Numerična matematika in programski jezik Julia

Martin Vuk

2024

Predgovor

Knjiga je prvenstveno namenjena študentom predmeta Numerična matematika na Fakulteti za računalništvo in informatiko, Univerze v Ljubljani. Vsebuje gradivo za vaje, domače naloge in stare izpitne naloge.

Vaje so zasnovane za delo v računalniški učilnici. Vaje so zasnovane tako, da omogočajo študentu, samostojno reševanje in ne zgolj prepisovanje s table. Vsaka vaja se začne z opisom naloge in jasnimi navodili, kaj na naj bo končni rezultat. Nato sledijo vedno bolj podrobni namigi, kako se naloge lotiti. 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, kako uporabljati in implementirati osnovne algoritme numerične matematike. Primeri programov so napisnani v programskem jeziku Julia, ki je zasnovan za učinkovito izvajanje računsko zahtevnih problemov.

Na tem mestu bi se rad zahvalil Bojanu Orlu, Emilu Žagarju, Petru Kinku, 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 LMMRI posebej Neži Mramor Kosta, Damiru Franetiču in Aljažu Zalarju, 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.

Uvod

Ta knjiga vsebuje gradiva za izvedbo laboratorijskih vaj pri predmetu Numerična matematika na Fakulteti za računalništvo in informatiko, Univerze v Ljubljani.

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 preiskusi 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 preverbe ali implementacijo robnih primerov. Včasih opustimo obravnavo pričakovanih napak. Prednost smo dali berljivosti, pred kompletnostjo, da je bralcu lažje razumeti, kaj koda počne.

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.

Cilj tega poglavja je, da si pripravimo okolje za delo v programskem jeziku *julia* in ustvarimo prvi program. Na koncu te vaje bomo pripravili svoj prvi paket v *julii*, ki bo vseboval preprosto funkcijo. Napisali bomo teste, ki bodo preverili pravilnost funkcije. Nato bomo napisali skripto, ki funkcijo iz našega paketa uporabi in iz skripte generirali lično pročilo v formatu PDF z ilustracijo uporabe funkcije in kakšnim lepim grafom.

1.1 Namestitev in prvi koraki

Ko namestite programski jezik julia, lahko v terminalu poženete ukaz `julia`, ki odpre interaktivno zanko (angl. *Read Eval Print Loop* ali s kratico REPL). V zanko lahko pišemo posamezne ukaze, ki jih nato `julia` prevede in izvede.

```
$ julia
julia> 1 + 1
2
```

1.2 Priprava delovnega okolja

Vnašanje ukazov v interaktivni zanki je lahko uporabno namesto kalkulatorja. Vendar je za resnejše delo bolje kodo shraniti v datoteke. 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 terminal in sledite ukazom spodaj.

```
$ julia
julia > # pritisnemo `]`, da pridemo v način paketov
(@v1.10) pkg> add Revise
(@v1.10) pkg> # pritisnemo vračalko, da pridemo iz načina paketov
julia> startup = """
       try
         using Revise
       catch e
         @warn "Error initializing Revise" exception=(e, catch_backtrace())
       end
"""
julia> write(ENV[HOME]*"/.config/julia/startup.jl", startup)
```

Okolje za delo z *julio* je pripravljeno.

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.

1.2.1 Priprava projektne mape

Pripravimo mapo, v kateri bomo hranili programe. Datoteke bomo organizirali tako, da bo vsaka vaja paket v svoji mapi, korenska mapa pa bo služila kot delovno okolje.

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.

```
julia > # pritisnemo ], da pridemo v način paketov
(@v1.10) pkg> activate . # pripravimo virtualno okolje v korenski mapi
(nummat-julia) pkg> add Plot # paket za risanje grafov
```

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! 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, glej docker, NixOS in GNU Guix.

