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. Za podrobnejši opis metod in izpeljave si lahko ogledate v učbeniku prof. Orla *Osnove numerične matematike* [1].

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

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

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

```
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`.

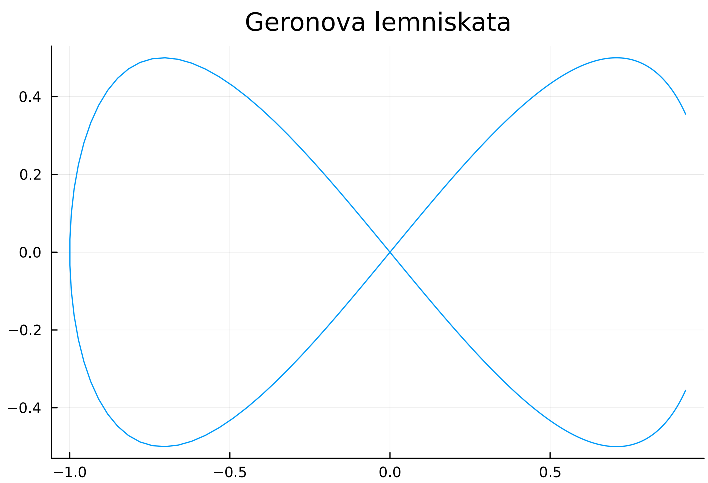
```
using Vaja00
#' Krivuljo narišemo tako, da koordinati tabeliramo za veliko število parametrov.
t = range(-5, 5, 300) # generiramo zaporedje 300 vrednosti na [-5, 5]
x = lemniskata_x.(t) # funkcijo apliciramo na elemente zaporedja
y = lemniskata_y.(t) # tako da imenu funkcije dodamo .
#' Za risanje 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. 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 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, da funkcija dosega zahtevano natančnost za poljubne vrednosti argumenta `x`.

2.2 Rešitev naloge

Najprej ustvarimo projekt za trenutno vajo in ga dodamo v delovno okolje.

```
(nummat-julia) pkg> generate Vaja02Koren
(nummat-julia) pkg> develop Vaje02Koren//
```

2.2.1 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* Enačba (2.1). Za numerično reševanje nelinearnih enačb obstaja cela vrsta metod. Ena najbolj popularnih 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 Enačba (2.3) konvergira, potem konvergira k rešitvi enačbe $f(x) = 0$.

Enačbo Enačba (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 $f'(y) = 2y$, 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 Enačba (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, ki računa člene izjemno preprost. Poglejmo si za primer, kako izračunamo $\sqrt{2}$:

```
julia>  let
          x = 1.5
          for n = 1:5
            x = (x + 2 / x) / 2
            println(x)
          end
        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 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 kvadratni 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.

2.2.2 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, kot tudi število korakov. Najprej bomo poskrbeli, da je število korakov ravno dovolj veliko, da dosežemo želeno natančnost.

Opomba! *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, če je absolutna napaka manjša od **absolutne tolerance** ali pa če 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 Enačba (2.6) nikoli ni izpolnjena. **Število** 0 **nikoli ni približno enako nobenemu neničelnemu številu, če ju primerjamo z relativno toleranco.**

Opomba! *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:

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

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 kore 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 si 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 je

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

Opomba! 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.

Če uporabimo pogoj Enačba (2.11), lahko napišemo funkcijo, ki sama določi število korakov iteracije.

```julia
"""
  y = koren(x, y0)

