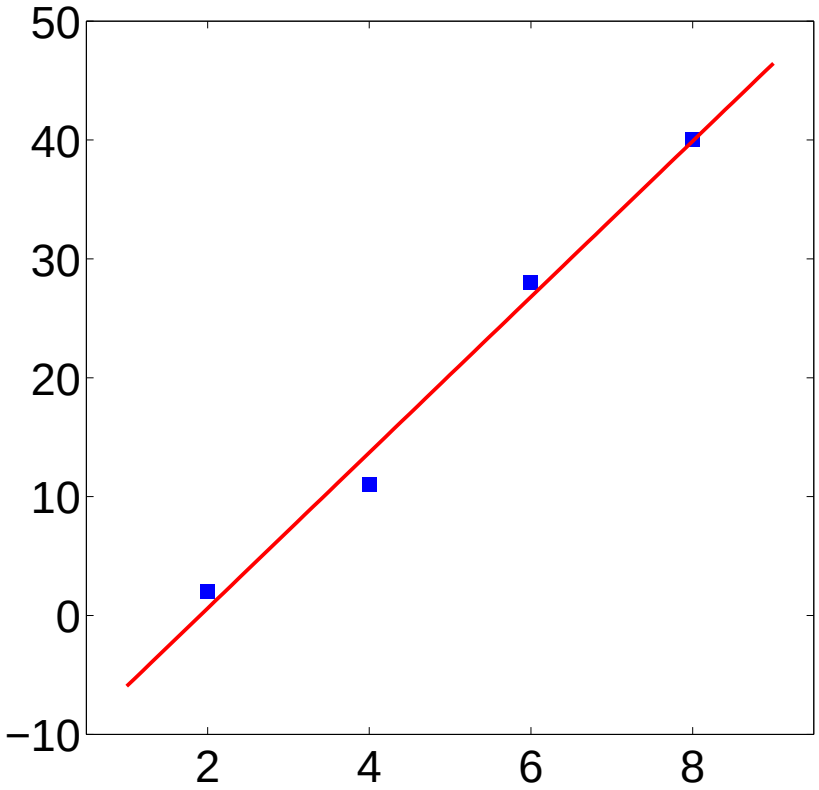
- metoda Eulera,
- metoda Rungego-Kutty,
- metoda różnic skończonych,
- metoda elementów skończonych,
- metoda elementów brzegowych.

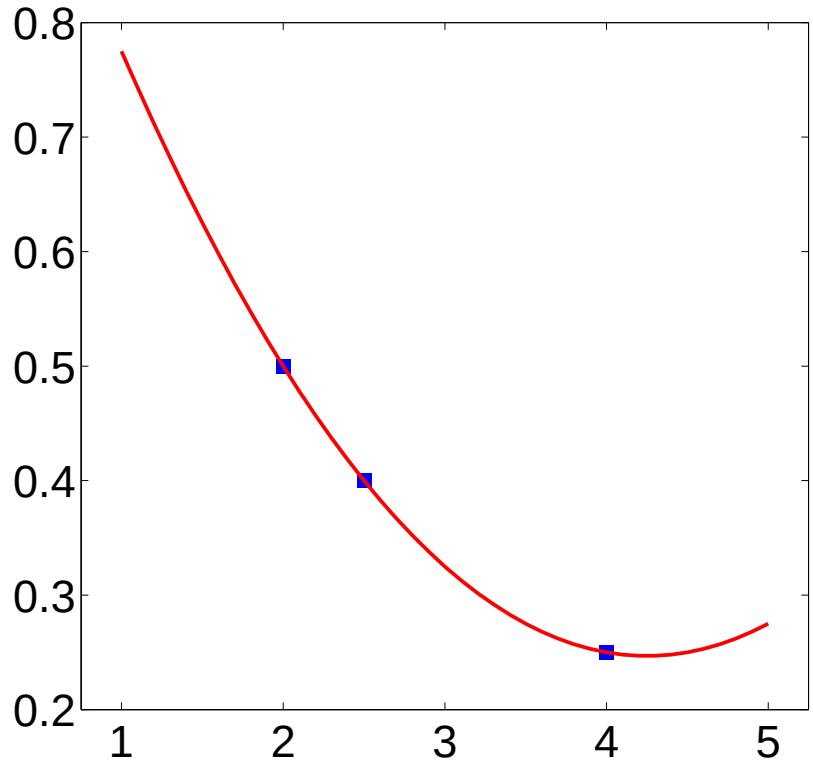
Aproksymacja i interpolacja

Aproksymacją nazywamy procedurę zastępowania jednej funkcji (funkcja aproksymowana) inną funkcją (funkcja aproksymująca) w taki sposób, aby funkcje te niewiele się różniły w sensie określonej normy.

Jeżeli dana funkcja określona jest w całym przedziale, w którym dokonujemy aproksymacji, mamy do czynienia z aproksymacją **funkcji ciągłej**, natomiast, gdy funkcja dana określona jest wyłącznie na skończonym zbiorze punktów zwanych **węzłami**, mówimy o aproksymacji **funkcji dyskretnej**. Jest to częsty przypadek w praktycznych zastosowaniach inżynierskich. Funkcja aproksymowana zwykle reprezentującą dane pomiarowe znane dla n różnych wartości zmiennej niezależnej.

Celem aproksymacji jest więc wyznaczenie funkcji (zwykle ciągłej), która będzie przebiegała w pobliżu danych punktów. W szczególnym przypadku funkcja aproksymująca może przebiegać przez niektóre z tych punktów. Jeżeli natomiast funkcja będzie przebiegać przez wszystkie podane punkty, będziemy mieć do czynienia z **interpolacją**.





Przyczyną stosowania aproksymacji mogą być między innymi:

- chęć zastąpienia funkcji niedogodnej do obliczeń numerycznych inną, dogodniejszą funkcją, która będzie niewiele odbiegać od funkcji wyjściowej (taką funkcją może być odpowiednio dobrany wielomian),
- potrzeba wyznaczenia wartości funkcji danej dyskretnie w innym punkcie obszaru,
- konieczność znalezienia dostatecznie gładkiej funkcji ciągłej przechodzącej w pobliżu zadanych punktów.

Dyskretna aproksymacja w bazie jednomianów

Rozwiązanie problemu aproksymacji danej dyskretnej funkcji $f(x_i)$, gdzie $i = 0, 1, \ldots, n$ wymaga przyjęcia bazy funkcyjnej aproksymacji (tutaj bazy jednomianów) i zapisaniu funkcji aproksymującej ($k = 0, 1, \ldots, m$)

$$P_m(x) = a_0 \cdot x^0 + a_1 \cdot x^1 + a_2 \cdot x^2 + \ldots + a_m \cdot x^m = \sum_{k=0}^{m} a_k \cdot x^k,$$

oraz na ustaleniu kryterium oceny jakości aproksymacji, które posłuży do jednoznacznego określenia wartości a_k, którego najprostsza postać jest następująca (metoda najmniejszych kwadratów):

$$r = \sum_{i=0}^{n} |f(x_i) - P_m(x_i)|^2.$$

Wartość funkcji r jest pewną miarą odchylenia funkcji aproksymującej $P_m(x)$ od aproksymowanej $f(x)$ (błąd aproksymacji). W przypadku aproksymacji $m < n$, wówczas $r > 0$ i osiąga minimum, gdy:

$$\frac{\partial r}{\partial a_k} = 0, \quad k = 0, 1, \ldots, m.$$

Warunków tych jest $m + 1$, a więc tyle ile niewiadomych współczynników wielomianu aproksymującego a_k.

Obliczając pochodne cząstkowe funkcji r otrzymamy

$$\frac{\partial}{\partial a_k} \left(\sum_{i=0}^{n} (f(x_i) - P_m(x_i))^2 \right) = 0,$$

$$2 \sum_{i=0}^{n} (f(x_i) - P_m(x_i)) \frac{\partial}{\partial a_k} P_m(x_i) = 0, \quad k = 0, 1, \ldots, m.$$

I po pewnych przekształceniach otrzymamy układ równań normalnych

$$a_0 \sum_{i=0}^{n} x_i^0 + a_1 \sum_{i=0}^{n} x_i^1 + a_2 \sum_{i=0}^{n} x_i^2 + \ldots + a_m \sum_{i=0}^{n} x_i^m = \sum_{i=0}^{n} f(x_i) x_i^0,$$

$$a_0 \sum_{i=0}^{n} x_i^1 + a_1 \sum_{i=0}^{n} x_i^2 + a_2 \sum_{i=0}^{n} x_i^3 + \ldots + a_m \sum_{i=0}^{n} x_i^{m+1} = \sum_{i=0}^{n} f(x_i) x_i^1,$$

$$\ldots$$

$$a_0 \sum_{i=0}^{n} x_i^m + a_1 \sum_{i=0}^{n} x_i^{m+1} + a_2 \sum_{i=0}^{n} x_i^{m+2} + \ldots + a_m \sum_{i=0}^{n} x_i^{2m} = \sum_{i=0}^{n} f(x_i) x_i^m.$$

Przykład

Obliczmy liniowy wielomian aproksymacyjny $P_1 = a_0 + a_1 \cdot x$ dla danych:

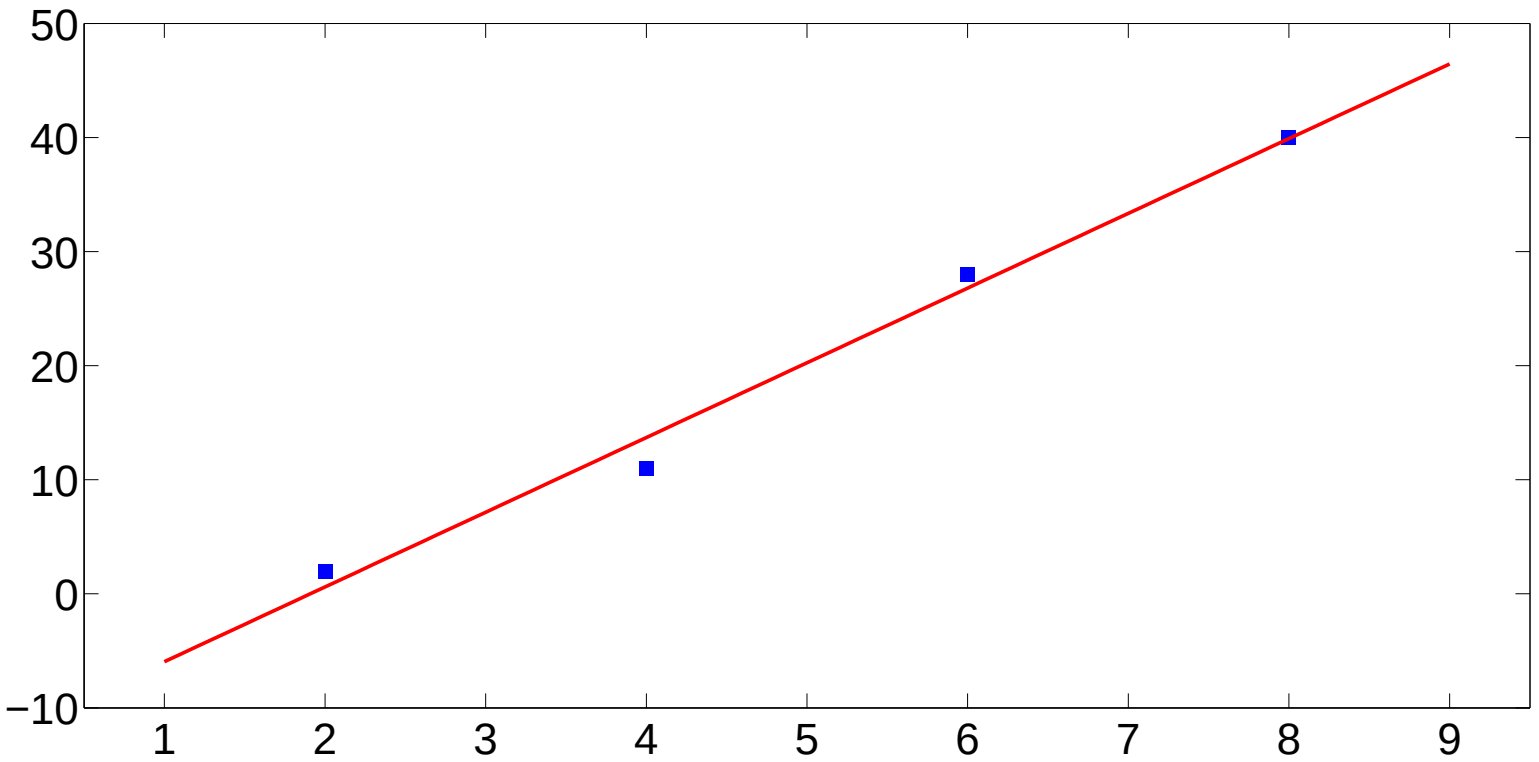
i	x_i	$f(x_i)$
0	2	2
1	4	11
2	6	28
3	8	40

W tym przypadku $n = 3$ i $m = 1$. Wykorzystując układ równań normalnych, możemy napisać

$$\begin{cases} 4a_0 + 20a_1 = 81 \\ 20a_0 + 120a_1 = 563 \end{cases}.$$

Rozwiązanie układu prowadzi do końcowego rozwiązania: $a_0 = -12,5$ i $a_1 = 6,55$. Wielomian aproksymacyjny wynosi

$$P_1(x) = -12,5 + 6,55x.$$



Interpolacja Lagrange'a

Interpolacja może być traktowana jako szczególny przypadek aproksymacji, polegający na tym, że funkcja aproksymowana i funkcja aproksymująca przyjmują te same wartości w wybranych punktach obszaru zwanych węzłami.

Funkcję interpolującą $P_n(x)$, najczęściej przedstawia się w postaci **wielomianu uogólnionego** utworzonego z odpowiednio dobranych funkcji bazowych

$$P_n(x) = a_0 \cdot \varphi_0(x) + a_1 \cdot \varphi_1(x) + a_2 \cdot \varphi_2(x) + \ldots + a_n \cdot \varphi_n(x) = \sum_{i=0}^{n} a_i \cdot \varphi_i(x).$$

Funkcje bazowe wielomianu interpolującego muszą być liniowo niezależne. Korzystając z warunku, że wartości funkcji interpolowanej i interpolującej mają takie same wartości, możliwe jest obliczenie współczynników funkcji interpolującej.

Najczęściej stosowaną i najprostszą bazą, jest baza funkcji jednomianowych

$$\varphi_n(x) = \left[x^0, x^1, x^2, \ldots, x^n\right],$$

i wtedy wielomian interpolacyjny będzie miał postać

$$P_n(x) = a_0 \cdot x^0 + a_1 \cdot x^1 + a_2 \cdot x^2 + \ldots + a_n \cdot x^n = \sum_{i=0}^{n} a_i \cdot x^i.$$

Wspomniana wcześniej procedura poszukiwania funkcji interpolującej jest w przypadku ogólnym dość złożona obliczeniowo: rozwiązywanie układu równań w celu określenia współczynników a_i; a współczynniki a_i zwykle nie mają interpretacji fizycznej.

Dlatego też funkcję interpolującą można zapisać w sposób

$$P_n(x) = f_0 \cdot N_{n,0}(x) + f_1 \cdot N_{n,1}(x) + f_2 \cdot N_{n,2}(x) + \ldots + f_n \cdot N_{n,n} = \sum_{i=0}^{n} f_i \cdot N_{n,i}(x),$$

z tak zwanymi funkcjami bazowymi Lagrange,a o postaci

$$N_{n,i}(x) = \prod_{\substack{j=0 \\ j \neq i}}^{n} \frac{x - x_j}{x_i - x_j} = \frac{(x - x_0)(x - x_1)\ldots(x - x_{i-1})(x - x_{x+1})\ldots(x - x_n)}{(x_i - x_0)(x_i - x_1)\ldots(x_i - x_{i-1})(x_i - x_{i+1})\ldots(x_i - x_n)}.$$

Warto zauważyć, że wartości współczynników, poprzednio oznaczonych przez a_i to teraz wartości funkcji interpolowanej f_i.

Przykład

Wyprowadźmy wzór interpolacyjny Lagrange'a stopnia drugiego przybliżający funkcję $f(x) = \frac{1}{x}$ i przyjmując węzły interpolacji: $x_0 = 2$, $x_1 = 2,5$ i $x_3 = 4$.

