KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

Współfinansowane przez Unię Europejską w ramach Europejskiego Funduszu Społecznego

Metody komputerowe matematyki przemysłowej

Część III.
Podstawy obliczeń i przykłady

Vladimir Mityushev
Wojciech Nawalaniec
Natalia Ryłko
Alexander Malevich

Instytut Matematyki Uniwersytetu Pedagogicznego w Krakowie

2010

Metody komputerowe matematyki przemysłowej

Część III.
Podstawy obliczeń i przykłady

Vladimir Mityushev
Wojciech Nawalaniec
Natalia Ryłko
Alexander Malevich

Wydawnictwo Pracowni Komputerowej Jacka Skalmierskiego

Paszczyńska 44, 44-100 Gliwice

tel. 032 729 80 97, tel. kom. 0 506 132 960 fax 032 729 85 49

www.pkjs.com.pl
pkjs@pkjs.com.pl

Projekt okładki: Katarzyna Kopańska (www.kopanska.pl)

Printed in Poland, Gliwice 2010

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

Niniejszy podręcznik został przygotowany na potrzeby nowej specjalności Matematyka stosowana kierunku Matematyka Uniwersytetu Pedagogicznego im. Komisji Edukacji Narodowej w Krakowie w ramach oferty „Zamawianie kształcenia na kierunkach technicznych, matematycznych i przyrodniczych - pilotaż" - Priorytet IV PO KL „Szkolnictwo wyższe i nauka", Działanie 4.1 „Wzmocnienie i rozwój potencjału dydaktycznego uczelni oraz zwiększenie liczby absolwentów kierunków o kluczowym znaczeniu dla gospodarki opartej na wiedzy", Poddziałanie 4.1.2 „Zwiększenie liczby absolwentów kierunków o kluczowym znaczeniu dla gospodarki opartej na wiedzy".

Spis treści

1. Podstawy pracy z *Mathematica*®

1.1. Poznaj *Mathematica*®

1.1.1. Struktura programu I

Głównymi elementami programu *Mathematica*® jest jądro programu (*Kernel*) oraz graficzny interfejs użytkownika w postaci notatnika (*Front End*). Kernel jest to jednostka wykonująca obliczenia, mająca dołączone wbudowane funkcje, procedury działania itd. Natomiast Front End umożliwia użytkownikowi łatwą komunikację z jądrem. Po uruchomieniu programu wyświetlone zostaną składowe interfejsu: menu programu oraz okno z obszarem roboczym - tzw. notatnik (*Notebook*). Wykonajmy proste działanie: dodawanie liczb. Wpisujemy w notatniku treść polecenia (czyli **2+2**) i naciskając SHIFT + ENTER (lub ENTER z klawiatury numerycznej) wysyłamy komendę do jądra programu (co od tego momentu będziemy nazywać *obliczeniem zawartości komórki*).

```
In[1]:= 2 + 2
Out[1]= 4
```

Zauważmy, że po prawej stronie pojawiły sie kwadratowe nawiasy. Otóż notatnik, w trakcie pracy jest automatycznie dzielony na tzw. komórki (*Cells*), które z kolei łączone są w grupy. Domyślnym typem komórek są: komórka wejścia (*Input* - **In[n]**) i wyjścia (*Output* - **Out[n]**). W pierwszej wpisujemy komendę skierowaną do programu, natomiast w postaci komórki wyjściowej otrzymujemy rezultat (odpowiedź) podany przez *Mathematica*®. Każda para komórek typu **In**/**Out** posiada numer, co pozwala w trakcie pracy odwoływać się do wcześniejszych wyników. Numerowanie przebiega rosnąco, począwszy od komórki obliczonej na samym początku a skończywszy na tej obliczonej na końcu. Aby użyć wyniku z ostatnio obliczonej komórki używamy symbolu **%**. Natomiast **%n** odnosi się do komórki o numerze n.

```
In[2]:= %
Out[2]= 4
```

⚠ **Uwaga** : W dalszej części książki etykiety **In**/**Out** komórek będą pomijane.

1.1.2. Podstawowe działania

Na chwilę zostawmy kwestię struktury programu *Mathematica*® i zajmijmy się podstawami pracy w notatniku. Program umożliwia operacje na zmiennych (symbolach).

```
3 x + 1.2 x
4.2 x
```

⚠ **Uwaga** : Użycie na końcu wyrażenia symbolu średnika spowoduje, że wynik nie zostanie wyświetlony, mimo obliczenia wartości wyrażenia.

```
1/2 p + 1/3 p;
%
```

$$\frac{5\,p}{6}$$

Omówimy teraz problem używania znaku równości. Otóż w *Mathematica*® pojawiają się następujące konstrukcje z użyciem symbolu **=**

1) **a=b** - obiektowi **b** nadajemy nazwę **a**; następuje obliczenie obiektu **b** i od tego momentu **a** symbolizuje wynik obliczenia **b**

```
nz1 = 5 t + 1 t
```

```
6 t
```

```
nz1
```

```
6 t
```

2) **a:=b** - dla obiektu **b** rezerwujemy nazwę **a**, jednak **b** pozostaje w formie nieobliczonej aż do momentu pierwszego użycia nazwy **a** (każde następne użycie nazwy **a** powoduje ponowne obliczenie **b**)

```
nz2 := 7 t + t
```

Zauważmy, że nie został wygenerowany żaden wynik. Dopiero użycie nazwy spowoduje obliczenie wyrażenia.

```
nz2
```

```
8 t
```

3) **a==b** - obliczenie tego wyrażenia spowoduje sprawdzenie czy **a** i **b** są równe (**==** symbolizuje również znak równości w równaniach)

```
nz1 + 2 t == nz2
```

```
True
```

Jeżeli obiektowi nadamy nazwę, która posiada już wartość, to wartość ta zmieni się i od tego momentu każde wyrażenie zawierające ów obiekt, zmieni swoją postać (po ponownym obliczeniu). W następujący sposób łatwo zauważymy różnicę między "**=**" a "**:=**". Niech

```
nz3 := nz1 + nz2
```

```
nz4 = nz1 + nz2;
```

wtedy

```
nz3 == nz4
```

```
True
```

Teraz nadajmy **nz1** inną wartość

```
nz1 = t
```

```
t
```

wówczas

```
nz3
```

```
9 t
```

```
nz4
```

```
14 t
```

Użycie nazwy **nz3** spowodowało ponowne obliczenie wyrażenia i nadanie jej nowej wartości. Natomiast **nz4** symbolizuje otrzymany już wcześniej wynik.

⚠ **Uwaga** : Nazwa nie może zaczynać sie od liczby.

⚠ **Uwaga** : Konstrukcje **a**␣**b** albo **a*b** oznaczają mnożenie elementów **a** i **b**, natomiast **ab** (bez spacji) określa nowy symbol.

```
nz1 nz2
```

```
8 t^2
```

```
nz1nz2
```

```
nz1nz2
```

⚠ **Uwaga** : W jednej linii możemy obliczyć kilka wyrażeń, oddzielonych średnikami. W jednej komórce (po przejściu do nowej linii) też jest to możliwe, niekoniecznie z użyciem średników.

Aby uwolnić nazwę od dotychczasowej wartości lub definicji użyjemy polecenia **Clear**. Jest to szczególnie przydatne, gdy w trakcie pracy zaistnieje potrzeba użycia "zajętej" nazwy jako symbolu (zmiennej). **Clear** uwalnia symbol od definicji i wartości, ale nie usuwa np. atrybutów. Za pomocą **=.** usuwamy tylko definicję, natomiast **Remove** powoduje kompletne uwolnienie symbolu. **Clear["Global`*"]** powoduje wyczyszczenie wszystkich użytych symboli (ze względu na wartości i definicje).

```
nz2 =.
```

```
Clear[nz3]
```

```
Remove[nz4]
```

Zauważmy, że **nz3** oraz **nz4** już nie posiadają przypisanych im wcześniej wartości.

```
nz4
```

```
nz4
```

```
nz3
```

```
nz3
```

Jednak **nz2** oraz **nz3** pozostają w *Mathematica*® jako symbole.

```
?? nz2
```

```
Global`nz2
```

```
?? nz3
```

```
Global`nz3
```

Z kolei **nz4** nie jest już rozpoznawane

```
?? nz4
```

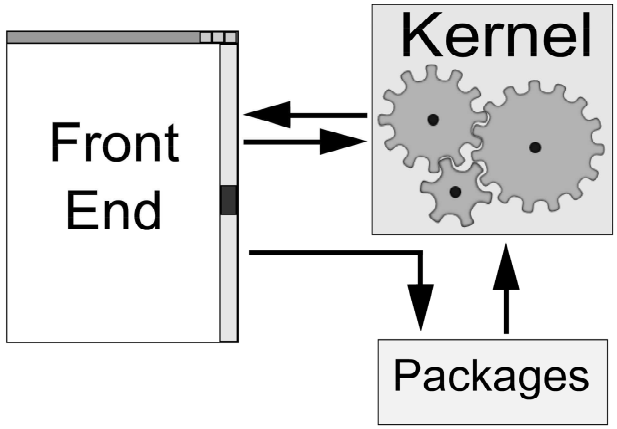
Information::notfound : Symbol nz4 not found. ≫

■ 1.1.3. Struktura programu II

Powróćmy do kwestii struktury programu *Mathematica*®. Jak już wspomnieliśmy, jądro programu operuje wbudowanymi funkcjami i procedurami. Jednak czasem funkcje znajdujące się w bibliotekach jądra mogą okazać się niewystarczajace dla konkretnych zastosowań. Dlatego też *Mathematica*® została wyposażona w obsługę pakietów rozszerzeń (*Packages*) zawierających definicje funkcji niedostępnych w podstawowym zakresie (ograniczenie to jest uzasadnione chociażby szybkością działania programu). Interesujący nas pakiet wczytujemy poleceniem postaci **<<nazwa_pakietu`**. Jako przykład wczytajmy pakiet **Vector Analysis** umożliwiajacy używanie m. in. operatorów różniczkowych (patrz rozdział *Analiza wektorowa*).

```
<< VectorAnalysis`
```

Reasumując, praca z *Mathematica*® przebiega według schematu przedstawionego na poniższym rysunku. Użytkownik łączy się z jądrem za pomoca interfejsu, wysyłając polecenie. Kernel oblicza wyrażenia, stosując funkcje z wbudowanych bibliotek i zwraca użytkownikowi obliczoną wartość. Natomiast gdy funkcja musi zostać pobrana z pakietu, to wówczas komunikacja przebiega za jego pośrednictwem.



Zadaniem jądra programu jest wykonanie wszystkich, skierowanych do niego przez użytkownika poleceń. W rezultacie może powstać szereg nowych definicji, obiektów, funkcji itd. Wszystkie efekty pracy jądra programu pod kierunkiem użytkownika nazywamy sesją (*Session*). Łatwo wyobrazić sobie w takim razie, jak podczas pracy zmienia się zasób "wiedzy" sesji. W momencie uruchomienia programu i wydania pierwszego polecenia obejmuje on wszystkie wbudowane funkcje i procedury kernela. Po wczytaniu dowolnego pakietu, powiększa sie on o elementy tego właśnie pakietu. W końcu wszystkie wyniki, otrzymane w trakcie pracy z programem zwiekszają ów zasób. Wymienione jako ostatnie, obiekty pamięci tymczasowej (efekty pracy użytkownika), otrzymane w trakcie trwania sesji można w każdym momencie usunąć. Służy do tego polecenie *Quit Kernel→Local* z menu *Evaluation*. Wówczas rozpoczynamy zupełnie nową sesję.

⚠ **Uwaga** : Może sie okazać, że ilość operacji jaką musi wykonać kernel w celu obliczenia wyrażenia powoduje zbyt długi czas pracy jądra. Wówczas w każdej chwili możemy przerwać obliczenia opcją *Evaluation→Abort Evaluation* lub za pomocą skrótu ALT + . .

■ 1.1.4. Wszystko jest wyrażeniem

Główną strukturą danych w *Mathematica*® jest *wyrażenie*. Wyrażenie składa się z tzw. głowy (*head of expression*) oraz podwyrażeń (które również są wyrażeniami). Podwyrażenia są zagnieżdżone aż do miejsca w którym otrzymujemy wyrażenie nie dające się już rozłożyć na podwyrażenia (tzw. *wyrażenie atomowe*). Rozpatrzmy następujący przykład.

```
a + b
```
```
a + b
```

Jest to dla nas zupełnie zrozumiały zapis. Jak natomiast interpretuje go *Mathematica*®? Użyjemy polecenia **FullForm**, zwracającego pełną formę wyrażenia

```
% // FullForm
```
```
Plus[a, b]
```

Jest to właśnie wspomniane wyżej wyrażenie. Jego głową jest **Plus**, a podwyrażeniami są **a** i **b**. Wyrażenia **a** i **b** są już wyrażeniami atomowymi. Zatem gdybyśmy chcieli komunikować się z programem na jego zasadach, to aby obliczyć sumę **1+3** musielibyśmy użyć zapisu.

```
Plus[1, 2]
```
```
3
```

Może nie byłoby to w tym przypadku bardzo kłopotliwe, jednak przyjrzyjmy się wyrażeniu $2\,a^2 + \frac{1}{3}\sqrt{b}$, które na pierwszy rzut oka nie wydaje sie bardzo skomplikowane. Nadajmy mu nazwę **w1**.

```
w1 = 2 a^2 + 1/3 Sqrt[b]
```

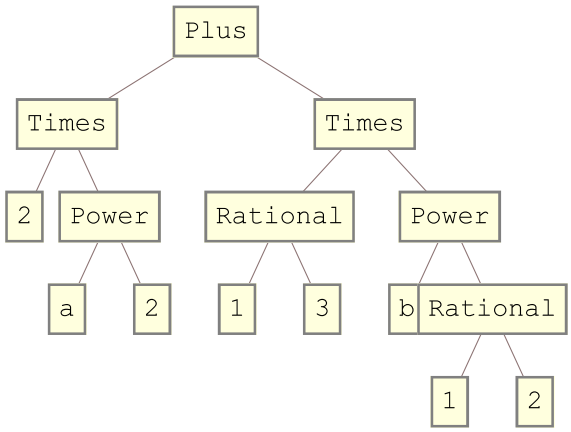
$$2\,a^2 + \frac{\sqrt{b}}{3}$$

```
% // FullForm
```

```
Plus[Times[2, Power[a, 2]],
 Times[Rational[1, 3], Power[b, Rational[1, 2]]]]
```

Jak łatwo zauważyć, jest to wyrażenie zawierające podwyrażenia. Zapis staje się już coraz mniej czytelny, nie mówiąc już o wprowadzaniu danych w takiej formie. Oczywistym jest, że byłoby to niewygodne i pracochłonne. *Mathematica*® ułatwia nam pracę, konwertując czytelny zapis do postaci wyrażeń, na których wykonuje operacje, i odwrotnie - zwraca wynik w formie przyjaznej dla użytkownika. Możemy przedstawić strukturę powyższego wyrażenia w postaci drzewa.

```
w1 // TreeForm
```



Mamy wówczas wyróżnione głowy poszczególnych wyrażeń oraz przypisane do nich podwyrażenia. Brzeg grafu stanowią wyrażenia atomowe. Powyższe drzewo sugeruje, że w każdym przypadku możemy mówić o poziomie zagnieżdżenia wyrażeń. Operator **Level** zwraca wszystkie podwyrażenia z danego poziomu lub przedziału poziomów.

```
Level[w1, {3}]
```

$$\left\{a,\ 2,\ b,\ \frac{1}{2}\right\}$$

```
Level[w1, 2]
```

$$\left\{2,\ a^2,\ 2\,a^2,\ \frac{1}{3},\ \sqrt{b},\ \frac{\sqrt{b}}{3}\right\}$$

Funkcja **AtomQ** pozwala stwierdzić czy dane wyrażenie jest wyrażeniem atomowym.

```
AtomQ[a]
```

```
True
```

Mając dane wyrażenie, możemy odczytać jego głowę za pomocą polecenia **Head**.

```
Head[a b]
```

```
Times
```

```
ab // Head
```

```
Symbol
```

```
Head[{a, b, c}]
```

```
List
```

Do dyspozycji mamy także operator zmiany głowy wyrażenia **Apply** (więcej w rozdziale *Operacje funkcyjne na listach*).

```
Apply[Plus, {a, b, c}]
```

```
a + b + c
```

Podobnie każda funkcja, operator lub komenda wpisywana w *Mathematica*® ma postać wyrażenia (głowa + treść).

```
Factor[x^2 + 2 x + 1]
```

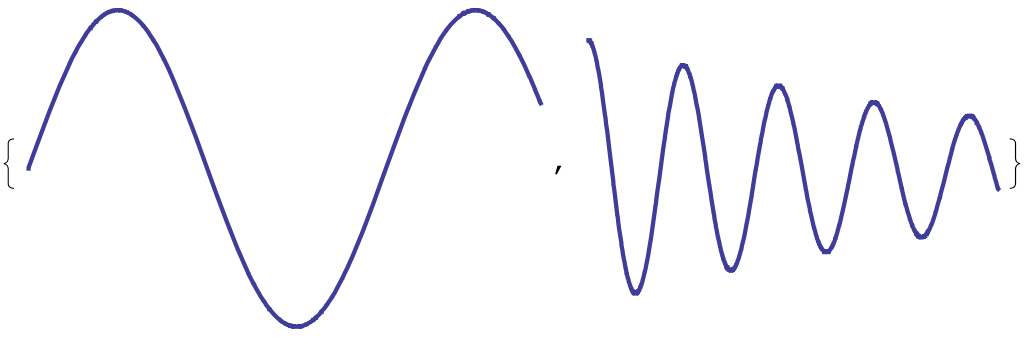
```
(1 + x)^2
```

```
Sin[3.14]
```

```
0.00159265
```

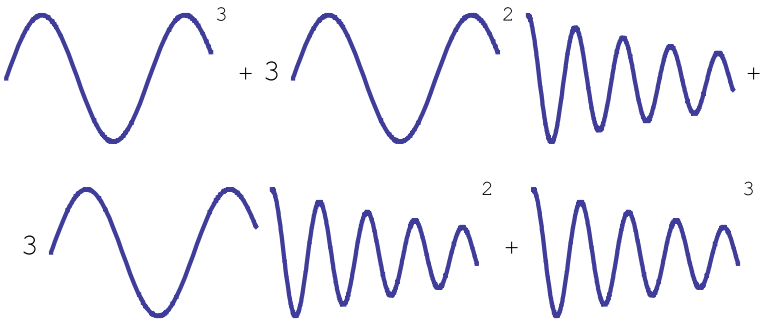
Mathematica® nie sprawdza, czy obliczana struktura ma sens: wszystko jest wyrażeniem! Jeżeli tylko podane wyrażenie jest poprawne pod względem składni, to program wykona obliczenie. Jeżeli będziemy chcieli rozwinąć sześcian sumy następujących dwóch wyrażeń graficznych

```
{Plot[Sin[x], {x, 0, 9}, Axes → False],
 Plot[e^(-x/10) Cos[3 x], {x, 0, 9}, Axes → False]}
```



to *Mathematica*® bez problemu wykona to polecenie

```
(Plot[Sin[x], {x, 0, 9}, Axes → False] +
    Plot[e^(-x/10) Cos[3 x], {x, 0, 9}, Axes → False])^3 // Expand
```



W poniższych przykładach obliczane są wyrażenie z argumentem postaci .

```
Sin[]^2 + Cos[]^2
```

```
Cos[]^2 + Sin[]^2
```

```
% // Simplify
```

```
1
```

```
∫ Sqrt[1 + ^2] d + ∫ d / (1 + ^2)
```

```
ArcTan[] + 1/2 (ArcSinh[] + Sqrt[1 + ^2])
```

Omówimy teraz formy, w jakich mogą występować operatory. Otóż zapisy tej samej funkcji w postaciach:
1) **`Funkcja[argument]`** (*Full Form*);
2) **`argument//Funkcja`** (*Postfix Form*)
3) **`Funkcja@argument`** (*Prefix Form*)
są równoważne. Pokażemy to na przykładzie.

```
x^2 + 2 x + 1 // Factor
```

```
(1 + x)^2
```

```
Factor@ (x^2 + 2 x + 1)
```

```
(1 + x)^2
```

```
3.14 // Sin
```

```
0.00159265
```

```
Sin@3.14
```
```
0.00159265
```

Czasem możemy stosować tzw. *Infix Form*. Mianowicie, postacie funkcji **Funkcja[a,b]** oraz **a~Funkcja~b** są równoważne.

```
a~Plus~b~Plus~c
```
```
a + b + c
```
```
a~Power~x
```
```
a^x
```
```
x~Power~a
```
```
x^a
```

⚠ **Uwaga** : Przy okazji zauważmy, że wyrażenia typu $\cos^2(x)$ wpisujemy w postaci **Cos[x]^2** a nie, jak jest przyjęte w matematyce - **Cos^2[x]**. Następującym przykładem wyjaśnimy dlaczego druga forma jest niepoprawna.

```
Cos[1.4]^2
```
```
0.0288888
```
```
Cos^2[1.4]
```
```
Cos^2[1.4]
```
```
% // Head
```
```
Cos^2
```

Jak widzimy, drugi zapis został potraktowany jako wyrażenie z głową **Cos^2**. Pierwszy natomiast jest poprawny: najpierw obliczamy wartość wyrażenia **Cos[1.4]**, a następnie podnosimy ją do kwadratu.

Z powyższych rozważań wynika jasno fakt: w *Mathematica*® wszystko jest wyrażeniem. Stąd mamy dowolność używania wyrażeń (niedozwoloną w tradycyjnej matematyce). Pewna zmienna może np. przyjmować wartość rzeczywistą albo oznaczać wykres funkcji. Zatem przy wprowadzaniu wyrażeń należy mieć na uwadze to, co ono oznacza, ponieważ przeważnie każde (nawet błędne) sformułowanie *Mathematica*® akceptuje i próbuje obliczać. Warto korzystać z pomocy (*Help*), która jest kolejną, integralną częścią programu. Możemy w tym celu użyć operatora **?** lub **??** w następujacy sposób.

```
? Cos
```

Cos[*z*] gives the cosine of *z*. ≫

```
?? Cos
```

Cos[z] gives the cosine of z. ≫

```
Attributes[Cos] = {Listable, NumericFunction, Protected}
```

Sprawdźmy czy w *Mathematica*® istnieje element, którego chcieliśmy użyć w ostatnim przykładzie.

```
? Cos^2
```

Information::ssym: Cos^2 is not a symbol or a valid string pattern. ≫

```
Information[Cos^2, LongForm → False]
```

Innym sposobem jest bezpośrednie odwołanie sie do systemu pomocy, naciskając klawisz F1 w momencie, gdy kursor znajduje sie na konkretnym zagadnieniu. Zostanie wówczas otwarty dokument pomocy w postaci notatnika zawierający wyczerpujące informacje. *Help* w programie *Mathematica*® jest w pełni interaktywny pozwala na zmianę konstrukcji przykładów, wprowadzenie własnych ustawień, zmiennych itd. Radzimy odwoływać się do pomocy nie tylko w przypadku wątpliwości, ale także w celu zgłębiania wiedzy dotyczącej użytkowania programu *Mathematica*®.

■ 1.1.5. Typy liczbowe oraz wyniki precyzyjne i przybliżone

W *Mathematica*® obowiazują cztery podstawowe typy liczbowe.

1) **Integer** - całkowite;
2) **Rational** - wymierne;
3) **Real** - przybliżone liczby rzeczywiste (z określoną dokładnością); wprowadzane zawsze z kropką dziesiętną
4) **Complex** - zespolone; mają postać **a+bi**, gdzie **a** i **b** są dowolnego "typu".

⚠ **Uwaga** : Od tego momentu będziemy używać następującej konstrukcji. Otóż w komórce *Input* zamieszczamy listę wyrażeń (patrz rozdział *Listy*). Wówczas *Mathematica*® zwróci nam odpowiedź w postaci listy zawierającej wyniki w kolejności odpowiadającej wyrażeniom.

```
{Head[1], Head[1/2], Head[1.], Head[2 + 3. I]}
```

```
{Integer, Rational, Real, Complex}
```

Liczby "typu" **Integer** i **Rational** są tzw. liczbami precyzyjnymi. *Mathematica*® często podaje wynik w postaci precyzyjnej. Jednak, gdy nie jest to możliwe, bądź potrzebujemy przybliżonej wartości pewnego wyrażenia, to używane są przybliżone liczby rzeczywiste (**Rational**). Aby otrzymać numeryczną wartość wyrażenia używamy operatora **N.**

```
{1 // N, 123/321 // N, 20^5 // N, π // N}
```

```
{1., 0.383178, 3.2 × 10^6, 3.14159}
```

⚠ **Uwaga** : Jeżeli w wyrażeniu użyjemy co najmniej jednej wartości w postaci dziesiętnej, to *Mathematica*® zwróci nam numeryczną wartość tego wyrażenia. Zilustrujmy to poniższymi przykładami.

```
{4./12, 10 + 1.}
```

```
{0.333333, 11.}
```

```
{Sin[1], Sin[1.]}
```

```
{Sin[1], 0.841471}
```

Powyżej mamy wartości funkcji sinus w punkcie 1. Pierwszy z wyników jest precyzyjny, natomiast drugi jest w postaci numerycznej. W tym przypadku jest to skrócony zapis (wyświetlone jest tylko 6 cyfr po przecinku). Aby wyświetlić wszystkie cyfry można użyć **InputForm**.

```
InputForm[Sin[1.]]
```

```
0.8414709848078965
```

⚠ **Uwaga** : Warto odnotować następujący fakt. Jeżeli zapiszemy liczbę zespoloną z częścią urojoną równą zero typu **Integer**, to *Mathematica*® zamieni ją do typu **Real** lub **Integer**.

```
{Head[1. + 0 i], Head[1 + 0 i]}
```

```
{Real, Integer}
```

Jeżeli użycie liczby zespolonej z zerową częścią urojoną jest konieczne, wówczas można użyć części urojonej równej zero typu **Real**.

```
Head[1 + 0. i]
```

```
Complex
```

Przybliżona liczba rzeczywista ma niepewność w wartości, czyli błąd obliczeniowy (spowodowany właśnie przybliżeniem). Jeżeli liczba x ma niepewność równą u, to jej prawdziwą wartością może być dowolną liczbą postaci $x \pm u_1$ (gdzie $0 \le u_1 \le \frac{u}{2}$).

Podstawowymi argumentami odpowiedzialnymi za dokładność przybliżonych wyników są **Precision** oraz **Accuracy**. Pierwszy z nich informuje nas o liczbie cyfr w przybliżeniu. Jeżeli u jest niepewnością przybliżonej liczby rzeczywistej x, to precyzję liczby x definiujemy jako $-\log_{10} \frac{u}{x}$. Domyślną wartością jest **$MachinePrecision** - sprzętowa precyzja komputera. Natomiast wartość dla wyników precyzyjnych jest oznaczana jako nieskończoność (∞).

```
Precision /@ {Sqrt[2], Sqrt[2.]}
```

```
{∞, MachinePrecision}
```

```
$MachinePrecision
```

```
15.9546
```

Z kolei parametr **Accuracy**, zdefiniowany jako $-\log_{10} u$, dotyczy liczby cyfr znaczących po przecinku.

```
Accuracy /@ {√2 , 10 √2.}
```

```
{∞, 14.8041}
```

Precision możemy traktować jako miarę niepewności względnej, natomiast **Accuracy** - niepewności absolutnej. Wówczas, mając daną przybliżoną liczbę rzeczywistą x, jej niepewność u równa jest 10^{-A}, gdzie A =**Accuracy[x]**. Dla P =**Precision[x]** oraz $x \neq 0$ mamy $u = |x| 10^{-P}$.

Precyzję obliczeń numerycznych komputera charakteryzuje tzw. *epsilon maszynowy* - najmniejsza liczba nieujemna, która po dodaniu jej do 1 daje liczbę większą od 1. Oznaczmy ten parametr symbolem ε_m. Wtedy

$$\varepsilon_m = \min\{\varepsilon > 0 : 1 + \varepsilon > 1\}$$

W *Mathematica*® ε_m odpowiada **$MachineEpsilon**. Niech

```
εm = $MachineEpsilon
```

```
2.22045 × 10^-16
```

wówczas

```
{1. + εm, 1. + εm/2}
```

```
{1., 1.}
```

Zauważmy, że tylko jeden z wyników jest równy 1.

```
% // InputForm
```

```
{1.0000000000000002, 1.}
```

Oprócz domyślnej precyzji możemy, dzięki następującej składni operatora **N** uzyskać liczby z dowolną wartością **Precision**.

```
N[π, 25]
```

```
3.1415926535897932384626433
```

Istnieje także możliwość ustalenia dokładności dla wszystkich liczb w odrębie pewnego wyrażenia. **SetPrecision** zwraca postać wyrażenia, w której wszystkie liczby zostały zapisane z ustaloną precyzją.

```
SetPrecision[2 x + 10, 41]
```

```
10.00000000000000000000000000000000000000000 +
 2.00000000000000000000000000000000000000000 x
```

Natomiast za pomocą poniższej konstrukcji wykonujemy obliczenia z ustaloną wartością **Precision**.

```
Block[{$MInPrecision = 35, $MaxPrecision = 35},
 SetPrecision[(1 + ℯ^√π - √1.2) / 2^30, 35]]
```

$$5.3922013705668004446507865804957826 \times 10^{-9}$$

Zatem przybliżone liczby rzeczywiste w *Mathematica*® możemy podzielić na te z maszynową dokładnością (tzw. *machine-precision numbers*) oraz te z dokładnością ustaloną przez użytkownika (tzw. *arbitrary-precision numbers*). W obliczeniach możemy używać zarówno pierwszych, jak i drugich. Podstawową różnicą jest czas obliczeń: w przypadku ustalenia większej dokładności niż **$MachinePrecision**, sprzętowe algorytmy muszą być zastąpione przez wolniejsze programowe. Natomiast minusem maszynowej precyzji jest to, iż różne komputery mogą posiadać inne wartości **$MachinePrecision**, co mogłoby skutkować innymi wynikami. Niewątpliwie na korzyść ustalonej przez użytkownika precyzji przemawia sama możliwość jej ustalania (w sytuacjach, w których precyzja maszynowa nie jest wystarczająca).

Przedstawimy teraz przykład innej notacji liczb z określoną precyzją, za pomocą znaku *gravis* (` - klawisz z *tyldą*).

```
{3`, 3`20}
```

{3., 3.00000000000000000000}

```
Precision /@ %
```

{MachinePrecision, 20.}

Jeżeli w wyniku obliczeń otrzymamy wielkości niewiele różniące się od zera, to możemy zażądać od programu, aby je nim zastąpił. Służy do tego polecenie **Chop**. Zeruje ono wszystkie wartości w wyrażeniu, których rząd wielkości jest mniejszy od 10^{-10}.

```
ℯ^N[2 Pi I]
```

$$1. - 2.44929 \times 10^{-16}\, i$$

Część urojona otrzymanej liczby jest bliska zeru, stąd użycie **Chop** spowoduje jej zamianę na zero.

```
% // Chop
```

1.

Należy jednak być ostrożnym podczas używania tego operatora. Rozpatrzmy następujący iloczyn.

```
p1 = 1. × 10^-24 (1. × 10^26 + z);
```

```
p1 // Chop
```
```
0
```

Dokonajmy rozwinięcia wyrażenia **p1** za pomocą operatora **Expand** i wówczas zastosujmy **Chop**.

```
p1 // Expand
```
```
100. + 1. × 10^-24 z
```
```
% // Chop
```
```
100.
```

Dlaczego tak jest? Zauważmy, że

```
(1. × 10^26 + z) // Chop
```
```
1. × 10^26 + z
```

Zatem w wyniku działania **Chop** otrzymaliśmy $0\times\left(1.\times 10^{26}+z\right)=0,$ ponieważ jedynie zastępuje on pewne wartości w wyrażeniu, nie dbając o konsekwencje. Myślenie w takich przypadkach jest obowiązkiem użytkownika: nie wystarczy wiedzieć "czego można użyć", ale także "w jakiej sytuacji jest to sensowne" (patrz zasada głupiego komputera i inne zasady obliczeniowe z Części I). Możemy sobie wyobrazić skutki takiego błędu w dokładnych inżynierskich obliczeniach.

1.2. Listy

1.2.1. Tworzenie list

Esencją pracy z programem *Mathematica*® jest umiejętność konstruowania odpowiednich list oraz wykonywania na nich różnego rodzaju operacji. **List** jest wyrażeniem, które przedstawia listę jego podwyrażeń.

```
Clear["Global`*"]
```
```
List[1, 2, 3]
```
```
{1, 2, 3}
```

Ten sam efekt otrzymamy wprowadzając listę w poniższy sposób.

```
{1, 2, 3}
```
```
{1, 2, 3}
```
```
% // FullForm
```
```
List[1, 2, 3]
```

Utwórzmy następujące listy o nazwach **list1**, **list2** oraz **list3**.

```
list1 = {a, b, c, d, e, f};
```

```
list2 = Table[x_k, {k, 6}]
```

$$\{x_1, x_2, x_3, x_4, x_5, x_6\}$$

```
list3 = Table[n, {n, 4, 21, 3}]
```

```
{4, 7, 10, 13, 16, 19}
```

W przypadku dwóch ostatnich list wykorzystaliśmy operator **Table**. W składni tego operatora podajemy wyrażenie, natomiast w nawiasie znajduje się parametr, od którego zależy wyrażenie, zakres zmienności tego parametru oraz skok.

Istnieje możliwość tworzenia macierzy w postaci list, których elementami są listy o równych długościach.

```
mc1 = {list1, list2, list3 }
```

$$\{\{a, b, c, d, e, f\},\ \{x_1, x_2, x_3, x_4, x_5, x_6\}, \{4, 7, 10, 13, 16, 19\}\}$$

Przedstawmy tę listę w postaci macierzowej.

```
% // MatrixForm
```

$$\begin{pmatrix} a & b & c & d & e & f \\ x_1 & x_2 & x_3 & x_4 & x_5 & x_6 \\ 4 & 7 & 10 & 13 & 16 & 19 \end{pmatrix}$$

Odpowiednie użycie **Table** również da nam macierz.

```
Table[f[a, b], {a, 1, 4}, {b, 3, 7}]
```

```
{{f[1, 3], f[1, 4], f[1, 5], f[1, 6], f[1, 7]},
 {f[2, 3], f[2, 4], f[2, 5], f[2, 6], f[2, 7]},
 {f[3, 3], f[3, 4], f[3, 5], f[3, 6], f[3, 7]},
 {f[4, 3], f[4, 4], f[4, 5], f[4, 6], f[4, 7]}}
```

⚠ **Uwaga** : Na powyższych przykładach widzimy, że wektory, macierze oraz tensory wygodnie jest przedstawiać w postaci list. Odpowiednia konstrukcja listy doskonale odzwierciedla każdy z tych obiektów.

