Martin Vuk

Rešeni problemi iz numerične matematike

Oktober 2021
Fakulteta za računalništvo in informatiko, Univerza v Ljubljani

Kazalo

Uvod

To je dokumentacija z gradivi za vaje in domače naloge pri predmetu Numerična matematika.

Za praktično delo pri tem predmetu bomo uporabljali platformo GitLab, ki omogoča vodenje projektov in sodelovanje. Preberite si več o načinu dela v vodiču za sodelovanje in kako poteka delo v GitLab.

Navodila za hiter začetek

Prenesite kodo na svoj računalnik z ukazom `git clone`

```
git clone https://gitlab.com/nummat/nummat-1920.git
```

Programi so napisani v programskem jeziku julia. Za urejanje programov priporočamo uporabo urejevalnikov Atom, Visual Studio Code ali emacs, a vsak sodoben urejevalnik bi moral zadoščati. Za začetek poženite julia REPL, v katerem lahko preiskusite programe in primere iz vaj

```
cd nummat-1920
julia --project=@.
```

V julia REPL naložimo paket/knjižnico `Nummat` in pogledamo opis modula z makro ukazom `@doc`

```
julia> using NumMat

julia> @doc NumMat
  Programi pri numerični matematiki na FRI
```

> **Posebnosti julia REPL**
>
> Julia REPL omogoča različne načine, v katerih je vnos različno interpretiran. V druge načine pridemo, če na začetku vnesemo enega od posebnih znakov `;`, `?` ali `]`:
>
> - če pritisnemo `;`, se vsi ukazi izvedejo v sistemski lupini
> - če pritisnemo `?` lahko iščemo po dokumentaciji za funkcije in tipe
> - če pritisnemo `]` lahko upravljamo s paketi

Testi

Teste za cel projekt lahko poženemo z ukazom `Pkg.test()`:

```
julia> import Pkg
julia> Pkg.test()
```

Če želimo pognati le teste v določeni datoteki, uporabimo funkcijo `include`

```
include("../test/vaje/vaja03/laplace2D.jl")
```

Dokumentacija

Dokumentacija je shranjena v mapi `docs/src/`. Za generiranje html strani z dokumentacijo uporabljamo Documenter.jl. Najprej moramo pripraviti okolje za pripravo dokumentacije

```
julia> import Pkg
julia> Pkg.activate("docs")
julia> Pkg.instantiate()
```

Html strani nato generiramo z ukazom

```
julia> include("docs/make.jl")
```

Ko je dokumentacija izdelana, se jo najde v `docs/build/`.

Povezave

- Algoritmi numerične matematike
- Predmet Numerična matematika na spletni učilnici

Računanje elementarnih funkcij

Računanje kvadratnega korena

Računalniški procesorji navadno implementirajo le osnovne številske operacije: seštevanje, množenje in deljenje. Za računanje drugih matematičnih funkcij mora nekdo napisati program.

> **Elementarne funkcije so v stadardni knjižnici večine jezikov**
>
> Večina programskih jezikov vsebuje elementarne funkcije v standardni knjižnici. Tako tudi `julia`. Lokacijo metod, ki računajo kvadratni koren lahko dobite z ukazom `methods(sqrt)`.

Naloga

Na različne načine izračunaj kvadratni koren. Napiši več metod za funkcijo `koren`, tako da uporabiš večlično razpošiljanje (multiple dispatch) na abstraktne tipe, ki predstavljajo različne metode. Na primer definiramo dve različni metodi za koren

```
struct NapacenKoren end
struct VgrajenaFunkcija end

function koren(x, metoda::NapacenKoren)
 x/2
end

function koren(x, metoda::VgrajenaFunkcija)
 sqrt(x)
end
```

```
koren (generic function with 2 methods)
```

ki jih nato lahko pokličemo, tako da spremenimo 2. argument:

```
julia> koren(2, NapacenKoren())
1.0

julia> koren(2, VgrajenaFunkcija())
1.4142135623730951
```

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. Moderna verzija metode računanja približka predstavlja rekurzivno zaporedje, ki konvergira k vrednosti kvadratnega korena danega števila x. Rekurzivna formula

$$a_{n+1} = \frac{1}{2} \cdot (a_n + \frac{x}{a_n})$$

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 primer računanje $\sqrt{2}$

```
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 a lahko v 64 bitnih številih s plavjočo vejico.

Napišimo zgornji algoritem še kot funkcijo.

```
"""
    koren_babilonski(x, x0, n)

Izračuna približek za koren števila `x` z `n` koraki babilonskega obrazca z začetnim približkom
↪  `x0`.
"""
function koren_babilonski(x, x0,  n)
    a = x0
    for i = 1:n
        a = (a + x/a)/2
    end
    return a
end
```

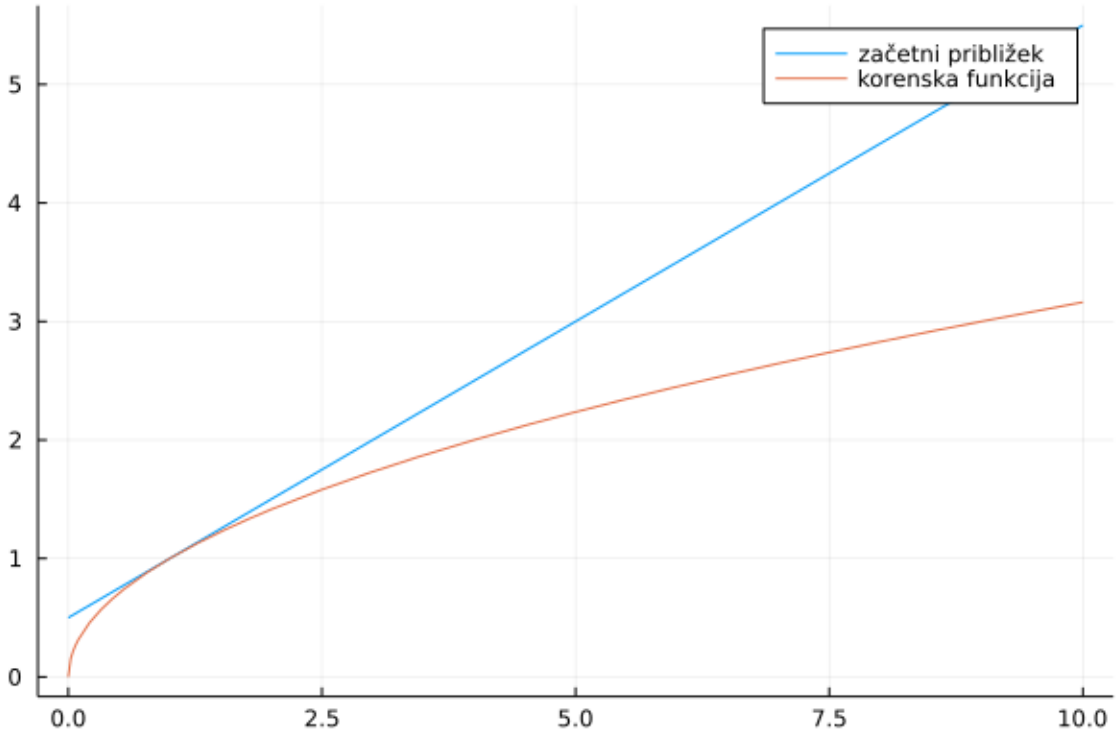
Preskusimo funkcijo na številu 3.

```
x  = koren_babilonski(3, 1.7, 5)
println("Koren števila 3: $x")
println("Razlika z vgrajeno funkcijo: $(x-sqrt(3))")
```

```
Koren števila 3: 1.7320508075688772
Razlika z vgrajeno funkcijo: 0.0
```

Metoda navadne iteracije in tangentna metoda

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



Slika 1: začetni približek v primerjavi z dejansko vrednostjo korena

Izbira začetnega približka

Funkcija `koren_babilonski(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. Funkcija mora sama izbrati začetni približek, kot tudi število korakov.

Kako bi učinkovito izbrali dober začetni približek? Ker rekurzivno zaporedje konvergira ne glede na izbran začetni približek, lahko uporabimo kar samo število x. Ali pa uporabimo diferencial(prva dva člena v Taylorjevem razvoju) okrog števila 1

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

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

```
using Plots
plot(x->1/2 + x/2, 0, 10, label="začetni približek")
plot!(x->sqrt(x), 0, 10, label="korenska funkcija")
savefig("koren_zacetni_priblizek.png")
```

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

$$x = m2^e$$

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).

Koren števila x lahko potem izračunamo kot

$$\sqrt{x} = \sqrt{m}2^{e/2}$$

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

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

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

$$\sqrt{2^e} \approx 2^d,$$

pri čemer smo ostanek zanemarili. Celi del števila pri deljenju z 2 lahko dobimo z binarnim premikom v desno (right shift). Tako lahko zapišemo naslednjo funckijo za začetni približek

```
"""
    zacetni_priblizek(x)

Izračunaj začetni približek za tangentno metodo za računanjekvadratnega korena števila `x`.
"""
function zacetni_priblizek(x)
  d = exponent(x) >> 1 # desni premik, oziroma deljenje z 2
  m = significand(x)
  return (0.5 + 0.5*m)*(2^d)
end
```

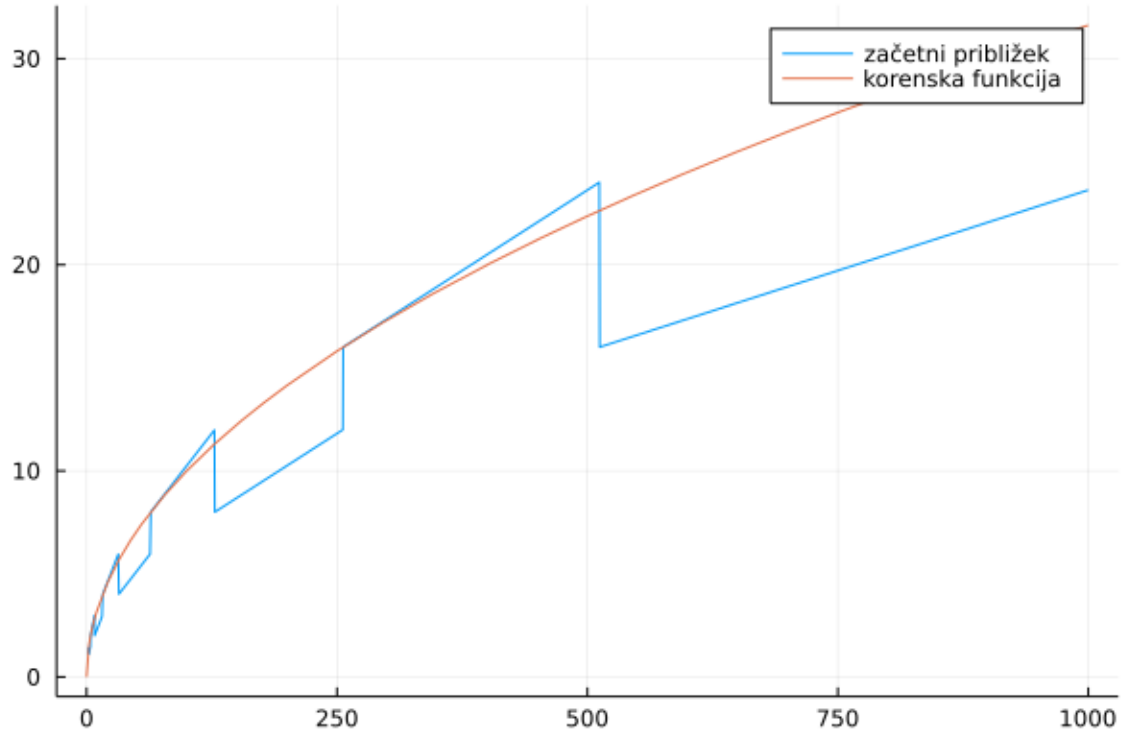
```
Main.ex-koren_priblizek.zacetni_priblizek
```

Začetni približek se sedaj bistveno bolje prilega korenski funkciji

```
  using Plots
  plot(zacetni_priblizek, 0, 1000, label="čzaetni žpribliek")
  plot!(sqrt, 0, 1000, label="korenska funkcija")
  savefig("izboljsan_priblizek_koren.png")
```

Poglejmo, kako se obnaša relativna napaka, če uporabimo izboljšano verzijo začetnega približka.

```
using Plots
rel_napaka(x) = (koren_babilonski(x, zacetni_priblizek(x), 6)^2 - x)/x
scatter(rel_napaka, 0, 10000, label="Relativna napaka", markersize=2)
savefig("napaka_koren.png")
```



Slika 2: Izboljšan začetni približek

Transformacija definicijskega območja

Določena formula ali metoda ni vedno uporabna na celotnem definicijskem območju funkcije. Tako smo videli v prejšnjem razdelku, da je fiksno število korakov tangentne metode s preprosto formulo za začetni približek dalo dober rezultat le v bližini števila 1. Rešitev lahko naslovimo s transformacijo definicijskega območja na ožji interval, na katerem izbrana metoda dobro deluje. Poglejmo si zopet zapis s plavajočo vejico

$$x = m \cdot 2^{2d+o}$$

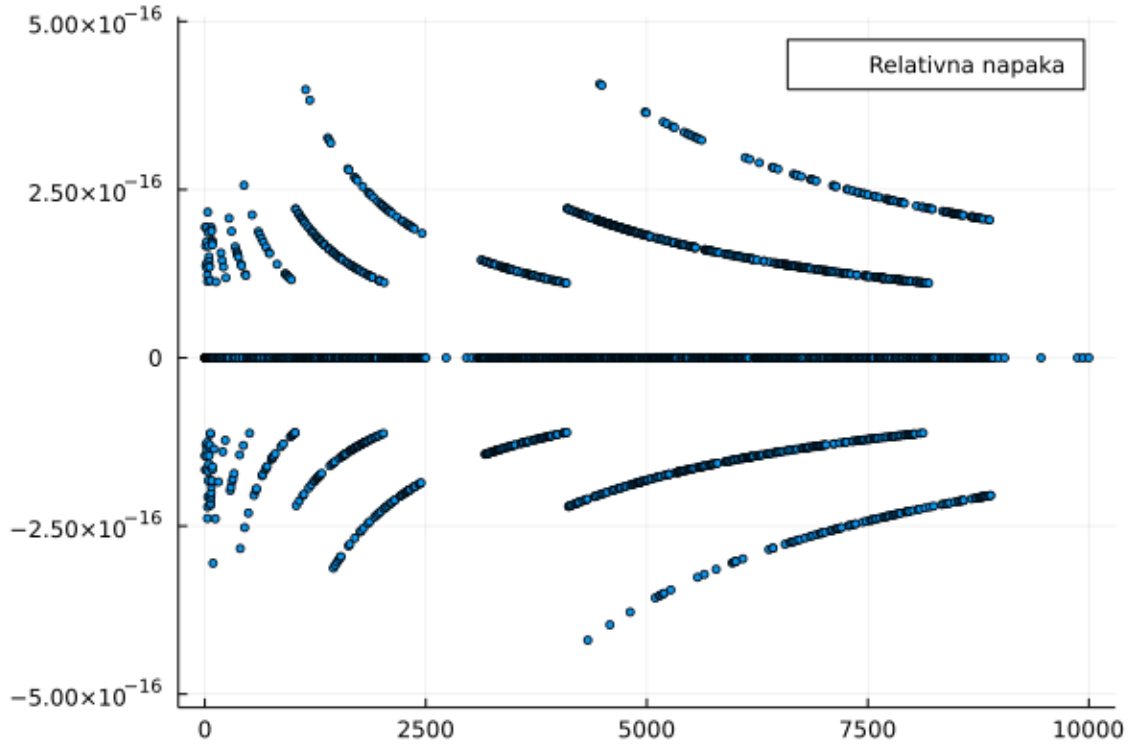
kjer smo eksponent že razstavili na sodo število in ostanek. Če izračunamo koren

$$\sqrt{x} = \sqrt{m \cdot 2^o} \cdot 2^d$$

Ker je $m \in [0.5, 1)$, je $x \cdot 2^{-2d} = m \cdot 2^o \in [0.5, 2)$, lahko za izračun korena uporabimo formulo, ki deluje na intervalu $[0.5, 2)$.

Izbrati moramo število korakov, pri katerem bo relativna napaka ustrezno majhna na intervalu $[0.5, 2)$. To se zgodi pri $n = 6$ za izbiro začetnega približka $1/2 + x/2$.

```
using Plots
rel_napaka(x) = (koren_babilonski(x, 0.5 + x/2, 6)^2 - x)/x
plot(rel_napaka, 0.5, 2)
savefig("napaka_koren_polovica_2.png")
```

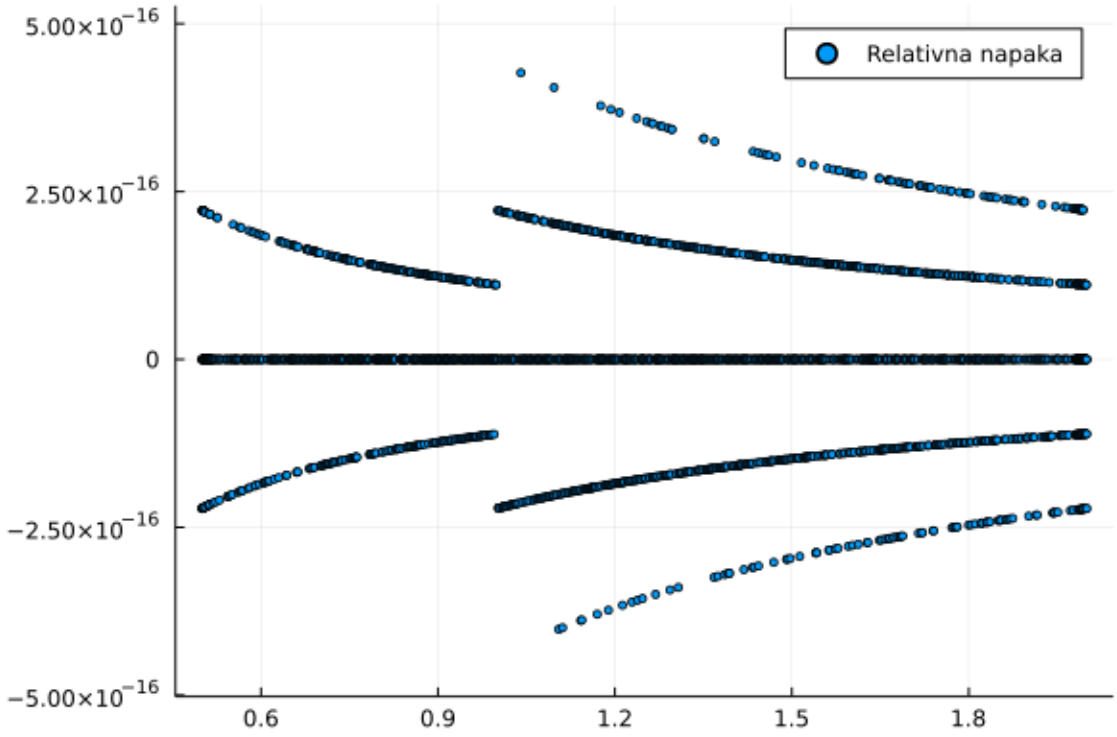


Slika 3: Relativna napaka metode z izboljšanim začetnim približkom

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

```
"""
    metoda = BabilonskiObrazec()

Pomožni podatkovni tip, ki predstavlja metodo računanja korena z babilonskim obrazcem
"""
struct BabilonskiObrazec
end

"""
    koren(x, BabilonskiObrazec())

Izračunaj kvadratni koren danega števila `x` z babilonskim obrazcem.
"""
function koren(x, metoda::BabilonskiObrazec)
    d = (exponent >> 1) # polovični eksponent
    x_trans = x * 2^(-(d << 1)) # transfromacija na interval [0.5, 2)
    x_0 = 0.5 + x_trans/2
    return koren_babilonski(x_trans, x_0, 6) * 2^d
end
```

```
Main.##ex-#263.koren
```



Slika 4: Relativna napaka na [0.5, 2]

Relativna napaka je neodvisna od izbranega števila, prav tako za izračun tudi potrebujemo enako število operacij.

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. Več o računanju obratne vrednosti kvadratnega korena.

Linearni sistemi

Tridiagonalni sistemi

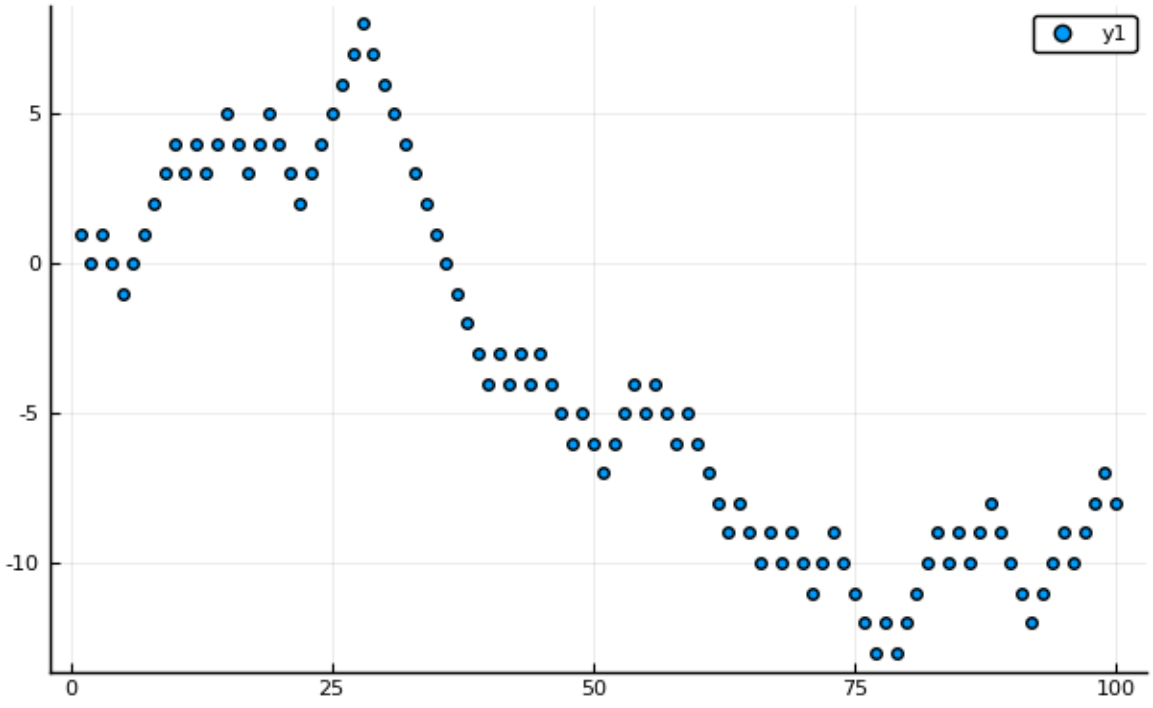
Slučajni sprehod

Poglejmo si primer slučajnega sprehoda v eni dimenziji. Slučajni sprehod je vrsta stohastičnega procesa, ki ga lahko opišemo z Markovsko verigo z množico stanj enako celim številom. Če se trenutno nahajamo v stanju n, se lahko v naslednjem koraku z verjetnostjo p premaknemo v stanje $n-1$ ali z verjetnostjo $q = 1 - p$ pa v stanje $n+1$.