Wielomian będzie miał postać

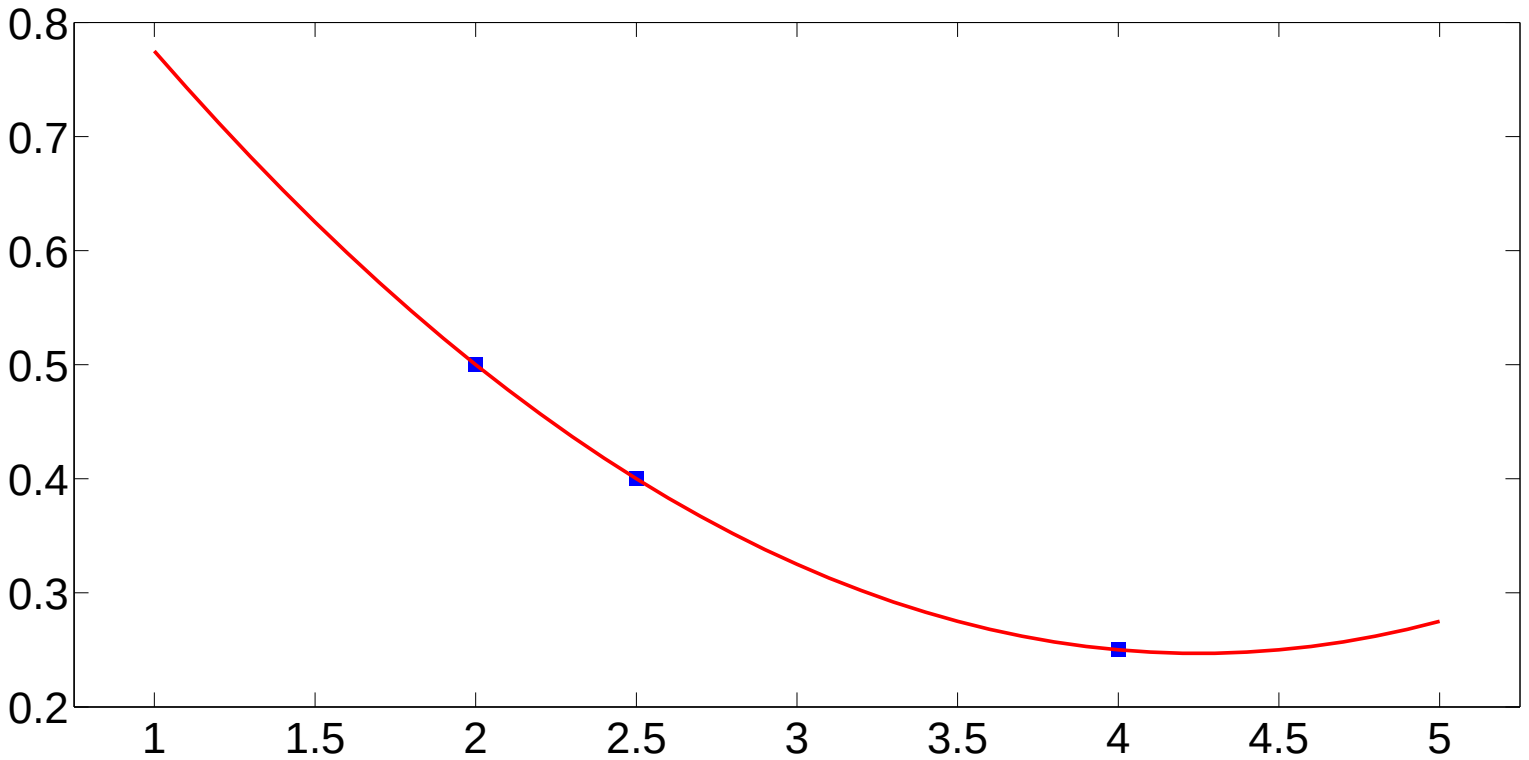
$$P_2(x) = f_0 \cdot N_{2,0}(x) + f_1 \cdot N_{2,1}(x) + f_2 \cdot N_{2,2}(x),$$

Wartości funkcji w węzłach interpolacji wynoszą odpowiednio: $f_0 = 0,5$, $f_1 = 0,4$ i $f_3 = 0,25$. Obliczmy funkcje bazowe

$$N_{2,0} = \frac{(x-2,5)(x-4)}{(2-2,5)(2-4)},$$
$$N_{2,1} = \frac{(x-2)(x-4)}{(2,5-2)(2,5-4)},$$
$$N_{2,2} = \frac{(x-2)(x-2,5)}{(4-2)(4-2,5)}.$$

Podstawienie funkcji bazowych i uproszczenie prowadzi do wielomianu interpolacyjnego Lagrange'a w postaci

$$P_2(x) = 0,5\frac{(x-2,5)(x-4)}{(2-2,5)(2-4)} + 0,4\frac{(x-2)(x-4)}{(2,5-2)(2,5-4)} + 0,25\frac{(x-2)(x-2,5)}{(4-2)(4-2,5)} = \\ = 1,15 - 0,425x + 0,05x^2.$$



W przypadkach elementarnych obliczenie wartości całki oznaczonej odbywa się na podstawie wzoru Newtona-Leibnitza

$$I(f) = \int_a^b f(x)\,\mathrm{d}x = F(b) - F(a).$$

Powyższy wzór można stosować wtedy, gdy znana jest tzw. **funkcja pierwotna** $F(x)$ spełniająca warunek

$$\frac{dF(x)}{dx} f(x).$$

Jeśli wyznaczenie funkcji pierwotnej jest bardzo trudne lub niemożliwe, albo też jeśli f(x) dane jest w postaci dyskretnej, to wartość całki można obliczyć tylko numerycznie stosując **wzory kwadraturowe** (kwadratury).

Idea postępowania przy całkowaniu numerycznym jest zawsze taka sama, a mianowicie zastępujemy funkcję podcałkową funkcją łatwą do scałkowania analitycznego (wielomian) w drodze interpolacji, a następnie funkcję interpolującą całkujemy ściśle (analitycznie).

W zależności od sposobu postępowania przy wyborze położeń węzłów interpolacji w przedziale całkowania możemy mieć do czynienia z kwadraturami z:

- **zamkniętymi** końcami (końce przedziału całkowania wchodzą do wzorów), a węzły są rozmieszczone równomiernie,
- **otwartymi** końcami (końce przedziału całkowania nie wchodzą do wzorów), a węzły są rozmieszczone w przedziale całkowania nierównomiernie.

Przykładem kwadratur pierwszego typu są kwadratury Newtona-Cotesa, a kwadratur drugiego typu kwadratury Gaussa.

Stosując funkcje bazowe interpolacji Lagrange'a

$$\int_a^b f(x)\,\mathrm{d}x \approx \int_a^b P_n(x)\,\mathrm{d}x = \int_a^b \sum_{i=0}^{n} f(x_i) N_{n,i}(x)\,\mathrm{d}x$$

wyznaczmy kilka kwadratur całkowania

- **kwadratura prostokątów** czyli interpolacja wielomianowa stopnia 0 ($n = 0$, funkcja stała)

$$\int_a^b f(x)\,\mathrm{d}x \approx \int_a^b f(x_0)\,\mathrm{d}x = (b-a) f(x_0),$$

- **kwadratura trapezów** czyli interpolacja wielomianowa stopnia 1 ($n = 1$, funkcja liniowa)

$$\int_a^b f(x)\,\mathrm{d}x \approx \int_a^b \left(f(x_0)\frac{x-x_1}{x_0-x_1} + f(x_1)\frac{x-x_0}{x_1-x_0} \right)\,\mathrm{d}x = \frac{b-a}{2}(f(x_0)+f(x_1)),$$

- **kwadratura Simpsona** czyli interpolacja wielomianowa stopnia 2 ($n = 2$, funkcja kwadratowa)

$$\int_a^b f(x)\,\mathrm{d}x \approx \int_a^b \left(f(x_0)\frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)} + f(x_1)\frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)} + \right.$$
$$\left. + f(x_1)\frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)} \right) \mathrm{d}x = \frac{b-a}{3}\left(f(x_0) + 4f(x_1) + f(x_2)\right).$$

Przykład

Policzmy całkę

$$I = \int_0^{\frac{1}{2}} \sqrt{1+x}\,\mathrm{d}x = \frac{2}{3}\left(\left(\frac{3}{2}\right)^{\frac{3}{2}} - 1\right) = \frac{1}{2}\sqrt{6} - \frac{2}{3} = 0,5580782.$$

Policzmy przykład jeszcze raz korzystając ze wzorów przybliżonych:

- wzór trapezów

$$I \approx \frac{1}{2}\cdot\frac{1}{2}\left((1+0)^{\frac{1}{2}} + \left(1+\frac{1}{2}\right)^{\frac{1}{2}}\right) = 0,5561862,$$

- wzór Simpsona

$$I \approx \frac{1}{3}\cdot\frac{1}{4}\left((1+0)^{\frac{1}{2}} + 4\left(1+\frac{1}{4}\right)^{\frac{1}{2}} + \left(1+\frac{1}{2}\right)^{\frac{1}{2}}\right) = 0,5580734.$$

Wady kwadratur Newtona-Cotesa:

- duże przedziały całkowania wymagają interpolacji wysokiego stopnia co utrudnia obliczanie współczynników interpolacji,
- wielomiany interpolacyjne wysokiego stopnia z równo rozmieszczonymi węzłami mają wysoki stopień oscylacji na brzegach.

Wyjściem z tej sytuacji są **kwadratury złożone**. Kwadratury złożone otrzymujemy dzieląc przedział całkowania na pewną liczbę równych pod przedziałów i stosując dla każdego z nich tę samą kwadraturę.

Do podwyższenia dokładności wzorów kwadraturowych można zastosować propozycję **Gaussa**, polegającą na optymalizacji położenia węzłów interpolacyjnych, rezygnacji z warunku równomiernego rozmieszczenia węzłów interpolacji oraz doborze odpowiednich wartości współczynników wagowych.

We wzorach kwadraturowych Gaussa wykorzystuje się aproksymację wielomianami Legendre'a. Wartość przybliżoną całki można policzyć korzystając z zależności

$$\int_a^b f(x)\,\mathrm{d}x \approx \sum_{i=1}^{n} w_i \cdot f(x_i),$$

gdzie w_i to współczynniki wagowe, a x_i to współrzędne węzłów interpolacji.

Wygodnie jest korzystać z wartości tablicowych współczynników wagowych i współrzędnych węzłów interpolacji, które można znaleźć w każdym podręczniku dotyczącym metod numerycznych.

Rozwiązywanie równań nieliniowych

Kolejnym prostym zastosowaniem metod numerycznych jest rozwiązywanie nieliniowego równania (algebraicznego, trygonometrycznego) o ogólnej postaci

$$f(x) = 0.$$

Przyjmiemy, ze znany jest **przedział izolacji**, w którym znajduje się szukany pierwiastek i jest tylko jeden (w badanym przedziale).

Pierwiastek będziemy liczyć **metodą iteracyjną**, polegającą na powtarzaniu pewnego ustalonego ciągu operacji arytmetycznych. Wymaga to przyjęcia odpowiedniego **przybliżenia początkowego** (punktu startowego).

Obliczenia uznamy za zakończone jeśli osiągniemy odpowiednie kryterium. Mogą to być kryteria związane z **liczbą iteracji** (niezależnie od znalezienia rozwiązania), z tempem zbieżności (zmiana kolejnych przybliżeń rozwiązania) lub z **residuum** (zmiana kolejnych wartości funkcji dla kolejnych przybliżeń rozwiązania).

Metoda bisekcji

Metoda bisekcji należy do najstarszych i naiprostszych metod iteracyjnych, oprócz znajdowania pierwiastków równania jest wykorzystywana przy zagadnieniach optymalizacji funkcji.

Dla wyjściowego równania $f(x) = 0$ szuka przybliżenia w przedziale $x \in (a, b)$. Stąd, aby metoda zadziałała, musi być gwarancja istnienia miejsca zerowego w tym przedziale: $f(a) \cdot f(b) < 0$.

Przy każdej iteracji oblicza się środek przedziału $x = \frac{a+b}{2}$. Następnie sprawdza się, w którym z podprzedziałów (a, x) oraz (x, b) leży miejsce zerowe i ten podprzedział podlega dalszemu dzieleniu.

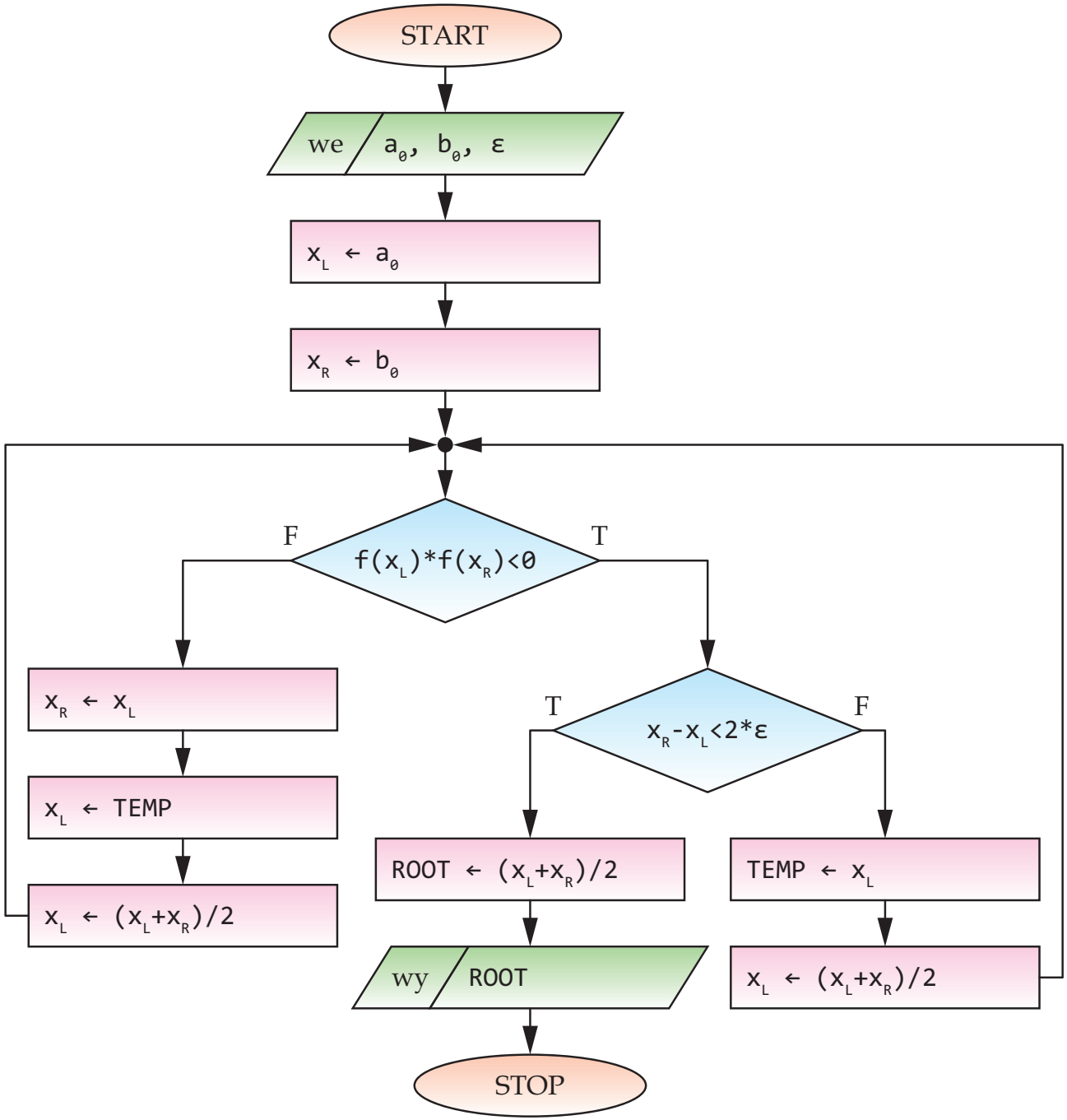
Podział przedziału (a, b) niekoniecznie musi odbywać się na dwie równe części.

Przykład

Rozwiążmy równanie $sin(x) + x^2 = 2$, przyjmując przedział początkowy $(1, 3)$.
Rozważane równanie ma pierwiastki $x_1^{sc} = -1,06155$ i $x_2^{sc} = 1,728466$.

Równanie wyjściowe zapiszemy w postaci: $f(x) = sin(x) + x^2 - 2$, $f(x) = 0$,
a końce przedziałów: $a_0 = 1$ i $b_0 = 3$.

i	$x_i = \frac{a_{i-1}+b_{i-1}}{2}$	$f(x_i) \cdot f(a_{n-1})$	a_i	b_i	$\varepsilon_i = \left\lvert \frac{x_i - x_{i-1}}{x_i} \right\rvert$	$\delta_i = \lvert f(x_i) \rvert$
1	2,000	-2,008497	1,000	2,000	0,500	1,090703
2	1,500	1,376490	1,500	2,000	0,333333	0,747495
...	...	...	...	...	...	...
11	1,727539	0,000023	1,727539	1,728516	0,000565	0,003350


START
we
a_0, b_0, ε
$x_L \leftarrow a_0$
$x_R \leftarrow b_0$
F
$f(x_L)*f(x_R)<0$
T
$x_R \leftarrow x_L$
$x_L \leftarrow$ TEMP
$x_L \leftarrow (x_L+x_R)/2$
T
$x_R-x_L<2*\varepsilon$
F
ROOT $\leftarrow (x_L+x_R)/2$
wy
ROOT
STOP
TEMP $\leftarrow x_L$
$x_L \leftarrow (x_L+x_R)/2$

Metoda stycznych (Newtona-Raphsona)

Dla pewnego otoczenia h punktu x rozwinięcie wartości wyjściowej funkcji $f(x+h)$ w szereg Taylora będzie wynosiło

$$f(x+h) = f(x) + f'(x) \cdot h + \frac{1}{2} f''(x) \cdot h^2 + \frac{1}{6} f'''(x) \cdot h^3 + \ldots \approx f(x) + f'(x) \cdot h$$

Po odrzuceniu wyrazów rozwinięcia wyższych rzędem niż pierwszy, ustalając x i podstawiając $f(x+h) = 0$ można obliczyć przyrost h jako

$$h = -\frac{f(x)}{f'(x)}.$$

Dla danej pary sąsiednich przybliżeń zachodzi: $x_{i+1} = x_i + h$, stąd otrzymujemy schemat metody

$$\begin{cases} x_0 \\ x_{i+1} = x_i - \dfrac{f(x)}{f'(x)} \end{cases}.$$

Graficznie metoda polega na budowaniu stycznych w kolejnych przybliżeniach x_i począwszy od punktu startowego oraz na szukaniu miejsc zerowych tych stycznych.