Budując listy, możemy użyć operatorów **Range** oraz **Array**. Poniżej znajduje się kilka przykładów ich użycia.

```
Range[9]
```

```
{1, 2, 3, 4, 5, 6, 7, 8, 9}
```

```
Range[3, 11]
```

```
{3, 4, 5, 6, 7, 8, 9, 10, 11}
```

```
Range[x, x + 4]
```

```
{x, 1 + x, 2 + x, 3 + x, 4 + x}
```

```
Range[5.2, 6.8, .2]
```

```
{5.2, 5.4, 5.6, 5.8, 6., 6.2, 6.4, 6.6, 6.8}
```

```
Array[g, 6]
```

```
{g[1], g[2], g[3], g[4], g[5], g[6]}
```

```
Array[h, 6, 3]
```

```
{h[3], h[4], h[5], h[6], h[7], h[8]}
```

```
Array[f, {2, 3}, {9, 7}] // MatrixForm
```

$$\begin{pmatrix} f[9, 7] & f[9, 8] & f[9, 9] \\ f[10, 7] & f[10, 8] & f[10, 9] \end{pmatrix}$$

Często wygodniej jest definiować macierz w innej formie. Przytrzymujemy klawisz CTRL i naciskamy ENTER lub przecinek - tworzymy w ten sposób odpowiednio wiersze oraz kolumny. Powstaje tabela, którą wystarczy tylko wypełnić wyrazami (poruszamy się po niej klawiszami kierunków lub klawiszem TAB).

```
□ □ □
□ □ □
```

```
{{□, □, □}, {□, □, □}}
```

```
11 12 13
21 22 23
```

```
{{11, 12, 13}, {21, 22, 23}}
```

Stosowanie "czystych" funkcji w połączeniu z **Array** daje nam więcej możliwości tworzenia list - patrz rozdział *Funkcje w czystej postaci.*

■ 1.2.2. Dodawanie i usuwanie elementów, łączenie list

Od tego momentu, w najbliższych kilku rozdziałach, będziemy podawać jedynie przykłady użycia operatorów. Czytelnik powinien bez problemu zrozumieć ich działanie.

```
A1 = {a, b, c, d, e};
```

```
{Append[A1, x], Prepend[A1, x], Insert[A1, x, 2]}
```

```
{{a, b, c, d, e, x}, {x, a, b, c, d, e}, {a, x, b, c, d, e}}
```

```
{Drop[A1, {2, 3}], Drop[A1, -3], ReplacePart[A1, x, 3]}
```

```
{{a, d, e}, {a, b}, {a, b, x, d, e}}
```

```
Join[A1, 2 A1, 3 A1]
```

```
{a, b, c, d, e, 2 a, 2 b, 2 c, 2 d, 2 e, 3 a, 3 b, 3 c, 3 d, 3 e}
```

■ 1.2.3. Wybieranie elementów listy

```
A1 = {a, b, c, d, e};
```

```
{First[A1], Last[A1], Most[A1], Rest[A1]}
```

```
{a, e, {a, b, c, d}, {b, c, d, e}}
```

```
{Take[A1, -2], Take[A1, {3}], Take[A1, {2, 4}]}
```

```
{{d, e}, {c}, {b, c, d}}
```

```
{A1[[2]], A1[[-2]], A1[[{1, 3, 5}]], A1[[2 ;; 5]]}
```

```
{b, d, {a, c, e}, {b, c, d, e}}
```

```
M1 = Table[k A1, {k, 3}]
```

```
{{a, b, c, d, e},
 {2 a, 2 b, 2 c, 2 d, 2 e}, {3 a, 3 b, 3 c, 3 d, 3 e}}
```

```
{M1[[All, 3]], M1[[3, 2]]}
```

```
{{c, 2 c, 3 c}, 3 b}
```

■ 1.2.4. Sortowanie, zmiana kolejności, transpozycja

```
A1 = {a, b, c, d, e};
A2 = {1, 2, 3, 4, 5};
```

```
M2 = {RotateLeft[A1, 2], RotateRight[A1, 3]}
```

```
{{c, d, e, a, b}, {c, d, e, a, b}}
```

```
{Sort[A2], Sort[A2, Greater], Reverse[A1]}
```

```
{{1, 2, 3, 4, 5}, {5, 4, 3, 2, 1}, {e, d, c, b, a}}
```

```
Transpose[M2] // MatrixForm
```

$$\begin{pmatrix} c & c \\ d & d \\ e & e \\ a & a \\ b & b \end{pmatrix}$$

Identyczny wynik otrzymamy wpisując po nazwie macierzy: ESC **tr** ESC.

```
M2^T
```

```
{{c, c}, {d, d}, {e, e}, {a, a}, {b, b}}
```

■ 1.2.5. Listy jako zbiory

Listy możemy traktować jako zbiory. *Mathematica*® posiada wbudowane operatory odpowiadające działaniom na zbiorach.

```
{X = {a, b, c}, Y = {b, c}, Z = {c, d, e}}
```

```
{{a, b, c}, {b, c}, {c, d, e}}
```

```
{Union[X, Z], Intersection[X, Z]}
```

```
{{a, b, c, d, e}, {c}}
```

```
{Complement[Z, Y], Complement[Y, X]}
```

```
{{d, e}, {}}
```

```
Subsets[Z]
```

```
{{}, {c}, {d}, {e}, {c, d}, {c, e}, {d, e}, {c, d, e}}
```

⚠ **Uwaga** : Możemy także używać symboli działań ⋃ (ESC**un**ESC) oraz ⋂ (ESC**inter**ESC).

Operator **Union** może być użyty w przypadku procedury kolekcjonowania kolejnych wyników. Poniżej znajduje sie prosty przykład skonstruowany przy pomocy pętli **For**.

```
M = {};
For[n = 1, n ≤ 100, n++, If[Mod[n, 7] == 0, M = M ⋃ {n}]];
M
```

```
{7, 14, 21, 28, 35, 42, 49, 56, 63, 70, 77, 84, 91, 98}
```

Czyli stworzyliśmy w ten sposób listę wszystkich liczb nauralnych z przedziału [1, 100], które są podzielne przez 7.

■ 1.2.6. Inne operacje na listach

```
A3 = {x, x, z, y, z, z, y, z}
```

```
{x, x, z, y, z, z, y, z}
```

```
{Length[A3], Tally[A3]}
```

```
{8, {{x, 2}, {z, 4}, {y, 2}}}
```

```
{Partition[A3, 2], Split[A3]}
```

```
{{{x, x}, {z, y}, {z, z}, {y, z}},
 {{x, x}, {z}, {y}, {z, z}, {y}, {z}}}
```

```
Flatten[%]
```

```
{x, x, z, y, z, z, y, z, x, x, z, y, z, z, y, z}
```

```
DeleteDuplicates[A3]
```

```
{x, z, y}
```

```
Join[%, {a, b, c, d}]
```

```
{x, z, y, a, b, c, d}
```

Poniższy przykład przedstawia róznicę między **Join** a poznanym wcześniej **Union**.

```
{Union[{a}, {a}], Join[{a}, {a}]}
{{a}, {a, a}}
```

Zauważmy, że zazwyczaj nasze listy nie zmieniały swojej postaci, ponieważ użyte operatory dają w wyniku przeważnie nową listę. Poniższy prosty przykład pokazuje, w jaki sposób można zmieniać postać już istniejącej listy.

```
A4 = {a, b, c, d};
A4 = Drop[A4, -2]; A4
{a, b}
```

■ 1.3. Manipulacja wyrażeniami

Pokażemy, w jaki sposób wykorzystać operatory, które stosowaliśmy w przypadku list. W rozdziale *Wybieranie elementów listy* używaliśmy operatora **Part** (**[[.]]**), który wydobywa z list wskazane elementy. Przeanalizujmy jego działanie.

```
{x, y, z} // FullForm
List[x, y, z]
%[[2]]
y
```

Czyli wybieramy w ten sposób z wyrażenia **List[x,y,z]** drugie z kolei podwyrażenie. Sprawdźmy, że procedura ta funkcjonuje nie tylko w przypadku list. Rozpatrzmy następujący przykład. Z wyrażenia

```
{xzy → abc}
{xzy → abc}
```

chcemy "wydobyć" symbol **abc**. Zapiszmy **{xzy→abc}** w pełnej formie.

```
% // FullForm
List[Rule[xzy, abc]]
```

Musimy zatem najpierw dostać się do wyrażenia **Rule[xzy,abc]**. Jest to pierwszy (mimo, że jedyny) element listy. Następnie wskazujemy w tym wyrażeniu drugie podwyrażenie.

```
{xzy → abc}[[1]] // FullForm
Rule[xzy, abc]
%[[2]]
abc
```

Cała operacja zamyka się w poniższym poleceniu.

```
{xzy → abc}[[1, 2]]
```

```
abc
```

Warto odnotować fakt, że użycie **[[0]]** zwraca głowę wyrażenia.

```
Sin[x][[0]]
```

```
Sin
```

W rozdziałach *Wzorce* oraz *Funkcje w czystej postaci* zostaną omówione bardziej zaawansowane metody wyboru i usuwania elementów list przy użyciu operatorów **Cases**, **DeleteCases** oraz **Select**.

2. Wzorce, funkcje oraz reguły

2.1. Wzorce

Każde wyrażenie w *Mathematica*® można skojarzyć z pewnym wzorcem. W celu sprawdzenia, czy dane wyrażenie pasuje do wzorca użyjemy operatora **MatchQ**. Jako przykład rozpatrzmy sumę **x+1**.

```
x + 1
```

```
1 + x
```

Sprawdźmy, czy **x+1** spełnia wzorzec: "*suma wyrażeń* **x** *oraz* **1**".

```
MatchQ[x + 1, x + 1]
```

```
True
```

Możemy także określić inne wzorce, używając znaku podkreślenia **_** (**Blank[]**), który jest podstawowym elementem rodziny wzorców i oznacza "dowolne wyrażenie".

```
MatchQ[x + 1, _] (*wzorzec: pewne wyrażenie*)
```

```
True
```

```
MatchQ[x + 1, 1 + _] (*wzorzec: suma 1 i pewnego wyrażenia*)
```

```
True
```

W celu tworzenia dokładniejszych wzorców możemy użyć kontrukcji **n_**

```
n_ // FullForm
```

```
Pattern[n, Blank[]]
```

która oznacza pewne wyrażenie z etykietą **n**, zawarte we wzorcu. Wówczas

```
MatchQ[x + 1, x_ + y_] (*wzorzec: suma pewnych dwóch wyrażeń*)
```

```
True
```

W danym wzorcu możemy użyć dodatkowych ograniczeń, np. w stosunku do głowy wyrażenia. Konstrukcje **_head** oraz **n_head** oznaczają odpowiednio wyrażenie z głową **head** oraz wyrażenie z głową **head** opatrzone etykietą **n**.

```
{x + 1 // Head, x // Head, 1 // Head}
```

```
{Plus, Symbol, Integer}
```

```
MatchQ[x + 1, _Plus]
```

```
True
```

```
MatchQ[x + 1, _Integer + _Symbol]
```

```
True
```

MatchQ należy do rodziny tzw. *wyrazeń testujących*, które zwracają wartości **True** (prawda) albo **False** (fałsz). Za pomocą konstrukcji **PatternTest** występującej w postaci

wzorzec?wyrażenie_testujące,

narzucamy kolejne wymagania dopasowania wyrażenia. Poniżej znajdują sie przykładowe funkcje testujące oraz ich zastosowanie w **PatternTest**.

```
{OddQ[3], PrimeQ[4]}
```

```
{True, False}
```

```
{MatchQ[x + 1, x_ + y_?OddQ], MatchQ[3, _Integer?PrimeQ]}
```

```
{True, True}
```

Podobnymi konstrukcjami do **Blank[]** są **BlankSequence[]** oraz **BlankNullSequence[]** (odpowiednio dwa oraz trzy znaki podkreślenia). Pierwsza z nich oznacza dowolną sekwencję wyrażeń, natomiast druga dopuszcza dodatkowo "brak wyrażeń".

```
{MatchQ[{a, b, c}, {__}], MatchQ[{}, {__}]}
```

```
{True, False}
```

```
{MatchQ[{a, b, c}, {___}], MatchQ[{}, {___}]}
```

```
{True, True}
```

Konstruując wzorce, możemy używać warunku (**Condition /;**) w formie

wzorzec/;warunek

Wówczas dopasowanie wyrażenia do wzorca wymaga dodatkowo spełnienia określonego warunku.

```
{MatchQ[3, x_ /; x ≥ 0], MatchQ[symb1, x_ /; Head[x] == Symbol]}
```

```
{True, True}
```

Z wykorzystaniem **BlankSequence[]** oraz **BlankNullSequence[]** możemy tworzyć bardziej skomplikowane wzorce. Przykładem niech będzie wzorzec listy, w której:
1) występują symbole **a** oraz **b**,
2) **a** poprzedza **b**,
3) między **a** i **b** znajduje się co najmniej jeden element.

```
A1 = {a, b, 0, 0, 0, 0, 0};
```

```
B1 = Table[RandomSample[A1], {10}]
```

```
{{0, 0, b, 0, 0, a, 0}, {0, b, a, 0, 0, 0, 0},
 {a, 0, 0, b, 0, 0, 0}, {b, 0, 0, 0, 0, 0, a},
 {0, 0, a, b, 0, 0, 0}, {0, a, 0, 0, 0, b, 0},
 {0, a, 0, b, 0, 0, 0}, {a, 0, 0, 0, b, 0, 0},
 {0, 0, a, 0, 0, b, 0}, {0, b, a, 0, 0, 0, 0}}
```

Za pomocą operatora **Cases** wybierzmy z **B1** wszystkie listy spełniające ustalone wyżej warunki.

```
Cases[B1, {___, a, __, b, ___}]
```

```
{{a, 0, 0, b, 0, 0, 0},
 {0, a, 0, 0, 0, b, 0}, {0, a, 0, b, 0, 0, 0},
 {a, 0, 0, 0, b, 0, 0}, {0, 0, a, 0, 0, b, 0}}
```

Możemy ustalić regułę (→), według której będą transformowane wyrażenia pasujące do wzorca. Dla przykładu wyznaczmy odległości między **a** i **b** w listach spełniających nasz wzorzec.

```
Cases[B1, {x___, a, y__, b, z___} → {y}]
```

```
{{0, 0}, {0, 0, 0}, {0}, {0, 0, 0}, {0, 0}}
```

```
Length /@ %
```

```
{2, 3, 1, 3, 2}
```

Konstrukcja **Cases** może mieć szerokie zastosowanie. Z dowolnej listy możemy wybierać elementy spełniające pewne warunki - wystarczy tylko utworzyć odpowiedni wzorzec. Przeciwne działanie ma **DeleteCases**, który usuwa wyrażenia spełniające dany warunek.

```
DeleteCases[B1, {___, a, __, b, ___}]
```

```
{{0, b, 0, a, 0, 0, 0}, {b, 0, 0, 0, 0, 0, a},
 {0, 0, 0, b, 0, 0, a}, {0, 0, 0, 0, a, b, 0}}
```

Natomiast **Count** podaje liczbę elementów pasujących do wzorca.

```
Count[B1, {___, a, __, b, ___}]
```

```
6
```

DeleteCases działa w przypadku dowolnego wyrażenia. Możliwe jest także określenie poziomu działania operatora.

```
DeleteCases[a^x - Sin[2 x], x, 2]
```

```
a - Sin[2 x]
```

```
{1, a^w, f[y], 3, g[x], (b + c)^v};
```

```
{Cases[%, x_[y_] → x], Cases[%, x_^y_ → x]}
```

```
{{f, g}, {a, b + c}}
```

2.2. Funkcje

Funkcją w *Mathematica*® możemy nazwać operację transformacji wyrażenia spełniającego dany wzorzec w nowe wyrażenie.

wyrażenie (spełniające dany wzorzec)
↓
Funkcja[wyrażenie]

Np. funkcja

```
f1[x_] := x^2
```

dowolnemu (wzorzec **x_**) wyrażeniu przypisuje jego kwadrat.

```
{f1[3], f1[wyrazenie]}
```

$\{9, \text{wyrazenie}^2\}$

Zdefiniujmy ciąg $a(n) = \frac{n-1}{n}$, $n = 1, 2, 3 \ldots$.

```
Clear[a]
test[x_] := TrueQ[x > 0];
a[n_Integer?test] := (n - 1)/n
{a[0], a[1], a[3], a[Sqrt[2]]}
```

$\{a[0], 0, \frac{2}{3}, a[\sqrt{2}]\}$

Podobny skutek otrzymamy przy użyciu **Condition** (**/;**) oraz konstrukcji **If**. Przy czym pierwszy przypadek jest lepszy, ponieważ konstrukcja oparta na **If** zwróci **Null**, jeżeli nie zostanie spełniony warunek.

```
Clear[a]
a[n_] := (n - 1)/n /; IntegerQ[n] ∧ n > 0
{a[0], a[1], a[3], a[Sqrt[2]]}
```

$\{a[0], 0, \frac{2}{3}, a[\sqrt{2}]\}$

```
Clear[a]
a[n_] := If[IntegerQ[n] ∧ n > 0, (n - 1)/n]
```

```
{a[0], a[1], a[3], a[Sqrt[2]]}
```

$$\left\{\text{Null}, 0, \frac{2}{3}, \text{Null}\right\}$$

Dokonajmy teraz kolejno następujących definicji funkcji **f**

```
Remove[f]
f[x_] := x^2
f[1/2] = 3;
f[x_] := 2 /; x < 0 ∧ x > -2
f[x_Symbol] := x
```

Wyznaczmy teraz wartości funkcji dla kilku argumentów

```
{f[1/3], f[1/2], f[-1/2], f[symb]}
```

$$\left\{\frac{1}{9}, 3, 2, \text{symb}\right\}$$

oraz sprawdźmy, w jaki sposób jest teraz zdefiniowana.

```
? f
```

```
Global`f

f[1/2] = 3

f[x_] := 2 /; x < 0 && x > -2

f[x_Symbol] := x

f[x_] := x^2
```

Kolejność elementów powyższej listy odpowiada kolejności, według której *Mathematica*® sprawdza, czy argument spełnia dany wzorzec bądź warunek. Program stara się aby ta kolejność odpowiadała precyzyjności wzorców, jednak nie zawsze potrafi to dobrze ocenić: problem może pojawić sie np. gdy używamy **PatternTest**. Zdefiniujmy wartości funkcji **f** dla wszystkich elementów całkowitych (czyli z głową **Integer**).

```
Clear[f]
f[x_Integer] := x/2
```

Załóżmy, że w trakcie pracy z funkcją chcemy zmienić jej formę, tak aby każdej liczbie typu **Numeric** przypisana została wartość równa 1. Naturalnym odruchem byłoby wykonanie przypisania

```
f[x_?NumericQ] := 1
```

jednak zauważmy, że

```
f[3]
```

$$\frac{3}{2}$$

ponieważ

```
? f
```

```
Global`f
```

```
f[x_Integer] := x/2
```

```
f[x_?NumericQ] := 1
```

Jak widzimy powyżej, spełnienie wzorca dla typu **Integer** jest sprawdzane wcześniej niż dla **Numeric**. Należy więc być ostrożnym podczas definiowania funkcji ten sposób i czasem najbezpieczniejszym wyjściem jest definiowanie funkcji na nowo.

Możemy także definiować funkcje wielu zmiennych. Jest to wtedy przypisanie sekwencji wyrażeń (czyli pewnemu wzorcowi!) określonego wyrażenia.

```
g[x_, y_, z_] := 2 x + y z
```

```
g[t1, t2, t3]
```

```
2 t1 + t2 t3
```

Za pomocą **BlankSequence[]** oraz **BlankNullSequence[]** możemy budować funkcję o nieokreślonej liczbie argumentów (działającej także w przypadku ich braku).

```
Clear[f]
```

```
f[v___] := {v} // Length
```

```
{f[1, Plot[x, {x, 0, 1}], 1/2, "tekst"], f[]}
```

```
{4, 0}
```

■ 2.2.1. Funkcje z lokalnymi zmiennymi

W trakcie pracy w dokumencie może zaistnieć potrzeba wykonania obliczeń z lokalnymi zmiennymi (nie kolidującymi z tymi już używanymi). Możemy wówczas zastosować moduł **Module**.

```
x = 1;
Module[{x, x0 = 2}, Print[x]; x = x0; x]
```

```
x$7558
2
```

```
{x, x0 // Head}
```

```
{1, Symbol}
```

Jak widzimy, wartość **x** nie zmieniła się, a **x0** pozostaje symbolem wolnym od wartości. Lista w powyższej konstrukcji zawiera symbole lokalnych zmiennych, przy czym **x0** posiada wartość początkową. Jak widzimy, nazwa symbolu **x** została zmieniona w odrębie **Module**. Dzięki temu zabiegowi możliwe jest wykonanie operacji, bez kolizji ze zmiennymi globalnymi.

Natomiast operator **Block** uwalnia wskazane symbole od wartości, lecz po wykonaniu obliczeń na powrót im je przypisuje.

```
Block[{x, x0 = 2}, Print[{x, x0 // Head}]; x = x0; x]
```

```
{x, Integer}
2
```

```
{x, x0 // Head}
```

```
{1, Symbol}
```

Nazwy symboli pozostają niezmienione, zatem **Block** można użyć do lokalnej zmiany parametrów wbudowanych w *Mathematica*® (czytelnik znajdzie odpowiedni przykład z parametrem **$RecursionLimit** w rozdziale *Poprawne definiowanie funkcji*). Podstawową różnicę pomiędzy powyższymi konstrukcjami przedstawia następujący przykład.

```
Clear[x, x0]; x = x0;
{Block[{x0 = 1}, x], Module[{x0 = 1}, x],
 Module[{x0 = 1}, x = x0; x]}
```

```
{1, x0, 1}
```

Czyli w **Module** zmiany w pewnym wyrażeniu, zależnym od lokalnej zmiennej, zachodzą tylko wówczas, gdy występuje ona w nim w sposób jawny.

Możemy skorzystać z powyższych konstrukcji w celu definiowania funkcji, w której musi zostać wykonana procedura bądź ich szereg. Realizujemy to według wzoru

```
funkcja[x1_,..,xk_]:=Module[{wart_lokalne},procedura1;
procedura2; wartość]
```

W rozdziale *Wizualizacje oraz moduły dynamiczne* poznamy moduł (**DynamicModule**) ograniczający zasięg zmiennych dynamicznych.

■ 2.2.2. Poprawne definiowanie funkcji

```
Clear[f1, f2]
```

```
f1[x_] := x^2
```

Powyższą funkcję **f1** zdefiniowaliśmy za pomocą **SetDealyed** (**:=**). Możemy także definiować funkcje przy użyciu **Set** (**=**).

```
f2[x_] = x^2;
```

Różnica polega na tym, iż w pierwszym przypadku lewa strona jest obliczana za każdym razem, natomiast w drugim - wartość dla dowolnego wyrażenia **x** obliczona zostaje w momencie zdefiniowania i od tego momentu używana jest w tej postaci. Poniższe przykłady pozwolą zrozumieć istotę obydwu definicji.

Skorzystamy z wiadomości z rozdziału *Różniczkowanie i całkowanie*. Zdefiniujmy funkcje $h(x) = \sin(x)$ oraz $f(x) = h'(x)$.

```
Clear[h, f]
```

```
h[x_] := Sin[x]
```

```
f[x_] := D[h[x], x]
```

Zauważmy, że obliczając wartość w konkretnym punkcie otrzymamy błąd.

```
f[1]
```

```
∂1 Sin[1]
```

```
General::ivar: 1 is not a valid variable. >>
```

Spowodowane jest to tym, iż powyższa definicja nakazuje najpierw podstawić liczbę 1 w miejsce **x** i dopiero wówczas obliczyć wartość. Zatem otrzymujemy wyrażenie **D[Sin[1],1]** o błędnej składni. Możemy wówczas użyć operatora **Evaluate** lub zdefiniować funkcję za pomocą **Set** (**=**).

```
Clear[f1, f2]
```

```
{f1[x_] := Evaluate[D[h[x], x]], f2[x_] = D[h[x], x]};
```

```
{f1[1], f2[1]}
```

```
{Cos[1], Cos[1]}
```

Jednak definiowanie w ten sposób funkcji niesie za sobą pewne konsekwencje. Zmieńmy postać funkcji *h*

```
h[x_] := x^2
```

i obliczmy ponownie wartości **f1[1]** oraz **f2[1]**. Czytelnik zauważy, że straciliśmy zależność wyrażeń od funkcji *h*. Aby rozwiązać ten problem dobrze jest posłużyć się definicją za pomocą operatorów **Module** oraz **ReplaceAll** (patrz rozdział *Reguły*).

```
Clear[f3]
f3[x_] := Module[{x0}, D[h[x0], x0] /. x0 → x]

h[x_] := Cos[x]; f3[2]

-Sin[2]

h[x_] := x^3; f3[2]

12
```

Jednak istnieją sytuacje, w których lepiej użyć **Set** (**=**). Zdefiniujmy następujące funkcje.

```
Remove[f1, f2]

f1[x_] := ∫_0^x Cos[t] dt

f2[x_] = ∫_0^x Cos[t] dt;
```

Obliczmy teraz 1000 wartości funkcji i operatorem **Timing** zmierzmy czas pracy.

```
Table[f1[x], {x, 1000}]; // Timing

{32.579, Null}

Table[f2[x], {x, 1000}]; // Timing

{4.30211 × 10^-15, Null}
```

Jak widzimy, różnica w czasie potrzebnym do obliczenia wyrażeń jest znaczna, mimo iż otrzymujemy ten sam wynik.

Przykładem użycia **SetDelayed** jest definicja rekurencyjna. Jednak i tutaj musimy być ostrożni. Zdefiniujmy następujący ciąg

$$a_n = \begin{cases} 1, & n < 3 \\ a_{n-1} - 2\, a_{n-2}, & n \geq 3 \end{cases}$$

```
a // Clear
a[1] = 1; a[2] = 2;
a[n_] := a[n - 1] - 2 a[n - 2]
```

Zauważmy, jak długo trwa obliczenie **a[35]**.

```
a[35] // Timing

{36.031, -167 028}
```

Dzieje się tak z powodu niewłaściwego zdefiniowania funkcji. Prześledźmy początek obliczania **a[35]** według powyższej definicji.

$$a_{35} = a_{34} - 2\,a_{33} \qquad \text{obliczenie wyrazu } a_{33}$$
$$= a_{33} - 2\,a_{32} \qquad \text{ponowne obliczenie } a_{33}$$

Zatem już w tym momencie wyraz **a[33]** będzie obliczany dwukrotnie (wynika to z faktu definiowania funkcji za pomocą **SetDelayed** (**:=**)), a kolejne kroki będą zwiększać liczbę niepotrzebnych obliczeń. Możemy zmodyfikować powyższą definicję, niejako wplatając w nią **Set** (**=**) w celu przechowywania w pamięci obliczonych już wyrazów ciągu.

```
a1 // Clear
a1[1] = 1; a1[2] = 2;
a1[n_] := a1[n] = a1[n - 1] - 2 a1[n - 2]
```

wówczas

```
a1[35] // Timing
```

```
{0., -167 028}
```

Jednak, chcąc obliczyć wyraz o indeksie 270, otrzymamy następujący komunikat

```
a1[270]
```

```
$RecursionLimit::reclim: Recursion depth of 256 exceeded. ≫

$RecursionLimit::reclim: Recursion depth of 256 exceeded. ≫

$RecursionLimit::reclim: Recursion depth of 256 exceeded. ≫

General::stop:
  Further output of $RecursionLimit::reclim will be suppressed during this
      calculation. ≫
```

informujący o tym, iż domyślna liczba wykonywanych kroków rekurencyjnych w programie jest równa 256. Odpowiada za to parametr **$RecursionLimit**, który możemy lokalnie zmienić na potrzeby obliczeń, poprawiając powyższą definicję.

```
a1rec[n_] := Block[{$RecursionLimit = ∞}, a1[n]]
```

```
a1rec[1500]
```

```
-5 179 812 730 506 852 004 970 483 975 422 720 003 284 590 239 675 ⁖
  154 595 254 794 685 738 328 042 933 477 807 123 086 708 266 197 477 ⁖
  897 587 514 063 152 522 167 220 827 804 627 086 883 345 230 076 850 ⁖
  941 062 297 625 269 647 835 924 880 942 196 117 200 904 323 656 377 ⁖
  376 782 374 733 262 189 722 812
```

■ 2.2.3. Definiowanie funkcji wielonormowo

Przedstawimy trzy sposoby definiowania funkcji, które przyjmują różne wartości w zależności od tego, jaki wzorzec spełnia dane wyrażenie. Posłużmy sie następującym przykładem

$$h(x) = \begin{cases} \cos(x) - 1, & \left|\cos(x)\right| \geq \frac{1}{2} \\ \cos(5\,x), & \left|\cos(x)\right| < \frac{1}{2} \end{cases}$$

Korzystamy z konstrukcji **Piecewise**: używamy skrótu ESC **pw** ESC, a następnie przytrzymując CTRL naciskamy ENTER. Pojawią się wówczas dwa wiersze zawierające dwa okienka; kolejny wiersz dodajemy, powtarzając tę operację. W pierwszym okienku wiersza wpisujemy przepis funkcji, a w drugim dziedzinę, w której ma ten przepis działać - postępujemy w ten sposób w przypadku każdego wiersza. Jeżeli jeden ze wzorów ma działać "dla pozostałych wartości", to wówczas wystarczy w drugim okienku wpisać **True**.

```
Clear[h1, h2, h3]
h1[x_] := { Cos[x] - 1   Abs[Cos[x]] ≥ 1 / 2
          { Cos[5 x]           True
```

Czasem można posłużyć się znanym już sposobem definiowania

```
h2[x_] := Cos[x] - 1 /; Abs[Cos[x]] ≥ 1 / 2
h2[x_] := Cos[5 x]
```

lub operatorem **Which**, w którym podajemy na przemian warunki oraz odpowiadające im wartości funkcji.

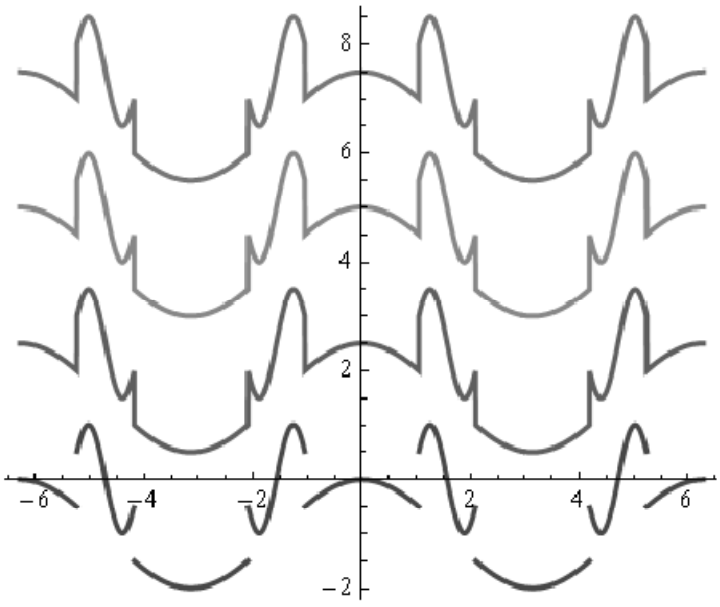
```
h3[x_] := Which[Abs[Cos[x]] ≥ 1 / 2,
  Cos[x] - 1, Abs[Cos[x]] < 1 / 2, Cos[5 x]]
```

Także konstrukcja **If** może być wykorzystana.

```
h4[x_] := If[Abs[Cos[x]] ≥ 1 / 2, Cos[x] - 1, Cos[5 x]]
```

Przedstawmy wykresy funkcji **h1**, **h2+2.5**, **h3+5** oraz **h4+7.5** (patrz rozdział *Wykresy*).

```
Plot[{h1[x], h2[x] + 2.5, h3[x] + 5, h4[x] + 7.5},
 {x, -2 π, 2 π}, AspectRatio → Automatic, PlotStyle → Thick]
```



Jak widzimy, tylko w przypadku pierwszej definicji (**h1**) wykres nie jest rysowany jako wykres funkcji ciagłej.

■ 2.2.4. Funkcje w czystej postaci

W powyższych przykładach używaliśmy funkcji, które posiadają nazwę (czyli zostały zdefiniowane przed użyciem). Funkcje w czystej postaci (*pure functions*) pozwalają zastosować operację bez potrzeby definiowania nowej funkcji.

Omówimy jednargumentową funkcję w czystej postaci. Za przykład posłuży nam funkcja $a \mapsto a^2 - 2$. Można to zrealizować na kilka równoważnych sposobów.

```
Function[a, a^2 - 2][t]
Function[#^2 - 2][t]
(#^2 - 2) & [t]
```

$$-2 + t^2$$

$$-2 + t^2$$

$$-2 + t^2$$

Znak **#** oznacza tzw. *slot* - symbolizuje argument funkcji.

Analogicznie używamy funkcji wielu zmiennych. Wówczas **#n** oznacza n-ty argument funkcji

```
{#1, f[#2 + #3]} &[x, y, z]
```
```
{x, f[y + z]}
```

Oczywiście czystej funkcji możemy nadać nazwę - otrzymamy wówczas zwykłą funkcję.

```
f1 = Function[a, a^2 - 2];
```
```
f1[t]
```
```
-2 + t^2
```

Dlaczego funkcja w czystej postaci jest czasem wygodniejsza? Załóżmy, że chcemy jakąś funkcję zastosować tylko raz. Wówczas nie ma potrzeby budowania nowej definicji, lepiej więc użyć jej w czystej postaci. Zmniejszamy tym samym zużycie pamięci operacyjnej (nie jest przetrzymywana informacja o jej nazwie i postaci), zwiększamy przejrzystość kodu, szybkość działania programu itd. Ma to szczególne znaczenie przy bardziej skomplikowanych aplikacjach.

