Sveučilište u Zagrebu
Prirodoslovno−matematički fakultet
Fizički odsjek

Krešimir Kumerički

Simboličko programiranje (*Mathematica*)

Skripte

Zagreb, 2006.

■ Predgovor

Ovo su skripte za kolegij *Simboličko programiranje* kojeg držim na Fizičkom odsjeku Prirodoslovno−matematičkog fakulteta u Zagrebu studentima smjera "profesor fizike i informatike". Iako su zbog toga neki primjeri inspirirani zadacima iz fizike, vjerujem da skripte mogu poslužiti i ne−fizičarima kao uvod u *Mathematicu* i simboličko programiranje. Elektronska verzija, u obliku *Mathematica* bilježnica, dostupna je na adresi http://www.phy.hr/~kkumer/mma/ .

Simboličko programiranje je postalo moguće razvojem modernih sustava za računalnu algebru (Axiom, Form, Macsyma, Maple, Mathematica, MuPAD, Maxima, Reduce, ...) koji su se u međuvremenu nametnuli kao neizostavan alat fizičara, matematičara i mnogih drugih. Za ovaj kolegij *Mathematica*® (komercijalni proizvod tvrtke *Wolfram Research Inc.)* je odabrana zbog svoje rasprostranjenosti, postojanja licence na Sveučilištu, te relativno pristupačne studentske verzije programa. Svejedno, većinu sadržanog gradiva moguće je obraditi i na drugim sustavima za računalnu algebru. (Od besplatno dostupnih sustava, zgodan je MuPAD, http://www.mupad.com/.) Za detaljnu usporedbu mogućnosti raznih sustava dobro je konzultirati rad Simon and Wester, *A Critique of the Mathematical Abilities of CA Systems*, http://www.math.unm.edu/%7Ewester/cas_review.html

Mislim da nema puno smisla u jednom ovakvom kursu predstavljati sve raspoložive matematičke funkcije koje *Mathematica* sadrži jer sadrži naprosto sve ono što student susreće i sve je zadovoljavajuće dokumentirano. Tako sam se, naročito u kasnijim poglavljima, usredotočio na razlike i prednosti koje rad s ovakvim sustavom pruža u odnosu na klasično programiranje u standardnim računalnim jezicima.

Krešimir Kumerički
U Regensburgu, 27. veljače 2006.

2006−02−27 −− Version 1−S

Sadržaj

Predgovor 3
Sadrž aj 5
1. Osnove rada s *Mathematicom* 7
1.1 Osnove grafičkog sučelja 7
Mathematica bilježnice i čelije 7
Rad s čelijama i izvršavanje komandi 7
Spremanje i predaja bilježnica 8
1.2 Elementarno računanje 8
Mathematica kao kalkulator 8
Ugrađene funkcije 10
"Help" sustav 10
Poruke o grešci 11
2. Elementarna i viša matematika na *Mathematici* 13
2.1 Algebarske manipulacije 13
2.2 Jednadžbe 14
2.3 Matematička analiza 18
2.4 Linearna algebra 19
2.5 Crtanje grafova 21
2D 21
3D 23
2.6 Grafički prikaz i fitanje podataka 25
2.7 Diferencijalne jednadžbe 27
3. Osnove *Mathematica* jezika 33
3.1 Liste 33
Liste i elementi 33
Konstruiranje lista 35
Liste kao skupovi 36
Listabilnost funkcija 36
3.2 Funkcije 37
Vrste pridruživanja 37
Definiranje funkcija 38
3.3 Izrazi (*expressions*) 42
Sve su samo izrazi 42

Dijelovi izraza 44
Nivoi izraza (*) 45
3.4 Uzorci (*patterns*) 46
3.5 Transformacijska pravila 47
4. Programiranje u *Mathematici* 51
Uvod 51
4.1 Proceduralno programiranje 52
If−then grananje 52
Petlje 53
Moduli 54
4.2 Funkcionalno programiranje 56
`Map i Apply` 56
`Nest i Fold` 57
Rekurzivno definiranje funkcije (*) 60
Primjeri razvoja funkcionalnih programa 60
4.3 Programiranje transformacijskim pravilima 64
Globalna i lokalna pravila 64
Uzorci koji se ponavljaju 65
Predikati 67
Primjeri 67
5. Fizika i *Mathematica* 73
5.1 Klasična mehanika 73
5.2 Elektrodinamika 73
Gibanje nabijene čestice u homogenom magnetskom polju 73
5.3 Kvantna fizika 74
Bohrov model atoma 74
6. *Mathematica* i "vanjski svijet" 75
6.1 Izvršavanje *Mathematica* programa u pozadini 75
6.2 Pokretanje kernela na drugom računalu 75
6.3 Povezivanje *Mathematice* s programima u C−u 76

1. Osnove rada s *Mathematicom*

■ 1.1 Osnove grafičkog sučelja

■ *Mathematica* bilježnice i ćelije

Mathematica bilježnica (*notebook*) je datoteka (uobičajena ekstenzija imena datoteke je `.nb`) koja se sastoji od niza ćelija (*cell*). Ćelije sadrže raznovrsne tipove informacija (matematičke izraze, grafiku, tekst) koje je ili unio korisnik ili ih je proizvela *Mathematica* kao rezultat nekog računa. Svaka ćelija je označena uglatom zagradom na desnoj strani bilježnice. Tri vrste ćelija koje ćemo najčešće susretati su:

1. *Tekstualna* ćelija, koja obično sadrži komentare računa (npr. ovaj tekst)

2. *Input* ćelija, koja sadrži izvršne *Mathematica* komande

3. *Output* ćelija, koja sadrži rezultate izvršavanja tih komandi

Najlakši način razlikovanja ćelija je pomoću tipa slova (fonta). Evo primjera jedne input i jedne output ćelije:

```
2 + 2
```

```
4
```

■ Rad s ćelijama i izvršavanje komandi

Novu ćeliju stvorimo tako da na željeno mjesto ili kliknemo mišem ili dođemo kursorskim tipkama tako da se prikaže horizontalna linija. Počnemo tipkati i automatski se kreira *input* ćelija. Želimo li *tekstualnu* moramo konvertirati tip ćelije tako da istu označimo klikom na njenu zagradu (koja pocrni) i onda u izborniku na vrhu bilježnice biramo `Format` → `Style` → `Text`.

Izvršenje svega onog što smo upisali u input ćeliju zatražimo pritiskom na `ctrl-Enter`. (Česta je početnička greška zaboravljanje pritiskanja `ctrl` tipke. Samo `Enter` nas stavlja u novi red iste ćelije.)

Važno svojstvo *Mathematica* bilježnice je da svaku ćeliju bilježnice možemo i naknadno editirati i onda ponovno izvršiti tako da pritisnemo `ctrl-Enter` dok je kursor bilo gdje (!) u toj ćeliji (ili je ona označena klikom na njenu zagradu). To rad u *Mathematici* čini sličnijim radu u tabličnom kalkulatoru (*spreadsheet*) nego standardnom programiranju.

Zadatak: Editiranjem gornje "**2+2**" ćelije izračunajte 2+3.

Ćelije se mogu grupirati tako da iznad njih stavimo naslovnu ćeliju tipa *(sub)title* ili *sub(section)* putem izbornika `Format` → `Style`. Grupa ćelija može biti *otvorena* (sve se vidi) ili *zatvorena* (vidi se samo naslovna ćelija grupe). Ćelije se mogu otvarati ili zatvarati dvostrukim klikom na uglatu zagradu koja obuhvaća grupu. To često doprinosi preglednosti rada.

Zadatak: Zatvorite i otvorite ovu grupu ćelija (s naslovom "*Rad s ćelijama i izvršavanje komandi*")

Brisanje ćelije se izvodi njenim označavanjem (klik na zagradu) pa tipka `Del` ili izbornik `Edit->Clear`

■ Spremanje i predaja bilježnica

Bilježnicu koju je priredio nastavnik najčešće nije poželjno mijenjati. Stoga je zgodno svaku bilježnicu prije rada spremiti pod novim imenom putem izbornika `File→Save as ...`. Radi lakšeg snalaženja datoteke s domaćim zadaćama koje se predaju nastavniku treba uniformno nazivati i to ovako:

<*prezime bez hrvatskih dijakritičkih znakova*>‾<*broj domaće zadaće* >.nb

Primjer: `Kostelic-01.nb`. Za slanje emailom putem modema velike bilježnice se mogu komprimirati (zip ili rar). Kako je bilježnica ASCII tekstualna datoteka, kompresija je često znatna.

■ 1.2 Elementarno računanje

■ *Mathematica* kao kalkulator

Mathematica se može koristiti kao obični kalkulator proizvoljne preciznosti

```
3 * 2 + 1
```

```
7
```

```
Sqrt[9]
```

```
3
```

```
13^23
```

```
417539054134131163670457 97
```

```
22!
```

```
1124000727777607680000
```

Množenje je, osim zvjezdicom *****, moguće iskazati i razmakom što je ponekad preglednije

```
3.2 10^4
```

```
32000.
```

Ova gore, uobičajena notacija za zapis velikih brojeva ima i alternativni zapis

```
3.2*^4
```

```
32000.
```

Fortranski zapis 3.2E4 *nije* dopušten.

Rad u *Mathematica* bilježnici omogućuje i ljepši zapis standardnih matematičkih operacija i simbola. Tako se npr. potenciranje može izvesti na slijedeće načine:

1. "Jednodimenzionalno"

```
2^3
```

```
8
```

2. "Dvodimenzionalno", korištenjem `ctrl-^` tipke:

```
2^3
```

```
8
```

3. "Dvodimenzionalno", korištenjem palete `File→Palletes→BasicInput` za unos, gdje je moguće bilo odmah kliknuti na prvu tipku na paleti, unijeti "2", kliknuti tipku Tab, pa unijeti "3" ili, alternativno, unijeti "2", označiti to mišem pa tek onda kliknuti na paletu i unijeti "3".

```
2^3
```

```
8
```

Palete su zgodne kad je korisnik početnik ili za rijetko korištene funkcije, ali je njihovo korištenje sporo u odnosu na unos samo preko tipkovnice. Za lakše učenje zgodno je primijetiti da kad se kursor nalazi iznad neke tipke na paleti, na dnu palete je prikazano kako izvesti ekivalentan unos putem tipkovnice.

Zadatak: Izračunajte $\sqrt{2\sqrt{e^{\pi}}}$. Konstante π i e se također mogu naći na paleti `BasicInput`.

Primijetite da nismo dobili numerički već simbolički rezultat. *Mathematica* će se uvijek odlučiti za egzatni simbolički rezultat ako takav postoji. Ako nas ipak zanima numeričko približenje možemo koristiti funkciju **N** :

```
N[%]
```

```
8.
```

Ovdje smo također iskoristili vrlo korisni simbol **%**, koji uvijek znači "rezultat izvršenja posljednje komande", gdje treba naglasiti da se to odnosi na posljednje izvršenje u trenutnoj sesiji, a ne na rezultat koji stoji neposredno iznad u bilježnici. Naime, kako je input čelije dozvoljeno izvršavati proizvoljnim redom te dvije stvari se mogu razlikovati. Zbog toga simbol **%** treba koristiti pažljivo.

%	zamjenjuje posljednji rezultat
%%	zamjenjuje pretposljednji rezultat
% n	zamjenjuje n – ti rezultat trenutne*Mathematica* sesije tj. **Out[n]**

Funkcija **N** ima i inačicu **N[..., n]** koja daje numeričko približenje s *n* točnih decimala

```
N[%%, 100]
```

```
8.00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
 000000000000000000
```

Kako se u praksi funkcija **N** često rabi, dobro je znati da se ona može primijeniti i tzv. *postfiksno* :

```
e // N
```

```
2.71828
```

Zadatak: Ispišite prvih 500 decimala broja π.

S kompleksnim brojevima radimo jednostavno. Samo treba imati na umu da je kompleksna jedinica $\left(\sqrt{-1}\right)$ reprezentirana velikim slovom "**I**" ili specijalnim simbolom $\imath$ kojeg unosimo s palete `BasicInput` ili s tipkovnice sekvencom `Esc-i-i-Esc.`

```
(2 + 3 i)^4
```

```
-119 - 120 i
```

Zadatak: Izračunajte $e^{i\pi}$.

■ Ugrađene funkcije

U *Mathematicu* su ugrađene ("*built−in*") i brojne poznate matematičke funkcije i operacije.

```
Log[Sin[π]]
```

```
-∞
```

Za upotrebu svih tih funkcija važno je imati na umu dvije stvari:

1. Ime funkcije počinje velikim početnim slovom i često je slično uobičajenom matematičkom imenu funkcije. (Ipak, treba paziti. Npr. **Log** je prirodni logaritam (ln = $\log_e$), a dekadski logaritam ($\log_{10}$) se dobije kao **Log[10, <arg>]**

2. Argument funkcije je dan u uglatim zagradama. Upotreba zagrada u *Mathematici* je vrlo stroga:

( )	grupiranje elemenata u izrazima poput **2 (3 + 2) - 1**
[ ]	ograđuju argumente funkcija **Log[2]**
{ }	liste **{1, 3, a, 9}** i listama srodni objekti poput vektora
[[ ]] ili ⟦ ⟧	specificiranje elemenata liste **{a, b}⟦2⟧**
(* *)	komentar

Kao primjer ugrađene funkcije uzmimo Besselovu funkcija $J_2(z)$ u točki z=0

```
BesselJ[2, 0]  (* Ovo je samo demonstracija komentara *)
```

```
0
```

Postavlja se pitanje: Recimo da Vam zatrebaju Besselove ili kakve druge konkretne matematičke funkcije. Kako saznati njihov naziv i sintaksu u *Mathematici*? Za to postoji razrađeni "Help" sustav.

■ "Help" sustav

Help sustav nudi nekoliko raznih pristupa:

1. `Help→Help Browser...` sadrži cijelu debelu *Mathematica* knjigu i još više toga. (Usput, plava imena funkcija u ovom tekstu vode klikom miša na odgovarajuću stranicu "Help" sustava.)

2. Ponekad je brže zatražiti pomoć izravno iz input čelije, koristeći simbol "**?**" i wildcard "*****". Nakon toga jedan klik daje osnovnu informaciju o funkciji, a drugi nas vodi na odgovarajući dio *Mathematica* knjige.

```
?Bess*
```

System‘

BesselI BesselJ BesselK BesselY

3. Ako smo funkciju već koristili i samo nam treba podsjetnik kako se točno zove zgodno je započeti upis funkcije: `Bess...` i onda stisnuti `ctrl−K`. To nam omogućuje izravno nadopunjavanje naziva klikom miša i pogodno je i kod

funkcija kojima znamo ime, ali želimo uštedjeti na tipkanju. Nadalje, nakon upisa ili izbora funkcije i s kursorom odmah iza njenog imena pritisak na `ctrl-shift-K` daje predložak za upis argumenata funkcije.

```
BesselJ[n, z]
```

```
BesselJ[n, z]
```

Argumenti su označeni simbolima koji olakšavaju prepoznavanje njihovog ispravnog redoslijeda. Ovdje **n** sugerira prirodni, a **z** kompleksni broj, po čemu znamo da je na prvom mjestu red, a na drugom argument funkcije $J_n(z)$.

Zadatak: Pronađite odgovarajuću *Mathematica* funkciju i izvrijednite pridruženi Laguerrov polinom $L_3^{\,2}(1)$.

Domaći zadatak 1–1: Proučite upotrebu funkcije **Sum** za zbrajanje matematičkih redova i izračunajte:
(a) **1 + 3 + 5 + 7 + ⋯ + 61**
(b) $\sum_1^{\infty} \frac{1}{n^2} = 1 + \frac{1}{4} + \frac{1}{9} + \cdots$

Poruke o grešci

Ako se ogriješimo o matematička ili sintaktička pravila, *Mathematica* će nam uzvratiti porukom o grešci

```
1
0
```

```
Power::infy : Infinite expression 1/0 encountered. More...
```

```
ComplexInfinity
```

```
sin[2.3]
```

```
General::spell1 :
 Possible spelling error: new symbol name "sin" is similar to existing symbol "Sin". More...
```

```
sin[2.3]
```

Ovo drugo strogo gledano nije greška (jer možda mislimo na neku funkciju koja se zove *sin(x)*, a koja nema veze sa sinusom), ali *Mathematica* nas upozorava da smo možda htjeli sinus koji, kao i sve ugrađene funkcije treba pisati s velikim početnim slovom.

2. Elementarna i viša matematika na *Mathematici*

■ 2.1 Algebarske manipulacije

Moć paketa za simboličku matematiku, poput *Mathematice*, leži u sposobnosti manipulacije simboličkim izrazima. Kao prvo, svakom izrazu možemo radi lakšeg snalaženja pridijeliti ime.

```
izraz1 = (b + a)^3
```

```
(a + b)^3
```

Kod davanja imena mogu se upotrebljavati i grčka slova, indeksi itd. kako bi se postigla sličnost sa standardnom matematičkom notacijom $(\beta_1, \hat{\mathcal{F}}, \ldots)$, ali takva imena su čest izvor grešaka i ne preporučaju se početnicima.

Mathematica ne provodi skoro nikakve operacije na izrazima dok je eksplicitno ne instruiramo. Recimo da želimo raspisati ("ekspandirati") **izraz1** koristeći binomni teorem

```
Expand[izraz1]
```

```
a^3 + 3 a^2 b + 3 a b^2 + b^3
```

To bi mogli i na ruke. Međutim:

```
izraz2 = (a + 2 b + 3 c)^3 (x + y)^3
```

```
(a + 2 b + 3 c)^3 (x + y)^3
```

```
Expand[izraz2]
```

```
a^3 x^3 + 6 a^2 b x^3 + 12 a b^2 x^3 + 8 b^3 x^3 + 9 a^2 c x^3 + 36 a b c x^3 + 36 b^2 c x^3 +
 27 a c^2 x^3 + 54 b c^2 x^3 + 27 c^3 x^3 + 3 a^3 x^2 y + 18 a^2 b x^2 y + 36 a b^2 x^2 y + 24 b^3 x^2 y +
 27 a^2 c x^2 y + 108 a b c x^2 y + 108 b^2 c x^2 y + 81 a c^2 x^2 y + 162 b c^2 x^2 y + 81 c^3 x^2 y +
 3 a^3 x y^2 + 18 a^2 b x y^2 + 36 a b^2 x y^2 + 24 b^3 x y^2 + 27 a^2 c x y^2 + 108 a b c x y^2 +
 108 b^2 c x y^2 + 81 a c^2 x y^2 + 162 b c^2 x y^2 + 81 c^3 x y^2 + a^3 y^3 + 6 a^2 b y^3 + 12 a b^2 y^3 +
 8 b^3 y^3 + 9 a^2 c y^3 + 36 a b c y^3 + 36 b^2 c y^3 + 27 a c^2 y^3 + 54 b c^2 y^3 + 27 c^3 y^3
```

Ukoliko ne želimo vidjeti ispis rezultata neke komande, stavimo na kraj "**;**".

```
izraz3 = Expand[izraz2^3];
```

```
Length[izraz3]
```

```
550
```

Komanda **Short** ispisuje samo početak i kraj izraza te naznačuje broj ispuštenih članova pa tako možemo dobiti neku ideju o rezultatu bez da zakrčimo ekran:

```
Short[izraz3]
```

```
a^9 x^9 + 18 a^8 b x^9 + <<547>> + 19683 c^9 y^9
```

Inverzna operacija ekspanziji je faktorizacija:

```
Factor[izraz3]
```

```
(a + 2 b + 3 c)^9 (x + y)^9
```

Ovo faktorizira cijeli izraz u najmanji broj faktora. Faktoriziranje konkretne varijable tj. prikazivanje izraza u obliku polinoma u toj varijabli:

```
Collect[izraz2, y]
```

$$(a + 2\,b + 3\,c)^3\,x^3 + 3\,(a + 2\,b + 3\,c)^3\,x^2\,y + 3\,(a + 2\,b + 3\,c)^3\,x\,y^2 + (a + 2\,b + 3\,c)^3\,y^3$$

Traženje koeficijenta uz određeni izraz:

```
Coefficient[izraz2, y^2]
```

$$3\,(a + 2\,b + 3\,c)^3\,x$$

Stavljanje na zajednički nazivnik:

```
Together[a/(1 - a) + a/(1 + a)]
```

$$-\frac{2\,a}{(-1 + a)\,(1 + a)}$$

Jedna od najčešće korištenih funkcija je **Simplify** koja obično daje najjednostavniji mogući zapis izraza

```
Simplify[%]
```

$$-\frac{2\,a}{-1 + a^2}$$

Zadatak: Uzmite `izraz3 = (a + b) ((c + d x) x + f x^2)` i natjerajte *Mathematicu* da ga prikaže u slijedeća tri oblika:
(a) $(a + b)\,c\,x + (a + b)\,(d + f)\,x^2$
(b) $a\,c\,x + b\,c\,x + a\,d\,x^2 + b\,d\,x^2 + a\,f\,x^2 + b\,f\,x^2$
(c) `(a+b) x (c+d x+f x)`

Funkcija **Simplify** zna i trigonometrijske identitete

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

```
1
```

```
1/Cot[b] // Simplify
```

```
Tan[b]
```

Zadatak: Uvjerite se da vrijedi identitet
$\frac{\cos^3 x + \sin^3 x}{\cos x + \sin x} = 1 - \cos x \sin x$

■ 2.2 Jednadžbe

U *Mathematici* kao znak jednakosti u jednadžbama stoji "==", jer je uobičajeni znak "=" , kako smo gore već vidjeli, rezerviran za pridjeljivanje vrijednosti simbolima (imenima). Rješavanje jednadžbi se izvodi ugrađenom funkcijom **Solve**:

```
rjes = Solve[2 x^2 - 1 == 0, x]
```

$$\left\{\left\{x \to -\frac{1}{\sqrt{2}}\right\}, \left\{x \to \frac{1}{\sqrt{2}}\right\}\right\}$$

Valja primijetiti:
1. Prilikom poziva funkcije **Solve** treba eksplicitno naznačiti po kojoj varijabli se traži rješavanje.
2. *Mathematica* je pronašla oba rješenja kvadratne jednadžbe.
3. Rezultat je ispisan u obliku liste **{...}**takozvanih *transfromacijskih pravila* ("*transformation rules*"). Ta se pravila mogu onda primijeniti na proizvoljni izraz pomoću simbola **/.** koji znači "primijeni na prethodni izraz transformacijska pravila koja slijede":

```
x + a /. rjes
```

$$\left\{-\frac{1}{\sqrt{2}} + a, \frac{1}{\sqrt{2}} + a\right\}$$

Tako provjeru dobivenih rješenja možemo napraviti primjenom transformacijskih pravila na zadanu jednadžbu

```
2 x^2 - 1 == 0 /. rjes
```

```
{True, True}
```

(Uvrštavanjem rješenja u jednadžbu dobije se jednadžba poput **0==0** koji je uvijek zadovoljena pa odgovara logičkoj istini i *Mathematica* je automatski transformira u logičku Booleovu konstantu **True**.)