1.3 Priprava paketa za vajo

Ob začetku vsake vaje si bomo v mapi, ki smo jo ustvarili pred tem (`nummmat-julia`) najprej ustvarili direktorij oziroma paket za *julio*, kjer 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
julia> # pritisnemo ], da pridemo v način pkg
(@v1.10) pkg> generate Vaja00 # 00 bomo kasneje nadomestili z zaporedno številko vaje
(@v1.10) pkg> activate .
(nummat-julia) pkg> develop Vaja00
(nummat-julia) pkg> # pritisnemo tipko za brisanje nazaj, da pridemo v navaden način
julia>
```

Zgornji ukazi ustvarijo direktorij `Vaja00` z osnovno strukturo paketa v Juliji. Za bolj obsežen projekt, ki ga želite objaviti, lahko uporabite PkgTemplates ali PkgSkeleton.

```
julia> cd("Vaja00") # pritisnemo ;, da pridemo v način lupine
shell> tree .
.
├── Project.toml
└── src
    └── Vaja00.jl

1 directory, 2 files
```

Direktoriju dodamo še teste, skripte z demnostracijsko kodo in README dokument.

```
shell> mkdir test
shell> touch test/runtests.jl
shell> touch README.md
shell> mkdir scripts
shell> touch scripts/demo.jl
shell> tree .
.
├── Manifest.toml
├── Project.toml
├── scripts
│   └── demo.jl
├── src
│   └── Vaja00.jl
└── test
    └── runtests.jl
```

Opomba! V okolju Windows lupina ne deluje najbolje

Zgornji ukazi v lupini v okolju Microsoft Windows mogoče ne bodo delovali. V tem primeru lahko mape in datoteke ustvarite v *raziskovalcu*. Lahko pa si okolje za *julio* ustvarite v Linuxu v Windowsih (WSL).

1.3.1 Geronova lemniskata

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

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

definiramo 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 interaktivni zanki lahko sedaj pokličemo novo definirani funkciji.

```
import Pkg; Pkg.activate(".")
using Vaja00
lemniskata_x(1.2)
```

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

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

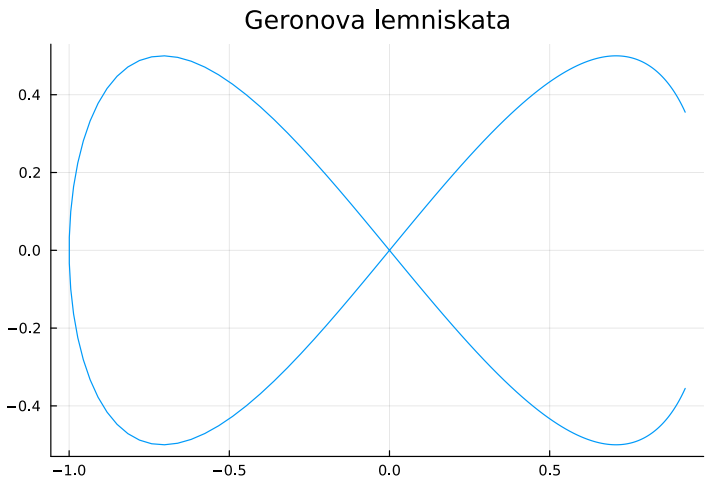
```
#' # Geronova lemniskata
using Vaja00
#' Krivuljo narišemo tako, da koordinati tabeliramo za veliko število parametrov.
t = range(-5, 5, 300) # generiramo zaporedje 300 vrednosti na [-5, 5]
x = lemniskata_x.(t) # funkcijo apliciramo na elemente zaporedja
y = lemniskata_y.(t) # tako da imenu funkcije 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

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

in dobimo sliko lemniskate.



Slika 1: Geronova lemniskata

1.3.2 Testi

V prejšnjem razdelku smo definirali funkcije in napisali skripto, s katero smo omenjene funkcije uporabili. Vstopna točka za teste je `test\runtests.jl`. Paket [Test](https://docs.julialang.org/en/v1/stdlib/Test/) omogoča pisanje enotskih testov, ki se lahko avtomatično izvedejo v sistemu [nenehne integracije (Continuous Integration)](https://en.wikipedia.org/wiki/Continuous_integration).

