Numerična matematika v programskem jeziku Julia

Martin Vuk

2024

Predgovor

Ta knjiga vsebuje gradiva za izvedbo laboratorijskih vaj pri predmetu Numerična matematika na Fakulteti za računalništvo in informatiko Univerze v Ljubljani. Kljub temu je primerna za vse, ki bi se želeli bolje spoznati z uporabo numeričnih metod in se naučiti uporabljati programski jezik Julia. Predpostavljamo, da je bralec vešč programiranja v kakem drugem programskem jeziku.

Knjige o numerični matematiki se pogosto posvečajo predvsem matematičnim vprašanjem. Vsebina te knjige pa poskuša nasloviti bolj praktične vidike numerične matematike. Tako so primeri, če je le mogoče, povezani s problemom praktične narave s področja fizke, matematičnega modeliranja in računalništva, pri katerih lahko za rešitev problema uporabimo numerične metode.

Bralcu svetujemo, da vso kodo napiše in preskusi sam. Še bolje je, če kodo razširi in spreminja. Bralca spodbujamo, da se čim bolj igra z napisano kodo. Koda, ki je navedena v tej knjigi, je minimalna različica kode, ki reši določen problem in še ustreza minimalnim standardom pisanja kvalitetne kode. Pogosto izpustimo preverjanje ali implementacijo robnih primerov. Včasih opustimo obravnavo pričakovanih napak. Prednost smo dali berljivosti pred kompletnostjo, da je bralcu lažje razumeti, kaj koda počne.

Vaje so zasnovane za samostojno delo z računalnikom. Vsaka vaja se začne z opisom naloge in jasnimi navodili, kaj na naj bo končni rezultat. Nato sledijo podrobnejša navodila, kako se naloge lotiti. Na koncu je rešitev z razlago posameznih korakov. Na ta način lahko študentje rešijo nalogo z različno mero samostojnosti. Na koncu je rešitev naloge, ki ji lahko študentje sledijo brez dodatnega dela. Rešitev vključuje matematične izpeljave, programsko kodo in rezultate, ki jih dobimo, če programsko kodo uporabimo.

Domače naloge so brez rešitev in naj bi jih študenti oziroma bralec rešili povsem samostojno. Nekatere stare izpitne naloge so rešene, večina pa nima rešitve. Odločitev, da niso vključene rešitve za vse izpitne naloge je namerna, saj bralec lahko verodostojno preveri svoje znanje le, če rešuje tudi naloge, za katere nima dostopa do rešitev.

Kljub temu, da je knjiga namenjena študentom, je zasnovana tako, da je primerna za vse, ki bi se radi naučili uporabljati in implementirati osnovne algoritme numerične matematike. Primeri programov so napisani v programskem jeziku Julia.

Na tem mestu bi se rad zahvalil Bojanu Orlu, Emilu Žagarju, Petru Kinku in Aljažu Zalarju, s katerimi sem sodeloval 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* 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 zahvalil študentom, ki so obiskovali numerične predmete, ki sem jih učil in so me naučili marsikaj novega.

Kazalo

1 Uvod v programski jezik Julia

V knjigi bomo za implementacijo algoritmov in ilustracijo uporabe izbrali programski jezik julia. Zavoljo učinkovitega izvajanja, uporabe dinamičnih tipov, funkcij specializiranih glede na signaturo in dobre podporo za interaktivno uporabo, je julia zelo primerna za implementacijo numeričnih metod in ilustracijo njihove uporabe. V nadaljevanju sledijo kratka navodila, kako začeti z `julio` in si pripraviti delovno okolje v katerem bo pisanje kode steklo čim bolj gladko.

Cilji tega poglavja so
- da se naučimo uporabljati Julio v interaktivni ukazni zanki,
- da si pripravimo okolje za delo v programskem jeziku Julia,
- da ustvarimo prvi paket in
- da ustvarimo prvo poročilo v formatu PDF.

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

1.1 Namestitev in prvi koraki

Sledite navodilom in namestite programski jezik Julia. Nato lahko v terminalu poženete ukaz `julia`. Ukaz odpre interaktivno ukazno zanko (angl. *Read Eval Print Loop* ali s kratico REPL) in v terminalu se pojavi ukazni poziv `julia>`. Za ukazni poziv 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 so v programskem jeziku Julia osnovne enote kode. Lahko jih definiramo na več načinov. Kratke funkcije, ki jih lahko definiramo v eni vrstici, lahko definiramo z izrazom `ime(x) = ...`.

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

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

Funkcija ima lahko tudi več argumentov.

```
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)
```

Funkcije lahko uporabljamo kot vsako drugo spremenljivko. Lahko jih podamo kot argumente drugim funkcijam, združujemo v podatkovne strukture, kot so seznami, vektorji ali matrike. Funkcije lahko definiramo tudi kot anonimne funkcije. To so funkcije, ki jih ne moremo poklicati po imenu, lahko pa jih podajamo kot argumente v druge funkcije.

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

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

1.1.2 Vektorji in matrike

Vektorje lahko vnesemo z oglatimi oklepaji `[]`:

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

julia> v[1] # prvi indeks je 1
1

julia> v[2:end] # zadnja dva elementa
2-element Vector{Int64}:
 2
 3

julia> sin.(v) # funkcije lahko apliciramo na elementih (operator .)
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

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

Osnovne operacije delujejo tudi za vektorje in matrike. Pri tem moramo vedeti, da gre za matrične operacije. Tako je * 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.

```
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 $Ax = b$. Operator `\` deluje za veliko različnih primerov. Med drugim ga lahko uporabimo tudi za iskanje rešitve pre-določenega sistema po metodi najmanjših kvadratov.