Funkcje w czystej postaci mogą służyć np. do budowy wyrażeń testujących, o których mowa w rozdziale *Wzorce*.

```
-2 // (# > 0) &
```
```
False
```
```
MatchQ[2/3, _Rational?(# > 0 &)]
```
```
True
```

Wyrażeń testujących możemy używać wraz z operatorem **Select**, który jest podobny w działaniu do **Cases** oraz **DeleteCases**. Operator ten zwróci wszystkie elementy listy, dla których wybrane wyrażenie testujące zwróci wartość **True**. Jako prosty przykład, wyodrębnimy spośród 20 losowo wybranych liczb z kwadratu $\{z \in \mathbb{C} : |\operatorname{re} z| < 2 \wedge |\operatorname{im} z| < 2\}$ te, które należą do zbioru $\{z \in \mathbb{C} : |\operatorname{Re} z| < 1 \wedge |\operatorname{Im} z| < 1\}$.

```
T1 = Table[RandomComplex[{-2 - 2 i, 2 + 2 i}], {20}];
```
```
Select[T1, Abs[Re[#]] < 1 ⋀ Abs[Im[#]] < 1 &]
```
```
{-0.270723 - 0.912682 i,
 0.276839 - 0.227072 i, 0.593817 - 0.853945 i}
```

W połączeniu z operatorem **Array**, funkcje w czystej postaci pozwalają generować różnego rodzaju listy.

```
Array[{#1^2, √#2} &, {3, 2}, {2, 2}]
```
```
{{{4, √2}, {4, √3}},
 {{9, √2}, {9, √3}}, {{16, √2}, {16, √3}}}
```
```
Array[#1 - #2 &, {3, 4}]
```
```
{{0, -1, -2, -3}, {1, 0, -1, -2}, {2, 1, 0, -1}}
```

```
Array[1 + #^2 &, 10]
```

```
{2, 5, 10, 17, 26, 37, 50, 65, 82, 101}
```

```
Array[(#1 + #2)/#3 &, {2, 3, 4}, {1, 2, 1}]
```

$$\left\{\left\{\left\{3, \frac{3}{2}, 1, \frac{3}{4}\right\}, \left\{4, 2, \frac{4}{3}, 1\right\}, \left\{5, \frac{5}{2}, \frac{5}{3}, \frac{5}{4}\right\}\right\},\right.$$
$$\left.\left\{\left\{4, 2, \frac{4}{3}, 1\right\}, \left\{5, \frac{5}{2}, \frac{5}{3}, \frac{5}{4}\right\}, \left\{6, 3, 2, \frac{3}{2}\right\}\right\}\right\}$$

W kolejnym rozdziale poznamy operatory pozwalające stosować funkcje do elementów list, ułatwiające i przyspieszające pracę z programem *Mathematica*®.

■ 2.2.5. Operacje funkcyjne na listach

Sprawdźmy, jaki wynik otrzymamy, jeżeli użyjemy listy jako argumentu funkcji **Sin**.

```
Sin[{x, y, z}]
```

```
{Sin[x], Sin[y], Sin[z]}
```

W tym przypadku wartość funkcji została obliczona dla każdego elementu listy, zwracając nową listę. Takie zachowanie operatora jest możliwe tylko w przypadku operatorów posiadających atrybut **Listable**.

```
Attributes[Sin]
```

```
{Listable, NumericFunction, Protected}
```

Załóżmy, że mamy funkcję **f**, która takiego atrybutu nie posiada. Możemy wówczas nadać jej tę cechę.

```
{f[{x, y, z}], Attributes[f]}
```

```
{f[{x, y, z}], {}}
```

```
Attributes[f] = {Listable}
```

```
{Listable}
```

```
f[{x, y, z}]
```

```
{f[x], f[y], f[z]}
```

Aby "oczyścić" symbol z atrybutów po prostu przypisujemy mu ich brak.

```
{Attributes[f] = {}, Attributes[f]}
```

```
{{}, {}}
```

Mathematica® posiada wbudowane operatory przekształcające listę argumentów na listę wartości pewnej funkcji. Jednym z takich operatorów jest **Map**.

```
Map[f, {a, b, c}]
{f[a], f[b], f[c]}
```

Istnieje również inna forma użycia tego operatora.

```
f /@ {a, b, c}
{f[a], f[b], f[c]}
```

Atrybut można przypisać jedynie do symbolu (nazwy funkcji), dlatego też **Map** jest szczególnie przydatny w przypadku funkcji występującej w czystej postaci.

```
{(g[#] + 3 #) & /@ {a, b, c}, Function[x, 2 x] /@ {a, b, c}}
{{3 a + g[a], 3 b + g[b], 3 c + g[c]}, {2 a, 2 b, 2 c}}
```

```
f[#[[1]]] g[#[[2]]] & /@ {{a, b}, {c, d}}
{f[a] g[b], f[c] g[d]}
```

Operator **Map** domyślnie działa na pierwszym poziomie listy, jednak pozwala stosować funkcję także na dowolnym poziomie tabeli.

```
Map[f, {{1, 2}, {3, 4}}, {2}]
{{f[1], f[2]}, {f[3], f[4]}}
```

```
Map[f, {{1, 2}, {3, 4}}, 2]
{f[{f[1], f[2]}], f[{f[3], f[4]}]}
```

Znany nam już operator **Apply** zmienia "głowę" wyrażenia. Występuje on także w dwóch równoważnych formach.

```
{Apply[f, g[x]], f @@ g[x]}
{f[x], f[x]}
```

Opcjonalnie możemy ustalić poziom działania operatora.

```
{Apply[f, (a + b)/Sqrt[x + y], {2}], Plus @@@ (x^a + y^b)}
```

$$\left\{\frac{a+b}{\sqrt{f[x,\,y]}},\ a+b+x+y\right\}$$

```
Apply[f, {a[x], {b[y], c[z]}}, {2}]
{a[x], {f[y], f[z]}}
```

Poniżej przedstawiono przykłady działania kilku innych operatorów o podobnym charakterze.

```
Through[h[p, q][x, y]]
h[p[x, y], q[x, y]]
```

```
{Fold[f, x, {a, b, c}], FoldList[f, x, {a, b}]}
```

```
{f[f[f[x, a], b], c], {x, f[x, a], f[f[x, a], b]}}
```

```
Sqrt@Rest@FoldList[Plus, 0, {a, b, c, d, e}]
```

$$\left\{\sqrt{a},\ \sqrt{a+b},\ \sqrt{a+b+c},\ \sqrt{a+b+c+d},\ \sqrt{a+b+c+d+e}\right\}$$

```
{Nest[f, x, 3], NestList[f, x, 3]}
```

```
{f[f[f[x]]], {x, f[x], f[f[x]], f[f[f[x]]]}}
```

```
{Inner[f, {a, b}, {c, d}, g], Outer[f, {a, b}, {x, y, z}]}
```

```
{g[f[a, c], f[b, d]],
 {{f[a, x], f[a, y], f[a, z]}, {f[b, x], f[b, y], f[b, z]}}}
```

Jako przykład zbudujmy następującą listę reguł

$$\{\{a_1 \to b_2\}, \{a_2 \to b_3\}, \{a_3 \to b_4\}, \{a_4 \to b_5\}, \{a_5 \to b_6\}\}.$$

```
Outer[#1_#2 &, {a, b}, Range[6]]
```

```
{{a1, a2, a3, a4, a5, a6}, {b1, b2, b3, b4, b5, b6}}
```

```
Inner[Rule, Most[%[[1]]], Rest[%[[2]]], List]
```

```
{a1 → b2, a2 → b3, a3 → b4, a4 → b5, a5 → b6}
```

■ 2.3. Reguły

Przedstawimy teraz konstrukcję często używaną podczas pracy z *Mathematica*® - **ReplaceAll**. Rozpatrzmy następujący przykład.

```
rep1 = 3 x y^2 + Sqrt[x]
```

$$\sqrt{x} + 3\,x\,y^2$$

```
ReplaceAll[rep1, x → b]
```

$$\sqrt{b} + 3\,b\,y^2$$

Jak łatwo zauważyć, dokonuje on w wyrażeniu podstawienia według podanej reguły (**Rule**) **x→b**. Istnieje krótsza postać tego polecenia przy użyciu symboli **/.**

```
rep1 /. x → 3 + y
```

$$\sqrt{3+y} + 3\,y^2\,(3+y)$$

Zauważmy, że samo wyrażenie **rep1** nie zostało zmienione.

```
rep1
```

$$\sqrt{x} + 3\,x\,y^2$$

Możemy dokonywać transformacji według kilku reguł. Wówczas podajemy je w postaci listy.

```
rep1 /. {y → z, x → z^2}
```

$$3\,z^4 + \sqrt{z^2}$$

Prześledźmy działanie **ReplaceAll** w zależności od sposobu użycia.

```
{x, y} /. {x → y, y → x}
```

```
{y, x}
```

```
{x, y} /. x → y /. y → x
```

```
{x, x}
```

```
f[x] /. x → {1, 2, 3, 4, 5}
```

```
f[{1, 2, 3, 4, 5}]
```

```
f[x] /. Table[{x → k}, {k, 1, 5}]
```

```
{f[1], f[2], f[3], f[4], f[5]}
```

```
{x, y} /. {{List → Divide},
  {List → Subscript}, {List → Times, x → {a, b}}}
```

$$\left\{\frac{x}{y},\ x_y,\ \{a\,y,\ b\,y\}\right\}$$

Oparatorem o podobnym sposobie działania jest **RuleDealyed** (**a:→b**). Różnica polega na tym, że wyrażenie **b** jest obliczane ilekroć **a** wystąpi w wyrażeniu. Poniższy, bardzo prosty przykład pokazuje różnicę między tym wyrażeniem a **ReplaceAll**.

```
{x, x, x} /. x → Random[]
```

```
{0.274247, 0.274247, 0.274247}
```

```
{x, x, x} /. x :→ Random[]
```

```
{0.559243, 0.445375, 0.129247}
```

⚠ **Uwaga** : Symbol **:→** wstawiamy za pomocą ESC **:>** ESC.

ReplaceAll stosuje regułę (badź reguły) tylko raz, natomiast jego uogólnienie w postaci **ReplaceRepeated** (**//.**) pozwala zapętlać operację podstawienia. Poniższy przykład pokazuje, że należy go ostrożnie używać.

```
{x, y} //. {x → y, y → x}
```

```
ReplaceRepeated::rrlim: Exiting after {x, y} scanned 65536 times. ≫
```

```
{x, y}
```

Obliczanie zapętliło się i w końcu zostało przerwane przez program. Możemy jednak sterować liczbą powtórzeń podstawienia, co daje nam możliwość budowania różnych konstrukcji.

```
ReplaceRepeated[Sqrt[x] + Sqrt[y]
  , {x -> 1/(x + Sqrt[y]), y -> 1/(y + Sqrt[x])}, MaxIterations → 3] // Quiet
```

$$\sqrt{\frac{1}{\frac{1}{x+\sqrt{y}}+\sqrt{\frac{1}{\sqrt{x}+y}}}+\sqrt{\frac{1}{\sqrt{\frac{1}{x+\sqrt{y}}}+\frac{1}{\sqrt{x}+y}}}}+\sqrt{\frac{1}{\sqrt{\frac{1}{x+\sqrt{y}}+\sqrt{\frac{1}{\sqrt{x}+y}}}+\frac{1}{\sqrt{\frac{1}{x+\sqrt{y}}}+\frac{1}{\sqrt{x}+y}}}}$$

Możemy poddawać zmianom fragmenty wyrażenia odpowiadające danemu wzorcowi.

```
Clear[g, a, b, c, d]
```

```
(a + b^3)/Sqrt[x + y] Sqrt[g] /. {a_ + b_^3 → a^3 + b, z_Power → z^2/2}
```

$$\frac{\left(a^3+b\right)\,g}{4\,(x+y)}$$

```
ReplaceAll[Permutations[{1, 2, 3}],
  {a_, b_, c_} -> D[f[x, y], {x, a}, {y, b}]]
```

```
{f^(1,2)[x, y], f^(1,3)[x, y], f^(2,1)[x, y],
  f^(2,3)[x, y], f^(3,1)[x, y], f^(3,2)[x, y]}
```

W jednym z poprzednich rozdziałów definiowaliśmy ciąg

$$a_n = \begin{cases} 1, & n < 3 \\ a_{n-1} - 2\,a_{n-2}, & n \geq 3 \end{cases}$$

w następujący sposób

```
Clear[a1, a2]
```

```
a1[1] = 1; a1[2] = 2;
a1[n_] := a1[n] = a1[n - 1] - 2 a1[n - 2]
```

Teraz możemy podać inną konstrukcję opartą na poznanych operatorach.

```
a2[n_] := ReplaceRepeated[{2, 1},
    {x_, y_} → {x - 2 y, x}, MaxIterations → n - 2][[1]] // Quiet
```

```
{a1[6], a2[6]}
```

```
{4, 4}
```

Zauważmy, że definicja ta jest lepsza pod tym względem, iż nie wymaga zmiany parametru **$RecursionLimit**

```
a2[350]
```

```
49 351 612 571 641 687 571 271 335 738 600 549 301 448 525 773 761 188
```

oraz nie pozostawia niepotrzebnych wyników w pamięci.

```
?a1
```

```
Global`a1
```

```
a1[1] = 1

a1[2] = 2

a1[3] = 0

a1[4] = -4

a1[5] = -4

a1[6] = 4

a1[n_] := a1[n] = a1[n - 1] - 2 a1[n - 2]
```

```
?a2
```

```
Global`a2
```

```
a2[n_] := Quiet[ReplaceRepeated[{2, 1},
    {x_, y_} → {x - 2 y, x}, MaxIterations → n - 2][[1]]]
```

■ 2.4. Funkcja - Sumowanie według danego zbioru wskaźników

Wbudowany w *Mathematica*® operator sumowania ma postać

```
∑_i a_i // FullForm
```

```
Sum[Subscript[a, i], i]
```

```
Sum[a_i, {i, {x, y, z, t, u}}]
```

```
a_t + a_u + a_x + a_y + a_z
```

Opierając się na tym spostrzeżeniu, utwórzmy sumę

$$\sum_{s,t \in \text{Freq}} a_{s,t} \cosh(s\,x + t\,y),$$

gdzie Freq jest zbiorem wskaźników w postaci par (s, t).

```
Sum[
 Function[{s, t}, a[s, t] Cosh[s x + t y]] @@ st,
 {st, Freq}]
```

$$\frac{1}{2}\,\text{Freq}\,(1+\text{Freq})$$

Opeartor **@@** (**Apply**) zamienia głowę wyrażenia **List[s,t]** według schematu

```
List → Function[{s, t}, a[s, t] Cosh[s x + t y]],
```

czyli otrzymujemy wyrażenie

```
Function[{s, t}, a[s, t] Cosh[s x + t y]][a, b]
```

```
a[a, b] Cosh[a x + b y]
```

Zatem dla

```
Freq = {{1, 1}, {3, 2}, {-1, 0}, {0, 1}};
```

mamy

```
Sum[Function[{s, t}, a[s, t] Cosh[s x + t y]] @@ st, {st, Freq}]
```

```
a[-1, 0] Cosh[x] + a[0, 1] Cosh[y] +
 a[1, 1] Cosh[x + y] + a[3, 2] Cosh[3 x + 2 y]
```

Zbudujemy teraz operator **SumEach**, który będzie sumował wyrażenia według dowolnego zbioru wskaźników. W pierwszej kolejności określamy sumowanie po indeksach z całego zbioru **Integer**.

```
ClearAll[SumEach];
Attributes[SumEach] = {HoldFirst};

SumEach[expr_, i_Symbol ∈ Integers] := Sum[expr, {i, -∞, ∞}];
```

Dowolnemu wyrażeniu (wzorzec **expr_**) oraz konstrukcji spełniającej wzorzec

```
i_Symbol ∈ Integers
```

przypisuje sumę po prawej stronie.

```
fs[x_] := { 1/2^x   x > 0
          { 0       True

SumEach[fs[a], a ∈ Integers]
```

```
1
```

Następnie definiujemy sumę dla wskaźników zawartych w wektorze (**{a1,a2,a3,...,ak}**).

```
SumEach[expr_, i_Symbol ∈ iSet_List?VectorQ] := Module[{p},
   Plus @@ Map[(Hold@expr /. i → #) &, iSet] // ReleaseHold];
```

Prześledźmy działanie tego operatora. Niech

```
expr = (1 + a)/a^2; set = Range[6]
```

```
{1, 2, 3, 4, 5, 6}
```

Operator **Hold** pozostawia wyrażenie w formie nieobliczonej.

```
Map[(Hold@(1 + a)/a^2 /. a → #) &, set]
```

$$\left\{\text{Hold}\left[\frac{1+1}{1^2}\right], \text{Hold}\left[\frac{1+2}{2^2}\right], \text{Hold}\left[\frac{1+3}{3^2}\right],\right.$$
$$\left.\text{Hold}\left[\frac{1+4}{4^2}\right], \text{Hold}\left[\frac{1+5}{5^2}\right], \text{Hold}\left[\frac{1+6}{6^2}\right]\right\}$$

Po wykonaniu podstawienia następuje zmiana głowy powyższego wyrażenia na **Plus** (w celu otrzymania sumy).

```
Plus @@ %
```

$$\text{Hold}\left[\frac{1+1}{1^2}\right] + \text{Hold}\left[\frac{1+2}{2^2}\right] + \text{Hold}\left[\frac{1+3}{3^2}\right] +$$
$$\text{Hold}\left[\frac{1+4}{4^2}\right] + \text{Hold}\left[\frac{1+5}{5^2}\right] + \text{Hold}\left[\frac{1+6}{6^2}\right]$$

W tym momencie zostaje zakończone działanie operatora **Hold** oraz obliczenie sumy wyrażenia.

```
% // ReleaseHold
```

$$\frac{14\,189}{3600}$$

Kolejnym krokiem będzie zdefiniowanie sumowania po listach indeksów.

```
SumEach[expr_, i_?(VectorQ[#, MatchQ[#, _Symbol] &] &) ∈
    iSet_List?MatrixQ] :=
 Module[{p},
   Plus @@ Map[(Hold@expr /. Thread[i → #]) &, iSet] //
    ReleaseHold
  ] /; Length[i] == Length[First@iSet]
```

W tym przypadku wzorcem jest

wektor symboli ∈ macierz indeksów,

natomiast warunek na końcu definicji dopuszcza wyrażenia do obliczenia tylko wówczas, gdy liczba elementów listy **i** jest równa liczbie kolumn macierzy **iSet**. Działanie operatora różni się od poprzedniego przypadku użyciem **Thread** w celu podstawiania indeksów do wyrażenia **expr**. Powoduje to powstanie listy reguł, w sposób podobny do poniższego.

```
Thread[{x, y, z} → #] &@{a, b, c}
```

```
{x → a, y → b, z → c}
```

```
set = Table[k {a, b, c}, {k, 3}]
```

```
{{a, b, c}, {2 a, 2 b, 2 c}, {3 a, 3 b, 3 c}}
```

```
Map[(Hold@expr[x, y, z] /. Thread[{x, y, z} → #]) &, set]
```

```
{Hold[expr[a, b, c]],
 Hold[expr[2 a, 2 b, 2 c]], Hold[expr[3 a, 3 b, 3 c]]}
```

Teraz możemy powrócić do przykładu rozpatrywanego na początku rozdziału i obliczyć żądaną sumę za pomocą jednego polecenia.

```
SumEach[a[s, t] Cosh[s x + t y], {s, t} ∈ Freq]
```

```
a[-1, 0] Cosh[x] + a[0, 1] Cosh[y] +
 a[1, 1] Cosh[x + y] + a[3, 2] Cosh[3 x + 2 y]
```

3. Zagadnienia matematyczne

3.1. Przekształcanie i upraszczanie wyrażeń

3.1.1. Upraszczanie wyrażeń

Funkcja **Simplify** wykonuje szereg przekształceń nad danym wyrażeniem i zwraca jego najprostszą formę, jaką jest w stanie znaleźć. Obejmuje standardowe algebraiczne i trygonometryczne tożsamości (te ostatnie możemy "wyłączyć").

```
w1 = 1/(3 (1 + x)) - (-1 + 2 x)/(6 (1 - x + x^2)) + 2/(3 (1 + 1/3 (-1 + 2 x)^2));
```

```
% // Simplify
```

$$\frac{1}{1 + x^3}$$

```
w2 = 8 Cos[x]^4 - 8 Cos[x]^2 + 1;
```

```
{Simplify[w2], Simplify[w2, Trig → False]}
```

$$\{\text{Cos}[4\,x],\ 1 - 8\,\text{Cos}[x]^2 + 8\,\text{Cos}[x]^4\}$$

Natomiast **FullSimplify** wykorzystuje szerszy zakres przekształceń (zawiera m. in. funkcje elementarne i specjalne).

```
{Simplify[x Gamma[x]], FullSimplify[x Gamma[x]]}
```

$$\{x\,\text{Gamma}[x],\ \text{Gamma}[1 + x]\}$$

Opcja **ExcludedForms** pozwala wskazać wzorzec wyrażeń, do których *Mathematica*® nie zastosuje uproszczeń.

```
Simplify[2 Sin[x] Cos[x] + w1, ExcludedForms → Sin[x_]]
```

$$\frac{1 + 2\left(1 + x^3\right)\text{Cos}[x]\,\text{Sin}[x]}{1 + x^3}$$

3.1.2. Rozwijanie wyrażeń

Rozważmy następujące wyrażenie.

```
w3 = (1 + x)^2/(1 - x) + 3 x^2/(1 + x)^2 + (2 - x)^2;
```

Operator **Expand** rozwija iloczyny bądź potęgi w wyrażeniu. Istnieje kilka jego odmian. **ExpandNumerator** i **ExpandDenominator** rozwijają odpowiednio tylko liczniki i mianowniki każdego składnika w wyrażeniu.

```
{ExpandNumerator[w3], ExpandDenominator[w3]}
```

$$\left\{4 - 4\,x + x^2 + \frac{3\,x^2}{(1+x)^2} + \frac{1 + 2\,x + x^2}{1 - x},\ (2 - x)^2 + \frac{(1+x)^2}{1-x} + \frac{3\,x^2}{1 + 2\,x + x^2}\right\}$$

Polecenie **Expand** rozwija każdy licznik oraz rozkłada wyrażenie na ułamki proste, natomiast **ExpandAll** rozwija zarówno liczniki, jak i mianowniki wyrażenia.

```
{Expand[w3], ExpandAll[w3]}
```

$$\left\{4 + \frac{1}{1-x} - 4\,x + \frac{2\,x}{1-x} + x^2 + \frac{x^2}{1-x} + \frac{3\,x^2}{(1+x)^2},\ 4 + \frac{1}{1-x} - 4\,x + \frac{2\,x}{1-x} + x^2 + \frac{x^2}{1-x} + \frac{3\,x^2}{1 + 2\,x + x^2}\right\}$$

Możemy żądać, aby zostały rozwinięte tylko wyrażenia zawierające pewien wzorzec.

```
{Expand[(x + 1)^3 + (z - 3)^2, z],
 Expand[(Sqrt[x + 1] + Sqrt[x])^2 + (z - 3)^3, Sqrt[_]]}
```

$$\left\{9 + (1+x)^3 - 6\,z + z^2,\ 1 + 2\,x + 2\sqrt{x}\,\sqrt{1+x} + (-3+z)^3\right\}$$

Expand i **ExpandAll** różnią się także tym, że tylko **ExpandAll** działa na podwyrażeniach.

```
{Expand[Sqrt[(1 + x)^2]], ExpandAll[Sqrt[(1 + x)^2]]}
```

$$\left\{\sqrt{(1+x)^2},\ \sqrt{1 + 2\,x + x^2}\right\}$$

Po użyciu opcji **Trig→True** możliwe jest rozwijanie wyrażeń trygonometrycznych.

```
Expand[Sin[x + y], Trig → True]
```

```
Cos[y] Sin[x] + Cos[x] Sin[y]
```

■ 3.1.3. Rozkład na czynniki

Teraz skupimy się na funkcji **Factor**. Rozkłada ona liczniki i mianowniki wyrażeń na czynniki o współczynnikach całkowitych. Możemy także rozszerzyć zbiór, do którego należą współczynniki, o wymierne kombinacje liczby algebraicznej. Możliwe jest także rozkładanie na czynniki wyrażeń trygonometycznych (wówczas należy użyć opcji **Trig → True**).

```
{Factor[x^2 - 1], Factor[Sin[x] + Sin[y], Trig → True]}
```

$$\left\{(-1+x)\,(1+x),\ 2\,\text{Cos}\left[\frac{x}{2}-\frac{y}{2}\right]\,\text{Sin}\left[\frac{x}{2}+\frac{y}{2}\right]\right\}$$

```
Factor[1 + x^4, Extension -> {Sqrt[2], I}]
```

$$\frac{1}{4}\left(\sqrt{2}-(1+i)\,x\right)\left(\sqrt{2}-(1-i)\,x\right)\left(\sqrt{2}+(1-i)\,x\right)\left(\sqrt{2}+(1+i)\,x\right)$$

Przy opcji **Extension→Automatic**, *Mathematica*® użyje rozszerzenia na ciało pokrywające współczynniki wielomianu.

■ 3.1.4. Wyrażenia trygonometryczne

Omówimy teraz sposoby przekształcania wyrażeń zawierających funkcje trygonometryczne. Wiemy już, że jest to możliwe w przypadku **Simplify**, **FullSimplify** oraz **Factor** (w tym ostatnim należy użyć opcji **Trig→True**). Odpowiednikami operatorów **Expand** oraz **Factor** dla wyrażeń trygonometrycznych są **TrigExpand** i **TrigFactor**.

```
TrigExpand[Sin[2 x] Cos[2 y]]
```

$$2\,\text{Cos}[x]\,\text{Cos}[y]^2\,\text{Sin}[x]-2\,\text{Cos}[x]\,\text{Sin}[x]\,\text{Sin}[y]^2$$

```
TrigFactor[8 Cos[x]^4 - 8 Cos[x]^2 + 1]
```

$$2\,\text{Sin}\left[\frac{\pi}{4}-2\,x\right]\,\text{Sin}\left[\frac{\pi}{4}+2\,x\right]$$

Aby przekształcić w pewnym wyrażeniu funkcje trygonometryczne do funkcji wykładniczych przy podstawie e, użyjemy **TrigToExp**. Natomiast do wykonania operacji odwrotnej - **ExpToTrig**.

```
{TrigToExp[Cos[x] + I Sin[x]], ExpToTrig[Sin[x + y] E^(I x)]}
```

$$\left\{e^{i\,x},\ \text{Cos}[x]\,\text{Sin}[x+y]+i\,\text{Sin}[x]\,\text{Sin}[x+y]\right\}$$

■ 3.1.5. Inne przydatne operatory

Rozważmy następujące wyrażenie.

```
w5 = (x^2/(x^2 - 1) + x/(x^2 - 1)) (x^3 - 1)/(x - 1)
```

$$\frac{\left(-1+x^3\right)\left(\frac{x}{-1+x^2}+\frac{x^2}{-1+x^2}\right)}{-1+x}$$

Operację sprowadzania do wspólnego mianownika możemy wykonać za pomocą **Together**. Natomiast **Apart** zapisuje wyrażenie jako sumę składników, których mianowniki występują w jak najprostszej postaci. Skracanie wspólnych czynników licznika i mianownika możemy wykonać operatorem **Cancel**.

```
{Together[w5], Apart[w5], Cancel[w5]}
```

$$\left\{\frac{x\left(1+x+x^2\right)}{-1+x},\ 3+\frac{3}{-1+x}+2x+x^2,\ \frac{x\left(1+x+x^2\right)}{-1+x}\right\}$$

Załóżmy, że mamy wyrażenie, w którym pojawiają się różne symbole zmiennych. Wówczas operator **Apart** może być zastosowany ze względu na tylko jedną z nich.

```
Apart[(x^2 + y^2)/(x + x y), x]
```

$$\frac{x}{1+y}+\frac{y^2}{x\,(1+y)}$$

Operator **PowerExpand** wykonuje przekształcenie typu $(x\,y)^a \to x^a\,y^a$.

```
PowerExpand[Sqrt[x y]]
```

$$\sqrt{x}\ \sqrt{y}$$

■ 3.1.6. Założenia

Czasem uproszczenie wyrażenia jest możliwe tylko pod warunkiem spełnienia konkretnego założenia. W przypadku **Simplify** i **FullSimplify** dodajemy je po wyrażeniu przekształcanym.

```
w4 = Sqrt[x^2] ;
{Simplify[w4], Simplify[w4, x > 0], Simplify[w4, x ∈ Reals]}
```

$$\left\{\sqrt{x^2},\ x,\ \text{Abs}[x]\right\}$$

⚠ **Uwaga** : Funkcja **Abs** określa moduł liczby zespolonej, czyli w szczególnym przypadku wartość bezwzględną liczby rzeczywistej.

Polecenie **FunctionExpand** pozwala także rozwijać wyrażenia przy odpowiednich założeniach.

```
{FunctionExpand[Log[x y]],
 FunctionExpand[Log[x y], x > 0 && y > 0]}
{Log[x y], Log[x] + Log[y]}
```

Dzięki **Refine** możemy podać postać, jaką przyjmie wyrażenie, jeżeli symbole w nim zawarte będą spełniać określone założenia.

```
{w4, Refine[w4, x < 0]}
```

$$\left\{\sqrt{x^2}\,,\ -x\right\}$$

```
{Sin[k Pi], Refine[Sin[k Pi], k ∈ Integers]}
```

```
{Sin[kπ], 0}
```

Sprawdźmy, czy dana równość zachodzi gdy spełniony jest pewien warunek.

```
Refine[a^2 - b^2 + 1 == 0, a + b == -1/(a - b)]
```

```
True
```

Omówmy krótko w jakiej formie wprowadzamy założenia. Fakt, że element **x** (lub elementy **{a,b,c}**) należy do pewnego zbioru zapisujemy **Element[x,X]** (odpowiednio **Element[{a,b,c}, X]**). Istnieją równoważne formy: **x∈X , {a,b,c}∈X** (symbol **∈** wprowadzamy za pomocą ESC**el**ESC). Przykładami zbiorów są: **Complexes** (liczby zespolone), **Reals** (rzeczywiste), **Algebraics** (algebraiczne), **Rationals** (wymierne), **Integers** (całkowite). Symbole tych zbiorów możemy zapisać za pomocą ESC**dsX**ESC, gdzie w miejsce **X** wstawiamy odpoowiednio: **C**, **R**, **A**, **Q**, **I**.
Przy korzystaniu z założeń oczywistym jest używanie symboli **>**, **<**, **≥**, **≤** (ESC**=>**ESC, ESC**<=**ESC) oraz spójników logicznych **∧** (ESC**&&**ESC), **∨** (ESC**or**ESC).

■ 3.1.7. Operator Reduce

Za pomocą operatora **Reduce** możemy redukować wyrażenia do prostszej postaci poprzez rozwiązywanie równań i nierówności oraz eliminację zmiennych.

```
Reduce[Abs[z] < 2 , z, Complexes]
```

$$-2 < \text{Re}[z] < 2 \;\&\&\; -\sqrt{4 - \text{Re}[z]^2} < \text{Im}[z] < \sqrt{4 - \text{Re}[z]^2}$$

```
% /. {Re[z] → x, Im[z] → y}
```

$$-2 < x < 2 \;\&\&\; -\sqrt{4 - x^2} < y < \sqrt{4 - x^2}$$

```
Reduce[{x^2 + 3 x - 1 > 0, x^2 + x - 5 < 0}, x]
```

$$\frac{1}{2}\left(-3 + \sqrt{13}\right) < x < \frac{1}{2}\left(-1 + \sqrt{21}\right)$$

W rozdziale *Równania i układy równań* przedstawimy dalsze zastosowania **Reduce**.

▪ 3.2. Granice

Mathematica® w pewnym stopniu umożliwia obliczanie granic wyrażeń.

```
{Limit[(1 + f[x]/n)^n, n → Infinity], Limit[Sin[x]/x, x → 0]}
```

```
{e^f[x], 1}
```

Istnieje również możliwość użycia opcji **Direction** w celu wyznaczenia granic jednostronnych.

```
{Limit[Tan[x], x → π/2, Direction → 1],
 Limit[Tan[x], x → π/2, Direction → -1]}
```

```
{∞, -∞}
```

Poniżej obliczamy granice wyrażenia przy pewnych założeniach.

```
Limit[x^c, x → ∞, Assumptions → #] & /@ {c < 0, c > 0, c == 0}
```

```
{0, ∞, 1}
```

Jako zastosowanie operatora **Limit** znajdziemy asymptotę ukośną funkcji $f(x) = \frac{x^3}{x^2-5}$.

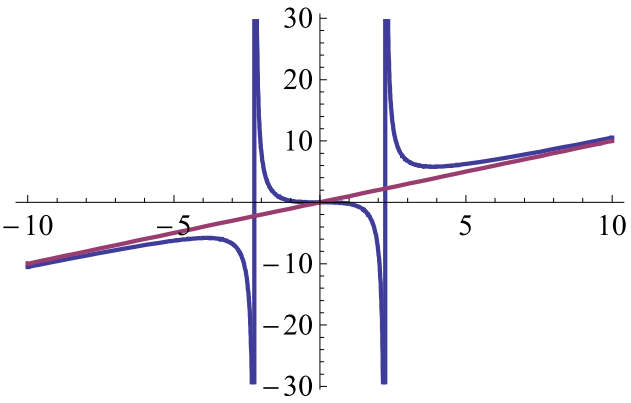
```
Clear[f]
```

```
f[x_] := x^3/(x^2 - 5)
```

```
{a = Limit[f[x]/x, x → ∞], b = Limit[f[x] - a x, x → ∞]}
```

```
{1, 0}
```

```
Plot[{f[x], a x + b}, {x, -10, 10}]
```



W pewnych przypadkach **Limit** zwraca przedział możliwych wartości.

```
Limit[Sin[x], x → Infinity]
```
```
Interval[{-1, 1}]
```

■ 3.3. Macierze

Zgodnie z treścią rozdziału *Listy,* odpowiednia konstrukcja list może reprezentować macierz. Pokażemy na przykładach, w jaki sposób wykonywać najprostsze operacje na macierzach. Obliczmy wyznacznik (**Det**) oraz wartości własne (**Eigenvalues**) pewnej macierzy.

```
A = {{-1, 2, 3}, {2, -1, 3}, {-1, 3, 2}}
```
```
{{-1, 2, 3}, {2, -1, 3}, {-1, 3, 2}}
```
```
% // MatrixForm
```
$$\begin{pmatrix} -1 & 2 & 3 \\ 2 & -1 & 3 \\ -1 & 3 & 2 \end{pmatrix}$$
```
{Det[A], Eigenvalues[A]}
```
```
{12, {4, -3, -1}}
```

Poniżej wyznaczone zostały (w formie macierzowej) odpowiednio: transpozycja macierzy **A** (**A**ESC**tr**ESC), macierz odwrotna do **A**, suma, iloczyn macierzy oraz macierz kwadratów elementów z **A**.

```
MatrixForm /@ {Transpose[A], Inverse[A], α A + A, A.A, A^2}
```
$$\left\{\begin{pmatrix} -1 & 2 & -1 \\ 2 & -1 & 3 \\ 3 & 3 & 2 \end{pmatrix}, \begin{pmatrix} -\frac{11}{12} & \frac{5}{12} & \frac{3}{4} \\ -\frac{7}{12} & \frac{1}{12} & \frac{3}{4} \\ \frac{5}{12} & \frac{1}{12} & -\frac{1}{4} \end{pmatrix},\right.$$
$$\left.\begin{pmatrix} -1-\alpha & 2+2\alpha & 3+3\alpha \\ 2+2\alpha & -1-\alpha & 3+3\alpha \\ -1-\alpha & 3+3\alpha & 2+2\alpha \end{pmatrix}, \begin{pmatrix} 2 & 5 & 9 \\ -7 & 14 & 9 \\ 5 & 1 & 10 \end{pmatrix}, \begin{pmatrix} 1 & 4 & 9 \\ 4 & 1 & 9 \\ 1 & 9 & 4 \end{pmatrix}\right\}$$

■ 3.4. Szeregi

Aby wyznaczyć szereg Taylora stopnia *n* funkcji *f* w punkcie *a* użyjemy konstrukcji **Series[f[x],{x,a,n}]**.

