Numerična matematika v programskem jeziku Julia

Martin Vuk

2024

Predgovor

Knjige o numerični matematiki se pogosto posvečajo predvsem matematičnim vprašanjem. Pričujoča knjiga poskuša nasloviti bolj praktične vidike numerične matematike, zato so primeri, če je le mogoče, povezani s problemom praktične narave s področja fizike, matematičnega modeliranja ali računalništva. Za podrobnejši matematični opis uporabljenih metod in izpeljav bralcu priporočam učbenik Osnove numerične matematike Bojana Orla [1].

Pričujoča 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 bi želeli 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 podrobnejša navodila, 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.

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 kode, ki reši določen problem in še ustreza minimalnim standardom pisanja kvalitetne kode. Pogosto je izpuščeno preverjanje ali implementacija robnih primerov, včasih tudi obravnava pričakovanih napak. Da je bralcu lažje razumeti, kaj koda počne, sem dal prednost berljivosti pred kompletnostjo.

Na tem mestu bi se rad zahvalil Bojanu Orlu, Emilu Žagarju, Petru Kinku in 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. Prav tako bi se zahvalil članom Laboratorija za matematične metode v računalništvu in informatiki, še posebej Neži Mramor-Kosta in Damirju Franetiču, ki so tako ali drugače prispevali k nastanku te knjige. Moja draga žena Mojca Vilfan je opravila delo urednika, za kar sem ji izjemno hvaležen. 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.

avtor Martin Vuk

Kazalo

1 Uvod v programski jezik Julia 5
1.1 Namestitev in prvi koraki 5
1.2 Avtomatsko posodabljanje kode 11
1.3 Priprava korenske mape 12
1.4 Vodenje različic s programom Git 13
1.5 Priprava paketa za vajo 13
1.6 Koda 14
1.7 Testi 15
1.8 Dokumentacija 17
1.9 Zaključek 21
2 Računanje kvadratnega korena 22
2.1 Naloga 22
2.2 Izbira algoritma 22
2.3 Določitev števila korakov 24
2.4 Izbira začetnega približka 26
2.5 Zaključek 29
3 Tridiagonalni sistemi 31
3.1 Naloga 31
3.2 Tridiagonalne matrike 31
3.3 Reševanje tridiagonalnega sistema 33
3.4 Slučajni sprehod 33
3.5 Pričakovano število korakov 35
3.6 Rešitve 39
4 Minimalne ploskve 43
4.1 Naloga 43
4.2 Matematično ozadje 44
4.3 Diskretizacija in linearni sistem enačb 44
4.4 Matrika sistema linearnih enačb 45
4.5 Izpeljava sistema s Kronekerjevim produktom 46
4.6 Numerična rešitev z LU razcepom 47
4.7 Napolnitev matrike ob eliminaciji 50
4.8 Iteracijske metode 51
4.9 Rešitve 54
5 Interpolacija z implicitnimi funkcijami 57
5.1 Naloga 57
5.2 Interpolacija z radialnimi baznimi funkcijami 57
5.3 Program 59
5.4 RBF s kompaktnim nosilcem 61
5.5 Rešitve 61
6 Fizikalna metoda za vložitev grafov 63
6.1 Naloga 63
6.2 Ravnovesje sil 63
6.3 Metoda konjugiranih gradientov 64
6.4 Krožna lestev 65
6.5 Mreža 66
6.6 Rešitve 68

7 Invariantna porazdelitev Markovske verige ... 72
7.1 Naloga ... 72
7.2 Invariantna porazdelitev Markovske verige ... 72
7.3 Potenčna metoda ... 72
7.4 Razvrščanje spletnih strani ... 73
7.5 Skakanje konja po šahovnici ... 74
7.6 Rešitve ... 77
8 Spektralno razvrščanje v gruče ... 79
8.1 Podobnostni graf in Laplaceova matrika ... 79
8.2 Algoritem ... 79
8.3 Primer ... 80
8.4 Inverzna potenčna metoda ... 80
8.5 Algoritem k-povprečij ... 82
8.6 Literatura ... 82
9 Konvergenčna območja nelinearnih enačb ... 83
9.1 Naloga ... 83
9.2 Newtonova metoda za sisteme enačb ... 83
9.3 Konvergenčno območje ... 84
9.4 Rešitve ... 85
10 Nelinearne enačbe v geometriji ... 86
10.1 Naloga ... 86
11 Aproksimacija z linearnim modelom ... 87
11.1 Naloga ... 87
12 Interpolacija z zlepki ... 88
12.1 Naloga ... 88
13 Porazdelitvena funkcija normalne porazdelitve ... 89
13.1 Naloga ... 89
13.2 Aproksimacija s polinomi Čebiševa ... 89
13.3 Čebiševa aproksimacija funkcije Φ za majhne x ... 91
13.4 Izračun funkcije $\Phi(x)$ na $[c, \infty)$ ... 92
14 Povprečna razdalja med dvema točkama na kvadratu ... 93
14.1 Naloga ... 93
15 Avtomatsko odvajanje z dualnimi števili ... 94
15.1 Naloga ... 94
16 Reševanje začetnega problema za NDE ... 95
16.1 Hermitova interpolacija ... 95
16.2 Poševni met z zračnim uporom ... 95
16.3 Rešitve ... 95
17 Aproksimacija podatkov z dinamičnim modelom ... 97
17.1 Naloga ... 97
18 Domače naloge ... 98
18.1 Navodila za pripravo domačih nalog ... 98
18.2 1. domača naloga ... 102
18.3 2. domača naloga ... 108
18.4 3. domača naloga ... 112
Literatura ... 115

1 Uvod v programski jezik Julia

V knjigi bomo uporabili programski jezik Julia. Zavoljo učinkovitega izvajanja, uporabe dinamičnih tipov in funkcij, 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 programskem jeziku Julia,
- ustvariti prvi paket in
- ustvariti prvo poročilo v formatu PDF.

Tekom te vaje bomo pripravili svoj prvi paket v Juliji, 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

Namestite programski jezik Julia, tako da sledite navodilom, in 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, izvede 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 programskem 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 uporabljamo kot vsako drugo spremenljivko. Lahko jih podamo kot argumente drugim funkcijam in jih združujemo v podatkovne strukture, kot so seznami, vektorji ali matrike. Funkcije lahko definiramo 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 uporabljamo 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 programskem jeziku Julia ima lahko več različnih definicij, glede na kombinacijo tipov argumentov, ki jih podamo. Posamezno definicijo funkcije 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 od 2. do zadnje komponente vektorja
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]
2×3 Matrix{Int64}:
 1  2  3
 4  5  6
```

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

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

julia> M[2:end, 1:end-1]
1×2 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\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
  4.499999999999999
```

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 tudi za iskanje rešitve pre-določenega 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 Podatkovni tipi

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

```
julia> struct Tocka
         x
         y
       end
```

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

```
julia> T = Tocka(1, 2) # ustvari vrednost tipa Tocka
Tocka(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 `::`. Za primer definirajmo funkcijo, ki izračuna razdaljo med dvema točkama:

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

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

1.1.4 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 `ImeModula`, uporabiti izven modula, moramo modul naložiti z `using ImeModula`. 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 ImeModula`. V tem primeru moramo vsem funkcijam iz modula dodati ime modula in piko kot predpono.

1.1.5 Paketi

Nabor funkcij, ki so na voljo v Juliji, je omejen, zato pogosto uporabimo knjižnice, ki vsebujejo dodatne funkcije. Knjižnica funkcij v Juliji 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 Julije, 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
```

Če 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

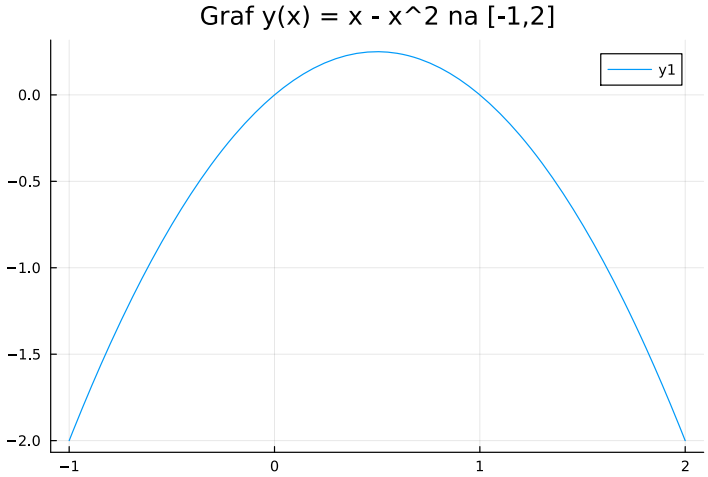
Julia ukazna zanka (REPL) pozna več načinov, ki so namenjeni različnim opravilom.

- Osnovni način s pozivom `julia>` je namenjen vnosu kode v Juliji.
- 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(⌫).

Za primer si oglejmo, kako namestiti knjižnico za ustvarjanje slik in grafov Plots.jl. 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, title="Graf y(x) = x - x^2 na [-1,2]")
```



1.1.6 Datoteke s kodo

Kodo lahko zapišemo tudi v datoteke. Vnašanje ukazov v interaktivni zanki je uporabno za preproste ukaze, na primer namesto kalkulatorja, za resnejše delo pa 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 Julijo

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 odprete 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 datoteko ponovno naložiti z ukazom `include`. Paket Revise.jl poskrbi za to, da se nalaganje zgodi avtomatično vsakič, ko se datoteke spremenijo. Zato najprej namestimo paket Revise in poskrbimo, da se zažene ob vsakem zagonu interaktivne zanke.

Naslednji ukazi namestijo paket Revise, ustvarijo mapo `$HOME/.julia/config` in datoteko `startup.jl`, ki naloži paket Revise in se izvede 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 `nummat`. Ustvarimo jo z ukazom:

```
$ mkdir nummat
```

Korenska mapa bo služila kot projektno okolje, v katerem bodo zabeleženi vsi paketi, ki jih bomo potrebovali.

```
$ cd nummat
$ julia
```

```
julia> # s pritiskom na ] vključimo paketni način
(@v1.10) pkg> activate . # pripravimo projektno okolje v korenski mapi
(nummat) pkg>
```

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

Projektno okolje v Juliji

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 rabili v nadaljevanju. Zaenkrat je to le paket Plots.jl, ki ga uporabljamo za risanje grafov:

```
(nummat) pkg> add Plots
```

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

```
[deps]
Plots = "91a5bcdd-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 `nummat` 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 jo bo napisal pri spremljanju te knjige.

Git repozitorij lahko hranimo zgolj lokalno na lastnem računalniku, lahko pa ga repliciramo na lastnem strežniku ali na enem od javnih spletnih skladišč programske kode, na primer Github ali Gitlab.

Z naslednjim ukazom v mapi `nummat` 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` lahko 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 razdelitvi dela na majhne zaključene probleme, ki so lažje obvladljivi.

1.5 Priprava paketa za vajo

Ob začetku vsake vaje bomo v korenski mapi (`nummat`) 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 ponavljajoča 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 `nummat` ustvarimo paket `Vaja01`, v katerega bomo shranili kodo. Nov paket ustvarimo v paketnem načinu z ukazom `generate`:

```
$ cd nummat
$ julia
```

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

Ukaz `generate` ustvari mapo `Vaja01` z osnovno strukturo paketa v Juliji:

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

1 directory, 2 files
```

Paket `Vaja01` nato dodamo v projektno okolje v korenski mapi `nummat`, da bomo lahko kodo iz paketa uporabili v programih in ukazni zanki:

```
(@v1.10) pkg> activate .
(nummat) pkg> develop ./Vaja01 # paket dodamo projektnemu okolju
```

> *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 projekte ustvarjali s `Pkg.generate`.

Osnovna struktura paketa je pripravljena. Paketu bomo v nadaljevanju dodali še:
- kodo (Poglavje 1.6),
- teste (Poglavje 1.7) in
- dokumentacijo (Poglavje 1.8).

1.6 Koda

Ko je mapa s paketom `Vaja01` pripravljena, lahko začnemo. Napisali bomo funkcije, ki izračunajo koordinate Geronove lemniskate:

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

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 imena funkcij, da so dostopna brez predpone `Vaja01`
export lemniskata_x, lemniskata_y
end # module Vaja01
```

Program 1: Definicije funkcij v paketu `Vaja01`

Funkcije iz datoteke `Vaja01/src/Vaja01.jl` lahko 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 lahko sedaj pokličemo novo definirani funkciji:

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

V datoteko `Vaja01/doc/01uvod.jl` bomo zapisali 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 zaporedje 300 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")
```

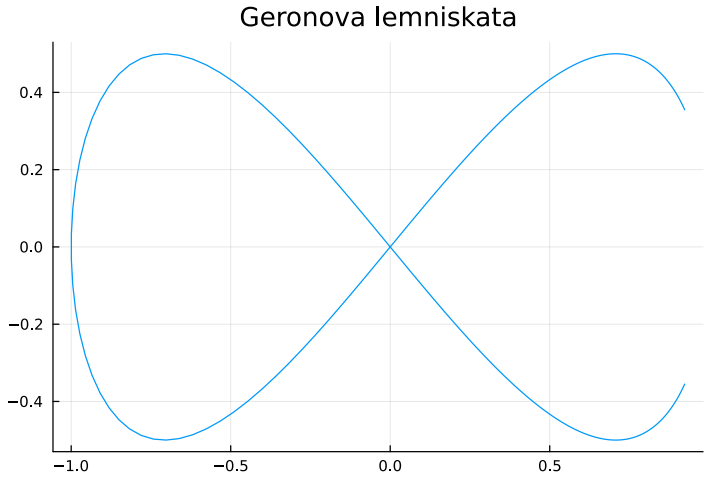
Program `01uvod.jl` poženemo z ukazom:

```
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. Če pritisnete kombinacijo tipk `Shift + Enter`, se bo izvedla vrstica, v kateri je trenutno kazalka.

Rezultat je slika lemniskate.



Slika 2: Geronova lemniskata

1.7 Testi

Naslednji korak je, da dodamo avtomatske teste, s katerimi preizkusimo pravilnost kode, ki smo je napisali v prejšnjem poglavju. Avtomatski test je preprost program, ki pokliče določeno funkcijo in preveri rezultat.

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 dovzetno 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 avtomatizacijo 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 teste za obe koordinatni funkciji, ki primerjajo 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 3: Rezultat funkcij primerjamo s pravilno vrednostjo

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, pride do zaokrožitvenih napak. Zato se različni načini izračuna za isto število praviloma razlikujejo na zadnjih decimalkah. Na primer izraz `asin(sin(pi/4)) - pi/4` ne vrne točne ničle ampak vrednost `-1.1102230246251565e-16`, ki pa je zelo majhno število. Za približno primerjavo dveh vrednosti `a` in `b` zato uporabimo izraz

$$|a - b| < \varepsilon, \tag{1.2}$$

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

Preden lahko poženemo teste, moramo ustvariti 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`.

```
(nummat) 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 generira avtomatično 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 Juliji 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 Julijo.

1.8.2 README dokument

Dokument README (preberi me) je namenjen najbolj osnovnim 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 Julije.

## 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 4: README.md 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 ([2]). Za pisanje besedila bomo uporabili format Markdown, ki ga bomo dodali kot komentarje v kodi.

Za generiranje PDF dokumentov je potrebno 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 zaporedje 300 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 5: Vrstice, ki se začnejo z znakoma `#'`, so v formatu Markdown

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

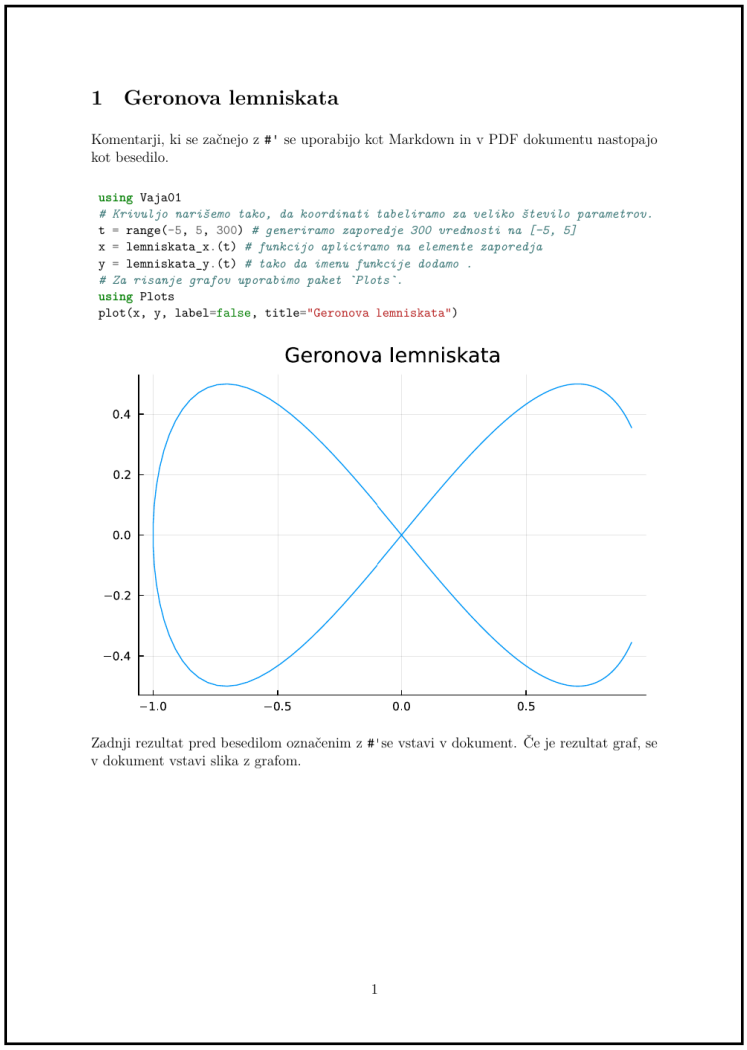
```
using Weave
# Poročilo generiramo z ukazom `Weave.weave`
Weave.weave("Vaja01/doc/01uvod.jl",
  doctype="minted2pdf", out_path="Vaja01/pdf")
```

Program 6: Program za pripravo PDF dokumenta

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

```
(nummat) 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 zaporedje 300 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.

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 Juliji:

- IJulia,
- Literate.jl in
- Quadro.

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

- TeX/LaTeX,
- pandoc,
- AsciiDoctor,
- 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 ime za 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 nadaljujete, ponovno preverite, če vse deluje tako, kot bi moralo. V Juliji aktivirajte projektno okolje:

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

Nato najprej poženemo teste:

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

Na koncu pa poženemo še program `01uvod.jl`:

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

in pripravimo poročilo:

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

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

2 Računanje kvadratnega korena

Računalniški procesorji navadno implementirajo le osnovne številske operacije: seštevanje, množenje in deljenje. Za računanje 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 korensko funkcijo.