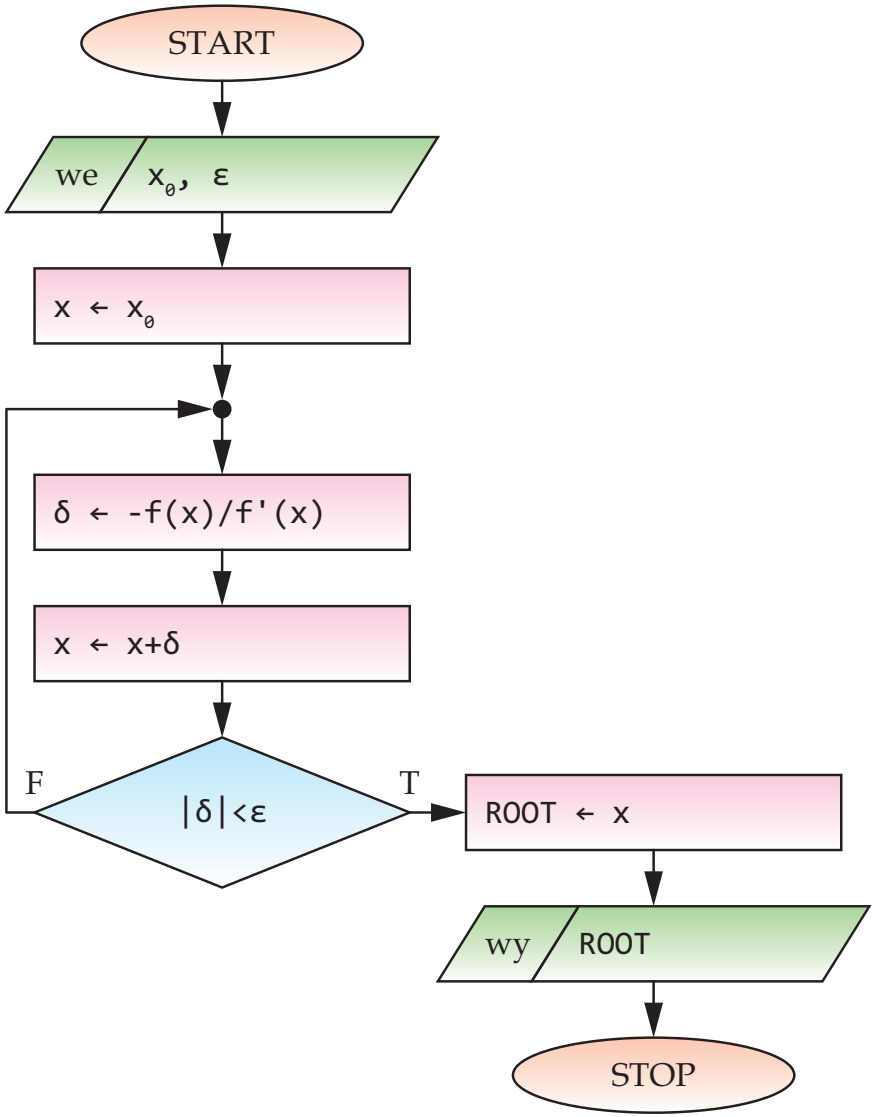
Przykład

Rozwiążmy równanie $sin(x) + x^2 = 2$, przyjmując punkt startowy $x_0 = -2$. Rozważane równanie ma pierwiastki $x_1^{sc} = -1,06155$ i $x_2^{sc} = 1,728466$.

Równanie wyjściowe zapiszemy w postaci: $f(x) = sin(x) + x^2 - 2$, $f(x) = 0$. Pochodna będzie równa: $f'(x) = cos(x) + 2x$. Schemat iteracyjny będzie miał postać:

$$\begin{cases} x_0 = -2 \\ x_{i+1} = x_i - \dfrac{sin(x_i) + x_i^2 - 1}{cos(x_i) + 2x_i} \end{cases}.$$

i	x_i	$\varepsilon_i = \left\lvert \frac{x_i - x_{i-1}}{x_i} \right\rvert$	$\delta_i = \left\lvert \frac{f(x_i)}{f(x_{i-1})} \right\rvert$
1	-1,188221	0,683189	0,116721
2	-1,064728	0,115985	0,002854
...	...	...	...
4	-1.061550	$< 10^{-6}$	$< 10^{-8}$


START
we
x_0, ε
x ← x_0
δ ← -f(x)/f'(x)
x ← x+δ
F
|δ|<ε
T
ROOT ← x
wy
ROOT
STOP

Metoda siecznych

W metodzie stycznych do schematu iteracyjnego potrzebna jest znajomość pochodnej rozpatrywanej funkcji. Aby uniknąć jej różniczkowania, można liczbową pochodną obliczać w sposób przybliżony korzystając z wartości ilorazu różnicowego

$$f'(x_i) \approx \frac{f(x_{i-1}) - f(x_i)}{x_{i-1} - x_i}$$

i wymaga dwóch punktów startowych. Prowadzi to do schematu iteracyjnego metody

$$\begin{cases} x_0, \ x_1 \\ x_{i+1} = x_i - f(x_i)\dfrac{x_i - x_{i-1}}{f(x_i) - f(x_{i-1})} \end{cases}.$$

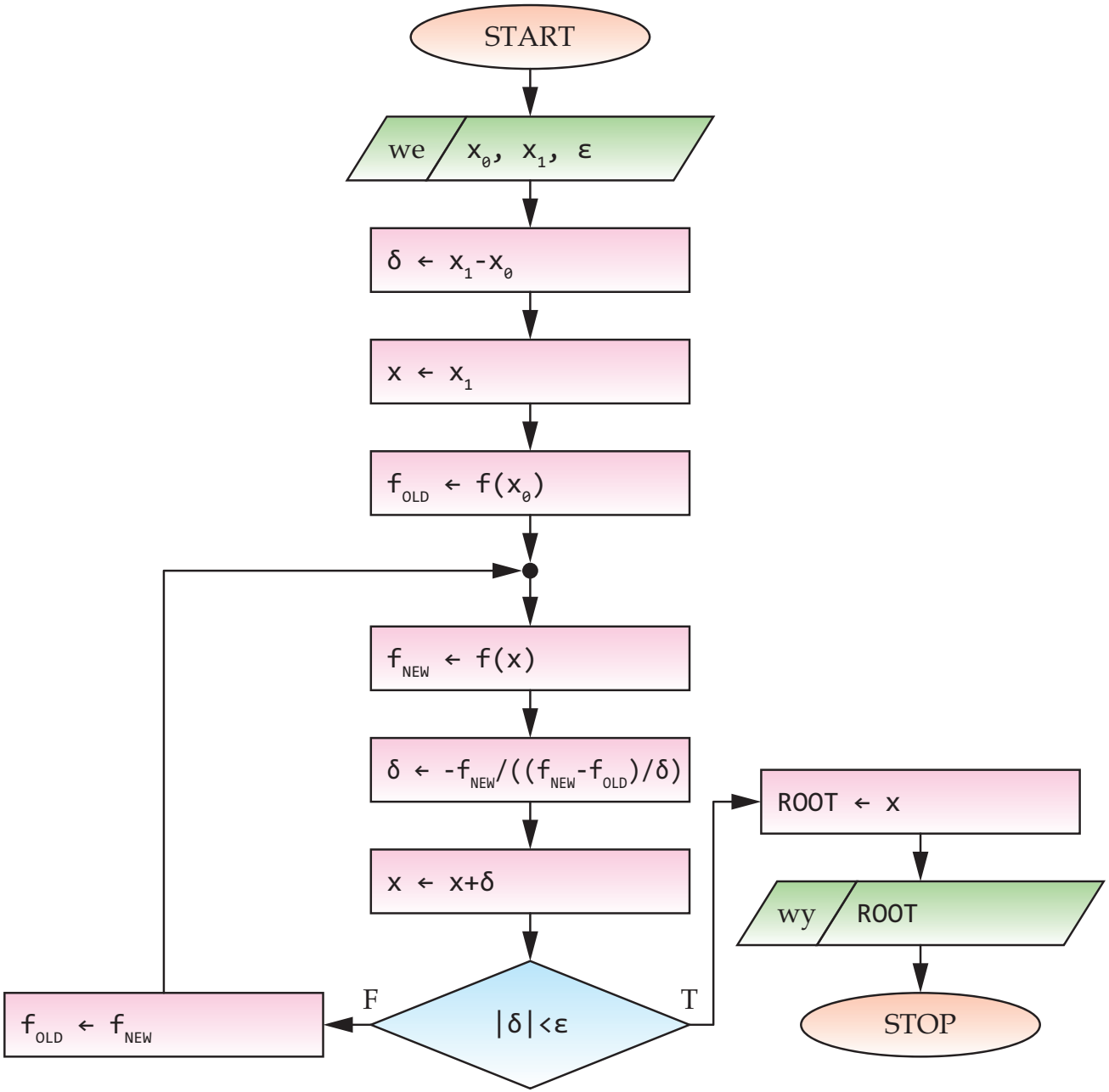
Przykład

Rozwiążmy równanie $sin(x) + x^2 = 2$, przyjmując punkty startowe $x_0 = -2$ i $x_1 = -0,5$. Rozważane równanie ma pierwiastki $x_1^{sc} = -1,06155$ i $x_2^{sc} = 1,728466$.

Równanie wyjściowe zapiszemy w postaci: $f(x) = sin(x) + x^2 - 2$, $f(x) = 0$. Schemat iteracyjny będzie miał postać:

$$\begin{cases} x_0 = -2, \ x_1 = -0,5 \\ x_{i+1} = x_i - (sin(x_i) + x_i^2 - 2)\dfrac{x_{i-1} - x_i}{sin(x_{i-1}) + x_{i-1}^2 - sin(x_n) - x_i^2} \end{cases}.$$

i	x_i	$\varepsilon_i = \left\lvert \frac{x_i - x_{i-1}}{x_i} \right\rvert$	$\delta_i = \left\lvert \frac{f(x_i)}{f(x_{i-1})} \right\rvert$
1	-0,955962	0,476967	0,092554
2	-1,078578	0,113683	0,015336
...	...	...	...
5	-1.061550	$< 10^{-6}$	$< 10^{-8}$


START
we
x_0, x_1, ε
δ ← x_1-x_0
x ← x_1
f_{OLD} ← f(x_0)
f_{NEW} ← f(x)
δ ← -f_{NEW}/((f_{NEW}-f_{OLD})/δ)
x ← x+δ
|δ|<ε
F
T
f_{OLD} ← f_{NEW}
ROOT ← x
wy
ROOT
STOP

Układ liniowych równań algebraicznych

Z liniowym (oznaczonym) układem równań algebraicznych mamy do czynienia w sytuacji, gdy wszystkie zmienne występujące w równaniach układu występują jedynie w pierwszej potędze. Można go skrótowo zapisać w następującej postaci

$$\sum_{l=1}^{n} c_{kl} \cdot x_l = r_k, \quad k = 1, 2, \ldots, m,$$

w którym oznaczono odpowiednio:

- c_{kl} – wartości współczynników stojących przy niewiadomych (elementy macierzy głównej),
- x_l – symbole niewiadomych (elementy wektora niewiadomych),
- r_l – wartości prawej strony (elementy wektora prawej strony).

Dla układu równań liniowych, warunek liniowej niezależności jest równoważny żądaniu by macierz główna układu $\mathbf{C}$, o elementach c_{kl}, miała wyznacznik różny od zera ($\det(\mathbf{C}) \neq 0$). W takiej sytuacji układ równań ma jedno i tylko jedno rozwiązanie. Podstawowe metody jego znajdowania można podzielić na dwie kategorie:

- metody eliminacyjne (np. Gaussa),
- metody iteracyjne (np. Gaussa-Seidla).

Do zalet metod eliminacyjnych należą między innymi łatwość rozwiązywania układu równań z wieloma prawymi stronami (sytuacja często występująca w praktyce), możliwość precyzyjnego oszacowania czasu obliczeń na podstawie rozmiaru zadania czy wreszcie gwarancja uzyskania wyniku po wykonaniu możliwej do określenia z góry liczby operacji.

Zaletą metod iteracyjnych jest prostota algorytmu i niewielkie zapotrzebowanie na pamięć operacyjną w trakcie obliczeń.

Metoda eliminacji Gaussa

Układ równań przekształcamy tak, aby macierz główną układu doprowadzić do postaci trójkątnej górnej (górnotrójkątnej).

Przekształcenia prowadzimy korzystając z faktu, że:

- przestawienie dwóch wierszy (równań),
- przemnożenie wszystkich elementów wiersza (współczynników równania) przez tę samą liczbę,
- dodanie do elementów jednego wiersza (współczynników równania) odpowiednio elementów innego wiersza,
- dodanie do elementów jednego wiersza odpowiednio kombinacji liniowej elementów innych wierszy,

przekształcają tablicę równoważnie, czyli w sposób nie zmieniający rozwiązania układu równań.

Obliczenia rozpoczynamy od pierwszej niewiadomej pierwszego równania

$$\begin{cases} c_{11} \cdot x_1 + c_{12} \cdot x_2 = r_1 \\ c_{21} \cdot x_1 + c_{22} \cdot x_2 = r_2 \end{cases},$$

naszym celem jest takie przekształcenie równania, aby współczynnik c_{11} miał wartość 1, a współczynnik c_{21} miał wartość 0. W tym celu podzielimy pierwsze równanie przez c_{11} (element wiodący)

$$\begin{cases} 1 \cdot x_1 + c_{12}/c_{11} \cdot x_2 = r_1/c_{11} \\ c_{21} \cdot x_1 + c_{22} \cdot x_2 = r_2 \end{cases},$$

następnie, od równania drugiego odejmujemy równanie pierwsze pomnożone przez c_{21}

$$\begin{cases} 1 \cdot x_1 + c_{12}/c_{11} \cdot x_2 = r_1/c_{11} \\ 0 \cdot x_1 + (c_{22} - c_{12}/c_{11} \cdot c_{21}) \cdot x_2 = r_2 - r_1/c_{11} \cdot c_{21} \end{cases},$$

w kolejnym kroku dzielimy drugie (kolejne) równanie przez $(c_{22} - c_{12}/c_{11} \cdot c_{21})$ (element wiodący)

$$\begin{cases} 1 \cdot x_1 + c_{12}/c_{11} \cdot x_2 = r_1/c_{11} \\ 0 \cdot x_1 + 1 \cdot x_2 = (r_2 - r_1/c_{11} \cdot c_{21})/(c_{22} - c_{12}/c_{11} \cdot c_{21}) \end{cases}.$$

W kolejnych krokach wyznaczamy wartości niewiadomych.

Przykład

Rozwiążmy metodą eliminacji Gaussa układ równań

$$\begin{cases} 2x_1 + 1x_2 - 2x_3 = 0 \\ -1x_1 - 3x_2 + 4x_3 = 3 \\ 3x_1 + 2x_2 + 1x_3 = 4 \end{cases}.$$

W pierwszym kroku, pierwsze równanie podzielimy przez 2 (element wiodący)

$$\begin{cases} 1x_1 + 0,5x_2 - 1x_3 = 0 \\ -1x_1 - 3x_2 + 4x_3 = 3 \\ 3x_1 + 2x_2 + 1x_3 = 4 \end{cases},$$

następnie, od równania 2 odejmiemy równanie 1 pomnożone przez -1

$$\begin{cases} 1x_1 + 0,5x_2 - 1x_3 = 0 \\ 0x_1 - 2,5x_2 + 3x_3 = 3 \\ 3x_1 + 2x_2 + 1x_3 = 4 \end{cases},$$

w kolejnym kroku, od równania 3 odejmiemy równanie 1 pomnożone przez 3

$$\begin{cases} 1x_1 + 0,5x_2 - 1x_3 = 0 \\ 0x_1 - 2,5x_2 + 3x_3 = 3 \\ 0x_1 + 0,5x_2 + 4x_3 = 4 \end{cases}.$$

Drugie równanie dzielimy przez 2,5 (element wiodący)

$$\begin{cases} 1x_1 + 0,5x_2 - 1x_3 = 0 \\ 0x_1 + 1x_2 - 1,2x_3 = -1,2 \\ 0x_1 + 0,5x_2 + 4x_3 = 4 \end{cases},$$

następnie, od równania 3 odejmiemy równanie 2 pomnożone przez 0,5

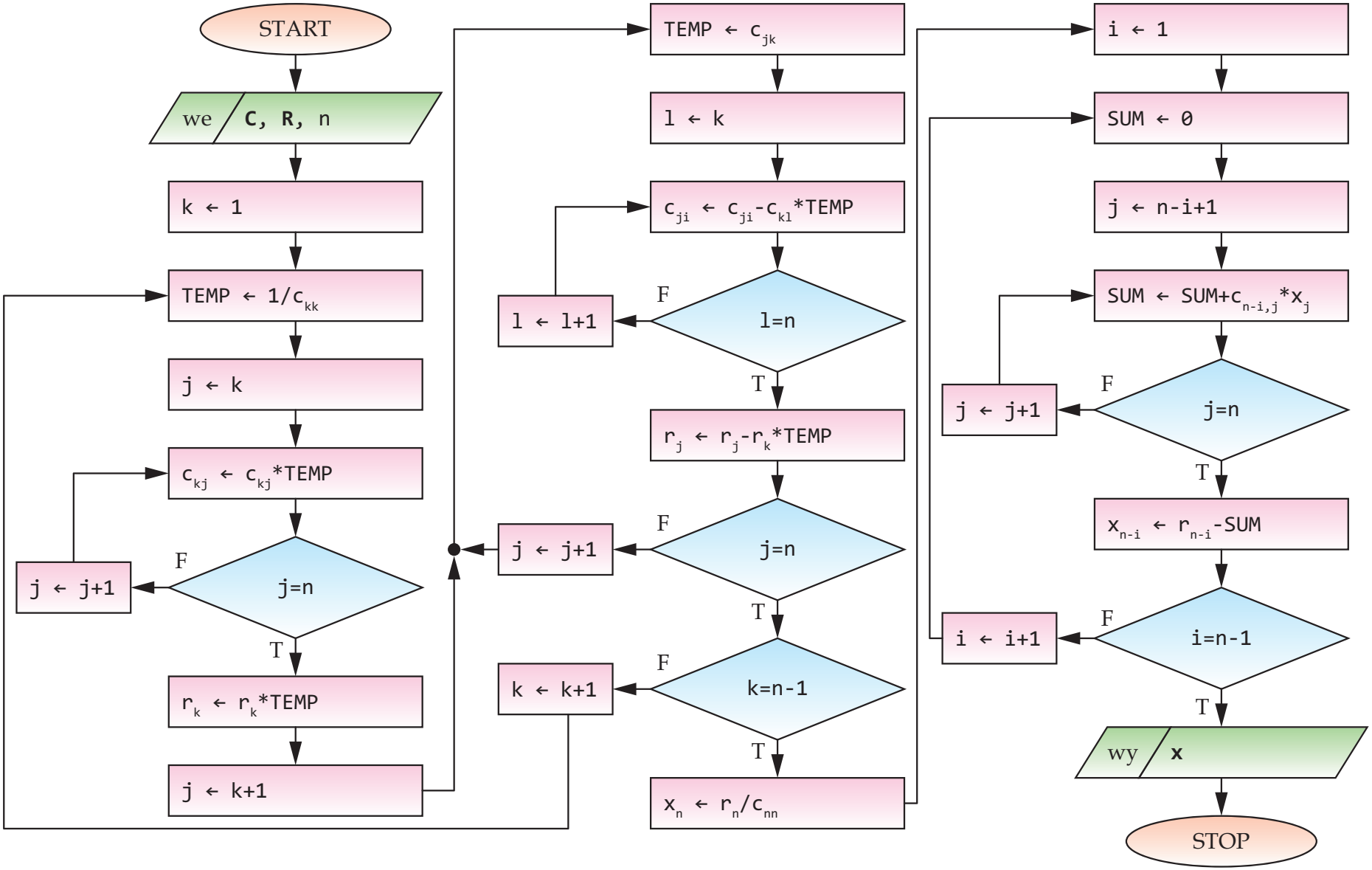
$$\begin{cases} 1x_1 + 0,5x_2 - 1x_3 = 0 \\ 0x_1 + 1x_2 - 1,2x_3 = -1,2 \\ 0x_1 + 0x_2 + 4,6x_3 = 4,6 \end{cases},$$

ostatnim krokiem będzie podzielenie równania 3 przez 4,6

$$\begin{cases} 1x_1 + 0,5x_2 - 1x_3 = 0 \\ 0x_1 + 1x_2 - 1,2x_3 = -1,2 \\ 0x_1 + 0x_2 + 1x_3 = 1 \end{cases}.$$