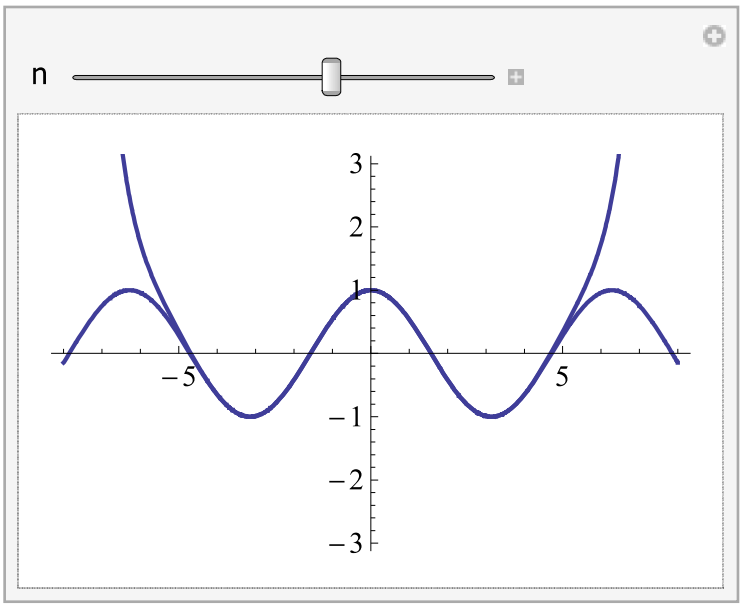
```
Series[Cos[x], {x, 0, 10}]
```
$$1 - \frac{x^2}{2} + \frac{x^4}{24} - \frac{x^6}{720} + \frac{x^8}{40\,320} - \frac{x^{10}}{3\,628\,800} + O[x]^{11}$$

```
% // Normal
```

$$1 - \frac{x^2}{2} + \frac{x^4}{24} - \frac{x^6}{720} + \frac{x^8}{40\,320} - \frac{x^{10}}{3\,628\,800}$$

Poniżej możemy zaobserwować w jaki sposób przybliżana jest funkcja wielomianami kolejnych stopni (informacje dotyczące konstruowania wizualizacji zawarte są w rozdziale *Wizualizacje oraz moduły dynamiczne*).

```
Manipulate[
 Show[Plot[Series[Cos[x], {x, 0, n}] // Normal // Evaluate,
   {x, -8, 8}], Plot[Cos[x], {x, -8, 8}],
  PlotRange → {-3, 3}], {n, 1, 20, 1}]
```



■ 3.4.1. Szeregi Fouriera

Podajmy rozwinięcie stopnia 5 w szereg Fouriera funkcji $f(x) = x^2$ na przedziale $[-\pi, \pi]$. W tym celu użyjemy operatora **FourierSeries**.

```
f[x_] := x^2
FourierSeries[f[x], x, 5]
```

$$-2\,e^{-i\,x} - 2\,e^{i\,x} + \frac{1}{2}\,e^{-2\,i\,x} + \frac{1}{2}\,e^{2\,i\,x} - \frac{2}{9}\,e^{-3\,i\,x} -$$

$$\frac{2}{9}\,e^{3\,i\,x} + \frac{1}{8}\,e^{-4\,i\,x} + \frac{1}{8}\,e^{4\,i\,x} - \frac{2}{25}\,e^{-5\,i\,x} - \frac{2}{25}\,e^{5\,i\,x} + \frac{\pi^2}{3}$$

Domyślnie rozwinięcie stopnia n jest definiowane jako

$$\sum_{k=-n}^{n} c_k\, e^{i\,k\,t},\ \text{gdzie}\ c_k = \frac{1}{2\,\pi}\int_{-\pi}^{\pi} f(t)\, e^{-i\,k\,t}\, dt.$$

Opcja **FourierParameters→{a,b}** pozwala zmienić wyjściowe rozwinięcie na

$$\left|\frac{b}{2\,\pi}\right|^{\frac{1-a}{2}} \sum_{k=-n}^{n} c_k\, e^{i\,b\,k\,t},$$

gdzie

$$c_k = \left|\frac{b}{2\,\pi}\right|^{\frac{a+1}{2}} \int_{-\frac{\pi}{|b|}}^{\frac{\pi}{|b|}} f(t)\, e^{-i\,b\,k\,t}\, dt.$$

Współczynnik o indeksie *n* rozwinięcia w szereg Fouriera znajdujemy dzięki **FourierCoefficient**.

```
FourierCoefficient[f[x], x, n]
```

$$\frac{2\,(-1)^n}{n^2}$$

Dzięki poleceniom **FourierSinCoefficient**, **FourierCosCoefficient** wyznaczamy współczynniki rozwinięcia w szereg według odpowiednio sinusów i cosinusów.

```
FourierSinCoefficient[f[x], x, n]
```

$$-\frac{2\left(2-2\,(-1)^n+(-1)^n\,n^2\,\pi^2\right)}{n^3\,\pi}$$

```
FourierCosCoefficient[f[x], x, n]
```

$$\frac{4\,(-1)^n}{n^2}$$

W celu utworzenia rozwinięcia funkcji w trygonometryczny szereg Fouriera określonego stopnia użyjemy **FourierTrigSeries**.

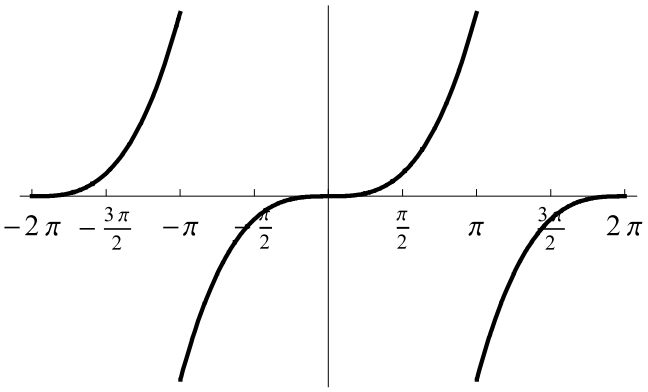
```
FourierTrigSeries[(x - π)^2, x, 6]
```

$$\frac{4\,\pi^2}{3} + 4\left(-\text{Cos}[x] + \frac{1}{4}\,\text{Cos}[2\,x] - \frac{1}{9}\,\text{Cos}[3\,x] + \frac{1}{16}\,\text{Cos}[4\,x] - \frac{1}{25}\,\text{Cos}[5\,x] + \frac{1}{36}\,\text{Cos}[6\,x]\right) + 4\,\pi\left(-\text{Sin}[x] + \frac{1}{2}\,\text{Sin}[2\,x] - \frac{1}{3}\,\text{Sin}[3\,x] + \frac{1}{4}\,\text{Sin}[4\,x] - \frac{1}{5}\,\text{Sin}[5\,x] + \frac{1}{6}\,\text{Sin}[6\,x]\right)$$

Zaobserwujmy, w jaki sposób następująca funkcja przybliżana jest za pomocą szeregu Fouriera.

```
g[x_] := x^3
```

```
Plot[g[ x - Round[ x, 2 π ]], {x, -2 π, 2 π},
 Ticks → {Table[k π / 2, {k, -4, 4}], None},
 Exclusions → {-π, π}, PlotStyle → {Thick, Black}]
```



Utwórzmy najpierw listę wykresów rozwinięć kolejnych stopni. Umożliwi nam ona płynną wizualizację (wówczas rozwinięcia kolejnych stopni nie będą obliczane na bieżąco). Zauważmy, że sposób wykonania, który może wydawać się najprostszy, zajmuje sporo czasu (już przy 15 wykresach).

```
FTS1 = Table[Plot[FourierTrigSeries[g[x], x, n] // Evaluate,
      {x, -2 π, 2 π}], {n, 1, 15}]; // Timing
```

```
{48.39, Null}
```

Dlatego skorzystamy z wiadomości zawartych w rozdziale *Poprawne definiowanie funkcji*. Obliczmy wzór na współczynnik rozwinięcia o indeksie n.

```
c[n_] = FourierCoefficient[g[x], x, n]
```

$$\frac{i\,(-1)^n\left(-6+n^2\pi^2\right)}{n^3}$$

Zdefiniujmy ciąg kolejnych rozwinięć w następujacy sposób.

```
fts[x_, 0] = FourierCoefficient[g[x], x, 0];
fts[x_, n_] :=
 fts[x, n] = fts[x, n - 1] + c[n] e^(i n x) + c[-n] e^(-i n x)
```

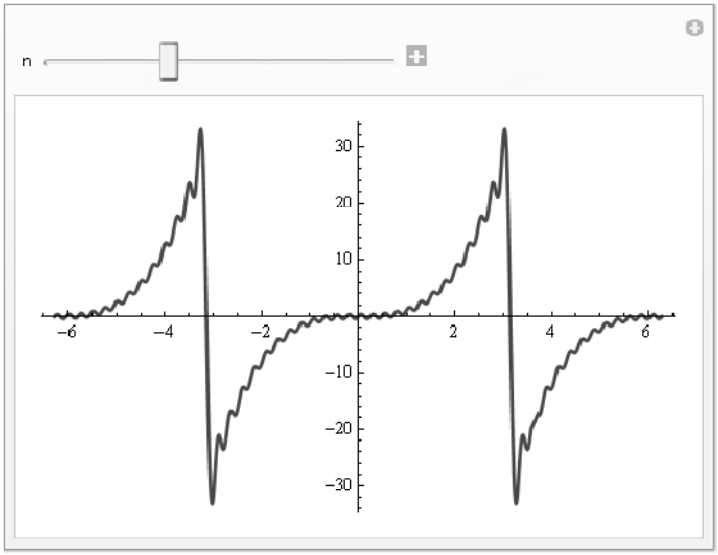
Utwórzmy listę 70 wykresów. Zauważmy, że zajmie to mniej czasu niż w przypadku listy **FTS1**.

```
FTS = Table[Plot[fts[x, n] // Evaluate, {x, -2 π, 2 π}],
    {n, 1, 70}]; // Timing
```

```
{17.312, Null}
```

Wówczas możemy przystąpić do wizualizacji. Mamy do czynienia z tzw. efektem Gibbsa (patrz Część II).

```
Manipulate[FTS[[n]], {n, 1, 70, 1}]
```



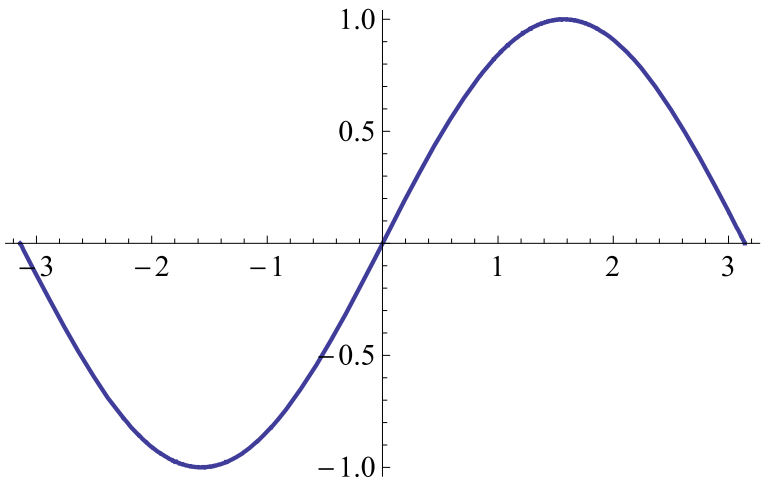
■ 3.5. Wykresy

W rozdziale tym przybliżymy problematykę generowania różnego rodzaju wykresów za pomocą *Mathematica*® z uwzględnieniem tylko niektórych opcji ich formatowania, wystarczających do tworzenia czytelnych ilustracji. Więcej funkcji dotyczących grafiki poznamy w rozdziale *Grafika*.

■ 3.5.1. Wykres funkcji $f : \mathbb{R} \to \mathbb{R}$

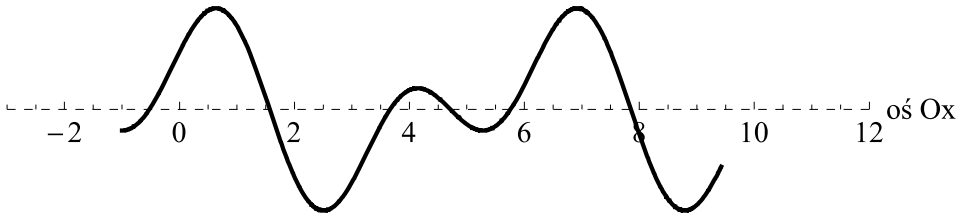
Poleceniem **`Plot`** generujemy wykresy funkcji na pewnym przedziale. Dla przykładu przedstawmy wykres funkcji sinus na przedziale $[-\pi, \pi]$.

```
Plot[Sin[x], {x, -π, π}]
```



Wygląd powyższej ilustracji jest domyślny, jednak istnieje możliwość jego zmiany poprzez określenie własności odpowiednich parametrów obrazka. Na poniższym przykładzie przedstawimy niektóre z nich (w tym momencie wystarczające dla naszych zastosowań).

```
Plot[Cos[x] + Sin[2 x], {x, -1, 3 π},
 PlotStyle → {Black, Thick}, AxesStyle → {Dashed, Gray},
 AxesLabel → {"oś Ox", y}, Axes → {True, None},
 PlotRange → {{-3, 12}, {-3, 3}}, AspectRatio → Automatic]
```



a) **Formatowanie linii wykresu**

Za styl samego wykresu funkcji odpowiada **`PlotStyle→{cechy linii]}`**. Po strzałce, w postaci listy wpisujemy cechy linii takie jak: kolor (jego nazwa w j. angielskim) , **`Dashed`** (linia przerywana), **`Thick`** oraz **`Thickness[wartość]`** (odpowiednio linia pogrubiona oraz linia o ustalonej grubości), **`Opacity[wartość]`** (przezroczystość) .

b) **Formatowanie osi**

W analogiczny sposób jak powyżej formatujemy osie (**`AxesStyle`**). Możemy dodatkowo ukryć je wszystkie (**`Axes → None`**), bądź tylko jedną z nich (np. **`Axes → {True, None}`**). Każdej z osi możemy nadać etykietę (**`AxesLabel`**).

c) **Formatowanie obrazu**

Wyświetlany obszar (w tym przypadku część układu współrzędnych) ustalamy za pomocą **`PlotRange`**, podając listę zakresów wartości dla każdej osi (zauważmy, że nie jest to to samo co zakres zmienności funkcji) bądź wartość **`All`** w celu wyświetlenia wszystkich wartości na podanym przedziale. **`AspectRatio`** określa natomiast stosunek wysokości obrazka do jego szerokości.

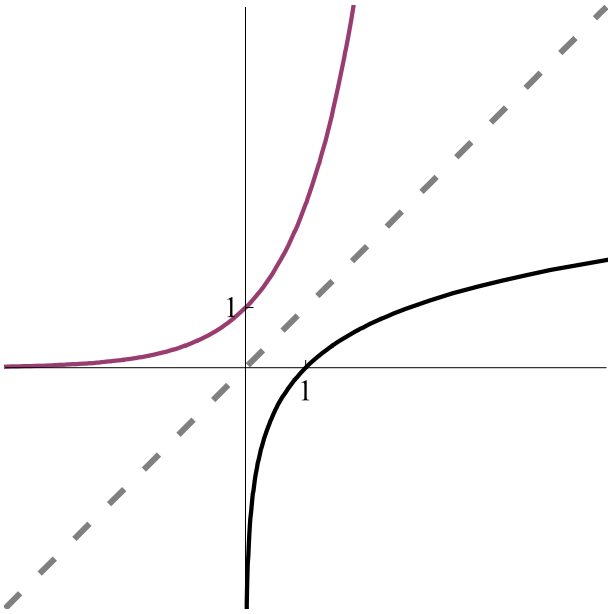
Wyświetlmy wszystkie opcje dotyczące wyrażenia **`Plot`**.

Plot // Options

$$\left\{\text{AlignmentPoint} \to \text{Center},\right.$$
$$\text{AspectRatio} \to \frac{1}{\text{GoldenRatio}}, \text{Axes} \to \text{True},$$
$$\text{AxesLabel} \to \text{None}, \text{AxesOrigin} \to \text{Automatic}, \text{AxesStyle} \to \{\},$$
$$\text{Background} \to \text{None}, \text{BaselinePosition} \to \text{Automatic},$$
$$\text{BaseStyle} \to \{\}, \text{ClippingStyle} \to \text{None},$$
$$\text{ColorFunction} \to \text{Automatic}, \text{ColorFunctionScaling} \to \text{True},$$
$$\text{ColorOutput} \to \text{Automatic}, \text{ContentSelectable} \to \text{Automatic},$$
$$\text{CoordinatesToolOptions} \to \text{Automatic},$$
$$\text{DisplayFunction} :\to \$\text{DisplayFunction}, \text{Epilog} \to \{\},$$
$$\text{Evaluated} \to \text{Automatic}, \text{EvaluationMonitor} \to \text{None},$$
$$\text{Exclusions} \to \text{Automatic}, \text{ExclusionsStyle} \to \text{None},$$
$$\text{Filling} \to \text{None}, \text{FillingStyle} \to \text{Automatic},$$
$$\text{FormatType} :\to \text{TraditionalForm}, \text{Frame} \to \text{False},$$
$$\text{FrameLabel} \to \text{None}, \text{FrameStyle} \to \{\}, \text{FrameTicks} \to \text{Automatic},$$
$$\text{FrameTicksStyle} \to \{\}, \text{GridLines} \to \text{None}, \text{GridLinesStyle} \to \{\},$$
$$\text{ImageMargins} \to 0., \text{ImagePadding} \to \text{All}, \text{ImageSize} \to \text{Automatic},$$
$$\text{ImageSizeRaw} \to \text{Automatic}, \text{LabelStyle} \to \{\},$$
$$\text{MaxRecursion} \to \text{Automatic}, \text{Mesh} \to \text{None}, \text{MeshFunctions} \to \{\#1\ \&\},$$
$$\text{MeshShading} \to \text{None}, \text{MeshStyle} \to \text{Automatic},$$
$$\text{Method} \to \text{Automatic}, \text{PerformanceGoal} :\to \$\text{PerformanceGoal},$$
$$\text{PlotLabel} \to \text{None}, \text{PlotPoints} \to \text{Automatic},$$
$$\text{PlotRange} \to \{\text{Full}, \text{Automatic}\}, \text{PlotRangeClipping} \to \text{True},$$
$$\text{PlotRangePadding} \to \text{Automatic}, \text{PlotRegion} \to \text{Automatic},$$
$$\text{PlotStyle} \to \text{Automatic}, \text{PreserveImageOptions} \to \text{Automatic},$$
$$\text{Prolog} \to \{\}, \text{RegionFunction} \to (\text{True}\ \&),$$
$$\text{RotateLabel} \to \text{True}, \text{Ticks} \to \text{Automatic},$$
$$\left.\text{TicksStyle} \to \{\}, \text{WorkingPrecision} \to \text{MachinePrecision}\right\}$$

Możemy oczywiście narysować więcej wykresów w jednym układzie współrzędnych. W wyrażeniu podajemy wówczas listę funkcji których wykresy chcemy wygenerować. Każdy z wykresów możemy sformatować w inny sposób. Style poszczególnych wykresów umieszczamy wówczas w postaci listy.

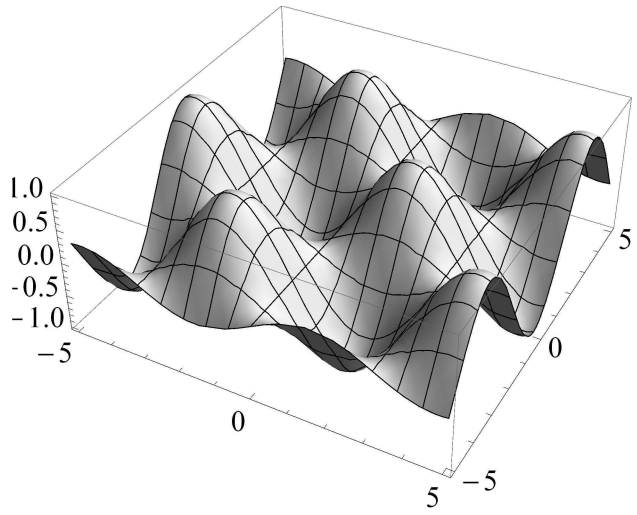
```
Plot[{Log[x], e^x, x}, {x, -10, 10},
 PlotRange → {{-4, 6}, {-4, 6}}, Ticks → {{1}, {1}},
 AspectRatio -> Automatic, PlotStyle → {{Black, Thick},
   Thick, {Dashing[Large], Gray, Thickness[.01]}}]
```



■ 3.5.2. Wykres funkcji $f : \mathbb{R}^2 \to \mathbb{R}$

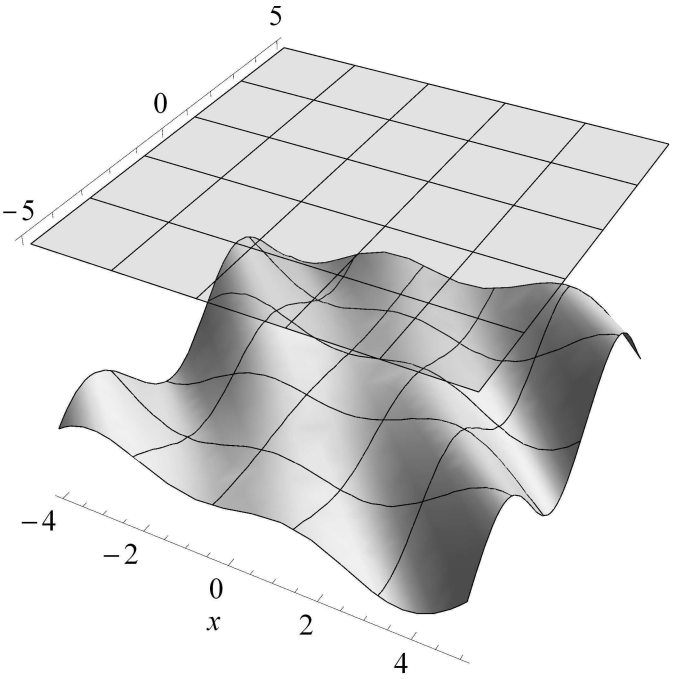
Mathematica® pozwala także na generowanie wykresów funkcji rzeczywistych, określonych na podzbiorach płaszczyzny (polecenie **Plot3D**). Zarówno tworzenie, jak i formatowanie wykresu przebiega w sposób analogiczny do przypadku funkcji jednej zmiennej. Składnia **Plot3D** jest podobna do **Plot** - podajemy funkcję oraz zakresy obydwu zmiennych.

```
Plot3D[Cos[y] Sin[x], {x, -5, 5}, {y, -5, 5}]
```



Zauważmy, że cała trójwymiarowa ilustracja jest ograniczona "pudełkiem". Trzy z jego krawędzi opatrzone są wartościami osi liczbowych. Wykres możemy obserwować z dowolnego kierunku, chwytając lewym klawiszem myszki i poruszając kursorem.
Niektóre elementy formatowania wykresu 3D przedstawimy na poniższym przykładzie.

```
Plot3D[{1 / 5 x Sin[y + x], 4},
 {x, -4, 5}, {y, -5, 5}, BoxRatios → {3, 3, 2},
 PlotStyle → {Automatic, {Orange, Opacity[.2]}},
 AxesLabel → {x, None, None},
 Axes → {True, True, False}, Boxed → False, Mesh → 4]
```

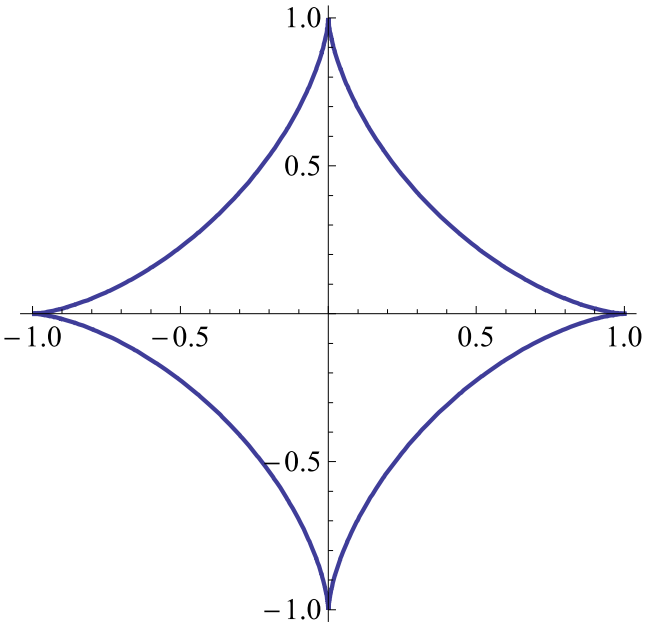


Część opcji pokrywa sie z tymi z wykresu dwuwymiarowego. Zauważmy, że już nie mamy "pudełka" (**Boxed→False**), wyświetlone są tylko dwie osie (**Axes → {True, True, False}**) oraz siatka narzucona na powierzchnię jest rzadsza (**Mesh → 4** - użycie wartości **False** spowoduje wyświetlenie rysunku bez siatki). **BoxRatios** spełnia podobną funkcję jak **AspectRatio** w przypadku wykresów 2D - określa trójstosunek krawędzi "pudełka" w jakim znajduje się trójwymiarowy rysunek (powyżej jest to 3:3:2).

■ 3.5.3. Wykresy krzywych i powierzchni zadanych w postaci parametrycznej

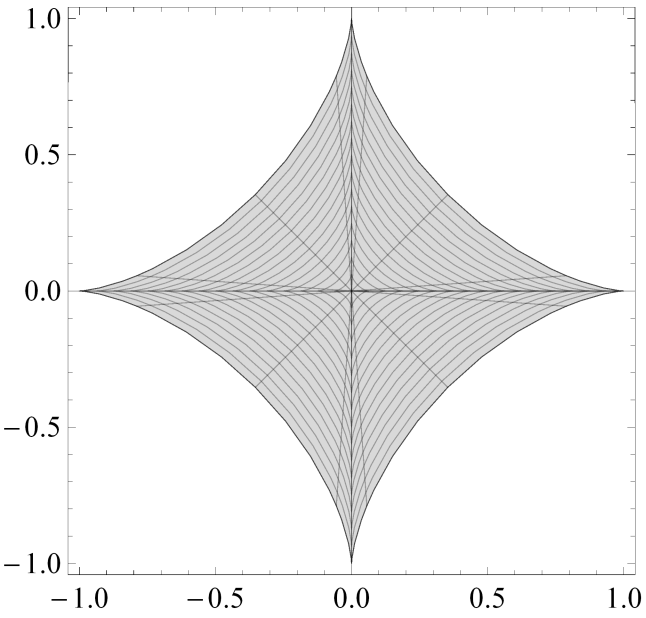
Krzywą na płaszczyźnie często zadajemy przy pomocy opisu parametrycznego, tzn. wykresem krzywej $\gamma : [a,\ b] \ni t \mapsto (x, y) \in \mathbb{R}^2$ jest zbiór $W = \{(x(t), y(t)) \in \mathbb{R}^2 : a \le t \le b\}$. Przy użyciu **ParametricPlot** możemy ilustrować wykresy funkcji zadanych parametrycznie.

```
ParametricPlot[{Cos[ϕ]^3, Sin[ϕ]^3}, {ϕ, 0, 2 π}]
```



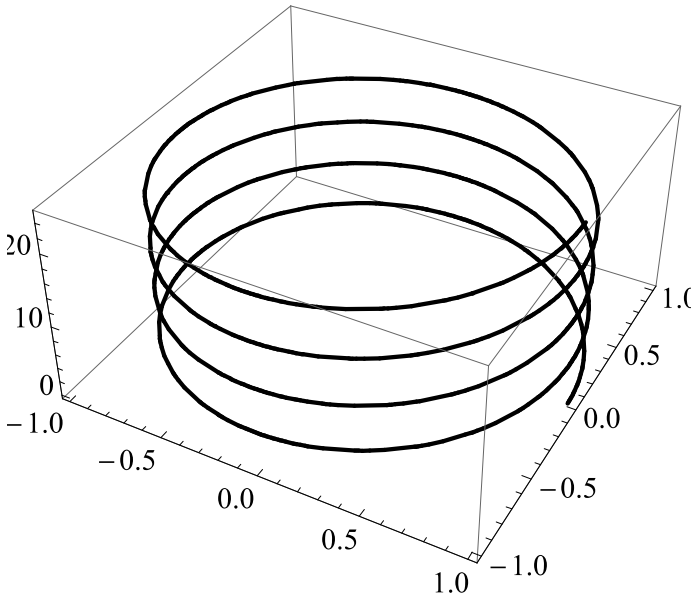
Składnia polecenia zawiera listę funkcji w postaci **{x[t],y[t]}** oraz zakres zmienności parametru. W prosty sposób możemy narysować wnętrze powyższej asteroidy, używając dodatkowego parametru.

```
ParametricPlot[r {Cos[ϕ]^3, Sin[ϕ]^3}, {ϕ, 0, 2 π}, {r, 0, 1}]
```



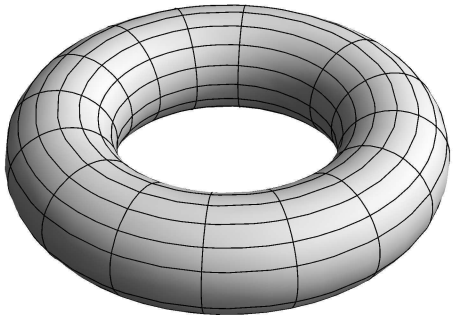
W analogiczny sposób do powyższego działa **ParametricPlot3D** w przestrzeni trójwymiarowej .

```
ParametricPlot3D[{Cos[t], Sin[t], t},
 {t, -0, 8 π}, BoxRatios → {2, 2, 1}]
```



Możemy także generować ilustracje powierzchni opisanych parametrycznie.

```
ParametricPlot3D[
 {4 + (3 + Cos[v]) Sin[u], 4 + (3 + Cos[v]) Cos[u], 4 + Sin[v]},
 {u, 0, 2 Pi}, {v, 0, 2 Pi}, Boxed → False, Axes → None]
```



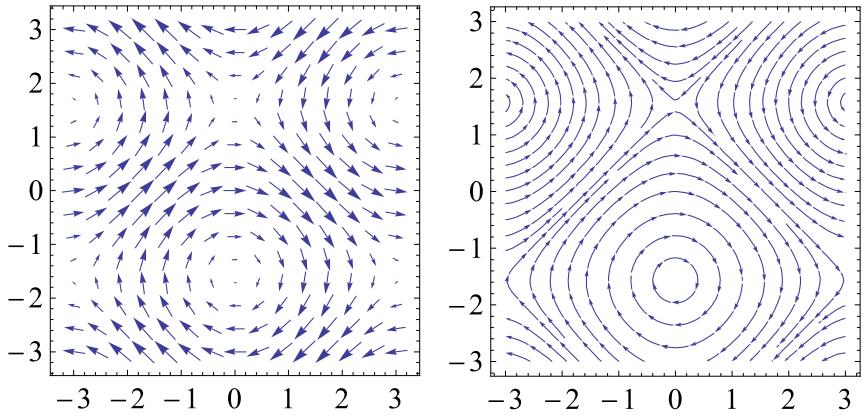
■ 3.5.4. Ilustracje pól wektorowych

Mathematica® posiada szereg operatorów generujących wykresy użyteczne w przypadku funkcji wektorowych. Zaczniemy od **VectorPlot** oraz **StreamPlot**. Pierwszy pozwala naszkicować pole wektorowe postaci $\boldsymbol{u}(x, y) = (u_1(x, y), u_2(x, y))$, natomiast drugi - linie prądu pola. Przedstawmy w ten sposób pole dla $u_1 = \cos(y)$, $u_2 = -\sin(x)$ w kwadracie $[-3, 3]\times[-3, 3]$,

```
u1 = Cos[y];
u2 = -Sin[x];
```

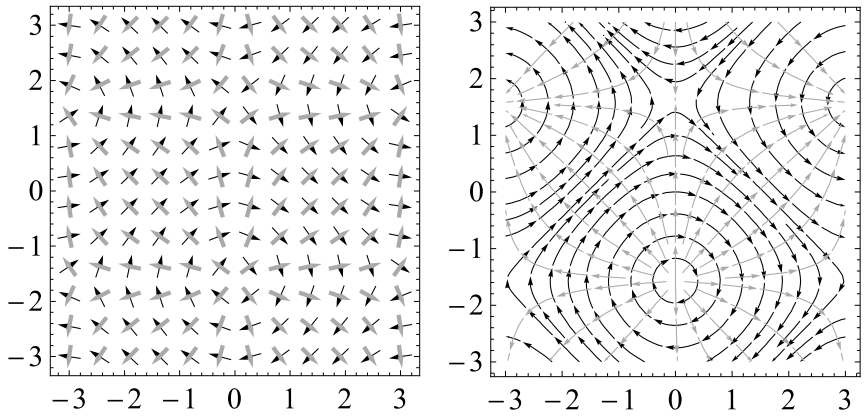
Wyświetlmy wykresy za pomocą operatora **GraphicsArray**.

```
{VectorPlot[{u1, u2}, {x, -3, 3}, {y, -3, 3}], StreamPlot[
    {u1, u2}, {x, -3, 3}, {y, -3, 3}]} // GraphicsArray
```



Zauważmy, że **VectorPlot** uwzględnia długość wektorów zadanych przez ***u***. Jednak możemy zmienić ustawienia operatora tak, aby wektory były jednej długości. Poniższe przykłady przedstawiają niektóre opcje formatowania wykresów w przypadku przedstawienia pola ***u*** oraz pola do niego ortogonalnego.

```
{VectorPlot[{{u1, u2}, {-u2, u1}}, {x, -3, 3}, {y, -3, 3},
    VectorPoints → 12, VectorScale → {.05, .9, None},
    VectorStyle → {Black, {Lighter[Gray], Thick}}],
  StreamPlot[{{u1, u2}, {-u2, u1}}, {x, -3, 3},
    {y, -3, 3}, StreamPoints → 30, StreamScale → .1,
    StreamStyle → {Black, {Lighter[Gray]}}]} // GraphicsArray
```



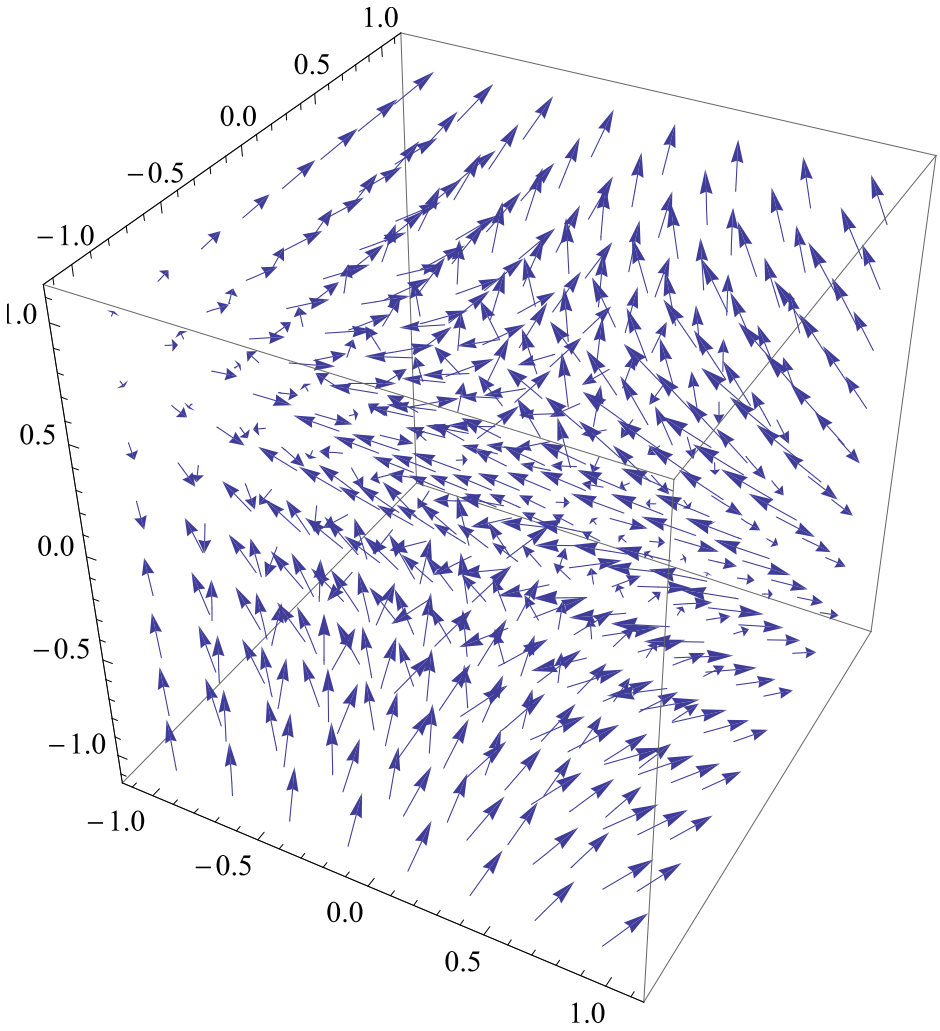
Trzy atrybuty opcji **VectorScale** oznaczają odpowiednio wielkość wektora, wielkość strzałki oraz skalowanie (**None** w tym przypadku gwarantuje brak skalowania). W podobny sposób działa **StreamScale**, natomiast **VectorPoints** oraz **StreamPoints** odpowiadają za gęstość wyświetlanego pola. Z kolei **VectorStyle** oraz **StreamStyle** zawierają formatowania typu: kolor, grubość, przezroczystość itd.

