

NUMERIČNA MATEMATIKA

V PROGRAMSKEM JEZIKU JULIA

Martin Vuk

2025

Univerza v Ljubljani

Fakulteta za računalništvo in informatiko

Kataložni zapis o publikaciji (CIP) pripravili v Narodni in univerzitetni knjižnici v Ljubljani

COBISS.SI-ID=218309379

ISBN 978-961-7059-16-8 (PDF)

Copyright © 2025 Založba UL FRI. All rights reserved.

Elektronska izdaja knjige je na voljo na:

URL: <http://zalozba.fri.uni-lj.si/vuk2024.pdf>

DOI: [10.51939/0005](https://doi.org/10.51939/0005)

Datum izdelave PDF: 21. 02. 2025

Recenzenta: doc. dr. Aljaž Zalar, prof. dr. Emil Žagar

Založnik: Založba UL FRI, Ljubljana

Izdajatelj: UL Fakulteta za računalništvo in informatiko, Ljubljana

Urednik: prof. dr. Franc Solina

Predgovor

Knjige o numerični matematiki se pogosto posvečajo predvsem matematičnim vprašanjem. Pričujoča poskuša nasloviti bolj praktične vidike numerične matematike, zato so primeri, če je le mogoče, povezani s problemi s področja fizike, matematičnega modeliranja ali računalništva. Za podrobnejši matematični opis uporabljenih metod in izpeljav bralcu priporočam učbenika Osnove numerične matematike Bojana Orla [1] in Razširjen uvod v numerične metode Bora Plestenjaka [2].

Knjiga je prvenstveno namenjena študentom Fakultete za računalništvo in informatiko Univerze v Ljubljani kot gradivo za izvedbo laboratorijskih vaj pri predmetu Numerična matematika. Kljub temu je primerna za vse, ki želijo bolje spoznati algoritme numerične matematike, uporabo numeričnih metod ali se naučiti uporabljati programski jezik [Julia](#). Pri sem se od bralca pričakuje osnovno znanje programiranja v kakšnem drugem programskem jeziku.

V knjigi so naloge razdeljene na vaje in na domače naloge. Vaje so zasnovane za samostojno delo z računalnikom, pri čemer lahko bralec naloge rešuje z različno mero samostojnosti. Vsaka vaja se začne z opisom naloge in jasnimi navodili, kaj je njen cilj oziroma končni rezultat. Sledijo podrobnejši napotki, kako se naloge lotiti, na koncu pa je rešitev z razlago posameznih korakov. Rešitev vključuje matematične izpeljave, programsko kodo in rezultate, ki jih dobimo, če programsko kodo uporabimo. Vsi programi iz te knjige so na voljo na spletni strani: <https://gitlab.com/nummat/nummat-knjiga/>

Domače naloge rešuje bralec povsem samostojno, zato so naloge brez rešitev. Odločitev, da rešitve niso vključene, je namerna, saj bralec lahko verodostojno preveri svoje znanje le, če rešuje tudi naloge, za katere nima dostopa do rešitev.

Vsekakor bralcu svetujem, da vso kodo napiše in preskusi sam. Še bolje je, če kodo razširi, jo spreminja in se z njo igra. Koda, ki je navedena v tej knjigi, je najosnovnejša različica, ki reši določen problem in še ustreza minimalnim standardom pisanja kvalitetne kode. Pogosto sta izpuščena preverjanje in implementacija robnih primerov, včasih tudi obravnava pričakovanih napak. Da je bralcu lažje razumeti, kaj koda počne, sem dal prednost berljivosti pred celovitostjo.

Na tem mestu bi se rad zahvalil prof. dr. Bojanu Orlu, prof. dr. Emilu Žagarju, asist. dr. Petru Kinku in doc. dr. Aljažu Zalarju, s katerimi sem sodeloval ali še sodelujem pri numeričnih predmetih na FRI. Veliko idej za naloge, ki so v tej knjigi, prihaja prav od njih. Posebna zahvala gre recenzentoma, ki sta knjigo podrobno pregledala in veliko prispevala h kakovosti vsebine. Moja draga žena doc. dr. Mojca Vilfan mi je pomagala „*zbrusiti ostre robe*“, za kar sem ji izjemno hvaležen. Prav tako se zahvaljujem članom Laboratorija za matematične metode v računalništvu in informatiki, še posebej prof. dr. Neži Mramor-Kosta in asist. dr. Damirju Franetiču, ki so tako ali drugače prispevali k nastanku te knjige. Hvala Založbi FRI in njenemu uredniku prof. dr. Francu Solini, ki sta omogočila izid. Na koncu bi se rad zahvalil študentom, ki so obiskovali numerične predmete. Čeprav sem jih jaz učil, so bili oni tisti, ki so me naučili marsikaj novega.

Martin Vuk

Kazalo

1	Uvod v programski jezik Julia	7
1.1	Namestitev in prvi koraki	7
1.2	Avtomatsko posodabljanje kode	15
1.3	Priprava korenske mape	15
1.4	Vodenje različic s programom Git	16
1.5	Priprava paketa za vajo	17
1.6	Koda	18
1.7	Testi	19
1.8	Dokumentacija	20
1.9	Zaključek	25
2	Računanje kvadratnega korena	26
2.1	Naloga	26
2.2	Izbira algoritma	26
2.3	Določitev števila korakov	28
2.4	Izbira začetnega približka	30
2.5	Zaključek	33
3	Tridiagonalni sistemi	35
3.1	Naloga	35
3.2	Tridiagonalne matrike	35
3.3	Reševanje tridiagonalnega sistema	37
3.4	Slučajni sprehod	38
3.5	Pričakovano število korakov	40
3.6	Rešitve	43
4	Minimalne ploskve	48
4.1	Naloga	48
4.2	Matematično ozadje	49
4.3	Diskretizacija in sistem linearnih enačb	49
4.4	Matrika sistema linearnih enačb	51
4.5	Izpeljava sistema s Kroneckerjevim produktom	52
4.6	Numerična rešitev z LU razcepom	53
4.7	Napolnitev matrike ob eliminaciji	55
4.8	Iteracijske metode	56
4.9	Rešitve	60
5	Interpolacija z implicitnimi funkcijami	63
5.1	Naloga	63
5.2	Interpolacija z radialnimi baznimi funkcijami	63
5.3	Program	65
5.4	Rešitve	68
6	Fizikalna metoda za vložitev grafov	70
6.1	Naloga	70
6.2	Ravnovesje sil	70
6.3	Rešitev v Julii	72
6.4	Krožna lestev	72
6.5	Dvodimenzionalna mreža	74
6.6	Rešitve	76
7	Invariantna porazdelitev Markovske verige	81

7.1	Naloga	81
7.2	Limitna porazdelitev Markovske verige	82
7.3	Potenčna metoda	82
7.4	Razvrščanje spletnih strani	82
7.5	Skakanje skakača po šahovnici	84
7.6	Rešitve	87
8	Spektralno razvrščanje v gruče	89
8.1	Naloga	89
8.2	Podobnostni graf in Laplaceova matrika	89
8.3	Algoritem	90
8.4	Primer	90
8.5	Inverzna iteracija	92
8.6	Inverzna iteracija s QR razcepom	93
8.7	Premik	94
8.8	Rešitve	96
8.9	Testi	98
9	Konvergenčna območja sistemov nelinearnih enačb	99
9.1	Naloga	99
9.2	Newtonova metoda za sisteme enačb	99
9.3	Konvergenčno območje	101
9.4	Rešitve	102
10	Nelinearne enačbe v geometriji	105
10.1	Naloga	105
10.2	Presečišča parametrično podanih krivulj	105
10.3	Minimalna razdalja med dvema krivuljama	108
10.4	Rešitve	114
11	Aproksimacija z linearnim modelom	116
11.1	Naloga	116
11.2	Linearni model	116
11.3	Opis sprememb koncentracije CO ₂	117
11.4	Normalni sistem	119
11.5	QR razcep	120
11.6	Kaj pa CO ₂ ?	121
12	Interpolacija z zlepki	123
12.1	Naloga	123
12.2	Hermitov kubični zlepak	123
12.3	Ocena za napako	126
12.4	Newtonov interpolacijski polinom	128
12.5	Rungejev pojav	129
12.6	Rešitve	130
12.7	Testi	134
13	Integrali	136
13.1	Naloga	136
13.2	Trapezno pravilo in sestavljeni trapezni pravilo	136
13.3	Simpsonovo pravilo	139
13.4	Gaussove kvadraturne formule	140
13.5	Primeri	142
13.6	Testi	146
13.7	Rešitve	147

13.8	Izpeljava Simpsonovega pravila	150
14	Povprečna razdalja med dvema točkama na kvadratu	152
14.1	Naloga	152
14.2	Dvojni integral in integral integrala	152
14.3	Metoda Monte Carlo	154
14.4	Povprečna razdalja med točkama na kvadratu $[0, 1]^2$	154
14.5	Rešitve	158
14.6	Testi	160
15	Avtomatsko odvajanje z dualnimi števili	162
15.1	Naloga	162
15.2	Ideja avtomatskega odvoda	162
15.3	Dualna števila	164
15.4	Keplerjeva enačba	166
15.5	Računanje gradientov	168
15.6	Gradient Ackleyeve funkcije	170
15.7	Rešitve	172
16	Začetni problem za navadne diferencialne enačbe	176
16.1	Naloga	176
16.2	Reševanje enačbe z eno spremenljivko	176
16.3	Eulerjeva metoda	178
16.4	Sistemi NDE	179
16.5	Ogrodje za reševanje NDE	180
16.6	Metode Runge-Kutta	182
16.7	Hermitova interpolacija	182
16.8	Poševni met z upoštevanja zračnega upora	183
16.9	Čas in dolžina meta	187
16.10	Rešitve	191
16.11	Testi	196
17	Navodila za pripravo domačih nalog	197
17.1	Kontrolni seznam	197
17.2	Kako pisati in kako ne	197
17.3	Kako pisati avtomatske teste	199
17.4	Priprava zahteve za združitev	200
18	Domače naloge	201
18.1	Prva domača naloga	201
18.2	Druga domača naloga	205
18.3	Tretja domača naloga	209
	Literatura	212

1 Uvod v programski jezik Julia

V knjigi bomo uporabili programski jezik [Julia](#) [3]. Zavoljo učinkovitega izvajanja, uporabe [dinamičnih tipov](#) in [metod, specializiranih glede na signaturo](#), ter dobre podpore za interaktivno uporabo, je Julia zelo primerna za programiranje numeričnih metod in ilustracijo njihove uporabe. V nadaljevanju sledijo kratka navodila, kako začeti z Julio.

Cilji tega poglavja so:

- naučiti se uporabljati Julio v interaktivni ukazni zanki,
- pripraviti okolje za delo v programskejem jeziku Julia,
- ustvariti prvi paket in
- ustvariti prvo poročilo v formatu PDF.

Tekom te vaje bomo pripravili svoj prvi paket v Julii, ki bo vseboval parametrično enačbo [Geronove lemniskate](#), in napisali teste, ki bodo preverili pravilnost funkcij v paketu. Nato bomo napisali skripto, ki uporabi funkcije iz našega paketa in nariše sliko Geronove lemniskate. Na koncu bomo pripravili lično poročilo v formatu PDF.

1.1 Namestitev in prvi koraki

Programski jezik Julia namestite tako, da sledite [navodilom](#), nato v terminalu poženite ukaz `julia`. Ukaz odpre interaktivno ukazno zanko (angl. *Read Eval Print Loop* ali s kratico REPL) in v terminalu se pojavi ukazni pozivnik `julia>`. Za ukaznim pozivnikom lahko napišemo posamezne ukaze, ki jih nato Julia prevede in izpiše rezultate. Poskusimo najprej s preprostimi izrazi:

```
julia> 1 + 1
2

julia> sin(pi)
0.0

julia> x = 1; 2x + x^2
3

julia> # vse, kar je za znakom #, je komentar, ki se ne izvede
```

1.1.1 Funkcije

Funkcije, ki so v programskejem jeziku Julia osnovne enote kode, definiramo na več načinov. Kratke enovrstične funkcije definiramo z izrazom `ime(x) = ...`

```
julia> f(x) = x^2 + sin(x)
f (generic function with 1 method)

julia> f(pi / 2)
3.4674011002723395
```

Funkcije z več argumenti definiramo podobno:

```
julia> g(x, y) = x + y^2
g (generic function with 1 method)

julia> g(1, 2)
5
```

Za funkcije, ki zahtevajo več kode, uporabimo ključno besedo `function`:

```
julia> function h(x, y)
    z = x + y
    return z^2
end
h (generic function with 1 method)

julia> h(3, 4)
49
```

Funkcije lahko uporabljammo kot vsako drugo spremenljivko. Lahko jih podamo kot argumente drugim funkcijam ali jih združujemo v podatkovne strukture, kot so seznamy, vektorji ali matrike. Definiramo jih lahko tudi kot anonimne funkcije. To so funkcije, ki jih vpeljemo brez imena in jih kasneje ne moremo poklicati po imenu.

```
julia> (x, y) -> sin(x) + y
#1 (generic function with 1 method)
```

Anonimne funkcije uporabljammo predvsem kot argumente v drugih funkcijah. Funkcija `map(f, v)` na primer zahteva za prvi argument funkcijo `f`, ki jo nato aplicira na vsak element vektorja `v`:

```
julia> map(x -> x^2, [1, 2, 3])
3-element Vector{Int64}:
 1
 4
 9
```

Vsaka funkcija v programskejem jeziku Julia ima lahko več različnih definicij, glede na kombinacijo tipov argumentov, ki jih podamo. Posamezno definicijo imenujemo `metoda`. Ob klicu funkcije Julia izbere najprimernejšo metodo.

```
julia> k(x::Number) = x^2
k (generic function with 1 method)

julia> k(x::Vector) = x[1]^2 - x[2]^2
k (generic function with 2 methods)

julia> k(2)
4

julia> k([1, 2, 3])
-3
```

1.1.2 Vektorji in matrike

Vektorje vnesemo z oglatimi oklepaji []:

```
julia> v = [1, 2, 3]
3-element Vector{Int64}:
1
2
3

julia> v[1] # vrne prvo komponento vektorja
1

julia> v[2:end] # vrne komponente vektorja od druge do zadnje
2-element Vector{Int64}:
2
3

julia> sin.(v) # funkcijo uporabimo na komponentah vektorja, če imenu dodamo .
3-element Vector{Float64}:
0.8414709848078965
0.9092974268256817
0.1411200080598672
```

Matrike vnesemo tako, da elemente v vrstici ločimo s presledki, vrstice pa s podpičji:

```
julia> M = [1 2 3; 4 5 6]
2x3 Matrix{Int64}:
1 2 3
4 5 6
```

Za razpone indeksov uporabimo :, s ključno besedo end označimo zadnji indeks. Julia avtomačno določi razpon indeksov v matriki:

```
julia> M[1, :] # prva vrstica
3-element Vector{Int64}:
1
2
3

julia> M[2:end, 1:end-1]
1x2 Matrix{Int64}:
4 5
```

Osnovne operacije delujejo tudi na vektorjih in matrikah. Pri tem moramo vedeti, da gre za matrične operacije. Tako je na primer * operacija množenja matrik ali matrike z vektorjem in ne morda množenja po komponentah.

```
julia> [1 2; 3 4] * [6, 5] # množenje matrike z vektorjem
2-element Vector{Int64}:
16
38
```

Če želimo operacije izvajati po komponentah, moramo pred operator dodati piko, na kar nas Julia opozori z napako:

```
julia> [1, 2] + 1 # seštevanje vektorja in števila ni definirano
ERROR: MethodError: no method matching +(::Vector{Int64}, ::Int64)
For element-wise addition, use broadcasting with dot syntax: array .+ scalar

julia> [1, 2] .+ 1
2-element Vector{Int64}:
 2
 3
```

Posebej uporaben je operator \, ki poišče rešitev sistema linearnih enačb. Izraz $A \backslash b$ vrne rešitev matričnega sistema $Ax = b$:

```
julia> A = [1 2; 3 4]; # podpičje prepreči izpis rezultata

julia> x = A \ [5, 6] # reši enačbo A * x = [5, 6]
2-element Vector{Float64}:
 -3.9999999999999987
 4.49999999999999
```

Izračun se izvede v aritmetiki s plavajočo vejico, zato pride do zaokrožitvenih napak in rezultat ni povsem točen. Naredimo še preizkus:

```
julia> A * x
2-element Vector{Float64}:
 5.0
 6.0
```

Operator \ deluje za veliko različnih primerov. Med drugim ga lahko uporabimo za iskanje rešitve predoločenega sistema po metodi najmanjših kvadratov:

```
julia> [1 2; 3 1; 2 2] \ [1, 2, 3] # rešitev za predoločen sistem
2-element Vector{Float64}:
 0.5999999999999999
 0.5111111111111114
```

1.1.3 Zanke in kontrolne strukture

Za zanko z znanim številom korakov uporabimo ukaz **for**:

```
julia> for i=1:3
           println("Trenutni števec je $i")
       end
Trenutni števec je 1
Trenutni števec je 2
Trenutni števec je 3
```

Julia podpira tudi sintakso z ukazom **for i in vektor** podobno kot Python. Namesto razpona 1:3, ki je tipa [LinRange](#), lahko for zanko izvedemo tudi po vektorju:

```
julia> for i in [2, 3, 1]
        println("Trenutni števec je $i")
    end
Trenutni števec je 2
Trenutni števec je 3
Trenutni števec je 1
```

Julia omogoča še vrsto drugih konstruktorov, ki so v bistvu zanke. Poglejmo tri različne načine, kako iz vektorja $v = [1, 2, 3]$ sestavimo vektor funkcijskih vrednosti $[f(1), f(2), f(3)]$:

```
julia> v = [1, 2, 3]
julia> f(x) = x^2
julia> [f(vi) for vi in v] # podobno kot v Pythonu
3-element Vector{Int64}:
1
4
9

julia> f.(v) # operator . je alias za funkcijo broadcast, ki funkcijo aplicira na komponente
3-element Vector{Int64}:
1
4
9

julia> map(f, v)
3-element Vector{Int64}:
1
4
9
```

Zanko lahko izvedemo tudi z ukazom `while`. Podobno kot v drugih programskeh jezikih deluje `if` stavek:

```
julia> if 1 < 2
        println("1 je manj kot 2")
    else
        println("1 je več ali enako 2")
    end
1 je manj kot 2
```

Rezultat `if` stavka je enak rezultatu veje, ki se izvede:

```
julia> x = if 1 < 2
        1
    else
        2
    end
1
```

Prejšnji izraz `if/else` krajše zapišemo s tričlenskim operatorjem (pogoj ? a : b):

```
julia> x = 1 < 2 ? 1 : 2
1
```

Če izpustimo `else` del, je rezultat `if` stavka bodisi rezultat telesa, če je pogoj izpolnjen, bodisi enak vrednosti `nothing`, če pogoj ni izpolnjen:

```
julia> x = if 1 > 2
           1
           end
julia> typeof(x) # typeof vrne tip argumenta
Nothing
```

1.1.4 Podatkovni tipi

Podatkovne tipe definiramo z ukazom `struct`. Ustvarimo tip, ki predstavlja točko z dvema koordinatama:

```
julia> struct Točka
           x
           y
           end
```

Ko definiramo nov tip, se avtomatično ustvari tudi funkcija z istim imenom, s katero lahko ustvarimo vrednost novog definiiranega tipa. Vrednost tipa Točka ustvarimo s funkcijo `Točka(x, y)`:

```
julia> T = Točka(1, 2) # ustvari vrednost tipa Točka
Točka(1, 2)

julia> T.x
1

julia> T.y
2
```

Julia omogoča različne definicije iste funkcije za različne podatkovne tipe. Za določitev tipa argumenta funkcije uporabimo operator `::`. Definirajmo funkcijo, ki izračuna razdaljo med dvema točkama:

```
julia> razdalja(T1::Točka, T2::Točka) = sqrt((T2.x - T1.x)^2 + (T2.y - T1.y)^2)
razdalja (generic function with 1 method)

julia> razdalja(Točka(1, 2), Točka(2, 1))
1.4142135623730951
```

1.1.5 Moduli

Moduli pomagajo organizirati funkcije v enote in omogočajo uporabo istega imena za različne funkcije in tipe. Module definiramo z `module ImeModula ... end`:

```
julia> module KrNeki
    kaj(x) = x + sin(x)
    čaj(x) = cos(x) - x
    export kaj
end
Main.KrNeki
```

Če želimo funkcije, ki so definirane v modulu NekiModul, uporabiti izven modula, moramo modul naložiti z `using NekiModul`. Funkcije, ki so izvožene z ukazom `export ime_funkcije` lahko kličemo kar po imenu, ostalim funkcijam pa moramo dodati ime modula kot predpono. Modulom, ki niso del paketa in so definirani lokalno, moramo dodati piko, ko jih naložimo:

```
julia> using .KrNeki

julia> kaj(1)
1.8414709848078965

julia> KrNeki.čaj(1)
-0.45969769413186023
```

Modul lahko naložimo tudi z ukazom `import NekiModul`. V tem primeru moramo vsem funkcijam iz modula dodati ime modula in piko kot predpono.

1.1.6 Paketi

Nabor funkcij, ki so na voljo v Julii, je omejen, zato pogosto uporabimo knjižnice, ki vsebujejo dodatne funkcije. Knjižnica funkcij v Julii se imenuje **paket**. Funkcije v paketu so združene v modul, ki ima isto ime kot paket.

Julia ima vgrajen upravljalnik s paketi, ki omogoča dostop do paketov, ki so del Julie, kot tudi tistih, ki jih prispevajo uporabniki. Poglejmo si primer, kako uporabiti ukaz `norm`, ki izračuna različne norme vektorjev in matrik. Ukaz `norm` ni del osnovnega nabora funkcij, ampak je del modula `LinearAlgebra`, ki je že vključen v program Julia. Če želimo uporabiti `norm`, moramo najprej uvoziti funkcije iz modula `LinearAlgebra` z ukazom `using LinearAlgebra`:

```
julia> norm([1, 2, 3]
ERROR: UndefVarError: `norm` not defined

julia> using LinearAlgebra
julia> norm([1, 2, 3])
3.7416573867739413
```

Kadar želimo uporabiti pakete, ki niso del osnovnega jezika Julia, jih moramo prenesti z interneta. Za to uporabimo modul `Pkg`. Paketom je namenjen poseben paketni način vnosa v ukazni zanki. Do paketnega načina pridemo, če za pozivnik vnesemo znak `]`.

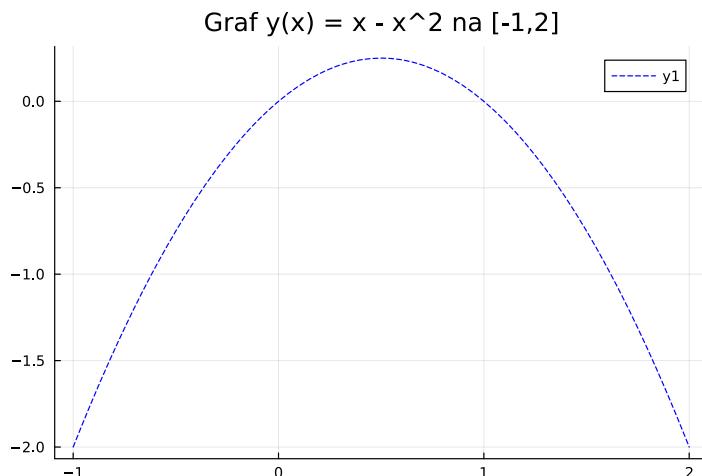
Različni načini ukazne zanke

Ukazna zanka (REPL) v Julii pozna več načinov, ki so namenjeni različnim opravilom.

- Osnovni način s pozivom `julia>` je namenjen vnosu kode.
- Paketni način s pozivom `pkg>` je namenjen upravljanju s paketi. V paketni način pridemo, če vnesemo znak `]`.
- Način za pomoč s pozivom `help?>` je namenjen pomoči. V način za pomoč pridemo z znakom `?`.
- Lupinski način s pozivom `shell>` je namenjen izvajanju ukazov v sistemski lupini. V lupinski način vstopimo z znakom `;`.
- Iz posebnih načinov pridemo nazaj v osnovni način s pritiskom na vračalko (`⊟`).

Poglejmo, kako namestiti knjižnico za ustvarjanje slik in grafov `Plots.jl`[4]. Najprej aktiviramo paketni način z vnosom znaka `]` za pozivnikom. Nato paket dodamo z ukazom `add`:

```
(@v1.10) pkg> add Plots
...
julia> using Plots # naložimo modul s funkcijami iz paketa
julia> plot(x -> x - x^2, -1, 2, color=:blue,
    linestyle=:dash, title="Graf y(x) = x - x^2 na [-1,2]")
```



1.1.7 Datoteke s kodo

Vnašanje ukazov v interaktivni zanki je uporabno za preproste ukaze, na primer namesto kalkulatorja. Za resnejše delo je bolje kodo shraniti v datoteke. Praviloma imajo datoteke s kodo v jeziku Julia končnico `.jl`.

Napišimo preprost program. Ukaze, ki smo jih vnesli doslej, shranimo v datoteko z imenom `01uvod.jl`. Ukaze iz datoteke poženemo z ukazom `include` v ukazni zanki:

```
julia> include("01uvod.jl")
```

ali pa v lupini operacijskega sistema:

```
$ julia 01uvod.jl
```

Urejevalniki in programska okolja za Julio

Za lažje delo z datotekami s kodo potrebujemo dober urejevalnik besedila, ki je namenjen programiranju. Če še nimate priljubljenega urejevalnika, priporočam [VS Code](#) in [razširitev za Julio](#).

Če odpremo datoteko s kodo v urejevalniku VS Code, lahko s kombinacijo tipk **Ctrl + Enter** posamezno vrstico kode pošljemo v ukazno zanko za Julio, da se izvede. Na ta način združimo prednosti interaktivnega dela in zapisovanja kode v datoteke .jl.

Priporočam, da večino kode napišete v datoteke. V nadaljevanju bomo spoznali, kako organizirati datoteke v projekte in pakete tako, da lahko kodo uporabimo na več mestih.

1.2 Avtomatsko posodabljanje kode

Ko uporabimo kodo iz datoteke v interaktivni zanki, je treba ob vsaki spremembi datoteke ponovno naložiti z ukazom `include`. Paket [Revise.jl](#) poskrbi, da se koda ponovno naloži vsakič, ko se datoteke spremenijo. Zato najprej namestimo paket Revise in poskrbimo, da se zažene ob vsakem zagonu Julie.

Naslednji ukazi namestijo paket Revise, ustvarijo mapo `$HOME/.julia/config` in datoteko `startup.jl`, ki naloži modul Revise ob vsakem zagonu programa julia:

```
julia> # pritisnemo ], da pridemo v paketni način
(@v1.10) pkg> add Revise
julia> startup = """
try
    using Revise
catch e
    @warn "Error initializing Revise" exception=(e, catch_backtrace())
end
"""

...
julia> path = homedir() * "./.julia/config"
julia> mkdir(path)
julia> write(path * "/startup.jl", startup) # zapišemo startup.jl
```

Okolje za delo z Julio je pripravljeno.

1.3 Priprava korenske mape

Programe, ki jih bomo napisali v nadaljevanju, bomo hranili v mapi `num_mat`. Ustvarimo jo z ukazom:

```
$ mkdir num_mat
```

Korenska mapa bo služila kot [projektno okolje](#), v katerem bodo zabeleženi vsi paketi, ki jih bomo potrebovali. Projektno okolje aktiviramo z ukazom `activate` pot/do/okolja v paketnem načinu.

```
$ cd num_mat
$ julia

julia> # s pritiskom na ] vključimo paketni način
(@v1.10) pkg> activate .
(num_mat) pkg>
```

Zgornji ukaz ustvari datoteko `Project.toml` in pripravi novo projektno okolje v mapi `num_mat`.

Projektno okolje v Julii

Projektno okolje je mapa, ki vsebuje datoteko `Project.toml` z informacijami o paketih in zahtevanih različicah paketov. Projektno okolje aktiviramo z ukazom `Pkg.activate("pot/do/mape/z/okoljem")` oziroma v paketnem načinu z:

```
(@v1.10) pkg> activate pot/do/mape/z/okoljem
```

Uporaba projektnega okolja delno rešuje problem [ponovljivosti](#), ki ga najlepše ilustriramo z izjavo „Na mojem računalniku pa koda dela!“. Projektno okolje namreč vsebuje tudi datoteko `Manifest.toml`, ki hrani različice in kontrolne vsote za pakete iz `Project.toml` in vse njihove odvisnosti. Ta informacija omogoča, da Julia naloži vedno iste različice vseh odvisnosti, kot v času, ko je bila datoteka `Manifest.toml` zadnjič posodobljena.

Projektna okolja v Juliji so podobna [virtualnim okoljem v Pythonu](#).

Projektnemu okolju dodamo pakete, ki jih bomo uporabili v nadaljevanju. Zaenkrat je to le paket `Plots.jl`, ki ga uporabljam za risanje grafov:

```
(num_mat) pkg> add Plots
```

Datoteka `Project.toml` vsebuje le ime paketa `Plots` in identifikacijski niz:

```
[deps]
Plots = "91a5bcd-55d7-5caf-9e0b-520d859cae80"
```

Točna verzija paketa `Plots` in vsi paketi, ki jih potrebuje, so zabeleženi v datoteki `Manifest.toml`.

1.4 Vodenje različic s programom Git

Za vodenje različic priporočam uporabo programa [Git](#). V nadaljevanju bomo opisali, kako v korenski mapi `num_mat` pripraviti Git repozitorij in vpisati datoteke, ki smo jih do sedaj ustvarili.

Sistem za vodenje različic Git

[Git](#) je sistem za vodenje različic, ki je postal *de facto* standard v razvoju programske opreme in tudi drugod, kjer se dela z besedilnimi datotekami. Priporočam, da si bralec ustvari svoj Git repozitorij, kjer si uredi kodo in zapiske, ki jih bo napisal pri spremeljanju te knjige.

Git repozitorij lahko hranimo zgolj lokalno na lastnem računalniku, lahko pa ga kloniramo na lastni strežnik ali na enega od javnih spletnih skladišč programske kode, na primer [GitHub](#) ali [GitLab](#).

Z naslednjim ukazom v mapi `num_mat` ustvarimo repozitorij za `git` in registriramo novo ustvarjene datoteke.

```
$ git init .
$ git add .
$ git commit -m "Začetni vpis"
```

Z ukazoma `git status` in `git diff` pregledamo, kaj se je spremenilo od zadnjega vpisa. Ko smo zadovoljni s spremembami, jih zabeležimo z ukazoma `git add` in `git commit`. Priporočamo redno uporabo ukaza `git commit`. Pogosti vpisi namreč precej olajšajo nadzor nad spremembami kode in spodbujajo k delitvi dela na majhne zaključene probleme, ki so lažje obvladljivi.

1.5 Priprava paketa za vajo

Ob začetku vsake vaje bomo v korenki mapi (`num_mat`) najprej ustvarili mapo oziroma [paket](#), v katerem bo shranjena koda za določeno vajo. S ponavljanjem postopka priprave paketa za vsako vajo posebej se bomo naučili, kako hitro začeti s projektom. Obenem bomo optimizirali potek dela in odpravili ozka grla v postopkih priprave projekta. Ponavljanje vedno istih postopkov nas prisili, da postopke kar se da poenostavimo in ponavljača se opravila avtomatiziramo. Na dolgi rok se tako lahko bolj posvečamo dejanskemu reševanju problemov.

Za vajo bomo ustvarili paket `Vaja01`, s katerim bomo narisali [Geronovo lemniskato](#).

V mapi `num_mat` ustvarimo paket `Vaja01`, v katerega bomo shranili kodo. Nov paket ustvarimo v paketnem načinu z ukazom `generate`:

```
$ cd num_mat
$ julia

julia> # pritisnemo ] za vstop v paketni način
(@v1.10) pkg> generate Vaja01
```

Ukaz `generate` ustvari mapo `Vaja01` z osnovno strukturo [paketa v Julii](#):

```
$ tree Vaja01
Vaja01
├── Project.toml
└── src
    └── Vaja01.jl

1 directory, 2 files
```

Paket `Vaja01` nato dodamo v projektno okolje v korenki mapi `num_mat`, da bomo lahko kodo iz paketa uporabili v programih in ukazni zanki. Namesto ukaza `add` uporabimo ukaz `develop`, ker želimo paket `Vaja01` še spremenjati:

```
(@v1.10) pkg> activate .
(num_mat) pkg> develop ./Vaja01
```

Za obsežnejši projekti uporabite šablone

Za obsežnejši projekt ali projekt, ki ga želite objaviti, je bolje uporabiti že pripravljene šablone [PkgTemplates](#) ali [PkgSkeleton](#). Zavoljo enostavnosti bomo v sklopu te knjige pakete ustvarjali s `Pkg.generate`.

Osnovna struktura paketa je pripravljena. Paketu bomo v nadaljevanju dodali še:

- kodo (Poglavlje 1.6),
- teste (Poglavlje 1.7) in
- dokumentacijo (Poglavlje 1.8).

1.6 Koda

Ko je mapa s paketom `Vaja01` pripravljena, lahko začnemo. Napisali bomo funkciji, ki izračunata koordinati Geronove lemniskate:

$$x(t) = \frac{t^2 - 1}{t^2 + 1}, \quad y(t) = \frac{2t(t^2 - 1)}{(t^2 + 1)^2}. \quad (1.1)$$

V urejevalniku odpremo datoteko `Vaja01/src/Vaja01.jl` in vanjo shranimo definiciji:

```
module Vaja01

    """Izračunaj `x` kordinato Geronove lemniskate."""
    lemniskata_x(t) = (t^2 - 1) / (t^2 + 1)
    """Izračunaj `y` kordinato Geronove lemniskate."""
    lemniskata_y(t) = 2t * (t^2 - 1) / (t^2 + 1)^2

    # izvozimo imeni funkcij, da sta dostopni brez predpone `Vaja01`
    export lemniskata_x, lemniskata_y
end # module Vaja01
```

Program 1: Definiciji funkcij v paketu `Vaja01`

Funkcije iz datoteke `Vaja01/src/Vaja01.jl` uvozimo z ukazom `using Vaja01`, če smo paket `Vaja01` dodali v projektno okolje (`Project.toml`). V mapo `src` sodijo splošno uporabne funkcije, ki jih želimo uporabiti v drugih programih. V interaktivni zanki pokličemo novo definirano funkcijo:

```
julia> using Vaja01
julia> lemniskata_x(1.2)
0.180327868852459
```

V datoteko `Vaja01/doc/01uvod.jl` zapišemo preprost program, ki uporabi kodo iz paketa `Vaja01` in nariše lemniskato:

```
using Vaja01
# Krivuljo narišemo tako, da koordinati tabeliramo za veliko število parametrov.
t = range(-5, 5, 300) # generiramo 300 enakomerno razporejenih vrednosti na [-5, 5]
x = lemniskata_x.(t) # funkcijo apliciramo na elemente zaporedja
y = lemniskata_y.(t) # tako da imenu funkcije dodamo .
# Za risanje grafov uporabimo paket `Plots`.
using Plots
plot(x, y, label=false, title="Geronova lemniskata")
```

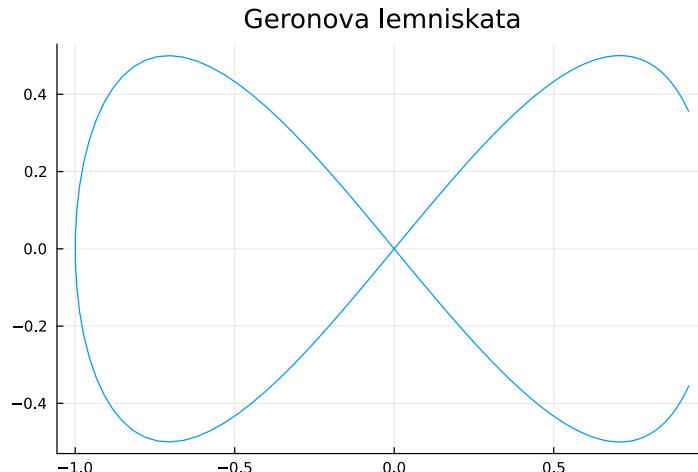
Program `01uvod.jl` poženemo z ukazom:

```
julia> include("Vaja01/doc/01uvod.jl")
```

Poganjanje ukaz za ukazom v VS Code

Če uporabljate urejevalnik [VS Code](#) in [razširitev za Julio](#), lahko ukaze iz programa poganjate vrstico za vrstico kar iz urejevalnika. S pritiskom kombinacije tipk Shift + Enter se bo izvedla vrstica, v kateri je trenutno kazalka.

Rezultat je slika lemniskate.



Slika 2: Geronova lemniskata

1.7 Testi

V naslednjem koraku dodamo avtomatske teste, s katerimi preizkusimo pravilnost kode, ki smo jo napisali v prejšnjem poglavju. Avtomatski test je preprost program, ki pokliče določeno funkcijo in preveri pravilnost rezultata.

Avtomatsko testiranje programov

Pomembno je, da pravilnost programov preverimo. Najlažje to naredimo „na roke“, tako da program poženemo in preverimo rezultat. Testiranje „na roke“ ima veliko pomankljivosti. Zahteva veliko časa, je lahko nekonsistentno in je dovezetno za človeške napake.

Alternativa ročnemu testiranju programov so avtomatski testi. To so preprosti programi, ki izvedejo testirani program in rezultate preverijo. Avtomatski testi so pomemben del [agilnega razvoja programske opreme](#) in omogočajo automatizacijo procesov razvoja programske opreme, ki se imenuje [nenehna integracija](#).

Uporabili bomo paket [Test](#), ki olajša pisanje testov. Vstopna točka za teste je datoteka `test/runtests.jl`. Uporabili bomo makroje `@test` in `@testset` iz paketa `Test`.

V datoteko `test/runtests.jl` dodamo testa za obe koordinatni funkciji, ki primerjata izračunane vrednosti s pravimi vrednostmi, ki smo jih izračunali „na roke“:

```

using Vaja01, Test

@testset "Koordinata x" begin
    @test lemniskata_x(1.0) ≈ 0.0
    @test lemniskata_x(2.0) ≈ 3 / 5
end

@testset "Koordinata y" begin
    @test lemniskata_y(1.0) ≈ 0.0
    @test lemniskata_y(2.0) ≈ 12 / 25
end

```

Program 2: Test funkcij lemniskata_x in lemniskata_y

Primerjava števil s plavajočo vejico

Pri računanju s števili s plavajočo vejico se izogibajmo primerjanju števil z operatorjem `==`, ki števili primerja bit po bit. Pri izračunih, v katerih nastopajo števila s plavajočo vejico, namreč pride do zaokrožitvenih napak. Različni načini izračuna za isto število se zato praviloma razlikujejo na zadnjih decimalkah. Na primer izraz `asin(sin(pi/4)) - pi/4` ne vrne točne ničle, temveč zelo majhno vrednost `-1.1102230246251565e-16`. Za približno primerjavo dveh vrednosti `a` in `b` zato uporabimo izraz

$$|a - b| < \varepsilon,$$

kjer je ε večji od pričakovane zaokrožitvene napake. V Julii lahko za približno primerjavo števil in vektorjev uporabimo operator `≈`, ki je alias za funkcijo `isapprox`.

Preden poženemo teste, ustvarimo testno okolje. Sledimo [priporočilom za testiranje paketov](#). V mapi `Vaja01/test` ustvarimo novo okolje in dodamo paket `Test`:

```

(@v1.10) pkg> activate Vaja01/test
(test) pkg> add Test
(test) pkg> activate .

```

Teste poženemo tako, da v paketnem načinu poženemo ukaz `test Vaja01`.

```

(num_mat) pkg> test Vaja01
Testing Vaja01
    Testing Running tests
    ...
    ...
Test Summary: | Pass  Total  Time
Koordinata x |    2      2  0.1s
Test Summary: | Pass  Total  Time
Koordinata y |    2      2  0.0s
    Testing Vaja01 tests passed

```

1.8 Dokumentacija

Dokumentacija programske kode je sestavljena iz različnih besedil in drugih virov, npr. videov, ki so namenjeni uporabnikom in razvijalcem programa ali knjižnice. Dokumentacija vključuje komentarje

v kodi, navodila za namestitev in uporabo programa ter druge vire z razlagami ozadja, teorije in drugih zadev, povezanih s projektom. Dobra dokumentacija lahko veliko pripomore k uspehu določenega programa. To še posebej velja za knjižnice.

Slabo dokumentirane kode ne želi nihče uporabljati. Tudi če vemo, da kode ne bo uporabljal nihče drug razen nas samih, bodimo prijazni do samega sebe v prihodnosti in pišimo dobro dokumentacijo.

V tej knjigi bomo pisali tri vrste dokumentacije:

- dokumentacijo za posamezne funkcije v sami kodi,
- navodila za uporabnika v datoteki README.md,
- poročilo v formatu PDF.

Zakaj format PDF

Izbira formata PDF je mogoče presenetljiva za pisanje dokumentacije programske kode. V praksi so precej uporabnejše HTML strani. Dokumentacija v obliki HTML strani, ki se avtomatično ustvari v procesu [nenehne integracije](#), je postala *de facto* standard. V kontekstu popravljanja domačih nalog in poročil za vaje pa ima format PDF še vedno prednosti, saj ga je lažje pregledovati in popravljati.

1.8.1 Dokumentacija funkcij in tipov

Funkcije in podatkovne tipe v Julii dokumentiramo tako, da pred definicijo dodamo niz z opisom funkcije, kot smo to naredili v programu Program 1. Več o tem si lahko preberete [v poglavju o dokumentaciji](#) priročnika za Julio.

1.8.2 README dokument

Dokument README (preberi me) je namenjen najosnovnejšim informacijam o paketu. Dokument je vstopna točka za dokumentacijo in navadno vsebuje:

- kratek opis projekta,
- povezavo na dokumentacijo,
- navodila za osnovno uporabo in
- navodila za namestitev.

Vzorčni projekt za vajo

Avtor: Martin Vuk <martin.vuk@fri.uni-lj.si>

Preprost paket, ki definira koordinatne funkcije [Geronove lemniskate](https://sl.wikipedia.org/wiki/Geronova_lemniskata). Primer uporabe je opisan v programu [01uvod.jl]([doc/01uvod.jl](#)), ki ga poženemo z ukazom

```
```jl
include("Vaja01/doc/01uvod.jl")
```
v interaktivni zanki Julie.
```

Testi

Teste poženemo z ukazom:

```
```
julia --project=Vaja01 -e "import Pkg; Pkg.test()"
```
```

Poročilo PDF

Poročilo pripravimo z ukazom:

```
```
julia --project=@. Vaja01/doc/makedocs.jl
```
```

Program 3: Vsebina datoteke README.md, ki vsebuje osnove informacije o projektu.

1.8.3 PDF poročilo

V nadaljevanju bomo opisali, kako poročilo pripraviti s paketom [Weave.jl](#). Paket Weave.jl omogoča mešanje besedila in programske kode v enem dokumentu: [literarnemu programu](#), kot ga je opisal D. E. Knuth [5]. Za pisanje besedila bomo uporabili format [Markdown](#), ki ga bomo dodali v komentarje h kodi.

Za ustvarjanje PDF dokumentov je treba namestiti [TeX/LaTeX](#). Priporočam namestitev [TinyTeX](#) ali [TeX Live](#), ki pa zasede več prostora na disku. Po [namestitvi](#) programa TinyTeX moramo dodati še nekaj LaTeX paketov, ki jih potrebuje paket Weave. V terminalu izvedemo naslednji ukaz

```
$ tlmgr install microtype upquote minted
```

Poročilo pripravimo v obliki literarnega programa. Uporabili bom kar datoteko `Vaja01/doc/01uvod.jl`, s katero smo pripravili sliko. V datoteko dodamo besedilo v obliki komentarjev. Če želimo, da se komentarji uporabijo kot besedilo v formatu [Markdown](#), uporabimo `#'`. Koda in navadni komentarji se v poročilu izpišejo nespremenjeni.

```

#' # Geronova lemniskata
#' Komentarji, ki se začnejo z `#`' se uporabijo kot Markdown in
#' v PDF dokumentu nastopajo kot besedilo.
using Vaja01
# Krivuljo narišemo tako, da koordinati tabeliramo za veliko število parametrov.
t = range(-5, 5, 300) # generiramo 300 enakomerno razporejenih vrednosti na [-5,
5]
x = lemniskata_x.(t) # funkcijo apliciramo na elemente zaporedja
y = lemniskata_y.(t) # tako da imenu funkcije dodamo .
# Za risanje grafov uporabimo paket `Plots`.
using Plots
plot(x, y, label=false, title="Geronova lemniskata")
#' Zadnji rezultat pred besedilom, označenim z `#`', se vstavi v dokument.
#' Če je rezultat graf, se v dokument vstavi slika z grafom.

```

Program 4: Vsebina datoteke `01uvod.jl`, iz katere ustvarimo poročilo. Vrstice, ki se začnejo z znakoma `#'`, so v formatu Markdown in bodo v poročilo vključene kot oblikovano besedilo.

Poročilo pripravimo z ukazom `Weave.weave`. Ustvarimo program `Vaja01/doc/makedocs.jl`, ki pripravi pdf dokument:

```

using Weave
# Poročilo generiramo z ukazom `Weave.weave`
Weave.weave("Vaja01/doc/01uvod.jl",
doctype="minted2pdf", out_path="Vaja01/pdf")

```

Program 5: Program za pripravo PDF dokumenta

Program poženemo z ukazom `include("Vaja01/doc/makedocs.jl")` v Julii. Preden poženemo program `makedocs.jl`, moramo projektnemu okolju `num_mat` dodati paket `Weave.jl`.

```

(num_mat) pkg> add Weave
julia> include("Vaja01/doc/makedocs.jl")

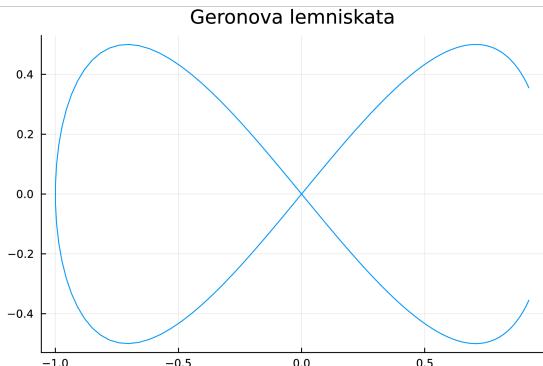
```

Poročilo se shrani v datoteko `Vaja01/pdf/01uvod.pdf`.

1 Geronova lemniskata

Komentarji, ki se začnejo z `#!` se uporabijo kot Markdown in v PDF dokumentu nastopajo kot besedilo.

```
using Vaja01
# Krivuljo narišemo tako, da koordinati tabeliramo za veliko število parametrov.
t = range(-5, 5, 300) # generiramo 300 enakomerno razporejenih vrednosti na [-5, 5]
x = lemniskata_x.(t) # funkcijo apliciramo na elemente zaporedja
y = lemniskata_y.(t) # tako da imenu funkcije dodamo .
# Za risanje grafov uporabimo paket "Plots".
using Plots
plot(x, y, label=false, title="Geronova lemniskata")
```



Zadnji rezultat pred besedilom, označenim z `#!`, se vstavi v dokument. Če je rezultat graf, se v dokument vstavi slika z grafov.

1

Slika 3: Poročilo v PDF formatu

Alternativni paketi za pripravo PDF dokumentov

Poleg paketa `Weave.jl` je na voljo še nekaj programov, ki so primerni za pripravo PDF dokumentov s programi v Julii:

- vmesnika v obliki zvezka [Julia](#) in [Pluo.jl](#),
- [Literate.jl](#),
- [TypstJlyfish.jl](#) in
- [Quadro](#).

Če potrebujemo več nadzora pri pripravi PDF dokumenta, priporočam uporabo naslednjih programov:

- [TeX/LaTeX](#),
- [pandoc](#),
- [AsciiDoctor](#) ali
- [Typst](#).

Povezave na temo pisanja dokumentacije

- Pisanje dokumentacije v jeziku Julia.
- Priporočila za stil za programski jezik Julia.
- Documenter.jl je najbolj razširjen paket za pripravo dokumentacije v Julii.
- Diátaxis je sistematičen pristop k pisanju dokumentacije.
- Dokumentacija kot koda je način dela, pri katerem z dokumentacijo ravnamo na enak način kot s kodo.

1.9 Zaključek

Ustvarili smo svoj prvi paket, ki vsebuje kodo, avtomatske teste in dokumentacijo. Mapa Vaja01 bi morala imeti naslednjo strukturo:

```
$ tree Vaja01
Vaja01
├── Manifest.toml
├── Project.toml
├── README.md
└── doc
    ├── 01uvod.jl
    └── makedocs.jl
└── src
    └── Vaja01.jl
└── test
    ├── Manifest.toml
    ├── Project.toml
    └── runtests.jl
```

Preden nadaljuješ, ponovno preveri, ali vse deluje tako, kot bi moralo. V Julii aktiviraj projektno okolje:

```
julia> # pritisnite ] za vstop v paketni način
(@v1.10) pkg> activate .
```

Nato poženi teste:

```
(num_mat) pkg> test Vaja01
...
Testing Vaja01 tests passed
```

Na koncu pa poženi še program 01uvod.jl:

```
julia> include("Vaja01/doc/01uvod.jl")
```

in pripravi poročilo:

```
julia> include("Vaja01/doc/makedocs.jl")
```

Priporočam, da si pred branjem naslednjih poglavij vzameš čas in poskrbiš, da se zgornji ukazi izvedejo brez napak.

2 Računanje kvadratnega korena

Računalniški procesorji navadno implementirajo le osnovne aritmetične operacije: seštevanje, množenje in deljenje. Za izračun vrednosti drugih matematičnih funkcij mora nekdo napisati program. Večina programskih jezikov vsebuje implementacijo elementarnih funkcij v standardni knjižnici. V tej vaji si bomo ogledali, kako implementirati korenko funkcijo.

Implementacija elementarnih funkcij v Julii

Lokacijo metod, ki računajo določeno funkcijo, dobimo z ukazoma `methods` in `@which`. Ukaz `methods(sqrt)` izpiše implementacije kvadratnega korena za vse podatkovne tipe, ki jih Julia podpira, ukaz `@which(sqrt(2.0))` pa razkrije metodo, ki računa koren za vrednost 2.0, to je za števila s plavajočo vejico.

2.1 Naloga

Napiši funkcijo $y = \text{koren}(x)$, ki bo izračunala približek za kvadratni koren števila x . Poskrbi, da bo rezultat pravilen na 10 decimalnih mest in da bo časovna zahtevnost neodvisna od argumenta x .

- Zapiši enačbo, ki ji zadošča kvadratni koren.
- Uporabi [Newtonovo metodo](#) in izpelji [Heronovo rekurzivno formulo](#) za računanje kvadratnega korena.
- Kako je konvergenca odvisna od vrednosti x ?
- Nariši graf potrebnega števila korakov v odvisnosti od argumenta x .
- Uporabi lastnosti [zapisa s plavajočo vejico](#) in izpelji formulo za približno vrednost korena, ki uporabi eksponent (funkcija `exponent` v Julii).
- Implementiraj funkcijo `koren(x)`, tako da je časovna zahtevnost neodvisna od argumenta x . Grafično preveri, ali funkcija dosega zahtevano natančnost za poljubne vrednosti argumenta x .

Preden se lotimo reševanja, ustvarimo projekt za trenutno vajo in ga dodamo v delovno okolje.

```
(num_mat) pkg> generate Vaja02
(num_mat) pkg> develop Vaja02/
```

Tako bomo imeli v delovnem okolju dostop do vseh funkcij, ki jih bomo definirali v paketu `Vaja02`.

2.2 Izbira algoritma

Z računanjem kvadratnega korena so se ukvarjali že pred 3500 leti v Babilonu. O tem si lahko več preberete v članku v reviji Presek [6]. Če želimo poiskati algoritem za računanje kvadratnega korena, se moramo najprej vprašati, kaj sploh je kvadratni koren. Kvadratni koren števila x je definiran kot pozitivna vrednost y , katere kvadrat je enak x . Število y je torej pozitivna rešitev enačbe:

$$y^2 = x. \quad (2.1)$$

Da bi poiskali vrednost \sqrt{x} , moramo rešiti [nelinearno enačbo \(2.1\)](#). Za numerično reševanje nelinearnih enačb obstaja celo vrsta metod. Ena najpopularnejših je [Newtonova ali tangentna metoda](#), ki jo bomo uporabili tudi mi. Pri Newtonovi metodi rešitev enačbe

$$f(x) = 0 \quad (2.2)$$

poiščemo z rekurzivnim zaporedjem približkov:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}. \quad (2.3)$$

Če zaporedje (2.3) konvergira, potem konvergira k rešitvi enačbe $f(x) = 0$.

Enačbo (2.1) najprej preoblikujemo v obliko, ki je primerna za reševanje z Newtonovo metodo. Prema-knemo vse člene na eno stran in dobimo:

$$y^2 - x = 0. \quad (2.4)$$

V formulo za Newtonovo metodo vstavimo funkcijo $f(y) = y^2 - x$ in odvod $f'(y) = \frac{d}{dy}f(y) = 2y$, da dobimo:

$$\begin{aligned} y_{n+1} &= y_n - \frac{y_n^2 - x}{2y_n} = \frac{2y_n^2 - y_n^2 + x}{2y_n} = \frac{1}{2} \left(\frac{y_n^2 + x}{y_n} \right) \\ y_{n+1} &= \frac{1}{2} \left(y_n + \frac{x}{y_n} \right). \end{aligned} \quad (2.5)$$

Rekurzivno formulo (2.5) imenujemo **Heronov obrazec**. Zgornja formula določa zaporedje, ki vedno konvergira bodisi k \sqrt{x} ali $-\sqrt{x}$, odvisno od izbire začetnega približka. Poleg tega, da zaporedje hitro konvergira k limiti, je program izjemno preprost. Poglejmo, kako izračunamo $\sqrt{2}$:

```
julia> y = 1.5
      x = 2
      for n = 1:5
          y = (y + x / y) / 2
          println(y)
      end
1.4166666666666665
1.4142156862745097
1.4142135623746899
1.414213562373095
1.414213562373095
```

Vidimo, da se približki začnejo ponavljati že po 4. koraku. To pomeni, da se zaporedje ne bo več spremenjalo in smo dosegli najboljši približek, kot ga lahko predstavimo s 64-bitnimi števili s plavajočo vejico.

Napišimo zgornji program še kot funkcijo. Da lažje spremojamo, kaj se dogaja med izvajanjem kode, uporabimo makro @info iz modula **Logging**, ki je del standardne knjižnice.

```

using Logging
"""

y = koren_heron(x, x0, n)

Izračuna približek za koren števila `x` z `n` koraki Heronovega obrazca z začetnim
približkom `x0`.

"""

function koren_heron(x, x0, n)
    y = x0
    for i = 1:n
        y = (y + x / y) / 2
        @info "Približek na koraku $i je $y"
    end
    return y
end

```

Program 6: Funkcija, ki računa kvadratni koren s Heronovim obrazcem.

Preskusimo funkcijo `koren_heron` na številu 3.

```

x = koren_heron(3, 1.7, 5)
println("Koren 3 je $(x).")

[ Info: Približek na koraku 1 je 1.7323529411764707
[ Info: Približek na koraku 2 je 1.7320508339159093
[ Info: Približek na koraku 3 je 1.7320508075688776
[ Info: Približek na koraku 4 je 1.7320508075688772
[ Info: Približek na koraku 5 je 1.7320508075688772
Koren 3 je 1.7320508075688772.

```

Metoda navadne iteracije in tangentna metoda

Metoda računanja kvadratnega korena s Heronovim obrazcem je poseben primer **tangentne metode**, ki je poseben primer **metode fiksne točke**. Obe metodi si bomo podrobnejše ogledali kasneje.

2.3 Določitev števila korakov

Funkcija `koren_heron(x, x0, n)` ni uporabna za splošno rabo, saj mora uporabnik poznati tako začetni približek kot tudi število korakov, ki so potrebni, da dosežemo želeno natančnost. Da bi bila funkcija zares uporabna, bi morala sama izbrati začetni približek in število potrebnih korakov. Najprej se bomo naučili poiskati dovolj veliko število korakov, da dosežemo želeno natančnost.

Kako vemo, kdaj smo dosegli želeno natančnost? Navadno nekako ocenimo napako približka in jo primerjamo z želeno natančnostjo. To lahko storimo na dva načina:

- preverimo, ali je absolutna napaka manjša od **absolutne tolerance** ali
- preverimo, ali je relativna napaka manjša od **relativne tolerance**.

Julia za namen primerjave dveh števil ponuja funkcijo `isapprox`, ki pove ali sta dve vrednosti približno enaki. Funkcija `isapprox` omogoča relativno in absolutno primerjavo vrednosti. Primerjava števil z relativno toleranco δ se prevede na neenačbo:

$$|a - b| < \delta(\max(|a|, |b|)). \quad (2.6)$$

Previdno pri primerjanju s številom nič

Ko uporabljamo relativno primerjavo, moramo biti previdni, če primerjamo vrednosti s številom 0. Če je namreč eno od števil, ki ju primerjamo, enako 0 in $\delta < 1$, potem neenačba (2.6) nikoli ni izpolnjena.

Število 0 nikoli ni približno enako nobenemu neničelnemu številu, če ju primerjamo z relativno toleranco.

Število pravilnih decimalnih mest

Ko govorimo o številu pravilnih decimalnih mest, imamo navadno v mislih število signifikantnih mest v zapisu s plavajočo vejico. V tem primeru moramo poskrbeti, da je relativna napaka dovolj majhna. Če želimo, da bo pravilnih 10 signifikantnih mest, mora biti relativna napaka manjša od $5 \cdot 10^{-11}$. Naslednja števila so vsa podana s 5 signifikantnimi mesti:

$$\begin{aligned}\frac{1}{70} &\approx 0.014285, \quad \frac{1}{7} \approx 0.14285 \\ \frac{10}{7} &\approx 1.4285, \quad \frac{10^{10}}{7} \approx 1428500000.\end{aligned}\tag{2.7}$$

Pri iskanju kvadratnega korena napako ocenimo tako, da primerjamo kvadrat približka z danim argumentom. Pri tem je treba raziskati, kako sta povezani relativni napaki približka za koren in njegovega kvadrata. Naj bo y točna vrednost kvadratnega korena \sqrt{x} . Če je \hat{y} približek z relativno napako δ , potem je $\hat{y} = y(1 + \delta)$. Poglejmo, kako je relativna napaka δ povezana z relativno napako kvadrata \hat{y}^2 :

$$\varepsilon = \frac{\hat{y}^2 - x}{x} = \frac{(y(1 + \delta))^2 - x}{x} = \frac{x(1 + \delta)^2 - x}{x} = (1 + \delta)^2 - 1 = 2\delta + \delta^2.\tag{2.8}$$

Pri tem smo upoštevali, da je $\hat{y}^2 = x$. Relativna napaka kvadrata je enaka $\varepsilon = 2\delta + \delta^2$. Ker je $\delta^2 \ll \delta$, dobimo dovolj natančno oceno, če δ^2 zanemarimo:

$$\delta = \frac{1}{2}(\varepsilon - \delta^2) < \frac{\varepsilon}{2}.\tag{2.9}$$

Od tod dobimo pogoj, kdaj je približek dovolj natančen. Če je

$$|\hat{y}^2 - x| < 2\delta \cdot x,\tag{2.10}$$

potem velja začetna zahteva:

$$|\hat{y} - \sqrt{x}| < \delta \cdot \sqrt{x}.\tag{2.11}$$

Ocene za napako ni vedno enostavno poiskati

V primeru računanja kvadratnega korena je bila analiza napak relativno enostavna in smo lahko dobili točno oceno za relativno napako metode. Večinoma ni tako. Točne ocene za napako ni vedno lahko ali sploh mogoče poiskati. Zato pogosto v praksi napako ocenimo na podlagi različnih indicev brez zagotovila, da je ocena točna.

Pri iterativnih metodah konstruiramo zaporedje približkov x_n , ki konvergira k iskanemu številu. Razlika med dvema zaporednima približkoma $|x_{n+1} - x_n|$ je pogosto dovolj dobra ocena za napako iterativne metode. Toda zgolj dejstvo, da je razlika med zaporednima približkoma majhna, še ne zagotavlja, da je razlika do limite prav tako majhna. Če poznamo oceno za hitrost konvergence (ozziroma odvod iteracijske funkcije), lahko izpeljemo zvezo med razliko dveh sosednjih približkov in napako metode. Vendar se v praksi pogosto zanašamo, da sta razlika sosednjih približkov in napaka sorazmerni. Problem nastane, če je konvergenca počasna.

Uporabimo pogoj (2.11) in napišemo funkcijo, ki sama določi število korakov iteracije:

```
'''  
y = koren(x, y0)
```

Izračunaj vrednost kvadratnega korena števila `x` s Heronovim obrazcem z začetnim približkom `y0`.

```
'''  
  
function koren(x, y0)  
    if x == 0.0  
        # Vrednost 0 obravnavamo posebej, saj je relativna primerjava z 0  
        # problematična  
        return 0.0  
    end  
    delta = 5e-11 # zahtevana relativna natančnost rezultata  
    maxit = 10 # 10 korakov je dovolj, če je začetni približek dober  
    for i = 1:maxit  
        y = (y0 + x / y0) / 2  
        if abs(x - y^2) <= 2 * delta * abs(x)  
            @info "Število korakov $i"  
            return y  
        end  
        y0 = y  
    end  
    throw("Iteracija ne konvergira!")  
end
```

Program 7: Metoda `koren(x, y0)`, ki avtomatsko določi število korakov iteracije, da dosežemo zahtevano natančnost.