```
# Primer prvih 100 korakov slučajnega sprehoda p=q=1/2
using Plots
scatter(cumsum(2*round.(rand(100)).-1))
savefig("sprehod.png")
```

Naloga

Izračunaj povprečno število korakov, ki jih potrebujemo, da se od izhodišča oddaljimo za k korakov.



Slika 5: Slučajni sprehod

Markovske verige

Prehodna matrika Markovske verige

Markovsko verigo lahko opišemo z zaporedjem slučajnih spremenljivk X_k, $k = 1, \ldots$, ki opisujejo trenutno stanje Markovske verige z Markovsko lastnostjo

$$P(X_{n+1} = x_{n+1} | X_1 = x_1, X_2 = x_2, \ldots, X_n = x_n) = P(X_{n+1} = x_{n+1} | X_n = x_n),$$

ki pomeni, da je verjetnost za prehod v naslednje stanje odvisna le od prejšnjega stanja in ne od starejše zgodovine stanj. Matriko P, katere elementi so prehodne verjetnosti prehodov med stanji Markovske verige

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

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

$$P\mathbf{1} = \mathbf{1}$$

Absorpcijska stanja

Absorpcijsko stanje je stanje, iz katerega se ne moremo več premakniti, medtem ko je *prehodno stajne* stanje, ki ga obiščemo le končno mnogo krat. Markovske verige z absorpcijskimi takimi stanji imenujemo absobirajoča Markovska veriga. Vrstica v prehodni matriki, ki ustreza absorpcijskemu stanju ima le diagonalni element enak 1, vsi ostali so nič.

Fundamentalna matrika

Če ima Markovska veriga absorpcijska stanja, lahko prehodno matriko zapišemo v naslednji bločni obliki

$$P = \begin{bmatrix} Q & T \\ 0 & I \end{bmatrix},$$

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

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

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

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

$$\mathbf{k} = N\mathbf{1} = (I - Q)^{-1}\mathbf{1}.$$

Če želimo poiskati pričakovano število korakov, moramo rešiti sistem linearnih enačb

$$(I - Q)\mathbf{k} = \mathbf{1}.$$

Prehodna in fundamentalna matrika slučajnega sprehoda

Če nas zanima le kdaj bo sprehod za k oddaljen od izhodišča, lahko začnemo v 0 in stanji k in $-k$ proglasimo za absorpcijska stanja. Prehodna matrika, ki jo dobimo je tridiagonalna z 0 na diagonali. Matrika $I - Q$ je prav tako tridiagonalna z 1 na diagonali in z negativnimi verjetnostmi $-p$ in $-q = p - 1$ na obdiagonalnih elementih:

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

Prilagojen podatkovni tip

Naj bo $A \in \mathbb{R}^{n \times n}$ tri-diagonalna, diagonalno dominantna matrika. Primer tridiagonalne 4×4 matrike

$$A = \begin{pmatrix} 3 & 1 & 0 & 0 \\ 2 & 4 & -1 & 0 \\ 0 & 1 & 3 & -1 \\ 0 & 0 & -1 & 8 \end{pmatrix}.$$

Definirajte podatkovni tip `Tridiagonalna`, ki hrani le neničelne elemente tridiagonalne matrike. Za podatkovni tip `Tridiagonalna` definirajte metode za naslednje funkcije:

- indeksiranje: `Base.getindex`,`Base.setindex!`,`Base.firstindex` in `Base.lastindex`
- množenje z desne `Base.*` z vektorjem
- „deljenje“ z leve `Base.\`

Časovna zahtevnost omenjenih funkcij naj bo linearna. Več informacij o tipih in vmesnikih.

Poissonova enačba na krogu

Drug primer, ko dobimo tridiagonalni sistem lineranih enačb, če iščemo rešitev za robni problem na krogu $x^2 + y^2 \leq 1$ za Poissonovo enačbo

$$\triangle u(x,y) = f(r)$$

z robnim pogojem $u(x,y) = 0$ za $x^2 + y^2 = 1$. Pri tem je $f(r) = f(\sqrt{x^2 + y^2})$ podana funkcija, ki je odvisna le od razdalje do izhodišča.

Laplaceov operator zapišemo v polarnih koordinatah in enačbo diskretiziramo z metodo končnih diferenc.

Minimalne ploskve (Laplaceova enačba)

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 wikipedia

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

$$\Delta u(x, y) = \rho(x, y).$$

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

$$\Delta u(x, y) = 0.$$

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*:

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

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.

Slika 6: Muenchenski Olimpijski park (slike so vzete iz wikipedie)

Diskretizacija in linearni sistem enačb

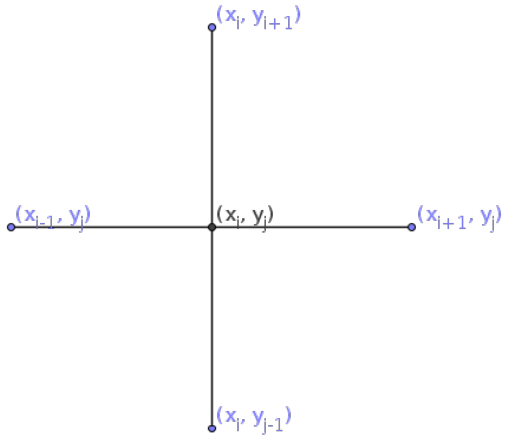
Problema se bomo lotili numerično, zato bomo vrednosti $u(x, y)$ poiskali le v končno mnogo točkah: problem bomo *diskretizirali.* Za diskretizacijo je najpreprosteje uporabiti enakomerno razporejeno pravokotno mrežo točk. Točke na mreži imenujemo *vozlišča*. Zaradi enostavnosti bomo obravnavali le mreže z enakim razmikom v obeh koordinatnih smereh. Omejimo se le na pravokotna območja v ravnini $[a, b] \times [c, d]$. Interval $[a, b]$ razdelimo na $n + 1$ delov, interval $[c, d]$ pa na $m + 1$ delov in dobimo zaporedje koordinat

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

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

$$u_{ij} = u(x_i, y_j), \quad i = 1, \ldots n, j = 1, \ldots m$$

Iščemo torej enačbe, ki jim zadoščajo elementi matrike u_{ij}. 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



Slika 7: sosednja vozlišča

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

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.

Matrika sistema linearnih enačb

Sisteme linearnih enačb običajno zapišemo v matrični obliki

$$A\mathbf{x} = \mathbf{b},$$

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

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

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

desne strani pa so

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

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")
```

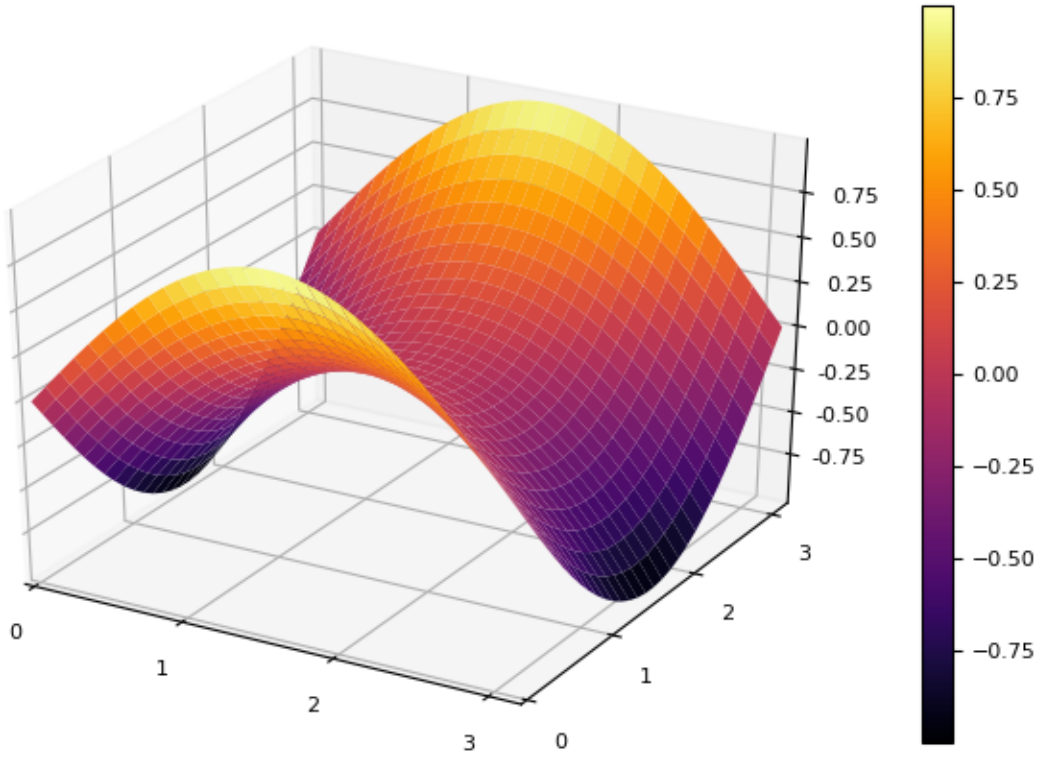
Napolnitev matrike ob eliminaciji

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)
```



Slika 8: minimalna ploskev

Koda

- NumMat.LaplaceovOperator
- NumMat.LaplaceovOperator
- NumMat.RobniProblemPravokotnik
- NumMat.desne_strani
- NumMat.matrika
- NumMat.resi

NumMat.LaplaceovOperator – Type.

```
L = LaplaceovOperator{2}()
```

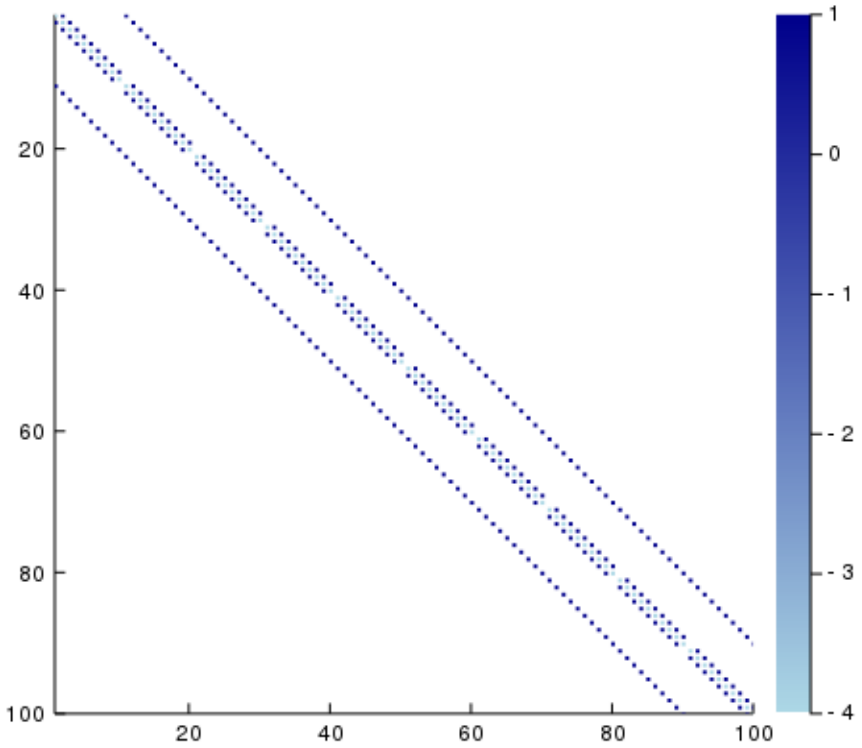
Podatkovni tip brez vrednosti, ki predstavlj laplaceov operator v d dimenzijah.

NumMat.LaplaceovOperator – Method.

```
LaplaceovOperator(d)
```

Vrni vrednost tipa LaplaceovOperator{d} v d dimenzijah.

NumMat.RobniProblemPravokotnik – Type.



Slika 9: Redka struktura Laplaceove matrike

```
RobniProblemPravokotnik(operator, ((a, b), (c, d)), [f_s, f_d, f_z, f_l])
```

Definiraj robni problem za enačbo z danim diferencialnim operatorjem

$$\mathcal{L}u(x, y) = 0$$

na pravokotniku $[a, b] \times [c, d]$, kjer so vrednosti na robu podane s funkcijami $u(x, c) = f_s(x)$, $u(b, y) = f_d(y)$, $u(x, d) = f_z(x)$ in $u(a, y) = f_l(y)$.

NumMat.desne_strani – Method.

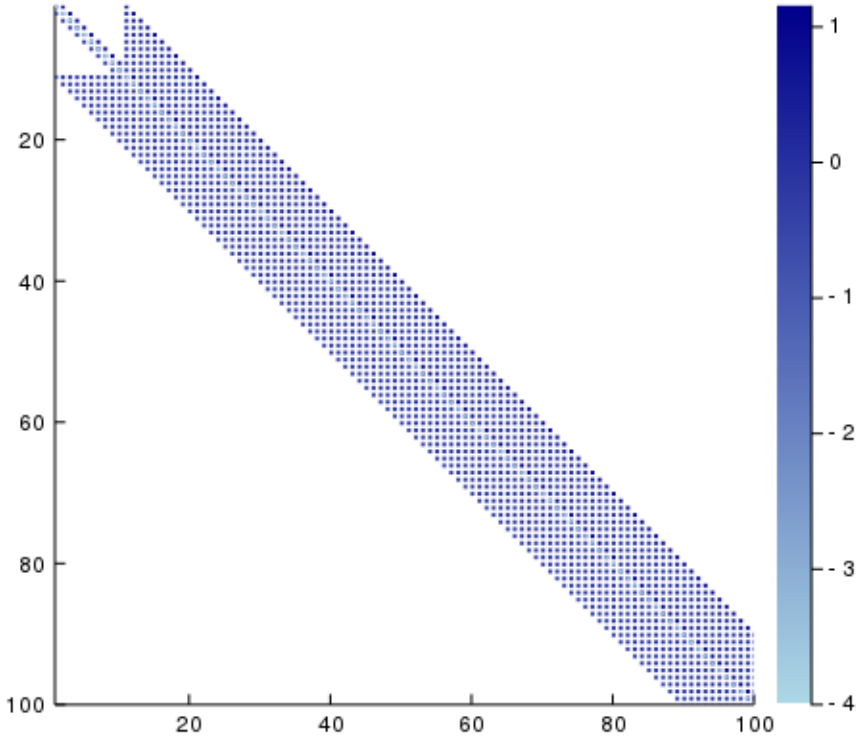
```
desne_strani(s, d, z, l, LaplaceovOperator(2))
```

Izračunaj desne strani pri reševanju robnega problema za Laplaceovo enačbo v 2 dimenzijah.

Argumenti

- `s::Vector`: robne vrednosti na spodnjem robu
- `d::Vector`: robne vrednosti na desnem robu
- `z::Vector`: robne vrednosti na zgornjem robu
- `l::Vector`: robne vrednosti na levem robu

NumMat.matrika – Method.



Slika 10: Napolnitev ob razcepu

```
L = matrika(n, m, LaplaceovOperator(2))
```

Zapiši matriko za Laplaceov operator v 2D na pravokotnem območju. Matrika L je matrika sistema enačb za diskretizirano laplaceovo enačbo

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

`NumMat.resi` – Method.

```
resi(::RobniProglemPravokotnik; nx=100, ny=100)
```

Izračunaj približek za rešitev robnega problema za operator z metodo deljenih diferenc.

Rezultat

- `Z::Matrix` je matrika vrednosti rešitve v notranjosti in na robu.
- `x::Vector` je vektor vrednosti abscise
- `y::Vector` je vektor vrednosti ordinate

Primer

```
using Plots
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)
```

Iteracijske metode

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.

> **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

> **Pogoji konvergence**
>
> Rekli boste, to je preveč enostavno, če enačbe le pruredimo in se potem rešitel kar sama pojavi, če le dovolj dolgo računamo. Gotovo se nekje skriva kak hakelc. Res je! Težave se pojavijo, če zaporedje približkov **ne konvergira dovolj hitro** ali pa sploh ne. Jakobijeva, Gauss-Seidlova in SOR iteracija **ne konvergirajo vedno**, zagotovo pa konvergirajo, če je matrika po vrsticah diagonalno dominantna.

Konvergenco jacobijeve iteracije lahko izboljšamo, če namesto vrednosti 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

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

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

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

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

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)
```

Konvergenca Gauss-Seidlove iteracije

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.png")
```

Slika 11: začetni priblizek za iteracijo

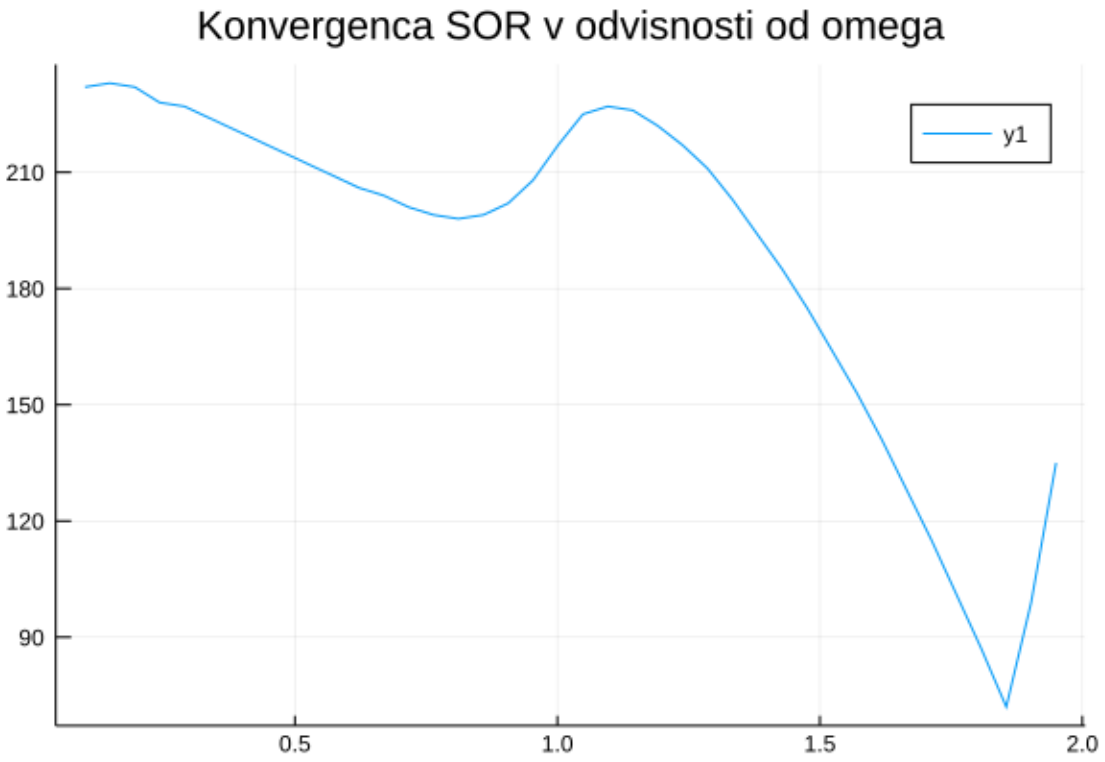
Metoda konjugiranih gradientov

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.

```
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")
```

```
Š
tevilo korakov: 34
```



Slika 12: Konvergenca SOR v odvisnosti od omega

Koda

- Base.:*
- NumMat.conjgrad
- NumMat.desne_strani
- NumMat.iteracija
- NumMat.korak_sor

Base.:* – Method.

```
*(L::LaplaceovOperator{2}, X::Matrix)
```

množi matriko X z matriko za laplaceov operator.

Primer

```
julia> L = LaplaceovOperator{2}();

julia> L*[1 2 3 4; 5 0 1 6; 7 8 9 10]
3×4 Array{Int64,2}:
 1   2  3   4
 5  -1  4   6
 7   8  9  10
```

NumMat.conjgrad – Function.

```
it = conjgrad!(A, b, x; tol=1e-10, maxit=1000)
```

metoda konjugiranih gradientov za reševanje sistema enačb

$$Ax = b$$

Primer

```
julia> A = [2 1 0;1 2 1; 0 1 2]; b = ones(3);

julia> x0 = zeros(3);

julia> x, it = conjgrad(A, b, x0)
([-0.5, -1.0, -0.5], 2)
```

NumMat.desne_strani – Method.

```
b = desne_strani(L::LaplaceovOperator{2}, U::Matrix)
```

Izračuna matriko desnih strani za Laplaceovo enačbo v 2D iz začetnih pogojev.

NumMat.iteracija – Method.

```
U = iteracija(korak, U0, robni_indeksi)
```

Poišče limito rekurzivnega zaporedja $U_n = korak(U_{n-1})$ z začetnim členom `U0`

Rezultat

- `U` limita zaporedja
- `it` število korakov iteracija

NumMat.korak_sor – Function.

```
U1 = korak_sor(L, U0, ω = 1, spremeni_indekse = [])
```

Izvede en korak SOR iteracije pri reševanju enačb

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

Parametri

- `L::LaplaceovOperator{2}`‘
- `U0::Matrix` matika vrednosti u_{ij}
- `ω::Float` relaksacijski parameter $\omega \in [0, 2]$
- `spremeni_indekse::Array{Tuple,2}` seznam indeksov točk, ki niso podane kot robni pogoji

Rezultat

- `U1::Matrix` vrednost matrike U na naslednji iteraciji

Primer

```
julia> korak_sor(LaplaceovOperator{2}(), [1. 1 1; 2  0 3; 1 4 1])
3×3 Array{Float64,2}:
 1.0  1.0  1.0
 2.0  2.5  3.0
 1.0  4.0  1.0
```

Interpolacija z implicitnimi funkcijami

Radialne bazne funkcije (RBF) so funkcije, katere vrednosti so odvisne od razdalje do izhodiščne točke $f(\vec{x}) = \varphi(\|\vec{x} - \vec{x}_0\|)$ Uporabljajo se za interpolacijo ali aproksimacijo s funkcijo oblike $\sum_i w_i \varphi(\|\vec{x} - \vec{x}_i\|)$, npr. za rekonstrukcijo 2D in 3D oblik v računalniški grafiki. Funkcija φ je navadno pozitivna soda funkcija zvončaste oblike in jo imenujemo funkcija oblike.

Podan je 2D ali 3D oblak točk $\{\vec{x}_1, \ldots \vec{x}_n\}$ in realne vrednosti $\{f_1, \ldots, f_n\}$. Napiši funkcijo, ki interpolira omenjene podatke s funkcijo oblike

$$F(\vec{x}) = \sum_i w_i \varphi(\|\vec{x} - \vec{x}_i\|).$$

To pomeni, da poiščeš vrednosti uteži w_i tako, da bodo izpolnjene enačbe $F(\vec{x_i}) = f_i$.

Naloga

Napiši dve funkciji:

- `w = koeficienti_rbf(phi, x0, f)`, ki poišče vrednosti uteži, če so podane funkcija oblike `phi`, oblak točk podan z matriko `x0` in tabela vrednosti `f`.
- `z = vrednost_rbf(x, w, x0)`, ki izračuna vrednost

$F(\vec{x}) \sum_i w_i \varphi(\|\vec{x} - \vec{x}_i\|)$ za argument `x`, pri čemer je `w` vektor uteži w_i, `x0` pa oblak točk, podan kot matrika.

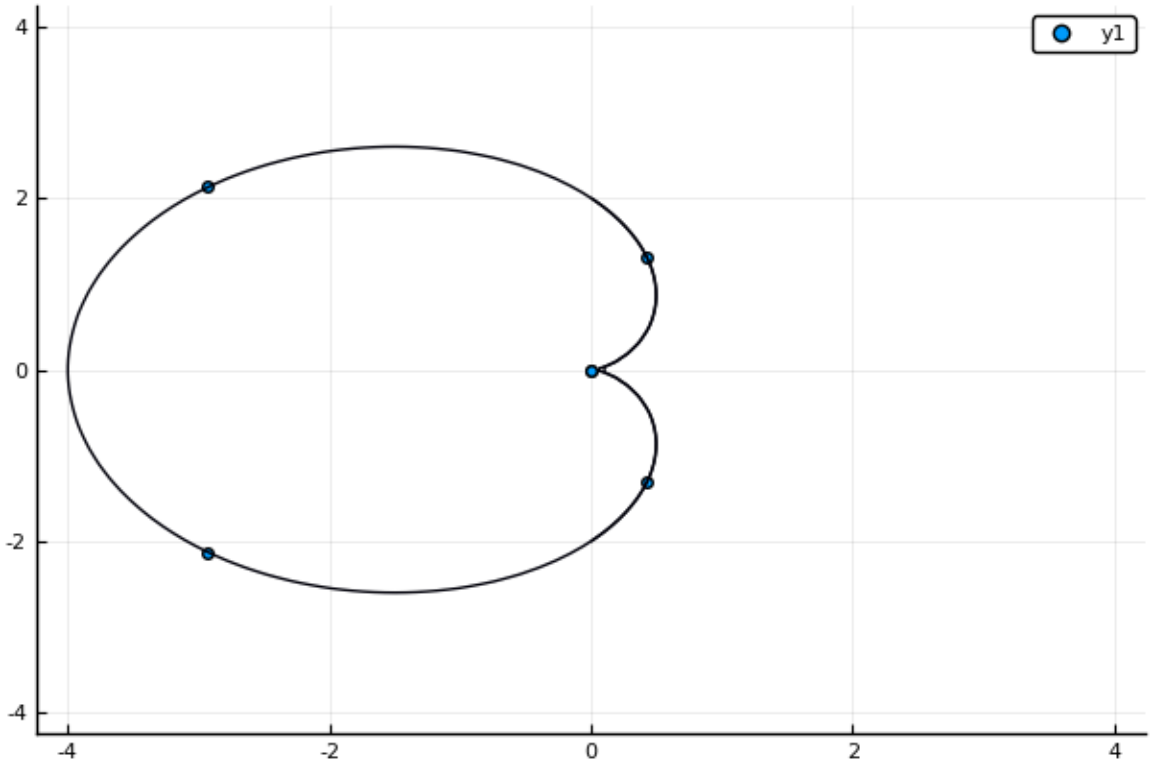
Funkciji uporabi za interpolacijo točk v ravnini z implicitno podano krivuljo, kot v naslednjem primeru:

```
using Plots
fi = range(0, 2π, length=6)
tocke = [2(1-cos(t)).*(cos(t), sin(t)) for t in fi]
scatter(tocke)
f(x,y) = (x^2 + y^2)^2 + 4x*(x^2 + y^2) - 4y^2
x = y = range(-4, 4, length = 100)
contour!(x, y, f, levels = [0])
```

Točke ležijo na nivojnici funkcije $f(x, y) = (x^2 + y^2)^2 + 4x(x^2 + y^2) - 4y^2$ za nivo $f(x, y) = 0$.

Danes bomo spoznali

1. kako napisatni sistem enačb za praktičen primer
2. primerno izbiro metode za reševanje
3. implementacija v programskem jeziku in težave



Slika 13: Nivojska krivulja funkcije, ki gre skozi dane točke

Opis krivulj z implicitno interpolacijo

Iz množice točk želimo rekonstruirati krivuljo, ki gre skozi te točke. Krivulje v ravnini lahko opišemo na različne načine

1. **eksplicitno**: $y = f(x)$
2. **parametrično**: $(x, y) = (x(t), y(t))$
3. **implicitno** z enačbo $F(x, y) = 0$

Tokrat se bomo posvetili implicitni predstavitvi krivulje.

Problem

Imamo točke v ravnini s koordinatami $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$. Iščemo krivuljo, ki gre skozi vse točke. Po možnosti naj bo krivulja gladka, poleg tega ni nujno, da do zaporedne točke v seznamu, tudi zaporedne točke na krivulji. Krivuljo iščemo v **implicitni** obliki, torej v obliki enačbe

$$F(x, y) = 0.$$

Iskano krivuljo bomo zapisali kot ničto nivojnico neke funkcije $F(x, y)$. Iščemo torej funkcijo $F(x, y)$, za katero velja

$$F(x_i, y_i) = 0 \quad i \leq n.$$

Ta pogoj žal ne zadošča. Dodamo moramo še nekaj točk, ki so znotraj območja omejenega s krivuljo. Označimo jih z $(x_{n+1}, y_{n+1}), \ldots, (x_m, y_m)$, v katerih predpišemo vrednost 1

$$F(x_i, y_i) = 1 \quad i \geq n + 1.$$

Naloga

Napiši program, ki za dane točke poišče interpolacijsko funkcijo oblike

$$F(\mathbf{x}) = \sum_i d_i \phi(\mathbf{x} - \mathbf{x}_i) + P(\mathbf{x}),$$

kjer so

- $\mathbf{x} = (x, y)$
- $P(\mathbf{x})$ polinom stopnje 1 (linearna funkcija v x in y)
- d_i primerno izbrane uteži.
- ϕ radialna bazna funkcija, ki je odvisna zgolj od razdalje do *i*-te točke $r = \|\mathbf{x} - \mathbf{x}_i\|$.
  - "thin plate": $\phi(r) = |r|^2 \log(|r|)$ za 2D in $\phi(r) = |r|^3$ za 3D
  - Gaussova: $\phi(r) = \exp(-r^2/\sigma^2)$
  - racionalni približek za Gaussovo

$$\phi(r) = \frac{1}{1 + r^{2p}}$$

Časovna in prostorska zahtevnost

- zgraditev matrike: $\mathcal{O}(n^2)$
- rešitev sistema: $\mathcal{O}((n^2))$, če uporabimo iteracijske metode
- računanje vrednosti funkcije: $\mathcal{O}(n)$

RBF s kompaktnim nosilcem

Matrika sistema, če uporabimo klasične RBF iz prejšnjega razdelka je polna. Čeprav je večina členov izven diagonale zelo majhnih npr. pri gaussovi RBF. Zato so [Morse et. all]@ref(Povezave) prišli na idejo, da uporabijo RBF s kompaktnim nosilcem. V tem primeru je matrika precej bolj redka in se tako prostorska kot tudi časovna zahtevnost algoritmov bistveno zmanjšata.

Povezave

- Savchenko V. V., Pasko, A. A., Okunev, O. G. and Kunii T. L. *Function representation of solids reconstructed from scattered surface points and contours*, Computer Graphics Forum 14(4) (1995),pdf
- G. Turk and J. O'Brien, *Variational Implicit Surfaces*, Technical Report GIT-GVU-99-15, Georgia Institute of Tech-nology, 1998.pdf
- Morse, B. S., Yoo, T. S., Rheingans, P., et al. Interpolating implicit surfaces from scattered surface data using compactly supported radial basis functions, SMI 2001 International Conference on Shape Modeling and Applications, Genova Italy, (2001) pdf

Lastne vrednosti

Konj na šahovnici

Konj naključno skače po šahovnici. Katera polja bolj pogosto obišče?

Kaj pa če po šahovnici skačeta 2 konja? Prej ali slej bo en konj pojedel drugega. Na katerih poljih se bo to najverjetneje zgodilo?

Možne poteze konja

Markovske verige

Problem modeliramo z markovskimi verigami. Prostor stanj markovske verige je v primeru enega konja polje na šahovnici, v primeru dveh konjev pa par šahovskih polj. Markovsko verigo predstavimo konjevim grafom, ki povezuje dve polji, če lahko konj skoči z enega na drugega. Če predpostavimo, da konj skoči na vsako dostopno polje z enako verjetnostjo, dobimo markovsko verigo. Verjetnosti za prehod med stanji predstavimo z matriko prehodnih verjetnosti:

$$P = [p_{kl}]_{k,l} = P(\text{iz stanja k preide v stanje l})$$

Za primer konja na šahovnici lahko stanje označimo s pari števil `(i,j)`, ki označujejo polje na šahovnici, kjer je konj trenutno postavljen. V prehodni matriki smo predpostavili, da je stanje označeno z enim samim številom, zato moramo dvojni indeks `(i,j)` preslikati v zaporedna števila. To lahko storimo, tako da polja številčimo po stolpcih. Če je šahovsko polje velikosti $m \times n$, dobimo naslednjo preslikavo med indeksi:

$$\begin{array}{cccc} (1,1) \to 1 & (1,2) \to m+1 & \dots & (1,m) \to 1+m(n-1) \\ (2,1) \to 2 & (2,2) \to m+2 & \dots & (2,m) \to 2+m(n-1) \\ \vdots & \vdots & \vdots & \vdots \\ (m,1) \to m & (m,2) \to 2m & \dots & (m,n) \to mn \end{array}$$

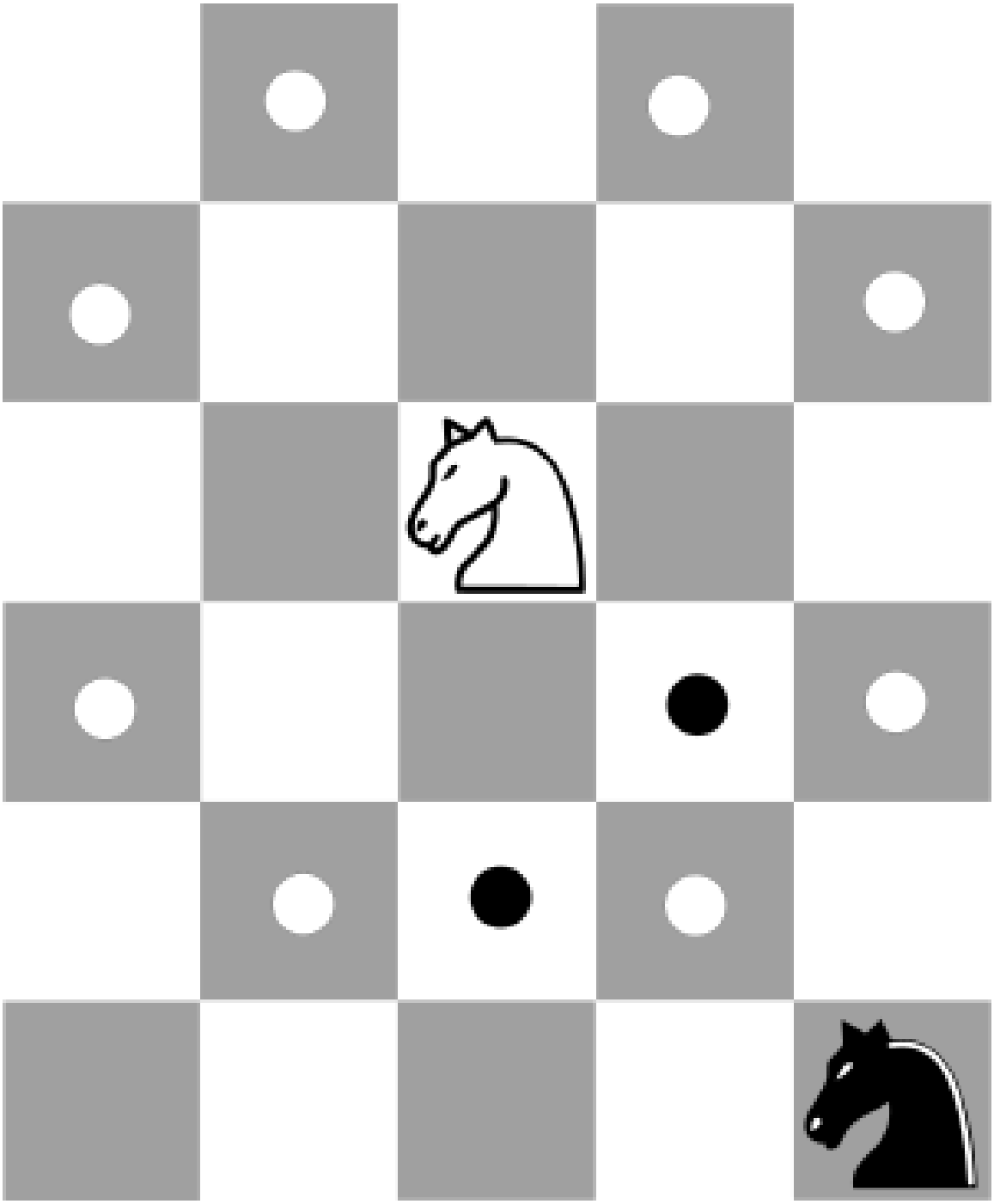
in zaporedni indeks posameznega polja lahko izrazimo s formulo:

$$k = i + (j-1) * m$$

Poglejmo si polje velikosti 3 krat 3. Število stanj je 9 Polja so razvrščena v naslednjem vrstnem redu

$$(1,1), (2,1), (3,1), (1,2), (2,2), (3,2), (1,3), (2,3), (3,3)$$

Prehodna matrika je dimenzije 9×9.



Slika 14: Možne poteze konja

```
julia> using NumMat
julia> P = prehodna_matrika_konj(3, 3)
julia> Matrix(P)
9×9 Array{Float64,2}:
 0.0  0.0  0.0  0.0  0.0  0.5  0.0  0.5  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.5  0.0  0.5
 0.0  0.0  0.0  0.5  0.0  0.0  0.0  0.5  0.0
 0.0  0.0  0.5  0.0  0.0  0.0  0.0  0.0  0.5
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.5  0.0  0.0  0.0  0.0  0.0  0.5  0.0  0.0
 0.0  0.5  0.0  0.0  0.0  0.5  0.0  0.0  0.0
 0.5  0.0  0.5  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.5  0.0  0.5  0.0  0.0  0.0  0.0  0.0
```

Iz vsakega polja razen iz sredine lahko konj skoči le v dve polji, zato so prehodne verjetnosti enake `0.5`. Kot primer vzemimo $p_{1,6} = 0.5$, kar pomeni, da iz polja $(1, 1)$ konj z verjetnostjo 0.5 skoči na polje $(3, 2)$. V tem primeru je prehodna matrika simetrična, a to nikakor ni praivlo.

Limitna porazdelitev Markovske verige

Limitna porazdelitev Markovske verige je lastni vektor transponirane matrike prehodnih verjetnosti P za lastno vrednost 1.

$$P\vec{p} = \vec{p}$$

Da se pokazati, da je 1 največja lastna vrednost prehodne matrike za markovsko verigo, zato lahko za izračun lastnega vektorja uporabimo potenčno metodo.

Potenčna metoda

S potenčno metodo zgradimo zaporedje približkov, tako da prejšnji približek $x^{(n)}$ pomnožimo z matriko A in dobljeni rezultat normiramo:

$$x^{(n+1)} = \frac{Ax^{(n)}}{\|Ax^{(n)}\|}$$

Običajno uporabimo maksimalno normo, saj jo je mogoče najhitreje izračunati. Namesto norme lahko uporabimo tudi katerokoli neničelno komponento približka. Približek za lastno vrednost λ lahko izračunamo z Rayleighovim kvocientom

$$\lambda = \frac{x^T Ax}{x^T x} = x^T Ax$$

Če smo uporabili za normiranje uporabili k-to komponento vektorja, pa lahko približek za lastno vrednost izračunamo kar kot kvocient $(Ax)_k / x_k$.

Premik

Če poskusimo potenčno metodo za primer konja na šahovnici, ugotovimo, da metoda ne konvergira.

```
julia> using NumMat
julia> using Random; Random.seed(321);
julia> P = prehodna_matrika_konj(8, 8);
julia> λ, v, it = potencna(P', rand(64); maxit=10000, tol=1e-2)
ERROR: Potenčna metoda ne konvergira
```

Razlog se skriva v lastnosti konjevega grafa, ki je dvodelen. Zato ima prehodna matrika tudi lastno vrednost -1. Približki potenčne metode tako skačejo med dvema vektorjema

$$\alpha v_1 + \beta v_2 \text{ in } \alpha v_1 - \beta v_2,$$

kjer je v_1 lastni vektor za 1 in v_2 lastni vektro za -1.

Nastali problem lahko rešimo s premikom

> **Premik lastnih vrednosti**
>
> Če matriki A prištejemo večkratnik identične matrike σI, se bodo lastne vrednosti premaknile za σ, lastni vektorji pa bodo ostali enaki. Namestno, da iščemo lastne vektorje matrike A, je včasih bolje poiskati lastne vektorje premaknjene matrike
>
> $$A + \sigma I$$
>
> Še posebej je to uporabno pri inverzni iteraciji, kjer s lastni vektor za poljubno lastno vrednost.

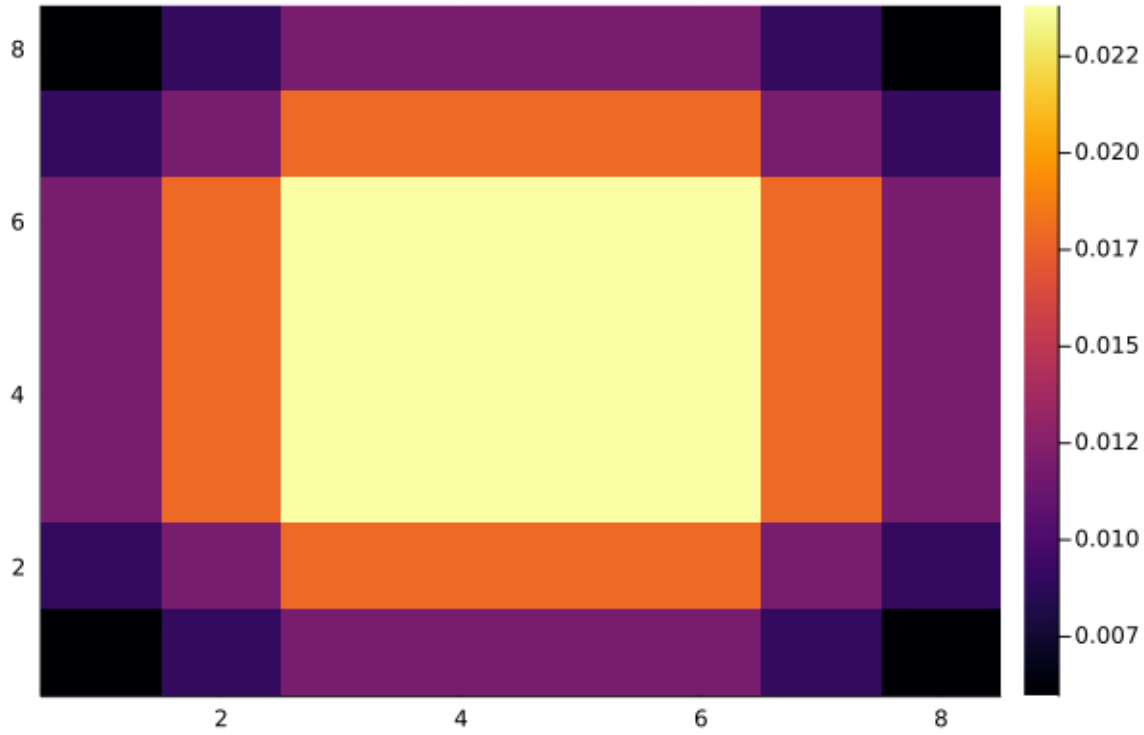
Če prehodno matriko premaknemo za I, se bodo lastne vrednosti premaknile tako, da se bo lastna vrednost 1 premaknila v 2, lastna vrednost -1 pa v 0. Potenčna metoda bo potem konvergirala k invariantni porazdelitvi.

```
using NumMat
import LinearAlgebra.I
import Random.seed!;
seed!(321);
P = prehodna_matrika_konj(8, 8);
λ, v, it = potencna(P'+I, rand(64))
using Plots
heatmap(reshape(v/sum(v), 8, 8))
savefig("invariantna_porazdelitev_konj.png")
```

Invariantna porazdelitev za slučajni sprehod konja na standardni šahovnici

Naloga

Določi invariantno porazdelitev za dva konja. Na katerem polju bo en konj najverjetneje pojedel drugega?



Slika 15: Invariantna porazdelitev za konja na 8x8 šahovnici

Povprečni čas do konca igre

Recimo, da na šahovnici naključno skačeta dva konja. Igra se konča, ko en konj poje drugega (pride na isto polje kot drugi). Zanima nas, koliko je povprečni čas, da se to zgodi. Pomagamo si lahko z matriko prehodnih verjetnosti. Matriko prehodnih verjetnosti lahko zapišemo v kanonični bločni obliki

$$P = \begin{bmatrix} Q & R \\ 0 & I \end{bmatrix}$$

kjer sta Q in R neničelni matriki, I je identiteta in 0 ničelna matrika ustreznih dimenzij.

$$(I - Q)^{-1} \cdot \mathbf{1},$$

kjer je $\mathbf{1} = [1, 1, \ldots 1]^T$ vektor samih enic.

Koda

- `NumMat.MCKonj`
- `Base.:*`

- NumMat.invariantna_porazdelitev
- NumMat.potencna
- NumMat.prehodna_matrika_konj
- NumMat.skoki

NumMat.potencna – Method.

```
λ, v, it = potencna(A, x0; maxit=100, tol=1e-5)
```

Poišči po absolutni vrednosti največjo lastno vrednost in pripadajoči lastni vektor za matriko A s potenčno metodo. Če potenčna metoda ne konvergira, javi napako tipa `ErrorException`.

Base.:* – Method.

```
*(MCKonj(), p)
```

Izračuna produkt porazdelitve p po šahovskih poljih s transponirano prehodno matriko za slučajni sprehod konja na šahovnici.

NumMat.invariantna_porazdelitev – Method.

```
invariantna_porazdelitev(p0; maxit=100, tol=1e-5)
```

Izračunaj invariantno porazdelitev markovske verige, ki predstavlja slučajni sprehod konja po šahovnici. Argument `p0` je začetna porazdelitev po poljih na šahovnici.

Primer

```
p0 = rand(8,8)
p = invariantna_porazdelitev(p0)
```

NumMat.prehodna_matrika_konj – Method.

```
prehodna_matrika_konj(n, m)
```

Vrne prehodno matriko za slučajni sprehod konja na šahovnici dimenzije `n` krat `m`.

NumMat.skoki – Method.

```
skoki(i, j, m, n)
```

Vrne vse možne skoke konja iz polja (i,j) na m x n šahovnici.

NumMat.MCKonj – Type.

Markovska veriga za konja na šahovnici

Nelinearne enačbe

Konvergenčna območja iteracijskih metod

Za reševanje nelinearnih enačb so najbolj pogosto uporablja iteracijske metode, pri katerih približek za rešitev konstruiramo z rekurzivno formulo. Konstrukcija rekurzivnega zaporedja zagotavlja, da je limita zaporedja približkov vedno rešitev enačbe. Glavna težava je, če zaporedje sploh konvergira in h kateri rešitvi konvergira. To je v veliki meri odvisno od začetnega približka in lastnosti funkcije v okolici rešitve.

Kompleksni koreni enote

Kot primer si poglejmo enačbo za kompleksne korene enote. Vemo, da ima enačba

$$z^n = 1$$

$$n$$

kompleksnih rešitev, ki ležijo enakomerno razporejene na enotskem krogu. Poglejmo si, kako konvergira Newtonova metoda, v odvisnosti od začetnega približka.

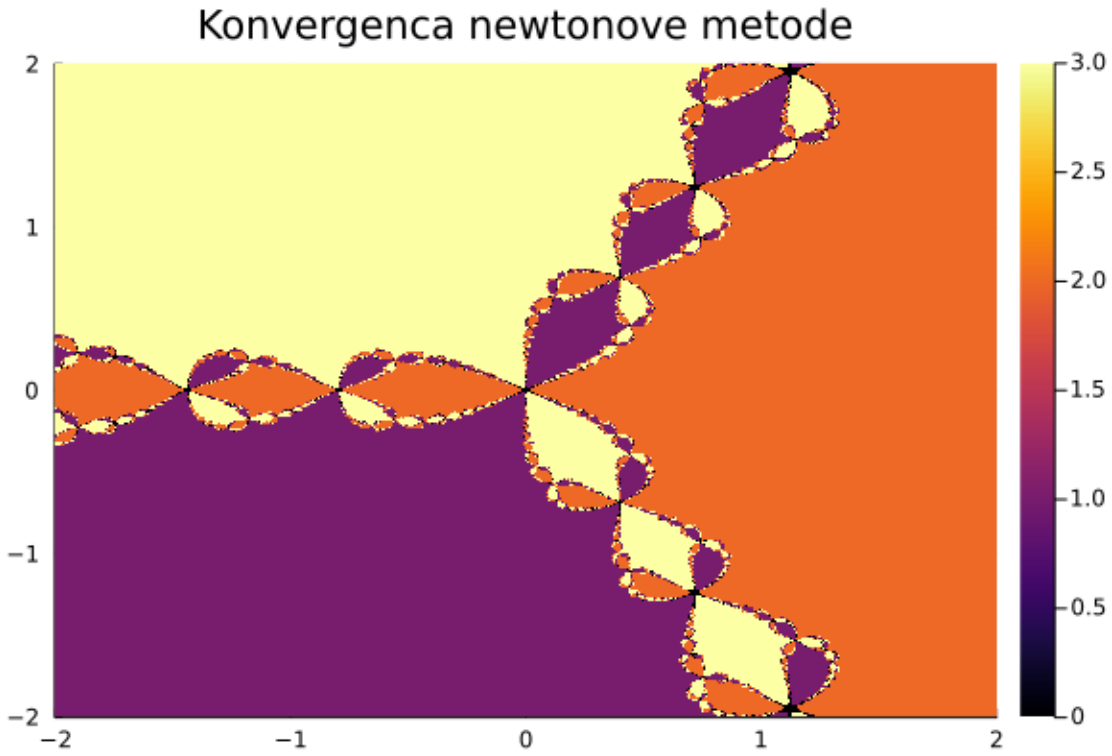
```
using NumMat
using Plots
n = 3
f(z) = z^(n)-1
df(z) = n*z^(n-1)
metoda((x, y); maxit=20, tol=1e-3) = newton(f, df, x + y*im; maxit=maxit, tol=tol)
meje = (-2, 2, -2, 2)
x, y, Z = konvergencno_obmocje(meje, metoda, n=500, m=500)
heatmap(x, y, Z', title="Konvergenca newtonove metode")
savefig("01_konvergenca.png")
```

Koda

- NumMat.konvergencno_obmocje
- NumMat.newton

NumMat.konvergencno_obmocje – Method.

```
x, y, Z = konvergencno_obmocje(obmocje, metoda; n=50, m=50, maxit=50, tol=1e-3)
```



Slika 16: Konvergenca newtonove metode

Izračuna, h katerim vrednostim konvergira metoda `metoda`, če uporabimo različne začetne približke.

Primer

Konvergenčno območje za Newtonovo metodo za kompleksno enačbo $z^3 = 1$

```
julia> F((x, y)) = [x^3-3x*y^2; 3x^2*y-y^3];
julia> JF((x, y)) = [3x^2-3y^2 -6x*y; 6x*y 3x^2-3y^2]
julia> metoda(x0) = newton(F, JF, x0; maxit=10; tol=1e-3);

julia> x, y, Z = konvergencno_obmocje((-2,2,-2,2), metoda; n=5, m=5); Z
5×5 Array{Float64,2}:
 1.0  1.0  2.0  3.0  3.0
 1.0  1.0  2.0  3.0  3.0
 1.0  1.0  0.0  3.0  3.0
 2.0  2.0  2.0  2.0  2.0
 2.0  2.0  2.0  2.0  2.0
```

`NumMat.newton` – Method.