```
julia> A = [1 2; 3 4];

julia> x =  A \ [5, 6] # rešimo enačbo A * x = [5, 6]
2-element Vector{Float64}:
 -3.9999999999999987
 4.499999999999999

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

1.1.3 Paketi

Nabor funkcij, ki so na voljo v Juliji, je omejen. Dodatne funkcije lahko naložimo iz knjižnic. Knjižnica funkcij v Juliji se imenuje paket. Funkcije v paketu so združene v module.

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, ampak je del modula `LinearAlgebra`, ki je vključen v program Julia. Če želimo uporabiti `norm`, moramo najprej uvoziti definicije iz modula `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 iz interneta. Za to uporabimo modul `Pkg`. Paketom je namenjen poseben paketni način vnosa v ukazni zanki. Do paketnega načina pridemo, če v ukazni zanki vpišemo znak `]`.

Opomba! *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 `;`.

Oglejmo si, kako namestiti knjižnico za ustvarjanje slik in grafov Plots.jl. Najprej aktiviramo paketni način z vnosom znaka `]` v ukazno zanko. Nato paket dodamo z ukazom `add`.

```
(@v1.10) pkg> add Plots
...

julia> using Plots

julia> plot(x -> x - x^2, -1, 2, title="Graf y(x) = x - x^2 na [-1,2]")
```

1.1.4 Datoteke s kodo

Kodo lahko zapišemo tudi v datoteke. Vnašanje ukazov v interaktivni zanki je lahko uporabno namesto kalkulatorja. Vendar je za resnejše delo bolje kodo shraniti v datoteke. Praviloma imajo datoteke s kodo v jeziku Julia končnico `.jl`. Definicije funkcij in ukaze zapišemo v tekstovno datoteko s končnico `.jl`.

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

```
julia> include("demo.jl")
```

ali pa v lupini operacijskega sistema:

```
$ julia demo.jl
```

> *Opomba! Urejevalniki in programska okolja za* julio
>
> Za lažje delo z datotekami s kodo potrebujete dober urejevalnik golega besedila, ki je namenjen programiranju. Če še nimate priljubljenega urejevalnika, priporočam VS Code in razširitev za Julio.
>
> Če odprete datoteko s kodo v urejevalniku VsCode, 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 s kodo tako, da lahko čim več kode ponovno uporabimo.

1.2 Priprava delovnega okolja

Med razvojem, se datoteke s kodo nenehno spreminjajo, zato je treba kodo v interaktivni zanki vseskozi posodabljati. Paket Revise.jl poskrbi za to, da se nalaganje zgodi avtomatično, ko se datoteke spremenijo. Zato najprej namestimo paket *Revise* in poskrbimo, da se zažene ob vsakem zagonu interaktivne zanke.

Odprite julia in sledite ukazom spodaj.

```
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
```

Zgornji ukazi ustvarijo mapo `$HOME/.julia/config` in datoteko `startup,jl`, ki se izvede ob vsakem zagonu programa `julia`. Okolje za delo z Julio je pripravljeno.

1.3 Priprava mape s kodo

V nadaljevanju bomo pripravili mapo, v kateri bomo hranili datoteke s kodo. Datoteke bomo organizirali tako, da bo vsaka vaja paket v svoji mapi, korenska mapa pa bo služila kot delovno okolje. Za več informacij o načinu dela si oglejte nasvete za delo z Julijo.

Opomba! *Razlika med paketom in delovnim okoljem*
Julia v datoteki `Project.toml` hrani odvisnosti od paketov. Vsaka mapa, ki vsebuje datoteko `Project.toml` je bodisi delovno okolje, bodisi paket. Delovno okolje je namenjeno interaktivnemu delu ali programom, medtem ko je paket namenjen predvsem deljenju funkcij med različnim projekti.

Pripravimo najprej korensko mapo. Imenovali jo bomo `nummat-julia`, lahko si pa izberete tudi drugo ime.

```
$ mkdir nummat-julia
$ cd nummat-julia
$ julia
```

Nato v korenski mapi pripravimo okolje s paketi in dodamo nekaj paketov, ki jih bomo potrebovali pri delu v interaktivni zanki.

```
(@v1.10) pkg> pkg> activate . # pripravimo virtualno okolje v korenski mapi
```

Zgornji ukaz ustvari datoteko `Project.toml` v mapi `nummat-julia`. Delovnemu okolju dodamo pakete, ki jih bomo potrebovali v nadaljevanju. Zaenkrat je to le paket `Plots`.

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

Opomba! *Na mojem računalniku pa koda dela!*
Izjava „Na mojem računalniku pa koda dela!“ je postala sinonim za problem ponovljivosti rezultatov, ki jih generiramo z računalnikom. Eden od mnogih faktorjev, ki vplivajo na ponovljivost, je tudi dostop do zunanjih knjižnic/paketov, ki jih naša koda uporablja in jih ponavadi ne hranimo skupaj s kodo. V `julii` lahko pakete, ki jih potrebujemo, deklariramo v datoteki `Project.toml`. Vsak direktorij, ki vsebuje datoteko `Project.toml` definira bodisi delovno okolje ali pa paket in omogoča, da preprosto obnovimo vse zunanje pakete, od katerih je odvisna naša koda. Za ponovljivost sistemskega okolja, lahko uporabimo docker, NixOS ali GNU Guix.