2.4 Izbira začetnega približka

Kako bi učinkovito izbrali dober začetni približek? Dokazati je mogoče, da rekurzivno zaporedje (2.5) konvergira ne glede na izbran začetni približek. Vendar je število korakov iteracije večje, dlje kot je začetni približek oddaljen od rešitve. Če želimo, da bo časovna zahtevnost funkcije neodvisna od argumenta, moramo poskrbeti, da za poljubni argument uporabimo dovolj dober začetni približek. Za začetni približek lahko uporabimo kar samo število x . Malce boljši približek dobimo s Taylorjevim razvojem (tangento) korenske funkcije okrog števila 1:

$$\sqrt{x} = 1 + \frac{1}{2}(x - 1) + \dots \approx \frac{1}{2} + \frac{x}{2}. \quad (2.12)$$

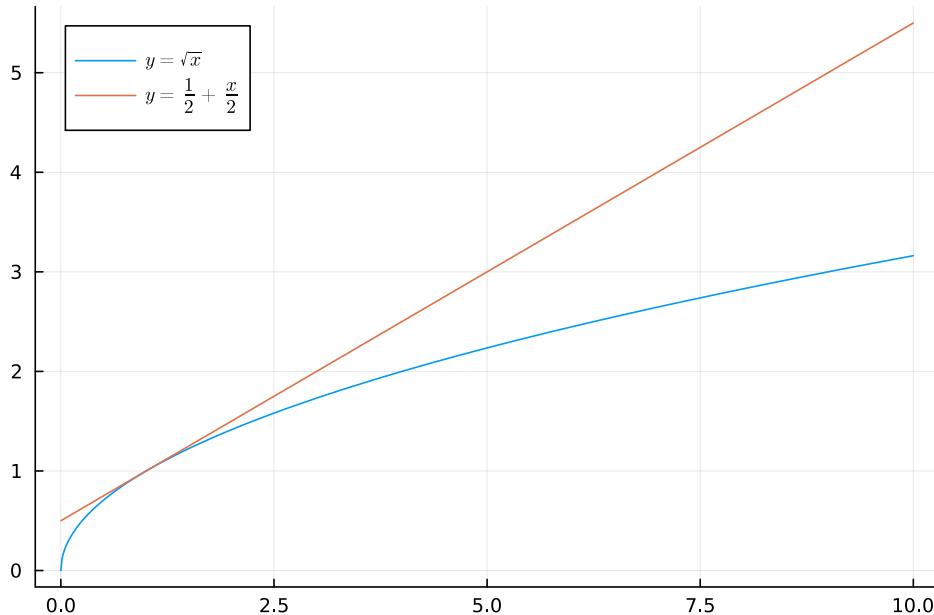
Opazimo, da za večja števila iteracija potrebuje več korakov:

```
julia> tangenta(x) = 0.5 + x / 2
julia> y = koren(10, tangenta(10))
[ Info: Število korakov 5
3.162277660168379

julia> y = koren(1000, tangenta(1000))
[ Info: Število korakov 8
31.622776601684336
```

Začetni približek $\frac{1}{2} + \frac{x}{2}$ dobro deluje za števila blizu 1. Če isto formulo za začetni približek uporabimo na večjih številih, dobimo večjo relativno napako oziroma potrebujemo več korakov zanke, da pridemo do enake natančnosti. Na isti graf narišimo korensko funkcijo in tangento $\frac{1}{2} + \frac{x}{2}$:

```
using Plots
plot(sqrt, 0, 10, label="\$y=\sqrt{x}\$")
plot!(x -> 0.5 + x / 2, 0, 10, label="\$y=\frac{1}{2}+\frac{x}{2}\$")
```



Slika 4: Korenska funkcija in tangentna $\frac{1}{2} + \frac{x}{2}$ v točki $x = 1$

Za boljši približek si pomagamo z načinom predstavitev števil v računalniku. Realna števila predstavimo s [števili s plavajočo vejico](#). Število je zapisano v obliki

$$x = m2^e, \quad (2.13)$$

kjer je $1 \leq m < 2$ mantisa, e pa eksponent. Za 64-bitna števila s plavajočo vejico se za zapis mantise uporabi 53 bitov (52 bitov za decimalke, en bit pa za predznak), 11 bitov pa za eksponent (glej [IEE 754 standard](#)). Koren števila x potem izračunamo kot:

$$\sqrt{x} = \sqrt{m} \cdot 2^{\frac{e}{2}}. \quad (2.14)$$

Koren mantise, ki leži na $[1, 2)$, približno ocenimo s tangento v $x = 1$

$$\sqrt{m} = \frac{1}{2} + \frac{m}{2}. \quad (2.15)$$

Če eksponent delimo z 2 in upoštevamo ostanek $o = e - 2d$, vrednost $\sqrt{2^e}$ zapišemo kot:

$$\sqrt{2^e} \approx 2^d \cdot \begin{cases} 1, & o = 0, \\ \sqrt{2}, & o = 1. \end{cases} \quad (2.16)$$

Formula za približek je enaka:

$$\sqrt{x} \approx \left(\frac{1}{2} + \frac{m}{2} \right) \cdot 2^d \cdot \begin{cases} 1, & o = 0, \\ \sqrt{2}, & o = 1. \end{cases} \quad (2.17)$$

Potenco števila 2^n izračunamo s premikom binarnega zapisa števila 1 v levo za n mest. Julia ima funkcijo `ldexp(x, n)`, ki uporabi premik eksponenta in učinkovito pomnoži x s potenco 2^n . Funkciji exponent in significand pa vrneta eksponent in mantiso števila s plavajočo vejico. Tako lahko zapišemo naslednjo funkcijo za začetni približek:

```
"""
y0 = začetni(x)

Izračunaj začetni približek za kvadratni koren števila `x` z uporabo
eksponenta za števila s plavajočo vejico.

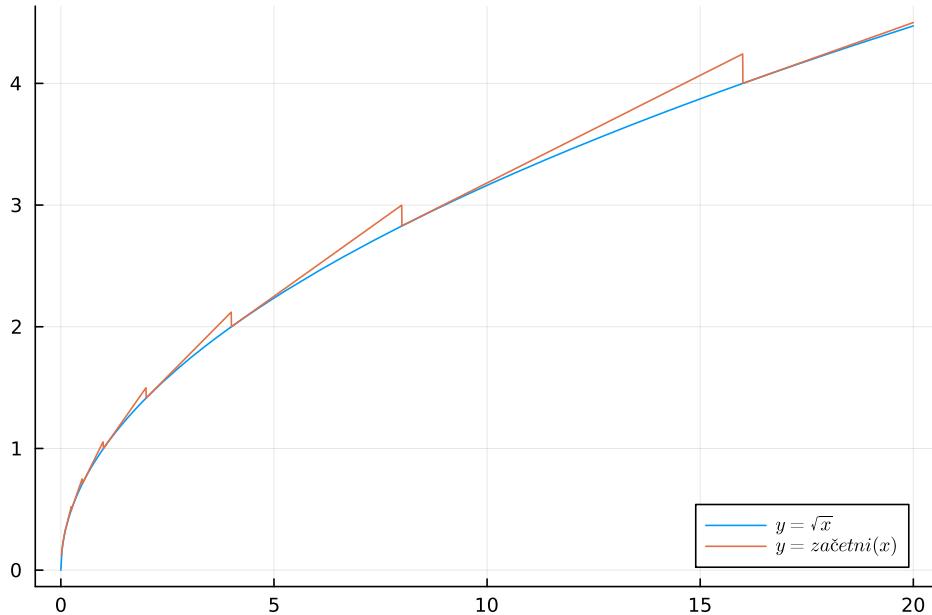
"""

function začetni(x)
    d, ost = divrem(exponent(x), 2)
    m = significand(x) # mantisa
    koren2ost = 1.0
    if (ost == 1)
        koren2ost = 1.4142135623730951 # koren(2)
    elseif (ost == -1)
        koren2ost = 0.7071067811865475 # 1/koren(2)
    end
    koren2e = ldexp(koren2ost, d) # koren(2^e) = koren(2^ost) * 2^d
    return (0.5 + m / 2) * koren2e
end
```

Program 8: Funkcija `začetni(x)`, ki izračuna začetni približek.

Primerjajmo izboljšano verzijo začetnega približka s pravo korensko funkcijo:

```
plot(sqrt, 0, 20, label="$y=\sqrt{x}$")
plot!(Vaja02.začetni, 0, 20, label="$y = začetni(x)$")
```



Slika 5: Korenska funkcija in izboljšani začetni približek

2.5 Zaključek

Ko izberemo dober začetni približek, Newtonova iteracija hitreje konvergira, ne glede na velikost argumenta. Tako lahko definiramo metodo `koren(x)` brez dodatnega argumenta.

```
''''
y = koren(x)
```

Izračunaj vrednost kvadratnega korena danega števila `x`.

```
''''
koren(x) = koren(x, začetni(x))
```

Program 9: Funkcija `koren(x)`

Julia omogoča več definicij iste funkcije

Julia uporablja posebno vrsto **polimorfizma** imenovano **večlična razdelitev** (angl. multiple dispatch). Za razliko od polimorfizma v objektno usmerjenih jezikih, kjer se metoda izbere le na podlagi razreda objekta, ki to metodo kliče, se v Julii metoda izbere na podlagi tipov vseh vhodnih argumentov. Ta lastnost omogoča pisanje generične kode, ki deluje za zelo različne vhodne argumente.

Večlična razdelitev omogoča, da za isto funkcijo definiramo več različic, ki se uporabijo glede na argumente podane funkciji. Tako smo definirali dve metodi za funkcijo `koren`. Prva metoda sprejme 2 argumenta, druga pa en argument. Ko pokličemo `koren(2.0, 1.0)`, se izvede različica Program 7, ko pa pokličemo `koren(2.0)`, se izvede Program 9.

Metode, ki so definirane za neko funkcijo `fun`, lahko vidimo z ukazom `methods(fun)`. Metodo, ki se uporabi za določen klic funkcije, poiščemo z makrojem `@which`, npr. `@which koren(2.0, 1.0)`.

Opazimo, da se število korakov z naraščanjem argumenta ne spreminja, kar pomeni, da je časovna zahtevnost funkcije `koren(x)` neodvisna od izbire argumenta.

```
julia> koren(2.0), koren(200.0), koren(2e10)
[ Info: Število korakov 1
[ Info: Število korakov 3
[ Info: Število korakov 2
(1.414213562373095, 14.142135623730965, 141421.35623853415)
```

Hitro računanje obratne vrednosti kvadratnega korena

Pri razvoju računalniških iger, ki poskušajo verodostojno prikazati 3-dimenzionalni svet na zaslonu, se veliko uporablja normiranje vektorjev. Pri normiraju je treba komponente vektorja deliti z normo vektorja, ki je enaka korenui vsote kvadratov komponent. Kot smo spoznali pri računanju kvadratnega korena s Heronovim obrazcem, je posebej problematično najti ustrezni začetni približek, ki je dovolj blizu pravi rešitvi. Tega problema so se zavedali tudi inženirji igre Quake, ki so razvili posebej zvit, skoraj magičen način za izračun funkcije $\frac{1}{\sqrt{x}}$. Metoda uporabi posebno vrednost `0x5f3759df`, da pride do dobrega začetnega približka, nato pa še en korak [Newtonove metode](#). Več o [računanju obratne vrednosti kvadratnega korena](#).

Kaj smo se naučili?

- Tudi za izračun preprostih funkcij potrebujemo numerični algoritem.
- Pri iterativnih metodah je pomembna izbira dobrega začetnega približka.
- Numerični algoritmi so pogosto preprosti, vendar moramo paziti, da je napaka omejena.

3 Tridiagonalni sistemi

3.1 Naloge

- Ustvari podatkovni tip za tridiagonalno matriko in implementiraj operacije množenja $*$ z vektorjem ter reševanja sistema $Ax = b$ z operatorjem \.
- Za slučajni sprehod v eni dimenziji izračunaj povprečno število korakov, ki jih potrebujemo, da se od izhodišča oddaljimo za k korakov.
 - ▶ Zapiši fundamentalno matriko za [Markovsko verigo](#), ki modelira slučajni sprehod, ki se lahko od izhodišča oddalji le za k korakov.
 - ▶ Reši sistem s fundamentalno matriko in vektorjem enic.
 - ▶ Povprečno število korakov oceni še z vzorčenjem velikega števila simulacij slučajnega sprehoda.
 - ▶ Primerjaj oceno z rešitvijo sistema.

3.2 Tridiagonalne matrike

Matrika je *tridiagonalna*, če ima neničelne elemente le na glavni diagonali in na dveh najbližjih diagonalah. Primer 5×5 tridiagonalne matrike:

$$\begin{pmatrix} 1 & 2 & 0 & 0 & 0 \\ 3 & 4 & 5 & 0 & 0 \\ 0 & 6 & 7 & 6 & 0 \\ 0 & 0 & 5 & 4 & 3 \\ 0 & 0 & 0 & 2 & 1 \end{pmatrix}. \quad (3.1)$$

Elementi tridiagonalne matrike, za katere se indeksa razlikujeta za več kot 1, so vsi enaki 0:

$$|i - j| > 1 \Rightarrow a_{ij} = 0. \quad (3.2)$$

Z implementacijo posebnega podatkovnega tipa za tridiagonalno matriko prihranimo tako na prostoru kot tudi pri časovni zahtevnosti algoritmov, saj jih lahko prilagodimo posebnim lastnostim tridiagonalnih matrik.

Preden se lotimo naloge, ustvarimo nov paket `Vaja03`, kamor bomo postavili kodo:

```
(num_mat) pkg> generate Vaja03
(num_mat) pkg> develop Vaja03/
```

Podatkovni tip za tridiagonalne matrike imenujemo `Tridiag` in vsebuje tri polja z elementi na posameznih diagonalah. Definicijo postavimo v `Vaja03/src/Vaja03.jl`:

```

"""
Tridiag(sd, d, zd)

Sestavi tridiagonalno matriko iz prve poddiagonale `sd`, glavne diagonale `d`
in prve naddiagonale `zd`. Rezultat je tipa `Tridiag`, ki hrani le neničelne
elemente matrike in omogoča učinkovito reševanje tridiagonalnega sistema
linearnih enačb. Dolžina vektorjev `sd` in `zd` mora biti za ena manj od dolžine
vektorja `d`.

"""
struct Tridiag
    sd::Vector # spodnja poddiagonala
    d::Vector # glavna diagonala
    zd::Vector # zgornja naddiagonala
    function Tridiag(sd, d, zd)
        if (length(sd) != length(d) - 1) || (length(zd) != length(d) - 1)
            error("Napačne dimenzijs diagonali.")
        end
        new(sd, d, zd)
    end
end
export Tridiag

```

Zgornja definicija omogoča, da ustvarimo nove objekte tipa `Tridiag`:

```
julia> using Vaja03
julia> Tridiag([3, 6, 5, 2], [1, 4, 7, 4, 1], [2, 5, 6, 3])
```

Preverjanje skladnosti polj v objektu

V zgornji definiciji `Tridiag` smo poleg deklaracije polj dodali tudi **notranji konstruktor** v obliki funkcije `Tridiag`. Vemo, da mora biti dolžina vektorjev `sd` in `zd` za ena manjša od dolžine vektorja `d`. Zato je pogoj najbolje preveriti, ko ustvarimo objekt, in se nam s tem v nadaljevanju ni več treba ukvarjati. Z notranjim konstruktorjem te pogoje uveljavimo ob nastanku objekta in preprečimo ustvarjanje objektov z nekonsistentnimi podatki.

Želimo, da se matrike tipa `Tridiag` obnašajo podobno kot generične matrike vgrajenega tipa `Matrix`. Zato funkcijam, ki delajo z matrikami, dodamo specifične metode za podatkovni tip `Tridiag`. Argumentu funkcije lahko dodamo informacijo o tipu, tako da dodamo `::Tip` in na ta način definiramo specifično metodo, ki deluje le za dan podatkovni tip. Če želimo, da metoda deluje za argumente tipa `Tridiag`, argumentu dodamo `::Tridiag`. Več informacij o **tipih** in **vmesnikih**.

Samostojno delo

Implementiraj naslednje metode, specifične za tip `Tridiag` (rešitve so na koncu poglavja):

- `size(T::Tridiag)` vrne dimenzijs matrike (Program 13),
- `getindex(T::Tridiag, i, j)` vrne element `T[i, j]` (Program 14),
- `setindex!(T::Tridiag, x, i, j)` nastavi element `T[i, j]` (Program 15) in
- `*(T::Tridiag, x::Vector)` izračuna produkt matrike `T` z vektorjem `x` (Program 16).

Za tridiagonalne matrike je časovna zahtevnost množenja matrike z vektorjem bistveno manjša kot v splošnem ($\mathcal{O}(n)$ namesto $\mathcal{O}(n^2)$).

Preden nadaljujemo, preverimo, ali so funkcije pravilno implementirane. Napišemo avtomatske teste, ki jih lahko kadarkoli poženemo. V projektu `Vaja03` ustvarimo datoteko `Vaja03/test/runtests.jl` in vanjo zapišemo kodo, ki preveri pravilnost zgoraj definiranih funkcij.

```
using Vaja03
using Test

@testset "Velikost" begin
    T = Tridiag([1, 2], [3, 4, 5], [6, 7])
    @test size(T) == (3, 3)
end
```

V paket `Vaja03` moramo dodati še paket `Test`:

```
(num_mat) pkg> activate Vaja03
(Vaja03) pkg> add Test
```

Teste poženemo v paketnem načinu z ukazom `test Vaja03`:

```
(Vaja03) pkg> activate .
(num_mat) pkg> test Vaja03
...
      Testing Running tests...
Test Summary: | Pass  Total  Time
Velikost     |    1      1  0.0s
```

Samostojno delo

Napiši teste za ostale funkcije (rešitve so v podoglavlju 3.6.1).

3.3 Reševanje tridiagonalnega sistema

Poiskali bomo rešitev sistema linearnih enačb $T\mathbf{x} = \mathbf{b}$, kjer je matrika sistema T tridiagonalna. Sistem lahko rešimo z Gaussovo eliminacijo in obratnim vstavljanjem [1]. Ker je v tridiagonali matriki bistveno manj elementov, se število potrebnih operacij tako za Gaussovo eliminacijo kot za obratno vstavljanje bistveno zmanjša. Dodatno predpostavimo, da je matrika T takšna, da med eliminacijo ni treba delati delnega pivotiranja (glej poglavje 2.5 v [1]). V nasprotnem primeru se tridiagonalna oblika matrike med Gaussovo eliminacijo podre in se algoritem nekoliko zakomplicira. Časovna zahtevnost Gaussove eliminacije brez pivotiranja je za tridiagonalni sistem $T\mathbf{x} = \mathbf{b}$ linearna $\mathcal{O}(n)$ namesto kubična $\mathcal{O}(n^3)$. Za obratno vstavljanje pa se časovna zahtevnost s kvadratne $\mathcal{O}(n^2)$ zmanjša na linearino $\mathcal{O}(n)$.

Samostojno delo

Priredi splošna algoritma Gaussove eliminacije in obratnega vstavljanja, da bosta upoštevala lastnosti tridiagonalnih matrik. Napiši funkcijo \:

```
function \(T::Tridiag, b::Vector),
```

ki poišče rešitev sistema $Tx = b$ (rešitev je Program 17).

V datoteko Vaja03/test/runtests.jl dodaj test, ki na primeru preveri pravilnost funkcije \.

3.4 Slučajni sprehod

Metodo za reševanje tridiagonalnega sistema bomo uporabili na primeru **slučajnega sprehoda** v eni dimenziji. Slučajni sprehod je vrsta **stohastičnega procesa**, ki ga lahko opišemo z **Markovsko verigo** z množico stanj, ki je enaka množici celih števil \mathbb{Z} . Če se na nekem koraku slučajni sprehod nahaja v stanju n , se lahko v naslednjem koraku z verjetnostjo $p \in [0, 1]$ premakne v stanje $n - 1$ ali z verjetnostjo $q = 1 - p$ v stanje $n + 1$. Prehodni verjetnosti slučajnega sprehoda sta enaki:

$$\begin{aligned} P(X_{i+1} = n - 1 \mid X_i = n) &= p \\ P(X_{i+1} = n + 1 \mid X_i = n) &= q. \end{aligned} \tag{3.3}$$

Definicija Markovske verige

Markovska veriga je zaporedje slučajnih spremenljivk

$$X_1, X_2, X_3, \dots \tag{3.4}$$

z vrednostmi v množici stanj (\mathbb{Z} za slučajni sprehod), za katere velja Markovska lastnost:

$$P(X_{i+1} = x \mid X_1 = x_1, X_2 = x_2, \dots, X_i = x_i) = P(X_{i+1} = x \mid X_i = x_i). \tag{3.5}$$

Ta pove, da je verjetnost za prehod v naslednje stanje odvisna le od prejšnjega stanja in ne od starejše zgodovine stanj. V Markovski verigi tako zgodovina, kako je proces prišel v neko stanje, ne odloča o naslednjem stanju, odloča le stanje, v katerem se proces trenutno nahaja.

Verjetnosti $P(X_{i+1} = x \mid X_i = x_i)$ imenujemo *prehodne verjetnosti* Markovske verige. V nadaljevanju bomo privzeli, da so prehodne verjetnosti enake za vse korake k :

$$P(X_{k+1} = x \mid X_k = y) = P(X_2 = x \mid X_1 = y). \tag{3.6}$$

Simulirajmo prvih 100 korakov slučajnega sprehoda:

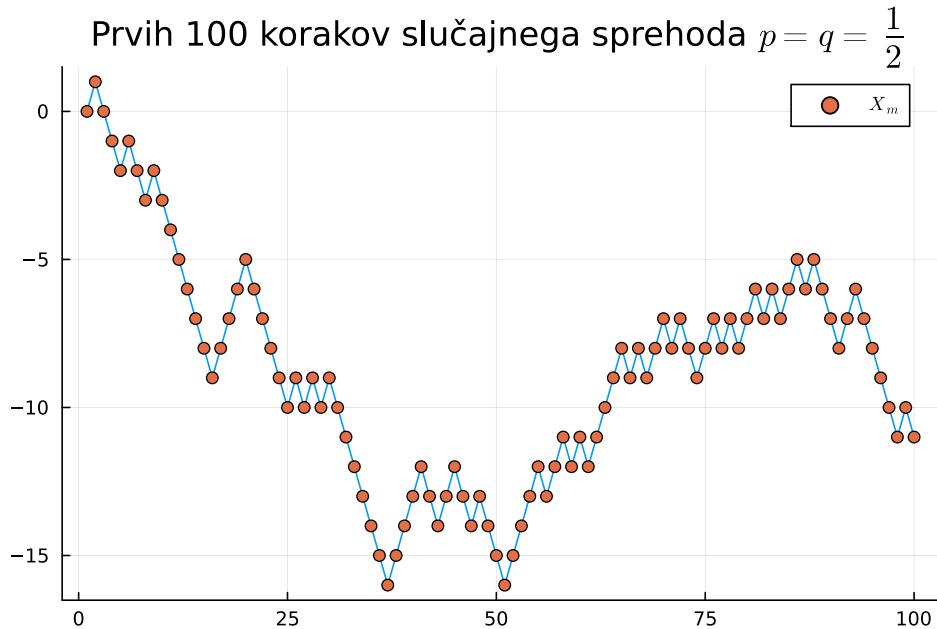
```
"""
Simuliraj `n` korakov slučajnega sprehoda s prehodnima verjetnostima `p`
in `1-p`.
"""

function sprehod(p, n)
    x = zeros(n)
    for i = 1:n-1
        x[i+1] = rand() < p ? x[i] + 1 : x[i] - 1
    end
    return x
end
```

```

using Plots
x = sprehod(0.5, 100)
plot(x, label=false)
scatter!(x, title="Prvih 100 korakov slučajnega sprehoda  $p=q=\frac{1}{2}$ ",
label="\$X_m\$")

```



Slika 6: Simulacija slučajnega sprehoda

Prehodna matrika Markovske verige

Za Markovsko verigo s končno množico stanj $\{x_1, x_2, \dots, x_n\}$, lahko prehodne verjetnosti zložimo v matriko. Brez škode lahko stanja $\{x_1, x_2, \dots, x_n\}$ nadomestimo z naravnimi števili $\{1, 2, \dots, n\}$. Matriko $\mathcal{P} = [p_{ij}]$, katere elementi so prehodne verjetnosti prehodov med stanji Markovske verige

$$p_{ij} = P(X_n = j | X_{n-1} = i), \quad (3.7)$$

imenujemo **prehodna matrika** Markovske verige. Za prehodno matriko velja, da vsi elementi ležijo na $[0, 1]$ in da je vsota elementov po vrsticah enaka 1:

$$\sum_{j=1}^n p_{ij} = 1. \quad (3.8)$$

Posledično je vektor samih enic $\mathbf{1} = [1, 1, \dots, 1]^T$ lastni vektor matrike \mathcal{P} za lastno vrednost 1:

$$\mathcal{P}\mathbf{1} = \mathbf{1}. \quad (3.9)$$

Prehodna matrika povsem opiše porazdelitev Markovske verige. Potence prehodne matrike \mathcal{P}^m na primer določajo prehodne verjetnosti po m korakih:

$$[\mathcal{P}^m]_{ij} = P(X_m = j | X_1 = i). \quad (3.10)$$

3.5 Pričakovano število korakov

Poiskati želimo pričakovano število korakov, po katerem se slučajni sprehod prvič pojavi v stanju k ali $-k$. Zato bomo privzeli, da se sprehod v stanjih $-k$ in k ustavi in se ne premakne več.

Stanje, iz katerega se veriga ne premakne več, imenujemo *absorbirajoče stanje*. Za absorbirajoče stanje k je diagonalni element prehodne matrike enak 1, vsi ostali elementi v vrstici pa 0:

$$\begin{aligned} p_{kk} &= P(X_{i+1} = k \mid X_i = k) = 1 \\ p_{kl} &= P(X_{i+1} = l \mid X_i = k) = 0. \end{aligned} \quad (3.11)$$

Stanje, ki ni absorbirajoče, imenujemo *prehodno stanje*. Markovsko verigo, ki vsebujejo vsaj eno absorbirajoče stanje, imenujemo **absorbirajoča Markovska veriga**.

Privzamemo, da je začetno stanje enako 0. Iščemo pričakovano število korakov, po katerem se slučajni sprehod prvič pojavi v stanju k ali $-k$. Stanji k in $-k$ spremenimo v absorbirajoči stani, saj stanj, ki so več kot k oddaljena od izhodišča, ni treba upoštevati. Obravnavamo torej absorbirajočo verigo z $2k + 1$ stanj, pri kateri sta stanji $-k$ in k absorbirajoči, ostala stanja pa ne. Iščemo pričakovano število korakov, da iz začetnega stanja pridemo v eno od absorbirajočih stanj.

Za izračun iskane pričakovane vrednosti uporabimo **kanonično obliko prehodne matrike**.

Kanonična oblika prehodne matrike

Če ima Markovska veriga absorbirajoča stanja, lahko prehodno matriko zapišemo v bločni obliki:

$$\mathcal{P} = \begin{pmatrix} Q & T \\ 0 & I \end{pmatrix}, \quad (3.12)$$

kjer vrstice $[Q, T]$ ustrezano prehodnim, vrstice $[0, I]$ pa absorbirajočim stanjem. Matrika Q opiše prehodne verjetnosti za sprehod med prehodnimi stani, matrika Q^m pa prehodne verjetnosti po m korakih, če se sprehajamo le po prehodnih stanjih.

Vsoto vseh potenc matrike Q :

$$N = \sum_{m=0}^{\infty} Q^m = (I - Q)^{-1} \quad (3.13)$$

imenujemo *fundamentalna matrika* absorbirajoče Markovske verige. Element n_{ij} predstavlja pričakovano število obiskov stanja j , če začnemo v stanju i .

Pričakovano število korakov, da dosežemo absorbirajoče stanje iz začetnega stanja i , je i -ta komponenta produkta matrike N z vektorjem samih enic:

$$\mathbf{m} = N\mathbf{1} = (I - Q)^{-1}\mathbf{1}. \quad (3.14)$$

Če želimo poiskati pričakovano število korakov \mathbf{m} , moramo rešiti sistem linearnih enačb:

$$(I - Q)\mathbf{m} = \mathbf{1}. \quad (3.15)$$

Če nas zanima, kdaj bo sprehod prvič za k oddaljen od izhodišča, lahko začnemo v 0 in stani k in $-k$ proglašimo za absorbirajoča stanja. Prehodna matrika, ki jo dobimo, je tridiagonalna z 0 na diagonali. Matrika $I - Q$ je prav tako tridiagonalna z 1 na diagonali in z negativnimi verjetnostmi $-p$ na prvi poddiagonali in $-q = p - 1$ na prvi naddiagonali:

$$I - Q = \begin{pmatrix} 1 & -q & 0 & \dots & 0 \\ -p & 1 & -q & \dots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & -p & 1 & -q \\ 0 & \dots & 0 & -p & 1 \end{pmatrix}. \quad (3.16)$$

Matrika $I - Q$ je tridiagonalna. Poleg tega je po stolpcih diagonalno dominantna in v prvem in zadnjem stolpcu strogo diagonalno dominantna¹, zato lahko uporabimo Gaussovo eliminacijo brez pivotiranja. Najprej napišemo funkcijo, ki zgradi matriko $I - Q$:

```
using Vaja03
"""
N = matrika_sprehod(k, p)

Sestavi fundamentalno matriko za slučajni sprehod, ki se konča, ko se prvič
za `k` korakov oddalji od izhodišča.
"""

matrika_sprehod(k, p) = Tridiag(-p * ones(2k - 2), ones(2k - 1), -(1 - p) *
ones(2k - 2))
```

Program 10: Funkcija, ki sestavi tridiagonalno matriko $I - Q$ za slučajni sprehod. Slučajni sprehod se konča, ko se prvič oddalji za k korakov od izhodišča.

Pričakovano število korakov izračunamo kot rešitev sistema $(I - Q)\mathbf{k} = \mathbf{1}$. Uporabimo operator \ za tridiagonalno matriko:

```
"""
Em = koraki(k, p)

Izračunaj pričakovano število korakov `Em`, ki jih potrebuje slučajni sprehod,
da doseže stanje `0` ali `2k`. Komponente vektorja `Em` vsebujejo pričakovano
število korakov, da sprehod pride v stanje `0` ali `2k`, če začne v stanju med
`1` in `2k - 1`.

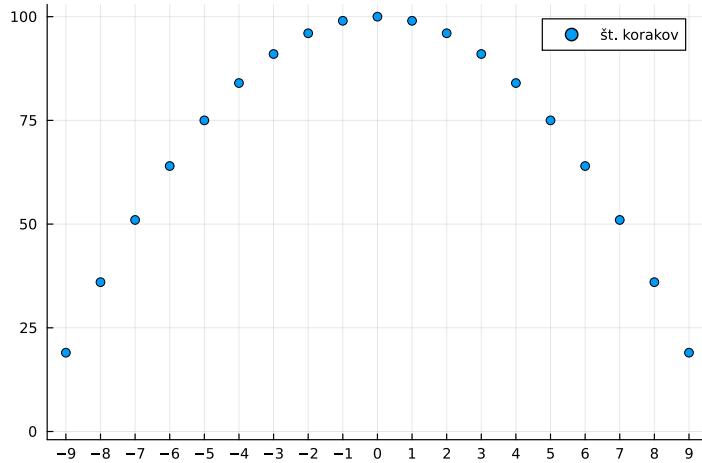
koraki(k, p) = matrika_sprehod(k, p) \ ones(2k - 1)
```

Program 11: Funkcija, ki izračuna vektor pričakovanih števil korakov, ki jih potrebuje slučajni sprehod, da se iz začetnega stanja med 1 in $2k - 1$ premakne v stanje 0 ali $2k$.

V matriki Q so stanja označena z indeksi matrike od 1 do $2k - 1$. Zato stanja premaknemo za $-k$, dobimo stanja $-k, -k + 1, \dots, 0, \dots, k$. Komponente vektorja \mathbf{k} tako predstavljajo pričakovano število korakov, ki jih slučajni sprehod potrebuje, da prvič doseže stanje $-k$ ali k , če začnemo v stanju $i \in \{-k + 1, -k + 2, \dots, 0, 1, \dots, k - 1\}$.

```
Em = koraki(10, 0.5)
scatter(-9:9, Em, label="št. korakov", xticks=-9:9)
```

¹Matrika je po stolpcih *diagonalno dominantna*, če za vsak stolpec velja, da je absolutna vrednost diagonalnega elementa večja ali enaka vsoti absolutnih vrednosti vseh ostalih elementov v stolpcu: $|a_{ii}| \geq \sum_{i \neq j} |a_{ji}|$. Če je matrika strogo diagonalno dominantna vsaj v enem stolpcu, potem je Gaussova eliminacija brez pivotiranja numerično stabilna [2].



Slika 7: Pričakovano število korakov, da slučajni sprehod prvič doseže stanji -10 ali 10 , v odvisnosti od začetnega stanja $i \in \{-9, -8, \dots, -1, 0, 1, \dots, 8, 9\}$.

Za konec se prepričajmo še s [simulacijo Monte Carlo](#), da so rešitve, ki jih dobimo kot rešitev sistema, res prave. Slučajni sprehod simuliramo z generatorjem naključnih števil in izračunamo [vzorčno povprečje](#) za število korakov m .

```

using Random

"""
x1 = naslednje_stanje(p, x0)

Simuliraj naslednje stanje slučajnega sprehoda z naključnim generatorjem števil.

naslednje_stanje(p, x0) = x0 + (rand() < p ? -1 : 1)

"""

st_korakov = simuliraj_sprehod(k, p)

Simuliraj slučajni sprehod s prehodnima verjetnostima `p` in `1-p`.
Vrni število korakov, ki jih slučajni sprehod potrebuje, da se prvič
oddalji za `k` korakov od izhodišča.

"""

function simuliraj_sprehod(k, p, x0=0)
    koraki = 0
    while (abs(x0) < k)
        x0 = naslednje_stanje(p, x0)
        koraki += 1
    end
    koraki
end

```

Program 12: Simulacija z generatorjem naključnih števil. Vzorčno povprečje da oceno za pričakovano število korakov.

Za $k = 10$ je pričakovano število korakov enako 100. Poglejmo, kako se izračunani rezultat ujema z vrednostjo, ki jo dobimo, če slučajni sprehod velikokrat ($n = 100\,000$) ponovimo in izračunamo vzorčno povprečje:

```
Random.seed!(691)
n = 100000
k = 10
p = 0.5
kp = sum([simuliraj_sprehod(k, p) for _ in 1:n]) / n
println("Vzorčno povprečje za $n slučajnih sprehodov je $kp.")
```

Vzorčno povprečje za 100000 slučajnih sprehodov je 100.09526.

Kaj smo se naučili?

- Z upoštevanjem lastnosti sistemov linearnih enačb prihranimo veliko prostora in časa.
- Sistemi linearnih enačb se pogosto pojavijo, ko rešujemo matematične probleme, čeprav na prvi pogled nimajo nobene zveze z linearno algebro.

3.6 Rešitve

```
# Vgrajene funkcije moramo naložiti, če jim želimo dodati nove metode.
import Base: size, getindex, setindex!, *, \
"""
size(T::Tridiag)

Vrni dimenzije tridiagonalne matrike `T`.
"""
size(T::Tridiag) = (length(T.d), length(T.d))
```

Program 13: Metoda `size` za tridiagonalno matriko, ki vrne dimenzije matrike.

```

"""
    elt = getindex(T, i, j)

Vrni element v `i`-ti vrstici in `j`-tem stolpcu tridiagonalne matrike `T`.
Ta funkcija se pokliče, ko dostopamo do elementov matrike z izrazom `T[i, j]`.

"""
function getindex(T::Tridiag, i, j)
    n, _m = size(T)
    if (i < 1) || (i > n) || (j < 1) || (j > n)
        throw(BoundsError(T, (i, j)))
    end
    if i == j - 1
        return T.zd[i]
    elseif i == j
        return T.d[i]
    elseif i == j + 1
        return T.sd[j]
    else
        return zero(T.d[1])
    end
end

```

Program 14: Metoda `getindex` za tridiagonalno matriko, ki se pokliče, ko uporabimo izraz `T[i, j]`.

```

"""
    setindex!(T, x, i, j)

Nastavi element `T[i, j]` na vrednost `x`. Ta funkcija se pokliče, ko uporabimo
zapis `T[i, j] = x`.

"""
function setindex!(T::Tridiag, x, i, j)
    n, _m = size(T)
    if (i < 1) || (i > n) || (j < 1) || (j > n)
        throw(BoundsError(T, (i, j)))
    end
    if i == j - 1
        T.zd[i] = x
    elseif i == j
        T.d[i] = x
    elseif i == j + 1
        T.sd[j] = x
    else
        error("Elementa [$i, $j] ni mogoče spremeniti.")
    end
end

```

Program 15: Metoda `setindex!` se pokliče, ko uporabimo izraz `T[i, j]=x`.

```

"""
y = T*x

Izračunaj produkt tridiagonalne matrike `T` z vektorjem `x`.

function *(T::Tridiag, x::Vector)
    n = length(T.d)
    if (n != length(x))
        error("Dimenzijs se ne ujemajo!")
    end
    y = zero(x)
    y[1] = T[1, 1] * x[1] + T[1, 2] * x[2]
    for i = 2:n-1
        y[i] = T[i, i-1] * x[i-1] + T[i, i] * x[i] + T[i, i+1] * x[i+1]
    end
    y[n] = T[n, n-1] * x[n-1] + T[n, n] * x[n]
    return y
end

```

Program 16: Metoda `*` za množenje tridiagonalne matrike z vektorjem

```

"""
x = T\b

Izračunaj rešitev sistema `Tx = b`, kjer je `T` tridiagonalna matrika in `b` vektor desnih strani.

function \(T::Tridiag, b::Vector)
    n, _ = size(T)
    # ob eliminaciji se spremeni le glavna diagonala
    T = Tridiag(T.sd, copy(T.d), T.zd)
    b = copy(b)
    # eliminacija
    for i = 2:n
        l = T[i, i-1] / T[i-1, i-1]
        T[i, i] = T[i, i] - l * T[i-1, i]
        b[i] = b[i] - l * b[i-1]
    end
    # obratno vstavljanje
    b[n] = b[n] / T[n, n]
    for i = (n-1):-1:1
        b[i] = (b[i] - T[i, i+1] * b[i+1]) / T[i, i]
    end
    return b
end

```

Program 17: Metoda `\` za reševanje tridiagonarnega sistema linearnih enačb

3.6.1 Testi

```
@testset "Dostop do elementov" begin
    T = Tridiag([1, 2], [3, 4, 5], [6, 7])
    # diagonalna
    @test T[1, 1] == 3
    @test T[2, 2] == 4
    @test T[3, 3] == 5
    # spodaj
    @test T[2, 1] == 1
    @test T[3, 2] == 2
    @test T[3, 1] == 0
    # zgoraj
    @test T[1, 2] == 6
    @test T[2, 3] == 7
    @test T[1, 3] == 0
    # izven obsega
    @test_throws BoundsError T[1, 4]
end
```

Program 18: Testi za funkcijo getindex

```
@testset "Nastavljanje elementov" begin
    T = Tridiag([1, 1], [1, 1, 1], [1, 1])
    T[2, 2] = 2
    T[2, 3] = 3
    T[2, 1] = 4
    @test T[1, 1] == 1
    @test T[2, 2] == 2
    @test T[2, 3] == 3
    @test T[2, 1] == 4
    # izven obsega
    @test_throws ErrorException T[1, 3] = 2
end
```

Program 19: Testi za funkcijo setindex!

```
@testset "Množenje z vektorjem" begin
    T = Tridiag([1, 2], [3, 4, 5], [6, 7])
    A =
        3 6 0;
        1 4 7;
        0 2 5
    ]
    x = [1, 2, 3]
    @test T * x == A * x
end
```

Program 20: Testi za množenje

```

@testset "Reševanje sistema" begin
    # Deljenje ima težave z vnosim tipa `Integer`, zato dodamo decimalne pike,
    # da vrednosti uporabi kot `Float64`.
    T = Tridiag([1.0, 1], [2.0, 2, 2], [1.0, 1])
    x = [1.0, 2, 3]
    b = T * x
    @test T \ b ≈ x

    T = Tridiag(-0.5 * ones(4), ones(5), -0.5 * ones(4))
    x = T \ ones(5)
    @test x ≈ [5, 8, 9, 8, 5]
end

```

Program 21: Testi za operator \ (reševanje tridiagonalnega sistema)

4 Minimalne ploskve

Žično zanko s pravokotnim tlorisom potopimo v milnico, tako da se nanjo napne milna opna. Naša naloga bo poiskati obliko milne opne. Malo brskanja po fizikalnih knjigah ali internetu hitro razkrije, da ploskve, ki tako nastanejo, sodijo med **minimalne ploskve**, ki so burile domisljijo mnogih matematikov in nematematikov. Minimalne ploskve so navdihovale tudi umetnike in arhitekte. Eden najbolj znanih primerov uporabe minimalnih ploskev v arhitekturi je streha münchenskega olimpijskega stadiona, ki jo je zasnoval **Frei Otto** s sodelavci. Frei Otto je eksperimentiral z milnimi mehurčki in elastičnimi tkaninami, s katerimi je ustvarjal nove oblike.



Slika 8: Streha olimpijskega stadiona v Münchenu (vir: [Wikipedia](#))

4.1 Naloga

Namen te vaje je primerjava eksplisitnih in iterativnih metod za reševanje sistemov linearnih enačb. Prav tako se bomo naučili, kako zgradimo matriko sistema in desne strani enačb za spremenljivke, ki niso podane z vektorjem, temveč kot elementi matrike. V okviru te vaje zato opravi naslednje naloge:

- Izpelji matematični model za minimalne ploskve s pravokotnim tlorisom.
- Zapiši problem iskanja minimalne ploskve kot **robni problem** za **Laplaceovo enačbo** na pravokotniku.
- Robni problem diskretiziraj in zapiši v obliki sistema linearnih enačb.
- Reši sistem linearnih enačb z LU razcepom. Uporabi knjižnico **SparseArrays** za varčno hranjenje matrike sistema.
- Preveri, kako se število neničelnih elementov poveča pri LU razcepu razpršene matrike.
- Uporabi iterativne metode (Jacobijeva, Gauss-Seidlova in SOR iteracija) na elementih matrike višinskih vrednosti ploskve in reši sistem enačb brez eksplisitne uporabe matrike sistema.
- Nariši primer minimalne ploskve.
- Animiraj konvergenco iterativnih metod.

4.2 Matematično ozadje

Ploskev v trirazsežnem prostoru lahko predstavimo eksplisitno s funkcijo dveh spremenljivk $z = u(x, y)$, ki predstavlja višino ploskve nad točko (x, y) . Naša naloga je poiskati približek za funkcijo u na danem pravokotnem območju, ki opisuje obliko milne opne, napete na žični zanki s pravokotnim tlorisom.

Funkcija u , ki opisuje milno opno, zadošča [Young-Laplaceovi enačbi](#):

$$(1 + u_x^2)u_{yy} - u_x u_y u_{xy} + (1 + u_y^2)u_{xx} = \rho(x, y), \quad (4.1)$$

kjer so $u_x = \frac{\partial u}{\partial x}$, $u_y = \frac{\partial u}{\partial y}$, $u_{xx} = \frac{\partial^2 u}{\partial x^2}$, $u_{xy} = \frac{\partial^2 u}{\partial x \partial y}$ in $u_{yy} = \frac{\partial^2 u}{\partial y^2}$ parcialni odvodi funkcije u . Funkcija ρ je sorazmerna tlačni razlike med zgornjo in spodnjo površino milne opne in je posledica teže milnice. Enačba (4.1) vsebuje parcialne odvode in jo zato uvrščamo med [parcialne diferencialne enačbe](#) ali s kratico PDE. Parcialni odvodi nastopajo nelinearno, zato enačbo (4.1) uvrščamo med nelinearne PDE.

Če zanemarimo tlačno razliko ρ in višje potence odvodov u_x^2 , u_y^2 in $u_x u_y$, dobimo [Laplaceovo enačbo](#):

$$\Delta u(x, y) = u_{xx}(x, y) + u_{yy}(x, y) = 0. \quad (4.2)$$

Diferencialni operator

$$\Delta u = u_{xx} + u_{yy} \quad (4.3)$$

imenujemo [Laplaceov operator](#).

Vrednosti u na robu območja so določene z obliko zanke, medtem ko za vrednosti v notranjosti velja enačba (4.2). Problem za diferencialno enačbo, pri katerem so podane vrednosti na robu, imenujemo [robni problem](#). Ker je oblika milnice določena na robu, iskanje oblike milnice prevedemo na robni problem za Laplaceovo PDE na območju, omejenem s tlorisom žične zanke.

Naj bo območje pravokotnik $[a, b] \times [c, d]$. Poleg Laplaceove enačbe (4.2) veljajo za vrednosti funkcije u tudi [robni pogoji](#):

$$\begin{aligned} u(x, c) &= f_s(x), \\ u(x, d) &= f_z(x), \\ u(a, y) &= f_l(y), \\ u(b, y) &= f_d(y). \end{aligned} \quad (4.4)$$

Pri tem so f_s , f_z , f_l in f_d dane funkcije. Rešitev robnega problema je odvisna od izbire območja in robnih pogojev.

4.3 Diskretizacija in sistem linearnih enačb

Problema se lotimo numerično, zato vrednosti $u(x, y)$ poiščemo le v končno mnogo točkah: problem [diskretiziramo](#). Za diskretizacijo je najpreprosteje uporabiti enakomerno razporejeno pravokotno mrežo točk na pravokotniku. Točke na mreži imenujemo [vozlišča](#). Zaradi enostavnosti se omejimo na mreže z enakim razmikom v obeh koordinatnih smereh. Interval $[a, b]$ razdelimo na $n + 1$ delov, interval $[c, d]$ pa na $m + 1$ delov tako, da sta razmika v obeh smereh približno enaka. Dobimo zaporedje koordinat, ki definirajo pravokotno mrežo točk (x_j, y_i) :

$$\begin{aligned} a = x_0, x_1, \dots, x_{n+1} &= b \quad \text{in} \\ c = y_0, y_1, \dots, y_{m+1} &= d. \end{aligned} \quad (4.5)$$

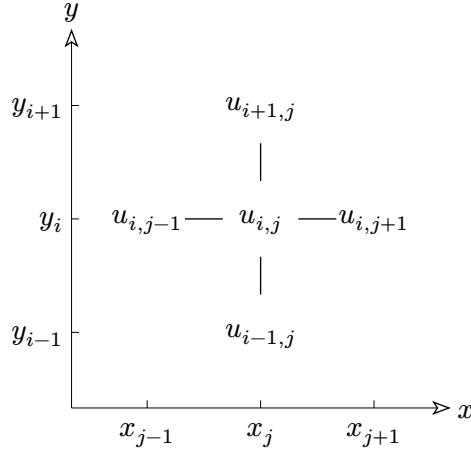
Namesto funkcije $u : [a, b] \times [c, d] \rightarrow \mathbb{R}$ tako iščemo le vrednosti

$$u_{ij} = u(x_j, y_i), \quad i = 1, \dots, m, \quad j = 1, \dots, n. \quad (4.6)$$

Vrstni red indeksov j in i smo v matriki zamenjali, saj prvi indeks določa položaj elementa matrike v navpični smeri, drugi indeks pa položaj v vodoravni smeri.

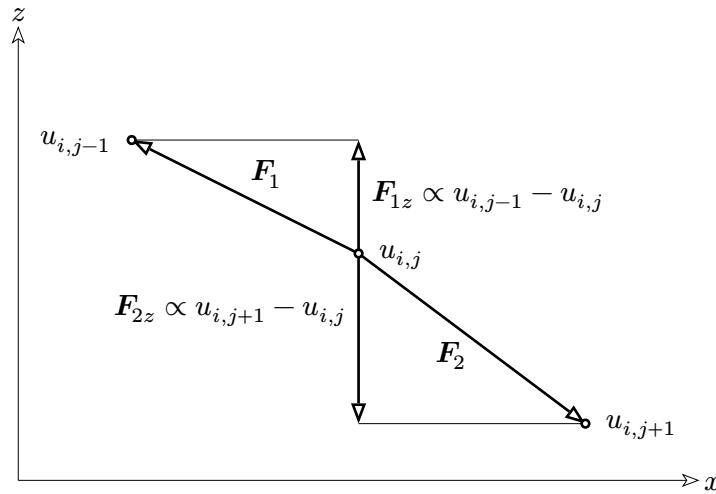
Elemente matrike u_{ij} določimo tako, da je v limiti, ko gre razmik med vozlišči proti 0, izpolnjena Laplaceova enačba (4.2).

Laplaceovo enačbo lahko diskretiziramo s **končnimi diferencami**. Lahko pa dobimo navdih pri arhitektu Ottu, ki je minimalne ploskve raziskoval z elastičnimi tkaninami. Ploskev si predstavljamo kot elastično tkanino, ki je fina kvadratna mreža iz elastičnih nitk. Vsako vozlišče v mreži je povezano s štirimi sosednjimi vozlišči.



Slika 9: Vrednosti v sosednjih vozliščih

Vozlišče je v ravnovesju, ko je vsota vseh sil nanj enaka 0.



Slika 10: Vektorske komponente sil, ki delujejo na vozlišče (x_j, y_i) iz sosednjih vozlišč (x_{j-1}, y_i) in (x_{j+1}, y_i) .

Predpostavimo, da so vozlišča povezana z idealnimi vzmetmi in je sila sorazmerna z vektorjem med položaji vozlišč. Če zapišemo enačbo za komponente sile v smeri z , dobimo za točko (x_j, y_i, u_{ij}) enačbo:

$$(u_{i-1j} - u_{ij}) + (u_{ij-1} - u_{ij}) + (u_{i+1j} - u_{ij}) + (u_{ij+1} - u_{ij}) = 0 \Rightarrow \\ \Rightarrow u_{i-1j} + u_{ij-1} - 4u_{ij} + u_{i+1j} + u_{ij+1} = 0. \quad (4.7)$$

Za vsako vrednost u_{ij} dobimo eno enačbo. Tako dobimo sistem $n \cdot m$ linearnih enačb za $n \cdot m$ neznank. Ker so vrednosti na robu določene z robnimi pogoji, moramo elemente u_{0j}, u_{m+1j}, u_{i0} in u_{in+1} prestaviti na desno stran in jih upoštevati kot konstante.

4.4 Matrika sistema linearnih enačb

Sisteme linearnih enačb navadno zapišemo v matrični obliki:

$$Ax = b, \quad (4.8)$$

kjer je A kvadratna matrika, x in b pa vektorja. V našem primeru je to nekoliko bolj zapleteno, saj so spremenljivke u_{ij} elementi matrike. Zato jih moramo najprej razvrstiti v vektor $\mathbf{x} = [x_1, x_2, \dots, x_N]^T$, kjer je $N = m \cdot n$ število vseh elementov matrike. Najpogosteje elemente u_{ij} razvrstimo v vektor \mathbf{x} po stolpcih, tako da je:

$$\mathbf{x} = [u_{11}, u_{21}, \dots, u_{m1}, u_{12}, u_{22}, \dots, u_{1n}, \dots, u_{m-1n}, u_{mn}]^T. \quad (4.9)$$

Iz enačb (4.7) lahko potem razberemo matriko A . Za $n = m = 3$ dobimo matriko velikosti 9×9 :

$$A^{9,9} = \begin{pmatrix} -4 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & -4 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & -4 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & -4 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & -4 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & -4 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & -4 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & -4 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & -4 \end{pmatrix}, \quad (4.10)$$

ki je sestavljena iz 3×3 blokov

$$L^{3,3} = \begin{pmatrix} -4 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & -4 \end{pmatrix}, \quad I^{3,3} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}. \quad (4.11)$$

Vektor desnih strani prav tako razberemo iz enačbe (4.7). Za $n = m = 3$ dobimo vektor:

$$\mathbf{b} = -[u_{01} + u_{10}, u_{20}, u_{30} + u_{41}, u_{02}, 0, u_{42}, u_{03} + u_{14}, u_{24}, u_{34} + u_{43}]^T. \quad (4.12)$$

V splošnem je formulo za vektor desnih strani lažje sprogramirati, zato bomo zapis izpustili.

Razvrstitev po stolpcih in operator vec

Elemente matrike razvrstimo v vektor tako, da stolpce matrike enega za drugim postavimo v vektor. Indeks v vektorju k izrazimo z indeksi v matriki i, j s formulo

$$k = i + (j - 1)m. \quad (4.13)$$

Ta način preoblikovanja matrike v vektor označimo s posebnim operatorjem vec:

$$\begin{aligned} \text{vec} : \mathbb{R}^{m \times n} &\rightarrow \mathbb{R}^{m \cdot n} \\ \text{vec}(A)_{i+(j-1)m} &= a_{ij}. \end{aligned} \quad (4.14)$$

4.5 Izpeljava sistema s Kroneckerjevim produktom

Množenje matrike A z vektorjem $\mathbf{x} = \text{vec}(U)$ lahko zapišemo kot:

$$A \text{vec}(U) = \text{vec}(LU + UL), \quad (4.15)$$

kjer je L matrika Laplaceovega operatorja v eni dimenziji, ki ima -2 na diagonali in 1 na spodnji in zgornji obdiagonali:

$$L = \begin{pmatrix} -2 & 1 & 0 & \dots & 0 \\ 1 & -2 & 1 & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & 1 & -2 & 1 \\ 0 & \dots & 0 & 1 & -2 \end{pmatrix}. \quad (4.16)$$

Res! Moženje matrike U z matriko L z leve je ekvivalentno množenju stolpcev matrike U z matriko L , medtem ko je množenje z matriko L z desne ekvivalentno množenju vrstic matrike U z matriko L . Prispevek množenja z leve predstavlja vsoto sil sosednjih vozlišč v smeri y , medtem ko množenje z desne predstavlja vsoto sil sosednjih vozlišč v smeri x . Element produkta $LU + UL$ na mestu (i, j) je enak:

$$\begin{aligned} (LU + UL)_{ij} &= \sum_{k=1}^m l_{ik} u_{kj} + \sum_{k=1}^n u_{ik} l_{kj} = \\ &= u_{i-1j} - 2u_{ij} + u_{i+1j} + u_{ij-1} - 2u_{ij} + u_{ij+1}, \end{aligned} \quad (4.17)$$

kar je enako levi strani enačbe (4.7).

Operacijo množenja matrike $U : U \mapsto LU + UL$ lahko predstavimo s Kroneckerjevim produktom \otimes , saj velja $\text{vec}(AXB) = A \otimes B \cdot \text{vec}(X)$. Tako velja:

$$\begin{aligned} A \text{vec}(U) &= \text{vec}(LU + UL) = \text{vec}(LUI + IUL) = \\ &= \text{vec}(LUI) + \text{vec}(IUL) = (L \otimes I) \text{vec}(U) + (I \otimes L) \text{vec}(U) \end{aligned} \quad (4.18)$$

in

$$A^{N,N} = L^{m,m} \otimes I^{n,n} + I^{m,m} \otimes L^{n,n}. \quad (4.19)$$

Kroneckerjev produkt in operator vec v Juliiji

Programski jezik Julia ima vgrajene funkcije `vec` in `kron` za preoblikovanje matrik v vektorje in računanje Kroneckerjevega produkta. Z ukazom `reshape` iz vektorja znova zgradimo matriko.

4.6 Numerična rešitev z LU razcepom

Preden se lotimo programiranja, ustvarimo nov paket za to vajo:

```
(num_mat) pkg> generate Vaja04  
(num_mat) pkg> develop Vaja04/
```

Nato dodamo pakete, ki jih bomo potrebovali:

```
(num_mat) pkg> activate Vaja04  
(Vaja04) pkg> add SparseArrays
```

Kodo bomo organizirali tako, da bomo najprej ustvarili podatkovni tip, ki opiše robni problem za PDE na pravokotniku:

```
"""  
rp = RobniProblemPravokotnik(op, ((a, b), (c, d)), [fs, fz, fl, fd])  
  
Ustvari objekt tipa `RobniProblemPravokotnik`, ki hrani podatke za robni problem  
za diferencialni operator `op` na pravokotniku `[a, b] x [c, d]` z robnimi  
pogoji, podanimi s funkcijami `fs`, `fz`, `fl`, `fd`. Funkcija `fs` določa robni  
pogoj na spodnjem robu `y = c`, funkcija `fz` robni pogoj na zgornjem robu `y =  
d`,  
funkcija `fl` na levem robu `x = a` in funkcija `fd` robni pogoj na desnem robu  
`x = b`.  
"""  
struct RobniProblemPravokotnik  
    op # diferencialni operator  
    meje # meje pravokotnika [a, b] x [c, d] v obliku [(a, b), (c, d)]  
    rp # funkcije na robu [fs, fz, fl, fd], f(a, y) = fl(y), f(x, c) = fs(x) ...  
end
```

Definiramo še tip brez polj, ki predstavlja Laplaceov diferencialni operator (4.3) in ga bomo lahko dodali v polje za operator v `RobniProblemPravokotnik`:

```
"""  
L = Laplace()  
  
Ustvari objekt tipa `Laplace`, ki predstavlja Laplaceov diferencialni  
operator.  
"""  
struct Laplace end
```

Podatkovni tipi brez polj

Programski jezik Julia ne pozna razredov. **Podatkovni tipi brez polj**, kot je `Laplace`, nadomestijo razrede brez stanja in omogočajo podobno obliko **polimorfizma**.

Robni problem za Laplaceovo enačbo na pravokotniku $[0, 1.5\pi] \times [0, \pi]$ z robnimi pogoji:

$$\begin{aligned} u(x, 0) &= u(x, \pi) = \sin(x) \quad \text{in} \\ u(0, y) &= u(1.5\pi, y) = \sin(y) \end{aligned} \tag{4.20}$$

predstavimo z objektom:

```
rp = RobniProblemPravokotnik(  
    Laplace(),           # operator  
    ((0, 1.5*pi), (0, pi)), # pravokotnik  
    (x -> 0, x -> 0, sin, sin) # funkcije na robu  
)
```

Zaenkrat si s tem objektom še ne moremo nič pomagati. Zato napišemo funkcije, ki bodo poiskale rešitev za dani robni problem. Kot smo videli v poglavju 4.3, lahko približek za rešitev robnega problema poiščemo kot rešitev sistema linearnih enačb (4.7).

Samostojno delo

Najprej napiši funkcijo, ki vrne matriko sistema:

```
function matrika(_::Laplace, n, m)
```

za dane dimenzijsje notranje mreže $n \times m$ (za rešitev glej Program 23).

Nato na robu mreže izračunamo robne pogoje in sestavimo vektor desnih strani sistema (4.7). Ker je preslikovanje dvojnega indeksa v enojnega in nazaj precej sitno, bomo večino operacij naredili na matriki $U = [u_{ij}]$ dimenzij $(m+2) \times (n+2)$, ki vsebuje tudi vrednosti na robu.

Samostojno delo

Napiši funkcijo:

```
U0, x, y = diskretiziraj(rp::RobniProblemPravokotnik, h),
```

ki poišče pravokotno mrežo z razmikom med vozlišči približno enakim h in izračuna vrednosti na robu. Rezultati funkcije `diskretiziraj` so matrika $U0$, vektor x in vektor y . Matrika $U0$ ima notranje elemente enake 0, robni elementi pa so določeni z robnimi pogoji. Vektorja x in y vsebujeta delilne točke na intervalih $[a, b]$ in $[c, d]$ (rešitev je Program 24).

Iz matrike $U0$ preprosto sestavimo desne strani enačb. Notranje indekse zaporedoma zamaknemo v levo, desno, gor in dol ter seštejemo ustrezne podmatrike. Rezultat nato spremenimo v vektor s funkcijo `vec`.

Samostojno delo

Napiši funkcijo `desne_strani(U0)`, ki iz rezultata funkcije `diskretiziraj` sestavi vektor desnih strani sistema (rešitev je Program 25).

Ko imas pripravljeno matriko in desne strani, vse skupaj zloži v funkcijo:

```
U, x, y = resi(rp::RobniProblemPravokotnik, h),
```

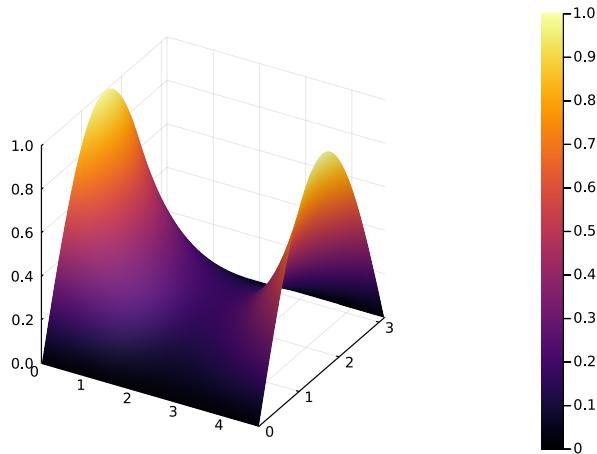
ki za dani robni problem `rp` in razmik med vozlišči h sestavi matriko sistema, izračuna desne strani na podlagi robnih pogojev in reši sistem. Funkcija vrne matriko vrednosti U in vektorja delilnih točk x in y (rešitev je Program 26).

Napisane programe uporabimo za rešitev robnega problema na pravokotniku $[0, 1.5\pi] \times [0, \pi]$ z robnimi pogoji:

$$\begin{aligned} u(0, y) &= 0, \\ u(1.5\pi, y) &= 0, \\ u(x, 0) &= \sin(x), \\ u(x, \pi) &= \sin(x). \end{aligned} \tag{4.21}$$

Definiramo robni problem in uporabimo funkcijo `resi`. Ploskev narišemo s funkcijo `surface`.

```
U, x, y = resi(rp, 0.1)
using Plots
surface(x, y, U)
```

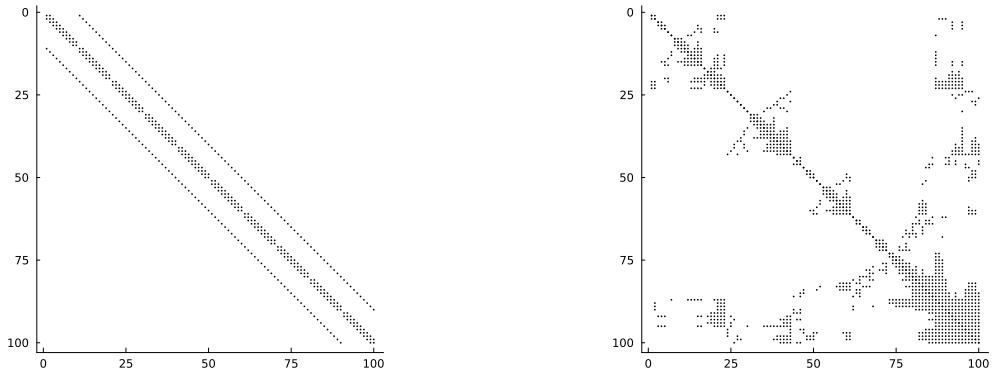


Slika 11: Rešitev robnega problema za Laplaceovo enačbo z robnimi pogoji (4.20)

4.7 Napolnitev matrike ob eliminaciji

Matrika Laplaceovega operatorja ima veliko ničelnih elementov. Takim matrikam pravimo **razpršene ali redke matrike**. Razpršenost matrike izkoristimo za prihranek prostora in časa, kot smo že videli pri tridiagonalnih matrikah (Poglavlje 3). Vendar se pri LU razcepnu, ki ga uporablja operator `\` za rešitev sistema, delež neničelnih elementov matrike pogosto poveča. Poglejmo, kako se odreže matrika za Laplaceov operator.

```
using LinearAlgebra
A = Vaja04.matrika(Laplace(), 10, 10)
p1 = spy(A .!= 0, legend=false) # na grafu prikažemo neničelne elemente
F = lu(A)
p2 = spy(F.L .!= 0, legend=false) # neničelni elementi za faktor L
spy!(p2, F.U .!= 0, legend=false) # in za faktor U
```



Slika 12: Neničelni elementi matrike za Laplaceov operator (levo) in njenega LU razcepa (desno). Število ničelnih elementov se pri LU razcepnu poveča. Kljub temu sta L in U v razcepu še vedno precej redki matriki.

Matrični razcepi v Julii

V knjižnici [LinearAlgebra](#) najdemo implementacije standardnih matričnih razcepov, kot so LU razcep, razcep Choleskega, QR razcep in drugi. Rezultat matričnega razcepa v Julii je poseben podatkovni tip, ki učinkovito hrani rezultate razcepa. Poleg tega so za različne razcepe definirane specializirane metode splošnih funkcij. Posebej uporabna je funkcija `\`, ki z izbranim matričnim razcepom učinkovito reši sistem linearnih enačb.

Poglejmo si, kako z LU razcepom rešimo sistem linearnih enačb. Uporabimo funkcijo `lu`, ki vrne rezultat tipa LU, s katerim nadomestimo matriko A v izrazu $A \backslash b$:

```
A = [1 2; 3 4]
b = [1, 1]
F = lu(A) # F je tipa LU,
x = F \ b # ki ga lahko uporabimo za rešitev sistema
```

Funkcija `factorize` vrne najprimernejši razcep za dano matriko. Tako za simetrično pozitivno definitno matriko vrne razcep Choleskega.

4.8 Iteracijske metode

V prejšnjih podpoglavljih smo poiskali približno obliko minimalne ploskve, tako da smo sistem linearnih enačb (4.7) rešili z LU razcepom. Največ težav smo imeli z zapisom matrike sistema in desnih strani. Poleg tega je matrika sistema redka, ko izvedemo LU razcep pa se matrika deloma napolni. Pri razpršenih matrikah tako pogosto uporabimo [iterativne metode](#) za reševanje sistemov enačb, pri katerih se matrika ne spreminja in zato prihranimo veliko na prostorski in časovni zahtevnosti.

Ideja iteracijskih metod je preprosta. Enačbe preuredimo tako, da ostane na eni strani le en element s koeficientom 1. Tako dobimo iteracijsko formulo za zaporedje približkov $u_{ij}^{(k)}$. Če zaporedje konvergira, je limita ena od rešitev rekurzivne enačbe. V primeru sistemov linearnih enačb je rešitev enolična.

V našem primeru enačb za minimalne ploskve (4.7) izpostavimo element u_{ij} in dobimo rekurzivne enačbe:

$$u_{ij}^{(k+1)} = \frac{1}{4} \left(u_{ij-1}^{(k)} + u_{i-1j}^{(k)} + u_{i+1j}^{(k)} + u_{ij+1}^{(k)} \right), \quad (4.22)$$

ki ustreza Jacobijevi iteraciji. Približek za rešitev dobimo tako, da zaporedoma uporabimo rekurzivno formulo (4.22).