Izračunaj vrednost kvadratnega korena danega števila `x` s Heronovim
obrazcem z začetnim približkom `y0`.
"""
function koren(x, y0)
  if x == 0.0
    # Vrednost 0 obravnavamo posebej, saj relativna primerjava z 0
    # problematična
    return 0.0
  end
  delta = 5e-11
  for i = 1:10
    y = (y0 + x / y0) / 2
    if abs(x - y^2) <= 2 * delta * abs(x)
      @info "Število korakov $i"
      return y
    end
    y0 = y
  end
  throw("Iteracija ne konvergira")
end
```

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

2.2.3 Izbira začetnega približka

Kako bi učinkovito izbrali dober začetni približek? Dokazati je mogoče, da rekurzivno zaporedje Enačba (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

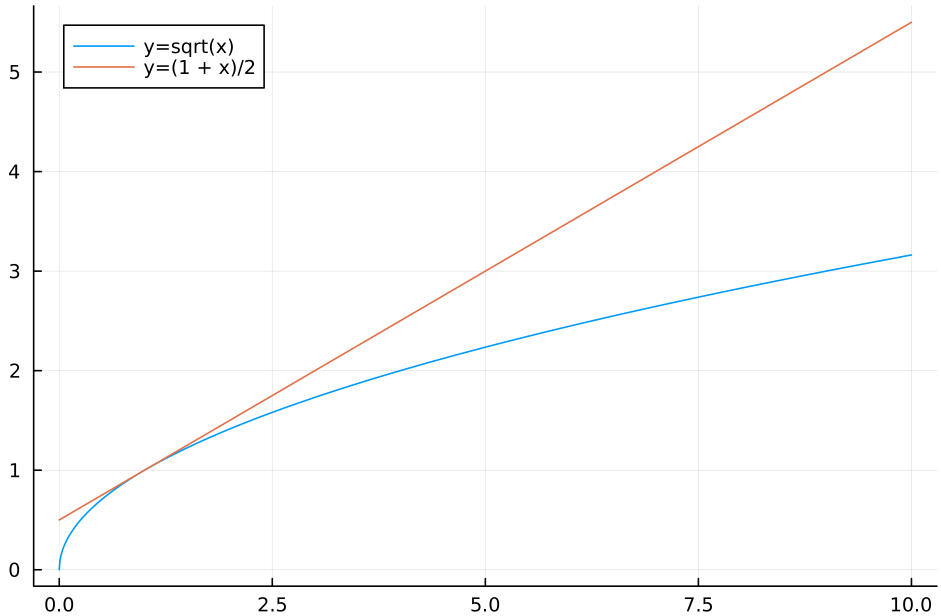
$$\sqrt{x} = 1 + \frac{1}{2}(x-1) + \ldots \approx \frac{1}{2} + \frac{x}{2}. \tag{2.12}$$

Vendar opazimo, da za večja števila, potrebuje iteracija 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 preskusimo za večja števila, dobimo večjo relativno napako. Oziroma potrebujemo več korakov zanke, da pridemo do enake natančnosti.

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



Slika 2: Korenska funkcija in tangenta v $x = 1$.

Da bi dobili boljši približek, si pomagamo s tem, kako so števila predstavljena 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 lahko potem izračunamo kot

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

Koren mantise lahko 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 $e = 2d + o$, lahko $\sqrt{2^e}$ 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:

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

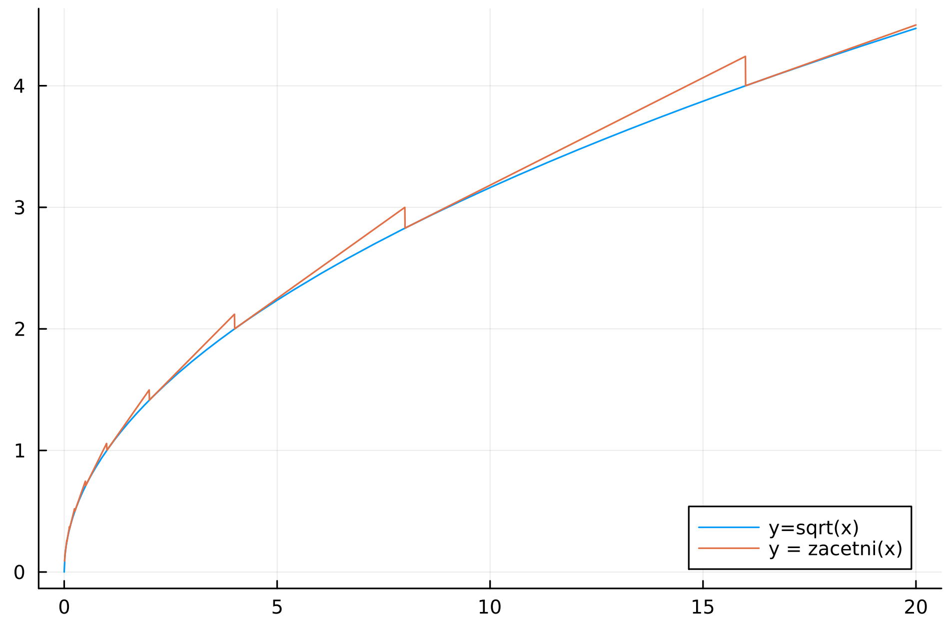
Potenco števila 2^n lahko izračunamo z binarnim premikom števila 1 v levo za n mest. Tako lahko zapišemo naslednjo funkcijo za začetni približek:

```julia
"""
  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)
  s2 = (ost == 0) ? 1 : 1.4142135623730951

  if x > 1
    return (1 << d) * (0.5 + m / 2) * s2
  else
    return (0.5 + m / 2) / (1 << d) / s2
  end
end
```

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

Primerjajmo izboljšano verzijo začetnega približka s pravo korensko funkcijo.

```julia
plot(sqrt, 0, 20, label="y=sqrt(x)")
plot!(Vaja02Koren.zacetni, 0, 20, label="y = zacetni(x)")
```



Slika 3: Korenska funkcija in začetni približek.

2.2.4 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 9: Funkcija `koren(x)`.

> *Opomba! 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 7, ko pokličemo `koren(2.0)` se izvede Program 9.
>
> Metode, ki so definirane za neko funkcijo `fun` lahko vidimo z ukazom `methods(fun)`. Metodo, ki se uporabi za določen klic funkcije lahko poiščemo z makrojem `@which`, npr. `@which koren(2.0, 1.0)`.

Opazimo, da se število korakov ne spreminja več z naraščanjem argumenta, to 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)
```

Opomba! *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 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 dober začetni približek. Metoda uporabi posebno vrednost `0x5f3759df`, da pride do začetnega približka, nato pa še en korak Newtonove metode. Več o računanju obratne vrednosti kvadratnega korena.

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 4: 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