Z powyższego układu równań, poczynając od ostatniego będziemy mogli wyznaczyć kolejne niewiadome

$$\begin{cases} 1x_3 = 1 & \Rightarrow & x_3 = 1 \\ 1x_2 - 1,2x_3 = -1,2 & \Rightarrow & x_2 = 0 \\ 1x_1 + 0,5x_2 - 1x_3 = 0 & \Rightarrow & x_1 = 1 \end{cases}.$$


START
we C, R, n
k ← 1
TEMP ← 1/c_kk
j ← k
c_kj ← c_kj*TEMP
F
j=n
j ← j+1
T
r_k ← r_k*TEMP
j ← k+1
TEMP ← c_jk
l ← k
c_ji ← c_ji-c_kl*TEMP
F
l=n
l ← l+1
T
r_j ← r_j-r_k*TEMP
F
j=n
j ← j+1
T
F
k=n-1
k ← k+1
T
x_n ← r_n/c_nn
i ← 1
SUM ← 0
j ← n-i+1
SUM ← SUM+c_n-i,j*x_j
F
j=n
j ← j+1
T
x_n-i ← r_n-i-SUM
F
i=n-1
i ← i+1
T
wy x
STOP

Metoda Gaussa-Seidla

W tej metodzie układ równań

$$\begin{cases} c_{11} \cdot x_1 + c_{12} \cdot x_2 = r_1 \\ c_{21} \cdot x_1 + c_{22} \cdot x_2 = r_2 \end{cases},$$

przekształcamy tak, aby z kolejnych równań móc obliczyć kolejne niewiadome

$$\begin{cases} x_1 = (r_1 - c_{12} \cdot x_2)/c_{11} \\ x_2 = (r_2 - c_{21} \cdot x_1)/c_{22} \end{cases}.$$

Obliczenia rozpoczynamy od założenia początkowych wartości niewiadomych $x_{1\{0\}}$ i $x_{2\{0\}}$. W pierwszej iteracji wyznaczamy nowe przybliżenie szukanych wartości niewiadomych, wykorzystując ich ostatnie dostępne wartości

$$\begin{cases} x_{1\{1\}} = (r_1 - c_{12} \cdot x_{2\{0\}})/c_{11} \\ x_{2\{1\}} = (r_2 - c_{21} \cdot x_{1\{1\}})/c_{22} \end{cases}.$$

Po pierwszej iteracji otrzymujemy przybliżone wartości rozwiązania $x_{1\{1\}}$ i $x_{2\{1\}}$. Powtarzamy procedurę iteracji

$$\begin{cases} x_{1\{2\}} = (r_1 - c_{12} \cdot x_{2\{1\}})/c_{11} \\ x_{2\{2\}} = (r_2 - c_{21} \cdot x_{1\{2\}})/c_{22} \end{cases},$$

i kolejno

$$\begin{cases} x_{1\{i+1\}} = (r_1 - c_{12} \cdot x_{2\{i\}})/c_{11} \\ x_{2\{i+1\}} = (r_2 - c_{21} \cdot x_{1\{i+1\}})/c_{22} \end{cases},$$

aż do otrzymania wyników spełniających zakładaną dokładność, zwykle określaną jako dopuszczalną, względną wielkość zmiany wyników pomiędzy kolejnymi iteracjami.

Przykład

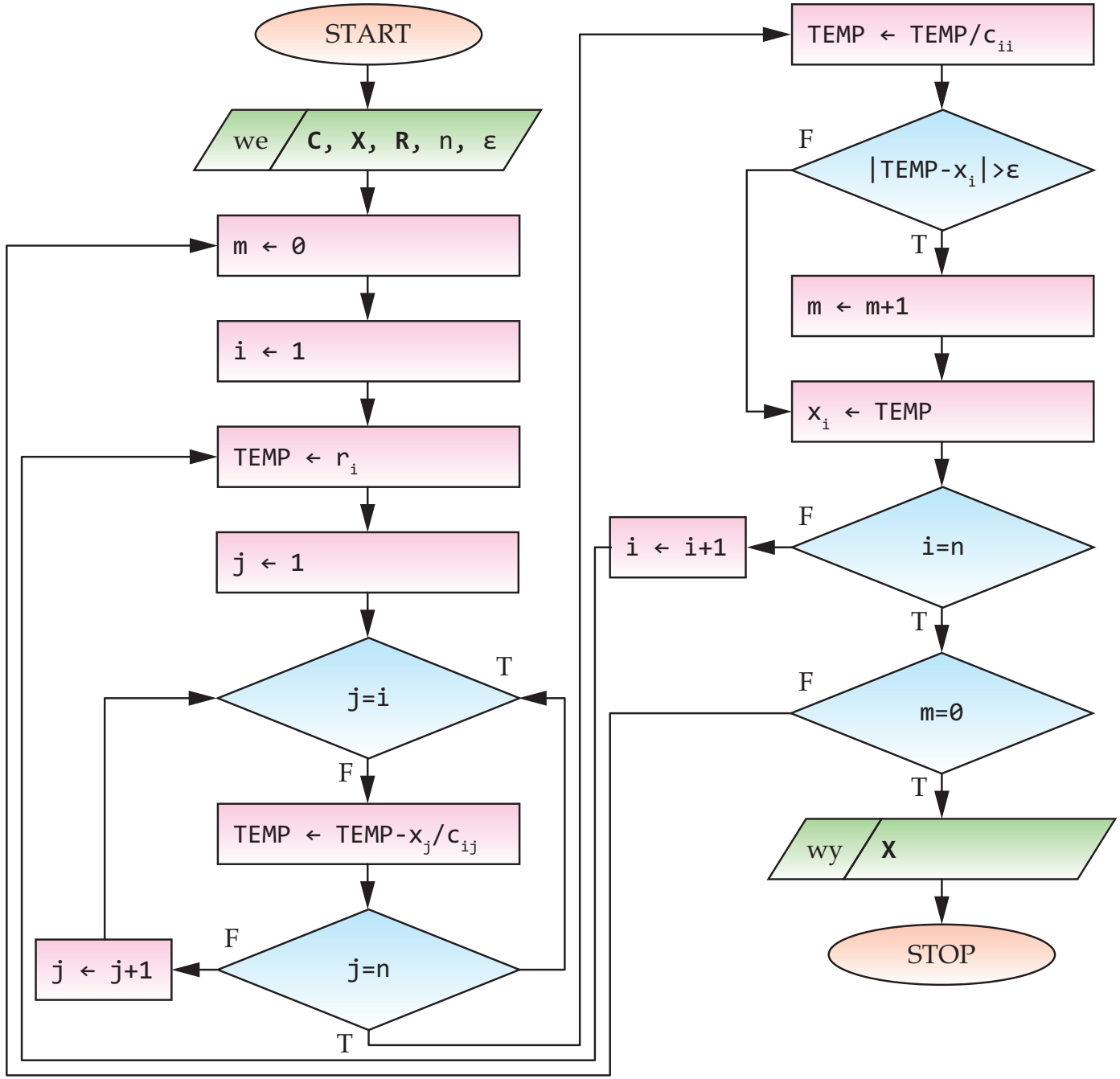
Rozwiążmy metodą iteracyjną Gaussa-Seidla układ równań,

$$\begin{cases} 3x_1 + 1x_2 - 1x_3 = 2 \\ 1x_1 + 4x_2 + 1x_3 = 12 \\ 2x_1 + 1x_2 + 2x_3 = 10 \end{cases},$$

przyjmując wartości startowe: $x_{1\{0\}} = 1$, $x_{2\{0\}} = 1$ i $x_{3\{0\}} = 1$. Przekształcony układ równań:

$$\begin{cases} x_1 = (2 - 1x_2 + 1x_3)/3 \\ x_2 = (10 - 1x_1 - 1x_3)/4 \\ x_3 = (12 - 2x_1 - 1x_2)/2 \end{cases}.$$

i	$x_{1\{i\}}$	$x_{2\{i\}}$	$x_{3\{i\}}$	$\varepsilon_1^{\{i\}} = \left\lvert \frac{x_{1\{i\}} - x_{1\{i-1\}}}{x_{1\{i\}}} \right\rvert$	$\varepsilon_2^{\{i\}} = \left\lvert \frac{x_{2\{i\}} - x_{2\{i-1\}}}{x_{2\{i\}}} \right\rvert$	$\varepsilon_3^{\{i\}} = \left\lvert \frac{x_{3\{i\}} - x_{3\{i-1\}}}{x_{3\{i\}}} \right\rvert$
1	0.6667	2.5834	3.0416	0.5	0.6129	0.6712
2	0.8194	2.0347	3.1632	0.1864	0.2696	0.0384
...	...	...	...	...	...	...
8	1.0000	2.0000	2.9999	0.0004	0.0001	0.0000


START
we C, X, R, n, ε
m ← 0
i ← 1
TEMP ← r_i
j ← 1
j=i
T
F
TEMP ← TEMP-x_j/c_ij
j=n
F
j ← j+1
T
TEMP ← TEMP/c_ii
|TEMP-x_i|>ε
F
T
m ← m+1
x_i ← TEMP
i=n
F
i ← i+1
T
m=0
F
T
wy X
STOP

Wolne i Otwarte Oprogramowanie (*Free Libre/Open Source Software, FOSS, FLOSS*) – neutralny skrót pozwalający objąć jednym mianem zarówno **Wolne Oprogramowanie** (*Free Software*) jak i **Otwarte Oprogramowanie** (*Open Source*).

Wolne Oprogramowanie – oprogramowanie, które może być uruchamiane, kopiowane, rozpowszechniane, analizowane oraz zmieniane i poprawiane przez użytkowników.

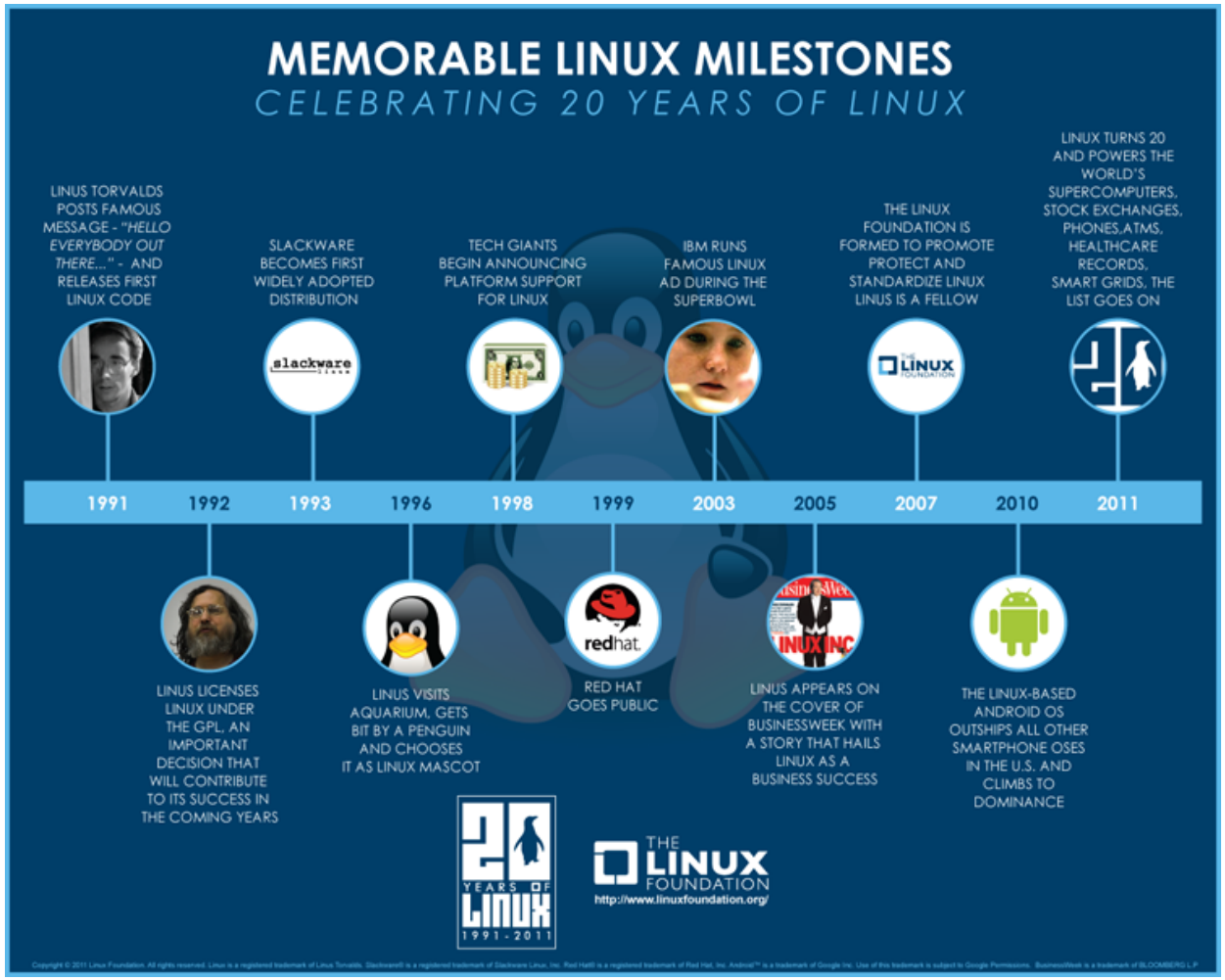
Definicja Wolnego Oprogramowania została opublikowanej przez **Free Software Foundation** i wynikają z niej następujące wolności:

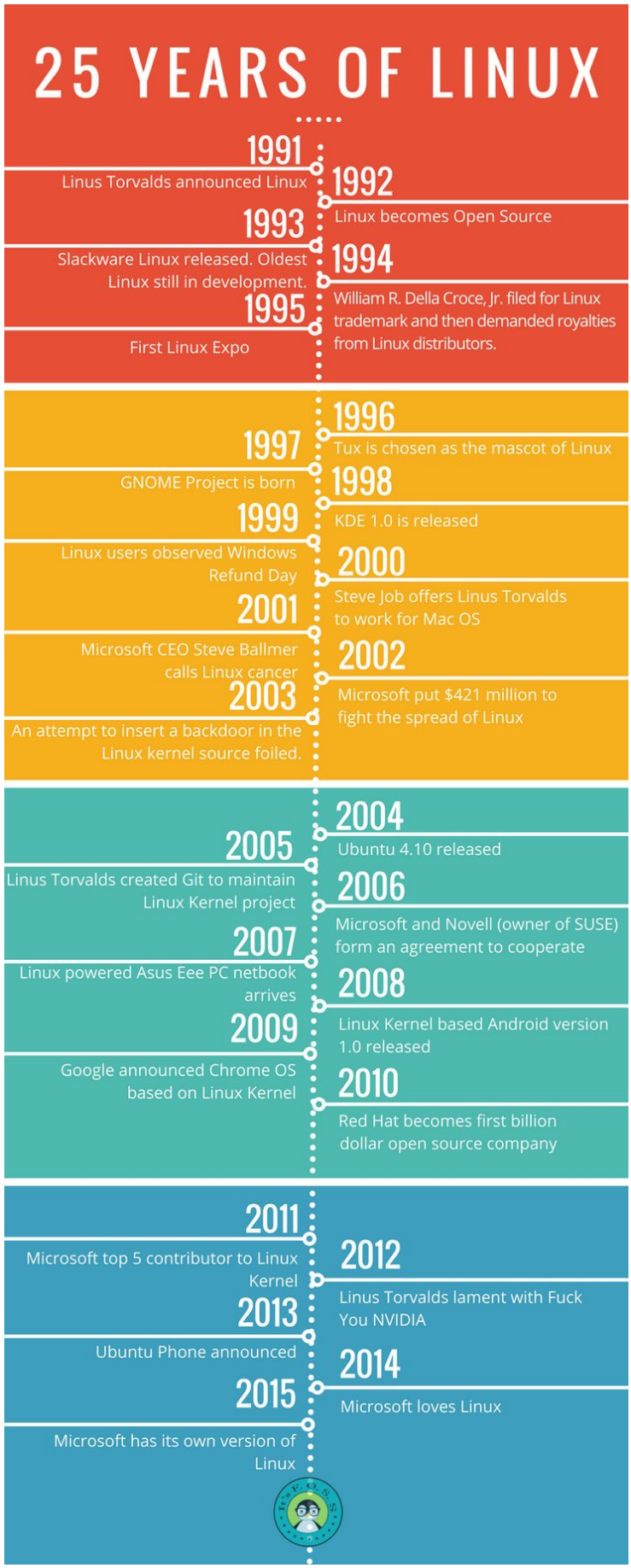
- wolność uruchamiania programu, w dowolnym celu,
- wolność analizowania programu oraz dostosowywania go do swoich potrzeb,
- wolność rozpowszechniania kopii programu,
- wolność udoskonalania programu i publicznego rozpowszechniania własnych ulepszeń.