Pogoji konvergencije

Rekli boste, da je preveč enostavno enačbe le preurediti in se potem rešitev kar sama pojavi, če le dovolj dolgo računamo. Gotovo se nekje skriva kak „hakelc“. Res je! Težave se pojavijo, če zaporedje približkov **ne konvergira dovolj hitro** ali pa sploh ne. Jacobijeva, Gauss-Seidlova in SOR iteracija **ne konvergirajo vedno**, zagotovo pa konvergirajo, če je matrika **diagonalno dominantna po vrsticah**.

Konvergenco Jacobijeve iteracije lahko izboljšamo, če namesto vrednosti $u_{i-1,j}^{(k)}$ in $u_{ij-1}^{(k)}$ uporabimo nove vrednosti $u_{i-1,j}^{(k+1)}$ in $u_{ij-1}^{(k+1)}$, ki so bile že izračunane (elemente $u_{ij}^{(k+1)}$ računamo po leksikografskem vrstnem redu). Če nove vrednosti uporabimo v iteracijski formuli, dobimo **Gauss-Seidlovo iteracijo**:

$$u_{i,j}^{(k+1)}\{GS\} = \frac{1}{4}\left(u_{i,j-1}^{(k+1)} + u_{i-1,j}^{(k+1)} + u_{i+1,j}^{(k)} + u_{i,j+1}^{(k)}\right). \quad (4.23)$$

Konvergenco še izboljšamo, če približek $u_{ij}^{(k)}$, ki ga dobimo z Gauss-Seidlovo metodo, malce „pokvarimo“ s približkom na prejšnjem koraku $u_{ij}^{(k)}$. Tako dobimo **metodo SOR**:

$$\begin{aligned} u_{i,j}^{(k+1)}\{GS\} &= \frac{1}{4}\left(u_{i,j-1}^{(k+1)} + u_{i-1,j}^{(k+1)} + u_{i+1,j}^{(k)} + u_{i,j+1}^{(k)}\right), \\ u_{i,j}^{(k+1)}\{SOR\} &= \omega u_{i,j}^{(k+1)}\{GS\} + (1 - \omega)u_{i,j}^{(k)}. \end{aligned} \quad (4.24)$$

Parameter ω je lahko poljubno število na intervalu $(0, 2)$. Pri $\omega = 1$ dobimo Gauss-Seidlovo iteracijo.

Prednost iteracijskih metod je, da jih je zelo enostavno implementirati. Za Laplaceovo enačbo je en korak Gauss-Seidlove iteracije podan s preprosto zanko.

```
"""
U = korak_gs(U0)

Izvedi en korak Gauss-Seidlove iteracije za Laplaceovo enačbo. Matrika `U0`
vsebuje približke za vrednosti funkcije na mreži.

"""
function korak_gs(U0)
    U = copy(U0)
    m, n = size(U)
    # spremenimo le notranje vrednosti
    for i = 2:m-1
        for j = 2:n-1
            # Gauss Seidel
            U[i, j] = (U[i+1, j] + U[i, j+1] + U[i-1, j] + U[i, j-1]) / 4
        end
    end
    return U
end
```

Program 22: Funkcija, ki poišče naslednji približek Gauss-Seidlove iteracije za diskretizacijo Laplaceove enačbe.

Samostojno delo

Napišite še funkciji `korak_jacobi(U0)` in `korak_sor(U0, omega)`, ki izračunata naslednji približek za Jacobijevo in SOR iteracijo za sistem za Laplaceovo enačbo. Nato napišite še funkcijo

```
x, k = iteracija(korak, x0),
```

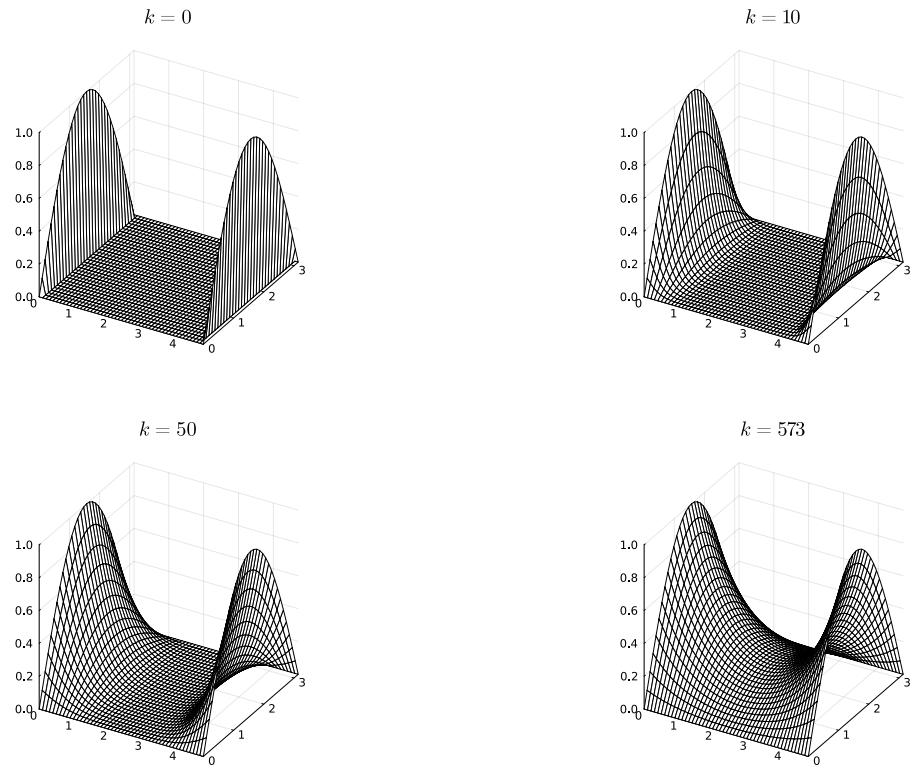
ki izračuna zaporedje približkov za poljubno iteracijsko metodo, dokler se rezultat ne spreminja več znotraj določene tolerance. Argument `korak` je funkcija, ki iz danega približka izračuna naslednjega, argument `x0` pa začetni približek iteracije.

Rešitve so na koncu poglavja v programih Program 27, Program 28 in Program 29.

4.8.1 Konvergenca

Poglejmo si, kako zaporedje približkov Gauss-Seidlove iteracije konvergira k rešitvi.

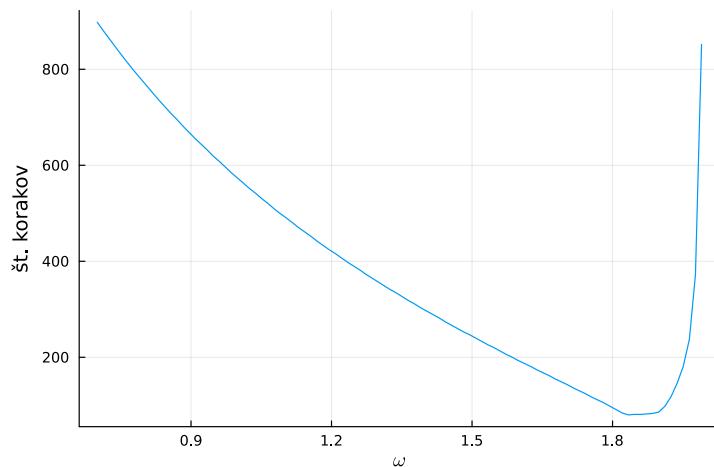
```
U0, x, y = Vaja04.diskretiziraj(rp, 0.1)
wireframe(x, y, U0, legend=false, title="\$ k=0 \$")
U = U0
for i = 1:10
    U = Vaja04.korak_gs(U)
end
wireframe(x, y, U, legend=false, title="\$k=10 \$")
U, it = Vaja04.iteracija(Vaja04.korak_gs, U0; atol=1e-3)
wireframe(x, y, U, legend=false, title="\$k=$it \$")
```



Slika 13: Približki Gauss-Seidlove iteracije za $k = 0, 10, 50$ in končni približek

Za metodo SOR je hitrost konvergencije odvisna od izbire parametra ω . Odvisnost od parametra ω je različna za različne matrike in začetne približke. Oglejmo si odvisnost za primer sistema, ki ga dobimo z diskretizacijo Laplaceove enačbe.

```
ω = range(0.7, 1.99, 100)
koraki = Vector{Float64}()
for ω_i in ω
    _, k = Vaja04.iteracija(U -> Vaja04.korak_sor(U, ω_i), U0; atol=1e-3)
    push!(koraki, k)
end
plot(ω, koraki, label=false, ylabel="št. korakov", xlabel="\$\\omega\$")
```



Slika 14: Odvisnost potrebnega števila korakov SOR iteracije od parametra ω

Kaj smo se naučili?

- Kako rešiti sistem linearnih enačb s spremenljivkami, ki so postavljene v matriko.
- Diskretizacija diferencialnih enačb privede do sistemov linearnih enačb.
- Iterativne metode so posebej uporabne za reševanje sistemov velikih dimenzij z redkimi matrikami.

4.9 Rešitve

```
using SparseArrays

laplace(n) = spdiagm(1 => ones(n - 1), 0 => -2 * ones(n), -1 => ones(n - 1))
enota(n) = spdiagm(0 => ones(n))

"""

A = matrika(Laplace(), n, m)

Ustvari matriko za diskretizacijo Laplaceovega operatorja v dveh dimenzijah
na pravokotni mreži dimenzije `n` krat `m`. Parameter `m` je število delilnih
točk v y smeri, `n` pa v x smeri.
"""

function matrika(_::Laplace, m, n)
    return kron(laplace(n), enota(m)) + kron(enota(n), laplace(m))
end
```

Program 23: Funkcija, ki zgradi matriko za diskretizacijo Laplaceovega operatorja.

```
"""

U0, x, y = diskretiziraj(rp::RobniProblemPravokotnik, h)

Diskretiziraj robni problem na pravokotniku `rp` s korakom `h`.

"""

function diskretiziraj(rp::RobniProblemPravokotnik, h)
    (a, b), (c, d) = rp.meje
    m = Integer(floor((d - c) / h))
    n = Integer(floor((b - a) / h))
    U0 = zeros(m + 2, n + 2)
    fs, fz, fl, fd = rp.rp
    x = range(a, b, n + 2)
    y = range(c, d, m + 2)
    U0[:, 1] = fl.(y)
    U0[:, end] = fd.(y)
    U0[1, :] = fs.(x)
    U0[end, :] = fz.(x)
    return U0, x, y
end
```

Program 24: Funkcija, ki diskretizira robni problem na pravokotniku.

```

function desne_strani(U0)
    return -vec(U0[2:end-1, 1:end-2] + U0[2:end-1, 3:end] +
                U0[1:end-2, 2:end-1] + U0[3:end, 2:end-1])
end

```

Program 25: Funkcija, ki izračuna robne pogoje in desne strani sistema za diskretizacijo Laplaceove enačbe.

```

"""
U, x, y = resi(rp, h, metoda)

Poišči rešitev robnega problema na pravokotniku na pravokotni mreži z razmikom
`h` med posameznimi vozlišči v obeh dimenzijah.
"""

function resi(rp::RobniProblemPravokotnik, h)
    U, x, y = diskretiziraj(rp, h)
    n = length(x) - 2
    m = length(y) - 2
    A = matrika(rp.op, m, n)
    d = desne_strani(U)
    res = A \ d # reši sistem
    U[2:end-1, 2:end-1] = reshape(res, m, n) # preoblikuj rešitev v matriko
    return U, x, y
end

```

Program 26: Funkcija, ki poišče približno rešitev robnega problema za Laplaceovo enačbo.

```

"""
U = korak_jacobi(U0)

Izvedi en korak Jacobijeve iteracije za Laplaceovo enačbo. Matrika `U0` vsebuje
približke za vrednosti funkcije na mreži, funkcija vrne naslednji približek.
"""

function korak_jacobi(U0)
    U = copy(U0)
    m, n = size(U)
    # spremenimo le notranje vrednosti
    for i = 2:m-1
        for j = 2:n-1

```

Program 27: Funkcija, ki poišče naslednji približek Jacobijeve iteracije za diskretizacijo Laplaceove enačbe.

```

"""
U = korak_sor(U0, ω)

Izvedi en korak SOR iteracije za Laplaceovo enačbo. Matrika `U0` vsebuje
približke za vrednosti funkcije na mreži, funkcija vrne naslednji približek.

"""

function korak_sor(U0, ω)
    U = copy(U0)
    m, n = size(U)
    # spremenimo le notranje vrednosti
    for i = 2:m-1
        for j = 2:n-1
            U[i, j] = (U[i+1, j] + U[i, j+1] + U[i-1, j] + U[i, j-1]) / 4
            U[i, j] = (1 - ω) * U0[i, j] + ω * U[i, j] # SOR popravek
        end
    end
    return U
end

```

Program 28: Funkcija, ki poišče naslednji približek SOR iteracije za diskretizacijo Laplaceove enačbe.

```

"""
x, it = iteracija(korak, x0; maxit=maxit, atol=atol)

Poišči približek za limito rekurzivnega zaporedja podanega rekurzivno s
funkcijo `korak` in začetnim členom `x0`.

"""

function iteracija(korak, x0; maxit=1000, atol=1e-8)
    for i = 1:maxit
        x = korak(x0)
        if isapprox(x, x0, atol=atol)
            return x, i
        end
        x0 = x
    end
    throw("Iteracija ne konvergira po $maxit korakih!")
end

```

Program 29: Funkcija, ki poišče približek za limito rekurzivnega zaporedja.

5 Interpolacija z implicitnimi funkcijami

Krivulje v ravnini in ploskve v prostoru lahko opišemo na različne načine:

| | krivulje v \mathbb{R}^2 | ploskve v \mathbb{R}^3 |
|--------------|---------------------------|---|
| eksplicitno | $y = f(x)$ | $z = f(x, y)$ |
| parametrično | $(x, y) = (x(t), y(t))$ | $(x, y, z) = (x(u, v), y(u, v), z(u, v))$ |
| implicitno | $F(x, y) = 0$ | $F(x, y, z) = 0$ |

Tabela 1: Različni načini predstavitev krivulj v \mathbb{R}^2 in ploskev v \mathbb{R}^3

Implicitne enačbe oblike $F(x_1, x_2, \dots) = 0$ so zelo dober način za opis krivulj in ploskev. Hitri algoritmi za izračun nivojskih krivulj in ploskev kot sta **korakajoči kvadrati** in **korakajoče kocke** omogočajo učinkovito generiranje poligonske mreže za implicitno podane krivulje in ploskev. Predstavitev s **predznačeno funkcijo razdalje** je osnova za mnoge grafične programe, ki delajo s ploskvami v 3D prostoru.

V tej vaji bomo spoznali, kako poiskati implicitno krivuljo ali ploskev, ki dobro opiše dani oblak točk v ravnini ali prostoru. Funkcijo F v implicitni enačbi $F(x, y) = 0$ bomo poiskali kot linearo kombinacijo **radialnih baznih funkcij (RBF)** ([7], [8], [9]).

5.1 Naloga

- Definiraj podatkovni tip za linearo kombinacijo radialnih baznih funkcij v \mathbb{R}^d . Podatkovni tip naj vsebuje središča RBF $\mathbf{x}_i \in \mathbb{R}^d$, funkcijo oblike $\varphi : \mathbb{R} \rightarrow \mathbb{R}$ in koeficiente $w_i \in \mathbb{R}$ v linearni kombinaciji:

$$F(\mathbf{x}) = \sum_{i=1}^n w_i \varphi(\|\mathbf{x} - \mathbf{x}_i\|). \quad (5.1)$$

- Napiši sistem za koeficiente w_i v linearni kombinaciji RBF, če so podane vrednosti $f_i = F(\mathbf{x}_i) \in \mathbb{R}$ v središčih RBF. Napiši funkcijo, ki za dane vrednosti f_i , funkcijo φ in središča \mathbf{x}_i , poišče koeficiente w_1, w_2, \dots, w_n . Katero metodo za reševanja sistema lahko uporabimo?
- Napiši funkcijo **vrednost**, ki izračuna vrednost funkcije $F(\mathbf{x})$ v dani točki \mathbf{x} .
- Uporabi napisane metode in interpoliraj oblak točk v ravnini z implicitno podano krivuljo. Oblak točk ustvari na krivulji, podani v parametrični obliki:

$$\begin{aligned} x(\varphi) &= 8 \cos(\varphi) - \cos(4\varphi), \\ y(\varphi) &= 8 \sin(\varphi) - \sin(4\varphi). \end{aligned} \quad (5.2)$$

5.2 Interpolacija z radialnimi baznimi funkcijami

V ravnini² je podan oblak točk $\{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n\} \subset \mathbb{R}^2$. Iščemo krivuljo, ki dobro opiše dane točke. Če zahtevamo, da vse točke ležijo na krivulji, problemu rečemo *interpolacija*, če pa dovolimo, da je krivulja zgolj blizu danih točk in ne nujno vsebuje vseh točk, problem imenujemo *aproximacija*. Krivuljo bomo

²Postopek, ki ga bomo opisali, deluje ravno tako dobro tudi za točke v prostoru. Zavoljo enostavnosti se bomo omejili na točke v ravnini.

poiskali v implicitni obliki kot nivojsko krivuljo funkcije dveh spremenljivk. Za izbrano vrednost $c \in \mathbb{R}$ iščemo funkcijo $f(x, y)$, za katero velja:

$$f(x_i, y_i) = c \quad (5.3)$$

za vse točke $\mathbf{x}_i = (x_i, y_i)$ v danem oblaku točk. Problem bomo rešili malce bolj splošno. Denimo, da imamo za vsako dano točko v oblaku \mathbf{x}_i , podano vrednost funkcije f_i . Iščemo zvezno funkcijo $f(x, y)$, tako da so izpolnjene enačbe:

$$\begin{aligned} f(x_1, y_1) &= f_1, \\ &\vdots \\ f(x_n, y_n) &= f_n. \end{aligned} \quad (5.4)$$

Zveznih funkcij, ki zadoščajo enačbam (5.4), je neskončno. Zato se moramo omejiti na podmnožico funkcij, ki je dovolj raznolika, da je sistem rešljiv, hkrati pa dovolj majhna, da je rešitev ena sama. V tej vaji se bomo omejili na n -parametrično družino funkcij oblike:

$$F(\mathbf{x}, \mathbf{w}) = F(\mathbf{x}, w_1, w_2, \dots, w_n) = \sum_i w_i \varphi(\|\mathbf{x} - \mathbf{x}_i\|). \quad (5.5)$$

Funkcije $\varphi_k(\mathbf{x}) = \varphi(\|\mathbf{x} - \mathbf{x}_k\|)$ sestavljajo bazo za množico funkcij oblike (5.1).

Radialne bazne funkcije (RBF) so funkcije, katerih vrednosti so odvisne od razdalje do izhodiščne točke:

$$r(\mathbf{x}) = \varphi(\|\mathbf{x} - \mathbf{x}_0\|). \quad (5.6)$$

Uporabljam se za interpolacijo ali aproksimacijo podatkov s funkcijo oblike:

$$F(\mathbf{x}) = \sum_i w_i \varphi(\|\mathbf{x} - \mathbf{x}_i\|), \quad (5.7)$$

npr. za rekonstrukcijo 2D in 3D oblik v računalniški grafiki. Funkcija φ je navadno pozitivna soda funkcija zvončaste oblike in jo imenujemo *funkcija oblike*.

Problem (5.4) se prevede na iskanje vrednosti koeficientov $\mathbf{w} = [w_1, w_2, \dots, w_n]^T$, tako da je izpoljen sistem enačb:

$$\begin{aligned} F(\mathbf{x}_1, w_1, w_2, \dots, w_n) &= f_1, \\ &\vdots \\ F(\mathbf{x}_n, w_1, w_2, \dots, w_n) &= f_n. \end{aligned} \quad (5.8)$$

Enačbe (5.8) so linearne za koeficiente w_1, \dots, w_n :

$$\begin{aligned} w_1 \varphi_1(\mathbf{x}_1) + w_2 \varphi_2(\mathbf{x}_1) + \dots + w_n \varphi_n(\mathbf{x}_1) &= f_1, \\ &\vdots \\ w_1 \varphi_1(\mathbf{x}_n) + w_2 \varphi_2(\mathbf{x}_n) + \dots + w_n \varphi_n(\mathbf{x}_n) &= f_n. \end{aligned} \quad (5.9)$$

Vektor desnih strani sistema (5.9) je vektor funkcijskih vrednosti $[f_1, f_2, \dots, f_n]$, matrika sistema pa je enaka:

$$\begin{pmatrix} \varphi(\|\mathbf{x}_1 - \mathbf{x}_1\|) & \varphi(\|\mathbf{x}_1 - \mathbf{x}_2\|) & \dots & \varphi(\|\mathbf{x}_1 - \mathbf{x}_n\|) \\ \varphi(\|\mathbf{x}_2 - \mathbf{x}_1\|) & \varphi(\|\mathbf{x}_2 - \mathbf{x}_2\|) & \dots & \varphi(\|\mathbf{x}_2 - \mathbf{x}_n\|) \\ \vdots & \vdots & \ddots & \vdots \\ \varphi(\|\mathbf{x}_n - \mathbf{x}_1\|) & \varphi(\|\mathbf{x}_n - \mathbf{x}_2\|) & \dots & \varphi(\|\mathbf{x}_n - \mathbf{x}_n\|) \end{pmatrix}. \quad (5.10)$$

Ker je

$$\varphi_i(\mathbf{x}_j) = \varphi(\|\mathbf{x}_j - \mathbf{x}_i\|) = \varphi(\|\mathbf{x}_i - \mathbf{x}_j\|) = \varphi_j(\mathbf{x}_i), \quad (5.11)$$

je matrika sistema (5.10) simetrična. V literaturi [7] se pojavijo naslednje izbire za funkcijo oblike φ :

- **poliharmonični zlepek (pločevina):** $\varphi(r) = r^2 \log(r)$ za 2D in $\varphi(r) = r^3$ za 3D [8],
- Gaussova funkcija: $\varphi(r) = \exp(-r^2/\sigma^2)$,
- racionalni približek za Gaussovo funkcijo:

$$\varphi(r) = \frac{1}{1 + \left(\frac{r}{\sigma}\right)^{2p}}. \quad (5.12)$$

Če izberemo prizerno funkcijo oblike, lahko dosežemo, da je matrika sistema (5.10) pozitivno definitna. V tem primeru lahko za reševanje sistema uporabimo razcep Choleskega [1]. Za funkcijo oblike bomo izbrali Gaussovo funkcijo

$$\varphi(r) = \exp(-r^2/\sigma^2), \quad (5.13)$$

za katero je matrika sistema (5.9) pozitivno definitna, če so točke $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n$ različne [10].

5.3 Program

Najprej definiramo podatkovni tip, ki opisuje linearno kombinacijo RBF (5.1).

```
.....
RBF(točke, uteži, phi)

Podatkovni tip za linearno kombinacijo *radialnih baznih funkcij* oblike
`phi(norm(x - točke[i])^2)`.

.....
struct RBF
    točke
    uteži
    phi
end
```

Samostojno delo

Za podatkovni tip napiši funkcijo `vrednost(x, rbf::RBF)`, ki izračuna vrednost linearne kombinacije (5.1) v dani točki x (rešitev Program 30).

Za primer ustvarimo mešanico dveh Gaussovih RBF v točkah $(1, 0)$ in $(2, 1)$ in izračunamo vrednost v točki $(1.5, 1.5)$:

```

using Vaja05
""" Ustvari Gaussovo funkcijo z danim `sigma`."""
gauss(sigma) = r -> exp(-r^2 / sigma^2)

točke = [[1, 0], [2, 1]]
uteži = [2, 1]
rbf = RBF(točke, uteži, gauss(0.7))
# za izračun vrednosti v dani točki lahko uporabimo `vrednost([1.5, 1.5], rbf)`,
# lahko pa objekt tipa RBF kličemo direktno kot funkcijo
z = rbf([1.5, 1.5])
0.3726164224242583

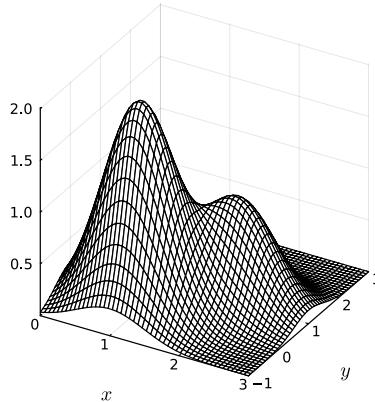
```

Narišimo še graf funkcije dveh spremenljivk, podane z linearno kombinacijo RBF.

```

using Plots
x = range(0, 3, 50)
y = range(-1, 3, 50)
wireframe(x, y, (x, y) -> rbf([x, y]), xlabel="\$x\$$", ylabel="\$y\$$")

```



Slika 15: Linearna kombinacija dveh RBF s središčema v točkah $(1, 0)$ in $(2, 1)$ ter funkcijo oblike $\varphi(r) = \exp(-r^2/0.7^2)$

Samostojno delo

Zapiši funkcijo `interpoliraj(točke, vrednosti, phi)`, ki poišče koeficiente v linearni kombinaciji (5.1) in vrne objekt tipa RBF, ki dane podatke interpolira (rešitev Program 31).

Funkcijo preskusimo na točkah, ki jih generiramo na parametrično podani krivulji (5.2). Sledimo [8] in točkam na krivulji dodamo točke znotraj krivulje, ki poskrbijo, da ne dobimo trivialne rešitve.

```

fi = range(0, 2π, 21)
točke = [[8cos(t) - cos(4t), 8sin(t) - sin(4t)] for t in fi[1:end-1]]
točke_noter = točke .* 0.9 # točke v smeri normal določimo približno
scatter(Tuple.(točke), label="točke na krivulji")
scatter!(Tuple.(točke_noter), label="točke v notranjosti")

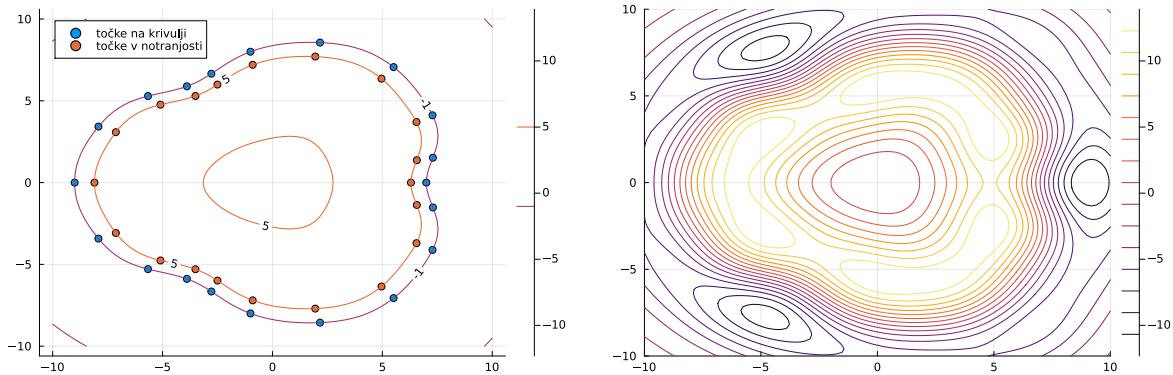
```

Vrednosti funkcije f_i za točke na krivulji izberemo tako, da so med seboj enake, hkrati pa se razlikujejo od vrednosti v notranjosti.

```

vse_točke = vcat(točke, točke_noter)
c1, c2 = -1, 5
vrednosti = vcat(
    c1 * ones(length(točke)), c2 * ones(length(točke)))
rbf = interpoliraj(vse_točke, vrednosti, gauss(3))
x = range(-10, 10, 100)
y = range(-10, 10, 100)
contour!(x, y, (x, y) -> rbf([x, y]), levels=[c1, c2], clabels=true)

```



Slika 16: Nivojske krivulje funkcije, podane z linearne kombinacijo RBF, ki interpolira dane vrednosti v izbranih točkah. Iskana krivulja, ki interpolira dane točke, je nivojska krivulja za vrednost -1 .

Kaj smo se naučili?

- Implicitna oblika krivulj ali ploskev je lahko zelo uporabna.
- Pri interpolaciji potrebujemo toliko parametrov, kot je na voljo podatkov.

5.4 Rešitve

```
using LinearAlgebra
"""
y = vrednost(x, rbf::RBF)

Izračunaj vrednost linearne kombinacije radialnih baznih funkcij
`rbf` v točki `x`.

"""

function vrednost(x, rbf::RBF)
    vsota = zero(x[1])
    n = length(rbf.točke)
    for i = 1:n
        norma = norm(rbf.točke[i] - x) # norma razlike
        vsota += rbf.uteži[i] * rbf.phi(norma) # utežena vsota
    end
    vsota
end
"""
rbf::RBF(x) = vrednost(x, rbf)
```

Program 30: Funkcija, ki izračuna vrednost linearne kombinacije RBF v dani točki.

```

"""
A = matrika(tocke, phi)

Poišči matriko sistema enačb za interpolacijo točk v seznamu `točke`,
z linearno kombinacijo radialnih baznih funkcij s funkcijo oblike
`phi`.

"""
function matrika(točke, phi)
    n = length(točke)
    A = zeros(n, n)
    for i = 1:n, j = i:n
        A[i, j] = phi(norm(točke[i] - točke[j]))
        A[j, i] = A[i, j]
    end
    return A
end

"""
rbf = interpoliraj(točke, vrednosti, phi)

Interpoliraj `vrednosti` v danih točkah s seznama `točke` z linearno
kombinacijo radialnih baznih funkcij s funkcijo oblike `phi`.

"""
function interpoliraj(točke, vrednosti, phi)
    A = matrika(točke, phi)
    F = cholesky!(A) # da prihranimo prostor, razcep naredimo kar v matriko A
    uteži = F \ vrednosti
    return RBF(točke, uteži, phi)
end

```

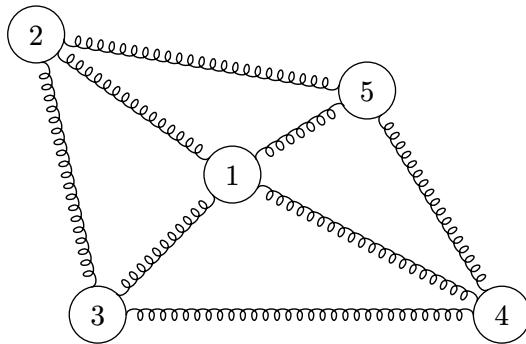
Program 31: Funkcija, ki interpolira dane vrednosti z linearno kombinacijo RBF.

6 Fizikalna metoda za vložitev grafov

Naj bo G neusmerjen povezan graf z množico vozlišč $V(G)$ in povezav $E(G) \subset V(G) \times V(G)$. Brez škode predpostavimo, da so vozlišča grafa G kar zaporedna naravna števila $V(G) = \{1, 2, \dots, n\}$. Vložitev grafa G v \mathbb{R}^d je preslikava $V(G) \rightarrow \mathbb{R}^d$, ki je podana z zaporedjem koordinat. Vložitev v \mathbb{R}^3 je podana z zaporedjem točk v \mathbb{R}^3

$$(x_1, y_1, z_1), (x_2, y_2, z_2), \dots, (x_n, y_n, z_n). \quad (6.1)$$

Za dani graf G želimo poiskati vložitev v \mathbb{R}^3 (ali \mathbb{R}^2). Pri fizikalni metodi grafu G priredimo fizikalni sistem in uporabimo fizikalne zakone za določanje položajev vozlišč. V tej vaji bomo grafu priredili sistem harmoničnih vzmeti, pri katerem na vsako vozlišče delujejo sosednja vozlišča s silo, ki je sorazmerna razdalji med vozlišči.



Slika 17: Sistem vzmeti za dani graf

6.1 Naloga

- Izpeli sistem enačb za koordinate vozlišč grafa, tako da so vozlišča v ravnotesju.
- Pokaži, da je matrika sistema diagonalno dominantna in negativno definitna.
- Napiši funkcijo, ki za dani graf in koordinate fiksiranih vozlišč poišče koordinate vseh vozlišč tako, da reši sistem enačb z metodo konjugiranih gradientov.
- V ravnini nariši [graf krožno lestev](#), tako da polovico vozlišč razporediš enakomerno po enotski krožnici.
- V ravnini nariši pravokotno mrežo. Fiksiraj vogale, nato točke na robu enakomerno razporedi po krožnici.

6.2 Ravnotesje sil

Harmonična vzmet je idealna vzmet dolžine 0, za katero sila ni sorazmerna sprememb dolžine, pač pa dolžini vzmeti. Sila harmonične vzmeti, ki je vpeta med točki (x_1, y_1, z_1) in (x_2, y_2, z_2) in deluje na prvo krajišče, je enaka:

$$\mathbf{F}_{21} = k \cdot \begin{pmatrix} x_2 - x_1 \\ y_2 - y_1 \\ z_2 - z_1 \end{pmatrix}, \quad (6.2)$$

kjer je k koeficient vzmeti.

Koordinate vozlišč določimo tako, da poiščemo ravnovesje sistema. To pomeni, da so v vsakem vozlišču j sile, s katerimi sosednja vozlišča delujejo nanj, v ravnovesju:

$$\sum_{i \in N(j)} \mathbf{F}_{ij} = \mathbf{0}, \quad (6.3)$$

kjer je $N(j) = \{i : (i, j) \in E(G)\}$ množica sosednjih točk v grafu za točko j in \mathbf{F}_{ij} sila, s katero vozlišče i deluje na vozlišče j . Iz enačbe (6.3) lahko izpeljemo sistem enačb za koordinate x_j, y_j in z_j . Iz vektorske enačbe za vozlišče j :

$$\sum_{i \in N(j)} \mathbf{F}_{ij} = \sum_{i \in N(j)} k \begin{pmatrix} x_i - x_j \\ y_i - y_j \\ z_i - z_j \end{pmatrix} = \mathbf{0}, \quad (6.4)$$

dobimo 3 enačbe za posamezne koordinate:

$$\begin{aligned} -st(j)x_j + x_{i_1} + x_{i_2} + \dots + x_{i_{st(i)}} &= 0, \\ -st(j)y_j + y_{i_1} + y_{i_2} + \dots + y_{i_{st(i)}} &= 0, \\ -st(j)z_j + z_{i_1} + z_{i_2} + \dots + z_{i_{st(i)}} &= 0, \end{aligned} \quad (6.5)$$

kjer je $st(j) = |N(j)|$ stopnja vozlišča j in $i_1, i_2, \dots, i_{st(j)} \in N(j)$. Ker so koordinate x, y in z med seboj neodvisne, dobimo za vsako koordinato en sistem enačb. Za koordinato x dobimo naslednji sistem:

$$\begin{aligned} -st(1)x_1 + \sum_{i \in N(1)} x_i &= 0, \\ -st(2)x_2 + \sum_{i \in N(2)} x_i &= 0, \\ &\vdots \\ -st(n)x_n + \sum_{i \in N(n)} x_i &= 0. \end{aligned} \quad (6.6)$$

Enačbe (6.6) so homogene, kar pomeni, da ima sistem le ničelno rešitev³. Če želimo netrivialno rešitev, moramo nekatera vozlišča v grafu pritrdati in jim predpisati koordinate. Brez škode lahko predpostavimo, da so vozlišča, ki jih pritrdimo, na koncu. Označimo z $F = \{m+1, \dots, n\} \subset V(G)$; $m < n$ množico vozlišč, ki imajo določene koordinate. Koordinate za vozlišča iz F niso več spremenljivke, ampak jih moramo prestaviti na drugo stran enačbe. Sistem enačb (6.6) postane nehomogen:

$$\begin{aligned} -st(1)x_1 + a_{12}x_2 + \dots + a_{1m}x_m &= -a_{1m+1}x_{m+1} - \dots - a_{1n}x_n, \\ a_{21}x_1 - st(2)x_2 + \dots + a_{2m}x_m &= -a_{2m+1}x_{m+1} - \dots - a_{2n}x_n, \\ &\vdots \\ a_{m1}x_1 - a_{m2}x_2 + \dots - st(m)x_m &= -a_{mm+1}x_{m+1} - \dots - a_{mn}x_n, \end{aligned} \quad (6.7)$$

kjer je vrednost a_{ij} enaka 1, če sta i in j sosed, in 0 sicer:

$$a_{ij} = \begin{cases} 1, & (i, j) \in E(G), \\ 0, & (i, j) \notin E(G). \end{cases} \quad (6.8)$$

Matrika sistema (6.7) je odvisna le od povezav v grafu in izbire točk, ki niso pritrjene, medtem ko so desne stani odvisne od koordinat pritrjenih točk:

³Rešitev je enolična le, če je matrika sistema (6.6) polnega ranga. To velja, če graf nima izoliranih točk.

$$A = \begin{pmatrix} -\text{st}(1) & a_{12} & \dots & a_{1m} \\ a_{21} & -\text{st}(2) & \dots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & -\text{st}(m) \end{pmatrix} \quad \text{in} \quad \mathbf{b} = - \begin{pmatrix} \sum_{i=m+1}^n a_{1i} x_i \\ \sum_{i=m+1}^n a_{2i} x_i \\ \vdots \\ \sum_{i=m+1}^n a_{ni} x_i \end{pmatrix}. \quad (6.9)$$

Sistema za y in z imata iste koeficiente kot sistem (6.6), razlikujeta se le v desnih straneh, ki so odvisne od koordinat pritrjenih točk.

Kakšne posebnosti ima matrika sistema (6.9)? Matrika je simetrična in diagonalno dominantna. Res! Velja namreč $\text{st}(i) = |N(i)|$ in zato:

$$|a_{ii}| = |N(i)| \geq |N(i) \cap F^C| = \sum_{j \neq i} |a_{ij}|. \quad (6.10)$$

Za sosedne fiksni vozlišč je neenakost stroga, zato je matrika diagonalno dominantna in vsaj za eno vrstico je neenakost stroga. Ker so vsi elementi na diagonali negativni in je matrika diagonalno dominantna (z vsaj eno vrstico, ki je strogo diagonalno dominantna), je matrika A negativno definitna in matrika $-A$ pozitivno definitna. Za večino grafov, za katere uporabimo zgornji postopek, je matrika sistema A redka. Zato lahko za reševanje sistema $-Ax = -b$ uporabimo [metodo konjugiranih gradientov](#), ki deluje za sisteme s pozitivno definitno matriko. Metoda konjugiranih gradientov in druge iterativne metode so zelo primerne za redke matrike. Za razliko od eliminacijskih metod, iterativne metode ne izvedejo sprememb na matriki, ki bi dodale neničelne elemente.

6.3 Rešitev v Julii

Za predstavitev grafa bomo uporabili paket [Graphs.jl](#)[11], ki definira podatkovne tipe in vmesnike za lažje delo z grafi.

Samostojno delo

Napiši naslednje funkcije:

- `krožna_lestev(n)`, ki ustvari graf krožne lestve z $2n$ vozlišči (rešitev Program 32).
- `matrika(G::AbstractGraph, sprem)`, ki vrne matriko sistema (6.9) za dani graf G in seznam vozlišč sprem, ki niso pritrjena (rešitev Program 33),
- `desne_strani(G::AbstractGraph, sprem, koordinate)`, ki vrne vektor desnih strani za sistem (6.7) (rešitev Program 34),
- `cg(A, b; atol=1e-8)`, ki poišče rešitev sistema $Ax = b$ z metodo konjugiranih gradientov (rešitev Program 35) in
- `vloži!(G::AbstractGraph, fix, točke)`, ki poišče vložitev grafa G v \mathbb{R}^d s fizikalno metodo. Argument `fix` naj bo seznam fiksnih vozlišč, argument `točke` pa matrika s koordinatami točk. Metoda naj ne vrne ničesar, ampak naj vložitev zapiše kar v matriko `točke` (rešitev Program 36).

6.4 Krožna lestev

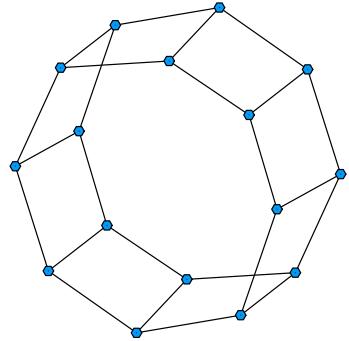
Uporabimo napisano kodo za primer grafa krožna lestev. Graf je sestavljen iz dveh ciklov enake dolžine n , ki sta med seboj povezana z n povezavami. Za grafično predstavitev grafov bomo uporabili paket [GraphRecipes.jl](#).

```

using Vaja06
G = krožna_lestev(8)

using GraphRecipes, Plots
graphplot(G, curves=false)

```



Slika 18: Graf krožna lestev s 16 vozlišči

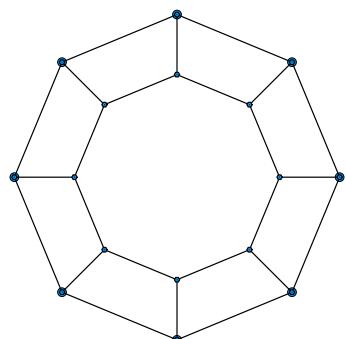
Poишčimo drugačno vložitev s fizikalno metodo, tako da vozlišča enega cikla enakomerno razporedimo po krožnici.

```

t = range(0, 2pi, 9)[1:end-1]
x = cos.(t)
y = sin.(t)
točke = hcat(hcat(x, y)', zeros(2, 8))
# funkcija hcat zloži vektorje po stolpcih v matriko
fix = 1:8

vloži!(G, fix, točke)
graphplot(G, x=točke[1, :], y=točke[2, :], curves=false)

```



Slika 19: Graf krožna lestev s 16 vozlišči, vložen s fizikalno metodo. Zunanja vozlišča so fiksna, notranja pa postavljena tako, da so sile vzmeti na povezavah v ravovesju.

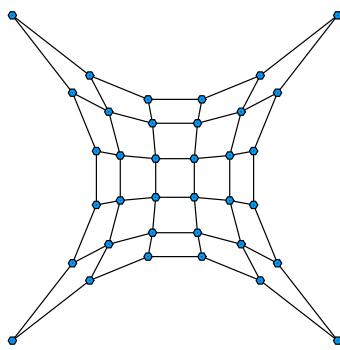
6.5 Dvodimenzionalna mreža

Preizkusimo algoritem na dvodimenzionalni mreži. Dvodimenzionalna mreža je graf, ki ga dobimo, če pravokotnik v ravnini razdelimo v pravokotno mrežo. Najprej pritrdimo štiri točke druge stopnje, ki ustrezajo oglščem pravokotnika.

```
using Graphs
m, n = 6, 6
G = Graphs.grid((m, n), periodic=false)

# vogali imajo stopnjo 2
vogali = filter(v -> degree(G, v) <= 2, vertices(G))
tocke = zeros(2, n * m)
tocke[:, vogali] = [0 0 1 1; 0 1 0 1]

vloži!(G, vogali, tocke)
graphplot(G, x=tocke[1, :], y=tocke[2, :], curves=false)
```



Slika 20: Dvodimenzionalna mreža, vložena s fizikalno metodo. Pritrjeni so le vogali.

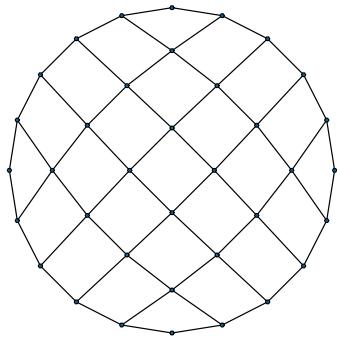
Sedaj pritrdimo cel rob in ga enakomerno razporedimo po krožnici.

```

m, n = 6, 6
G = Graphs.grid((m, n), periodic=false)
rob = filter(v -> degree(G, v) <= 3, vertices(G))
urejen_rob = [rob[1]]

# uredi točke na robu v cikel
for i = 1:length(rob)-1
    sosedи = neighbors(G, urejen_rob[end])
    sosedи = intersect(sosedи, rob)
    sosedи = setdiff(sosedи, urejen_rob)
    push!(urejen_rob, sosedи[1])
end
t = range(0, 2pi, length(rob) + 1)[1:end-1]
točke = zeros(2, n * m)
točke[:, urejen_rob] = hcat(cos.(t), sin.(t))'
vloži!(G, urejen_rob, točke)
graphplot(G, x=točke[1, :], y=točke[2, :], curves=false)

```



Slika 21: Dvodimenzionalna mreža, vložena s fizikalno metodo. Rob mreže je enakomerno razporejen po krožnici.

Kaj smo se naučili?

- Kako zapisati sistem linearnih enačb za spremenljivke, ki ustrezajo vozliščem grafa.
- Spoznali smo še eno iterativno metodo, ki deluje za pozitivno definitne matrike.

6.6 Rešitve

```
using Graphs
"""
G = krožna_lestev(n)

Ustvari graf krožna lestev z `2n` točkami.

function krožna_lestev(n)
    G = SimpleGraph(2 * n)
    # prvi cikel
    for i = 1:n-1
        add_edge!(G, i, i + 1)
    end
    add_edge!(G, 1, n)
    # drugi cikel
    for i = n+1:2n-1
        add_edge!(G, i, i + 1)
    end
    add_edge!(G, n + 1, 2n)
    # povezave med obema cikloma
    for i = 1:n
        add_edge!(G, i, i + n)
    end
    return G
end
```

Program 32: Funkcija, ki ustvari graf krožna lestev.

```

using SparseArrays
"""

A = matrika(G::AbstractGraph, sprem)

Poišči matriko sistema linearnih enačb za vložitev grafa `G` s fizikalno metodo.
Argument `sprem` je vektor vozlišč grafa, ki nimajo določenih koordinat.
Indeksi v matriki `A` ustrezajo vozliščem v istem vrstnem redu,
kot nastopajo v argumentu `sprem`.

"""
function matrika(G::AbstractGraph, sprem)
    # preslikava med vozlišči in indeksi v matriki
    v_to_i = Dict([sprem[i] => i for i in eachindex(sprem)])
    m = length(sprem)
    A = spzeros(m, m)
    for i = 1:m
        vertex = sprem[i]
        sosedi = neighbors(G, vertex)
        for vertex2 in sosedi
            if haskey(v_to_i, vertex2)
                j = v_to_i[vertex2]
                A[i, j] = 1
            end
        end
        A[i, i] = -length(sosedi)
    end
    return A
end

```

Program 33: Funkcija, ki ustvari matriko sistema za ravnovesje sil v grafu.

```

"""
    b = desne_strani(G::AbstractGraph, sprem, koordinate)

Poišči desne strani sistema linearnih enačb za eno koordinato vložitve grafa `G`
s fizikalno metodo. Argument `sprem` je vektor vozlišč grafa, ki nimajo
določenih koordinat. Argument `koordinate` vsebuje eno koordinato za vsa
vozlišča grafa. Metoda uporabi le koordinato vozlišč, ki so pritrjena.
Indeksi v vektorju `b` ustrezajo vozliščem v istem vrstnem redu,
kot nastopajo v argumentu `sprem`.

function desne_strani(G::AbstractGraph, sprem, koordinate)
    set = Set(sprem)
    m = length(sprem)
    b = zeros(m)
    for i = 1:m
        v = sprem[i]
        for v2 in neighbors(G, v)
            if !(v2 in set) # dodamo le točke, ki so fiksirane
                b[i] -= koordinate[v2]
            end
        end
    end
    return b
end

```

Program 34: Funkcija, ki izračuna desne strani sistema za ravnovesje sil v grafu na podlagi koordinat pritrjenih vozlišč.

```

using Logging
"""

x = cg(A, b; atol=1e-10)

Metoda konjugiranih gradientov za reševanje sistema enačb `Ax = b`
s pozitivno definitno matriko `A`. Argument `A` ni nujno matrika, lahko je
tudi drugega tipa, če ima implementirano množenje z vektorjem `b`.

Metoda ne preverja, ali je argument `A` pozitivno definiten.

"""

function cg(A, b; atol=1e-8)
    # za začetni približek vzamemo kar desne strani
    x = copy(b)
    r = b - A * b
    p = r
    res0 = sum(r .* r)
    for i = 1:length(b)
        Ap = A * p
        alpha = res0 / sum(p .* Ap)
        x = x + alpha * p
        r = r - alpha * Ap
        res1 = sum(r .* r)
        if sqrt(res1) < atol
            @info "Metoda KG konvergira po $i korakih."
            break
        end
        p = r + (res1 / res0) * p
        res0 = res1
    end
    return x
end

```

Program 35: Funkcija, ki reši sistem linearnih enačb $Ax = b$ z metodo konjugiranih gradientov.

```
"""
vloži!(G::AbstractGraph, fix, točke)
```

Pošči vložitev grafa `G` v prostor s fizikalno metodo. Argument `fix` vsebuje vektor vozlišč grafa, ki imajo določene koordinate. Argument `točke` je začetna vložitev grafa. Koordinate vozlišč, ki niso pritrjena, bodo nadomeščene z novimi koordinatami.

Metoda ne vrne ničesar, ampak zapiše izračunane koordinate v matriko `točke`.

```
"""
function vloži!(G::AbstractGraph, fix, točke)
    sprem = setdiff(vertices(G), fix)
    dim, _ = size(točke)
    A = matrika(G, sprem)
    for k = 1:dim
        b = desne_strani(G, sprem, točke[k, :])
        x = cg(-A, -b) # matrika A je negativno definitna
        točke[k, sprem] = x
    end
end
```

Program 36: Funkcija, ki poišče koordinate vložitve grafa v \mathbb{R}^d s fizikalno metodo.

7 Invariantna porazdelitev Markovske verige

Z [Markovskimi verigami](#) smo se že srečali v poglavju o tridiagonalnih sistemih (Poglavlje 3). Spomnimo se, da je Markovska veriga zaporedje slučajnih spremenljivk X_k , ki opisujejo slučajni prehod po množici stanj. Stanja Markovske verige bomo označili kar z zaporednimi naravnimi števili $1, 2, \dots, n$. Na vsakem koraku je Markovska veriga v določenem stanju $X_k \in \{1, 2, \dots, n\}$. Porazdelitev na naslednjem koraku X_{k+1} je odvisna zgolj od porazdelitve na prejšnjem koraku X_k in prehodnih verjetnosti za prehod iz stanja i v stanje j :

$$p_{ij} = P(X_{k+1} = j \mid X_k = i). \quad (7.1)$$

Matrika $\mathcal{P} = [p_{ij}]$, katere elementi so prehodne verjetnosti za prehod iz stanja i v stanje j , imenujemo *prehodna matrika Markovske verige*.

Naj bo X_k Markovska veriga z n stanji in naj bo $\mathbf{p}^{(k)} = [p_1^{(k)}, p_2^{(k)}, \dots, p_n^{(k)}]$ porazdelitev po stanjih na k -tem koraku ($p_i^{(k)} = P(X_k = i)$).

Porazdelitev \mathbf{p}^k Markovske verige je *invariantna*, če je za vse k enaka:

$$\mathbf{p}^{k+1} = \mathbf{p}^k. \quad (7.2)$$

Porazdelitev na naslednjem koraku X_{k+1} dobimo tako, da seštejemo verjetnosti po vseh možnih stanjih na prejšnjem koraku, pomnožene s pogojnimi verjetnostmi, da iz enega stanja preidemo v drugega:

$$\begin{aligned} p_i^{(k+1)} &= \sum_{j=1}^n P(X_{k+1} = i \mid X_k = j) P(X_k = j) = \sum_{j=1}^n p_{ji} p_j^{(k)} \\ \mathbf{p}^{(k+1)} &= \mathcal{P}^T \mathbf{p}^{(k)}. \end{aligned} \quad (7.3)$$

Od tod sledi, da je porazdelitev \mathbf{p} *invariantna porazdelitev* Markovske verige s prehodno matriko \mathcal{P} , če velja:

$$\mathbf{p} = \mathcal{P}^T \mathbf{p}. \quad (7.4)$$

Povedano drugače: invariantna porazdelitev za Markovsko verigo s prehodno matriko \mathcal{P} je lastni vektor matrike \mathcal{P} z lastno vrednostjo 1.

7.1 Naloga

- Implementiraj potenčno metodo za iskanje največje lastne vrednosti in pripadajočega lastnega vektorja dane matrike.
- Uporabi potenčno metodo in poišči invariantno porazdelitev Markovske verige z dano prehodno matriko \mathcal{P} . Poišči invariantne porazdelitve za naslednja primera:
 - ▶ veriga, ki opisuje skakanje skakača (konja) po šahovnici,
 - ▶ veriga, ki opisuje brskanje po mini spletu s 5-10 stranmi (podobno spletni iskalniki [razvrščajo strani po relevantnosti](#)).

7.2 Limitna porazdelitev Markovske verige

Porazdelitev \mathbf{p} je *limitna porazdelitev* Markovske verige, če porazdelitve na posameznih korakih \mathbf{p}^k konvergirajo k \mathbf{p} . Limitna porazdelitev je vedno invariantna in potem takem lastni vektor \mathcal{P}^T z lastno vrednostjo 1:

$$\mathbf{p}^\infty = \lim_{k \rightarrow \infty} \mathbf{p}^{(k)} = \lim_{k \rightarrow \infty} \mathbf{p}^{(k+1)} = \lim_{k \rightarrow \infty} \mathcal{P}^T \mathbf{p}^{(k)} = \mathcal{P}^T \lim_{k \rightarrow \infty} \mathbf{p}^{(k)} = \mathcal{P}^T \mathbf{p}^\infty. \quad (7.5)$$

Ker so vsote elementov po vrsticah za prehodno matriko \mathcal{P} enake 1, je 1 lastna vrednost matrike \mathcal{P} in zato tudi lastna vrednost matrike \mathcal{P}^T . Posledično limitna porazdelitev \mathbf{p}^∞ vedno obstaja, ni pa nujno enolična.

Da se pokazati, da je 1 po absolutni vrednosti največja lastna vrednost matrik \mathcal{P} in \mathcal{P}^T , zato lahko invariantno porazdelitev poiščemo s [potenčno metodo](#).

7.3 Potenčna metoda

S potenčno metodo poiščemo lastni vektor matrike A , ki pripada lastni vrednosti z največjo absolutno vrednostjo. Izberemo neničelen začetni vektor $\mathbf{p}^{(0)} \neq 0$ in sestavimo zaporedje približkov:

$$\mathbf{x}^{(k+1)} = \frac{A\mathbf{x}^{(k)}}{\|A\mathbf{x}^{(k)}\|}, \quad k = 0, 1, \dots \quad (7.6)$$

Zaporedje $\mathbf{x}^{(k)}$ konvergira k lastnemu vektorju matrike A z lastno vrednostjo, ki je po absolutni vrednosti največja. Če je takih lastnih vrednosti več (npr. 1 in -1), se lahko zgodi, da potenčna metoda ne konvergira. Za normo, s katero delimo produkt $A\mathbf{x}^{(k)}$, lahko izberemo katerokoli vektorsko normo. Navadno je to neskončna norma $\|\cdot\|_\infty$, saj jo lahko najhitreje izračunamo.

Samostojno delo

Napiši program `x, it = potencna(A, x0)`, ki poišče lastni vektor za po absolutni vrednosti največjo lastno vrednost matrike A (za rešitev glej Program 37).

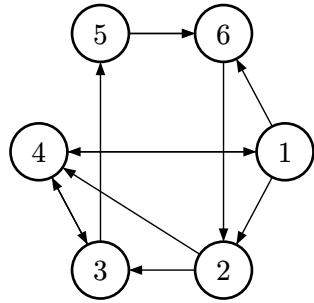
7.4 Razvrščanje spletnih strani

Spletni iskalniki želijo uporabniku prikazati čim relevantnejše rezultate. Zato morajo ugotoviti, katere spletne strani so pomembnejše od drugih. Brskanje po spletu lahko modeliramo z Markovsko verigo, kjer na vsakem koraku obiščemo eno spletno stran. Na vsaki spletne strani, ki jo obiščemo, naključno izberemo povezavo, ki nas vodi do naslednje strani. Če spletne strani nima povezav, se lahko vrnemo nazaj na prejšnjo stran ali pa naključno izberemo novo stran. Limitna porazdelitev pove, kolikšen delež vseh obiskov pripada posamezni spletnej strani, če se naključno sprehajamo po spletu. Večji delež obiskov ima spletne strane, pomembnejša je.

Limitno porazdelitev Markovske verige s prehodno matriko \mathcal{P} poiščemo s potenčno metodo, kot lastni vektor matrike \mathcal{P}^T za lastno vrednost 1.

Približno tako deluje algoritem za razvrščanje spletnih strani po pomembnosti [Page Rank](#), ki sta ga prva opisala in uporabila ustanovitelja podjetja Google Larry Page in Sergey Brin.

Poglejmo preprost primer spletja s šestimi stranmi in povezavami med njimi (Slika 22).



Slika 22: Mini splet s 6 stranmi

Prehodna matrika verige za mini splet je:

$$\mathcal{P} = \begin{pmatrix} 0 & \frac{1}{3} & 0 & \frac{1}{3} & 0 & \frac{1}{3} \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}. \quad (7.7)$$

Poščimo invariantno porazdelitev s potenčno metodo:

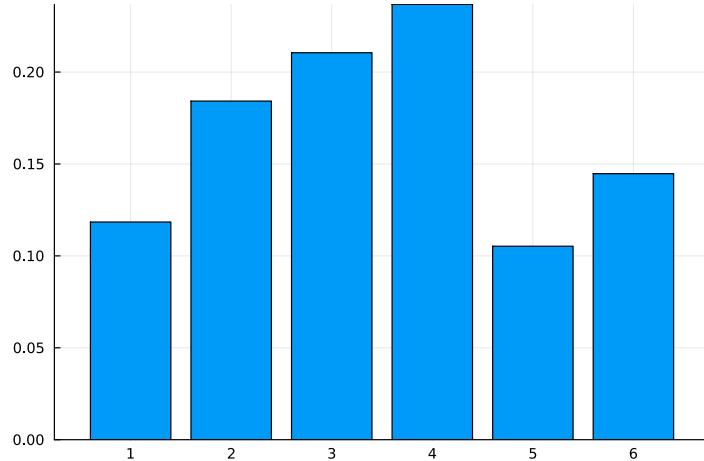
```
P = [
0 1/3 0 1/3 0 1/3;
0 0 1/2 1/2 0 0;
0 0 0 1/2 1/2 0;
1/2 0 1/2 0 0 0;
0 0 0 0 0 1;
0 1 0 0 0 0
]
x, it = potenčna(P', rand(6))
```

Preverimo, ali je dobljeni vektor res lastni vektor za lastno vrednost 1, tako da izračunamo razliko $\mathcal{P}^T x - x$ in preverimo, da je razlika blizu 0:

```
delta = P' * x - x
6-element Vector{Float64}:
-1.761793266830125e-9
8.700961284802133e-9
-5.06670871924797e-9
-4.618036397729952e-9
-2.595118120396478e-9
5.3406952194023916e-9
```

Invariantno porazdelitev predstavimo s stolpčnim diagramom:

```
x = x / sum(x)
using Plots
bar(x, label=false)
```

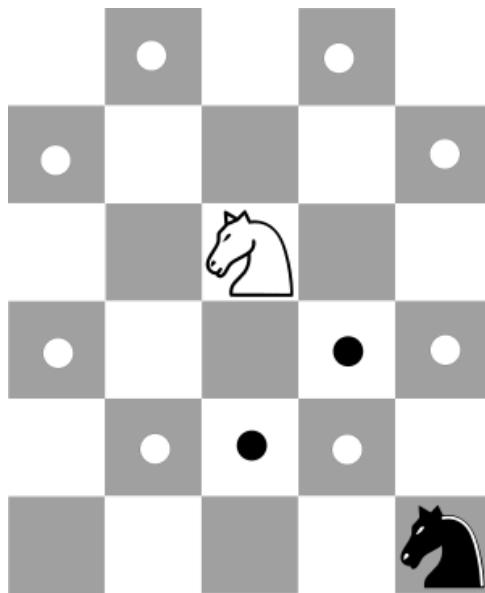


Slika 23: Delež obiskov posamezne strani v limitni porazdelitvi

Iz diagrama vidimo, da je najpogosteje obiskana spletna stran 4, najredkeje pa spletna stran 5.

7.5 Skakanje skakača po šahovnici

Tudi naključno skakanje skakača po šahovnici lahko opišemo z Markovsko verigo. Stanja Markovske verige so polja na šahovnici, prehodne verjetnosti pa določimo tako, da skakač v naslednji potezi naključno skoči na eno od polj, ki so mu dostopna. Predpostavimo, da so vsa dostopna polja enako verjetna.



Slika 24: Možne poteze, ki jih lahko naredita beli in črni skakač na 5×5 šahovnici.

Stanja označimo s pari indeksov (i, j) , ki označujejo posamezno polje. Prvi indeks označuje vrstico, drugi pa stolpce. Indeks $(1, 1)$ označuje polje levo zgoraj. Invariantna porazdelitev je podana z matriko, katere elementi p_{ij} so enaki verjetnosti, da je skakač na polju (i, j) . Ponovno se srečamo s problemom iz prejšnjega poglavja (Poglavlje 4), kako elemente matrike postaviti v vektor. Elemente matrike zložimo v vektor po stolpcih. Preslikava med indeksi i, j v matriki in indeksom k v vektorju je podana s formulami:

$$\begin{aligned}
 k &= i + (j - 1)m, \\
 j &= \lfloor (k - 1)/m \rfloor \\
 i &= ((k - 1) \bmod m) + 1.
 \end{aligned} \tag{7.8}$$

Samostojno delo

Za lažje delo napiši funkciji

- $k = \text{ij_v_k}(i, j, m)$ in
- $i, j = \text{k_v_ij}(k, m)$,

ki izračunata preslikavo med indeksi i, j v matriki in indeksom k v vektorju (Program 38).

Nato definiraj:

- podatkovno strukturo $\text{Skakač}(m, n)$, ki predstavlja Markovsko verigo za skakača na $m \times n$ šahovnici (Program 39) in
- funkcijo $\text{prehodna_matrika}(k : \text{Skakač})$, ki vrne prehodno matriko za Markovsko verigo za skakača (Program 40).

Invariantno porazdelitev poskusimo poiskati s potenčno metodo:

```
P = prehodna_matrika(Skakač(8, 8))

x, it = potenčna(P^, rand(64))
```

Potenčna metoda ne konvergira, saj ima matrika \mathcal{P}^T dve dominantni lastni vrednosti 1 in -1 . Skoraj vsi začetni približki vsebujejo tako komponento v smeri lastnega vektorja za 1 kot tudi komponento v smeri lastnega vektorja za -1 . Zaporedje približkov v limiti začne preskakovati med dvema vrednostima:

$$\frac{\mathbf{v}_1 + \mathbf{v}_{-1}}{\|\mathbf{v}_1 + \mathbf{v}_{-1}\|} \quad \text{in} \quad \frac{\mathbf{v}_1 - \mathbf{v}_{-1}}{\|\mathbf{v}_1 - \mathbf{v}_{-1}\|}, \tag{7.9}$$

kjer je \mathbf{v}_1 lastni vektor za 1 in \mathbf{v}_{-1} lastni vektor za -1 . S funkcijo `eigen` poiščimo lastne pare matrike \mathcal{P} in preverimo, da sta dominantni lastni vrednosti res -1 in 1:

```
# funkcija `eigen` iz modula LinearAlgebra izračuna lastni razcep matrike
lambda, v = eigen(Matrix(P^))
# lambda ima tudi imaginarne komponente, ki pa so zanemarljivo majhne
lambda = real.(lambda)
println("Največja in najmanjša lastna vrednost matrike P':")
println("$(maximum(lambda)), $(minimum(lambda))")

Največja in najmanjša lastna vrednost matrike P':
1.000000000000018, -1.0000000000000018
```

Težavo rešimo s preprostim premikom. Če matriki prištejemo večkratnik identitete, se lastni vektorji ne spremeniijo, le lastne vrednosti se premaknejo. Če so $(\lambda_1, \mathbf{v}_1), (\lambda_2, \mathbf{v}_2), \dots$ lastni pari matrike A , so

$$(\lambda_1 + \delta, \mathbf{v}_1), (\lambda_2 + \delta, \mathbf{v}_2), \dots \tag{7.10}$$

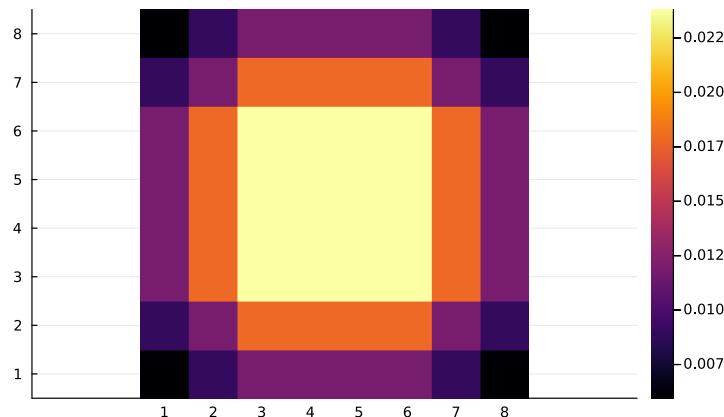
lastni pari matrike

$$A + \delta I. \tag{7.11}$$

S premikom $\mathcal{P}^T + I$ dosežemo, da se lastne vrednosti premaknejo za 1 v pozitivni smeri: lastna vrednost -1 se premakne v 0 , lastna vrednost 1 pa v 2 . Tako lastna vrednost 2 postane edina dominantna lastna vrednost. Za matriko $\mathcal{P}^T + I$ potenčna metoda konvergira k lastnemu vektorju za lastno vrednost 2 , ki je hkrati lastni vektor matrike \mathcal{P}^T za lastno vrednost 1 .

```
x, it = potenčna(P' + I, rand(64))
x = x / sum(x) # vrednosti normiramo, da je vsota enaka 1
porazdelitev = reshape(x, 8, 8)

using Plots
heatmap(porazdelitev, aspect_ratio=1, xticks=1:8, yticks=1:8)
```



Slika 25: Limitna porazdelitev za skakača na standardni 8×8 šahovnici. Svetlejša polja so pogosteje obiskana. Limitna porazdelitev je ena sama in ni odvisna od porazdelitve začetne lege skakača.