Transformacija izraza pomoću simbola **/.** je vrlo korisna operacija. Više riječi o njoj će biti u odjeljku 3.5, a sada ćemo samo pokazati njenu najčešću upotrebu: pridjeljivanje vrijednosti simbolima u nekom izrazu:

```
(a + b - c)^4 // Expand
```

$$a^4 + 4\,a^3\,b + 6\,a^2\,b^2 + 4\,a\,b^3 + b^4 - 4\,a^3\,c - 12\,a^2\,b\,c - 12\,a\,b^2\,c - 4\,b^3\,c + 6\,a^2\,c^2 + 12\,a\,b\,c^2 + 6\,b^2\,c^2 - 4\,a\,c^3 - 4\,b\,c^3 + c^4$$

```
% /. b → c
```

$$a^4$$

```
% /. a → 2
```

```
16
```

Funkcija **Solve** može rješavati i sustave jednadžbi, ako se kao argumenti zadaju liste jednadžbi odnosno varijabli:

```
Solve[{x^2 + y^2 == 1, x - 2 y == 0}, {x, y}]
```

$$\left\{\left\{x \to -\frac{2}{\sqrt{5}}, y \to -\frac{1}{\sqrt{5}}\right\}, \left\{x \to \frac{2}{\sqrt{5}}, y \to \frac{1}{\sqrt{5}}\right\}\right\}$$

Zadatak: Riješite sustav jednadžbi:
$x^4 + a^4 = 1$
$x^2 + a^2 = 1$
Koliko ima rješenja?

Funkcija **Solve** nastoji naći simboličko rješenje, no to je ponekad nemoguće. Npr. jednadžbe viših stupnjeva uopće ne moraju imati sva rješenja u zatvorenom obliku.

```
Solve[9 x^6 + 4 x^4 + 3 x^3 + x - 17 == 0, x]
```

```
{{x → 1}, {x → Root[17 + 16 #1 + 16 #1^2 + 13 #1^3 + 9 #1^4 + 9 #1^5 &, 1]},
 {x → Root[17 + 16 #1 + 16 #1^2 + 13 #1^3 + 9 #1^4 + 9 #1^5 &, 2]},
 {x → Root[17 + 16 #1 + 16 #1^2 + 13 #1^3 + 9 #1^4 + 9 #1^5 &, 3]},
 {x → Root[17 + 16 #1 + 16 #1^2 + 13 #1^3 + 9 #1^4 + 9 #1^5 &, 4]},
 {x → Root[17 + 16 #1 + 16 #1^2 + 13 #1^3 + 9 #1^4 + 9 #1^5 &, 5]}}
```

x=1 je trivijalno rješenje, ali ostalih pet rješenja *Mathematica* ne može napisati u zatvorenom obliku već ih zapisuje na poseban način u koji sad nećemo ulaziti. U takvim slučajevima možemo pribjeći numeričkom rješavanju koje radi funkcija **NSolve**:

```
NSolve[9 x^6 + 4 x^4 + 3 x^3 + x - 17 == 0, x]
```

```
{{x → -1.10302}, {x → -0.491102 - 0.988331 i}, {x → -0.491102 + 0.988331 i},
 {x → 0.54261 - 1.05431 i}, {x → 0.54261 + 1.05431 i}, {x → 1.}}
```

(Često simboličke funkcije u Mathematici imaju svoj numerički pandan koji ima isto ime do na početno slovo **N:** **Integrate ↔ NIntegrate, Sum ↔ NSum,** ...)

S transcendentalnim jednadžbama situacija je nešto teža. Naime, i **Solve** i **NSolve** su predviđene samo za rješavanje polinomnih jednadžbi.

```
NSolve[2 ArcTan[x] == x^2, x]
```

```
Solve::tdep : The equations appear to involve the
   variables to be solved for in an essentially non-algebraic way. More...
```

```
NSolve[2 ArcTan[x] == x^2, x]
```

Funkcija kojom zaista numerički rješavamo jednadžbe je **FindRoot**

```
? FindRoot
```

```
FindRoot[lhs==rhs, {x, x0}] searches for a numerical solution to the equation lhs==
   rhs, starting with x=x0. FindRoot[{eqn1, eqn2, ... }, {{x, x0}, {y, y0}, ... }]
   searches for a numerical solution to the simultaneous equations eqni. More...
```

Vidimo da **FindRoot** traži da mu zadamo početnu točku od koje kreće tražiti nultočku nekom od numeričkih metoda poput Newtonove.

```
FindRoot[2 ArcTan[x] == x^2, {x, 0}]
```

```
{x → 0.}
```

```
FindRoot[2 ArcTan[x] == x^2, {x, 1}]
```

```
{x → 1.37177}
```

Pronašli smo dva rješenja. Daljnji pokušaji daju uvijek jedno od ova dva:

```
FindRoot[2 ArcTan[x] == x^2, {x, 5}]
```

```
{x → 1.37177}
```

```
FindRoot[2 ArcTan[x] == x^2, {x, 1000}]
```

```
{x → 1.37177}
```

```
FindRoot[2 ArcTan[x] == x^2, {x, -5}]
```

```
{x → -1.86608 × 10^-23}
```

Ovo zadnje rješenje je zapravo 0 jer *Mathematica*, ako joj se ne kaže drugačije, radi numeriku s preciznošću od otprilike 10^{-15}.

```
Options[FindRoot]
```

```
{AccuracyGoal → Automatic, Compiled → True, DampingFactor → 1, EvaluationMonitor → None,
 Jacobian → Automatic, MaxIterations → 100, Method → Automatic,
 PrecisionGoal → Automatic, StepMonitor → None, WorkingPrecision → MachinePrecision}
```

```
$MachinePrecision
```

```
15.9546
```

Ove varijable s "**$**" na početku su tzv. *environment* varijable koje definiraju okolinu rada *Mathematice*.

```
$Version
```

```
5.0 for Linux (November 18, 2003)
```

Važna varijabla je **$Path** koja specificira direktorije po kojima će *Mathematica* tražiti datoteke koje želimo učitati.

```
$Path
```

```
{/home/kkumer/.mma/Applications, /usr/local/Wolfram/Mathematica/5.0/AddOns/JLink,
 /usr/local/Wolfram/Mathematica/5.0/AddOns/NETLink,
 /home/kkumer/.Mathematica/Kernel, /home/kkumer/.Mathematica/Autoload,
 /home/kkumer/.Mathematica/Applications, /usr/share/Mathematica/Kernel,
 /usr/share/Mathematica/Autoload, /usr/share/Mathematica/Applications, .,
 /home/kkumer, /usr/local/Wolfram/Mathematica/5.0/AddOns/StandardPackages,
 /usr/local/Wolfram/Mathematica/5.0/AddOns/StandardPackages/StartUp,
 /usr/local/Wolfram/Mathematica/5.0/AddOns/Autoload,
 /usr/local/Wolfram/Mathematica/5.0/AddOns/Applications,
 /usr/local/Wolfram/Mathematica/5.0/AddOns/ExtraPackages,
 /usr/local/Wolfram/Mathematica/5.0/SystemFiles/Graphics/Packages,
 /usr/local/Wolfram/Mathematica/5.0/Configuration/Kernel}
```

```
FindRoot[2 ArcTan[x] == x^2, {x, -5}, WorkingPrecision → 50]
```

```
{x → -1.5154949774453108169163268497033939159685369049224× 10^-92 }
```

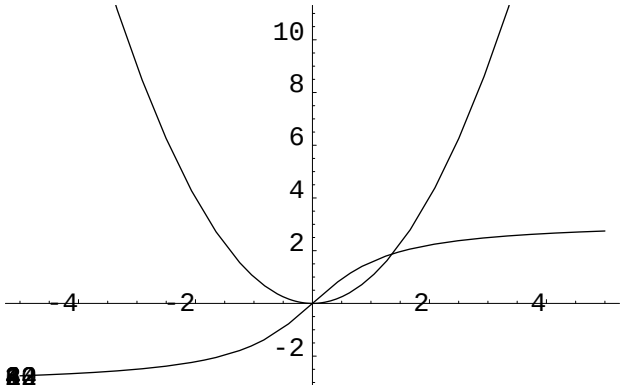
Vidimo da se s povećanjem radne preciznosti rješenje još više približilo nuli. Funkcija **Chop** stavlja u egzaktnu nulu sve brojeve koji su joj bliži od 10^{-10} i korisna je za povećanje preglednosti računa.

```
Chop[%]
```

```
{x → 0}
```

Možemo se uvjeriti da su ova gore dva rješenja zaista jedina, tako da skiciramo grafove funkcija 2 arctg(x) i x^2 i vidimo da se sijeku na samo dva mjesta. Koristimo funkciju **Plot** o kojoj će kasnije biti više riječi.

```
Plot[{2 ArcTan[x], x^2}, {x, -5, 5}]
```




```
- Graphics -
```

Domaći zadatak 2−1:

Riješite jednadžbu:
$\mathrm{tg}(x) - \frac{x}{10} = 0$

Domaći zadatak 2−2:

Nađite (kompleksno!) rješenje jednadžbe:
$\sin(x) = 2$
i provjerite uvrštavanjem. Pokušajte pronaći još neko rješenje

Domaći zadatak 2−3:

Riješite nejednadžbu:
$x^2 + x - 12 < 0$

Domaći zadatak 2−4:

Pronađite pozicije minimuma i maksimuma Besselove funkcije $J_1(x)$ koji su najbliži točki x=0.

2.3 Matematička analiza

Mathematica sadrži sve standardne operacije koje se uče u matematičkoj analizi, poput limesa, nizova, redova i diferencijalnog računa. Limesi se izvrijednjavaju funkcijom **Limit**

```
Limit[Sin[x]/x, x → 0]
```

```
1
```

Razvoj neke funkcije po nekoj varijabli oko neke točke do nekog reda radi **Series**:

```
Series[Sin[x], {x, 0, 5}]
```

$$x - \frac{x^3}{6} + \frac{x^5}{120} + O[x]^6$$

Zadatak: Odredite prva četiri člana u razvoju funkcije arctg(x) oko točke x=∞.

Domaći zadatak 2−5

Kolika je relativna pogreška koju radimo ako za izvrijednjavanje arctg(5) koristimo četveročlani red dobiven u gornjem zadatku? Da li bi greška bila manja da smo upotrebljavali četveročlani red oko x=0?

Deriviranje se radi funkcijom **D**:

```
D[e^(2 x) Cos[3 x], x]
```

$$2\, e^{2x} \text{Cos}[3x] - 3\, e^{2x} \text{Sin}[3x]$$

Višestruko deriviranje:

```
D[e^(2 x) Cos[3 x], {x, 4}]
```

$$-119\, e^{2x} \text{Cos}[3x] + 120\, e^{2x} \text{Sin}[3x]$$

Deriviranje (nepoznate) opće funkcije f(x), korištenjem Leibnitzovog lančanog pravila:

```
D[x^2 f[x], x]
```

$$2x\, f[x] + x^2\, f'[x]$$

Simboličko neodređeno integriranje:

```
Integrate[2 e^(2 x) Cos[3 x] - 3 e^(2 x) Sin[3 x], x]
```

$$e^{2x} \text{Cos}[3x]$$

Primijetite da se konstanta integracije podrazumijeva bez navođenja. Alternativno, postoji i zapis koji grafički bolje izgleda:

```
∫ x Log[x] dx
```

$$-\frac{x^2}{4} + \frac{1}{2} x^2 \text{Log}[x]$$

Simboličko rješavanje određenih integrala:

```
Integrate[x Log[x], {x, 0, 1}]
```

$$-\frac{1}{4}$$

Neki integrali su preteški ili se naprosto ne daju prikazati u zatvorenoj formi pa *Mathematica*, kao i uvijek kad nešto ne zna izračunati, vraća zadani izraz:

```
Integrate[ArcTan[BesselJ[1, x]], {x, 0, 1}]
```

$$\int_0^1 \text{ArcTan}[\text{BesselJ}[1, x]]\, dx$$

Tada nam ostaje numerička integracija pomoću **NIntegrate**

```
NIntegrate[ArcTan[BesselJ[1, x]], {x, 0, 1}]
```

```
0.227279
```

Zadatak: Izračunajte neodređeni integral
$\int \frac{x^2+3}{x^5+x^4-x-1}\,dx$
i onda provjerite dobiveni rezultat deriviranjem i algebarskim manipulacijama.

Zadatak: Izračunajte određene integrale
$\int_0^1 \sin(\sin(x))\,dx$ $\int_0^\pi \sin(\sin(x))\,dx$
simbolički i numerički.

Domaći zadatak 2−6: Izračunajte dvostruki određeni integral
$\int_0^1 dx \int_0^{1-x} dy\,(x+y)\sqrt{x\,y^3}$

Domaći zadatak 2−7: Izračunajte
$\int dx \left(\int dy\,(x+y)\sqrt{x\,y^3}\right)$
i onda provjerite dobiveni rezultat deriviranjem i algebarskim manipulacijama.

2.4 Linearna algebra

Vektori su u *Mathematici* reprezentirani kao liste

```
vec1 = {1, 1, 2}
vec2 = {2, 2, 4}
```

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

```
{2, 2, 4}
```

Skalarni produkt vektora se dobije točkom "**.**" (ili funkcijom **Dot**),

```
vec1.vec2
```

```
12
```

a vektorski produkt funkcijom **Cross**

```
Cross[vec1, vec2]
```

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

Matrice su liste odgovarajućih red−vektora

```
mat = {{1, 2, 1}, {4, 3, 3}, {9, 1, 7}}
```

```
{{1, 2, 1}, {4, 3, 3}, {9, 1, 7}}
```

Za prirodniji ispis koristimo funkciju **MatrixForm**

```
MatrixForm[%]
```

$$\begin{pmatrix} 1 & 2 & 1 \\ 4 & 3 & 3 \\ 9 & 1 & 7 \end{pmatrix}$$

Za upis matrice možemo koristiti i izbornik `Input->Create Table/Matrix/Pallete`

$$\text{mat} = \begin{pmatrix} \mathbf{1} & \mathbf{2} & \mathbf{1} \\ \mathbf{4} & \mathbf{3} & \mathbf{3} \\ \mathbf{9} & \mathbf{1} & \mathbf{7} \end{pmatrix}$$

```
{{1, 2, 1}, {4, 3, 3}, {9, 1, 7}}
```

S matricama i vektorima možemo raditi sve ono na što smo navikli. Važno je samo znati da se množenje izvodi točkom "**.**" (koja označava *nekomutativno* množenje), a ne zvjezdicom "*****" ili razmakom kao kod običnog množenja koje je *komutativno*. Zaboravljanje točke je najčešći uzrok pogrešaka kod rada s matricama.

Množenje matrica (uočite točku!):

```
mat.mat
```

```
{{18, 9, 14}, {43, 20, 34}, {76, 28, 61}}
```

```
MatrixForm[%]
```

$$\begin{pmatrix} 18 & 9 & 14 \\ 43 & 20 & 34 \\ 76 & 28 & 61 \end{pmatrix}$$

Množenje matrice i vektora (uočite točku!):

```
mat.vec1
```

```
{5, 13, 24}
```

Inverz matrice:

```
Inverse[mat]
```

$$\left\{\left\{-\frac{18}{7}, \frac{13}{7}, -\frac{3}{7}\right\}, \left\{\frac{1}{7}, \frac{2}{7}, -\frac{1}{7}\right\}, \left\{\frac{23}{7}, -\frac{17}{7}, \frac{5}{7}\right\}\right\}$$

Provjera

```
mat % // MatrixForm
```

$$\begin{pmatrix} -\frac{18}{7} & \frac{26}{7} & -\frac{3}{7} \\ \frac{4}{7} & \frac{6}{7} & -\frac{3}{7} \\ \frac{207}{7} & -\frac{17}{7} & 5 \end{pmatrix}$$

Primijetite da smo upotrijebili *postfiksni* oblik funkcije **`MatrixForm`**.

Moguće je i mijenjati pojedine elemente matrice. Sjetite se da se element liste dobiva pomoću dvostrukih uglatih zagrada. (Njihov elegantan zapis se postiže sekvencama `Esc-[[-Esc` i `Esc-]]-Esc`

```
mat[[1, 1]] = 0
```

```
0
```

```
mat // MatrixForm
```

$$\begin{pmatrix} 0 & 2 & 1 \\ 4 & 3 & 3 \\ 9 & 1 & 7 \end{pmatrix}$$

Zadatak: Odredite svojstvene vrijednosti (*eigenvalues*) i svojstvene vektore (*eigenvectors*) matrice

$$\begin{pmatrix} 2.3 & 4.5 \\ 6.7 & -1.2 \end{pmatrix}$$

i provjerite za jedan od tih vektora da je zaista svojstveni eksplicitnim množenjem.

Domaći zadatak 2−8:

(1)Koristeći funkcije **Table** i **Random** kreirajte 3x3 matricu sa slučajnim realnim brojevima između 0 i 10. (2)Zamijenite središnji element matrice simbolom **x**. (3)Invertirajte novonastalu matricu te je (4)pomnožite s neinvertiranom i provjerite da je rezultat jedinična matrica.

Domaći zadatak 2−9:

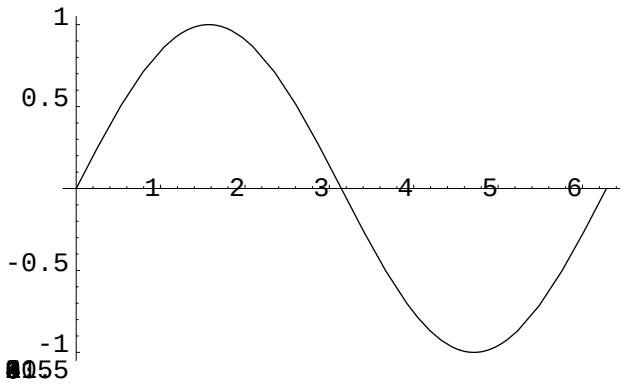
Kakvo množenje matrica dobivamo ako izostavimo točku u A.B?

2.5 Crtanje grafova

2D

Glavna komanda za dvodimenzionalno crtanje je **Plot**.

```
Plot[Sin[x], {x, 0, 2 π}]
```




```
- Graphics -
```

Poput većine *Mathematica* komandi i **Plot** ima svoje *opcije* pomoću kojih možemo podešavati njegovo ponašanje. Evo ih zajedno s njihovim defaultnim vrijednostima:

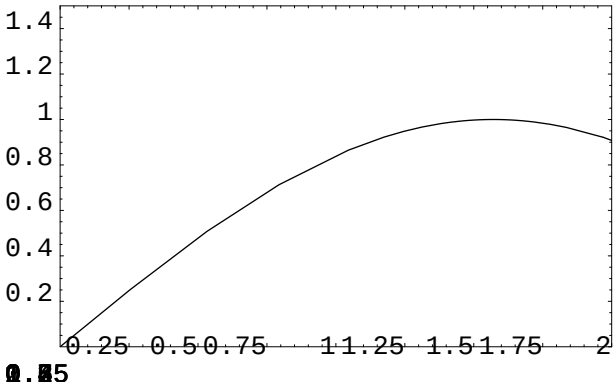
```
Options[Plot]
```

```
{AspectRatio → 1/GoldenRatio, Axes → Automatic, AxesLabel → None,
 AxesOrigin → Automatic, AxesStyle → Automatic, Background → Automatic,
 ColorOutput → Automatic, Compiled → True, DefaultColor → Automatic,
 DefaultFont :→ $DefaultFont, DisplayFunction :→ $DisplayFunction, Epilog → {},
 FormatType :→ $FormatType, Frame → False, FrameLabel → None, FrameStyle → Automatic,
 FrameTicks → Automatic, GridLines → None, ImageSize → Automatic,
 MaxBend → 10., PlotDivision → 30., PlotLabel → None, PlotPoints → 25,
 PlotRange → Automatic, PlotRegion → Automatic, PlotStyle → Automatic,
 Prolog → {}, RotateLabel → True, TextStyle :→ $TextStyle, Ticks → Automatic}
```

Za detaljno značenje pojedinih opcija pogledajte dokumentaciju. One najzanimljivije su:

PlotRange	Dio koordinatne ravnine koji se crta		
Axes	Da li crtamo osi?		
Frame	Da li crtamo okvir?		
AxesLabel	Oznake na osima		
DisplayFunction	Kamo crtamo?	ekran	**$DisplayFunction**
		datoteka	**Function[Display[** < *ime datoteke* >**, #]]**
		nigdje	**Identity**

```
Plot[Sin[x], {x, 0, 2 π}, AxesLabel → {"x", "sin(x)"},
 PlotRange → {{0, 2}, {0, 1.5}}, Frame → True]
```




```
- Graphics -
```

Redosljed kojim specificiramo opcije je nebitan.