```
x, it = newton(f, df, x0; maxit=100, tol=1e-12)
```

Poišči ničlo funkcije `f` z Newtonovo metodo. Če Newtonova metoda ne konvergira naj metoda vrne par `nothing, maxit`.

Primer

Poiščimo $\sqrt{2}$ kot ničlo funkcije $f(x) = x^2 - 2$:

```
julia> newton(x->x^2-2, x->2x, 1.5)
(1.4142135623730951, 4)
```

Nelinearne enačbe in geometrija

Ogledali si bomo nekaj primerov uporabe metod za reševanje *nelinearnih enačb* v geometriji krivulj in ploskev v 2D in 3D.

Samopresečišče krivulje

Poišči samopresečišča različnih lissajousjevih krivulj. Lissajousjeva krivulja je podana parametrično z enačbama

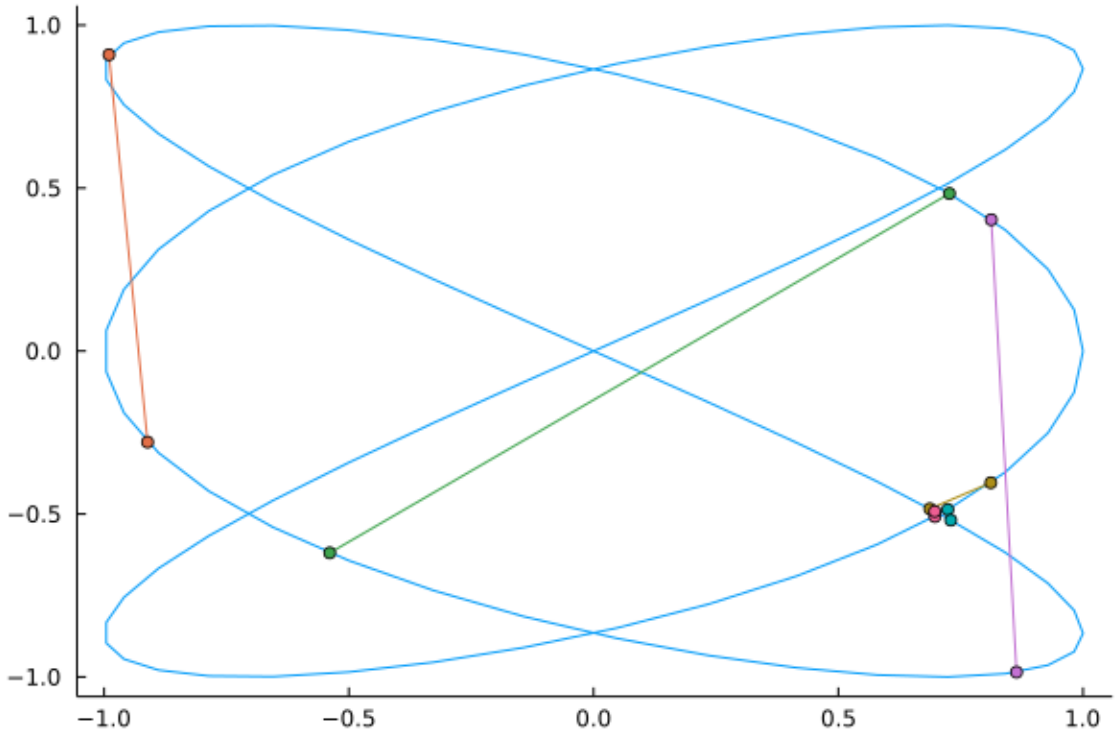
$$(x(t), y(t)) = (a\sin(nt), b\cos(mt)).$$

Primer lissajoujeve krivulje za $n = 3, m = 2$

```
using Plots
x(t) = cos(3*t); y(t) = sin(2*t);
t = range(0, stop=2pi, length=100)
plot(x.(t), y.(t), grid=false, legend=:none)
t = range(0, stop=6, length=21)
# scatter!(x.(t), y.(t))
# annotate!([(x(tt)+0.05, y(tt)+0.05, "t=$tt") for tt in t])
```

Samopresečišče lahko poiščemo z Newtonovo metodo

```
dx(t) = -3sin(3t); dy(t) = 2cos(t);
f(t, s) = [x(t) - x(s), y(t) - y(s)]
Jf(t, s) = [dx(t) -dx(s); dy(t) -dy(s)]

let t0 = 1; s0 = 3
plot!([x(t0), x(s0)], [y(t0), y(s0)], marker=:circle)
for i=1:5
  t0, s0 = [t0, s0] - Jf(t0, s0)\f(t0, s0)
  plot!([x(t0), x(s0)], [y(t0), y(s0)], marker=:circle)
end
end # let
plot!(legend=false)
```



Območje konvergence

Konvergenca newtonove metode je odvina od začetnega približka.

Razdalja med dvema krivuljama

Naj bosta K_1 in K_2 parametrično podani krivulji

$$K_1 : (x(t), y(t)), \quad t \in \mathbb{R}$$
$$K_2 : (\tilde{x}(s), \tilde{y}(s)), \quad s \in \mathbb{R}.$$

Razdaljo med krivuljama lahko definiramo na različne načine

- najmanjša razdalja $d(K_1, K_2) = \min_{x \in K_1, y \in K_2} d(x, y)$
- hausdorffova razdalja

$$d_h(K1, K2) = \max \left\{ \max_{x \in K_2} \min_{y \in K_1} d(x, y), \max_{x \in K_1} \min_{y \in K_2} d(x, y) \right\}$$

Hausdorffova razdalja

Hausdorffova razdalja pove, koliko je lahko točka na eni krivulji največ oddaljena od druge krivulje. Če sta množici blizu v hausdorffovi razdalji, je vsaka točka ene množice blizu drugi množici.

> Medtem, ko je minimalna razdalja med dvema krivuljama vedno končna, pa je lahko hausdorffova razdalja tudi neskončna (na primer, če je ena krivulja omejena, druga pa neomejena).

Najlažje je poiskati minimalno razdaljo. Iščemo točki na krivuljah $(x(t_0), y(t_0)) \in K_1$ in $(\tilde{y}(s_0), \tilde{y}(s_0)) \in K_2$, za katere bo razdalja

$$d(t,s) = \sqrt{(x(t) - \tilde{x}(s))^2 + (y(t) - \tilde{y}(s))^2}$$

najmanjša. Ker je koren naraščajoča funkcija, ga lahko pri iskanju minimuma brez škode izpustimo in obravnavamo funkcijo

$$D(t,s) = d(t,s)^2 = (x(t) - \tilde{x}(s))^2 + (y(t) - \tilde{y}(s))^2,$$

ki je bolj enostavna za računanje. Opazimo, da je razdalja odvisna le od parametrov t in s. Dovolj je, da poiščemo vrednosti (t_0, s_0), v katerih bo vrednost funkcije $d(t,s)$ oziroma $D(t,s)$ minimalna. Če je območje parametrov $(t,s) \in \mathbb{R}^2$ cela ravnina, bo najmanjša vrednost nastopila v *lokalnem ekstremu*, v katerem je tudi *stacionarna točka* funkcije $D(t,s)$. Iskanja lokalnega ekstrema se bomo lotili na dva različna načina, a še prej si oglejmo primer.

Primer

Poglejmo si primer dveh elips v ravnini

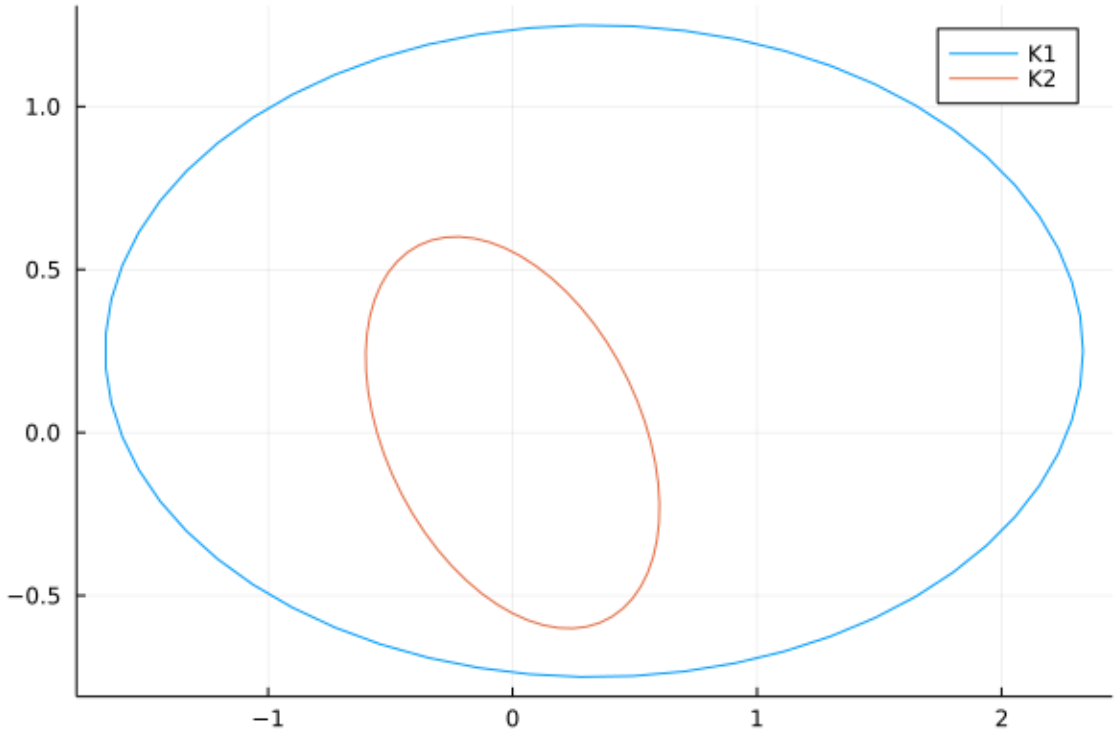
$$\begin{aligned} x(t) &= 2\cos(t) + \frac{1}{3} \\ y(t) &= \sin(t) + \frac{1}{4} \end{aligned}$$

in

$$\begin{aligned} \tilde{x}(s) &= \frac{1}{3}\cos(s) - \frac{1}{2}\sin(s) \\ \tilde{y}(s) &= \frac{1}{3}\cos(s) + \frac{1}{2}\sin(s) \end{aligned}$$

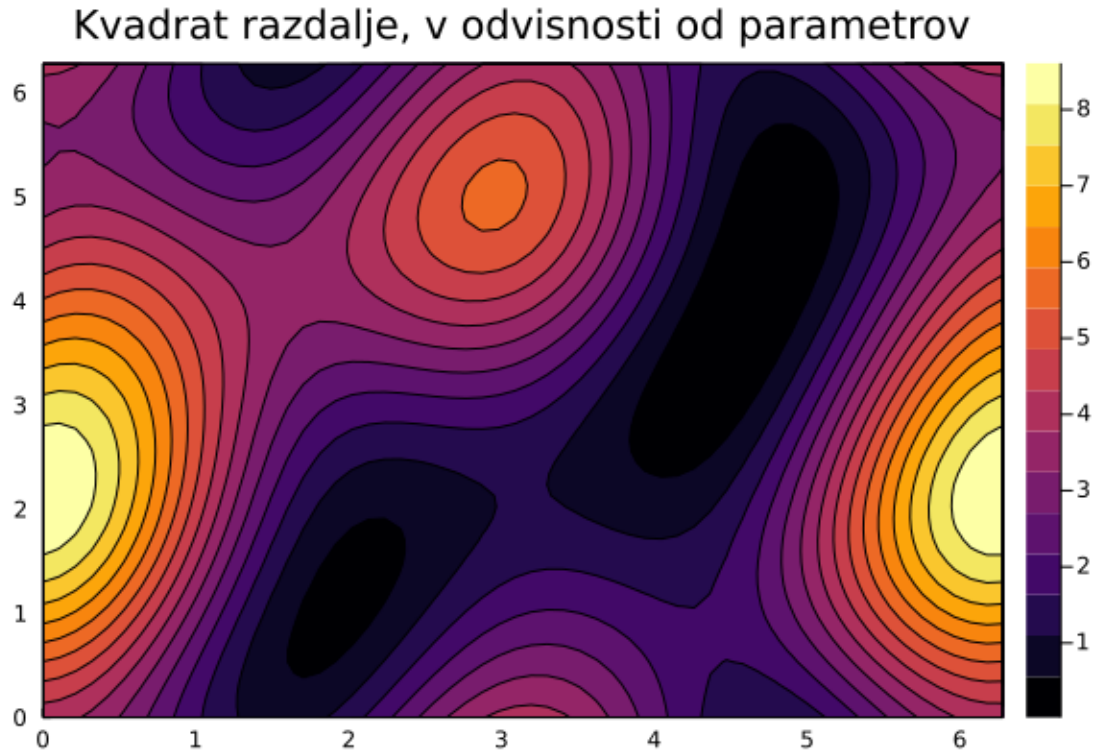
Narišimo ju z

```
using Plots
tocka(a) = tuple(a...)
K1(t) = [2*cos(t) + 1/3, sin(t) + 0.25]
K2(s) = [cos(s)/3 - sin(s)/2, cos(s)/3 + sin(s)/2]
t = LinRange(0, 2*pi, 60);
plot(tocka.(K1.(t)), label="K1")
plot!(tocka.(K2.(t)), label="K2")
```



Narišimo tudi graf funkcije $D(t, s)$ na pravokotniku $[0, 2\pi] \times [0, 2\pi]$.

```
D(t, s) = sum((K1(t) - K2(s)).^2)
contourf(t, t, D, title="Kvadrat razdalje, v odvisnosti od parametrov")
```



Metoda najhitrejšega spusta

Metoda je sila enostavna. Najlažje si jo predstavljamo za iskanje lokalnega mimimuma nadmorske višine. Sestopili bi radi na dno kraške vrtače. Če se držimo metode najhitrejšega spusta, na vsakem koraku izberemo smer, v kateri je pobočje najbolj strmo. V jeziku funkcij to pomeni, da na vsakem koraku izberemo smer, v kateri funkcija najhitreje pada. To je ravno v nasprotni smeri gradienta funkcije. Lokalnemu minimumu funkcije $f(\mathbf{x})$ se lahko tako približamo z naslednjim zaporedjem približkov

$$\mathbf{x}_n = \mathbf{x}_{n-1} - h_n \bigtriangledown f(\mathbf{x}),$$

kjer je $\mathbf{x} = [x_1, \ldots, x_k]^T$ vektor spremenljivk, $h_n \in \mathbb{R}_+$ pa omejeno zaporedje, ki je lahko tudi konstantno.

```
using NumMat
dK1(t) = [-2sin(t), cos(t)]
dK2(s) = [-sin(s)/3 - cos(s)/2, -sin(s)/3 + cos(s)/2]
gradD = gradient_razdalje(K1, K2, dK1, dK2)
slika = contour(t, t, D, title="Zaporedje korakov gradientne metode")

let
x0 = [0.4, 3.3]
h = 0.2
priblizki = [tuple(x0...)]
for i=1:40
  x0 = x0 - h*gradD(x0)
  push!(priblizki, tuple(x0...))
end
scatter!(slika, priblizki)
```

```
end # let
slika #hide
```

Konvergenca je odvisna od zažetnega približka

Limita zaporedja $\mathbf{x}_n$ je zelo odvisna od začetnega približka x_0, pa tudi od izbire parametra h_k. Kaj lahko se nam zgodi, da bomo pristali v "napačnem" minimumu. Prav tako je konvergenca vedno počasnejša, bližje kot smo minimu.

Newtonova metoda

Fermat je med drugim dokazal izrek, ki pove, da je v lokalnem ekstremu vrednost odvoda vedno enaka 0. Isti izrek velja tudi za funkcije več spremenljivk, le da je v tem primeru gradient funkcije enak 0.

Ta izrek morda ni tako razvpit kot njegov zadnji izrek, je pa zato toliko bolj uporaben. Potreben pogoj za nastop lokalnega ekstrema je namreč vektorska enačba

$$\bigtriangledown D(s,t) = 0,$$

ki je v našem primeru sistem dveh enačb z dvema neznankama

$$\begin{aligned}\frac{\partial D(t,s)}{\partial t} &= 2(x(t)-\tilde{x}(s))\dot{x}(t) + 2(y(t)-\tilde{y}(s))\dot{y}(t) = 0\\ \frac{\partial D(t,s)}{\partial s} &= -2(x(t)-\tilde{x}(s))\dot{\tilde{x}}(s) - 2(y(t)-\tilde{y}(s))\dot{\tilde{y}}(s) = 0.\end{aligned}$$

Rešitev zgornjega sistema lahko poiščemo numerično z uporabo Newtonove metode za sisteme enačb. Podobno kot pri Newtonovi metodi za eno spremenljivko potrebujemo odvod, vendar imamo pri več enačbah z več spremenljivkami opravka z odvodom vektorske funkcije več spremenljivk. Sistem lahko zapišemo kot eno samo vektorsko enačbo

$$\mathbf{F}(t,s) = 0,$$

kjer pa je

$$\mathbf{F}(t,s) = \begin{pmatrix} F_1(t,s) \\ F_2(t,s) \end{pmatrix} = \begin{pmatrix} \frac{\partial D(t,s)}{\partial t} \\ \frac{\partial D(t,s)}{\partial s} \end{pmatrix}$$

vektorska funkcija. Odvod funkcije $\mathbf{F}$ lahko predstavimo z linearno preslikavo, ki je dana z matriko (imenovano tudi Jacobijeva matrika sistema):

$$J(\mathbf{F}(t,s)) = \begin{pmatrix} \frac{\partial F_1(t,s)}{\partial t} & \frac{\partial F_1(t,s)}{\partial s} \\ \frac{\partial F_2(t,s)}{\partial t} & \frac{\partial F_2(t,s)}{\partial s} \end{pmatrix}$$

Pri Newtonovi metodi sestavimo zaporedje približkov definirano z naslednjo rekurzivno formulo

$$\mathbf{x}_n = \mathbf{x}_{n-1} - J(\mathbf{F}(\mathbf{x}_{n-1}))^{-1} \cdot \mathbf{F}(\mathbf{x}_{n-1}), \quad \mathbf{x}_n = (t_n, s_n)^T.$$

Če je zaporedje $\mathbf{x}_n$ konvergentno, potem je limita rešitev enačbe $\mathbf{F}(\mathbf{x}) = 0$. Zaporedje $\mathbf{x}_n$ ponavadi zelo hitro konvergira, zato za rešitev potrebujemo relativno malo korakov. Prav tako Newtonovo iteracijo hitro sprogramiramo, saj je dovolj že ena zanka.

Vrnimo se k našem primeru razdalje med dvema krivuljama. Jacobijeva matrika je v tem primeru kar matrika drugih odvodov funkcije razdalje $D(t, s)$

$$J(\mathbf{F}(t,s)) = \begin{pmatrix} \frac{\partial^2 D(t,s)}{\partial t^2} & \frac{\partial^2 D(t,s)}{\partial s \partial t} \\ \frac{\partial^2 D(t,s)}{\partial t \partial s} & \frac{\partial^2 D(t,s)}{\partial s^2} \end{pmatrix}.$$

Enačbe za druge odvode $D(s, t)$ hitro izpeljemo iz gradienta $D(t, s)$

$$\begin{aligned} \frac{\partial^2 D(t,s)}{\partial t^2} &= 2(\ddot{x}(t)(x(t) - \tilde{x}(s)) + \dot{x}^2(t) + \ddot{y}(t)(y(t) - \tilde{y}(s)) + \dot{y}^2(t)) \\ \frac{\partial^2 D(t,s)}{\partial s^2} &= 2(-\ddot{\tilde{x}}(s)(x(t) - \tilde{x}(s)) + \dot{\tilde{x}}^2(s) - \ddot{\tilde{y}}(s)(y(t) - \tilde{y}(s)) + \dot{\tilde{y}}^2(s)) \\ \frac{\partial^2 D(t,s)}{\partial t \partial s} &= -2(\dot{\tilde{x}}(s)\dot{x}(t) + \dot{\tilde{y}}(s)\dot{y}(t). \end{aligned}$$

Poleg tega velja

$$\frac{\partial^2 D(t,s)}{\partial s \partial t} = \frac{\partial^2 D(t,s)}{\partial t \partial s}.$$

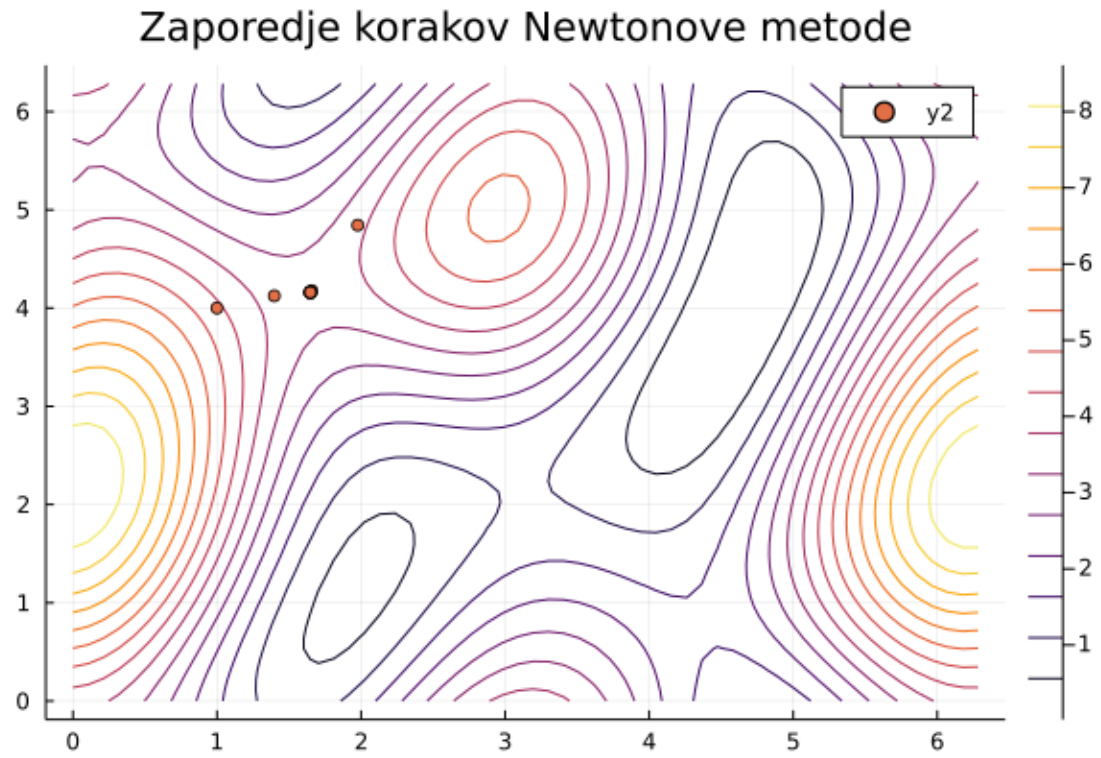
Preskusimo sedaj Newtonovo metodo za naš primer dveh elips. Funkcija, katere ničle iščemo je kar enaka gradientu. Napisati moramo še funkcijo, ki vrne Jacobijevo matriko.

Avtomatsko odvajanje

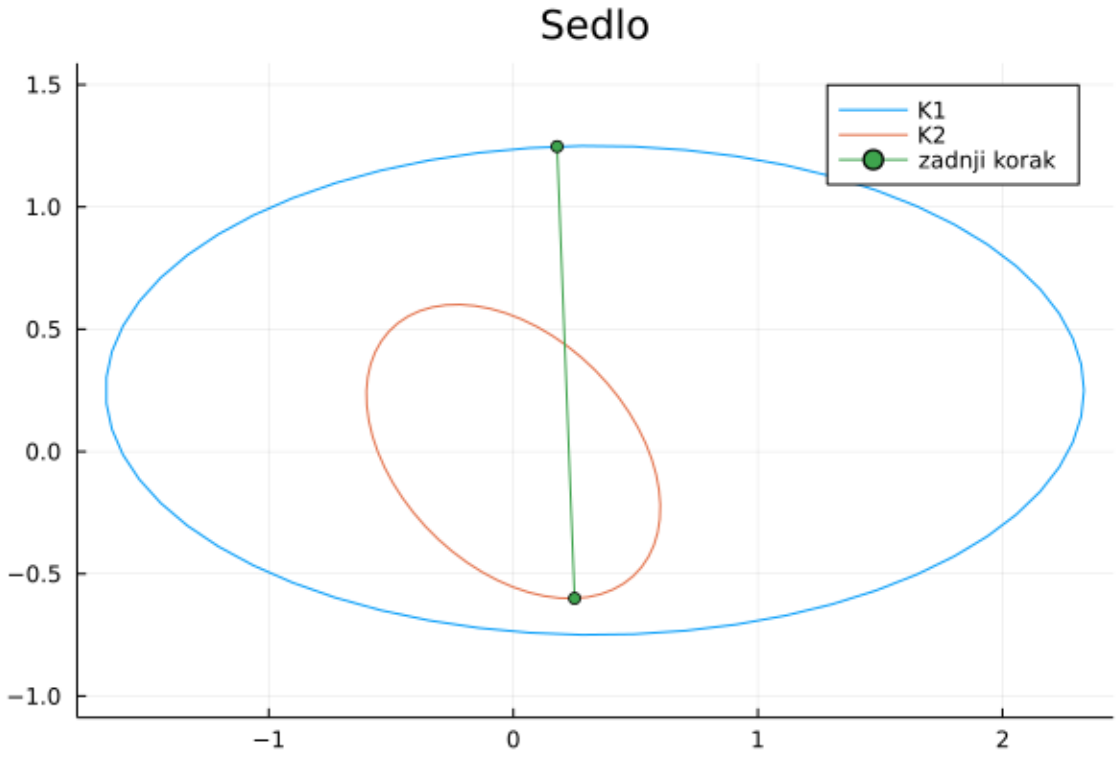
Iskanje odvodov postane hitro zamudno in mimogrede se pri računanju človek zmoti. Na srečo si lahko pri tem si lahko pomagamo z avtomatskim odvajanjem, ki programsko iz programa za funkcijo generira program za odvod. Julija ima več knjižnic za ta namen, mi bomo uporabili ForwardDiff.jl.

```
using ForwardDiff
D2((t,s)) = sum((K1(t)-K2(s)).^2)
gradD2(x) = ForwardDiff.gradient(D2, x)
hessD2(x) = ForwardDiff.hessian(D2, x)

slika = contour(t, t, (t, s)-> D2((t,s)), title="Zaporedje korakov Newtonove metode")

x0 = [1., 4.]
priblizki = [tuple(x0...)]

let
```

```
x = x0
for i=1:10
  x = x - hessD2(x)\gradD2(x)
  push!(priblizki, tuple(x...))
end
end # let
scatter!(slika, priblizki)
```




```
plot(tocka.(K1.(t)), label="K1", aspect_ratio=:equal, title="Sedlo")
plot!(tocka.(K2.(t)), label="K2")
t0, s0 = priblizki[end]
plot!(tocka.([K1(t0), K2(s0)]), marker=:circle, label="zadnji korak")
```



Konvergenčno območje

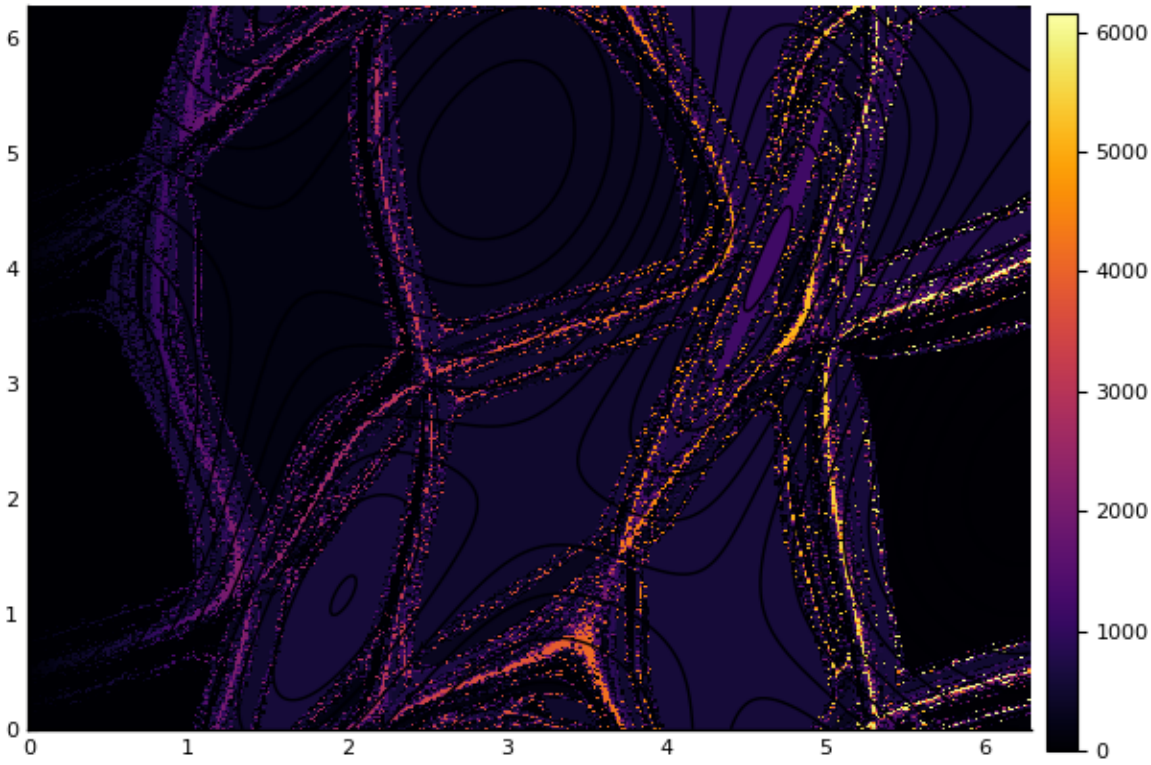
Konvergenca tako gradientne kot tudi newtonove metode je zelo odvisna od začetnega približka.