Kaj smo se naučili?

- Iterativne metode se dobro obnesejo tudi pri iskanju lastnih vektorjev in lastnih vrednosti.
- Potenčna metoda se obnese tudi pri matrikah zelo velikih dimenzij.
- Problem lastnih vrednosti se pojavi pri študiju Markovskih verig.

7.6 Rešitve

```
using LinearAlgebra

"""
x, it = potenčna(A)

Poišči lastni vektor matrike `A` za največjo lastno vrednost s potenčno metodo.

"""

function potenčna(A, x0; atol=1e-8, maxit=1000)
    for i = 1:maxit
        x = A * x0
        x = x / norm(x, Inf)
        if norm(x - x0, Inf) < atol
            return x, i
        end
        x0 = x
    end
    throw("Potenčna metoda ne konvergira po $maxit korakih!")
end
```

Program 37: Funkcija, ki s potenčno metodo poišče lastni vektor za po absolutni vrednosti največjo lastno vrednost dane matrike.

```
ij_v_k(i, j, n) = i + (j - 1) * n

function k_v_ij(k, m)
    j, i = divrem(k - 1, m)
    return (i + 1, j + 1)
end
```

Program 38: Preslikave med indeksi v matriki in indeksi v vektorju, ki je sestavljen iz stolpcev matrike.

```
"""
Skakač(m, n)

Podatkovna struktura, ki označuje Markovsko verigo za skakača na šahovnici
dimenzijs `m` x `n`.

"""

struct Skakač
    m
    n
end
```

Program 39: Podatkovni tip, ki predstavlja Markovsko verigo za skakača na šahovnici.

```

using SparseArrays
"""

P = prehodna_matrika(k::Skakač)

Poišči prehodno matriko za Markovsko verigo, ki opisuje
skakanje figure skakača po šahovnici.
"""

function prehodna_matrika(skakač::Skakač)
    m = skakač.m
    n = skakač.n
    N = m * n
    P = spzeros(N, N)
    skoki = [(1, 2), (2, 1), (-1, 2), (-2, 1),
              (1, -2), (2, -1), (-1, -2), (-2, -1)]
    for k = 1:N
        i0, j0 = k_v_ij(k, m)
        for skok in skoki
            i = i0 + skok[1]
            j = j0 + skok[2]
            if i >= 1 && i <= m && j >= 1 && j <= n
                k1 = ij_v_k(i, j, m)
                P[k, k1] = 1
            end
        end
        P[k, :] /= sum(P[k, :]) # normiramo vrstico, da je vsota enaka 1
    end
    return P
end

```

Program 40: Funkcija, ki ustvari prehodno matriko za Markovsko verigo za skakača na šahovnici.

8 Spektralno razvrščanje v gruče

Razvrščanje v gruče ali gručenje je postopek, pri katerem množico objektov razdelimo v nekaj skupin ali gruč, v katerih so objekti, ki so si v nekem smislu podobni. Ogledali si bomo metodo, ki s spektralno analizo Laplaceove matrike podobnognega grafa podatke preslika v prostor, v katerem jih nato preprosteje razvrstimo z algoritmi gručenja. Sledili bomo postopku, imenovanemu spektralno gručenje, kot je opisan v [12].

8.1 Naloge

- Napiši funkcijo, ki zgradi podobnostni graf za podatke, podane kot oblak točk v \mathbb{R}^d . V podobnostnem grafu je vsaka točka v oblaku vozlišče, povezani pa so vsi pari točk i in j , ki so oddaljeni za manj kot ε za primerno izbrani ε .
- Napiši funkcijo, ki za dano simetrično matriko poišče k po absolutni vrednosti najmanjših lastnih vrednosti in pripadajoče lastne vektorje. Uporabi inverzno iteracijo za k vektorjev in na vsakem koraku s QR razcepom poskrbi, da so paroma ortogonalni. Faktor Q v QR razcepnu konvergira k lastnim vektorjem, diagonalna faktorja R pa h k po absolutni vrednosti najmanjšim lastnim vrednostim matrike.
- Funkcijo za iskanje lastnih vektorjev uporabi na Laplaceovi matriki podobnognega grafa podatkov. Za primer podatkov naključno generiraj mešanico treh različnih Gaussovih porazdelitev. Komponente lastnih vektorjev uporabi kot nove koordinate in podatke predstavi v novih koordinatah.

8.2 Podobnostni graf in Laplaceova matrika

Podatke (množico točk v \mathbb{R}^d) želimo razvrstiti v več gruč. Osnova za spektralno gručenje je *podobnostni uteženi graf*, ki povezuje točke, ki so si v nekem smislu blizu. Podobnostni graf lahko ustvarimo na več načinov:

- **ε okolice:** s točko x_k povežemo vse točke, ki ležijo v ε okolini te točke,
- **k najbližji sosedji:** x_k povežemo z x_i , če je x_i med k najbližnjimi točkami. Tako dobimo usmerjen graf, zato navadno upoštevamo povezavo v obe smeri.
- **poln utežen graf:** povežemo vse točke, vendar povezave utežimo glede na razdaljo. Pogosto uporabljena utež je nam znana [radialna bazna funkcija](#):

$$w(x_i, x_k) = \exp\left(-\frac{\|x_i - x_k\|^2}{2\sigma^2}\right), \quad (8.1)$$

pri kateri s parametrom σ določamo velikost okolic.

Uteženemu grafu podobnosti z matriko uteži:

$$W = [w_{ij}] \quad (8.2)$$

priredimo Laplaceovo matriko:

$$L = D - W, \quad (8.3)$$

kjer je $D = [d_{ij}]$ diagonalna matrika z elementi $d_{ii} = \sum_{j \neq i} w_{ij}$. Če graf ni utežen, namesto matrike uteži uporabimo [matriko sosednosti](#). Laplaceova matrika L je simetrična, nenegativno definitna in ima vedno eno lastno vrednost enako 0 za lastni vektor iz samih enic. Laplaceova matrika je pomembna v

spektralni teoriji grafov, ki preučuje lastnosti grafov s pomočjo analize lastnih vrednosti in vektorjev matrik. Knjižnica [Laplacians.jl](#) je namenjena spektralni teoriji grafov.

8.3 Algoritem

Velja izrek, da ima Laplaceova matrika natanko toliko lastnih vektorjev za lastno vrednost 0, kot ima graf komponent za povezanost. Na prvi pogled se zdi, da bi lahko bile gruče kar komponente grafa, a se izkaže, da to ni najbolje. Namesto tega bomo gruče poiskali s standardnimi metodami gručenja v drugem koordinatnem sistemu, ki ga določajo lastni podprostori Laplaceove matrike podobnostnega grafa. Postopek je naslednji:

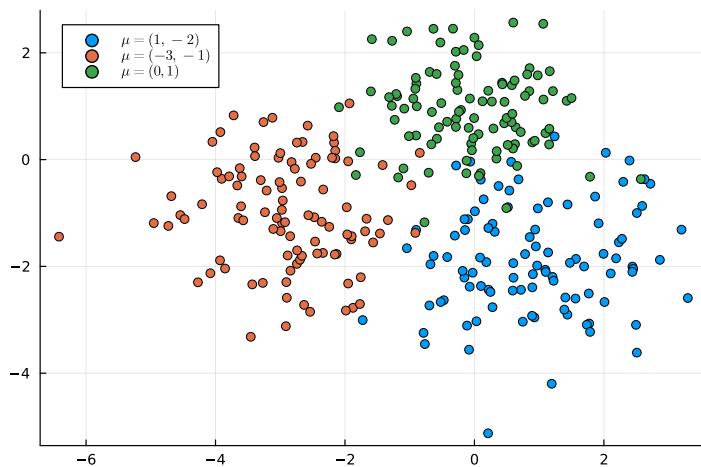
- Poiščemo k najmanjših lastnih vrednosti za Laplaceovo matriko in izračunamo njihove lastne vektorje.
- Označimo matriko lastnih vektorjev $Q = [v_1, v_2, \dots, v_k]$. Stolpci Q^T ustrezajo koordinatam točk v novem prostoru.
- Za stolpce matrike Q^T izvedemo izbran algoritem gručenja (npr. algoritem [*k-povprečij*](#)).

V tej vaji bomo zadnji korak izpustili. Grafično si bomo ogledali, kako je videti oblak točk v novem koordinatnem sistemu, določenem z matriko Q .

8.4 Primer

Algoritem preverimo na mešanici treh Gaussovih porazdelitev.

```
using Plots
using Random
m = 100;
Random.seed!(12)
x = [1 .+ randn(m, 1); -3 .+ randn(m, 1); randn(m, 1)];
y = [-2 .+ randn(m, 1); -1 .+ randn(m, 1); 1 .+ randn(m, 1)];
scatter(x[1:100], y[1:100], label="\$\mu = (1, -2)\$")
scatter!(x[101:200], y[101:200], label="\$\mu = (-3, -1)\$")
scatter!(x[201:300], y[201:300], label="\$\mu = (0, 1)\$")
```



Slika 26: Mešanica treh različnih Gaussovih porazdelitev v ravnini

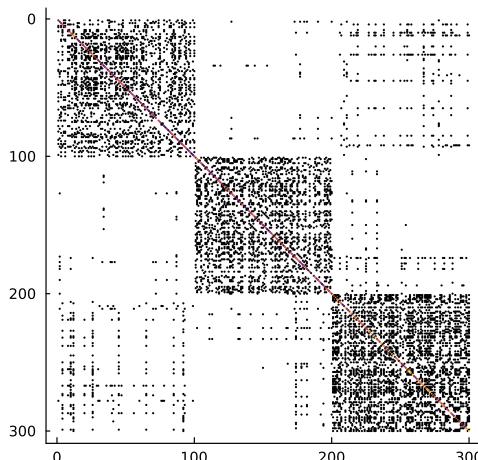
Samostojno delo

Napiši naslednji funkciji:

- `graf_eps(točke, r)`, ki za dane točke določi matriko sosednosti za podobnostni graf z ε okolicami (rešitev je Program 43) in
- `laplace(A)`, ki za dano matriko sosednosti izračuna Laplaceovo matriko grafa (rešitev je Program 44).

Izračunamo graf sosednosti z metodo ε okolic in poiščemo Laplaceovo matriko dobljenega grafa.

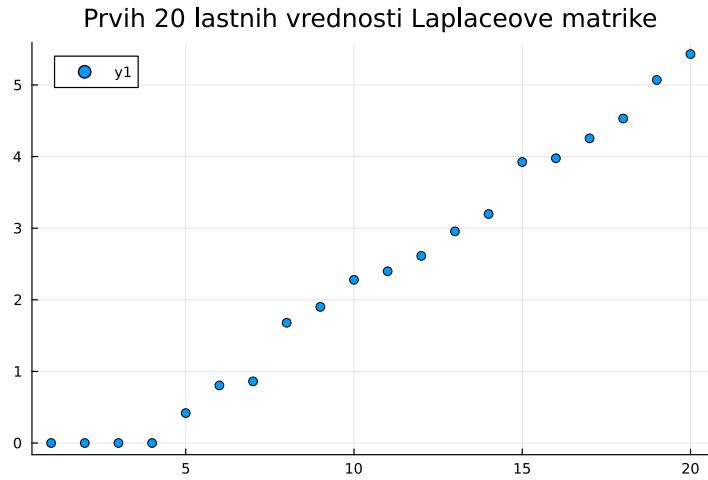
```
using Vaja08
using SparseArrays
točke = hcat(x, y)'
r = 1.0
G = graf_eps(točke, r)
L = laplace(G)
spy(L, legend=false)
```



Slika 27: Neničelni elementi Laplaceove matrike

Izračunamo lastne vrednosti Laplaceove matrike dobljenega grafa:

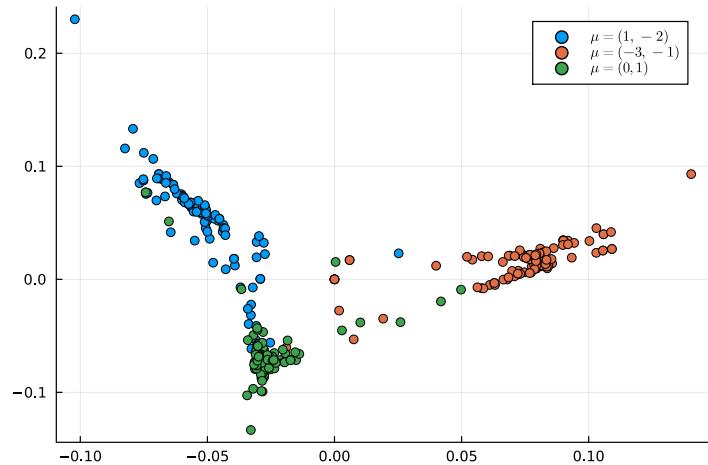
```
using LinearAlgebra
razcep = eigen(Matrix(L))
scatter(razcep.values[1:20], title="Prvih 20 lastnih vrednosti Laplaceove
matrike")
```



Slika 28: Lastne vrednosti Laplaceove matrike

Vidimo, da sta peta in šesta lastna vrednost najmanjši vrednosti, ki sta različni od 0. Komponente lastnih vektorjev za peto in šesto lastno vrednost uporabimo za nove koordinate.

```
x_nov = razcep.vectors[:, 5]
y_nov = razcep.vectors[:, 6]
scatter(x_nov[1:100], y_nov[1:100], label="$\mu=(1, -2)$")
scatter!(x_nov[101:200], y_nov[101:200], label="$\mu=(-3, -1)$")
scatter!(x_nov[201:300], y_nov[201:300], label="$\mu=(0, 1)$")
```



Slika 29: Vložitev točk v nov prostor, določen z lastnima vektorjema Laplaceove matrike. Slika ilustrira, kako lahko s preslikavo v drug prostor gruče postanejo bolj očitne.

Seveda se pri samem algoritmu gručenja ni treba omejiti le na dva lastna vektorja in iskat, katere kombinacije komponent najbolj ločijo gruče. Preprosto izberemo lastne vektorje za k najmanjših neničelnih lastnih vrednosti in algoritem gručenja avtomatsko bolj upošteva dimenzije, v katerih so gruče najbolj razčlenjene.

8.5 Inverzna iteracija

Ker nas zanima le nekaj najmanjših lastnih vrednosti, lahko za njihov izračun uporabimo [inverzno iteracijo](#). Pri tej metodi zgradimo zaporedje približkov z rekurzivno formulo:

$$\mathbf{x}^{(k+1)} = \frac{A^{-1}\mathbf{x}^{(k)}}{\|A^{-1}\mathbf{x}^{(k)}\|}, \quad k = 0, 1, \dots \quad (8.4)$$

Zaporedje približkov $\mathbf{x}^{(k)}$ konvergira k lastnemu vektorju za najmanjo lastno vrednost matrike A za skoraj vse izbire začetnega približka $\mathbf{x}^{(0)}$.

Namesto inverza uporabite LU razcep ali drugo metodo za reševanje sistema linearnih enačb

V inverzni iteraciji moramo večkrat zaporedoma izračunati vrednost:

$$A^{-1}\mathbf{x}^{(k)}. \quad (8.5)$$

Za izračun te vrednosti v resnici ne potrebujemo inverzne matrike A^{-1} . Računanje inverzne matrike je namreč časovno zelo zahtevna operacija, zato se ji, razen v nizkih dimenzijah, če je le mogoče, izognemo. Produkt $\mathbf{x} = A^{-1}\mathbf{b}$ je rešitev sistema linearnih enačb $A\mathbf{x} = \mathbf{b}$ in metode za reševanje sistema so učinkovitejše od računanja inverza A^{-1} in množenja z vektorjem \mathbf{x} .

Inverz A^{-1} matrike A lahko nadomestimo tudi z razcepom matrike A . Če na primer uporabimo LU razcep $A = LU$, lahko $A^{-1}\mathbf{b}$ izračunamo tako, da rešimo sistem $A\mathbf{x} = \mathbf{b}$ oziroma $LU\mathbf{x} = \mathbf{b}$ v dveh korakih:

$$\begin{aligned} L\mathbf{y} &= \mathbf{b} \quad \text{in} \\ U\mathbf{x} &= \mathbf{y}, \end{aligned} \quad (8.6)$$

ki sta časovno toliko zahtevna, kot je množenje z matriko A^{-1} . Programski jezik Julia ima za ta namen posebno metodo `factorize`, ki za različne vrste matrik izračuna najprimernejši razcep. Rezultat metode `factorize` je vrednost posebnega tipa, za katero lahko uporabimo operator `\`, da učinkovito izračunamo rešitev sistema:

```
julia> F = factorize(A)
julia> x = F\b # ekvivalentno A\mathbf{b}, a učinkovitejše
```

Samostojno delo

Napiši funkcijo `inviter(resi, x0)`, ki poišče lastni par za najmanjo lastno vrednost matrike (rešitev je Program 41). Matrika ni podana eksplisitno, ampak je podana le funkcija `resi`, ki reši sistem $A\mathbf{x} = \mathbf{b}$ za dani vektor \mathbf{b} .

8.6 Inverzna iteracija s QR razcepom

Laplaceova matrika je simetrična, zato so lastni vektorji ortogonalni. Lastne vektorje lahko poiščemo tako, da iteracijo izvajamo na več vektorjih hkrati in nato na dobljeni bazi izvedemo ortogonalizacijo s QR razcepom. Tako dobljeno zaporedje lastnih vektorjev konvergira k lastnim vektorjem za po absolutni vrednosti najmanjše lastne vrednosti. Priredimo sedaj funkcijo `inviter`, da za začetni približek spremjme $k \times n$ matriko in izvede inverzno iteracijo s QR razcepom.

Samostojno delo

Napiši funkcijo `inviterqr(resi, X0)`, ki poišče lastne vektorje za prvih nekaj najmanjših lastnih vrednosti (rešitev je Program 42). Število lastnih vektorjev, ki jih metoda poišče, naj bo določeno z dimenzijami začetnega približka $X0$.

Laplaceova matrika grafa je pogosto redka, zato se splača uporabiti eno izmed iterativnih metod. Poleg tega je Laplaceova matrika simetrična in pozitivno semidefinitna, zato za rešitev sistema uporabimo [metodo konjugiranih gradientov](#). Težava je, ker ima Laplaceova matrika grafa tudi lastno vrednost 0, zato metoda konjugiranih gradientov ne konvergira, če jo uporabimo na njej. To lahko rešimo s preprostim premikom Laplaceove matrike za εI .

8.7 Premik

Inverzna iteracija (8.4) konvergira k lastnemu vektorju za najmanjšo lastno vrednost. Lastne vektorje za druge lastne vrednosti poiščemo s premikom. Če ima matrika A lastne vrednosti $\lambda_1, \lambda_2, \dots, \lambda_n$, potem ima matrika:

$$A - \delta I \quad (8.7)$$

lastne vrednosti $\lambda_1 - \delta, \lambda_2 - \delta, \dots, \lambda_n - \delta$. Če izberemo δ dovolj blizu λ_k , lahko poskrbimo, da je $\lambda_k - \delta$ najmanjša lastna vrednost matrike $A - \delta I$. Tako z inverzno iteracijo za matriko $A - \delta I$ poiščemo lastni vektor za poljubno lastno vrednost.

Podobno premaknemo Laplaceovo matriko, da postane strogo pozitivno definitna. Potem lahko za reševanje sistema uporabimo metodo konjugiranih gradientov. Namesto lastnih vrednosti in vektorjev matrike L , isčemo lastne vrednosti in vektorje malce premaknjene matrike $L + \varepsilon I$ z enakimi lastnimi vektorji kot L .

```

A = L + 0.1 * I # premik, da dobimo pozitivno definitno matriko
n = size(L, 1)
# poiščemo prvih 10 lastnih vektorjev
X, lambda = inviterqr(B -> Vaja08.cgmat(A, B), ones(n, 10), 1000)

[ Info: Metoda KG konvergira po 4 korakih.
[ Info: Metoda KG konvergira po 8 korakih.
[ Info: Metoda KG konvergira po 9 korakih.
Inverzna iteracija s QR razcepom se je končala po 205 korakih.
([-0.05773502691368182 -0.005802588523008474 ... -0.005040782182521457
 -0.009367575049040844; -0.057735026915228076 -0.0058025885339489935 ...
 -0.0350631801228755 -0.022053224269263837; ... ; -0.05773502691797562
 -0.005802588530438843 ... 0.005847112228594752 -0.0012243787826357164;
 -0.05773502691890123 -0.005802588531702566 ... 0.009046192382646001
 0.0025458880730825474], [0.10000000000000134, 0.09999999999999998,
 0.5179003965208074, 0.903836883326621, 0.9609470847648324, 1.7793419216657385,
 0.1000000000000002, 2.001222553034165, 2.378682969684288, 2.4982258638596693])

```

Vidimo, da metoda konjugiranih gradientov za naš primer zelo hitro konvergira. Z inverzno iteracijo s QR razcepom smo učinkovito poiskali lastne vektorje Laplaceove matrike za najmanjše lastne vrednosti. Ti lastni vektorji pa izboljšajo proces gručenja.

Velike količine podatkov zahtevajo učinkovite algoritme

V našem primeru je bila količina podatkov majhna. Vendar bi z inverzno iteracijo s QR razcepom in metodo konjugiranih gradientov lahko obdelali tudi bistveno večje količine podatkov, pri katerih bi splošne metode, kot na primer QR iteracija za iskanje lastnih parov ali LU razcep za reševanje sistema, odpovedale.

Z naraščanjem količine podatkov je nujno izbrati učinkovite metode. V praksi se količine podatkov merijo v milijonih in milijardah. Metode s kvadratno ali višjo časovno ali prostorsko zahtevnostjo so pri tako velikih količinah podatkov neuporabne. V tem primeru je mogoče izvesti spektralno gručenje le, če uporabimo učinkovite metode, kot sta *inverzna iteracija s QR razcepom* in *metoda konjugiranih gradientov*.

8.8 Rešitve

```
"""
v, lambda = inviter(resi, x0)

Poišči lastno vrednost `lambda` in lastni vektor `v` za matriko z inverzno
iteracijo.
Argument `resi` je funkcija, ki za dani vektor `b` poišče rešitev sistema `Ax=b`.
Argument `x0` je začetni približek.

# Primer

```jl
A = [3 1 1; 1 1 3; 1 3 4]
F = lu(A)
resi(b) = F \ b
v, lambda = inviter(resi, [1., 1., 1.])
```
"""

function inviter(resi, x0, maxit=100, tol=1e-10)
    # poiščemo po absolutni največji element v x0
    ls, index = findmax(abs, x0)
    ls = x0[index]
    x = x0 / ls # normiramo, da je največji element enak 1
    for i = 1:maxit
        x = resi(x) # poišči rešitev sistema Ay = x
        ln, index = findmax(abs, x)
        ln = x[index]
        x = x / ln # x normiramo, da je največji element enak 1
        if abs(ln - ls) < tol
            println("Inverzna potenčna metoda se je končala po $i korakih.")
            return x, 1 / ln
        end
        ls = ln
    end
    throw("Inverzna potenčna metoda ne konvergira v $maxit korakih.")
end
```

Program 41: Funkcija, ki z inverzno iteracijo poišče lastni par za po absolutni vrednosti najmanjšo lastno vrednost matrike.

```

"""
x, lambda = inviterqr(resi, x0)

Poišči nekaj najmanjših lastnih vrednosti in lastnih vektorjev z inverzno
iteracijo
s QR razcepom.

Argument `resi` je funkcija, ki za dani vektor `b` poišče rešitev sistema `Ax=b`.
Argument `x0` je matrika začetnih približkov. Toliko kot je stolpcev v matriki
`x0`, toliko lastnih parov vrne funkcija.

"""

function inviterqr(resi, x0, maxit=100, tol=1e-10)
    _, m = size(x0)
    x = x0
    ls = Inf * ones(m)
    for i = 1:maxit
        x = resi(x) # poišči rešitev sistema Ay = x
        Q, R = qr(x)
        x = Matrix(Q)
        ln = diag(R) # lastne vrednosti A^(-1) so na diagonali R
        if norm(ln - ls, Inf) < tol
            println("Inverzna iteracija s QR razcepom se je končala po $i korakih.")
            return x, 1 ./ ln
        end
        ls = ln
    end
    throw("Inverzna iteracija s QR razcepom ne konvergira v $maxit korakih.")
end

```

Program 42: Funkcija, ki z inverzno iteracijo in QR razcepom poišče k lastnih parov za k po absolutni vrednosti najmanjših lastnih vrednosti matrike.

```

"""
A = graf_eps(oblak, epsilon)

Poišči podobnostni graf  $\epsilon$  okolic za dani oblak točk.
Argument `oblak` je `k x n` matrika, katere stolpci so koordinate točk v oblaku.
Funkcija vrne matriko sosednosti A za podobnostni graf.

"""

function graf_eps(oblak, epsilon)
    _, n = size(oblak)
    A = spzeros(n, n)
    for i = 1:n
        for j = i+1:n
            if norm(oblak[:, i] - oblak[:, j]) < epsilon
                A[i, j] = 1
                A[j, i] = 1
            end
        end
    end
    return A
end

```

Program 43: Funkcija, ki ustvari matriko sosednosti za graf podobnosti z ϵ okolicami za dane podatke.

```

"""
L = laplace(A)

Poišči Laplaceovo matriko za dani graf, podan z matriko sosednosti `A`.
Laplaceova matrika grafa ima na diagonali stopnje točk, izven diagonale
pa vrednosti -1 ali 0, odvisno, ali sta indeksa povezana v grafu ali ne.
"""

laplace(A) = spdiags(vec(sum(A, dims=2))) - A

```

Program 44: Funkcija, ki izračuna Laplaceovo matriko grafa.

8.9 Testi

```

@testset "Inverzna iteracija" begin
    Q = [2 2 1; 1 -2 2; -2 1 2]
    A = Q * diagm([2.0, 3.0, 4.0]) / Q
    F = factorize(A)
    v, lambda = inviter(b -> F \ b, [1, 1, 1])
    @test isapprox(lambda, 2)
    cosinus = dot(v, Q[:, 1]) / norm(v) / norm(Q[:, 1])
    @test isapprox(abs(cosinus), 1)
end

```

Program 45: Test za inverzno iteracijo

```

@testset "Inverzna iteracija QR" begin
    Q = [2 2 1; 1 -2 2; -2 1 2]
    A = Q * diagm([2.0, 3.0, 4.0]) / Q
    F = factorize(A)
    v, lambda = inviterqr(b -> F \ b, [1 1; 1 1; 1 1])
    @test isapprox(lambda, [2, 3])
    cosinus = dot(v[:, 1], Q[:, 1]) / norm(v[:, 1]) / norm(Q[:, 1])
    @test isapprox(abs(cosinus), 1)
end

```

Program 46: Test za inverzno iteracijo s QR razcepom

```

@testset "Graf sosednosti" begin
    oblak = [1 2 3; 1 2 3]
    A = graf_eps(oblak, 2)
    @test isapprox(A, [0 1 0; 1 0 1; 0 1 0])
end

```

Program 47: Test za matriko sosednosti z ε okolicami

9 Konvergenčna območja sistemov nelinearnih enačb

Za razliko od sistemov linearnih enačb, ki imajo preproste množice rešitev, so množice rešitev za sisteme nelinearnih enačb zapletene in nepredvidljive. Zato tudi reševanje sistemov nelinearnih enačb z numeričnimi metodami ni tako preprosto kot za sisteme linearnih enačb. Numerične metode lahko konvergenco k določeni rešitvi zagotovijo le ob dodatnih predpostavkah, ki jih je težko vnaprej izpolniti. V tej vaji si bomo na preprostem primeru ogledali, kako se obnaša Newtonova metoda za različne začetne približke.

9.1 Naloga

- Implementiraj Newtonovo metodo za reševanje sistemov nelinearnih enačb.
- Poišči rešitev dveh nelinearnih enačb z dvema neznankama:

$$\begin{aligned}x^3 - 3xy^2 &= 1, \\ 3x^2y - y^3 &= 0.\end{aligned}\tag{9.1}$$

- Sistem nelinearnih enačb ima navadno več rešitev. Grafično predstavi, h kateri rešitvi konvergira Newtonova metoda v odvisnosti od začetnega približka. Začetne približke izberi na pravokotni mreži. Vozliščem v mreži priredi različne barve, glede na to, h kateri rešitvi konvergira Newtonova metoda. Cel postopek zapiši v funkcijo `konvergencno_obmocje`.

9.2 Newtonova metoda za sisteme enačb

Iščemo rešitev sistem n nelinearnih enačb z n neznankami:

$$\begin{aligned}f_1(x_1, x_2, \dots, x_n) &= 0, \\ f_2(x_1, x_2, \dots, x_n) &= 0, \\ &\vdots \\ f_n(x_1, x_2, \dots, x_n) &= 0,\end{aligned}\tag{9.2}$$

kjer so f_1, f_2, \dots, f_n nelinearne funkcije več spremenljivk. Sistem (9.2) zapišemo v vektorski obliki:

$$\mathbf{F}(\mathbf{x}) = \mathbf{0},\tag{9.3}$$

kjer sta $\mathbf{0} = [0, 0, \dots, 0]^T$ in $\mathbf{x} = [x_1, x_2, \dots, x_n]^T \in \mathbb{R}^n$ n -dimenzionalna vektorja, $\mathbf{F} : \mathbb{R}^n \rightarrow \mathbb{R}^n$ pa je vektorska funkcija z vektorskim argumentom. Komponente vektorske funkcije $\mathbf{F}(\mathbf{x})$ so leve strani nelinearnih enačb (9.2):

$$\mathbf{F}(\mathbf{x}) = \begin{pmatrix} f_1(x_1, x_2, \dots, x_n) \\ f_2(x_1, x_2, \dots, x_n) \\ \vdots \\ f_n(x_1, x_2, \dots, x_n) \end{pmatrix}.\tag{9.4}$$

Denimo, da je $\mathbf{x}^{(k)}$ približek za rešitev enačbe (9.3). Funkcijo \mathbf{F} lahko, podobno kot funkcijo ene spremenljivke, v točki $\mathbf{x}^{(k)}$ aproksimiramo z linearno funkcijo:

$$\mathbf{F}(\mathbf{x}) = \mathbf{F}(\mathbf{x}^{(k)}) + J_{\mathbf{F}}(\mathbf{x}^{(k)})(\mathbf{x} - \mathbf{x}^{(k)}) + \mathcal{O}\left((\mathbf{x} - \mathbf{x}^{(k)})^2\right),\tag{9.5}$$

kjer je $J_{\mathbf{F}}$ Jacobijeva matrika parcialnih odvodov komponent f_i po koordinatah x_j :

$$J_{\mathbf{F}}(\mathbf{x}) = \begin{pmatrix} \frac{\partial f_1}{\partial x_1}(\mathbf{x}) & \frac{\partial f_1}{\partial x_2}(\mathbf{x}) & \dots & \frac{\partial f_1}{\partial x_n}(\mathbf{x}) \\ \frac{\partial f_2}{\partial x_1}(\mathbf{x}) & \frac{\partial f_2}{\partial x_2}(\mathbf{x}) & \dots & \frac{\partial f_2}{\partial x_n}(\mathbf{x}) \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1}(\mathbf{x}) & \frac{\partial f_n}{\partial x_2}(\mathbf{x}) & \dots & \frac{\partial f_n}{\partial x_n}(\mathbf{x}) \end{pmatrix}. \quad (9.6)$$

Naslednji približek $\mathbf{x}^{(k+1)}$ v Newtonovi iteraciji dobimo kot rešitev sistema linearnih enačb:

$$\mathbf{F}(\mathbf{x}^{(k)}) + J_{\mathbf{F}}(\mathbf{x}^{(k)})(\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}) = \mathbf{0}. \quad (9.7)$$

Formulo za naslednji približek $\mathbf{x}^{(k+1)}$ formalno zapišemo kot:

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - J_{\mathbf{F}}(\mathbf{x}^{(k)})^{-1} \mathbf{F}(\mathbf{x}^{(k)}), \quad (9.8)$$

pri čemer formule ne smemo jemati dobesedno, saj inverzne matrike $J_{\mathbf{F}}(\mathbf{x}^{(k)})^{-1}$ dejansko ne izračunamo. Izraz $J_{\mathbf{F}}(\mathbf{x}^{(k)})^{-1} \mathbf{F}(\mathbf{x}^{(k)})$ poiščemo tako, da rešimo sistem $J_{\mathbf{F}}(\mathbf{x}^{(k)}) \mathbf{x} = \mathbf{F}(\mathbf{x}^{(k)})$ (npr. z LU razcepom ali s kakšno drugo metodo za reševanje sistemov linearnih enačb).

Samostojno delo

Napiši funkcijo `newton(f, jf, x0)`, ki poišče rešitev sistema nelinearnih enačb z Newtonovo metodo (Program 48).

Poglejmo, kako uporabimo Newtonovo metodo za enačbe (9.1). Spremenljivke x, y postavimo v vektor $\mathbf{x} = [x, y]^T$ in za lažje pisanje programa vpeljemo komponente $x_1 = x$ in $x_2 = y$. Sistem enačb (9.1) preuredimo tako, da je na desni strani 0:

$$\begin{aligned} x_1^3 - 3x_1x_2^2 - 1 &= 0, \\ 3x_1^2x_2 - x_2^3 &= 0. \end{aligned} \quad (9.9)$$

Funkcija levih strani $\mathbf{F}(\mathbf{x})$ je enaka:

$$\mathbf{F}(\mathbf{x}) = \begin{pmatrix} x_1^3 - 3x_1x_2^2 - 1 \\ 3x_1^2x_2 - x_2^3 \end{pmatrix}, \quad (9.10)$$

Jacobijeva matrika $J_{\mathbf{F}}(\mathbf{x})$ pa:

$$J_{\mathbf{F}}(\mathbf{x}) = \begin{pmatrix} 3x_1^2 - 3x_2^2 & -6x_1x_2 \\ 6x_1x_2 & 3x_1^2 - 3x_2^2 \end{pmatrix}. \quad (9.11)$$

```
f(x) = [x[1]^3 - 3x[1] * x[2]^2 - 1, 3x[1]^2 * x[2] - x[2]^3]
function jf(x)
    a = 3x[1]^2 - 3x[2]^2
    b = 6x[1] * x[2]
    jf = [a -b; b a]
end
```

Avtomatsko odvajanje

V našem primeru smo Jacobijevo matriko izračunali na roke, saj je bila funkcija preprosta. Če je funkcija \mathbf{F} kompleksnejša ali je ne poznamo vnaprej, lahko Jacobijevo matriko odvodov učinkovito izračunamo z [avtomatskim odvajanjem](#). V Julii uporabimo funkcijo `jacobian` iz paketa [Forward-Diff](#)[13].

Sistem (9.1) izhaja iz kompleksne enačbe $z^3 = 1$ in ima tako 3 rešitve:

$$\begin{aligned}x_1 &= 1, \quad y_1 = 0 \quad (z_1 = 1), \\x_2 &= -\frac{1}{2}, \quad y_2 = \frac{\sqrt{3}}{2} \quad \left(z_2 = -\frac{1}{2} + \frac{\sqrt{3}}{2}i \right) \text{ in} \\x_3 &= -\frac{1}{2}, \quad y_3 = -\frac{\sqrt{3}}{2} \quad \left(z_3 = -\frac{1}{2} - \frac{\sqrt{3}}{2}i \right).\end{aligned}\tag{9.12}$$

Za različne začetne približke Newtonova metoda konvergira k različnim rešitvam:

```
julia> x1, it1 = newton(f, jf, [2, 0])
([1.0, 0.0], 7)

julia> x2, it2 = newton(f, jf, [-1, 1.0])
([-0.4999999999999994, 0.8660254037844386], 6)

julia> x2, it2 = newton(f, jf, [-1, 1.0])
([-0.4999999999999994, 0.8660254037844386], 6)
```

9.3 Konvergenčno območje

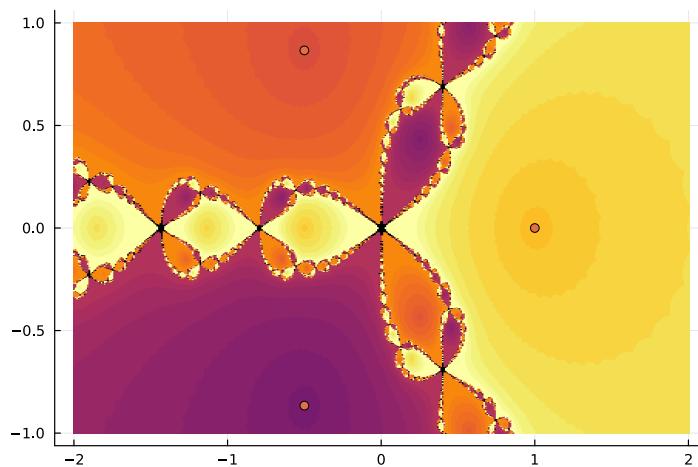
Newtonova metoda je občutljiva glede izbire začetnega približka. Če je začetni približek dovolj blizu neke rešitve, Newtonova metoda konvergira k tisti rešitvi. Če pa je približek med ničlami, postane obnašanje Newtonove metode precej nepredvidljivo.

Samostojno delo

- Definiraj podatkovni tip `Box2d`, ki opiše pravokotnik v ravnini s stranicami vzporednimi koordinatnim osem (Program 49) in
- Napiši funkcijo `konvergenca(območje::Box2d, metoda, nx, ny)`, ki za dano območje izračuna dele območja, na katerem `metoda` konvergira k določeni ničli. Argument `metoda` naj bo funkcija oblike `x, it = metoda(x0)`, ki sprejme začetni približek in vrne rešitev ter število iteracij (Program 50).

Program konvergenca uporabimo na našem primeru (9.1):

```
using Plots
maxit = 20
območje = Box2d(Interval(-2, 2), Interval(-1, 1))
metoda(x0) = newton(f, jf, x0; atol=1e-4, maxit=maxit)
x, y, Z, ničle, koraki = konvergenca(območje, metoda, 800, 400; atol=1e-3)
heatmap(x, y, Z + 0.8 * min.(koraki / 10, 1), legend=false)
scatter!(Tuple.(ničle), label="rešitve")
```



Slika 30: Newtonova metoda konvergira k različnim rešitvam odvisno od začetnega približka.

Kaj smo se naučili?

- Sisteme nelinearnih enačb lahko rešimo z Newtonovo metodo.
- Sistemi nelinearnih enačb imajo navadno več rešitev.
- Konvergenca Newtonove metode je odvisna od začetnega približka.
- Za izbrani začetni približek je težko predvideti, h kateri izmed rešitev bo Newtonova metoda konvergirala.

9.4 Rešitve

```
using LinearAlgebra
"""
x, it = newton(f, jf, x0)

Poišči rešitev sistema nelinearnih enačb `f(x) = 0` z Newtonovo metodo.
Argument `jf` je funkcija, ki vrne Jacobijevu matriko funkcije `f`.
Argument `x0` je začetni približek za Newtonovo metodo.
"""

function newton(f, jf, x0; maxit=100, atol=1e-8)
    for i = 1:maxit
        x = x0 - jf(x0) \ f(x0)
        if norm(x - x0, Inf) < atol
            return x, i
        end
        x0 = x
    end
    throw("Metoda ne konvergira po $maxit korakih!")
end
```

Program 48: Newtonova metoda za reševanje sistemov nelinearnih enačb

```

"Podatkovna struktura za interval"
struct Interval
    min
    max
end

"Ali interval `I` vsebuje točko `x`?"
vsebuje(x, I::Interval) = x >= I.min && x <= I.max

"""

Podatkovna struktura za pravokotnik, vzporeden s koordinatnimi osmi (škatla).
Pravokotnik je podan kot produkt dveh intervalov za spremenljivki `x` in `y`.
"""

struct Box2d
    int_x
    int_y
end

"Ali škatla `b` vsebuje dano točko `x`?"
vsebuje(x, b::Box2d) = vsebuje(x[1], b.int_x) && vsebuje(x[2], b.int_y)

"""

x = diskretiziraj(I, n)

Razdeli interval `I` na `n` enakih podintervalov. Vrni seznam krajišč
podintervalov.
"""

diskretiziraj(I::Interval, n) = range(I.min, I.max, n)

"""

x = diskretiziraj(b, m, n)

Razdeli škatlo `b` na manjše škatle. Vrni seznama krajišč
podintervalov v smereh `x` in `y`.
"""

diskretiziraj(b::Box2d, m, n) =
    diskretiziraj(b.int_x, m), diskretiziraj(b.int_y, n))

```

Program 49: Pomožne funkcije za delo s pravokotnimi območji

```
"""
    x, y, Z, ničle, koraki = konvergenca(območje, metoda, n=50, m=50; maxit=50,
tol=1e-3)
```

Izračunaj, h katerim vrednostim konvergira metoda `metoda`, če uporabimo različne začetne približke na pravokotniku `[a, b]x[c, d]`, podanim z argumentom `območje`.

Funkcija vrne:

- seznam krajišč podintervalov v x smeri,
- seznam krajišč podintervalov v y smeri,
- matriko z indeksi ničle, h kateri metoda konvergira,
- seznam ničel na izbranem območju,
- matriko s številom korakov, ki jih metoda potrebuje, da najde ničlo.

Primer

Konvergenčno območje za Newtonovo metodo za reševanje kompleksne enačbe `` $z^3=(x + i y)^3 = 1$ ``

```
```jl
F((x, y)) = [x^3-3x*y^2-1; 3x^2*y-y^3];
JF((x, y)) = [3x^2-3y^2 -6x*y; 6x*y 3x^2-3y^2];
metoda(x0) = newton(F, JF, x0; maxit=10; tol=1e-3);
območje = Box2d(Interval(-2, 2), (-1, 1))
x, y, Z, ničle, koraki = konvergenca(območje, metoda; n=5, m=5)
```
"""

function konvergenca(območje::Box2d, metoda, m=50, n=50; atol=1e-3)
    Z = zeros(m, n)
    koraki = zeros(m, n)
    x, y = diskretiziraj(območje, n, m)
    ničle = []
    for i = 1:n, j = 1:m
        z = [x[i], y[j]]
        it = 0
        try
            z, it = metoda(z)
        catch
            continue
        end
        k = findfirst([norm(z - z0, Inf) < 2atol for z0 in ničle])
        if isnuthing(k)
            if vsebuje(z, območje)
                push!(ničle, z)
                k = length(ničle)
            else
                continue
            end
        end
        Z[j, i] = k # vrednost elementa je enaka indeksu ničle
        koraki[j, i] = it # število korakov metode
    end
    return x, y, Z, ničle, koraki
end
```

Program 50: Funkcija, ki razišče konvergenco izbrane metode na danem pravokotniku.

10 Nelinearne enačbe v geometriji

Ko obravnavamo geometrijske objekte, ki so ukrivljeni (na primer krožnice, krivulje, ukrivljene ploskve), probleme pogosto prevedemo na reševanje sistema nelinearnih enačb. Ogledali si bomo dva primera, kjer problem rešimo na ta način: iskanje samopresečišča krivulje in minimalne razdalje med dvema krivuljama.

10.1 Naloga

- Napiši funkcijo, ki poišče eno od samopresečišč [Lissajousove krivulje](#):

$$(x(t), y(t)) = (a \sin(nt), b \cos(mt)) \quad (10.1)$$

za parametre $a = b = 1$, $n = 3$ in $m = 2$.

- Poišci točki na parametrično podanih krivuljah:

$$\begin{aligned} (x_1(t), y_1(t)) &= \left(2 \cos(t) + \frac{1}{3}, \quad \sin(t) + \frac{1}{4} \right), \\ (x_2(s), y_2(s)) &= \left(\frac{1}{3} \cos(s) - \frac{1}{2} \sin(s), \quad \frac{1}{3} \cos(s) + \frac{1}{2} \sin(t) \right), \end{aligned} \quad (10.2)$$

ki sta najbližje.

- Zapiši razdaljo med točko na prvi krivulji in točko na drugi krivulji kot funkcijo $d(t, s)$ parametrov t in s .
- Z [gradientnim spustom](#) poišči minimum funkcije $d(t, s)$ oziroma $d^2(t, s)$.
- Minimum funkcije $d^2(t, s)$ poišči še z Newtonovo metodo kot rešitev sistema nelinearnih enačb:

$$\nabla d^2(t, s) = \mathbf{0}. \quad (10.3)$$

- Grafično predstavi zaporedja približkov za gradientno in Newtonovo metodo.
- Primerjaj konvergenčna območja za gradientno in Newtonovo metodo (glej Poglavlje 9).

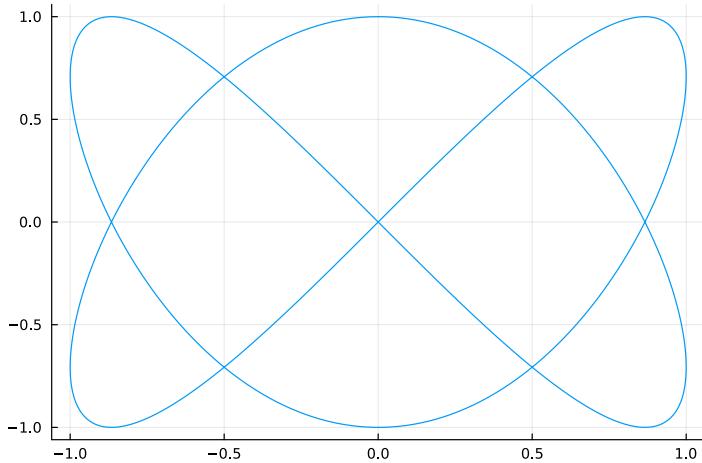
10.2 Presečišča parametrično podanih krivulj

Poишčimo samopresečišča [Lissajousove krivulje](#):

$$\begin{aligned} x(t) &= a \sin(nt), \\ y(t) &= b \cos(mt) \end{aligned} \quad (10.4)$$

za parametre $a = b = 1$, $n = 2$ in $m = 3$. Za lažjo predstavo najprej narišemo krivuljo.

```
using Plots
l(t) = [sin(2t), cos(3t)]
t = range(0, 2pi, 500)
# Funkcija plot drugače interpretira vektor vektorjev kot vektor parov (Tuple).
# Zato vektorje koordinat spremenimo v pare koordinat (Tuple).
plot(Tuple.(l.(t)), label=nothing)
```



Slika 31: Lissajousova krivulja za $a = b = 1$, $n = 2$ in $m = 3$

Iščemo različni vrednosti parametra t_1 in t_2 , za katera velja

$$\begin{aligned} x(t_1) &= x(t_2), \\ y(t_1) &= y(t_2), \quad t_1 \neq t_2. \end{aligned} \tag{10.5}$$

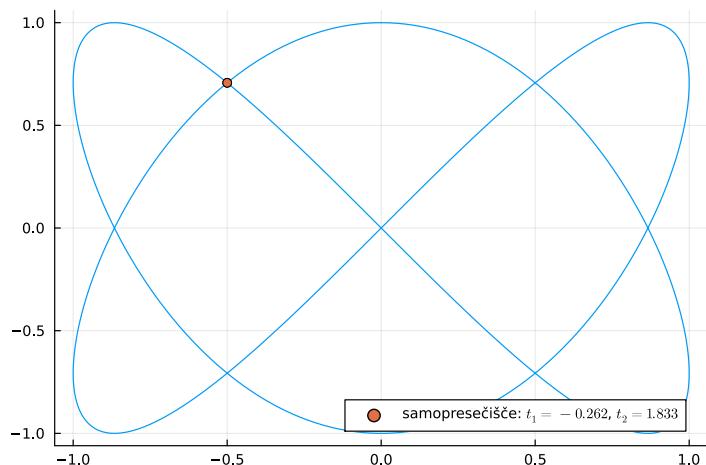
Dobili smo sistem dveh nelinearnih enačb z dvema neznankama. Rešitve sistema (10.5) poiščemo z Newtonovo metodo, ki smo jo spoznali v poglavju 9. Newtonova metoda zahteva, da sistem enačb prevedemo v vektorsko enačbo $\mathbf{F}(\mathbf{t}) = \mathbf{0}$, kjer je $\mathbf{t} = [t_1, t_2]^T$. Funkcija, katere ničlo iščemo, je

$$\mathbf{F}(\mathbf{t}) = \mathbf{F}\left(\begin{bmatrix} t_1 \\ t_2 \end{bmatrix}\right) = \begin{bmatrix} x(t_1) - x(t_2) \\ y(t_1) - y(t_2) \end{bmatrix}, \tag{10.6}$$

njena Jacobijeva matrika pa

$$J_{\mathbf{F}}\left(\begin{bmatrix} t_1 \\ t_2 \end{bmatrix}\right) = \begin{pmatrix} \dot{x}(t_1) & -\dot{x}(t_2) \\ \dot{y}(t_1) & -\dot{y}(t_2) \end{pmatrix}. \tag{10.7}$$

```
using Vaja09: newton
using Printf
f(tt) = l(tt[1]) - l(tt[2])
dl(t) = [2cos(2t), -3sin(3t)]
df(tt) = hcat(dl(tt[1]), -dl(tt[2]))
tt, it = newton(f, df, [0.0, pi / 2])
scatter!(Tuple.(l.(tt)),
    label=@sprintf "samopresečišče: \$t_1=% .3f, \$t_2=% .3f" tt...)
```



Slika 32: Krivulja doseže izbrano samopresečišče pri dveh različnih vrednostih parametra t .

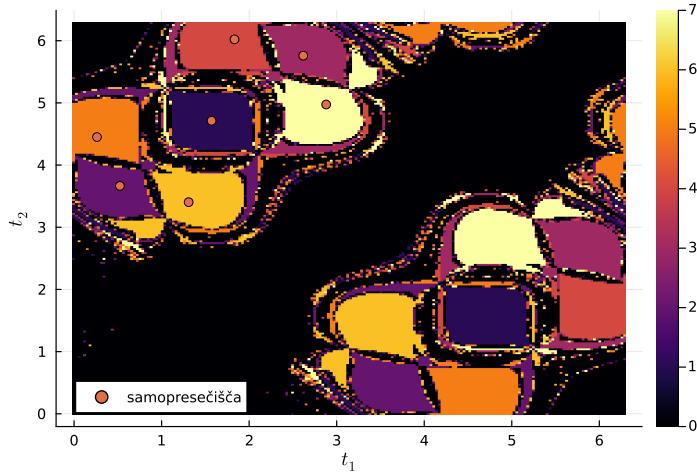
Samostojno delo

Napiši funkcijo `samopres(k, dk, tt0)`, ki poišče eno od samopresečišč krivulje z Newtonovo metodo z danim začetnim približkom (rešitev Program 51).

Uporabimo Program 50 in poiščemo vsa samopresečišča ter določimo konvergenčna območja. Ker sta funkciji sin in cos, ki nastopata v definiciji krivulje, periodični s periodo 2π , vrednosti parametrov t in s računamo po modulu 2π .

```
using Vaja10: samopres
mod2pi(x) = rem(x, 2pi)
""" Poišči samopresečišče Lissajousove krivulje. Upoštevaj periodičnost."""
function splissajous(tt0)
    tt, it = samopres(l, dl, tt0)
    tt = mod2pi.(tt)
    if abs(tt[1] - tt[2]) < 1e-12
        throw("Isti vrednosti parametra ne pomenita samopresečišča.")
    end
    return sort(tt), it
end

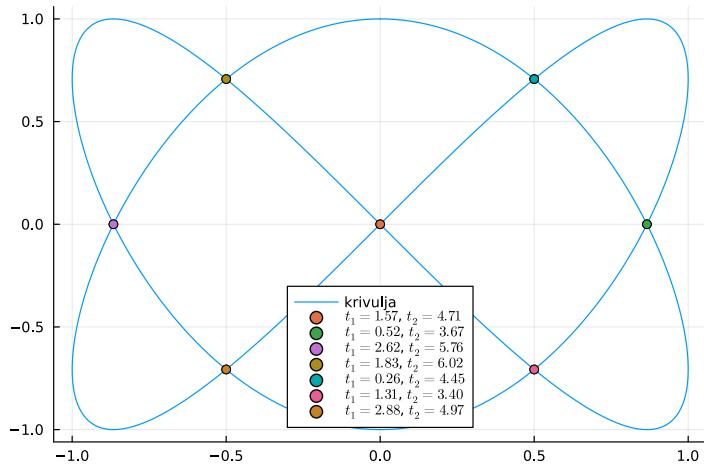
using Vaja09: konvergenca, Box2d, Interval
x, y, Z, ničle, koraki = konvergenca(Box2d(Interval(0, 2pi), Interval(0, 2pi)),
                                         splissajous, 200, 200)
heatmap(x, y, Z, xlabel="\$t_1\$", ylabel="\$t_2\$")
scatter!(Tuple.(ničle), label="samopresečišča", legend=:bottomleft)
```



Slika 33: Območje konvergencije za samopresečišča Lissajousove krivulje

Narišimo sedaj še krivuljo in na njej označimo vse samopresečišča.

```
p = plot(Tuple.(l.(t)), label="krivulja", legend=:bottom)
for tt in ničle
    scatter!(p, Tuple.(l.(tt)), label=@sprintf "\$t_1=% .2f\$, \$t_2=% .2f\$" tt...)
end
display(p)
```



10.3 Minimalna razdalja med dvema krivuljama

Naj bosta K_1 in K_2 parametrično podani krivulji:

$$\begin{aligned} K_1 : \mathbf{k}_1(t) &= (x_1(t), y_1(t)); \quad t \in \mathbb{R}, \\ K_2 : \mathbf{k}_2(s) &= (x_2(s), y_2(s)); \quad s \in \mathbb{R}. \end{aligned} \tag{10.8}$$

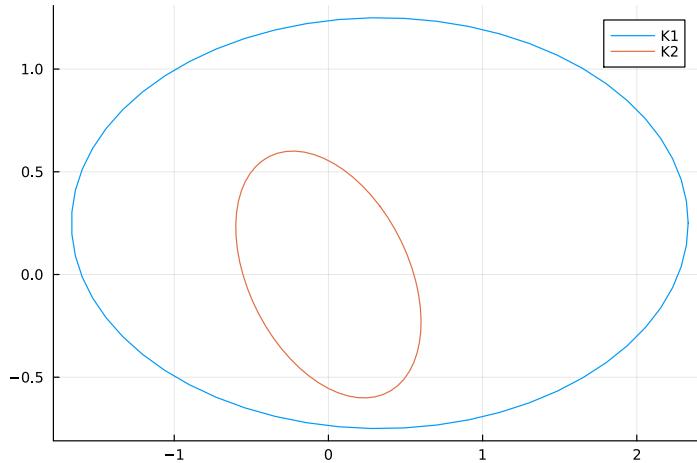
Narišimo obe krivulji iz naloge:

$$\begin{aligned} \mathbf{k}_1(t) &= \left(2 \cos(t) + \frac{1}{3}, \quad \sin(t) + \frac{1}{4} \right), \\ \mathbf{k}_2(t) &= \left(\frac{1}{3} \cos(s) - \frac{1}{2} \sin(s), \quad \frac{1}{3} \cos(s) + \frac{1}{2} \sin(s) \right). \end{aligned} \tag{10.9}$$

```

k1(t) = [2 * cos(t) + 1 / 3, sin(t) + 0.25]
k2(s) = [cos(s) / 3 - sin(s) / 2, cos(s) / 3 + sin(s) / 2]
t = range(0, 2pi, 60);
plot(Tuple.(k1.(t)), label="K1")
plot!(Tuple.(k2.(t)), label="K2")

```



Slika 35: Krivulji v ravnini

Iščemo minimalno razdaljo med krivuljama K_1 in K_2 . Minimalna razdalja je najmanjšo razdalja med točko na eni krivulji in točko na drugi krivulji:

$$d_m(K_1, K_2) = \min_{T_1 \in K_1, T_2 \in K_2} d(T_1, T_2). \quad (10.10)$$

Funkcija d_m ni prava razdalja v smislu metričnih prostorov.

Hausdorffova razdalja

Alternativna definicije razdalje med dvema množicama je *Hausdorffova razdalja*. Hausdorffova razdalja pove, koliko je lahko točka na eni krivulji največ oddaljena od druge krivulje in je definirana kot:

$$d_h(K_1, K_2) = \max \left(\max_{T_1 \in K_1} \min_{T_2 \in K_2} d(T_1, T_2), \max_{T_2 \in K_2} \min_{T_1 \in K_1} d(T_1, T_2) \right). \quad (10.11)$$

Če sta množici blizu v Hausdorffovi razdalji, je vsaka točka ene množice blizu druge množici. Minimalna razdalja med dvema krivuljama je vedno končna, medtem ko je Hausdorffova razdalja lahko tudi neskončna (na primer, če je ena krivulja omejena, druga pa neomejena).

Iščemo točko $\mathbf{k}_1(t)$ na krivulji K_1 in točko $\mathbf{k}_2(s)$ na krivulji K_2 , ki sta najbližji med vsemi pari točk. Iščemo vrednosti parametrov t in s , pri katerih funkcija razdalje

$$d(t, s) = \sqrt{(x_1(t) - x_2(s))^2 + (y_1(t) - y_2(s))^2} \quad (10.12)$$

doseže minimum. Ker je koren naraščajoča funkcija, imata d in d^2 minimum v isti točki. Zato namesto minimuma funkcije d poiščemo minimum funkcije

$$D(t, s) = d^2(t, s) = (x_1(t) - x_2(s))^2 + (y_1(t) - y_2(s))^2. \quad (10.13)$$

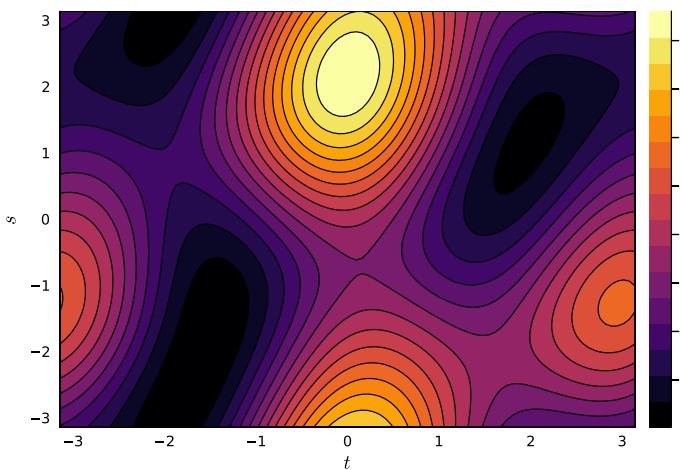
Samostojno delo

Napiši funkcijo razdalja2(k1, k2), ki za dani krivulji k1 in k2 vrne funkcijo kvadrata razdalje $D(t_1, t_2)$. Vrnjena funkcija naj sprejme dva argumenta t1 in t2 in vrne kvadrat razdalje med točkama k1(t1) in k2(t2) (rešitev je Program 52).

Funkcijo $D(t, s)$ za krivulji (10.9) grafično predstavimo z nivojnimi in barvami na kvadratu $[-\pi, \pi]^2$.

```
using Vaja10: razdalja2
d2 = razdalja2(k1, k2)

t = range(-pi, pi, 100)
s = t
contourf(t, s, d2, xlabel="\$t\$", ylabel="\$s\$")
```



Slika 36: Razdalja med točkama na krivuljah K_1 in K_2 v odvisnosti od parametrov na krivulji

10.3.1 Gradientni spust

Metoda gradientnega spusta je sila enostavna. Predstavljamо si, da je gosta megle in da smo na pobočju gore. Želimo čim prej priti v dno doline. Na vsakem koraku izberemo smer, v kateri je pobočje najstrmejše in se spustimo v tej smeri. Na ta način najhitreje izgubljamo višino. Vendar ni nujno, da bomo prišli v dno doline, saj lahko prej pristanemo v kakšni kotanji ali vrtači na pobočju gore. V vsakem primeru bomo prišli nekam na dno, od koder bo šlo le še navzgor.

V jeziku funkcij iščemo minimum funkcije več spremenljivk f . Na vsakem koraku izberemo smer, v kateri funkcija najhitreje pada, in se premaknemo za določen korak v tej smeri. To je ravno v nasprotni smeri gradijenta funkcije. Če koraki niso preveliki, bomo prej ali slej pristali v lokalnemu minimumu funkcije f . Računamo naslednje zaporedje približkov:

$$\mathbf{x}^{(n+1)} = \mathbf{x}^{(n)} - h_n \nabla f(\mathbf{x}^{(n)}), \quad (10.14)$$

kjer je $\nabla f(\mathbf{x}^{(n)})$ vrednost gradijenta v točki $\mathbf{x}^{(n)}$, h_n pa je parameter, ki poskrbi, da zaporedje približkov ne skače preveč po domeni in se lahko na vsakem koraku spremeni.

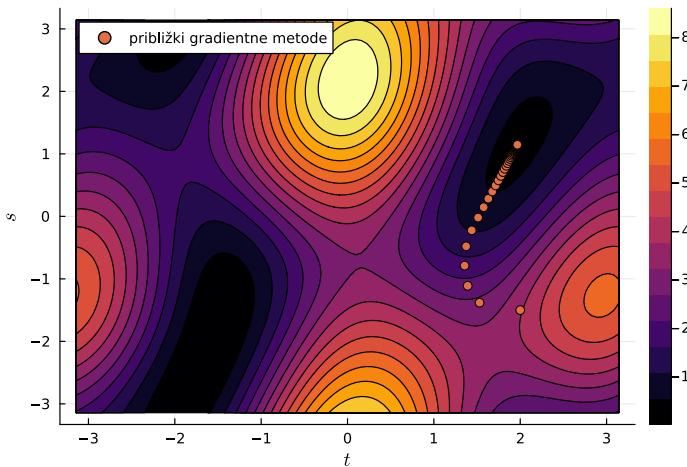
Samostojno delo

Napiši funkcijo spust(gradf, x0, h), ki poišče lokalni minimum funkcije z metodo gradientnega spusta (rešitev Program 53).

Gradient funkcije D lahko izračunamo na roke, vendar je to zamudno in se pri tem lahko hitro zmotimo. Raje uporabimo knjižnico za [avtomatsko odvajanje ForwardDiff.jl](#)[13], ki učinkovito izračuna vrednosti parcialnih odvodov funkcije v posameznih točkah. Knjižnica ForwardDiff zna odvajati le funkcije vektorske spremenljivke, zato funkcijo dveh spremenljivk $d2(t, s)$ sprememimo v funkcijo vektorske spremenljivke $ts \rightarrow d2(ts\dots)$. Operator \dots elemente vektorja razporedi kot argumente funkcije.

```
using ForwardDiff
gradd2(ts) = ForwardDiff.gradient(ts → d2(ts\dots), ts)

"Izračunaj zaporedje približkov podano z rekurzivno funkcijo `f`."
function približki(f, x0, n)
    p = [x0]
    x = x0
    for _ = 1:n
        x = f(x)
        push!(p, x)
    end
    return p
end
korak_grad(x0) = x0 - 0.2 * gradd2(x0)
pribl = približki(korak_grad, [2.0, -1.5], 40)
scatter!(Tuple.(pribl), label="približki gradientne metode")
```



Slika 37: Zaporedje približkov gradientnega spusta

Slike je razvidno, da gradientni spust konvergira k lokalnemu minimumu, vendar postane konvergenca počasna, ko se približamo minimumu. Konvergenco lahko pohitrimo s primerno izbiro parametra h_n , na primer z metodo [minimiziranja v dani smeri](#).