Istnieje także możliwość zobrazowania pola wektorowego

$$u(x, y, z) = (u_1(x, y, z), u_2(x, y, z), u_3(x, y, z)),$$

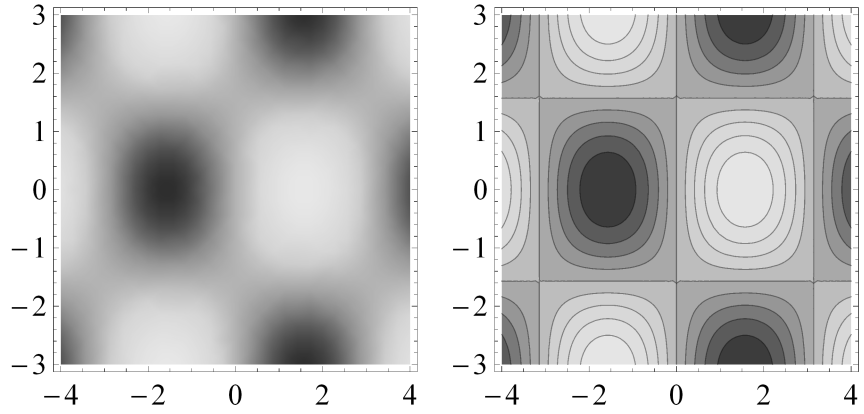
czyli trójwymiarowego.

```
VectorPlot3D[{-x t, y t, t^2}, {x, -1, 1}, {y, -1, 1},
 {t, -1, 1}, VectorPoints → 8, VectorScale → {.06, .5, None}]
```



Jeżeli mamy do czynienia z funkcjami skalarnymi (np. rozkład temperatury), to wówczas możemy skorzystać z **DensityPlot** bądź **ContourPlot**.

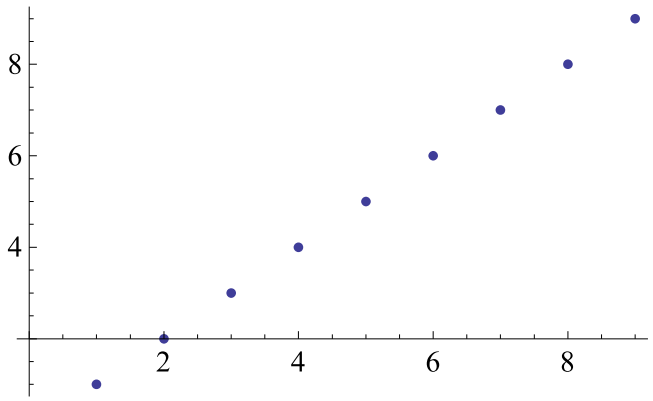
```
GraphicsRow[{
  DensityPlot[Sin[x] Cos[y], {x, -4, 4}, {y, -3, 3}],
  ContourPlot[Sin[x] Cos[y], {x, -4, 4}, {y, -3, 3}]}]
```



■ 3.5.5. Wykresy punktowe

W celu wykonania wykresu punktowego wykorzystamy **ListPlot**.

```
ListPlot[{1, 2, 3, 4, 5, 6, 7, 8, 9}]
```

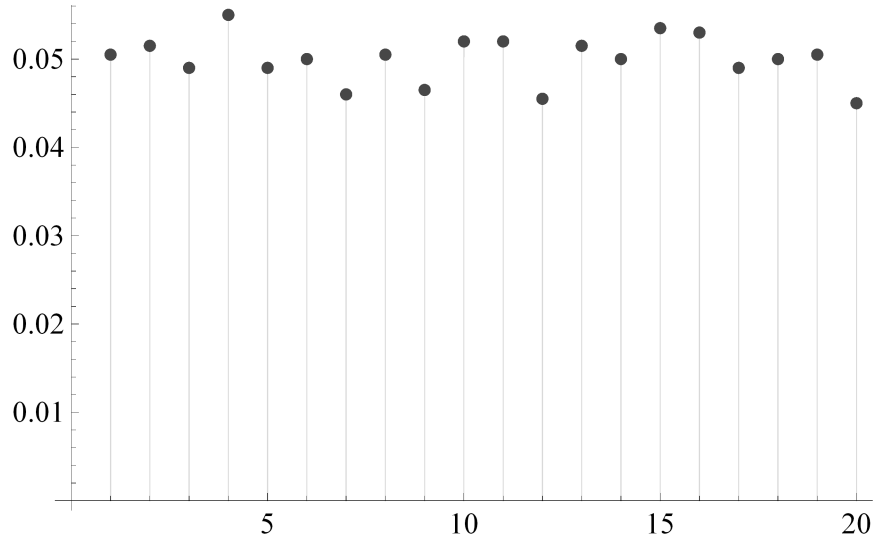


Wykonajmy symulację 2000 rzutów kostką 20-ścienną.

```
los1 = Table[RandomChoice[Range[20]], {2000}];
```

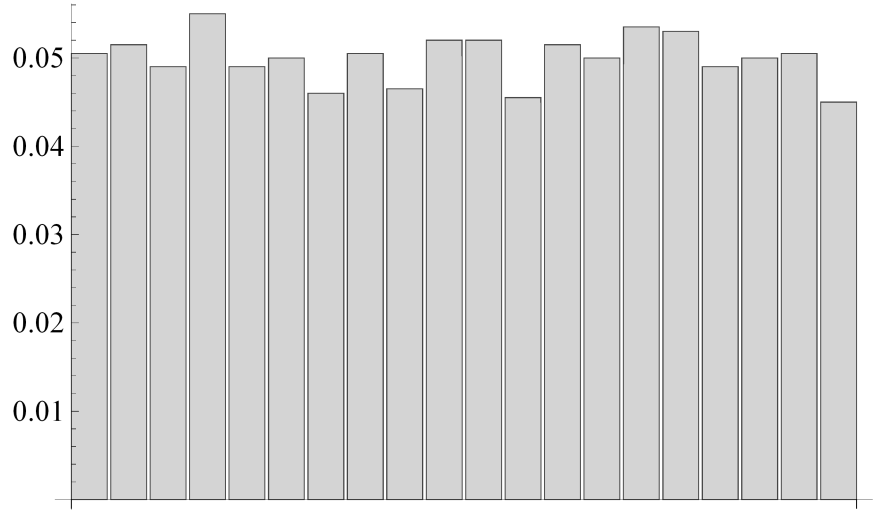
Utwórzmy wykres funkcji, która liczbie, będącej numerem ścianki, przyporządkowuje częstość jej wystąpienia podczas doświadczenia.

```
ListPlot[Count[los1, #] / 2000 & /@ Range[20],
 AxesOrigin → {0, 0}, Filling → Axis,
 PlotStyle → PointSize[.015]]
```



Możemy także użyć wykresu słupkowego.

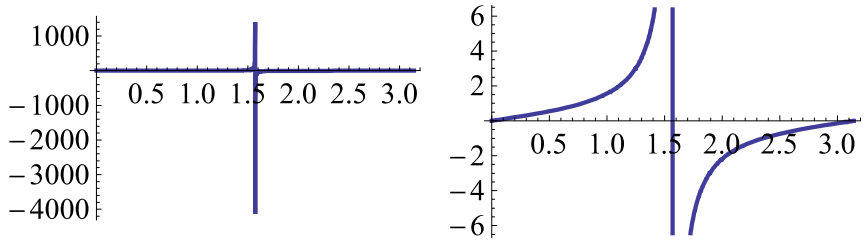
```
BarChart[Count[los1, #] / 2000 & /@ Range[20]]
```



■ 3.5.6. Jak powstaje wykres w *Mathematica*®

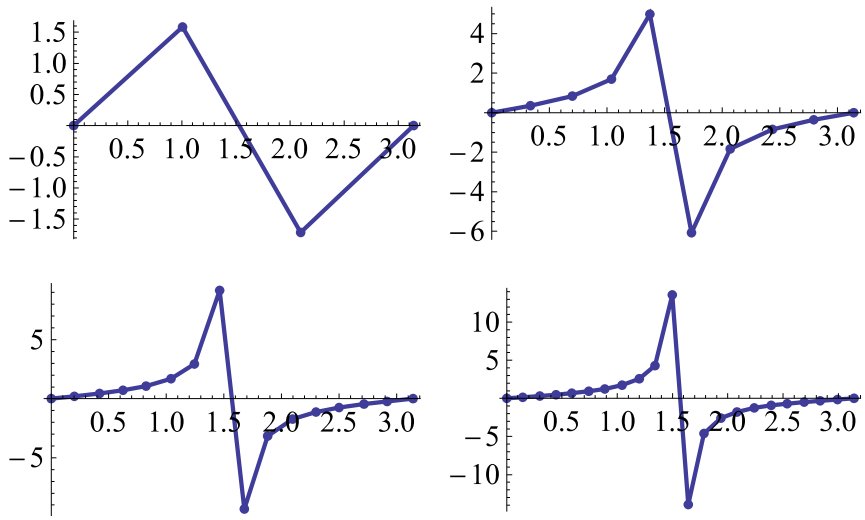
W powyższych rozdziałach zajmowaliśmy się wykresami funkcji. W tej części książki skupimy się na tym, w jaki sposób powstają owe wykresy. Narysujmy dwa wykresy funkcji tangens: jeden z opcją **PlotRange → All** (czyli zostanie wyświetlony cały wykres na określonym przedziale), a drugi z domyślną wartością **PlotRange**.

```
GraphicsRow[{
  Plot[Tan[x], {x, 0, π}, PlotRange → All],
  Plot[Tan[x], {x, 0, π}]}]
```

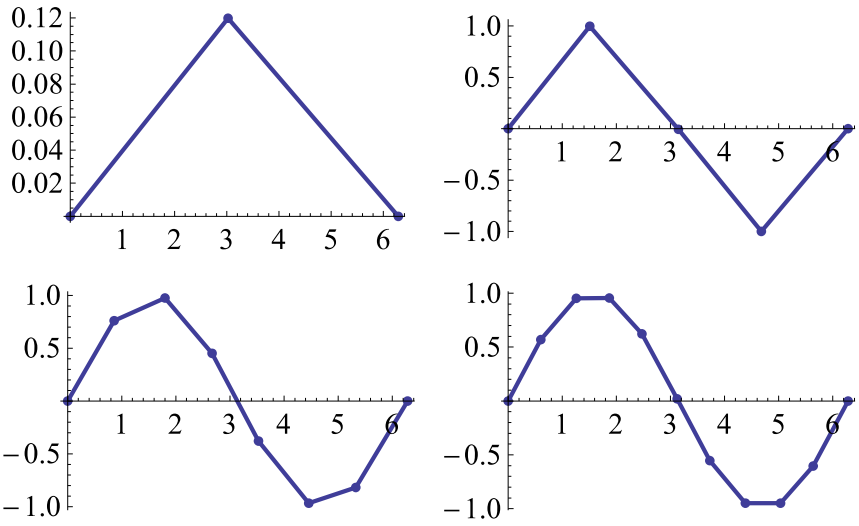


Wiemy doskonale, że tangens jest funkcją nieograniczoną. Dlaczego w takim razie *Mathematica*® narysowała ograniczony wykres? Natomiast, w przypadku drugiej ilustracji narzuca sie pytanie o przecięcie wykresu z osią w okolicy wartości $\frac{\pi}{2}$. Na te pytania jest prosta odpowiedź. Otóż *Mathematica*®, otrzymując polecenie narysowania wykresu, wybiera skończoną liczbę równo odległych od siebie punktów we wskazanym przez użytkownika przedziale, następnie znajduje wartości funkcji w tych punktach, po czym łączy je kolejno odcinkami. Owa liczba punktów jest na tyle duża, że łamana powstająca w ten sposób przypomina krzywą wykresu. Parametrami odpowiadającymi za "dokładność" rysowania są **`PlotPoints`** oraz **`MaxRecursion`**. Określają one liczbę punktów, które będą tworzyć podział oraz maksymalną liczbę podpodziałów dla każdego jednego odcinka podziału. Narysujmy kilka wykresów tangensa i sinusa przy niskich wartościach tych parametrów.

```
Plot[Tan[x], {x, 0, π }, Mesh → All, MaxRecursion → 0,
      PlotPoints → #, PlotRange → All] & /@ # & /@
  {{4, 10}, {16, 22}} // GraphicsArray
```



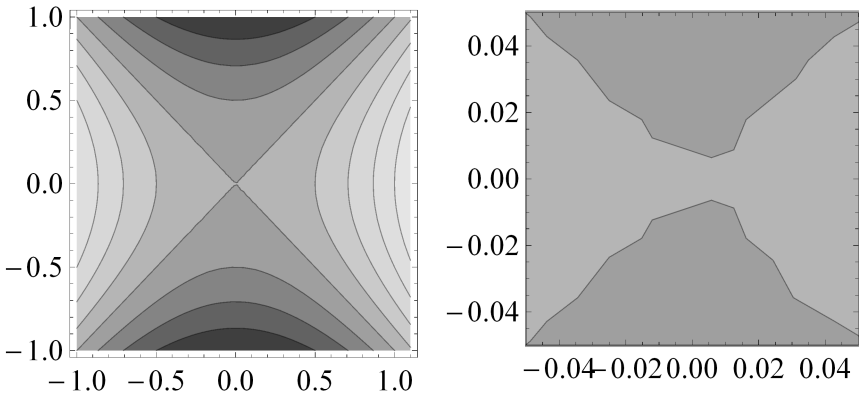

```
Plot[Sin[x], {x, 0, 2 π}, Mesh → All, MaxRecursion → 0,
      PlotPoints → #, PlotRange → All] & /@ # & /@
  {{3, 5}, {8, 11}} // GraphicsArray
```



Wykresy te powstały przy wyborze małej liczby punktów podziału wskazanego przedziału. Widzimy, że im więcej punktów wybierzemy, tym "gładszy" będzie wykres funkcji. W obliczu powyższych informacji, odpowiedzi na postawione pytania pozostawiamy czytelnikowi.

W analogiczny sposób są generowane pozostałe rodzaje wykresów. Poniższy przykład dotyczy **ContourPlot**. Pierwsza ilustracja przedstawia wygenerowany wykres, natomiast w drugim (pod powiększeniem za pomocą **PlotRange**) widzimy niedokładność w środku układu. Jest tam punkt osobliwy, z którym *Mathematica*® nie może sobie "poradzić".

```
{ContourPlot[x^2 - y^2, {x, -1, 1.1}, {y, -1, 1}],
  ContourPlot[x^2 - y^2, {x, -1, 1.1}, {y, -1, 1},
   PlotRange → {{-.05, .05}, {-.05, .05}}]} // GraphicsRow
```



Wniosek jest następujący: musimy mieć na względzie to, iż program komputerowy, ograniczony skończoną liczbą wykonywanych operacji, nie jest w stanie dorównać doskonałości pojęć matematyki. Użytkownik powinien być tego świadom podczas interpretowania wyników pracy z *Mathematica*® (patrz zasada głupiego komputera z Części I).

3.6. Równania i układy równań

■ 3.6.1. Równania

Równaniem w *Mathematica*® nazywamy wyrażenia połączone spójnikiem **==.**

```
2 x == x + x
```

```
True
```

Mathematica® pozwala rozwiązywać równania i układy równań. Przedstawimy kilka operatorów, dzięki którym jest to możliwe. Rozwiążmy równanie algebraiczne $x - 54\ x^2 + 5 = 0$ za pomoca operatora **Solve**.

```
Solve[x - 4 x^2 + 5 == 0, x]
```

$$\left\{\{x \to -1\}, \left\{x \to \frac{5}{4}\right\}\right\}$$

Zauważmy, że istotne jest określenie symbolu niewiadomej. Rozwiązania można przedstawić w postaci listy za pomocą **ReplaceAll**.

```
x /. %
```

$$\left\{-1, \frac{5}{4}\right\}$$

Sprawdźmy czy podane liczby spełniają równanie.

```
x - 4 x^2 + 5 /. x -> %
```

```
{0, 0}
```

Istnieją równania, dla których nie da się wprost wyznaczyć rozwiązania. Wówczas *Mathematica*® zwraca wynik w następującej formie.

```
Solve[2 - 4 x + x^5 == 0]
```

$$\{\{x \to \text{Root}[2 - 4\,\#1 + \#1^5\ \&,\ 1]\},\ \{x \to \text{Root}[2 - 4\,\#1 + \#1^5\ \&,\ 2]\},\ \{x \to \text{Root}[2 - 4\,\#1 + \#1^5\ \&,\ 3]\},\ \{x \to \text{Root}[2 - 4\,\#1 + \#1^5\ \&,\ 4]\},\ \{x \to \text{Root}[2 - 4\,\#1 + \#1^5\ \&,\ 5]\}\}$$

Root[f,k] reprezentuje k-ty pierwiastek równania wielomianowego $f(x) = 0$. Możemy wówczas wyznaczyć numerycznie przybliżone wartości.

```
% // N
```

```
{{x → -1.51851}, {x → 0.508499}, {x → 1.2436},
 {x → -0.116792 - 1.43845 i}, {x → -0.116792 + 1.43845 i}}
```

Za pomoca **Solve** można rozwiązać wszystkie równania algebraiczne o jednej niewiadomej stopnia niższego od pięciu oraz niektóre stopni wyższych. Jeżeli **Solve** nie poradzi sobie z rozwiązaniem jakiegoś równania, to za pomocą **NSolve** możemy próbować znaleźć numerycznie przybliżone rozwiązania.

```
NSolve[x^7 + x + 2 == 0, x]
```

```
{{x → -1.}, {x → -0.722647 - 0.774104 i},
 {x → -0.722647 + 0.774104 i},
 {x → 0.17701 - 1.12266 i}, {x → 0.17701 + 1.12266 i},
 {x → 1.04564 - 0.536005 i}, {x → 1.04564 + 0.536005 i}}
```

Funkcji **FindRoot** użyjemy w celu znalezienia pierwiastka funkcji w pobliżu określonego punktu. Pozwala ona także znaleźć numeryczne rozwiązanie równania.

```
FindRoot[Cos[x] == e^x, {x, -1}]
```

```
{x → -1.2927}
```

Spróbujmy rozwiązać równanie $\sin(x) = 0$.

```
Solve[Sin[x] == 0, x]
```

```
Solve::ifun :
  Inverse functions are being used by Solve, so some solutions may not be
      found; use Reduce for complete solution information. >>
```

```
{{x → 0}}
```

Mathematica® informuje nas, że mogły zostać podane nie wszystkie rozwiązania i sugeruje użycie operatora **Reduce**.

```
Reduce[Sin[x] == 0, x]
```

```
C[1] ∈ Integers && (x == 2 π C[1] || x == π + 2 π C[1])
```

Reduce pozwala określić dziedzinę zmiennych.

```
Reduce[Abs[x] == 1, x, Integers]
```

```
x == -1 || x == 1
```

```
Reduce[Abs[x] == 1, x, Complexes]
```

$$-1 \le \text{Re}[x] \le 1 \;\&\&\; \left(\text{Im}[x] == -\sqrt{1 - \text{Re}[x]^2} \;||\; \text{Im}[x] == \sqrt{1 - \text{Re}[x]^2}\right)$$

Innym przykładem, który wskazuje na użyteczność operatora **Reduce** jest rozwiązywanie równania $a\,x = b$. **Solve** podaje następujące rozwiązanie.

```
Solve[a x + b == 0, x]
```

$$\left\{\left\{x \to -\frac{b}{a}\right\}\right\}$$

Jednak wiemy, że może istnieć sytuacja, w której mamy nieskończenie wiele rozwiazań, mianowicie gdy $a = b = 0$.

```
Reduce[a x + b == 0, x]
```

$$(b == 0 \;\&\&\; a == 0) \;||\; \left(a \neq 0 \;\&\&\; x == -\frac{b}{a}\right)$$

■ 3.6.2. Układy równań

W analogiczny do powyższego sposób rozwiązujemy układy równań.

```
Solve[{-4 x^2 + y == -5, x + 3 y == -4}, {x, y}]
```

$$\left\{\left\{y \to -\frac{59}{36},\ x \to \frac{11}{12}\right\},\ \{y \to -1,\ x \to -1\}\right\}$$

Sprawdźmy otrzymane rozwiązania.

```
{-4 x^2 + y == -5, x + 3 y == -4} /. %
```

```
{{True, True}, {True, True}}
```

Skonstruujmy układ równań za pomocą macierzy.

```
A = {{3, 1, 4}, {2, -5, 2}, {-3, 11, 2}};
B = {7, 8, 1};
Solve[A.{x, y, z} == B, {x, y, z}]
```

$$\left\{\left\{x \to -\frac{67}{39},\ y \to -\frac{38}{39},\ z \to \frac{128}{39}\right\}\right\}$$

Możemy również szukać rozwiazań w sposób numeryczny

```
NSolve[{-4 x^3 + y == -z, Sqrt[x] + 3 y^3 == 2 z, Sqrt[z] == 2 + y}, {x, y, z}]
```

```
{{x → -0.856728 + 1.50398 i, y → 2.29486 - 0.0376467 i,
  z → 18.4444 - 0.323375 i}, {x → -0.856728 - 1.50398 i,
  y → 2.29486 + 0.0376467 i, z → 18.4444 + 0.323375 i},
 {x → 1.72475, y → 2.27209, z → 18.2507},
 {x → -0.00329447 + 0.801632 i, y → -0.865208 - 0.63019 i,
  z → 0.890613 - 1.43027 i}, {x → -0.00329447 - 0.801632 i,
  y → -0.865208 + 0.63019 i, z → 0.890613 + 1.43027 i},
 {x → -0.71231 - 0.455704 i, y → -0.753576 - 0.685995 i,
  z → 1.08298 - 1.71008 i}, {x → -0.71231 + 0.455704 i,
  y → -0.753576 + 0.685995 i, z → 1.08298 + 1.71008 i},
 {x → 0.720437 - 0.375062 i, y → -0.792189 - 0.622134 i,
  z → 1.07176 - 1.50284 i}, {x → 0.720437 + 0.375062 i,
  y → -0.792189 + 0.622134 i, z → 1.07176 + 1.50284 i}}
```

Poniżej znajduje się przykład rozwiązany za pomocą **Reduce**.

```
Reduce[{Sin[x] == y, Cos[x] Log[y] == 0}, {x, y}]
```

$$C[1] \in \text{Integers} \;\&\&\; \left(x == \frac{\pi}{2} + 2\pi C[1] \;||\; x == -\frac{\pi}{2} + 2\pi C[1] \;||\; x == \frac{\pi}{2} + 2\pi C[1]\right) \;\&\&\; y == \text{Sin}[x]$$

■ 3.7. Różniczkowanie i całkowanie

■ 3.7.1. Pochodna funkcji jednej zmiennej oraz pochodna cząstkowa

Zdefiniujmy funkcję

```
f[x_, y_] := x^n y + 2 x
```

Pochodną cząstkową względem x obliczymy za pomocą operatora **D**. Wystarczy podać funkcję oraz zmienną, względem której ma być wyznaczona pochodna.

```
D[f[x, y], x]
```

$$2 + n\, x^{-1+n}\, y$$

Podobnie obliczamy $\frac{\partial^2 f}{\partial y\, \partial x}$ oraz $\frac{\partial^2 f}{\partial x^2}$.

```
{D[f[x, y], y, x], D[f[x, y], {x, 2}]}
```

$$\left\{n\, x^{-1+n},\ (-1+n)\, n\, x^{-2+n}\, y\right\}$$

Identyczny efekt otrzymamy używając operatora w następujacej postaci

```
{∂y,x f[x, y], ∂x,x f[x, y]}
```

$$\left\{n\, x^{-1+n},\ (-1+n)\, n\, x^{-2+n}\, y\right\}$$

Zatem komendę obliczenia pochodnej $\frac{\partial^3 f}{\partial y\, \partial x^2}$ można wydać w trzech postaciach.

```
{∂y,x,x f[x, y], D[f[x, y], {y, 1}, {x, 2}], D[f[x, y], y, x, x]}
```

$$\left\{(-1+n)\, n\, x^{-2+n},\ (-1+n)\, n\, x^{-2+n},\ (-1+n)\, n\, x^{-2+n}\right\}$$

⚠ **Wskazówka** : Symbol ∂, używany w notacji pochodnej cząstkowej, wstawiamy za pomocą ESC**pd**ESC. Natomiast symbole zmiennych wpisujemy w indeksie dolnym.

Możemy także obliczyć gradient funkcji skalarnej (jest nią zdefiniowana wyżej funkcja **f**). Wówczas zmienne podajemy w podwójnych nawiasach.

```
D[f[x, y], {{x, y}}]
```

$$\left\{2 + n\, x^{-1+n}\, y,\ x^n\right\}$$

Zdefiniujmy funkcję wektorową $F: \mathbb{R}^3 \to \mathbb{R}^2$

```
F[x_, y_, z_] := {x^2 y z, Cos[x y]}
```

W sposób analogiczny do poprzedniego obliczamy jej macierz Jacobiego. Przedstawimy go w formie macierzowej.

```
D[F[x, y, z], {{x, y, z}}]
```

```
{{2 x y z, x^2 z, x^2 y}, {-y Sin[x y], -x Sin[x y], 0}}
```

```
% // MatrixForm
```

$$\begin{pmatrix} 2\,x\,y\,z & x^2\,z & x^2\,y \\ -y\,\mathrm{Sin}[x\,y] & -x\,\mathrm{Sin}[x\,y] & 0 \end{pmatrix}$$

Jeżeli mamy do czynienia z funkcją jednej zmiennej, to wówczas pochodne możemy obliczać w sposób przedstawiony w poniższym przykładzie (przy użyciu znaku apostrofu).

```
g[x_] := a x^3 + Cos[x]
```

```
{g'[x], g''[x]}
```

```
{3 a x^2 - Sin[x], 6 a x - Cos[x]}
```

```
g''[x]
```

```
6 a x - Cos[x]
```

Oczywiście jest to równoważne obliczeniu za pomocą operatora **D**.

```
D[g[x], {x, 2}]
```

```
6 a x - Cos[x]
```

Obliczmy wartość pochodnej funkcji g w punkcie $x = 3$.

```
g'[3]
```

```
27 a - Sin[3]
```

Załóżmy, że stała a w opisie funkcji g jest funkcją zależną od x (czyli $a = a(x)$). Wówczas istnieje możliwość poinformowania *Mathematica*® o tym fakcie za pomocą polecenia **NonConstants→{a}**.

```
D[g[x], x, NonConstants → {a}]
```

```
3 a x^2 + x^3 D[a, x, NonConstants → {a}] - Sin[x]
```

Wyrażenie **D[a,x,NonConstants→{a}]** symbolizuje pochodną funkcji $a = a(x)$.

```
{Clear[f], Clear[g]};
```

Sprawdźmy, za pomocą **InputForm**, jaką składnię ma operator różniczkowania w pełnej formie.

```
D[fun[x, y, z], {x, 3}, {z, 2}] // InputForm
Derivative[3, 0, 2][fun][x, y, z]
```

Pierwszy nawias określa kolejno numery pochodnych cząstkowych według odpowiadających im zmiennych.

■ 3.7.2. Pochodna totalna

Istnieje także operator **Dt**, który oblicza tzw. pochodną totalną funkcji, to jest oblicza pochodną przy założeniu, że każdy parametr w wyrażeniu jest funkcją zmiennej względem której różniczkujemy.
Zatem pochodna totalna funkcji $f : \mathbb{R}^k \to \mathbb{R}^m$ jest jej różniczką zupełną. Obliczmy różniczkę zupełną odwzorowania $f : \mathbb{R}^3 \to \mathbb{R}^2$ o przepisie $f(x, y, z) \to \left(x\,y\,z, x^2 + y\,z\right)$.

```
Dt[{x y z, x^2 + yz}]
{y z Dt[x] + x z Dt[y] + x y Dt[z], 2 x Dt[x] + Dt[yz]}
```

Załóżmy, że z jest stałą, to znaczy $f(x, y) \to \left(x^2\,y\,z, \cos(x\,y)\right)$. Musimy jednak poinformować o tym program (polecenie **Constants**).

```
Dt[{x y z, x^2 + yz}, Constants → {z}]
{y z Dt[x, Constants → {z}] + x z Dt[y, Constants → {z}],
 2 x Dt[x, Constants → {z}] + Dt[yz, Constants → {z}]}
```

■ 3.7.3. Symboliczne obliczanie pochodnych

Załóżmy, że mamy dane funkcje $u(x)$ i $v(x)$. Obliczmy pochodną iloczynu $u\,v$.

```
D[u[x] v[x], x]
v[x] u'[x] + u[x] v'[x]
```

Następnie pochodną złożenia $u \circ v$.

```
D[u[v[x]], x]
u'[v[x]] v'[x]
```

Niech teraz $u = u(x, y)$ i $v = v(x, y)$. Obliczmy $\frac{\partial^2 u\left(v(x,y),y^2\right)}{\partial y\,\partial x}$

```
D[u[v[x, y], y^2], x, y]
u^(1,0)[v[x, y], y^2] v^(1,1)[x, y] + v^(1,0)[x, y]
   (2 y u^(1,1)[v[x, y], y^2] + v^(0,1)[x, y] u^(2,0)[v[x, y], y^2])
```

⚠ **Uwaga** : Zapis postaci $u^{(k_1, k_2, k_3, \ldots, k_n)}[x_1, x_2, x_3, \ldots, x_n]$ oznacza pochodną rzędu $\sum_n k_n$ odpowiednio k_j razy po zmiennej x_j.

Niech $G = G(u, v)$. Obliczmy $\frac{\partial G}{\partial x}$

```
D[G[u[x, y], v[x, y]], x]
```

```
G^(1,0)[u[x, y], v[x, y]] u^(1,0)[x, y] +
 G^(0,1)[u[x, y], v[x, y]] v^(1,0)[x, y]
```

Zatem w *Mathematica*® bez problemu przeprowadzimy proste obliczenia symboliczne na funkcjach bez konkretnego przepisu.

■ 3.7.4. Całki

Konstrukcja **Integrate** pozwala na całkowanie funkcji. Obliczmy całkę nieoznaczoną funkcji $f(x) = e^{-2x}$.

```
Integrate[e^(-2 x), x]
```

$$-\frac{1}{2} e^{-2x}$$

Integrate możemy używać w postaci zbliżonej do formalnego zapisu $\int f(x)\,dx$.

```
∫Cos[x] Sin[x] dx
```

$$-\frac{1}{2}\text{Cos}[x]^2$$

⚠ **Wskazówka** : Konstrukcję ∫□d□ wstawiamy za pomocą ESC**intt**ESC. Natomiast ESC**int**ESC oraz ESC**dd**ESC dają odpowiednio znak całki ∫ oraz symbol d.