Implementacija elementarnih funkcij v Juliji

Lokacijo metod, ki računajo določeno funkcijo, lahko dobite z ukazoma `methods` in `@which`. Tako bo ukaz `methods(sqrt)` izpisal 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 = 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 Juliji).
- 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.

```
(nummat) pkg> generate Vaja02
(nummat) 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. 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

$$y^2 = x. \tag{2.1}$$

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

$$f(x) = 0 \tag{2.2}$$

poiščemo z rekurzivnim zaporedjem približkov

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

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. Premaknemo vse člene na eno stran, da je na drugi strani nič

$$y^2 - x = 0. \tag{2.4}$$

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

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

Rekurzivno formulo (2.5) imenujemo Haronov 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>   x = 1.5
         for n = 1:5
           x = (x + 2 / x) / 2
           println(x)
         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 spreminjalo in smo dosegli najboljši približek, kot ga lahko predstavimo s 64 bitnimi števili s plavajočo vejico.

Napišimo zgornji program še kot funkcijo. Da lažje spremljamo, 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 7: 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 sta si bomo podrobneje 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.

Relativna in absolutna napaka

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

- 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

$$\mid a - b \mid < \delta(\max(|a|, |b|)). \tag{2.6}$$

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 10 signifikantnih mest pravilnih, 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 lahko 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 $y^2 = x$. 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 lahko 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 (oziroma 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 8: Metoda `koren(x, y0)`, ki avtomatsko določi število korakov iteracije

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. Problem je, da 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. Poskusimo lahko za začetni približek uporabiti kar samo število x. Malce boljši približek dobimo s Taylorjevem razvojem korenske funkcije okrog števila 1

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

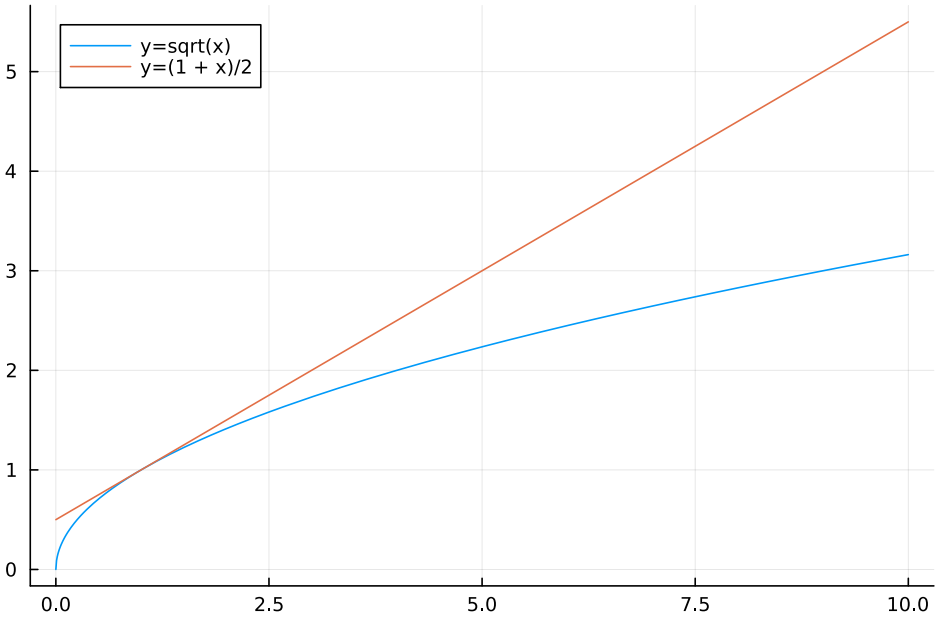
Opazimo, da za večja števila, iteracija potrebuje več korakov:

```
julia>   tangenta(x) = 0.5 + x / 2
         y = koren(10, tangenta(10))
         y = koren(200, tangenta(200))

[ Info: Število korakov 5
[ Info: Število korakov 7
14.142135623730955
```

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=(1 + x)/2")
```



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

Za boljši približek, si pomagamo z načinom predstavitve števil v računalniku. Realna števila predstavimo s števili s plavajočo vejico. Število je zapisano v obliki

$$x = m2^e, \tag{2.13}$$

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

$$\sqrt{x} = \sqrt{m} \cdot 2^{\frac{e}{2}}. \tag{2.14}$$

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

$$\sqrt{m} = \frac{1}{2} + \frac{m}{2}. \tag{2.15}$$

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

$$\sqrt{2^e} \approx 2^d \cdot \begin{cases} 1; & o = 0 \\ \sqrt{2}; & o = 1. \end{cases} \tag{2.16}$$

Formula za približek je enaka:

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

Potenco števila 2^n lahko izračunamo s premikom binarnega zapisa števila 1 v levo za n mest. V Juliji za levi premik uporabimo operator `<<`, s funkcijama `exponent` in `significand` pa dobimo eksponent in mantiso števila s plavajočo vejico. Tako lahko zapišemo naslednjo funkcijo za začetni približek:

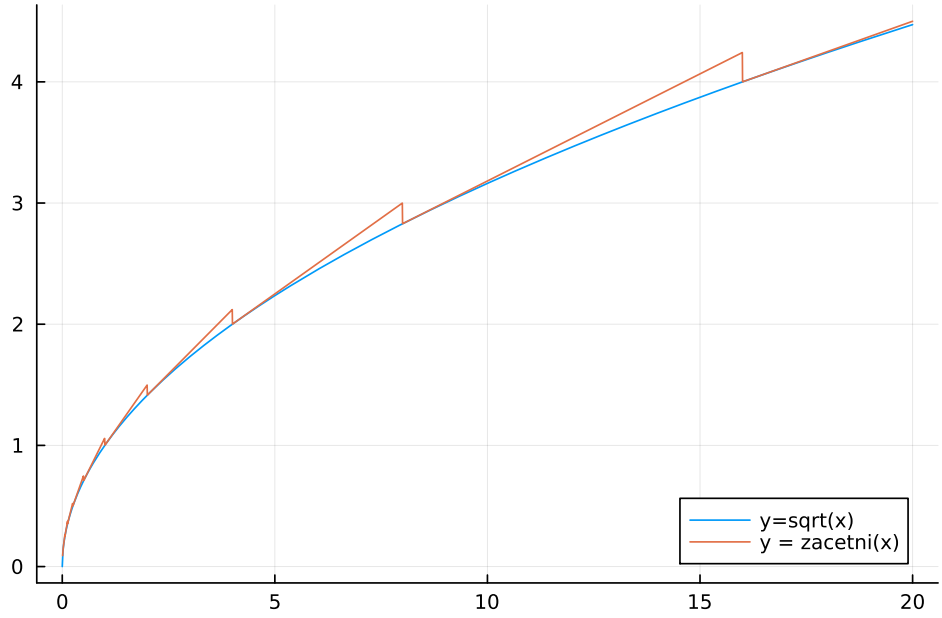
```
"""
  y0 = zacetni(x)

Izračunaj začetni približek za kvadratni koren števila `x` z uporabo
eksponenta za števila s plavajočo vejico.
"""
function zacetni(x)
  d, ost = divrem(abs(exponent(x)), 2)
  m = significand(x) # mantisa
  koren2ost = (ost == 0) ? 1 : 1.4142135623730951 # koren(2^ost)
  koren2e = (1 << d) * koren2ost # koren(2^e)
  if x > 1
    return (0.5 + m / 2) * koren2e
  else
    return (0.5 + m / 2) / koren2e
  end
end
```

Program 9: Funkcija `zacetni(x)`, ki izračuna začetni približek

Primerjajmo izboljšano verzijo začetnega približka s pravo korensko funkcijo.

```
plot(sqrt, 0, 20, label="y=sqrt(x)")
plot!(Vaja02.zacetni, 0, 20, label="y = zacetni(x)")
```



Slika 5: Korenska funkcija in izboljšani začetni približek

2.5 Zaključek

Ko smo enkrat izbrali dober začetni približek, tudi 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, zacetni(x))
```

Program 10: Funkcija `koren(x)`

Julia omogoča več definicij iste funkcije

Julia uporablja posebno vrsto polimorfizma imenovano večlična razdelitev (angl. multiple dispatch). Večlična razdelitev omogoča, da za isto funkcijo definiramo več različic, ki se uporabijo glede na to, katere argumente podamo 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 8, ko pa pokličemo `koren(2.0)`, se izvede Program 10.

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

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

```
julia>   koren(10.0), koren(200.0), koren(2e10)

[ Info: Število korakov 3
[ Info: Število korakov 3
[ Info: Število korakov 2
(3.162277660168379, 14.142135623730965, 141421.35623853415)
```

Hitro računanje obratne vrednosti kvadratnega korena

Pri razvoju računalniških iger, ki poskušajo verno prikazati 3-dimenzionalni svet na zaslonu, se veliko uporablja normiranje vektorjev. Pri normiranju je treba komponente vektorja deliti z normo vektorja, ki je enaka korenu vsote kvadratov komponent. Kot smo spoznali pri računanju kvadratnega korena s Heronovim obrazcem, je posebej problematično poiskati ustrezen 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.

3 Tridiagonalni sistemi

3.1 Naloga

- Ustvari podatkovni tip za tridiagonalno matriko ter implementiraj operacije množenja * z vektorjem in reševanja sistema $Ax = b$ \.
- 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 oddalji od izhodišča 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:

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

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

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

Z implementacijo posebnega podatkovnega tipa za tridiagonalno matriko lahko 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:

```
(nummat) pkg> generate Vaja03
(nummat) 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 dimenzije diagonal.")
    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 lahko 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.

Implementirajmo naslednje metode specifične za tip `Tridiag`:

- `size(T::Tridiag)` vrne dimenzije matrike (Program 14),
- `getindex(T::Tridiag, i, j)` vrne element `T[i,j]` (Program 15),
- `setindex!(T::Tridiag, x, i, j)` nastavi element `T[i,j]` (Program 16) in
- `*(T::Tridiag, x::Vector)` izračuna produkt matrike `T` z vektorjem `x` (Program 17).

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`:

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

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

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

Podobno definiramo teste še za druge funkcije. Primeri testov so v poglavju Poglavje 3.6.1.

3.3 Reševanje tridiagonalnega sistema

Poiskali bomo rešitev sistema linearnih enačb $Tx = b$, kjer je matrika sistema T tridiagonalna. Sistem lahko rešimo z Gaussovo eliminacijo in obratnim vstavljanjem (glej učbenik [1]). Ker je v tridiagonalni 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. V nasprotnem primeru se tridiagonalna oblika matrike med Gaussovo eliminacijo podre in se algoritem nekoliko zakomplicira. Za diagonalno dominantne matrike po stolpcih pri Gaussovi eliminaciji pivotiranje ni potrebno.

Časovna zahtevnost Gaussove eliminacije brez pivotiranja je za tridiagonalni sistem $Tx = b$ 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 linearno $\mathcal{O}(n)$.

Priredimo splošna algoritma Gaussove eliminacije in obratnega vstavljanja, da bosta upoštevala lastnosti tridiagonalnih matrik. Napišimo funkcijo `\`:

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

ki poišče rešitev sistema $Tx = b$ (rešitev je Program 18). V datoteko `Vaja03/test/runtests.jl` dodajte 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 enako 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 $n + 1$. Prehodne verjetnosti slučajnega sprehoda so enake:

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

Definicija Markovske verige

Markovska veriga je zaporedje slučajnih spremenljivk

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

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

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

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.

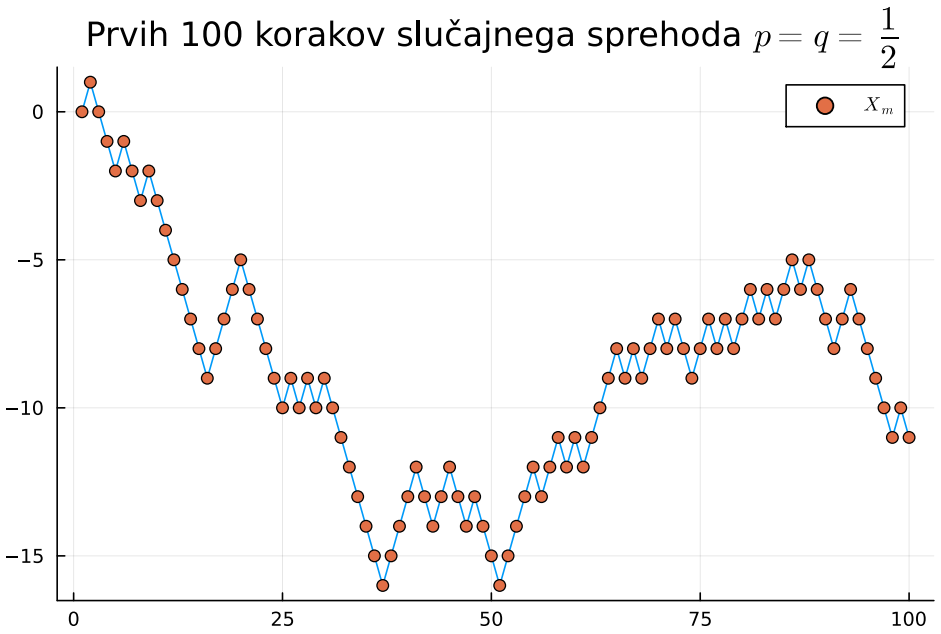
Verjetnosti $P(X_{i+1} = x \mid X_i = x_i)$ imenujemo *prehodne verjetnosti* Markovske verige. V nadaljevanju bomo privzeli, da so prehodne verjetnosti enake za vse korake k:

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

Simulirajmo prvih 100 korakov slučajnega sprehoda

```
""" Simuliraj `n` korakov slučajnega sprehoda s prehodno 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 $\{x_1, x_2, ...x_n\}$, lahko prehodne verjetnosti zložimo v matriko. Brez škode lahko stanja $\{x_1, x_2, ...x_n\}$ nadomestimo z naravnimi števili $\{1, 2, ...n\}$. Matriko P, katere elementi so prehodne verjetnosti prehodov med stanji Markovske verige

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

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

$$\sum_{j=1}^{n} p_{ij} = 1. \tag{3.8}$$

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

$$P\mathbf{1} = \mathbf{1}. \tag{3.9}$$

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

$$P(X_m = j|X_1 = i). \tag{3.10}$$

3.5 Pričakovano število korakov

Poiskati želimo pričakovano število korakov, ko 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:

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

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

Predpostavimo lahko, da je začetno stanje enako 0. Iščemo pričakovano število korakov, ko se slučajni sprehod prvič pojavi v stanju k ali $-k$. Zanemarimo stanja, ki so več kot k oddaljena od izhodišča in stanji k in $-k$ spremenimo v absorbirajoči stanji. Obravnavamo torej absorbirajočo verigo z $2k+1$ stanji, 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

$$P = \begin{pmatrix} Q & T \\ 0 & I \end{pmatrix}, \tag{3.12}$$

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

Vsoto vseh potenc matrike Q

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

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:

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

Če želimo poiskati pričakovano število korakov $|(\boldsymbol{m})$, moramo rešiti sistem linearnih enačb:

$$(I - Q)\ |(\boldsymbol{m}) = \mathbf{1}. \tag{3.15}$$

Če nas zanima, kdaj bo sprehod prvič za k oddaljen od izhodišča, lahko začnemo v 0 in stanji k in $-k$ proglasimo za absorpcijska 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:

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

Matrika $I - Q$ je tridiagonalna in po stolpcih diagonalno dominantna, 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 11: Sestavi tridiagonalno matriko $I - Q$ za slučajni sprehod, ki se konča, ko se prvič oddalji za k korakov od izhodišča

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

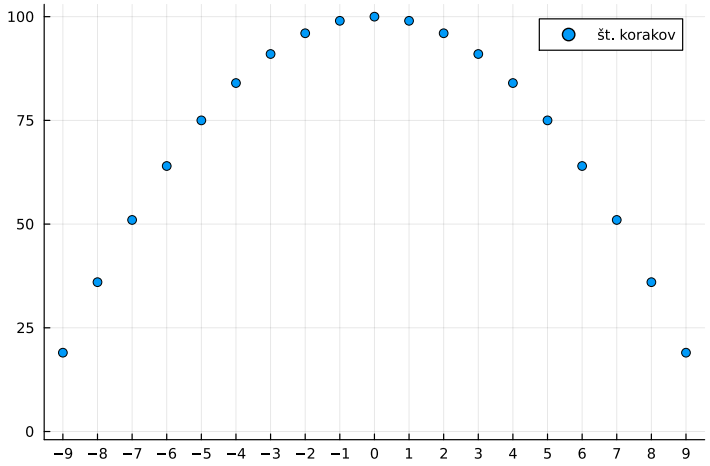
```
"""
    Em = koraki(k, p)

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 12: Izračunaj 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 $-k, -k + 1, \ldots 0 \ldots k$. Komponente vektorja $\boldsymbol{k}$ tako predstavljajo pričakovano število korakov, ki jih slučajni sprehod potrebuje, da prvič doseže stanji $-k$ ali k, če začnemo v stanju $i \in \{-k + 1, -k + 2, \ldots 0, 1, \ldots k - 1\}$.

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



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

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 13: 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 rezultat ujema z vzorčnim povprečjem po velikem številu sprehodov.

```
  Random.seed!(691)
  n = 100_000
  k, p = 10, 0.5
  kp = sum([simuliraj_korake(k, p) for _ in 1:n]) / n
  println("Vzorčno povprečje za vzorec velikosti $n je $kp.")
```

```
Vzorčno povprečje za vzorec velikosti 100000 je 100.09526
```

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 14: Metoda `size` 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 15: Metoda `getindex` 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 16: 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("Dimenzije 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 17: 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 18: Reševanje tridiagonalnega sistema linearnih enačb

3.6.1 Testi

```
@testset "Dostop do elementov" begin
  T = Tridiag([1, 2], [3, 4, 5], [6, 7])
  # diagonala
  @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 19: 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 20: 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 21: Testi za množenje