10.3.2 Newtonova metoda

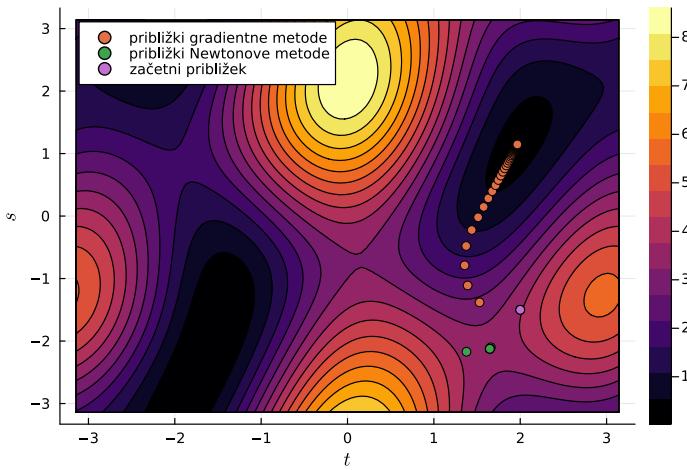
Fermatov izrek pravi, da je v lokalnem ekstremu vrednost odvoda vedno enaka 0. Isti izrek velja tudi za funkcije več spremenljivk, le da je v tem primeru gradient funkcije enak 0.

Ta Fermatov izrek morda ni tako razvpit kot njegov [zadnji](#), je pa za nas uporabnejši. Potreben pogoj za nastop lokalnega ekstrema je tako vektorska enačba

$$\nabla D(t, s) = \begin{pmatrix} \frac{\partial D}{\partial t}(t, s) \\ \frac{\partial D}{\partial s}(t, s) \end{pmatrix} = \mathbf{0}. \quad (10.15)$$

Rešitev enačbe (10.15) poiščemo z Newtonovo metodo.

```
jacd2(x0) = ForwardDiff.jacobian(gradd2, x0)
korak_newton(x0) = x0 - jacd2(x0) \ gradd2(x0)
približki_n = približki(korak_newton, [2.0, -1.5], 10)
scatter!(Tuple.(približki_n), label="približki Newtonove metode")
scatter!([(2.0, -1.5)], label="začetni približek")
```



Slika 38: Zaporedje približkov gradientnega spusta in Newtonove metode z istim začetnim približkom

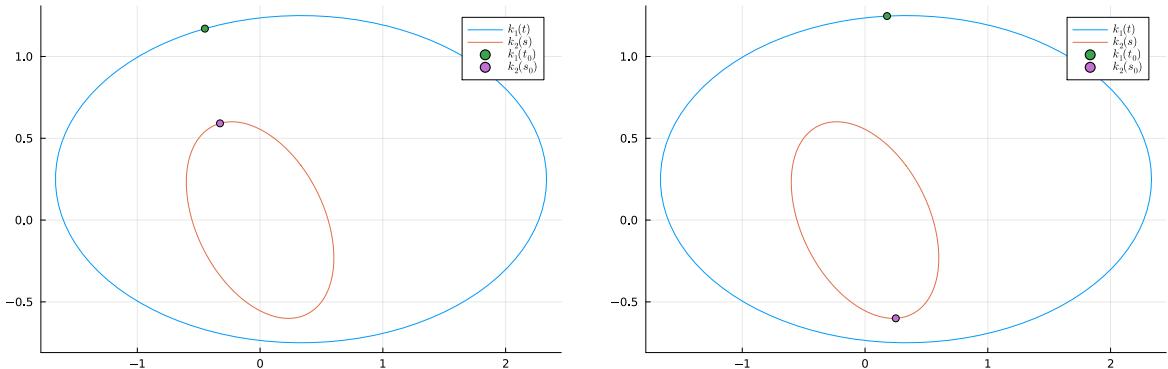
Za razliko od gradientnega spusta, Newtonova metoda ne konvergira nujno k lokalnemu minimumu, ampak k eni od stacionarnih točk funkcije D , med katerimi so tudi sedla in maksimumi. Zato je Newtonova metoda precej občutljivejša na izbiro začetnega približka kot gradientni spust.

Poglejmo si točki na krivuljah, ki ustrezata parametrom, poiskanima z gradientnim spustom:

```
using Vaja10: spust
t = range(0, 2pi, 100)
plot(Tuple.(k1.(t)), label="\$k_1(t)\$")
plot!(Tuple.(k2.(t)), label="\$k_2(s)\$")
ts, it = spust(gradd2, [2, -1.5], 0.2)
scatter!(Tuple(k1(ts[1])), label="\$k_1(t_0)\$")
scatter!(Tuple(k2(ts[2])), label="\$k_2(s_0)\$")
```

in z Newtonovo metodo:

```
using Vaja09: newton
t = range(0, 2pi, 100)
plot(Tuple.(k1.(t)), label="\$k_1(t)\$")
plot!(Tuple.(k2.(t)), label="\$k_2(s)\$")
ts, it = newton(gradd2, jacd2, [2, -1.5])
scatter!(Tuple(k1(ts[1])), label="\$k_1(t_0)\$")
scatter!(Tuple(k2(ts[2])), label="\$k_2(s_0)\$")
```



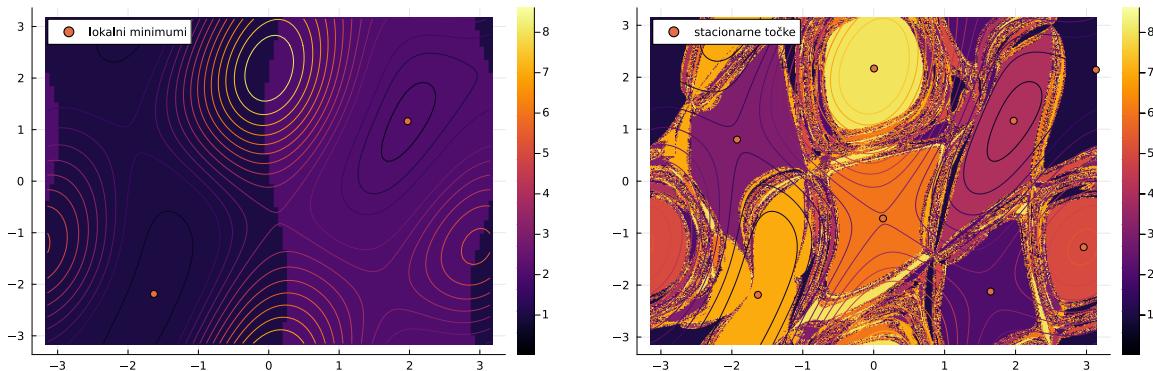
Slika 39: Točki na krivuljama, h katerima konvergira gradientna metoda, sta lokalni minimum, ki pa ni globalni (levo). Newtonova metoda konvergira k sedlu. Točka na k_1 je lokalni minimum, točka na k_2 pa lokalni maksimum (desno).

Za konec si oglejmo še konvergenčna območja za gradientni spust:

```
using Vaja09: konvergenca
using Vaja10: spust
function spustd2(x0)
    ts, it = spust(gradd2, x0, 0.2; maxit=1000)
    ts = map(t -> mod(t + pi, 2pi) - pi, ts)
    return ts, it
end
x, y, Z, ničle, koraki = konvergenca(
    Box2d(Interval(-pi, pi), Interval(-pi, pi)), spustd2, 100, 100;
    atol=1e-2)
heatmap(x, y, Z)
scatter!(Tuple.(ničle), label="lokalni minimumi")
contour!(x, y, d2)
```

in Newtonovo metodo:

```
function newtond2(x0)
    ts, it = newton(gradd2, jacd2, x0; atol=1e-5)
    ts = map(t -> mod(t + pi, 2pi) - pi, ts)
    return ts, it
end
x, y, Z, ničle, koraki = konvergenca(
    Box2d(Interval(-pi, pi), Interval(-pi, pi)), newtond2, 500, 500;
    atol=1e-2)
heatmap(x, y, Z)
scatter!(Tuple.(ničle), label="stacionarne točke")
contour!(x, y, d2)
```



Slika 40: Območja konvergencije za gradientni spust (levo) in Newtonovo metodo (desno)

Problem iskanja minimuma funkcije, ki smo ga reševali, uvrščamo med optimizacijske probleme. Študij optimizacijskih problemov je obsežno raziskovalno področje. Več o tem si lahko preberete v knjigi [14].

10.4 Rešitve

```
#####
ts, it = samopres(k, dk, ts0)

Poišči samopresečišče krivulje `k` s smernim odvodom `dk` z Newtonovo metodo z
začetnim približkom `ts0`. Začetni približek `ts0` in rezultat `ts` sta
dvodimensionalna vektorja z dvema različnima parametroma.

# Primer
```jl
k(t) = [t^2, t^3-t]
dk(t) = [2t, 3t^2-1]
ts, it = samopres(k, dk, [-0.5, 0.5])
```
#####

function samopres(k, dk, ts0)
  f(ts) = k(ts[1]) - k(ts[2])
  df(ts) = hcat(dk(ts[1]), -dk(ts[2]))
  ts, it = newton(f, df, ts0)
  ts = sort(ts)
  if abs(ts[1] - ts[2]) < 1e-12
    throw("Ista parametra ne pomenita samopresečišča.")
  end
  return ts, it
end
```

Program 51: Funkcija, ki poišče samopresečišče krivulje z Newtonovo metodo.

```

using LinearAlgebra

"""
d2 = razdalja2(k1, k2)

Vrni funkcijo kvadrata razdalje `d2(t, s)` med točkama na krivuljah
`k1` in `k2`. Rezultat `d2` je funkcija spremenljivk `t` in `s`, kjer sta
`t` parameter na krivulji `k1` in `s` parameter na krivulji `k2`.
# Primer
```jl
k1(t) = [t, t^2 - 2]
k2(s) = [cos(s), sin(s)]
d2 = razdalja(k1, k2)
d2(1, pi)
```
"""

function razdalja2(k1, k2)
    function d2(t, s)
        delta = k1(t) - k2(s)
        return dot(delta, delta)
    end
    return d2
end

```

Program 52: Funkcija, ki za dani krivulji vrne funkcijo kvadrata razdalje med dvema točkama na krivuljah.

```

"""
x0, it = spust(gradf, x0, h)

Poišči lokalni minimum za funkcijo, podano z gradientom `gradf`, z metodo
najhitrejšega spusta. Argument `x0` je začetni približek, `h` skalar s
katerim pomnožimo gradient.
"""

function spust(gradf, x0, h; maxit=500, atol=1e-8)
    for i = 1:maxit
        x = x0 - h * gradf(x0)
        if norm(x0 - x) < atol
            return x, i
        end
        x0 = x
    end
    throw("Gradientni spust ne konvergira po $maxit korakih!")
end

```

Program 53: Gradientni spust

11 Aproksimacija z linearnim modelom

V znanosti pogosto želimo opisati odvisnost ene količine od druge. Matematičnemu opisu povezave med dvema ali več spremenljivkami pravimo **matematični model**. Primer modela je Hookov zakon za vzem, ki pravi, da je sila F sorazmerna z raztezkom x , torej:

$$F = kx. \quad (11.1)$$

Model povezuje dve količini: silo F in raztezek x . Poleg tega Hookov zakon vpelje še koeficient vzem, k . Koeficientu k pravimo **parameter modela** in ga lahko določimo za vsako vzem posebej z meritvami sile in raztezka.

V tej vaji bomo podatke o koncentraciji CO₂ v zadnjih desetletjih opisali z linearnim modelom.

11.1 Naloga

- Podatke o koncentraciji CO₂ v ozračju aproksimiraj s kombinacijo kvadratnega polinoma in nihanja s periodo 1 leto.
- Parametre modela poišči z normalnim sistemom in QR razcepom.
- Model uporabi za napoved obnašanja koncentracije CO₂ za naslednjih 20 let.

11.2 Linearni model

Najpreprostejši matematični model je **linearni model**, pri katerem odvisno količino y zapišemo kot linearno kombinacijo baznih funkcij φ_j , $j = 1, 2, \dots, k$ neodvisne spremenljivke x :

$$y(x, p) = p_1\varphi_1(x) + p_2\varphi_2(x) + \dots + p_k\varphi_k(x). \quad (11.2)$$

Koeficientom p_j pravimo parametri modela in jih določimo na podlagi meritov. Znanstveniki iščejo model, pri katerem imajo parametri p_j preprosto interpretacijo in pomagajo pri razumevanju pojava, ki ga opisujejo. Bazne funkcije so zato pogosto elementarne funkcije, iz katerih je jasno razvidna narava odvisnosti.

11.2.1 Metoda najmanjših kvadratov

Koeficiente modela, ki najbolje opisujejo izmerjene podatke, lahko poiščemo z **metodo najmanjših kvadratov**. Najprej napišemo pogoje, ki bi jim zadoščali parametri, če bi izmerjeni podatki povsem sledili modelu. Za vsako meritev $y_i = y(x_i)$ bi bila vrednost odvisne količine y_i natanko enaka vrednosti, ki jo predvidi model $M(p, x_i)$. To predpostavko zapišemo s sistemom enačb:

$$\begin{aligned} y_1 &= M(p, x_1) = p_1\varphi_1(x_1) + \dots + p_k\varphi_k(x_1), \\ y_2 &= M(p, x_2) = p_1\varphi_1(x_2) + \dots + p_k\varphi_k(x_2), \\ &\vdots \\ y_n &= M(p, x_n) = p_1\varphi_1(x_n) + \dots + p_k\varphi_k(x_n). \end{aligned} \quad (11.3)$$

Neznanke v zgornjem sistemu so parametri p_j in za **linearni model** so enačbe linearne. To je tudi ena glavnih prednosti linearnega modela. Meritve redko povsem sledijo modelu, zato sistem (11.3) v splošnem ni rešljiv, saj je meritev običajno več kot parametrov sistema. Sistem (11.3) je **predoločen**. Lahko pa poiščemo vrednosti parametrov p_j , pri katerih bo razlika med meritvami in modelom kar

se da majhna. Izkaže se, da je najboljša mera za odstopanje modela od podatkov kar vsota kvadratov razlik med meritvami in napovedjo modela:

$$(y_1 - M(p, x_1))^2 + \dots + (y_n - M(p, x_n))^2 = \sum_{i=1}^n (y_i - M(p, x_i))^2. \quad (11.4)$$

Sistem (11.3) zapišemo v matrični obliki $A\mathbf{p} = \mathbf{y}$, kjer so stolpci matrike sistema A enaki vrednostim baznih funkcij:

$$A = \begin{pmatrix} \varphi_1(x_1) & \varphi_2(x_1) & \dots & \varphi_k(x_1) \\ \varphi_1(x_2) & \varphi_2(x_2) & \dots & \varphi_k(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ \varphi_1(x_n) & \varphi_2(x_n) & \dots & \varphi_k(x_n) \end{pmatrix}, \quad (11.5)$$

stolpec desnih strani pa je enak meritvam:

$$\mathbf{y} = [y_1, y_2, \dots, y_n]^T. \quad (11.6)$$

Pogoj najmanjših kvadratov razlik (11.4) za optimalne vrednosti parametrov \mathbf{p}_{opt} potem zapišemo s kvadratno vektorsko normo:

$$\mathbf{p}_{opt} = \operatorname{argmin}_{\mathbf{p}} \|A\mathbf{p} - \mathbf{y}\|_2^2. \quad (11.7)$$

Statistična interpretacija metode najmanjših kvadratov

Če linearni model obravnavamo kot **statistični model**, so vrednosti parametrov, ki jih dobimo z metodo najmanjših kvadratov, v določenem smislu najboljše cenilke za parametre modela. Natančneje: **Gauss-Markov izrek** pravi, da so cenilke za parametre linearnega modela z najmanjšo varianco ravno vrednosti parametrov, ki jih dobimo z metodo najmanjših kvadratov. Ob predpostavki, da so napake meritve nekorelirane slučajne spremenljivke z enakimi variancami in pričakovano vrednostjo 0.

11.3 Opis sprememb koncentracije CO₂

Na observatoriju **Mauna Loa** na Havajih že vrsto let spremljajo koncentracijo CO₂ v ozračju in podatke objavlja na svoji spletni strani v različnih oblikah. Oglejmo si tedenska povprečja koncentracije od začetka meritev leta 1974. Za prenos podatkov uporabimo knjižnico **FTPClient.jl**.

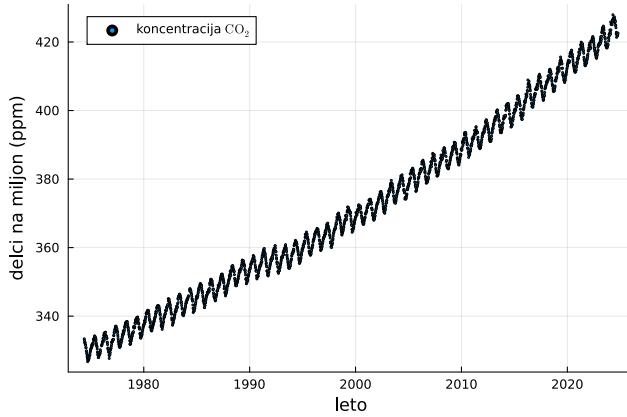
```
using FTPClient
url = "ftp://aftp.cmdl.noaa.gov/products/trends/co2/co2_weekly_mlo.txt"
io = download(url)
data = readlines(io)
```

Nato iz datoteke odstranimo komentarje in izluščimo podatke.

```

filter!(l -> l[1] != '#', data)
data = strip.(data)
data = [split(line, r"\s+") for line in data]
data = [[parse(Float64, x) for x in line] for line in data]
filter!(l -> l[5] > 0, data)
t = [l[4] for l in data]
co2 = [l[5] for l in data]
using Plots
scatter(t, co2, xlabel="leto",
        ylabel="delci na milijon (ppm)",
        label="koncentracija \${CO}_2\$, markersize=1)

```



Slika 41: Koncentracija atmosferskega CO_2 v zadnjih desetletjih

Časovni potek koncentracije CO_2 matematično opišemo kot funkcijo koncentracije v odvisnosti od časa:

$$y = \text{CO}_2(t). \quad (11.8)$$

Model, ki dobro opisuje spremembe CO_2 , lahko sestavimo iz kvadratne funkcije, ki opisuje naraščanje letnih povprečij, in periodičnega dela, ki opiše nihanja med letom:

$$\text{CO}_2(\mathbf{p}, t) = p_1 + p_2 t + p_3 t^2 + p_4 \sin(2\pi t) + p_5 \cos(2\pi t), \quad (11.9)$$

kjer je čas t podan v letih. Predoločen sistem (11.3), ki ga dobimo za naš model, ima $n \times 5$ matriko sistema:

$$A = \begin{pmatrix} 1 & t_1 & t_1^2 & \sin(2\pi t_1) & \cos(2\pi t_1) \\ 1 & t_2 & t_2^2 & \sin(2\pi t_2) & \cos(2\pi t_2) \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & t_n & t_n^2 & \sin(2\pi t_n) & \cos(2\pi t_n) \end{pmatrix}, \quad (11.10)$$

desne strani pa so vrednosti koncentracij CO_2 .

Po metodi najmanjših kvadratov iščemo vrednosti parametrov \mathbf{p} modela $\text{CO}_2(\mathbf{p}, t)$, pri katerih je vsota kvadratov razlik med napovedjo modela in izmerjenimi vrednostmi najmanjša. Zapišimo vsoto kvadratov kot evklidsko normo razlike med vektorjem napovedi modela $A\mathbf{p}$ in vektorjem izmerjenih vrednosti \mathbf{y} . Iščemo torej vektor parametrov \mathbf{p} , pri katerem je vrednost

$$\|A\mathbf{p} - \mathbf{y}\|_2^2 \quad (11.11)$$

najmanjša.

11.4 Normalni sistem

Osvežimo nekaj pojmov iz linearne algebре. *Stolpčni prostor* matrike A je prostor

$$C(A) = \{\mathbf{y} : \mathbf{y} = A\mathbf{x} \text{ za nek } \mathbf{x}\}, \quad (11.12)$$

ki ga razpenjajo stolpci matrike A . *Ničelni prostor* matrike A je prostor

$$N(A) = \{\mathbf{x} : A\mathbf{x} = \mathbf{0}\}, \quad (11.13)$$

v katerem so vektorji, ki jih A preslika v $\mathbf{0}$.

Po metodi najmanjših kvadratov vektor parametrov modela \mathbf{p} izberemo tako, da je norma razlike med napovedjo modela $A\mathbf{p}$ in vrednostmi \mathbf{y} najmanjša. Geometrijsko je $A\mathbf{p}$ pravokotna projekcija vektorja \mathbf{y} na stolpčni prostor $C(A)$, torej je razlika $A\mathbf{p} - \mathbf{y}$ pravokotna na $C(A)$:

$$A\mathbf{p} - \mathbf{y} \perp C(A). \quad (11.14)$$

Iz linearne algebре vemo, da je $C(A)^\perp = N(A^T)$:

$$A\mathbf{p} - \mathbf{y} \in N(A^T) \Rightarrow A^T(A\mathbf{p} - \mathbf{y}) = \mathbf{0}. \quad (11.15)$$

Tako lahko izpeljemo *normalni sistem* za dani predoločen sistem $A\mathbf{p} = \mathbf{y}$:

$$\begin{aligned} A^T(A\mathbf{p} - \mathbf{y}) &= 0 \Rightarrow \\ A^T A\mathbf{p} &= A^T \mathbf{y}. \end{aligned} \quad (11.16)$$

Normalni sistem $A^T A\mathbf{p} = A^T \mathbf{y}$ je kvadraten in ima enolično rešitev, če je matrika A polnega ranga.

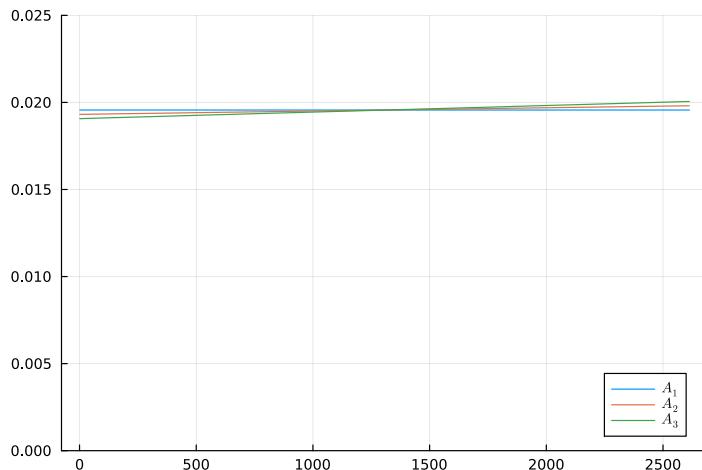
```
using LinearAlgebra
A = hcat(ones(size(t)), t, t .^ 2, cos.(2pi * t), sin.(2pi * t))
N = A' * A
b = A' * co2
p = N \ b
```

Problem normalnega sistema (11.16) je, da je zelo občutljiv. Število občutljivosti matrike sistema pove, ali je matrika slabo pogojena in je sistem posledično občutljiv. Izračunamo ga z ukazom cond:

```
julia> cond(A), cond(N)
(8.448575836430208e10, 4.180774966930753e21)
```

Manjše kot je število občutljivosti, manj je sistem občutljiv. Iz rezultata vidimo, da je že sama matrika A slabo pogojena, matrika N pa še toliko bolj. Razlog je v izbiri baznih funkcij. Če narišemo normirane stolpce A kot funkcije, vidimo, da so zelo podobni.

```
plot(A[:, 1] / norm(A[:, 1]), ylims=[0, 0.025], label="\$A_1\$")
plot!(A[:, 2] / norm(A[:, 2]), label="\$A_2\$")
plot!(A[:, 3] / norm(A[:, 3]), label="\$A_3\$")
```



Slika 42: Normirani prvi trije bazni vektorji (stolpci matrike A)

Veliko število občutljivosti matrike je deloma posledica dejstva, da čas merimo v letih od začetka našega štetja. Vrednosti 1975 in 2020 sta relativno blizu in tako ima vektor vrednosti t_i skoraj enako smer kot vektor enic. Občutljivost matrike A lahko precej zmanjšamo, če časovno skalo premaknemo, da je ničla bliže dejanskim podatkom. Namesto t uporabimo spremenljivko $t - \tau$, kjer je τ premik časovne skale. Najboljša izbira za τ je na sredini podatkov:

```
julia> τ = sum(t) / length(t)
julia> A = hcat(ones(size(t)), t .- τ, (t .- τ) .^ 2, cos.(2pi * t), sin.(2pi * t))
julia> cond(A), cond(A'A)
(425.05456570474456, 180671.3838264739)
```

Matrika A ima število občutljivosti precej manjše kot prej in posledično je manjše tudi za matriko normalnega sistema N .

Prednosti normalnega sistema

Čeprav je normalni sistem zelo občutljiv, se v praksi izkaže, da napaka vendarle ni tako velika. Ima pa normalni sistem nekatere prednosti pred QR razcepom.

Dimenzijske normalnega sistema so dane s številom parametrov in so bistveno manjše od dimenzije matrike predoločenega sistema A . Zato je prostor, ki ga potrebujemo za normalni sistem, bistveno manjši od prostora, ki ga potrebujemo za QR razcep.

Druga prednost normalnega sistema je možnost posodobitve, če dobimo nove podatke. To je uporabno, če na primer podatke dobivamo v toku. Normalni sistem lahko posodobimo vsakič, ko dobimo nov podatek, ne da bi bilo treba hraniti prejšnje podatke.

11.5 QR razcep

Normalni sistem se redko uporablja v praksi. Standardni postopek za iskanje rešitve predoločenega sistema z metodo najmanjših kvadratov je QR razcep. Pri QR razcepu $QR = A$ so stolpci matrike Q ortonormirana baza stolpčnega prostora matrike A , matrika R pa vsebuje koeficiente v razvoju stolpcov matrike A po ortonormirani bazi, določeni s Q . Projekcijo na stolpčni prostor ortogonalne matrike še lažje izračunamo, saj lahko koeficiente izračunamo s skalarnim produktom s stolpci Q . Če predoločen sistem $Ap = y$ pomnožimo z desne s Q^T in upoštevamo, da je $Q^T Q = I$, dobimo zopet kvadratni sistem za vektor parametrov p :

$$\begin{aligned}
 Ap &= y \Rightarrow \\
 \Rightarrow QRp &= y \\
 \Rightarrow Q^T QRp &= Q^T y \\
 \Rightarrow Rp &= Q^T y.
 \end{aligned} \tag{11.17}$$

Matrika R je zgornje trikotna, tako da lahko sistem rešimo z obratnim vstavljanjem. V Julii uporabimo funkcijo `qr`, ki vrne posebni podatkovni tip, posebej namenjen QR razcep matrike:

```

julia> F = qr(A)
julia> p_qr = F \ co2 # ekvivalentno R\Q'*b
julia> p_norm = (A' * A) \ (A' * co2) # rešitev z normalnim sistemom
julia> razlika = norm(p_norm - p_qr)
6.822543246114055e-13

```

Razlika med rešitvijo s QR razcepom in normalnim sistemom je zanemarljiva. Vendar se QR uporablja, ker je numerično stabilnejši v primerjavi z normalnim sistemom. Tudi vgrajen operator `\` v Julii in Matlabu uporabi QR razcep, če je sistem predoločen.

11.6 Kaj pa CO₂?

Koncentracija CO₂ se v ozračju vztrajno povečuje. Poglejmo, kaj nam o tem povedo parametri modela, ki smo jih izračunali na podlagi izmerjenih podatkov.

```

julia> p_qr
5-element Vector{Float64}:
368.81376214677266
1.8515079712064941
0.01411695002100984
-0.8053801587031669
2.852551492543684

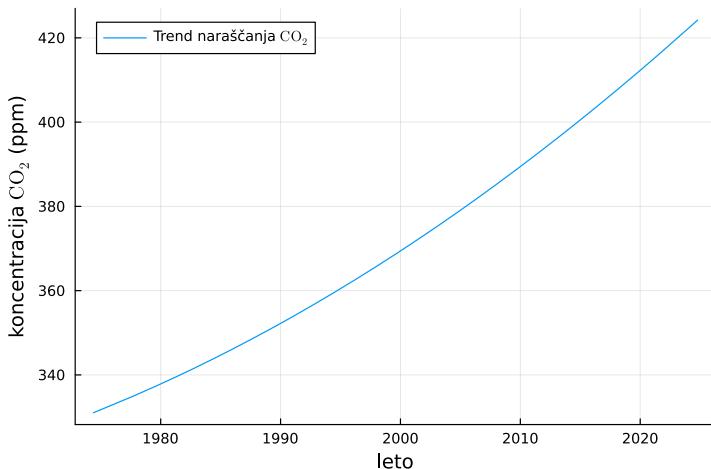
```

Koeficient p_1 pove povprečno koncentracijo na sredini merilnega obdobja, p_2 in p_3 pa sta koeficienta pri linearinem in kvadratnem členu. Amplituda letnih nihanj je enaka velikosti vektorja $[p_4, p_5]$. Kot je razvidno iz parametrov modela, je naraščanje kvadratno in ne linearne. Ne le, da se vsako leto poveča koncentracija, vsako leto se poveča za večjo vrednost. Če odmislimo nihanja zaradi letnih časov, dobimo trend naraščanja:

```

model_trend(t) = p_qr[1:3]' * [1, t - τ, (t - τ)^2]
plot(model_trend, t[1], t[end], label="Trend naraščanja \${\mathrm{CO}}_2",
      xlabel="leto", ylabel="koncentracija \${\mathrm{CO}}_2\$( ppm)")

```



Slika 43: Rezultati modela brez letnih nihanj

Lahko poskusimo tudi napovedati prihodnost:

```
julia> model(t) = p_qr' * [1, t - τ, (t - τ)^2, cos(2pi * t), sin(2pi * t)]
julia> napoved = model.([2030, 2040, 2050])
3-element Vector{Float64}:
437.1444876692537
465.63393287633704
496.9467680876224
```

Naš model napoveduje, da bo leta 2050 koncentracija CO₂ v ozračju znašala skoraj 500 ppm.

Kaj smo se naučili?

- Linearni model je funkcija, pri kateri *parametri* nastopajo *linearno*.
- Parametre modela poiščemo z *metodo najmanjših kvadratov*.
- Za iskanje parametrov po metodi najmanjših kvadratov je numerično najprimernejši *QR razcep*, če smo v stiski s prostorom, pa lahko uporabimo *normalni sistem*.
- Premik neodvisne spremenljivke lahko bistveno izboljša numerično stabilnost.
- Koncentracija CO₂ prav zares narašča.

12 Interpolacija z zlepki

Pri interpolaciji iščemo *interpolacijsko funkcijo*, ki se v danih točkah ujema z danimi vrednostmi. Najbolj znana je interpolacija s polinomi. Če je danih točk veliko, je pogosto bolje namesto ene interpolacijske funkcije z veliko parametri interval razdeliti na več podintervalov in na vsakem uporabiti različne interpolacijske funkcije z malo parametri. Funkcijam, ki so definirane z različnimi predpisi na različnih intervalih, pravimo zlepki.

12.1 Naloga

- Podatke iz Tabele 2 interpoliraj s [Hermitovim kubičnim zlepkom](#).

| x | x_1 | x_2 | \dots | x_n |
|---------|--------|--------|---------|--------|
| $f(x)$ | y_1 | y_2 | \dots | y_n |
| $f'(x)$ | dy_1 | dy_2 | \dots | dy_n |

Tabela 2: Podatki, ki jih potrebujemo za Hermitov kubični zlepek.

- Uporabi Hermitovo bazo kubičnih polinomov, ki zadoščajo pogojem iz Tabele 3 in jih z linearno funkcijo preslikaj z intervala $[0, 1]$ na interval $[x_i, x_{i+1}]$.

| | $p(0)$ | $p(1)$ | $p'(0)$ | $p'(1)$ |
|----------|--------|--------|---------|---------|
| h_{00} | 1 | 0 | 0 | 0 |
| h_{01} | 0 | 1 | 0 | 0 |
| h_{10} | 0 | 0 | 1 | 0 |
| h_{11} | 0 | 0 | 0 | 1 |

Tabela 3: Vrednosti baznih polinomov $h_{ij}(t)$ in njihovih odvodov v točkah $t = 0$ in $t = 1$

- Definiraj podatkovni tip [HermitovZlepek](#) za Hermitov kubični zlepek, ki vsebuje podatke iz Tabele 2.
- Napiši funkcijo [vrednost\(zlepek, x\)](#), ki izračuna vrednost Hermitovega kubičnega zlepka za dano vrednost argumenta x . Omogoči, da se vrednosti tipa [HermitovZlepek](#) [kliče kot funkcije](#).
- S Hermitovim zlepkom interpoliraj funkcijo $f(x) = \cos(2x) + \sin(3x)$ na intervalu $[0, 6]$ v 10 točkah. Napako oceni s formulo za napako polinomske interpolacije:

$$f(x) - p_3(x) = \frac{f^{(4)}(\xi)}{4!}(x - x_1)(x - x_2)(x - x_3)(x - x_4) \quad (12.1)$$

in oceno primerjaj z dejansko napako. Upoštevaj, da je pri Hermitovi interpolaciji $x_1 = x_2$ in $x_3 = x_4$. Nariši graf napake.

- Z oceno za napako (12.1) določi število interpolacijskih točk, pri katerem bo napaka Hermitovega zlepka manjša od 10^{-7} .
- Funkcijo f interpoliraj tudi z Newtonovim polinomom in primerjaj napako z napako Hermitovega zlepka.

12.2 Hermitov kubični zlepek

Hermitov kubični zlepek, ki interpolira podatke iz Tabele 2, je sestavljen iz kubičnih polinomov p_k na intervalih $[x_k, x_{k+1}]$. Kubični polinom je podan s štirimi parametri, ravno toliko kot je podatkov v krajiščih intervala $[x_k, x_{k+1}]$. Zato lahko vsak polinom p_k določimo le na podlagi podatkov v točkah

x_k in x_{k+1} . Polinom p_k poiščemo tako, da interval $[x_k, x_{k+1}]$ preslikamo z linearno funkcijo na $[0, 1]$ in uporabimo Lagrangeovo bazo h_{00}, h_{01}, h_{10} in h_{11} za podatke iz Tabele 3. Polinome poiščemo v standardni bazi:

$$h_{ij}(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3. \quad (12.2)$$

Če izračunamo še odvod $h'_{ij}(t) = a_1 + 2a_2 t + 3a_3 t^2$, dobimo naslednji sistem enačb za koeficiente baznega polinoma h_{00} :

$$\begin{aligned} h_{00}(0) &= a_0 = 1, \\ h'_{00}(0) &= a_1 = 0, \\ h_{00}(1) &= a_0 + a_1 + a_2 + a_3 = 0, \\ h'_{00}(1) &= a_1 + 2a_2 + 3a_3 = 0. \end{aligned} \quad (12.3)$$

Za ostale polinome dobimo podobne sisteme, ki imajo isto matriko sistema, razlikujejo pa se v desnih straneh. Če desne strani postavimo v matriko, dobimo identično matriko. Inverzna matrika sistema (12.3) ima v stolpcih koeficiente baznih polinomov h_{ij} . Inverz izračunamo z Julio.

```
A = [1 0 0 0; 1 1 1 1; 0 1 0 0; 0 1 2 3]
inv(A)
```

```
4x4 Matrix{Float64}:
1.0 -0.0 0.0 -0.0
0.0 0.0 1.0 -0.0
-3.0 3.0 -2.0 -1.0
2.0 -2.0 1.0 1.0
```

Bazni polinomi so enaki

$$\begin{aligned} h_{00}(t) &= 1 - 3t^2 + 2t^3, \\ h_{01}(t) &= 3t^2 - 2t^3, \\ h_{10}(t) &= t - 2t^2 + t^3, \\ h_{11}(t) &= -t^2 + t^3. \end{aligned} \quad (12.4)$$

Določiti moramo še preslikavo z intervala $[x_k, x_{k+1}]$ na $[0, 1]$. Naj bo $x \in [x_k, x_{k+1}]$ in $t \in [0, 1]$. Potem je preslikava med x in t enaka:

$$t(x) = \frac{x - x_k}{x_{k+1} - x_k}, \quad (12.5)$$

medtem ko je preslikava med t in x enaka:

$$x(t) = x_k + t(x_{k+1} - x_k). \quad (12.6)$$

Želimo uporabiti bazo h_{ij} , zato podatke interpoliramo s polinomom $p_k(x(t))$ na intervalu $[0, 1]$. Če t izračunamo kot $t(x)$ iz (12.5), velja:

$$\begin{aligned}
 p_k(x(0)) &= p_k(x_k) = y_k, \\
 \frac{d}{dt}p_k(x(0)) &= p'_k(x_k)x'(0) = p'_k(x_k)(x_{k+1} - x_k) = dy_k(x_{k+1} - x_k), \\
 p_k(x(1)) &= p_k(x_{k+1}) = y_{k+1}, \\
 \frac{d}{dt}p_k(x(1)) &= p'_k(x_k)x'(1) = p'_k(x_{k+1})(x_{k+1} - x_k) = dy_{k+1}(x_{k+1} - x_k)
 \end{aligned} \tag{12.7}$$

in

$$p_k(x) = y_k h_{00}(t) + y_{k+1} h_{01}(t) + (x_{k+1} - x_k)(dy_k h_{10}(t) + dy_{k+1} h_{11}(t)). \tag{12.8}$$

Samostojno delo

Sedaj napiši naslednje funkcije in tipe:

- funkcijo `hermiteint(x, x_int, y, dy)`, ki izračuna vrednost Hermitovega polinoma v x (rešitev Program 54) in
- podatkovni tip `HermitovZlepek` ter funkcijo `vrednost(x, Z::HermitovZlepek)`, ki izračuna vrednost Hermitovega zlepka Z v dani točki x (rešitev Program 55 in Program 56).

Vrednosti kot funkcije

Na primeru Hermitovega zlepka ilustriramo, kako v Julii ustvarimo vrednosti, ki se obnašajo kot funkcije. Tako zapis v programskemu jeziku približamo matematičnemu zapisu. Za Hermitov zlepek smo definirali tip `HermitovZlepek` in funkcijo `vrednost`, s katero izračunamo vrednost Hermitovega zlepka v dani točki. Vrednost tipa `HermitovZlepek` hrani interpolacijske podatke in hkrati predstavlja zlepek kot funkcijo. V Julii lahko definiramo, da se vrednosti tipa `HermitovZlepek` obnašajo kot funkcije:

```
(z::HermitovZlepek)(x) = vrednost(z, x)
```

Vrednosti zlepka v dani točki izračunamo tako, kot bi to naredili v matematičnem zapisu:

```
z = HermitovZlepek([0, 1, 2], [1, 2, 3], [2, 1, 2])
z(1.23)
```

Napisane funkcije preskusimo tako, da funkcijo $f(x) = \cos(2x) + \sin(3x)$ interpoliramo v 10 ekvidistančnih točkah na intervalu $[0, 6]$:

```

f(x) = cos(2x) + sin(3x)
df(x) = -2sin(2x) + 3cos(3x)
x = range(0, 6, 10)
y = f.(x)
dy = df.(x)

z = HermitovZlepek(x, y, dy)

```

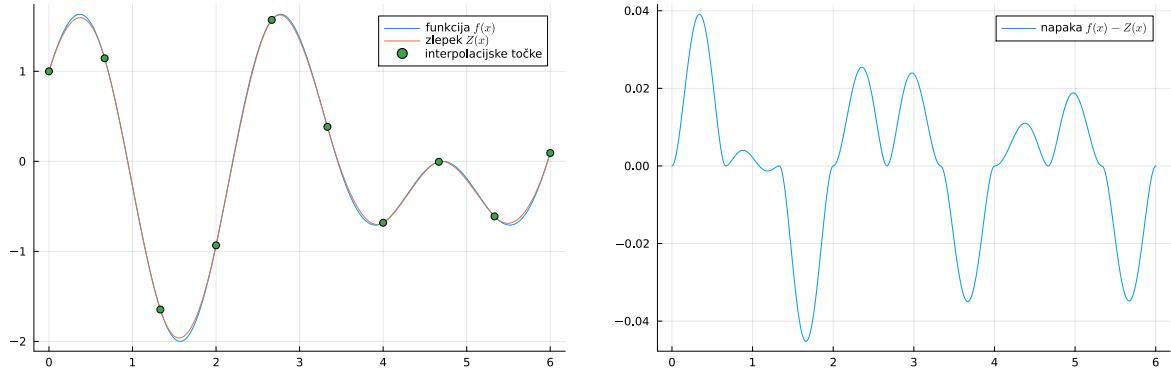
Na eno sliko narišemo graf funkcije in zlepka, na drugo pa napako interpolacije (razliko med funkcijo in zlepkom).

```

using Plots
plot(f, 0, 6, label="funkcija  $f(x)$ ", legend=:topright)
plot!(x -> z(x), 0, 6, label="zlepak  $Z(x)$ ")
scatter!(x, y, label="interpolacijske točke")

plot(x -> f(x) - z(x), 0, 6, label="napaka  $f(x) - Z(x)$ ")

```



Slika 44: Graf funkcije $f(x) = \cos(2x) + \sin(3x)$ in Hermitovega zlepka, ki interpolira funkcijo f na 10 točkah (levo). Graf napake interpolacije (desno). Zlepak interpolira tudi vrednosti odvodov, zato ima napaka v interpolacijskih točkah stacionarne točke.

12.3 Ocena za napako

Ocene za napako Hermitove interpolacije lahko izračunamo analitično. Napako polinomske interpolacije v splošnem zapišemo kot:

$$f(x) - p_n(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!} \prod_{k=1}^n (x - x_i), \quad (12.9)$$

kjer je ξ neznana vrednost znotraj interpolacijskega območja, odvisna od vrednosti x . Poleg vrednosti imamo na voljo tudi odvode, zato interpolacijski točki štejemo dvojno. Tako dobimo naslednjo formulo za napako:

$$f(x) - p_3(x) = \frac{f^{(4)}(\xi)}{4!} (x - x_1)^2 (x - x_2)^2. \quad (12.10)$$

Poščimo še oceno za največjo vrednost napake. Vrednosti $f^{(4)}(\xi)$ ne poznamo in jo lahko zgolj ocenimo. Poleg tega moramo poiskati po absolutni vrednosti največjo vrednost polinoma $p(x) = (x - x_1)^2 (x - x_2)^2$ na intervalu $[x_1, x_2]$. V krajiščih intervala je vrednost polinoma $p(x)$ enaka nič, zato je maksimum dosežen v notranjosti in je dosežen v stacionarni točki. Poščimo torej stacionarno točko $p(x)$, ki leži znotraj intervala $[x_1, x_2]$. Odvod je enak:

$$\begin{aligned} p'(x) &= 2(x - x_1)(x - x_2)^2 + 2(x - x_1)^2(x - x_2) \\ &= 2(x - x_1)(x - x_2)(x - x_2 + x - x_1) \\ &= 4(x - x_1)(x - x_2) \left(x - \frac{x_1 + x_2}{2} \right). \end{aligned} \quad (12.11)$$

Polinom p ima tri stacionarne točke: dve v krajiščih intervala in eno v njegovem središču $\frac{x_1+x_2}{2}$. Vrednost polinoma v središču je tudi največja vrednost, dosežena na $[x_1, x_2]$:

$$p\left(\frac{x_1+x_2}{2}\right) = \left(\frac{x_1+x_2}{2} - x_1\right)^2 \left(\frac{x_1+x_2}{2} - x_2\right)^2 = \frac{1}{4}(x_2 - x_1)^4. \quad (12.12)$$

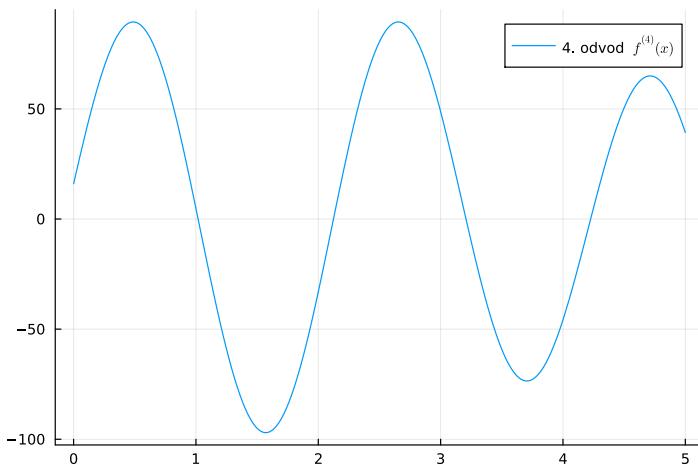
Največjo napako ocenimo z:

$$|f(x) - p_3(x)| \leq \frac{f^{(4)}(\xi)}{4!} \frac{(x_2 - x_1)^4}{4} = \frac{1}{96} f^{(4)}(\xi)(x_2 - x_1)^4. \quad (12.13)$$

Če želimo uporabiti oceno (12.13), moramo poznati oceno za četrти odvod funkcije. Za približno oceno uporabimo avtomatsko odvajanje in narišemo graf četrtega odvoda:

```
using ForwardDiff
ddf(x) = ForwardDiff.derivative(df, x)
d3f(x) = ForwardDiff.derivative(ddf, x)
d4f(x) = ForwardDiff.derivative(d3f, x)

plot(d4f, 0, 5, label="4. odvod \$f^{(4)}(x)", legend=:topright)
```



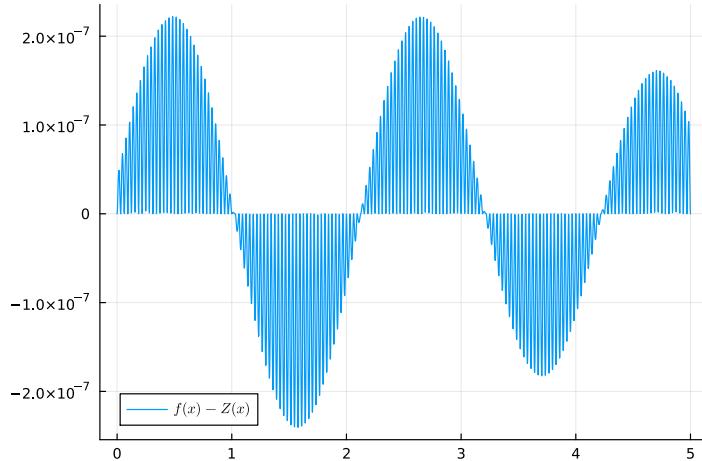
Slika 45: Četrti odvod funkcije $f(x) = \cos(2x) + \sin(3x)$ na intervalu $[0, 6]$

Ocena za napako omogoča, da vnaprej izberemo interpolacijske točke, za katere bo napaka manjša od predpisane. Če želimo, da je napaka manjša od ε , potem mora biti širina intervala manjša od:

$$|x_2 - x_1| \leq \sqrt[4]{\frac{96}{f^{(4)}(\xi)}} \varepsilon. \quad (12.14)$$

Preverimo še numerično, ali izračunana formula deluje:

```
eps = 1e-6
d4fmax = 100
h = (96 * eps / d4fmax)^0.25
n = Integer(ceil((5 - 0) / h))
x = range(0, 5, n + 1)
Z = HermitovZlepek(x, f.(x), df.(x))
plot(x -> f(x) - Z(x), 0, 5, label="\$f(x) - Z(x)\$")
```



Slika 46: Napaka interpolacije, pri čemer smo število interpolacijskih točk izbrali tako, da je napaka manjša od 10^{-6} .

Teoretične ocene za napako niso vedno uporabne

Za določitev napake smo uporabili oceno (12.13). Pri tem smo meje za četrti odvod, ki nastopa v oceni, določili kar z grafa. V praksi teoretične ocene, kakršna je (12.1), ni mogoče vedno uporabiti, saj je nepraktično določiti meje za višje odvode. Pogosto zato uporabimo manj eksaktne načine, da dobimo vsaj neko informacijo o napaki, četudi ni povsem zanesljiva.

12.4 Newtonov interpolacijski polinom

Naj bodo x_1, x_2, \dots, x_n vrednosti neodvisne spremenljivke in y_1, y_2, \dots, y_n vrednosti neznane funkcije. Interpolacijski polinom, ki interpolira podatke x_i, y_i , je polinom p , za katerega velja:

$$\begin{aligned} p(x_1) &= y_1, \\ p(x_2) &= y_2, \\ &\vdots \\ p(x_n) &= y_n. \end{aligned} \tag{12.15}$$

Newtonov interpolacijski polinom je interpolacijski polinom, zapisan v obliki:

$$p(x) = a_0 + a_1(x - x_1) + a_2(x - x_1)(x - x_2) + \dots + a_n \prod_{k=1}^{n-1} (x - x_k). \tag{12.16}$$

Pri tem smo uporabili Newtonovo bazo polinomov za dane interpolacijske točke:

$$1, x - x_1, (x - x_1)(x - x_2), \dots, \prod_{k=1}^{n-2} (x - x_k) \text{ in } \prod_{k=1}^{n-1} (x - x_k). \tag{12.17}$$

Koeficiente a_i je namreč lažje izračunati, kot če bi bil polinom zapisan v standardni bazi. Poleg tega je računanje vrednosti polinoma v standardni bazi numerično nestabilno, če so vrednosti x_i relativno blizu. Koeficiente a_i poiščemo bodisi tako, da rešimo spodnje trikotni sistem, ki ga dobimo iz enačb (12.15), ali pa z [deljenimi diferencami](#).

Samostojno delo

Definiraj naslednje tipe in funkcije:

- podatkovno strukturo za Newtonov interpolacijski polinom `NewtonovPolinom` (Program 57),
- funkcijo `vrednost(p, x)`, ki izračuna vrednost Newtonovega polinoma v dani točki (Program 58),
- funkcijo `interpoliraj(NewtonovPolinom, x, y)`, ki poišče koeficiente polinoma za dane interpolacijske podatke (Program 59).

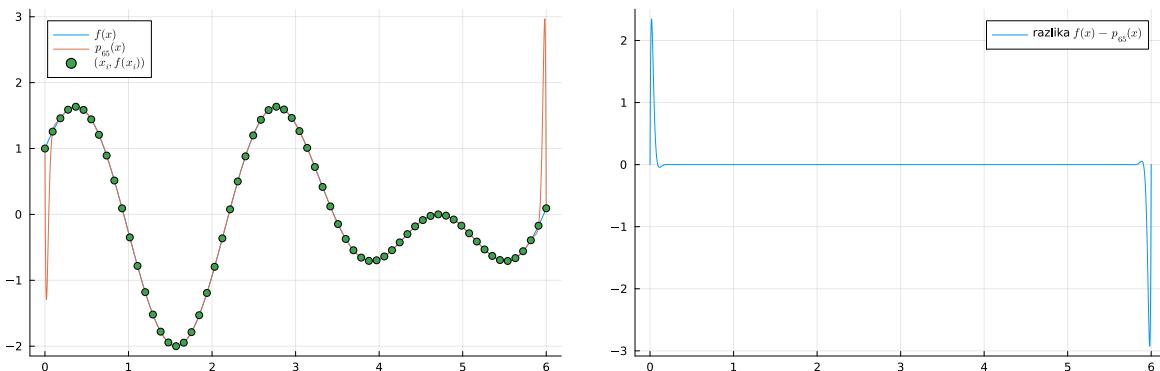
12.5 Rungejev pojav

Pri nizkih stopnjah polinomov se z večanjem števila interpolacijskih točk napaka interpolacije zmanjšuje. A le do neke mere! Če stopnjo polinoma preveč povečamo, začne napaka naraščati.

```
f(x) = cos(2x) + sin(3x)
n = 65
x = range(0, 6, n + 1)
p_newton = interpoliraj(NewtonovPolinom, x, f.(x))

plot(f, 0, 6, label="$f(x)$")
plot!(x -> p_newton(x), 0, 6, label="$p_{65}(x)$")
scatter!(x, f.(x), label="$f(x_i)$")

plot(x -> f(x) - p_newton(x), 0, 6, label="razlika $f(x) - p_{65}(x)$")
```



Slika 47: Interpolacija s polinomi visokih stopenj na ekvidistančnih točkah na robu območja močno niha. Interpolacija funkcije $f(x) = \cos(2x) + \sin(3x)$ s polinomom stopnje 65 (levo). Razlika med funkcijo in interpolacijskim polinomom (desno).

Opazimo, da se napaka na robu znatno poveča. Povečanje je posledica velikih oscilacij, ki se pojavijo na robu, če interpoliramo s polinomom visoke stopnje na ekvidistančnih točkah. To je znan pojav pod imenom [Rungejev pojav](#). Problemu se pogosto izognemo, če namesto ekvidistančnih točk uporabimo [Čebiševe točke](#).

Kaj smo se naučili?

- Metodo deljenih diferenc in Newtonov polinom lahko uporabimo tudi, če so poleg vrednosti podani tudi odvodi.
- Zaradi Rungejevega pojava interpolacija s polinomi visokih stopenj na ekvidistančnih točkah ni najboljša izbira. Interpolacija na Čebiševih točkah pogosto deluje tudi za visoke stopnje polinoma.
- Zlepki so enostavni za uporabo, učinkoviti (malo operacij za izračun) in imajo v določenih primerih boljše lastnosti od polinomov visokih stopenj.

12.6 Rešitve

```
# bazne funkcije na [0, 1]
h00(t) = 1 + t^2 * (-3 + 2 * t)
h01(t) = t^2 * (3 - 2 * t)
h10(t) = t * (1 + t * (-2 + t))
h11(t) = t^2 * (-1 + t)

"""
hermiteint(x, x_int, y, dy)

Izračunaj vrednost Hermitovega kubičnega polinoma p v točki `x`, ki interpolira
podatke `p(x_int[1]) == y[1]`, `p(x_int[2]) == y[2]` in
`p'(x_int[1]) == dy[1]`, `p'(x_int[2]) == dy[2]`.
"""

function hermiteint(x, x_int, y, dy)
    dx = x_int[2] - x_int[1]
    t = (x - x_int[1]) / dx
    return y[1] * h00(t) + y[2] * h01(t) + dx * (dy[1] * h10(t) + dy[2] * h11(t))
end
```

Program 54: Funkcija, ki interpolira podatke iz Tabele 2 s Hermitovim kubičnim polinomom.

```
"""
Podatkovna struktura, ki hrani podatke za Hermitov kubični zlepak
v interpolacijskih točkah `x` z danimi vrednostmi `y` in vrednostmi
odvoda `dy` .
"""

struct HermitovZlepak
    x
    y
    dy
end
```

Program 55: Podatkovni tip za Hermitov kubični zlepak

```

"""
y = vrednost(x, Z)

Izračunaj vrednost Hermitovega kubičnega zlepka `Z` v dani točki `x`.

function vrednost(x, Z::HermitovZlepek)
    i = searchsortedfirst(Z.x, x)
    if (x == first(Z.x))
        return first(Z.y)
    end
    if (i > lastindex(Z.x)) || (i == firstindex(Z.x))
        throw(BoundsError(Z, x))
    end
    return hermiteint(x, Z.x[i-1:i], Z.y[i-1:i], Z.dy[i-1:i])
end

(Z::HermitovZlepek)(x) = vrednost(x, Z)

```

Program 56: Funkcija, ki izračuna vrednost Hermitovega kubičnega zlepka.

```

"""
NewtonovPolinom(a, x)

Vrni [Newtonov interpolacijski polinom](https://en.wikipedia.org/wiki/Newton_
polynomial)
oblike `a[1]+a[2](x-x[1])+a[3](x-x[1])(x-x[2])+\dots`
s koeficienti `a` in vozlišči `x`.

# Primer

Poglejmo polinom ``1+x+x(x-1)``, ki je definiran s koeficienti `[1, 1, 1]` in z
vozliščema
``x_0 = 0`` in ``x_1 = 1``

```jldoctest
julia> p = NewtonovPolinom([1, 1, 1], [0, 1])
NewtonovPolinom([1, 1, 1], [0, 1])
```

"""

struct NewtonovPolinom
    a # koeficienti
    x # vozlišča
end

```

Program 57: Podatkovni tip za polinom v Newtonovi obliki

```

"""
vrednost(p::NewtonovPolinom, x)

Izračunaj vrednot Newtonovega polinoma `p` v `x` s Hornerjevo metodo.

# Primer

```jldoctest
julia> p = NewtonovPolinom([0, 0, 1], [0, 1]);
julia> vrednost(p, 2)
2
```
"""

function vrednost(p::NewtonovPolinom, x)
    n = length(p.x)
    v = p.a[n+1]
    for i = n:-1:1
        v = p.a[i] + (x - p.x[i]) * v
    end
    return v
end

(p::NewtonovPolinom)(x) = vrednost(p, x)

```

Program 58: Funkcija, ki izračuna vrednost Newtonovega polinoma.

```

using LinearAlgebra
"""
    p = interpoliraj(NewtonovPolinom, x, f)

Izračunaj koeficiente Newtonovega interpolacijskega polinoma, ki interpolira
podatke `y(x[k])=f[k]`. Če se katere vrednosti `x` ponovijo,
metoda predvideva, da so v `f` poleg vrednosti podani tudi odvodi.

# Primer

Polinom ``x^2-1``, ki interpolira podatke `x=[0,1,2]` in `y=[-1, 0, 3]`, v
Newtonovi obliki zapišemo kot ``1 + x + x(x-1)``

```jldoctest
julia> p = interpoliraj(NewtonovPolinom, [0, 1, 2], [-1, 0, 3])
NewtonovPolinom([-1.0, 1.0, 1.0], [0, 1])
```

Če imamo več istih vrednosti abscise `x`, moramo v `f` podati vrednosti funkcije
in odvodov. Na primer polinom ``p(x) = x^4 = x + 3x(x-1) + 3x(x-1)^2 + x(x-1)^3``
ima v ``x=1`` vrednosti ``p(1)=1, p'(1)=4, p''(1)=12``

```jldoctest
julia> p = interpoliraj(NewtonovPolinom, [0,1,1,1,2], [0,1,4,12,16])
NewtonovPolinom([0.0, 1.0, 3.0, 3.0, 1.0], [0, 1, 1, 1])

julia> x = (1,2,3,4,5); p.(x) .- x.^4
(0.0, 0.0, 0.0, 0.0, 0.0)
```
"""

function interpoliraj(P::Type{<:NewtonovPolinom}, x, f)
    n = length(x) - 1
    m = length(f) - 1
    @assert n == m
    a = zeros(n + 1, n + 1)
    a[:, 1] = f;
    fakteta = 1
    for j = 2:n + 1
        fakteta *= j - 1
        for i = j:n + 1
            if x[i] != x[i-j+1]
                a[i,j] = (a[i,j-1] - a[i-1,j-1])/(x[i] - x[i-j+1]);
            else
                a[i,j] = a[i,j-1]/fakteta
                a[i,j-1] = a[i-1,j-1]
            end
        end
    end
    return NewtonovPolinom(diag(a), x[1:end-1])
end

```

Program 59: Izračun koeficientov Newtonovega polinoma z deljenimi diferencami

12.7 Testi

```
@testset "Hermitova baza" begin
    @test Vaja12.h00.([0, 1]) ≈ [1, 0]
    @test Vaja12.h01.([0, 1]) ≈ [0, 1]
    @test Vaja12.h10.([0, 1]) ≈ [0, 0]
    @test Vaja12.h11.([0, 1]) ≈ [0, 0]

end
```

Program 60: Test, ki preveri Hermitovo bazo (12.4).

```
@testset "Hermitov polinom" begin
    # kubični polinom se bo vedno ujemal s Hermitovim polinomom
    p3(x) = x^3 - x
    dp3(x) = 3x^2 - 1
    xint = [0, 2]
    h(x) = hermiteint(x, xint, p3.(xint), dp3.(xint))
    @test isapprox(p3(0), h(0))
    @test isapprox(p3(0.5), h(0.5))
    @test isapprox(p3(1.5), h(1.5))
end
```

Program 61: Test za izračun Hermitovega kubičnega polinoma

```
@testset "Hermitov zlepek" begin
    p3(x) = x^3 - 2x^2 + 1
    dp3(x) = 3x^2 - 4x
    x = [-1, 0.5, 3.5, 4]
    z = HermitovZlepek(x, p3.(x), dp3.(x))
    @test_throws BoundsError z(-2)
    for xi in x
        @test isapprox(z(xi), p3(xi))
    end
    @test isapprox(z(-0.2), p3(-0.2))
    @test isapprox(z(1.1), p3(1.1))
    @test isapprox(z(3.7), p3(3.7))
    @test_throws BoundsError z(5.1)
end
```

Program 62: Test za izračun vrednosti zlepka

```
@testset "Vrednost Newtonovega polinoma" begin
    p = NewtonovPolinom([1, 1, 2], [0, 1])
    f(x) = 1 + x*(1 + 2*(x - 1))
    @test isapprox(p(1), f(1))
    @test isapprox(p(2.5), f(2.5))
    @test isapprox(p(pi), f(pi))
end
```

Program 63: Test za izračun vrednosti Newtonovega polinoma

```
@testset "Interpolacija z Newtonovim polinomom" begin
    f(x) = 1 + (x-2)*(2.5 + (x-1)*3.5)
    p = interpoliraj(NewtonovPolinom, [2, 1, 0], f.([2, 1, 0]))
    @test isapprox(p.a, [1, 2.5, 3.5])
    @test isapprox(p.x, [2, 1])
end
```

Program 64: Test za izračun koeficientov Newtonovega interpolacijskega polinoma

13 Integrali

Za numerični izračun določenega integrala funkcije f na intervalu $[a, b]$

$$\int_a^b f(x)dx \quad (13.1)$$

obstaja veliko različnih metod, ki jih imenujemo *numerične kvadrature* ali na kratko *kvadrature*. V tej vaji si bomo ogledali, kako izbrati primerno metodo, glede na funkcijo, ki jo integriramo.

13.1 Naloga

- Definiraj podatkovni tip, ki predstavlja določeni integral funkcije na danem intervalu.
- Definiraj podatkovni tip `Kvadratura`, ki hrani podatke o vozilih in utežeh za dano kvadraturo na danem intervalu.
- Implementiraj funkcijo `integriraj(int, kvad::Kvadratura)`, ki s kvadraturo `kvad` izračuna integral `int`.
- Implementiraj sestavljeno trapezno, sestavljeno Simpsonovo, Gauss-Legendrove kvadrature in adaptivno Simpsonovo metodo. Za izračun vozlov in uteži Gauss-Legendrovih kvadratur uporabi paket [FastGaussQuadrature.jl](#).
- Implementirane metode uporabi za izračun naslednjih integralov:

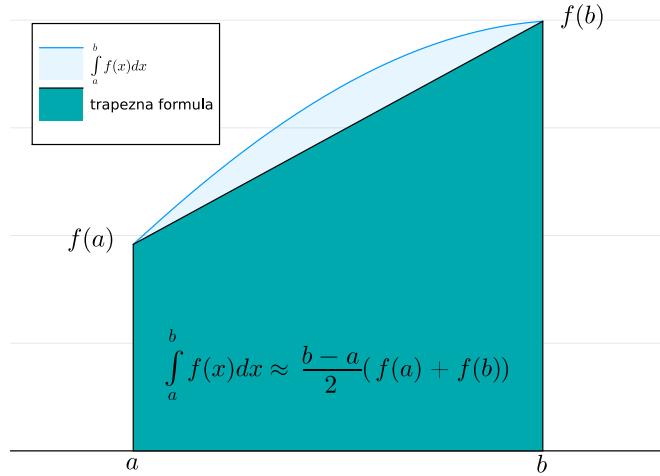
$$\int_0^3 \sin(x^2)dx \quad \text{in} \quad \int_{-1}^2 |2 - x^2| dx. \quad (13.2)$$

Napako oceni tako, da rezultat primerjaš z rezultatom, ki ga dobiš, če uporabiš dvakrat več vozlov in uporabiš [Richardsonovo ekstrapolacijo](#).