Slijedeći primjer bi trebao rezultirati datotekom *sinus.gif* u radnom direktoriju. (Za popis mogućih grafičkih formata vidi u manualu funkciju **Export**.)

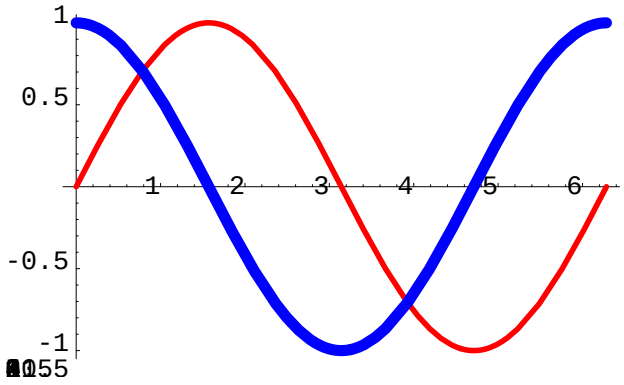
```
Plot[Sin[x], {x, 0, 2 π}, DisplayFunction → Function[Display["sinus.gif", #, "GIF"]]]
```

```
- Graphics -
```

Zadatak: Pronađite ovu datoteku na hard disku, prikažite je na ekranu pa onda izbrišite. (Radni direktorij možete pronaći izdavanjem komande **Directory[]**).

Za specificiranje svojstava samih grafova funkcija koristi se opcija **PlotStyle** prema donjem primjeru koji pokazuje i kako na istoj slici istovremeno prikazati više grafova.

```
Plot[{Sin[x], Cos[x]}, {x, 0, 2 π}, PlotStyle →
  {{Thickness[0.01], RGBColor[1, 0, 0]}, {Thickness[0.02], RGBColor[0, 0, 1]}}]
```




```
- Graphics -
```

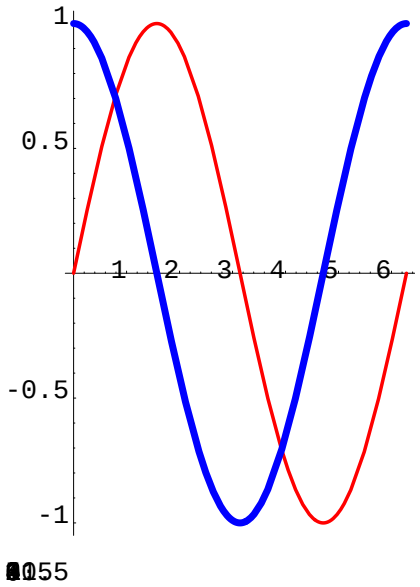
Važno je primijetiti kako *output* **Plot** komande nije sam graf već tzv. grafički (*Graphics*) objekt. Graf na ekranu je samo usputni rezultat proizvodnje tog grafičkog objekta. Zato "**%**" ne daje ono što bismo naivno očekivali:

```
%
```

```
- Graphics -
```

Za ponovno prikazivanje tog objekta možemo koristiti funkciju **Show**, koja omogućava i modificiranje opcija grafa.

```
Show[%%, AspectRatio → 1.5]
```




```
- Graphics -
```

Zadatak: Nacrtajte graf funkcije *ln(x)* između x=0.5 i x=1.5 bez ikakvih oznaka, okvira i osi, dakle samo liniju.

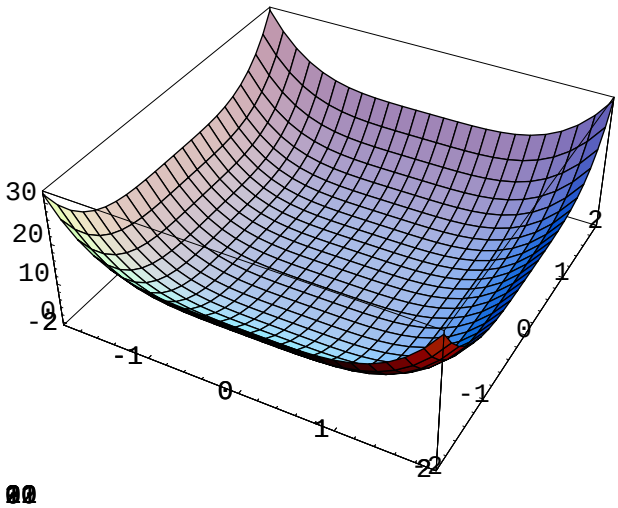
Domaći zadatak 2−9:

Koristeći funkciju **ParametricPlot** nacrtajte kružnicu (na ekranu treba izgledati baš kao kružnica, a ne kao elipsa).

■ 3D

Glavna komanda za trodimenzionalno crtanje je **Plot3D**.

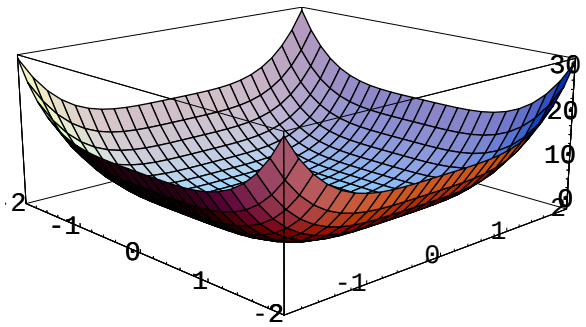
```
Plot3D[x^4 + y^4, {x, -2, 2}, {y, -2, 2}]
```




```
- SurfaceGraphics -
```

Za promjenu kuta gledanja koristimo opciju **ViewPoint**, čiju vrijednost možemo odrediti pomoću dijaloga `Input→3D ViewPoint Selector ...`

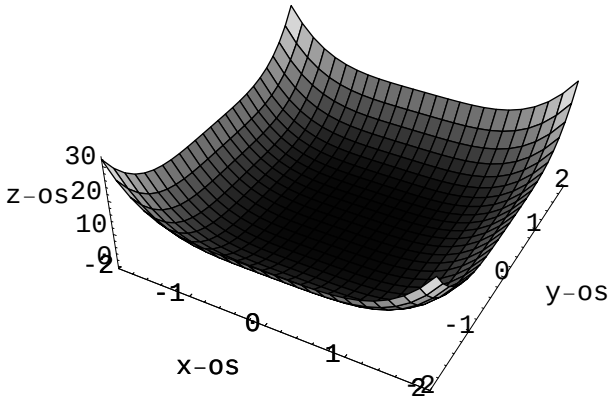
```
Show[%, ViewPoint -> {2.375, -2.232, 0.910}]
```




```
- SurfaceGraphics -
```

Plot3D ima većinu opcija kao i **Plot** plus još dosta novih

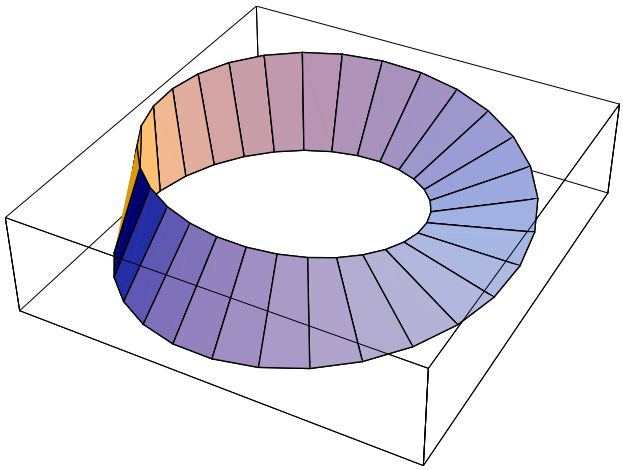
```
Show[%%, Boxed → False, AxesLabel → {"x-os", "y-os", "z-os"}, Lighting → False]
```




```
00
- SurfaceGraphics -
```

Osim velikog broja ugrađenih ("*built−in*") funkcija, s *Mathematicom* dolazi i određen broj *paketa* s dodatnim funkcijama. One su isto dokumentirane u Help−u (vidi tamo odjeljak "*Add−ons & Links*"), ali je za njihovu upotrebu prvo potrebno učitati odgovarajući paket. Npr. paket **Graphics'Shapes'** definira neke zanimljive plohe

```
<<Graphics'Shapes'

Show[Graphics3D[MoebiusStrip[2, 0.6, 30]]]
```



```
- Graphics3D -
```

Još više dodatnih paketa se može skinuti s http://library.wolfram.com/infocenter/MathSource/

■ 2.6 Grafički prikaz i fitanje podataka

Često je potrebno neke podatke dobivene npr. mjerenjima grafički prikazati i približno opisati nekom funkcijom (tzv. *fitanje*). Prvo je potrebno učitati podatke u *Mathematica* listu oblika

$$\{\{x_1, y_1\}, \{x_2, y_2\}, \ldots\}$$

Neka u radnom direktoriju bude poddirektorij *files* s datotekom *fit1.dat* s podacima složenim u dva stupca. Izravan ispis datoteke je moguć ovako (na MS Windows platformi je možda nužno zamijeniti "/" → "\")

```
!! "files/fit1.dat"
```

```
0.2  0.1
0.35  0.2
1    0.6
1.8  0.9
2.5  1.3
4    2.06
5    2.6
```

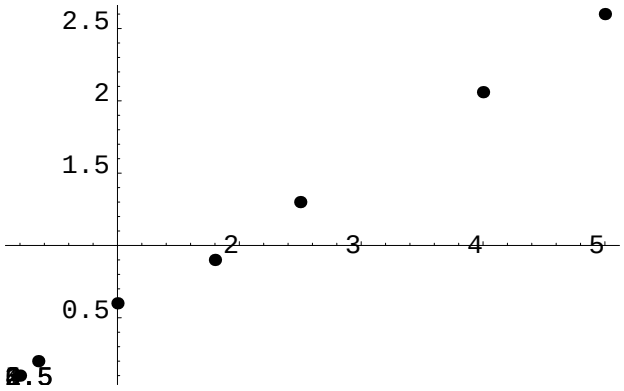
Funkcija **ReadList** omogućuje učitavanje ovih podataka izravno u *Mathematica* listu

```
podaci1 = ReadList["files/fit1.dat", {Number, Number}]
```

```
{{0.2, 0.1}, {0.35, 0.2}, {1, 0.6}, {1.8, 0.9}, {2.5, 1.3}, {4, 2.06}, {5, 2.6}}
```

Za grafički prikaz možemo koristiti funkcije **ListPlot** i **ScatterPlot3D**.

```
ListPlot[podaci1, PlotStyle → {PointSize[0.02]}]
```




```
- Graphics -
```

(Default veličina točaka je obično presitna za ugodno gledanje.)

Fitanje se izvodi funkcijom

Fit[*<lista s podacima>*, *<lista funkcija kojima fitamo>*, *<lista varijabli>***]**

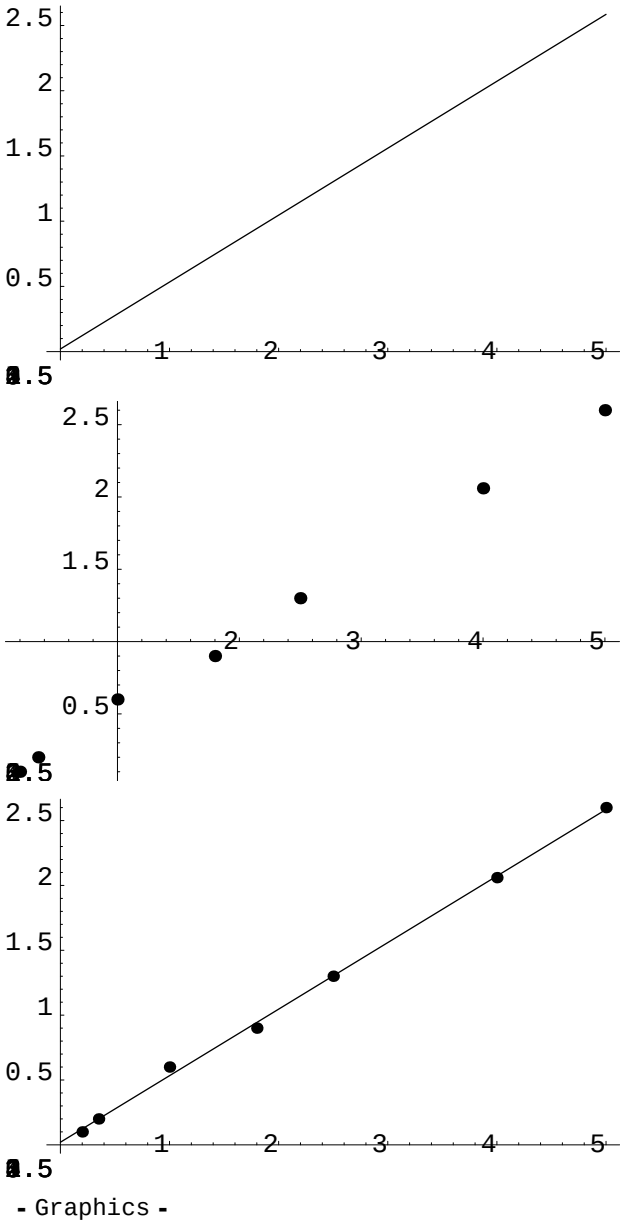
koja izvodi fitanje metodom najmanjih kvadrata na linearnu kombinaciju specificiranih funkcija. Npr. linearni fit ovih podataka na funkciju $f(x)=a\,x+b$ je

```
Fit[podaci1, {1, x}, {x}]
```

```
0.0201595 + 0.513056 x
```

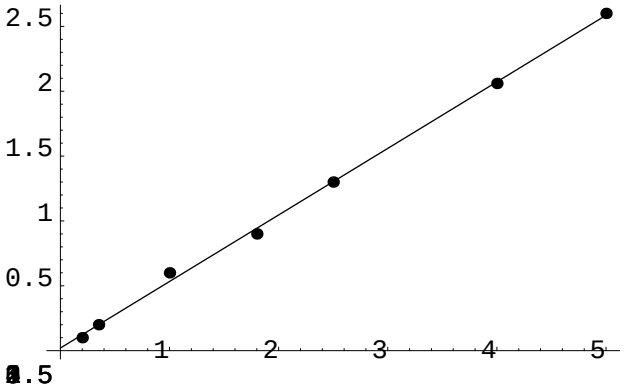
Grafičku provjeru kvalitete fita možemo provesti zajedničkim iscrtavanjem ovog pravca i podataka. (Primjetite kako cut'n'paste gornjeg rezultata u donju *input* ćeliju pokazuje da *Mathematica* interno reprezentira brojeve preciznije nego što ih ispisuje.)

```
Show[Plot[0.020159523556311256` + 0.5130561168421428` x, {x, 0, 5}],
 ListPlot[podaci1, PlotStyle → {PointSize[0.02]}]]
```



Vidimo da kao nusprodukt imamo dva neželjena grafa. To spriječimo korištenjem opcije **`DisplayFunction`** kojom isključimo njihovo iscrtavanje.

```
Show[Plot[0.020159523556311256` + 0.5130561168421428` x,
  {x, 0, 5}, DisplayFunction → Identity],
 ListPlot[podaci1, PlotStyle → {PointSize[0.02]}, DisplayFunction → Identity],
 DisplayFunction → $DisplayFunction]
```




```
- Graphics -
```

Zadatak: Podatke iz datoteke *fit2.dat* nafitajte na polinom trećeg reda, te na *f(x)=a sin(x)* te nacrtajte sve na jednom grafu. Neka sinusna funkcija bude u boji radi lakšeg snalaženja.

Fit radi s *linearnom* kombinacijom zadanih funkcija. Jedno moguće poboljšanje je korištenje funkcije **FindFit** (pojavljuje se tek u *Mathematici* 5.0) koja može raditi *nelinearni* fit.

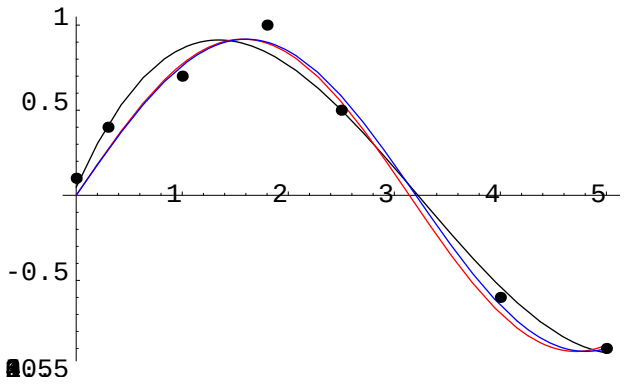
```
podaci2 = ReadList["files/fit2.dat", {Number, Number}]
```

```
{{0, 0.1}, {0.3, 0.4}, {1, 0.7}, {1.8, 1}, {2.5, 0.5}, {4, -0.6}, {5, -0.9}}
```

```
FindFit[podaci2, a Sin[b x], {a, b}, {x}]
```

```
{a → 0.917041, b → 0.980775}
```

```
Show[Plot[0.04731370851715653` + 1.4100207533957259` x - 0.6617882175300662` x^2 +
   0.06817318237314943` x^3, {x, 0, 5}, DisplayFunction → Identity],
 Plot[0.9174221173298753` Sin[x], {x, 0, 5}, DisplayFunction → Identity,
  PlotStyle → {RGBColor[1, 0, 0]}], Plot[0.917041 Sin[0.980775 x],
  {x, 0, 5}, DisplayFunction → Identity, PlotStyle → {RGBColor[0, 0, 1]}],
 ListPlot[podaci2, PlotStyle → {PointSize[0.02]}, DisplayFunction → Identity],
 DisplayFunction → $DisplayFunction]
```




```
- Graphics -
```

Koji je fit najbolji? Trebala bi pažljiva analiza koju ćemo ostaviti za neki kasniji zadatak.

Domaći zadatak 2−10:

Pronađite neka svoja mjerenja koja ste napravili na Fizičkom praktikumu i načinite fitanje nekom razumnom funkcijom. Nacrtajte grafički prikaz i stavite na njega legendu.

■ 2.7 Diferencijalne jednadžbe

Osnovna funkcija koja traži analitička rješenja običnih diferencijalnih jednadžbi je **DSolve**. Riješimo pomoću nje jednadžbu gušenog harmoničkog oscilatora

$$\frac{d^2 y}{dt^2} + 2\,\frac{dy}{dt} + 4\,y = 0$$

```
DSolve[y''[t] + 2 y'[t] + 4 y[t] == 0, y[t], t]
```

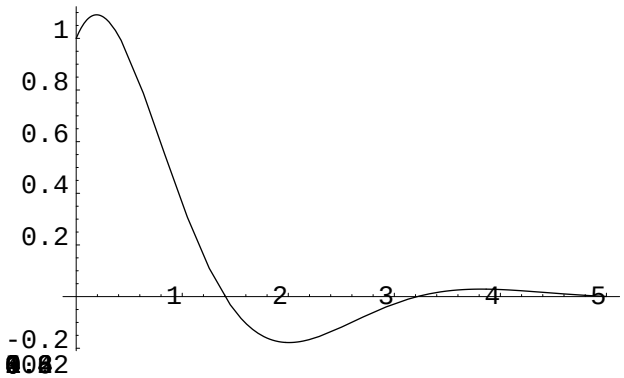
```
{{y[t] → ℯ^-t C[2] Cos[√3 t] + ℯ^-t C[1] Sin[√3 t]}}
```

Primijetite da smo dobili *opć e* rješenje s dvije nepoznate konstante **C[1]** i **C[2]** koje treba odrediti iz početnih uvjeta. Ukoliko želimo partikularno rješenje za neke konkretne početne uvjete, zadamo te uvjete kao dodatne jednadžbe:

```
sol = DSolve[{y''[t] + 2 y'[t] + 4 y[t] == 0, y'[0] == 1, y[0] == 1}, y[t], t]
```

```
{{y[t] → 1/3 ℯ^-t (3 Cos[√3 t] + 2 √3 Sin[√3 t])}}
```

```
Plot[1/3 ℯ^-t (3 Cos[√3 t] + 2 √3 Sin[√3 t]), {t, 0, 5}]
```




```
- Graphics -
```

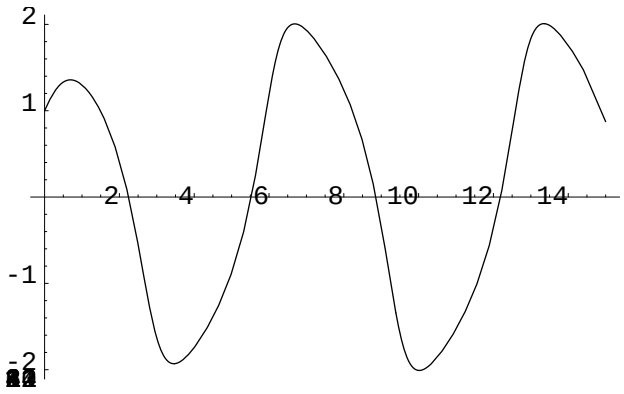
Neke jednadžbe se ne mogu riješiti analitički, pa moramo pribjeći numeričkom integriranju funkcijom **NDSolve**. Npr. na kolegiju *Diferencijalne jednadžbe i dinamički sustavi* numerički se analizira tzv. Van der Polova jednadžba.

```
sol =
 NDSolve[{x''[t] - (1 - x[t]^2) x'[t] + x[t] == 0, x'[0] == 1, x[0] == 1}, x[t], {t, 0, 15}]
```

```
{{x[t] → InterpolatingFunction[{{0., 15.}}, <>][t]}}
```

Rezultirajući objekt je specijalni oblik funkcije, **InterpolatingFunction**, koji nije analitički, ali je možemo izvrijedniti u nekoj konkretnoj točki ili nacrtati.

```
Plot[Evaluate[x[t] /. sol], {t, 0, 15}]
```



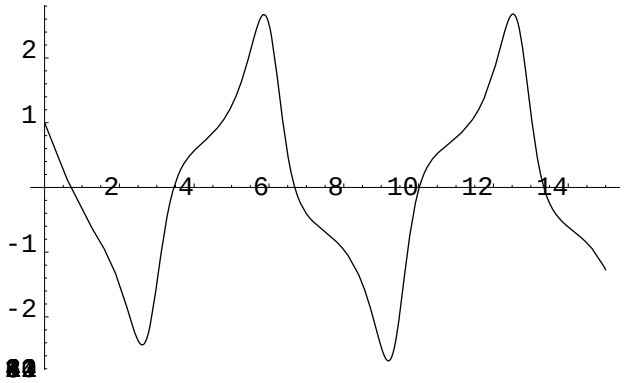

```
- Graphics -
```

InterpolatingFunction objekt možemo i derivirati

```
dersol = D[Evaluate[x[t] /. sol], t]
```

```
{InterpolatingFunction[{{0., 15.}}, <>][t]}
```

```
Plot[Evaluate[dersol], {t, 0, 15}]
```




```
- Graphics -
```

NDSolve može riješiti i neke jednostavnije *parcijalne* diferencijalne jednadžbe.