```
julia> using Plots;
julia> x, y, Z = konvergencno_obmocje((0, 2pi, 0, 2p),
  x -> newton(gradD2, hessD2, x; maxit=20, tol=1e-3), n=300, m=300);
julia> heatmap(x, y, Z')
julia> contour!(x, y, (x,y)->razdalja((x,y)))
julia> maximum(Z)
6157.0
```

Konvergenčno območje newtonove metode je razdeljeno. Iz slike je razvidno, da je veliko različnih približkov (6157), h kateri konvergira newtonova metoda na območju $[0, 2\pi] \times [0, 2\pi]$, čeprav je na tem območju le nekaj rešitev.

```
julia> using Plots;
julia> x, y, Z = konvergencno_obmocje((0, 2pi, 0, 2pi),
       x -> spust(grad, x; maxit=1000, tol=1e-3), tol=1e-1, n=200, m=200)
julia> heatmap(x, y, Z')
julia> contour!(x, y, (x,y)->razdalja((x,y)))
julia> maximum(Z)
8.0
```

Za razliko od newtonove metode, je konvergenčno območje gradientne metode precej bolj predvidljivo.



Slika 17: Konvergenca newtonove metode

Drugi primeri nelinearnih enačb iz 3D geometrije

- presečišče dveh krivulj v 2D
- presečišče krivulje in ploskve v 3D
- presečišče dveh ploskev v 3D

Koda

`NumMat.gradient_razdalje` – Method.

```
fgrad = gradient_razdalje(K1, K2, dK1, dK2)
```

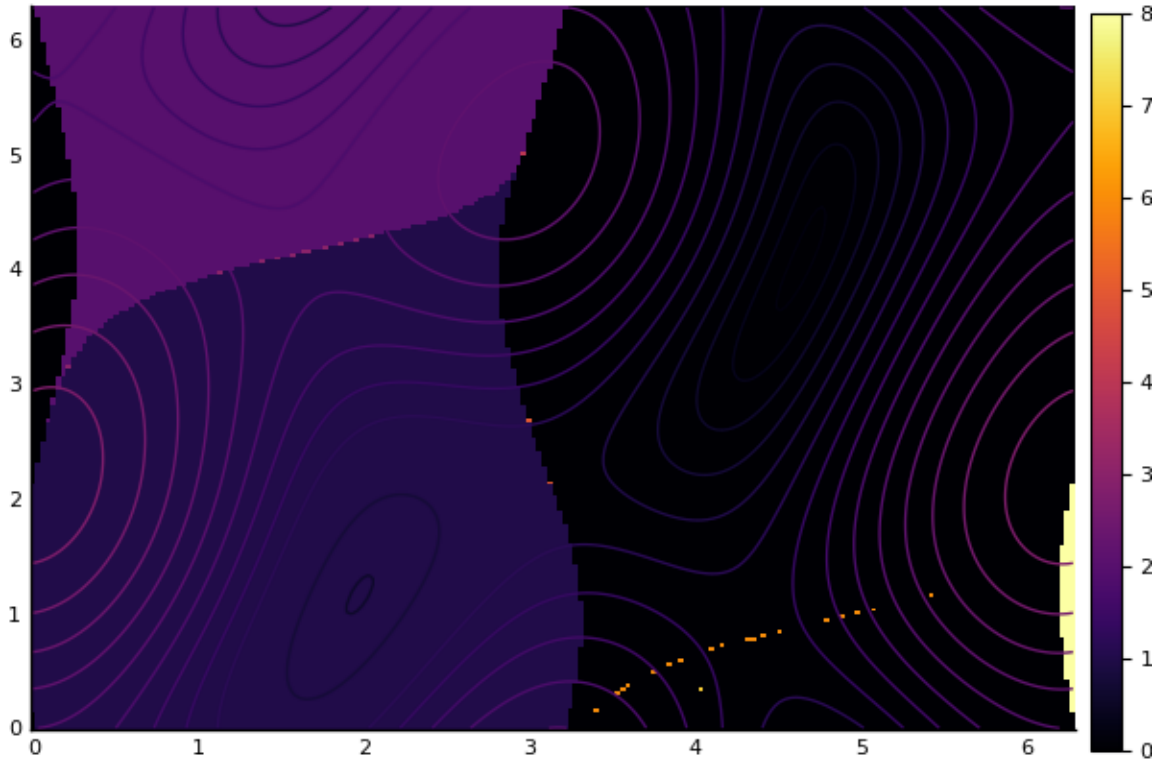
Vrne funkcijo `fgrad`, ki izračuna gradient kvadrata razdalje med dvema točkama na krivuljah K1 in K2. Argumenti K1, K2, dK1 in dK2 so funkcije parametra, ki opisujejo krivulji in njuna smerna odvoda.

Funkcija uporabi `ForwardDiff.gradient` za izračun gradienta.

`NumMat.hessian_razdalje` – Method.

```
fhess = hessian_razdalje(K1, K2, dK1, dK2, d2K1, d2K2)
```

Vrne funkcijo `fhess`, ki izračuna hessian kvadrata razdalje med dvema točkama na krivuljah K1 in K2. Argumenti K1, K2, dK1, dK2, d2K1, d2K2 so funkcije parametra, ki opisujejo krivulji, njuna smerna odvoda in njuna druga odvoda po parametru.



Slika 18: Konvergenca gradientne metode

Funkcija uporabi `ForwardDiff.hessian` za izračun hessove matrike.

`NumMat.razdalja` – Method.

```
d = razdalja(K1, K2)
```

Vrne funkcijo razdalje

$$d(t,s) = ||K1(t) - K2(s)||$$

med dvema točkama na krivuljama `K1(t)` in `K2(s)`.

Primer

```
julia> K1(t) = [2*cos(t)+1/3, (sin(t))+1/4];
julia> K2(s) = [cos(s)/3-sin(s)/2, cos(s)/3+sin(s)/3];
julia> razdalja(K1, K2)
#15 (generic function with 1 method)
```

Interpolacija, aproksimacija

Interpolacija z zlepki

Interpolacija s polinomi

Poglejmo si najprej primer interpolacije s polinomom. Interpolirali bomo funkcijo e^{2x} v tokah $-1, 0, 1, 2$ z newtonovim interpolacijskim polinomom. Nato napako ocenimo z oceno

$$f(x) - p_n(x) = \frac{f^{n+1}(\xi)}{(n+1)!}\Pi_{k=1}^{n}(x - x_i)$$

in oceno primerjamo z numerično napako.

Newtonova interpolacija

Naj bodo $x_1, \dots x_{n+1}$ vrednosti neodvisne spremenljivke in $y_1, \dots y_{n+1}$ vrednosti neznane funkcije. Interpolacijski polinom je polinom, ki zadošča enačbam

$$p(x_1) = y_1, p(x_2) = y_2, \dots p(x_{n+1}) = y_{n+1}$$

Newtonov interpolacijski je interpolacijski polinom

$$p(x) = a_0 + a_1(x - x_1) + a_2(x - x_1)(x - x_2) + \dots a_n(x - x_1) \dots (x - x_{n-1}),$$

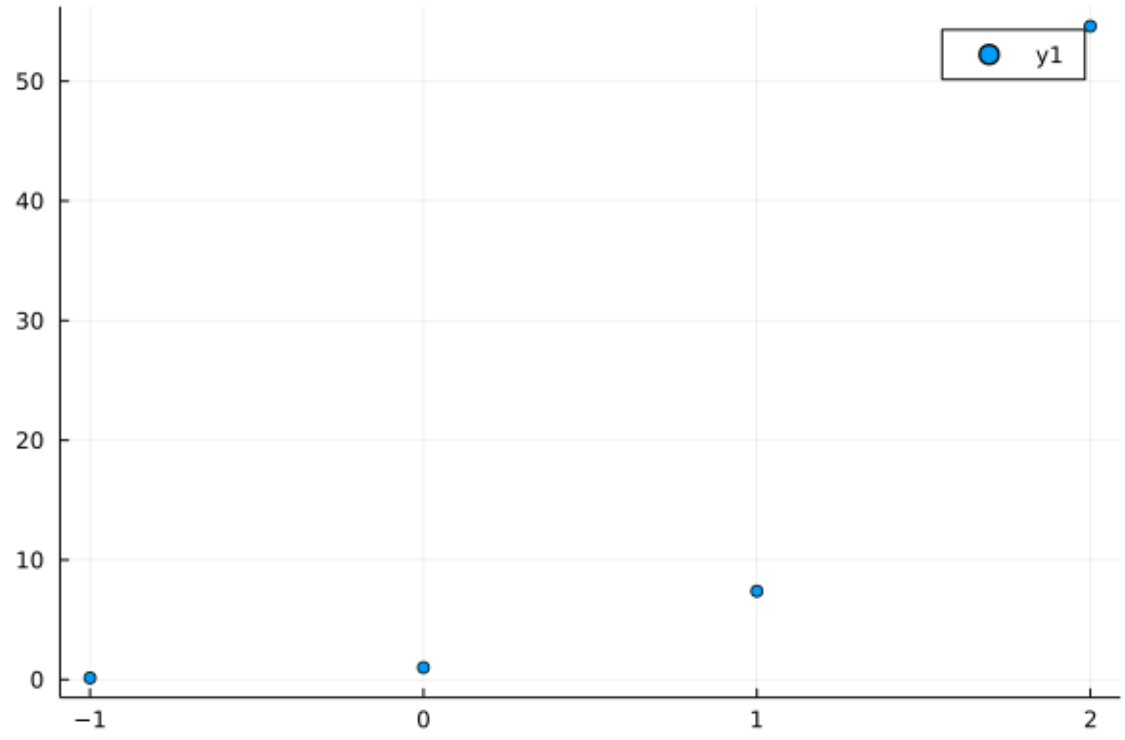
ki je zapisan v bazi

$$1, x - x_1, (x - x_1)(x - x_2), \dots (x - x_1) \dots (x - x_{n-1}).$$

Koeficiente a_i je tako lažje izračunati, kot če bi bil polinom zapisan v standardni bazi. Poleg tega je računanje vrednosti polinoma v standardni bazi lahko numerično nestabilno, če so vrednosti x_i relativno skupaj.

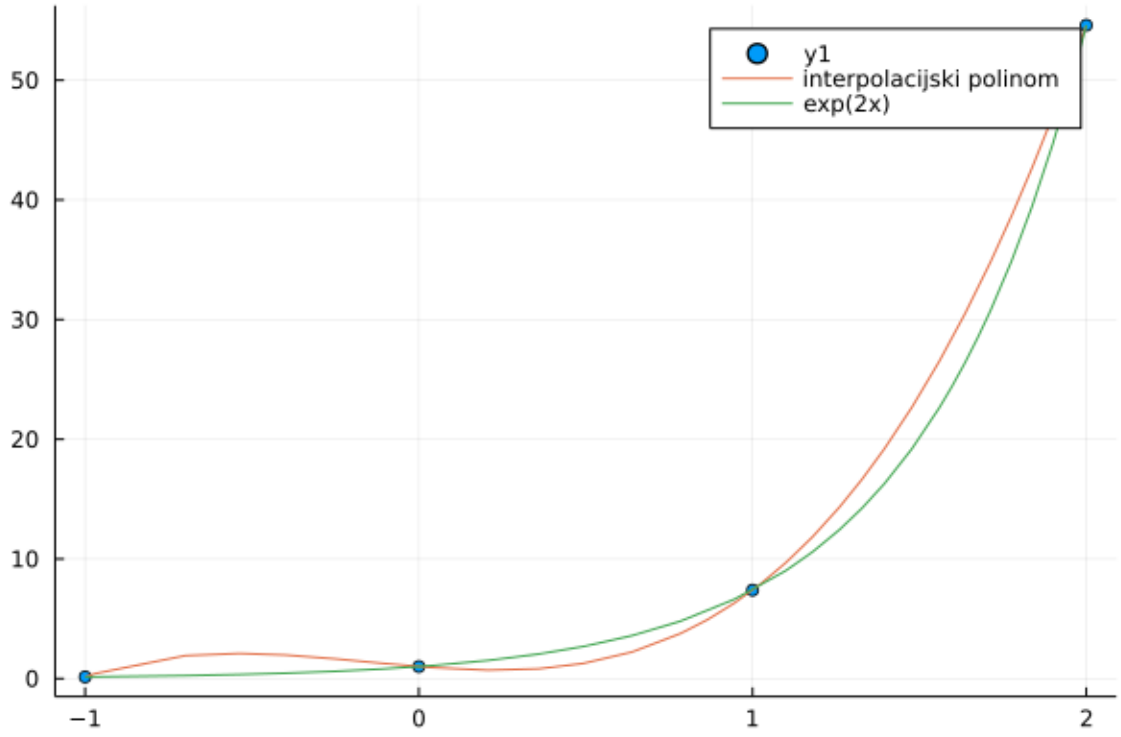
Koeficiente a_i lahko poiščemo bodisi tako, da rešimo spodnje trikotni sistem, ki ga dobimo iz enačb \eqref{eq:enacbe} ali pa z deljenimi diferencami

```
using Plots
x = [-1, 0, 1, 2]
y = exp.(2x)
slika = scatter(x, y)
```



Ustvarimo tabelo deljenih diferenc in zapišemo Newtonov interpolacijski polinom

```
using NumMat
p = deljene_diference(x,y)
plot!(slika, p, -1, 2, label="interpolacijski polinom")
plot!(slika, x->exp(2x), -1, 2, label="exp(2x)")
```

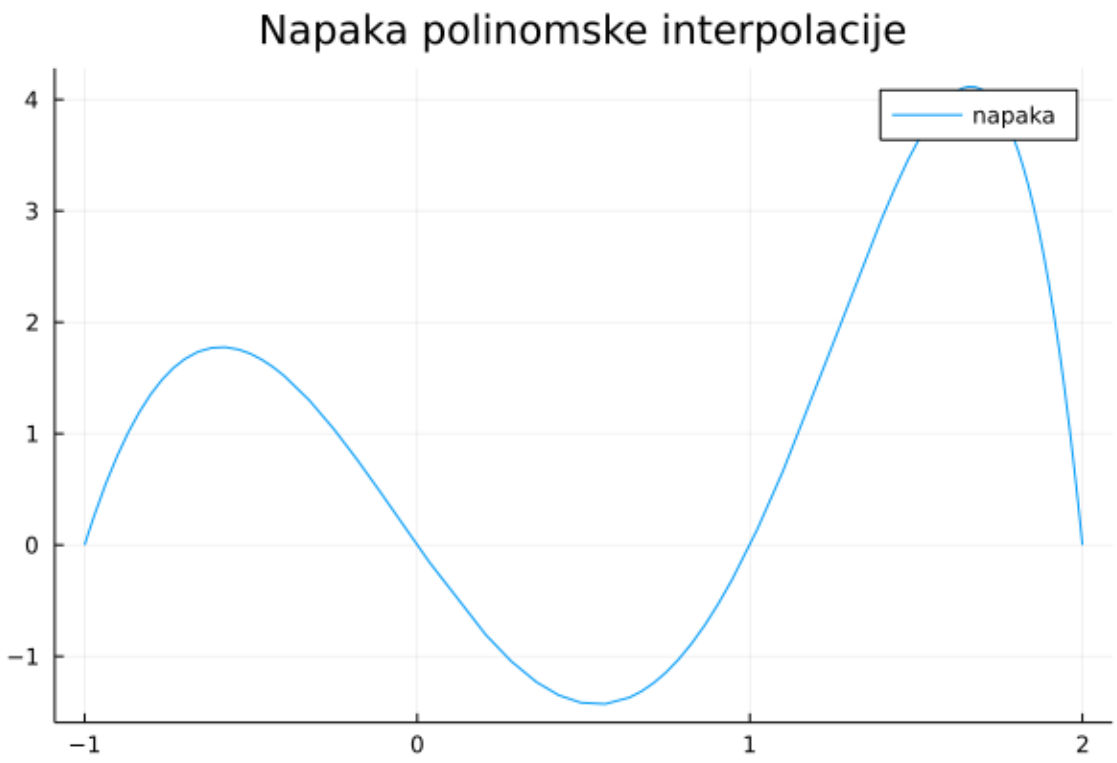


Razlika med funkcijo in interpolacijskim polinomom lahko ocenimo s formulo \eqref{eq:napaka}, če upoštevamo, da je $f^{(4)} = 16e^{2x} \leq 16e^4 \simeq 873$. Potem je

$$|p(x) - f(x)| \leq 36.4(x+1)x(x-1)(x-2) \leq 36.4$$

medtem ko dejansko napako najlepše prikažemo grafično

```
plot(x->p(x)-exp(2x), -1, 2, label="napaka", title="Napaka polinomske interpolacije")
```



Ocena, ki smo jo dobili s formulo \eqref{eq:napaka} je tako precej pesimistična, čeprav je vsaj red velikosti približno pravilen.

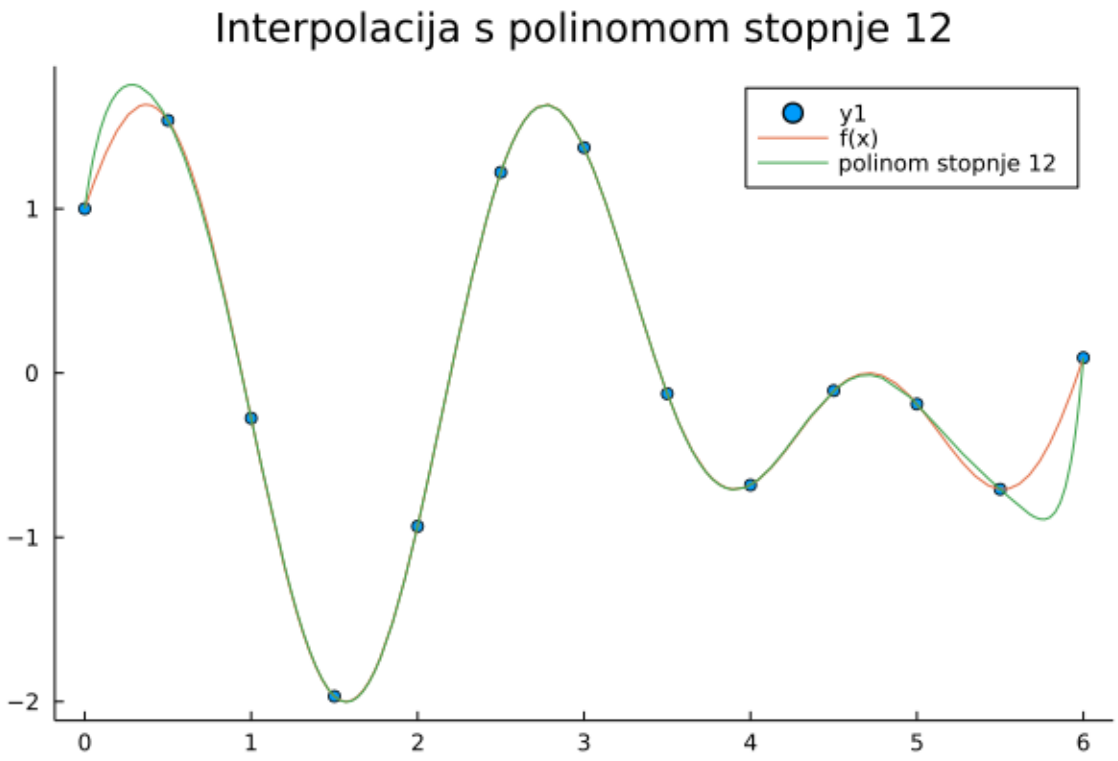
> **Tipi in večlična dodelitev omogoča izražanje podobno kot v matematičnemu jeziku**
>
> Na primeru Newtonovega polinoma lahko ilustriramo, kako izkoristimo tipe in večlično dodelitev, da lahko transparentno definiramo operacije, ki so v matematiki naravne, v programskih jeziki pa pogosto njihova uporaba ni več tako intuitivna. Za Newtonov interpolacijski polinom smo tako definirali tip `NewtonovPolinom` in metodo `polyval`, s katero lahko izračunamo vrednost Newtonovega interpolacijskega polinoma v dani točki. Lahko gremo še dlje in definiramo, da se vrednosti tipa `NewtonovPolinom` obnašajo kot funkcije
>
> ```
> (p::NewtonovPolinom)(x) = polyval(p, x)
> ```
>
> tako da lahko vrednosti polinoma dobimo tako, da kličemo polinom sam, kot bi bil funkcija
>
> ```
> p = NewtonovPolinom([1,0,0],[1, 2, 3])
> p(1.23)
> ```

Rungejev pojav

Pri nizkih stopnjah polinomov se napaka interpolacije zmanjšuje, če povečamo število interpolacijskih točk. A le do neke mere, če stopnja polinoma preveč povečamo, začne napaka spet naraščati. Interpolirajmo funkcijo $\sin(3x) + \cos(2x)$ v točkah $x_i = \frac{i-1}{2}$, za $i = 1 \ldots 13$.

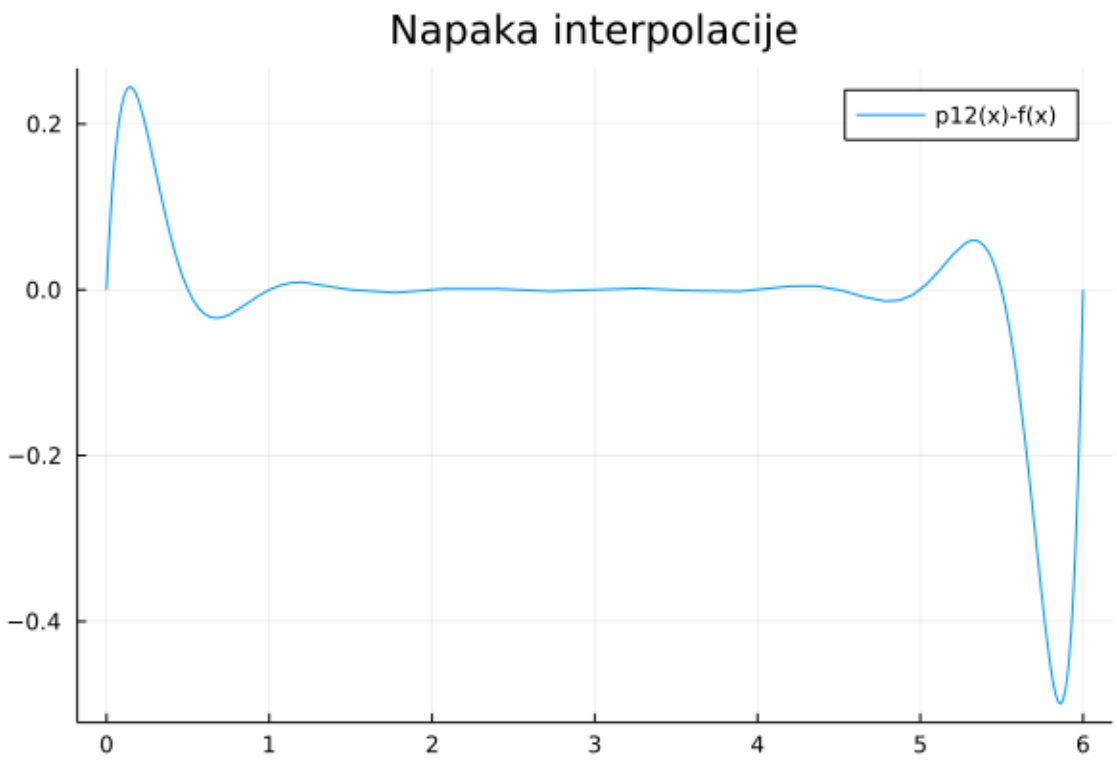
```
using Plots
using NumMat
```

```
x = 0:0.5:6
f(x) = sin(3x)+cos(2x)
scatter(x, f.(x), title="Interpolacija s polinomom stopnje 12", grid=false)
plot!(f, 0, 6; label="f(x)")
p = deljene_diference(Array(x), f.(x))
plot!(p, 0, 6; label="polinom stopnje 12")
```



Večja napaka se v tem primeru pojavi blizu roba interpolacijskega intervala

```
plot(x->p(x)-f(x), 0, 6; title="Napaka interpolacije", label="p12(x)-f(x)")
```



Interpolacija s polinomi visokih stopenj je lahko problematična

V prejšnjem primeru opazimo, da se napaka na robu znatno poveča. Povečanje je posledica velikih oscilacij, ki se pojavijo na robu, če interpoliramo s polinomom **visoke stopnje** na **ekvidistančnih točkah**. To je znan pojav pod imenom Rungejev pojav in se pojavi, če interpoliramo s polinomom visoke stopnje v ekvidistančnih točkah. Problemu se lahko izognemo, če namesto ekvidistančnih točk uporabimo Čebiševe točke.

Zlepki

Zakaj zlepki

Pogosto je bolje uporabiti različne definicije funkcije na različnih intervalih, kot eno funkcijo z veliko parametri. Funkcijam, ki so sestavljene iz več različnih definicij pravimo **zlepki**. Če interpoliramo z zlepkom, se izognemo Rungejevemu pojavu in ne dobimo velikih oscilacij na robu.

Linearen zlepek

Interpoliraj funkcijo sin z linearnim zlepkom. Numerično oceni napako.

Hermitovi zlepki

Za točke $x_1, \ldots x_n$ imamo podane vrednosti funkcije $f_1, \ldots f_n$ in vrednosti odvoda $df_1, \ldots df_n$. Poiskati želimo gladek zlepek sestavljen iz polinomov $S_i(x)$ na intervalu $[x_i, x_{i+1}]$, ki interpolira dane podatke. To pomeni, da so izpolnjene naslednje enačbe

$$S_i(x_i) = f_i \quad S_i(x_{i+1}) = f_{i+1}$$
$$S_i'(x_i) = df_i \quad S_i'(x_{i+1}) = df_{i+1}$$

Hermitov zlepek je vedno zvezen in zvezno odvedljiv, ker se vrednosti funkcije in vrednosti odvoda predpisane v točkah, kjer se dva predpisa dotikata in se zato avtomatično ujemajo.

Newtonova interpolacija vrednosti in odvodov

Če poleg funkcijskih vrednosti podamo še odvode, lahko še vedno uporabimo deljene diference za izračun Newtonovega interpolacijskega polinoma. Pri tem vrednosti neodvisne spremenljivke x_i, v katerih je podan tudi odvod, v tabeli deljenih diferenc napišemo večkrat (točko ponovimo tolikokrat, kolikor je podanih višjih odvodov). Poglejmo si primer. Interpolirajmo podatke $x_1 = 0, x_2 = 1, f(x_1) = 0, f(x_2) = 0$ in $f'(x_1) = 1, f''(x_1) = -4$. V tabelo deljenih diferenc zapišemo $0, 0, 0, 1$ za vrednosti x in $0, 1, --4, 0$ za vrednosti f.

x	f	$f[\cdot,\cdot]$	$f[\cdot,\cdot,\cdot]$	$f[\cdot,\cdot,\cdot,\cdot]$
0	0	1	$\frac{-4}{2!}$	1
0	1	1	-1	
0	-4	0		
1	0			

Hermitove zlepke sedaj lahko preprosto poiščemo tako da na vsakem intervalu uporabimo deljene diference za vrednosti in odvode v krajščih (enačbe \eqref{eq:hermite}).