13.2 Trapezno pravilo in sestavljeni trapezni pravilo

Trapezno pravilo izhaja iz formule za ploščino trapeza. Integral f na $[a, b]$ ocenimo s ploščino trapeza z oglišči $(a, 0), (a, f(a)), (b, f(b))$ in $(b, 0)$:

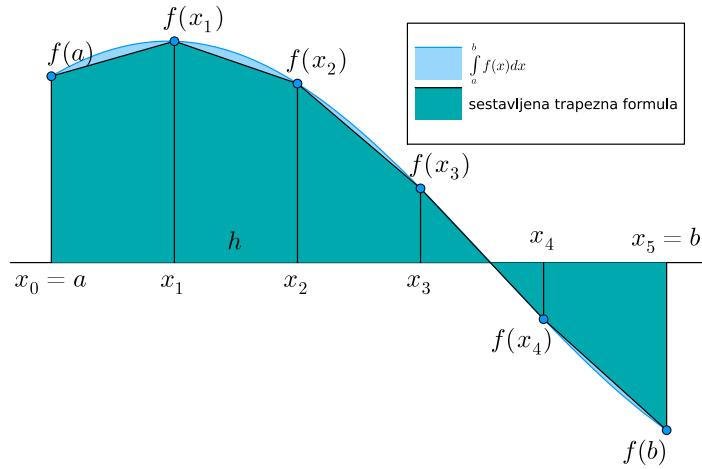
$$\int_a^b f(x)dx \approx (b - a) \frac{f(a) + f(b)}{2}. \quad (13.3)$$



Slika 48: Ilustracija trapeznega pravila

Sestavljeni trapezni pravilo izpeljemo tako, da interval $[a, b]$ razdelimo na manjše dele in na vsakem podintervalu uporabimo trapezno formulo. Interval $[a, b]$ razdelimo na n podintervalov in označimo z $x_0 = a < x_1 < x_2 < \dots < x_{n-1} < x_n = b$ krajišča podintervalov. Integral na $[a, b]$ razbijemo na vsoto integralov po intervalih $[x_i, x_{i+1}]$ in na vsakem podintervalu uporabimo trapezno formulo:

$$\begin{aligned} \int_a^b f(x)dx &= \int_a^{x_1} f(x)dx + \int_{x_1}^{x_2} f(x)dx + \dots + \int_{x_{n-1}}^b f(x)dx \approx \\ &\approx (x_1 - a)\frac{f(a) + f(x_1)}{2} + (x_2 - x_1)\frac{f(x_1) + f(x_2)}{2} + \dots + (b - x_{n-1})\frac{f(x_{n-1}) + f(b)}{2}. \end{aligned} \quad (13.4)$$



Slika 49: Ilustracija sestavljenega trapeznega pravila

Če točke x_k razporedimo enakomerno, tako da je $h = x_1 - a = x_2 - x_1 = \dots = b - x_{n-1}$ in $x_k = a + kh$, dobimo naslednjo formulo

$$\begin{aligned} \int_a^b f(x)dx &\approx \frac{h}{2}(f(a) + f(x_1)) + \frac{h}{2}(f(x_1) + f(x_2)) + \dots \\ &\dots + \frac{h}{2}(f(x_{n-2}) + f(x_{n-1})) + \frac{h}{2}(f(x_{n-1}) + f(b)) = \\ &= h\left(\frac{1}{2}f(a) + f(x_1) + f(x_2) + \dots + f(x_{n-1}) + \frac{1}{2}f(b)\right). \end{aligned} \quad (13.5)$$

Predznačeni dolžini podintervala $x_{k+1} - x_k$ v sestavljeni kvadraturni formuli pogosto rečemo *korak*. Korak je lahko tudi negativen, če je spodnja meja integrala večja od zgornje. Za formulo (13.5) pravimo, da je sestavljena formula z enakomernim korakom.

Manjša kot je širina intervala $h = \frac{b-a}{n}$, bližje je vsota (13.5) pravi vrednosti integrala. V nadaljevanju se bomo prepričali, da za trapezno formulo napaka pada sorazmerno s h^2 .

Preden se lotimo implementacije trapeznega in sestavljenega trapeznega pravila, bomo napisali teste, ki bodo preverili pravilnost bodoče implementacije.

Testno voden razvoj

V tej vaji bomo sledili načinu razvoja programske opreme, ki se imenuje **testno voden razvoj**. Ideja testno vodenega razvoja je, da se najprej napiše teste in šele nato kodo, ki zadovolji te teste. S tem se pozornost preusmeri iz implementacije na uporabo in iz ustvarjalca kode na uporabnika. S tem ko najprej napišemo test, moramo najprej razmisliti, kako bo naša koda uporabljena. Tako se vživimo v vlogo uporabnika in začasno odmislimo implementacijo. Prednosti testno vodenega razvoja so tako dvojne. Uporabniška izkušnja je boljša, poleg tega pa je koda preverjena vsaj na enem primeru.

Vemo, da je trapezno pravilo in zato tudi sestavljeno trapezno pravilo točno za linearne funkcije. Zato bomo v testu integrirali linearne funkcije.

Da lahko napišemo test, si moramo najprej zamisliti, kako bomo kodo uporabljali. V našem primeru želimo izračunati približek za določeni integral in za to uporabiti sestavljeno trapezno pravilo. Osnovni pojem je integral, zato si zasluži svoj podatkovni tip `Integral`. Določeni integral ima dve sestavini: funkcijo in interval. Tako bo imel podatkovni tip `Integral` dve polji: `f` za funkcijo in `interval` za interval. Za daljše ime `interval` se odločimo, da preprečimo zamenjavo s precej podobno besedo `integral`.

Interval bi lahko predstavili s parom ali nizom števil, vendar raje definiramo nov podatkovni tip `Interval`, ki vsebuje dve polji `min` in `max`. Na prvi pogled se zdi, da vpeljava novega tipa nima smisla, saj par števil `(1.0, 2.0)` in vrednost `Interval(1.0, 2.0)` vsebuje iste podatke. Vendar pa nosi vrednost tipa `Interval` dodatno informacijo o vlogi, ki jo ima v programu. Ločevanje po vlogi in ne le po obliki podatkov je priporočljivo, saj lahko prepreči napake, ki jih je sicer težko odkriti in so posledica zamenjave podatkov iste oblike, ki nastopajo v različnih vlogah.

Ko smo določili zapis integrala, moramo razmisliti, kako v programu predstaviti sestavljeno trapezno pravilo. V objektno usmerjenih programskih jezikih bi definirali vmesnik za splošno kvadraturo in razred za sestavljeno trapezno pravilo, ki ta vmesnik implementira. Julia uporablja večlično razdelitev, ki omogoča definicijo več specializiranih metod za isto funkcijo. Izbera metode je odvisna od tipa vhodnih podatkov. Zato je naravno, da za sestavljeno trapezno pravilo definiramo svoj podatkovni tip `Trapez` in metodo `integriraj(i::Integral, m::Trapez)` za splošno funkcijo `integriraj`. Test povzame odločitve in zasnovno programskega vmesnika, kot smo si ga zamislili.

```
@testset "Sestavljeno trapezno pravilo" begin
    i = Integral(x -> 2x + 1, Interval(1.0, 3.0))
    @test integriraj(i, Trapez(4)) ≈ 10
end
```

Program 65: Test za sestavljeno trapezno formulo

Samostojno delo

Definiraj naslednje tipe in funkcije in poskrbi, da bo test uspešen:

- tip `Interval(a, b)`, ki predstavlja zaprti interval $[a, b]$ (Program 69),
- tip `Integral(fun, interval::Interval)`, ki predstavlja določeni integral funkcije `fun` na intervalu `interval` (Program 70),
- tip `Trapez(n::Int)`, ki predstavlja sestavljen trapezno formulo z n enakomernimi koraki (Program 71) in
- metodo `integriraj(i::Integral, k::Trapez)`, ki izračuna približek za integral `i` s sestavljenem trapezno formulo `k` (Program 72).

13.3 Simpsonovo pravilo

Simpsonovo pravilo dobimo tako, da poleg vrednosti funkcije v krajičih intervala uporabimo še vrednost funkcije v središču intervala. Tako dobimo formulo:

$$\int_0^h f(x)dx = \frac{h}{6} \left(f(a) + 4f\left(\frac{a+b}{2}\right) + f(b) \right) + R_f, \quad (13.6)$$

kjer je napaka enaka $R_f = -\frac{1}{2880}h^5 f^{(4)}(\xi)$ za neznano vrednost $\xi \in [0, h]$. Obstoj vrednosti ξ je posledica Lagrangeevega izreka in vrednost ξ je odvisna od funkcije f , ki jo integriramo. Izpeljavo Simpsonovega pravila si lahko ogledate v podpoglavlju 13.8.

Podobno kot trapezno lahko tudi Simpsonovo pravilo preoblikujemo v sestavljen pravilo. Sestavljen Simpsonovo pravilo z $2n$ enakomernimi koraki dolžine h je dano kot:

$$\int_a^b f(x)dx \approx \frac{h}{3} \left(f(a) + f(b) + 4 \sum_{k=1}^n f(a + (2k-1)h) + 2 \sum_{j=1}^{n-1} f(a + 2jh) \right), \quad (13.7)$$

kjer je $h = \frac{b-a}{2n}$. Napaka sestavljenega pravila (13.7) je enaka:

$$-\frac{1}{180}h^4(b-a)f^{(4)}(\xi). \quad (13.8)$$

Podobno kot pri trapeznem pravilu sledimo testno vodenemu razvoju in najprej napišemo test. Večino zasnove smo določili že pri trapeznem pravilu. Preostane nam le še podatkovni tip za Simpsonovo pravilo, ki ga imenujemo `Simpson`.

Samostojno delo

Napiši test za sestavljen Simpsonovo pravilo. Vemo, da je Simpsonovo pravilo točno za polinome stopnje 3, zato v testu uporabi polinom stopnje 3 (Program 66).

Definiraj naslednje tipe in metode ter poskrbi, da se test pravilno izvede:

- podatkovni tip `Simpson(n::Int)`, ki predstavlja sestavljen Simpsonovo formulo (Program 73) in
- metodo `integriraj(i::Integral, k::Simpson)`, ki izračuna približek za integral `i` s sestavljenem Simpsonovo formulo `k` (Program 74).

13.4 Gaussove kvadraturne formule

Sestavljeni trapezna (13.5) in Simpsonova formula (13.7) nista nič drugega kot uteženi vsoti funkcijskih vrednosti v izbranih točkah na integracijskem intervalu

$$\int_a^b f(x)dx \approx \sum_{k=0}^n u_k f(x_k). \quad (13.9)$$

Vrednostim u_k pravimo *uteži*, x_k pa so *vozli* kvadraturne formule. Za sestavljeni trapezno formulo so uteži enake $u_0 = u_n = \frac{h}{2}$ in $u_k = h$ za $1 \leq k \leq n - 1$, vozli pa so $x_k = a + kh$ za $k = 0, 1, \dots, n$.

Pri trapezni in Simpsonovi kvadraturi so vozli razporejeni na robu in sredini integracijskega intervala, uteži pa so določene tako, da je formula točna za polinome čim višjih stopenj. To ni optimalna izbira. Če dovolimo, da so vozli razporejeni drugače, lahko dobimo kvadraturo, ki je točna za polinome še višjih stopenj. Za integral na intervalu $[-1, 1]$ dobimo [Gauss-Legendrove kvadrature](#)

$$\int_{-1}^1 f(t)dt \approx \sum_{k=1}^n w_k f(t_k), \quad (13.10)$$

ki so del družine [Gaussovih kvadratur](#). Za Gauss-Legendrovo kvadraturo z n vozli so ničle Legendrovega polinoma stopnje n optimalna izbira. Gauss-Legendrova kvadratura z n vozli je točna za polinome stopnje $2n - 1$. Za $n = 1$ dobimo sredinsko pravilo z vozлом v $x_1 = 0$, ki je točno za linearne funkcije. Za $n = 2$ dobimo formulo

$$\int_{-1}^1 f(x)dx \approx f\left(-\frac{1}{\sqrt{3}}\right) + f\left(\frac{1}{\sqrt{3}}\right), \quad (13.11)$$

ki je točna za kubične polinome.

Če želimo Gauss-Legendrovo kvadraturo uporabiti na poljubnem intervalu $[a, b]$, moramo interval $[-1, 1]$ preslikati na $[a, b]$. To naredimo z linearno funkcijo oziroma uvedbo nove spremenljivke $t \in [-1, 1]$:

$$t(x) = 2\frac{x-a}{b-a} - 1 \quad \text{in} \quad x(t) = \frac{b-a}{2}t + \frac{a+b}{2}. \quad (13.12)$$

Integral na $[a, b]$ izrazimo z integralom na $[-1, 1]$:

$$\int_a^b f(x)dx = \int_{-1}^1 f(x(t))x'(t)dt = \frac{1}{2}(b-a) \int_{-1}^1 f(x(t))dt, \quad (13.13)$$

kjer smo upoštevali, da je $x'(t) = \frac{1}{2}(b-a)$. Gauss-Legendrove formule na $[a, b]$ so tako:

$$\int_a^b f(x)dx = \frac{1}{2}(b-a) \sum_{k=1}^N w_k f(x_k), \quad (13.14)$$

kjer je

$$x_k = \frac{b-a}{2}t_k + \frac{a+b}{2}. \quad (13.15)$$

Vozlov x_k in uteži u_k za Gauss-Legendrove kvadrature ni tako enostavno poiskati, saj moramo poiskati ničle ustreznega Legendrovega polinoma in določiti uteži. Obstaja tudi [Golub-Welschev algoritem](#), ki vozle in uteži poišče kot lastne vrednosti posebej ustvarjene tridiagonalne matrike. A

računanje vozlov in uteži presega obseg te vaje in zato bomo za njihov izračun uporabili knjižnico `FastGaussQuadrature.jl`.

Pri trapeznem in Simpsonovem pravilu smo uteži in vozle računali sproti med računanjem približka za integral. To smo si lahko privoščili, ker je izračun precej enostaven in ne zahteva veliko časa v primerjavi z izračunom utežene vsote. Pri Gauss-Legendrovih kvadraturah pa je izračun vozlov in uteži precej zahtevnejši, zato jih je smiselno shraniti za večkratno uporabo. Kvadrature bomo obravnavali splošno in definirali podatkovni tip `Kvadratura`, ki predstavlja splošno kvadraturo oblike:

$$\int_a^b f(x)dx \approx \sum_{k=1}^n u_k f(x_k). \quad (13.16)$$

Če pogledamo podrobno, je kvadratura (13.16) podana z vozli x_1, x_2, \dots, x_n , utežmi u_1, u_2, \dots, u_n in intervalom $[a, b]$.

Samostojno delo

Definiraj podatkovni tip `Kvadratura` s tremi polji:

- `x` vektor vozlov,
- `u` vektor uteži in
- interval interval.

Rešitev je Program 75.

Ko imamo kvadraturo definirano, hočemo z njo izračunati integral. Napišimo v duhu testno vodenega razvoja test za uporabo podatka tipa `Kvadratura`. Kot smo videli, lahko kvadraturo oblike (13.16) uporabimo na poljubnem intervalu, zato pričakujemo, da funkcija `integriraj` deluje tudi za integrale po intervalih, ki niso enaki intervalu za kvadraturo. Za primer bomo uporabili Gauss-Legendrovo kvadraturo z dvema vozloma (13.11), ki je točna za kubične polinome.

Samostojno delo

- Napiši test za integracijo s kvadraturo tipa `Kvadratura` (Program 67).
- Napiši metodo `integriraj(i::Integral, k::Kvadratura)`, ki izračuna približek za dani integral `i` z dano kvadraturo `k` (Program 76).
- Napiši test za funkcijo, ki vrne Gauss-Legendrove kvadrature. Uporabi dejstvo, da so kvadrature z n vozli točne za polinome stopnje $2n - 1$ (Program 68).
- Napiši funkcijo `glkvad(n::Int)`, ki vrne Gauss-Legendrovo kvadraturo z n vozli (Program 77).

Zakaj so vse kvadraturne formule utežene vsote?

Vse kvadrature, ki jih poznamo, lahko na nek način prevedemo na uteženo vsoto funkcijskih vrednosti:

$$\int_a^b f(x) = \sum_{i=1}^n u_i f(x_i). \quad (13.17)$$

Zakaj je tako? Integral lahko obravnavamo kot preslikavo na prostoru integrabilnih funkcij:

$$I_{[a,b]} : f \mapsto \int_a^b f(x) dx. \quad (13.18)$$

Preslikava $I_{[a,b]}$ je linearji funkcional, zato želimo, da je tudi kvadratura $K(f)$ linearji funkcional. Če želimo kvadraturo v končnem času izračunati, mora biti odvisna le od končnega števila funkcijskih vrednosti. To pa pomeni, da mora biti funkcija $K(f)$ linearja kombinacija končnega števila funkcijskih vrednosti $f(x_1), f(x_2), \dots, f(x_n)$:

$$K(f) = k(f(x_1), f(x_2), \dots, f(x_n)) = \sum_{i=1}^n u_i f(x_i). \quad (13.19)$$

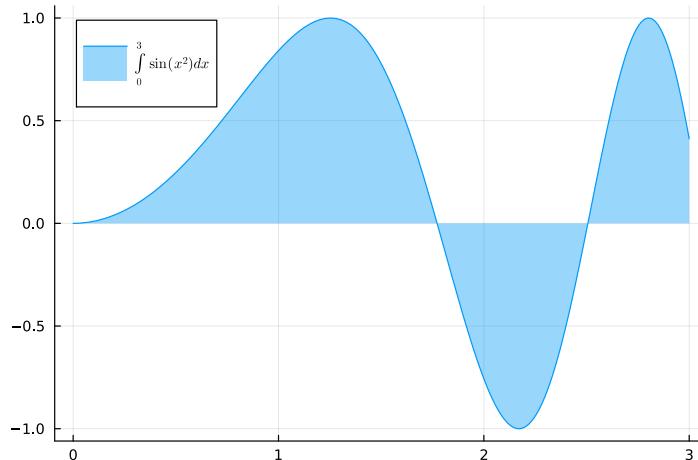
Izjema so adaptivne metode, pri katerih je izbira vozlov x_i odvisna od izbire funkcije f , ki jo integriramo. Zato adaptivne metode strogo gledano niso linearji funkcionali, kljub temu, da jih lahko prevedemo na uteženo vsoto.

13.5 Primeri

Metode, ki smo jih implementirali v prejšnjih poglavjih, bomo uporabili za izračun integrala:

$$\int_0^3 \sin(x^2) dx. \quad (13.20)$$

```
using Plots
f(x) = sin(x^2)
plot(f, 0, 3, fillrange=0, fillalpha=0.4, label="\$\\int_0^3 \\sin(x^2) dx\$")
```



Slika 50: Graf funkcije, ki jo integriramo.

Izračunajmo integral (13.20) z metodami, ki smo jih implementirali. Za vse metode uporabimo enako število vozlov.

```
julia> integral = Integral(f, Interval(0.0, 3.0))
julia> I_t11 = integriraj(integral, Trapez(10))
0.7303937723036156

julia> I_s11 = integriraj(integral, Simpson(5))
0.7834780936263591

julia> I_gl11 = integriraj(integral, glkvad(11))
0.7735624553692868
```

Kako vemo, kateri rezultat je natančnejni? Točnega rezultata ne znamo izračunati, saj integral (13.20) ni elementaren. Nekako moramo oceniti napako. Uveljavili sta se dve strategiji za oceno napake. Za sestavljenje kvadraturne formule rezultat primerjamo z rezultatom iste kvadrature s polovičnim korakom. Za Gauss-Legendrove kvadrature pa uporabimo kvadraturo višjega reda.

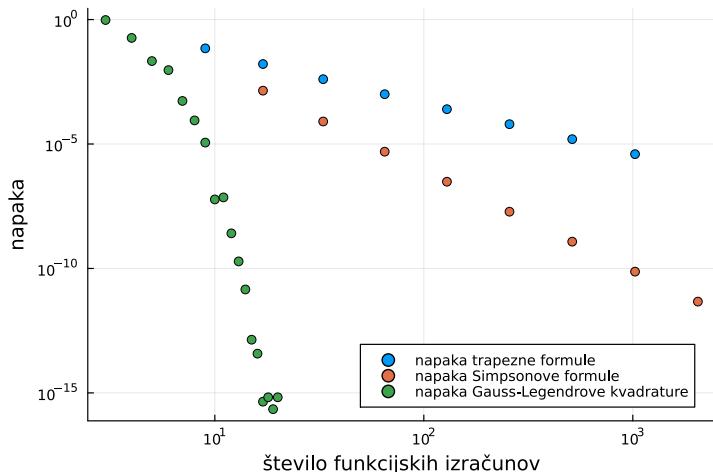
```
julia> napaka_t11 = I_t11 - integriraj(integral, Trapez(20))
-0.0327887111119024

julia> napaka_s11 = I_s11 - integriraj(integral, Simpson(10))
0.009366039840207008

julia> napaka_gl11 = I_gl11 - integriraj(integral, glkvad(12))
-7.409709779082618e-8
```

Vidimo, da se je najbolje odrezala Gauss-Legendrova kvadratura. Oglejmo si na grafu, kako absolutna vrednost napake pojema s številom izračunanih funkcijskih vrednosti za vse tri metode.

```
I_p = integriraj(integral, glkvad(21))
n = 2 .^ (3:10)
I_t = [integriraj(integral, Trapez(k)) for k in n]
I_s = [integriraj(integral, Simpson(k)) for k in n]
n_gl = 3:20
I_gl = [integriraj(integral, glkvad(k)) for k in n_gl]
scatter(n .+ 1, abs.(I_t .- I_p), label="napaka trapezne formule")
scatter!(2n .+ 1, abs.(I_s .- I_p), label="napaka Simpsonove formule",)
scatter!(n_gl, abs.(I_gl .- I_p), scale=:log10,
        label="napaka Gauss-Legendrove kvadrature",
        xlabel="število funkcijskih izračunov", ylabel="napaka",
        legend=:bottomright, yticks=10.0 .^ (0:-5:-15))
```



Slika 51: Absolutna vrednost napake pri izračunu $\int_0^3 \sin(x^2) dx$ z različnimi kvadraturami. Naklon premice pri trapeznem in Simpsonovem pravilu je enak redu metode.

Richardsonova ekstrapolacija

Če imamo kvadraturo reda n za dani integral

$$\int_a^b f(x)dx = K(h) + Ch^n + \mathcal{O}(h^{n+1}), \quad h = \frac{b-a}{N} \quad (13.21)$$

lahko z [Richardsonovo ekstrapolacijo](#) izpeljemo kvadraturo višjega reda:

$$\begin{aligned} I &= K(h) + Ch^n + \mathcal{O}(h^{n+1}), \\ I &= K\left(\frac{h}{2}\right) + C\frac{h^n}{2^n} + \mathcal{O}(h^{n+1}) \Rightarrow \\ &\Rightarrow 2^n I - I = 2^n K\left(\frac{h}{2}\right) - K(h) + \mathcal{O}(h^{n+1}) \\ &\Rightarrow I = \frac{2^n K\left(\frac{h}{2}\right) - K(h)}{2^n - 1} + \mathcal{O}(h^{n+1}). \end{aligned} \quad (13.22)$$

Napako lahko ocenimo tako, da rezultat, dobljen s kvadraturo $K(h)$, primerjamo z Richardsonovo ekstrapolacijo (13.22):

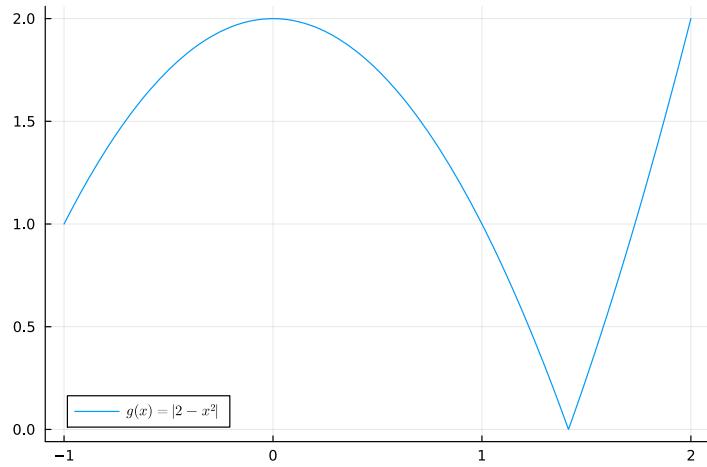
$$\begin{aligned} I - K(h) &\approx \frac{2^n}{2^n - 1} \left(K\left(\frac{h}{2}\right) - K(h) \right) \\ I - K\left(\frac{h}{2}\right) &\approx \frac{1}{2^n - 1} \left(K\left(\frac{h}{2}\right) - K(h) \right). \end{aligned} \quad (13.23)$$

Poglejmo še primer:

$$\int_{-1}^2 |2 - x^2| dx. \quad (13.24)$$

Narišimo graf funkcije $g(x) = |2 - x^2|$ na intervalu $[-1, 2]$.

```
using Plots
g(x) = abs(2 - x^2)
plot(g, -1, 2, label="g(x) = |2 - x^2|")
```



Slika 52: Graf funkcije $g(x) = |2 - x^2|$ na intervalu $[-1, 2]$

Podobno kot prej bomo na grafu predstavili, kako je napaka odvisna od števila izračunanih vrednosti funkcije. V tem primeru lahko vrednost integrala izračunamo:

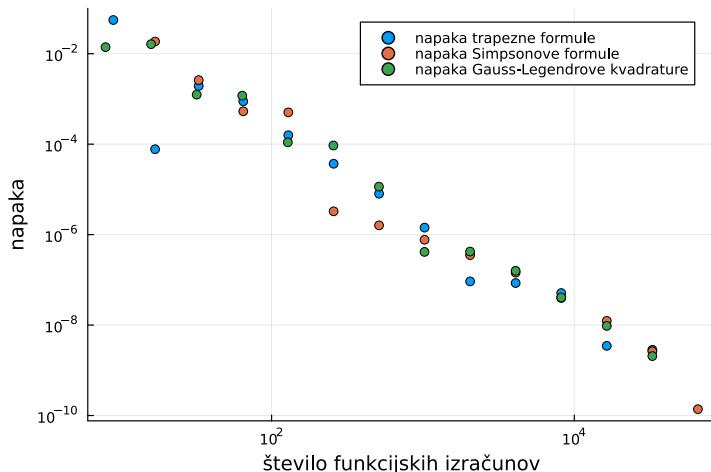
$$\begin{aligned} \int_{-1}^2 |2 - x^2| dx &= \int_{-1}^{\sqrt{2}} (2 - x^2) dx + \int_{\sqrt{2}}^2 -(2 - x^2) dx = \\ &= F(\sqrt{2}) - F(-1) - (F(2) - F(\sqrt{2})) = 2F(\sqrt{2}) - F(-1) - F(2), \end{aligned} \quad (13.25)$$

kjer je $F(x) = \int (2 - x^2) dx = 2x - \frac{x^3}{3}$.

```
julia> F(x) = 2x - x^3/3
julia> I_p = 2F(sqrt(2)) - F(-1) - F(2)
4.1045694996615865
```

Narišimo graf napake:

```
integral = Integral(g, Interval(-1.0, 2.0))
n = 2 .^ (3:15)
I_t = [integriraj(integral, Trapez(k)) for k in n]
I_s = [integriraj(integral, Simpson(k)) for k in n]
I_gl = [integriraj(integral, glkvad(k)) for k in n]
scatter(n .+ 1, abs.(I_t .- I_p), label="napaka trapezne formule")
scatter!(2n .+ 1, abs.(I_s .- I_p), label="napaka Simpsonove formule",)
scatter!(n, abs.(I_gl .- I_p), scale=:log10,
       label="napaka Gauss-Legendrove kvadrature",
       xlabel="število funkcijskih izračunov", ylabel="napaka",
       legend=:topright, yticks=10.0 .^ (0:-2:-15))
```



Slika 53: Absolutna vrednost napake pri izračunu integrala (13.24) z različnimi kvadraturami

Vidimo, da pri integralu (13.24) visok red kvadratur ne pomaga kaj dosti. Vse tri metode se obnašajo približno enako. Vzrok je v tem, da funkcije $g(x)$ ni mogoče dobro aproksimirati s polinomi. V takih primerih se bolje obnesejo **adaptivne kvadraturne formule**.

Kaj smo se naučili?

- Kvadraturne formule se večinoma prevedejo na uteženo vsoto funkcijskih vrednosti v vozlih kvadrature.
- Gauss-Legendrove kvadrature so najprimernejše za integrale funkcij, ki jih lahko dobro aproksimiramo s polinomi.
- Za splošno rabo so najprimernejše adaptivne kvadrature.

13.6 Testi

```
@testset "Sestavljeni Simpsonovo pravilo" begin
    # Simpsonovo pravilo je točno za kubične polinome
    integral = Integral(x -> x^3 + x^2, Interval(-1.0, 3.0))
    # nedoločeni integral je  $x^4/4 + x^3/3$ 
    I(x) = x^4 / 4 + x^3 / 3
    @test integriraj(integral, Simpson(3)) ≈ I(3) - I(-1)
end
```

Program 66: Test sestavljenega Simpsonovega pravila

```
@testset "Splošna kvadratura" begin
    # Gauss-Legendrova kvadratura na dveh točkah
    k = Kvadratura(1 / sqrt(3) * [-1, 1], [1.0, 1.0], Interval(-1.0, 1.0))
    # formula je točna za kubične polinome
    integral = Integral(x -> x^3 - x^2, Interval(-1.0, 3.0))
    I(x) = x^4 / 4 - x^3 / 3
    @test integriraj(integral, k) ≈ I(3) - I(-1)
end
```

Program 67: Test za integracijo s splošno kvadraturo oblike (13.16)

```

@testset "Gauss-Legendrove kvadrature" begin
    k = glkvad(3)
    @test length(k.x) == 3
    # formula je točna za polinome do vključno stopnje 5
    for n=1:5
        integral = Integral(x -> x^n, Interval(0., 1.))
        @test integriraj(integral, k) ≈ 1/(n+1)
    end
end

```

Program 68: Test za generiranje Gauss-Legendrovih kvadratur

13.7 Rešitve

```

"""
Podatkovna struktura za interval `[min, max]`.

"""
struct Interval{T}
    min::T
    max::T
end

```

Program 69: Podatkovni tip za zaprti interval $[a, b]$

```

"""
Podatkovna struktura za določeni integral funkcije `fun` na danem intervalu.

"""
struct Integral{T}
    f
    interval::Interval{T}
end

```

Program 70: Podatkovni tip za določeni integral $\int_a^b f(x)dx$

```

"""Krovni podatkovni tip za vse vrste kvadratur"""
abstract type AbstraktnaKvadratura end

"""
Parametri za sestavljen trapezno pravilo. Sestavljen trapezno
pravilo ima en sam parameter `n`, ki pove, na koliko podintervalov
razdelimo interval.
"""

struct Trapez <: AbstraktnaKvadratura
    n::Integer
end

```

Program 71: Podatkovni tip za sestavljen trapezno pravilo z n enakomernimi koraki

```

"""
I = integriraj(i::Integral, k::Trapez)

Izračunaj približek za integral `i` s sestavljenou trapezno kvadraturo.
"""

function integriraj(i::Integral, k::Trapez)
    a = i.interval.min
    b = i.interval.max
    h = (b - a) / k.n
    I = (i.f(a) + i.f(b)) / 2
    for j in 1:k.n-1
        I += i.f(a + j * h)
    end
    return h * I
end

```

Program 72: Funkcija, ki izračuna integral s sestavljenim trapeznim pravilom.

```

"""
Parametri za sestavljenou Simpsonovo pravilo. Pravilo ima en sam
parameter `n`, ki pove, na koliko podintervalov razdelimo interval.
"""

struct Simpson <: AbstraktnaKvadratura
    n::Integer
end

```

Program 73: Podatkovni tip za sestavljenou Simpsonovo pravilo z $2n$ enakomernimi koraki

```

"""
I = integriraj(i::Integral, k::Simpson)

Izračunaj približek za integral `i` s sestavljenou Simpsonovo kvadraturo.
"""

function integriraj(i::Integral, k::Simpson)
    a = i.interval.min
    b = i.interval.max
    h = (b - a) / k.n
    I = (i.f(a) + i.f(b) + 4 * i.f(b - h / 2))
    for j in 1:k.n-1
        I += 4 * i.f(a + (j - 0.5) * h)
        I += 2 * i.f(a + j * h)
    end
    return (h / 6) * I
end

```

Program 74: Funkcija, ki izračuna integral s sestavljenim Simpsonovim pravilom.

```

"""
Kvadratura(x, u, interval)

Podatkovna struktura za splošno kvadraturo z danimi vozli in utežmi.

"""
struct Kvadratura{T} <: AbstraktnaKvadratura
    x::Vector{T}
    u::Vector{T}
    interval::Interval{T}
end

```

Program 75: Podatkovni tip za splošno kvadraturno formulo

```

"""Izračunaj predznačeno dolžino intervala."""
dolžina(int::Interval) = int.max - int.min

"""
I = integriraj(i::Integral, k::Kvadratura)

Izračunaj približek za integral `i` s kvadraturo `k`.
# Primer
```jl
i = Integral(x->sin(x^2), Interval(2., 5.))
k = Kvadratura([0., 1.], [0.5, 0.5], Interval(0., 1.)) # trapezna formula
integriraj(i, k)
```

"""
function integriraj(i::Integral, k::Kvadratura)
    # funkcija, ki vozle kvadrature preslikava na interval integrala
    razteg = dolžina(i.interval) / dolžina(k.interval)
    preslikaj(x) = razteg * (x - k.interval.min) + i.interval.min
    # zaporedje parov (u(j), x(j)) ustvarimo s funkcijo `zip`
    I = sum(u * i.f(preslikaj(x)) for (u, x) in zip(k.u, k.x))
    return razteg * I
end

```

Program 76: Funkcija, ki izračuna integral z dano kvadraturo.

```

using FastGaussQuadrature
````

 kvadratura = glkvad(n)

Izračunaj vozle in uteži za Gauss-Legendrove kvadraturne formule z
`n` vozli. Funkcija vrne vrednost tipa `Kvadratura`, ki jo lahko uporabimo
v funkciji `integriraj`.

Primer
```jl
k = glkvad(10)
i = Integral(x->exp(-x^2), Interval(1.0, 2.0))
integriraj(i, k)
````

````
function glkvad(n)
    x, u = gausslegendre(n)
    return Kvadratura(x, u, Interval(-1.0, 1.0))
end
````
```

Program 77: Funkcija, ki vrne Gauss-Legendrovo kvadraturo z  $n$  vozli.

### 13.8 Izpeljava Simpsonovega pravila

Simpsonovo pravilo za izračun približka integrala poleg vrednosti funkcije v krajiščih uporabi še vrednost funkcije na sredini. Pravilo zapišemo v naslednji obliki:

$$\int_0^h f(x)dx = Af(a) + Bf\left(\frac{a+b}{2}\right) + Cf(b) + R_f, \quad (13.26)$$

kjer so  $A$ ,  $B$  in  $C$  uteži ter  $R_f$  razlika med pravo vrednostjo in približkom. Uteži določimo z metodo nedoločenih koeficientov. Vrednosti  $A$ ,  $B$  in  $C$  določimo tako, da je  $R_f = 0$  za polinome čim višjih stopnj. Začnemo s polinomom stopnje 0, se pravi s konstanto 1:

$$\int_0^h dx = h = A + B + C. \quad (13.27)$$

Nadaljujemo s polinomi  $x$  in  $x^2$ :

$$\begin{aligned} \int_0^h x dx &= \frac{h^2}{2} = A \cdot 0 + B \cdot \frac{h}{2} + C \cdot h, \\ \int_0^h x^2 dx &= \frac{h^3}{3} = A \cdot 0 + B \cdot \frac{h^2}{4} + C \cdot h^2. \end{aligned} \quad (13.28)$$

Za uteži  $A$ ,  $B$  in  $C$  dobimo sistem linearnih enačb:

$$\begin{aligned} h &= A + B + C, \\ h &= B + 2C, \\ 4h &= 3B + 12C, \end{aligned} \quad (13.29)$$

ki ga v matrični obliki lahko zapišemo kot

$$\begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 2 \\ 0 & 3 & 12 \end{pmatrix} \begin{pmatrix} A \\ B \\ C \end{pmatrix} = h \begin{pmatrix} 1 \\ 1 \\ 4 \end{pmatrix} \quad (13.30)$$

in ga rešimo z Julio

```
julia> abc = [1 1 1; 0 1 2; 0 3 12] \ [1, 1, 4]
3-element Vector{Float64}:
0.1666666666666652
0.6666666666666669
0.1666666666666663
```

Vrednosti uteži so enake  $A = \frac{h}{6}$ ,  $B = \frac{2h}{3}$  in  $C = \frac{h}{6}$ , Simpsonovo pravilo pa se glasi:

$$\int_0^h f(x)dx = \frac{h}{6} \left( f(a) + 4f\left(\frac{a+b}{2}\right) + f(b) \right) + R_f. \quad (13.31)$$

Formulo za napako  $R_f$  dobimo tako, da v formulo vstavimo polinome še višjih stopenj, dokler napaka ni več enaka 0. Za  $x^3$  je napaka  $R_f$  enaka 0:

$$\int_0^h x^3 dx = \frac{h^4}{4} = \frac{2h}{3} \cdot \frac{h^3}{8} + \frac{h}{6} \cdot h^3 = \frac{1}{4}h^4, \quad (13.32)$$

za  $x^4$ , pa ne več

$$\begin{aligned} \int_0^h x^4 dx &= \frac{h^5}{5} = \frac{2h}{3} \cdot \frac{h^4}{16} + \frac{h}{6} \cdot h^4 + R_{x^4} = \frac{5h^5}{24} + R_{x^4} \\ R_{x^4} &= \frac{h^5}{5} - \frac{5h^5}{24} = -\frac{1}{120}h^5. \end{aligned} \quad (13.33)$$

Ker je  $x^4$  najnižja stopnja polinoma, pri kateri napaka vedno enaka 0, bo v formuli za napako nastopala vrednost četrtega odvoda  $f^{(4)}(\xi)$  v neznani točki  $\xi$ . Napako zapišemo kot  $R_f = Ch^5 f^{(4)}(\xi)$  in:

$$\begin{aligned} R_{x^4} &= Ch^5 (x^4)^{(4)} = Ch^5 4! = -\frac{1}{120}h^5 \Rightarrow \\ \Rightarrow C &= -\frac{1}{2880}. \end{aligned} \quad (13.34)$$

## 14 Povprečna razdalja med dvema točkama na kvadratu

V poglavju o enojnih integralih smo spoznali, da je večina kvadraturnih formul preprosta utežena vsota

$$\int_a^b f(x)dx \approx \sum w_k f(x_k), \quad (14.1)$$

kjer so uteži  $w_k$  in vozli  $x_k$  primerno izbrani, da se formula čim bolj približa pravi vrednosti.

Izračun večkratnih integralov je zapletenejši, a v principu enak. Kvadrature so tudi za večkratne integrale večinoma navadne utežene vsote vrednosti v izbranih točkah na integracijskem območju. V tej vaji si bomo ogledali produktne kvadrature za večkratne integrale.

### 14.1 Naloga

- Izpeljite algoritem, ki izračuna integral na večdimensionalnem kvadru kot večkratni integral tako, da za vsako dimenzijo uporabite isto kvadraturno formulo za enkratni integral.
- Pri implementaciji pazite, da ne delate nepotrebnih dodelitev pomnilnika.
- Uporabite algoritem za izračun povprečne razdalje med dvema točkama na enotskem kvadratu  $[0, 1]^2$  in enotski kocki  $[0, 1]^3$ .
- Za sestavljeno Simpsonovo formulo in Gauss-Legendrove kvadrature ugotovite, kako napaka pada s številom izračunov funkcije, ki jo integriramo. Primerjajte rezultate s preprosto Monte Carlo metodo (računanje vzorčnega povprečja za enostaven slučajni vzorec).

### 14.2 Dvojni integral in integral integrala

Oglejmo si najenostavnnejši primer integrala funkcije na kvadratu  $[a, b]^2$ . Dvojni integral lahko zapišemo kot dva gnezdena enojna integrala<sup>4</sup>:

$$\int \int_{[a,b]^2} f(x, y) dx dy = \int_a^b \left( \int_a^b f(x, y) dy \right) dx = \int_a^b \left( \int_a^b f(x, y) dx \right) dy. \quad (14.2)$$

Kvadrature za večkratni integral je najenostavnje izpeljati, če za vsakega od gnezdenih enojnih integralov uporabimo isto kvadraturno formulo:

$$\int_a^b f(x) dx \approx \sum_{k=1}^n w_k f(x_k) \quad (14.3)$$

z danimi utežmi  $w_1, w_2, \dots, w_n$  in vozli  $x_1, x_2, \dots, x_n$ . Če za zunanji integral uporabimo kvadrature (14.3), dobimo:

$$\int \int_{[a,b]^2} f(x, y) dx dy = \int_a^b \left( \int_a^b f(x, y) dy \right) dx = \sum w_i F_y(x_i), \quad (14.4)$$

kjer smo z  $F_y(x)$  označili integral po  $y$ . Za izračun vrednosti  $F_y(x_i)$  ponovno uporabimo kvadrature (14.3):

<sup>4</sup>Več o tem, kdaj je mogoče večkratni integral zamenjati z gnezdenimi enojnimi integrali, pove [Fubinijev izrek](#).

$$F_y(x_i) = \int_a^b f(x_i, y) dy \approx \sum w_j f(x_i, y_j). \quad (14.5)$$

Dvojni integral približno izračunamo kot dvojno vsoto:

$$\int_a^b \int_a^b f(x, y) dx dy \approx \sum_{i,j} w_i w_j f(x_i, y_j). \quad (14.6)$$

Formulo (14.6) pospolimo za integrale v več dimenzijah:

$$\int_{[a,b]^d} f(x_1, x_2, \dots, x_d) dx_1 dx_2 \dots dx_d \approx \sum_{i_1, i_2, \dots, i_d} \left( \prod_{j=1}^d w_{i_j} \right) f(x_{i_1}, x_{i_2}, \dots, x_{i_d}), \quad (14.7)$$

kjer seštevamo po vseh možnih multi indeksih  $(i_1, i_2, \dots, i_d) \in \{1, 2, \dots, n\}^d$ . S preprosto linearno funkcijo formulo (14.7) razširimo na poljuben  $d$ -dimenzionalen kvader  $[a_1, b_1] \times [a_2, b_2] \times \dots \times [a_d, b_d]$ :

$$\begin{aligned} & \int_{a_1}^{b_1} \int_{a_2}^{b_2} \dots \int_{a_n}^{b_n} f(x_1, x_2, \dots, x_d) dx_1 dx_2 \dots dx_d \approx \\ & \approx \prod_{i=1}^d \left( \frac{b_i - a_i}{b - a} \right) \sum_{i_1, i_2, \dots, i_d} \prod_{j=1}^d w_{i_j} f(t_{1i_1}, t_{1i_2}, \dots, t_{di_d}), \end{aligned} \quad (14.8)$$

kjer so  $t_{ij}$  vozli  $x_j$  preslikani na interval  $[a_i, b_i]$ . Kvadraturnim formulam (14.6), (14.7) in (14.8) pravimo *produktné formule*.

#### Produktne formule trpijo za prekletstvom dimenzionalnosti

Število vozlov, ki jih dobimo v produktni formuli, narašča eksponentno z dimenzijo prostora. Zato produktne kvadrature postanejo hitro (že v dimenzijah nad 6 ali 7) časovno tako zahtevne, da konvergirajo počasneje kot metoda Monte Carlo (Poglavlje 14.3). Pojav imenujemo **prekletstvo dimenzionalnosti** in se pojavi tudi pri drugih problemih, ko dimenzija prostora narašča.

Z dimenzijo narašča delež volumna, ki je „na robu“. Oglejmo si  $d$ -dimenzionalno enotsko kocko  $[-1, 1]^d$ . Če interval  $[-1, 1]$  razdelimo na točke v notranjosti  $[-\frac{1}{2}, \frac{1}{2}]$  in točke na robu  $[-1, -\frac{1}{2}] \cup [\frac{1}{2}, 1]$ , sta v eni dimenziji oba dela enako dolga. V višjih dimenzijah pa delež točk v kocki, ki so blizu robu, v primerjavi s točkami v notranjosti narašča. Delež točk v notranjosti lahko preprosto izračunamo:

$$P\left(\left[-\frac{1}{2}, \frac{1}{2}\right]^d\right) = \frac{1}{2^d} \quad (14.9)$$

in vidimo, da pada eksponentno z dimenzijo  $d$ . Zato je smiselno na robu uporabiti gostejšo mrežo kot v notranjosti. Tako so razvili **razpršene mreže**, ki vsaj delno omilijo prekletstvo dimenzionalnosti.

#### Samostojno delo

Definiraj naslednje tipe in funkcije:

- podatkovni tip **VečkratniIntegral** (`fun, box`), ki opiše večkratni integral na kvadru  $\prod_i [a_i, b_i]$  (Program 78),
- metodo **integriraj** (`I::VečkratniIntegral, kvad::Kvadratura`), ki izračuna večkratni integral s kvadraturno formulo (14.8) (Program 79, Program 80, Program 81).

### 14.3 Metoda Monte Carlo

Naj bo

$$X \sim U([a_1, b_1] \times [a_2, b_2] \times \dots \times [a_d, b_d]) \quad (14.10)$$

enakomerno porazdeljena slučajna spremenljivka na  $B_d = [a_1, b_1] \times [a_2, b_2] \times \dots \times [a_d, b_d]$ . Potem je pričakovana vrednost funkcije  $f(X)$  enaka večkratnemu integralu:

$$E(f(X)) = \frac{1}{V(B_d)} \int_{B_d} f(x) dx. \quad (14.11)$$

Po [centralnem limitnem izreku](#) je vzorčno povprečje za enostaven vzorec  $x_1, x_2, \dots, x_n$  porazdeljeno približno normalno:

$$\overline{f(x)} = \frac{1}{n}(f(x_1) + f(x_2) + \dots + f(x_n)) \sim N(\mu, \sigma) \quad (14.12)$$

s parametri  $\mu = E(f(X))$  in  $\sigma = \sqrt{\frac{\sigma(f(X))}{n}}$ . Razpršenost porazdelitve pada s  $\sqrt{n}$ , zato je vzorčno povprečje za velike vzorce blizu vrednosti integrala (14.11). Približno vrednost integrala (14.11) lahko ocenimo z vzorčnim povprečjem za dovolj velik vzorec:

$$\int_{B_d} f(x) dx \approx \overline{f(x)} \cdot V(B_d). \quad (14.13)$$

Metoda Monte Carlo ne konvergira zelo hitro<sup>5</sup>, ima pa prednost, da ne trpi za prekletstvom dimenzionalnosti. Zato se jo najpogosteje uporablja za računanje integralov v viših dimenzijah.

#### Samostojno delo

Definiraj naslednji tip in metodo:

- podatkovni tip `MonteCarlo(rng, n)` za parametre metode Monte Carlo in
- metodo `integriraj(integral::VeckrantiIntegral, mc::MonteCarlo)`, ki dani integral izračuna z metodo Monte Carlo (Program 83).

### 14.4 Povprečna razdalja med točkama na kvadratu $[0, 1]^2$

Povprečno razdaljo med dvema točkama na kvadratu  $[0, 1]^2$  izračunamo s štirikratnim integralom:

$$\bar{d} = \int_{[0,1]^4} \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} dx_1 dx_2 dy_1 dy_2. \quad (14.14)$$

Za izračun integrala bomo uporabili produktno kvadraturo s sestavljenim Simpsonovo formulo in metodo Monte Carlo.

<sup>5</sup>Napaka pada s  $\sqrt{n}$ , kjer je  $n$  število izračunov funkcijskih vrednosti.

Najprej definiramo funkcijo razdalje:

```
"""
Izračunaj razdaljo med dvema točkama podanimi v enem vektorju `x`.
Koordinate prve točke so podane v prvi polovici, koordinate druge točke pa v
drugi polovici komponent vektorja `x`.
"""

function razdalja(x)
 d = div(length(x), 2)
 return norm(x[1:d] - x[d+1:2d])
end
```

Nato pa še funkcijo, ki izračuna povprečno razdaljo.

```
"""Izračunaj povprečno razdaljo med dvema točkama na danem pravokotniku."""
function povprečna_razdalja(box::Vector{Interval{Float64}}, kvadratura)
 integral = VečkratniIntegral{Float64,Float64}(razdalja, vcat(box, box))
 I = integriraj(integral, kvadratura)
 return I / volumen(box)^2
end
```

Za izračun uporabimo sestavljeni Simpsonovo formulo.

```
kvadrat = [Interval(0.0, 1.0), Interval(0.0, 1.0)]
integral = VečkratniIntegral{Float64,Float64}(razdalja, vcat(kvadrat, kvadrat))
n = 15
d0 = integriraj(integral, simpson(0.0, 1.0, n))
```

```
0.5213916369440644
```

Napako ocenimo tako, da izračun ponovimo z natančnejšo kvadraturno formulo. Na primer tako, da podvojimo število vozlov v osnovni kvadraturi.

```
n = 30
d1 = integriraj(integral, simpson(0.0, 1.0, n))
napaka = d0 - d1
```

```
-1.2083201086476869e-5
```

Isti rezultat izračunajmo še z metodo Monte Carlo. Približek, ki ga dobimo z metodo Monte Carlo je odvisen od naključnega vzorca. Zato dobimo različne približke, če metodo večkrat ponovimo.

```

using Random
rng = Xoshiro(4526) # ustvarimo nov generator psevdonakljucnih stevil
mc = MonteCarlo(rng, 16^4) # uporabimo isto stevilo izracunov kot prvič
dmc = [
 integriraj(integral, mc) for i in 1:5] # vsaka ponovitev vrne nekaj drugega

5-element Vector{Float64}:
0.5201770605788114
0.5217087179128533
0.5212543535233094
0.5221131942172096
0.5208611243179776

```

Poglejmo, kako napaka pada, če povečamo število podintervalov v sestavljeni Simpsonovi formuli.

```

using Plots
nsim = 3:20
napakesim = [povprečna_razdalja(kvadrat, simpson(0.0, 1.0, i)) - d1 for i in
nsim]
scatter((2 * nsim .+ 1) .^ 4, abs.(napakesim), yscale=:log10, xscale=:log10,
label="napaka Simpson")

```

Iz rezultatov lahko ocenimo red metode. Zapišemo predoločen sistem za logaritem absolutne vrednosti napake v odvisnosti od logaritma števila funkcijskih izračunov:

$$\begin{aligned}\delta(n) &= n^{-r} \\ \log(\delta(n)) &= -r \log(n).\end{aligned}\tag{14.15}$$

Tako dobimo  $k \times 1$  predoločen sistem za parameter  $r$ :

$$\begin{pmatrix} \log(n_1) \\ \log(n_2) \\ \vdots \\ \log(n_k) \end{pmatrix} \cdot r = -\begin{pmatrix} \log(|\delta_1|) \\ \log(|\delta_2|) \\ \vdots \\ \log(|\delta_k|) \end{pmatrix},\tag{14.16}$$

kjer so  $\delta_i$  izračunane napake za  $n_i$  funkcijskih izračunov. Sistem rešimo po metodi najmanjših kvadratov z operatorjem \:

```

k = reshape(4 * log.((2 * nsim .+ 1)), length(nsim), 1) \ log.(abs.(napakesim))

1-element Vector{Float64}:
-0.8252312869463272

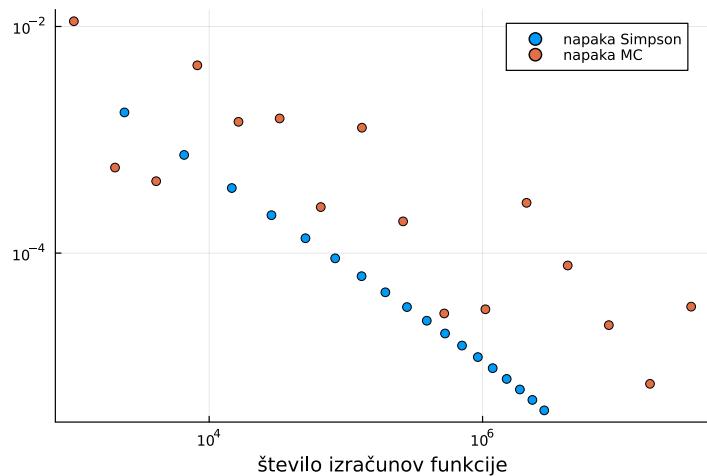
```

Vidimo, da je red produktne Simpsonove formule malo manj kot 1. Za vsako novo točno decimalko moramo število funkcijskih izračunov povečati za faktor  $10^{\frac{1}{r}}$ , kar je malo več kot 10. Poglejmo, kakšen je red metode Monte Carlo.

```

using Random
rng = Xoshiro(526)
nmc = 2 .^ (10:25)
napakamc = [povprečna_razdalja(kvadrat, MonteCarlo(rng, i)) - d1 for i in nmc]
scatter!(nmc, abs.(napakamc), yscale=:log10, xscale=:log10,
label="napaka MC", xlabel="število izračunov funkcije")

```



Slika 54: Napaka Simpsonove produktne kvadrature in metode Monte Carlo v odvisnosti od števila funkcijskih izračunov

Vidimo, da je napaka pri metodi Monte Carlo nepredvidljiva. To je posledica nepredvidljivosti vzorčenja. Kljub temu je trend jasen. Podobno kot prej lahko ocenimo red metode Monte Carlo. Centralni limitni izrek pove, da bi moral biti red približno  $\frac{1}{2}$ .

```

k = reshape(log.(nmc), length(nmc), 1) \ log.(abs.(napakamc))

1-element Vector{Float64}:
-0.6613614335692505

```

Izračunan red je nekoliko večji, a to je zgolj posledica variabilnosti, ki ga prinaša vzorčenje.

#### Kaj smo se naučili?

- Večkratni integral lahko izračunamo s produktnimi kvadraturami.
- Produktne kvadrature trpijo za prekletstvom dimenzionalnosti.
- Metoda Monte Carlo je enostavna in ne trpi za prekletstvom dimenzionalnosti, a konvergira počasi.

## 14.5 Rešitve

```
"""Podatkovni tip za večkratni integral."""
struct VečkratniIntegral{T,TIntegral}
 fun # funkcija, ki jo integriramo
 box::Vector{Interval{T}}
end

"""Vrni dimenzijo večkratnega integrala."""
dim(i::VečkratniIntegral) = length(i.box)
```

Program 78: Podatkovni tip, ki opiše večkratni integral.

```
import Vaja13: preslikaj, dolžina
"""

kvad2 = preslikaj(kvad1::Kvadratura{T}, interval::Interval{T})

Preslikaj kvadraturo `kvad1` na drug `interval`.

"""

function preslikaj(kvad::Kvadratura{T}, interval::Interval{T}) where {T}
 x = map(x -> preslikaj(x, kvad.interval, interval), kvad.x)
 u = dolžina(interval) / dolžina(kvad.interval) * kvad.u
 return Kvadratura(x, u, interval)
end
```

Program 79: Funkcija, ki preslika kvadraturo na drug interval.

```
"""

naslednji!(index, n)

Izračunanj naslednji multiindeks v zaporedju vseh multiindeksov
```\{1, 2, ... n\}``` in ga zapiši v vektor `index`.

"""

function naslednji!(index, n)
    d = length(index)
    for i = 1:d
        if index[i] < n
            index[i] += 1
            return
        else
            index[i] = 1
        end
    end
end
```

Program 80: Funkcija, ki izračuna naslednji multiindeks $(i_1, i_2, \dots, i_d) \in \{1, 2, \dots, n\}^d$.

```
"""
I = integriraj(int::VečkratniIntegral{T, TI}, kvad::Kvadratura{T})
```

Integriraj večkratni integral `int` s produktno kvadraturo, ki je podana kot produkt enodimenzionalne kvadrature `kvad`.

```
# Primer
```jldoctest
int = VečkratniIntegral{Float64, Float64}(
 x -> x[1] - x[2], [Interval(0., 2.), Interval(-1., 2.)]);
kvad = Kvadratura([0.5], [1.0], Interval(0.0, 1.0)); # sredinsko pravilo
integriraj(int, kvad)

output

3.0
```
"""

function integriraj(
    int::VečkratniIntegral{T,TI}, kvad::Kvadratura{T}) where {T,TI}
    # kvadrature preslikamo pred glavno zanko
    kvadrature = [preslikaj(kvad, interval) for interval in int.box]
    d = dim(int)
    index = ones(Int, d)
    n = length(kvad.x)
    x = zeros(T, d) # alociramo vektorja vozlov in uteži pred zanko
    w = zeros(T, d)
    I = zero(TI)
    for _ in 1:n^d
        for j in eachindex(x)
            kvad = kvadrature[j]
            x[j] = kvad.x[index[j]]
            w[j] = kvad.u[index[j]]
        end
        z::TI = int.fun(x)
        I += z * prod(w)
        naslednji!(index, n)
    end
    return I
end
```

Program 81: Funkcija, ki izračuna večkratni integral s produktno kvadraturo.

```
"""Poišči vozle in uteži za sestavljeni Simpsonovo pravilo na intervalu
`[a, b]` z delitvijo na `n` podintervalov."""
simpson(a, b, n) = Kvadratura(collect(LinRange(a, b, 2n + 1)),
    (b - a) / (6 * n) * vcat([1.0], repeat([4, 2], n - 1), [4, 1]), Interval(a, b))
```

Program 82: Funkcija, ki izračuna vozle in uteži za sestavljeni Simpsonovo pravilo.

```

"""Izračunaj volumen večrazsežne škatle `box`."""
volumen(box::Vector{Interval{T}}) where {T} = prod(dolžina.(box))

"""Podatkovna struktura za parametre metode Monte Carlo."""
struct MonteCarlo
    rng
    n::Int # število vzorcev
end

"""
I = integriraj(int::::VečkratniIntegral{T, TI}, mc::MonteCarlo)

Izračunaj približek za večkratni integral `int` z metodo Monte Carlo s parametri
`mc`.
"""

function integriraj(int::VečkratniIntegral{T, TI}, mc::MonteCarlo) where {T, TI}
    I = zero(TI)
    x = zeros(T, dim(int))
    for _ in 1:mc.n
        for i in eachindex(x)
            x[i] = preslikaj(rand(mc.rng), int.box[i], Interval(0.0, 1.0))
        end
        z::TI = int.fun(x)
        I += z
    end
    return volumen(int.box) * I / mc.n
end

```

Program 83: Funkcija, ki izračuna večkratni integral z metodo Monte Carlo.

14.6 Testi

```

@testset "Naslednji indeks" begin
    index = [1, 1, 1]
    Vaja14.naslednji!(index, 3)
    @test index == [2, 1, 1]
    index = [3, 1, 1]
    Vaja14.naslednji!(index, 3)
    @test index == [1, 2, 1]
    index = [3, 3, 3]
    Vaja14.naslednji!(index, 3)
    @test index == [1, 1, 1]
end

```

Program 84: Test izračuna naslednjega multiindeksa

```

@testset "Simpsonovo sestavljeni pravilo" begin
    k = Vaja14.simpson(1.0, 3.0, 2)
    @test (k.interval.min, k.interval.max) == (1.0, 3.0)
    @test k.x == [1.0, 1.5, 2.0, 2.5, 3.0]
    @test isapprox(k.u, [1, 4, 2, 4, 1] * 1 / 6)
end

```

Program 85: Test izračuna sestavljenega Simpsonovega pravila

```

@testset "Preslikaj kvadraturo" begin
    k = Kvadratura([1.0, 2.0, 3.0], [1.0, 2.0, 1.0], Interval(0.0, 4.0))
    k1 = Vaja14.preslikaj(k, Interval(-1.0, 1.0))
    @test k1.x ≈ [-0.5, 0, 0.5]
    @test k1.u ≈ [0.5, 1, 0.5]
end

```

Program 86: Test preslikave kvadrature na nov interval

```

@testset "Integriraj večkratni integral" begin
    kvad = Kvadratura([0.0, 1.0], [0.5, 0.5], Interval(0.0, 1.0)) # trapezna
    int = VečkratniIntegral{Float64,Float64}(x -> x[1] - x[2],
        [Interval(0.0, 2.0), Interval(-1.0, 2.0)])
    @test integriraj(int, kvad) ≈ 3.0
end

```

Program 87: Test izračuna integrala s produktno kvadraturo

15 Avtomatsko odvajanje z dualnimi števili

V grobem poznamo tri načine, kako izračunamo odvod funkcije z računalnikom:

- simbolično odvajanje,
- numerično odvajanje s končnimi diferencami in
- avtomatsko odvajanje programske kode z uporabo verižnega pravila.

Pri praktični uporabi v programiranju ima simbolično odvajanje težave pri odvajanju kompleksnih programov, saj mora program prevesti v en sam matematični izraz. Ti izrazi lahko za programe, ki vsebujejo zanke ali rekurzijo, hitro postanejo neobvladljivi. Pri numeričnem odvajanju imamo težave z zaokrožitvenimi napakami. Tako simbolično kot tudi numerično odvajanje je počasno pri računanju gradientov funkcij več spremenljivk. Avtomatsko odvajanje ne trpi za omenjenimi težavami.

V tej vaji bomo v Julii implementirali avtomatsko odvajanje z [dualnimi števili](#).

15.1 Naloga

- Definiraj podatkovni tip za dualna števila.
- Za podatkovni tip dualnega števila definiraj osnovne operacije in elementarne funkcije, kot so sin, cos in exp.
- Uporabi dualna števila in izračunaj hitrost nebesnega telesa, ki se giblje po Keplerjevi orbiti. Keplerjevo orbito izrazimo z rešitvijo [Keplerjeve enačbe](#), ki jo lahko rešiš z Newtonovo metodo.
- Pospoliši dualna števila, da je komponenta pri ε lahko vektor. Uporabi pospolena dualna števila za izračun gradienta funkcije več spremenljivk.
- Uporabi funkcijo za računanje gradienta in izračunaj gradient [Ackleyeve funkcije](#):

$$f(\mathbf{x}) = -a \exp\left(-b \sqrt{\frac{1}{d} \sum_{i=1}^d x_i^2}\right) - \exp\left(\frac{1}{d} \sum_{i=1}^d \cos(cx_i)\right) + a + \exp(1), \quad (15.1)$$

ki se uporablja za testiranje optimizacijskih algoritmov.

