Ćwiczenie 11. Mathematica: Manipulacja. Elementy programowania

11.1. Animacja wykresów

Program *Mathematica* daje ogromne możliwości animacji obiektów matematycznych, co znacznie rozszerza nasze możliwości poznawcze. Animacje mogą przekazać znacznie więcej informacji niż wykresy statyczne. *Mathematica* posiada wbudowane funkcje **Animate** i **ListAnimate**, które zapewniają natychmiastowy sposób konstruowania animacji, grafiki lub innego rodzaju ekspresji w dokumencie. Funkcję **Animate** możemy wywołać w następujący sposób:

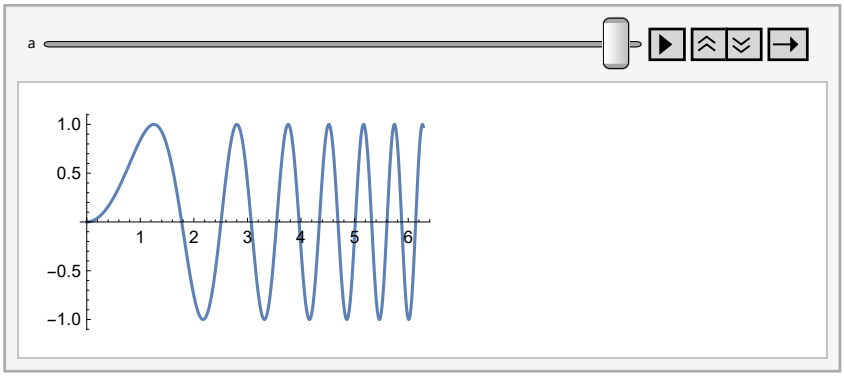
Animate[expr, {u, umin, umax}] – generuje animację w której wartość u zmienia się w sposób ciągły od wartości umin do umax.

Animate[expr, {u, umin, umax, du}] – generuje animację w której wartość u zmienia się w krokach o wartości du.
Zademonstrujemy możliwości animacyjne stosując następujące przykłady.

Zastosujemy funkcję **Animate** dla animacji wykresu funkcji $f(x) = \cos(x^2 - \alpha)$ w zależności od wartości parametru α:

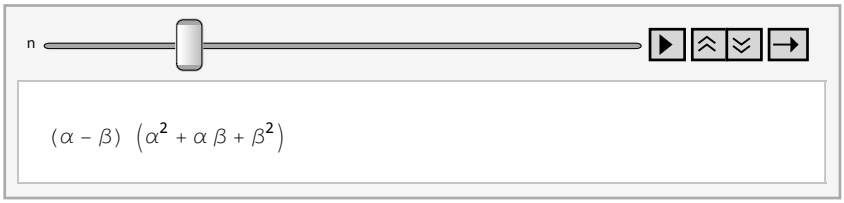
In[355]:=
```
Animate[Plot[Sin[x^2 + a], {x, 0, 2 π}, ImageSize → 200], {a, 0, 2 π}, AnimationRunning → False]
```
animuj w··· sinus rozmiar obrazu animacja aktywna fałsz

Out[355]=



Mathematica może „odzyskać" dowolne wyrażenie, nie tylko grafikę. W następnym przykładzie funkcja **Animate** stopniowo pokazuje rozkład wielomianu $\alpha^n - \beta^n$ na współczynniki. Opcja **DefaultDuration** określa długość animacji od początku do końca w ciągu kilku sekund

```
Animate[Factor[a^n - b^n], {n, 1, 10, 1}, DefaultDuration → 30]
```
animuj rozłóż na czynniki domyślny czas trwania