Otwarte Oprogramowanie (*Open Source*) – odłam ruchu Wolnego Oprogramowania, alternatywny dla Free Software, głównie z przyczyn praktycznych.

W praktyce każdy program na licencji zgodnej z definicją Free Software Foundation jest jednocześnie zgodny z bardziej liberalną definicją **Open Source Movement**.

GNU/Linux – system operacyjny.

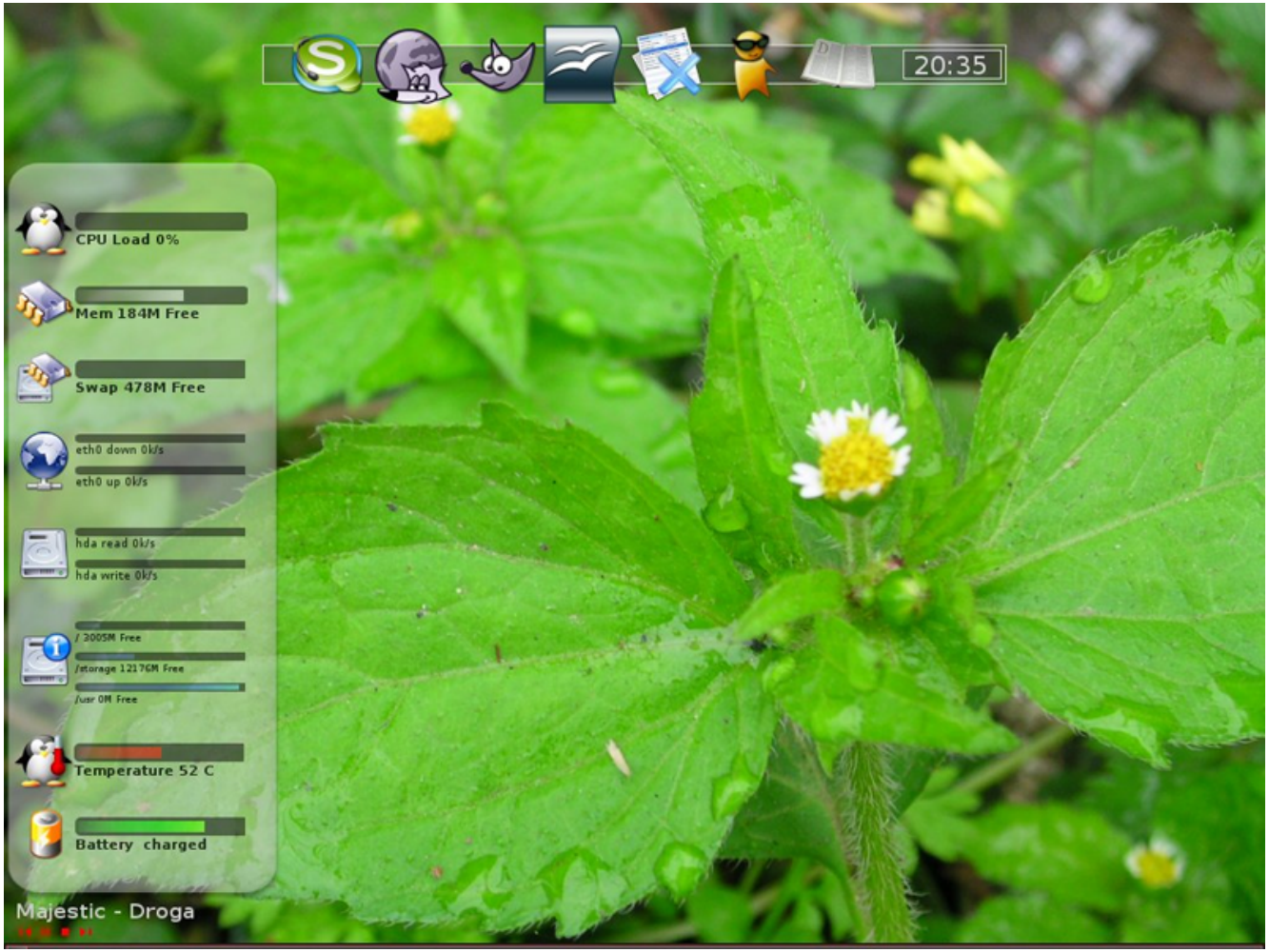



25 YEARS OF LINUX
1991
Linus Torvalds announced Linux
1992
Linux becomes Open Source
1993
Slackware Linux released. Oldest Linux still in development.
1994
William R. Della Croce, Jr. filed for Linux trademark and then demanded royalties from Linux distributors.
1995
First Linux Expo
1996
Tux is chosen as the mascot of Linux
1997
GNOME Project is born
1998
KDE 1.0 is released
1999
Linux users observed Windows Refund Day
2000
Steve Job offers Linus Torvalds to work for Mac OS
2001
Microsoft CEO Steve Ballmer calls Linux cancer
2002
Microsoft put $421 million to fight the spread of Linux
2003
An attempt to insert a backdoor in the Linux kernel source foiled.
2004
Ubuntu 4.10 released
2005
Linus Torvalds created Git to maintain Linux Kernel project
2006
Microsoft and Novell (owner of SUSE) form an agreement to cooperate
2007
Linux powered Asus Eee PC netbook arrives
2008
Linux Kernel based Android version 1.0 released
2009
Google announced Chrome OS based on Linux Kernel
2010
Red Hat becomes first billion dollar open source company
2011
Microsoft top 5 contributor to Linux Kernel
2012
Linus Torvalds lament with Fuck You NVIDIA
2013
Ubuntu Phone announced
2014
Microsoft loves Linux
2015
Microsoft has its own version of Linux

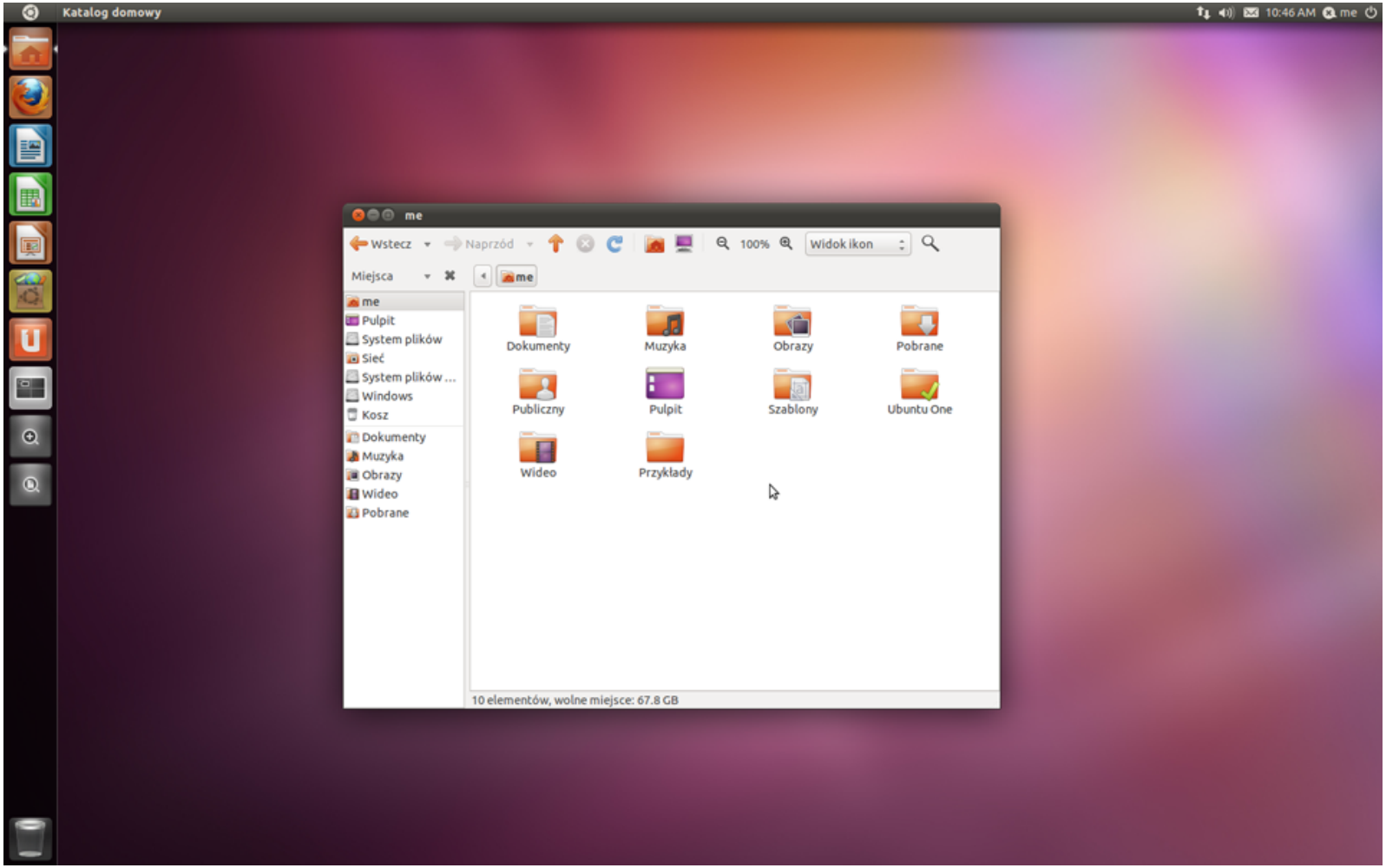
Krótka charakterystyka i ważniejsze cechy systemu GNU/Linux:

- efektywność i stabilność systemu,

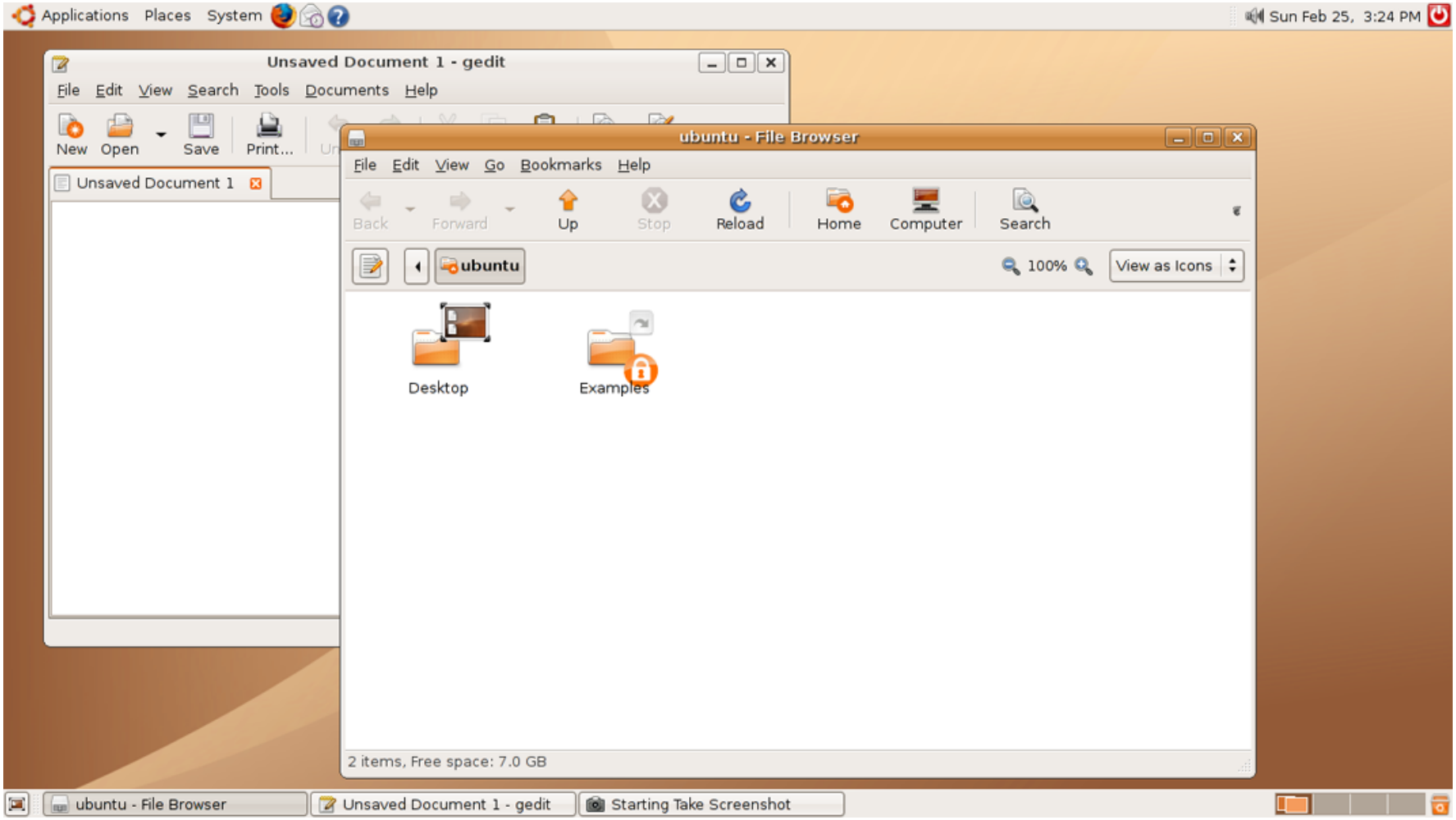
- powszechna dostępność bez jakichkolwiek opłat licencyjnych,
- bogaty zestaw oprogramowania umożliwiający szeroki zakres zastosowań,



- możliwość pracy na wielu platformach sprzętowych przy stosunkowo niewielkich wymaganiach,
- możliwość łatwej współpracy z innymi popularnymi systemami operacyjnymi,



- bogata dokumentacja w wersji elektronicznej,
- dostępność kodu źródłowego.

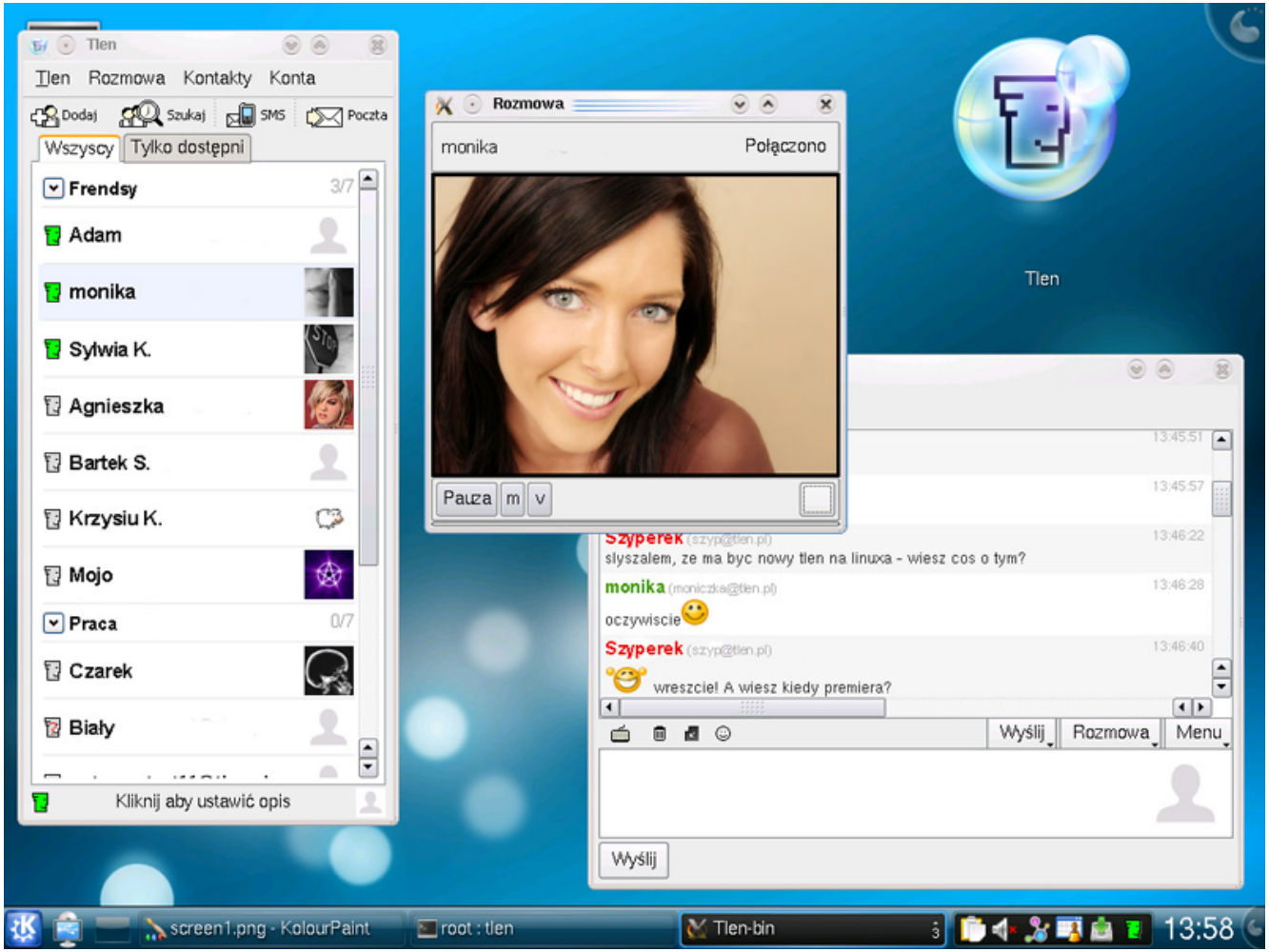


- wielodostęp,
- wielozadaniowość, czyli praca z podziałem czasu procesora pomiędzy wiele zadań,