15.2 Ideja avtomatskega odvoda

Računalniški program ni nič drugega kot zaporedje osnovnih operacij, ki jih zaporedoma opravimo na vhodnih podatkih. Matematično lahko vsak korak programa zapišemo kot preslikavo k_i , ki stare vrednosti spremenljivk pred izvedbo koraka preslika v nove vrednosti po izvedbi koraka. Program si lahko predstavljamo kot kompozitum posameznih korakov

$$P = k_n \circ k_{n-1} \circ \dots \circ k_2 \circ k_1. \quad (15.2)$$

Pri avtomatskem odvajanju želimo program za računanje vrednosti neke funkcije spremeniti v program, ki poleg vrednosti funkcije računa tudi vrednost odvoda pri istih argumentih. Matematično lahko program ali funkcijo obravnavamo kot preslikavo med vhodnimi argumenti in izhodnimi vrednostmi:

$$P : \mathbb{R}^n \rightarrow \mathbb{R}^m \quad (15.3)$$

Vsek korak programa je prav tako preslikava $k_i : \mathbb{R}^{n_i} \rightarrow \mathbb{R}^{m_i}$. Če uporabimo verižno pravilo, je odvod programa P z danimi argumenti \mathbf{x} enak produktu:

$$DP(\mathbf{x}) = Dk_n(\mathbf{x}_{n-1}) \cdot Dk_{n-1}(\mathbf{x}_{n-2}) \cdots Dk_1(\mathbf{x}_0), \quad (15.4)$$

kjer je $x_i = k_i(k_{i-1} \dots k_2(k_1(x)) \dots)$ stanje lokalnih spremenljivk po tem, ko se je izvedel i -ti korak. Oglejmo si preprost primer funkcije, ki z Newtonovo metodo izračuna kvadratni koren.

```
"""
y = koren(x)

Izračunaj vrednost korenske funkcije za argument `x`.

function koren(x)
    y = 1 + (x - 1) / 2
    for i = 1:5
        y = (y + x / y) / 2
    end
    return y
end
```

V programu moramo odvajati vsako vrstico kode posebej. Poglejmo prvo vrstico funkcije `koren`:

```
y = 1 + (x-1)/2
```

Nova lokalna spremenljivka y je funkcija x . Njen odvod je

$$y'(x) = \left(1 + \frac{x-1}{2}\right)' = \frac{1}{2}. \quad (15.5)$$

V zanki nato ponavljamo

```
y = (y - x/y)/2
```

Označimo z y_i vrednost spremenljivke y na i -tem koraku zanke. Vrednost y_i je odvisna od vrednosti x , za njen izračun pa potrebujemo tudi vrednost y na prejšnjem koraku $y_{i-1}(x)$. Vrstico programa zapišemo kot rekurzivno enačbo:

$$y_i(x) = \frac{1}{2} \left(y_{i-1}(x) - \frac{x}{y_{i-1}(x)} \right). \quad (15.6)$$

Če rekurzivno enačbo odvajamo, dobimo:

$$y'_i(x) = \frac{1}{2} \left(y'_{i-1}(x) - \frac{1}{y_{i-1}(x)} - \frac{x y'_{i-1}(x)}{y_{i-1}(x)^2} \right). \quad (15.7)$$

Program `koren` dopolnimo, da hkrati računa vrednosti funkcije in vrednosti odvoda. To storimo tako, da na vsakem koraku posodobimo vrednosti odvodov spremenljivk, ki jih na tem koraku posodobimo.

```

y, dy = dkoren(x)

Izračunaj vrednost korenske funkcije in njenega odvoda za argument `x`.

function dkoren(x)
    y = 1 + (x - 1) / 2
    dy = 1 / 2
    for i = 1:5
        y = (y + x / y) / 2
        dy = (dy + 1 / y - x / y^2 * dy) / 2
    end
    return y, dy
end

```

Preverimo, če funkcija deluje. Odvod korenske funkcije je enak $(\sqrt{x})' = \frac{1}{2\sqrt{x}}$.

```

julia> y, dy = dkoren(2)
(1.414213562373095, 0.35355339059327373)

julia> napaka = dy - 1 / (2sqrt(2))
0.0

```

15.3 Dualna števila

Dualna števila so števila oblike $a + b\varepsilon$, kjer sta $a, b \in \mathbb{R}$, medtem ko je dualna enota ε neničelno število, katerega kvadrat je nič: $\varepsilon \neq 0$ in $\varepsilon^2 = 0$. Podobno kot dobimo kompleksna števila, če realna števila razširimo z imaginarno enoto $i = \sqrt{-1}$, dobimo dualna števila, če realna števila razširimo z dualno enoto ε .

Z dualnimi števili računamo kot z navadnimi binomi, pri čemer upoštevamo, da je $\varepsilon^2 = 0$. Pri vsoti dveh dualnih števil se realna in dualna komponenta seštejeta:

$$(a + b\varepsilon) + (c + d\varepsilon) = (a + c) + (b + d)\varepsilon. \quad (15.8)$$

Pri izpeljavi pravila za produkt moramo upoštevati lastnost $\varepsilon^2 = 0$ in komutativnost produkta z ε :

$$(a + b\varepsilon)(c + d\varepsilon) = ac + ad\varepsilon + bc\varepsilon + bd\varepsilon^2 = ac + (ad + bc)\varepsilon. \quad (15.9)$$

Pravilo za deljenje oziroma inverz dobimo tako, da število pomnožimo z ulomkom $1 = \frac{a-b\varepsilon}{a-b\varepsilon}$:

$$\frac{1}{a+b\varepsilon} = \frac{a-b\varepsilon}{(a+b\varepsilon)(a-b\varepsilon)} = \frac{a-b\varepsilon}{a^2+b^2\varepsilon^2} = \frac{1}{a} - \frac{b}{a^2}\varepsilon. \quad (15.10)$$

Pri izpeljavi pravila za potenciranja si pomagamo z razvojem v binomsko vrsto:

$$(a + b\varepsilon)^n = a^n + \binom{n}{n-1} a^{n-1} b\varepsilon + \binom{n}{n-2} a^{n-2} b^2 \varepsilon^2 + \dots = a^n + n a^{n-1} b\varepsilon. \quad (15.11)$$

Za racionalne potence uporabimo binomsko vrsto, če pa ε nastopa v eksponentu, uporabimo vrsto za e^x .

Dualna števila lahko uporabimo za računanje odvodov. Z dualnimi števili se namreč računa podobno kot z diferenciali, oziroma linearnim delom Taylorjeve vrste. Linearni del Taylorjeve vrste imenujemo

tudi **1-tok**. Množica 1-tokov v neki točki predstavlja vse možne tangente na vse možne funkcije, ki gredo skozi to točko. V točki x_0 lahko 1-tok funkcije f zapišemo kot:

$$f(x_0) + f'(x_0)dx, \quad (15.12)$$

kjer je $dx = x - x_0$ diferencial neodvisne spremenljivke. Poglejmo si primer 1-toka za produkt dveh funkcij f in g :

$$\begin{aligned} & (f(x_0) + f'(x_0)dx)(g(x_0) + g'(x_0)dx) = \\ & = f(x_0)g(x_0) + (f(x_0)g'(x_0) + f'(x_0)g(x_0))dx + \mathcal{O}(dx^2). \end{aligned} \quad (15.13)$$

Vse potence dx^k za $k \geq 2$ potisnemo v ostanek $\mathcal{O}(dx^2)$ in v limiti zanemarimo. Pravila računanja 1-tokov in dualnih števil so povsem enaka. Pri računanju z diferenciali ravno tako upoštevamo, da je $dx^2 \approx 0$ in vse potence dx^k za $k \geq 2$ zanemarimo. Vrednosti odvoda v neki točki lahko izračunamo z dualnimi števili. Če poznamo vrednost funkcije in vrednost odvoda funkcije v neki točki, z dualnimi števili izračunamo vrednosti odvodov različnih operacij. Z dualnimi števili lahko predstavimo 1-tokove. Če sta f in g funkciji, potem dualni števili

$$f(x_0) + f'(x_0)\varepsilon \quad \text{in} \quad g(x_0) + g'(x_0)\varepsilon \quad (15.14)$$

predstavljata 1-tokova za funkciji f in g v točki x_0 . Če dualni števili vstavimo v nek izraz, npr. x^2y , dobimo 1-tok funkcije $f(x)^2g(x)$ in s tem tudi vrednost odvoda v točki x_0 .

Izračunajmo odvod $f(x)^2g(x)$ v točki $x_0 = 1$ za funkciji $f(x) = x^2$ in $g(x) = 2 - x$. Dualno število za 1-tok za f je:

$$f(1) + f'(1)\varepsilon = 1 + 2\varepsilon, \quad (15.15)$$

dualno število za 1-tok za g pa:

$$g(1) + g'(1)\varepsilon = 1 - \varepsilon. \quad (15.16)$$

Dualni števili vstavimo v izraz x^2y in upoštevamo $\varepsilon^2 = 0$:

$$\begin{aligned} (1 + 2\varepsilon)^2(1 - \varepsilon) &= (1 + 4\varepsilon + 4\varepsilon^2)(1 - \varepsilon) = (1 + 4\varepsilon)(1 - \varepsilon) \\ &= 1 + 4\varepsilon - \varepsilon - 4\varepsilon^2 = 1 + 3\varepsilon. \end{aligned} \quad (15.17)$$

Od tod lahko razberemo, da je 1-tok za (f^2g) v točki 1 enak:

$$(f^2g)(1) + (f^2g)'(1)\varepsilon = 1 + 3\varepsilon \quad (15.18)$$

in odvod $(f^2g)'(1) = 3$.

Samostojno delo

Definiraj podatkovni tip `DualNumber`, ki predstavlja dualno število, nato pa še osnovne računske operacije za ta tip in elementarne funkcije `sin`, `cos`, `exp` in `log` (Program 88, Program 89, Program 90).

Napiši funkcijo `odvod(f, x)`, ki izračuna vrednost funkcije f in njenega odvoda v točki x (Program 91).

15.4 Keplerjeva enačba

Keplerjeva enačba

$$M = E - e \sin(E) \quad (15.19)$$

določa ekscentrično anomalijo za telo, ki se giblje po Keplerjevi orbiti v odvisnosti od ekscentričnosti orbite e in povprečne anomalije M .

Keplerjevo orbito lahko izračunamo, če poznamo $E(t)$:

$$\begin{aligned} x(t) &= a(\cos(E(t)) - e), \\ y(t) &= b \sin(E(t)), \end{aligned} \quad (15.20)$$

kjer sta a in b polosi elipse ($a \leq b$). Elipsa je premaknjena tako, da je eno od gorišč v točki $(0, 0)$. Ekscentričnost e je odvisna od razmerja polosi:

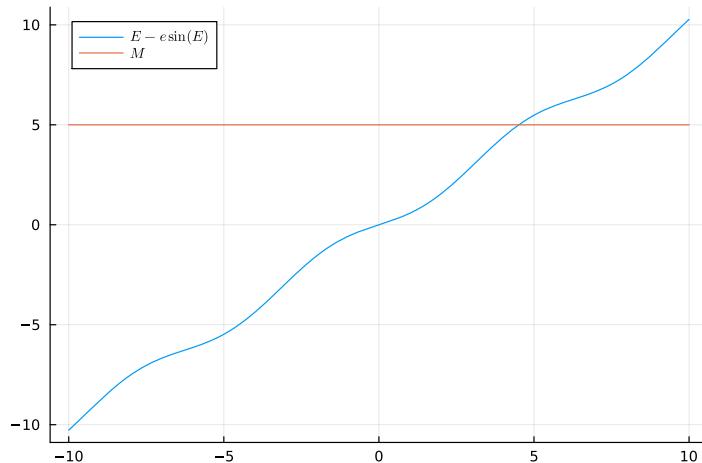
$$e = \sqrt{1 - \frac{b^2}{a^2}} \in [0, 1]. \quad (15.21)$$

Vrednost $E(t)$ izračunamo iz Keplerjeve enačbe, saj velja drugi Keplerjev zakon, ki pravi, da se povprečna anomalija $M(t)$ spreminja enakomerno:

$$M(t) = n(t - t_0). \quad (15.22)$$

Keplerjeva enačba ima eno samo rešitev. Res, funkcija $f(E) = E - e \sin(E)$ je naraščajoča in surjektivna na $(-\infty, \infty)$, saj je odvod $f'(E) = 1 - e \cos(E) \geq 0$ vedno nenegativnen. Keplerjevo enačbo (15.19) predstavimo grafično:

```
using Plots
e = 0.5
plot(E -> E - e * sin(E), -10, 10, label="\$E - e\sin(E)\$")
M = 5
plot!(x -> M, -10, 10, label="\$M\$")
```



Slika 55: Leva in desna stran Keplerjeve enačbe za $M = 5$ in $e = 0.5$

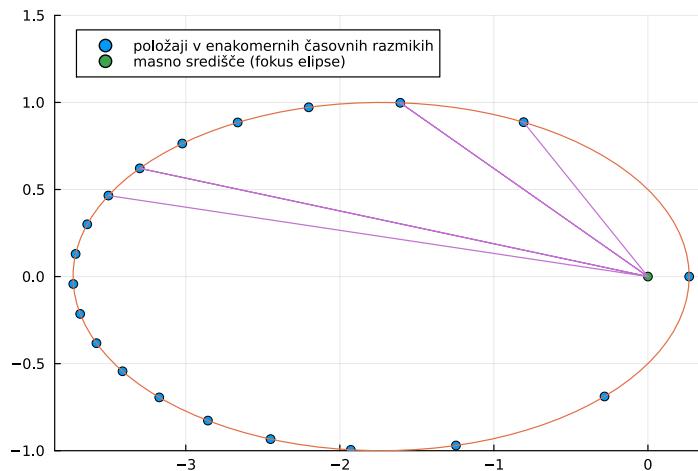
Vidimo, da je rešitev Keplerjeve enačbe blizu vrednosti M , saj je $e \sin(E)$ relativno majhen v primerjavi z M . Zato je vrednost M dober začetni približek za rešitev enačbe (15.19). Rešitev hitro poiščemo z Newtonovo metodo za funkcijo:

$$g(E) = M - E + e \sin(E). \quad (15.23)$$

Samostojno delo

Napiši naslednji funkciji:

- `keplerE(M, e)`, ki izračuna vrednost ekscentrične anomalije za dane vrednosti ekscentričnosti e in povprečne anomalije M (Program 93) in
- `orbita(t, a, b, n)`, ki izračuna položaj orbite v času t , če je v času $t = 0$ telo najbližje masnemu središču (Program 92).



Slika 56: Položaji telesa na orbiti v enakomernih časovnih razmikih. Drugi Keplerjev zakon pravi, da zveznica med telesom in masnim središčem v enakem času pokrije enako ploščino.

Hitrost telesa v določenem trenutku lahko izračunamo z avtomatskim odvodom. Najprej definiramo dualno število $t + \varepsilon$ za vrednost t , v kateri bi radi izračunali hitrost. Nato v funkcijo `orbita` vstavimo $t + \varepsilon$ in dobimo koordinate, podane z dualnimi števili.

```
julia> t0 = t[5] + ε
1.431578947368421 + 1.0ε

julia> o0 = orbita(t0, a, b, n)
(-2.6629236468300297      +      -1.1354638659453598ε,      0.8850813178920173      +
 -0.29855236219660986ε)
```

Dualni del koordinat je enak vektorju hitrosti:

```
julia> o0 = orbita(t0, a, b, n)
(-1.1354638659453598, -0.29855236219660986)
```

Avtomatsko odvajanje kontrolnih struktur

Kontrolne strukture, kakršen je `if` stavek, lahko povzročijo razvejitve v programu in funkcija, ki jo program izračuna, ni nujno zvezno odvedljiva. Če avtomatsko odvajamo tak program, dobimo odvod tiste veje, ki se za dane vhodne vrednosti izvede. Za večino vhodnih vrednosti se ta ujema z dejanskim odvodom. Za mejne vrednosti, pri katerih se program razveji, pa dobimo bodisi levi bodisi desnji odvod. Odvisno od tega, katera veja se izvede. Poglejmo si primer implementacije funkcije $f(x) = |x|$:

```
function f(x)
  if x>=0
    return x
  else
    return -x
  end
end
```

Funkcija je zvezno odvedljiva povsod, razen v točki 0. Če v funkcijo vstavimo $x = 0$ oziroma dualno število $0 + \varepsilon$, se izvede prva veja. Izračunani odvod je enak 1 in to je enako desnemu odvodu funkcije $|x|$ v točki 0.

Tabeliranih funkcij ne moremo avtomatsko odvajati

Avtomatsko odvajanje deluje le za funkcije, ki so implementirane s programom. Če je funkcija podana s tabelo funkcijskih vrednosti, avtomatsko odvajanje ni mogoče. V tem primeru uporabimo končne difference.

15.5 Računanje gradientov

Parcialne odvode funkcije več spremenljivk lahko prav tako izračunamo z dualnimi števili. Pri tem moramo paziti, da za odvode po različnih spremenljivkah uporabimo neodvisne dualne enote. Poglejmo si primer funkcije dveh spremenljivk:

$$f(x, y) = x^2y + y^3 \sin(x). \quad (15.24)$$

Ker imamo dva neodvisna parcialna odvoda $\frac{\partial}{\partial x}$ in $\frac{\partial}{\partial y}$, potrebujemo dve neodvisni dualni enoti ε_x in ε_y , ki zadoščata pogoju:

$$\varepsilon_x^2 = 0, \quad \varepsilon_y^2 = 0 \quad \text{in } \varepsilon_x \varepsilon_y = 0. \quad (15.25)$$

V funkcijo f vstavimo $x + \varepsilon_x$ in $y + \varepsilon_y$. Dobimo 1-tok funkcije $f(x, y)$:

$$\begin{aligned} f(x + \varepsilon_x, y + \varepsilon_y) &= (x + \varepsilon_x)^2(y + \varepsilon_y) + (y + \varepsilon_y)^3 \sin(x + \varepsilon_x) = \\ &= (x^2 + 2x\varepsilon_x)(y + \varepsilon_y) + (y^3 + 3y^2\varepsilon_y)(\sin(x) + \cos(x)\varepsilon_x) = \\ &= x^2y + y^3 \sin(x) + 2xy\varepsilon_x + x^2\varepsilon_y + 3y^2 \sin(x)\varepsilon_y + y^3 \cos(x)\varepsilon_x = \\ &= f(x, y) + (2xy + y^3 \cos(x))\varepsilon_x + (x^2 + 3y^2 \sin(x))\varepsilon_y. \end{aligned} \quad (15.26)$$

Vidimo, da sta koeficienta pri ε_x in ε_y ravno parcialna odvoda:

$$\begin{aligned}\frac{\partial f}{\partial x}(x, y) &= 2xy + y^3 \cos(x), \\ \frac{\partial f}{\partial y}(x, y) &= x^2 + 3y^2 \sin(x).\end{aligned}\tag{15.27}$$

Za lažjo implementacijo koeficiente pri ε_x in ε_y postavimo v vektor in zapišemo:

$$\varepsilon_x = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \varepsilon, \quad \varepsilon_y = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \varepsilon.\tag{15.28}$$

Za funkcijo n spremenljivk $f(x_1, x_2, \dots, x_n)$ dobimo:

$$\varepsilon_1 = \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \varepsilon, \quad \varepsilon_2 = \begin{pmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{pmatrix} \varepsilon, \quad \dots \quad \varepsilon_n = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 1 \end{pmatrix} \varepsilon\tag{15.29}$$

in

$$f(x_1 + \varepsilon_1, x_2 + \varepsilon_2, \dots, x_n + \varepsilon_n) = f(x_1, x_2, \dots, x_n) + \nabla f(x_1, x_2, \dots, x_n) \varepsilon.\tag{15.30}$$

Samostojno delo

Definiraj vektorska dualna števila tipa **Dual**, ki opisujejo elemente oblike

$$a + \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix} \varepsilon, \quad a, b_1, b_2, \dots, b_n \in \mathbb{R}.\tag{15.31}$$

Podobno kot pri navadnih dualnih številih nato definiraj osnovne računske operacije in nekatere elementarne funkcije (Program 94, Program 95 in Program 96).

Definiraj funkcijo **gradient(f, x)**, ki izračuna vrednost funkcije f in njenega gradiента v točki x (Program 97).

Odvajanje naprej, odvajanje nazaj

Z dualnimi števili učinkovito računamo odvode in gradiante funkcij, ki so implementirane s programom. To metodo lahko posplošimo tudi na računaje Jacobijevih matrik in višjih odvodov. V osnovi metoda temelji na verižnem pravilu in zapisu programa kot kompozituma posameznih korakov. Formula (15.4) predstavlja odvod kot produkt matrik:

$$DP(\mathbf{x}) = Dk_n(\mathbf{x}_{n-1}) \cdot Dk_{n-1}(\mathbf{x}_{n-2}) \cdots Dk_1(\mathbf{x}). \quad (15.32)$$

Matrike $Dk_i(\mathbf{x}_{i-1})$ so lahko različnih dimenzij in vrstni red množenja posameznih faktorjev vpliva na velikost spomina, ki ga potrebujemo.

V praksi sta se uveljavila dva načina računanja produkta. Produkt (15.32) lahko računamo sproti, tako da zmnožimo matrike takoj, ko so na voljo. To pomeni, da množimo z desne. Če metoda deluje na ta način, pravimo, da uporablja *način odvajanje naprej* (angl. *forward mode*).

Drug razred metod, ki ga imenujemo *način odvajanja nazaj* (angl. *backward mode*), množi matrike v produktu (15.32) z leve. To pomeni, da odvodov ni mogoče izračunati, dokler izračun vrednosti funkcije ni končan. Pri odvajanju nazaj si mora program zapomniti vmesne vrednosti \mathbf{x}_k do konca izračuna.

Metode računanja nazaj so navadno implementirane kot transformacija izvirne kode, medtem ko metode računanja naprej navadno implementiramo z definicijo novih podatkovnih tipov ali razredov.

Računanje z dualnimi števili uporablja način odvajanja naprej.

15.6 Gradient Ackleyeve funkcije

Izračunajmo gradient funkcije Ackleyeve funkcije:

$$f(\mathbf{x}) = -a \exp\left(-b \sqrt{\frac{1}{d} \sum_{i=1}^d x_i^2}\right) - \exp\left(\frac{1}{d} \sum_{i=1}^d \cos(cx_i)\right) + a + \exp(1), \quad (15.33)$$

ki se uporablja za testiranje optimizacijskih algoritmov.

Samostojno delo

Najprej napiši funkcijo, ki izračuna vrednost Ackleyeve funkcije (Program 98).

Gradient izračunamo tako, da za vhodni vektor \mathbf{x}_0 ustvarimo vektor dualnih vektorskih števil in ga vstavimo v funkcijo. Dimenzija dualnega dela (vektorja ob ε) je enaka dimenziji vhodnega vektorja \mathbf{x}_0 .

```

julia> x0 = [1.1, 1.2, 1.3]
julia> x0 = Vaja15.spremenljivka(x0)
3-element Vector{Dual}:
Dual{Float64}(1.1, [1.0, 0.0, 0.0])
Dual{Float64}(1.2, [0.0, 1.0, 0.0])
Dual{Float64}(1.3, [0.0, 0.0, 1.0])

julia> y = ackley(x0, 4, 5, 6)
Dual{Float64}(4.9967587772563045,      [1.0814632936346151,      2.733373180740012,
3.436117175843327])

julia> dy = odvod(y)
3-element Vector{Float64}:
1.0814632936346151
2.733373180740012
3.436117175843327

```

Dualni del rezultata je enak gradientu funkcije.

Knjižnice za avtomatsko odvajanje

Knjižnice za avtomatsko odvajanje obstajajo za večino programskih jezikov, ki se uporabljajo za numerične izračune. V Julii so poleg `ForwardDiff` na voljo še drugi paketi. Paketi so zbrani v neformalni organizaciji [JuliaDiff](#).

Kaj smo se naučili?

- Odvode programov učinkovito računamo z avtomatskim odvajanjem.
- Obstajata dva načina avtomatskega odvajanja: *način odvajanja naprej* in *način odvajanja nazaj*.
- Odvajanje naprej lahko implementiramo z dualnimi števili.

15.7 Rešitve

```
"""
DualNumber(x, dx)

Predstavlja skalarno spremenljivko x in njen diferencial dx.

"""
struct DualNumber <: Number
    x::Float64
    dx::Float64
end

import Base: show, promote_rule, convert
"""Lepo izpiši dualno število `DualNumber(a, b)` kot `a + bε`."""
show(io::IO, a::DualNumber) = print(io, a.x, " + ", a.dx, "ε")

"""Dualna enota."""
ε = DualNumber(0, 1)
"Spremeni realno število v vrednost tipa `DualNumber`."
convert(::Type{DualNumber}, x::Real) = DualNumber(x, zero(x))
"Navadna števila avtomatsko promoviraj v dualna števila."
promote_rule(::Type{DualNumber}, ::Type{<:Number}) = DualNumber
```

Program 88: Podatkovni tip za dualna števila

```
import Base: +, -, *, /, ^
*(a::DualNumber, b::DualNumber) = DualNumber(a.x * b.x, a.x * b.dx + a.dx * b.x)
+(a::DualNumber, b::DualNumber) = DualNumber(a.x + b.x, a.dx + b.dx)
-(a::DualNumber) = DualNumber(-a.x, -a.dx)
-(a::DualNumber, b::DualNumber) = DualNumber(a.x - b.x, a.dx - b.dx)
/(a::DualNumber, b::DualNumber) = DualNumber(a.x / b.x,
    (b.x * a.dx - a.x * b.dx) / b.x^2)
^(a::DualNumber, b::Number) = DualNumber(a.x^b, b * a.x^(b - 1) * a.dx)
^(a::Number, b::DualNumber) = DualNumber(a^b.x, log(a)a^b.x * b.dx)
```

Program 89: Osnovne operacije med dualnimi števili

```
import Base: sin, cos, exp, log, abs, isless, sqrt
sqrt(a::DualNumber) = DualNumber(sqrt(a.x), 0.5 / sqrt(a.x) * a.dx)
sin(a::DualNumber) = DualNumber(sin(a.x), cos(a.x) * a.dx)
cos(a::DualNumber) = DualNumber(cos(a.x), -sin(a.x) * a.dx)
exp(a::DualNumber) = DualNumber(exp(a.x), exp(a.x) * a.dx)
log(a::DualNumber) = DualNumber(log(a.x), a.dx / a.x)
abs(a::DualNumber) = DualNumber(abs(a.x), sign(a.x) * a.dx)
isless(a::DualNumber, b::DualNumber) = isless(a.x, b.x)
isless(a::DualNumber, b::Number) = isless(a.x, b)
```

Program 90: Elementarne funkcije za dualna števila

```

odvod(a::DualNumber) = a.dx
vrednost(a::DualNumber) = a.x

"""
y = odvod(f<:Function, x)

Izračuna vrednost odvoda funkcije `f` v točki `x`.

odvod(f, x) = odvod(f(x + ε))

```

Program 91: Funkcija, ki izračuna odvod funkcije ene spremenljivke.

```

"""
Izračunaj x in y koordinati na Keplerjevi orbiti.

function orbita(t, a, b, n)
    M = n * t
    e = sqrt(1 - b^2 / a^2)
    E = keplerE(M, e)
    return (a * (cos(E) - e), b * sin(E))
end

```

Program 92: Funkcija, ki izračuna položaj na orbiti v danem trenutku.

```

"""
E = keplerE(M, e)

Izračunaj ekscentrično anomalijo Keplerjeve orbite z ekscentričnostjo `e`
za povprečno anomalijo `M`.

function keplerE(M, e, maxit=10, atol=1e-10)
    g(E) = M - E + e * sin(E)
    dg(E) = -1 + e * cos(E)
    E0 = M
    for i in 1:maxit
        E = E0 - g(E0) / dg(E0)
        if abs(E - E0) < atol
            @info "Newtonova metoda konvergira po $i korakih."
            return E
        end
        E0 = E
    end
    throw("Iteracija ne konvergira po $maxit korakih")
end

```

Program 93: Funkcija za izračun ekscentrične anomalije E

```

"""
Dual(x, dx::Vector)

Predstavlja vrednost `f`, odvisno od `n` spremenljivk, in njen gradient
``\nabla f = f_1 \frac{\partial}{\partial x_1} + f_2 \frac{\partial}{\partial x_2} + \dots f_n \frac{\partial}{\partial x_n}``.
"""

struct Dual{T} <: Number
    x::T
    dx::Vector{T}
end

odvod(y::Dual) = y.dx
vrednost(y::Dual) = y.x

```

Program 94: Podatkovni tip za mešanico dualnih števil

```

*(a::Dual, b::Dual) = Dual(a.x * b.x, a.x * b.dx + a.dx * b.x)
*(a::Number, b::Dual) = Dual(a * b.x, a * b.dx)
*(a::Dual, b::Number) = Dual(a.x * b, a.dx * b)
+(a::Dual, b::Dual) = Dual(a.x + b.x, a.dx + b.dx)
+(a::Number, b::Dual) = Dual(a + b.x, b.dx)
+(a::Dual, b::Number) = Dual(a.x + b, a.dx)
-(a::Dual) = Dual(-a.x, -a.dx)
-(a::Dual, b::Dual) = Dual(a.x - b.x, a.dx - b.dx)
-(a::Number, b::Dual) = Dual(a - b.x, -b.dx)
-(a::Dual, b::Number) = Dual(a.x - b, a.dx)
^(a::Dual, b::Number) = Dual(a.x .^ b, b * a.x^(b - 1) * a.dx)
^(a::Number, b::Dual) = Dual(a .^ b.x, log(a) * a^b.x * b.dx)
/(a::Dual, b::Dual) = Dual(a.x / b.x, (1 / b.x^2) * (b.x * a.dx - a.x * b.dx))
/(a::Number, b::Dual) = Dual(a / b.x, (-a * b.dx / b.x^2))
/(a::Dual, b::Number) = Dual(a.x / b, (1 / b) * a.dx)

```

Program 95: Osnovne operacije med vektorskimi dualnimi števili

```

sqrt(a::Dual{T}) where {T} = Dual(sqrt(a.x), (0.5 / sqrt(a.x)) * a.dx)
sin(a::Dual) = Dual(sin(a.x), cos(a.x) * a.dx)
cos(a::Dual) = Dual(cos(a.x), -sin(a.x) * a.dx)
exp(a::Dual) = Dual(exp(a.x), exp(a.x) * a.dx)
log(a::Dual) = Dual(log(a.x), a.dx / a.x)

```

Program 96: Elementarne funkcije za vektorska dualna števila

```

"""
x = spremenljivka(v)

Generiraj vektor `x` vektorskih dualnih števil za vrednost vektorske
spremenljivke `v`. Komponente vektorja `x` so vektorska dualna števila
z vrednostmi, enakimi komponentam vektorja `v`, in parcialnimi odvodi enakimi
1, če se indeksa komponente in parcialnega odvoda ujemata, in 0 sicer.
Rezultat te funkcije lahko uporabimo kot vhodni argument pri računanju
gradienta vektorske funkcije z argumentom `v`.
"""

function spremenljivka(v::Vector{T}) where {T}
    n = length(v)
    x = Vector{Dual}()
    for i = 1:n
        xi = Dual(convert(Float64, v[i]), zeros(Float64, n))
        xi.dx[i] = 1
        push!(x, xi)
    end
    return x
end

"""

g = gradient(f, x)

Izračunaj gradient vektorske funkcije `f` v danem vektorju `x`.

# Primer
```jl
f(x) = x[1]*x[2] + x[2]*sin(x[1])
gradient(f, [1., 2.])
```
"""

gradient(f, x::Vector) = odvod(f(spremenljivka(x)))

```

Program 97: Funkcija, ki izračuna gradient funkcije vektorske spremenljivke v dani točki.

```

"""

Izračunaj vrednost Ackleyeve funkcije za argument `x`
s parametri `a`, `b` in `c`.

"""

function ackley(x, a, b, c)
    d = length(x)
    S1 = 0.0
    S2 = 0.0
    for xi in x
        S1 += xi * xi
        S2 += cos(c * xi)
    end
    y = a + MathConstants.e
    y -= a * exp(-b * sqrt(S1 / d))
    y -= exp(S2 / d)
    return y
end

```

Program 98: Funkcija za izračun Ackleyeve funkcije

16 Začetni problem za navadne diferencialne enačbe

Navadna diferencialna enačba (NDE) prvega reda je funkcionalna enačba za neznano funkcijo u , v kateri nastopa prvi odvod u' . Obravnavali bomo le enačbe, ki jih lahko zapišemo v obliki:

$$u'(t) = f(t, u(t), p), \quad (16.1)$$

kjer so t neodvisna spremenljivka, u odvisna spremenljivka, p parametri enačbe in f funkcija, ki izrazi odvod v odvisnosti od t , u in p . Enačba (16.1) ima enolično rešitev za vsak začetni pogoj $u(t_0) = u_0$. Iskanje rešitve NDE z danim začetnim pogojem imenujemo **začetni problem**.

V tej vaji bomo napisali knjižnico za reševanje začetnega problema za NDE.

16.1 Naloga

1. Definiraj podatkovno strukturo, ki hrani podatke o začetnem problemu za NDE.
2. Definiraj podatkovno strukturo, ki hrani podatke o rešitvi začetnega problema.
3. Implementiraj funkcijo `resi`, ki poiščejo približek za rešitev začetnega problema z različnimi metodami:
 - z Eulerjevo metodo,
 - z metodo Runge-Kutta reda 2,
 - z metodo Runge-Kutta reda 4.
4. Napiši funkcijo `vrednost`, ki za dano rešitev začetnega problema izračuna vrednost rešitve v vmesnih točkah s Hermitovim kubičnim zlepkom (Poglavlje 12).

Napisane funkcije uporabi za obravnavo poševnega meta z upoštevanjem zračnega upora. Iz Newtonovih zakonov gibanja in kvadratnega zakona upora zapiši sistem NDE. Kako daleč leti telo, preden pade na tla? Koliko časa leti?

Za vse tri metode oceni, kako se napaka spreminja v odvisnosti od dolžine koraka. Namesto točne rešitve uporabi približek, ki ga izračunaš s polovičnim korakom.

16.2 Reševanje enačbe z eno spremenljivko

Poglejmo, kako numerično poiščemo rešitev diferencialne enačbe z eno spremenljivko. Reševali bomo le enačbe, pri katerih lahko odvod eksplisitno izrazimo in jih zapišemo v obliki:

$$u'(t) = f(t, u(t)) \quad (16.2)$$

za neko funkcijo dveh spremenljivk $f(t, u)$. Funkcija f določa odvod u' in s tem tangento na rešitev u v točki $(t, u(t))$. Za vsako točko (t, u) dobimo tangento oziroma smer, v kateri se rešitev premakne. Funkcija f tako določa smerno polje v ravnini (t, u) .

Za primer vzemimo enačbo:

$$u'(t) = -2tu(t), \quad (16.3)$$

ki jo znamo tudi analitično rešiti in ima splošno rešitev:

$$u(t) = Ce^{-t^2}, \quad C \in \mathbb{R}. \quad (16.4)$$

Poglejmo, kako je videti smerno polje za enačbo (16.3). Tangente vzorčimo na pravokotni mreži na pravokotniku $(t, u) \in [-2, 2] \times [0, 4]$. Za eksplisitno podano krivuljo $u = u(t)$ je vektor v smeri

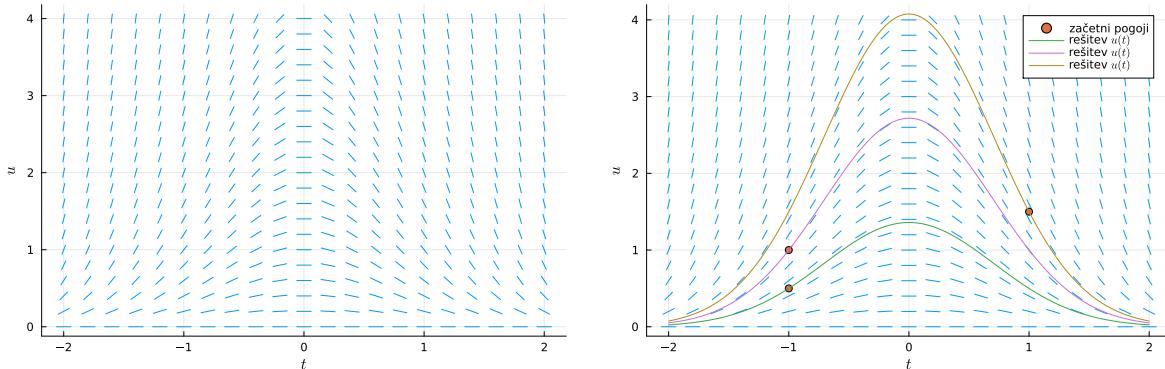
tangente podan s koordinatami $[1, u'(t)]$. Za nazornejšo sliko, vektor v smeri tangente normaliziramo in pomnožimo s primernim faktorjem, da se tangente na sliki ne prekrivajo. Uporabimo funkcijo `riši_polje(fun, (t0, tk), (u0, uk), n)`, ki nariše polje tangent za diferencialno enačbo (glej Program 99 in Program 100).

Narišimo polje smeri za enačbo (16.3):

```
using Plots
fun(t, u) = -2 * t * u
plt = riši_polje(fun, (-2, 2), (0, 4))
```

Narišimo še nekaj začetnih pogojev in rešitev, ki gredo skoznje.

```
t0 = [-1, -1, 1]
u0 = [0.5, 1, 1.5]
scatter!(plt, t0, u0, label="začetni pogoji")
for i in 1:3
    C = u0[i] / exp(-t0[i]^2)
    plot!(plt, t -> C * exp(-t^2), -2, 2, label="rešitev \$u(t)\$")
end
plt
```



Slika 57: Smerno polje za enačbo $u'(t) = -2tu(t)$. V vsaki točki ravnine (t, u) enačba določa odvod $u'(t) = f(t, u)$ in s tem tudi tangento na rešitev enačbe $u(t)$ (levo). Vsaka točka (t_0, u_0) v ravnini t, u določa začetni pogoj. Za različne začetne pogoje dobimo različne rešitve. Rešitev NDE se v vsaki točki dotika tangente, določene z $u'(t) = f(t, u)$ (desno).

Diferencialna enačba (16.3) ima neskončno rešitev. Čim pa določimo vrednost v neki točki $u(t_0) = u_0$, ima enačba (16.3) enolično rešitev u . Pogoj $u(t_0) = u_0$ imenujemo *začetni pogoj*, diferencialno enačbo z začetnim pogojem:

$$\begin{aligned} u'(t) &= f(t, u), \\ u(t_0) &= u_0, \end{aligned} \tag{16.5}$$

pa *začetni problem*.

Rešitev začetnega problema (16.5) je funkcija $u(t)$. Funkcijo u lahko numerično opišemo na mnogo različnih načinov. Dva načina, ki se pogosto uporablja, sta:

- z vektorjem vrednosti $[u_0, u_1, \dots, u_n]$ v danih točkah t_0, t_1, \dots, t_n ali
- z vektorjem koeficientov $[a_0, a_1, \dots, a_n]$ v razvoju $u(t) = \sum a_k \varphi_k(t)$ po dani bazi φ_k .

Metode, ki poiščejo približek za vektor vrednosti, imenujemo [kolokacijske metode](#), metode, ki poiščejo približek za koeficiente v razvoju po bazi, pa [spektralne metode](#). Metode, ki jih bomo spoznali v nadaljevanju, uvrščamo med kolokacijske metode.

16.3 Eulerjeva metoda

Najpreprostejša metoda za reševanje začetnega problema je [Eulerjeva metoda](#). Za dani začetni problem (16.5) bomo poiskali vrednosti $u_i = u(t_i)$ v točkah $t_0 < t_1 < \dots < t_n = t_k$ na intervalu $[t_0, t_k]$. Vrednosti u_i poiščemo tako, da najprej izračunamo približek u_1 za t_1 in nato z isto metodo rešimo začetni problem z začetnim pogojem $u(t_1) \approx u_1$. Pri Eulerjevi metodi naslednjo vrednost izračunamo iz vrednosti tangente skozi (t_k, u_k) pri $t = t_{k+1}$. Tako dobimo rekurzivno formulo za približke v točkah t_1, t_2, \dots, t_n :

$$u_{k+1} = u_k + (t_{k+1} - t_k)f(t_k, u_k). \quad (16.6)$$

Samostojno delo

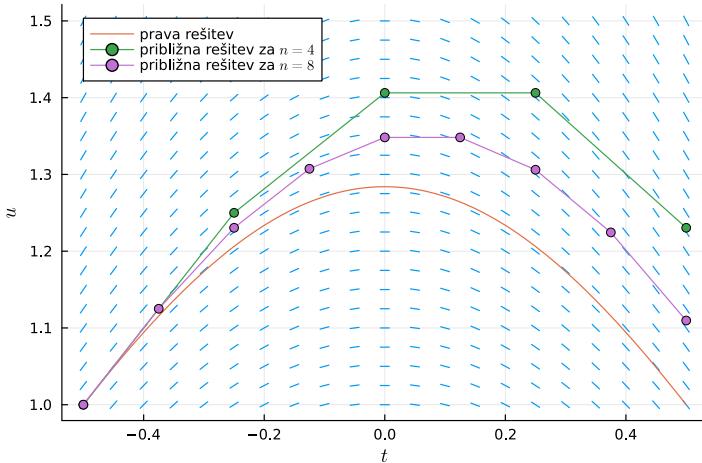
Napiši funkcijo `u`, `t = euler(fun, u0, (t0, tk), n)`, ki implementira Eulerjevo metodo s konstantnim korakom (Program 101).

```
tint = (-0.5, 0.5)
u0 = 1
riši_polje(fun, tint, (u0, 1.5))

C = u0 / exp(-tint[1]^2)
plot!(x -> C * exp(-x^2), tint[1], tint[2],
    label="prava rešitev", legend=:topleft)

t4, u4 = Vaja16.euler(fun, 1.0, (-0.5, 0.5), 4)
plot!(t4, u4, marker=:circle, label="približna rešitev za \$n=4\$",
    xlabel="\$t\$$, ylabel="\$u\$$")

t8, u8 = Vaja16.euler(fun, 1.0, [-0.5, 0.5], 8)
plot!(t8, u8, marker=:circle, label="približna rešitev za \$n=8\$",
    xlabel="\$t\$$, ylabel="\$u\$$")
```



Slika 58: Približna rešitev začetnega problema za enačbo $u' = -2tu$ z začetnim pogojem $u(-0.5) = 1$ na intervalu $[-0.5, 0.5]$. Približki so izračunani s 4 in 8 koraki Eulerjeve metode. Več korakov kot naredimo, boljši je približek za rešitev.

16.4 Sistemi NDE

Podobno kot za eno enačbo tudi za sistem navadnih diferencialnih enačb definiramo začetni problem. Sistem NDE za k spremenljivk u_1, u_2, \dots, u_k zapišemo v obliki:

$$\begin{aligned} u'_1 &= f_1(t, u_1, u_2, \dots, u_k, p), \\ u'_2 &= f_2(t, u_1, u_2, \dots, u_k, p), \\ &\vdots \\ u'_k &= f_k(t, u_1, u_2, \dots, u_k, p), \end{aligned} \tag{16.7}$$

začetni pogoj pa:

$$u_1(t_0) = u_{0,1}, u_2(t_0) = u_{0,2}, \dots, u_k(t_0) = u_{0,k}. \tag{16.8}$$

Sistem (16.7) obravnavamo kot eno samo enačbo, če ga zapišemo v vektorski obliki:

$$\mathbf{u}' = \mathbf{f}(t, \mathbf{u}, p), \tag{16.9}$$

kjer je $\mathbf{u} = [u_1, u_2, \dots, u_k]^T$ vektor posameznih odvisnih spremenljivk sistema. Začetni pogoj (16.8) zapišemo kot:

$$\mathbf{u}(t_0) = \mathbf{u}_0, \tag{16.10}$$

kjer je začetna vrednost $\mathbf{u}_0 = [u_{0,1}, u_{0,2}, \dots, u_{0,k}]^T$ prav tako vektor. Vektorska enačba (16.9) in začetni pogoj (16.10) imata povsem enako obliko kot pri enačbi z eno spremenljivko. Edina razlika je ta, da je \mathbf{u} vektorska spremenljivka, \mathbf{f} vektorska funkcija in \mathbf{u}_0 vektor.

Začetni problem za sistem NDE (16.7) z začetnim pogojem (16.8) ima tudi enolično rešitev in ga obravnavamo povsem enako kot začetni problem za enačbo z eno spremenljivko. Tudi Eulerjeva metoda in vse metode, ki jih bomo spoznali v nadaljevanju, delujejo povsem enako za posamezne enačbe in sisteme, le da številske vrednosti za u in u' nadomestimo z vektorskimi vrednostmi \mathbf{u} in \mathbf{u}' .

V nadaljevanju bomo predstavili ogrodje za numerično reševanje začetnih problemov za NDE, ki deluje tako za enačbe z eno spremenljivko kot tudi za sisteme.

16.5 Ogrodje za reševanje NDE

Definirali bomo tipe in funkcije, ki bodo omogočali enotno obravnavo reševanja začetnega problema in enostavno dodajanje različnih metod za reševanje NDE. Pri tem se bomo zgledovali po paketu [DifferentialEquations.jl](#) (glej [15] za podrobnejšo razlag).

Definirajmo podatkovni tip `ZačetniProblem`, ki vsebuje vse podatke o začetnem problemu (16.5). Uporabili ga bomo kot vhodni podatek za funkcije, ki bodo poiskale numerično rešitev.

```
"""
zp = ZačetniProblem(f, u0 tint, p)
```

Podatkovna struktura s podatki za začetni problem za navadne diferencialne enačbe (NDE) oblike:

```
```math
u'(t) = f(t, u, p).
```
```

Desna stran NDE je podana s funkcijo `f`, ki je odvisna od vrednosti `u`, `t` in parametrov enačbe `p`. Začetni pogoj `` $u(t_0) = u_0$ `` je določen s poljem `u0` v začetni točki intervala `` $tint=[t_0, t_k]$ ``, na katerem iščemo rešitev.

```
## Atributi

* `f`: funkcija, ki izračuna odvod (desne strani NDE)
* `u0`: začetni pogoj
* `tint`: časovni interval za rešitev
* `p`: vrednosti parametrov enačbe
"""

struct ZačetniProblem{TU,TT,TP}
    f          # desne strani NDE  $u' = f(t, u, p)$ 
    u0::TU    # začetna vrednost
    tint::Tuple{TT,TT}  # interval, na katerem iščemo rešitev
    p::TP      # parametri sistema
end
```

Približke, ki jih bomo izračunali z različnimi metodami, bomo shranili v poseben podatkovni tip `RešitevNDE`. Poleg vrednosti neodvisne spremenljivke in izračunanih približkov rešitve bomo v tipu `RešitevNDE` hranili tudi vrednosti odvodov, ki jih izračunamo s smernim poljem NDE. Odvode bomo potrebovali za izračun vmesnih vrednosti s Hermitovim zlepkom.

```
"""

Podatkovna struktura, ki hrani približek za rešitev začetnega problema za NDE.
Uporablja se kot tip, ki ga vrnejo metode za reševanje začetnega problema.
"""

struct RešitevNDE{TU,TT,TP}
    zp::ZačetniProblem{TU,TT,TP} # referenca na začetni problem
    t::Vector{TT}    # vrednosti časa (argumenta)
    u::Vector{TU}    # približki za vrednosti rešitve
    du::Vector{TU}   # izračunane vrednosti odvoda
end
```

Samostojno delo

- Definiraj podatkovni tip Euler, ki vsebuje parametre za Eulerjevo metodo.
- Napiši funkcijo resi(p::ZačetniProblem, metoda::Euler), ki poišče rešitev začetnega problema z Eulerjevo metodo (rešitev Program 103).

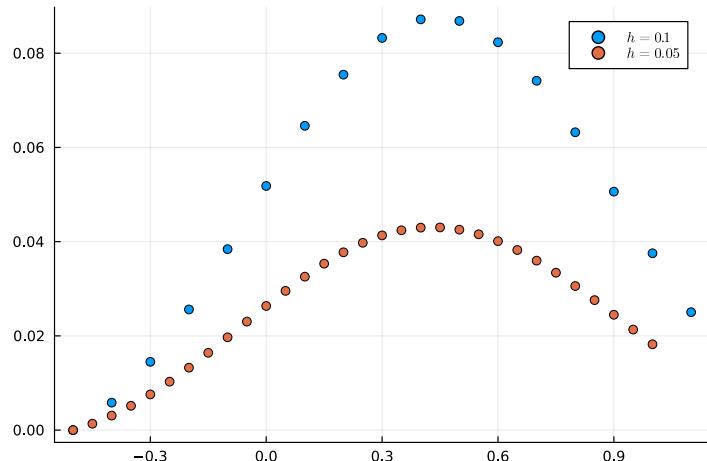
Ponovno rešimo začetni problem:

$$\begin{aligned} u'(t) &= -2tu, \\ u(-0.5) &= 1. \end{aligned} \tag{16.11}$$

Faktor -2 v enačbi $u'(t) = -2tu$ lahko obravnavamo kot parameter enačbe. Ker poznamo točno rešitev $u(t) = e^{-\frac{t^2}{2}}$, lahko izračunamo napako. Narišimo napako Eulerjeve metode za dve različni velikosti koraka.

```
fun(t, u, p) = p * t * u
problem = ZačetniProblem(fun, 1.0, (-0.5, 1.0), -2.0)
upravi(t) = exp(-t^2) / exp(-0.5^2)

res1 = resi(problem, Euler(0.1))
scatter(res1.t, res1.u - upravi.(res1.t), label="\$h=0.1\$")
res2 = resi(problem, Euler(0.05))
scatter!(res2.t, res2.u - upravi.(res2.t), label="\$h=0.05\$")
```



Slika 59: Napaka Eulerjeve metode za začetni problem $u' = -2ut$; $u(-0.5) = 1$ pri različnih velikostih koraka $h = 0.1$ in $h = 0.05$

Večlična razdelitev in posebni podatkovni tipi omogočajo abstraktno obravnavo

Uporaba specifičnih tipov omogoča definicijo specifičnih metod, ki so posebej napisane za posamezen primer. Taka organizacija kode omogoča večjo abstrakcijo in definicijo [domenskega jezika](#), ki je prilagojen posameznemu področju. Tako lahko obravnavamo ZačetniProblemNDE namesto funkcije in vektorja, ki vsebuje dejanske podatke. Vrednost tipa RešitevNDE se razlikuje od vektorjev in matrik, ki jih vsebuje, predvsem v tem, da Julia ve, da gre za podatke, ki so numerični približek za rešitev začetnega problema. To prevajalniku omogoča, da za dane podatke avtomatično uporabi metode glede na vlogo, ki jo imajo v danem kontekstu. Takšna organizacija je zelo prilagodljiva in omogoča enostavno dodajanje novih numeričnih metod ali novih formulacij problema samega.

16.6 Metode Runge-Kutta

Implementirajmo še metodi Runge-Kutta drugega in četrtega reda. Naslednji približek za $u_{n+1} = u(t_n + h)$ za metodo drugega reda lahko zapišemo kot:

$$\begin{aligned} k_1 &= hf(t_n, u_n, p) \\ k_2 &= hf(t_n + h, u_n + k_1, p) \\ u_{n+1} &= u_n + \frac{1}{2}(k_1 + k_2). \end{aligned} \tag{16.12}$$

Za metodo četrtega reda pa je naslednji približek enak:

$$\begin{aligned} k_1 &= hf(t_n, u_n, p) \\ k_2 &= hf\left(t_n + \frac{1}{2}h, u_n + \frac{1}{2}k_1, p\right) \\ k_3 &= hf\left(t_n + \frac{1}{2}h, u_n + \frac{1}{2}k_2, p\right) \\ k_4 &= hf(t_n + h, u_n + k_3, p) \\ u_{n+1} &= u_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4). \end{aligned} \tag{16.13}$$

Samostojno delo

Definiraj naslednje tipe in metode:

- podatkovni tip RK2, ki predstavlja metodo drugega reda (16.12) in metodo korak(m::RK2, fun, t0, u0, par, smer), ki izračuna približek na naslednjem koraku (Program 105),
- podatkovni tip RK4, ki predstavlja metodo četrtega reda (16.13) in metodo korak(m::RK4, fun, t0, u0, par, smer), ki izračuna približek na naslednjem koraku (Program 106).

16.7 Hermitova interpolacija

Numerične metode za začetni problem NDE izračunajo približke za rešitev zgolj v nekaterih vrednostih spremenljivke t . Vrednosti rešitve diferencialne enačbe lahko interpoliramo s **kubičnim Hermitovim zlepkom**, ki smo ga že spoznali v poglavju o zlepkih (Poglavlje 12). Hermitov zlepek je na intervalu $[t_i, t_{i+1}]$ določen z vrednostmi rešitve in odvodi v krajiščih intervala. Ti podatki so shranjeni v vrednosti tipa RešitevNDE.

Samostojno delo

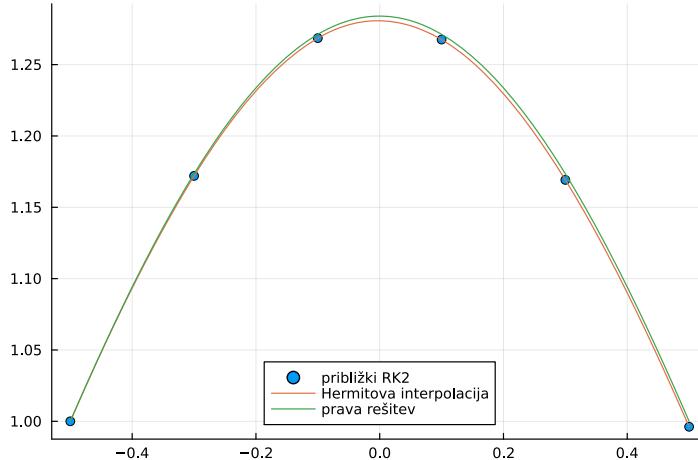
Napiši metodo vrednost(res::RešitevNDE, t), ki vrne približek za rešitev NDE v dani točki (Program 104). Vrednosti rešitve lahko na ta način izračunamo tudi za argumente t , ki so med približki, ki jih izračuna Eulerjeva ali kakšna druga metoda.

Prikažimo Hermitovo interpolacijo na grafu:

```

using Plots
fun(t, u, p) = p * t * u
upravi(t) = exp(-t^2) / exp(-0.5^2)
zp = ZačetniProblem(fun, upravi(-0.5), (-0.5, 0.5), -2.0)
res = resi(zp, RK2(0.2))
scatter(res.t, res.u, label="približki RK2")
plot!(t -> res(t), res.t[1], res.t[end], label="Hermitova interpolacija")
plot!(upravi, res.t[1], res.t[end], label="prava rešitev", legend=:bottom)

```



Slika 60: Vmesne vrednosti med približki metode Runge-Kutta reda 2 interpoliramo s Hermitovim zlepkom.

16.8 Poševni met z upoštevanja zračnega upora

Poševni met opisuje gibanje točkastega telesa pod vplivom gravitacije. Enačbe, ki opisujejo poševni met, izpeljemo iz [Newtonovih zakonov gibanja](#). Položaj telesa v vsakem trenutku opišemo z vektorjem položaja $\mathbf{x} = [x, y, z]^T \in \mathbb{R}^3$. Trajektorija opisuje položaj v odvisnosti od časa in je podana kot krivulja $\mathbf{x}(t)$. Označimo vektor hitrosti z:

$$\mathbf{v}(t) = \dot{\mathbf{x}}(t) = \frac{d\mathbf{x}}{dt}(t) \quad (16.14)$$

in vektor pospeška z:

$$\mathbf{a}(t) = \ddot{\mathbf{x}}(t) = \frac{d\mathbf{v}}{dt}(t) = \frac{d^2\mathbf{x}}{dt^2}(t). \quad (16.15)$$

Gibanje točkastega telesa z maso m pod vplivom vsote vseh sil \mathbf{F} , ki delujejo na dano telo, opiše drugi Newtonov zakon:

$$\mathbf{F} = m\mathbf{a}(t). \quad (16.16)$$

Sile, ki delujejo na telo, so lahko odvisne tako od položaja kot tudi od hitrosti. Sili, ki delujeta na telo pri poševnem metu, sta sila teže \mathbf{F}_g in sila zračnega upora \mathbf{F}_u . Privzamemo, da velja [kvadratni zakon upora](#), po katerem je velikost sile upora sorazmerna kvadratu velikosti hitrosti. Sila upora vedno kaže v nasprotno smer kot hitrost, zato zapišemo:

$$\mathbf{F}_u = -C' \mathbf{v} \|\mathbf{v}\|, \quad (16.17)$$

kjer je parameter C' odvisen od gostote medija in oblike ter velikosti telesa. Sila teže kaže vertikalno navzdol in je sorazmerna masi in težnemu pospešku g :

$$\mathbf{F}_g = m\mathbf{g} = -mg \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}, \quad (16.18)$$

kjer je $\mathbf{g} = g[0, 0, -1]^T$ vektor težnega pospeška. Vsoto vseh sil zapišemo kot:

$$\mathbf{F} = -m\mathbf{g} - C'\mathbf{v}\|\mathbf{v}\|. \quad (16.19)$$

Ko vstavimo (16.19) v drugi Newtonov zakon (16.16), dobimo sistem enačb drugega reda:

$$\ddot{\mathbf{x}}(t) = \mathbf{a}(t) = \frac{\mathbf{F}}{m} = \mathbf{g} - C\dot{\mathbf{x}}(t)\|\dot{\mathbf{x}}(t)\|, \quad (16.20)$$

kjer je $C = \frac{C'}{m}$. Če želimo uporabiti metode za numerično reševanje diferencialnih enačb, moramo sistem (16.20) prevesti na sistem enačb prvega reda. To naredimo tako, da vpeljemo nove spremenljivke za komponente odvoda $\dot{\mathbf{x}}$. Oznake za nove spremenljivke se ponujajo kar same $\mathbf{v}(t) = \dot{\mathbf{x}}(t)$. Sistem enačb drugega reda (16.20) je ekvivalenten sistemu enačb prvega reda:

$$\begin{aligned} \dot{\mathbf{x}}(t) &= \mathbf{v}(t), \\ \dot{\mathbf{v}}(t) &= \mathbf{g} - C\dot{\mathbf{x}}(t)\|\dot{\mathbf{x}}(t)\|. \end{aligned} \quad (16.21)$$

V enačbah (16.21) se v resnici skriva 6 enačb, po ena za vsako komponento vektorjev \mathbf{x} in \mathbf{v} . Če ni bočnega vetra, se telo giblje v navpični ravnini. Koordinatni sistem postavimo tako, da je $y = 0$ in število odvisnih spremenljivk in enačb zmanjšamo na 4. Spremenljivke združimo v en vektor s 4 komponentami:

$$\mathbf{u}(t) = \begin{pmatrix} x \\ z \\ v_x \\ v_z \end{pmatrix}. \quad (16.22)$$

Nato napišemo funkcijo `f_poševni(t, u, p)`, ki izračuna vektor desnih stani enačb (16.21):

```
using LinearAlgebra
"""

Izračunaj desne strani enačb za poševni met.

"""

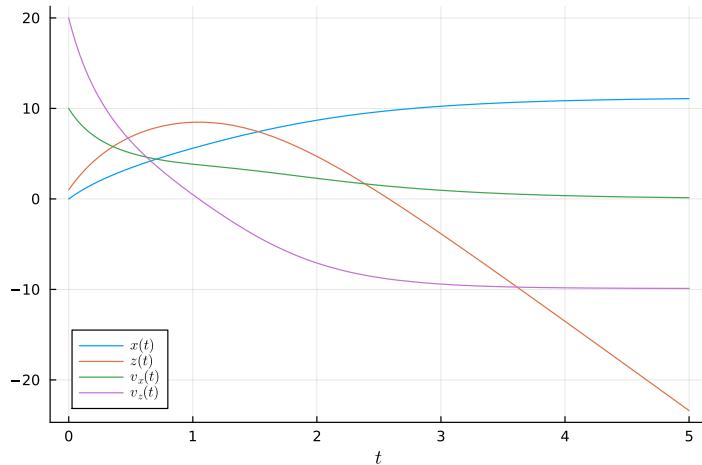
function f_poševni(_, u, par)
    g, c = par
    n = div(length(u), 2)
    v = u[n+1:end]
    fg = zero(v)
    fg[end] = -g
    f = fg - c * v * norm(v)
    return vcat(v, f)
end
```

Definirajmo vrednost tipa `ZačetniProblem`, ki opiše začetni problem za enačbe (16.21) za prvih 5s leta z začetnimi pogoji $x_0 = 0m$, $z_0 = 1m$, $\mathbf{v}_0 = [10\frac{m}{s}, 20\frac{m}{s}]^T$ in parametri $g = 9.8\frac{m}{s^2}$ ter $C = \frac{0.1}{m}$. Nato uporabimo funkcijo `resi` in problem rešimo z metodo Runge-Kuta reda 4 s korakom $h = 0.1s$.

```

x0 = [0.0, 1.0]
v0 = [10.0, 20.0]
tint = (0.0, 5.0)
g = 9.8
c = 0.1
zp = ZačetniProblem(f_poševni, vcat(x0, v0), tint, (g, c))
res = resi(zp, RK4(0.1))
using Plots
plot(t -> res(t)[1], tint..., label="$x(t)$")
plot!(t -> res(t)[2], tint..., label="$z(t)$")
plot!(t -> res(t)[3], tint..., label="$v_x(t)$")
plot!(t -> res(t)[4], tint..., label="$v_z(t)$", xlabel="$t$")

```



Slika 61: Komponente rešitve začetnega problema za poševni met. Graf prikazuje, kako se lega (koordinati x in z) ter hitrost (komponenti v_x in v_z) spreminjajo v odvisnosti od časa.

Iz grafa razberemo, da se hitrost približuje limitni vrednosti. V vodoravni smeri gre hitrost proti $0 \frac{m}{s}$, v vertikalni smeri pa se približuje vrednosti, ko sta sila teže in sila upora v ravnovesju

$$|\mathbf{F}_u| = |\mathbf{F}_g| \Rightarrow Cv^2 = g \Rightarrow v = \sqrt{\frac{g}{C}}. \quad (16.23)$$

Hitrost, ko se to zgodi imenujemo *terminalna hitrost* prostega pada. V našem primeru je terminalna hitrost enaka $v = \sqrt{9.8/0.1} \frac{m}{s} \approx 9.9 \frac{m}{s}$.

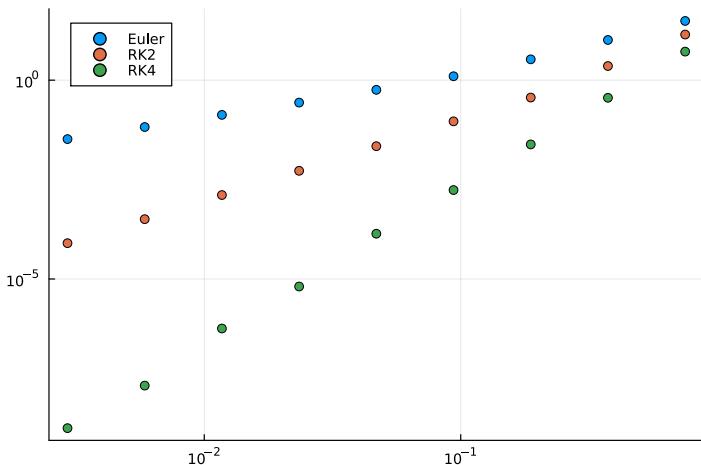
Poglejmo, kako se obnesejo različne metode. Primerjali bomo vse tri metode, ki smo jih spoznali. Za različne vrednosti koraka bomo izračunali približek in ga primerjali s pravo rešitvijo. Ker prave rešitve ne poznamo, bomo namesto nje uporabili približek, ki ga dobimo z metodo Runge-Kutta reda 4 s polovičnim korakom. Napako bomo ocenili tako, da bomo poiskali največjo napako med različnimi vrednostmi t na danem intervalu $[t_0, t_k]$.

```

zp = ZačetniProblem(f_poševni, [0.0, 2.0, 10.0, 20.0], (0.0, 3.0), (9.8, 0.1))
function napaka(reševalec, zp, rešitev, nvzorca=100)
    približek = resi(zp, reševalec)
    t0, tk = zp.tint
    t = range(t0, tk, nvzorca)
    maximum(t -> norm(približek(t) - rešitev(t)), t)
end

h = 3 ./ (2 .^ (2:10))
rešitev = resi(zp, RK4(h[end] / 2))
napakaEuler = [napaka(Euler(hi), zp, rešitev) for hi in h]
napakaRK2 = [napaka(RK2(hi), zp, rešitev) for hi in h]
napakaRK4 = [napaka(RK4(hi), zp, rešitev) for hi in h]
scatter(h, napakaEuler, xscale=:log10, yscale=:log10, label="Euler")
scatter!(h, napakaRK2, xscale=:log10, yscale=:log10, label="RK2")
scatter!(h, napakaRK4, xscale=:log10, yscale=:log10, label="RK4",
legend=:topleft)

```



Slika 62: Napaka za različne metode v odvisnosti od velikosti koraka v logaritemski skali

Kako izbrati prizerno metodo?

V tej vaji smo spoznali tri različne metode: Eulerjevo, Runge-Kutta reda 2 in reda 4. Poleg omenjenih, obstaja še cel živalski vrt različnih metod za reševanje začetnega problema NDE. Tule je na primer [seznam metod](#), implementiranih v paketu [DifferentialEquations](#), podrobneje pa so opisane v [16]. Kako se odločimo, katero metodo izbrati?

Izberemo metodo, ki ima red vsaj 4, sicer je treba korak zelo zmanjšati, da dobimo dovolj dobro natančnost. Za splošno rabo so najprimernejše metode s kontrolo koraka. Zelo popularna je metoda [Dormand-Prince reda 5](#) s kratico DOPRI5, ki jo privzeto uporablja Matlab, Octave in paket [DifferentialEquations](#) za Julio.