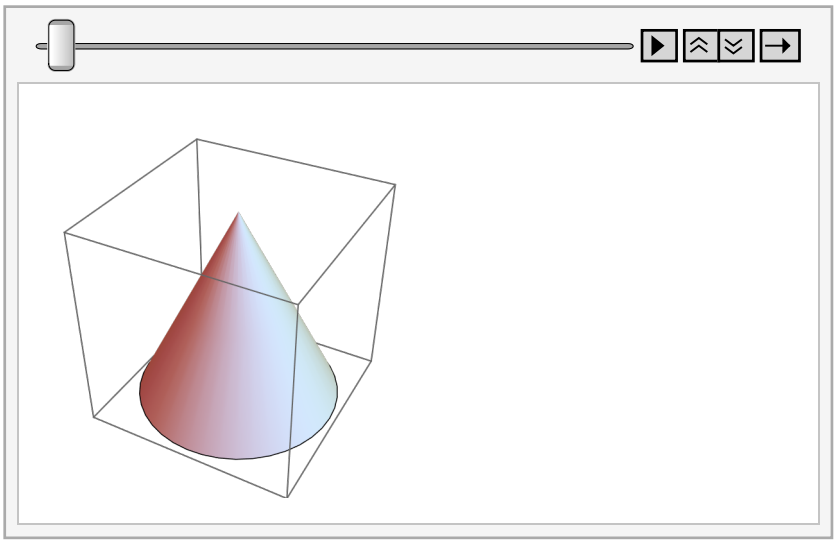


Zastosujemy funkcję **ListAnimate** dla animowania listy obiektów. Ustawienie wartości **False** dla opcji **AnimationRunning** uniemożliwia automatyczne uruchomienie animacji

In[360]:=
```
list = {Graphics3D[Cone[]], Graphics3D[Cuboid[]], PolyhedronData["Dodecahedron"], PolyhedronData["Icosahedron"]};
```
trójwymia··· stożek trójwymia··· prostopa·· dane o wielości··· dwunastościan dane o wielości··· dwudziestościan

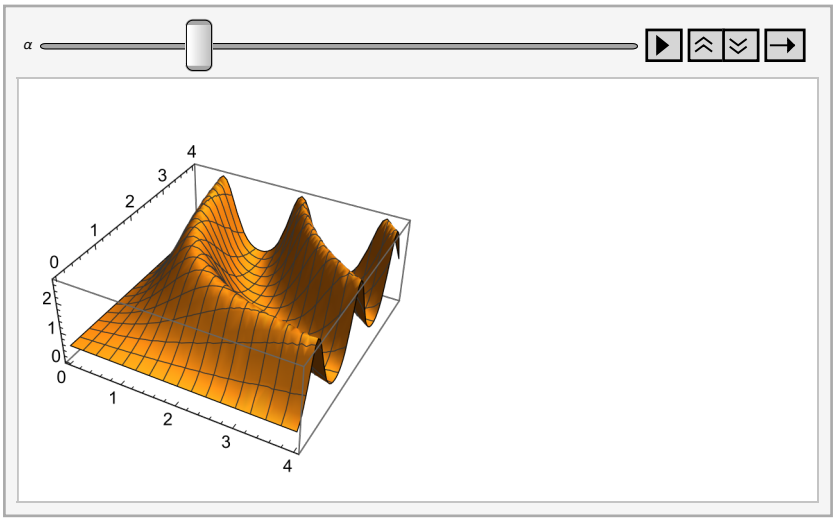
In[361]:=
```
ListAnimate[list, AnimationRunning → False, ImageSize → 200]
```
animuj liste animacja aktywna fałsz rozmiar obrazu

Out[361]=



Poniższy przykład ilustruje działanie instrukcji **Animate** w trójwymiarowej przestrzeni dla powierzchni określonej wzorem $z = e^{-\sin(x\,y+\alpha)}$ poprzez zmianę wartości parametru α z przedziału [0,10].

```
Animate[Plot3D[Exp[−Sin[x y + α]], {x, 0, 4}, {y, 0, 4}, ImageSize → 200], {α, 0, 10}, AnimationRunning → False]
```
animuj wykr··· fun··· sinus rozmiar obrazu animacja aktywna fałsz