```
zlepek = hermitov_zlepek([0, 1, 2, 4], [0, 1, 2, 0], [0, -1, 0, 1])
plot(zlepek, title="Heremitov zlepek")
```

Laplaceovi $\mathcal{C}^1$ zlepki

Za točke $x_1, \dots x_n$ so podane vrednosti funkcije $f_1, \dots f_n$. Predpostavimo, da je n liho število. Poiskati želimo zlepek sestavljen iz kubičnih polinomov $S_i(x)$ na intervalu $[x_{2i-1}, x_{2i+1}]$, ki interpolira dane podatke. Poleg tega zahtevamo, da se v točkah, v katerih se predpisi stikajo ujemajo odvodi, da dobimo zvezno odvedljiv zlepek.

Število parametrov naj bo enako številu zahtev

Pri interpolaciji s polinomi lahko zadostimo toliko zahtevanim pogojem, kolikor ima polinom koeficientov oz kolikor je dimenzija prostora poplinomov. Za polinome 3. stopnje je dimenzija 4, kar pomeni, da lahko in moramo predpisati 4 zahteve, bodisi za vrednosti polinoma ali vrednosti odvodov v 4 različnih točkah. Na vsakem intevalu $[x_{2i-1}, x_{2i+1}]$ so 3 vrednosti predpisane (v točkah x_{2i-1}, x_{2i}, x_{2i+1}), to pomeni, da lahko dodamo še eno zahtevo, s katero dosežemo zvezno odvedljivost zlepka.

Omenjene zahteve lahko zapišemo z naslednjimi enačbami

$$\begin{array}{lll} S_i(x_{2i-1}) = f_{2i-1} & S_i(x_{2i}) = f_{2i} & S_i(x_{2i+1}) \quad = f_{2i+1} \\ S'_{i-1}(x_{2i-1}) = S'_i(x_{2i-1}) & S'_i(x_{2i+1}) = S'_{i+1}(x_{2i+1}) & \end{array}$$

To je sicer 5 zahtev za vsak polinom S_i, a če preštejemo vse enačbe in koeficiente, vidimo, da manjka le še ena enačba za polinom na robu S_1 ali $S_{(n-1)/2}$. Predpišemo lahko vrednost prvega ali drugega odvoda v eni od robnih točk x_1 ali x_n[1].

Koeficiente polinomov S_i lahko tako računamo zaporedoma tako, da izpolnijo naslednje 4 enačbe

$$\begin{array}{ll} S_i(x_{2i-1}) = f_{2i-1} & S_i(x_{2i}) = f_{2i} \\ S_i(x_{2i+1}) = f_{2i+1} & S'_i(x_{2i-1}) = S'_{i-1}(x_{2i-1}) \end{array}$$

Za prvi polinom S_1 pa zadnjo enačbo nadomestimo z enačbo

$$S''_1(x_1) = 0$$

Običajno se uporabljajo naravni zlepki

Lagrangeovega C^1 zlepka, kot smo ga opisali v tem razdelku običajno ne uporabljamo, saj je lahko za iste podatke sestavimo naravni kubični zlepek, ki ima boljlše lastnosti. Naravni zlepek je dvakrat zvezno odvedljiv in med vsemi funkcijami, ki interpolirajo dane točke ima najmanjšo povprečno ukrivljenost.

Kaj smo se naučili

- deljene diference in Newtonov polinom lahko uporabimo tudi, če so poleg vrednosti podani tudi odvodi
- zaradi Rungejevega pojava interpolacija s polinomi visokih stopenj na ekvidistančnih točkah ni najboljša izbira
- zlepki so enostavni za uporabo, učinkoviti (malo operacij za izračun) in imajo v določenih primerih boljše lastnosti kot polinomi visokih stopenj

- poznamo različne vrste zlepkov glede na zveznost in podatke, ki jih potrebujemo za določitev (linearni zlepek, Hermitov zlepek in Lagrangeov zlepek)

Koda

Julia omogoča, da definirmo metode za transparentno risanje in računanje z zlepki PlotRecipies.jl

- NumMat.NewtonovPolinom
- NumMat.Zlepek
- NumMat.deljene_diference
- NumMat.findinterval
- NumMat.hermitov_zlepek
- NumMat.lagrangev_zlepek
- NumMat.polyder
- NumMat.polyval
- RecipesBase.apply_recipe
- RecipesBase.apply_recipe

NumMat.deljene_diference – Method.

```
p::NewtonovPolinom{T} = deljene_diference(x::Array{T}, f::Array{U})
```

Izračuna koeficiente Newtonovega interpolacijskega polinoma, ki interpolira podatke `y(x[k])=f[k]`. Če se katere vrednosti x večkrat ponovijo, potem metoda predvideva, da so v `f` poleg vrednosti, podani tudi odvodi.

Primer

Polinom $x^2 - 1$ interpolira podatke `x=[0,1,2]` in `y=[-1, 0, 3]` lahko v Newtonovi obliki zapišemo kot $1 + x + x(x - 1)$

```
julia> p = deljene_diference([0, 1, 2], [-1, 0, 3])
NewtonovPolinom{Float64,Int64}([-1.0, 1.0, 1.0], [0, 1])
```

Če imamo več istih vrednosti abscise x, moramo v `f` podati vrednosti funkcije in odvodov. Na primer polinom $p(x) = x^4 = x + 3x(x - 1) + 3x(x - 1)^2 + x(x - 1)^3$ ima v $x = 1$ vrednosti $p(1) = 1, p'(1) = 4, p''(1) = 12$

```
julia> p = deljene_diference([0,1,1,1,2], [0,1,4,12,16])
NewtonovPolinom{Float64,Int64}([0.0, 1.0, 3.0, 3.0, 1.0], [0, 1, 1, 1])

julia> x = (1,2,3,4,5); p.(x).-x.^4
(0.0, 0.0, 0.0, 0.0, 0.0)
```

[1] Najpogosteje zahtevamo, da je drugi odvod v robnih točkah enak 0. Če predpišemo prvi odvod, določimo tangente v robnih točkah, kar je pogosto preveč vpliva na končno obliko zlepka.

NumMat.findinterval – Method.

```
findinterval(x, endpoints)
```

Poišči index intervala na katerem leži x.

NumMat.hermitov_zlepek – Method.

```
z::Zlepek = hermitov_zlepek(x, f, df)
```

Izračuna koeficiente kubičnega C^1 zlepka, ki interpolira podatke $p_i(x_i) = f_i, p_i(x_{i+1}) = f_{i+1}$ in $p_i'(x_i) = df_i, p_i'(x_{i+1}) = df_{i+1}$.

Primer

```
julia> z = hermitov_zlepek([1, 2, 3], [0, 1, 0], [0, 0, 0]);

julia> z.(1:3) - [0, 1, 0]
3-element Array{Float64,1}:
 0.0
 0.0
 0.0

 julia> polyder(z.funkcije[1], 1)
 0
```

NumMat.lagrangev_zlepek – Method.

```
lagrangev_zlepek(x,f)
```

Izračuna koeficiente kubičnega C^1 zlepka, ki interpolira podatke $p_i(x_{2i-1}) = f_{2i-1}, p_i(x_{2i}) = f_{2i}$ in $p_i(x_{2i+1}) = f_{2i+1}$, poleg tega pa je zvezno odvedljiv. Dodatna lasntost zlepka je $p_1''(x_1) = 0$.

Primer

```
julia> z = lagrangev_zlepek([1, 2, 3, 4, 5], [0, 1, 2, 1, 0]);

julia> z.(1:5) - [0, 1, 2, 1, 0]
5-element Array{Float64,1}:
 0.0
 0.0
 0.0
 0.0
 0.0
```

NumMat.polyder – Method.

```
dy = polyder(p::NewtonovPolinom, x)
```

Izračuna vrednost odvoda newtonovega polinoma p v točki x s hornerjevim algoritmom, ki je dopolnjen tako, da računa tudi odvod.

Primer

Vzemimo primer polinoma $1 + x^2 = 1 - x + x(x + 1)$, katerega odvod je $2x$

```
julia> p = NewtonovPolinom([1,-1,1], [0,-1]);

julia> odvodp(x) = polyder(p, x); odvodp.((1, 2, 3))
(2, 4, 6)
```

NumMat.polyval – Method.

```
polyval(p::NewtonovPolinom, x)
```

Izračuna vrednot newtonovega polinoma p v x s Hornerjevo methodo. Objekte tipe `NewtonovPolinom` lahko kličemo kot funkcijo, ki pokliče to funkcijo.

Primer

```
julia> p = NewtonovPolinom([0, 0, 1], [0, 1]);

julia> polyval(p, 2)
2

julia> p(2)
2
```

RecipesBase.apply_recipe – Method.

Recept za risanje grafa zlepka

RecipesBase.apply_recipe – Method.

Recept za risanje Newtonovega polinoma

NumMat.NewtonovPolinom – Type.

```
NewtonovPolinom{T}(a::Vector{T}, x::Vector{T})
```

Vrne newtonov interpolacijski polinom oblike `a[1]+a[2](x-x[1])+a[3](x-x[1])(x-x[2])+...` s koeficienti a in vozlišči x.

Primer

Poglejmo si polinom $1 + x + x(x - 1)$, ki je definiran s koeficienti `[1,1,1]` in z vozlišči $x_0 = 0$ in $x_1 = 1$

```
julia> p = NewtonovPolinom([1, 1, 1], [0, 1])
NewtonovPolinom{Int64}([1, 1, 1], [0, 1])

julia> p.([1,2,3])
3-element Array{Int64,1}:
  2
  5
 10
```

NumMat.Zlepek – Type.

```
Zlepek(funkcije::Vector{F}, vozlisca::Vector{T})
```

Vrednost tipa Zlepek predstavlja funkcijo podano kot zlepek različnih predpisov na različnih intervalih

$$f(x) = \begin{cases} f_1(x); x \in [x_1, x_2] \\ f_2(x); x \in (x_2, x_3] \\ \vdots \end{cases}$$

Krajišča intervalov so podana v seznamu `vozlisca`, medtem ko so predpisi zbrani v seznamu `funkcije`.

Primer

```
julia> z = Zlepek([x->x, x->2-x], [0, 1, 2]);

julia> z(0.5), z(1.5)
(0.5, 0.5)
```

Aproksimacija z linearnim modelom

Linearni model

V znanosti pogosto želimo opisati odvisnost dveh količin npr. kako se spreminja koncentracija CO_2 v odvisnosti od časa. Matematičnemu opisu povezave med dvema ali več količinami pravimo **matematični model**. Primer modela je Hookov zakon za vzmet, ki pravi, da je sila vzmeti F sorazmerna z raztezkom x: $F = kx$ Model povezuje dve količini silo F in raztezek x. Poleg tega Hookov zakon vpelje še koeficient vzmeti k. Koeficientu k pravimo **parameter modela** in ga lahko določimo za vsako vzmet posebej z meritvami sile in raztezka.

Najpreporstejši je **linearni model**, pri katerem odvisno količino y zapišemo kot linearno kombinacijo baznih funkcij $\phi_j(x)$ neodvisne spremenljivke x:

$$y(x) = M(p, x) = p_1\phi_1(x) + p_2\phi_2(x) + \ldots + p_k\phi_k(x).$$

Koeficientom p_j pravimo parametri modela in jih določimo na podlagi meritev. Znanstveniki hočejo model, pri katerem imajo parametri p_i preprosto interpretacijo in pomagajo pri razumevanju pojava, ki ga opisujejo. Zato so bazne funkcije pogosto elementarne funkcije, pri katerih je jasno razvidna narava odvisnosti.

Metoda najmanjših kvadratov

Koeficiente modela, ki najbolje opisujejo izmerjene podatke lahko poiščemo z metodo najmanjših kvadratov. Napišemo najprej pogoje, ki bi jim zadoščali parametri, če bi izmerjeni podatki povsem sledili modelu. Za vsako meritev $y_i = y(x_i)$ bi bila vrednost odvisne količine y_i natanko enaka vrednosti, ki jo predvidi model $M(p, x_i)$. To predpostavko lahko zapišemo s sistemom enačb

$$y_i = M(p, x_i) = p_1\phi_1(x_i) + \ldots p_k\phi_k(x_i)$$

Neznanke v zgornjem sistemu so parametri p_j in za **linearni model** so enačbe linearne. To je tudi ena glavnih prednosti linearnega modela. Meritve redko povsem sledijo modelu, zato sistem \eqref{eq:sistem} v splošnem ni rešljiv, saj je meritev običajno več kot je parametrov sistema. Sistem \eqref{eq:sistem} je **predoločen**. Lahko pa poiščemo vrednosti parametrov p_j pri katerih bo razlika med meritvami in modelom kar se da majhna. Izkaže se, da je najboljša mera za odstopanje modela od podatkov kar vsota kvadratov razlik med meritvami in napovedjo modela:

$$(y_1 - M(p, x_1))^2 + \ldots + (y_n - M(p, x_n))^2 = \sum_{i=1}^{n}(y_i + M(p, x_i))^2$$

Sistem (\ref{eq:sistem}) lahko zapišemo v matrični obliki $A\mathbf{p} = \mathbf{y}$, kjer so stolpci matrike sistema enaki vrednostim baznih funkcij

$$A = \begin{bmatrix} \phi_1(x_1) & \phi_2(x_1) & \dots & \phi_k(x_1) \\ \phi_1(x_2) & \phi_2(x_2) & \dots & \phi_k(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_1(x_n) & \phi_2(x_n) & \dots & \phi_k(x_n) \end{bmatrix}$$

in stolpec desnih strani je enak meritvam

$$\mathbf{y} = [y_1, y_2, \dots, y_n]^\mathsf{T}.$$

Pogoj najmanjših kvadratov razlik (\ref{eq:minkvad}) za optimalne vrednosti parametrov $\mathbf{p}_{opt}$ lahko sedaj zapišemo s kvadratno vektorsko normo

$$\mathbf{p}_{opt} = \operatorname{argmin}_{\mathbf{p}} \|A\mathbf{p} - \mathbf{y}\|_2^2 .$$

Opis sprememb koncentracije CO2

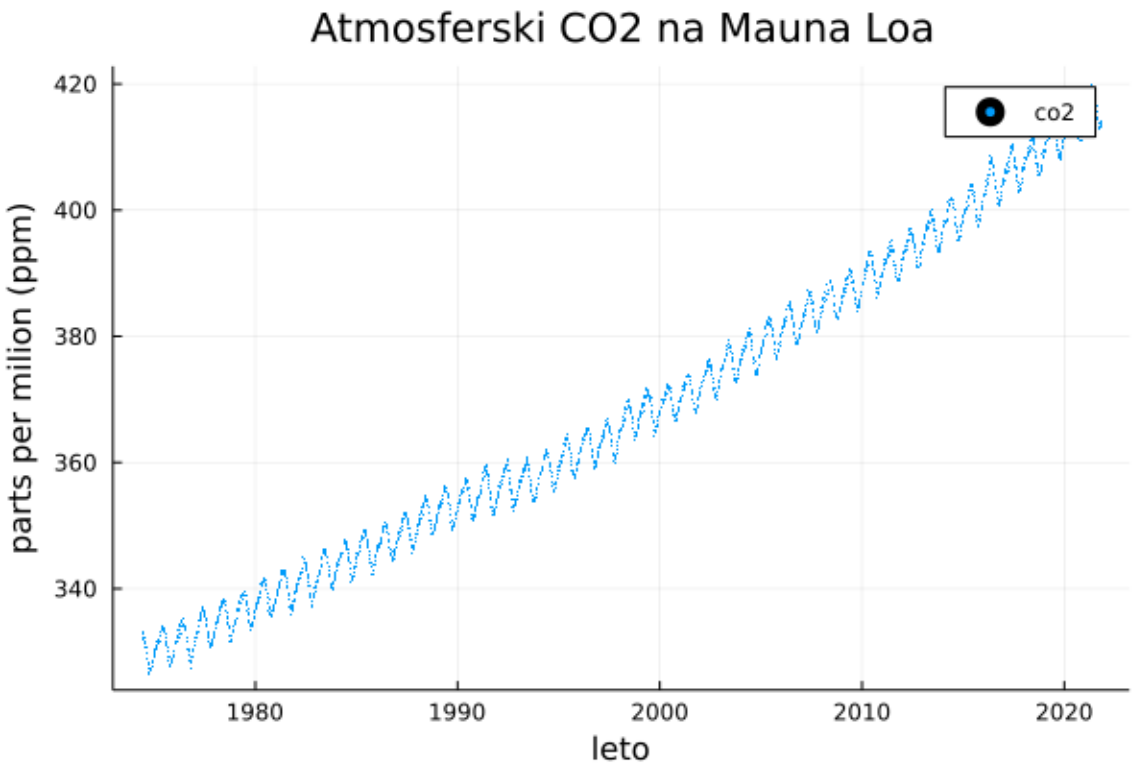
Na observatoriju Mauna Loa na Hawaiih že leta spremljajo koncentracijo CO_2 v ozračju. Podatke lahko dobimo na njihovi spletni strani v različnih oblikah. Oglejmo si tedenska povprečja od začetka maritev leta 1974

```
using FTPClient
url = "ftp://aftp.cmdl.noaa.gov/products/trends/co2/co2_weekly_mlo.txt"
io = download(url)
data = readlines(io)
```

```
2524-element Vector{String}:
 "# ----------------------------------------------------------------------"
 "# USE OF NOAA GML DATA"
 "# "
 "# These data are made freely available to the public and the"
 "# scientific community in the belief that their wide dissemination"
 "# will lead to greater understanding and new scientific insights."
 "# The availability of these data does not constitute publication"
 "# of the data.  NOAA relies on the ethics and integrity of the user to"
 "# ensure that GML receives fair credit for their work.  If the data "
 "# are obtained for potential use in a publication or presentation, "
 □
 "  2021   8  22  2021.6397    413.90  6          412.28    389.49    136.58"
 "  2021   8  29  2021.6589    413.22  4          411.90    389.78    136.32"
 "  2021   9   5  2021.6781    413.36  7          411.68    389.05    136.79"
 "  2021   9  12  2021.6973    413.09  6          411.65    389.38    136.75"
 "  2021   9  19  2021.7164    413.30  7          411.27    389.16    137.09"
 "  2021   9  26  2021.7356    413.37  6          411.26    388.93    137.18"
 "  2021  10   3  2021.7548    413.63  7          411.29    388.93    137.36"
 "  2021  10  10  2021.7740    413.89  7          411.33    389.09    137.43"
 "  2021  10  17  2021.7932    414.36  7          411.70    389.25    137.64"
```

Nato odstranimo komentarje in izluščimo podatke

```
using Plots
filter!(l->l[1]!='#', data)
data = strip.(data)
data = [split(line, r"\s+") for line in data]
data = [[parse(Float64, x) for x in line] for line in data]
filter!(l->l[5]>0, data)
t = [l[4] for l in data]
co2 = [l[5] for l in data]
scatter(t, co2, title="Atmosferski CO2 na Mauna Loa",
        xlabel="leto", ylabel="parts per milion (ppm)", label="co2",
        markersize=1)
```



Časovni potek koncentracije CO_2 matematično opišemo kot funkcijo koncentracije v odvisnosti od časa

$$y = \mathrm{CO}_2(t).$$

Model, ki dobro opisuje spremembe CO_2 lahko sestavimo iz kvadratne funcije, ki opisuje naraščanje letnih povprečij in periodičnega dela, ki opiše nihanja med letom:

$$\mathrm{CO}_2(t) = p_1 + p_2 t + p_3 t^2 + p_4 \sin(2\pi t) + p_5 \cos(2\pi t).$$

Čas t naj bo podan v letih. Predoločeni sistem (\ref{eq:sistem}), ki ga dobimo za naš model ima $n \times 5$ matriko sistema

$$A = \begin{bmatrix} 1 & t_1 & t_1^2 & \sin(2\pi t_1) & \cos(2\pi t_1) \\ 1 & t_2 & t_2^2 & \sin(2\pi t_2) & \cos(2\pi t_2) \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & t_n & t_n^2 & \sin(2\pi t_n) & \cos(2\pi t_n) \end{bmatrix}$$

desne strani pa so vrednosti koncentracij.

Normalni sistem

Po metodi najmanjših kvadratov iščemo vrednosti parametrov p modela, pri katerih bo vsota kvadratov razlik med napovedjo modela in izmerjenimi vrednostmi najmanjša. Zapišimo vsoto kvadratov kot evklidsko normo razlike med vektorjem napovedi modela Ap in vektorjem izmerjenih vrednosti y. Iščemo torej vektor parametrov p, pri katerem bo

$$\|Ap - y\|^2$$

najmanjša. Iščemo torej pravokotno projekcijo vektorja y na stolpčni prostor matrike A, katere stolpci so podani kot vrednosti baznih funkcij, ki nastopajo v modelu.

$$\begin{aligned} Ap - y &\perp C(A) \\ A^T(Ap - y) &= 0 \\ A^T Ap &= A^T y \end{aligned}$$

Tako dobimo normalni sistem $A^T Ap = A^T y$, ki je kvadraten in je vedno rešljiv, če so le bazne funkcije modela linearno neodvisne (izračunane v izmerjenih vrednostih neodvisne spremenljivke).

```
using LinearAlgebra
A = hcat(ones(size(t)), t, t.^2, cos.(2pi*t), sin.(2pi*t))
norm(A*p-co2)
```

```
52.64993956141902
```

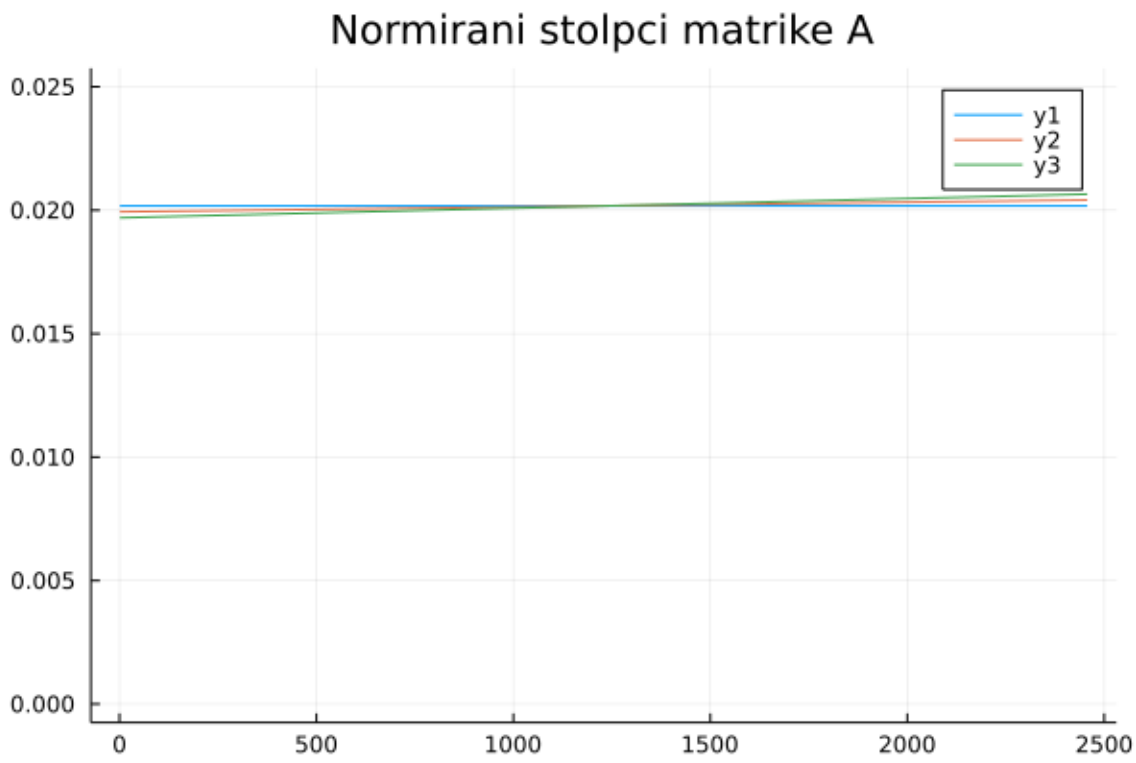
Problem normalnega sistema je, da je zelo občutljiv

```
cond(N), cond(A)
```

```
(2.4039638658029684e22, 9.515612999176315e10)
```

Pravzaprav je že sama matrika A zelo občutljiva. Razlog je v izbiri baznih funkcij. Če narišemo normirane stolpce A kot funkcije, vidimo, da so zelo podobni.