W analogiczny sposób liczymy całkę oznaczoną na określonym przedziale.

```
Integrate[x^2 Sin[x] , {x, 0, 2 π}]
```

$$-4\,\pi^2$$

Poniższy przykład przedstawia sposób obliczenia całki iterowanej $\int_0^1 \int_{-\pi}^{\pi} f(x, y)\,dx dy$, czyli całki $\int_D f(x, y)\,dx\,dy$ gdzie $D = (-\pi, \pi) \times (0, 1)$.

```
Integrate[Cos[x]^2 + y, {x, -π, π}, {y, 0, 1}]
```

$$2\,\pi$$

Zapiszmy powyższe całki w równoważnej formie.

```
{∫_0^(2 π) x^2 Sin[x] dx, ∫_0^1 ∫_(-π)^π (Cos[x]^2 + y) dx dy}
```

$$\{-4\,\pi^2,\ 2\,\pi\}$$

⚠ **Wskazówka** : Konstrukcję $\int_{\square}^{\square} \square \, d\square$ wstawiamy za pomocą ESC **dintt** ESC. Natomiast $\int_{\square}^{\square}$ otrzymamy wstawiając znak całki $\int$, a następnie używając skrótów CTRL+_ oraz CTRL+%, odpowiednio dla dolnej i górnej granicy całkowania.

Obliczmy całkę wielokrotną $\int_0^1 \int_0^x x \sin(x)\, dy\, dx$.

```
Integrate[y Sin[y]^2, {y, 0, x}, {x, 0, 1}]
```

$$\frac{1}{24}(5 + 3\cos[2] - 3\sin[2])$$

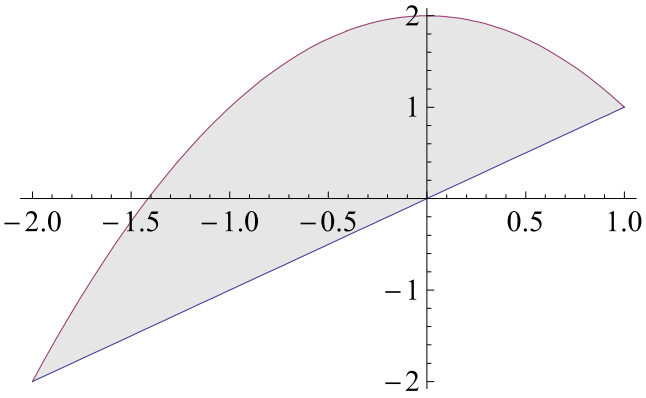
Podajmy wartość numeryczną wyniku.

```
% // N
```

```
0.0426528
```

Obliczmy całkę z funkcji $f(x, y) = x\,y$ na obszarze ograniczonym wykresami funkcji $y = x$ oraz $y = 2 - x^2$.

```
Plot[{x, 2 - x^2}, {x, -2, 1},
 Filling → {1 → {{2}, Lighter[Gray, .8]}}]
```




```
Solve[x == 2 - x^2, x]
```

```
{{x → -2}, {x → 1}}
```

```
{a = %[[1, 1, 2]], b = %[[2, 1, 2]]}
```

```
{-2, 1}
```

```
Integrate[x y, {y, x, -x^2+2}, {x, a, b}]
```

$$\frac{9}{8}$$

■ 3.7.5. Całkowanie numeryczne

Całki, których *Mathematica*® nie jest w stanie policzyć zostawiane są w postaci symbolicznej.

```
Integrate[x^x, {x, 0, 1}]
```

$$\int_0^1 x^x \, dx$$

Możemy wówczas użyć operatora **N**, w celu uzyskania wartości tej całki metodą numeryczną.

```
% // N
```

```
0.783431
```

Strukturą, która oblicza numeryczne wartości całek oznaczonych jest **NIntegrate**. Składnia polecenia jest wówczas identyczna z przypadkiem **Integrate**.

```
NIntegrate[x^x, {x, 0, 1}]
```

```
0.783431
```

Integrate daje przybliżony, numeryczny wynik. Możemy sterować dokładnością pracy tego operatora. Sprawdźmy jakie opcje są dla niego dostępne.

```
Options[NIntegrate]
```

```
{AccuracyGoal → ∞,
 Compiled → Automatic, EvaluationMonitor → None,
 Exclusions → None, MaxPoints → Automatic,
 MaxRecursion → Automatic, Method → Automatic,
 MinRecursion → 0, PrecisionGoal → Automatic,
 WorkingPrecision → MachinePrecision}
```

Zachęcamy czytelnika, aby za pomocą plików pomocy *Help* przeanalizował większość opcji operatora **Integrate**.

■ 3.7.6. Konstrukcja nowego operatora całkowania

Przedstawimy konstrukcję nowego operatora całkowania **Int** opartego na **Integrate**. Pierwsza część naszej definicji ma następującą postać.

```
ClearAll[Int];
Int[f_Plus, z_] := Int[#, z] & /@ f;
Int[a_, {z_Symbol, z0_, z1_}] := a z1 - a z0 /; FreeQ[a, z];
Int[P_, {z_Symbol, z0_, z1_}] :=
  Integrate[P, {z, z0, z1}] /; PolynomialQ[P, z];
Int[a_. ℯ^(ω_. z_), {z_Symbol, z0_, z1_}] :=
  a/ω ℯ^(ω z1) - a/ω ℯ^(ω z0) /; FreeQ[a, z] ∧ FreeQ[ω, z];
Int[ℯ^(ω_. z_) P_, {z_Symbol, z0_, z1_}] := (1/ω P ℯ^(ω z) /. z → z1) -
    (1/ω P ℯ^(ω z) /. z → z0) - 1/ω Int[D[P, z] ℯ^(ω z), {z, z0, z1}] /;
   FreeQ[ω, z] ∧ PolynomialQ[P, z];
```

Poniżej znajdują się przykłady, odróżniające **Int** od **Integrate**.

```
{Integrate[ℯ^(w z), {z, z0, z1}], Int[ℯ^(w z), {z, z0, z1}]}
```

$$\left\{\frac{-e^{w\,z0}+e^{w\,z1}}{w},\ -\frac{e^{w\,z0}}{w}+\frac{e^{w\,z1}}{w}\right\}$$

```
{Integrate[ℯ^(w z) z + z^2, {z, z0, z1}], Int[ℯ^(w z) z + z^2, {z, z0, z1}]}
```

$$\left\{\frac{1}{3}\left(-z0^3+z1^3+\frac{3\left(e^{w\,z0}\,(1-w\,z0)+e^{w\,z1}\,(-1+w\,z1)\right)}{w^2}\right),\right.$$
$$\left.-\frac{-\frac{e^{w\,z0}}{w}+\frac{e^{w\,z1}}{w}}{w}-\frac{e^{w\,z0}\,z0}{w}-\frac{z0^3}{3}+\frac{e^{w\,z1}\,z1}{w}+\frac{z1^3}{3}\right\}$$

Natomiast poniższa część przykładu pokazuje, jak przy użyciu operatora **MakeBoxes** możemy zdefiniować wyjściową formę symbolu **Int**.

```
MakeBoxes[Int[expr_, x_Symbol], form_] :=
  RowBox[{StyleBox["∫", FontWeight → "Bold",
     FontColor → RGBColor[0, 0, .8]],
    MakeBoxes[expr, form], StyleBox["𝕕",
     FontWeight → "Bold", FontColor → RGBColor[0, 0, .8]],
    MakeBoxes[x, form]}];
MakeBoxes[Int[expr_, {x_Symbol, xFrom_, xTo_}], form_] :=
  RowBox[
   {UnderoverscriptBox[StyleBox["∫", FontWeight → "Bold",
      FontColor → RGBColor[0, 0, .8]], MakeBoxes[xFrom,
      form], MakeBoxes[xTo, form]], MakeBoxes[expr, form],
    StyleBox["𝕕", FontWeight → "Bold",
     FontColor → RGBColor[0, 0, .8]], MakeBoxes[x, form]}];
```

Zauważmy skutek na przykładzie całki z funkcji zadanej symbolicznie.

```
{Integrate[f[z], {z, z0, z1}], Int[f[z], {z, z0, z1}]}
```

$$\left\{\int_{z0}^{z1} f[z]\, dz,\ \int_{z0}^{z1} f[z]\, dz\right\}$$

```
InputForm /@ %
```

```
{Integrate[f[z], {z, z0, z1}], Int[f[z], {z, z0, z1}]}
```

Nie będziemy omawiać budowy i znaczenia poszczególnych definicji w przedstawionym przykładzie. Niech będzie to ćwiczeniem dla zaawansowanego czytelnika (wskazana pomoc *Help*).

■ 3.8. Równania różniczkowe

■ 3.8.1. Równania różniczkowe zwyczajne

W rozwiązywaniu równań różniczkowych pomoże nam operator **DSolve**. Składnia może występować w dwóch poniższych postaciach.

```
rr1 = DSolve[y'[x] + x^2 == 3, y[x], x]
```

$$\left\{\left\{y[x] \to 3x - \frac{x^3}{3} + C[1]\right\}\right\}$$

```
rr2 = DSolve[y'[x] + x^2 == 3, y, x]
```

$$\left\{\left\{y \to \text{Function}\left[\{x\}, 3x - \frac{x^3}{3} + C[1]\right]\right\}\right\}$$

Druga skutkuje zwróceniem rozwiązania w czystej postaci. Zauważmy, że w rozwiązaniach pojawił sie symbol `C[1]` . Oznacza on stałą całkowania (parametryzuje rodzinę rozwiązań równania różniczkowego). Rozwiązania zostały podane w postaci listy zawierającej regułę. Zdefiniujmy na ich podstawie funkcje (patrz rozdział *Manipulacja wyrażeniami*)

```
y1[x_] = y[x] /. rr1[[1]]
```

$$3x - \frac{x^3}{3} + C[1]$$

```
y2 = rr2[[1, 1, 2]]
```

$$\text{Function}\left[\{x\}, 3x - \frac{x^3}{3} + C[1]\right]$$

```
{y1[x], y2[x]}
```

$$\left\{3x - \frac{x^3}{3} + C[1], 3x - \frac{x^3}{3} + C[1]\right\}$$

Sprawdźmy, że dana funkcja jest rozwiązaniem naszego równania.

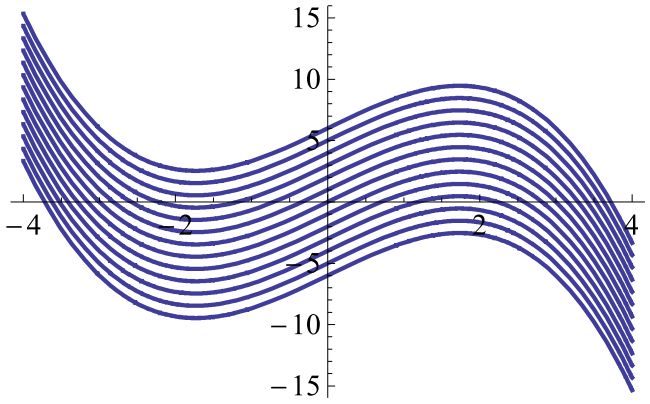
```
y'[x] + x^2 == 3 /. y → # & /@ {y1, y2}
{True, True}
```

Skoro już zdefiniowaliśmy rozwiązania, możemy przystąpić do wizualizacji krzywych całkowych równania. Użyjemy operatora **ReplaceAll** aby utworzyć podzbiór rodziny rozwiązań.

```
y2[x] /. C[1] → Table[c, {c, -6, 6, 1}]
```

$$\left\{-6 + 3x - \frac{x^3}{3}, -5 + 3x - \frac{x^3}{3}, -4 + 3x - \frac{x^3}{3}, -3 + 3x - \frac{x^3}{3},\right.$$
$$-2 + 3x - \frac{x^3}{3}, -1 + 3x - \frac{x^3}{3}, 3x - \frac{x^3}{3}, 1 + 3x - \frac{x^3}{3}, 2 + 3x - \frac{x^3}{3},$$
$$\left.3 + 3x - \frac{x^3}{3}, 4 + 3x - \frac{x^3}{3}, 5 + 3x - \frac{x^3}{3}, 6 + 3x - \frac{x^3}{3}\right\}$$

```
Plot[y2[x] /. C[1] → Table[c, {c, -6, 6, 1}], {x, -4, 4}]
```



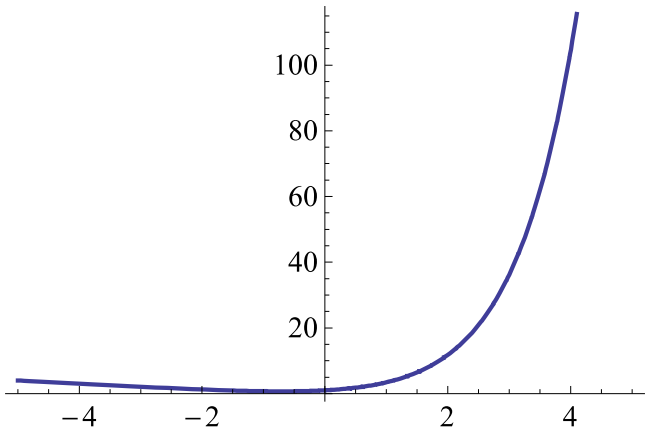
Znajdźmy rozwiązanie równania różniczkowego, spełniające warunek początkowy.

```
DSolve[{y'[x] == y[x] + a x, y[0] == 1}, y[x], x]
```

```
{{y[x] → -1 + 2 e^x - x}}
```

Narysujmy wykres rozwiązania, przy ustalonym parametrze **a**.

```
Plot[%[[1, 1, 2]] /. a → 25, {x, -5, 5}]
```



Rozwiążmy układ równań różniczkowych.

```
DSolve[{x'[t] == y[t], y'[t] == -x[t]}, {x[t], y[t]}, t]
```

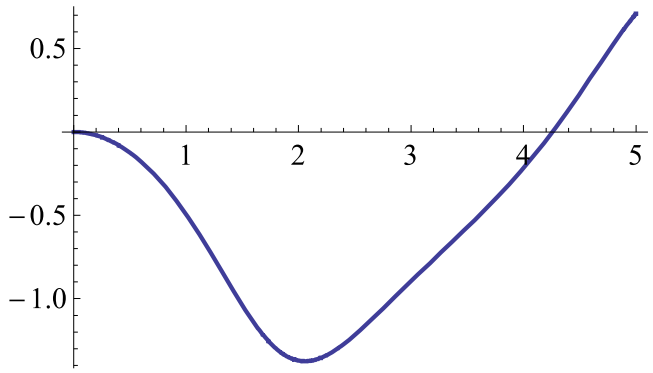
```
{{x[t] → C[1] Cos[t] + C[2] Sin[t],
  y[t] → C[2] Cos[t] - C[1] Sin[t]}}
```

Jeżeli *Mathematica*® nie znajdzie rozwiązania w sposób symboliczny, to wówczas możemy użyć operatora numerycznego obliczania rozwiazań **Ndsolve**.

```
NDSolve[{y'[x] + Cos[x] y[x]^2 == -Sin[x], y[0] == 0},
 y[x], {x, 0, 5}]
```

```
{{y[x] → InterpolatingFunction[{{0., 5.}}, <>][x]}}
```

```
Plot[%[[1, 1, 2]], {x, 0, 5}]
```



■ 3.8.2. Równania różniczkowe cząstkowe

Omówimy teraz rozwiązywanie równań różniczkowych cząstkowych. Rozwiążmy równanie

$$\frac{\partial y}{\partial x_1} - 2\,\frac{\partial y}{\partial x_2} = {x_2}^2$$

```
DSolve[D[y[x1, x2], x1] - 2 D[y[x1, x2], x2] == x2^2,
 y[x1, x2], {x1, x2}]
```

$$\left\{\left\{\text{y[x1, x2]} \to \frac{1}{6}\left(-\text{x2}^3 + 6\,\text{C[1][2 x1 + x2]}\right)\right\}\right\}$$

W tym przypadku symbol `C[1]` oznacza dowolną funkcję. Równania możemy wprowadzać także za pomocą symbolu ∂.

```
DSolve[∂x1 y[x1, x2] - ∂x2 y[x1, x2] == x1 y[x1, x2], y, {x1, x2}]
```

$$\left\{\left\{\text{y} \to \text{Function}\left[\{\text{x1, x2}\},\ e^{\frac{\text{x1}^2}{2}}\,\text{C[1][x1 + x2]}\right]\right\}\right\}$$

■ 3.9. Analiza zespolona

■ 3.9.1. Podstawowe operatory

Podstawowe operatory dla liczb zespolonych przedstawimy w postaci listy.

```
z1 = 2 e^(3 i)
```

$$2\,e^{3\,i}$$

```
{Re[z1], Im[z1], Abs[z1], Arg[z1], Conjugate[z1]}
```

$$\left\{2\,\text{Cos[3]},\ 2\,\text{Sin[3]},\ 2,\ 3,\ 2\,e^{-3\,i}\right\}$$

Oznaczają one odpowiednio: część rzeczywistą, część urojoną, moduł, argument główny oraz sprzężenie liczby zespolonej. Zauważmy, że *Mathematica*® jako argument główny liczby zespolonej traktuje argument z przedziału $(-\pi, \pi]$.

```
{Arg[-1], Arg[-i]}
```

$$\left\{\pi, -\frac{\pi}{2}\right\}$$

■ 3.9.2. Przekształcanie wyrażeń zespolonych

W celu wykonywania przekształceń wyrażeń zespolonych możemy korzystać z poznanych już metod. Ze wzgledu na postać trygonometryczną liczby zespolonej użytecznymi będą operatory przekształcające wyrażenia trygonometryczne

```
e^(i φ) == Cos[φ] + i Sin[φ] // Simplify
```

```
True
```

Niektóre z poznanych już operatorów mogą nie spełniać naszych oczekiwań. Rozpatrzmy następujący przykład. Użyjemy operatora **Expand** w celu rozwinięcia wyrażenia

$$\mathrm{Re}(a + i\,b)\,\mathrm{Im}(c + i\,d).$$

```
Expand[Re[a + i b] Im[c + i d]]
```

```
-Im[b] Im[c] + Im[c] Re[a] - Im[b] Re[d] + Re[a] Re[d]
```

Wynik jest jak najbardziej poprawny, ponieważ nie są znane informacje na temat symboli **a**, **b**, **c**, **d**. Za pomocą **Refine** wykonajmy to samo polecenie przy założeniu **a**,**b**,**c**,**d**∈ℝ

```
Refine[Expand[Re[a + i b] Im[c + i d]], {a, b, c, d} ∈ Reals]
```

```
a d
```

Zatem na czym polega główny problem w przypadku operowania wyrażeniami zespolonymi? Otóż *Mathematica*® musi wiedzieć z jakimi liczbami ma do czynienia. Pracę z liczbami zespolonymi w znaczny sposób ułatwi nam operator **ComplexExpand**. W domyślnych ustawieniach traktuje wszystkie zmienne jako elementy rzeczywiste. Możemy dodatkowo określić które z nich są zespolone.

```
{ComplexExpand[(a + i b) (c + i d)],
 ComplexExpand[x y], ComplexExpand[x y, {x, y}]}
```

```
{a c - b d + i (b c + a d), x y,
 -Im[x] Im[y] + Re[x] Re[y] + i (Im[y] Re[x] + Im[x] Re[y])}
```

Widzimy, że powyższe wyrażenia zostały rozwinięte do postaci algebraicznej ($z = \mathrm{Re}(z) + i\,\mathrm{Im}(z)$). Używając opcji **TargetFunction**, określamy funkcję, za pomocą której ma być podane rozwinięcie. Możliwymi opcjami są: **Re**, **Im**, **Abs**, **Conjugate** (sprzężenie), **Sign** (tutaj $\mathrm{sign}(z) = \frac{z}{\mathrm{Abs}(z)}$).

```
ComplexExpand[Re[z^2], {z}]
ComplexExpand[Re[z^2], {z}, TargetFunctions -> {Abs, Arg}]
ComplexExpand[Re[z^2], {z}, TargetFunctions -> Conjugate]
```

$$-\text{Im}[z]^2 + \text{Re}[z]^2$$

$$\text{Abs}[z]^2 \text{Cos}[\text{Arg}[z]]^2 - \text{Abs}[z]^2 \text{Sin}[\text{Arg}[z]]^2$$

$$\frac{z^2}{2} + \frac{\text{Conjugate}[z]^2}{2}$$

Jeżeli zamierzamy prowadzić dłuższe obliczenia, to uciążliwe może być ciągłe ustalanie założeń. Możemy wówczas spróbować zbudować funkcję, która symbole określonej postaci będzie traktować jako liczby zespolone. Rozpatrzmy następujący przykład. Załóżmy, że chcemy aby w trakcie naszej pracy używać odpowiedników funkcji **Re** oraz **Im**, które traktowałyby każdą zmienną postaci z_k, gdzie k jest dowolnym symbolem jako zespoloną. Musimy najpierw "nauczyć" *Mathematica*® rozpoznawać w danym wyrażeniu elementy tej postaci. Posłużmy się operatorem **Cases**.

```
wyb[wyr_] := Cases[{wyr}, z_, ∞]
```

Powyższa funkcja ma wybrać z wyrażenia każdy obiekt typu "z posiadający indeks dolny" (powyżej znak podkreślenia znajduje się w indeksie symbolu **z**). Wartość ∞ określa w tym przypadku poziom, do którego ma działać **Cases**, a więc do ostatniego. Sprawdźmy jak działa nasz operator.

```
wyb[Sqrt[f[z_i] Sin[z_6^2] + (a + z_indeks) zk]]
```

$$\{z_i, z_6, z_{\text{indeks}}\}$$

Jak widzimy, otrzymaliśmy w wyniku listę wszystkich obiektów wyrażenia, o których chcemy założyć, że są zespolone. Zatem możemy zastosować podobną konstrukcję w składni funkcji **ComplexExpand** i zarazem w definicjach nowych funkcji **Re2**, **Im2**.

```
Re2[wyr_] :=
 ComplexExpand[Re[wyr], Cases[{wyr}, z_, Infinity]]
Im2[wyr_] := ComplexExpand[Re[wyr],
  Cases[{wyr}, z_, Infinity]]

Re2[x + i y + z_1]
```

$$x + \text{Re}[z_1]$$

```
{Re[r e^(i φ)], Re2[r e^(i φ)]}
```

$$\{\text{Re}[e^{i\phi} r], r \text{Cos}[\phi]\}$$

3.10. Analiza wektorowa

Działania w zakresie analizy wektorowej znacznie ułatwia pakiet **VectorAnalysis`**. Wczytajmy go w znany już nam sposób.

```
<< VectorAnalysis`
```

⚠ **Uwaga** : Symbol następujący po **VectorAnalysis** nie jest zwykłym apostrofem - jest to tylko tzw. *gravis* (patrz klawisz z tyldą ~).

■ 3.10.1. Współrzędne

Jednym z pierwszych kroków jest określenie w jakim systemie współrzędnych będziemy pracować. Domyślnym (zaraz po wczytaniu pakietu) jest układ kartezjański (**Cartesian**). Poleceniami **CoordinateSystem** oraz **Coordinates[]** sprawdzamy odpowiednio jaki jest aktualnie ustawiony system współrzędnych oraz w jakiej postaci występują zmienne. Postać współrzędnych jest bardzo ważna - *Mathematica*® musi wiedzieć, które symbole oznaczają zmienne.

```
CoordinateSystem
```

```
Cartesian
```

```
Coordinates[]
```

```
{Xx, Yy, Zz}
```

Mathematica® oferuje możliwość pracy w różnych układach współrzędnych (patrz *Help*). Dla naszych zastosowań wystarczające będą: sferyczny (**Spherical**), cylindryczny (**Cylindrical**) oraz wspomniany już kartezjański. Ustawmy jako domyślny układ sferyczny oraz ustalmy postać zmiennych.

```
SetCoordinates[Spherical[r1, θ1, ϕ1]]
```

```
Spherical[r1, θ1, ϕ1]
```

```
{CoordinateSystem, Coordinates[]}
```

```
{Spherical, {r1, θ1, ϕ1}}
```

Możemy sprawdzić jakie są zakresy poszczególnych zmiennych.

```
CoordinateRanges[]
```

```
{0 ≤ r1 < ∞, 0 ≤ θ1 ≤ π, -π < ϕ1 ≤ π}
```

```
{CoordinateRanges[Cartesian],
 CoordinateRanges[Cylindrical]}
```

```
{{-∞ < Xx < ∞, -∞ < Yy < ∞, -∞ < Zz < ∞},
 {0 ≤ Rr < ∞, -π < Ttheta ≤ π, -∞ < Zz < ∞}}
```

Aby sprawdzić, jaką postać mają współrzędne domyślnego systemu (bądź innego wskazanego przez nas) w kartezjańskim układzie użyjemy **CoordinatesToCartesian**.

```
{CoordinatesToCartesian[{x, y, z}],
 CoordinatesToCartesian[{3, π/3, -π/2}, Spherical]}
```

$$\left\{\{x\,\text{Cos}[z]\,\text{Sin}[y],\ x\,\text{Sin}[y]\,\text{Sin}[z],\ x\,\text{Cos}[y]\},\ \left\{0,\ -\frac{3\sqrt{3}}{2},\ \frac{3}{2}\right\}\right\}$$

■ 3.10.2. Działania na wektorach

Ustawmy jako jako domyślny układ kartezjański wraz ze zmiennymi w postaci (x1, x2, x3).

```
SetCoordinates[Cartesian[x1, x2, x3]]
```

```
Cartesian[x1, x2, x3]
```

Omówimy teraz działania na wektorach. Poniżej przedstawione są w następującej kolejności iloczyny: skalarny (**DotProduct**), wektorowy (**CrossProduct**) oraz potrójny skalarny (**ScalarTripleProduct**), czyli $(\mathbf{x}\times\mathbf{y})\cdot\mathbf{z}$ - objętość równoległościanu rozpiętego na czynnikach iloczynu (ze znakiem zależnym od kąta między **z** oraz $\mathbf{x}\times\mathbf{y}$).

```
{x = {x1, x2, x3}, y = {y1, y2, y3}, z = {z1, z2, z3}};
```

```
{DotProduct[x, y], CrossProduct[x, y],
 ScalarTripleProduct[x, y, z]}
```

```
{x1 y1 + x2 y2 + x3 y3,
 {-x3 y2 + x2 y3, x3 y1 - x1 y3, -x2 y1 + x1 y2},
 -x3 y2 z1 + x2 y3 z1 + x3 y1 z2 - x1 y3 z2 - x2 y1 z3 + x1 y2 z3}
```

Oczywiście obliczenia wykonywane są w domyślnym układzie. Możemy je także wykonać w innych współrzędnych, bez potrzeby zmiany domyślnego.

```
DotProduct[x, y, Spherical]
```

```
x1 y1 Cos[x2] Cos[y2] + x1 y1 Cos[x3] Cos[y3] Sin[x2] Sin[y2] +
 x1 y1 Sin[x2] Sin[x3] Sin[y2] Sin[y3]
```

■ 3.10.3. Operatory różniczkowe

Zdefiniujmy następującą funkcję

```
F[x_, y_, z_] := {x^2 + 2 y z, z Sin[x], z^3 Cos[y]}
```

Podstawowe operatory różniczkowe takie jak dywergencja ($\nabla \cdot F$), wirowość ($\nabla \times F$), gradient (∇F), laplasjan ($\nabla^2 F$) oraz operator biharmoniczny ($\nabla^4 F$) przedstawimy na przykładzie funkcji **F**. W *Mathematica*® są nimi odpowiednio **Div**, **Curl**, **Grad**, **Laplacian** oraz **Biharmonic**. Operatory te używają aktualnie ustawionych: układu współrzędnych oraz postaci zmiennych.

```
SetCoordinates[Cartesian[x1, x2, x3]]
```

```
Cartesian[x1, x2, x3]
```

```
Div[F[x1, x2, x3]]
```

```
2 x1 + 3 x3^2 Cos[x2]
```

Jeżeli w wyrażeniu zmienne lub układ współrzędnych występują w postaci innej niż domyślna, to konieczne jest uwzględnienie tego faktu w składni.

```
{Curl[F[x1, y2, z3]],
 Curl[F[xx, yy, zz], Cartesian[xx, yy, zz]]}
```

```
{{0, 0, z3 Cos[x1]},
 {-Sin[xx] - zz^3 Sin[yy], 2 yy, -2 zz + zz Cos[xx]}}
```

Obliczmy gradient pierwszej oraz operator biharmoniczny trzeciej funkcji składowej odwzorowania F .

```
{Grad[F[x1, x2, x3][[1]]], Biharmonic[F[x1, x2, x3][[3]]]}
```

```
{{2 x1, 2 x3, 2 x2}, -12 x3 Cos[x2] + x3^3 Cos[x2]}
```

Poniższy przykład przedstawia laplasjan odwzorowania $(r, \phi, \theta) \mapsto (\sin(\phi), \cos(\theta), r)$ we współrzędnych sferycznych.

```
Laplacian[{Sin[ϕ], Cos[θ], r}, Spherical[r, ϕ, θ]]
```

$$\left\{-\frac{\text{Csc}[\phi]\left((\text{Cos}[\theta]-\text{Cos}[\phi])\,\text{Cos}[\phi]+\text{Sin}[\phi]^2\right)}{r^2}+\right.$$
$$\frac{\text{Csc}[\phi]\left(\text{Cos}[\theta]\,\text{Cos}[\phi]+2\,\text{Sin}[\phi]^2\right)}{r^2}-$$
$$\frac{2\,\text{Csc}[\phi]\left(r\,\text{Cos}[\theta]\,\text{Cos}[\phi]+2\,r\,\text{Sin}[\phi]^2\right)}{r^3},\ -\frac{\text{Cos}[\theta]\,\text{Csc}[\phi]^2}{r^2}+$$
$$\frac{1}{r}\left(\frac{1}{r^2}\text{Csc}[\phi]\,(-r\,\text{Cos}[\theta]\,\text{Sin}[\phi]+4\,r\,\text{Cos}[\phi]\,\text{Sin}[\phi])-\right.$$
$$\left.\frac{1}{r^2}\text{Cot}[\phi]\,\text{Csc}[\phi]\left(r\,\text{Cos}[\theta]\,\text{Cos}[\phi]+2\,r\,\text{Sin}[\phi]^2\right)\right),$$
$$\left.-\frac{\text{Cot}[\phi]\,\text{Csc}[\phi]\,\text{Sin}[\theta]}{r^2}-\frac{-1+\frac{\text{Cot}[\phi]\,\text{Csc}[\phi]\left(r^2\,\text{Cos}[\phi]+r\,\text{Sin}[\theta]\right)}{r^2}}{r}\right\}$$

```
% // Simplify
```

$$\left\{\frac{-2\,\mathrm{Cos}[\theta]\,\mathrm{Cot}[\phi]+\mathrm{Cos}[\phi]\,\mathrm{Cot}[\phi]-3\,\mathrm{Sin}[\phi]}{r^2},\ \frac{2\left(\mathrm{Cos}[\phi]-\mathrm{Cos}[\theta]\,\mathrm{Csc}[\phi]^2\right)}{r^2},\ \frac{r-r\,\mathrm{Cot}[\phi]^2-2\,\mathrm{Cot}[\phi]\,\mathrm{Csc}[\phi]\,\mathrm{Sin}[\theta]}{r^2}\right\}$$

■ 3.10.4. Obliczenia symboliczne

W rozdziale *Symboliczne obliczanie pochodnych* obliczaliśmy symbolicznie pochodne funkcji. Istnieje taka możliwość również w przypadku operatorów różniczkowych. Zdefiniujmy funkcję trzech zmiennych o wartościach w postaci wektora trójelementowego.

```
U[x_, y_, z_] := {u1[x, y, z], u2[x, y, z], u3[x, y, z]}
Div[U[x1, x2, x3]]
```

```
u3^(0,0,1)[x1, x2, x3] + u2^(0,1,0)[x1, x2, x3] + u1^(1,0,0)[x1, x2, x3]
```

Jako przykład użycia symbolicznych obliczeń przedstawimy dowód tożsamości

$$\tfrac{1}{2}\nabla(U\cdot U)=(U\cdot\nabla)\,U+U\times(\nabla\times U),$$

gdzie $U(x, y, z) = (u(x, y, z), v(x, y, z), w(x, y, z))$. Oznaczmy lewą stronę równania literą L, natomiast czynniki sumy po prawej stronie odpowiednio $P1$ oraz $P2$. Zaczniemy od ustalenia wygodnej postaci zmiennych, zdefiniowania funkcji U oraz wektora zawierającego domyślną formę zmiennych. Czytelnik zauważy, że poniższe definicje znacznie uproszczą formę wpisywanych poleceń.

```
SetCoordinates[Cartesian[x1, x2, x3]];
x = {x1, x2, x3};
U[{x_, y_, z_}] := {u[x, y, z], v[x, y, z], w[x, y, z]};
```

Obliczmy wyrażenie L.

```
L = 1/2 Grad[U[x].U[x]] // Simplify
```

```
{u[x1, x2, x3] u^(1,0,0)[x1, x2, x3] +
   v[x1, x2, x3] v^(1,0,0)[x1, x2, x3] +
   w[x1, x2, x3] w^(1,0,0)[x1, x2, x3],
  u[x1, x2, x3] u^(0,1,0)[x1, x2, x3] +
   v[x1, x2, x3] v^(0,1,0)[x1, x2, x3] +
   w[x1, x2, x3] w^(0,1,0)[x1, x2, x3],
  u[x1, x2, x3] u^(0,0,1)[x1, x2, x3] +
   v[x1, x2, x3] v^(0,0,1)[x1, x2, x3] +
   w[x1, x2, x3] w^(0,0,1)[x1, x2, x3]}
```

W celu wyznaczenia $P1$ wygodnie jest utworzyć odpowiedni operator. Zauważmy, że

$$(U \cdot \nabla)\, U = (U \cdot \nabla u,\, U \cdot \nabla v,\, U \cdot \nabla w),$$

zatem wyrażenie $P1$ otrzymamy w następujący sposób.

```
P1 = (U[x].Grad[U[x][[#]]]) & /@ {1, 2, 3}
```

```
{w[x1, x2, x3] u^(0,0,1)[x1, x2, x3] +
   v[x1, x2, x3] u^(0,1,0)[x1, x2, x3] +
   u[x1, x2, x3] u^(1,0,0)[x1, x2, x3],
  w[x1, x2, x3] v^(0,0,1)[x1, x2, x3] +
   v[x1, x2, x3] v^(0,1,0)[x1, x2, x3] +
   u[x1, x2, x3] v^(1,0,0)[x1, x2, x3],
  w[x1, x2, x3] w^(0,0,1)[x1, x2, x3] +
   v[x1, x2, x3] w^(0,1,0)[x1, x2, x3] +
   u[x1, x2, x3] w^(1,0,0)[x1, x2, x3]}
```