Pri nekaterih NDE postanejo običajne metode kot so Runge-Kutta in DOPRI5 numerično nestabilne. Take enačbe imenujemo [toge diferencialne enačbe](#). Za toge diferencialne enačbe so razvili veliko specjalnih metod (glej [17]).

Prav tako obstajajo metode, ki so prilagojene posebnim razredom diferencialnih enačb, na primer enačbam na Liejevih grupah in homogenih prostorih, Hamiltonskim enačbam in še mnogo drugih.

16.9 Čas in dolžina meta

Za različne začetne pogoje in parametre želimo poiskati, kako daleč leti telo, preden pade na tla. Naj bodo tla na višini 0. Najprej poiščemo, kdaj telo zadene tla. To se zgodi takrat, ko je telo na višini 0. Iskani čas je torej rešitev enačbe:

$$z(t) = u_2(t) = 0. \quad (16.24)$$

Vrednost $z(t)$ je ena komponenta rešitve začetnega problema. Enačba (16.24) je nelinearna enačba, za katero ne poznamo eksplisitne formule. Kljub temu lahko za iskanje rešitev uporabimo metode za reševanje nelinearnih enačb. Problema se lotimo splošnejše. Enačbo (16.24) zapišemo kot:

$$h(t) = F(\mathbf{u}(t)) = 0, \quad (16.25)$$

kjer je \mathbf{u} rešitev začetnega problema (16.21), funkcija $F(\mathbf{u})$ pa vrne vertikalno komponento vektorja \mathbf{u} :

$$F([x, z, v_x, v_z]) = z. \quad (16.26)$$

Za reševanje enačbe $h(t) = 0$ uporabimo metode za reševanje nelinearnih enačb, na primer bisekcijo ali Newtonovo metodo. Lahko uporabimo Newtonovo metodo, saj je vrednost odvoda $h'(t)$ na voljo. Vrednost odvoda $\dot{\mathbf{u}}(t)$ in s tem tudi $h'(t) = \frac{d}{dt}F(\mathbf{u}(t))$ izračunamo iz desnih strani diferencialne enačbe:

$$h'(t) = \frac{d}{dt}F(\mathbf{u}(t)) = \nabla F(\mathbf{u}(t)) \cdot \dot{\mathbf{u}}(t) = \nabla F \cdot f(t, \mathbf{u}(t), p). \quad (16.27)$$

Z numeričnimi metodami dobimo približek za začetni problem v obliki zaporedja približkov:

$$\mathbf{u}_0, \mathbf{u}_1, \dots, \mathbf{u}_n \quad (16.28)$$

za vrednosti ob časih t_0, t_1, \dots, t_n . Za vsak izračun $h(t)$ bi morali vsakič znova izračunati začetni del zaporedja \mathbf{u}_i . Da se temu izognemo, najprej poiščemo interval $[t_i, t_{i+1}]$, na katerem leži ničla. V tabeli poiščemo i , za katerega je:

$$F(\mathbf{u}_i)F(\mathbf{u}_{i+1}) < 0. \quad (16.29)$$

Ko poiščemo interval $[t_i, t_{i+1}]$, na katerem je ničla, smo ničlo že poiskali z natančnostjo $\delta = |t_{i+1} - t_i|$. Zmanjšanje koraka osnovne metode bi sicer dalo boljši približek, vendar se tudi časovna zahtevnost poveča za enak faktor, kot se zmanjša natančnost. Bistveno bolje je uporabiti eno od metod za reševanje nelinearnih enačb.

Vrednosti t_i in \mathbf{u}_i uporabimo kot nov začetni približek za začetni problem. Tako lahko zgolj z enim korakom izbrane metode izračunamo $h(t) = F(\mathbf{u}(t))$ za poljuben $t \in [t_i, t_{i+1}]$.

Iskanje ničle v tabeli

Imamo tabelo vrednosti funkcije $[f_1, f_2, \dots, f_n]$ in želimo poiskati ničlo. Pogoj $f_i = 0$ ni najboljši, saj zaradi zaokrožitvenih napak skoraj zagotovo ni nikoli izpolnjen. Tudi pogoj $|f_i| < \varepsilon$ ni dosti boljši, ker je ničla lahko med dvema vrednostima f_i in f_{i+1} , čeprav sta vrednosti daleč stran od ničle. Ničla pa je zagotovo tam, kjer funkcija spremeni predznak. Pravi pogoj je zato:

$$f_i \cdot f_{i+1} < 0. \quad (16.30)$$

Samostojno delo

Napiši naslednje funkcije:

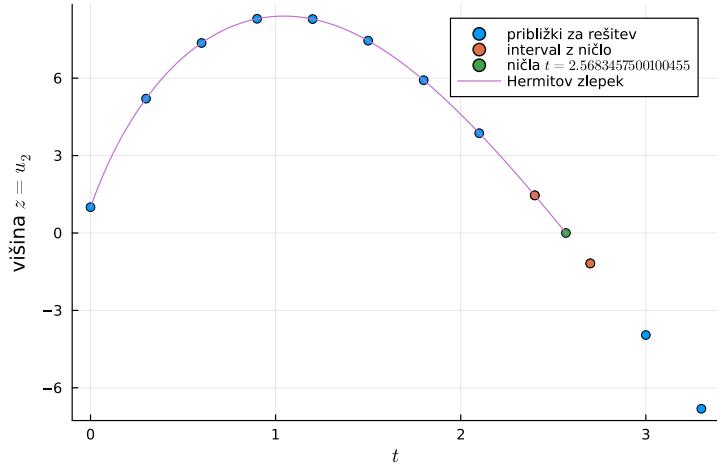
- `ničla_interval(res::RešitevNDE, F)`, ki poišče interval, na katerem je dana funkcija $F(t, u, du)$ enaka 0 (Program 107) in
- `ničla(res::RešitevNDE, F, DF)`, ki poišče ničlo funkcije $F(t, u, du)$ za dano rešitev začetnega problema. Za računanje novih vrednosti naj uporabi metodo RK4 (Program 108).

Funkcijo `ničla` uporabimo za poševni met. Uporabili bomo relativno velik korak, da bo postopek iskanja ničle nazornejši. Rešimo začetni problem za poševni met (16.21) s parametri $g = 9.8 \frac{m}{s^2}$, $c = 0.1 \frac{m}{s}$, z začetnim položajem $x = 0m, z = 1m$ in začetno hitrostjo $v_x = 10 \frac{m}{s^2}, v_z = 20 \frac{m}{s}$. Enote so v numeričnem izračunu izpuščene.

```
x0 = [0.0, 1.0]
v0 = [10.0, 20.0]
tint = (0.0, 3.0)
g = 9.8
c = 0.1
zp = ZačetniProblem(f_poševni, vcat(x0, v0), tint, (g, c))
res = resi(zp, RK4(0.3))
scatter(res.t, [u[2] for u in res.u], label="približki za rešitev")
```

Približki za rešitev so precej narazen, saj smo za izračun uporabili relativno velik korak $h = 0.3$. Kljub temu je zaradi visokega reda metode Runge-Kutta izračun dokaj natančen. V tabeli približkov, ki smo jo dobili z metodo Runge-Kutta, poiščemo interval, na katerem druga komponenta u_2 spremeni predznak. Nato z Newtonovo metodo rešimo nelinearno enačbo $u_2(t) = 0$.

```
fun(_t, u, _du) = u[2]
dfun(_t, u, _du) = u[4]
i = Vaja16.ničla_interval(res, fun)
scatter!(res.t[i:i+1], [res.u[i][2], res.u[i+1][2]], label="interval z ničlo")
t0 = ničla(res, fun, dfun)
scatter!([t0], [res(t0)[2]], label="ničla \$(t=\$(t0))$")
plot!(t -> res(t)[2], 0, t0, label="Hermitov zlepek", xlabel="\$t\$",
      ylabel="višina \$z=u_2\$")
```



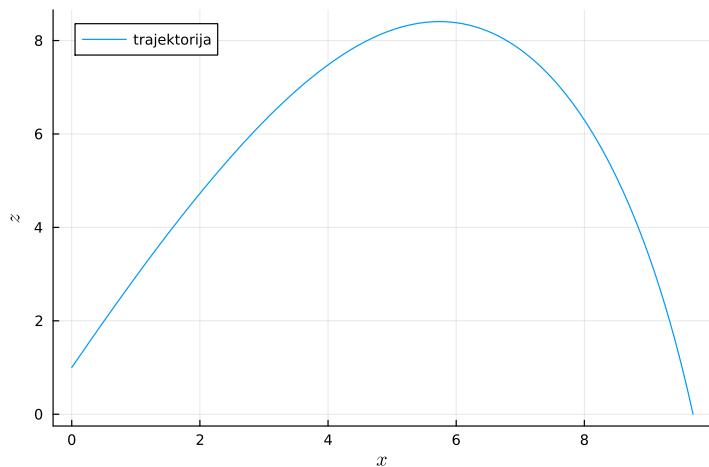
Slika 63: Slika ilustrira postopek, s katerim poiščemo, koliko časa izstrelki leti, preden pade na tla. Najprej poiščemo približke za rešitev z metodo Runge-Kutta reda 4. Nato poiščemo interval med približki, na katerem višina $z = u_2$ spremeni predznak. Enačbo $u_2(t) = 0$ rešimo z Newtonovo metodo.

Vrednost t_0 predstavlja čas leta, medtem ko dolžino leta razberemo iz prve komponente rešitve.

```
julia> (t0, res(t0))
(2.5683457500100455, [9.69610631974621, -4.334343792807005e-5, 1.4160245365127713,
-8.836708202579016])
```

Iz rezultata razberemo, da je čas leta približno $2.57s$, medtem ko je dolžina, ki jo izstrelek doseže enaka $9.67m$. Narišimo še trajektorijo, ki je podana s parametrično krivuljo $(x(t), z(t)) = (u_1(t), u_2(t))$.

```
t = range(0, t0, 100)
x = [res(ti)[1] for ti in t]
y = [res(ti)[2] for ti in t]
plot(x, y, label="trajektorija", xlabel="\$x\$", ylabel="\$z\$")
```



Slika 64: Trajektorija poševnega meta od začetka do trenutka, ko izstrelek pada na tla.

16.10 Rešitve

```
using LinearAlgebra

"""
Izračunaj enotski vektor v smeri tangente na rešitev diferencialne enačbe
`u'(t)=f(t, u(t))``"""
function tangenta(f, t, u)
    v = [1, f(t, u)]
    return v / norm(v)
end

function daljica(f, t, u, l)
    v = l * tangenta(f, t, u) / 2
    dt, du = v
    return [t - dt, t + dt], [u - du, u + du]
end

"""
Vzorči smerno polje za NDE 1. reda `u' = fun(t, u)` na pravokotniku
`[t0, tk] x [u0, uk]` z `n` delilnimi točkami v obeh smereh.
"""
function vzorči_polje(fun, (t0, tk), (u0, uk), n=21)
    t = range(t0, tk, n)
    u = range(u0, uk, n)

    dt = t[2] - t[1]
    du = u[2] - u[1]
    l = min(dt, du) * 0.6

    # enotske vektorje skaliramo, da se ne prekrivajo
    polje = [daljica(fun, ti, ui, l) for ti in t for ui in u]
    return polje
end
```

Program 99: Funkcija izračuna polje smeri za NDE prvega reda v vozliščih pravokotne mreže na danem pravokotniku.

```

using Plots
function riši_polje(fun, (t0, tk), (u0, uk), n=21)
    polje = vzorči_polje(fun, (t0, tk), (u0, uk), n)
    N = length(polje)
    x = polje[1][1]
    y = polje[1][2]
    for i in 2:N
        # med daljice vrinemo vrednosti NaN, da plot prekine črto
        push!(x, NaN)
        append!(x, polje[i][1])
        push!(y, NaN)
        append!(y, polje[i][2])
    end
    return plot(x, y, xlabel="\$t\$$", ylabel="\$u\$$", label=false)
end

```

Program 100: Funkcija nariše polje smeri za NDE prvega reda v ravnini t, u .

```

"""
u, t = euler(fun, u0, (t0, tk), n)

Izračunaj približek za rešitev začetnega problema za diferencialno enačbo z
Eulerjevo metodo z enakimi koraki.

# Argumenti

- `fun` desne strani DE `u'=fun(t, u)`
- `u0` začetni pogoj `u(t0) = u0`
- `(t0, tk)` interval, na katerem iščemo rešitev
- `n` število korakov Eulerjeve metode
"""

function euler(fun, u0, tint, n)
    t0, tk = tint
    t = range(t0, tk, n + 1)
    h = t[2] - t[1]
    u = [u0]
    for i = 1:n
        u0 += h * fun(t[i], u0)
        push!(u, u0)
    end
    return t, u
end

```

Program 101: Eulerjeva metoda za reševanje začetnega problema NDE

```

"""
r = resi(zp::ZačetniProblem, reševalec::TR) where {TR<:ReševalecNDE}

Reši začetni problem za NDE `zp` z danim reševalcem `reševalec`.

# Primer

Rešimo ZP za enačbo `u'(t) = -2t u` z začetnim pogojem `u(-0.5) = 1.0`:
```julia-repl
julia> fun(t, u, p) = -p * t * u;
julia> problem = ZačetniProblem(fun, 1., (-0.5, 1), 2);
julia> res = resi(problem, Euler(0.5)) # reši problem s korakom 0.5
```
"""

function resi(zp::ZačetniProblem{TU,TT,TP}, metoda::TM) where
{TU,TT,TP,TM<:ReševalecNDE}
    t0, tk = zp.tint
    u0 = zp.u0
    smer = sign(tk - t0)
    t = [t0]
    u = [u0]
    du = TU[]
    while smer * t0 < smer * tk
        t0, u0, du0 = korak(metoda, zp.f, t0, u0, zp.p, smer)
        push!(t, t0)
        push!(u, u0)
        push!(du, du0)
    end
    push!(du, zp.f(t[end], u[end], zp.p)) # odvod v zadnjem približku
    return RešitevNDE(zp, t, u, du)
end

```

Program 102: Funkcija `resi`, ki poišče rešitev začetnega problema z različnimi metodami. Posamezne metode implementiramo tako, da definiramo metode za funkcijo `korak`.

```

plain
"""
    u, t = euler(fun, u0, (t0, tk), n)

Izračunaj približek za rešitev začetnega problema za diferencialno enačbo z
Eulerjevo metodo z enakimi koraki.

# Argumenti

- `fun` desne strani DE `u'=fun(t, u)`
- `u0` začetni pogoj `u(t0) = u0`
- `(t0, tk)` interval, na katerem iščemo rešitev
- `n` število korakov Eulerjeve metode
"""

function euler(fun, u0, tint, n)
    t0, tk = tint
    t = range(t0, tk, n + 1)
    h = t[2] - t[1]
    u = [u0]
    for i = 1:n
        u0 += h * fun(t[i], u0)
        push!(u, u0)
    end
    return t, u
end

```

Program 103: Metoda za funkcijo **korak**, ki poišče rešitev začetnega problema z enim korakom Eulerjeve metode.

```

using Vaja12
"""
    y = vrednost(r, t)

Izračunaj vrednost rešitve `r` v točki `t`. Funkcija za izračun vrednosti
uporabi interpolacijo s Hermitovim zlepkom.

"""

function vrednost(r::RešitevNDE, t)
    z = Vaja12.HermitovZlepek(r.t, r.u, r.du)
    return Vaja12.vrednost(t, z)
end

# Omogočimo, da rešitev NDE kličemo kot funkcijo.
(res::RešitevNDE)(t) = vrednost(res, t)

```

Program 104: Vmesne vrednosti rešitve NDE izračunamo s Hermitovim kubičnim zlepkom.

```

struct RK2{T} <: ReševalecNDE
    h::T # dolžina koraka
end

"""
Izračunaj en korak metode Runge-Kutta reda 2.
"""

function korak(m::RK2, fun, t0, u0, par, smer)
    h = smer * m.h
    du = fun(t0, u0, par)
    t = t0 + h
    k1 = h * du
    k2 = h * fun(t, u0 + k1, par)
    return t, u0 + (k1 + k2) / 2, du
end

```

Program 105: Metoda za funkcijo korak, ki poišče rešitev začetnega problema z enim korakom metode Runge-Kutta drugega reda.

```

struct RK4{T} <: ReševalecNDE
    h::T
end

function korak(res::RK4, fun, t0, u0, par, smer)
    h = smer * res.h
    du = fun(t0, u0, par)
    k1 = h * du
    k2 = h * fun(t0 + h / 2, u0 + k1 / 2, par)
    k3 = h * fun(t0 + h / 2, u0 + k2 / 2, par)
    k4 = h * fun(t0 + h, u0 + k3, par)
    return t0 + h, u0 + (k1 + 2(k2 + k3) + k4) / 6, du
end

```

Program 106: Metoda za funkcijo korak, ki poišče rešitev začetnega problema z enim korakom metode Runge-Kutta četrtega reda.

```

function ničla_interval(res::RešitevNDE, fun)
    t, u, du = res.t, res.u, res.du
    n = length(t)
    for i in 1:n-1
        if fun(t[i], u[i], du[i]) * fun(t[i+1], u[i+1], du[i+1]) < 0
            return i
        end
    end
    throw("Ni intervala z ničlo")
end

```

Program 107: Funkcija, ki poišče interval $[t_i, t_{i+1}]$, na katerem ima funkcija $F(u(t))$ ničlo.

```

function newton(fdf, x0, maxit=10, atol=1e-8)
    for _ in 1:maxit # Newtonova metoda
        z, dz = fdf(x0)
        dx = -z / dz
        x0 += dx
        if abs(dx) < atol
            return x0
        end
    end
    throw("Newtonova metoda ne konvergira po $maxit korakih!")
end

function ničla(res::RešitevNDE, fun, dfun, maxit=10, atol=1e-8)
    t, u = res.t, res.u
    i = ničla_interval(res, fun)
    function rhs(tk) # desne strani enačbe
        reševalec = RK4(tk - t[i])
        smer = sign(tk - t[i])
        t0, u0, du0 = korak(reševalec, res.zp.f, t[i], u[i], res.zp.p, smer)
        return fun(t0, u0, du0), dfun(t0, u0, du0)
    end
    newton(rhs, t[i], maxit, atol)
end

```

Program 108: Funkcija, ki poišče vrednost t , pri kateri je $F(\mathbf{u}(t)) = 0$.

16.11 Testi

```

@testset "Vrednost za RešitevNDE" begin
    f(t) = [t^3, t^2]
    df(t) = [3t.^2, 2t]
    t = [0., 1., 2.]
    zp = ZačetniProblem((t, x, p) -> df(t), [0.0, 0], (0.0, 2.0), nothing)
    res = RešitevNDE(zp, t, f.(t), df.(t))
    @test res(0.) ≈ f(0)
    @test res(1.) ≈ f(1)
    @test res(2.) ≈ f(2)
end

```

Program 109: Test za računanje vmesnih vrednosti rešitev začetnega problema

17 Navodila za pripravo domačih nalog

V nadaljevanju so navodila za pripravo domačih nalog v programskejem jeziku [Julia](#). Navodila so uporabna tudi za reševanje v kakšnem drugem programskejem jeziku. V tem primeru jih smiselno prilagodimo.

17.1 Kontrolni seznam

Seznam delov, ki jih mora vsebovati vsaka domača naloga:

- koda (`src\NalogaXY.jl`),
- testi (`test\runtests.jl`),
- dokument `README.md`,
- demo skripta, s katero ustvarite rezultate za poročilo,
- poročilo v formatu PDF.

Preden končaš domačo nalogo, uporabi naslednji *kontrolni seznam*:

- vse funkcije imajo dokumentacijo,
- testi pokrivajo večino kode,
- *README* vsebuje naslednje:
 - ▶ ime in priimek avtorja,
 - ▶ opis naloge,
 - ▶ navodila kako uporabiti kodo,
 - ▶ navodila, kako pognati teste,
 - ▶ navodila, kako ustvariti poročilo,
- *README* ni predolg,
- poročilo vsebuje naslednje:
 - ▶ ime in priimek avtorja,
 - ▶ splošen (matematični) opis naloge,
 - ▶ splošen opis rešitve,
 - ▶ primer uporabe (slikice prosim :-).

17.2 Kako pisati in kako ne

V nadaljevanju je nekaj primerov dobre prakse, kako pisati kodo, teste in poročilo. Pri pisanju besedil je vedno treba imeti v mislih, komu je poročilo namenjeno.

Pisec naj uporabi empatijo do bralca in naj poskuša napisati zgodbo, ki ji bralec lahko sledi. Tudi če je pisanje namenjeno strokovnjakom, je dobro, da je čim več besedila razumljivega tudi širši publik. Tudi strokovnjaki radi beremo besedila, ki jih hitro razumemo. Zato je dobro začeti z okvirnim opisom z malo formulami in splošnimi izrazi. V nadaljevanju besedilo stopnjujemo k vedno podrobnejšim informacijam.

Določene podrobnosti, ki so povezane s konkretno implementacijo, brez škode izpustimo.

Opis rešitve naj bo okviren

Opis rešitve naj bo zgolj okviren. Izogibaj se uporabi programerskih izrazov. Raje uporabi matematične. Na primer: izraz **uporabimo „for“ zanko**, nadomesti s **postopek ponavljam**. Od bralca zahteva splošen

opis manj napora in mu da širšo sliko. Če želiš dodati izpeljave, jih napiši z matematičnimi formulami, ne v programskejem jeziku. Koda sodi zgolj v del, kjer je opisana uporaba za konkreten primer.

DOBRO! Splošen opis algoritma

Algoritem za LU razcep smo prilagodili tridiagonalni strukturi matrike. Namesto trojne zanke smo uporabili le enojno, saj je pod pivotnim elementom neničelen le en element. Časovna zahtevnost algoritma je tako z $\mathcal{O}(n^3)$ padla na zgolj $\mathcal{O}(n)$.

SLABO! Podrobna razлага kode, vrstico po vrstico

V programu za LU razcep smo uporabili for zanko od 2 do velikosti matrike. V prvi vrstici zanke smo izračunali $L.s[i]$, tako da smo element $T.s[i]$ delili z $U.z[i-1]$. Nato smo izračunali diagonalni element, tako da smo uporabili formulo $U.d[i] - L.s[i]*U.d[i-1]$. Na koncu zanke smo vrnili matriki L in U .

Podrobnosti implementacije ne sodijo v poročilo

Podrobnosti implementacije so razvidne iz kode, zato jih nima smisla ponavljati v poročilu. Algoritme opišemo okvirno, tako da izpustimo podrobnosti, ki niso nujno potrebne za razumevanje. Podrobnosti lahko dodamo v nadaljevanju, če so potrebne.

DOBRO! Algoritem opišemo okvirno, podrobnosti razložimo kasneje

V matriki želimo eliminirati spodnji trikotnik. To dosežemo tako, da stolpce enega za drugim pre-slikamo s Householderjevimi zrcaljenji. Za vsak stolpec poiščemo vektor, preko katerega zrcalimo. Vektor poiščemo tako, da ima zrcalna slika ničle pod diagonalnim elementom.

Tu lahko z razlago zaključimo. Če želimo dodati podrobnosti, jih navedemo za okvirno idejo.

DOBRO! Podrobnosti sledijo za okvirno razlago

Vektor zrcaljenja dobimo kot:

$$u = [s(k) + A_{k,k}, A_{k+1,k}, \dots A_{n,k}], \quad (17.1)$$

kjer je $s(k) = \text{sign}(A_{k,k}) * \|A(k : n, k)\|$. Podmatriko $A(k : n, k + 1 : n)$ prezrcalimo preko vektorja u , tako da podmatriki odštejemo matriko:

$$2u \frac{u^T A(k : n, k + 1 : n)}{u^T u}. \quad (17.2)$$

Na k -tem koraku prezrcalimo le podmatriko $k : n \times k : n$, ostali deli matrike pa ostanejo nespremenjeni.

Takošnje razlaganje podrobnosti, brez predhodnega opisa osnovne ideje, ni dobro, saj bralec težko razloči, kaj je zares pomembno in kaj ne.

SLABO! *Takoj dodamo vse podrobnosti, ne da bi razložili namen*

Za vsak k poiščemo vektor $u = [s(k) + A_{k,k}, A_{k+1,k}, \dots, A_{n,k}]$, kjer je $s(k) = \text{sign}(A_{k,k}) * \| [A_{k,k}, \dots, A_{n,k}] \|$.

Nato matriko popravimo:

$$A(k : n, k + 1 : n) = A(k : n, k + 1 : n) - 2 * u * \frac{u^T * A(k : n, k + 1 : n)}{u^T * u}. \quad (17.3)$$

Če implementacija vsebuje posebnosti, kot na primer uporabo posebne podatkovne strukture ali algoritma, jih lahko opišemo v poročilu. Vendar tudi tu pazimo, da bralca ne obremenjujemo s podrobnostmi.

DOBRO! *Posebnosti implementacije opisemo v grobem in se ne spuščamo v podrobnosti*

Za tridiagonalne matrike definiramo posebno podatkovno strukturo `Tridiag`, ki hrani le neničelne elemente matrike. Julia omogoča, da LU razcep tridiagonalne matrike implementiramo kot specializirano metodo funkcije `lu` iz paketa `LinearAlgebra`. Pri tem upoštevamo posebnosti tridiagonalne matrike in algoritem za LU razcep prilagodimo tako, da se časovna in prostorska zahtevnost zmanjšata na $\mathcal{O}(n)$.

Pazimo, da v poročilu ne povzemamo direktno posameznih korakov kode.

SLABO! *Opisovanje, kaj počnejo posamezni koraki kode, ne sodi v poročilo.*

Za tridiagonalne matrike definiramo podatkovni tip `Tridiag`, ki ima 3 atribute `s`, `d` in `z`. Atribut `s` vsebuje elemente pod diagonalo, ...

LU razcep implementiramo kot metodo za funkcijo `LinearAlgebra.lu`. V `for` zanki izračunamo naslednje:

1. element $l[i] = a[i, i-1] / a[i-1, i-1]$
2. ...

17.3 Kako pisati avtomatske teste

Nekaj nasvetov, kako lahko testiramo kodo:

- Na roke poiščemo rešitev za preprost primer in jo primerjamo z rezultati funkcije.
- Ustvarimo testne podatke, za katere je znana rešitev. Na primer za testiranje kode, ki reši sistem $Ax=b$, izberemo A in x in izračunamo desne strani kot $b=A*x$.
- Preverimo lastnost rešitve. Za enačbe $f(x)=0$ lahko rešitev, ki jo izračuna program, preprosto vstavimo nazaj v enačbo in preverimo, ali je enačba izpolnjena.
- Red metode lahko preverimo tako, da naredimo simulacijo in primerjamo red metode z redom programa, ki ga eksperimentalno določimo.
- Če je le mogoče, v testih ne uporabljamo rezultatov, ki jih proizvede koda sama. Ko je koda dovolj časa v uporabi, lahko rezultate kode same uporabimo za [regresijske teste](#).

Pokritost kode s testi

Pri pisanju testov je pomembno, da testi izvedejo vse veje v kodi. Delež kode, ki se izvede med testi, imenujemo [pokritost kode \(angl. Code Coverage\)](#). V Julii lahko pokritost kode dobimo z argumentom `coverage=true`, ki ga dodamo metodi `Pkg.test`:

```
julia> import Pkg; Pkg.test("NalogaXY"; coverage=true)
```

Zgornji ukaz bo za vsako datoteko iz mape `src` ustvaril ustrezeno datoteko s končnico `.cov`, v kateri je shranjena informacija o tem, kateri deli kode so bili uporabljeni med izvajanjem testov. Za pripravo povzetka o pokritosti kode uporabimo paket [Coverage.jl](#):

```
using Coverage
cov = process_folder("NalogaXY")
pokrite_vrstice, vse_vrstice = get_summary(cov)
delež = pokrite_vrstice / vse_vrstice
println("Pokritost kode s testi: $(round(delež*100))%.")
```

17.4 Priprava zahteve za združitev

Za lažjo komunikacijo predlagam, da rešitev domače naloge postaviš v svojo vejo in ustvariš zahtevo za združitev (*Pull request* za GitHub ozziroma *Merge request* za GitLab). V nadaljevanju bomo opisali, kako to storиш, če repozitorij z domačimi nalogami gostiš na GitLabu. Postopek za GitHub in druge platforme je podoben.

Preden začneš z delom, ustvari vejo na svoji delovni kopiji repozitorija in jo potisno na GitLab. Ime veje naj bo domača-X, se pravi domača-01 za prvo domačo nalogu in tako naprej. To narediš z ukazom

```
$ git checkout -b domača-01
$ git push -u origin domača-01
```

Stikalo -u pove gitu, naj z domačo vejo sledi veji na GitLabu.

Med delom sproti dodajaj vnose z `git commit` in jih prenesi na GitLab z ukazom `git push`. Ko je domača naloga končana, na GitLabu ustvari zahtevo za združitev (angl. Merge request):

- Klikni na zavihek `Merge requests` in nato na gumb `New merge request`.
- Na desni strani izberi vejo `domača-01` in klikni na gumb `Compare branches and continue`.
- Ko je koda pripravljena na pregled, odstrani besedo `Draft`: v naslovu in v komentarju povabi asistenta k pregledu. To storиш tako, da v komentar dodaš uporabniško ime asistenta (npr. `@mojZlobniAsistent`):

`@mojZlobniAsistent Prosim za pregled.`

Pri domačih nalogah se posvetuj s kolegi

Nič ni narobe, če za pomoč pri domači nalogi prosiš kolega. Seveda moraš kodo in poročilo napisati sam, lahko pa kolega prosiš za pregled kode v zahtevi za združitev ali za pomoč, če kaj ne dela.

Domačo nalogu lahko rešujete skupinsko, vendar morate v tem primeru rešiti toliko različnih nalog, kot vas je v skupini.

18 Domače naloge

Pomemben del učenja je samostojno reševanje nalog. Naloge v tem poglavju so razdeljene v tri sklope. Prvi vsebuje naloge, ki zahtevajo izdelavo rešitve po točno določenih specifikacijah. Naloge iz drugih dveh sklopov so bolj odprtrega tipa in zahtevajo več samostojnosti pri reševanju. Priporočam, da iz vsakega sklopa rešiš po eno nalogo in da naloge rešite v obliki paketa za Julio z vsem, kar sodi zraven (glej Poglavlje 1 in Poglavlje 17).

Namen nalog ni, da na internetu poiščeš optimalen algoritem in ga implementiraš, ampak da uporabiš lastno znanje, čeprav na koncu rešitev morda ne bo optimalna. Kljub temu pazi na **časovno in prostorsko zahtevnost** in zahtevano natančnost.

18.1 Prva domača naloga

18.1.1 SOR iteracija za razpršene matrike

Naj bo A $n \times n$ diagonalno dominantna razpršena matrika (velika večina elementov je ničelnih, $a_{ij} = 0$).

Definiraj nov podatkovni tip RedkaMatrika, ki matriko zaradi prostorskih zahtev hrani v dveh matrikah V in I , kjer sta V in I $n \times m$ matriki, tako da velja:

$$V(i, j) = A(i, I(i, j)). \quad (18.1)$$

V matriki V se torej nahajajo neničelni elementi matrike A . Vsaka vrstica matrike V vsebuje neničelne elemente iz iste vrstice v A . V matriki I pa so shranjeni indeksi stolpcev teh neničelnih elementov.

Za podatkovni tip RedkaMatrika definiraj metode za naslednje funkcije:

- indeksiranje: `Base.getindex`, `Base.setindex!`, `Base.firstindex` in `Base.lastindex`
- množenje z desne `Base.*` z vektorjem

Več informacij o [tipih in vmesnikih](#).

Napiši funkcijo `x, it = sor(A, b, x0, omega, tol=1e-10)`, ki reši razpršeni sistem $Ax = b$ z SOR iteracijo. Pri tem je $x0$ začetni približek, tol pogoj za ustavitev iteracije in $omega$ parameter pri SOR iteraciji. Iteracija naj se ustavi, ko je

$$\|Ax^{(k)} - b\|_\infty < \delta, \quad (18.2)$$

kjer je δ podan z argumentom `tol`.

Metodo uporabi za vložitev grafa v ravnino ali prostor s fizikalno metodo (Poglavlje 6).

Za uporabljene primere poišči optimalni ω , pri katerem SOR najhitreje konvergira in predstavi odvisnost hitrosti konvergence od izbire ω .

18.1.2 Metoda konjugiranih gradientov za razpršene matrike

Definirajte nov podatkovni tip RedkaMatrika, kot je opisano v prejšnji nalogi.

Napišite funkcijo `[x, i]=conj_grad(A, b)`, ki reši sistem

$$Ax = b, \quad (18.3)$$

z metodo konjugiranih gradientov za tip matrike `RedkaMatrika`.

Metodo uporabite na primeru vložitve grafa v ravnino ali prostor s fizikalno metodo, kot je opisano v prejšnji nalogi.

18.1.3 Metoda konjugiranih gradientov s predpogojevanjem

Za pohitritev konvergencije iterativnih metod se velikokrat izvede t. i. predpogojevanje (angl. preconditioning). Za simetrične pozitivno definitne matrike je to pogosto nepopolni razcep Choleskega, pri katerem sledimo algoritmu za razcep Choleskega, le da ničelne elemente pustimo pri miru.

Naj bo A $n \times n$ pozitivno definitna razpršena matrika (velika večina elementov je ničelnih, $a_{ij} = 0$). Matriko zaradi prostorskih zahtev hranimo kot *razpršeno* matriko. Poglej si dokumentacijo za [razpršene matrike](#).

Napišite funkcijo `L = nep_chol(A)`, ki izračuna nepopolni razcep Choleskega za matriko tipa `AbstractSparseMatrix`. Napišite še funkcijo `x, i = conj_grad(A, b, L)`, ki reši sistem linearnih enačb

$$Ax = b \quad (18.4)$$

s predpogojeno metodo konjugiranih gradientov za matriko $M = L^T L$ kot predpogojevalcem. Pri tem pazи, da matrike M ne izračunate, ampak uporabite razcep $M = L^T L$. Za različne primere preverite, ali se izboljša hitrost konvergencije.

18.1.4 QR razcep zgornje Hessenbergove matrike

Naj bo H $n \times n$ zgornja Hessenbergova matrika (velja $a_{ij} = 0$ za $j < i - 1$). Definiraj podatkovni tip `ZgornjiHessenberg` za zgornjo Hessenbergovo matriko.

Napiši funkcijo `Q, R = qr(H)`, ki izvede QR razcep matrike H tipa `ZgornjiHessenberg` z Givenovimi rotacijami. Matrika R naj bo zgornja trikotna matrika enakih dimenzij kot H , v Q pa naj bo matrika tipa `Givens`.

Podatkovni tip `Givens` definirajte sami tako, da hrani le zaporedje rotacij, ki se med razcepom izvedejo in indekse vrstic, na katere te rotacije delujejo. Posamezno rotacijo predstavite s parom

$$[\cos(\alpha); \sin(\alpha)], \quad (18.5)$$

kjer je α kot rotacije na posameznem koraku (kota α ni treba računati). Za tip `Givens` definiraj še množenje `Base.*` z vektorji in matrikami.

Uporabi QR razcep za QR iteracijo zgornje Hessenbergove matrike. Napišite funkcijo `lastne_vrednosti, lastni_vektorji = eigen(H)`, ki poišče lastne vrednosti in lastne vektorje zgornje Hessenbergove matrike.

Preverite časovno zahtevnost vaših funkcij in ju primerjajte z metodo `qr` za navadne matrike.

18.1.5 QR razcep simetrične tridiagonalne matrike

Naj bo A $n \times n$ simetrična tridiagonalna matrika (velja $a_{ij} = 0$ za $|i - j| > 1$).

Definiraj podatkovni tip `SimTridiag` za simetrično tridiagonalno matriko, ki hrani glavno in stransko diagonalo matrike. Za tip `SimTridiag` definiraj metode za naslednje funkcije:

- indeksiranje: `Base.getindex`, `Base.setindex!`, `Base.firstindex` in `Base.lastindex`,

- množenje z desne `Base.*` z vektorjem ali matriko.

Časovna zahtevnost omenjenih funkcij naj bo linearnejša. Napiši funkcijo `Q, R = qr(T)`, ki izvede QR razcep matrike `T` tipa `Tridiag` z Givensovimi rotacijami. Matrika `R` naj bo zgornje trikotna tridiagonalna matrika tipa `ZgornjeTridiag`, v `Q` pa naj bo matrika tipa `Givens`.

Definiraj podatkovna tipa `ZgornjeTridiag` in `Givens` (glej nalogo 18.1.4). Poleg tega implementiraj množenje `Base.*` matrik tipa `Givens` in `ZgornjeTridiag`.

Uporabi QR razcep za QR iteracijo simetrične tridiagonalne matrike. Napiši funkcijo `lastne_vrednosti, lastni_vektorji = eigen(T)`, ki poišče lastne vrednosti in lastne vektorje simetrične tridiagonalne matrike.

Preveri časovno zahtevnost vaše funkcije in jih primerjaj z metodo `qr` za navadne matrike.

18.1.6 Inverzna potenčna metoda za zgornjo Hessenbergovo matriko

Lastne vektorje matrike A lahko računamo z **inverzno potenčno metodo**. Naj bo $A_\lambda = A - \lambda I$. Če je λ približek za lastno vrednost, potem zaporedje vektorjev

$$\mathbf{x}^{(n+1)} = \frac{A_\lambda^{-1} \mathbf{x}^{(n)}}{\|A_\lambda^{-1} \mathbf{x}^{(n)}\|_\infty}, \quad (18.6)$$

konvergira k lastnemu vektorju za lastno vrednost, ki je po absolutni vrednosti najbližje vrednosti λ .

Da bi zmanjšali število operacij na eni iteraciji, lahko poljubno matriko A prevedemo v zgornjo Hessenbergovo obliko (velja $a_{ij} = 0$ za $j < i - 1$). S Householderjevimi zrcaljenji lahko poiščemo zgornje Hessenbergovo matriko H , ki je podobna matriki A :

$$H = Q^T A Q. \quad (18.7)$$

Če je v lastni vektor matrike H , je Qv lastni vektor matrike A , lastne vrednosti matrik H in A pa so enake.

Napiši funkcijo `H`, `Q = hessenberg(A)`, ki s Householderjevimi zrcaljenji poišče zgornje Hessenbergovo matriko `H` tipa `ZgornjiHessenberg`, ki je podobna matriki `A`.

Definiraj tip `ZgornjiHessenberg`, kot je opisano v nalogi o QR razcepnu zgornje Hessenbergove matrike. Poleg tega implementirajte metodo `L, U = lu(A)` za matrike tipa `ZgornjiHessenberg`, ki bo pri razcepnu upoštevala lastnosti zgornje Hessenbergovih matrik. Matrika `L` naj ne bo polna, ampak tipa `SpTridiag`. Tip `SpTridiag` definirajte sami, tako da bo hranil le neničelne elemente in za ta tip matrike definirajte operator `Base.\`, tako da bo upošteval strukturo matrike `L`.

Napišite funkcijo `lambda, vektor = inv_lastni(A, l)`, ki najprej naredi Hessenbergov razcep in nato izračuna lastni vektor in točno lastno matrike `A`, kjer je `l` približek za lastno vrednost. Inverza matrike `A` nikar ne računajte, ampak raje uporabite LU razcep in na vsakem koraku rešite sistem $L(Ux^{n+1}) = x^n$.

Metodo preskusiti za izračun ničel polinoma. Polinomu

$$x^n + a_{n-1}x^{n-2} + \dots + a_1x + a_0 \quad (18.8)$$

lahko priredimo matriko

$$\begin{pmatrix} 0 & 0 & \dots & 0 & -a_0 \\ 1 & 0 & \dots & 0 & -a_1 \\ 0 & 1 & \dots & 0 & -a_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & 1 & -a_{n-1} \end{pmatrix} \quad (18.9)$$

katere lastne vrednosti se ujemajo z ničlami polinoma.

18.1.7 Inverzna potenčna metoda za tridiagonalno matriko

Lastne vektorje matrike A lahko računamo z **inverzno potenčno metodo**. Naj bo $A_\lambda = A - \lambda I$. Če je λ približek za lastno vrednost, potem zaporedje vektorjev

$$\mathbf{x}^{(n+1)} = \frac{A_\lambda^{-1} \mathbf{x}^{(n)}}{\|A_\lambda^{-1} \mathbf{x}^{(n)}\|_\infty}, \quad (18.10)$$

konvergira k lastnemu vektorju za lastno vrednost, ki je po absolutni vrednosti najbližje vrednosti λ .

Naj bo A simetrična matrika. Da bi zmanjšali število operacij na eni iteraciji, lahko poljubno simetrično matriko A prevedemo v tridiagonalno obliko. S Householderjevimi zrcaljenji lahko poiščemo tridiagonalno matriko T , ki je podobna matriki A :

$$T = Q^T A Q. \quad (18.11)$$

Če je v lastni vektor matrike T , je Qv lastni vektor matrike A , lastne vrednosti matrik T in A pa so enake.

Napiši funkcijo T , $Q = \text{tridiag}(A)$, ki s Householderjevimi zrcaljenji poišče tridiagonalno matriko H tipa Tridiag , ki je podobna matriki A .

Definiraj tip Tridiag , kot je opisano v nalogi o QR razcepu tridiagonalne matrike. Poleg tega implementiraj metodo L , $U = \text{lu}(A)$ za matrike tipa Tridiag , ki bo pri razcepu upoštevala lastnosti tridiagonalnih matrik. Matrike L in U naj ne bodo polne matrike. Matrika L naj bo tipa SpTridiag , matrika U pa tipa ZgTridiag . Definiraj tipa SpTridiag in ZgTridiag , tako da bosta hranila le neničelne elemente. Za oba tipa definiraj operator $\text{Base}.\backslash$, ki z obratnim oziroma direktnim vstavljanjem reši sistem linearnih enačb.

Napiši funkcijo lambda , $\text{vektor} = \text{inv_lastni}(A, \lambda)$, ki najprej naredi Hessenbergov razcep in nato izračuna lastni vektor in točno lastno matrike A , kjer je λ približek za lastno vrednost. Inverza matrike A nikar ne računaj, ampak raje uporabi LU razcep in na vsakem koraku reši sistem $L(U\mathbf{x}^{n+1}) = \mathbf{x}^n$.

Metodo preskus na Laplaceovi matriki, ki ima vse elemente 0 razen $l_{ii} = -2$, $l_{i+1,j} = l_{i,j+1} = 1$. Poisci nekaj lastnih vektorjev za najmanjše lastne vrednosti in jih vizualiziraj z ukazom plot .

Lastni vektorji Laplaceove matrike so približki za rešitev robnega problema za diferencialno enačbo

$$y''(x) = \lambda^2 y(x), \quad (18.12)$$

katere rešitve sta funkciji $\sin(\lambda x)$ in $\cos(\lambda x)$.

18.1.8 Naravni zlepek

Danh je n interpolacijskih točk (x_i, f_i) , $i = 1, 2, \dots, n$. **Naravni interpolacijski kubični zlepek** S je funkcija, ki izpolnjuje naslednje pogoje:

1. $S(x_i) = f_i, \quad i = 1, 2, \dots, n.$
2. S je polinom stopnje 3 ali manj na vsakem podintervalu $[x_i, x_{i+1}], i = 1, 2, \dots, n - 1.$
3. S je dvakrat zvezno odvedljiva funkcija na interpolacijskem intervalu $[x_1, x_n]$
4. $S''(x_1) = S''(x_n) = 0.$

Zlepek S določimo tako, da postavimo

$$S(x) = S_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3, \quad x \in [x_i, x_{i+1}], \quad (18.13)$$

nato pa rešimo sistem linearnih enačb za koeficiente, ki jih izpeljemo iz zahtevanih pogojev (glej [2]).

Napiši funkcijo `Z = interpoliraj(x, y)`, ki izračuna koeficiente polinomov S_i in vrne element tipa Zlepek.

Definiraj tip Zlepek, ki naj vsebuje koeficiente polinoma in interpolacijske točke. Za tip Zlepek napiši dve funkciji

- `y = vrednost(Z, x)`, ki vrne vrednost zlepka v dani točki x .
- `plot(Z)`, ki nariše graf zlepka, tako da različne odseke izmenično nariše z rdečo in modro barvo (uporabi paket `Plots`).

18.1.9 QR iteracija z enojnim premikom

Naj bo A simetrična matrika. Napišite funkcijo, ki poišče lastne vektorje in vrednosti simetrične matrike z naslednjim algoritmom

- Izvedi Hessenbergov razcep matrike $A = U^T TU$ (uporabite lahko vgrajeno funkcijo `LinearAlgebra.hessenberg`)
- Za tridiagonalno matriko T ponavljam, dokler ni $h_{n-1,n}$ dovolj majhen:
 - za $T - \mu I$ za $\mu = h_{n,n}$ izvedi QR razcep
 - nov približek je enak $RQ + \mu I$
- Postopek ponovi za podmatriko brez zadnjega stolpca in vrstice

Napiši metodo `lastne_vrednosti`, `lastni_vektorji = eigen(A, EnojniPremik(), vektorji = false)`, ki vrne

- vektor lastnih vrednosti simetrične matrike A , če je vrednost `vektorji` enaka `false`,
- vektor lastnih vrednosti λ in matriko s pripadajočimi lastnimi vektorji V , če je `vektorji` enaka `true`.

Pazi na časovno in prostorsko zahtevnost algoritma. QR razcep tridiagonalne matrike izvedi z Givensovimi rotacijami in hrani le elemente, ki so nujno potrebni (glej nalogu *QR razcep simetrične tridiagonalne matrike*).

Funkcijo preskusiti na Laplaceovi matriki grafa podobnosti (glej Poglavlje 8).

18.2 Druga domača naloga

Druga domača naloga ima dve vrsti nalog. Prva vrsta zahteva program za računanje vrednosti dane funkcije $f(x)$, druga vrsta pa izračun ene vrednosti. Obe nalogi reši na **10 decimalk** (z relativno natančnostjo 10^{-10}). Uporabiš lahko le osnovne operacije, vgrajene osnovne matematične funkcije `exp`, `sin`, `cos`, ..., osnovne operacije z matrikami in razcepe matrik. Vse ostale algoritme implementiraj sam.

18.2.1 Naloge s funkcijami

Implementacija funkcije naj zadošča naslednjim zahtevam:

- relativna napaka je manjša od $5 \cdot 10^{-11}$ za vse argumente in
- časovna zahtevnost je omejena s konstanto, ki je neodvisna od argumenta.

18.2.2 Fresnelov integral (težja)

Napiši učinkovito funkcijo, ki izračuna vrednosti Fresnelovega kosinusa

$$C(x) = \int_0^x \cos\left(\frac{\pi t^2}{2}\right) dt. \quad (18.14)$$

Namig: Uporabi pomožni funkciji

$$\begin{aligned} f(z) &= \frac{1}{\pi\sqrt{2}} \int_0^\infty \frac{e^{-\frac{\pi z^2 t}{2}}}{\sqrt{t}(t^2 + 1)} dt, \\ g(z) &= \frac{1}{\pi\sqrt{2}} \int_0^\infty \frac{\sqrt{t}e^{-\frac{\pi z^2 t}{2}}}{t^2 + 1} dt, \end{aligned} \quad (18.15)$$

kot je opisano v [18].

18.2.3 Porazdelitvena funkcija za $N(0, 1)$

Napiši učinkovito funkcijo, ki izračuna porazdelitveno funkcijo standardne normalne porazdelitve:

$$F(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^x e^{-\frac{t^2}{2}} dt. \quad (18.16)$$

Poskrbi, da bo relativna napaka tudi za negativne vrednosti dovolj majhna in da je časovna zahtevnost omejena z isto konstanto na celiem definicijskem območju.

Namig: Definicijsko območje razdeli na več območij in na vsakem območju uporabi drugo metodo.

18.2.4 Funkcija kvantilov za $N(0, 1)$

Napiši učinkovito funkcijo, ki izračuna funkcijo kvantilov za standardno normalno porazdeljeno slučajno spremenljivko. Funkcija kvantilov je inverzna funkcija F^{-1} porazdelitvene funkcije:

$$F(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^x e^{-\frac{t^2}{2}} dt. \quad (18.17)$$

Poskrbi, da bo relativna napaka za vrednosti blizu 0 in 1 dovolj majhna in da je časovna zahtevnost omejena z isto konstanto na celiem intervalu $(0, 1)$.

18.2.5 Integralski sinus (težja)

Napišite učinkovito funkcijo, ki izračuna integralski sinus

$$\text{Si}(x) = \int_0^x \frac{\sin(t)}{t} dt. \quad (18.18)$$

Uporabite pomožni funkciji

$$\begin{aligned} f(z) &= \int_0^\infty \frac{\sin(t)}{t+z} = \int_0^\infty \frac{e^{-zt}}{t^2+1} dt, \\ g(z) &= \int_0^\infty \frac{\cos(t)}{t+z} = \int_0^\infty \frac{te^{-zt}}{t^2+1} dt, \\ \text{Si}(z) &= \frac{\pi}{2} - f(z) \cos(z) - g(z) \sin(z), \end{aligned} \quad (18.19)$$

kot je opisano v [19].

18.2.6 Naravni parameter (težja)

Napišite učinkovito funkcijo, ki izračuna **naravni parameter**:

$$s(t) = \int_0^t \sqrt{\dot{x}(\tau)^2 + \dot{y}(\tau)^2} d\tau \quad (18.20)$$

za parametrično krivuljo

$$(x(t), y(t)) = (t^3 - t, t^2 - 1). \quad (18.21)$$

Namig: Za velike vrednosti argumenta t interpoliraj funkcijo $\frac{1}{s(\frac{1}{t})}$ s polinomom v Čebiševih točkah (Poglavlje 18.2.7).

18.2.7 Interpolacija z baricentrično formulo

Napišite program, ki za dano funkcijo f na danem intervalu $[a, b]$ izračuna polinomski interpolant v Čebiševih točkah. Vrednosti naj računa z **baricentrično Lagrangeovo interpolacijo** po formuli

$$l(x) = \begin{cases} \frac{\sum \frac{f(x_j) \lambda_j}{x - x_j}}{\sum \frac{\lambda_j}{x - x_j}}, & x \neq x_j, \\ f(x_j), & \text{sicer.} \end{cases} \quad (18.22)$$

Čebiševe točke so podane na intervalu $[-1, 1]$ s formulo

$$x_k = \cos\left(\frac{2k-1}{2n}\pi\right), \quad k = 0, 1, \dots, n-1, \quad (18.23)$$

vrednosti uteži λ_k pa so enake

$$\lambda_k = (-1)^k \begin{cases} 1, & 0 < i < n, \\ \frac{1}{2}, & i = 0, \\ n, & \text{sicer.} \end{cases} \quad (18.24)$$

Za interpolacijo na splošnem intervalu $[a, b]$ si pomagaj z linearino funkcijo na interval $[-1, 1]$. Program uporabi za tri različne funkcije e^{-x^2} na $[-1, 1]$, $\frac{\sin x}{x}$ na $[0, 10]$ in $|x^2 - 2x|$ na $[1, 3]$. Za vsako funkcijo določi stopnjo polinoma, da napaka ne bo presegla 10^{-6} .

18.2.8 Naloge za izračun posamezne vrednosti

Pri naslednjih nalogah ravno tako nalogo rešite v obliki paketa za Julio. V demonstracijski skripti implementiraj funkcijo, ki izračuna iskano vrednost.

18.2.9 Sila težnosti

Izračunajte velikost sile težnosti med dvema vzporedno postavljenima enotskima homogenima kockama na razdalji 1. Predpostavite, da so vse fizikalne konstante, ki nastopajo v problemu, enake 1. Sila med dvema telesoma $T_1, T_2 \subset \mathbb{R}^3$ je enaka

$$\mathbf{F} = \int_{T_1} \int_{T_2} \frac{\mathbf{r}_1 - \mathbf{r}_2}{\|\mathbf{r}_1 - \mathbf{r}_2\|_2^2} d\mathbf{r}_1 d\mathbf{r}_2. \quad (18.25)$$

18.2.10 Ploščina hipotrohoide

Izračunajte ploščino območja, ki ga omejuje hipotrohoida podana parametrično z enačbama:

$$x(t) = (a + b) \cos(t) + b \cos\left(\frac{a+b}{b}t\right), \quad (18.26)$$

$$y(t) = (a + b) \sin(t) + b \sin\left(\frac{a+b}{b}t\right), \quad (18.27)$$

za parametra $a = 1$ in $b = -\frac{11}{7}$.

Namig: Uporabite formulo za [ploščino krivočrtnega trikotnika](#) pod krivuljo:

$$P = \frac{1}{2} \int_{t_1}^{t_2} (x(t)\dot{y}(t) - \dot{x}(t)y(t)) dt. \quad (18.28)$$

18.2.11 Povprečna razdalja (težja)

Izračunajte povprečno razdaljo med dvema točkama znotraj telesa T , ki je enako razliki dveh kock:

$$T = [-1, 1]^3 - [0, 1]^3. \quad (18.29)$$

Integral na produktu razlike dveh množic $(A - B) \times (A - B)$ lahko izrazimo kot vsoto integralov:

$$\begin{aligned} \int_{A-B} \int_{A-B} f(x, y) dx dy &= \int_A \int_A f(x, y) dx dy \\ &\quad - 2 \int_A \int_B f(x, y) dx dy + \int_B \int_B f(x, y) dx dy. \end{aligned} \quad (18.30)$$

18.2.12 Ploščina zanke Bézierjeve krivulje

Izračunajte ploščino zanke, ki jo omejuje Bézierjeva krivulja dana s kontrolnim poligonom:

$$(0, 0), (1, 1), (2, 3), (1, 4), (0, 4), (-1, 3), (0, 1), (1, 0). \quad (18.31)$$

Namig: Uporabite lahko formulo za [ploščino krivočrtnega trikotnika](#) pod krivuljo:

$$P = \frac{1}{2} \int_{t_1}^{t_2} (x(t)\dot{y}(t) - \dot{x}(t)y(t))dt. \quad (18.32)$$

18.2.13 Gauss-Legendrove kvadrature (lažja)

Izpelji [Gauss-Legendrovo integracijsko pravilo](#) na dveh točkah

$$\int_0^h f(x)dx = Af(x_1) + Bf(x_2) + R_f \quad (18.33)$$

vključno s formulo za napako R_f . Izpelji sestavljeni pravilo za $\int_a^b f(x)dx$ in napiši program, ki to pravilo uporabi za približno računanje integrala. Oceni, koliko izračunov funkcijskih vrednosti je potrebnih za izračun približka za

$$\int_0^5 \frac{\sin x}{x} dx \quad (18.34)$$

na 10 decimalnih mest natančno. *Namig:* Najprej izpelji pravilo na intervalu $[-1, 1]$ in ga nato prevedi na poljuben interval $[x_i, x_{i+1}]$. Za oceno napake uporabi izračun z dvojnim številom korakov.

18.3 Tretja domača naloga

Zahtevana števila izračunajte na **10 decimalih** (z relativno natančnostjo 10^{-10}) Uporabite lahko le osnovne operacije, vgrajene osnovne matematične funkcije `exp`, `sin`, `cos`, ..., osnovne operacije z matrikami in razcepe matrik. Vse ostale algoritme morate implementirati sami.

18.3.1 Ničle Airyjeve funkcije

Airyjeva funkcija je dana kot rešitev začetnega problema

$$Ai''(x) - x Ai(x) = 0, \quad Ai(0) = \frac{1}{3^{\frac{2}{3}}\Gamma(\frac{2}{3})}, \quad Ai'(0) = -\frac{1}{3^{\frac{1}{3}}\Gamma(\frac{1}{3})}. \quad (18.35)$$

Poščite čim več ničel funkcije Ai na 10 decimalnih mest natančno. Ni dovoljeno uporabiti vgrajene funkcije za reševanje diferencialnih enačb. Lahko pa uporabite Airyjevo funkcijo `airyai` iz paketa `SpecialFunctions.jl`, da preverite ali ste res dobili pravo ničlo.

Namig: Za računanje vrednosti lahko uporabite Magnusovo metodo reda 4 za reševanje enačb oblike:

$$y'(x) = A(x)y(x), \quad (18.36)$$

pri kateri nov približek y_{k+1} dobimo takole:

$$\begin{aligned}
A_1 &= A \left(x_k + \left(\frac{1}{2} - \frac{\sqrt{3}}{6} \right) h \right), \\
A_2 &= A \left(x_k + \left(\frac{1}{2} + \frac{\sqrt{3}}{6} \right) h \right), \\
\sigma_{k+1} &= \frac{h}{2}(A_1 + A_2) - \frac{\sqrt{3}}{12}h^2[A_1, A_2], \\
y_{k+1} &= \exp(\sigma_{k+1})y_k.
\end{aligned} \tag{18.37}$$

Izraz $[A, B]$ je komutator dveh matrik in ga izračunamo kot $[A, B] = AB - BA$. Eksponentno funkcijo na matriki $\exp(\sigma_{k+1})$ pa v programskejem jeziku Julia dobite z ukazom `exp`.

18.3.2 Dolžina implicitno podane krivulje

Poščite približek za dolžino krivulje, ki je dana implicitno z enačbama:

$$\begin{aligned}
F_1(x, y, z) &= x^4 + y^2/2 + z^2 = 12, \\
F_2(x, y, z) &= x^2 + y^2 - 4z^2 = 8.
\end{aligned} \tag{18.38}$$

Krivuljo lahko poiščete kot rešitev diferencialne enačbe

$$\dot{x}(t) = \nabla F_1 \times \nabla F_2. \tag{18.39}$$

18.3.3 Perioda limitnega cikla

Poščite periodo [limitnega cikla](#) za diferencialno enačbo

$$x''(t) - 4(1 - x^2)x'(t) + x = 0 \tag{18.40}$$

na 10 decimalk natančno.

18.3.4 Obhod lune

Sondo Appolo pošljite iz Zemljine orbite na [tir z vrnitvijo brez potiska](#), ki obkroži Luno in se vrne nazaj v Zemljino orbito. Rešujte sistem diferencialnih enačb, ki ga dobimo v koordinatnem sistemu, v katerem Zemlja in Luna mirujeta ([omejen krožni problem treh teles](#)). Naloge ni potrebno reševati na 10 decimalk.

Omejen krožni problem treh teles

Označimo z M maso Zemlje in z m maso Lune. Ker je masa sonde zanemarljiva, Zemlja in Luna krožita okrog skupnega masnega središča. Enačbe gibanja zapišemo v vrtečem koordinatnem sistemu, kjer masi M in m mirujeta. Označimo

$$\mu = \frac{m}{M+m} \quad \text{in} \quad \bar{\mu} = 1 - \mu = \frac{M}{M+m}. \tag{18.41}$$

V brezdimenzijskih koordinatah (dolžinska enota je kar razdalja med masama M in m) postavimo maso M v točko $(-\mu, 0, 0)$, maso m pa v točko $(\bar{\mu}, 0, 0)$. Označimo z R in r oddaljenost satelita s položajem (x, y, z) od mas M in m , tj.

$$R = R(x, y, z) = \sqrt{(x + \mu)^2 + y^2 + z^2}, \\ r = r(x, y, z) = \sqrt{(x - \bar{\mu})^2 + y^2 + z^2}. \quad (18.42)$$

Enačbe gibanja sonde so potem:

$$\ddot{x} = x + 2\dot{y} - \frac{\bar{\mu}}{R^3}(x + \mu) - \frac{\mu}{r^3}(x - \bar{\mu}), \\ \ddot{y} = y - 2\dot{x} - \frac{\bar{\mu}}{R^3}y - \frac{\mu}{r^3}y, \\ \ddot{z} = -\frac{\bar{\mu}}{R^3}z - \frac{\mu}{r^3}z. \quad (18.43)$$

18.3.5 Matematično nihalo (lažja)

Kotni odmik $\theta(t)$ (v radianih) pri nedušenem nihanju uteži obešene na vrvici opišemo z diferencialno enačbo

$$\frac{g}{l} \sin(\theta(t)) + \ddot{\theta}(t) = 0, \quad \theta(0) = \theta_0, \dot{\theta}(0) = \dot{\theta}_0, \quad (18.44)$$

kjer je g težni pospešek in l dolžina nihala. Napiši funkcijo, ki izračuna odmik nihala ob določenem času. Enačbo drugega reda prevedi na sistem prvega reda in računajte z metodo [DOPRI5](#).

Za različne začetne pogoje primerjaj rešitev z nihanjem harmoničnega nihala, ki je dano z enačbo

$$\frac{g}{l}\theta(t) + \ddot{\theta}(t) = 0. \quad (18.45)$$

Pri harmoničnem nihalu je nihajni čas neodvisen od začetnih pogojev, medtem ko je pri matematičnem nihalu nihajni čas narašča, ko se veča energija nihala (začetni odmik). Nariši graf odvisnosti nihajnega časa matematičnega nihala od energije nihala.

Literatura

1. Orel, B.: Osnove numerične matematike. Fakulteta za računalništvo in informatiko (2004).
2. Plestenjak, B.: Razširjen uvod v numerične metode. DMFA - založništvo (2015).
3. Bezanson, J., Edelman, A., Karpinski, S., Shah, V.B.: Julia: A Fresh Approach to Numerical Computing. SIAM Review. 59, 65–98 (2017). <https://doi.org/10.1137/141000671>.
4. Christ, S., Schwabeneder, D., Rackauckas, C., Borregaard, M.K., Breloff, T.: Plots.jl – a user extensible plotting API for the julia programming language. (2023). <https://doi.org/https://doi.org/10.5334/jors.431>.
5. Knuth, D.E.: Literate programming. The Computer Journal. 27, 97–111 (1984). <https://doi.org/10.1093/comjnl/27.2.97>.
6. Domajenko, V.: Babilonski približek za [kvadratni koren iz 2]. Presek. 21, 40–45 (1993).
7. Savchenko V. V., Pasko, A. A., Okunev, O. G., Kunii T. L.: Function representation of solids reconstructed from scattered surface points and contours. Computer Graphics Forum. 14, 181–188 (1995).
8. Turk, G., O'Brien, J.: Variational Implicit Surfaces, <https://www.semanticscholar.org/paper/Variational-Implicit-Surfaces-Turk-O'Brien/50dbc9f86af75dad7be6b2e92601e4ded7bee2d6>, (1999).
9. Morse, B.S., Yoo, T.S., Rheingans, P., Chen, D.T., Subramanian, K.R.: Interpolating implicit surfaces from scattered surface data using compactly supported radial basis functions, <https://ieeexplore.ieee.org/document/923379>, (2001). <https://doi.org/10.1109/SMA.2001.923379>.
10. Buhmann, M.: Radial Basis Functions. V: Engquist, B. (ur.) Encyclopedia of Applied and Computational Mathematics. str. 1216–1219 (2015).
11. Fairbanks, J., Besançon, M., Simon, S., Hoffman, J., Eubank, N., Karpinski, S.: JuliaGraphs/Graphs.jl: an optimized graphs package for the Julia programming language, <https://github.com/JuliaGraphs/Graphs.jl/>.
12. Luxburg, U. von: A tutorial on spectral clustering. Statistics and Computing. 17, 395–416 (2007). <https://doi.org/10.1007/s11222-007-9033-z>.
13. Revels, J., Lubin, M., Papamarkou, T.: Forward-Mode Automatic Differentiation in Julia. arXiv:1607.07892 [cs.MS]. (2016).
14. Kochenderfer, M.J., Wheeler, T.A.: Algorithms for Optimization. The MIT Press (2019).
15. Rackauckas, C., Nie, Q.: DifferentialEquations.jl--a performant and feature-rich ecosystem for solving differential equations in Julia. Journal of Open Research Software., 5, (2017).
16. Hairer, E., Wanner, G., Nørsett, S.P.: Solving Ordinary Differential Equations I. Springer (1993). <https://doi.org/10.1007/978-3-540-78862-1>.
17. Hairer, E., Wanner, G.: Solving Ordinary Differential Equations II. Springer (1996). <https://doi.org/10.1007/978-3-642-05221-7>.
18. Temme, N.M.: Digital Library of Mathematical Functions: Chapter 7 Error Functions, Dawson's and Fresnel Integrals, <https://dlmf.nist.gov/7>, pridobljeno 2024/06/15.
19. Temme, N.M.: Digital Library of Mathematical Functions: Chapter 6 Exponential, Logarithmic, Sine, and Cosine Integrals, <https://dlmf.nist.gov/6>, pridobljeno 2024/06/15.