Zadatak: Pomoću **ParametricPlot** nacrtajte graf derivacije gornjeg rješenja Van der Polove jednadžbe x'(t) u ovisnosti o rješenju x(t) (tzv. *fazni dijagram*).

Zadatak: Neka na česticu u ravnini djeluje sila $\vec{F}$=8 x y^3 $\vec{i}$ + 12 x^2 y^2 $\vec{j}$. Odredite odgovarajući potencijal U(x,y) bilo ručno bilo pomoću *Mathematice*. Provjerite da vrijedi $\vec{F} = -\nabla U$. Funkcija *gradijent* postoji kao **Grad** u standardnom dodatnom paketu **Calculus'VectorAnalysis'** . Nacrtajte potencijal U(x,y) i pomoću **Plot3D** i pomoću **ContourPlot**. Nadalje, rješavajući Newtonovu diferencijalnu jednadžbu gibanja, pronađite putanju čestice. Nacrtajte tu putanju, a onda je pokušajte superponirati na gornje dijagrame potencijala. Igrajte se malo s početnim uvjetima i uvjerite se da je sve OK. Izvrijednite ukupnu energiju za razna vremena i uvjerite se da je sustav zaista konzervativan tj. da je energija konstantna.

Domaći zadatak 2 – 11 : Isto kao gornji zadatak, ali neka putanje ne budu kontinuirane linije već točke koje su ekvidistantne u vremenu tako da njihov razmak opisuje trenutnu brzinu tijela. Veći razmak odgovara većoj brzini.

3. Osnove *Mathematica* jezika

■ 3.1 Liste

■ Liste i elementi

Liste su važan dio *Mathematica* jezika. Srodne su poljima (*array*) iz standardnih programskih jezika (C, Fortran), ali imaju bitno više svojstava. Elementi liste mogu biti praktički bilo koji *Mathematica* objekti.

```
l1 = List[1, 3, 5, 7, 9, 11]
l2 = {Sin, Cos, Log}
l3 = {{}, {"jedan"}, %}
```

```
{1, 3, 5, 7, 9, 11}
```

```
{Sin, Cos, Log}
```

```
{{}, {jedan}, {Sin, Cos, Log}}
```

Vitičaste zagrade su samo skraćeni oblik funkcije **`List`**. Puni oblik se rijetko koristi.

Elementima liste se pristupa putem funkcije **`Part`**, odnosno njenog skraćenog oblika **`[[ ]]`** koji se može ukucati i putem sekvence `Esc-[-[-Esc` što onda možda ljepše izgleda.

```
Part[l1, 2]
l1[[2]] = "tri"
l1⟦2⟧
```

```
3
```

```
tri
```

```
tri
```

```
l1⟦{3, 6}⟧
Take[l1, {3, 6}]
```

```
{5, 11}
```

```
{5, 7, 9, 11}
```

```
l1
```

```
{1, tri, 5, 7, 9, 11}
```

Listama se može manipulirati na skoro svaki zamislivi način. **`First, Last, Take, Drop, Rest, Extract, Length, Prepend, PrependTo, Append, AppendTo, Insert, Delete`**, ... (vidi Help).

Zadatak: Izbrišite u gornjoj listi **`l1`** treći element i dodajte na kraj još dva elementa, brojeve 13 i 15.

Elementi liste mogu biti i druge liste, pa onda imamo višedimenzionalne liste. Dvodimenzionalne liste (listu lista jednake duljine) smo već sreli kao matrice. Evo primjera trodimenzionalne liste:

```
l4 = {{{a, b}, {h}, {d, e}}, {{f}, {g, h}, {i, j, k}}}
```

```
{{{a, b}, {h}, {d, e}}, {{f}, {g, h}, {i, j, k}}}
```

```
l4[[1]]
l4[[1]][[3]]
l4[[1, 3]]
l4[[1, 3, 2]]
```

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

```
{d, e}
```

```
{d, e}
```

```
e
```

```
Depth[l4]
```

```
4
```

```
pos = Position[l4, h]
```

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

```
l4[[2, 2, 2]]
```

```
h
```

Rezultat funkcije **Position** je lista pozicija, formatom prilagođena za funkcije poput **ReplacePart**

```
ReplacePart[l4, "ejdz", pos]
```

```
{{{a, b}, {ejdz}, {d, e}}, {{f}, {g, ejdz}, {i, j, k}}}
```

Primijetite da **ReplacePart** ne mijenja originalnu listu:

```
l4
```

```
{{{a, b}, {h}, {d, e}}, {{f}, {g, h}, {i, j, k}}}
```

Često se javlja potreba za izborom elemenata liste koji zadovoljavaju neki kriterij. To se izvodi funkcijom **Select**.

```
l5 = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
Select[l5, EvenQ]
```

```
{0, 2, 4, 6, 8, 10}
```

Drugi argument funkcije **Select** može biti bilo koji *predikat*. Predikat je termin iz matematičke logike koji označava funkciju koja poprima isključivo vrijednosti **True** ili **False**.

```
IntegerQ[3.]
```

```
False
```

Određen broj predikata je već ugrađen u *Mathematicu* (svima ime završava na "**Q**" od "Question" jer postavljaju neko *da−ne* pitanje svojim argumentima), a kasnije ćemo vidjeti kako ih možemo i sami definirati.

```
?*Q
```

System‘

ArgumentCountQ	IntervalMemberQ	MatrixQ	SameQ
ArrayQ	InverseEllipticNomeQ	MemberQ	StringMatchQ
AtomQ	LegendreQ	NameQ	StringQ
DigitQ	LetterQ	NumberQ	SyntaxQ
EllipticNomeQ	LinkConnectedQ	NumericQ	TensorQ
EvenQ	LinkReadyQ	OddQ	TrueQ
ExactNumberQ	ListQ	OptionQ	UnsameQ
FreeQ	LowerCaseQ	OrderedQ	UpperCaseQ
HypergeometricPFQ	MachineNumberQ	PartitionsQ	ValueQ
InexactNumberQ	MatchLocalNameQ	PolynomialQ	VectorQ
IntegerQ	MatchQ	PrimeQ	

```
NumberQ[x]
NumberQ[3.]
```

```
False
```

```
True
```

Prilikom programiranja u *Mathematici* se razmjerno često javlja potreba za smanjivanjem dimenzije liste tj. za micanje zagrada. Funkcija **Flatten** bez dodatnog argumenta pretvara svaku listu u jednodimenzionalnu tj. miče sve unutarnje zagrade, dok dodatnim argumentom možemo zatražiti da to ide samo do neke razine.

```
Flatten[l4]
Flatten[l4, 1]
```

```
{a, b, h, d, e, f, g, h, i, j, k}
```

```
{{a, b}, {h}, {d, e}, {f}, {g, h}, {i, j, k}}
```

■ Konstruiranje lista

```
Range[10]
```

```
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

```
Range[2, 5, 0.5]
```

```
{2, 2.5, 3., 3.5, 4., 4.5, 5.}
```

Range radi samo s brojevima. Univerzalnija funkcija za konstruiranje lista je **Table**.

```
Table[x^n/n!, {n, 1, 5}]
```

$$\left\{x, \frac{x^2}{2}, \frac{x^3}{6}, \frac{x^4}{24}, \frac{x^5}{120}\right\}$$

```
tbl = Table[a[i, j], {i, 1, 3}, {j, 1, 2}]
```

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

```
tbl // TableForm
```

```
a[1, 1]    a[1, 2]
a[2, 1]    a[2, 2]
a[3, 1]    a[3, 2]
```

Nekoliko *Mathematica* funkcija (**Integrate**, **Sum**, ...) poput **Table** kao argumente imaju i objekte koji se zovu *iteratori* i dolaze u slijedećim oblicima:

{n, nmin, nmax, step}	izraz se ponavlja, a **n** ide od **nmin** do **nmax**, uz korak **step**
{n, nmin, nmax}	izraz se ponavlja, a **n** ide od **nmin** do **nmax**, uz korak **1**
{n, nmax}	izraz se ponavlja, a **n** ide od 1 do **nmax**, uz korak **1**
{n}	izraz se ponavlja **n** puta

Za konstruiranje dijagonalnih matrica postoje specijalne funkcije **DiagonalMatrix** i **IdentityMatrix**

```
DiagonalMatrix[{1, w, 2}] // MatrixForm
```

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & w & 0 \\ 0 & 0 & 2 \end{pmatrix}$$

```
IdentityMatrix[2] // MatrixForm
```

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

Zadatak: Konstruirajte listu svih prim−brojeva manjih od 100. Koliko ima prim−brojeva manjih od $10^6, 10^7, 10^8$, ...? Usporedite rezultat (i brzinu njegovog dobivanja, vidi funkciju **Timing**) s ugrađenom funkcijom **PrimePi**, te slijedećom aproksimacijom (koja je sadržaj slavnog teorema o prim−brojevima)

$$\pi(x) \approx \int_0^x \frac{dz}{\ln(z)} \qquad x>1$$

Domaći zadatak 3−1: Izračunajte funkciju π(x) (broj prim−brojeva manjih od x) za $x = 10^{10}$ statističkom metodom. Izaberite uzorak od n slučajnih cijelih brojeva između 1 i x i testirajte koliko ima prim – brojeva među njima. Iz tog udjela odredite π(x). Kolika veličina uzorka vam treba da bi greška prema **PrimePi[10^{10}]** pala ispod 1 %?

Liste kao skupovi

Liste nisu skupovi u matematičkom smislu jer u skupovima nema ponavljanja elemenata i redosljed elemenata u skupovima nije bitan. Međutim, *Mathematica* ima nekoliko funkcija koje rade s listama standardne operacije koje se inače rade sa skupovima (unija je **Union**, presjek **Intersection**, a razlika skupova **Complement**) i pritom eliminira duple elemente te uredi redosljed (tj. originalni redoslijed je nebitan).

```
l6 = {10, 9, 10, 8, 7, 7, 7, 4}
```

```
{10, 9, 10, 8, 7, 7, 7, 4}
```

```
Union[l6]
```

```
{4, 7, 8, 9, 10}
```

```
Union[l6, l1]
```

```
{1, 4, 5, 7, 8, 9, 10, 11, tri}
```

```
Intersection[l6, l1]
Complement[l6, l1]
```

```
{7, 9}
```

```
{4, 8, 10}
```

■ Listabilnost funkcija

Listabilnost je *atribut* koje imaju neke *Mathematica* fukcije, a riječ je o svojstvu da se one, primijenjene na listu, distribuiraju po elementima liste.

```
l7 = {1, 2, x, x^2};
Log[l7]
```

```
{0, Log[2], Log[x], Log[x^2]}
```

```
l7^2 + 1
```

```
{2, 5, 1 + x^2, 1 + x^4}
```

Ispitati da li neka funkcija ima to svojstvo moguće je funkcijom **Attributes**.

```
Attributes[{Log, Length, Power, Attributes}] // MatrixForm
```

```
⎛       {Listable, NumericFunction, Protected}       ⎞
⎜                    {Protected}                     ⎟
⎜{Listable, NumericFunction, OneIdentity, Protected}⎟
⎝           {HoldAll, Listable, Protected}           ⎠
```

Primijetite da je i **Attributes** listabilna što smo ovdje odmah i upotrijebili. Atribute možemo dobiti i Help−funkcijom "**??**" koja i inače daje više podataka o objektu od jednostrukog "**?**".

```
?? Log
```

```
Log[z] gives the natural logarithm of z (
   logarithm to base e). Log[b, z] gives the logarithm to base b. More...
```

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

Zadatak: Napišite *one−liner* (elegantni program koji je cijeli sadržan u jednom redu - ponos svakog programera) koji ispisuje slijedeću tablicu (desni stupac je derivacija lijevog):

x	1
x^2	$2\,x$
x^3	$3\,x^2$
x^4	$4\,x^3$
x^5	$5\,x^4$

Komanda ne smije uključivati nikave brojeve izuzev broja 5! (Jeftini trikovi poput 5/5+5/5 naravno ne dolaze u obzir.)

■ 3.2 Funkcije

■ Vrste pridruživanja

Već znamo da se pridruživanje ("*assignment*") vrijednosti simbolu izvodi jednostrukim znakom jednakosti

```
m = 3;
n = 4;
zbroj = m + n
```

```
7
```

Ako sad promijenimo vrijednost simbolu **m**

```
m = 6;
zbroj
```

```
7
```

vidimo da se zbroj nije promijenio. Takva je narav operacije pridruživanja vrijednosti znakom "**=**". Postoji i druga mogućnost, tzv. *odgođeno pridruživanje* ("*delayed assignment*"), koje se izvodi znakom "**:=**".

```
zbroj2 := m + n
```

Primijetite da *Mathematica* nije dala nikakav output. Dat će ga tek prilikom poziva varijable **zbroj2** kad će i izvršiti njegovo izračunavanje prema trenutnim vrijednostima varijabli **a** i **b**.

```
zbroj2
```

```
10
```

```
n = w;
zbroj2
```

```
6 + w
```

```
? zbroj
```

```
Global‘zbroj

zbroj = 7
```

```
? zbroj2
```

```
Global‘zbroj2

zbroj2 := m + n
```

■ Definiranje funkcija

Mogućnost definiranja novih funkcija je osnovna stepenica k naprednijem programiranju. Nove funkcije u *Mathematici* definiramo na slijedeći način

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

```
f[2]
```

```
4
```

```
f[(2 x + 3)^2]
```

```
(3 + 2 x)^4
```

Jasno je da je prilikom definicije funkcija potrebno upotrebljavati odgođeno pridruživanje "**:=**" jer želimo da se funkcija izvrijednjava tek kad je pozovemo. Nadalje, primijetite simbol "_" na lijevoj strani, tzv. *blank*. Izraz "**x_**" je tzv. *uzorak* (*pattern*).

x	jedinstveni simbol x
x_	bilo koji izraz, privremeno (za vrijeme izvrijednjavanja funkcije) zvan x

Da bolje uočimo razliku, pogledajmo "funkciju" definiranu bez blanka.

```
g[x] := x^2

g[x]

x^2

g[w]

g[w]
```

Ona djeluje samo na argument koji je upravo x i na nijedan drugi, što nikako nije željeno ponašanje za jednu funkciju.

Poželjno je sad izbrisati ovu lošu definiciju iz memorije, pomoću funkcije **Clear**, jer inače možemo naletiti na probleme poput ovih:

```
g[a_] := 3 a

g[2]
g[w]
g[x]

6

3 w

x^2
```

Vidimo da sad funkcija radi dobro za sve argumente *osim za* **x**. To je zbog toga jer je još uvijek važeće gornje specifično pravilo za **g[x]**, a *Mathematica* uvijek primjenjuje prvo specifična pravila, a opća tek onda ako nema primjenjivih specifičnih pravila.

```
?g

Global`g

g[x] := x^2

g[a_] := 3 a

Clear[g]
g[a_] := 3 a
g[x]

3 x
```

Sad je sve uredu.

Naravno, funkcija može biti i od više varijabli:

```
f[x_, a_] := x^a

f[4, 2]

16
```

```
f[Range[4], 3]
```

```
{1, 8, 27, 64}
```

Ovo radi zahvaljujući listabilnosti potenciranja. Naravno, funkcije mogu imati i složenije objekte kao argumente, poput liste u slijedećem primjeru:

```
g[list_, n_] := n + Length[list]

g[{a, b, c}, 1]
```

```
4
```

```
g[2, 1]
```

```
1
```

(*Broj* 2 nije lista, pa mu je duljina 0.)

Zapravo, bilo što može biti argument funkcije. Najmoćnija stvar, obilato korištena u funkcionalnom pristupu programiranju (odjeljak 4.2), je da i funkcije mogu biti argumenti funkcija (kompozicija funkcija).

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

h[Sin, 2]
h[Log, 0.1]
```

```
Sin[Sin[2]]
```

```
0.834032 + 3.14159 i
```

```
intder[f_, x_] := D[Integrate[f, x], x]

intder[x Log[Tan[x]], x]
```

$$\frac{1}{2}\left(\frac{1}{4}\left(-i+\frac{2\,i\,e^{2\,i\,(\frac{\pi}{2}-x)}}{1+e^{2\,i\,(\frac{\pi}{2}-x)}}\right)\pi^2-\pi\left(-i\left(\frac{\pi}{2}-x\right)+\frac{2\,i\,e^{2\,i\,(\frac{\pi}{2}-x)}\,(\frac{\pi}{2}-x)}{1+e^{2\,i\,(\frac{\pi}{2}-x)}}\right)-i\left(\frac{\pi}{2}-x\right)^2+\frac{2\,i\,e^{2\,i\,(\frac{\pi}{2}-x)}\,(\frac{\pi}{2}-x)^2}{1+e^{2\,i\,(\frac{\pi}{2}-x)}}\right)+\frac{1}{2}\left(-i\,x^2+\frac{2\,i\,e^{2\,i\,x}\,x^2}{1+e^{2\,i\,x}}\right)+x\,\text{Log}[\text{Tan}[x]]+\frac{1}{2}\,x^2\,\text{Csc}[x]\,\text{Sec}[x]$$

```
Simplify[%]
```

```
x Log[Tan[x]]
```

Moguće je i ograničiti domenu neke funkcije pomoću funkcije **`Condition`**, koja se obično zapisuje u skraćenom *infiks* obliku "**`/;`**" (čitaj npr. "uz uvjet"):

```
p[x_ /; x > 0] := x^3
```

"p(x), uz uvjet da je x veći od nula, je x^3."

```
p[3]
```

```
27
```

```
p[-3]
```

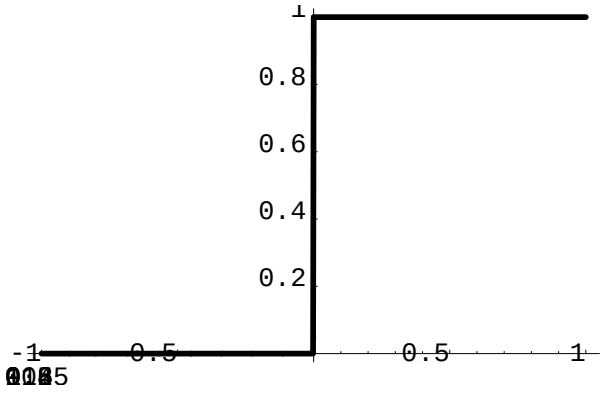
```
p[-3]
```

Tako možemo definirati funkciju po dijelovima

```
step[x_ /; x ≥ 0] := 1
step[x_] := 0  /; x < 0
```

Za negativni dio domene smo, demonstracije radi, upotrijebili alternativnu poziciju operatora "**/;**". U starijim verzijama *Mathematice* samo je taj način bio dozvoljen.

```
Plot[step[x], {x, -1, 1}, PlotStyle → Thickness[0.01]]
```




```
- Graphics -
```

Pripazite! Takve funkcije mogu imati problematično numeričko ponašanje oko točaka gdje su nederivabilne.

```
delta = step'
```

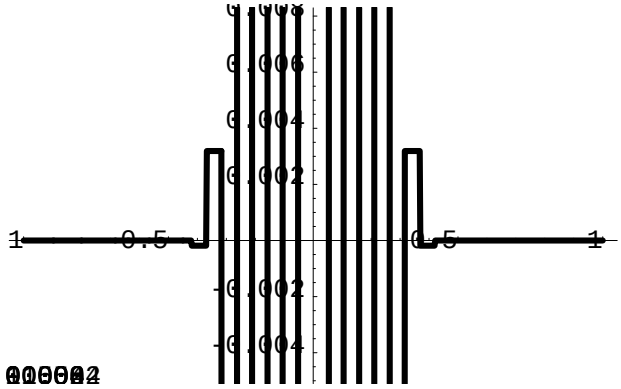
```
step′
```

```
delta[1] // N
delta[0.1] // N
```

```
-1.7133 × 10^-16
```

```
-4.29432
```

```
Plot[delta[x], {x, -1, 1}, PlotStyle → Thickness[0.01]]
```




```
- Graphics -
```

Ove nepostojeće oscilacije su artefakt numeričkih metoda koje *Mathematica* koristi prilikom crtanja.

Zadatak: Ugrađena funkcija **`UnitStep`** odgovara ovoj funkciji **`step`**. Uvjerite se da je njeno ponašanje oko nule bolje. Kojoj je (ugrađenoj) funkciji jednaka njena derivacija **`UnitStep`**'? Ta funkcija i **`UnitStep`** su dvije rijetke nederivabilne funkcije koje se relativno često koriste u fizici pa je stoga u *Mathematici* posvećena dodatna pažnja njihovoj kvalitetnoj definiciji koja ne pati od gornjih problema.