Z kolei zauważmy, że czynnik $\nabla \times U$ w $P2$ oznacza wirowość CurlU . Ten fakt pozwala zapisać $P2$ w poniższej postaci.

```
P2 = Cross[U[x], Curl[U[x]]]
```

```
{-w[x1, x2, x3] u^(0,0,1)[x1, x2, x3] -
   v[x1, x2, x3] u^(0,1,0)[x1, x2, x3] +
   v[x1, x2, x3] v^(1,0,0)[x1, x2, x3] +
   w[x1, x2, x3] w^(1,0,0)[x1, x2, x3],
  -w[x1, x2, x3] v^(0,0,1)[x1, x2, x3] +
   u[x1, x2, x3] u^(0,1,0)[x1, x2, x3] +
   w[x1, x2, x3] w^(0,1,0)[x1, x2, x3] -
   u[x1, x2, x3] v^(1,0,0)[x1, x2, x3],
  u[x1, x2, x3] u^(0,0,1)[x1, x2, x3] +
   v[x1, x2, x3] v^(0,0,1)[x1, x2, x3] -
   v[x1, x2, x3] w^(0,1,0)[x1, x2, x3] -
   u[x1, x2, x3] w^(1,0,0)[x1, x2, x3]}
```

Pozostaje już tylko sprawdzenie, że $L = P1 + P2$.

```
L == P1 + P2
```

```
True
```

■ 3.10.5. 0perator `ArcLengthFactor` - konstrukcja całki krzywoliniowej

Niech $\gamma(t) = (x(t), y(t), z(t))$ będzie krzywą określoną na przedziale (a, b). Jak wiemy, z podstawowego kursu analizy matematycznej, długość $l(\gamma)$ takiej krzywej obliczamy jako całkę $l(\gamma) = \int_a^b \frac{d\gamma}{dt}\, dt$, gdzie $\frac{d\gamma}{dt} = \sqrt{\left(\frac{dx}{dt}\right)^2 + \left(\frac{dy}{dt}\right)^2 + \left(\frac{dz}{dt}\right)^2}$. Operator **`ArcLengthFactor`** *Mathematica*® odpowiada $\frac{d\gamma}{dt}$.

```
Clear[x, y, z]
```

```
ArcLengthFactor[{x[t], y[t], z[t]}, t]
```

```
√(x'[t]^2 + y'[t]^2 + z'[t]^2)
```

Jako przykład zastosujmy **`ArcLengthFactor`** do obliczenia długości krzywej $c1(t) = (\sin(t), \cos(t), t)$.

```
c1[t_] := {Sin[t], Cos[t], t}
```

```
∫_0^(2 π) ArcLengthFactor[c1[t], t] dt
```

```
2 √2 π
```

Kolejnym przykładem będzie zdefiniowanie całki krzywoliniowej nieskierowanej z funkcji f określonej na krzywej γ.

```
CkrN[f_, γ_, a_, b_, N_] := If[N == 1, NIntegrate, Integrate][
  f[γ] ArcLengthFactor[γ, t], {t, a, b}]

f[{x_, y_, z_}] := x + y + z
γ := {Sin[t], Cos[t], t}

CkrN[f, γ, 0, 2 π, 1]
```

```
27.9155
```

```
c2 := {R Sin[t], R Cos[t], 0}
f1[{x_, y_, z_}] := 1

SetAttributes[R, Constant]

CkrN[f1, c2, 0, 2 π, 0]
```

$$2\,\pi\,\sqrt{R^2}$$

```
Simplify[%, Assumptions → R > 0]
```

$$2\,\pi\,R$$

Możemy pójść dalej: korzystając z twierdzenia o związku całki skierowanej z nieskierowaną, zdefiniujemy tę pierwszą. Niech $s(\gamma(t))$ będzie jednostkowym wektorem stycznym do krzywej γ w punkcie $\gamma(t)$.

```
s[γ_] := D[γ, t] / ArcLengthFactor[γ, t]
```

Wówczas całka krzywoliniowa skierowana z funkcji $F = (P, Q, R)$ określonej na krzywej γ,

```
CkrS[F_, γ_, a_, b_, N_] := CkrN[F[#].s[#] &, γ, a, b, N]

F1[{x_, y_, z_}] := {x + y, z x, 3 y}

CkrS[F1, γ, 0, 2 π, 1]
```

```
-6.72801
```

■ 3.10.6. Operatory JacobianMatrix oraz JacobianDdeterminant

Eperatory **JacobianMatrix** oraz **JacobianDeterminant** zwracają odpowiednio macierz Jacobiego przejścia od domyślnego (bądź wskazanego przez użytkownika) układu współrzędnych do kartezjańskiego.

```
{JacobianMatrix[Spherical[r, θ, ϕ]] // MatrixForm,
 JacobianDeterminant[Spherical[r, θ, ϕ]]}
```

$$\left\{\begin{pmatrix} \text{Cos}[\phi]\,\text{Sin}[\theta] & r\,\text{Cos}[\theta]\,\text{Cos}[\phi] & -r\,\text{Sin}[\theta]\,\text{Sin}[\phi] \\ \text{Sin}[\theta]\,\text{Sin}[\phi] & r\,\text{Cos}[\theta]\,\text{Sin}[\phi] & r\,\text{Cos}[\phi]\,\text{Sin}[\theta] \\ \text{Cos}[\theta] & -r\,\text{Sin}[\theta] & 0 \end{pmatrix}, r^2\,\text{Sin}[\theta]\right\}$$

```
{JacobianMatrix[{r, α, z}, Cylindrical] // MatrixForm,
 JacobianDeterminant[{R, θ, ϕ}, Cylindrical]}
```

$$\left\{\begin{pmatrix} \text{Cos}[\alpha] & -r\,\text{Sin}[\alpha] & 0 \\ \text{Sin}[\alpha] & r\,\text{Cos}[\alpha] & 0 \\ 0 & 0 & 1 \end{pmatrix}, R\right\}$$

4. Grafika

Obecnie skupimy się na zagadnieniu wykorzystania możliwości generowania obiektów graficznych w *Mathematica*®, które możemy podzielić na dwie grupy: dwuwymiarowe i trójwymiarowe. Z kolei owe grupy składają się głównie z wykresów oraz ilustracji graficznych, generowanych za pomocą wyrażeń **Graphics**, **Graphics3D** (odpowiednio dla grafiki dwuwymiarowej i trójwymiarowej). Użycie jednego z tych wyrażeń można porównać (w uproszczeniu) do rysowania na kartce papieru: pracujemy na podzbiorze płaszczyzny (bądź przestrzeni) używając współrzędnych do osadzania na nim elementów o ustalonych parametrach. Wszystkie wyrażenia graficzne, w odrębie ustalonego środowiska (2D albo 3D), możemy przedstawić w jednym układzie za pomocą operatora **Show**.
Głównym celem tego rozdziału jest przedstawienie czytelnikowi ogólnych zasad dotyczących generowania grafiki, bez wchodzenia w szczegóły. Przedstawimy konkretne przykłady, których analiza pozwoli czytelnikowi na opanowanie podstaw. Zainteresowanym tą tematyką polecamy skorzystanie z pomocy programu *Mathematica*®.

▪ 4.1. Ogólna budowa wyrażeń graficznych

Wyrażenie **Graphics** ma następującą budowę:

Graphics[{lista_elementów},opcje].

Elementy ilustracji mogą występować w postaci grupy obiektów z odpowiednim formatowaniem

{format_obiektu,Obiekt1[parametry],...,Obiektn[parametry]},

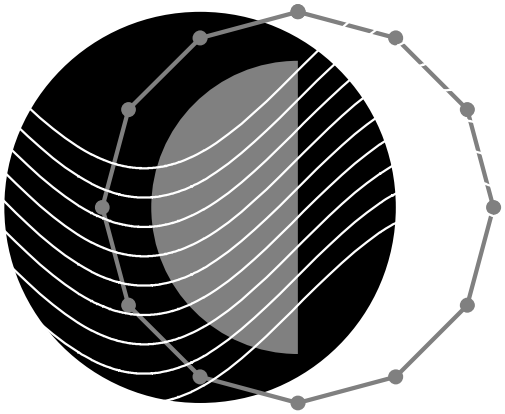
gdzie **format_obiektu** odnosi sie do wszystkich obiektów w nawiasach. Omówimy to szczegółowo na następującym przykładzie.

Niech **A1** będzie nazwą listy dwunastu punktów równo rozłożonych na okręgu o promieniu długości 2 i środku (0,0).

$$\mathbf{A1 = Table\left[\{2\,Sin[\phi],\,2\,Cos[\phi]\},\,\left\{\phi,\,0,\,2\,\pi,\,\frac{\pi}{6}\right\}\right];}$$

Poniżej znajduje sie składnia wyrażenia generującego konkretny rysunek. Wyrażenie jest zapisane w postaci umożliwiającej szybkie określenie zawartości odpowiednich nawiasów. Aby zobaczyć zawartość nawiasów, umieszczamy kursor na jednym z nich i naciskamy CTRL + SHIFT + **B** (zostanie wówczas wykonane zaznaczenie).

```
Show[Graphics[{
   Disk[{-1, 0}, 2],
   {Gray, Disk[{0, 0}, 1.5, {π / 2, 3 π / 2}],
    {PointSize[Large], Point /@ A1}},
   {Gray, Thick, Line[A1]}}],
 Plot[Table[Sin[x] + k, {k, -1, 1.5, .3}],
  {x, -π, π}, PlotStyle → {White, Thickness[.0035]}],
 PlotRange → All, AspectRatio → Automatic]
```



Omówmy kolejno budowę powyższego wyrażenia. Operator **Show** łączy dwa wyrażenia graficzne: grafiki **Graphics** oraz grupę wykresów kilku funkcji postaci $\sin(x)+k$ o kolorze białym. Pierwszy z nawiasów w wyrażeniu **Graphics** otwiera listę wszystkich elementów, w skład której wchodzą kolejno:

1) **Disk[{-1, 0},2]** - obiekt z domyślnym formatowaiem: kolor czarny (**Black**) - (koło o środku w punkcie (-1,0) i promieniu długości 2),

2) **{Gray,Disk[{0,0},1.5,{π/2,3π/2}],{PointSize[Large],Point/@A1}}** - lista obiektów, w odrębie której obowiązuje formatowanie: "zmiana koloru domyślnego na szary (**Gray**)" - (półkole oraz zbiór punktów listy **A1** - zauważmy, że

```
{PointSize[Large], Point /@ A1}
```

```
{PointSize[Large], {Point[{0, 2}],
  Point[{1, √3}], Point[{√3, 1}], Point[{2, 0}],
  Point[{√3, -1}], Point[{1, -√3}], Point[{0, -2}],
  Point[{-1, -√3}], Point[{-√3, -1}], Point[{-2, 0}],
  Point[{-√3, 1}], Point[{-1, √3}], Point[{0, 2}]}}
```

Zatem operator **Point** zastosowany został w postaci funkcji do każdego elementu listy, w wyniku czego powstała nowa lista, w obrębie której obowiązuje formatowanie wielkości punktu **PointSize[Large]**),
3) **{Gray,Thick,Line[A1]}** - łamana o wierzchołkach z listy **A1**, dla której określono kolor (**Gray**) oraz grubość (**Thick**).

Jak łatwo zauważyć, obiekty wyświetlone są "warstwowo" według kolejności występowania, tzn. każdy kolejny obiekt nakładany jest na pozostałe. Wyrażenie jest zakończone formatowaniem całego rysunku; w tym przypadku jest to znana nam już opcja **AspectRatio→ Automatic** oraz **PlotRange→All**.

▪ 4.2. Ilustracja liczby zespolonej oraz jej pierwiastków stopnia *n* na płaszczyźnie

Niech $z = 1 + i$ oraz $n = 5$.

```
{n = 5, z = 1 + i};
```

Korzystając ze wzoru de Moivre'a wyznaczmy zbiór pierwiastków stopnia *n* z liczby *z*.

```
pnz := Table[Sqrt[Abs[z], n]
    (Cos[(Arg[z] + 2 k π)/n] + i Sin[(Arg[z] + 2 k π)/n]), {k, 0, n - 1}]
pnz^n // Simplify
{1 + i, 1 + i, 1 + i, 1 + i, 1 + i}
```

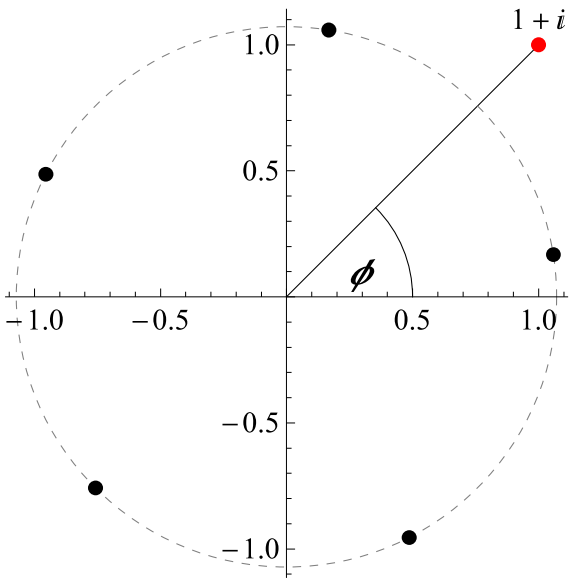
Aby umiejscowić elementy zbioru **pnz** na płaszczyźnie, musimy przedstawić je w postaci punktów. Skonstruujmy funkcję, która będzie przekształcać liczbę *z* na odpowiadający jej punkt $(\mathrm{re}\, z,\ \mathrm{im}\, z)$ na płaszczyźnie.

```
c2r[z_] := {Re[z], Im[z]}
```

Wówczas ilustracja pierwiastków może wyglądać nastepująco.

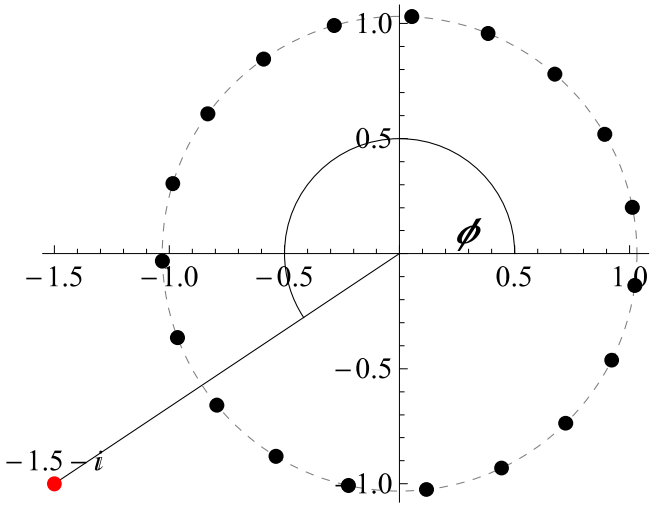
```
gr1 := Graphics[{Line[{{0, 0}, c2r[z]}],
    {Gray, Dashed, Circle[{0, 0}, Sqrt[Abs[z], n]]},
    Circle[{0, 0}, .5, {0, Arg[z] + If[Arg[z] < 0, 2 π, 0]}],
    {PointSize[Large], Point /@ c2r /@ pnz,
     {Red, Point[c2r[z]]}},
    Text[Style[z, FontSize → 14], c2r[z] + {0, .1}],
    Text[Style["ϕ", Italic, Bold, FontSize → 18], {.3, .1}]},
  AspectRatio → Automatic, Axes → True]
```

```
gr1
```



Należy wspomnieć o nieznanym jeszcze operatorze **Text**. Pozwala on włączyć do ilustracji tekst, poprzez określenie współrzędnych jego położenia. Jeżeli tekst podamy w cydzysłowiu, to bedzie on potraktowany jako łańcuch znaków (patrz np. **"ϕ"**), w przeciwnym wypadku może wyświetlić zawartość użytej nazwy (u nas np. symbol liczby **z**). Możemy dodatkowo określić jego parametry; np. wielkość - **FontSize**, pochylenie - **Italic**, pogrubienie - **Bold**. Omówmy także krótko konstrukcję **Point/@c2r/@pnz**. Złożenie to zamienia najpierw (funkcja **c2r**) listę **pnz** liczb zespolonych na listę punktów, następnie zwraca listę punktów. Użyliśmy także formatowania grafiki wyświetlającej na obrazku osie układu - **Axes→True**. Zauważmy, że po ponownym zdefiniowaniu parametrów *z* oraz *n* i obliczeniu symbolu **gr1** nasza ilustracja ulegnie zmianie.

```
{z = -1.5 - i, n = 19};
gr1
```



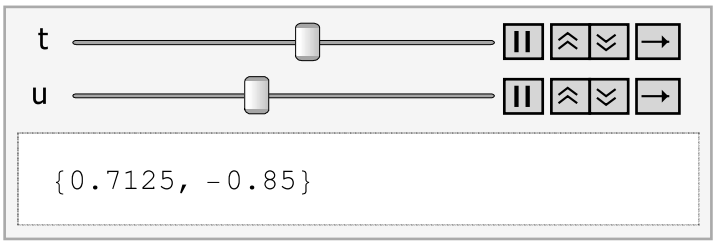
5. Wizualizacje oraz moduły dynamiczne

W rozdziale tym omówimy podstawowe operatory *Mathematica*® umożliwiające konstruowanie dynamicznych wizualizacji.

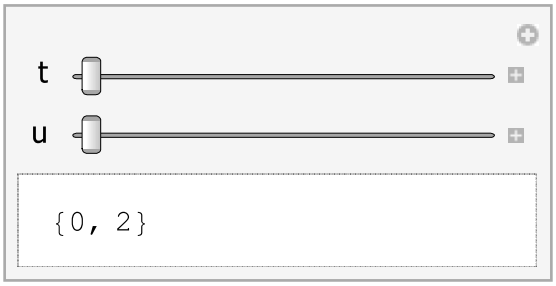
■ 5.1. Operatory `Manipulate` oraz `Animate`

Podstawowa grupa wyrażeń dynamicznych zawiera **`Manipulate`** oraz **`Animate`**. Konstrukcja każdego z nich zawiera z kolei wyrażenie oraz określenie parametrów zmiennych wraz z ich zakresami. Wyrażenie będzie aktualizowane z każdą zmianą któregokolwiek z parametrów. Rozpatrzmy nastepujące przykłady.

```
Animate[{t, u}, {t, 0, 1}, {u, 2, -2}]
```




```
Manipulate[{t, u}, {t, 0, 1}, {u, 2, -2}]
```



Omówmy krótko każde z wyrażeń. W **`Animate`** możemy zatrzymać animację ze względu na niektóre parametry, sterować prędkością animacji, oraz zmieniać kierunek. W przypadku **`Manipulate`** możemy, naciskając "**+**" obok manipulatora rozwinąć podmenu, w którym znajdują się m.in. opcje z **`Animate`**, okienko w którym można obserwować aktualną wartość parametru lub wpisać żądaną. W dalszym ciągu skupimy się jedynie na operatorze **`Manipulate`**.

Jako przykład zastosowania **`Manipulate`** zbudujemy wizualizację zagadnienia z Części I dotyczącego spadku swobodnego. Rozwiążmy w tym celu równanie różniczkowe. Przyjmijmy $g = 9.81$.

```
Clear[x, g, t, y, T]
DSolve[{x''[t] == 9.81, x[0] == 0, x'[0] == v0}, x[t], t]
```

```
{{x[t] → 4.905 t^2 + t v0}}
```

Powyższe rozwiązanie jest zależne od t oraz v_0. Na jego podstawie zdefiniujemy funkcję $y(t, v_0)$.

```
y[t_, v0_] = x[t] /. %[[1]]
```

```
4.905 t^2 + t v0
```

Wyznaczmy czas t, w którym ciało osiągnie wysokość $h = 0$ przy danej prędkości początkowej v_0. Wówczas różnica wysokości początkowej i całkowitej drogi przebytej przez ciało jest równa 0.

```
Solve[h0 - y[t, v0] == 0, t]
```

```
{{t → 0.101937 (-1. v0 - 4.42945 Sqrt[1. h0 + 0.0509684 v0^2])},
 {t → 0.101937 (-1. v0 + 4.42945 Sqrt[1. h0 + 0.0509684 v0^2])}}
```

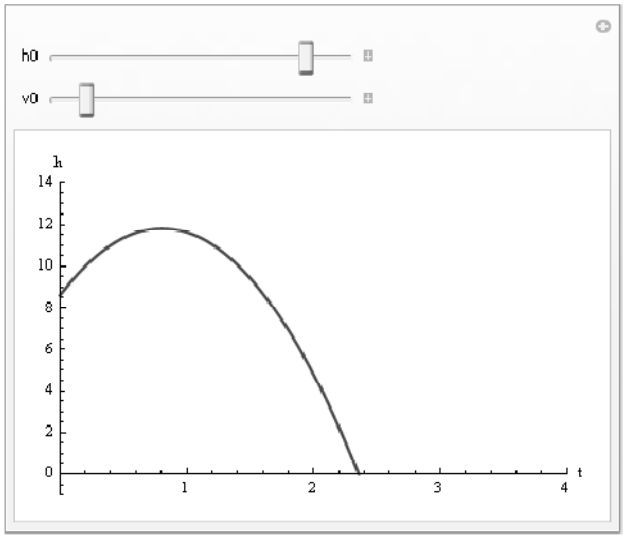
Zdefiniujmy funkcję zmiennych h oraz v_0.

```
T[h0_, v0_] = t /. %[[2, 1]]
```

```
0.101937 (-1. v0 + 4.42945 Sqrt[1. h0 + 0.0509684 v0^2])
```

Pozostaje skonstruowanie odpowiedniego wyrażenia za pomocą **Manipulate**. Teraz możemy obserwować, jak będzie się zmieniał wykres zależności $h(t)$ przy zmianie wartości początkowych: wysokości i prędkości.

```
Manipulate[Plot[h0 - y[t, v0], {t, 0, T[h0, v0]},
  AxesLabel → {"t", "h"}, PlotRange → {{0, 4}, {-1, 14}}],
 {h0, 0, 10}, {v0, -10, 10}, ControlPlacement → Top]
```

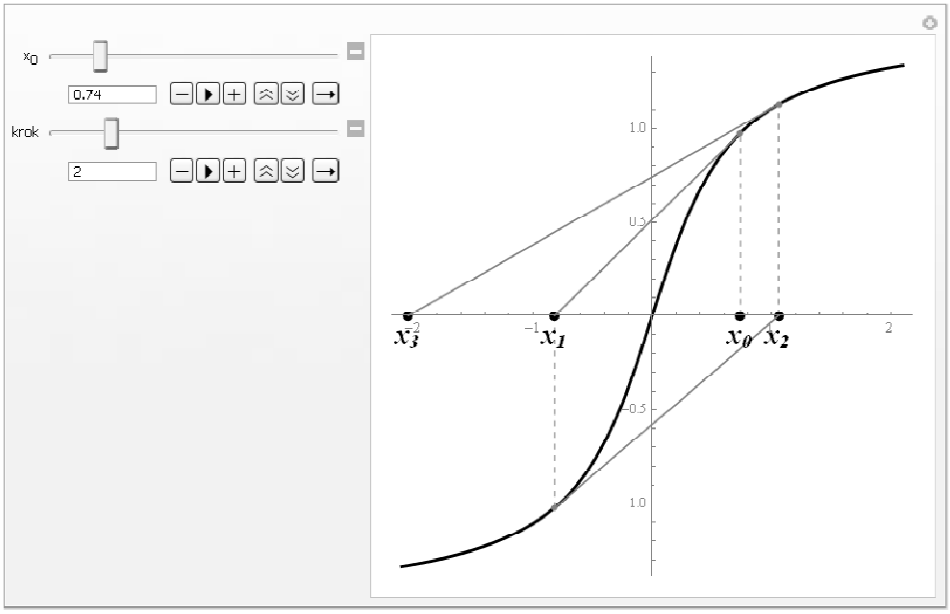


Poniższy kod realizuje wizualizację przykładu dotyczącego metody Newtona (patrz Część I).

$$x_{k\_} := x_{k-1} - \frac{f[x_{k-1}]}{f'[x_{k-1}]}$$

```
f[x_] := ArcTan[2 x]

Manipulate[x0 = t; A1 = Table[xk, {k, 0, i}];
 Show[Graphics[{{Lighter[Gray], Dashed, Thickness[.0035],
     Line[{{#, 0}, {#, f[#]}} & /@ A1]}, {PointSize[Large],
     Point[{x0, 0}], Point /@ {{# - f[#]/f'[#], 0}} & /@ A1}}],
  Plot[f[x], {x, -2.1, 2.1}, PlotStyle ->
    Directive[Black, Thickness[.006]]],
  Graphics[{{{Gray, Thickness[.0035],
     Line[{{#, f[#]}, {# - f[#]/f'[#], 0}} & /@ A1]},
    {PointSize[Medium], Gray, Point /@ {{{#, f[#]}}} & /@ A1},
    {Text[Style["x"#, FontSize → 20, Bold, Italic],
       {x#, -0.1}] & /@ Table[k, {k, 0, i + 1, 1}]}}],
  Axes → True, AxesStyle → Gray, AspectRatio → 1]
 , {{t, 0.65, "x0"}, .5, 2, .01, Appearance → "Open"},
 {{i, 0, "krok"}, 0, 10, 1, Appearance → "Open"},
 ControlPlacement → Top]
```



Animate oraz **Manipulate** oferują także dodatkowe opcje (np. autostart animacji, ustawienia początkowych wartości parametrów, rodzaj manipulatorów itp.), jednak nie będziemy ich omawiać - zainteresowany czytelnik bez trudu pozna je samodzielnie dzięki plikom pomocy programu *Mathematica*®.

5.2. Operator `Dynamic`

5.2.1. Zmienne dynamiczne i moduł dynamiczny

Uogólnieniem poznanych wyżej wyrażeń **Manipulate** oraz **Animate** jest zastosowanie operatora **Dynamic**. Aby zrozumieć jego istotę, zacznijmy od prostego przykładu

```
Clear[v]
```

```
Dynamic[v]
```

```
0.
```

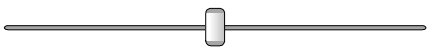
Powyższy zapis oznacza, że postać (wartość) obiektu **v** będzie automatycznie uaktualniana wraz ze zmianami w dokumencie. Istotnie, przypiszmy symbolowi **v** wartość równą 1. Wówczas zauważymy, że powyższa wartość zmieni się.

```
v = 1;
```

```
v = Graphics[Rectangle[], ImageSize → Tiny]
```

Rozpatrzmy operator **Slider**.

```
Slider[]
```



Otrzymaliśmy suwak; jednak jego użycie nie ma jeszcze żadnych konsekwencji. Możemy sprawić, aby powodował zmiany wartości **v** w dynamiczny sposób.

```
Slider[Dynamic[v]]
```

Teraz użycie suwaka skutkuje zmianami wartości sumbolu **v** w miejsu jego wystąpienia. Zatem niezależnie w którym miejscu dokumentu znajduje się **v**, jego postać będzie się zmieniać na bieżąco. Jeżeli chcemy, aby zmiany zachodziły tylko lokalnie możemy użyć otoczenia **DynamicModule**.

```
DynamicModule[{v2 = 0}, {Slider[Dynamic[v2]], Dynamic[v2]}]
```

```
Dynamic[v2]
```

```
v2
```

Jak widzimy, poza **DynamicModule** zmiany nie zachodzą. Istotnym elementem składni tego operatora jest pierwszy nawias, w którym występują symbole dynamiczne, użyte w odrębie wyrażenia znajdujacego się po przecinku. Jak w powyższym przykładzie, możemy im przypisać wartości początkowe.

W przypadku **Manipulate** mogliśmy wpisywać w okienku interesującą nas wartość, bez "szukania" jej za pomocą manipulatora. Tutaj odpowiednikiem tego postępowania jest zastosowanie operatora **InputField**.

```
InputField[Dynamic[tresc]]
```

```
Dynamic[tresc]
```

```
tresc
```

Po wpisaniu treści i naciśnieciu ENTER następuje zmiana postaci **tresc**.

Poznamy jeszcze jeden z manipulatorów przydanych w przypadku konstrukcji dynamicznych, mianowicie **Button**. Jego składnię i działanie przedstawimy na następującym, bardzo prostym przykładzie.

```
n = 0;

Button["powiększ n o 1", n = n + 1]
Dynamic[n]
```

```
0
```

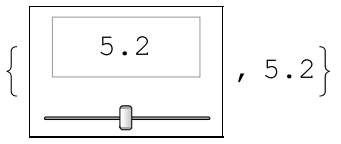
■ 5.2.2. Budowa nowego manipulatora

Wszystkie manipulatory w powyższych konstrukcjach posiadają ograniczony zakres zmiennych. Zbudujemy operator, dzięki któremu będziemy mogli w dowolny sposób zmieniać wartość określonej zmiennej.

```
DynamicModule[{xx = 0, w1 = 0},
 {Slider[Dynamic[xx], {-1, 1, .01}],
  Dynamic[w1 = w1 + (xx = xx - ControlActive[0, xx])]}]
```

Konstrukcja **ControlActive[wyr1,wyr2]** zwraca wartość **wyr1**, jeżeli manipulator jest "używany". W przeciwnym razie obliczane jest wyrażenie **wyr2**. Zatem w momencie odchylenia suwaka zaczyna być odejmowana od **w1** wartość **xx**. Im większe odchylenie, tym szybciej rośnie (maleje) wartość **w1**. Jeżeli uwolnimy suwak, to wówczas wróci on na swoje miejsce (tzn. **xx=0**). Użyjemy teraz operatora oraz dodatkowych opcji aby zbudować manipulator, w którym możemy wpisać z klawiatury wartość, która zastąpi aktualną po naciśnieciu ENTER.

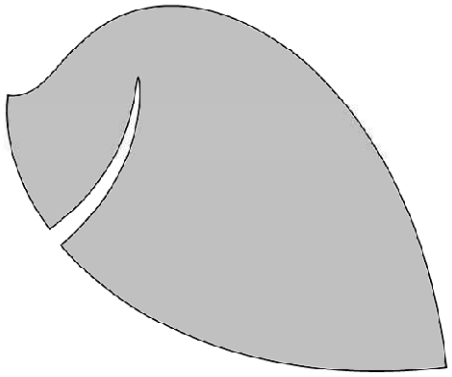
```
DynamicModule[{xx = 0, w1 = 0},
 {Column[{InputField[Dynamic[w1],
      FieldSize → {5, 1.5}, Alignment -> Center],
    {Slider[Dynamic[xx], {-.3, .3, .05}, ImageSize → 80], w1 =
       w1 + (xx = xx - ControlActive[0, xx])}[[1]] // Dynamic},
  Frame → True, Alignment -> Center], Dynamic[w1]}]
```



■ 5.2.3. Generowanie siatek obszarów za pomocą odwzorowań konforemnych

Podstawowe zastosowanie metody elementów skończonych jest oparte na podziale rozważanego obszaru D na elementy (np. prostokątne, trójkątne). Elementy czworokątne pozwalają w wystarczający sposób przybliżyć obszar o skomplikowanym kształcie. W celu otrzymania elementu czworokątnego o odpowiednim kształcie wystarczy dokonać przekształcenia elementu prostokątnego. Przekształcając wiele elementów, otrzymamy nową siatkę. Problemem jest właśnie generowanie siatek elementów skończonych obszarów o różnych kształtach. Zrealizujemy rozwiązanie tego problemu z wykorzystaniem odwzorowań konforemnych, konstruując gotowy program do generowania siatek elementów różnych obszarów.

Rozpatrzmy następujący przykład. Jest dany pewien obszar D



oraz odwzorowanie konforemne $w(z) = \sin\left(0.5 + e^{\frac{5\pi i}{4}}(z - 0.76)\right)$ przekształcające prostokąt $[-2.4, 2.4]\times[-0.7, 0.7]$ na D. Ustalmy następujące parametry

```
w1 = 4.8; w2 = 1.4; pxs = 12; pys = 4; px = 1; py = 1;
```

oraz listy

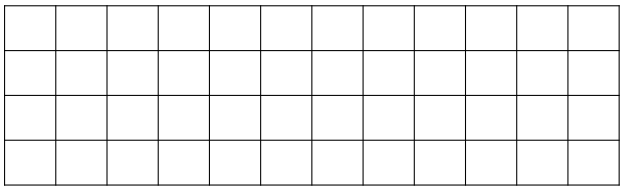
```
X := Table[x + i y, {x, -w1/2, w1/2, w1/(pxs px)}]
Y := Table[x + i y, {y, -w2/2, w2/2, w2/(pys py)}]
XX := Table[X, {y, -w2/2, w2/2, w2/pys}]
YY := Table[Y, {x, -w1/2, w1/2, w1/pxs}]
```

Otrzymane w ten sposób listy **XX**, **YY** składają się z list liczb zespolonych zawartych w prostokącie $[-2.4, 2.4]\times[-0.7, 0.7]$. Zdefiniujmy następujące operatory.

```
c2r[z_] := {Re[z], Im[z]}
linec[list_] := Line[c2r /@ list]
```

linec jest operatorem, który w środowisku **Graphics** tworzy łamaną o wierzchołkach z listy liczb zespolonych. Poniższa konstrukcja powoduje powstanie siatki prostokąta.

```
Graphics[(linec /@ # &) /@ {XX, YY}]
```



Niech $w(z)$ będzie wspomnianym odwzorowaniem konforemnym

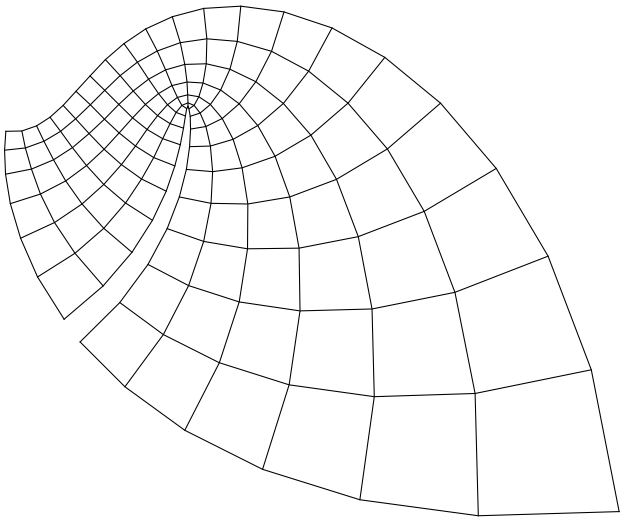
```
w[z_] := Sin[0.5 + e^(5 π i/4) (z - 0.76)]
```

Zbudujmy listy, które zawierają obrazy punktów zespolonych z list **XX**, **YY** przez odworowanie w.

```
XX1 := (w /@ #) & /@ XX
YY1 := (w /@ #) & /@ YY
```

Wówczas możemy wyświetlić obraz siatki przekształconej przez w.

```
Graphics[(linec /@ # &) /@ {XX1, YY1}]
```



Siatkę możemy uczynić dokładniejszą, poprzez zwiększenie parametrów **pxs**, **pys** (odpowiedzialne za ilość elementów) lub **px**, **py** (odpowiedzialne za liczbę węzłów w jednym elemencie). Obliczenie powyższych komórek na nowo spowoduje powstanie nowej siatki.