Program *Mathematica* daje możliwość rozwiązywania skomplikowanych problemów matematyki i jej zastosowań, zarówno robiąc wizualizacje, jak i animacje rozwiązań. Dla wizualizacji rozwiązań stosowaliśmy podstawowe instrukcje **Plot**, **ParametricPlot**, **ContourPlot**, **ListPlot** na płaszczyźnie i **Plot3D**, **ParametricPlot3D** w trójwymiarowej przestrzeni. *Mathematica* pozwala nie tylko obserwować zmiany obiektów zależne od występujących w nich parametrach, ale również symulować eksperymenty zmieniające się w miarę upływu czasu. Podstawową procedurą wykorzystywaną do manipulacji i animacji jest procedura **Manipulate**.

```
Manipulate[expr, {u, u_min, u_max, krok}]
```
zmieniaj

Instrukcja ta daje możliwość utworzenia wersji wyrażenia *expr* zależnej od parametru *u* z przedziału $[u_{min}, u_{max}]$. W przypadku

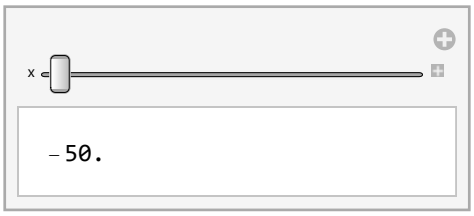
podania parametru *krok* parametr *u* zmienia się skokowo: $u_{\min}$, $u_{\min}+krok$, $u_{\min}+2krok$, ... , $u_{\min}+n\ krok$, gdzie $u_{\min}+(n+1)\ krok > u_{\max}$.

Mamy jeszcze inne możliwości wywołania tego polecenia:

1. **Manipulate** [*expr*, {*u*, umin, umax}] – generuje wersję wyrażenia *expr* z dodatkową kontrolą, która umożliwia manipulację interaktywnej wartości *u*;
2. **Manipulate**[*expr*,{{*u*, *uinit*}, umin, umax...}] – przyjmuję wartości początkowe *u* jako *uinit*;
3. **Manipulate**[*expr*,{{*u*, uinit, ulbl},...}] – dodatkowe etykiety kontroli dla *u*;
4. **Manipulate**[*expr*,{*u*,{u1, u2,...}}] – pozwala wartości *u* przyjąć wartości dyskretne u1,u2,… ;
5. **Manipulate**[*expr*,{*u*,...},{*v*,...},...] – zapewnia kontrolę do manipulowania każdą wartością *u*, *v*, ...

In[363]:=
```
Manipulate[5 x, {x, −10, 10}]
```
zmieniaj

Out[363]=



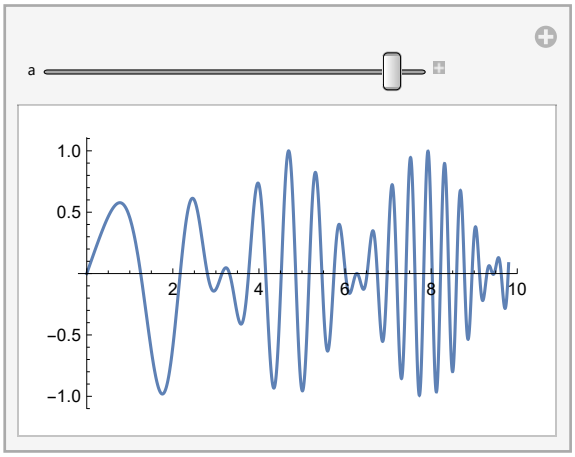
Kliknięcie przycisku [+] z prawej strony suwaka ujawni dodatkowe kontrole poniższe

Out[ ]=

Narysujemy wykres funkcji $f(x) = x^2 \cos\frac{1}{x^2}$, gdzie zmienna *x* należy do przedziału [0; *a*], parametr *a* zmienia się od $\frac{\pi}{8}$ do $\frac{\pi}{6}$.