Priporočljivo je uporabiti tudi program za vodenje različic Git. Z naslednjim ukazom v mapi `nummat-julia` ustvarimo repozitorij za `git` in registriramo novo ustvarjene datoteke.

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

Priporočam pogosto beleženje sprememb z `git commit`. Pogoste potrditve (angl. commit) olajšajo pregledovanje sprememb in spodbujajo k razdelitvi dela na majhne zaključene probleme, ki so lažje obvladljivi.

Opomba! *Sistem za vodenje različic Git*
Git je sistem za vodenje različic, ki je postal *de facto* standard v razvoju programske opreme pa tudi drugod, kjer se dela s tekstovnimi datotekami. Predlagam, da si bralec naredi 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. Če želimo svojo kodo deliti ali pa zgolj hraniti varnostno kopijo, ki je dostopna na internetu, lahko repozitorij repliciramo na lastnem strežniku ali na enem od javnih spletnih skladišč za programsko kodo na primer Github ali Gitlab.

1.4 Priprava paketa za vajo

Ob začetku vsake vaje si bomo v mapi, ki smo jo ustvarili v prejšnjem poglavju (`nummat-julia`) 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 način dela (angl. workflow), da bo pri delu čim manj nepotrebnih motenj.

```
$ cd nummat-julia
$ julia
```

```
(@v1.10) pkg> generate Vaja00
(@v1.10) pkg> activate .
(nummat-julia) pkg> develop ./Vaja00 # paket dodamo delovnemu okolju
```

Zgornji ukazi ustvarijo direktorij `Vaja00` z osnovno strukturo paketa v Juliji.

```
$ tree Vaja00
Vaja00
├── Project.toml
└── src
    └── Vaja00.jl

1 directory, 2 files
```

Opomba! *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`.

Paketu `Vaje00` dodamo še teste, skripte in README dokument, tako da bo imela mapa naslednjo strukturo.

```
$ tree Vaje00
Vaje00
├── Manifest.toml
├── Project.toml
├── doc
│   └── demo.jl
├── src
│   └── Vaja00.jl
└── test
    └── runtests.jl
```

1.5 Koda

Ko je mapa s paketom `Vaja00` pripravljena, lahko začnemo s pisanjem kode. Za vajo bomo narisali Geronove lemniskato. Najprej definiramo koordinatne funkcije

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

Definicije shranimo v datoteki `Vaja00/src/Vaja00.jl`.

```julia
module Vaja00

"""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 `Vaja00`
export lemniskata_x, lemniskata_y
end # module Vaja00
```

Program 1: Vsebina datoteke `Vaja00.jl`.

V ukazni zanki lahko sedaj pokličemo novo definirani funkciji.

```
(@v1.10) pkg> activate .
julia> using Vaja00
julia> lemniskata_x(1.2)
0.180327868852459
```

Nadaljujemo šele, ko se prepričamo, da lahko pokličemo funkcije iz paketa `Vaja00`.

Kodo, ki bo sledila, bomo sedaj pisali v scripto `Vaja00\doc\demo.jl`.

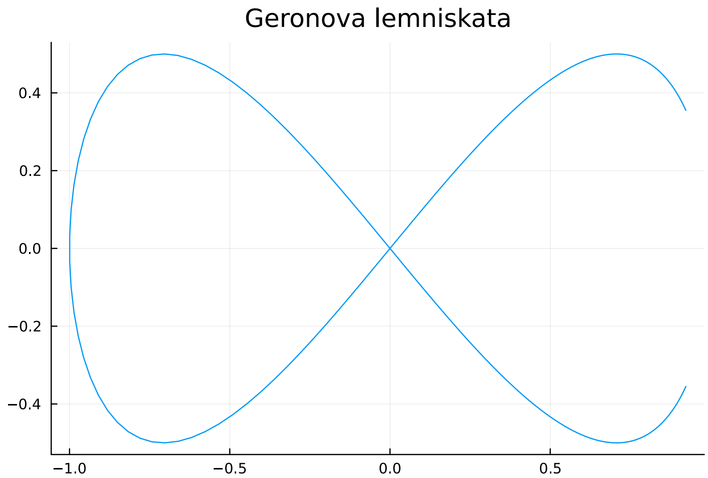
```
using Vaja00
#' Krivuljo narišemo tako, da koordinati tabeliramo za veliko število parametrov.
t = range(-5, 5, 300) # generiramo zaporedje 300 vrednosti na [-5, 5]
x = lemniskata_x.(t) # funkcijo apliciramo na elemente zaporedja
y = lemniskata_y.(t) # tako da imenu funkcije dodamo .
#' Za risanje grafov uporabimo paket `Plots`.
using Plots
plot(x, y, label=false, title="Geronova lemniskata")
```

Program 2: Vsebina datoteke `demo.jl`

Skripto poženemo z ukazom:

```
julia> include("Vaja00/doc/demo.jl")
```

Rezultat je slika lemniskate.



Slika 1: Geronova lemniskata

Opomba! *Poganjanje ukaz za ukazom v VsCode*
Če uporabljate urejevalnik VsCode in razširitev za Julio, lahko ukaze iz skripte poganjate vrstico za vrstico kar iz urejevalnika. Če pritisnete kombinacijo tipk `Shift + Enter`, se bo izvedla vrstica v kateri je trenutno kazalka.