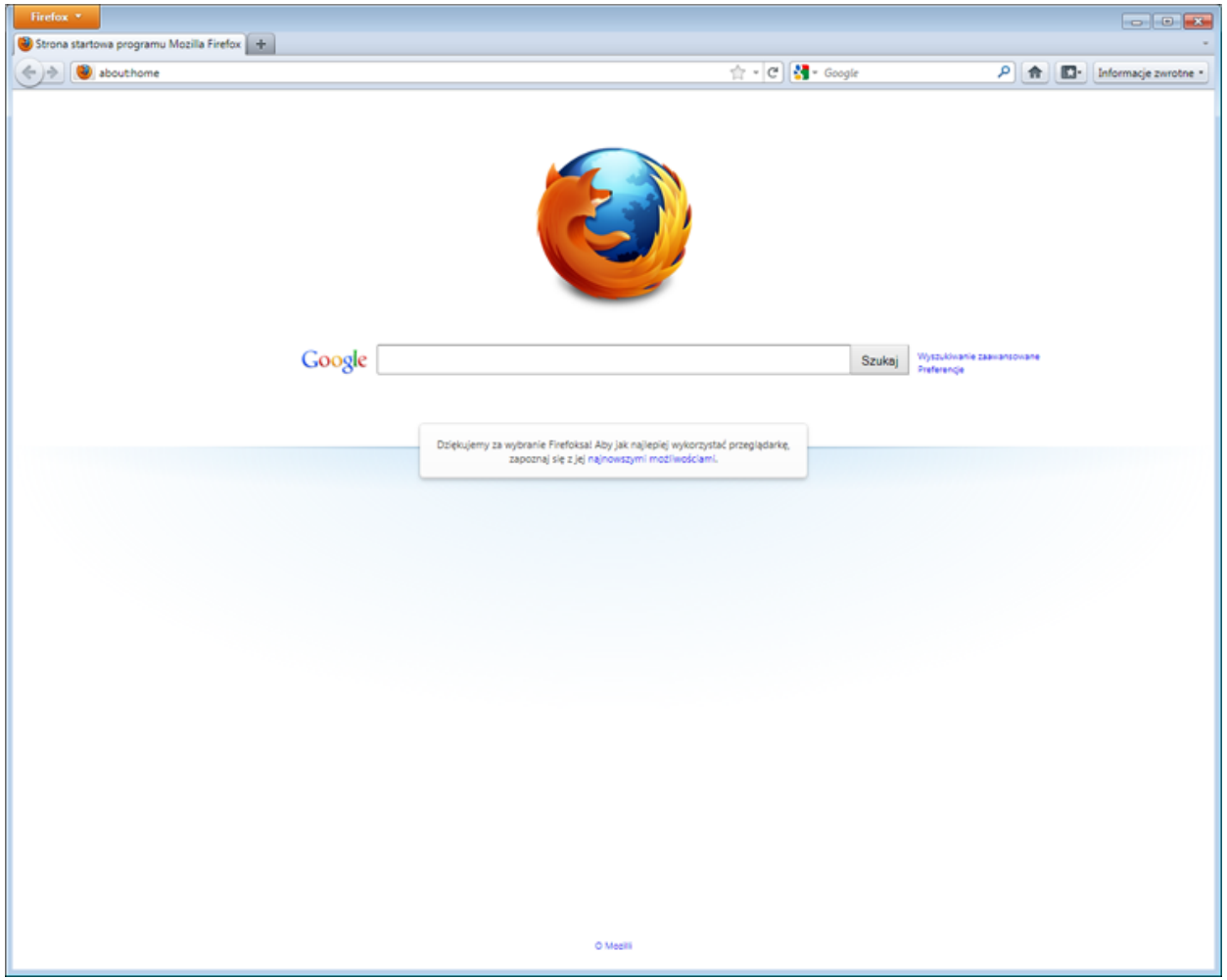
- wieloprzetwarzanie, czyli praca wieloprocesorowa,
- możliwość uruchamiania zadań w łagodnym czasie rzeczywistym,

- obsługa różnych typów systemów plików,
- obsługa różnych protokołów sieciowych,

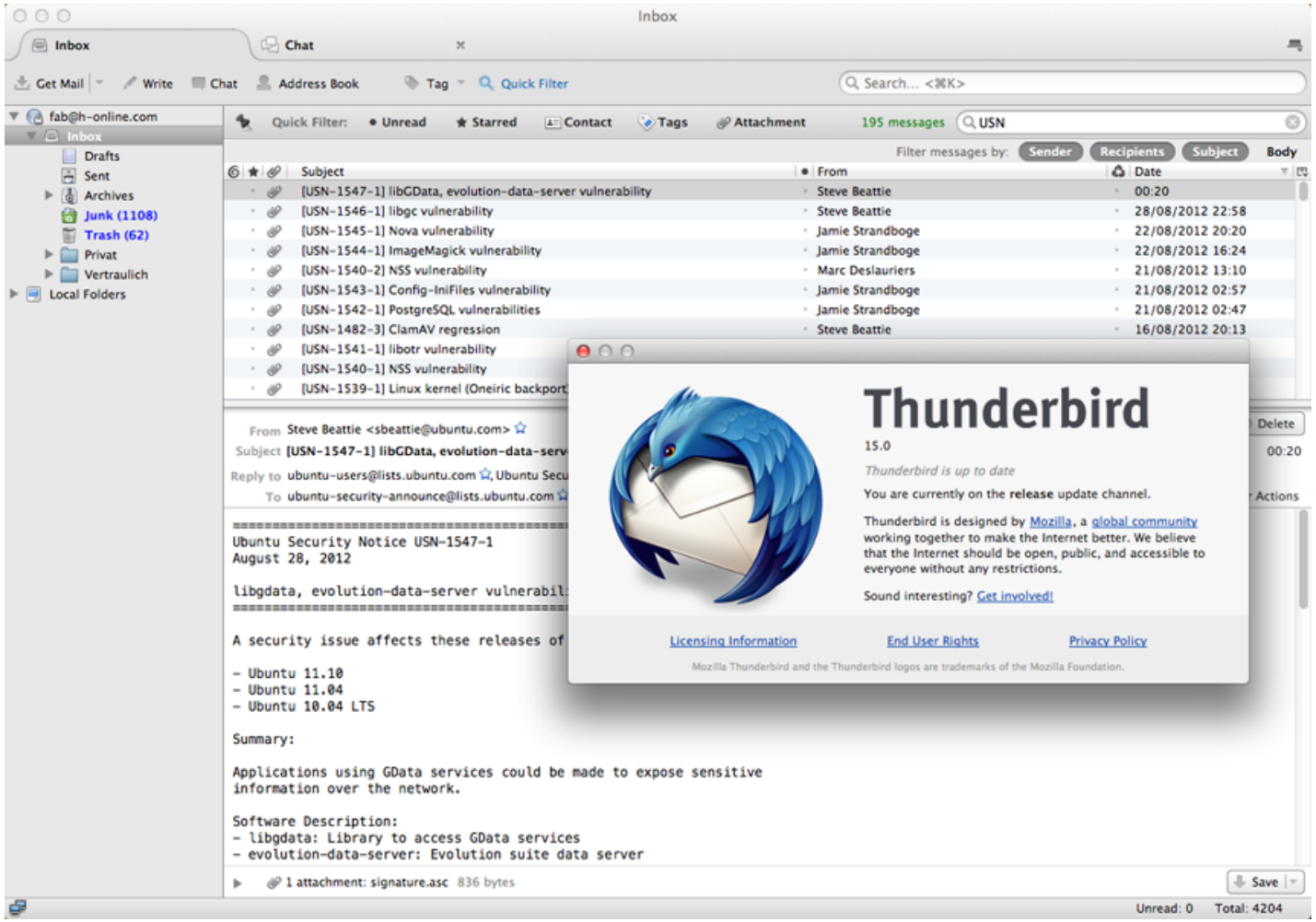
- obsługa różnych formatów plików wykonywalnych,
- wykorzystanie współdzielonych bibliotek.



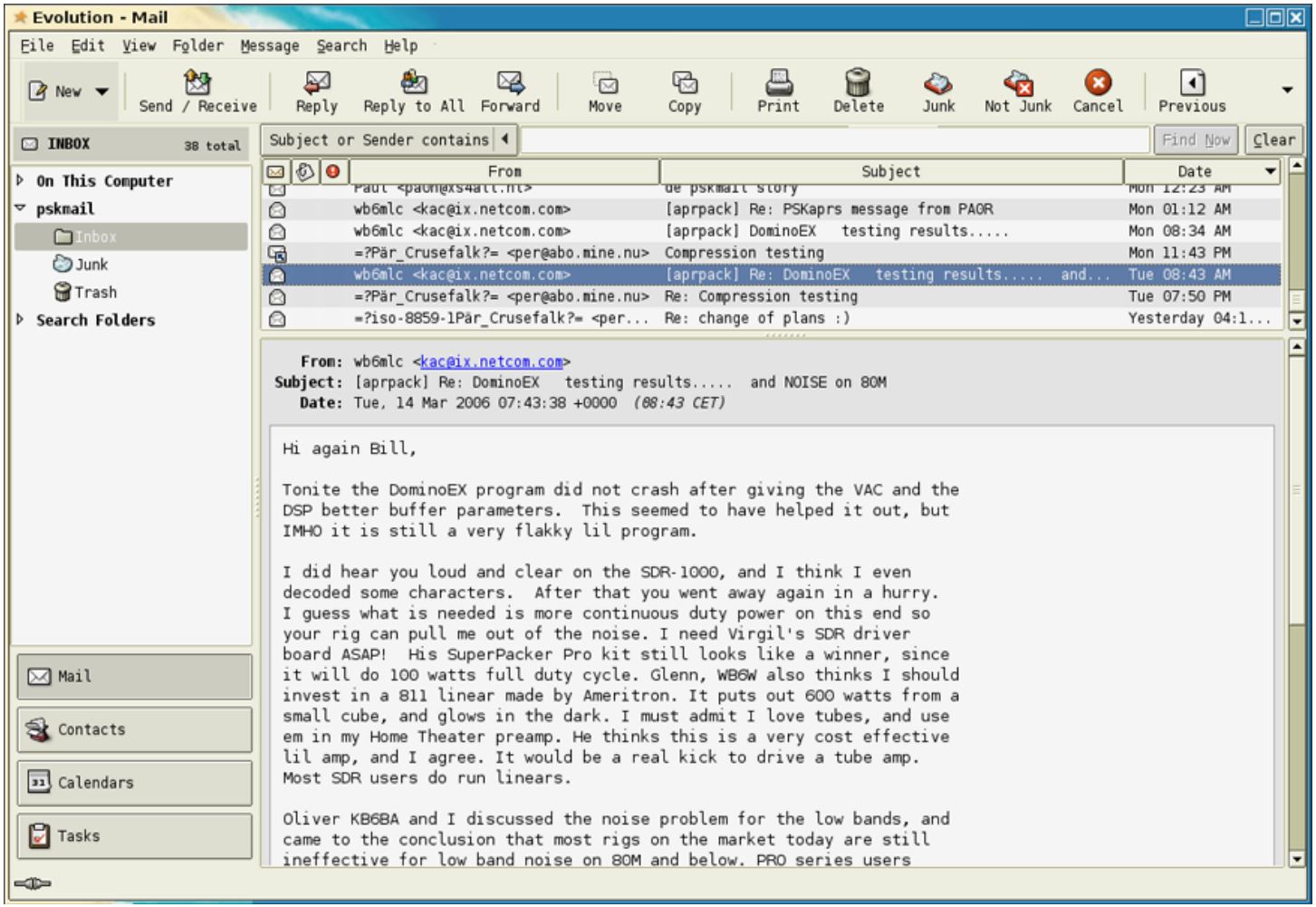
Firefox – przeglądarka internetowa.



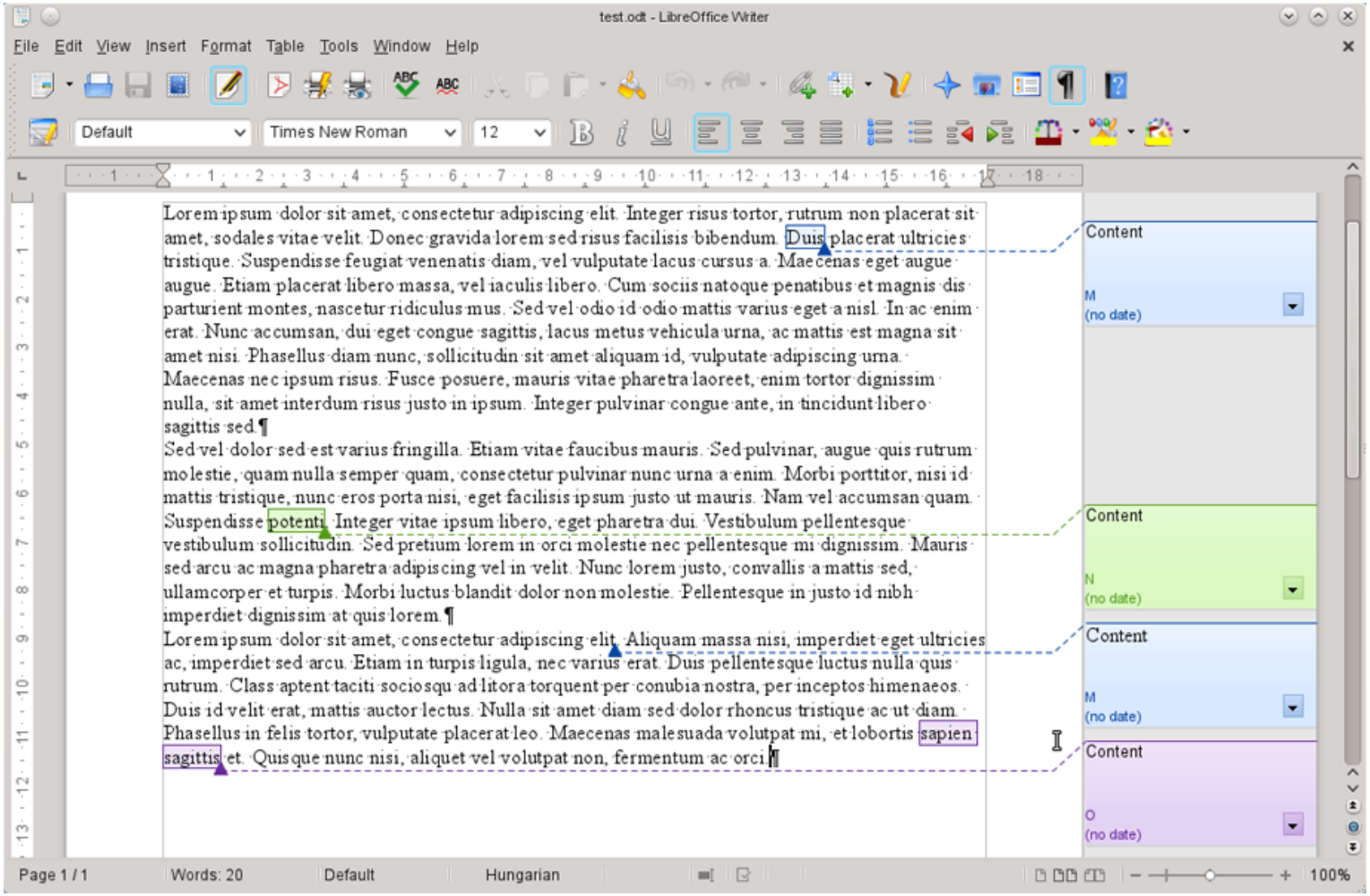
Thunderbird – klient poczty, czytnik grup dyskusyjnych i kanałów informacyjnych.



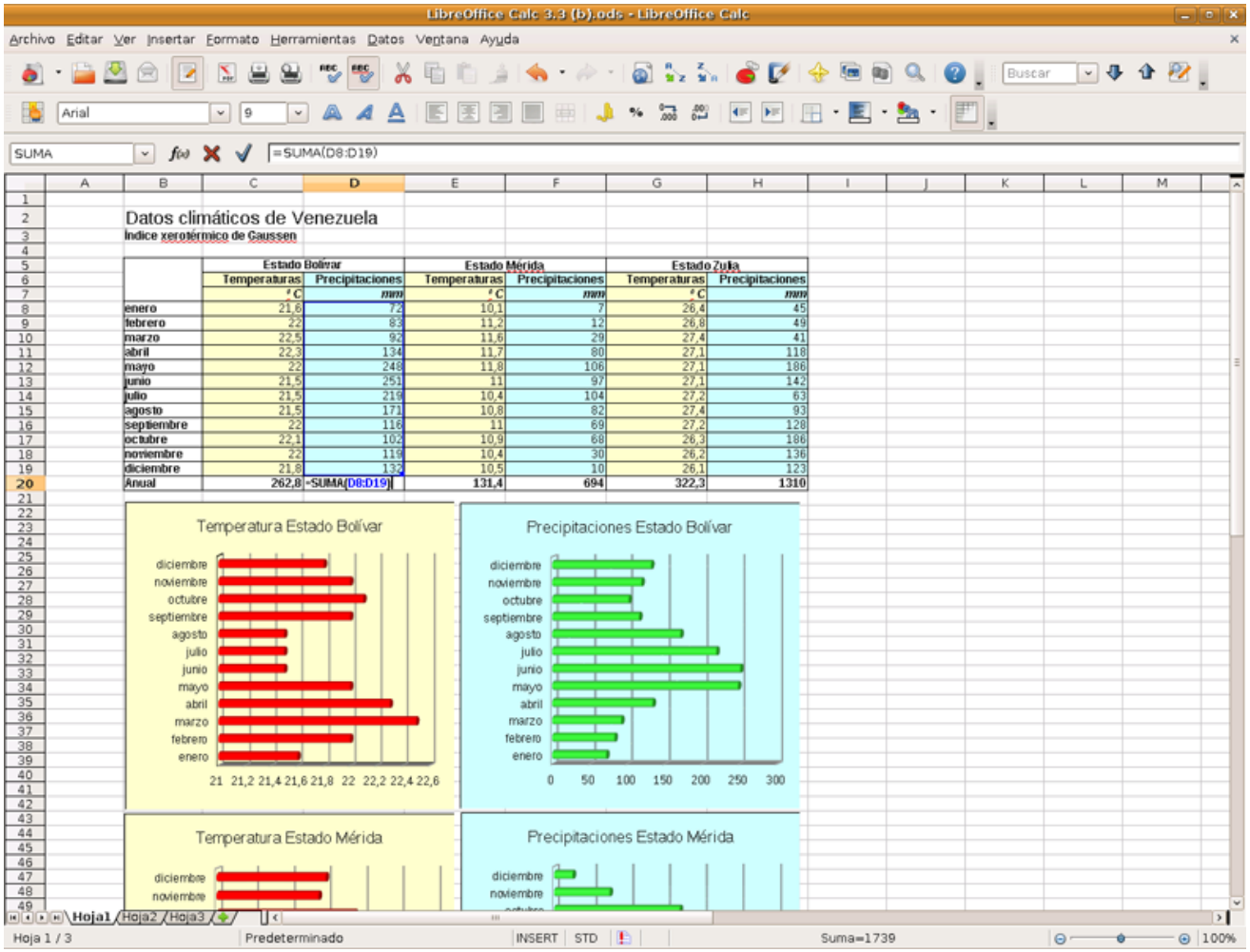
Evoliution – klient poczty, organizer.