In[375]:=
```
Manipulate[Plot[Sin[x] Cos[x^2], {x, 0, a}, ImageSize → 250], {a, 5, 10}]
```
zmieniaj w··· sinus cosinus rozmiar obrazu

Out[375]=



Instrukcja **Manipulate** ma podobne opcje jak instrukcja **Plot**. Poniżej zostały zaprezentowane wybrane opcje instrukcji **Manipulate**:

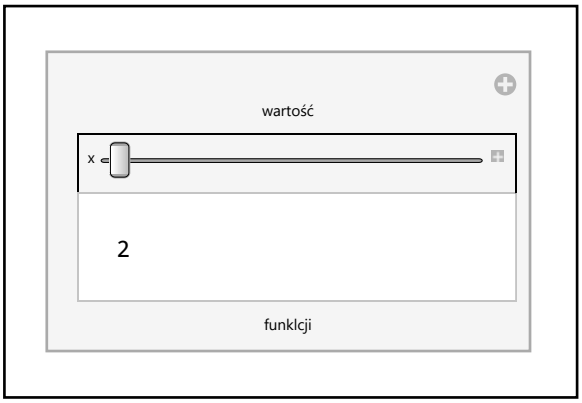
1. **Frame** rysuje ramkę wokół kontrolek; możliwe wartości – **False**, **True**;

2. **FrameMargins** margines wewnętrzny; przykłady wartości: **Automatic**, **Medium**, 12;
3. **FrameLabel** – etykieta po każdej stronie panelu; przykłady wartości: **None**,{„bottom”, „left”, „top”, „right”};
4. **RotateLabel** obracają etykiety w panelu; możliwe wartości – **True**, **False**.

Znajdziemy wartości funkcji $f(x) = 2\,\mathrm{tg}(3\,x)$, gdzie x zmienia się na przedziale $[\pi/12;\ \pi]$.

In[385]:=
```
Manipulate[2 Tan[3 x], {x, π/12, π}, Frame → True, FrameMargins → 15,
 zmieniaj  tangens  ramka  pr···  marginesy w ramce
  FrameLabel → {"funklcji", " ", "wartość"}, RotateLabel → True, ImageMargins → 16] // Framed
  etykieta ramki  obróć podpis  pr···  marginesy obrazu  z ramką
```

Out[385]=

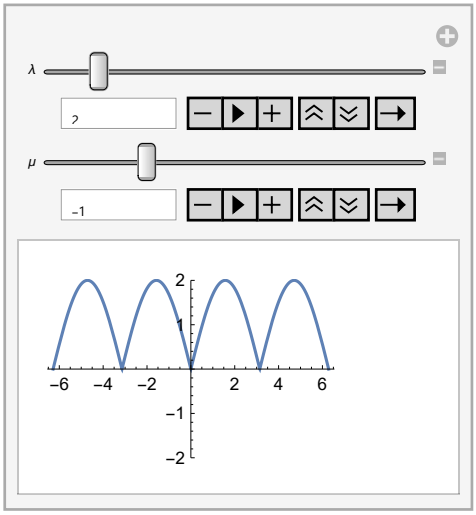


Sercem dynamiki jest możliwość interaktywnej zmiany wartości parametrów, żeby zobaczyć jaki wpływ ma to na wyrażenie. Mamy wiele typów kontrolek, którymi możemy zmieniać parametry. Podstawową kontrolką jest slider (suwak). Został on zaprezentowany na powyższych przykładach.

Do ciągłego zakresu wartości, funkcja **Manipulate** zawiera domyślnie ukryte elementy kontroli animacji. Do automatycznego wyświetlania trzeba ustawić wartość opcji **Appearance** jako „Open”. Możemy animować kilka parametrów jednocześnie (w tym przykładzie dwa)

In[388]:=
```
Manipulate[Plot[λ Abs[Sin[μ x]], {x, -2 π, 2 π}, PlotRange → 2, ImageSize → 150],
 zmieniaj  wyk···  w···  sinus  zakres wykresu  rozmiar obrazu
  {{λ, 2}, 1, 10, Appearance → "Open"}, {{μ, -1}, -2, 2, Appearance → "Open"}]
    wygląd  wygląd
```

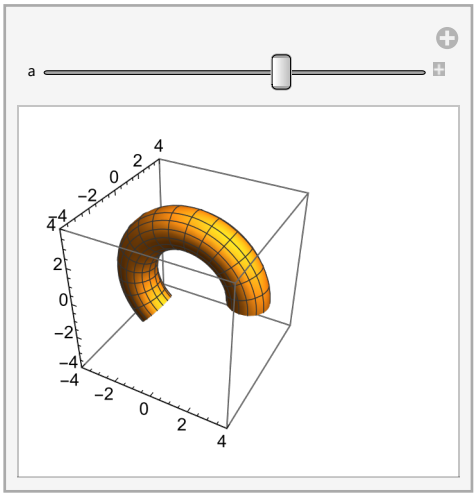
Out[388]=



Przykład manipulacji trójwymiarowych wykresów w przestrzeni.