1.6 Testi

V prejšnjem razdelku smo definirali funkcije in napisali skripto, s katero smo omenjene funkcije uporabili. Naslednji korak je, da dodamo teste, s katerimi preiskusimo pravilnost napisane kode.

Opomba! Avtomatsko testiranje programov

Pomembno je, da pravilnost programov preverimo. Najlažje to naredimo „na roke“, tako da program poženemo in preverimo rezultat. Testiranja „na roke“ ima veliko pomankljivosti. Zahteva veliko časa, je lahko nekonsistentno in 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`.

Avtomatski test je preprost program, ki pokliče določeno funkcijo in preveri rezultat. Najbolj enostavno je rezultat primerjati z v naprej znanim rezultatom, za katerega smo prepričani, da je pravilen. Uporabili bomo makroje @test in @testset iz paketa `Test`.

V datoteko `test/runtests.jl` dodamo teste za obe koordinatni funkciji, ki smo ju definirali:

```julia
using Vaja00, 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: Testi za paket `Vaja00`

Za primerjavo rezultatov smo uporabili operator ≈, ki je alias za funkcijo isapprox.

Opomba! 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, kot pričakovana zaokrožitvena napaka. Funkcija `isapprox` je namenjena ravno približni primerjavi.

Preden lahko poženemo teste, moramo ustvariti testno okolje. Sledimo priporočilom za testiranje paketov. V mapi `Vaje00/test` ustvarimo novo okolje in dodamo paket `Test`.

```
(@v1.10) pkg> activate Vaje00/test
(test) pkg> add Test
(test) pkg> activate .
```

Teste poženemo tako, da v paketnem načinu poženemo ukaz `test Vaja00`.

```
(nummat-julia) pkg> test Vaja00
Testing Vaja00
     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 Vaja00 tests passed
```

1.7 Dokumentacija

Dokumentacija programske kode je sestavljena iz različnih besedil in drugih virov, npr. videov, ki so namenjeni uporabnikom in razvijalcem programa ali knjižnice. Dokumentacija lahko vključuje komentarje v kodi, navodila za namestitev in uporabo programa in druge vire v raznih formatih z razlagami ozadja, teorije in drugih zadev povezanih s projektom. Dobra dokumentacija lahko veliko pripomore k uspehu določenega programa. Sploh to velja za knjižnice.

Tudi, če kode ne bo uporabljal nihče drug in verjamite, slabo dokumentirane kode, nihče ne želi uporabljati, bodimo prijazni do nas samih v prihodnosti in pišimo dobro dokumentacijo.

V tej knjigi bomo pisali 3 vrste dokumentacije:

- dokumentacijo posameznih funkcij in tipov,
- navodila za uporabnika v datoteki `README.md`,
- poročilo v formatu PDF.

> *Opomba! Zakaj format PDF*
>
> Izbira formata PDF je mogoče presenetljiva za pisanje dokumentacije programske kode. V praksi so precej bolj uporabne 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 na vajah pa ima format PDF še vedno prednosti. Saj ga je lažje pregledovati in popravljati.

1.7.1 Dokumentacija funkcij in tipov

Funkcije in tipe v Julii dokumentiramo tako, da pred definicijo dodamo niz z opisom funkcije. Več o tem si lahko preberete v priročniku za Julio.

1.7.2 Generiranje PDF poročila

Za pisanje dokumentacijo bomo uporabili format Markdown, ki ga bomo dodali kot komentarje v kodi. Knjižnica Weave.jl poskrbi za generiranje PDF poročila.

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 demo skripte. Uporabili bom kar `Vaja00/doc/demo.jl`, ki smo jo ustvarili, da smo generirali sliko.

V datoteko dodamo besedilo v obliki komentarjev. Komentarje, ki se začnejo z `#'`, paket `Weave` uporabi kot tekst v formatu Markdown, medtem ko se koda in navadni komentarji v poročilu izpišejo kot koda.

```julia
#' # Geronova lemniskata
#' Komentarji, ki se začnejo s `#'` se prevedejo v Markdown in
#' v PDF dokumentu nastopajo kot tekst.
using Vaja00
#' 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 tekstom se izpiše v dokument. Če je rezultat
#' tipa, ki predstavlja plot, se v dokument vstavi slika.
savefig("img/01_demo.svg")
```

Program 4: Vsebina `Vaje00/doc/demo.jl`, po tem, ko smo dodali komentarje s tekstom v formatu Markdown

Poročilo pripravimo z ukazom `Weave.weave`. Ustvarimo še eno skripto `Vaje00\doc\makedocs.jl`, v katero dodamo naslednje vrstice