```
plot([A[:,1]/norm(A[:,1]), A[:,2]/norm(A[:,2]), A[:,3]/norm(A[:,3])], ylims=[0,0.025],
↪  title="Normirani stolpci matrike A")
```



Težavo lahko omilimo tako, da časa ne štejemo od začetka našega štetja, pač pa od sredine podatkov.

```
cond(A)
```

```
376.06750481798775
```

Matrika A je sedaj precej dlje od singularne matrike in posledično je tudi normalni sistem manj občutljiv.

QR razcep

Normalni sistem se redko uporablja v praksi. Standarden postopek za iskanje rešitve predoločenega sistema z metodo najmanjših kvadratov je s QR razcepom. Če je $QR = A$ QR razcep matrike A, so stolpci matrike Q ortonormirana baza stolpčnega prostora matrike A, matrika R vsebuje koeficiente v razvoju stolpcev matrike A po ortonormirani bazi določeni s Q. Projekcija na stolpčni prostor ortogonalne matrike še lažje izračunamo, saj lahko koeficiente izračunamo s skalarnim produktom s stolpci Q. Matrično skalarni produkt s stolpci matrike pomeni množenje z transponirano matriko.

$$A = QRRp = Q^T y$$

```
norm(A*p-co2)
```

```
52.64993956140208
```

Na isti način deluje tudi vgrajen operator \, če je matrika dimenzij $n \times k$ in $k < n$.

```
p = A\co2
norm(A*p-co2)
```

```
52.64993956140205
```

Kaj pa CO2?

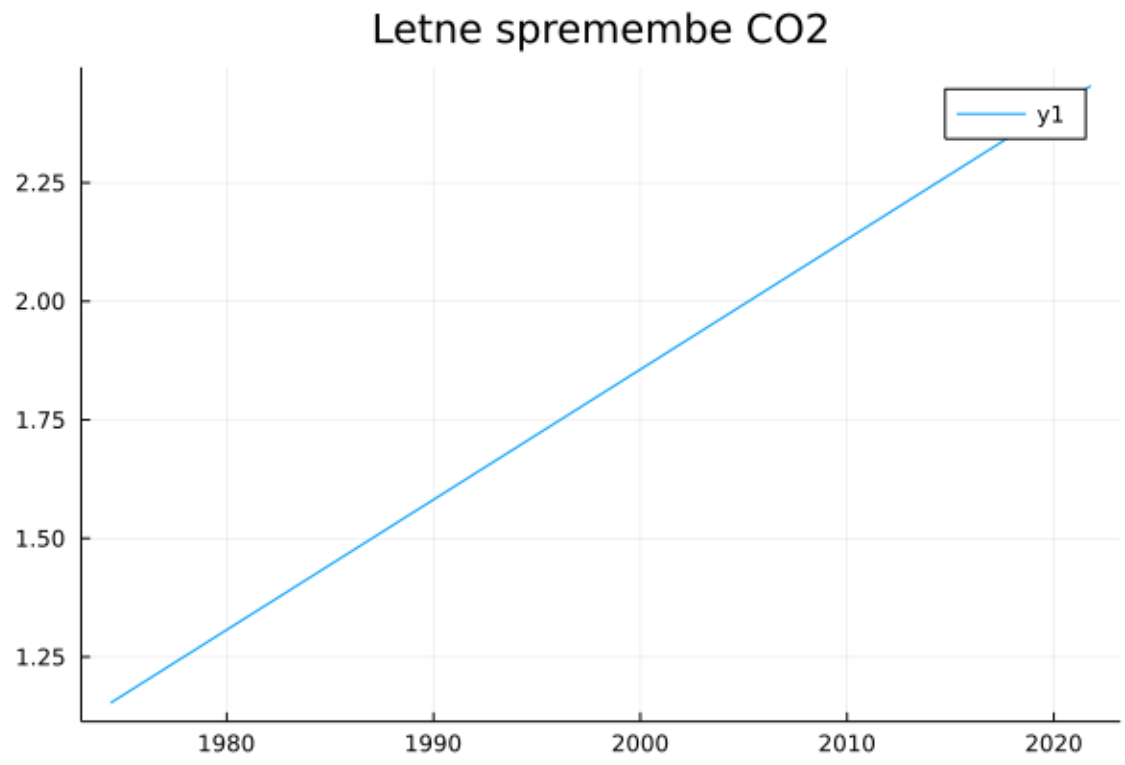
Koncentracije CO2 se vztrajno povečuje. Kot kaže naš model, je naraščanje kvadratično in ne le linearno. To pomeni, da ne le, da se vsako leto poveča koncentracija, pač pa se vsako leto poveča za večjo vrednost.

```
p
```

```
5-element Vector{Float64}:
 366.1197449700098
   1.8060746710199262
   0.013725678086865721
  -0.8049018934050689
   2.8563108111965567
```

Koeficient p_1 pove povprečno koncentracijo na sredini merilnega obdobja. Medtem ko odvod $p_2 + 2p_3(t - \tau)$ pove za koliko se v povprečju spremeni koncentracija v enem letu.

```
plot(t, p[2].+2*p[3]*(t.-τ), title="Letne spremembe CO2")
```



Lahko poskusimo tudi napovedati prihodnost:

```
model(t) =  p[1]+ p[2]*(t-τ) + p[3]*(t-τ)^2 + p[4]*cos(2*pi*t) + p[5]*sin(2*pi*t)
model.([2020, 2030, 2050])
```

```
3-element Vector{Float64}:
 411.2768374999925
 436.70214709843015
 495.7881731474146
```

Kaj smo se naučili

- Linearni model je opis, pri katerem **parametri nastopajo linearno**
- Parametre modela poiščemo z **metodo najmanjših kvadratov**
- Za iskanje parametrov po metodi najmanjših kvadratov je numerično najbolj primeren **QR-razcep**.
- koncentracija CO2 prav zares narašča

Aproksimacija s polinomi Čebiševa

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

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

kjer so T_n polinomi Čebiševa. Polinomi Čebiševa so definirani z relacijo

$$T_n(\cos(\phi)) = \cos(n\phi)$$

in zadoščajo dvočlenski rekurzivni enačbi

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

Namesto cele vrste, lahko obdržimo le prvih nekaj členov in funkcijo **aproksimiramo** s funkcijo oblike

$$f(x) \sim \sum_{n=0}^{N} a_n T_n(x).$$

Iščemo torej koeficiente funkcije $f(x)$ v razvoju po T_n.

$$a_k = \frac{2}{\pi} \int_{-1}^{1} \frac{f(x)T_k(x)}{\sqrt{1-x^2}} dx,$$

kjer za $k = 0$ faktor $\frac{2}{\pi}$ zamenjamo z $\frac{1}{\pi}$. Koeficiente lahko približno izračunamo z gaussovimi kvadraturnimi formulami.

Koeficiente Čebiševe vrste natančneje računamo s FFT

Na vajah bomo koeficiente a_n računali približno z gaussovimi kvadraturnimi formulami. V praksi je mogoče koeficiente a_k izračunati bolj natančno z diskretno Fourierovo kosinusno transformacijo funkcijskih vrednosti v Čebiševih interpolacijskih točkah.

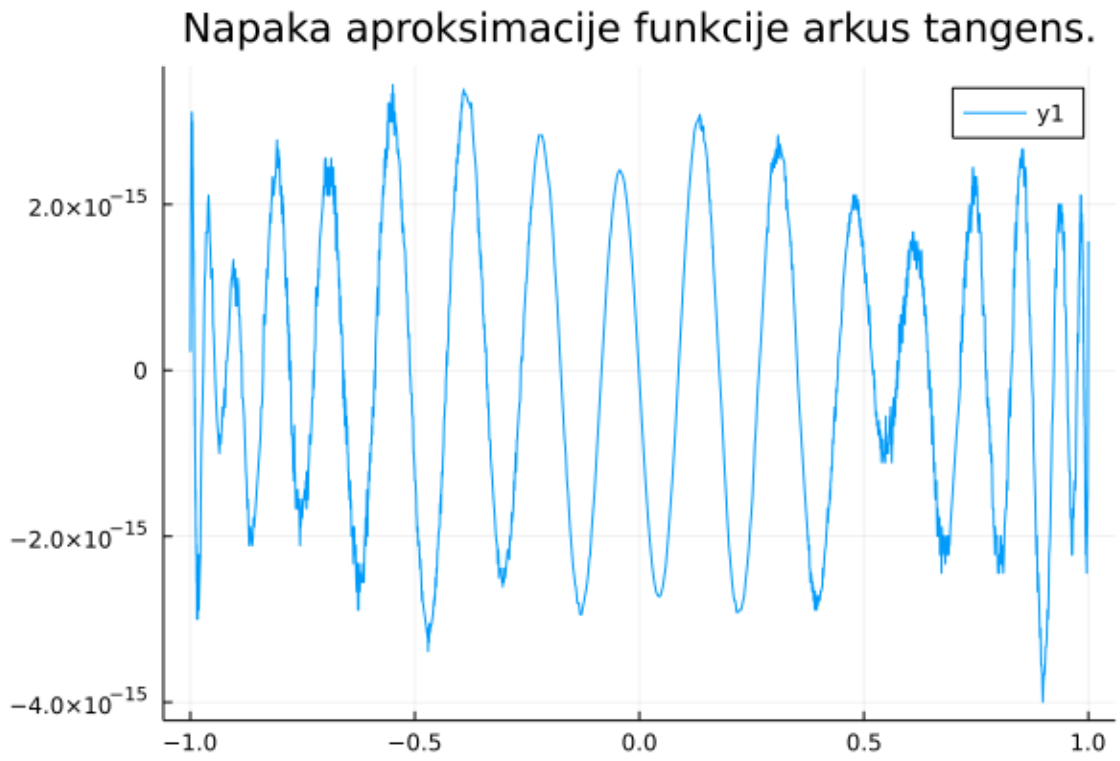
Primer

Uporabimo Čebiševo vrsto za implementacijo funkcije $\arctan(x)$. Definicijsko območje so vsa realna števila, zato si pomagamo z enakostjo

$$\arctan(x) + \arctan\left(\frac{1}{x}\right) = \begin{cases} \frac{\pi}{2}; x > 0 \\ -\frac{\pi}{2}; x < 0 \end{cases}$$

tako da lahko funkcijo računamo le na intervalu $[-1, 1]$.

```
using NumMat, Plots
catan = chebfun(atan, -1, 1)
println("Stopnja aproksimacijskega polinoma: ", stopnja(catan))
plot(x->catan(x)-atan(x), -1, 1, title="Napaka aproksimacije funkcije arkus tangens.")
```



Povezave

- Knjižnica ‘chebfun‘
- THE AUTOMATIC SOLUTION OF PARTIAL DIFFERENTIAL EQUATIONS USING A GLOBAL SPECTRAL METHOD
- Trefetnova knjiga

Koda

- NumMat.ChebFun
- NumMat.chebfun
- NumMat.chebfun
- NumMat.chebkoef
- NumMat.chebval

NumMat.chebfun – Method.

```
chebfun(fun, a, b, n)
```

vrne razvoj funkcije po Čebiševih polinomih stopnje največ n na intervalu [a, b]

NumMat.chebfun – Method.

```
chebfun(fun, a, b)
```

vrne razvoj funkcije v Čebiševo vrsto.

NumMat.chebkoef – Method.

```
fp = chebkoef(fun,n)
```

izračuna koeficiente v razvoju funkcije fun na intervalu [-1,1] po Čebiševih polinomih stopnje največ n.

NumMat.chebval – Method.

```
y = chebval(p,x)
```

izračuna vrednost polinoma, ki je podan v Čebiševi bazi p(x) = p(1)T0(x) + p(2)T1(x) + ... + p(n+1)Tn(x)

```
julia> p = [2 0 -1]; t = LinRange(-1, 1, 100);

julia> @assert chebval([0,0,1],t) ≈ 2t.^2 - 1
```

NumMat.ChebFun – Type.

Podatkovni tip za vrsto Čebiševih polinomov

Integral

Integrali

Metoda nedoločenih koeficientov

> **Kvadrature ali integral**
>
> **Kvadratura** je zgodovinsko ime za integral. Izraz se je ohranil pri formulah za numerično računanje integralov, ki jim pogosto rečemo **kvadraturne formule** ali preprosto **kvadrature**.

Kvadraturne formule lahko izpeljemo s preprostim postopkom imenovanim *metoda nedoločenih koeficientov*. Kvadraturna formula bo tem višjega reda, za kolikor višjo stopnjo polinomov bo formula točna, brez napake. Tako lahko koeficiente določimo preporsto tako, da v formulo zaporedoma vstavljamo polinome vedno višjih stopenj, dokler ne določimo vseh koeficientov kvadraturne formule.

Primer

Izpeljite kvadraturno formulo za integral

$$\int_{-1}^{1} f(x)dx = Af(-x_0) + Bf(0) + Cf(x_0)$$

z metodo nedoločenih koeficientov. Iz omenjenega pravila izpeljite še sestavljeno pravilo in izpeljite tudi oceno za napako.

Gaussove kvadraturne formule

Že pri interpolaciji smo videli, da ekvidistančne točke niso najboljša izbira. Podobno velja pri kvadraturnih formulah. Če želimo poiskati kvadraturno formulo s kar največjega reda, s čim manj izračuni funkcije, ekvidistančne točke niso prava izbira. Boljša izbira so ničle ortogonalnih polinomov.

Primer

Izračunajte integral

$$\int_{0}^{5} \sin(x)dx$$

s sestavljeno trapezno, sestavljeno Simpsonovo in Gauss-Lagendrovimi kvadraturami. Primerjajte število zahtevanih izračunov funkcije za različne metode, ki dajo isto natančnost 10^{-12}. Gauss-Legendrove kvadraturne formule izračunamo z Golub-Welschovim algoritmom.

Koda

- NumMat.gauss_quad_rule

NumMat.gauss_quad_rule – Function.

```
x, w = gauss_quad_rule(a, b, c, mu, n)
```

Izračuna uteži w in vozlišča x za Gaussove kvadraturne formule za integral

$$\int_a^b f(x)w(x)dx \simeq w_1 f(x_1) + \ldots w_n f(x_n)$$

z Golub-Welshovim algoritmom.

Parametri a, b in c so funkcije n, ki določajo koeficiente v tročlenski rekurzivni formuli za ortogonalne polinome na intervalu $[a, b]$ z utežjo w(x)

$$p_n(x) = (a(n)x + b(n))p_{n-1}(x) - c_n p_{n-2}(x)$$

mu je vrednost integrala uteži na izbranem intervalu

$$\mu = \int_a^b w(x)dx$$

Primer

za računanje integrala z utežjo $w(x) = 1$ na intervalu $[-1, 1]$, lahko uporabimo Legendrove ortogonalne polinome, ki zadoščajo rekurzivni zvezi

$$p_{n+1}(x) = \frac{2n+1}{n+1} x p_n(x) - \frac{n}{n+1} p_{n-1}$$

Naslednji program izpiše vozlišča in uteži za n od 1 do 5

```
a(n) = (2*n-1)/n; b(n) = 0.0; c(n) = (n-1)/n;
μ = 2;
println("Gauss-Legendrove kvadrature")
for n=1:5
  x0, w = gauss_quad_rule(a, b, c, μ, n);
  println("n=$n")
  println("vozlišča: ", x0)
  println("uteži: ", w)
end
```

Večrazsežni integrali

V poglavju o enojnih integralis smo spoznali, da je večina kvadraturnih formul preprosta utežena vsota

$$\int_a^b f(x)dx \approx \sum w_k f(x_k),$$

kjer so uteži w_k in vozlišča x_k izbrana tako, da je formula točna za polinome čim višjega reda.

Pri večkratnih integralih se stvari nekoliko zakomplicirajo, a v bistvu ostanejo enake. Kvadrature so tudi za večkratne integrale večinoma navadne utežene vsote vrednosti v izbranih točkah na območju.

Dvojni integral in integral integrala

Oglejmo si najbolj enostaven primer, ko integriramo funkcijo na kocki $[a,b]^2$. Dvojni integral lahko zapišemo kot dva gnezdena enojna integrala [1]

$$\int\int_{[a,b]^2} f(x,y)dxdy = \int_a^b \left(\int_a^b f(x,y)dy \right) dx = \int_a^b \left(\int_a^b f(x,y)dx \right) dy$$

Najbolj enostavno je izpeljati kvadrature za večkratni integral, če za vsak od gnezdenih enojnih integralov uporabimo isto kvadraturno formulo

$$\int_a^b f(x)dx \approx \sum_{k=1}^{n} w_k f(x_k)$$

z danimi utežmi $w_1, w_2, \ldots w_n$ in vozlišči $x_1, x_2, \ldots x_n$. Če za zunanji integral uporabimo kvadrature \eqref{eq:quad}, dobimo

$$\int\int_{[a,b]^2} f(x,y)dxdy = \int_a^b \left(\int_a^b f(x,y)dy \right) dx = \sum w_i Fy(x_i),$$

kjer je funkcija $Fy(x)$ enaka integralu po y, za katero lahko zopet uporabimo kvadrature \eqref{eq:quad}

$$Fy(x) = \int_a^b f(x,y)dy \approx \sum w_j f(x,y_j)$$

Dvojni integral lahko tako približno izračunamo kot dvojno vsoto

$$\int\int_{[a,b]^2} f(x,y)dxdy \approx \sum_{i,j} w_i w_j f(x_i, y_j).$$

Kvadraturni formuli, ki jo dobimo na ta način, pravimo *produktna formula*.

Produktne formule trpijo za prekletstvom dimenzionalnosti

Število vozlišč, ki jih dobimo, ko uporabimo produktno formulo, narašča eksponentno z dimenzijo prostora, na katerem integriramo. Zato produktne kvadrature postanejo hitro (že v dimenzijah nad 6, 7) časovno tako zahtevne, da celo slabše konvergirajo kot metoda Monte Carlo. Ta pojav, da s povečevanjem dimenzij, zahtevnost problemov eksponentno narašča imenujemo prekletstvo dimenznionalnosti in se pojavi tudi na drugih področjih.

Razpršene mreže omilijo prekletsvo dimenzionalnosti

Z dimenzijo narašča delež volumna, ki je „na robu“. Oglejmo si $d-$dimenzionalno enotsko kocko $[-1,1]^d$. Če interval $[-1,1]$ razdelimo na točke v notranjosti $[-\frac{1}{2}, \frac{1}{2}]$ in točke na robu $[-1,1] - [-\frac{1}{2}, \frac{1}{2}]$, sta v eni dimenziji oba dela enako dolga. V višjih dimenzijah pa delež točk v kocki, ki so na robu v primerjavi s točkami v notranjosti narašča. Delež točk v notranjosti lahko preprosto izračunamo:

$$P\left(\left[-\frac{1}{2}, \frac{1}{2}\right]^d\right) = \frac{1}{2^d}$$

in pada eksponentno z dimenzijo. Zato je smiselno na robu uporabiti bolj gosto mrežo kot v notranjosti. Tako je matematik Sergey A. Smolyak razvil razpršene mreže, ki izkoriščajo to idejo in delno omilijo prekletstvo dimenzionalnosti.

Dodatna naloga

Izpelji formulo za ostanek pri izračunu dvojnega integrala na kvadratu $[a,b]^2$, če za kvadraturo \eqref{eq:quad} uporabiš sestavljeno Simpsonovo pravilo

$$\int_a^b f(x)dx \approx \frac{h}{3}\sum_{i=0}^{n-1}\left(f(x_{2i}) + 4f(x_{2i+1}) + f(x_{2i+2})\right) + R_f,$$

kjer je $h = \frac{b-a}{2n}$, vozlišča $x_i = a + ih$ in ostanek je enak

$$R_f = -\frac{h^4}{180}(b-a)f^{(4)}(\xi),$$

za nek $\xi \in (a,b)$.

[1] Več o tem, kdaj je mogoče večkratni integral zamenjati z gnezdenimi enojnimi integrali pove Fubinijev izrek.

Povprečna razdalja med točkama na kvadratu $[0,1]^2$

Naloga

Izračunaj povprečno razdaljo med dvema točkama na kvadratu $[0,1] \times [0,1]$. Primerjaj različne metode.

Rešitev

Povprečna razdaljo lahko izračunamo s štirikratnem integralom

$$\bar{d} = \int_{[0,1]^4} \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} dx_1 dx_2 dy_1 dy_2.$$

Za izračun bom uporabili produktno kvadraturo s sestavljeno Simpsonovo formulo in metodo Monte Carlo.

```
using NumMat, LinearAlgebra
## povprečna razdalja med točkama v kocki [0,1]^2
f(x) = norm(x[1:2]-x[3:4]); # razdalja
x0 = LinRange(0, 1, 7); w = (x0[2]-x0[1])/3*[1 4 2 4 2 4 1];
I = ndquad(f, x0, w, 4)
```

```
0.5196600342442085
```

Poskusimo še z metodo Monte Carlo, kjer vozlišča izberemo slučajno enakomerno na izbranem območju.

```
function mcquad(fun, n, dim)
    Ef = 0
    for i=1:n
      Ef += fun(rand(dim))
    end
    return Ef/n
end
mcquad(f, 100000, 6)
```

```
0.5223765445088695
```

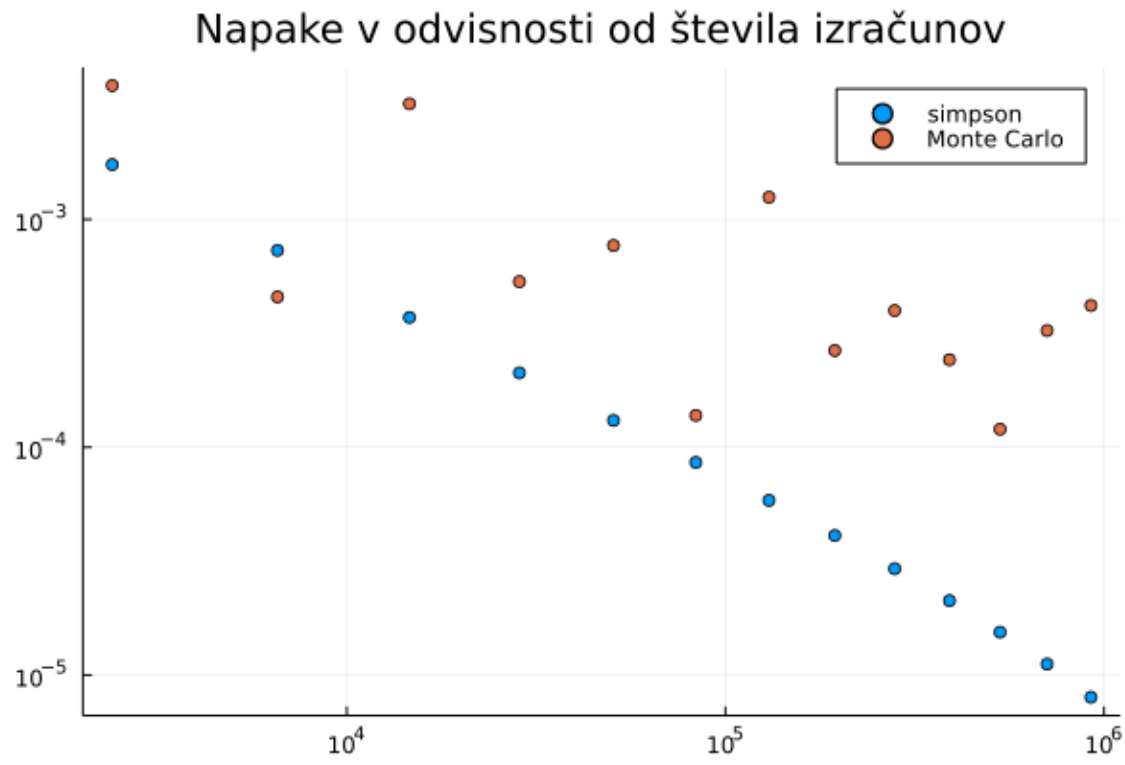
Primerjava različnih metod

Produktna kvadratura s sestavljeno Simpsonovo formulo

Poglejmo si, kako je s hitrostjo konvergence pri produktnih kvadraturah.

```
using Plots, Random
Random.seed!(1234)
ndquad_simpson(f, n, dim) = ndquad(f, LinRange(0, 1, 2n+1),
                            1/(6n)*vcat([1], repeat([4,2], n-1), [4, 1]), dim)
dim = 4
I = ndquad_simpson(f, 20, dim)
n = 3:15
napake_s = []
napake_mc = []
for nk in n
```

```
    push!(napake_s, ndquad_simpson(f, nk, dim) - I)
    push!(napake_mc, mcquad(f, (2nk+1)^4, dim) - I)
end
scatter((2n.+1).^4, abs.(napake_s), scale=:log10, label="simpson")
scatter!((2n.+1).^4, abs.(napake_mc), scale=:log10, label="Monte Carlo", title="Napake v odvisnosti
↪  od števila izračunov")
```



Z zbranimi podatki lahko določimo približni red simpsonove produktne metode za 4 kratne integrale

```
konst, red = hcat(ones(size(n)), log.((2n.+1).^4))\log.(abs.(napake_s))
println("Napaka produktne simpsonove formule pada z n^(", red, "), kjer je n število izračunov
↪  funkcijske vrednosti.")
```

```
Napaka produktne simpsonove formule pada z n^(-0.8871618142136484), kjer je n število čizraunov
    funkcijske vrednosti.
```

Podobno lahko vsaj približno ocenimo hitrost konvergence za metodo Monte Carlo. Pri čemer se moramo zavedati, da je vrednost in tudi napaka odvisna od zaporedja slučajnih vozlišč, zato je ocena zgolj okvirna:

```
konst, red = hcat(ones(size(n)), log.((2n.+1).^4))\log.(abs.(napake_mc))
println("Napaka pri Monte Carlo pada približno z n^(", red, ") za izbrane vzorce.")
```

```
Napaka pri Monte Carlo pada žpriblino z n^(-0.383520832756106) za izbrane vzorce.
```

Centralni limitni izrek in konvergenca Monte Carlo

Konvergenco metode Monte Carlo(MC) je posledica centralnega limitnega izreka, ki pove, da je vzorčno povprečje $\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$, s katerim ocenimo integral pri metodi MC, porazdeljen približno normalno

$$\bar{x} \sim N(\mu, \sigma) = N(\mu_0, \frac{\sigma_0}{\sqrt{n}}),$$

kjer je $\mu_0 = E(X)$ povprečje in $\sigma_0 = \sigma(X)$ standardni odklon porazdelitve, ki jo vzorčimo. Standardni odklon porazdelitve vzorčnih povprečij torej pada s korenom velikosti vzorca $\sqrt{n}$, s tem pa tudi širina porazdelitve za $\bar{x}$ in natančnost izračuna z metodo MC.

Koda

- NumMat.ndquad

NumMat.ndquad – Function.