In[391]:=
```
Manipulate[ParametricPlot3D[{Cos[u] (3 − Cos[v]), −Sin[v], Sin[u] (3 − Cos[v])},
 zmieniaj trójwymiarowy wy·· cosinus cosinus sinus sinus cosinus
 {u, 0, a}, {v, −π, π}, ImageSize → 150, PlotRange → 4], {{a, 4}, 0, 2 π}]
                                  rozmiar obrazu zakres wykresu
```

Out[391]=



11.2. Elementy programowania

Mathematica obsługuje wiele stylów programowania, takich jak proceduralny, funkcyjny, oparty na regułach i rekurencyjny. Ponadto, użycie prymitywów graficznych i dyrektyw prowadzi nas do programowania graficznego. *Mathematica* umożliwia nawet programowanie obiektowe. Programowanie funkcyjne i oparte na regułach stanowi sedno programowania w Mathematice. Przedstawmy pokrótce główne style programowania.
Programy napisane w tradycyjnych językach programowania, takich jak Fortran i C, nazywane są procedurami, i stąd pochodzi termin programowanie proceduralne. *Mathematica* oferuje również podobne polecenia, takie jak **For**, **While** i **If**, używane w programowaniu proceduralnym, dzięki czemu możemy programować w stylu proceduralnym.
W programowaniu funkcyjnym stosujemy funkcje do argumentów. Funkcje mogą być funkcjami wbudowanymi lub funkcjami zdefiniowanymi przez nas i mogą być stosowane w sposób zagnieżdżony. Funkcje są stosowane do argumentów za pomocą specjalnych, zaawansowanych poleceń, takich jak **Map**, **Apply**, **Nest**, **FixedPoint** i **Fold**. Programowanie funkcyjne jest skuteczne zwłaszcza w przypadku manipulacji listami i obliczeń iteracyjnych.

Programowanie oparte na regułach wykorzystuje reguły i wzorce. Definicja funkcji f**[x_] := expr** jest przykładem reguły globalnej: za każdym razem, gdy napotkana zostanie funkcja **f** z określonym argumentem, na przykład **a**, ta reguła zastępuje **f**[**a**] wartością **expr**, gdzie **x** jest zastępowane przez **a**. W **expr/. x → a** stosujemy regułę lokalną: za każdym razem, gdy napotkana zostanie **x** w **expr**, zastąp je przez **a**. Argument **x_** funkcji jest przykładem wzorca. Wzorzec **x_** jest bardzo ogólny i w rzeczywistości pasuje do niego wszystko; nazwa wzorca to **x**.
Możemy tworzyć bardziej restrykcyjne wzorce na wiele sposobów. Ogólnie rzecz biorąc, w programowaniu opartym na regułach podajemy kilka reguł dla tej samej funkcji. Reguły te obejmują kilka różnych sytuacji lub kilka wzorców argumentów.
Programowanie rekurencyjne naturalnie powstaje poprzez programowanie rekurencyjnych formuł matematycznych: program wywołuje sam siebie z innym argumentem. Wiele zadań związanych z manipulacją listami można również zaprogramować rekurencyjnie: lista jest transformowana, dopóki nie przestanie się zmieniać.
Zanim jednak omówimy programowanie proceduralne, funkcjonalne, oparte na regułach i rekurencyjne, najpierw przedstawimy przykłady prostych programów, w których nie potrzebujemy żadnych specjalnych poleceń programistycznych, ale używamy znanych poleceń, takich jak **Table**, **N**, **/.** i wielu poleceń manipulacji listami. Pamiętajmy również, że dzięki **Manipulate** i **Dynamic** możemy tworzyć aplikacje interaktywne.
Programowanie funkcjonalne.
W przeciwieństwie do języków takich jak BASIC i C, polecenia Mathematica mogą operować nie tylko na określonych typach liczb i struktur danych, ale również na dowolnych wyrażeniach. W szczególności argumentem jednej funkcji może być inna funkcja. To zapewnia potężny i elegancki paradygmat programowania. Chociaż nie będziemy prowadzić filozoficznej dyskusji na temat zalet i natury programowania funkcyjnego, wystarczy powiedzieć, że jeśli przeczytałeś i śledziłeś ten rozdział do tego momentu, już to robisz. **Map** i **Function** to twój punkt wejścia. Polecenia takie jak **Apply**, **Thread** i **MapThread** poszerzą