Zadatak: Definirajte funkciju **`argand[list_]`** koja crta kompleksnu ravninu s točkama zadanim u listi kompleksnih brojeva **`list`** = {$z_1, z_2, ...$}. (Treba vam funkcija **`ListPlot`**.) Upotrijebite funkciju **`argand`** da nacrtate na jednom dijagramu svih devet devetih korijena od 1.

Naputak: Prvo definirajte funkciju **`plotpoint[z_]`** koja crta samo jednu točku.

Zadatak: Definirajte funkciju **`kettle[masa_, donjatemp_, gornjatemp_, korak_]`** koja crta tablicu oblika

`7 Celsius`	`29.4 Joule`	2.94×10^8 `Erg`
`7.3 Celsius`	`30.66 Joule`	3.066×10^8 `Erg`
`7.6 Celsius`	`31.92 Joule`	3.192×10^8 `Erg`

gdje su prvi stupac temperature izmedju **`donjatemp`** i **`gornjatemp`** (u razmacima **`korak`**), drugi stupac su energije u SI sustavu koje trebaju da se **`masa`** vode zagrije od 0° C do te temperature, a treći stupac su te iste energije, ali u CGS sustavu jedinica.

Naputak:
1. Prisjetite se da je energija potrebna da se temperatura tvari mase *m* i specifičnog toplinskog kapaciteta *c* promijeni za ΔT jednaka E = m c ΔT. Toplinski kapacitet vode je 4200 $J\,kg^{-1}\,K^{-1}$.
2. Upoznajte se s paketom **`Miscellaneous'Units'`** i funkcijama **`SI`**, **`CGS`** koje rade konverziju između sustava jedinica.
3. Definirajte funkciju **`energija1`** koja za zadanu masu i temperaturu daje potrebnu energiju. Neka funkcija prima argumente s fizikalnom dimenzijom, poput **`energija1[0.01 Kilogram, 5.3 Celsius]`**.
4. Definirajte funkciju **`energija2`** koja za zadanu masu i temperaturu daje cijeli redak tablice
5. Koristeć i funkciju **`energija2`** definirajte traženu funkciju **`kettle`**.

Domaći zadatak 3−2 (Jacobijeva matrica)

Za prijelaz iz jednog sustava varijabli $\{x_1, x_2, \cdots\}$ u drugi $\{y_1(x_1, x_2, \cdots),\ y_2(x_1, x_2, \cdots),\ \cdots\}$ često se rabi tzv. Jacobijeva matrica transformacija

$$J=\begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \cdots \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} & \cdots \\ \vdots & \vdots & \ddots \end{pmatrix}$$

(1) Definirajte funkciju **`jacobiMatrix[ylist_, xlist_]`** koja izračunava Jacobijevu matricu.
(2) Izračunajte Jacobijevu matricu **`jac`** transformacija iz polarnog u kartezijev sustav x=r cos(θ), y=r sin(θ).
(3) Odredite (ručno) inverzne transformacije iz kartezijevog u polarni sustav i izračunajte odgovarajuću Jacobijevu matricu **`invjac`**.
(4) Provjerite da je umnožak matrica iz (2) i (3) jednak jediničnoj matrici.
(5) Ako podzadatak (3) pokušate riješiti tako da pomoću *Mathematice* invertirate jednadžbe x=r cos(θ), y=r sin(θ) dobit ćete četiri rješenja. Izračunajte sve te četiri moguće Jacobijeve matrice **`invjac[n], n=1,2,3,4`** i pokažite da su sve one inverzne originalnoj matrici, nakon što se obavi odgovarajuća zamjena varijabli.

Tu je za pojednostavljenje nekih izraza potrebno je *Mathematici* objasniti da je r>0, kako bi ona znala da je npr. $\sqrt{r^2} = r$. Za to je korisna funkcija **`Assuming`**.

■ Domaći zadatak 3−3 (relativistička kinetička energija)

Formule za nerelativističku i relativističku kinetičku energiju tijela mase *m* koje se giba brzinom *v* su

$$K_{\text{nr}} = \frac{1}{2} m v^2 \qquad K_{\text{rel}} = m c^2 \left[\frac{1}{\sqrt{1-\frac{v^2}{c^2}}} - 1 \right]$$

(a) Definirajte odgovarajuće funkcije **knr** i **krel** te usporedite na istom dijagramu ponašanje tih funkcija za brzine od 0 do 3×10^8 m/s za neku vrijednost mase. Označite koordinatne osi!
(b) Odredite brzinu pri kojoj je greška nerelativističke formule tisućinku promila.

■ 3.3 Izrazi (*expressions*)

■ Sve su samo izrazi

Sve objekte s kojima smo se dosad sreli, poput funkcija, lista, polinoma, jednadžbi, dijagrama itd., *Mathematica* interno tretira kao objekte iste vrste, tzv. izraze (*expressions*).

```
FullForm[x^2 + 1 == Sin[x]]
FullForm[b → g]
FullForm[Plot[x, {x, 0, 1}, PlotPoints → 5, DisplayFunction → Identity]]
```

```
Equal[Plus[1, Power[x, 2]], Sin[x]]

Rule[b, g]

Graphics[List[List[Line[
    List[List[2.5'*^-7, 2.5'*^-7], List[0.2434019494374947', 0.2434019494374947']],
     List[0.5088527991562422', 0.5088527991562422']],
     List[0.7581561980312317', 0.7581561980312317'], List[0.9979104466249685',
      0.9979104466249685'], List[0.99999975', 0.99999975']]]]],
 List[Rule[PlotRange, Automatic], Rule[AspectRatio, Power[GoldenRatio, -1]],
  RuleDelayed[DisplayFunction, Identity],
  Rule[ColorOutput, Automatic], Rule[Axes, Automatic],
  Rule[AxesOrigin, Automatic], Rule[PlotLabel, None],
  Rule[AxesLabel, None], Rule[Ticks, Automatic],
  Rule[GridLines, None], Rule[Prolog, List[]],
  Rule[Epilog, List[]], Rule[AxesStyle, Automatic],
  Rule[Background, Automatic], Rule[DefaultColor, Automatic],
  RuleDelayed[DefaultFont, $DefaultFont], Rule[RotateLabel, True],
  Rule[Frame, False], Rule[FrameStyle, Automatic],
  Rule[FrameTicks, Automatic], Rule[FrameLabel, None],
  Rule[PlotRegion, Automatic], Rule[ImageSize, Automatic],
  RuleDelayed[TextStyle, $TextStyle], RuleDelayed[FormatType, $FormatType]]]
```

Vidimo da su sva ova tri objekta izgrađena na isti način, od uglatih zagrada, zareza i objekata poput brojeva ili simbola koji se u *Mathematici* zovu *atomi*. Mogući tipovi atoma su *brojevi* (3, 4.1, 3+2*i*), *simboli* (x, *β*, Plot, wf3c) i *stringovi* ("dvije lipe", "če=*ħ*").

Objasnite slijedeću poruku o grešci:

```
Union[{2, 3}, 4]
```

```
Union::normal : Nonatomic expression expected at position 2 in {2, 3}⋃4. More...

{2, 3}⋃4
```

Od atoma se izgrađuju objekti koji se zovu *izrazi*, a koji se formalno definiraju na rekurzivni način: Izrazi su ili atomi ili objekti oblika

$$i_0[i_1,\ i_2,\ \ldots]$$

gdje su i_n izrazi. Dakle, izraz može biti npr. `fja[][w[2,b],p[wrus[r,t]]]`

Da bi vidjeli interni *Mathematica* izraz koji odgovara nekom izrazu na ekranu, koristimo funkciju **`FullForm`**. Objekt i_0 u gornjoj definiciji se zove *glava* izraza ("*head*"). Moguće ga je ekstrahirati funkcijom **`Head`**.

```
FullForm[b /; x > 3]
```

```
Condition[b, Greater[x, 3]]
```

```
Head[b /; x > 3]
```

```
Condition
```

```
Head[1/2]
```

```
Rational
```

Mi razne tipove izraza interpretiramo na različit način, kako je prikazano u tabeli

Funkcija[argumenti]	`Log[x]`
Komanda[argumenti]	`Expand[(x + 1)^2]`
Operacija[operandi]	`Plus[2, 3]`

no to različito interpretiranje je više posljedica navike. U gornjem primjeru **`Log[x]`**, **`Log`** možemo smatrati i komandom ili pak operacijom koja se vrši na jednom operandu i čiji je rezultat logaritam tog operanda.

Premda svi izrazi interno imaju oblik *glava*[... *argumenti* ...], *Mathematica* često dopušta i druge načine zapisa izraza, u skladu sa standardnom matematičkom notacijom

`f[x, y]`	standardni oblik
`f @ x`	prefiksni oblik
`x // f`	postfiksni oblik
`x ~ f ~ y`	infiksni oblik

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

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

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

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

Pritom treba paziti na pravila o redoslijedu izvršavanja komandi.

```
Sin @ π + π
```

```
π
```

```
π + π // Sin
```

```
0
```

Ovaj primjer pokazuje da je **@** po redosljedu izvršenja ispred, a **//** iza zbrajanja. Najbolje je, ukoliko nismo sasvim sigurni u prioritete izvršavanja, igrati na sigurno i koristiti zagrade.

■ Dijelovi izraza

Pojedine dijelove izraza je moguće ekstrahirati slično kao kod lista (izrazi interno i jesu objekti vrlo slični listama)

```
izr = x^3 + (1 + x)^2
```

$x^3 + (1 + x)^2$

```
Drop[izr, -1]
```

x^3

```
Take[izr, -1]
```

$(1 + x)^2$

```
Append[izr, w^6]
```

$w^6 + x^3 + (1 + x)^2$

```
izr2 = x (1 - x);
Append[izr2, w^6]
```

$w^6 (1 - x) x$

U prvom od zadnja dva primjera glava od **izr** je **Plus**, a u drugom glava od **izr2** je **Times**, pa **Append** priključuje argument w^6 shodno tome.

Pažnja: gore prikazane operacije ne mijenjaju originalne izraze!

```
izr2
```

$(1 - x) x$

Dakle, ako želimo transformacijama mijenjati izraze, to se treba raditi npr. ovako:

```
izr2 = Append[izr2, w^6]
```

$w^6 (1 - x) x$

```
izr2
```

$w^6 (1 - x) x$

Recimo da sada u ovom izrazu želimo promijeniti $x \to y$, ali samo unutar zagrade. (Globalnu zamjenu bi bilo lako izvesti transformacijskim pravilom **izr2 /. x->y**). Ponovno od pomoći može biti funkcija **Position**

```
Position[izr2, x]
```

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

```
izr2[[2, 2, 2]] = y;
izr2
```

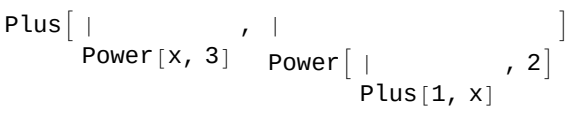
$w^6 x (1 - y)$

Koji rezultat funkcije **Position** odgovara kojem x−u iz izraza možemo odrediti metodom pokušaja i pogreške ili pomoću informacija iz slijedećeg odjeljka.No, često je mnogo lakše ovakve zamjene načiniti tako da naprosto *cut−and−paste*−amo izraze u novu input čeliju i izmjene napravimo ručno.

■ Nivoi izraza (*)

Vidimo da su izrazi "višedimenzionalni". Njihovu stabloliku strukturu je moguće i skicirati pomoću funkcije **TreeForm**

```
TreeForm[izr]
```



Razne operacije se mogu usredotočiti i samo na pojedine *nivoe* (*levels*) izraza.

n	nivoi od 1 do n
Infinity	svi nivoi (osim 0)
{n}	samo n – ti nivo
{n, m}	nivoi od n – tog do m – tog

```
Level[izr, {2}]
```

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

```
Level[izr, 2]
```

```
{x, 3, x^3, 1 + x, 2, (1 + x)^2}
```

```
Replace[izr, x → y, {2}]
```

```
(1 + x)^2 + y^3
```

Ovo ograničavanje na pojedine nivoe je korisno kod funkcionalnog programiranja gdje se time omogućuje fina kontrola područja djelovanja funkcija.

```
Sin[izr]
```

```
Sin[x^3 + (1 + x)^2]
```

```
Map[Sin, izr, {1}]
```

```
Sin[x^3] + Sin[(1 + x)^2]
```

```
Map[Sin, izr, {2}]
```

```
Sin[x]^Sin[3] + Sin[1 + x]^Sin[2]
```

```
Map[Sin, izr, {3}]
```

```
x^3 + (Sin[1] + Sin[x])^2
```

```
Map[Sin, izr, 3]
```

```
Sin[Sin[x]^Sin[3]] + Sin[Sin[Sin[1] + Sin[x]]^Sin[2]]
```

Zadatak: Zamijenite u izrazu

```
izr2 = Log[(Sin[a]^2 + Sin[b]) Sin[c] - Sin[d]]
```

```
Log[(Sin[a]^2 + Sin[b]) Sin[c] - Sin[d]]
```

sve sinuse osim **Sin[a]**, kosinusima.

■ **Domaći zadatak 3−4:** Objasnite riječima slijedeći izračun:

```
izr = x^3 + (1 + x)^2
```

```
x^3 + (1 + x)^2
```

```
Apply[Times, izr, 2]
```

```
5 x
```

■ 3.4 Uzorci (*patterns*)

Pojam *uzorka* (*pattern*) smo već upoznali u kontekstu definiranja funkcija. **f[x_]** je značilo da će funkcija **f** djelovati na bilo koji izraz. *Mathematica* radi sparivanje (uspoređivanje) s uzorkom (*pattern matching*) da bi odlučila da li da primijeni funkciju ili ne. **x_** se tako sparuje s bilo kojim izrazom, ali složeniji uzorci mogu biti selektivniji.

```
f2[x_^n_] := pwr[x, n]
```

```
f2[a^2]
```

```
pwr[a, 2]
```

```
f2[a]
```

```
f2[a]
```

Primijetite da se uspoređivanje s uzorkom obavlja u *strukturalnom*, a ne *matematičkom* smislu. Tako se u prethodnom primjeru **x_^n_** ne sparuje s **a**, premda je **a** matematički gledano isto oblika x^n, uz n=1.

```
f3[x_, x_] := "isti"
```

```
f3[3, 3] + f3[a, a] + f3[1, 1.0]
```

```
2 isti + f3[1, 1.]
```

Kod funkcija koje barataju elementima liste ili izraza, željene elemente možemo specificirati i pomoću uzoraka.

```
lst = {1, 3, 5^w, a, 1, c^2, d^w, 4};
```

```
Position[lst, x_^n_]
```

```
{{3}, {6}, {7}}
```

```
Cases[lst, x_^n_]
```

```
{5^w, c^2, d^w}
```

U ovim primjerima zapravo i nismo trebali davati imena uzorcima

```
DeleteCases[lst, _^_]
```

```
{1, 3, a, 1, 4}
```

Jedna od mogućnosti ograničavanja domene funkcija je specificiranje uzoraka **x_*head*** koji se sparuju samo s izrazima odgovarajuće glave (*head*)

```
gg[x_Integer] := 1
```

```
gg[3.]
```

```
gg[3.]
```

```
gg[3]
```

```
1
```

ili koji zadovoljavaju dodatne uvjete (to smo već vidjeli)

```
DeleteCases[lst, x_ /; x < 4]
```

```
{5^w, a, c^2, d^w, 4}
```

■ 3.5 Transformacijska pravila

Transformacijska pravila smo već sreli prilikom provjere rješenja jednadžbi, ali njihova je primjena univerzalna.

```
izr = a + b Log[x^2] + c x Sin[x];
```

```
izr /. {_^2 → sq, x → y, c → x}
```

```
a + b Log[sq] + x y Sin[y]
```

Primijetite da se pravila primjenjuju po redu kako su navedena i da u njima smijemo koristiti uzorke.

```
ff[a + ff[x]] + ff[a] /. ff[x_] → hh[x]
```

```
hh[a] + hh[a + ff[x]]
```

Uočavamo da je unutarnji **ff[x]** ostao netransformiran. To je stoga što simbol "**/.**" pravila primjenjuje samo jednom na svaki dio izraza. Postoji i simbol "**//.**" koja pravila primjenjuje iterativno sve dok ima ikakvih promjena izraza.

```
ff[a + ff[x]] + ff[a] //. ff[x_] → hh[x]
```

```
hh[a] + hh[a + hh[x]]
```

Ako transformacijsko pravilo involvira izvrijednjavanje neke funkcije to se izvrijednjavanje odvija čim je pravilo zadano (dakle prije nego se ono primijeni na izraz s lijeve strane simbola **/.**)

```
x + ff[x] /. x → Random[Integer, 100]
```

```
91 + ff[91]
```

Postoji i alternativa, tzv. *odgođeno* transformacijsko pravilo "**:>**" (*Mathematica* ga tipografski u bilježnici prikazuje kao "**⧴**") koje se izvrijednjuje tek prilikom upotrebe pravila:

```
x + ff[x] /. x ⧴ Random[Integer, 100]
```

```
13 + ff[47]
```

Vidimo da je razlika između "**->**" i "**:>**" za transformacijska pravila slična kao i razlika između "=" i ":=" za pridruživanja.

Naravno, i u transformacijskim pravilima na uzorke se mogu postavljati i dodatni uvjeti

```
Range[10] /. (x_ /; PrimeQ[x]) → 0
```

```
{1, 0, 0, 4, 0, 6, 0, 8, 9, 10}
```

Ukoliko na izraz primijenimo *listu listi* transformacijskih pravila, dobit ćemo *listu* transformiranih izraza. (To smo već koristili kod provjere rezultata funkcije **Solve**.)

```
x^2 /. {{x → 1}, {x → 2}}
```

```
{1, 4}
```

Može i

```
x^2 /. x → {1, 2}
```

```
{1, 4}
```

Zadatak: Definirajte funkciju **factors[n_, m_]** koja crta tablicu oblika

```
15      2
16      1
17      prim
18      2
```

gdje su lijevi stupac cijeli brojevi od n do m, a desni broj njihovih različitih faktora, odnosno string "prim" gdje treba.

Naputak:
1. Upoznajte se s funkcijom **FactorInteger** koja daje listu faktora s njihovim učestalostima.
2. Napravite transformacijsko pravilo koje radi pridruživanje $x \to \{x, \langle broj\ razl.\ faktora \rangle\}$, tj. pretvara cijeli broj u jedan redak traž ene tablice.
3. Trebate posebno pravilo za prim brojeve, koje ih pretvara u {x , *"prim"*}.
4. Primijenite pravila na listu brojeva **Range[6,8]** (za tablični ispis trebate funkciju **TableForm**).
5. Definirajte traž enu funkciju **factors.**

Domaći zadatak 3−5: Modificirajte funkciju **factors** tako da u desnom stupcu ne bude broj njihovih *različitih* faktora, već ukupan broj faktora, dakle

```
15      2
16      4
17      prim
18      3
```

(*) Slijedi napredniji primjer uporabe transformacijskih pravila. Neka je zadana je lista planetarnih podataka

```
planeti = {
   {"Merkur", 0.387, 0.24085},
   {"Venera", 0.7233, 0.61515},
   {"Zemlja", 1, 1},
   {"Mars", 1.5237, 1.8808},
   {"Jupiter", 5.2028, 11.867},
   {"Saturn", 9.5388, 29.461},
   {"Uran", 19.18, 84.013},
   {"Neptun", 30.0611, 164.793},
   {"Pluton", 39.44, 247.7}
  };
```

```
TableForm[planeti, TableHeadings →
  {Automatic, {"planet", "R-udaljenost od Sunca [AJ]", "T-period ophodnje [god]"}}]
```

	planet	R-udaljenost od Sunca [AJ]	T-period ophodnje [god]
1	Merkur	0.387	0.24085
2	Venera	0.7233	0.61515
3	Zemlja	1	1
4	Mars	1.5237	1.8808
5	Jupiter	5.2028	11.867
6	Saturn	9.5388	29.461
7	Uran	19.18	84.013
8	Neptun	30.0611	164.793
9	Pluton	39.44	247.7

Upotrebom transformacijskih pravila provjerit ćemo treći Keplerov zakon tj. da je R^3 / T^2 otprilike konstantno. Prvo kreiramo iz liste **planeti** dvije posebne liste s udaljenostima i periodima:

```
podaci = {imena, udaljenosti, periodi} = Transpose[planeti]
```

```
{{Merkur, Venera, Zemlja, Mars, Jupiter, Saturn, Uran, Neptun, Pluton},
 {0.387, 0.7233, 1, 1.5237, 5.2028, 9.5388, 19.18, 30.0611, 39.44},
 {0.24085, 0.61515, 1, 1.8808, 11.867, 29.461, 84.013, 164.793, 247.7}}
```

Zatim transformiramo izraz $\frac{R^3}{T^2}$ pomoću liste pravila te tako dobivamo listu omjera:

```
R^3/T^2 /. {R → udaljenosti, T → periodi}
```

```
{0.99917, 0.999985, 1, 1.00003, 1.00007, 0.999968, 0.999661, 1.00032, 0.999905}
```

Odredit ćemo još i odgovarajuću masu Sunca putem formule

$$M_{\text{Sunca}} = \frac{4\pi^2 R^3}{G T^2}$$

Za konverziju jedinica (AJ) i vrijednost Newtonove konstante G koristimo pakete **Miscellaneous'Units'** i **Miscellaneous'PhysicalConstants'.**

```
<< Miscellaneous'Units'
<< Miscellaneous'PhysicalConstants'

konstante = {
  G → GravitationalConstant,
  god → Convert[Year, Second],
  AJ -> Convert[AstronomicalUnit, Meter]
 }
```

$$\left\{G \to \frac{6.673\times 10^{-11}\ \text{Meter}^2\ \text{Newton}}{\text{Kilogram}^2},\ \text{god} \to 31536000\ \text{Second},\ \text{AJ} \to 1.49598\times 10^{11}\ \text{Meter}\right\}$$

```
mSun = 4 π^2 R^3/(G T^2) /. {R → udaljenosti * AJ, T → periodi * god} /. konstante /.
  Newton → Kilogram Meter / Second^2
```

```
{1.98995×10^30 Kilogram, 1.99157×10^30 Kilogram, 1.9916×10^30 Kilogram,
 1.99166×10^30 Kilogram, 1.99173×10^30 Kilogram, 1.99153×10^30 Kilogram,
 1.99092×10^30 Kilogram, 1.99223×10^30 Kilogram, 1.99141×10^30 Kilogram}
```

4. Programiranje u *Mathematici*

■ Uvod

Isti matematički problem je često moguće riješiti na različite načine, putem algoritama iza kojih stoje sasvim različiti načini razmišljanja. Razložimo to na primjeru funkcije *faktorijel*.

```
7!
```

```
5040
```

Klasični način programiranja, upotrebljavan od dana prvih računala, je tzv. *proceduralno* (ili *imperativno*) programiranje kod kojeg na program gledamo kao na niz naredbi koje obično postupno mijenjaju vrijednosti nekih varijabli pohranjenih u memoriji računala:

```
faktorijelProc[n_] := Module[{i = 1, fac = 1},
   While[i < n,
    i = i + 1; fac = fac * i];
   fac];

faktorijelProc[7]
```

```
5040
```

Zanemarite zasad precizno značenje komande **`Module`** koju ćemo objasniti kasnije i preciznu sintaksu komande **`While`**. Ovdje je očito riječ o standardnom algoritmu kakvog bismo isprogramirali u bilo kojem proceduralnom jeziku poput Fortrana ili C−a: Imamo petlju koja se prolazi *n* puta i svaki puta množimo rezultat prošlog prolaza sa za 1 većim brojem.