```
@testset "Reševanje sistema" begin
```

Program 22: Testi za operator \ (reševanje tridiagonalnega sistema)

4 Minimalne ploskve

4.1 Naloga

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 in internetu hitro razkrije, da ploskve, ki tako nastanejo, sodijo med minimalne ploskve, ki so burile domišljijo mnogih matematikov in nematematikov. Minimalne ploskve so navdihovale tudi umetnike npr. znanega arhitekta Frei Otto, ki je sodeloval pri zasnovi Muenchenskega olimpijskega stadiona, kjer ima streha obliko minimalne ploskve.

Slika 8: Streha olimpijskega stadiona v Münchnu (vir wikipedia)

Namen te vaje je primerjava eksplicitnih in iterativnih metod za reševanje linearnih sistemov enačb. Prav tako se bomo naučili, kako zgradimo matriko sistema in desne strani enačb za spremenljivke, ki niso podane z vektorjem ampak kot elementi matrike. V okviru te vaje 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) in reši sistem enačb direktno na elementih matrike višinskih vrednosti ploskve brez eksplicitne uporabe matrike sistema.
- Nariši primer minimalne ploskve.
- Animiraj konvergenco iterativnih metod.

4.2 Matematično ozadje

Ploskev lahko predstavimo s funkcijo dveh spremenljivk $u(x, y)$, ki predstavlja višino ploskve nad točko (x, y). Naša naloga je poiskati približek za funkcijo $u(x, y)$ na pravokotnem območju.

Funkcija $u(x, y)$, ki opisuje milno opno, zadošča matematična enačbi

$$\Delta u(x, y) = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = \rho(x, y), \tag{4.1}$$

znani pod imenom Poissonova enačba. Diferencialni operator

$$\Delta u(x, y) = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \tag{4.2}$$

imenujemo Laplaceov operator.

Funkcija $\rho(x, y)$ je sorazmerna tlačni razliki med zunanjo in notranjo površino milne opne. Tlačna razlika je lahko posledica višjega tlaka v notranjosti milnega mehurčka ali pa teže milnice. Če tlačno razliko zanemarimo, dobimo Laplaceovo enačbo:

$$\Delta u(x, y) = 0. \tag{4.3}$$

Vrednosti $u(x, y)$ na robu območja so določene z obliko zanke, medtem ko za vrednosti v notranjosti velja enačba (4.3). Enačbo, ki vsebuje parcialne odvode, imenujemo parcialna diferencialna enačba ali s kratico PDE. Rešitev PDE je funkcija več spremenljivk, ki zadošča enačbi. Problem za diferencialno enačbo, pri katerem so podane vrednosti na robu, imenujemo robni problem. Ker je oblika milnice določena na robu, lahko iskanje oblike milnice prevedemo na robni problem za Laplaceovo PDE na območju omejenem s tlorisom žične zanke.

V nadaljevanju predpostavimo, da je območje pravokotnik $[a, b] \times [c, d]$. Poleg Laplaceove enačbe (4.3), veljajo za vrednosti funkcije $u(x, y)$ tudi *robni pogoji*:

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

kjer so f_s, f_z, f_l in f_d dane funkcije. Rešitev robnega problema je tako odvisna od območja, kot tudi od robnih pogojev.

4.3 Diskretizacija in linearni sistem enačb

Problema se bomo lotili numerično, zato bomo vrednosti $u(x, y)$ poiskali le v končno mnogo točkah: problem bomo diskretizirali. 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 in dobimo zaporedje koordinat

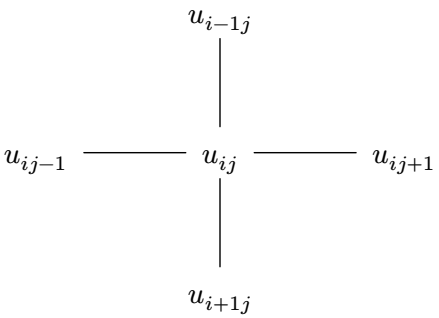
$$\begin{aligned} a = x_0, x_1, \dots \ x_{n+1} &= b \\ c = y_0, \ y_1, \dots y_{m+1} &= d, \end{aligned} \tag{4.5}$$

ki definirajo pravokotno mrežo točk (x_i, y_j). Namesto funkcije $u : [a, b] \times [c, d] \to \mathbb{R}$ tako iščemo le vrednosti

$$u_{ji} = u(x_i, y_j), \quad i = 1, ...n, \quad j = 1, ...m \tag{4.6}$$

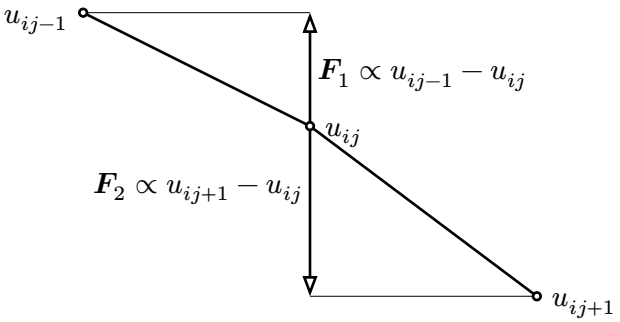
Elemente matrike u_{ji} določimo tako, da je v limiti, ko gre razmik med vozliči proti 0, izpolnjena Laplaceova enačba (4.3).

Laplaceovo enačbo lahko diskretiziramo s končnimi diferencami. Lahko pa dobimo navdih pri arhitektu Frei Otto, 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 4 sosednjimi vozlišči.



Slika 9: Sosednje vrednosti vozlišča (i, j).

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



Slika 10: Sile elastik iz sosednjih vozlišč $(i, j-1)$ in $(i, j+1)$ na vozlišče (i, j).

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_i, y_j, u_{ij}) enačbo

$$\begin{aligned} (u_{i-1j} - u_{ij}) + (u_{ij-1} - u_{ij}) + (u_{i+1j} - u_{ij}) + (u_{ij+1} - u_{i(i,j)}) &= 0 \\ u_{i-1j} + u_{ij-1} - 4u_{ij} + u_{i+1j} + u_{ij+1} &= 0. \end{aligned} \tag{4.7}$$

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

4.4 Matrika sistema linearnih enačb

Sisteme linearnih enačb običajno zapišemo v matrični obliki

$$A\boldsymbol{x} = \boldsymbol{b}, \tag{4.8}$$

kjer je A kvadratna matrika, $\boldsymbol{x}$ in $\boldsymbol{b}$ pa vektorja. Spremenljivke u_{ij} moramo nekako razvrstiti v vektor $\boldsymbol{x} = [x_1, x_2, ...]^T$. Najpogosteje elemente u_{ij} razvrstimo v vektor $\boldsymbol{x}$ po stolpcih, tako da je

$$\boldsymbol{x} = [u_{11}, u_{21} \ldots u_{n1}, u_{12}, u_{22} \ldots u_{1n} \ldots u_{m-1n}, u_{mn}]^T. \tag{4.9}$$

Za $n = m = 3$ dobimo 9×9 matriko

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

ki je sestavljena iz 3×3 blokov

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

in desne strani

$$\begin{aligned} \boldsymbol{b} = -[&u_{01} + u_{10}, u_{20} \ldots u_{n0} + u_{n+11}, \\ &u_{02}, 0 \ldots u_{n+1,2}, u_{03}, 0 \ldots u_{nm+1}, u_{nm+1} + u_{n+1m}]^T. \end{aligned} \tag{4.12}$$

Razvrstitev po stolpih in operator vec

Eden od načinov, kako lahko elemente matrike razvrstimo v vektor, je tako, da stolpce matrike enega za drugim postavimo v vektor. Indeks v vektorju k lahko izrazimo z indeksi i, j v matriki s formulo

$$k = i + (m-1)j. \tag{4.13}$$

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

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

4.5 Izpeljava sistema s Kronekerjevim produktom

Množenje vektorja $\boldsymbol{x} = \text{vec}(U)$ z matriko A lahko prestavimo kot množenje matrike U z matriko L z leve in desne:

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

kjer je L Laplaceova matrika v eni dimenziji, ki ima -2 na diagonali in 1 na spodnji pod-diagonali in zgornji nad-diagonali:

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

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 vsebuje vsoto sil sosednjih vozlišč v smeri y, medtem ko množenje z desne vsebuje vsoto sil sosednjih vozlišč v smeri x. Element produkta $LU + UL$ na mestu (i, j) je enak:

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

kar je enako desni strani enačbe (4.7).

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

$$\begin{aligned} A \cdot \mathrm{vec}(U) &= \mathrm{vec}(LU + UL) = \mathrm{vec}(LUI + IUL) \\ A^{N,N} &= L^{m,m} \otimes I^{n,n} + I^{m,m} \otimes L^{n,n}. \end{aligned} \tag{4.18}$$

Kroneckerjev produkt in operator vec *v Juliji*

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

4.6 Numerična rešitev z LU razcepom

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

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

Nato dodamo pakete, ki jih bomo potrebovali:

```
(nummat) 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`, ki določajo vrednosti na
robovih pravokotnika `y = c`, `y = d`, `x = a` in `x = b`.
"""
struct RobniProblemPravokotnik
  op # abstrakten podatkovni tip, ki opiše diferencialni operator
  meje # meje pravokotnika [a, b] x [c, d] v obliki [(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 abstrakten tip brez polj, ki predstavlja Laplaceov diferencialni operator (4.2) in ga bomo lahko doali v polje za operator v `RobniProblemPravokotnik`:

```
"""
    L = Laplace()

Ustvari abstrakten objekt tipa `Laplace`, ki predstavlja Laplaceov diferencialni
operator.
"""
struct Laplace end
```

Abstraktni podatkovni tipi
Programski jezik Julija ne pozna razredov. Uporaba abstraktnih podatkovnih tipov, kot je `Laplace`, omogoča polimorfizem. Na ta način lahko v kodo organiziramo tako, da odraža abstraktne matematične pojme, kot je v našem primeru robni problem za PDE.

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

$$\begin{aligned} u(x, 0) = u(x, \pi) = \sin(x) \\ u(0, y) = u(\pi, y) = \sin(y) \end{aligned} \tag{4.19}$$

lahko predstavimo z objektom

```
rp = RobniProblemPravokotnik(
  Laplace(),           # operator
  ((0, pi), (0, pi)),  # pravokotnik
  (sin, sin, 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 Poglavje 4.3, lahko približek za rešitev robnega problema poiščemo kot rešitev linearnega sistema enačb (4.7). Najprej napišemo funkcijo, ki generira matriko sistema:

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

za dane dimenzije notrajnje mreže `n` in `m` (za rešitev glej Program 24). Nato na robu mreže izračunamo robne pogoje in sestavimo vektor desnih strani sistema (4.7). Ker je preslikovanje dvojnega indeksa v enojni in nazaj precej sitno, bomo večino operacij naredili na matriki vrednosti $U = [u_{ij}]$ dimenzij $(m+2) \times (n+2)$, ki vsebuje tudi vrednosti na robu. Napisali bom funkcijo

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

ki vrne matriko `U0` dimenzije $(m+2) \times (n+2)$ za katero so vrednosti notranjih elementov enake 0 in vrednosti na robu podane z robnimi pogoji podanimi v robnem problemu `rp`. Poleg matrike `U` naj funkcija vrne vektorja `x` in `y`, ki vsebujeta delilne točke na intervalih $[a, b]$ in $[c, d]$.

Iz matrike `U` lahko sedaj dokaj preprosto sestavimo desne strani enačb, tako da indekse $i = 2...(m-1)$ in $j = 2...(n-1)$ zaporedoma zamaknemo v levo, desno, gor in dol in seštejemo ustrezne podmatrike. Rezultat nato spremenimo v vektor s funkcijo `vec` (za rešitev glej Program 25).

Ko imamo pripravljeno matriko in desne strani, vse skupaj zložimo 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. Rezultat nato vrne v obliki matrike vrednosti `U` in vektorjev vozlišč `x` in `y` (za rešitev glej Program 26).

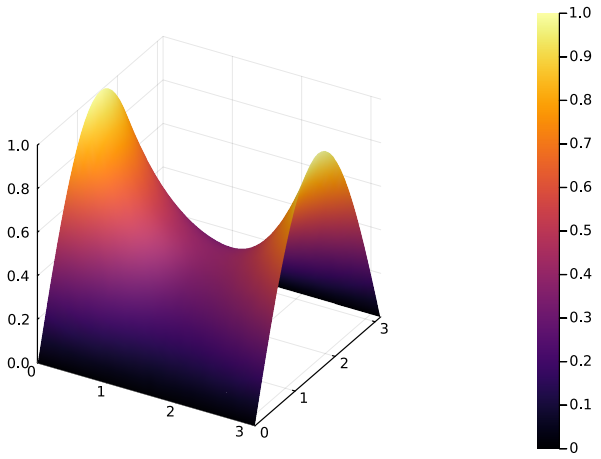
Napisane programe uporabimo za rešitev robnega problema za pravokotnik $[0, \pi] \times [0, \pi]$ z robnimi pogoji

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

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

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

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

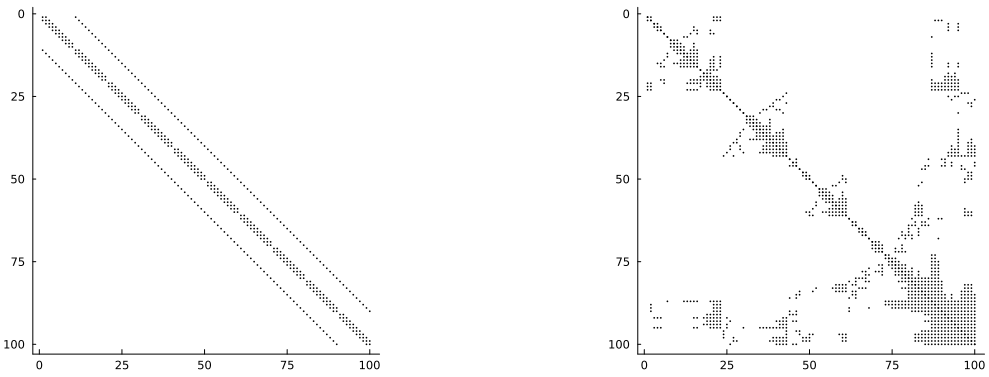


Slika 11: Rešitev robnega problema za Laplaceovo enačbo.

4.7 Napolnitev matrike ob eliminaciji

Matrika Laplaceovega operatorja ima veliko ničelnih elementov. Takim matrikam pravimo razpršene ali redke matrike. Razpršenost matirke lahko izkoristimo za prihranek prostora in časa, kot smo že videli pri tridiagonalnih matrikah v poglavju Poglavje 3. Vendar se pri LU razcepu, 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)
F = lu(A)
p2 = spy(F.L .!= 0, legend=false)
spy!(p2, F.U .!= 0, legend=false)
```



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

4.8 Iteracijske metode

V prejšnjih poglavjih smo poiskali približno obliko minimalne ploskve, tako da smo linearni sistem (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 ostane in tako lahko 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 enačbo. V primeru linearnih sistemov je rešitev enolična.

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

$$u_{ij}^{(k+1)} = \frac{1}{4}\Big(u_{ij-1}^{(k)} + u_{i-1j}^{(k)} + u_{i+1j}^{(k)} + u_{ij+1}^{(k)}\Big), \tag{4.21}$$

ki ustrezajo Jacobijevi iteraciji. Približek za rešitev tako dobimo, če zaporedoma uporabimo rekurzivno formulo (4.21).

Pogoji konvergence

Rekli boste, to je preveč enostavno, če enačbe le pruredimo in se potem rešitel 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. Jakobijeva, Gauss-Seidlova in SOR iteracija **ne konvergirajo vedno**, zagotovo pa konvergirajo, če je matrika po vrsticah diagonalno dominantna.

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

$$u_{i,j}^{(k+1)} = \frac{1}{4}\Big(u_{i,j-1}^{(k+1)} + u_{i-1,j}^{(k+1)} + u_{i+1,j}^{(k)} + u_{i,j+1}^{(k)}\Big) \tag{4.22}$$

Konvergenco še izboljšamo, če približek $u_{ij}^{(k+1)}$, ki ga dobimo z Gauss-Seidlovo metodo, malce „pokvarimo“ s približkom na prejšnjem koraku $u_{ij}^{(k)}$. Tako dobimo metodo SOR

$$\begin{aligned} u_{i,j}^{(\mathrm{GS})} &= \frac{1}{4}\Big(u_{i,j-1}^{(k+1)} + u_{i-1,j}^{(k+1)} + u_{i+1,j}^{(k)} + u_{i,j+1}^{(k)}\Big) \\ u_{i,j}^{(k+1)} &= \omega u_{i,j}^{(\mathrm{GS})} + (1-\omega)u_{i,j}^{(k)} \end{aligned} \tag{4.23}$$

Parameter ω je lahko poljubno število $(0, 2)$. 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 23: Poišči naslednji približek Gauss-Seidlove iteracije za diskretizacijo Laplaceove enačbe.

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),
```

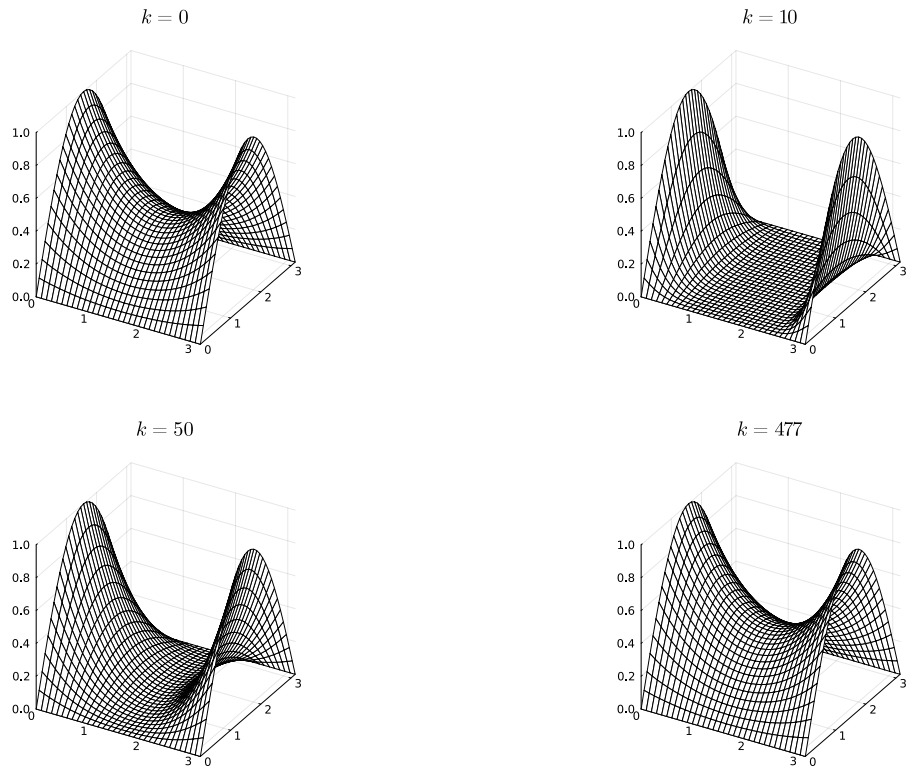
ki, računa zaporedne približke, dokler se rezultat ne spreminja več znotraj določene tolerance. Argument `korak` je funkcija, ki iz danega približka izračuna naslednjega.

Rešitve so v programih: Program 27, Program 28 in Program 29.

4.8.1 Konvergenca

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

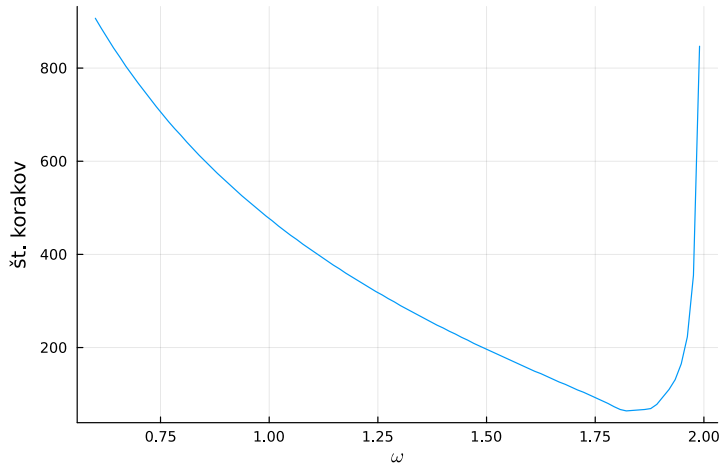
```
U0, x, y = Vaja04.diskretiziraj(rp, 0.1)
wireframe(x, y, U, legend=false, title="\$ k=0\$")
U = U0
for i = 1:10
  U = Vaja04.korak_gs(U)
end
wireframe(x, y, U, legend=false, title="\$k=10\$")
U, it = Vaja04.iteracija(Vaja04.korak_gs, U0; atol=1e-3)
wireframe(x, y, U, legend=false, title="\$k=$it\$")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



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

Za metodo SOR je hitrost konvergence odvisna od izbire parametra ω. Odvisnot 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(0.6, 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: Število korakov SOR iteracije je odvisno od parametra ω.

4.9 Rešitve

Funkcije, ki smo jih definirali v `Vaja04/src/Vaja04.jl`.

```
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 2 dimenzijah
na pravokotni mreži dimenzije` `n` krat `m`.
"""
function matrika(_::Laplace, n, m)
  return kron(laplace(n), enota(m)) + kron(enota(n), laplace(m))
end
```

Program 24: Generiraj 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((b - a) / h))
  n = Integer(floor((d - c) / h))
  U0 = zeros(n + 2, m + 2)
  fs, fz, fl, fd = rp.rp
  (a, b), (c, d) = rp.meje
  x = range(a, b, m + 2)
  y = range(c, d, n + 2)
  U0[1, :] = fl.(y)
  U0[end, :] = fd.(y)
  U0[:, 1] = fs.(x)
  U0[:, end] = fz.(x)
  return U0, x, y
end

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: Izračunaj 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(L, n, m)
  d = desne_strani(U)
  res = A \ d  # reši sistem
  U[2:end-1, 2:end-1] = reshape(res, n, m) # preoblikuj rešitev v matriko
  return U, x, y
end
```

Program 26: Poišči 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
      # Jacobi
      U[i, j] = (U0[i+1, j] + U0[i, j+1] + U0[i-1, j] + U0[i, j-1]) / 4
    end
  end
  return U
end
```

Program 27: Poišči 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
      # Gauss Seidel
      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: Poišči 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 začentim 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: Poišči približek za limito rekurzivnega zaporedja.

5 Interpolacija z implicitnimi funkcijami

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

	krivulje v 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 predstavitve krivulj v $\mathbb{R}^2$ in ploskev v $\mathbb{R}^3$.

Implicitne enačbe oblike $F(x_1, x_2, ...) = 0$ so zelo dober način za opis krivulj in ploskev. Hitri algoritmi za izračun nivojskih krivulj in ploskev kot sta korakajoče kocke in korakajoči kvadrati omogočajo učinkovito predstavitev implicitno podanih ploskev in krivulj s poligonsko mrežo. Predstavitev s predznačeno funkcijo razdalje pa 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 linearno kombinacijo radialnih baznih funkcij (RBF) ([3], [4]).