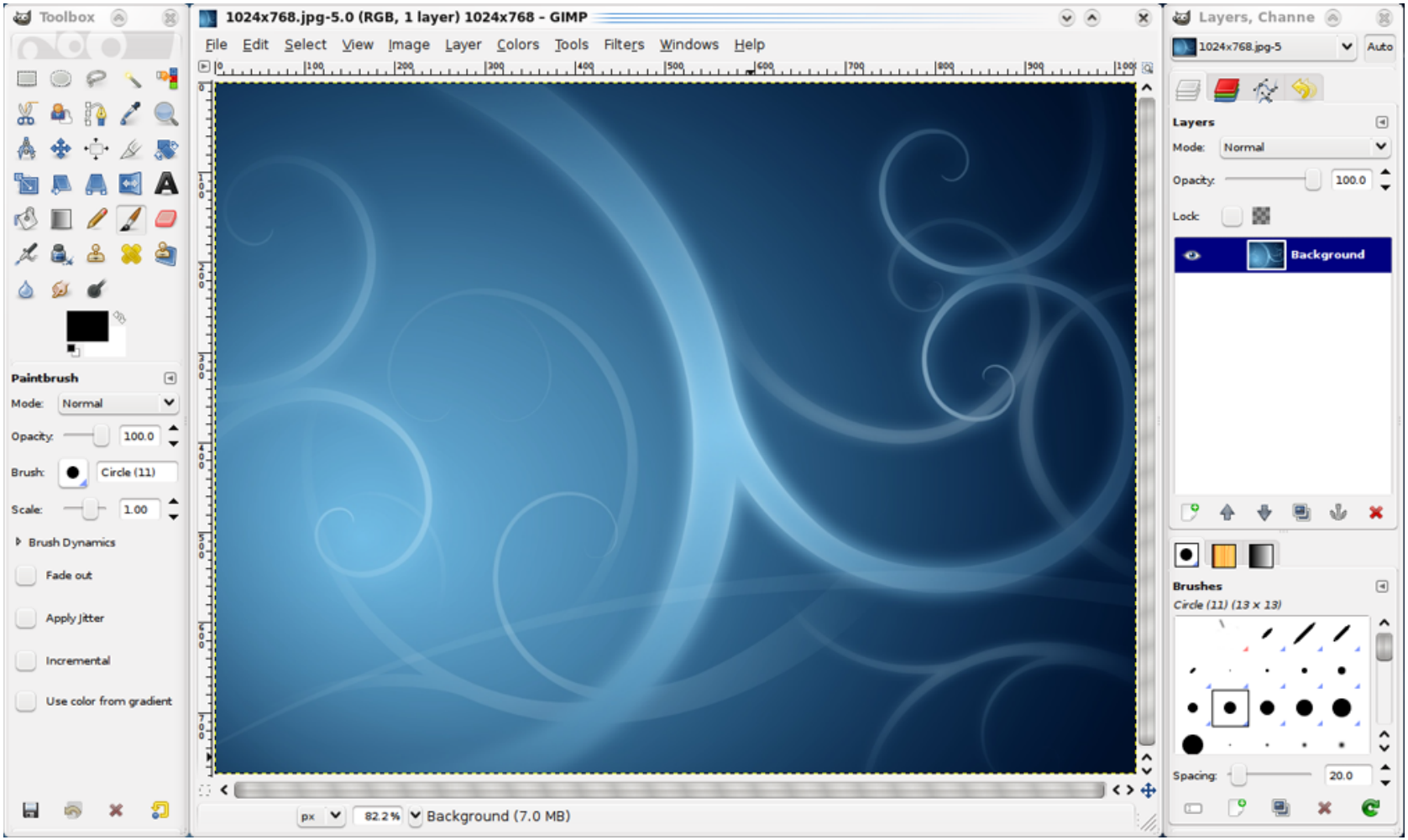
LibreOffice Writer – edytor tekstu.



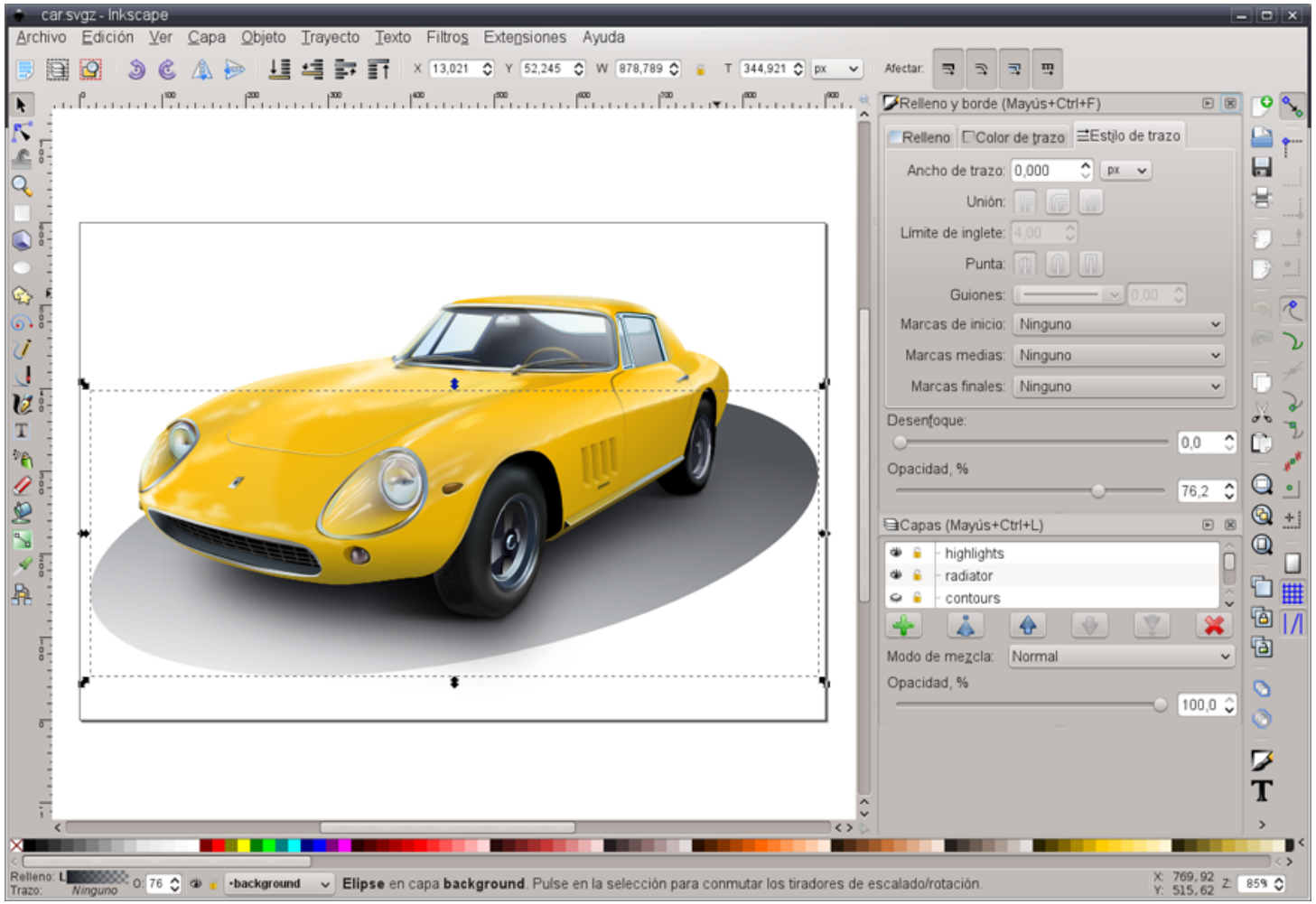
LibreOffice Calc – arkusz kalkulacyjny.



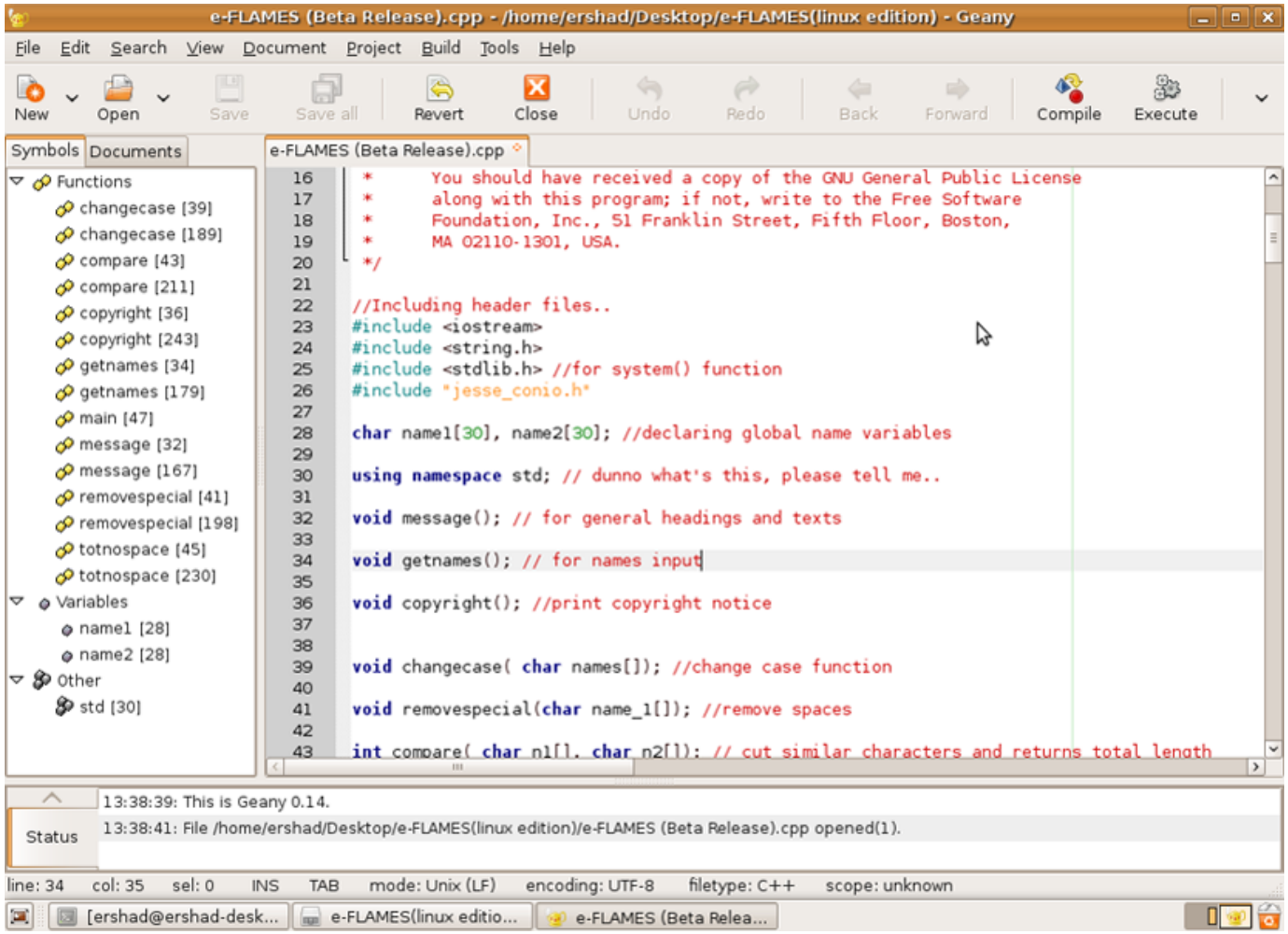
GIMP – edytor grafiki rastrowej.



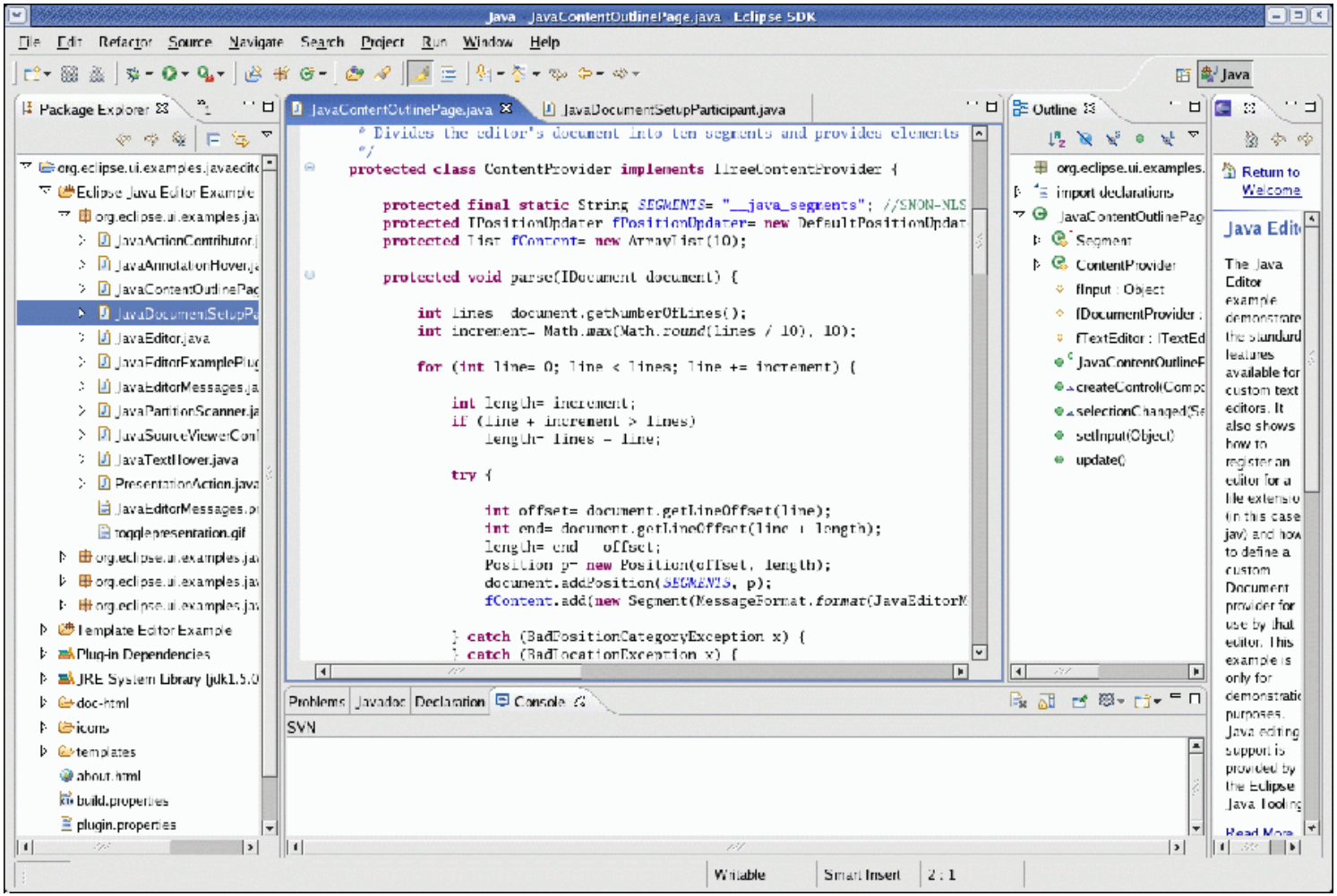
Inkscape – edytor grafiki wektorowej.