twoje horyzonty, a **Nest**, **Fold**, reguły zamiany i dopasowywanie wzorców przeniosą Cię na wyższy poziom.
Przydatnym poleceniem podobnym do **Map** jest **Apply**. Podobnie jak **Map**, **Apply** przyjmuje dwa argumenty: funkcję i wyrażenie (często listę). Wynik to drugi argument, którego nagłówek zastąpiono funkcją z pierwszego argumentu. To wszystko; **Apply** usunie nagłówek z dowolnego wyrażenia i zastąpi go czymś innym. Na przykład, **Apply**[**Plus**, **List**[a,b,c]] zwróci **Plus**[a,b,c]:

In[404]:=
```
Apply[Plus, List[a, b, c]]
```
zasto··· suma lista

Out[404]=
```
a + b + c
```

Polecenie **Apply** można przekazać w formie infiksowej za pomocą @@.

In[403]:=
```
Plus @@ {a, b, c}
```
suma

Out[403]=
```
a + b + c
```

Polecenie **Thread** może być użyte do „wątku" funkcji obejmującego kilka list. W poniższym przykładzie, (niezdefiniowana) funkcja nazywa się **f**. Jest ona wywoływana z dwoma argumentami, z których każdy jest listą. Opakowanie tego wyrażenia w **Thread** powoduje, że f jest wywoływane dla odpowiednich elementów obu list, a wyjściem jest lista wyników:

In[405]:=
```
Thread[f[{a, b, c}, {1, 2, 3}]]
```
nawlecz

Out[405]=
```
{f[a, 1], f[b, 2], f[c, 3]}
```

Oto dwie aplikacje. Pierwsza pokazuje, jak użyć **Thread** do programowego utworzenia listy reguł z listy wartości lewostronnych i listy wartości prawostronnych. W tym przypadku polecenie **Rule** pełni rolę funkcji **f** powyżej. Drugi przykład ilustruje, że **Thread** działa tak samo jak **Transpose** na liście **list** (tj. na macierzy). W tym przypadku **List** pełni rolę funkcji **f** powyżej.

In[406]:=
```
Thread[{a, b, c} → {1, 2, 3}]
```
nawlecz

Out[406]=
```
{a → 1, b → 2, c → 3}
```

In[407]:=
```
Thread[{a, b, c}, {1, 2, 3}]
```
nawlecz

Out[407]=
```
{a, b, c}
```

Polecenie **MapThread** łączy funkcjonalność **Map** i **Thread**. Poniższy przykład ilustruje typową implementację. Zasadniczo działa ono tak samo, jak Thread, ale ma dwa argumenty, przy czym funkcja (w tym przypadku **f**) jest podana osobno jako pierwszy argument.

In[408]:=
```
MapThread[f, {{a, b, c}, {1, 2, 3}}]
```
zastosuj w wątku

Out[408]=
```
{f[a, 1], f[b, 2], f[c, 3]}
```

11.3. Iterations: Nest and Fold

In[409]:=
```
Clear[f, x];
```
wyczyść

```
NestList[f, x, 3]
```
lista wyników iteracji

Out[410]=
```
{x, f[x], f[f[x]], f[f[f[x]]]}
```

Polecenie **NestList** to podstawowe narzędzie do iterowania funkcji. Po wpisaniu **NestList**[polecenie, start, n] tworzona jest lista o długości $n + 1$, której pierwszym wpisem jest start. Następnie wyświetlany jest wynik zastosowania polecenia do startu, a następnie wynik zastosowania polecenia do tego wyniku i tak dalej, aż do n zastosowań polecenia. Polecenie **Nest** jest podobne i

wyświetla tylko ostatni element na tej liście.