5.1 Naloga

- Definiraj podatkovni tip za linearno kombinacijo radialnih baznih funkcij (RBF). Podatkovni tip naj vsebuje središča RBF $\boldsymbol{x}_i$, funkcijo ene oblike φ in koeficiente w_i v linearni kombinaciji

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

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

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

5.2 Interpolacija z radialnimi baznimi funkcijami

V ravnini[1] je podan oblak točk $\{\boldsymbol{x}_1, ...\boldsymbol{x}_n\} \subset \mathbb{R}^2$. Iščemo krivuljo, ki dobro opiše dane točke. Če zahtevamo, da vse točke ležijo krivulji, problemu rečemo *interpolacija*, če pa dovolimo, da je krivulja zgolj blizu danih točk in ne nujno vsebuje vseh točk, problem imenujemo *aproksimacija*. Krivuljo bomo poiskali v implicitni obliki kot nivojsko krivuljo funkcije 2 spremenljivk. Za izbrano vrednost c iščemo funkcijo $f(x, y)$, za katero velja

[1]Postopek, ki ga bomo opisali, deluje ravno tako dobro tudi za točke v prostoru. Vendar se bomo zavoljo enostavnosti omejili na točke v ravnini

$$f(x_i, y_i) = c \tag{5.3}$$

za vse točke v danem oblaku točk. Problem bomo rešili malce bolj splošno. Denimo, da imamo za vsako dano točko v oblaku, podano tudi vrednost funkcije $\{f_1, ..., f_n\}$. Iščemo zvezno funkcijo $f(x, y)$, tako da so izpolnjene enačbe:

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

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

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

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

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

$$f(\boldsymbol{x}) = \varphi(\|\boldsymbol{x} - \boldsymbol{x}_0\|) \tag{5.6}$$

Uporabljajo se za interpolacijo ali aproksimacijo podatkov s funkcijo oblike

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

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 $\boldsymbol{w} = [w_1, ...w_n]^T$, tako da je izpolnjen sistem enačb

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

Enačbe (5.8) so linearne za koeficiente $w_1, ...w_n$:

$$\begin{aligned} w_1\varphi_1(\boldsymbol{x}_1) + w_2\varphi_2(\boldsymbol{x}_1)...w_n\varphi_{n(\boldsymbol{x}_1)} &= f_1 \\ &\vdots \\ w_1\varphi_1(\boldsymbol{x}_n) + w_2\varphi_2(\boldsymbol{x}_n)...w_n\varphi_{n(\boldsymbol{x}_n)} &= f_n, \end{aligned} \tag{5.9}$$

Z matriko sistema

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

Ker je

$$\varphi_{i(\boldsymbol{x}_j)} = \varphi(\|\boldsymbol{x}_j - \boldsymbol{x}_i\|) = \varphi(\|\boldsymbol{x}_i - \boldsymbol{x}_j\|) = \varphi_{j(\boldsymbol{x}_i)}, \tag{5.11}$$

je matrika sistema (5.10) simetrična. V literaturi [3] 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 [4]
- Gaussova funkcija: $\varphi(r) = e^{-\frac{r^2}{\sigma^2}}$
- racionalni približek za Gaussovo funkcijo:

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

Če izberemo primerno funkcijo oblike, lahko dosežemo, da je matrika sistema (5.10) pozitivno definitna. V tem primeru lahko za reševanje sistema uporabimo razcep Choleskega (poglavje 2.6 v [1]). Za funkcijo oblike bomo izbrali Gaussovo funkcijo

$$\varphi(r) = e^{-\frac{t^2}{\sigma^2}}, \tag{5.13}$$

za katero je matrika sistema (5.9) pozitivno definitna, če so točke $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots \boldsymbol{x}_n$ različne [5].

5.3 Program

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

```
"""
    RBF(tocke, utezi, phi)

Podatkovni tip za linearno kombinacijo *radialnih baznih funkcij* oblike
`phi(norm(x - tocke[i])^2)`.
"""
struct RBF
  tocke
  utezi
  phi
end
```

Za podatkovni tip napišimo 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 $(2, 1)$:

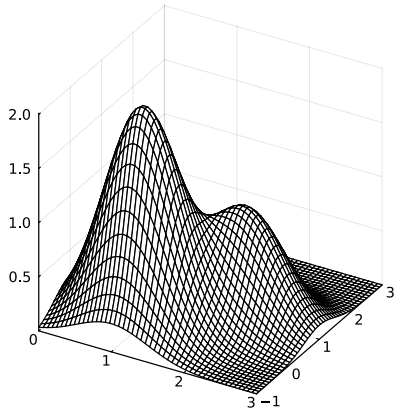
```
using Vaja05
""" Ustvari Gaussovo funkcijo z danim `sigma`."""
gauss(sigma) = r -> exp(-r^2 / sigma^2)

tocke = [[1, 0], [2, 1]]
utezi = [2, 1]
rbf = RBF(tocke, utezi, 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]))
```



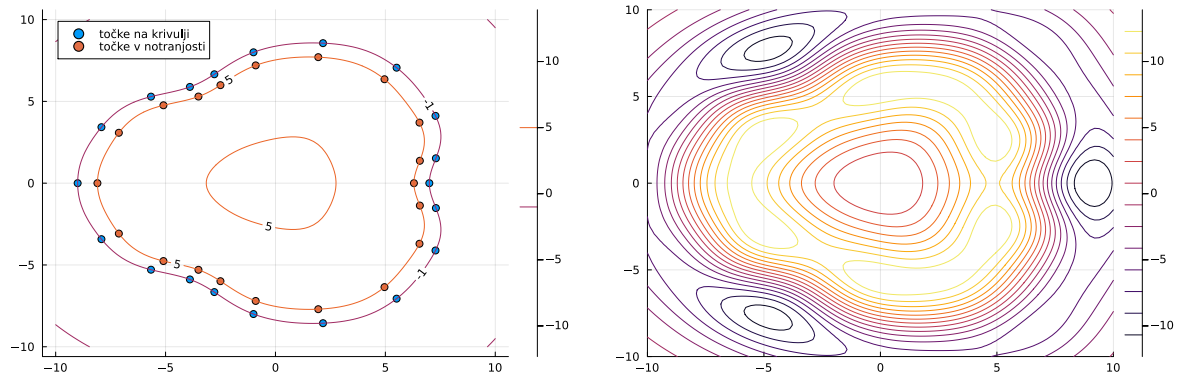
Slika 15: Linearna kombinacija dveh RBF v točkah $(1, 0)$ in $(2, 1)$ s funkcijo oblike $\varphi(r) = e^{-\frac{r^2}{0.7^2}}$.

Rešimo sedaj problem interpolacije. Zapišimo funkcijo `interpoliraj(tocke, 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 [4] in točkam na krivulji dodamo točke znotraj krivulje, v smeri normal, ki poskrbijo, da ne dobimo trivialne rešitve.

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

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

```
vse_tocke = vcat(tocke, tocke_noter)
c1, c2 = -1, 5
vrednosti = vcat(
  c1*ones(length(tocke)), c2*ones(length(tocke)))
rbf = interpoliraj(vse_tocke, 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 linearno kombinacijo RBF, ki interpolirajo dane točke

5.4 RBF s kompaktnim nosilcem

Če uporabimo klasične RBF iz prejšnjega razdelka je matrika sistema polna. Čeprav je večina členov izven diagonale zelo majhnih npr. pri Gaussovi RBF (5.13). Če uporabimo RBF s kompaktnim nosilcem, je matrika redka in se tako prostorska kot tudi časovna zahtevnost algoritmov bistveno zmanjšata [6].

5.5 Rešitve

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

Izračunaj vrednost linearne kombinacije radialnih baznih funkcij podane z
`rbf` v točki `x`.
"""
function vrednost(x, rbf::RBF)
  vsota = zero(x[1])
  n = length(rbf.tocke)
  for i = 1:n
    norma = norm(rbf.tocke[i] - x) # norma razlike
    vsota += rbf.utezi[i] * rbf.phi(norma) # utežena vsota
  end
  vsota
end
"""
    rbf::RBF(x)

Izračunaj vrednost linearne kombinacije radialnih baznih funkcij ``rbf`` v dani
točki `x`.
"""
(rbf::RBF)(x) = vrednost(x, rbf)
```

Program 30: Izračunaj vrednost linearne kombinacije RBF v dani točki

```
"""
    A = matrika(tocke, phi)

Poišči matriko sistema enačb za interpolacijo točk podanih v seznamu `tocke`
z linearno kombinacijo radialnih baznih funkcij s funkcijo oblike
`phi`.
"""
function matrika(tocke, phi)
  n = length(tocke)
  A = zeros(n, n)
  for i = 1:n, j = i:n
    A[i, j] = phi(norm(tocke[i] - tocke[j]))
    A[j, i] = A[i, j]
  end
  return A
end

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

Interpoliraj `vrednosti` v danih točkah iz seznama `tocke` z linearno
kombinacijo radialnih baznih funkcij s funkcijo oblike `phi`.
"""
function interpoliraj(tocke, vrednosti, phi)
  A = matrika(tocke, phi)
  F = cholesky!(A) # da prihranimo prostor, razcep naredimo kar v matriko A
  utezi = F \ vrednosti
  return RBF(tocke, utezi, phi)
end
```

Program 31: Interpoliraj vrednosti funkcij z linearno kombinacijo RBF

6 Fizikalna metoda za vložitev grafov

6.1 Naloga

- Izpelji sistem enačb za koordinate vozlišč grafa, tako da so v ravnovesju.
- 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 Ravnovesje sil

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

$$(x_1, y_1, z_1), (x_2, y_2, z_2), ..., (x_n, y_n, z_n). \tag{6.1}$$

Za dani graf G želimo poiskati vložitev v $\mathbb{R}^3$ (ali $\mathbb{R}^2$). V vsakem vozlišču mreže j morajo biti sile v ravnovesju.

$$\sum_{i \in N(j)} \vec{F}_{ij} = 0, \tag{6.2}$$

kjer je $N(j) = \{j; \quad (i, j) \in E(G)\}$ množica sosednjih točk v grafu za točko j in $\vec{F}_{ij}$ sila s katero vozlišče j deluje na vozlišče i.

Harmonična vzmet je idealna vzmet dolžine 0, za katero sila ni sorazmerna spremembi 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 drugo krajišče, je enaka

$$\vec{F}_{21} = k \cdot \begin{pmatrix} x_2 - x_1 \\ y_2 - y_1 \\ z_2 - z_1 \end{pmatrix}, \tag{6.3}$$

kjer je k koeficient vzmeti. Enačbe (6.2) lahko izpeljemo sistem enačb za koordinate x_i, y_i in z_i.

Iz vektorske enačbe

$$\sum_{j \in N(1)} k \begin{pmatrix} x_j - x_1 \\ y_j - y_1 \\ z_j - z_1 \end{pmatrix} = 0 \tag{6.4}$$

dobimo enačbe za posamezne koordinate

$$
\begin{aligned}
-\operatorname{st}(1)x_1 + x_{j_1} + x_{j_2} + \ldots + x_{j_{\operatorname{st}(1)}} &= 0 \\
-\operatorname{st}(1)y_1 + y_{j_1} + y_{j_2} + \ldots + y_{j_{\operatorname{st}(1)}} &= 0 \\
-\operatorname{st}(1)z_1 + z_{j_1} + z_{j_2} + \ldots + z_{j_{\operatorname{st}(1)}} &= 0 \\
&\vdots \\
x_{j_1} + x_{j_2} + \ldots + x_{j_{\operatorname{st}(n)}} - \operatorname{st}(n)x_n &= 0 \\
y_{j_1} + y_{j_2} + \ldots + y_{j_{\operatorname{st}(n)}} - \operatorname{st}(n)y_n &= 0 \\
z_{j_1} + z_{j_2} + \ldots + z_{j_{\operatorname{st}(n)}} - \operatorname{st}(n)z_n &= 0.
\end{aligned}
\tag{6.5}
$$

Koordinate x, y in z so neodvisni druga od druge in tako dobimo po en sistem enačb za vsako koordinato. Enačbe (6.5) so homogene, kar pomeni, da ima ničelno rešitev. Če želimo netrivialno rešitev, moramo nekaterim vozliščem v grafu predpisati koordinate. Brez škode lahko predpostavimo, da so vozlišča, ki jih fiksiramo na koncu. Označimo z $F = \{m+1, \ldots, n\} \subset V(G)$ množico vozlišč, ki jih fiksiramo. Koordinate za vozlišča iz F niso več spremenljivke, ampak jih moramo prestaviti na drugo stran enačbe. Za lažji zapis označimo z

$$
1_{A(k)} = \begin{cases} 1 \; k \in A \\ 0 \; k \notin A \end{cases}
\tag{6.6}
$$

indikatorsko funkcijo za množico A. Sistem enačb (6.5) postane nehomogen sistem:

$$
\begin{aligned}
-\operatorname{st}(1)x_1 + 1_{N(1)}(2)x_2 + \ldots + 1_{N(1)}(m)x_m &= -\sum_{j=m+1}^{n} 1_{N(1)}(j)x_j \\
1_{N(2)}(1)x_1 - \operatorname{st}(2)x_2 + \ldots + 1_{N(2)}(m)x_m &= -\sum_{j=m+1}^{n} 1_{N(2)}(j)x_j \\
&\vdots \\
1_{N(m)}(1)x_1 + 1_{N(m)}(2)x_2 + \ldots - \operatorname{st}(m)x_m &= -\sum_{j=m+1}^{n} 1_{N(m)}(j)x_j
\end{aligned}
\tag{6.7}
$$

in podobno za koordinati y in z. Matrika sistema (6.7) je odvisna le od grafa in izbire točk, ki so proste.

$$
A = \begin{pmatrix}
-\operatorname{st}(1) & 1_{N(1)}(2) & \ldots & 1_{N(1)}(m) \\
1_{N(2)}(1) & -\operatorname{st}(2) & \ldots & 1_{N(2)}(m) \\
\vdots & \vdots & \ddots & \vdots \\
1_{N(m)}(1) & 1_{N(m)}(2) & \ldots & -\operatorname{st}(m)
\end{pmatrix}
\tag{6.8}
$$

Za predstavitev grafa bomo uporabili paket Graphs.jl, ki definira podatkovne tipe in vmesnike za lažje delo z grafi.

Napišimo naslednji funkciji:

- `matrika(G::AbstractGraph, sprem)`, ki vrne matriko sistema (6.8) za dani graf `G` in seznam vozlišč, ki se lahko spreminjajo, `sprem` (rešitev Program 33) in
- `desne_strani(G::AbstractGraph, sprem, koordinate)`, ki vrne vektor desnih strani za sistem (6.7) (rešitev Program 34).

6.3 Metoda konjugiranih gradientov

Matrika (6.8) je simetrična in diagonalno dominantna. Res! Velja $\operatorname{st}(i) = |N(i)|$ in je zato

$$|a_{ii}| = |N(i)| \geq |N(i) \cap F^C| = \sum |a_{ij}|. \tag{6.9}$$

Za sosede fiksnih vozlišč je neenakost stroga. Ker so vsi elementi na diagonali negativni, je matrika A negativno definitna. Zato lahko za reševanje sistema $-Ax = -b$ uporabimo metodo konjugiranih gradientov.

Za večino grafov, za katere uporabimo zgornji postopek bo matrika sistema A redka. 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.

Uporabimo sedaj metodo konjugiranih gradientov za iskanje vložitve grafa s fizikalno metodo. Napišimo naslednji funkciji:

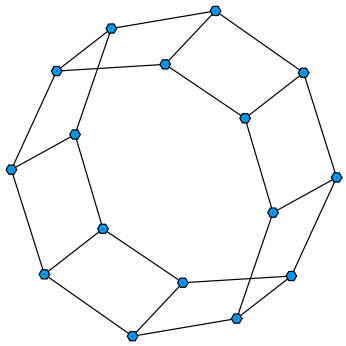
- `cg(A, b; atol=1e-8)`, ki poišče rešitev sistema $Ax = b$ z metodo konjugiranih gradientov (rešitev Program 36) in
- `vlozi!(G::AbstractGraph, fix, tocke)`, ki poišče vložitev grafa `G` v $\mathbb{R}^d$ s fizikalno metodo. Argument `fix` naj bo seznam fiksnih vozlišč, argument `tocke` pa matrika s koordinatami točk. Metoda naj ne vrne ničesar, ampak naj vložitev zapiše kar v matriko `tocke` (rešitev Program 35).

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 = krozna_lestev(8)

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



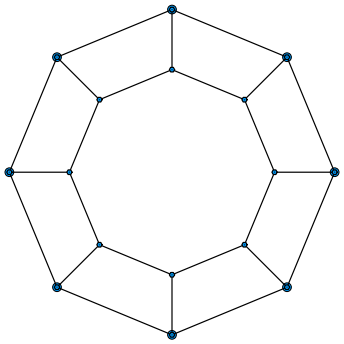
Slika 17: Graf krožna lestev s 16 vozlišči

Poiščimo drugačno vložitev s fizikalno metodo, tako da vozlišča enega cikla enakomerno razporedimo na krožnico.

```
t = range(0, 2pi, 9)[1:end-1]
x = cos.(t)
y = sin.(t)
scatter(x[1:8], y[1:8], label="fiksna vozlišča")


tocke = hcat(hcat(x, y)', zeros(2, 8))
fix = 1:8

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



Slika 18: 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 ravnovesju.

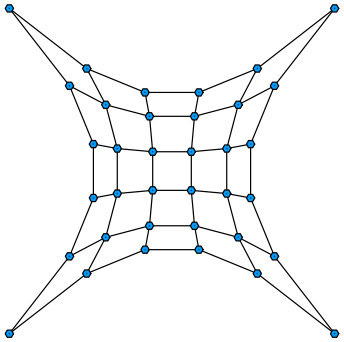
6.5 Mreža

Preizkusimo algoritem na pravokotni mreži.

```
m, n = 6, 6
G = 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]

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

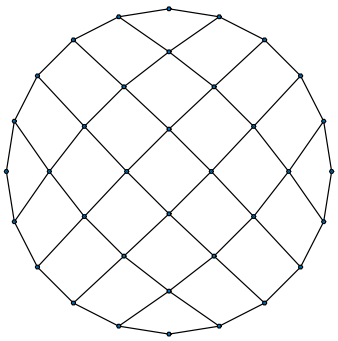


Slika 19: Pravokotna mreža vložena s fizikalno metodo. Fiksirani so le vogali.

Sedaj fiksiramo cel rob in ga enakomerno razporedimo na krožnico.

```
m, n = 6, 6
G = 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
  sosedi = neighbors(G, urejen_rob[end])
  sosedi = intersect(sosedi, rob)
  sosedi = setdiff(sosedi, urejen_rob)
  push!(urejen_rob, sosedi[1])
end
t = range(0, 2pi, length(rob) + 1)[1:end-1]
tocke = zeros(2, n * m)
tocke[:, urejen_rob] = hcat(cos.(t), sin.(t))'
vlozi!(G, urejen_rob, tocke)
graphplot(G, x=tocke[1, :], y=tocke[2, :], curves=false)
```



Slika 20: Pravokotna mreža vložena s fizikalno metodo. Rob mreže je enakomerno razporejen na krožnici.

6.6 Rešitve

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

Ustvari graf krožna lestev z `2n` točkami.
"""
function krozna_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: Ustvari graf krožna lestev

```
using SparseArrays

"""
    A = matrika(G::AbstractGraph, sprem)

Poišči matriko sistema za linearni sistem za fizikalno metodo za vložitev grafa
`G`.
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: Ustvari matriko sistema za ravnovesje sil v grafu

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

Poišči desne strani za linearni sistem za fizikalno metodo za vložitev grafa `G`
za eno koordinato.
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 fiksirana.
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: Izračunaj desne strani sistema za ravnovesje sil v grafu na podlagi koordinat vozlišč, ki so fiksirana

```
"""
    vlozi!(G::AbstractGraph, fix, tocke)

Poišči vložitev grafa `G` v prostor s fizikalno metodo. Argument `fix` vsebuje
vektor vozlišč
grafa, ki imajo določene koordinate. Argument `tocke` je začetna vložitev grafa.
Koordinate
vozlišč, ki niso fiksirane, bodo nadomeščene z novimi koordinatami.
Metoda ne vrne ničesar, ampak zapiše izračunane koordinate v matriko `tocke`.
"""
function vlozi!(G::AbstractGraph, fix, tocke)
  sprem = setdiff(vertices(G), fix)
  dim, _ = size(tocke)
  A = matrika(G, sprem)
  for k = 1:dim
    b = desne_strani(G, sprem, tocke[k, :])
    x = cg(-A, -b) # matrika A je negativno defnitna
    tocke[k, sprem] = x
  end
end
```

Program 35: Poišči koordinate vložitve grafa v $\mathbb{R}^d$ s fizikalno metodo

```julia
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 36: Metoda konjugiranih gradientov za reševanje sistema $Ax = b$ za pozitivno definitno matriko A

7 Invariantna porazdelitev Markovske verige

7.1 Naloga

- Implementiraj potenčno metodo za iskanje največje lastne vrednosti.
- Uporabi potenčno metodo in poišči invariantno porazdelitev Markovske verige z dano prehodno matriko P. Poišči invariantne porazdelitve za naslednja primera:
  - veriga, ki opisuje skakanje konja (skakača) po šahovnici,
  - veriga, ki opisuje brskanje po mini spletu z 5-10 stranmi (podobno spletni iskalniki razvrščajo strani po relevantnosti).