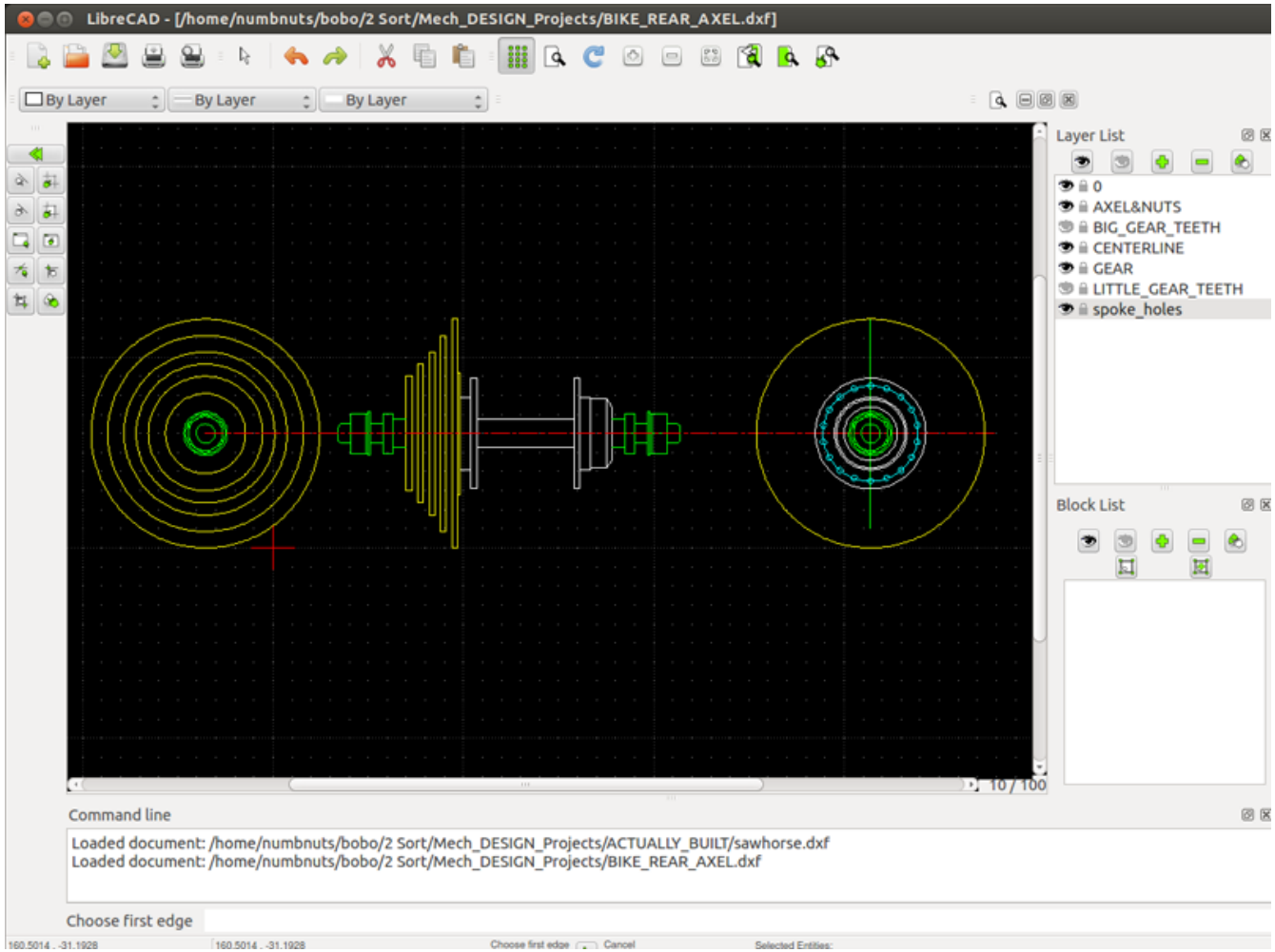
Geany – edytor tekstu programisty.



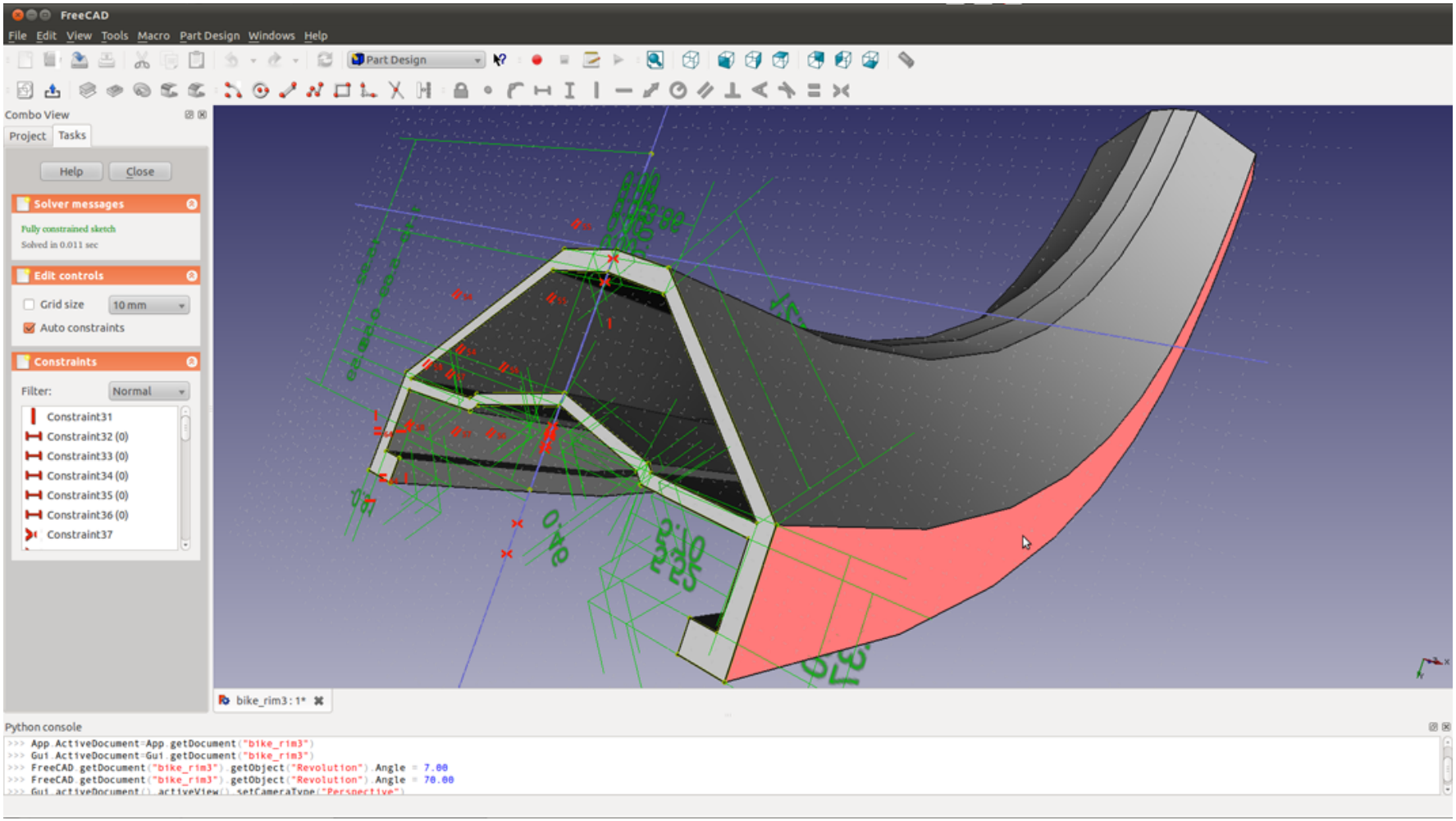
Eclipse – zintegrowane środowisko programistyczne.



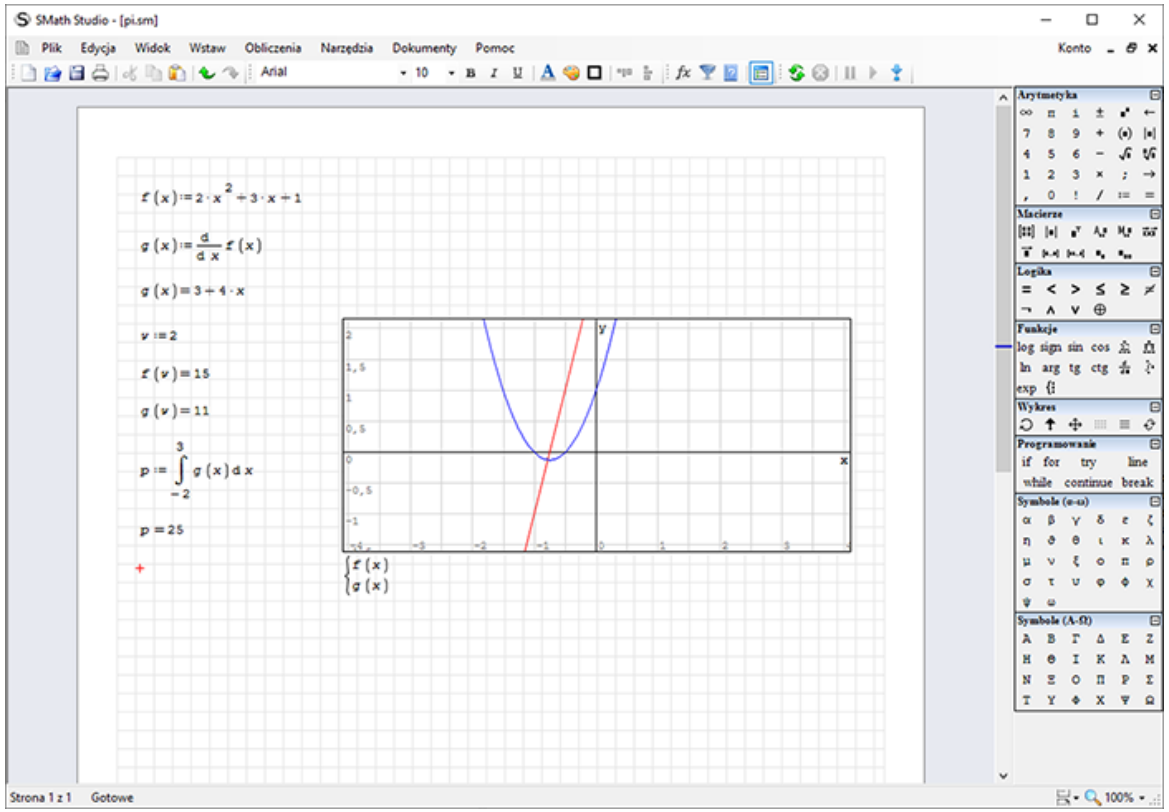
LibreCAD – aplikacja CAD 2D.



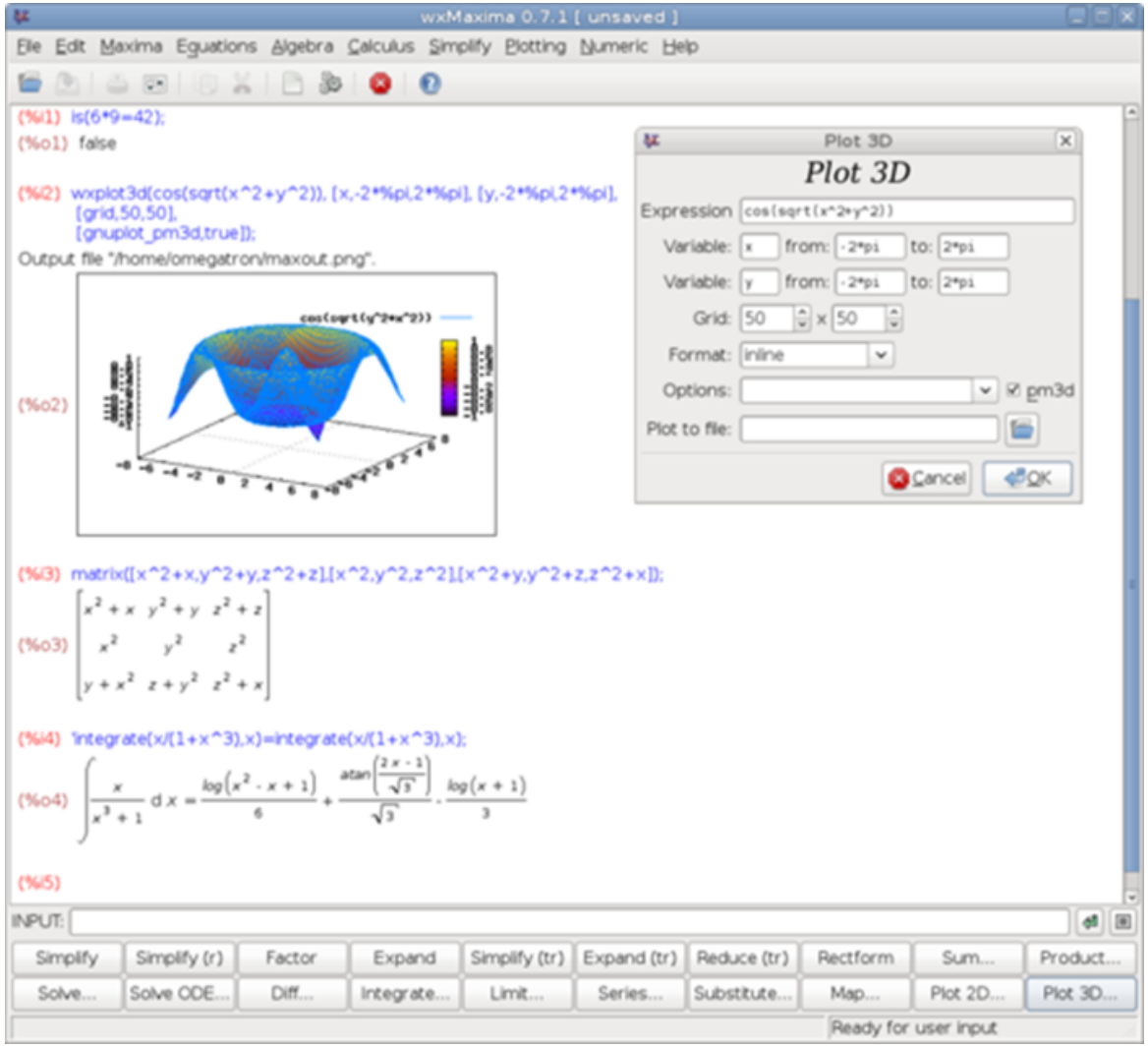
FreeCAD – aplikacja CAD 3D.



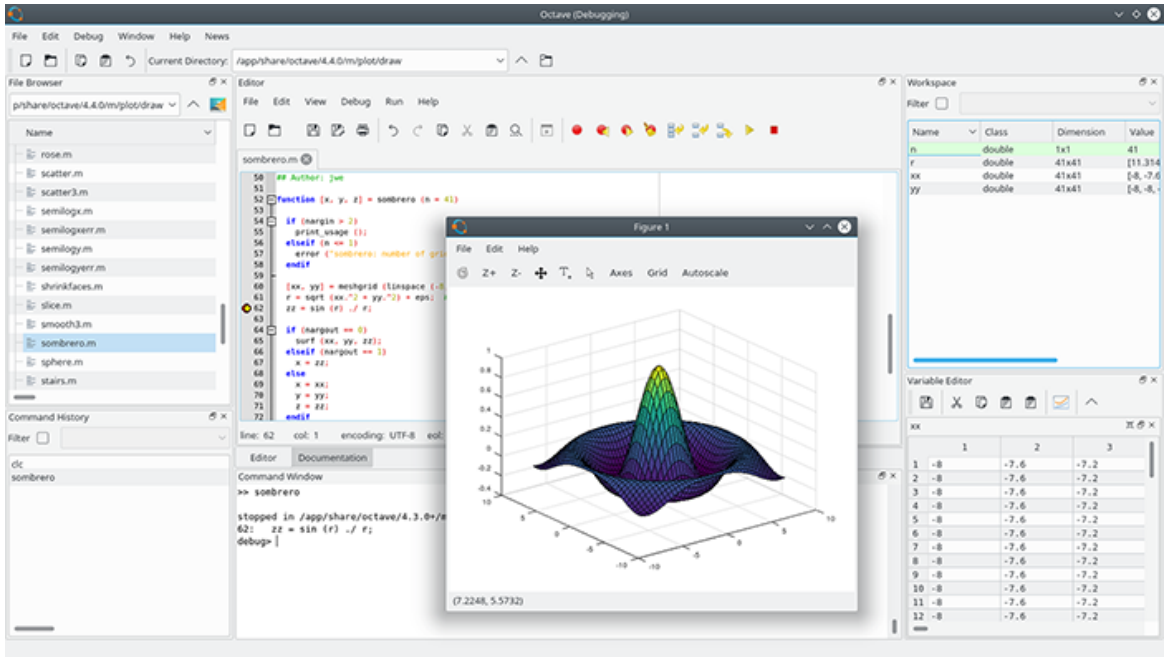
SMath Studio – aplikacja CAS (zamknięte źródło).



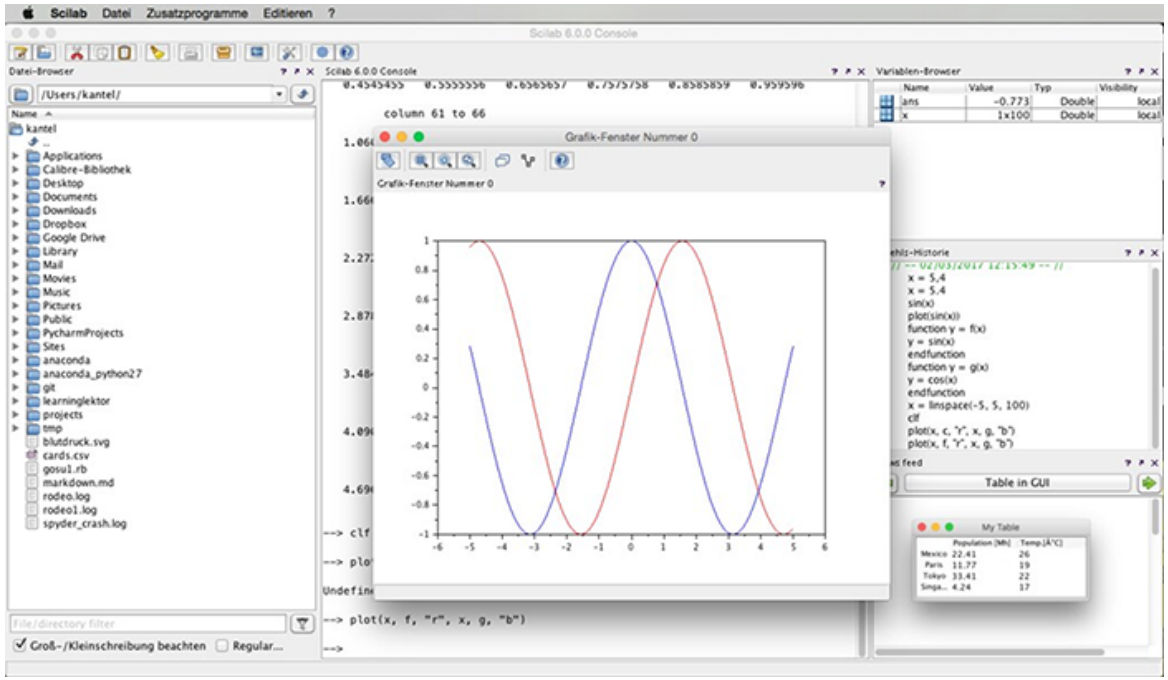
Maxima – aplikacja CAS.



Octave – aplikacja CAS.



SciLab – aplikacja CAS.



Python(x,y) – aplikacja CAS, język programowania.