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

kjer je A kvadratna matrika, $\boldsymbol{x}$ in $\boldsymbol{b}$ 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 $\boldsymbol{x} = \mathrm{vec}(Z)$ 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 prurediomo 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")
```

4 Domače naloge

4.1 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.

4.1.1 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 vso kodo (glej Coverage.jl)
- *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 :-)

4.1.2 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.

4.1.2.1 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 vrstici*
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`.

4.1.2.2 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

$$u = \left[s(k) + A_{k,k}, A_{k+1,k}, \ldots A_{n,k}\right], \tag{4.1}$$

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

$$2u\frac{u^T A(k:n,k+1:n)}{u^T u}. \tag{4.2}$$

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 $s(k) = \operatorname{sign}(A_{k,k}) * \| [A_{k,k}, ..., A_{n,k}] \|$.

Nato matriko popravimo

$$A(k:n, k+1:n) = A(k:n, k+1:n) - 2 * u * \frac{u^T * A(k:n, k+1:n)}{u^T * u}. \qquad (4.3)$$

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 tri-diagonalne matrike definiramo posebno podatkovno strukturo `Tridiag`, ki hrani le neničelne elemente matrike. Julia omogoča, da LU razcep tri-diagonalne matrike, implementiramo kot specializirano metodo funkcije `lu` iz paketa `LinearAlgebra`. Pri tem upoštevamo posebnosti tri-diagonalne 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 tri-diagonalne 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. …

4.1.3 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.

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

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

Za poročanje o pokritosti kode lahko uporabite paket Coverage.jl.

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

4.2 1. domača naloga

Izberite eno izmed spodnjih nalog.

Naloge

4.2.1 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

$$V(i,j) = A(i, I(i,j)). \tag{4.4}$$

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

$$|A\boldsymbol{x}^{(k)} - \boldsymbol{b}|_\infty < \delta, \tag{4.5}$$

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

$$
\begin{aligned}
-\mathrm{st}(i)x_i + \sum_{j\in N(i)} x_j = 0,\\
-\mathrm{st}(i)y_i + \sum_{j\in N(i)} y_j = 0,\\
-\mathrm{st}(i)z_i + \sum_{j\in N(i)} z_j = 0,
\end{aligned}
\tag{4.6}
$$

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.

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

4.2.2 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

$$Ax = b, \tag{4.7}$$

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.

4.2.3 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

$$Ax = b \tag{4.8}$$

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.

4.2.4 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

$$[\cos(\alpha); \sin(\alpha)], \tag{4.9}$$

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.

4.2.5 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 dvodiagonalna matrika tipa `ZgornjeDvodiagonalna`, v `Q` pa naj bo matrika tipa `Givens`. vmesnikih.

Podatkovna tipa `ZgornjeDvodiagonalna` in `Givens` definirajte sami (glejte tudi nalogo Poglavje 4.2.4. Poleg tega implementirajte množenje `Base.*` matrik tipa `Givens` in `ZgornjeDvodiagonalna`.

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.

4.2.6 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

$$x^{(n+1)} = \frac{A_\lambda^{-1} x^{(n)}}{|A_\lambda^{-1} x^{(n)}|}, \tag{4.10}$$

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:

$$H = Q^T A Q. \tag{4.11}$$

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

$$x^n + a_{\{n-1\}}x^{\{n-2\}} + \ldots a_1 x + a_0 \tag{4.12}$$

lahko priredimo matriko