Međutim, *Mathematica* nam omogućuje i drugačije pristupe. Tako je kod *funkcionalnog* programiranja naglasak na izvrijednjavanju funkcija tj. primjeni funkcija na izraze. Faktorijel prirodno zamišljamo kao funkciju koja daje "*umnožak svih brojeva od 1 do zadanog broja":*

```
faktorijelFunkc[n_] := Apply[Times, Range[n]]

faktorijelFunkc[7]
```

```
5040
```

(Primijetite da nam ovdje nije trebala pomoćna varijabla (poput **`i`** iz **`faktorijelProc`**), ali nam je trebalo više memorije jer smo trebali pohraniti čitavu listu {1, 2, …, n} koju daje **`Range[n]`**.

Programiranje *transformacijskim pravilima* u *Mathematici* je svojevrsna nadgradnja funkcionalnog, gdje funkcije promatramo kao skupove pravila za transformaciju svojih argumenata. Uz ime funkcije je vezan niz pravila koja čine da se funkcija ponaša kao složeno transformacijsko pravilo. Tu smo filozofiju u prošlom poglavlju primijenili za definiciju funkcije **`step`**. Faktorijel možemo unutar te paradigme definirati kao skup od dva pravila: jedno koje transformira n→ n (n−1)! i koje *Mathematica* primjenjuje rekurzivno, te jedno trivijalno koje transformira 1 → 1 koje "zaustavlja" rekurziju

```
faktorijelTrans[1] := 1
faktorijelTrans[n_] := n faktorijelTrans[n - 1]

faktorijelTrans[7]
```

```
5040
```

Primjetite da se ovdje koristi svojstvo da *Mathematica* uvijek prvo izvrijednjava specifična pravila pa tek onda prelazi na opća.

U ovom konkretnom slučaju, funkcionalni program se čini najelegantniji od sva tri. No, promotrimo funkciju koja izračunava n−ti član Fibonaccijevog niza (niz kod kojeg je svaki član definiran kao zbroj prethodna dva, a javlja se pri analizi idealizirane populacije zečeva). Ovdje se kao najelegantniji pokazuje pristup putem transformacijskih pravila:

```
fibTrans[0] := 0
fibTrans[1] := 1
fibTrans[n_] := fibTrans[n – 1] + fibTrans[n – 2]

fibTrans[7]

13
```

U posljednje vrijeme je vrlo popularno i tzv. *objektno* (*object oriented*) programiranje. *Mathematica* nije sasvim prilagođena istom pa ga nećemo razmatrati. (Na internetu se može naći dodatni paket `Class.m` koji omogućuje neku vrstu objektnog programiranja u *Mathematici.*)

■ 4.1 Proceduralno programiranje

Osnovna paradigma proceduralnog programiranja je da se kreće od nekog definiranog *stanja* programa (vrijednosti varijabli i nizova varijabli u memoriji računala). To stanje se specificira početnim naredbama pridruživanja (x=0, i=1) (*assignment*) i onda se algoritam izvodi primjenom raznih grananja, petlji i potprograma koji svi mijenjaju to stanje i vode na konačno stanje u kojem vrijednosti nekih varijabli odgovaraju rješenju problema.

■ If−then grananje

Grananje se u većini proceduralnih jezika izvodi nekim oblikom *if−then−else* petlje. Tu mogućnost nudi i *Mathematica* putem funkcije **`If`**

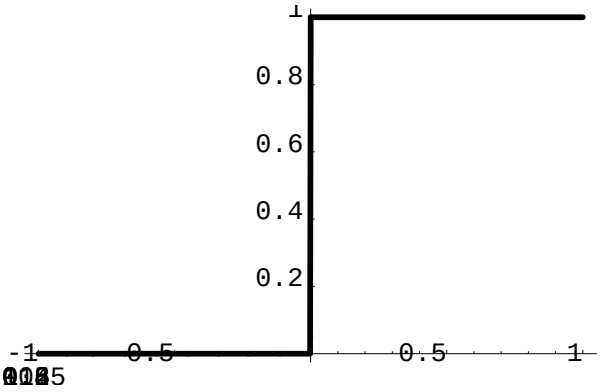
`If[`*test*,
<komande koje se izvršavaju ako je test True ("then")>,
*<komande koje se izvršavanju ako je test False ("else")>***`]`**

Naglasimo da je **`If`** normalna *Mathematica funkcija*. Rezultat njenog izvrijednjavanja je rezultat zadnje u nizu komandi sadržanih u argumentu br. 2 ili 3 (koji od tih nizova će se izvršiti ovisi o testu).

Na primjer, funkcija step koju smo sreli u prošlom poglavlju može se realizirati i ovako

```
step[x_] := If[x < 0, 0, 1]
```

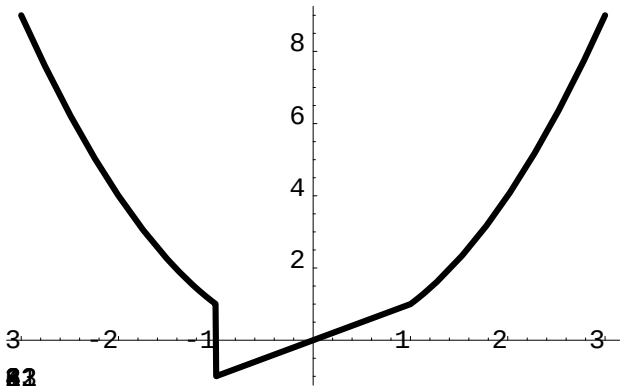
```
Plot[step[x], {x, -1, 1}, PlotStyle → Thickness[0.01]]
```




```
- Graphics -
```

Ako želimo konstrukciju s višestrukim grananjem *if−elif−elif−else*, onda moramo koristiti "ugniježđeni" (*nested*) **If**:

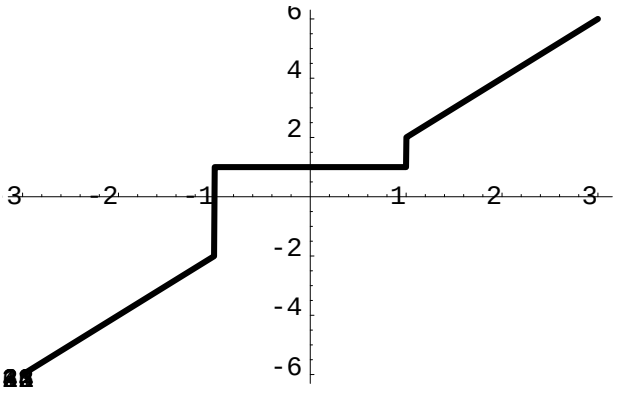
```
f[x_] := If[x < -1 , x^2, If[x > 1, x^2, x]]

Plot[f[x], {x, -3, 3}, PlotStyle → Thickness[0.01]]
```




```
- Graphics -
```

Spomenimo usput da ovako definiranu funkciju slobodno deriviramo.

```
Plot[f'[x], {x, -3, 3}, PlotStyle → Thickness[0.01]]
```




```
- Graphics -
```

Primijetite da smo gore mogli izbjeći ugnježđeni **If** korištenjem logičkih funkcija, danih u slijedećoj tablici (Četvrti stupac navodi niz tipaka koje daju znak u trećem stupcu. Ako zagrade ne kažu drugačije, prvenstvo izvršavanja je odozgo prema dolje.):

Not	!	¬	Esc – not – Esc
And	&&	∧	Esc – and – Esc
Or	||	∨	Esc – or – Esc

```
f[x_] := If[x < -1 || x > 1, x^2, x]
```

```
Or[1 > 3, -1 > 0, ¬ 5 > 10]
```

```
True
```

■ Petlje

Sve standardne vrste petlji, *do*−petlja, *while*−petlja i *for*−petlja, postoje i u *Mathematici.* Sintakse su

Do[*komande*, <*iterator*>**]**

gdje je <*iterator*> objašnjen u prošlom poglavlju.

While[*test*, *komande***]**

gdje se *komande* (niz *Mathematica* komandi odvojenih točka−zarezom "**;**") izvršavaju sve dok je god *test* **True**.

For[*start*, *test*, *incr*, *komande***]**

gdje se prvo izvrši start, a zatim se ponavlja izvršavanje *komandi* i *incr* sve dok je god *test* **True** (vrlo slično sintaksi for−petlje u C programskom jeziku).

Primjer upotrebe *while*−petlje smo vidjeli prilikom definiranja funkcije **faktorijelProc** u uvodu ovog poglavlja. Evo kratkih primjera za *do*− i *for*−petlje:

```
Do[2^n, {n, 0, 1, 0.5}]
```

Nema rezultata! Sve ove petlje, premda po sintaksi normalne *Mathematica* funkcije, imaju svojstvo po kojem se razlikuju od većine ugrađenih funkcija, a to je da ne vraćaju nikakav rezultat. Preciznije, vraćaju izraz **Null.** (Isti izraz vraćaju i one komande nakon kojih stavimo točka−zarez.)

```
FullForm[%]
```

```
Null
```

Ukoliko želimo neko ispisivanje rezultata koraka petlje, koristimo **Print**

```
Do[Print[2^n], {n, 0, 1, 0.5}]
```

```
1

1.41421

2.
```

Za iskakanje iz petlje možemo koristiti **Break** ili **Return**. Npr. slijedeći algoritam pronalazi prvi cijeli broj čiji rastav sadrži više od pet različitih prostih faktora.

```
For[i = 1,
 True,
 i = i + 1,
 If[Length[FactorInteger[i]] > 5, Break[],]]; i
```

```
30030
```

(Primijetite da **If** nema treći argument tj. da je on **Null**.) Ovo je relativno neelegantan program za pronalaženje najmanjeg broja koji ima više od pet različitih faktora. Bolje bi bilo

```
i = 1; While[Length[FactorInteger[i]] < 6, i = i + 1]; i
```

```
30030
```

a najbolje

```
Times @@ Table[Prime[n], {n, 6}]
```

```
30030
```

ali ovo zadnje spada već u funkcionalno programiranje.

■ Moduli

Jedan od problema s proceduralnim programiranjem je obilje popratnih efekata koje procedure izazivaju, poput pridjeljivanja vrijednosti pomoćnim varijablama. Na primjer, gornji račun je pridijelio vrijednost varijabli **i**

```
i
```

```
30030
```

što je zapravo nepotrebno i neželjeno, a često dovodi i do grešaka. Funkcionalno programiranje takve stvari obično automatski izbjegava (vidi zadnji primjer), dok se kod proceduralnog programiranja to može izbjeći korištenjem *modula* tj. funkcije **Module**

Module[{*x*, *y* = y_0, …}, *komande*]

koja specificira da su varijable *x, y,* … lokalne tj. da ne postoje izvan modula, te koja vraća kao rezultat rezultat posljednje iz niza *komandi*. Primijetite da se prilikom specificranja lokalnih varijabli one mogu i inicijalizirati na neku vrijednost. Primjenu modula smo vidjeli u uvodu, pri definiranju funkcije **faktorijelProc.**

```
a = 1
```

```
1
```

```
f[n_] := Module[{a}, a = (x + y)^n; Expand[a]]
```

```
f[3]
```

$x^3 + 3\,x^2\,y + 3\,x\,y^2 + y^3$

```
a
```

```
1
```

Varijabla **a** je zadržala svoju globalnu vrijednost bez obzira na to što joj je unutar modula pridjeljivana neka druga vrijednost.

```
Clear[a]
```

Zadatak: Isprogramirajte proceduralno funkciju **fibProc[n_]** koja računa n−ti Fibonaccijev broj.

■ **Domaći zadatak 4−1:** Isprogramirajte funkciju **`kredit[iznos_, kamata_, rok_]`** koja daje mjesečni anuitet i cijenu kredita za zadani iznos, kamatu i rok kredita u godinama. Cijena kredita je razlika između dobivenog i konačno otplačenog novca.

Nadalje, isprogramirajte funkciju **`kreditTablica[iznos_, kamata_, {start_,stop_,step_}]`** koja ispisuje tablicu za niz rokova kredita zadanih putem naznačenog iteratora. Stupci tablice neka budu: rok, anuitet, cijena.

Zadatak: Konstruirajte funkciju **`brown[n_]`** koja daje listu $\{\{x_0, y_0\}, \{x_1, y_1\}, \ldots, \{x_n, y_n\}\}$ pozicija čestice koja se giba u ravnini slučajnim gibanjem koje je definirano rekurzijom $x_i = x_{i-1} + rnd1$ i $y_i = y_{i-1} + rnd2$ gdje su *rnd1* i *rnd2* slučajni brojevi između −1 i 1. Nacrtajte odgovarajuću putanju čestice.

Naputak:
1. Programirajte proceduralno koristeći **`Module`**
2. Definirajte varijablu **`randomVec`** koja će pri svakoj upotrebi (pozivu) poprimati vrijednost novog slučajnog vektora (x, y). Za to će biti potrebno koristiti odgođeno pridruživanje vrijednosti toj varijabli.
3. Inicijalizirajte listu pozicija tako da sadrži (0,0) kao prvi vektor pozicije i putem petlje dodajte u svakom koraku toj listi novi slučajni vektor.
4. Za crtanje možete koristiti **`ListPlot`**

■ **Domaći zadatak 4−2:** Konstruirajte funkciju **`walk1D`** tako da simulira tzv. jednodimenzionalnog nasumičnog šetača. Svaki šetačev korak treba biti iste, jedinične duljine, ulijevo ili udesno. Dakle, umjesto vektora (*rnd1*, *rnd2*) iz gornjeg Brownovog gibanja trebamo slučajan odabir koraka +1 ili −1. Provjerite ispravnost tvrdnje da je očekivana udaljenost nasumičnog jednodimenzionalnog šetača od ishodišta nakon n koraka $\sqrt{n}$.

■ 4.2 Funkcionalno programiranje

Funkcionalno programiranje je stil programiranja koji stavlja naglasak na izvrijednjavanje izraza, a ne na sukcesivno izvršavanje komandi. Kod čistih funkcionalnih programa obično nema pomoćnih varijabli i nepotrebnih popratnih efekata izvrijednjavanja funkcija. U tom smislu funkcionalno programiranje je pomalo slično radu s tabličnim kalkulatorom (npr. MS Excel): redosljed izračunavanja čelija je nebitan tj. očekujemo automatsku konzistenciju. Isto, vrijednosti čelija su dane izrazima, a ne slijedovima komandi. (Više informacija o funkcionalnom programiranju nudi Functional Programming FAQ ili WWW stranice *Haskell* funkcionalnog jezika.)

Takav stil programiranja često rabi nekoliko specijalnih funkcija (mogli bismo ih zvati "meta−funkcije") čija je uloga upravljanje primjenivanjem *drugih funkcija* koje dolaze kao argumenti ovih meta−funkcija. Upoznajmo ih.

■ Map i Apply

Već smo usputno upoznali funkciju **`Map[`***funkcija*, *izraz*`]`** koja primjenjuje *funkciju* na sve pod−izraze prvog nivoa *izraza* (Vidi odjeljak *"Sve su izrazi"* iz prošlog poglavlja za objašnjenje nivoa izraza, ali trebalo bi biti jasno iz slijedećih primjera.). Tako ako je izraz lista, onda se funkcija primijenjuje na sve elemente liste.

```
Map[fun, {a, b, {c, d}}]
```

```
{fun[a], fun[b], fun[{c, d}]}
```

Opcionalni treći argument funkcije **`Map`** omogućuje specificiranje nivoa na koji se funkcija primjenjuje, ako ne želimo defaultni prvi nivo.

```
Map[fun, {a, b, {c, d}}, {2}]
```

```
{a, b, {fun[c], fun[d]}}
```

Zgodna je i *infiksna* forma ove funkcije koja se dobiva simbolom **/@**

```
fun /@ (a x^2 + b)
```

```
fun[b] + fun[a x^2]
```

MapAt[*funkcija, izraz, lista pozicija***]** precizno primjenjuje *funkciju* na pozicije u *izrazu* specificirane u *listi pozicija* (koju možemo dobiti i pomoću funkcije **Position**)

```
izr = a x^2 + Log[x] Sin[a^x];
MapAt[fun, izr, Position[izr, x]]
```

```
a fun[x]^2 + Log[fun[x]] Sin[a^fun[x]]
```

Ovo smo mogli jednostavnije izvesti ovako:

```
izr /. x → fun[x]
```

```
a fun[x]^2 + Log[fun[x]] Sin[a^fun[x]]
```

ali gornja metoda je nekad zgodnija, npr. kad se želimo ograničiti samo na **x**−eve iz nekih nivoa izraza.

Druga važna "meta−funkcija" je **Apply[***funkcija, izraz***]** koja zamjenjuje *glavu izraza* s *funkcijom*. Ona isto ima često korištenu *infiks* formu **@@**.

```
Apply[fun, {a, b, {c, d}}]
```

```
fun[a, b, {c, d}]
```

```
fun @@ izr
```

```
fun[a x^2, Log[x] Sin[a^x]]
```

Dakle, ako *izraze* promatramo kao u prošlom poglavlju tj. kao generičku formu $i_0[i_1, i_2, \ldots]$ onda **Map** djeluje na argumente $i_1, i_2, \ldots$, a **Apply** na glavu izraza i_0.

Naravno, ovdje možemo koristiti i funkcije koje smo sami definirali

```
faktorijelFunkc[4]
```

```
24
```

```
faktorijelFunkc /@ Range[6]
```

```
{1, 2, 6, 24, 120, 720}
```

Recimo da sad želimo ispisati tablicu s nizom prirodnih brojeva i njihovih faktorijela. Možemo prvo postupiti tako da prvo definiramo funkciju koja generira jedan red tablice i onda je pomoću **Apply** primijenimo na niz brojeva:

```
faklist[n_] := {n, n!}
faklist /@ Range[6] // TableForm
```

```
1   1
2   2
3   6
4   24
5   120
6   720
```

Međutim, baš kao i pomoćne varijable, tako ni pomoćne funkcije nisu u duhu funkcionalnog programiranja. Zato postoje tzv. *čiste funkcije* ("*pure functions*") koje su bezimene i definiraju se i upotrebljavaju na slijedeći način:

```
{#, #!} &[4]
```

```
{4, 24}
```

Dakle, "&" nakon izraza je samo oznaka da je ono što prethodi funkcionalni oblik funkcije, a "#" stoje na mjestu argumenata.

Tako gornju tablicu dobijemo ovako:

```
{#, #!} & /@ Range[6] // TableForm
```

```
1     1
2     2
3     6
4     24
5     120
6     720
```

Ako čista funkcija treba biti funkcija više argumenata, koristimo #1, #2, itd. Dakle, drugim riječima, ako nam negdje treba pomoćna funkcija koju bismo definirali kao:
aux[x1, x2, ...]:= (*izraz koji uključuje x*1, *x2, …***)**
Možemo na mjesta na kojem bismo je upotrijebili staviti čistu funkciju:
(*izraz koji uključuje #*1, *#2, …***)&**

■ Nest i Fold

Ove dvije funkcije služe repetitivnom primjenjivanju neke funkcije na dani argument.