V juliji teste pišemo z makroji @test in @testset. Če `test/runtests.jl` lahko napišemo

```
using Test, Vaja00

@test Vaja00.funkcija_ki_vrne_ena() == 1
```

Lahko teste poženemo tako, da v `pkg` načinu poženemo ukaz `test`

```
(Vaja00) pkg> test

    Testing Running tests...
    Testing Vaja00 tests passed
```

1.3.3 Dokumentacija

Za pisanje dokumentacijo navadno uporabimo format [Markdown](https://en.wikipedia.org/wiki/Markdown). S paketom [Documenter](https://documenter.juliadocs.org/stable/) lahko komentarje v kodi in markdown dokumentente združimo in generiramo HTML ali PDF dokumentacijo s povezavo na izvorno kodo.

Za pripravo posameznih poročil lahko uporabite [IJulia](https://github.com/JuliaLang/IJulia.jl), [Weave.jl](https://github.com/JunoLab/Weave.jl), [Literate.jl](https://github.com/fredrikekre/Literate.jl) ali [Quadro](https://quarto.org/docs/computations/julia.html).

1.4 Organizacija direktorijev

- `vaje` direktorij z vajami
- `vaje/Vaja00` vsaka vaja ima svoj direktorij
- posamezen direktorij za vajo je organiziran kot paket s kodo, testi in dokumentacijo

```
nummat-julia
  └── Vaja00
    ├── Project.toml
    ├── README.md
    ├── src
    |   └── Vaja00.jl
    ├── test
    |   └── runtests.jl
    ├── doc
    |   ├──  makedocs.jl
    |   └── index.md
    └── scripts
      └── demo.jl
```

1.5 Generiranje PDF dokumentov

Za generiranje PDF dokumentov s paketi [Documenter](https://documenter.juliadocs.org/stable/) ali [Weave.jl](https://github.com/JunoLab/Weave.jl) je potrebno namestiti [TeX/LaTeX](https://tug.org/). Priporočam uporabo [TinyTeX](https://yihui.org/tinytex/). Po [namestitvi](https://yihui.org/tinytex/#installation) tinytex, dodamo še nekaj `LaTeX` paketov, tako da v terminalu izvedemo naslednji ukaz

```
tlmgr install microtype upquote minted
```

1.6 Povezave

- [Način dela za Gitlab (Gitlab Flow)](https://docs.gitlab.com/ee/topics/gitlab_flow.html).
- [Priporočila za stil Julia](https://docs.julialang.org/en/v1/manual/style-guide/).
- [Naveti za delo z Julijo](https://docs.julialang.org/en/v1/manual/workflow-tips/).

1.7 Git

[Git](https://git-scm.com/) 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 lastnem strežniku ali na enm od javnih spletnih skladišč za programsko kodo npr. [Github](https://github.com/) ali [Gitlab](https://gitlab.com/).

1.7.1 Povezave

Spodaj je nekaj povezav, ki bodo bralcu v pomoč pri uporabi programa [Git](https://git-scm.com/):

- vmesnik za git [Git Extensions](https://gitextensions.github.io/) za Windows,
- [način dela za Github (Github flow)](https://docs.github.com/en/get-started/using-github/github-flow),
- [način dela za Gitlab (Gitlab Flow)](https://docs.gitlab.com/ee/topics/gitlab_flow.html).

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`.

2.2 Računajne kvadratnega korena z babilonskim obrazcem

Z računajnjem kvadratnega korena so se ukvarjali pred 3500 leti v Babilonu. O tem si lahko več preberete v [članku v reviji Presek](http://www.presek.si/21/1160-Domajnko.pdf). 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:

```
"""
  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 3: Funkcija, ki računa kvadrani koren s Heronovim obrazcem.

Preskusimo funkcijo na številu 3.

```
  x = koren_heron(3, 1.7, 5)
  println("koren 3 je $(x)!")
```

```
[ Info: Približek na koraku 1 je 1.7323529411764707
[ Info: Približek na koraku 2 je 1.7320508339159093
[ Info: Približek na koraku 3 je 1.7320508075688776
[ Info: Približek na koraku 4 je 1.7320508075688772
[ Info: Približek na koraku 5 je 1.7320508075688772
koren 3 je 1.7320508075688772!
```

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).