$$\begin{pmatrix} 0 & 0 & \ldots & 0 & -a_0 \\ 1 & 0 & \ldots & 0 & -a_1 \\ 0 & 1 & \ldots & 0 & -a_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \ldots & 1 & -a_{n-1} \end{pmatrix} \tag{4.13}$$

katere lastne vrednosti se ujemajo z ničlami polinoma.

4.2.7 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

$$x^{\{(n+1)\}} = \frac{A_\lambda^{-1} x^{(n)}}{|A_\lambda^{-1} x^{(n)}|}, \tag{4.14}$$

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:

$$T = Q^T A Q. \tag{4.15}$$

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

$$y''(x) = \lambda^2 y(x), \tag{4.16}$$

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

4.2.8 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

$$S(x) = S_{i(x)} = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3, \quad x \in [x_i, x_{i+1}], \tag{4.17}$$

nato pa izpolnimo zahtevane pogoje[1].

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`).

4.2.9 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:

[1] 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

  - 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).

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

4.3 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

4.3.1 Naloge s funkcijami

Naloge

4.3.1.1 Porazdelitvena funkcija normalne slučajne spremenljivke

Napišite učinkovito funkcijo, ki izračuna vrednosti porazdelitvene funkcije za standardno normalno porazdeljeno slučajno spremenljivko $X \sim N(0,1)$.

$$\Phi(x) = P(X < . = x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-\frac{t^2}{2}} dt \tag{4.18}$$

4.3.1.2 Fresnelov integral (težja)

Napišite učinkovito funkcijo, ki izračuna vrednosti Fresnelovega kosinusa

$$C(x) = \sqrt{2/\pi} \int_0^x \cos(t^2) dt. \tag{4.19}$$

Namig: Uporabite pomožni funkciji

$$f(x) = \sqrt{2/\pi} \int_0^\infty e^{-2xt} \cos(t^2) dt \tag{4.20}$$

$$g(x) = \sqrt{2/\pi} \int_0^\infty e^{-2xt} \sin(t^2) dt \tag{4.21}$$

kot je opisano v priročniku Abramowitz in Stegun.

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

Napišite učinkovito funkcijo, ki izračuna funkcijo kvantilov za normalno porazdeljeno slučajno spremenljivko. Funkcija kvantilov je inverzna funkcija porazdelitvene funkcije.

4.3.1.4 Integralski sinus (težja)

Napišite učinkovito funkcijo, ki izračuna integralski sinus

$$Si(x) = \int_0^x \frac{\sin(t)}{t} dt. \tag{4.22}$$

Uporabite pomožne funkcije, kot je opisano v priročniku Abramowitz in Stegun.

4.3.1.5 Besselova funkcija (težja)

Napišite učinkovito funkcijo, ki izračuna Besselovo funkcijo J_0:

$$J_0(x) = \frac{1}{\pi} \int_0^\pi \cos(x \sin t) dt. \tag{4.23}$$

4.3.2 Naloge s števili

Naloge

4.3.2.1 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

$$\boldsymbol{F} = \int_{T_1} \int_{T_2} \frac{\boldsymbol{r}_1 - \boldsymbol{r_2}}{\| \boldsymbol{r}_1 - \boldsymbol{r_2} \|^2} d\boldsymbol{r}_1 d\boldsymbol{r}_2. \tag{4.24}$$

4.3.2.2 Ploščina hipotrohoide

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

$$x(t) = (a+b)\cos(t) + b\cos\left(\frac{a+b}{b}t\right) \tag{4.25}$$

$$y(t) = (a+b)\sin(t) + b\sin\left(\frac{a+b}{b}t\right) \tag{4.26}$$

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

Namig: Uporabite formulo za ploščino krivočrtnega trikotnika pod krivuljo:

$$P = \frac{1}{2}\int_{t_1}^{t_2} (x(t)\dot{y}(t) - \dot{x}(t)y(t))dt \tag{4.27}$$

4.3.2.3 Povprečna razdalja (težja)

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

$$T = ([-1,1])^3 - ([0,1])^3. \tag{4.28}$$

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

$$\begin{aligned} \int_{A-B}\int_{A-B} f(x,y)dxdy = & \int_A\int_A f(x,y)dxdy \\ & -2\int_A\int_B f(x,y)dxdy + \int_B\int_B f(x,y)dxdy \end{aligned} \tag{4.29}$$

4.3.2.4 Ploščina Bézierove krivulje

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

$$(0,0),(1,1),(2,3),(1,4),(0,4),(-1,3),(0,1),(1,0). \tag{4.30}$$

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

$$P = \frac{1}{2}\int_{t_1}^{t_2} (x(t)\dot{y}(t) - \dot{x}(t)y(t))dt. \tag{4.31}$$

4.3.3 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 obeh nalog:

4.3.3.1 Ineterpolacija 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

$$l(x) = \begin{cases} \frac{\sum \frac{f(x_j)\lambda_j}{x-x_j}}{\sum \frac{\lambda_j}{x-x_j}} & x \neq x_j \\ f(x_j) & \text{sicer} \end{cases} \tag{4.32}$$

kjer so vrednosti uteži λ_j izbrane, tako da je $\prod_{i\neq j}(x_j - x_i) = 1$. Čebiševe točke so podane na intervalu $[-1,1]$ s formulo

$$x_k = \cos\left(\frac{2k-1}{2n}\pi\right), \quad k = 0, 1 \dots n-1, \tag{4.33}$$

vrednosti uteži λ_k pa so enake

$$\lambda_k = (-1)^k \begin{cases} 1 & 0 < i < n \\ \frac{1}{2} & i = 0 \\ n & \text{sicer.} \end{cases} \tag{4.34}$$

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 $|x^2 - 2x|$ na $[1,3]$. Za vsako funkcijo določi stopnjo polinoma, da napaka ne bo presegla 10^{-6}.

4.3.3.2 Gauss-Legendrove kvadrature

Izpelji Gauss-Legendreovo integracijsko pravilo na dveh točkah

$$\int_0^h f(x)dx = Af(x_1) + Bf(x_2) + R_f \tag{4.35}$$

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

$$\int_0^5 \frac{\sin x}{x} dx \tag{4.36}$$

na 10 decimalk natančno.

4.4 3. domača naloga

4.4.1 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.

4.4.2 Težje naloge

4.4.2.1 Ničle Airijeve funkcije

Airyjeva funkcija je dana kot rešitev začetnega problema

$$Ai''(x) - x\,Ai(x) = 0, \quad Ai(0) = \frac{1}{3^{\frac{2}{3}}\Gamma\left(\frac{2}{3}\right)}\, Ai'(0) = -\frac{1}{3^{\frac{1}{3}}\Gamma\left(\frac{1}{3}\right)}. \tag{4.37}$$

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.

4.4.2.1.1 Namig

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

$$y'(x) = A(x)y, \tag{4.38}$$

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

$$\begin{aligned}
A_1 &= A\left(x_k + \left(\frac{1}{2} - \frac{\sqrt{3}}{6}\right)h\right) \\
A_2 &= A\left(x_k + \left(\frac{1}{2} + \frac{\sqrt{3}}{6}\right)h\right) \\
\sigma_{k+1} &= \frac{h}{2}(A_1 + A_2) - \frac{\sqrt{3}}{12}h^2[A_1, A_2] \\
\boldsymbol{Y}_{k+1} &= \exp(\sigma_{k+1})\boldsymbol{Y}_k.
\end{aligned} \tag{4.39}$$

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`.

4.4.2.2 Dolžina implicinto podane krivulje

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

$$\begin{aligned}
F_1(x, y, z) &= x^4 + y^2/2 + z^2 = 12 \\
F_2(x, y, z) &= x^2 + y^2 - 4z^2 = 8.
\end{aligned} \tag{4.40}$$

Krivuljo lahko poiščete kot rešitev diferencialne enačbe

$$\dot{\boldsymbol{x}}(t) = \nabla F_1 \times \nabla F_2. \tag{4.41}$$

4.4.2.3 Perioda limitnega cikla

Poiščite periodo limitnega cikla za diferencialno enačbo

$$x''(t) - 4(1 - x^2)x'(t) + x = 0 \tag{4.42}$$

na 10 decimalk natančno.

4.4.2.4 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.

4.4.2.4.1 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

$$\mu = \frac{m}{M + m} \quad \text{ter} \quad \bar{\mu} = 1 - \mu = \frac{M}{M + m}\ . \tag{4.43}$$

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.

$$
\begin{aligned}
R = R(x, y, z) &= \sqrt{(x + \mu)^2 + y^2 + z^2}, \\
r = r(x, y, z) &= \sqrt{\left(x - \bar{\mu}\right)^2 + y^2 + z^2}.
\end{aligned}
\tag{4.44}
$$

Enačbe gibanja sonde so potem:

$$
\begin{aligned}
\ddot{x} &= x + 2\dot{y} - \frac{\bar{\mu}}{R^3}(x + \mu) - \frac{\mu}{r^3}\left(x - \bar{\mu}\right), \\
\ddot{y} &= y - 2\dot{x} - \frac{\bar{\mu}}{R^3} y - \frac{\mu}{r^3} y, \\
\ddot{z} &= -\frac{\bar{\mu}}{R^3} z - \frac{\mu}{r^3} z.
\end{aligned}
\tag{4.45}
$$

4.4.3 Lažja naloga (ocena največ 9)

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

4.4.3.1 Matematično nihalo

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

$$
\frac{g}{l}\sin(\theta(t)) + \theta''(t) = 0, \quad \theta(0) = \theta_0, \theta'(0) = \theta_0', \tag{4.46}
$$

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:

$$
\begin{aligned}
k_1 &= & h\, f(x_n, y_n) \\
k_2 &= & h\, f(x_n + h/2, y_n + k_1/2) \\
k_3 &= & h\, f(x_n + h/2, y_n + k_2/2) \\
k_4 &= & h\, f(x_n + h, y_n + k_3) \\
y_{n+1} &= y_n + (k_1 + 2k_2 + 2k_3 + k_4)/6.
\end{aligned}
\tag{4.47}
$$

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.