7.2 Invariantna porazdelitev Markovske verige

Z Markovskimi verigami smo se že srečali v poglavju o tridiagonalnih sistemih Poglavje 3. Porazdelitev Markovske verige X_k je podana z matriko P, katere elementi so prehodne verjetnosti:

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

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

$$\begin{gathered} p_i^{(k+1)} = \sum_{j=1}^{n} P(X_{k+1} = i \mid X_k = j) P(X_k = j) = \sum_{j=1}^{n} p_{ji} p_j^{(k)} \\ \boldsymbol{p}^{(k+1)} = P^T \boldsymbol{p}^{(k)}. \end{gathered} \tag{7.2}$$

Če zaporedje porazdelitev $p^{(k)}$ konvergira k limitni porazdelitvi $\boldsymbol{p}^{\infty}$, potem je limitna porazdelitev lastni vektor za P^T za lastno vrednost 1:

$$\boldsymbol{p}^{\infty} = \lim_{k\to\infty} \boldsymbol{p}^{(k)} = \lim_{k\to\infty} \boldsymbol{p}^{(k+1)} = \lim_{k\to\infty} P^T \boldsymbol{p}^{(k)} = P^T \lim_{k\to\infty} \boldsymbol{p}^{(k)} = P^T \boldsymbol{p}^{\infty}. \tag{7.3}$$

Ker so vsote elementov po vrsticah za prehodno matriko P enake 1, je 1 lastna vrednost matrike P in zato tudi lastna vrednost matrike P^T. Zato limitna porazdelitev $\boldsymbol{p}^{\infty}$ vedno obstaja, ni pa nujno enolična. Ker matrika P^T ne spremeni limitne porazdelitve $\boldsymbol{p}^{\infty}$, limitno porazdelitve imenujemo tudi *invariantna porazdelitev*.

Da se pokazati, da je 1 po absolutni vrednosti največja lastna vrednost matrike P in P^T, zato lahko lastni vektor poiščemo s potenčno metodo.

7.3 Potenčna metoda

S potenčno metodo poiščemo lastni vektor dane matrike A za po absolutni vrednosti največjo lastno vrednost matrike A. Če je takih lastnih vrednosti več (npr. 1 in -1), se lahko zgodi, da potenčna metoda ne konvergira. Izberemo neničelen začetni vektor $\boldsymbol{p}^{(0)} \neq 0$ in sestavimo zaporedje približkov

$$\boldsymbol{x}^{(k+1)} = \frac{A\boldsymbol{x}^{(k)}}{\left\|A\boldsymbol{x}^{(k)}\right\|}, \tag{7.4}$$

ki konvergira k lastnemu vektorju za največjo lastno vrednost. Za normo, s katero delimo produkt $A\boldsymbol{x}^{(k)}$, lahko izberemo katerakoli vektorsko normo. Ponavadi je to neskončna norma $\|\cdot\|_\infty$, saj jo lahko najhitreje določimo.

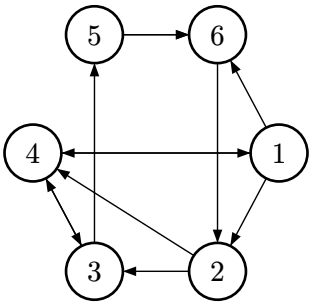
Napišimo program `x, it = potencna(A, x0)`, ki poišče lastni vektor za po absolutni vrednosti največjo lastno vrednost matrike A (Program 37).

7.4 Razvrščanje spletnih strani

Spletni iskalniki želijo uporabniku prikazati čim bolj relevantne rezultate. Zato morajo ugotoviti, katere spletne strani so bolj pomembne od drugih. Brskanje po spletu lahko modeliram 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 spletna stran nima povezav, lahko gremo nazaj na prejšnjo stran ali pa naključno izberemo drugo stran. Limitna porazdelitev pove, kolikšen delež vseh obiskov pripada posamezni spletni strani, če se naključno sprehajamo po spletu. Večji delež obiskov ima spletna stran, bolj je pomembna.

Limitno porazdelitev Markovske verige s prehodno matriko P poiščemo s potenčno metodo, kot lastni vektor matrike 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.



Slika 21: Mini splet s 6 stranmi

Prehodna matrika verige je

$$P = \begin{pmatrix} 0 & \frac{1}{3} & 0 & \frac{1}{3} & 0 & \frac{1}{3} \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix} \tag{7.5}$$

Poiščimo 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 = potencna(P', rand(6))
```

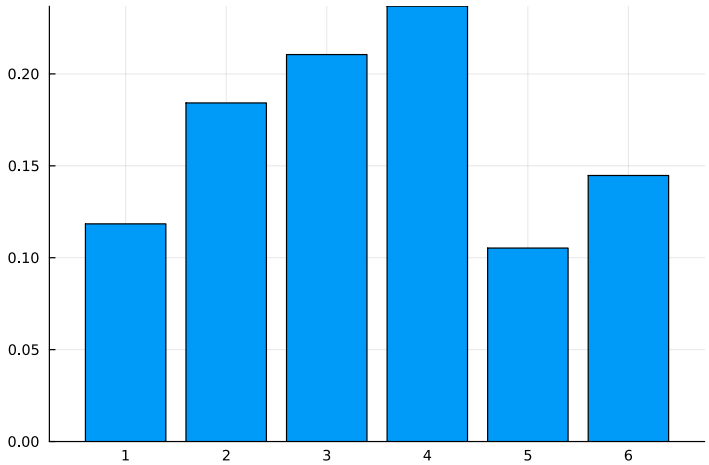
Preverimo, ali je dobljeni vektor res lastni vektor:

```
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:

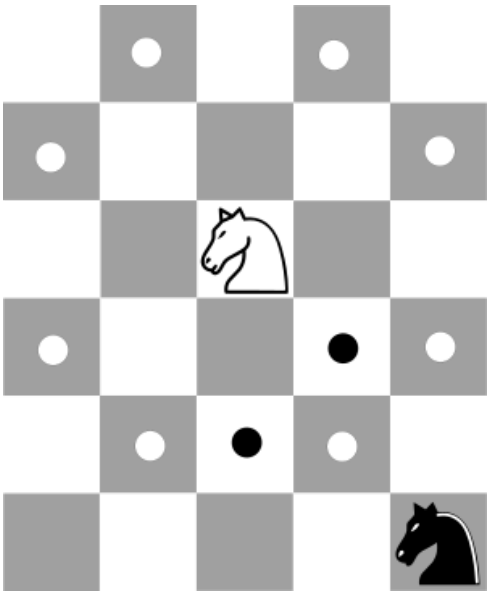
```
x = x / sum(x)
using Plots
bar(x, label=false)
```



Slika 22: Delež obiskov posamezne strani v limitni porazdelitvi

7.5 Skakanje konja po šahovnici

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



Slika 23: Možne poteze, ki jih lahko naredi konj.

Stanja označimo s pari indeksov, ki označujejo posamezno polje. Invariantna porazdelitev je podana z matriko verjetnosti, tako elementi matrike ustrezajo poljem na šahovnici. Zopet se srečamo s problemom iz prejšnjega poglavja Poglavje 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

$$\begin{aligned} k &= i + (j-1)m \\ j &= \lfloor (k-1)/m \rfloor \\ i &= ((k-1) \bmod m) + 1. \end{aligned} \tag{7.6}$$

Za lažje delo napišimo funkciji

- `k = ij_v_k(i, j, m)` in
- `i, j = k_v_ij(k, m)`,

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

Nato definirajmo

- podatkovno strukturo `Konj(m, n)`, ki predstavlja Markovsko verigo za konja na $m \times n$ šahovnici (Program 39) in
- funkcijo `prehodna_matrika(k::Konj)`, ki vrne prehodno matriko za Markovsko verigo za konja (Program 40).

Invariantno porazdelitev poskusimo poiskati s potenčno metodo:

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

x, it = potencna(P', rand(64))
```

Potenčna metoda ne konvergira, saj ima matrika 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

$$\frac{\boldsymbol{v_1} + \boldsymbol{v}_{-1}}{\|\boldsymbol{v}_1 + \boldsymbol{v}_{-1}\|} \text{ in } \frac{\boldsymbol{v_1} - \boldsymbol{v}_{-1}}{\|\boldsymbol{v}_1 + \boldsymbol{v}_{-1}\|}, \tag{7.7}$$

kjer je v_1 lastni vektor za 1 in v_{-1} lastni vektor za -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':
1.0000000000000018,  -1.0000000000000018
```

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

$$(\lambda_1 + \delta, \boldsymbol{v}_1), (\lambda_2 + \delta, \boldsymbol{v}_2), \ldots \tag{7.8}$$

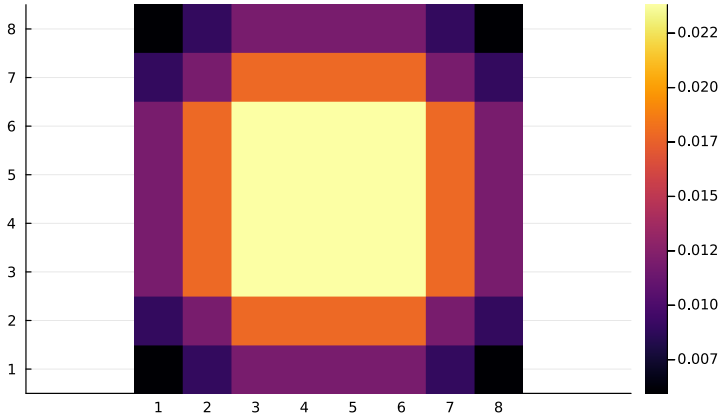
lastni pari matrike

$$A + \delta I. \tag{7.9}$$

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

```
x, it = potencna(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 24: Invariantna porazdelitev za konja na standardni 8×8 šahovnici. Svetlejša polja so pogosteje obiskana.

7.6 Rešitve

```
using LinearAlgebra

"""
    x, it = potencna(A)

Poišči lastni vektor matrike `A` za največjo lastno vrednost s potenčno metodo.
"""
function potencna(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: Potenčna metoda 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.

```
"""
  Konj(m, n)

Podatkovna struktura, ki označuje Markovsko verigo za konja na šahovnici
dimenzije `m` x `n`.
"""
struct Konj
  m
  n
end
```

Program 39: Podatkovni tip, ki predstavlja Markovsko verigo za konja, ki skače po šahovnici.

```
using SparseArrays
"""
  P = prehodna_matrika(k::Konj)

Poišči prehodno matriko za Markovsko verigo, ki opisuje skanje figure konja po
šahovnici.
"""
function prehodna_matrika(konj::Konj)
  m = konj.m
  n = konj.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 konja, ki skače po šahovnici.

8 Spektralno razvrščanje v gruče

Pokazali bomo metodo razvrščanja v gruče, ki uporabi spektralno analizo Laplaceove matrike podobnostnega grafa podatkov, zato da podatke preslika v prostor, kjer jih je lažje razvrstiti.

8.1 Podobnostni graf in Laplaceova matrika

Podatke (množico točk v $\mathbb{R}^n$) želimo razvrstiti v več gruč. Najprej ustvarimo *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-okolici te točke
- **k najbližji sosedi**: x_k povežemo z x_i, če je x_i med k najbližjimi točkami. Tako dobimo usmerjen graf, zato ponavadi upoštevmo 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

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

pri kateri s parametrom σ lahko določamo velikost okolic.

Grafu podobnosti priredimo matriko uteži

$$W = \left[w_{ij}\right], \tag{8.2}$$

in Laplaceovo matriko

$$L = D - W, \tag{8.3}$$

kjer je $D = \left[d_{ij}\right]$ diagonalna matrika z elementi $d_{ii} = \sum_j w_{ij}$. Laplaceova matrika L je simetrična, nenegativno definitna in ima vedno eno lastno vrednost 0 za lastni vektor iz samih enic.

8.2 Algoritem

Velja naslednji 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 komponente kar naše gruče, a se izkaže, da to ni najbolje.

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

Algoritem k povprečij

Izberemo si število gruč k. Najprej točke naključno razdelimo v k gruč. Nato naslednji postopek ponavljamo, dokler se rezultat ne spreminja več

- izračunamo center posamezne gruče $\boldsymbol{c}_i = \frac{1}{|G_i|} \sum_{j \in G_i} \boldsymbol{x}_i$,
- vsako točko razvrstimo v gručo, ki ima najbližji center.

8.3 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, y, title="Oblak točk v ravnini")
savefig("06_oblak.png")
```

Slika 25: Oblak točk

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

```
using SparseArrays
tocke = [(x[i], y[i]) for i=1:3*m]
r = 0.9
G = graf_eps_okolice(tocke, r)
L = LaplaceovaMatrika(G)
spy(sparse(Matrix(L)), title="Porazdelitev neničelnih elementov v laplaceovi matriki")
savefig("06_laplaceova_matrika_grafa.png")
```

Slika 26: Neničelni elementi Laplaceove matrike

Če izračunamo lastne vektorje in vrednosti laplaceove matrike dobljenega grafa, dobimo 4 najmanjše lastne vrednosti, ki očitno odstopajo od ostalih.

```
import LinearAlgebra.eigen
razcep = eigen(Matrix(L))
scatter(razcep.values[1:20], title="Prvih 20 lastnih vrednosti laplaceove matrike")
savefig("06_lastne_vrednosti.png")
```

Slika 27: Lastne vrednosti laplaceove matrike

```
scatter(razcep.vectors[:,4], razcep.vectors[:,5], title="Vložitev s komponentami 4. in
5. lastnega vektorja")
savefig("06_vlozitev.png")
```

Slika 28: Vložitev točk v nov prostor

8.4 Inverzna potenčna metoda

Ker nas zanima le najmanjših nekaj lastnih vrednosti, lahko njihov izračun in za izračun lastnih vektrojev uporabimo inverzno potenčno metodo. Pri inverzni potenčni metodi zgradimo zaporedje približkov z rekurzivno formulo

$$\boldsymbol{x}^{(k+1)} = \frac{A^{-1}\boldsymbol{x}^{(n)}}{\| A^{-1}\boldsymbol{x}^{(n)} \|} \tag{8.4}$$

in zaporedje približkov konvergira k lastnemu vektorju za najmanjšo lastno vrednost matrike A.

!!! warning „Namesto inverza uporabite razcep“

```
 Računanje inverza je časovno zelo zahtevna operacija, zato se jo razen v nizkih
dimenzijah,
 če je le mogoče izognemo. Namesto inverza raje uporabimo enega od razcepov matrike
```

```
$A$.
 Če naprimer 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

 $$
 \begin{aligned}
 L\mathbf{y}&=\mathbf{b}\cr
 U\mathbf{x}&=\mathbf{y}
 \end{aligned}
 $$

 Programski jezik `julia` ima za ta namen prav posebno metodo [factorize](https://
docs.julialang.org/en/v1/stdlib/LinearAlgebra/index.html#LinearAlgebra.factorize), ki
za različne matrike, izračuna najbolj primeren razcep.
```

Laplaceova matrika je simetrična, zato so lastne vrednosti ortogonalne. Lastne vektorje lahko tako poiščemo tako, da iteracijo izvajamo na več vektorjih hkrati in nato na dobljeni bazi izvedemo ortogonalizacijo (QR razcep), da zaporedje lastnih vektorjev za lastne vrednosti, ki so najbližje najmanjši lastni vrednosti.

Laplaceova matrika grafa okolic je simetrična in diagonalno dominantna. Poleg tega je zelo veliko elementov enakih 0. Zato za rešitev sistema uporabimo metodo konjugiranih gradientov. Za uporabo metode konjugiranih gradientov zadošča, da učinkovito izračunamo množenje matrike z vektorjem. Težava je, ker so je laplaceova matrika grafa izrojena, zato metoda konjugiranih gradientov ne konvergira. Težavo lahko rešimo s premikom. Namesto, da računamo lastne vreednosti in vektorje matrike L, iščemo lastne vrednosti in vektorje malce premaknjene matrike $L + \varepsilon I$, ki ima enake lastne vektorje, kot L.

!!! note

```
Programski jezik julia omogoča polimorfizem v obliki [večlične
dodelitve](https://docs.julialang.org/en/v1/manual/methods/index.html). Tako
lahko za isto funkcijo definiramo različne metode. Za razliko od polmorfizma
v objektno orientiranih jezikih, se metoda izbere ne le na podlagi tipa
objekta, ki to metodo kliče, ampak na podlagi tipov vseh vhodnih argumentov.
To lastnost lahko s pridom uporabimo, da lahko pišemo generično kodo, ki
deluje za veliko različnih vhodnih argumentov. Primer je funkcija
[`conjgrad`](@ref), ki jo lahko uporabimo tako za polne matrike, matrike tipa
`SparseArray` ali pa tipa `LaplaceovaMatrika` za katerega smo posebej
definirali operator množenja [`*`](@ref).
```

$$L\boldsymbol{x}^{(\boldsymbol{k}+\mathbf{1})} = \boldsymbol{x}^{(\boldsymbol{k})} \tag{8.5}$$

Primerjajmo inverzno potenčno metodo z vgrajeno metodo za iskanje lastnih vrednosti s polno matriko

```
import Base:*, size
struct PremikMatrike
   premik
   matrika
end
*(p::PremikMatrike, x) = p.matrika*x + p.premik.*x
size(p::PremikMatrike) = size(p.matrika)

Lp = PremikMatrike(0.01, L)
l, v = inverzna_iteracija(Lp, 5, (Lp, x) -> conjgrad(Lp, x)[1])
```

8.5 Algoritem k-povprečij

```
nove_tocke =  [tocka for tocka in zip(razcep.vectors[:,4], razcep.vectors[:,5])]
gruce = kmeans(nove_tocke, 3)

p1 = scatter(tocke[findall(gruce .== 1)], color=:blue, title="Originalne točke")
scatter!(p1, tocke[findall(gruce .== 2)], color=:red)
scatter!(p1, tocke[findall(gruce .== 3)], color=:green)

p2 = scatter(nove_tocke[findall(gruce .== 1)], color=:blue, title="Preslikane točke")
scatter!(p2, nove_tocke[findall(gruce .== 2)], color=:red)
scatter!(p2, nove_tocke[findall(gruce .== 3)], color=:green)

plot(p1,p2)
savefig("06_gruce.png")
```

Slika 29: Gruče

8.6 Literatura

- Ulrike von Luxburg A Tutorial on Spectral Clustering
- Peter Arbenz Lecture Notes on Solving Large Scale Eigenvalue Problems
- Knjižnica Laplacians.jl

9 Konvergenčna območja nelinearnih enačb

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. Vsakemu vozlišču v mreži priredi različne barve, glede na to, h kateri rešitvi konvergira Newtonova metoda. Ves postopek zapiši v funkcijo `konvergencno_obmocje`.