In[411]:=
```
NestList[f, x, 3]
```
lista wyników iteracji

Out[411]= {x, f[x], f[f[x]], f[f[f[x]]]}

Polecenie **NestList** to podstawowe narzędzie do iterowania funkcji. Po wpisaniu **NestList**[polecenie, start, n] tworzona jest lista o długości $n+1$, której pierwszym wpisem jest start. Następnie wyświetlany jest wynik zastosowania polecenia do startu, a następnie wynik zastosowania polecenia do tego wyniku i tak dalej, aż do n zastosowań polecenia. Polecenie **Nest** jest podobne i wyświetla tylko ostatni element na tej liście. **Nest** to podstawowe polecenie iteracyjne, które wykonuje iterację $x_{i+1} = f(x_i)$ ustaloną liczbę razy:

In[412]:=
```
Nest[f, x0, 3]
```
iteruj

Out[412]= f[f[f[x0]]]

FixedPoint wykonuje iterację, dopóki wynik się nie zmieni (można podać kryterium zatrzymania). **Fold** również iteruje funkcję, ale teraz funkcja ma dwie zmienne i w każdej iteracji pobiera jeden element z danej listy jako drugi argument. Mamy również polecenia **NestList**, **FixedPointList** i **FoldList**, które również drukują wszystkie kroki pośrednie.
Na przykład metodę Newtona można zapisać następująco:

In[413]:=
```
newton[f_, x_, x0_, max_, opts___] := With[{df = D[f, x]}, FixedPointList[(x - f / df) /. x → # &, N[x0], max, opts]]
```
przy oblicz ⋯ ścieżka do punktu stałego przybliżenie numeryc

Oto jego zastosowanie:

In[414]:=
```
newton[3 x^3 - E^x, x, 2, 20, SameTest → (Abs[#1 - #2] < 10^-6 &)]
```
liczba Eulera kryterium r⋯ wartość bezwzględna

Out[414]= {2., 1.41942, 1.1019, 0.975117, 0.953089, 0.952446, 0.952446}

Pierwszą zaletą poleceń iteracji funkcji jest to, że skracają one kod w porównaniu z programowaniem proceduralnym.

Zadania

Zadanie 1. Utwórz tabliczkę mnożenia 10×10 (Wskazówka: użyj polecenia **Grid** i **Table**).

Zadanie 2. Utwórz siatkę 8×8 o losowych kolorach.

Zadanie 3. Wizualizuj list {1, 4, 3, 5, 2} za pomocą wykresu kołowego, osi liczbowej, wykresu liniowego i wykresu słupkowego. Umieść je w siatce 2x2.

Zadanie 4. Stwórz manipulację składającą się z 1 do 20 koncentrycznych okręgów.

Zadanie 5. Utwórz obiekt manipulacyjny, w którym t zmienia się pomiędzy −2 i +2, i który zawiera okręgi o promieniu x i środku w punkcie {t*x, 0}, przy czym x zmienia się od 1 do 10.

Zadanie 6. Użyj funkcji Range i czystej funkcji, aby utworzyć listę pierwszych 20 kwadratów.

Zadanie 7. Stwórz listę wyników mieszania żółtego, zielonego i niebieskiego z czerwonym.

Zadanie 8. Zacznij od x, następnie utwórz listę, zagnieżdżając ramkę (Framed) do 10 razy, za każdym razem używając losowego koloru tła (Wskazówka: użyj polecenie **NestList**).

Zadanie 9. Utwórz listę pierwszych 10 potęg liczby 3 (zaczynając od 0) za pomocą **NestList**.

Zadanie 10. Użyj **Prime** i **Array**, aby wygenerować listę pierwszych 100 liczb pierwszych.