Nest[*funkcija*, *argument*, *n*] primjenjuje *funkciju* na *argument* pa onda na dobiveni rezultat opet primjenjuje *funkciju* i tako *n* puta ("ugnježđivanje", *nesting*).

```
Nest[Log, x, 3]
```

```
Log[Log[Log[x]]]
```

```
(Sqrt[5] + 1)/2 // N
```

```
1.61803
```

Npr. tzv. *zlatni omjer* $\frac{\sqrt{5}+1}{2}$ se može izraziti i kao beskonačni razlomak

$$\text{GoldenRatio} = 1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{\ddots}}}} = 1.618034\ldots$$

Takav razlomak možemo izvrijedniti na slijedeći način:

```
Nest[(1 + 1/#) &, x, 5]
```

$$1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{1 + \frac{1}{x}}}}}$$

gdje smo upotrijebili čistu funkciju koja odgovara funkciji $f(x) = 1 + \frac{1}{x}$.

```
% /. x → 1 // N
```

```
1.625
```

To isto, ali elegantnije:

```
Nest[(1 + 1/#) &, 1, 20] // N
```

```
1.61803
```

Funkcija **NestList[***funkcija*, *argument*, *n*] nam daje listu sa rezultatima primjene *funkcije* na *argument* i to od nula do *n* puta:

```
NestList[fun, x, 3]
```

```
{x, fun[x], fun[fun[x]], fun[fun[fun[x]]]}
```

```
NestList[(1 + 1/#) &, 1, 8] // N
```

```
{1., 2., 1.5, 1.66667, 1.6, 1.625, 1.61538, 1.61905, 1.61765}
```

Funkcija **FixedPoint** je slična **Nest** — ona isto "ugnježđuje" funkciju, ali ne neki zadani broj puta već sve dok nema promjene rezultata (do na određenu točnost koja se može specificirati).

```
FixedPoint[(1 + 1/#) &, 1] // N
```

```
$Aborted
```

Ovo se vrti beskonačno! Probajmo ovako:

```
FixedPoint[(1 + 1/#) & // N, 1]
```

```
1.61803
```

Zadatak: U čemu je bio problem tj. zašto se prva verzija vrtila beskonačno?

(Inaće, test koji **FixedPoint** provodi da bi provjerio da nema promjene rezultata (po defaultu je to ugrađeni predikat **SameQ**) se može i posebno specificirati putem opcije **SameTest**. Vidi donji program **mandelbrot**.)

Funkcija **Fold[***fun*, a_0, $\{a_1, a_2, \cdots, a_n\}$**]**, gdje je *fun* funkcija od dva argumenta, vraća fun[[⋯fun[fun[a_0, a_1], a_2]⋯]].

```
Fold[fun, a, {b, c, d}]
```

```
fun[fun[fun[a, b], c], d]
```

Npr. faktorijel možemo implementirati i ovako

```
faktorijelFunkc2[n_] := Fold[Times, 1, Range[n]]
```

```
faktorijelFunkc2[7]
```

```
5040
```

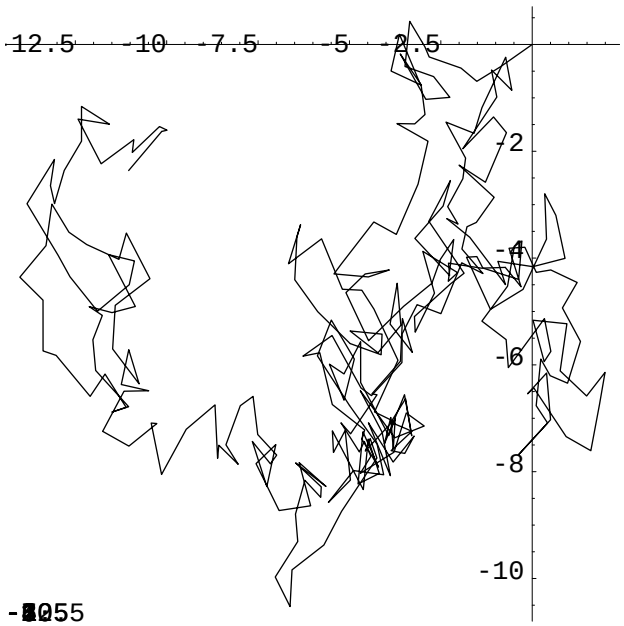
Ovdje imamo isto i funkciju **FoldList** koja nam daje listu međukoraka

```
FoldList[Times, 1, Range[7]]
```

```
{1, 1, 2, 6, 24, 120, 720, 5040}
```

Ove se funkcije na prvi pogled čine egzotične, ali često se mogu korisno primijeniti u funkcionalnom programiranju. Npr. funkcija **brown** koja daje listu pozicija čestice koja se giba nasumičnim Brownovim gibanjem se može implementirati i ovako

```
randomVec := {Random[Real, {-1, 1}], Random[Real, {-1, 1}]}
brown[n_] := FoldList[(#1 + #2) &, {0, 0}, Table[randomVec, {n}]]

ListPlot[brown[300], PlotJoined → True, AxesOrigin → {0, 0}, AspectRatio → 1]
```




```
- Graphics -
```

Primijetite da je program elegantniji od proceduralnog. Nema pomoćnih varijabli, izuzev **randomVec** kojeg bismo se isto mogli riješiti tako da njegovu definiciju stavimo u definiciju funkcije **brown** i tako dobijemo *one−liner*. S druge strane, program je nešto neoptimalniji od proceduralnog jer koristi više memorije—trebaju mu dvije liste veličine *n*: jedna da drži *n* vektora pomaka **randomVec**, a druga da drži pozicije čestice. Proceduralni program smo izveli samo s jednom listom.

■ Rekurzivno definiranje funkcije (*)

Jedna od značajki funkcionalnog programiranja je i upotreba rekurzivne definicije funkcija. Već smo je sreli u prošlom poglavlju pri definiciji faktorijela i Fibonaccijevih brojeva. Evo još jednog simpatičnog primjera, enormno brzorastuće Ackermannove funkcije (važne u matematičkoj teoriji složenosti) definirane putem dvostruke rekurzije:

```
ackermann[0, n_] := n + 1
ackermann[m_, 0] := ackermann[m - 1, 1]
ackermann[m_, n_] := ackermann[m - 1, ackermann[m, n - 1]]
$RecursionLimit = 10^4;
$IterationLimit = 10^8;

{ackermann[2, 0], ackermann[2, 1], ackermann[2, 2]}
```

```
{3, 5, 7}
```

```
{ackermann[3, 0], ackermann[3, 1], ackermann[3, 2]}
```

```
{5, 13, 29}
```

Pažnja, slijedeći red zamrzne moj notebook (da se to ne bi dogodilo, slijedeća ćelija nije izvršna: u izborniku `Cell->Cell Properties->Cell Evaluatable` je isključeno).

```
{ackermann[4, 0], ackermann[4, 1]}
```

```
{13, ??}
```

ackermann(4,1)=65533, a ackermann(4,2) je 19729−znamenkasti broj $2^{65536} - 3$ (http://www.kosara.net/thoughts/ackermann42.html). Ti brojevi još nisu sami po sebi preogromni za *Mathematicu*, ali potreban broj rekurzija da se do njih dođe je prevelik. S druge strane ackermann(5,2) se ne da zapisati u decimalnom zapisu, čak i u računalu napravljenom od svih čestica našeg svemira.

■ Primjeri razvoja funkcionalnih programa

Promotrimo razvoj programa koji ispisuje tablicu frekvencija pojavljivanja elemenata u listi. Načinimo prvo jednostavnu testnu listu

```
lst = {a, b, a, c, a, d, b, c, c};
```

Funkcija koja broji broj pojavljivanja je **Count**[*lista, element*]

```
Count[lst, c]
```

```
3
```

Da bismo ispisivali tablicu trebamo listu oblika {{element1, frekvencija elementa1}, {element2, frekvencija elementa2},...}. Element te liste (red tablice) se može dobiti ovako:

```
el[x_] := {x, Count[lst, x]};
el[c]
```

```
{c, 3}
```

Tablicu frekvencija za različite elemente dobijemo korištenjem čiste funkcije analogne **el** i njenom primjenom pomoću funkcije **Map** na popis elemenata

```
Map[{#, Count[lst, #]} &, {a, b, c, d}] // TableForm
```

```
a   3
b   2
c   3
d   1
```

To je sad to, jedino još želimo izbjeći da sami moramo ručno identificirati koji se sve elementi pojavljuju u listi. To nam može raditi funkcija **Union**.

```
Union[lst]
```

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

Dakle, imamo

```
Map[{#, Count[lst, #]} &, Union[lst]]
```

```
{{a, 3}, {b, 2}, {c, 3}, {d, 1}}
```

Kao zadnju stvar, poželjno je listu sortirati po frekvencijama. Funkcija **Sort** je već definirana tako da zna sortirati liste različitih objekata, no ovdje bi po defaultu sortirala parove po prvom elementu i to po abecednom redu, a nama treba sortiranje po drugom elementu i to po veličini. Stoga moramo kao drugi argument funkciji **Sort** dati sortirajuću funkciju koju ćemo sami definirati. Sortirajuća funkcija je *predikat* koji prima dva argumenta i vraća **True** ili **False** ovisno o tome da li prvi argument treba biti sortiran ispred ili iza drugog.

Razmislimo kako definirati funkciju koja kao argument prima dva para i vraća **True** ili **False** ovisno o tome da li je drugi element prvog para veći od drugog elementa drugog para ili ne. Može ovako:

```
frekventnijiQ[par1_, par2_] := par1[[2]] > par2[[2]]

Sort[Map[{#, Count[lst, #]} &, Union[lst]] , frekventnijiQ]

{{c, 3}, {a, 3}, {b, 2}, {d, 1}}
```

Naravno, elegantnije je to izvesti pomoću čiste funkcije:

```
Sort[Map[{#, Count[lst, #]} &, Union[lst]] , #1[[2]] > #2[[2]] &]

{{c, 3}, {a, 3}, {b, 2}, {d, 1}}
```

I to je to. Sad definiramo kompletnu funkciju:

```
frekvencije[l_] := Sort[Map[{#, Count[l, #]} &, Union[l]] , #1[[2]] > #2[[2]] &]

frekvencije[lst]

{{c, 3}, {a, 3}, {b, 2}, {d, 1}}
```

Primijenimo to na frekvenciju pojavljivanja slova u Shakespeareovom *Mletačkom trgovcu* (zapravo koristimo engleski original *The Merchant of Venice*, skinut sa WWW stranica projekta Gutenberg)

```
venice = ReadList["files/venice.txt", Character];

Length[venice]  (* ukupni broj slova *)

121057

Take[frekvencije[venice], 10] // TableForm

      20958
e     12267
o     7277
t     7241
a     6244
h     5570
n     5534
s     5326
r     5241
i     5215
```

Vidimo da je "e" daleko najčešće slovo (poslije razmaka), činjenica vrlo važna pri dešifriranju engleskih tekstova.

Kao slijedeći primjer, isprogramirat ćemo funkciju **mandelbrot** koja crta fraktal poznat pod imenom *Mandelbrotov skup*. Ovaj skup je definiran kao skup svih točaka *c* kompleksne ravnine za koje iteracija

$$z_0 = 0$$
$$z_{n+1} = z_n^{\,2} + c$$

ne divergira.

Promotrimo najprije prvih *n* iteracija za neki proizvoljni kompleksni broj, koje možemo dobiti funkcijom **NestList** kojom iterativno *n* puta primjenimo funkciju $f(z) = z^2 + c$ na $z_0 = c$ kao početnu vrijednost (čime samo štedimo jedan korak u odnosu na $z_0 = 0$ kao početnu vrijednost)

```
iter[c_, n_] := NestList[#^2 + c &, c, n]

iter[1 + i, 5]

{1 + i, 1 + 3 i, -7 + 7 i, 1 - 97 i, -9407 - 193 i, 88454401 + 3631103 i}
```

Pojavljivanje divergencije možemo uočiti gledajući apsolutne vrijednosti (udaljenost od ishodišta kompleksne ravnine)

```
iter[c_, n_] := Abs /@ NestList[#^2 + c &, c, n] // N

iter[1 + i, 5]

{1.41421, 3.16228, 9.89949, 97.0052, 9408.98, 8.85289 × 10^7}
```

Uočite gore primjenu funkcije **Map** za distribuciju funkcije **Abs** preko liste. Alternativno se može iskoristiti činjenica da je **Abs** *listabilna* funkcija i samo definirati:

```
iter[c_, n_] := Abs[NestList[#^2 + c &, c, n]] // N
```

U svakom slučaju, očito je da ovo divergira i da 1+i ne pripada Mandelbrotovom skupu. Pokušajmo s nekim drugim brojem:

```
iter[-1 + 0.2 i, 50]

{1.0198, 0.203961, 1.06063, 0.250592, 1.07668, 0.211683, 1.0413, 0.142667, 1.03646,
 0.190504, 1.05536, 0.216603, 1.06094, 0.195857, 1.04727, 0.171989, 1.04464,
 0.189194, 1.05275, 0.201417, 1.05484, 0.192042, 1.04938, 0.182988, 1.04821,
 0.189652, 1.05159, 0.194935, 1.05241, 0.190943, 1.0502, 0.187354, 1.0497, 0.19002,
 1.0511, 0.192232, 1.05143, 0.190571, 1.05053, 0.189121, 1.05032, 0.190203, 1.0509,
 0.191116, 1.05103, 0.190432, 1.05066, 0.189841, 1.05058, 0.190282, 1.05081}
```

Izgleda da −1+0.2i, ne divergira tj. da pripada skupu.

Kao nekakav prvi test divergencije možemo uzeti da ako apsolutna vrijednost ne pređe 2 nakon 50 koraka, vjerojatno nikad ni neće. (Matematičar Fatou je metodama kompleksne analize pokazao da kad jednom prođemo 2, divergiranje je neminovno.) Za jedan mali broj točaka sasvim blizu granice Mandelbrotovog skupa koje prelaze 2 tek nakon više od 50 koraka ovaj test će biti pogrešan, ali to možemo profiniti naknadno, povećavajući broj koraka.

Definiramo odgovarajući predikat

```
divergiraQ[c_] := Last[iter[c, 50]] > 2

divergiraQ[-1 + 0.2 i]

False

divergiraQ[1 + i]

$Aborted
```

Problem je da je 50 iteracija jako puno i brojevi mogu postati enormni i blokirati računalo (vidi gore 1+i već nakon 5 iteracija). Bolje bi bilo da prekinemo iteriranje čim aposlutna vrijednost pređe 2 i proglasimo točku divergentnom. Definirajmo novi predikat **divergiraQ2** u skladu s tim. Također ćemo kao vrijednosti ovog predikata umjesto **True** i **False** uzeti 1 i 0 radi lakšeg grafičkog prikaza.

```
divergiraQ2[c_] := If[For[z = c; i = 1, i < 50 && Abs[z] < 2, i++, z = z^2 + c]; i == 50, 0, 1]
```

(Razložite si algoritam ove definicije!)

```
divergiraQ2[1 + i]
```

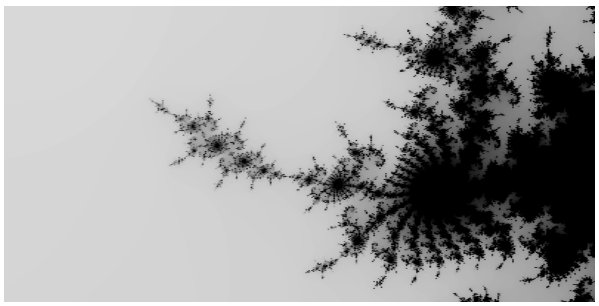
```
1
```

```
DensityPlot[divergiraQ2[x + i y], {x, -2.01, 0.7},
 {y, -1.1, 1.1}, PlotPoints → 200, Mesh → False, Frame → False]
```

```
- DensityGraphics -
```

Ovo je ispao zapravo proceduralni program. Ako ipak želimo upražnjavati čisto funkcionalno programiranje, onda možemo definirati funkciju ovako (primijetite da nema pomoćnih funkcija ni varijabli):

```
mandelbrot[xmin_, xmax_, ymin_, ymax_, n_] := DensityPlot[
  -Length[FixedPointList[#^2 + (x + I y) &, x + I y, 200, SameTest → (Abs[#2] > 2.0 &)]],
  {x, xmin, xmax}, {y, ymin, ymax}, PlotPoints → n,
  Mesh → False, Frame → False, AspectRatio → Automatic]

mandelbrot[-1.257, -1.247, 0.044, 0.049, 400]
```

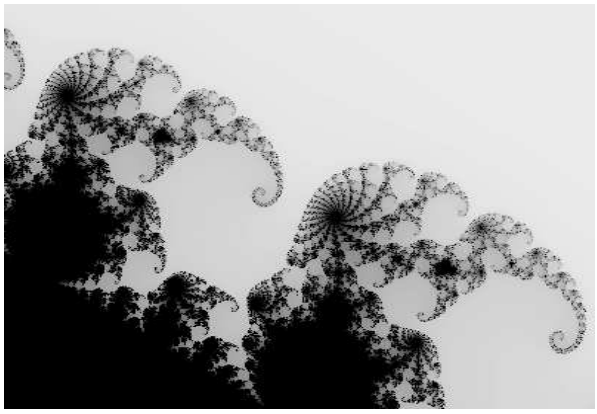


```
- DensityGraphics -
```

(Prikazan je dio Mandelbrotovog skupa poznat pod imenom "dolina morskog konjića" (*seahorse valley*).

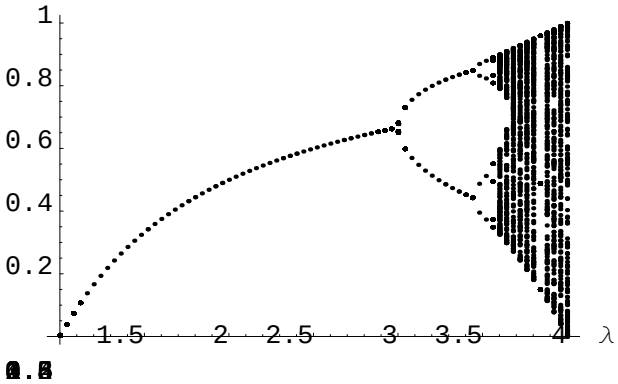
■ **Domaća zadaća 4−3:** Objasnite riječima algoritam ove funkcionalne inačice funkcije **`mandelbrot`**.

```
mandelbrot[0.278, 0.2853, -0.007, -0.012, 400]
```



```
- DensityGraphics -
```

■ **Domaća zadaća 4−4:** Logističko preslikavanje je dano iteracijskom formulom $x_{n+1} = \lambda\, x_n(1 - x_n)$. Za male vrijednosti parametra λ, to preslikavanje za veliki n konvergira k jednoj vrijednosti. Kad λ raste, negdje blizu λ=3, dolazi do bifurkacije i iteracije više ne konvergiraju već skaču između dvije vrijednosti. S daljnjim rastom λ dolazi do nove bifurkacije i sustav se za veliki n ponavlja s periodom četiri, itd. Rezultat se može prikazati kao tzv. Feigenbaumov bifurkacijski dijagram



Nacrtajte ga. Mogući način: za dati λ kreirajte listu $\{x_0, x_1, \cdots, x_{400}\}$ pa maknite prvih 100−200 članova da dođete u područje u kojem se sustav već stabilizirao. Nacrtajte to kao funkciju od λ, koristeći **`ListPlot`**.

■ 4.3 Programiranje transformacijskim pravilima

Ideja ove vrste programiranja u *Mathematici* je promatrati npr. funkciju $f(x) = \sin(x)$, ne kao matematičko pridruživanje koja izvrijednjava izraz x i pridružuje mu $\sin(x)$, već kao transformaciju izraza x u izraz $\sin(x)$. To je u duhu uobičajene procedure računanja rukom — iz reda u red prelazimo tako da matematičke izraze transformiramo u skladu s nekim pravilima.

■ Globalna i lokalna pravila

Pridruživanje vrijednosti simbolima, koje smo obilato koristili, može se shvatiti kao *globalno transformacijsko pravilo*. Npr. pridruživanje

```
x = w;
```

kaže "kad god se u nekom izrazu pojavi x, transformiraj ga u w".

```
Sin[x]^2 + myfun[x^2 + 1]
```

```
myfun[1 + w^2] + Sin[w]^2
```

To pravilo je *globalno* u smislu da će biti upotrijebljeno u svakom izrazu. Ono je *pridruženo simbolu x* i to je pridruživanje na snazi do kraja *Mathematica* sesije tj. komande **Quit**.

```
?x
```

```
Global`x
```

```
x = w
```

(Ovdje "Global" u **Global`x** nema veze s globalnošću pravila, već s globalnošću *konteksta* simbola *x* u što nećemo ovdje ulaziti. Koga zanima, neka pogleda pojam *context* u manualu.)

```
Clear[x]
```

Lokalna pravila su ona koja se primjenjuju samo na jedan izraz, putem "**/.**". I njih smo često koristili:

```
myfun[y^2 + 3] /. y → w
```

```
myfun[3 + w^2]
```

```
3 + y
```

```
3 + y
```

```
?y
```

```
Global`y
```

Vidimo da je simbol **y** "čist", u smislu da mu nije pridruženo nikakvo pravilo.

Ove dvije vrste pravila imaju, kako znamo, i inačice koje djeluju *odgođeno*, u smislu da se izrazi u samom pravilu izvrijednjuju tek prilikom primjene pravila. Za globalna pravila to se postiže upotrebom simbola **:=** umjesto **=**, a za lokalna pravila upotrebom **:>** umjesto **->**.