```julia
using Weave
# Poročilo generiramo z ukazom `Weave.weave`
Weave.weave("Vaja00/doc/demo.jl",
  doctype="minted2pdf", out_path="Vaja00/pdf")
```

Program 5: Program za generiranje PDF dokumenta

Skripto poženemo v julii z ukazom `include("Vaja00/doc/makedocs.jl")`. Poročilo najdemo v datoteki `Vaja00/pdf/demo.pdf`.

Poleg paketa Weave.jl je na voljo še nekaj programov, ki so primerni za pripravo poročil:

- IJulia,
- Literate.jl ali
- Quadro.

Navedimo še nekaj zanimivih povezav, ki so povezane s pisanjem 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 ravnamo s kodo.

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. Tako tudi `julia`. V tej vaji si bomo ogledali, kako implementirati korensko funkcijo.

> *Opomba!* *Implementacija elementarnih funkcij v julii*
>
> Lokacijo metod, ki računajo določeno funkcijo lahko dobite z ukazoma `methods` in `@match`. 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`.

2.1.1 Podrobna navodila

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

2.2 Računajne kvadratnega korena s Heronovim obrazcem

Z računajnjem kvadratnega korena so se ukvarjali že pred 3500 leti v Babilonu. O tem si lahko več preberete v članku v reviji Presek. Moderna verzija metode računanja približka predstavlja rekurzivno zaporedje, ki konvergira k vrednosti kvadratnega korena danega števila x. Zaporedje približkov lahko izračunamo, tako da uporabimo rekurzivno formulo

$$a_{n+1} = \frac{1}{2} \cdot \left(a_n + \frac{x}{a_n}\right). \tag{2.1}$$

Če izberemo začetni približek, 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, ki računa člene izjemno preprost. Poglejmo si za primer, kako izračunamo $\sqrt{2}$:

```
#| output: true
let
    x = 1.5
    for n = 1:5
        x = (x + 2 / x) / 2
```

```
        println(x)
    end
end
```

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 z 64 bitnimi števili s plavajočo vejico.

Napišimo zgornji algoritem še kot funkcijo:

```julia
"""
  y = koren_heron(x, x0, n)

Izračuna približek za koren števila `x` z `n` koraki Heronovega obrazca z začetnim
približkom `x0`.
"""
function koren_heron(x, x0, n)
  y = x0
  for i = 1:n
    y = (y + x / y) / 2
    @info "Približek na koraku $i je $y"
  end
  return y
end
```

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

Preskusimo funkcijo na številu 3.

```julia
  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!
```

Opomba! *Metoda navadne iteracije in tangentna metoda*

Metoda računanja kvadratnega korena s Heronovim obrazcem je poseben primer tangentne metode, ki je poseben primer metode fiksne točke. Obe metodi, si bomo podrobneje ogledali, v poglavju o nelinearnih enačbah.

2.2.1 Izbira začetnega približka

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 željeno natančnost. Da bi lahko funkcijo uporabljal kdor koli, bi morala funkcija sama izbrati začetni približek, kot tudi število korakov.

Kako bi učinkovito izbrali dober začetni približek? Dokazati je mogoče, da rekurzivno zaporedje konvergira ne glede na izbran začetni približek. Tako lahko uporabimo kar samo število x. Malce boljši približek dobimo s Taylorjevem razvojem korenske funkcije okrog števila 1

$$\sqrt{x} = 1 + \frac{1}{2}(x-1) + \ldots \approx \frac{1}{2} + \frac{x}{2}. \tag{2.2}$$

Število korakov lahko izberemo avtomatsko tako, da računamo nove približke, dokler relativna napaka ne pade pod v naprej predpisano mejo (v našem primeru bomo izbrali napako tako, da bomo dobili približno 10 pravilnih decimalnih mest). Program implementiramo kot novo metodo za funkcijo `koren`

```
"""
    y, st_iteracij = koren_babilonski(x, x0)

Izračunaj vrednost kvadratnega korena danega števila `x` z babilonskim obrazcem z
začetnim približkom `x0`. Funkcija vrne
vrednost približka za kvadratni koren in število iteracij (kolikokrat zaporedoma smo
uporabili babilonski obrazec, da smo dobili zahtevano natančnost).
"""
function koren_babilonski(x, x0)
    a = x0
    it = 0
    while abs(a^2 - x) > abs(x) * 0.5e-11
        a = (a + x / a) / 2
        it += 1
    end
    return a, it
end

y, it = koren_babilonski(10, 0.5 + 10 / 2)
println("Za izračun korena števila 10, potrebujemo $it korakov.")
y, it = koren_babilonski(1000, 0.5 + 1000 / 2)
println("Za izračun korena števila 1000, potrebujemo $it korakov.")
```

Opazimo, da za večje število, potrebujemo več korakov. Poglejmo si, kako se število korakov spreminja, v odvisnosti od števila `x`.

```
#| fig-cap: Število korakov v odvisnosti od argumenta
using Plots
plot(x -> koren_babilonski(x, 0.5 + x / 2)[2], 0.0001, 10000, xaxis=:log10,
minorticks=true, formatter=identity, label="število korakov")
```

Začetni približek $\frac{1}{2} + \frac{x}{2}$ dobro deluje za števila blizu 1, če isto formulo za začetni približek preskusimo za večja števila, dobimo večjo relativno napako. Oziroma potrebujemo več korakov zanke, da pridemo do enake natančnosti. Razlog je v tem, da je $\frac{1}{2} + \frac{x}{2}$ dober približek za majhna števila, če pa se od števila 1 oddaljimo, je približek slabši, bolj kot smo oddaljeni od 1:

```
#| fig-cap: Začetni približek v primerjavi z dejansko vrednostjo korena.
using Plots
plot(x -> 0.5 + x / 2, 0, 10, label="začetni približek")
plot!(x -> sqrt(x), 0, 10, label="korenska funkcija")
```

Da bi dobili boljši približek, si pomagamo s tem, kako so števila predstavljena v računalniku. Realna števila predstavimo s števili s [plavajočo vejico](https://sl.wikipedia.org/wiki/Plavajo%C4%8Da_vejica). Število je zapisano v obliki

$$x = m2^e \tag{2.3}$$

kjer je $0.5 \leq m < 1$ 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](https://en.wikipedia.org/wiki/IEEE_754)).

Koren števila x lahko potem izračunamo kot

$$\sqrt{x} = \sqrt{m} 2^{\frac{e}{2}} \tag{2.4}$$

dober začetni približek dobimo tako, da $\sqrt{m}$ aproksimiramo razvojem v Taylorjevo vrsto okrog točke 1

$$\sqrt{m} \approx 1 + \frac{1}{2}(m - 1) = \frac{1}{2} + \frac{m}{2} \tag{2.5}$$

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

$$\sqrt{2^e} \approx 2^d. \tag{2.6}$$

Celi del števila pri deljenju z 2 lahko dobimo z binarnim premikom v desno (right shift). Potenco števila 2^n, pa z binarnim premikom števila 1 v levo za n mest. Tako lahko zapišemo naslednjo funckijo za začetni približek:

```
"""
    zacetni_priblizek(x)

Izračunaj začetni približek za tangentno metodo za računanje kvadratnega korena števila
`x`.
"""
function zacetni_priblizek(x)
    d = exponent(x) >> 1 # desni premik oziroma deljenje z 2
    m = significand(x)
    if d < 0
        return (0.5 + 0.5 * m) / (1 << -d)
    end
    return (0.5 + 0.5 * m) * (1 << d)
end
```

Primrjajmo izboljšano verzijo začetnega približka s pravo korensko funkcijo:

```
#| fig-cap: Izboljšan začetni približek.
using Plots
plot(zacetni_priblizek, 0, 1000, label="začetni približek")
plot!(sqrt, 0, 1000, label="kvadratni koren")
```

Oglejmo si sedaj število korakov, če uporabimo izboljšani začetni približek.

```
#| fig-cap: Število korakov v odvisnosti od argumenta za izboljšan začetni približek.

using Plots
plot(x -> koren_babilonski(x, zacetni_priblizek(x))[2], 0.0001, 10000, xaxis=:log10,
minorticks=true, formatter=identity, label="število korakov")
```

Opazimo, da se število korakov ne spreminja več z naraščanjem argumenta, to pomeni, da bo časovna zahtevnost tako implemetirane korenske funkcije konstantna in neodvisna od izbire argumenta.

```
#| fig-cap: Relativna napaka na [0.5, 2].
using Plots
rel_napaka(x) = (koren_babilonski(x, 0.5 + x / 2, 4)^2 - x) / x
plot(rel_napaka, 0.5, 2)
```

Sedaj lahko sestavimo funkcijo za računanje korena, ki potrebuje le število in ima konstantno časovno zahtevnost

```
"""
    y = koren(x)
```

```
Izračunaj kvadratni koren danega števila `x` z babilonskim obrazcem.
"""
function koren(x)
    y = zacetni_priblizek(x)
    for i = 1:4
        y = (y + x / y) / 2
    end
    return y
end
```

Preverimo, da je relativna napaka neodvisna od izbranega števila, prav tako pa za izračun potrebujemo enako število operacij.

```
#| fig-cap: Relativna napaka korenske funkcije.
plot(x -> (koren(x)^2 - x) / x, 0.001, 1000.0, xaxis=:log, minorticks=true,
formatter=identity, label="relativna napaka")
```

2.3 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 operaciji normiranja je potrebno komponente vektorja deliti s korenom vsote kvadratov komponent. Kot smo spoznali pri računanju kvadratnega korena z babilonskim obrazcem, je posebej problematično poiskati ustrezen začetni približek, ki je dovolj blizu pravi rešitvi. Tega problema so se zavedali tudi inžinirji igre Quake, ki so razvili posebej zvit, skoraj magičen način za dober začetni približek. Metoda uporabi posebno vrednost `0x5f3759df`, da pride do začetnega približka, nato pa še en korak [tangentne metode](ttps: Več o [računanju obratne vrednosti kvadratnega korena](https://en.wikipedia.org/wiki/Fast_inverse_square_root).

3 Minimalne ploskve

3.1 Naloga

Žično zanko s pravokotnim tlorisom potopimo v milnico, tako da se nanjo napne milna opna.

Radi bi poiskali obliko milne opne, razpete na žični zanki. Malo brskanja po fizikalnih knjigah in internetu hitro razkrije, da ploskve, ki tako nastanejo, sodijo med minimalne ploskve, ki so burile domišljijo mnogim matematikom in nematematikom. Minimalne ploskve so navdihovale tudi umetnike npr. znanega arhitekta Otto Frei, ki je sodeloval pri zasnovi Muenchenskega olimpijskega stadiona, kjer ima streha obliko minimalne ploskve.

Slika 2: Slika olimpijskega stadiona v Münchnu.

3.2 Matematično ozadje

Ploskev lahko predstavimo s funkcijo dveh spremenljivk $u(x, y)$, ki predstavlja višino ploskve nad točko (x, y). Naša naloga bo poiskati funkcijo $u(x, y)$ na tlorisu žične mreže.

Funkcija $u(x, y)$, ki opisuje milno opno, zadošča matematična enačbi, znani pod imenom Poissonova enačba

$$\Delta u(x, y) = \rho(x, y) \tag{3.1}$$

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, v primeru opne, napete na žični zanki. V primeru minimalnih ploskev pa tlačno razliko kar zanemarimo in dobimo Laplaceovo enačbo:

$$\Delta u(x, y) = 0. \tag{3.2}$$

Če predpostavimo, da je oblika na robu območja določena z obliko zanke, rešujemo robni problem za Laplaceovo enačbo. Predpostavimo, da je območje pravokotnik $[a, b] \times [c, d]$. Poleg Laplacove enačbe, veljajo za vrednosti funkcije $u(x, y)$ tudi *robni pogoji*:

$$\begin{aligned} u(x, c) &= f_s(x) \\ u(x, d) &= f_z(x) \\ u(a, y) &= f_l(y) \\ u(b, y) &= f_d(y) \end{aligned} \tag{3.3}$$

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.

Za numerično rešitev Laplaceove enačbe za minimalno ploskev dobimo navdih pri arhitektu Frei Otto, ki je minimalne ploskve raziskoval tudi z elastičnimi tkaninami.

3.3 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

$$\begin{aligned} a = x_0, x_1, \dots x_{n+1} = b \\ c = y_0, \ y_1, \dots y_{m+1} = d \end{aligned} \tag{3.4}$$

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

$$u_{i,j} = u(x_i, y_j), \quad i = 1, \dots n, \quad j = 1, \dots m \tag{3.5}$$

Iščemo torej enačbe, ki jim zadoščajo elementi matrike $u_{i,j}$. Laplaceovo enačbo lahko diskretiziramo z končnimi diferencami, lahko pa izpeljemo enačbe, če si ploskev predstavljamo kot elastično tkanino, ki je fina kvadratna mreža iz elastičnih nitk. Vsako vozlišče v mreži je povezano s 4 sosednjimi vozlišči. Vozlišče bo v ravnovesju, ko bo vsota vseh sil nanj enaka 0. Predpostavimo, da so vozlišča povezana z idealnimi vzmetmi in je sila sorazmerna z razliko. Če zapišemo enačbo za komponente sile v smeri z, dobimo za točko (x_i, y_j, u_{ij}) enačbo

$$u_{i-1,j} + u_{i,j-1} - 4u_{i,j} + u_{i+1,j} + u_{i,j+1} = 0. \tag{3.6}$$

Za u_{ij} imamo tako sistem linearnih enačb. Ker pa so vrednotsi 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.

3.4 Matrika sistema linearnih enačb

Sisteme linearnih enačb običajno zapišemo v matrični obliki

$$Ax = \mathbf{b}, \tag{3.7}$$

kjer je A kvadratna matrika, x in $\mathbf{b}$ pa vektorja. Spremenljivke $u_{i,j}$ razvrstimo po stolpcih v vektor.

!!! note „Razvrstitev po stolpih“