Zdefiniujmy operator o nazwie **trob**, który będzie umożliwiał translację węzłów siatki na płaszczyźnie, obrót wokół początku układu współrzędnych oraz przekształcanie węzłów względem danego odwzorowania *w*.

```
trob[list_, z0_, ϕ_, w_] := w[z0 + e^(i ϕ) #] & /@ list
```

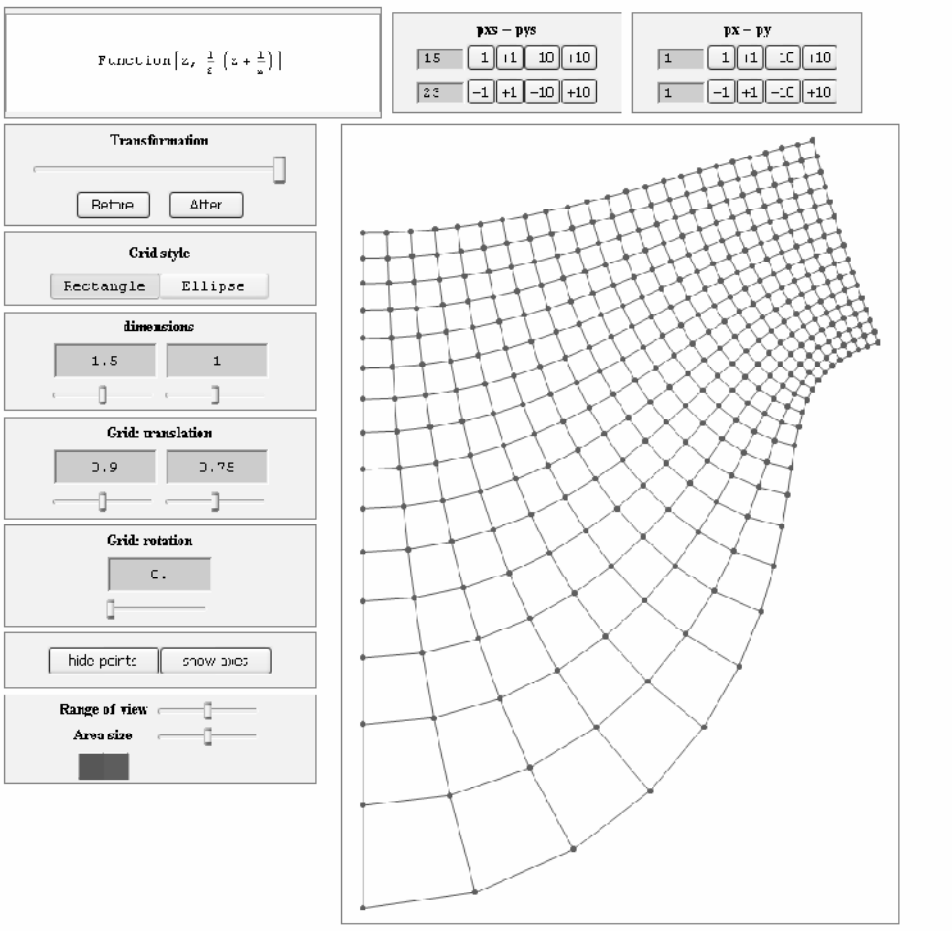
Wówczas nastepująca konstrukcja da wynik identyczny z powyższym (bez potrzeby uprzedniego definiowania odwzorowania *w* oraz list **XX1**, **YY1**).

```
Graphics[(linec /@ # &) /@
  (trob[#, 0.5 - e^(5 π i/4) 0.76, 5/4 π, Function[z, Sin[z]]] & /@
    {XX, YY})];
```

Poniższe wyrażenie daje możliwość obserwacji transformacji siatki wyjściowej na siatkę elementów, na które został podzielony obszar D.

```
Manipulate[Graphics[(linec /@ # &) /@
  (trob[#, 0, 0, Function[z, t Sin[0.5 + e^(5 π i/4) (z - 0.76)] +
      (1 - t) z]] & /@ {XX, YY})], {t, 0, 1}];
```

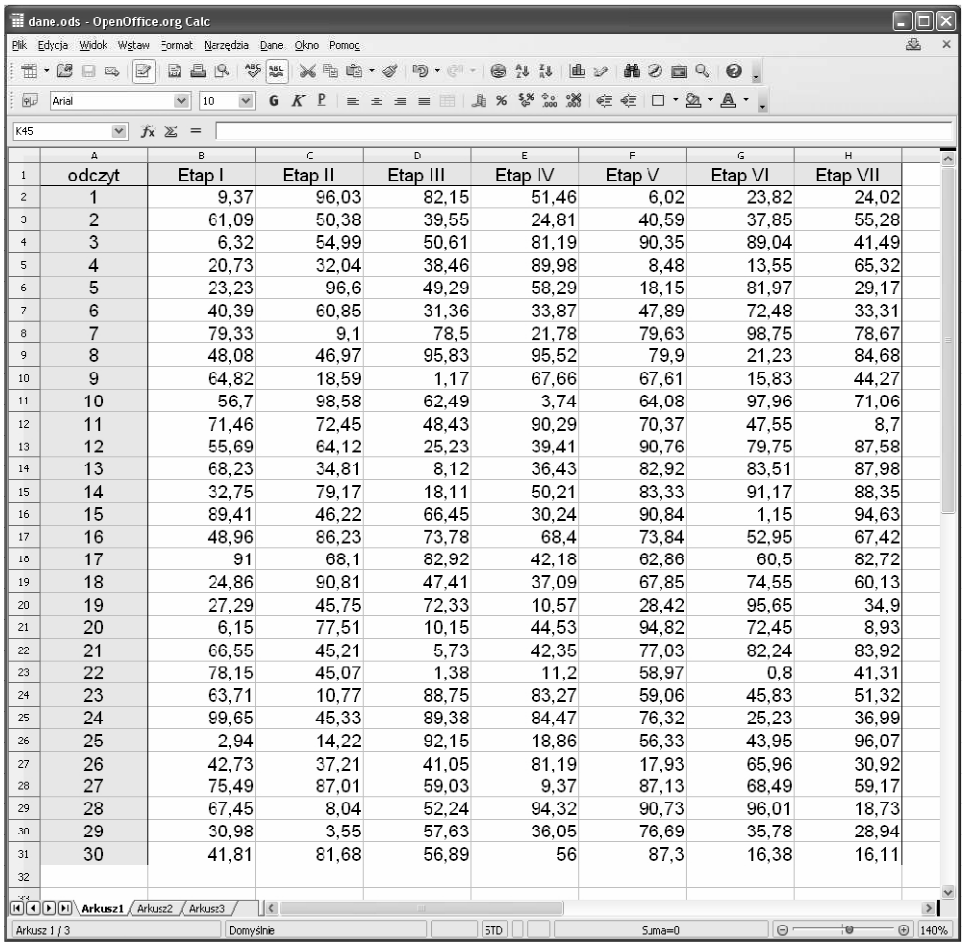
Skonstruowaliśmy w ten sposób podział obszaru D na elementy skończone. Załóżmy, że chcemy pracować z innymi obszarami (czyli z wykorzystaniem innego odwzorowania konforemnego *w* oraz innych niż powyższe parametrów). W oparciu o powyższe informacje, możliwe jest zbudowanie programu przeznaczonego do tego celu, i w nim - automatyczna konstrukcja siatki. Kod źródłowy takiego programu przedstawiony jest w *Dodatku A*. Zachęcamy czytelnika do zrozumienia jego budowy. Poniżej przedstawiona jest ilustracja wyglądu skompilowanej już aplikacji



6. Importowanie i eksportowanie

6.1. Importowanie danych z arkusza kalkulacyjnego OpenOffice.org Calc

Załóżmy, że chcemy wykorzystać w *Mathematica*® dane, zebrane w postaci arkusza kalkulacyjnego zapisanego w pliku **dane.ods**.



	A	B	C	D	E	F	G	H
1	odczyt	Etap I	Etap II	Etap III	Etap IV	Etap V	Etap VI	Etap VII
2	1	9,37	96,03	82,15	51,46	6,02	23,82	24,02
3	2	61,09	50,38	39,55	24,81	40,59	37,85	55,28
4	3	6,32	54,99	50,61	81,19	90,35	89,04	41,49
5	4	20,73	32,04	38,46	89,98	8,48	13,55	65,32
6	5	23,23	96,6	49,29	58,29	18,15	81,97	29,17
7	6	40,39	60,85	31,36	33,87	47,89	72,48	33,31
8	7	79,33	9,1	78,5	21,78	79,63	98,75	78,67
9	8	48,08	46,97	95,83	95,52	79,9	21,23	84,68
10	9	64,82	18,59	1,17	67,66	67,61	15,83	44,27
11	10	56,7	98,58	62,49	3,74	64,08	97,96	71,06
12	11	71,46	72,45	48,43	90,29	70,37	47,55	8,7
13	12	55,69	64,12	25,23	39,41	90,76	79,75	87,58
14	13	68,23	34,81	8,12	36,43	82,92	83,51	87,98
15	14	32,75	79,17	18,11	50,21	83,33	91,17	88,35
16	15	89,41	46,22	66,45	30,24	90,84	1,15	94,63
17	16	48,96	86,23	73,78	68,4	73,84	52,95	67,42
18	17	91	68,1	82,92	42,18	62,86	60,5	82,72
19	18	24,86	90,81	47,41	37,09	67,85	74,55	60,13
20	19	27,29	45,75	72,33	10,57	28,42	95,65	34,9
21	20	6,15	77,51	10,15	44,53	94,82	72,45	8,93
22	21	66,55	45,21	5,73	42,35	77,03	82,24	83,92
23	22	78,15	45,07	1,38	11,2	58,97	0,8	41,31
24	23	63,71	10,77	88,75	83,27	59,06	45,83	51,32
25	24	99,65	45,33	89,38	84,47	76,32	25,23	36,99
26	25	2,94	14,22	92,15	18,86	56,33	43,95	96,07
27	26	42,73	37,21	41,05	81,19	17,93	65,96	30,92
28	27	75,49	87,01	59,03	9,37	87,13	68,49	59,17
29	28	67,45	8,04	52,24	94,32	90,73	96,01	18,73
30	29	30,98	3,55	57,63	36,05	76,69	35,78	28,94
31	30	41,81	81,68	56,89	56	87,3	16,38	16,11
32								

Za pomocą operatora **Import** możemy importować do notatnika programu *Mathematica*® obiekty różnych formatów. Jednym z nich jest arkusz OpenOffice.org Calc. W składni operatora należy podać ścieżkę do pliku wraz z jego nazwą i rozszerzeniem.

```
Import["ścieżka_do_pliku\plik.rozszerzenie"]
```

Przedstawimy szybki sposób wydobycia konkretnych danych z arkusza. Niech notatnik, w którym pracujemy i plik **dane.ods** będą zapisane w tym samym folderze. Wówczas polecenie **NotebookDirectory[]** zwraca ścieżkę do owego katalogu.

```
NotebookDirectory[]
```

```
C:\Documents and Settings\wojtek\Pulpit\T3M\
```

Zbudujemy operator, który będzie nazwie pliku zapisanego w katalogu z dokumentem *Mathematica*® przypisywał gotową ścieżkę.

```
notkat[x_String] := FileNameJoin[{NotebookDirectory[], x}]
```

```
notkat["dane.ods"]
```

```
C:\Documents and Settings\wojtek\Pulpit\T3M\dane.ods
```

Teraz możemy importować dane z pliku **dane.ods**. Poniższe polecenie importuje w postaci list wszystkie dane z arkusza o numerze **k**.

```
Import["dane.ods" // notkat, {"Data", k}]
```

Jednak załóżmy, że chcemy wydobyć dane, zebrane w etapach II-VI z odczytami 1-25. Stanowi to wówczas kolumny 3-7 i wiersze 2-26.

```
data = Import["dane.ods" // notkat,
  {"Data", 1, Range[2, 26], Range[3, 7]}]
```

```
{{96.0266, 82.1503, 51.4618, 6.01501, 23.8159},
 {50.3815, 39.5538, 24.8077, 40.5853, 37.854},
 {54.9866, 50.6134, 81.1859, 90.3534, 89.035},
 {32.0374, 38.4583, 89.978, 8.48389, 13.5468},
 {96.6003, 49.2889, 58.2947, 18.1549, 81.9672},
 {60.8459, 31.3629, 33.8684, 47.8943, 72.4762},
 {9.10339, 78.4973, 21.7834, 79.6326, 98.7457},
 {46.9727, 95.8313, 95.52, 79.8981, 21.2341},
 {18.5913, 1.17188, 67.6605, 67.6147, 15.8295},
 {98.584, 62.4908, 3.74451, 64.0839, 97.9553},
 {72.4548, 48.4253, 90.2863, 70.3674, 47.5494},
 {64.1174, 25.2319, 39.4104, 90.7562, 79.7516},
 {34.8145, 8.12073, 36.4319, 82.9163, 83.5144},
 {79.1748, 18.1061, 50.2106, 83.3252, 91.1743},
 {46.225, 66.449, 30.2399, 90.8417, 1.15356},
 {86.2274, 73.7823, 68.399, 73.8373, 52.951},
 {68.1, 82.9193, 42.1753, 62.8601, 60.5042},
 {90.8142, 47.406, 37.0941, 67.8528, 74.5544},
 {45.7458, 72.3297, 10.5652, 28.421, 95.6543},
 {77.5116, 10.1532, 44.5313, 94.8242, 72.4548},
 {45.2118, 5.72815, 42.3523, 77.0325, 82.2449},
 {45.0714, 1.37634, 11.203, 58.9722, 0.802612},
 {10.7697, 88.7482, 83.2733, 59.0576, 45.8252},
 {45.3308, 89.3829, 84.4696, 76.3153, 25.2289},
 {14.2242, 92.1539, 18.8599, 56.3324, 43.9514}}
```

Otrzymaliśmy dane w postaci list. Od tego momentu możemy pracować z nimi za pomocą *Mathematica*®. Operator **TableForm** wyświetla listy w postaci tabeli.

```
data // TableForm
```

```
96.0266  82.1503  51.4618  6.01501  23.8159
50.3815  39.5538  24.8077  40.5853  37.854
54.9866  50.6134  81.1859  90.3534  89.035
32.0374  38.4583  89.978   8.48389  13.5468
96.6003  49.2889  58.2947  18.1549  81.9672
60.8459  31.3629  33.8684  47.8943  72.4762
9.10339  78.4973  21.7834  79.6326  98.7457
46.9727  95.8313  95.52    79.8981  21.2341
18.5913  1.17188  67.6605  67.6147  15.8295
98.584   62.4908  3.74451  64.0839  97.9553
72.4548  48.4253  90.2863  70.3674  47.5494
64.1174  25.2319  39.4104  90.7562  79.7516
34.8145  8.12073  36.4319  82.9163  83.5144
79.1748  18.1061  50.2106  83.3252  91.1743
46.225   66.449   30.2399  90.8417  1.15356
86.2274  73.7823  68.399   73.8373  52.951
68.1     82.9193  42.1753  62.8601  60.5042
90.8142  47.406   37.0941  67.8528  74.5544
45.7458  72.3297  10.5652  28.421   95.6543
77.5116  10.1532  44.5313  94.8242  72.4548
45.2118  5.72815  42.3523  77.0325  82.2449
45.0714  1.37634  11.203   58.9722  0.802612
10.7697  88.7482  83.2733  59.0576  45.8252
45.3308  89.3829  84.4696  76.3153  25.2289
14.2242  92.1539  18.8599  56.3324  43.9514
```

■ 6.2. Konwersja plików *.BMP oraz *.JPG do formatu *.TIFF

W pewnym katalogu znajdują się pliki z rozszerzeniami *.jpg albo *.bmp. Przeprowadzimy ich konwersję na format TIFF w przestrzeni barw CMYK i z rozdzialczością 300DPI. Ustalmy domyślny katalog roboczy (w miejscu **Path to Directory** wpisujemy ścieżkę do katalogu z plikami).

```
SetDirectory["Path to Directory"]
```

Niech **FIn** będzie listą wszystkich plików o rozszerzeniach *.jpg albo *.bmp, znajdujących się w katalogu zadeklarowanym powyżej.

```
FIn = FileNames@{"*.JPG", "*.BMP"}
```

Poniższa lista zawiera nazwy plików ze zmienionymi rozszerzeniami.

```
FOut =
 StringReplace[#, _ ~~ _ ~~ _ ~~ EndOfString → "TIF"] & /@ FIn
```

Istotnie

```
StringReplace["nazwa.jpg",
   _ ~~ _ ~~ _ ~~ EndOfString → "TIF"] & /@
 {nazwa1.jpg, nazwa2.bmp}
```

```
{nazwa.TIF, nazwa.TIF}
```

Zrozumienie konstrukcji następującego polecenia, konwertującego pliki ze wspomnianego katalogu, pozostawiamy czytelnikowi.

```
MapThread[
 Export[#2, ImageResize[Import[#1], Scaled[300/96]], "TIFF",
   ImageResolution → 300, "ColorSpace" → "CMYKColor",
   "ImageEncoding" → "ZIP"] &, {FIn, FOut}]
```

7. Dodatek A. Kod programu

Ponizej prezentujemy kod programu do generowania siatek za pomocą odwzorowań konforemnych, o którym mowa w rozdziale *Wizualizacje oraz moduły dynamiczne*. Pierwszym elementem budowy kodu jest wczytanie potrzebnych funkcji.

```
c2r[z_] := {Re[z], Im[z]}
linec[list_] := Line[c2r /@ list]
trobw[list_, z_, ϕ_, f_] := f[z + e^(i ϕ) (#)] & /@ list
trobf[list_, z_, ϕ_, f_] := trobw[#, z, ϕ, f] & /@ list
```

oraz parametrów ustawień.

```
{kol1, kol2} = {Hue[.6, 1, 1], Hue[.9, 1, 1]};
ust[1] := {Function[z, 1/2 (z + 1/z)], "elips",
  2, 2, 1, 2 π, 1, 3, 6, 13, 0, 0, 0, 1, 0, 20, 0}
set = ust[1];
```

Teraz następuje otwarcie modułu **DynamicModule**, w którym działać będzie nasz program.

```
DynamicModule[(* zbiór wartości lokalnych,
 wczytywany z listy ust[1] *)
 {w = set[[1]], fig = set[[2]], w1 = set[[3]], w2 = set[[4]], w3 = set[[5]],
  Θ = set[[6]], px = set[[7]], py = set[[8]], pxs = set[[9]],
  pys = set[[10]], ϕ = set[[11]], z01 = set[[12]], z02 = set[[13]],
  punkt = set[[14]], o1 = set[[15]], o2 = set[[16]], ax = set[[17]]},

 (********  definiowanie manipulatorów  ********)
 (* gestość pxs-pys *)
 opt21 =
  Column[{
    Column[{Text[Style["pxs - pys", Bold]],
```

```
      Row[{InputField[Dynamic[If[pys < 1, pys = 1;, pys]],
          FieldSize → {2, 1}, Enabled → False, Background →
           Lighter[kol1, .7]], Button["-1", pys = pys - 1;],
        Button["+1", pys = pys + 1;], Button["-10",
         pys = pys - 10;], Button["+10", pys = pys + 10;]}],
      Row[{InputField[Dynamic[If[pxs < 1, pxs = 1;, pxs]],
          FieldSize → {2, 1}, Enabled → False,
          Background → Lighter[kol2, .7]], Button["-1",
          pxs = pxs - 1;], Button["+1", pxs = pxs + 1;],
        Button["-10", pxs = pxs - 10;], Button["+10",
         pxs = pxs + 10;]}]}, Alignment → Center]},
  ItemSize → {13, 8}, Frame → True, Alignment →
   {Center, Center}, FrameStyle → Gray,
  Background → Lighter[Gray, .9]];
(*gestość px-py*)
opt22 = Column[
  {Column[{Text[Style["px - py", Bold]], Row[{InputField[
         Dynamic[If[px < 1, px = 1;, px]], FieldSize → {2, 1},
         Enabled → False, Background → Lighter[kol1, .7]],
        Button["-1", px = px - 1;], Button["+1", px = px + 1;],
        Button["-10", px = px - 10;],
        Button["+10", px = px + 10;]}],
      Row[{InputField[Dynamic[If[py < 1, py = 1;, py]],
          FieldSize → {2, 1}, Enabled → False,
          Background → Lighter[kol2, .7]],
        Button["-1", py = py - 1;], Button["+1", py = py + 1;],
        Button["-10", py = py - 10;], Button["+10",
         py = py + 10;]}]}, Alignment → Center]},
  ItemSize → {13, 8}, Frame → True, Alignment →
   {Center, Center}, FrameStyle → Gray,
  Background → Lighter[Gray, .9]];
(*okno funkcji*)
opt01 = Column[{InputField[Dynamic[w],
```

```
    FieldSize → {20, 4}, Alignment → Center]},
  ItemSize → {22, 8}, Frame → True,
  Alignment → {Center, Center}, FrameStyle → Gray,
  Background → Lighter[Gray, .9]];
(*transformacja*)
opt11 = Column[{
   Column[{Text[Style["Transformation", Bold]],
     Slider[Dynamic[t]],
     Row[{Button["Before", t = 0;, ImageSize → 60],
       "  ", Button["After", t = 1;,
        ImageSize → 60]}]}, Alignment → Center]
  }, ItemSize → {18, 8}, Frame → True,
  Alignment → {Center, Center}, FrameStyle → Gray,
  Background → Lighter[Gray, .9]];
(*grid translation*)
opt14 =
 Column[{Column[{Text[Style["Grid: translation", Bold]],
     Row[{DynamicModule[{xx = 0}, Column[{InputField[
            Dynamic[z01], FieldSize → {5, 1.5}, Alignment ->
             Center, Background → Lighter[kol1, .7]],
           {Slider[Dynamic[xx], {-.3, .3, .1},
               ImageSize → 80], z01 = z01 + (xx =
                  xx - ControlActive[0, xx])}[[1]] // Dynamic
         }]], DynamicModule[{xx = 0}, Column[{InputField[
           Dynamic[z02], FieldSize → {5, 1.5}, Alignment ->
            Center, Background → Lighter[kol1, .7]],
          {Slider[Dynamic[xx], {-.3, .3, .1}, ImageSize →
                80], z02 = z02 + (xx = xx - ControlActive[
                     0, xx])}[[1]] // Dynamic}]]}]},
    Alignment → Center]}, ItemSize → {18, 8},
  Frame → True, Alignment →
   {Center, Center},
  FrameStyle → Gray,
```

```
  Background →
   Lighter[Gray, .9]];
(*grid rotation*)
opt15 =
 Column[{Column[{Text[Style["Grid: rotation", Bold]],
     Column[{InputField[Dynamic[ϕ],
         FieldSize → {5, 1.5}, Alignment -> Center,
         Background → Lighter[kol1, .7]],
       Row[{Slider[Dynamic[ϕ], {0, 2 π},
          ImageSize → 80]}]]}]
  }, ItemSize → {18, 8}, Frame → True,
  Alignment → {Center, Center}, FrameStyle → Gray,
  Background → Lighter[Gray, .9]];
(*punkty / układ wsp*)
opt16 = Column[{
   Row[{Dynamic[If[punkt == 0,
       Button["show points", punkt = 1;, ImageSize → 90],
       Button["hide points", punkt = 0;, ImageSize → 90]]],
     Dynamic[If[ax == 0, Button["show axes",
         ax = 1;, ImageSize → 90],
       Button["hide axes", ax = 0;, ImageSize → 90]]]}]},
  ItemSize → {18, 4}, Frame → True, Alignment →
   {Center, Center}, FrameStyle → Gray,
  Background → Lighter[Gray, .9]];
(*Area Size, Range of view, colors *)
opt17 = Column[{
   Grid[{
     {Text[Style["Range of view", Bold]],
      DynamicModule[{xx = 0},
       {Slider[Dynamic[xx], {-.3, .3, .1}, ImageSize → 80],
          o1 = Max[o1 + (xx = xx - ControlActive[0, xx]), 0]}[[
         1]] // Dynamic]},
     {Text[Style["Area size", Bold]],
```

```
      DynamicModule[{xx = 0},
       {Slider[Dynamic[xx], {-.3, .3, .1}, ImageSize → 80],
          o2 = Max[o2 + (xx = xx - ControlActive[0, xx]), 0]}[[
         1]] // Dynamic]}, {Row[{ColorSetter[Dynamic[kol1]],
        ColorSetter[Dynamic[kol2]]}]}}]
  }, ItemSize → {18, 4}, Frame → True,
  Alignment → {Center, Center}, FrameStyle → Gray,
  Background → Lighter[Gray, .9]];

(******** front programu ********)
(* okno programu *)
Grid[{{Grid[{{opt01, opt21, opt22}}, Frame → None,
    FrameStyle → {Gray}], SpanFromLeft}, {Column[{opt11,
     (*wybór siatki*)
     Column[{Text[Style["Grid style", Bold]],
       Column[{SetterBar[Dynamic[fig],
          {"rect" → " Rectangle ", "elips" →
            "  Ellipse  "}]}, Alignment → Center]},
      ItemSize → {18, 2.5}, Frame → True,
      Alignment → {Center, Center}, FrameStyle → Gray,
      Background → Lighter[Gray, .9]],
     (*opt12*)
     Dynamic[If[fig == "rect",
       (*ustawienia kwadratu*)
       Column[{Column[{Text[Style["dimensions", Bold]],
           Row[{DynamicModule[{xx = 0}, Column[
               {InputField[Dynamic[w1],
                  FieldSize → {5, 1.5}, Alignment -> Center,
                  Background → Lighter[kol1, .7]],
                {Slider[Dynamic[xx], {-.3, .3, .1},
                    ImageSize → 80], w1 =
                    w1 + (xx = xx - ControlActive[0, xx])}[[
```

```
            1]] // Dynamic}]], DynamicModule[
        {xx = 0}, Column[{InputField[Dynamic[w2],
           FieldSize → {5, 1.5}, Alignment -> Center,
           Background → Lighter[kol2, .7]],
          {Slider[Dynamic[xx], {-.3, .3, .1},
              ImageSize → 80], w2 = w2 + (xx =
                 xx - ControlActive[0, xx])}[[1]] //
           Dynamic}]]}]}, Alignment → Center]},
 ItemSize → {18, 4}, Frame → True, Alignment →
  {Center, Center}, FrameStyle → Gray,
 Background → Lighter[Gray, .9]],
(*ustawienia elipsy*)
Column[{Grid[{{Text[Style["dimensions", Bold]],
     SpanFromLeft}, {DynamicModule[{xx = 0},
      Column[{InputField[Dynamic[w1],
         FieldSize → {5, 1.5}, Alignment -> Center,
         Background → Lighter[kol1, .7]],
        {Slider[Dynamic[xx], {-.3, .3, .1},
            ImageSize → 80], w1 = Max[
             w1 + (xx = xx - ControlActive[0, xx]),
             0]}[[1]] // Dynamic}]],
     DynamicModule[{xx = 0}, Column[
       {InputField[Dynamic[w2],
         FieldSize → {5, 1.5}, Alignment -> Center,
         Background → Lighter[kol1, .7]],
        {Slider[Dynamic[xx], {-.3, .3, .1},
            ImageSize → 80], w2 = Max[
             w2 + (xx = xx - ControlActive[0, xx]),
             0]}[[1]] // Dynamic}]]},
    {Column[{InputField[Dynamic[θ],
        FieldSize → {5, 1.5}, Alignment -> Center,
        Background → Lighter[kol1, .7]],
       Row[{Slider[Dynamic[θ], {-2 π, 2 π},
```

```
                ImageSize → 80]}]}], DynamicModule[
            {xx = 0}, Column[{InputField[Dynamic[w3],
               FieldSize → {5, 1.5}, Alignment -> Center,
               Background → Lighter[kol1, .7]],
              {Slider[Dynamic[xx], {-.3, .3, .1},
                  ImageSize → 80], w3 = Max[
                   w3 + (xx = xx - ControlActive[0, xx]),
                   0]}[[1]] // Dynamic}]]}
        }]}, ItemSize → {18, 8}, Frame → True,
      Alignment → {Center, Center}, FrameStyle → Gray,
      Background → Lighter[Gray, .9]]]],
  opt14, opt15, opt16, opt17}, Frame → None,
 FrameStyle → {Gray}, Alignment → Center],
(*work.area*)
Dynamic[nowset := {w, fig, w1, w2, w3, θ, px,
   py, pxs, pys, ϕ, z01, z02, punkt, o1, o2, ax};
 If[fig == "rect",
  (*siatka prostokąta*)
  Column[{"", X = Table[x + ⅈ y, {x, -w1/2, w1/2, w1/(pxs px)}];
    XX = Table[X, {y, -w2/2, w2/2, w2/pys}];
    Y = Table[x + ⅈ y, {y, -w2/2, w2/2, w2/(pys py)}];
    YY = Table[Y, {x, -w1/2, w1/2, w1/pxs}];}],
  (*siatka elipsy*)
  Column[{"", X = Table[r (w1 Cos[α] + ⅈ w2 Sin[α]),
      {r, w3/w1, 1, (1 - w3/w1)/(pxs px)}];
```

```
XX = Table[X, {α, 0, θ, θ/pys}];
Y = Table[
  r (w1 Cos[α] + i w2 Sin[α]), {α, 0, θ, θ/(pys py)}];
YY = Table[Y, {r, w3 / w1, 1, (1 - w3 / w1)/pxs}];}]]],

Dynamic[Grid[
  {{Graphics[{{Opacity[0], Line[{{-o1, 0}, {o1, 0}}],
       Line[{{0, -o1}, {0, o1}}]},
      PointSize[Medium], {kol1, If[punkt == 1,
        Point /@ # & /@ (c2r /@ # & /@ trobf[XX,
            z01 + i z02, ϕ, (t w[#] + (1 - t) #) &]),
        {Opacity[0], Point[{{0, 0}}]}], linec /@
        trobf[XX, z01 + i z02, ϕ, (t w[#] + (1 - t) #) &]},
      {kol2, PointSize[Medium],
       If[punkt == 1, Point /@ # & /@ (c2r /@ # & /@ trobf[YY,
            z01 + i z02, ϕ, (t w[#] + (1 - t) #) &]),
        {Opacity[0], Point[{{0, 0}}]}]
       , linec /@ trobf[YY, z01 + i z02, ϕ,
         (t w[#] + (1 - t) #) &]}}, AspectRatio →
      Automatic, Axes → ax, ImageSize → Scaled[.9],
     AxesStyle → {Lighter[Gray], Opacity[.5]}]}},
   ItemSize → {{o2}}, Frame → True, FrameStyle → Gray]]},
Frame → None, Alignment → {Left, Top}]]
```

8. Dodatek B. Skróty klawiaturowe

Wygodnie jest posługiwać się skrótami klawiaturowymi. Poniżej przedstawione są niektóre z nich, znacznie ułatwiające pracę z *Mathematica*®.

CTRL + **S** - zapisz (zaleca się częste jego używanie)
CTRL + **C** - kopiuj
CTRL + **X** - wytnij
CTRL + **V** - wklej
Home / **End** - przejdź na koniec / początek linii

ALT + **]** , ALT + **}** , ALT + **)** - wprowadza odpowiednio nawiasy [ ] , { } , ( )
CTRL + **L** - wklej ostatni "input"
CTRL + SHIFT + **L** - wklej ostatni "output"
CTRL + **'** - zwija (rozwija) grupy komórek
F12 - widok pełnoekranowy
F1 - pomoc (program automatycznie wskaże temat zwiazany z elementem na którym aktualnie znajduje sie kursor)

Zaznaczenia
CTRL + **.** - rozszerza zasięg zaznaczenia
CTRL + **A** - zaznaczenie całego notatnika
kliknięcie na komórkę lub grupę komórek - zaznaczenie komórki lub grupy komórek
CTRL + SHIFT + **B** - zaznaczenie nawiasu (wraz z zawartością), w odrębie którego znajduje się kursor
CTRL + SHIFT + → , CTRL + SHIFT + ← - rozszerza zaznaczenie o słowo w odpowiednim kierunku

Znaki specjalne, elementy wyrażeń
ESC*nazwa_elementu*ESC- wstawia element o podanej nazwie (np. ESCpiESC wstawia liczbę π)

p	π	inf	∞	[[	⟦
ee	e	ii	i	]]	⟧
a	α	el	$\in$		

⚠ **Uwaga** : Początkującym użytkownikom radzimy korzystać z palety *Basic Math Assistant* z menu *Pallets*. Znajdują się tam między innymi znaki specjalne, symbole matematyczne, struktury używane w *Mathematica*® itp.

CTRL + ^ , CTRL + **6** - indeks górny
CTRL + _ , - indeks dolny
CTRL + / - kreska ułamkowa
CTRL + **2** - pierwiastek
CTRL + **5** - stopień pierwiastka
CTRL + **Spacja** - wyjście poza zakres indeksu, ułamka itp.
CTRL + **,** - dodaje kolumnę
CTRL + ENTER - dodaje wiersz
CTRL + **K** - wyświetlenie możliwych nazw wyrażeń (w trakcie pisania)

Bibliografia

[1] Л. Л. Голубева, А. Э. Малевич, Н. Л. Щеглова, *Компьютерная математика. Автоматизированное рабочее место Математика : курс лекций*, Минск : БГУ, 2005.

[2] W. T. Shaw, *Complex Analysis with Mathematica®*, Cambridge University Press, 2006.

[3] S. Mangano, *Mathematica Cookbook*, O'Reilly Media, 2010.

[4] M. Trott, *The Mathematica GuideBook for Symbolics*, Springer, 2005.

[5] M. Trott, *The Mathematica GuideBook for Numerics*, Springer, 2005.

[6] M. Trott, *The Mathematica GuideBook for Programming*, Springer, 2004.

[7] M. Trott, *The Mathematica GuideBook for Graphics, Springer*, 2004.

[8] R. Hazreat, *Mathematica®: A Problem-Centered Approach* (Springer Undergraduate Mathematics Series), Springer, 2010.

[9] W. Nawalaniec, V. V. Mityushev, *Application of conformal mappings to grid generation in finite elements methods*, Problemes of modern techniques in engineering and education 2010, Pedagogical University, Cracow, 73-78.

[10] R. Grzymkowski, A. Kapusta, T. Kuboszek, D. Słota, *Mathematica 6*, Wydawnictwo Pracowni Komputerowej Jacka Skalmierskiego, 2008.