GIMNAZIJA JOŽETA PLEČNIKA LJUBLJANA

ISKANJE NIČEL POLINOMA

Seminarska naloga

Žiga Vaupotič, 4.a

Mentor: mag. Tomi Zebič

Ljubljana, 2023

Izvleček

V seminarski nalogi analiziramo različne algoritme za iskanje ničel polinomov in njihovo uporabo v računalništvu ter informatiki. Obravnavanih je nekaj glavnih algoritmov v $\mathbb{R}$ in $\mathbb{R}^n$ prostorih. Algoritmi so ovrednoteni na podlagi njihove konvergence in časovne zahtevnosti. Pokažemo tudi nekaj primerov, kjer je izbira posameznega predstavljenega algoritma ključna. Proti koncu naloge se srečamo z problemi v informatiki in računalništvu, ki neposredno uporabljajo analizirane algoritme.

Ključne besede: iskanje ničel polinomov, iskanje ničel, optimizacijske metode

Kazalo

Poglavje 1

Uvod

V računalništvu in informatiki se velikokrat pojavljajo problemi, ki temeljijo na elementarnih matematičnih funkcijah. Najpogosteje se pojavljajo polinomske funkcije. Redno se srečujemo tudi s problemi, kjer moramo iskati ničle teh polinomskih funkciji.

1.1 Namen

Cilj seminarske naloge je opredeliti konvergenco in časovno zahtevnost nekaterih najpogostejših algoritmov za iskanje ničel. Rezultati bodo uporabljeni predvsem za nadaljnjo lastno uporabo. Glavni **namen** naloge je vzbuditi zanimanje za numerično analizo in uporaba le te v informacijskih tehnologijah. Sicer se redko direktno srečamo s problemom iskanja ničel polinomov v programiranju, a se velikokrat srečamo z optimizacijskimi problemi, ki temeljijo na metodah za iskanje ničel funkciji. Sicer funkcije niso vedno polinomske, a osnovne metode kot so Newton-Raphsovnova, biskecija, ... ostajajo enake. Seveda pa nesemo pozabiti, da lahko funkcije po Taylorjevem izreku aproksimiramo s polinomi. Posledično se velikokrat vrnemo nazaj na metode iskanja ničel polinomov.

1.2 Metodologija

Analiza bo temeljila na pregledu že obstoječe literature. Posvetili se bomo predvsem iskanju ničel polinomov v $\mathbb{R}$ in $\mathbb{R}^n$, seveda pa obstajajo tudi metode za iskanje ničel polinomov v $\mathbb{C}$. V računalništvu imajo kompleksni polinomi vlogo predvsem v teoriji šifriranja in numerični analizi. Vsi algoritmi bodo spisani in implementirani v programskem jeziku Python in bodo dodani kot priloga. Za matematično podlogo bomo uporabili knjižnici `numpy` in `scipy`. Za risanje grafov in figur bomo uporabili `matplotlib`. Algoritmi bodo v nalogi zapisani v obliki pseudo-algoritmeskega zapisa. Izogibali s bomo zapisu nepotrebnih matematičnih izrekov, ter predstavil algoritme tudi grafično, kjer bo to mogoče. Večina programov oz. skript bo časovno analizirana na računalniku s procesorjem *AMD Ryzen 7 2700x*, medtem ko bodo nekateri zahtevnejši algoritmi izvedeni na procesorju *AMD EPYC GENOA 9684X*.

1.3 Približki in napake

V vseh računalniških algoritmih se pojavljajo napake. Napake se pojavljajo tudi v obliki natančnosti zapisa števil z plavajočo vejico. Po navadi se v obravnavanih algoritmih uporablja podatkovni tip `double`. Tovrstni podatkovni tip zavzame 8 bajtov. Prvih 12 bitov predstavlja prolog sestavljen iz predznaka (1 bit) in eksponenta (sledečih 11 bitov). Matisa predstavlja 52 bitov. Natančnost lahko izračunamo z izrazom za plavajočo vejico [**?**]

Dodaj sliko oz. diagram

$$fl(x) = m \cdot b^{e-1023} \tag{1.1}$$

m predstavlja mantiso, torej $0.m_1m_2...m_n$, b predstavlja bazo (ponavadi binarno), m_n so števke predstavljene v bazne zapisu (posledično so v mejah od 0 do $b-1$), e eksponent, ki je v mejah $L \leq e \leq U$. Absolutno in relativno napako izračunamo z izrazoma

$$E_{abs} = |x - fl(x)| \qquad E_{rel} = \frac{|x - fl(x)|}{x}$$

V našem primeru imamo na razpolago 11 bitov za zapis eksponenta, torej je zgornja meja $\sum_{i=0}^{10} 2^i = 2047$. To bi bilo sicer res, vendar sta eksponenta 00000000000 in 11111111111 [**?**] rezervirana. Tako je največji možen eksponent 11111111110 oziroma $\sum_{i=1}^{10} 2^i = 2046$. Zanima nas predvsem najmanjši možen eksponent, ki je posledično enak 00000000001 oz. 1. Če vstavimo navedene podatke v izraz dobimo najmanjšo možno število in posledično največjo možno napako za zapis ničle

$$fl(x) = 0,0 \overbrace{\ldots}^{50 \text{ ničel}} 1 \cdot 2^{1-1023} \approx 10^{-308}. \tag{1.2}$$

Sicer lahko zapišemo še nižja števila, vendar v tem primeru več nimamo popolne natančnosti decimalk [**?**]. Najmanjša možna napaka je enaka najmanjšemu možnem zapisu. Ta napaka velja za vse nadaljnje algoritme.