```
Eden od načinov, kako lahko elemente matrike razvrstimo v vektor, je, da
stolpce matrike enega za drugim postavimo v vektor. Indeks v vektorju $k$ lahko
izrazimo z indeksi $i,j$ v matriki s formulo
$$k = i+(n-1)j.$$
```

Za $n = m = 3$ dobimo 9×9 matriko

```
L = \begin{bmatrix} -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{bmatrix},
```

ki je sestavljena iz 3×3 blokov

```
\begin{bmatrix}-4&1&0\cr 1&-4&1\cr 0&1&-4\end{bmatrix},\quad
\begin{bmatrix}1&0&0\cr 0&1&0\cr 0&0&1\end{bmatrix}.
```

desne strani pa so

```
\mathbf{b} = -[u_{01}+u_{10}, u_{20}, \ldots u_{n0}+u_{n+1,1},u_{02},
0,\ldots u_{n+1,2}, u_{03}, 0\ldots u_{n, m+1},u_{n,m+1}+u_{n+1,m}]^T.
```

3.5 Izpeljava s Kronekerjevim produktom

Množenje vektorja $x = \mathrm{vec}(Z)$ z matriko L lahko prestavimo kot množenje z matriko

$$\mathrm{vec}(LZ + ZL) = L\ \mathrm{vec}(Z). \tag{3.8}$$

Ker velja $\mathrm{vec}(AXB) = A \otimes B \cdot \mathrm{vec}(X)$ je

$$L^{N,N} = L^{m,m} \otimes I^{n,n} + I^{m,m} \otimes L^{n,n} \tag{3.9}$$

3.6 Primer

```
robni_problem = RobniProblemPravokotnik( LaplaceovOperator{2}, ((0,
pi), (0, pi)), [sin, y->0, sin, y->0] ) Z, x, y = resi(robni_problem) surface(x,
y, Z) savefig("milnica.png")
```

3.7 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. Vendar se pri Gaussovi eliminaciji delež ničelnih elementov matrike pogosto zmanjša. Poglejmo kako se odreže matrika za Laplaceov operator.

```
using Plots L = matrika(100,100, LaplaceovOperator(2)) spy(sparse(L),
seriescolor = :blues)
```

Če izvedemo eliminacijo, se matrika deloma napolni z neničelnimi elementi:

```
import LinearAlgebra.lu LU = lu(L) spy!(sparse(LU.L), seriescolor =
:blues) spy!(sparse(LU.U), seriescolor = :blues)
```

3.8 Koda

```
@index Pages = ["03_minimalne_ploskve.md"]
```

```
@autodocs Modules = [NumMat] Pages = ["Laplace2D.jl"]
```

3.9 Iteracijske metode

```
@meta
CurrentModule = NumMat
DocTestSetup  = quote
    using NumMat
end
```

V nalogi o minimalnih ploskvah smo reševali linearen sistem enačb