Ako primjenjujemo *listu* transformacijskih pravila, ona će se sva primijeniti istovremeno

```
{a, b} /. {a → b, b → a}
```

```
{b, a}
```

dok ukoliko primjenjuemo *niz* transformacijskih pravila, ona će se primijeniti konsekutivno, jedno po jedno.

```
{a, b} /. a → b /. b → a
```

```
{a, a}
```

■ Uzorci koji se ponavljaju

Ranije smo upoznali *uzorke* (*patterns*) koji se u definicijama funkcija i transformacijskih pravila koriste kao zamjena (*wildcard*) za proizvoljne izraze. Postoje i specijalni uzorci za izraze koji se ponavljaju

x_	bilo koji izraz
x__	bilo koji izraz koji se ponavlja jednom ili više puta
x___	bilo koji izraz koji se ponavlja nula ili više puta

To omogućuje definiranje funkcije s proizvoljnim brojem argumenata.

```
h[x__] := Power[x]

h[a, b]
```

a^b

```
h[a, b, c]
```

a^{bc}

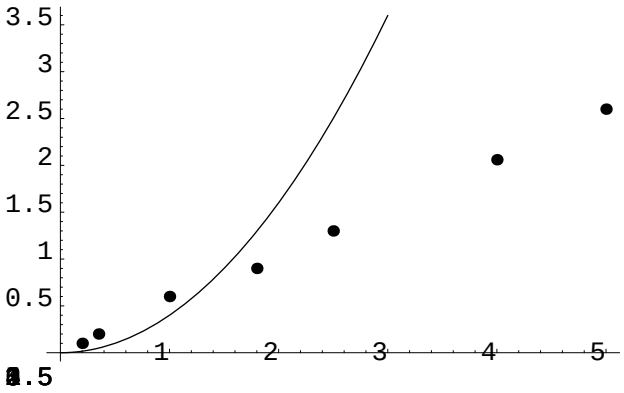
```
h[4, 4, 4]
```

```
1340780792994259709957402499820584612747936582059239337772356144372176403007354697
 680187429816690342769003185818648605085375388281194656994643364900608409 6
```

Primijetite da ovo odudara od matematičkog pojma funkcije koja uvijek mora imati dobro definiran broj argumenata, ali je sasvim u skladu s idejama transformacijskih pravila i uzoraka.

Ovo možemo iskoristiti npr. za definiranje funkcija s opcionalnim argumentima

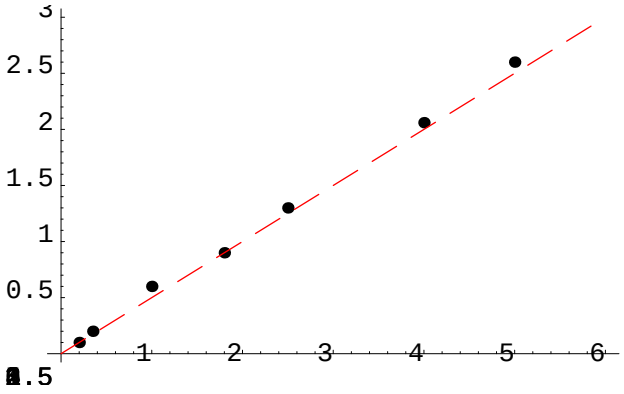
```
data = ReadList["files/fit1.dat", {Number, Number}];

myPlot[fun_, iter_, opts___] :=
 Show[ListPlot[data, PlotStyle → {PointSize[0.02]}, DisplayFunction -> Identity],
  Plot[fun, iter, DisplayFunction → Identity, opts],
  DisplayFunction → $DisplayFunction]

myPlot[0.4 x^2, {x, 0, 3}]
```




```
- Graphics -
```

Ova funkcija **myPlot** radi potpuno isto kao i obična funkcija **Plot**, jedino što preko svakog dijagrama crta još i gore učitane točke. Tako je pogodna npr. za testiranje kvalitete fita. Treći, opcionalni argument **opts**___ služi sa prosljeđivanje željenih opcija funkciji **Plot**.

```
myPlot[0.5 x, {x, 0, 6}, PlotStyle → {RGBColor[1, 0, 0], Dashing[{0.05, 0.03}]}]
```




```
- Graphics -
```

Ako želimo definirati funkciju kod koje opcionalni argumenti kad se izostave imaju neku *defaultnu* vrijednost, koristimo uzorak oblika: **uzorak_:**<*default*>. Npr:

```
f4[x_, n_: 1] := x^n
```

```
f4[a + b, 2]
```

```
(a + b)^2
```

```
f4[a + b]
```

```
a + b
```

■ Predikati

Već smo ranije upoznali *predikate* kao funkcije koje poprimaju samo vrijednosti **True** i **False**. Koristili smo ih kao argument funkcije **Select[**<*lista*>**,** <*predikat*>**]** za izbor elemenata liste koji zadovoljavaju neki uvjet. Predikate koristimo i za ograničavanje područja djelovanja transformacijskih pravila, odnosno, preciznije rečeno, za ograničavanje izraza koji će se spariti s određenim uzorkom. Tako će se npr. uzorak **x_ /; IntegerQ[x]** spariti samo sa izrazima koji su cijeli brojevi.

```
2 x + 3 w /. (x_ /; EvenQ[x]) → x^6
```

```
3 w + 64 x
```

Primijetite da uzorak **x_** ovdje nema nikakve veze sa simbolom **x** koji se pojavljuje u izrazu.

Ovo ima i jednostavniji oblik

```
2 x + 3 w /. (x_?EvenQ) → x^6
```

```
3 w + 64 x
```

Uočite da u prvom obliku, nakon **/;**, dolazi predikat *izvrijednjen* za argument **x** (ime uzorka), a u drugom obliku nakon **?** dolazi predikat kao funkcija.

Predikate možemo definirati i sami, kao bilo koju čistu funkciju koja poprima vrijednosti **True** ili **False:**

```
Select[Range[100], IntegerQ[√#] &]
```

```
{1, 4, 9, 16, 25, 36, 49, 64, 81, 100}
```

Ovdje smo pomoću predikata "da li je cijeli broj", konstruirali predikat "da li je kvadrat cijelog broja".

Koristan predikat je **FreeQ[** <*izraz*>, <*forma*> **]** koji provjerava da li se *forma* (simbol, uzorak, ...) pojavljuje bilo gdje u *izrazu* ili njegovim dijelovima (podizrazima).

```
lst = {1, Sin[w], 5^w, a, 1, c^2, d^w, 4, w};

Cases[lst, x_ /; FreeQ[x, w]]

{1, a, 1, c^2, 4}
```

Dakle ovdje smo izabrali elemente liste koji su "slobodni od w".

Zadatak: Napravite komplementarnu operaciju ove gore, tj. načinite listu čiji elementi će biti oni elementi liste **lst** u kojima se pojavljuje simbol **w**.

■ Primjeri

Koristeći uzorke te svojstvo da funkcije prvo primjenjuju specifičnija pa onda općenitija transformacijska pravila možemo definirati funkcije složenog ponašanja. Npr. logaritme *Mathematica* po defaultu ne ekspandira.

```
Log[a b]

Log[a b]

Expand[%]

Log[a b]
```

To je zapravo željeno ponašanje jer pravila $ln(a\ b) = ln(a)+ln(b)$ i $ln(a^n) = n\ ln(a)$ ne vrijede za proizvoljne argumente (npr. prva relacija ne vrijedi ako su i a i b negativni brojevi). Ukoliko ipak želimo koristiti ove relacije, možemo definirati svoju funkciju

```
logExpand[Log[e1_ e2_]] := Log[e1] + Log[e2]          (* 1 *)
logExpand[Log[e_^n_]] := n Log[e]                     (* 2 *)

logExpand[Log[a b]]

Log[a] + Log[b]

logExpand[Log[a^w]]

w Log[a]
```

No, za kompleksnije izraze **logExpand** radi loše. Npr. ne zna se distribuirati preko zbrajanja:

```
logExpand[Log[a b ] + 2]

logExpand[2 + Log[a b]]

logExpand[Log[a^2 b]]

Log[a^2] + Log[b]
```

U ovom drugom slučaju je problem u tome da nakon jedne primjene pravila ekspanzija prestaje. Trebamo nekako izvesti da se pravila primjenjuju iznova sve dok ih se više nema na što primjeniti.

```
Clear[logExpand]
logExpand[Log[e1_ e2_]] := logExpand[Log[e1]] + logExpand[Log[e2]](* 1 *)
logExpand[Log[e_^n_]] := n logExpand[Log[e]]   (* 2 *)
logExpand[e1_ + e2_] := logExpand[e1] + logExpand[e2]  (* 3 *)
logExpand[c_ e_ /; FreeQ[c, Log]] := c logExpand[e]    (* 4 *)
logExpand[Log[e_]] := Log[e]  (* 5 *)
```

Pravila su modificirana tako da glava **logExpand** preživljava svaku ekspanziju pa tako nakon jednog djelovanja slijedi novo, sve dok se ijedno pravilo može primijeniti. Na kraju zadnje pravilo (br. 5) miče glavu **logExpand**. Pravila br. 3 i 4 se brinu za distribuciju preko zbroja i produkta.

```
logExpand[3 Log[a b^g c] + 2 Log[p w^3]]
```

```
3 (Log[a] + g Log[b] + Log[c]) + 2 (Log[p] + 3 Log[w])
```

Postoji i ugrađena *Mathematica* funkcija **PowerExpand** koja radi isto to, i neke druge stvari povrh toga.

```
PowerExpand[3 Log[a b^g c] + 2 Log[p w^3]]
```

```
3 (Log[a] + g Log[b] + Log[c]) + 2 (Log[p] + 3 Log[w])
```

Zadatak: Korigirajte slijedeću manjkavost funkcije **logExpand:**

```
logExpand[Log[a b] + 2]
```

```
Log[a] + Log[b] + logExpand[2]
```

Želimo ponašanje kao za **PowerExpand** :

```
PowerExpand[Log[a b] + 2]
```

```
2 + Log[a] + Log[b]
```

Zadatak: Definirajte svoju funkciju **der[e_,x_]** koja može derivirati proizvoljne *polinome*, dakle slabiju verziju ugrađene funkcije **D[e_,x_]**.

Domaći zadatak 4−5: Poboljšajte funkciju **der[e_,x_]** tako da može izaći na kraj i s izrazima poput $x^2\ (b\,x + c)$ ili $\frac{x^2+bx}{a\,x}$. Naputak: treba implementirati Leibnizovo lančano pravilo deriviranja produkta funkcija.

Kako pronaći prvo pojavljivanje dva ista elementa za redom u nekoj listi? Postoji zgodna tehnika koja koristi "*triple−blank*" ___ ponavljajuće uzorke:

```
lst = {1, 65, 13, 44, 2, 7, 7, 11, 0};
```

```
lst /. {l___, a_, a_, r___} → a
```

```
7
```

To nam daje koji je to element. Da bi dobili njegovu poziciju u listi pokušajmo s

```
lst /. {l___, a_, a_, r___} → {a, Length[{l}] + 1}
```

```
{7, 2}
```

Problem je da se **Length[{l}]** izvrijednjava prije sparivanja uzorka **l___** pa je to naprosto duljina liste od jednog elementa. Treba upotrijebiti *odgođeno* pravilo.

```
lst /. {l___, a_, a_, r___} :> {a, Length[{l}] + 1}
```

```
{7, 6}
```

Ovo radi dobro jer se sad funkcija **Length[{l}]** izvrijednjava tek nakon što je uzorak **l___** već sparen tj. simbolu **l** je pridijeljena vrijednost — prvih pet elemenata liste **lst**.

Zadatak: Negdje u prvih 1000 decimala broja π pojavljuje se šest istih znamenaka za redom. Pronađite ih. Korisno je upotrijebiti funkcije **ToString** i **Characters** i pomoću njih pretvoriti decimalni zapis broja π u listu znamenaka.

Usput, funkcija **ToString** i njen par **ToExpression** mogu ponekad biti od koristi u konstrukciji raznih izraza.

```
StringJoin["Sin[", "π]"]
```

```
Sin[π]
```

```
ToExpression[%]
```

```
0
```

Skraćena infiks forma funkcije **StringJoin** je "**<>**"

```
ToExpression["2^" <> "3"]
```

```
8
```

Domaći zadatak 4−6: Definirajte funkciju **traziNtorku[list_, n_]** koja će u zadanoj listi pronaći prvo pojavljivanje n istih elemenata zaredom.

Domaći zadatak 4−7: *Prim−brojevi blizanci* su parovi prim−brojeva koji se razlikuju za dva, poput (3,5) ili (17,19). V. Brun je 1919. dokazao da suma recipročnih vrijednosti prim−brojeva blizanaca konvergira

$$B \equiv \left(\frac{1}{3}+\frac{1}{5}\right)+\left(\frac{1}{5}+\frac{1}{7}\right)+\left(\frac{1}{11}+\frac{1}{13}\right)+\left(\frac{1}{17}+\frac{1}{19}\right)+\cdots$$

Konstanta B se zove Brunova konstanta i iznosi približno $B=1.902160583104$. Ovako preciznu vrijednost je vrlo teško dobiti, no napišite komandu **brun[n_]** koja izračunava B koristeći listu od prvih n prim−brojeva. (Koristite ugrađenu funkciju **Prime**.) Inače, zanimljivo je da je upravo računalno određivanje ove konstante ukazalo na bug u prvoj generaciji Pentium procesora.

5. Fizika i *Mathematica*

■ 5.1 Klasična mehanika

Domaći zadatak 5−1: Slično kao na kraju drugog poglavlja, napravite simulaciju gibanja čestice u centralnom sferno−simetričnom gravitacijskom potencijalu

$$V(r) = -\frac{1}{r}$$

Igrajte se s početnim uvjetima da dobijete: (a) zatvorenu (eliptičnu) putanju (b) otvorenu (hiperboličnu) putanju.

Naputak: Kako znamo da je gibanje u ravnini, dovoljno je raditi 2D simulaciju tj. uzeti $r = \sqrt{x^2 + y^2}$.

■ 5.2 Elektrodinamika

■ Gibanje nabijene čestice u homogenom magnetskom polju

Zadatak: Jednadžba gibanja nabijene čestice u elektromagnetskom polju je

$$\mathrm{m}\,\vec{a} = q\left(\vec{E} + \vec{v} \times \vec{B}\right)$$

Riješite je za slučaj homogenog magnetskog polja duž z−osi i nacrtajte putanju za neke tipične početne uvjete.

Naputak: Možete postupiti ovako:

1. Definirajte vektore brzine, akceleracije i magnetskog polja

```
brzina = {x'[t], y'[t], z'[t]};
akceleracija = {x''[t], y''[t], z''[t]};
magnetskoPolje = {0, 0, B};
```

Tada su diferencijalne jednadžbe gibanja

```
jednadzbeGibanja = m akceleracija == q Cross[brzina, magnetskoPolje] // Thread
```

```
{m x''[t] == B q y'[t], m y''[t] == -B q x'[t], m z''[t] == 0}
```

(Pogledajte u manualu upotrebu funkcije **Thread.** Koja je funkcija gore "threaded"?**)**

2. Početne uvjete definirajte kao posebnu listu jednadžbi.

3. Riješite diferencijalne jednadžbe upotrebom **DSolve**

4. Nacrtajte rješenje upotrebom **ParametricPlot3D**

Domaći zadatak 5−2: Upotrebom paketa **Graphics'PlotField3D'** superponirajte na gornju putanju skicu vektorskog magnetskog polja

■ 5.3 Kvantna fizika

■ Bohrov model atoma

Zadatak: Koristeći Bohrov uvjet za orbitu elektrona u atomu vodika

$$\text{moment impulsa} = n\,\hbar$$

izračunajte i nacrtajte spektar vodika.

Naputak: Možete postupiti ovako:

Gornji uvjet

```
eq1 = m v r == n ħ
```

```
m r v == ħ n
```

i jednadžba gibanja (Coulombova sila = m*centripetalno ubrzanje)

```
eq2 = e^2 / (4 π ϵ0 r^2) == m v^2 / r
```

$$\frac{e^2}{4\pi r^2 \epsilon_0} == \frac{m v^2}{r}$$

daju dvije jednadžbe s dvije nepoznanice koje se mogu riješiti po *r* i *v*. Ta riješenja se ona mogu uvrstiti u izraz za ukupnu energiju E(n)=kinetička + potencijalna, što daje Bohrov spektar. Pomoću paketa **`Miscellaneous‘PhysicalConstants‘`** i **`Miscellaneous‘Units‘`** treba napraviti konverziju konstanti da se dobije energija u elektron−voltima (eV).

6. *Mathematica* i "vanjski svijet"

■ 6.1 Izvršavanje*Mathematica* programa u pozadini

Često je za dulje račune nezgodno čekati pred računalom do kraja i možda tako nepotrebno zuzimati radno mjesto. Računala pod Unix operativnim sustavom (Linux, FreeBSD, ...) omogućuju kontrolu računalnih procesa koje tamo možemo staviti u pozadinu (*background*) i osloboditi terminal.

Najjednostavniji način je naprosto upisati odgovarajuće *Mathematica* komande u datoteku, npr. (standardno je ekstenzija takve datoteke **.m**) i onda u komandnoj ljusci (*shell*−u) izvršiti

```
math  < infile.m  > outfile &
```

To poziva *Mathematica kernel*, izvršava komande u datoteci *infile.m* i prosljeđuje output u datoteku *outfile.* Znak "&" na kraju stavlja proces u pozadinu. Nakon toga se može izlogirati s računala. (Ponekad računalo ne dopušta odlogiravanje uz upozorenje "*There are jobs running!*". Tad ponekad pomaže shell naredba *disown*. U slučaju takvih problema proučite dio dokumentacije *shella* koji se tiče upravljanja procesima − *job control*.)

Mali problem s ovim gore pristupom je da su rezultati u *outfile* datoteci relativno teško upotrebljivi za daljnje računalno procesiranje. Tu pomažu dvije tehnike. Jedna je da se unutar samog *Mathematica* programa zatraži redirekcija rezultata komandi koje nas zanimaju u datoteku pomoću komandi "**>>**" i "**>>>**" (vidi manual). Druga je da se iskoristi dodatni paket **ProgrammingInMathematica'NotebookLog'** koji omogućuje da se cijela pozadinska sesija logira u *Mathematica* notebook.

```
<< ProgrammingInMathematica'NotebookLog'
```

```
? NotebookLog
```

```
NotebookLog["file.nb"] starts a transcript in notebook format. NotebookLog[] closes the log file.
```

Slijedeći problem je da datoteka *infile.m* mora biti čisti ASCII pa ne možemo izravno upisivati grčka slove i druge specijalne simbole. Isto tako, često račun priredimo u notebooku pa je poželjno naći način da prepišemo notebook ćelije u *infile.m*. To se radi na slijedeći način: Označimo sve ćelije koje sadrže željeni račun i označimo ih kao inicijalizacijske (`Cell->Cell Properties->Initialization Cell`, to inače služi da se te ćelije automatski izvrše prije bilo čega drugog u notebooku). Nakon toga spremimo notebook kao `File->Save As Special->Package Format`. To će spremiti cijeli notebook u file istog imena, ali s ekstenzijom `.m`, i samo s označenim ćelijama odkomentiranim.

Zadatak: isprobajte sve gore opisano na nekom računu koji traje umjereno dugo, npr. izračun Brunove konstante iz 4. poglavlja za prvih 10^5 ili 10^6 prim brojeva.

■ 6.2 Pokretanje kernela na drugom računalu

Ponekad nam na raspolaganju stoji udaljeno snažno računalo. Međutim, pokretanje *Mathematice* na njemu zbog sporosti mreže čini rad otežanim. Tada je zgodno pokrenuti *Mathematica* "Frontend" lokalno i samo se spojiti na *kernel* udaljenog računala.

Najprije provjerimo da li se na udaljeno računalo uopće možemo logirati ssh aplikacijom i tamo pokrenuti kernel komandom 'math'.

```
ssh username@jaki.stroj.hr math
```

Ako to radi onda u lokalnom notebooku idemo u `Kernel->Kernel Configuration Options` i podesimo potrebne opcije (naziv udaljenog hosta itd.).

Na Unixu je potrebno realizirati mogućnost spajanja bez passworda korištenjem 'ssh−agent'−a. Na MS Windowsima postoji mogućnost specificiranja passworda. (U ovo dvoje nisam apsolutno siguran.)

Nakon toga u `Kernel->Notebook's kernel` samo specificiramo koji kernel želimo koristiti.

■ 6.3 Povezivanje *Mathematice* s programima u C−u

Mathematica je u zadnje vrijeme dosta napredovala u pogledu brzine numeričke matematike, ali ponekad je neke zahtjevne račune zgodnije delegirati bržim kompajliranim jezicima poput C−a ili Fortrana. Isto tako, ponekad je poželjno iz drugih jezika pozvati neku *Mathematica* funkciju. To se izvodi *MathLink* sustavom.

Za pozvati C funkciju iz *Mathematice MathLink*, pored same definicije funkcije u C−u traži još i tzv. *template* datoteku s ekstenzijom **`.tm`** u kojoj je opisana sama funkcija i njeni argumenti. Npr, funkcija **`zbroj`** koja zbraja dva cijela broja ima ovakvu template datoteku (nazovimo je *zbroj.tm*):

```
:Begin:
:Function:      zbroj
:Pattern:       zbroj[x_Integer, y_Integer]
:Arguments:     {x, y}
:ArgumentTypes: {Integer, Integer}
:ReturnType:    Integer
:End:
```

Odgovarjući C−program (u datoteci *zbroj.c*) može izgledati ovako. Prvo uključujemo *MathLink* header

```
#include "mathlink.h"
```

Zatim dolazi C kod koji definira funkciju *zbroj*

```
int zbroj(int x, int y) {
   return x+y;
}
```

I na kraju glavni program koji se vrti i čeka poziv od *Mathematice*

```
int main(int argc, char *argv[]) {
   return MLMain(argc, argv);
}
```

Kompajliranje se vrši (na Unixu!) programom *mcc* ovako:

```
mcc −o zbroj.exe zbroj.tm zbroj.c
```

Da bi se funkcija mogla koristiti treba je u *Mathematica* sesiji prvo instalirati

```
Install["files/zbroj.exe"]

LinkObject[./files/zbroj.exe, 2, 2]
```

i nakon toga nam je dostupna

```
zbroj[4, 3]

7
```

Moguće je i povezivanje s programima u Fortranu, no samo posredno putem *wrapper* programa u C−u. Za sve detalje oko korištenja *MathLink*a treba konzultirati "*MathLink Tutorial*" na adresi http://library.wolfram.com/infocenter/TechNotes/174/