9.2 Newtonova metoda za sisteme enačb

Sistem nelinearnih enačb lahko zapišemo v obliki

$$\boldsymbol{F}(\boldsymbol{x}) = \boldsymbol{0}, \tag{9.2}$$

kjer sta $\boldsymbol{0} = [0, 0, ...]^T$ in $\boldsymbol{x} = [x_1, x_2, ...]^T \in \mathbb{R}^n$ n-dimenzionalna vektorja in $\boldsymbol{F} : \mathbb{R}^n \to \mathbb{R}^n$ vektorska funkcija z vektorskim argumentom:

$$\boldsymbol{F}(\boldsymbol{x}) = \begin{pmatrix} f_1(x_1, x_2 ... x_n) \\ f_2(x_1, x_2 ... x_n) \\ \vdots \\ f_n(x_1, x_2 ... x_n) \end{pmatrix}. \tag{9.3}$$

Denimo, da je $\boldsymbol{x}^{(k)}$ približek za rešitev enačbe (9.2). Funkcijo $\boldsymbol{F}$ lahko, podobno kot funkcijo ene spremenljivke, v točki $\boldsymbol{x}^{(k)}$ aproksimiramo z linearno funkcijo:

$$\boldsymbol{F}(\boldsymbol{x}) = \boldsymbol{F}\big(\boldsymbol{x}^{(k)}\big) + \mathrm{JF}\big(\boldsymbol{x}^{(k)}\big)\big(\boldsymbol{x} - \boldsymbol{x}^{(k)}\big) + \mathcal{O}\Big(\big(\boldsymbol{x} - \boldsymbol{x}^{(k)}\big)^2\Big), \tag{9.4}$$

kjer je $\mathrm{JF}(\boldsymbol{x})$ Jacobijeva matrika parcialnih odvodov komponent $f_i(x_1, x_2, ...)$ po koordinatah x_j

$$\mathrm{JF}(\boldsymbol{x}) = \begin{pmatrix} \frac{\partial f_1(\boldsymbol{x})}{\partial x_1} & \frac{\partial f_1(\boldsymbol{x})}{\partial x_2} & \cdots & \frac{\partial f_1(\boldsymbol{x})}{\partial x_n} \\ \frac{\partial f_2(\boldsymbol{x})}{\partial x_1} & \frac{\partial f_2(\boldsymbol{x})}{\partial x_2} & \cdots & \frac{\partial f_2(\boldsymbol{x})}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_{n(\boldsymbol{x})}}{\partial x_1} & \frac{\partial f_{n(\boldsymbol{x})}}{\partial x_2} & \cdots & \frac{\partial f_{n(\boldsymbol{x})}}{\partial x_n} \end{pmatrix}. \tag{9.5}$$

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

$$\begin{aligned} \boldsymbol{0} &= \boldsymbol{F}\big(\boldsymbol{x}^{(k)}\big) + \mathrm{JF}\big(\boldsymbol{x}^{(k)}\big)\big(\boldsymbol{x}^{(k+1)} - \boldsymbol{x}^{(k)}\big) \\ \mathrm{JF}\big(\boldsymbol{x}^{(k)}\big)\boldsymbol{x}^{(k+1)} &= \mathrm{JF}\big(\boldsymbol{x}^{(k)}\big)\boldsymbol{x}^{(k)} - \boldsymbol{F}\big(\boldsymbol{x}^{(k)}\big). \end{aligned} \tag{9.6}$$

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

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} - \mathrm{JF}\big(\boldsymbol{x}^{(k)}\big)^{-1} \boldsymbol{F}\big(\boldsymbol{x}^{(k)}\big), \tag{9.7}$$

pri čemer formule ne smemo jemati dobesedno, saj inverzne matrike $\mathrm{JF}\big(\boldsymbol{x}^{(k)}\big)^{-1}$ dejansko ne izračunamo. Izraz $\mathrm{JF}\big(\boldsymbol{x}^{(k)}\big)^{-1} \boldsymbol{F}\big(\boldsymbol{x}^{(k)}\big)$ poiščemo tako, da rešimo sistem $\mathrm{JF}\big(\boldsymbol{x}^{(k)}\big)\boldsymbol{x} = \boldsymbol{F}\big(\boldsymbol{x}^{(k)}\big)$ (npr. z LU razcepom ali kako drugače).

Poglejmo si, kako uporabimo Newtonovo metodo za enačbe (9.1). Spremenljivke x, y postavimo v vektor $\boldsymbol{x} = [x, y]$ in za lažje pisanje programa vpeljemo komponente $x_1 = x$ in $x_2 = y$. Funkcija $\boldsymbol{F}(\boldsymbol{x})$ je enaka

$$\boldsymbol{F}(\boldsymbol{x}) = \begin{pmatrix} x_1^3 - 3x_1x_2^2 - 1 \\ 3x_1^2x_2 - x_2^3 \end{pmatrix}, \tag{9.8}$$

Jacobijeva matrika $\mathrm{JF}(\boldsymbol{x})$ pa

$$\mathrm{JF}(\boldsymbol{x}) = \begin{pmatrix} 3x_1^2 - 3x_2^2 & -6x_1x_2 \\ -6x_1x_2 & 3x_1^2 - 3x_2^2 \end{pmatrix}. \tag{9.9}$$

```
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
```

Uporabimo funkcijo `newton` in za različne začetne približke, preverimo rezultat.

```
x1, it1 = newton(f, jf, [2, 0])
x2, it2 = newton(f, jf, [-1, 1.0])
x3, it3 = newton(f, jf, [-1, -1.0])
```

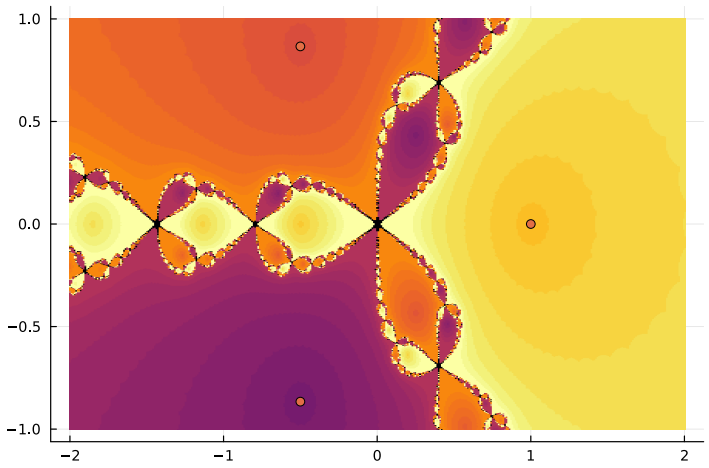
Avtomatsko odvajanje

Jacobijevo matriko odvodov lahko učinkovito izračunamo z avtomatskim odvajanjem. V Juliji uporabimo funkcijo `jacobian` iz paketa ForwardDiff.

9.3 Konvergenčno območje

Za razliko od linearnih enačb imajo nelinearne enačbe lahko zelo različno število rešitev. Newtonova metoda je občutljiva glede izbire začetnega približka. Če je začetni približek Blizu rešitve, Newtonova metoda konvergira k tisti ničli, za zastran od rešitevki so Newtonova metoda je občutljiva

```
using Plots
maxit = 20
obmocje = Box2d(Interval(-2, 2), Interval(-1, 1))
metoda(x0) = newton(f, jf, x0; atol=1e-4, maxit=maxit)
x, y, Z, nicle, koraki = konvergenca(obmocje, metoda, 800, 400; atol=1e-3)
heatmap(x, y, Z + 0.8*min.(koraki/10, 1), legend=false)
scatter!(Tuple.(nicle), label="rešitve")
```



Slika 30: Newtonova metoda konvergira k različnim rešitvam odvisno od začetnega približka

9.4 Rešitve

Naslednje funkcije smo napisali v `Vaja09/src/Vaja09.jl`.

```
using LinearAlgebra

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 41: Newtonova metoda za sisteme enačb.

10 Nelinearne enačbe v geometriji

10.1 Naloga

- Implementirajte Newtonovo metodo za sisteme nelinearnih enačb.
- Napišite funkcije, ki poišče presečišča geometrijskih objektov:
  - samopresečišče Lissajousove krivulje

$$(x(t), y(t)) = (a\sin(nt), b\cos(mt)) \tag{10.1}$$

  za parametre $a = b = 1$ in $n = 3$ in $m = 2$.
  - poltraka $\boldsymbol{x}(t) = \boldsymbol{x_0} + t\boldsymbol{e}$ in implicitne ploskve podane z enačbo

  $F(x, y, z) = 0$.
- Poiščite minimalno razdaljo med dvema parametrično podanima krivuljama:

$$\begin{aligned}(x_1(t), y_1(t)) &= \left(2\cos(t) + \frac{1}{3}, \sin(t) + \frac{1}{4}\right)\\ (x_2(s), y_2(s)) &= \left(\frac{1}{3}\cos(s) - \frac{1}{2}\sin(s), \frac{1}{3}\cos(s) + \frac{1}{2}\sin(t)\right).\end{aligned} \tag{10.2}$$

  - Zapišite razdaljo med točko na prvi krivulji in točko na drugi krivulji kot funkcijo $d(t, s)$ parametrov t in s.
  - Minimum funkcije $d(t, s)$ oziroma $d^2(t, s)$ poiščite z gradientnim spustom.
  - Minimum funkcije $d^2(t, s)$ poiščite z Newtonovo metodo kot rešitev vektorske enačbe

$$\nabla d^2(t, s) = 0. \tag{10.3}$$

  - Grafično predstavi zaporedja približkov za gradientno metodo in Newtonovo metodo.
  - Primerjaj konvergenčna območja za gradientno in Newtonovo metodo (glej Poglavje 9).

11 Aproksimacija z linearnim modelom

11.1 Naloga

- Podatke o koncentraciji CO_2 v ozračju aproksimiraj s kombinacijo kvadratnega polinoma in sinusnega nihanja s periodo 1 leto.
- Parametre modela poišči z normalnim sistemom in QR razcepom.
- Model uporabi za napoved obnašanja koncentracije CO_2 za naslednjih 20 let.

12 Interpolacija z zlepki

12.1 Naloga

- Podatke iz tabele

x	x_1	x_2	...	x_n
$f(x)$	y_1	y_2	...	y_n
$f'(x)$	dy_1	dy_2	...	dy_n

Tabela 2: Podatki, ki jih potrebujemo za Hermitov kubični zlepek.

  interpolirajte s Hermitovim kubičnim zlepkom.
- Uporabite Hermitovo bazo kubičnih polinomov, ki zadoščajo pogojem (Tabela 3) in jih z linearno preslikavo preslikate 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 $t = 1$.

- Definirajte podatkovni tip `HermitovZlepek` za Hermitov kubični zlepek, ki vsebuje podatke iz tabele Tabela 2.
- Napišite funkcijo `vrednost(zlepek, x)`, ki izračuna vrednost Hermitovega kubičnega zlepka v dani vrednosti argumenta x. Za podatkovni tip `HermitovZlepek` uporabite sintakso `julije`, ki omogoča, da se objekte kliče kot funkcije.
- S Hermitovim zlepkom interpolirajte funkcijo $\sin(x^2)$ na intervalu $[0, 5]$. Napako ocenite s formulo za napako polinomske interpolacije

$$f(x) - p_3(x) = \frac{f^{(4)}(\xi)}{4!}(x - x_1)(x - x_2)(x - x_3)(x - x_4) \tag{12.1}$$

  in oceno primerjajte z dejansko napako. Narišite graf napake in ocene za napako.
- Funkcijo $\sin(x^2)$ na intervalu $[0, 5]$ interpolirajte tudi z Newtonovim polinomom, in linearnim zlepkom.
- Hermitov kubični zlepek uporabite za interpolacijo zaporedja točk v ravnini s parametričnim zlepkom (vsako koordinatno funkcijo intepoliramo posebej, odvode pa določimo z deljenimi diferencami).

13 Porazdelitvena funkcija normalne porazdelitve

13.1 Naloga

- Implementiraj porazdelitveno funkcijo standardne normalne porazdelitve

$$\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-\frac{t^2}{2}} dt. \tag{13.1}$$

- Poskrbi, da je relativna napaka manjša od $0.5 \cdot 10^{\{-11\}}$. Definicijsko območje razdeli na več delov in na vsakem delu uporabi primerno metodo, da zagotoviš relativno natančnost.
- Interval $(-\infty, -1]$ transformiraj s funkcijo $\frac{1}{x}$ na interval $[-1, 0]$ in uporabi interpolacijo s polinomom na Čebiševih točkah.
  - Namesto funkcije $\Phi(x)$ aproksimiraj funkcijo $xe^{x^2}\Phi(x)$.
  - Vrednosti funkcije $\Phi(x)$ v Čebiševih točkah izračunaj
- Na intervalu $[-1, a]$ za primerno izbran a uporabi Gauss-Legendrove kvadrature.
- Izberi a, da je na intervalu $[a, \infty)$ vrednost na 10 decimalk enaka 1.

13.2 Aproksimacija s polinomi Čebiševa

Weierstrassov izrek pravi, da lahko poljubno zvezno funkcijo na končnem intervalu enakomerno na vsem intervalu aproksimiramo s polinomi. Polinom dane stopnje, ki neko funkcijo najbolje aproksimira je težko poiskati. Z razvojem funkcije po ortogonalnih polinomih Čebiševa, pa se optimalni aproksimaciji zelo približamo. Naj bo $f : [-1, 1] \to \mathbb{R}$ zvezna funkcija. Potem lahko f zapišemo z neskončno Furierovo vrsto

$$f(t) = \sum_{n=0}^{\infty} a_n T_{n(t)}, \tag{13.2}$$

kjer so T_n polinomi Čebiševa, a_n pa koeficienti. Koeficienti a_n so dani z integralom

$$\begin{aligned} a_0 &= \frac{1}{\pi} \int_{-1}^{1} \frac{f(x)}{\sqrt{1-x^2}} dx \\ a_n &= \frac{2}{\pi} \int_{-1}^{1} \frac{f(x)T_n(x)}{\sqrt{1-x^2}} dx. \end{aligned} \tag{13.3}$$

Polinomi Čebiševa so definirani z relacijo

$$T_n(\cos(\varphi)) = \cos(n\varphi) \tag{13.4}$$

in zadoščajo dvočlenski rekurzivni enačbi

$$T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x). \tag{13.5}$$

Prvih nekaj polinomov $T_n(x)$ je enakih:

$$
\begin{aligned}
T_0(x) &= 1 \\
T_1(x) &= x \\
T_2(x) &= 2x^2 - 1 \\
T_3(x) &= 2x(2x^2 - 1) - x = 4x^3 - 3x
\end{aligned}
\tag{13.6}
$$

Namesto cele vrste (13.2), lahko obdržimo le prvih nekaj členov in funkcijo aproksimiramo s končno vsoto

$$
f(x) \approx C_{N(x)} = \sum_{n=0}^{N} a_n T_n(x), \tag{13.7}
$$

koeficiente a_n pa poiščemo numerično z Gauss-Čebiševimi kvadraturami [7].

Vozlišča za Gauss-Čebiševa kvadraturo v n vozliščih so v Čebiševih vozliščih

$$
x_k = \cos\left(\frac{2k+1}{2n}\pi\right), \quad k = 0, \ldots n-1, \tag{13.8}
$$

uteži pa so vse enake $w_k = \frac{\pi}{n}$. Za vrsto $C_{N(x)}$ uporabimo kvadraturne formule z $N+1$ vozlišči. Za koeficiente tako na intervalu $[-1, 1]$ dobimo približne formule

$$
\begin{aligned}
a_0 &= \frac{1}{N+1} \sum_{k=0}^{N} f(x_k) \\
a_1 &= \frac{2}{N+1} \sum_{k=0}^{N} T_1(x_k) f(x_k) \\
a_2 &= \frac{2}{N+1} \sum_{k=0}^{N} T_2(x_k) f(x_k) \\
&\vdots \\
a_N &= \frac{2}{N+1} \sum_{k=0}^{N} T_N(x_k) f(x_k).
\end{aligned}
\tag{13.9}
$$

Koeficiente Čebiševe vrste natančneje in hitreje računamo s FFT

Na vajah bomo koeficiente an računali približno z Gauss-Čebiševimi kvadraturnimi formulami. V praksi je mogoče koeficiente a_n izračunati bolj natančno in hitreje ($\mathcal{O}(n\log(n))$ namesto $\mathcal{O}(n^2)$) z diskretno Fourierovo kosinusno transformacijo funkcijskih vrednosti v Čebiševih interpolacijskih točkah [8].

Če želimo aproksimirati funkcijo $f : [a, b] \to \mathbb{R}$, moramo argument preslikati na interval $[-1, 1]$ z linearno preslikavo. V splošnem sta linearni preslikavi med $x \in [a, b]$ in $t \in [c, d]$ podani kot:

$$
\begin{aligned}
t(x) &= \frac{d-c}{b-a}(x-a) + c \\
x(t) &= \frac{b-a}{d-c}(t-c) + b.
\end{aligned}
\tag{13.10}
$$

Namesto $f(x)$ aproksimiramo funkcijo $\tilde{f}(t) = f(x(t))$ na intervalu $[-1, 1]$.

Napako aproksimacije lahko ocenimo z velikostjo koeficientov a_n. Ker je

$$
|T_{n(x)}| \leq 1, \quad x \in [-1, 1], \tag{13.11}
$$

je napaka $f(x) - C_{N(x)}$ omejena s $\sum_{n=N+1}^{\infty} |a_n|$

$$|f(x) - C_{N(x)}| = |\sum_{n=N+1}^{\infty} a_n T_n(x)| \leq \sum_{n=N+1}^{\infty} |a_n| \tag{13.12}$$

Ker neskončne vrste $\sum_{n=N+1}^{\infty} |a_n|$ ne moremo sešteti, za približno oceno napake vzamemo kar zadnji koeficient a_N v končni vsoti $C_{N(x)}$.

13.3 Čebiševa aproksimacija funkcije Φ za majhne x

Za majhne x se vrednost Φ približuje 0

$$\lim_{x \leftarrow -\infty} \Phi(x) = 0. \tag{13.13}$$

Zato ni dovolj, da omejimo absolutno napako, ampak moramo poskrbeti, da je tudi relativna napaka dovolj majhna. Formula

$$\Phi(-x) = 1 - \Phi(x) \tag{13.14}$$

ni uporabna, saj pri odštevanju dveh skoraj enakih vrednosti relativna napaka nekontrolirano naraste. Zato definicijsko območje razdelimo na dva intervala $(-\infty, c]$ in $[c, \infty)$. Na intervalu $[c, \infty)$ je vrednost Φ navzdol omejena z $\Phi(c)$ in je relativna napaka največ $\frac{1}{\Phi(c)}$ kratnik absolutne napake. Zato je na $[c, \infty)$ dovolj, če poskrbimo, da je absolutna napaka majhna.

Pri aproksimaciji s polinomi Čebiševa imamo kontrolo le nad absolutno napako. Če blizu ničle funkcije pa majhna absolutna napaka ne pomeni nujno tudi majhne relativne napake. Težavo lahko rešimo tako, da funkcijo $\Phi(x)$ pomnožimo s faktorjem $k(x)$ tako, da je limita

$$\lim_{x \to -\infty} k(x)\Phi(x) = 1. \tag{13.15}$$

Namesto funkcije $\Phi(x)$ aproksimiramo funkcijo $g(x) = k(x)\Phi(x)$, ki je navzdol omejena z neničelno vrednostjo na $(-\infty, c]$. Za funkcijo $g(x)$ lahko poskrbimo, da je absolutna napaka enakomerno omejena na $(-\infty, c]$. Vrednost funkcije $\Phi(x)$ nato izračunamo tako, da izračunamo kvocient

$$\Phi(x) = \frac{g(x)}{k(x)}, \tag{13.16}$$

kar pa ne povzroči bistvenega povečanja relativne napake, saj je deljenje za razliko od odštevanja ne povzroči katastrofalnega krajšanja.

Če je $c < 0$, lahko za dodatni faktor izberemo $k(x)$ $\Phi(x)$ Izračun vrednosti za majhne vrednosti x lahko izračunamo z Gauss-Laguerreovimi kvadraturami [7]

$$\int_0^{\infty} f(x)e^{-1}dx \approx \sum_{k=1}^{N} w_k f(x_k)m, \tag{13.17}$$

kjer so x_k ničle Laguerrovega polinoma stopnje $N-1$, w_i pa primerno izbrane uteži. Vrednost

$\Phi(x)$ za majhne vrednosti x

$$\Phi(x) = \int_{-\infty}^{x} e^{-\frac{t^2}{2}} dt \tag{13.18}$$

lahko z uvedbo nove spremenljivke $u = x - t$, ki preslika interval $(-\infty, x)$ v interval $(0, \infty)$, prevedemo na integral

$$\int_{-\infty}^{x} e^{-\frac{t^2}{2}} dt = \int_{0}^{\infty} e^{-\frac{(u-x)^2}{2}} du = \int_{0}^{\infty} e^{-\frac{(u-x)^2}{2}+u} e^{-u} du \tag{13.19}$$

in uporabimo Gauss-Laguerove kvadrature (13.17) za funkcijo $f(u) = e^{-\frac{(u-x)^2}{2}+u}$.

13.4 Izračun funkcije $\Phi(x)$ na $[c, \infty)$

14 Povprečna razdalja med dvema točkama na kvadratu

14.1 Naloga

- Izpeljite algoritem, ki izračuna integral na več dimenzionalnem 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).

15 Avtomatsko odvajanje z dualnimi števili

15.1 Naloga

- Definirajte podatkovni tip za dualna števila.
- Za podatkovni tip dualnega števila definirajte osnovne operacije in elementarne funkcije, kot so sin, cos in exp.
- Uporabite dualna števila in izračunajte hitrost nebesnega telesa, ki se giblje po Keplerjevi orbiti. Keplerjevo orbito izrazite z rešitvijo Keplerjeve enačbe, ki jo rešite z Newtonovo metodo.
- Posploši dualna števila, da je komponenta pri ε lahko vektor. Uporabite posplošena dualna števila za izračun gradienta funkcije več spremenljivk.

16 Reševanje začetnega problema za NDE

Navadna diferencialna enačba

$$u'(t) = f(t, u, p) \tag{16.1}$$

ima enolično rešitev za vsak začetni pogoj $u(t_0) = u_0$. Iskanje rešitve NDE z danim začetnim pogojem imenujemo začetni problem.

V naslednji vaji bomo napisali knjižnico za reševanje začetnega problema za NDE. Napisali bomo naslednje:

1. Podatkovno strukturo, ki hrani podatke o začetnemu problemu.
2. Podatkovno strukturo, ki hrani podatke o rešitvi začetnega problema.
3. Različne metode za funkcijo `resi`, ki poiščejo približek za rešitev začetnega problema z različnimi metodami:
   - Eulerjevo metodo,
   - Runge-Kutta reda 2,
   - prediktor korektor z Eulerjevo in implicitno trapezno metodo in kontrolo koraka.
4. Funkcijo `vrednost`, ki za dano rešitev začetnega problema izračuna vrednost rešitve v vmesnih točkah s Hermitovim kubičnim zlepkom. Uporabite Hermitovo bazo kubičnih polinomov, ki zadoščajo pogojem v tabeli

	$p(0)$	$p(1)$	$p'(0)$	$p'(1)$
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 4: Vrednosti baznih polinomov $h_{ij}(t)$ in njihovih odvodov v točkah $t = 0$ in $t = 1$.

5. Napisane funkcije uporabite, da poiščete rešitev začetnega problema za poševni met z zračnim uporom. Kako daleč leti telo preden pade na tla? Koliko časa leti?
6. Ocenite napako, tako da rezultat izračunajte z dvakrat manjšim korakom.

16.1 Hermitova interpolacija

Približne 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. Hermitov zlepek je na intervalu (x_i, x_{i+1}) enak kubičnemu polinomu, ki se z rešitvijo ujema v vrednostih in odvodih v krajiščih intervala x_i in x_{i+1}.

16.2 Poševni met z zračnim uporom

16.3 Rešitve

Program 42:

17 Aproksimacija podatkov z dinamičnim modelom

17.1 Naloga

- Poišči parametre $\boldsymbol{p}$ dinamičnega modela Lotka-Volterra na podlagi izmerjenih vrednosti v določenih časovnih trenutkih.
- Napiši funkcijo $r(\boldsymbol{p})$, ki izračuna vsoto kvadratov razlik med vrednostmi modela in izmerjenimi vrednostmi.
- Poišči minimum funkcije r z gradientno metodo. Gradient izračunaj z avtomatskim odvodom.
- Algoritem preskusi na umetno generiranih podatkih.

18 Domače naloge

18.1 Navodila za pripravo domačih nalog

Ta dokument vsebuje navodila za pripravo domačih nalog. Navodila so napisana za programski jezik Julia. Če uporabljate drug programski jezik, navodila smiselno prilagodite.

18.1.1 Kontrolni seznam

Spodaj je seznam delov, ki naj jih vsebuje domača naloga.

- koda (`src\DomacaXY.jl`)
- testi (`test\runtests.jl`)
- dokument `README.md`
- demo skripta, s katero ustvarite rezultate za poročilo
- poročilo v formatu PDF

Preden oddate domačo nalogo, uporabite 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(matematičen) opis naloge
  - splošen opis rešitve
  - primer uporabe (slikice prosim :-)

18.1.2 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, če je čim več besedila razumljivega tudi širši publiki. Tudi strokovnjaki radi beremo besedila, ki jih hitro razumemo. Zato je dobro začeti z okvirnim opisom z malo formulami in splošnimi izrazi. V nadaljevanju lahko besedilo stopnjujemo k vedno večjim podrobnostim.

Določene podrobnosti, ki so povezane s konkretno implementacijo, brez škode izpustimo.

18.1.2.1 Opis rešitve naj bo okviren

Opis rešitve naj bo zgolj okviren. Izogibajte se uporabi programerskih izrazov ampak raje uporabljajte matematične. Na primer izraz uporabimo for zanko, lahko nadomestimo s postopek ponavljamo. Od bralca zahteva splošen opis manj napora in dobi širšo sliko. Če želite dodati izpeljave, jih napišite z matematičnimi formulami, ne v programskem 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 razlaga 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`.

18.1.2.2 Podrobnosti implementacije ne sodijo v poročilo

Podrobnosti implementacije so razvidne iz kode, zato jih nima smisla ponavljati v poročilu. Algoritme opišete okvirno, tako da izpustite podrobnosti, ki niso nujno potrebne za razumevanje. Podrobnosti lahko dodate, v nadaljevanju, če mislite, da so nujne za razumevanje.

Dobro! *Algoritem opišemo okvirno, podrobnosti razložimo kasneje*

V matriki želimo eleminirati spodnji trikotnik. To dosežemo tako, da stolpce enega za drugim preslikamo s Hausholderjevimi zrcaljenji. Za vsak stolpec poiščemo vektor, preko katerega bomo zrcalili. Vektor poiščemo tako, da bo imela zrcalna slika ničle pod diagonalnim elementom.

Tu lahko z razlago zaključimo. Če želimo dodati podrobnosti, pa jih navedemo za okvirno idejo.

Dobro! *Podrobnosti sledijo za okvirno razlago*

Vektor zrcaljenja dobimo kot

$$u = \left[s(k) + A_{k,k}, A_{k+1,k}, \ldots A_{n,k}\right], \tag{18.1}$$

kjer je $s(k) = \text{sign}\left(A_{k,k}\right) * \|A(k:n,k)\|$. Podmatriko $A(k:n, k+1:n)$ prezrcalimo preko vektorja u, tako da podmatriki odštejemo matriko

$$2u\frac{u^T A(k:n,k+1:n)}{u^T u}. \tag{18.2}$$

Na k-tem koraku prezrcalimo le podmatriko $k:n \times k:n$, ostali deli matrike pa ostanjejo nespremenjeni.

Takojšnje razlaganje podrobnosti, brez predhodnega opisa osnovne ideje, ni dobro. Bralec težko loči, kaj je zares pomembno in kaj je zgolj manj pomembna podrobnost.

SLABO! Takoj dodamo vse podrobnosti, ne da bi razložili zakaj

Za vsak k, poiščemo vektor $u = [s(k) + A_{k,k}, A_{k+1,k}, ...A_{n,k}]$, kjer je $s(k) = \text{sign}(A_{k,k}) * \|[A_{k,k}, ..., A_{n,k}]\|$.

Nato matriko popravimo

$$A(k:n, k+1:n) = A(k:n, k+1:n) - 2 * u * \frac{u^T * A(k:n, k+1:n)}{u^T * u}. \quad (18.3)$$

Če implementacija vsebuje posebnosti, kot na primer uporaba posebne podatkovne strukture ali algoritma, jih lahko opišemo v poročilu. Vendar pazimo, da bralca ne obremenjujemo s podrobnostmi.

DOBRO! Posebnosti implementacije opišemo 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. ...

18.1.3 Kako pisati teste

Nekaj nasvetov, kako lahko testiramo kodo.
- Na roke izračunajte rešitev za preprost primer in jo primerjajte z rezultati funkcije.
- Ustvarite testne podatke, za katere je znana rešitev. Na primer za testiranje kode, ki reši sistem `Ax=b`, izberete `A` in `x` in izračunate desne strani `b=A*x`.
- Preverite lastnost rešitve. Za enačbe `f(x)=0`, lahko rešitev, ki jo izračuna program preprosto vstavite nazaj v enačbo in preverite, če je enačba izpolnjena.
- Red metode lahko preverite tako, da naredite simulacijo in primerjate red

metode z redom programa, ki ga eksperimentalno določite.
- 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.

18.1.3.1 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 juliji lahko pokritost kode dobimo, če dodamo argument `coverage=true` metodi `Pkg.test`:

```
julia> import Pkg; Pkg.test("DomacaXY"; coverage=true)
```

Zgornji ukaz bo za vsako datoteko iz mape `src` ustvaril ustrezno datoteko s končnico `.cov`, v kateri je shranjena informacija o tem, kateri deli kode so bili uporabljeni med izvajanjem testov.

Za poročanje o pokritosti kode lahko uporabite paket Coverage.jl. Povzetek o pokritosti kode s testi lahko pripravite z naslednjim programom:

```
using Coverage
cov = process_folder("DomacaXY")
pokrite_vrstice, vse_vrstice = get_summary(cov)
delez = pokrite_vrstice / vse_vrstice
println("Pokritost kode s testi: $(round(delez*100))%.")
```

18.1.4 Priprava zahteve za združitev na Github

Za lažjo komunikacijo predlagam, da rešitev domače naloge postavite v svojo vejo in ustvarite zahtevo za združitev (*Pull request* na Githubu oziroma *Merge request* na Gitlabu). V nadaljevanju bomo opisali, kako to storiti, če repozitorij z domačimi nalogami gostite na Githubu. Postopek za Gitlab in druge platforme je podoben.

Preden začnete z delom, ustvarite vejo na svoji delovni kopiji repozitorija in jo potisnete na Github ali Gitlab. Ime veje naj bo `domača-X`, se pravi `domaca-1` za 1. domačo nalogo in tako naprej. To storite z ukazom

```
$ git checkout -b domaca-1
$ git push -u origin domaca-1
```

Stikalo `-u` pove git-u, da naj z domačo vejo sledi veji na Githubu/Gitlabu.

Med delom sproti dodajate vnose z `git commit` in jih prenesete na splet z ukazom `git push`. Ko je domača naloga končana, na Githubu ustvarite zahtevo za združitev (angl. Pull request).

- Kliknete na zavihek `Pull requests` in nato na zelen gumb `Create pull request`.
- Na desni strani izberete vejo `domaca-1` in kliknete na gumb `Create draft pull request`.
- Ko je koda pripravljena na pregled, kliknite na gumb `Ready for review`.
- V komentarju za novo ustvarjeno zahtevo povabite asistenta k pregledu. To storite tako, da v komentar dodate uporabniško ime asistenta (npr. `@mojZlobniAsistent`).