```
u_{i,j-1}+u_{i-1,j}-4u_{ij}+u_{i+1,j}+u_{i,j+1}=0
```

za elemente matrike $U = [u_{ij}]$, ki predstavlja višinske vrednosti na minimalni ploskvi v vozliščih kvadratne mreže. Največ težav smo imeli z zapisom matrike sistema in desnih strani. Poleg tega je matrika sistema L razpršena (ima veliko ničel), ko izvedemo LU razcep ali Gaussovo eliminacijo, veliko teh ničelnih elementov postane neničelni in matrika se napolni. Pri razpršenih matrikah tako pogosto uporabimo iterativne metode za reševanje sistemov enačb, pri katerih matrika ostane razpršena in tako lahko prihranimo veliko na prostorski in časovni zahtevnosti.

‼ note „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 $u_{ij}^{(k)}$. Limita rekurzivnega zaporedja je ena od fiksnih točk rekurzivne enačbo, če zaporedje konvergira. Ker smo rekurzivno enačbo izpeljali iz originalnih enačb, je njena fiksna točka ravno rešitev originalnega sistema.
```

V primeru enačb za laplaceovo enačbo(minimalne ploskve), tako dobimo rekurzivne enačbe

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

ki ustrezajo jacobijevi iteraciji

‼ tip „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](https://sl.wikipedia.org/wiki/Diagonalno_dominantna_matrika).
```

Konvergenco jacobijeve iteracije lahko izboljšamo, če namesto vrednosti na prejšnjem približku, uporabimo nove vrednosti, ki so bile že izračunani. Če računamo element u_{ij} po leksikografskem vrstnem redu, bodo elementi $u_{il}^{(k+1)}$ za $l < j$ in $u_{lj}^{(k+1)}$ za $l < i$ že na novo izračunani, ko računamo $u_{ij}^{(k+1)}$. Če jih upobimo v iteracijski formuli, dobimo gauss-seidlovo iteracijo

$$u_{i,j}^{(k+1)} = \frac{1}{4}\left(u_{i,j-1}^{(k+1)} + u_{i-1,j}^{(k)} + u_{i+1,j}^{(k)} + u_{i,j+1}^{(k)}\right) \tag{3.10}$$

Konvergenco še izboljšamo, če približek $u_{ij}^{(k+1)}$, ki ga dobimo z gauss-seidlovo metodo, malce zmešamo s približkom na prejšnjem koraku $u_{ij}^{(k)}$

$$
\begin{aligned}
u_{i,j}^{(\mathrm{GS})} &= \frac{1}{4}\Big(u_{i,j-1}^{(k+1)} + u_{i-1,j}^{(k)} + u_{i+1,j}^{(k)} + u_{i,j+1}^{(k)}\Big) \\
u_{i,j}^{(k+1)} &= \omega u_{i,j}^{(\mathrm{GS})} + (1-\omega) u_{i,j}^{(k)}
\end{aligned}
\tag{3.11}
$$

in dobimo metodo SOR. Parameter ω je lahko poljubno število $(0, 2]$ Pri $\omega = 1$ dobimo gauss-seidlovo iteracijo.

3.9.1 Primer

```
using Plots
U0 = zeros(20, 20)
x = LinRange(0, pi, 20)
U0[1,:] = sin.(x)
U0[end,:] = sin.(x)
surface(x, x, U0, title="Začetni približek za iteracijo")
savefig("zacetni_priblizek.png")
```

```
L = LaplaceovOperator(2)
U = copy(U0)
animation = Animation()
for i=1:200
    U = korak_sor(L, U)
    surface(x, x, U, title="Konvergenca Gauss-Seidlove iteracije")
    frame(animation)
end
mp4(animation, "konvergenca.mp4", fps = 10)
```

```
@raw html <video width="600" height="400" controls> <source src="../konvergenca.mp4"
type="video/mp4"> <source src="konvergenca.mp4" type="video/mp4"> </video>
```

Konvergenca Gauss-Seidlove iteracije

3.9.2 Konvergenca

Grafično predstavi konvergenco v odvisnoti od izbire ω.

```
using Plots
n = 50
U = zeros(n,n)
U[:,1] = sin.(LinRange(0, pi, n))
U[:, end] = U[:, 1]
L = LaplaceovOperator(2)
omega = LinRange(0.1, 1.95, 40)
it = [iteracija(x->korak_sor(L, x, om), U; tol=1e-3)[2] for om in omega]
plot(omega, it, title = "Konvergenca SOR v odvisnosti od omega")
savefig("sor_konvergenca.svg")
```

3.9.3 Metoda konjugiranih gradientov

Ker je laplaceova matrika diagonalno dominantna z -4 na diagonali je negativno definitna. Zato lahko uporabimo metodo konjugiranih gradientov. Algoritem konjugiranih gradientov potrebuje le množenje z laplaceovo matriko, ne pa tudi samih elementov. Zato lahko izkoristimo možnosti, ki jih ponuja programski jezik `julia`, da lahko za isto funkcijo napišemo različne metode za različne tipe argumentov.

Preprosto napišemo novo metodo za množenje `*`, ki sprejme argumente tipa `LaplaceovOperator{2}` in `Matrix`. Metoda konjugiranih gradientov še hitreje konvergira kot SOR.

```
@example
using NumMat
n = 50
U = zeros(n,n)
U[:,1] = sin.(LinRange(0, pi, n))
U[:, end] = U[:, 1]
L = LaplaceovOperator{2}()
b = desne_strani(L, U)
Z, it = conjgrad(L, b, zeros(n, n))
println("Število korakov: $it")
```