```
I = ndquad(f, x0, utezi, d)
```

izračuna integral funkcije f na d-dimenzionalni kocki $[a, b]^d$ z večkratno uporabo enodimenzionalne kvadrature za integral na intervalu $[a, b]$, ki je podana z utežmi utezi in vozlišči x0.

Primer

```
julia> f(x) = x[1] + x[2]; #f(x,y)=x+1;
julia> utezi = [1,1]; x0 = [0.5, 1.5]; #sestavljeno sredinsko pravilo
julia> ndquad(f, x0, utezi, 2)
8.0
```

Odvod

Automatsko odvajanje

V grobem poznamo 3 načine, kako lahko odvajamo funkcije z računalnikom

- simbolično odvajanje
- numerično odvajanje z končnimi diferencami
- avtomatsko odvajanje programske kode

Tokrat se bomo posvetili avtomatskemu odvajanju. Programi

Avtomatsko odvajanje z dualnimi števili

Avtomatično odvajanje lahko implementiramo tako, da realna števila razširimo s posebnim elementom ε, za katerega velja $\varepsilon \neq 0$ in $\varepsilon^2 = 0$. Tako dobimo množico dualnih števil, podobno kot dobimo množico kompleksnih številih, če realna števila razširimo z imaginarno enoto.

Dualna števila

Dualna števila so elementi oblike $a + b\varepsilon$, pri čemer sta a in b poljubni realni števili. Z dualna števili računamo kot z navadnimi binomi, le da upoštevamo, da je $\varepsilon^2 = 0$. Tako je produkt dveh dualnih števil enak

$$(a + b\varepsilon)(c + d\varepsilon) = ac + (ad + bc)\varepsilon + bd\varepsilon^2,$$

ker pa je $\varepsilon^2 = 0$, dobimo naslednje pravilo za produkt

$$(a + b\varepsilon)(c + d\varepsilon) = ac + (ad + bc)\varepsilon.$$

Podobno lahko izpeljemo pravilo za deljenje, oziroma inverz

$$\frac{1}{a + b\varepsilon} = \frac{a - b\varepsilon}{(a + b\varepsilon)(a - b\varepsilon)} = \frac{1}{a} - \frac{b}{a^2}\varepsilon$$

Za izpeljavo pravila za potenciranje z naravnimi števili, si pomagamo s potenco binoma

$$(a + b\varepsilon)^n = a^n + \binom{n}{n-1} a^{n-1} b\varepsilon + \varepsilon^2(\ldots ...) = a^n + na^{n-1}b\varepsilon,$$

za racionalne in realne potence, lahko uporabimo binomsko vrsto, za potence, kjer nastopa ε v eksponentu, pa bi potrebovali potenčno vrsto za funkcijo e^x.

Dualna števila lahko izkoristimo za računanje odvodov. Z dualnimi števili se namreč računa podobno kot z diferenciali oziroma z linearnim delom Taylorjeve vrste imenovanim tudi 1-tok (1-jet). Poglejmo si primer produkta dveh 1-tokov za dve funkciji v neki točki x_0

$$(f(x_0) + f'(x_0)dx)\,(g(x_0) + g'(x_0)dx) = f(x_0)g(x_0) + (f(x_0)g'(x_0) + f'(x_0)g(x_0))dx + O(dx^2).$$

Vse potence dx^k za $k > 1$, lahko potisnemo v člen $O(dx^2)$ oziroma jih zanemarimo. Diferencial dx se tako obnaša enako kot element ε. Dualna števila imajo isto aritmetiko kot 1-tokovi funkcij v neki točki x_0. To dejstvo izkoristimo za računanje odvoda.

Implementacija dualnih števil v programskem jeziku julia

Definirali bomo podatkovno strukturo `Dual` za dualna števila in osnovne aritmetične operacije za to strukturo.

Poleg tega bomo definirali funkcijo `odvod(f, x)`, ki izračuna odvod funkcije v dani točki.

Obe funkciji preiskusimo na primeru funkcije, ki računa kvadratni koren z Newtonovo metodo.

```
function koren(x)
    y = 1+(x-1)/2 # Taylor
    for i=1:100
      y = (y + x/y)/2
    end
    return y
end
```

Koda

`NumMat.DualNumber` – Type.

```
    DualNumber(x, dx)
```

Predstavlja skalarnoi spremenljivko x in njen diferencial dx.

`NumMat.odvod` – Function.

```
y = odvod(f<:Function, x)
```

Izračuna vrednost odvoda funkcije `f` v točki `x`.

Diferencialne enačbe

Populacijska dinamika

Perioda geostacionarne orbite

Gravitacijski potencial Zemlje lahko razvijemo z multipolnim razvojem

Koeficiente lahko razberemo iz geopotencialnega modela

Domače naloge

1. domača naloga

Oddaja naloge

Naloge oddajte na Gitlab.

> **Warning**
>
> Naloge, ki jih boste oddali na Gitlabu, bodo **javno dostopne**. Če kdo ne želi, da je njegov izdelek javno dostopen, naj si ustvari privatno različico(fork) repozitorija nummat-1920 in naj povabi zraven kolega, ki bo kodo pregledal in asistenta(@mrcinv). Če kdo ne želi uporabljati Gitlaba ali programskega jezika julia, naj se oglasi asistentu.

Navodila za oddajo naloge (glejte tudi dokument o delu z gitom in Gitlabom)

- Ustvarite svoj zahtevek(issue) in zahtevo za združitev(merge request).
- Napište kodo, teste in kratek dokument z opisom rešitve in primerom. Kodo potisnite na repozitorij v svoji git veji.
- Povabite kolega, da pregleda vašo kodo, tako da ga omenite kot `@vzdevek` v komentarju na zahtevi za združitev(merge request).
- Ko je pregled končan in pripombe kolega upoštevane, povabite še asistenta (`@mrcinv`).

Rešitev naloge naj vsebuje vsaj 3 datoteke:

- izvorna kodo v `src/domace/vzdevek/ImeNaloge.jl`
- testi v `test/domace/vzdevek/ImeNaloge.jl`, ki so vključeni v `test/runtests.jl`
- kratek dokument z opisom rešitve v `doc/src/domace/vzdevek/ImeNaloge.md` in „praktičnim“ primerom uporabe (slikice prosim :-)).

Po lastni presoji, lahko rešitev vsebuje tudi več drugih datotek. Vsaka funkcija naj tudi vsebuje docstring z opisom funkcije.

Naloga

Pasovne diagonalno dominantne matrike

Pasovna matrika je matrika, ki ima neničelne elemente le v glavni diagonali in v nekaj stranskih diagonalah ob glavni diagonali. Matrika je diagonalno dominantna po vrsticah, če za vse i velja

$$|a_{ii}| \geq \sum_{j \neq i} |a_{ij}|.$$

Definirajte podatkovni tipe

- `PasovnaMatrika`, ki predstavlja pasovno matriko,
- `ZgornjePasovnaMatrika`, ki predstavlja zgornje trikotno pasovno matriko in
- `SpodnjePasovnaMatrika`, ki predstavlja spodnje trikotno pasovno matriko

Podatkovni tipi naj hranijo le neničelne elemente matrike. Za omenjene podatkovne tipe definirajte metode za naslednje funkcije:

- indeksiranje: `Base.getindex`,`Base.setindex!` in `Base.size`. Podatkovni tipi naj ustrezajo vmesniku AbstractArray, da lahko do elementov dostopamo s sintaksno `A[i,j]`.
- množenje z desne `Base.*` z vektorjem
- „deljenje“ z leve `Base.\`
- `lu`, ki izračuna LU-razcep brez pivotiranja, če je matrika diagonalno dominantna, sicer pa javi napako. Vrnjena faktorja naj bosta tipa `SpodnjePasovnaMatrika` in `ZgornjePasovnaMatrika`.

Časovna zahtevnost omenjenih funkcij naj bo linearna (odvisna od širine pasu). Več informacij o tipih in vmesnikih.

Za primer rešite sistem za Laplaceovo matriko v 2D.

2. domača naloga

Navodila

Izberite eno izmed spodnjih nalog. Po možnosti tako, ki je ni še nihče drug izbral (preverite zahtevke na Gitlabu).

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

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

$$V(i,j) = A(i, I(i,j)).$$

V matriki V se torej nahajajo neničelni elementi matrike A. Vsaka vrstica matrike V vsebuje ničelne 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 tridiagonalni 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

$$|A\mathbf{x}^{(k)} - \mathbf{b}|_\infty < nat.$$

Metodo uporabite za vožitev 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

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

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 `omega`, pri katerem SOR najhitreje konvergira in predstavite odvisnost hitrosti konvergence od izbire ω.

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

$$Ax = b,$$

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.

Metoda konjugiranih gradientov s predpogojevanjem

Za pohitritev konvergence iterativnih metod, se velikokrat izvede t. i. predpogojevanje(angl. preconditioning). Za simetrične pozitivno definitne matrike je to pogosto nepopolni razcep Choleskega, pri katerem sledimo algoritmu za razcep Choleskega, le da ničelne elemente pustimo pri miru.

Naj bo A $n \times n$ pozitivno definitna razpršena matrika(velika večina elementov je ničelnih $a_{ij} = 0$). Matriko zaradi prostorskih zahtev hranimo kot *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

$$Ax = b$$

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

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

$$[\cos(\alpha); \sin(\alpha)],$$

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.

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 QR razcep zgornje hessenbergove matrike). 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.

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

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

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:

$$H = Q^T A Q.$$

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

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

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

Napišite funkcijo `lambda, vektor = inv_lastni(A, l)`, ki najprej naredi hessenbergov razcep in nato izračuna lastni vektor in točno lastno matrike A, kjer je `l` približek za lastno vrednost. Inverza matrike A nikar ne računajte, ampak raje uporabite LU razcep in na vsakem koraku rešite sistem $L(Ux^{n+1}) = x^n$.

Metodo preskusite za izračun ničel polinoma. Polinomu

$$x^n + a_{n-1}x^{n-2} + ...a_1 x + a_0$$

lahko priredimo matriko

$$\begin{bmatrix} 0 & 0 & \dots & 0 & -a_0 \\ 1 & 0 & \dots & 0 & -a_1 \\ 0 & 1 & \dots & 0 & -a_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & 1 & -a_{n-1} \end{bmatrix},$$

katere lastne vrednosti se ujemajo z ničlami polinoma.

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

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

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:

$$T = Q^T A Q.$$

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

$$y''(x) = \lambda^2 y(x),$$

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

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

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

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`).

QR iteracija z enojnim premikom

Naj bo A simetrična matrika. Napišite funkcijo, ki poišče lastne vektorje in vrednosti simetrične matrike z naslednjim algoritmom

- Izvedi Hessenbergov razcep matrike $A = U^T T U$ (uporabite lahko vgrajeno funkcijo `LinearAlgebra.hessenberg`)
- Za tridiagonalno matriko T ponavljaj, dokler ni $h_{n-1,n}$ dovolj majhen:
  - za $T - \mu I$ za $\mu = h_{n,n}$ izvedi QR razcep
  - nov približek je enak $RQ + \mu I$
- Postopek ponovi za podmatriko brez zadnjega stolpca in vrstice

Napiši metodo `lastne_vrednosti, lastni_vektorji = eigen(A, EnojniPremik(), vektorji = false)`, ki vrne

- vektor lastnih vrednosti simetrične matrike `A`, če je vrednost `vektorji` enaka `false`.
- vektor lastnih vrednosti `lambda` in matriko s pripadajočimi lastnimi vektorji `V`, če je `vektorji` enaka `true`

Pazi na časovno in prostorsko zahtevnost algoritma. QR razcep tridiagonalne matrike izvedi z Givensovimi rotacijami in hrani le elemente, ki so nujno potrebni (glej nalogo QR razcep simetrične tridiagonalne matrike).

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

[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

3. domača naloga

Navodila

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 s funkcijami

Porazdelitvena funkcija normalne slučajne spremenljivke

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

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

Fresnelov integral

Napišite učinkovito funkcijo, ki izračuna vrednosti Fresnelovega kosinusa

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

Namig: Uporabite pomožni funkciji

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

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

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.

Integralski sinus

Napišite učinkovito funkcijo, ki izračuna integralski sinus

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

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

Besselova funkcija

Napišite učinkovito funkcijo, ki izračuna Besselovo funkcijo J_0:

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

Naloge s števili

Sila težnosti

Izračunajte velikost sile težnosti med dvema vzporedno postavljenima enotskima homogenima kockama na razdalji 1. Predpostavite, da so vse fizikalne konstante, ki nastopajo v problemu, enake 1. Sila med dvema telesoma $T_1, T_2 \subset \mathbb{R}^3$ je enaka

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

Ploščina hipotrohoide

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

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

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

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

Povprečna razdalja

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

$$T = [-1,1]^3 - [0,1]^3.$$

Povprečno razdaljo na produktu razlik množic $(A-B)\times(A-B)$ lahko določimo z integralom:

$$\int_{A-B}\int_{A-B} \|\vec{r_1} - \vec{r_2}\| dr_1 dr_2 = \int_A\int_A \|\vec{r_1} - \vec{r_2}\| - 2\int_A\int_B \|\vec{r_1} - \vec{r_2}\| dr_1 dr_2 + \int_B\int_B \|\vec{r_1} - \vec{r_2}\| dr_1 dr_2$$

Ploščina Bézierove krivulje

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

$$(0,0), (1,1), (2,3), (1,4), (0,4), (-1,3), (0,1), (1,0).$$

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:

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

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

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

$$x_i = \cos\left(\frac{i\pi}{n}\right); \quad i = 0 \ldots n,$$

vrednosti uteži λ_i pa so enake

$$\lambda_i = (-1)^i \begin{cases} 1 & 0 < i < n \\ \frac{1}{2} & i = 0, n. \end{cases}$$

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}.

Gauss-Legendrove kvadrature

Izpelji Gauss-Legendreovo integracijsko pravilo na dveh točkah

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

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

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

na 10 decimalk natančno.

4. domača naloga

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 pridobilili 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.

Težje naloge

Ničle Airijeve funkcije

Airyjeva funkcija je dana kot rešitev začetnega problema

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

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.

Namig

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

$$y'(x) = A(x)y,$$

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

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

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.

Dolžina implicinto podane krivulje

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

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

Krivuljo lahko poiščete kot rešitev diferencialne enačbe

$$\dot{\mathbf{x}}(t) = \nabla F_1 \times \nabla F_2.$$

Perioda limitnega cikla

Poiščite periodo limitnega cikla za diferencialno enačbo

$$x''(t) - 4(1 - x^2)x'(t) + x = 0$$

na 10 decimalk natančno.

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.

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

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

V brezdimenzijskih koordinatah (dolžinska enota je kar razdalja med masama M in m) postavimo maso M v točko $(-\mu, 0, 0)$, maso m pa v točko $(\bar{\mu}, 0, 0)$. Označimo z R in r oddaljenost satelita s položajem (x, y, z) od mas M in m, tj.

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

Enačbe gibanja sonde so potem:

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

Lažja naloga (ocena največ 9)

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

Matematično nihalo

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

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

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:

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

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.

Kako sodelovati pri predmetu Numerična matematika

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

Ta repozitorij je namenjem zbiranju gradiv in izdelavi domačih nalog pri predmetu Numerična matematika. Želimo si, da skupaj ustvarimo lepo urejen in zaokrožen repozitorij z vsemi gradivi, ki jih bomo ustvarili na vajah in z domačimi nalogami. Zato ste vsi študenti, ki ste vpisani na ta predmet vabljeni, da se pridružite temu projektu in sodelujete pri tem.

Laboratorijske vaje

Na laboratorijskih vajah bomo iskali ravnotežje med razlago na tablo in programiranjem asistenta in samostojnim delom študentov. Vaje bodo (upam) uravnotežena mešanica:

- razlage na tablo
- programiranja na projektorju
- nekaj samostojnega programiranja

Note

Del svojih obveznosti lahko študent opravi že na vajah, če za naloge, ki jih bomo reševali na vajah, izdela rešitev in poda zahtevo za združitev (*merge request*).

Domače naloge/sprotno delo

Domače naloge oziroma sprotno delo bo potekalo sodelovalno. To pomeni, da bomo skupaj razvijali knjižnico numeričnih funkcij v tem repozitoriju. Poleg tega naj bi vsak študent naredil eno nalogo v svojem repozitoriju.

Za pozitivno oceno naj bi študent prispeval več stvari v eni od naslednjih oblik:

- rešitve domačih nalog v obliki *zahteve za združitev(merge request)*
- pregled rešitev domačih nalog svojih kolegov
- sodelovanje na vajah
  - pisanje kode
  - pisanje dokumentacije
  - pisanje testov

- sodelovanje preko zaznamkov (issues)

Ocena se določi na podlagi kvalitete prispevkov

- na oceno vplivajo le dobri prispevki, če kdo kdaj pošlje kakšno neumnost, se mu to ne šteje v minus
- ko študent zbere dovolj prispevkov, mu asistent dodeli oceno
- ocena se lahko le še popravi z bolj kvalitetnimi prispevki

Kako oddati domačo nalogo

Domače naloge oddajte kot zahtevo za združitev (merge request). Spodaj je zelo na kratek opis, kako to naredite. Predpostavljam, da ste vsaj malo vešči z orodjem git.

Seznam opravil za domačo nalogo:

- odprete zahtevek
- pripravite zahtevo za združitev
- napišete kodo, teste in dokumentacijo
- povabite kolega za pregled
- ko vaš kolega potrdi vašo nalogo, povabite še asistenta za pregled
- asistent združi vašo vejo v master (ali pa vas pošlje na 3. korak

Odpiranje zahtevka

Zahteva za združitev (merge request)

Delo z izvorno kodo z Git-om

- najprej si na svojem računalniku ustvarite klon repozitorija

```
git clone https://gitlab.com/nummat/nummat-1920.git
cd nummat-1920
```

- nato ustvarite novo

vejo

```
git branch nickname-dn1
git checkout nickname-dn1
```

> **Note**
>
> Zahtevo za združitev in vejo lahko enostavno ustvarite s klikom na gumb `Create merge request`, ko ustvarite zahtevek v gitlabu.

- nalogo rešite in sproti spremembe z `git commit` beležite v repozitorij.
- nalogo prenesete na strežnik z ukazom `git push`

```
git push origin nickname-dn1
```

Merge request na Gitlab

Ko ste svojo rešitev dokončali in jo uspešno prenesli na gitlab, lahko ustvarite merge request. Za *source* izberete svojo vejo, za *target* pa vejo `master`.

> **Note**
>
> Bolj podroben opis načina dela z git in gitlab je opisan v dokumentu o načinu dela(workflow).

Viri

- priporočila za Gitlab
  - 11 pravil za Gitlab
- Kako v kodo dodamo licenco
- Stil pisanja kode - Octave
- Stil pisanja kode - julia
- Sodelovalni urejevalnik za Markdown

Developer documentation

This document describes the best practices to be followed when working with Git and GitLab in this project.

> **Note**
>
> We *strongly* suggest you use Git from the *command line*. Our suggestion is zsh with Oh my ZSH! on Unix and Babun on Windows.

The Goals

The goals we want to achieve are

- to have a *codebase* that is manageable and *well documented*
- to communicate among the members of the team as conveniently as possible
- to know and document what each one of us is doing

To achieve these goals, we will use the features of Git and Gitlab.

The recomended Workflow

> **Note**
>
> A *workflow* is a way of doing things. It can help us or present a burden.

Overview

1. Create an issue
2. Create a WIP merge request linking to the issue
3. Code, test, commit, code, test, commit, ... (in your branch)
4. Rebase your branch to `master`
5. Remove WIP from the merge request and invite someone to review your changes
6. Merge the code to `master`, remove the source branch and close the issue

Create an issue

Issues should be the primary source for documenting the development process. Label issues apropriatelly and use milestones to group issues.

Create a WIP merge request

WIP merge request is a way to create a merge request that can not be accidentally merged until is ready. Creating a WIP merge requests will help others on your team to better understand what it is that you are doing at the moment, and what is the purpose of different branches.

> **Note**
>
> If you make a merge request from issue, gitlab automatically makes it WIP, creates a new branch and adds a link to the issue. The issue is automatically closed once the mere request is accepted.

> **Don't forget!**
>
> Checkout newly created branch, before you start coding

```
git fetch
git checkout 314-my-awesome-pi
```

Code, test, commit, code

Now you can finaly start coding.

> **Remember to**
>
> - make incremental commits
> - test the code with automated tests
> - write your own tests that test your new feature

Rebase your branch onto `master`

Rebasing is similar to merging, but the order of commits is rearanged so that we get a linear history. Make sure you read about Merging vs. Rebasing before you proceed.

> **Warning**
>
> *Never rebase the `master` branch onto any other branch!!!*

To rebase your branch onto master

```
git fetch origin master
git checkout my-new-feature
git rebase master
```

After rebase, the commits of your branch will simply follow the tip of `master` branch.

If Git can not merge automatically, read the message carefully and proceed as suggested.

> **Note**
>
> If git complains with `error: failed to push some refs to ...` you c an use a switch `--force-with-lease` which is a safer variant of `--force`

> **Warning**
>
> *Never push `--force` to the origin!!!*

Invite someone to review your changes

Remove WIP from the merge request and invite someone to review your changes by mentioning them in a comment to the merge request.

> **Remember to**
>
> - remove WIP from the title
> - mention the `@nicknames` of whoever you want to review the changes

Merge the code to `master`, remove the source branch and close the issue

Dos and don'ts

Everything should be in Git

Every contribution in text format and especially *the code itself should be put into the Git version control system*.

Don't commit in `master`!

> **Don't commit in master**
>
> **Use branches!!!**

Create a branch and merge with `master` when your changes are ready. The exceptions are minor chanages (code that does not affect core functionality like printing to console, etc.) or changes that do not touch the code itself, like documentation, comments, print statements, indentation, etc. *The `master` branch should always pass all tests.*

Commit frequently

> **Tip**
>
> Every change that is rounded should be committed.

If a change can be split into two separate changes that make sense on their own, then do this. Exceptions to the rule are the initial commits and commits of new features - however, the next rule still applies.

DOs and DON'Ts of commit messages

Don't:

- *Don't end the summary line with a period* as it is a title and these don't end with periods.

Do:

- *Use the imperative mode* when writing the summary line and describing what you have done, as if you were commanding someone - e.g., start the line with *Fix*, *Add*, *Change* instead of *Fixed*, *Added*, *Changed*;
- *Leave the second line blank*;
- *Line break the commit message* and make it readable without having to scroll horizontally (line = 80 characters).

Tip

If you feel it's difficult to summarize what you are trying to commit, this may be due to the nature of the commit, which would be better split up into several separate commits.

See also

- GitHub's Writing good commit messages article
- chapter 5 of the Git book

Knjižnica

Vidne funkcije

Dokumentacija za funkcije, ki jih modul `NumMat` izvozi.

Kazalo

- NumMat.hessian_razdalje
- NumMat.invariantna_porazdelitev
- NumMat.iteracija
- NumMat.kddrevo
- NumMat.kmeans
- NumMat.konvergencno_obmocje
- NumMat.korak_sor
- NumMat.lagrangev_zlepek
- NumMat.matrika
- NumMat.najdi_okolico
- NumMat.ndquad
- NumMat.newton
- NumMat.odvod
- NumMat.polyder
- NumMat.polyval
- NumMat.potencna
- NumMat.prehodna_matrika_konj
- NumMat.razdalja
- NumMat.resi
- NumMat.skoki
- RecipesBase.apply_recipe
- RecipesBase.apply_recipe

Skrite funkcije

- NumMat.ChebFun
- NumMat.DualNumber
- NumMat.LaplaceovOperator
- NumMat.LaplaceovOperator
- NumMat.MCKonj
- NumMat.NewtonovPolinom
- NumMat.RobniProblemPravokotnik
- NumMat.Zlepek
- Base.:*
- Base.:*
- NumMat.chebfun
- NumMat.chebfun
- NumMat.chebkoef
- NumMat.chebval
- NumMat.conjgrad
- NumMat.deljene_diference
- NumMat.desne_strani
- NumMat.desne_strani
- NumMat.findinterval
- NumMat.gauss_quad_rule
- NumMat.gradient_razdalje
- NumMat.hermitov_zlepek
- NumMat.hessian_razdalje
- NumMat.invariantna_porazdelitev
- NumMat.iteracija

- NumMat.kddrevo
- NumMat.kmeans
- NumMat.konvergencno_obmocje
- NumMat.korak_sor
- NumMat.lagrangev_zlepek
- NumMat.matrika
- NumMat.najdi_okolico
- NumMat.ndquad
- NumMat.newton
- NumMat.odvod
- NumMat.polyder
- NumMat.polyval
- NumMat.potencna
- NumMat.prehodna_matrika_konj
- NumMat.razdalja
- NumMat.resi
- NumMat.skoki
- RecipesBase.apply_recipe
- RecipesBase.apply_recipe