```
@mojZlobniAsistent Prosim za pregled.
```

Pri domačih nalogah se posvetujte s kolegi
Nič ni narobe, če za pomoč pri domači nalogi prosite kolega. Seveda morate kodo in poročilo napisati samo, lahko pa kolega prosite za pregled ali za pomoč, če vam kaj ne dela. Domačo nalogo tudi napišete v skupini, vendar morate v tem primeru rešiti toliko različnih nalog, kot je študentov v skupini.

18.2 1. domača naloga

Izberite eno izmed spodnjih nalog.

Naloge

18.2.1 SOR iteracija za razpršene matrike 103
18.2.2 Metoda konjugiranih gradientov za razpršene matrike 104
18.2.3 Metoda konjugiranih gradientov s pred-pogojevanjem 104

18.2.4 QR razcep zgornje hessenbergove matrike 104
18.2.5 QR razcep simetrične tridiagonalne matrike 105
18.2.6 Inverzna potenčna metoda za zgornje hessenbergovo matriko 105
18.2.7 Inverzna potenčna metoda za tridiagonalno matriko 106
18.2.8 Naravni zlepek 107
18.2.9 QR iteracija z enojnim premikom 107

18.2.1 SOR iteracija za razpršene matrike

Naj bo A $n \times n$ diagonalno dominantna razpršena matrika (velika večina elementov je ničelnih $a_{ij} = 0$).

Definirajte nov podatkovni tip `RazprsenaMatrika`, ki matriko zaradi prostorskih zahtev hrani v dveh matrikah V in I, kjer sta V in I matriki $n \times m$, tako da velja

$$V(i,j) = A(i, I(i,j)). \tag{18.4}$$

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 `RazprsenaMatrika` definirajte 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.

Napišite 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

$$|A\boldsymbol{x}^{(k)} - \boldsymbol{b}|_\infty < \delta, \tag{18.5}$$

kjer je δ podan s argumentom `tol`.

Metodo uporabite za vložitev grafa v ravnino ali prostor s fizikalno metodo. Če so (x_i, y_i, z_i) koordinate vozlišč grafa v prostoru, potem vsaka koordinata posebej zadošča enačbam

$$\begin{aligned} -\mathrm{st}(i)x_i + \sum_{j \in N(i)} x_j &= 0, \\ -\mathrm{st}(i)y_i + \sum_{j \in N(i)} y_j &= 0, \\ -\mathrm{st}(i)z_i + \sum_{j \in N(i)} z_j &= 0, \end{aligned} \tag{18.6}$$

kjer je $st(i)$ stopnja i-tega vozlišča, $N(i)$ pa množica indeksov sosednjih vozlišč. Če nekatera vozlišča fiksiramo, bodo ostala zavzela ravnovesno lego med fiksiranimi vozlišči. Napišite funkcijo `ravnovesni_sistem`, ki za dani graf in koordinate vozlišč, ki so fiksirana, vrne matriko sistema in desne strani enačb za posamezne koordinate za vozlišča, ki niso fiksirana.

Za primer lahko upodobite [graf krožno lestev](https://en.wikipedia.org/wiki/Ladder_graph#Circular_ladder_graph), kjer polovica vozlišč enakomerno razporedite na enotski krožnici.

Za risanje grafa lahko uporabite GraphRecipes.jl.

Za primere, ki jih boste opisali, poiščite optimalni ω, pri katerem SOR najhitreje konvergira in predstavite odvisnost hitrosti konvergence od izbire ω.

18.2.2 Metoda konjugiranih gradientov za razpršene matrike

Definirajte nov podatkovni tip `RazprsenaMatrika`, kot je opisano v prejšnji nalogi.

Napišite funkcijo `[x,i]=conj_grad(A, b)`, ki reši sistem

$$Ax = b, \tag{18.7}$$

z metodo konjugiranih gradientov za `A` tipa `RazprsenaMatrika`.

Metodo uporabite na primeru vložitve grafa v ravnino ali prostor s fizikalno metodo, kot je opisano v prejšnji nalogi.

18.2.3 Metoda konjugiranih gradientov s pred-pogojevanjem

Za pohitritev konvergence iterativnih metod, se velikokrat izvede t. i. pred-pogojevanje(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 *sparse* matriko. Poglejte 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 linearni sistem

$$Ax = b \tag{18.8}$$

s pred-pogojeno metodo konjugiranih gradientov za matriko $M = L^T L$ kot pred-pogojevalcem. Pri tem pazite, da matrike M ne izračunate, ampak uporabite razcep $M = L^T L$. Za različne primere preverite, ali se izboljša hitrost konvergence.

18.2.4 QR razcep zgornje hessenbergove matrike

Naj bo H $n \times n$ zgornje hessenbergova matrika (velja $a_{ij} = 0$ za $j < j - 2i$). Definirajte podatkovni tip `ZgornjiHessenberg` za zgornje hessenbergovo matriko.

Napišite funkcijo `Q, R = qr(H)`, ki izvede QR razcep matrike `H` tipa `ZgornjiHessenberg` z Givensovimi rotacijami. Matrika `R` naj bo zgornje 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

$$[\cos(\alpha); \sin(\alpha)], \tag{18.9}$$

kjer je α kot rotacije na posameznem koraku. Za podatkovni tip definirajte še množenje `Base.*` z vektorji in matrikami.

Uporabite QR razcep za QR iteracijo zgornje hesenbergove 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 metodami `qr` in `eigen` za navadne matrike.

18.2.5 QR razcep simetrične tridiagonalne matrike

Naj bo A $n \times n$ simetrična tridiagonalna matrika (velja $a_{ij} = 0$ za $|i - j| > 1$).

Definirajte podatkovni tip `SimetricnaTridiagonalna` za simetrično tridiagonalno matriko, ki hrani glavno in stransko diagonalo matrike. Za tip `SimetricnaTridiagonalna` definirajte 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 linearna. Več informacij o tipih in Napišite funkcijo `Q, R = qr(T)`, ki izvede QR razcep matrike `T` tipa `Tridiagonalna` z Givensovimi rotacijami. Matrika `R` naj bo zgornje trikotna tridiagonalna matrika tipa `ZgornjeTridiagonalna`, v `Q` pa naj bo matrika tipa `Givens`. vmesnikih.

Podatkovna tipa `ZgornjeTridiagonalna` in `Givens` definirajte sami (glejte tudi nalogo Poglavje 18.2.4. Poleg tega implementirajte množenje `Base.*` matrik tipa `Givens` in `ZgornjeTridiagonalna`.

Uporabite QR razcep za QR iteracijo simetrične tridiagonalne matrike. Napišite funkcijo `lastne_vrednosti, lastni_vektorji = eigen(T)`, ki poišče lastne vrednosti in lastne vektorje simetrične tridiagonalne matrike.

Preverite časovno zahtevnost vaših funkcij in ju primerjajte z metodami `qr` in `eigen` za navadne matrike.

18.2.6 Inverzna potenčna metoda za zgornje 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

$$x^{(n+1)} = \frac{A_\lambda^{-1} x^{(n)}}{|A_\lambda^{-1} x^{(n)}|}, \tag{18.10}$$

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 zgornje hessenbergovo obliko (velja $a_{ij} = 0$ za $j < i - 2$). S hausholderjevimi zrcaljenji lahko poiščemo zgornje hesenbergovo matriko H, ki je podobna matriki A:

$$H = Q^T A Q. \tag{18.11}$$

Če je v lastni vektor matrike H, je Qv lastni vektor matrike A, lastne vrednosti matrik H in A pa so enake.

Napišite funkcijo `H, Q = hessenberg(A)`, ki s Hausholderjevimi zrcaljenji poišče zgornje hesenbergovo matriko `H` tipa `ZgornjiHessenberg`, ki je podobna matriki `A`.

Tip `ZgornjiHessenberg` definirajte sami, kot je opisano v nalogi o QR razcepu zgornje hessenbergove matrike. Poleg tega implementirajte metodo `L, U = lu(A)` za matrike tipa `ZgornjiHessenberg`, ki bo pri razcepu upoštevala lastnosti zgornje hessenbergovih matrik. Matrika `L` naj ne bo polna, ampak tipa `SpodnjaTridiagonalna`. Tip `SpodnjaTridiagonalna` definirajte sami, tako da bo hranil le neničelne elemente in za ta tip matrike definirajte operator `Base.\`, tako da bo upošteval strukturo matrikw `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 preskusite za izračun ničel polinoma. Polinomu

$$x^n + a_{\{n-1\}}x^{\{n-2\}} + ...a_1x + a_0 \tag{18.12}$$

lahko priredimo matriko

$$\begin{pmatrix} 0 & 0 & ... & 0 & -a_0 \\ 1 & 0 & ... & 0 & -a_1 \\ 0 & 1 & ... & 0 & -a_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & ... & 1 & -a_{n-1} \end{pmatrix} \tag{18.13}$$

katere lastne vrednosti se ujemajo z ničlami polinoma.

18.2.7 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

$$x^{\{(n+1)\}} = \frac{A_\lambda^{-1}x^{(n)}}{|A_\lambda^{-1}x^{(n)}|}, \tag{18.14}$$

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 hausholderjevimi zrcaljenji lahko poiščemo tridiagonalno matriko T, ki je podobna matriki A:

$$T = Q^TAQ. \tag{18.15}$$

Če je v lastni vektor matrike T, je Qv lastni vektor matrike A, lastne vrednosti matrik T in A pa so enake.

Napišite funkcijo `T, Q = tridiag(A)`, ki s Hausholderjevimi zrcaljenji poišče tridiagonalno matriko `H` tipa `Tridiagonalna`, ki je podobna matriki `A`.

Tip `Tridiagonalna` definirajte sami, kot je opisano v nalogi o QR razcepu tridiagonalne matrike. Poleg tega implementirajte metodo `L, U = lu(A)` za matrike tipa `Tridiagonalna`, ki bo pri razcepu upoštevala lastnosti tridiagonalnih matrik. Matrike `L` in `U` naj ne bodo polne matrike. Matrika `L` naj bo tipa `SpodnjaTridiagonalna`, matrika `U` pa tipa `ZgornjaTridiagonalna`. Tipa `SpodnjaTridiagonalna` in `ZgornjaTridiagonalna` definirajte sami, tako da bosta hranila le neničelne elemente. Za oba tipa definirajte operator `Base.\`, tako da bo upošteval strukturo matrik.

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 preskusite na laplaceovi matriki, ki ima vse elemente 0 razen $l_{ii} = -2, l_{i+1,j} = l_{i,j+1} = 1$. Poiščite nekaj lastnih vektorjev za najmanjše lastne vrednosti in jih vizualizirajte z ukazom `plot`.

Lastni vektorji laplaceove matrike so približki za rešitev robnega problema za diferencialno enačbo

$$y''(x) = \lambda^2 y(x), \tag{18.16}$$

katere rešitve sta funkciji $\sin(\lambda x)$ in $\cos(\lambda x)$.

18.2.8 Naravni zlepek

Danih je n interpolacijskih točk (x_i, f_i), $i = 1, 2...n$. **Naravni interpolacijski kubični zlepek** S je funkcija, ki izpolnjuje naslednje pogoje:

1. $S(x_i) = f_i, \quad i = 1, 2...n.$
2. S je polinom stopnje 3 ali manj na vsakem podintervalu $[x_i, x_{i+1}]$, $i = 1, 2...n-1$.
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

$$S(x) = S_{i(x)} = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3, \quad x \in [x_i, x_{i+1}], \tag{18.17}$$

nato pa izpolnimo zahtevane pogoje[2].

Napišite funkcijo `Z = interpoliraj(x, y)`, ki izračuna koeficient polinoma S_i in vrne element tipa `Zlepek`.

Tip `Zlepek` definirajte sami in naj vsebuje koeficiente polinoma in interpolacijske točke. Za tip `Zlepek` napišite 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`).

18.2.9 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 $A = U^T T U$ (uporabite lahko vgrajeno funkcijo `LinearAlgebra.hessenberg`)
- Za tridiagonalno matriko T ponavljaj, 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 `lambda` 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 nalogo QR razcep simetrične tridiagonalne matrike).

[2] pomagajte si z: Bronštejn, Semendjajev, Musiol, Mühlig: **Matematični priročnik**, Tehniška založba Slovenije, 1997, str. 754 ali pa J. Petrišič: **Interpolacija**, Univerza v Ljubljani, Fakulteta za strojništvo, Ljubljana, 1999, str. 47

Funkcijo preiskusi na Laplaceovi matriki grafa podobnosti (glej vajo o spektralnem gručenju).

18.3 2. domača naloga

Tokratna domača naloga je sestavljena iz dveh delov. V prvem delu morate implementirati program za računanje vrednosti dane funkcije $f(x)$. V drugem delu pa izračunati eno samo številko. Obe nalogi rešite na **10 decimalk** (z relativno natančnostjo $\mathbf{10^{-10}}$) 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.

Namen te naloge ni, da na internetu poiščete optimalen algoritem in ga implementirate, ampak da uporabite znanje, ki smo ga pridobilili pri tem predmetu, čeprav na koncu rešitev morda ne bo optimalna. Uporabite lahko interpolacijo ali aproksimacijo s polinomi, integracijske formule, Taylorjevo vrsto, zamenjave spremenljivk, itd. Kljub temu pazite na **časovno in prostorsko zahtevnost**, saj bo od tega odvisna tudi ocena.

Izberite **eno** izmed nalog. Domačo nalogo lahko delate skupaj s kolegi, vendar morate v tem primeru rešiti toliko različnih nalog, kot je študentov v skupini.

Če uporabljate drug programski jezik, ravno tako kodi dodajte osnovno dokumentacijo, teste in demo.

Naloge

18.3.1 Naloge s funkcijami 108
18.3.2 Naloge s števili 109
18.3.3 Lažje naloge (ocena največ 9) 111

18.3.1 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.

Naloge

18.3.1.1 Fresnelov integral (težja) 108
18.3.1.2 Funkcija kvantilov za $N(0, 1)$ 109
18.3.1.3 Integralski sinus (težja) 109
18.3.1.4 Naravni parameter (težja) 109

18.3.1.1 Fresnelov integral (težja)

Napišite učinkovito funkcijo, ki izračuna vrednosti Fresnelovega kosinusa

$$C(x) = \int_0^x \cos\left(\frac{\pi t^2}{2}\right) dt. \tag{18.18}$$

Namig: Uporabite pomožni funkciji

$$
\begin{aligned}
f(z) &= \frac{1}{\pi\sqrt{2}} \int_0^\infty \frac{e^{-\frac{\pi z^2 t}{2}}}{\sqrt{t}(t^2+1)} dt \\
g(z) &= \frac{1}{\pi\sqrt{2}} \int_0^\infty \frac{\sqrt{t} e^{-\frac{\pi z^2 t}{2}}}{t^2+1} dt,
\end{aligned}
\tag{18.19}
$$

kot je opisano v [9].

18.3.1.2 Funkcija kvantilov za $N(0,1)$

Napišite učinkovito funkcijo, ki izračuna funkcijo kvantilov za standardno normalno porazdeljeno slučajno spremenljivko. Funkcija kvantilov je inverzna funkcija $\Phi^{-1}(x)$ porazdelitvene funkcije:

$$
\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-\frac{t^2}{2}} dt. \tag{18.20}
$$

Poskrbite, da bo relativna napaka za vrednosti blizu 0 in 1 dovolj majhna in da je časovna zahtevnost omejena z isto konstanto na celem intervalu $(0,1)$.

18.3.1.3 Integralski sinus (težja)

Napišite učinkovito funkcijo, ki izračuna integralski sinus

$$
\mathrm{Si}(x) = \int_0^x \frac{\sin(t)}{t} dt. \tag{18.21}
$$

Uporabite pomožni funkciji

$$
\begin{aligned}
f(z) &= \int_0^\infty \frac{\sin(t)}{t+z} = \int_0^\infty \frac{e^{-zt}}{t^2+1} dt \\
g(z) &= \int_0^\infty \frac{\cos(t)}{t+z} = \int_0^\infty \frac{te^{-zt}}{t^2+1} dt \\
\mathrm{Si}(z) &= \frac{\pi}{2} - f(z)\cos(z) - g(z)\sin(z),
\end{aligned}
\tag{18.22}
$$

kot je opisano v [10].

18.3.1.4 Naravni parameter (težja)

Napišite učinkovito funkcijo, ki izračuna naravni parameter:

$$
s(t) = \int_0^t \sqrt{\dot{x}(\tau)^2 + \dot{y}(\tau)^2} d\tau \tag{18.23}
$$

za parametrično krivuljo

$$
(x(t), y(t)) = (t^3 - t, t^2 - 1). \tag{18.24}
$$

Za velike vrednosti argumenta t aproksimirajte funkcijo $s\left(\frac{1}{t}\right)^{-1}$ s polinomom.

18.3.2 Naloge s števili

Naloge

18.3.2.1 Sila težnosti 110
18.3.2.2 Ploščina hipotrohoide 110
18.3.2.3 Povprečna razdalja (težja) 110
18.3.2.4 Ploščina Bézierove krivulje 110

18.3.2.1 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

$$\boldsymbol{F} = \int_{T_1} \int_{T_2} \frac{\boldsymbol{r_1} - \boldsymbol{r_2}}{\| \boldsymbol{r_1} - \boldsymbol{r_2} \|^2} d\boldsymbol{r}_1 d\boldsymbol{r}_2. \tag{18.25}$$

18.3.2.2 Ploščina hipotrohoide

Izračunajte ploščino območja, ki ga omejuje hypotrochoida podana parametrično z enačbama:

$$x(t) = (a+b)\cos(t) + b\cos\left(\frac{a+b}{b}t\right) \tag{18.26}$$

$$y(t) = (a+b)\sin(t) + b\sin\left(\frac{a+b}{b}t\right) \tag{18.27}$$

za parametra $a = 1$ in $b = -\frac{11}{7}$.

Namig: Uporabite formulo za ploščino krivočrtnega trikotnika pod krivuljo:

$$P = \frac{1}{2}\int_{t_1}^{t_2} (x(t)\dot{y}(t) - \dot{x}(t)y(t))dt \tag{18.28}$$

18.3.2.3 Povprečna razdalja (težja)

Izračunajte povprečno razdaljo med dvema točkama znotraj telesa T, ki je enako razliki dveh kock:

$$T = ([-1,1])^3 - ([0,1])^3. \tag{18.29}$$

Integral na produktu razlike dveh množic $(A-B)\times(A-B)$ lahko izrazimo kot vsoto integralov:

$$\begin{aligned}\int_{A-B}\int_{A-B} f(x,y)dxdy = &\int_A\int_A f(x,y)dxdy \\ &-2\int_A\int_B f(x,y)dxdy + \int_B\int_B f(x,y)dxdy\end{aligned} \tag{18.30}$$

18.3.2.4 Ploščina Bézierove krivulje

Izračunajte ploščino zanke, ki jo omejuje Bézierova krivulja dana s kontrolnim poligonom:

$$(0,0),(1,1),(2,3),(1,4),(0,4),(-1,3),(0,1),(1,0). \tag{18.31}$$

Namig: Uporabite lahko formulo za ploščino krivočrtnega trikotnika pod krivuljo:

$$P = \frac{1}{2}\int_{t_1}^{t_2} (x(t)\dot{y}(t) - \dot{x}(t)y(t))dt. \tag{18.32}$$

18.3.3 Lažje naloge (ocena največ 9)

Naloge so namenjen tistim, ki jih je strah eksperimentiranja ali pa za to preprosto nimajo interesa ali časa. Rešiti morate eno od nalog:

18.3.3.1 Gradientni spust z iskanjem po premici

18.3.3.2 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 Lagrangevo interpolacijo, po formuli

$$l(x) = \begin{cases} \frac{\sum \frac{f(x_j)\lambda_j}{x-x_j}}{\sum \frac{\lambda_j}{x-x_j}} & x \neq x_j \\ f(x_j) & \text{sicer} \end{cases} \tag{18.33}$$

Čebiševe točke so podane na intervalu $[-1, 1]$ s formulo

$$x_k = \cos\left(\frac{2k-1}{2n}\pi\right), \quad k = 0, 1 \ldots n-1, \tag{18.34}$$

vrednosti uteži λ_k pa so enake

$$\lambda_k = (-1)^k \begin{cases} 1 & 0 < i < n \\ \frac{1}{2} & i = 0 \\ n & \text{sicer.} \end{cases} \tag{18.35}$$

Za interpolacijo na splošnem intervalu $[a, b]$ si pomagaj z linearno preslikavo na interval $[-1, 1]$. Program uporabi za tri različne funkcije e^{-x^2} na $[-1, 1]$, $\frac{\sin x}{x}$ na $[0, 10]$ in $\left|x^2 - 2x\right|$ na $[1, 3]$. Za vsako funkcijo določi stopnjo polinoma, da napaka ne bo presegla 10^{-6}.

18.3.3.3 Gauss-Legendrove kvadrature

Izpelji Gauss-Legendreovo integracijsko pravilo na dveh točkah

$$\int_0^h f(x)dx = Af(x_1) + Bf(x_2) + R_f \tag{18.36}$$

vključno s formulo za napako R_f. Izpelji sestavljeno pravilo za $\int_a^b f(x)dx$ in napiši program, ki to pravilo uporabi za približno računanje integrala. Oceni, koliko izračunov funkcijske vrednosti je potrebnih, za izračun približka za

$$\int_0^5 \frac{\sin x}{x}dx \tag{18.37}$$

na 10 decimalk natančno. *Namig*: Najprej izpelji pravilo na intervalu $[-1, 1]$ in ga nato prevedi na poljuben interval $[x_i, x_{i+1}]$. Za oceno napake uporabite izračun z dvojnim številom korakov.

18.4 3. domača naloga

18.4.1 Navodila

Zahtevana števila izračunajte na **10 decimalk** (z relativno natančnostjo $\mathbf{10^{-10}}$) 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.

Namen te naloge ni, da na internetu poiščete optimalen algoritem in ga implementirate, ampak da uporabite znanje, ki smo ga pridobili pri tem predmetu, čeprav na koncu rešitev morda ne bo optimalna. Kljub temu pazite na **časovno in prostorsko zahtevnost**, saj bo od tega odvisna tudi ocena.

Izberite **eno** izmed nalog. Domačo nalogo lahko delate skupaj s kolegi, vendar morate v tem primeru rešiti toliko različnih nalog, kot je študentov v skupini.

Če uporabljate drug programski jezik, ravno tako kodi dodajte osnovno dokumentacijo in teste.

18.4.2 Težje naloge

18.4.2.1 Ničle Airijeve funkcije

Airyjeva funkcija je dana kot rešitev začetnega problema

$$Ai''(x) - x\,Ai(x) = 0, \quad Ai(0) = \frac{1}{3^{\frac{2}{3}}\Gamma\left(\frac{2}{3}\right)}\, Ai'(0) = -\frac{1}{3^{\frac{1}{3}}\Gamma\left(\frac{1}{3}\right)}. \tag{18.38}$$

Poiščite čim več ničel funkcije Ai na 10 decimalnih mest natančno. Ni dovoljeno uporabiti vgrajene funkcijo za reševanje diferencialnih enačb. Lahko pa uporabite Airyjevo funkcijo `airyai` iz paketa `SpecialFunctions.jl`, da preverite ali ste res dobili pravo ničlo.

18.4.2.1.1 Namig

Za računanje vrednosti $y(x)$ lahko uporabite Magnusovo metodo reda 4 za reševanje enačb oblike

$$y'(x) = A(x)y, \tag{18.39}$$

pri kateri nov približek $\boldsymbol{Y}_{k+1}$ dobimo takole:

$$\begin{aligned} A_1 &= A\left(x_k + \left(\frac{1}{2} - \frac{\sqrt{3}}{6}\right)h\right) \\ A_2 &= A\left(x_k + \left(\frac{1}{2} + \frac{\sqrt{3}}{6}\right)h\right) \\ \sigma_{k+1} &= \frac{h}{2}(A_1 + A_2) - \frac{\sqrt{3}}{12}h^2[A_1, A_2] \\ \boldsymbol{Y}_{k+1} &= \exp(\sigma_{k+1})\boldsymbol{Y}_k. \end{aligned} \tag{18.40}$$

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 programskem jeziku julia dobite z ukazom `exp`.

18.4.2.2 Dolžina implicinto podane krivulje

Poiščite približek za dolžino krivulje, ki je dana implicitno z enačbama

$$\begin{aligned} F_1(x, y, z) &= x^4 + y^2/2 + z^2 = 12 \\ F_2(x, y, z) &= x^2 + y^2 - 4z^2 = 8. \end{aligned} \tag{18.41}$$

Krivuljo lahko poiščete kot rešitev diferencialne enačbe

$$\dot{\boldsymbol{x}}(t) = \nabla F_1 \times \nabla F_2. \tag{18.42}$$

18.4.2.3 Perioda limitnega cikla

Poiščite periodo limitnega cikla za diferencialno enačbo

$$x''(t) - 4(1 - x^2)x'(t) + x = 0 \tag{18.43}$$

na 10 decimalk natančno.

18.4.2.4 Obhod lune

Sondo Appolo pošljite iz Zemljine orbite na tir z vrnitvijo brez potiska (free-return trajectory), 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.

18.4.2.4.1 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

$$\mu = \frac{m}{M+m} \quad \text{ter} \quad \bar{\mu} = 1 - \mu = \frac{M}{M+m}\ . \tag{18.44}$$

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 $\left(\bar{\mu}, 0, 0\right)$. Označimo z R in r oddaljenost satelita s položajem (x, y, z) od mas M in m, tj.

$$\begin{aligned} R &= R(x, y, z) = \sqrt{(x+\mu)^2 + y^2 + z^2}, \\ r &= r(x, y, z) = \sqrt{\left(x-\bar{\mu}\right)^2 + y^2 + z^2}. \end{aligned} \tag{18.45}$$

Enačbe gibanja sonde so potem:

$$
\begin{aligned}
\ddot{x} &= x + 2\dot{y} - \frac{\bar{\mu}}{R^3}(x+\mu) - \frac{\mu}{r^3}\left(x - \bar{\mu}\right),\\
\ddot{y} &= y - 2\dot{x} - \frac{\bar{\mu}}{R^3}y - \frac{\mu}{r^3}y,\\
\ddot{z} &= -\frac{\bar{\mu}}{R^3}z - \frac{\mu}{r^3}z.
\end{aligned}
\tag{18.46}
$$

18.4.2.5 Perioda geostacionarne orbite

Oblika planeta Zemlja ni čisto pravilna krogla. Zato tudi gravitacijsko polje ne deluje v vseh smereh enako. Gravitacijsko polje lahko zapišemo kot odvod gravitacijskega potenciala

$$
F_{g(\boldsymbol{r})} = m \cdot \nabla V(\boldsymbol{r}),
\tag{18.47}
$$

kjer je $V(\boldsymbol{r})$ skalarna funkcija položaja $\boldsymbol{r}$. Zemljina gravitacija Zemljin gravitacijski potencial.

18.4.3 Lažja naloga (ocena največ 9)

Naloga je namenjena tistim, ki jih je strah eksperimentiranja ali pa za to preprosto nimajo interesa ali časa.

18.4.3.1 Matematično nihalo

Kotni odmik $\theta(t)$ (v radianih) pri nedušenem nihanju nitnega nihala opišemo z diferencialno enačbo

$$
\frac{g}{l}\sin(\theta(t)) + \theta''(t) = 0, \quad \theta(0) = \theta_0, \theta'(0) = \theta'_0,
\tag{18.48}
$$

kjer je $g = 9.80665 m/s^2$ težni pospešek in l dolžina nihala. Napišite funkcijo `nihalo`, ki računa odmik nihala ob določenem času. Enačbo drugega reda prevedite na sistem prvega reda in računajte z metodo Runge-Kutta četrtega reda:

$$
\begin{aligned}
k_1 &= h\, f(x_n, y_n)\\
k_2 &= h\, f(x_n + h/2, y_n + k_1/2)\\
k_3 &= h\, f(x_n + h/2, y_n + k_2/2)\\
k_4 &= h\, f(x_n + h, y_n + k_3)\\
y_{n+1} &= y_n + (k_1 + 2k_2 + 2k_3 + k_4)/6.
\end{aligned}
\tag{18.49}
$$

Klic funkcije naj bo oblike `odmik=nihalo(l,t,theta0,dtheta0,n)`

- kjer je `odmik` enak odmiku nihala ob času `t`,
- dolžina nihala je `l`,
- začetni odmik (odmik ob času 0) je `theta0`
- in začetna kotna hitrost ($\theta'(0)$) je `dtheta0`,
- interval $[0, t]$ razdelimo na `n` podintervalov enake dolžine.

Primerjajte rešitev z nihanjem harmoničnega nihala. Za razliko od harmoničnega nihala (sinusno nihanje), je pri matematičnem nihalu nihajni čas odvisen od začetnih pogojev (energije). Narišite graf, ki predstavlja, kako se nihajni čas spreminja z energijo nihala.

Literatura

[1] B. Orel, *Osnove numerične matematike*. 2020.

[2] D. E. Knuth, „Literate programming“, *The Computer Journal*, let. 27, št. 2, str. 97–111, 1984, doi: 10.1093/comjnl/27.2.97.

[3] Savchenko V. V., Pasko, A. A., Okunev, O. G., in Kunii T. L., „Function representation of solids reconstructed from scattered surface points and contours“, *Computer Graphics Forum*, let. 14, št. 4, str. 181–188, 1995, Pridobljeno: 22. julij 2024. [Na spletu]. Dostopno na: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.48.80&rep=rep1&type=pdf

[4] Turk G. in O'Brien J., „Variational Implicit Surfaces“. 1999. Pridobljeno: 22. julij 2024. [Na spletu]. Dostopno na: https://www.semanticscholar.org/paper/Variational-Implicit-Surfaces-Turk-O'Brien/50dbc9f86af75dad7be6b2e92601e4ded7bee2d6

[5] M. Buhmann, „Radial Basis Functions“, v *Encyclopedia of Applied and Computational Mathematics*, B. Engquist, Ur., 2015, str. 1216–1219.

[6] B. S. Morse, T. S. Yoo, P. Rheingans, D. T. Chen, in K. R. Subramanian, „Interpolating implicit surfaces from scattered surface data using compactly supported radial basis functions“. str. 89–98, maj 2001. doi: 10.1109/SMA.2001.923379.

[7] N. M. Temme, „Digital Library of Mathematical Functions: Chapter 3 Numerical Methods“. Pridobljeno: 15. junij 2024. [Na spletu]. Dostopno na: https://dlmf.nist.gov/3

[8] L. Trefethen, *Approximation Theory and Approximation Practice, Extended Edition*. 2019. doi: 10.1137/1.9781611975949.

[9] N. M. Temme, „Digital Library of Mathematical Functions: Chapter 7 Error Functions, Dawson's and Fresnel Integrals“. Pridobljeno: 15. junij 2024. [Na spletu]. Dostopno na: https://dlmf.nist.gov/7

[10] N. M. Temme, „Digital Library of Mathematical Functions: Chapter 6 Exponential, Logarithmic, Sine, and Cosine Integrals“. Pridobljeno: 15. junij 2024. [Na spletu]. Dostopno na: https://dlmf.nist.gov/6