Poglavje 2

Algoritmi v $\mathbb{R}$

Ko govorimo o polinomih si največkrat zamislimo funkcijo $p : \mathbb{R} \to \mathbb{R}$, ki je definirana kot

$$p(x) = \sum_{i=0}^{k} \alpha_i x^i = \vec{x} \cdot \vec{\alpha}. \tag{2.1}$$

Velikokrat tovrstni izraz zapišem kar z skalarnim produktom dveh vektorjev, vektorja $\vec{x}$, ki vsebuje stopnje polinomov in vektorja $\vec{\alpha}$, ki vsebuje koeficiente polinoma. Zanimajo nas rešitve polinomske enačbe oblike $p(x) = 0$, kjer je $k > 4$. Za polinome reda $k \leq 4$ obstajajo rešitve v obliki izrazov, kot je na primer kvadratna formula $x_{1,2} = \frac{-b \pm \sqrt{D}}{2a}$. Abel–Ruffini izrek [**?**] trdi, da za polinome višjega reda tovrstnih formul ni. Zato se ničle polinomov višje stopnje iščejo numerično, dandanes s pomočjo računalnika. Za iskanje polinomov obstaja ogromno različnih algoritmov. Nekateri algoritmi so splošni in veljajo tako za poljubne funkcije, kot tudi za polinome. Seveda velja, da vsako poljubno odvedljivo funkcijo lahko zapišemo v obliki polinoma s pomočjo Taylorjeve vrste. Posledično se nekateri algoritmi posplošijo.

2.1 Splošne Metode

Splošne metode kot so bisekcija, navadna iteracija, tangentna metoda, ... se velikokrat uporabljajo za iskanje ničel. Omogočajo iskanje ničle poljubne funkcije f inje le polinomov. V primeru metod, ki temeljijo na navadni

iteracije je pogoj, da je funkcija k-krat odvedljiva. Odvode ponavadi optimizacijske knjižnice izračunajo kar diskretno, zato lahko ta pogoj včasih kršimo.

Problem splošnih metod nastane pri iskanju več ničel polinomov. Ničle moramo ponavadi izločati. Posledično moramo ničle izločati v pravilne zaporedju, da zagotovimo konvergenco in stabilnost. Tovrstnega problema pri optimizacijskih problemih ne zasledimo, ker je pomemben le en minimum oz. lokalni minimum.

2.1.1 Bisekcija

Najpreprostejši algoritem, ki se uporablja za iskanje ničle je bisekcija. Bisekcija deluje tako, da iteracijsko manjšamo interval na katerem se nahaja ničla. Naj sta točki a_1 in a_2 poljubni točki, tako da velja $f(a_1) < 0$ in $f(a_2) > 0$. Zato, ker je funkcija f zvezna vemo, da obstaja ničla med a_1 in a_2. Sledi preprosto algoritem.

Algoritem 1 Bisekcija

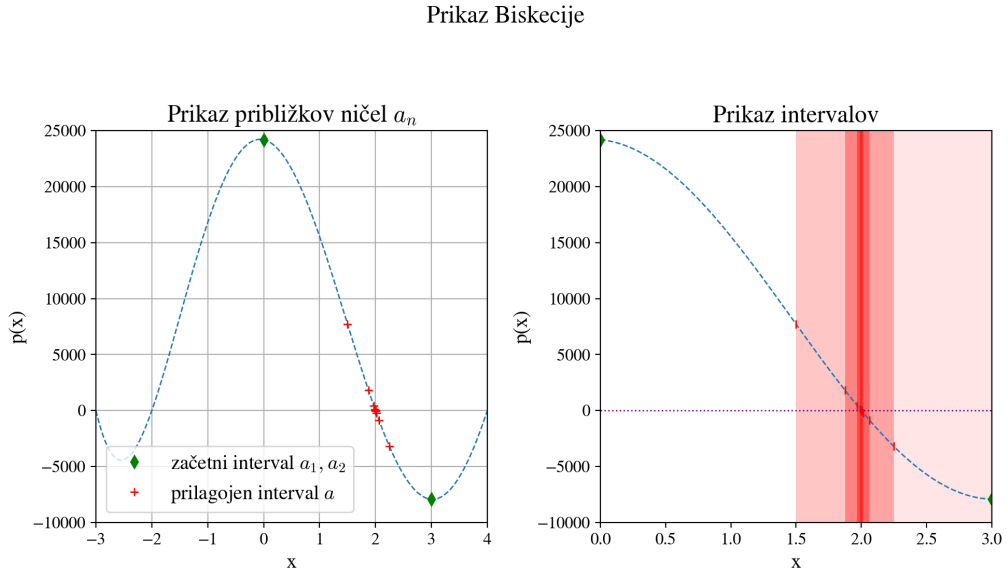
Require: a_1, a_2, ϵ
  while $|f(a)| > \epsilon$ **do** ▷ Iteriraj dokler ni $p(a)$, v okolici ničle
    $a \leftarrow \frac{a_1+a_2}{2}$ ▷ Vsako iteracijo razpolovimo interval
    if $\operatorname{sign} f(a_1) = \operatorname{sign} f(a)$ **then** ▷ Določi točko glede na predznak
      $a_1 \leftarrow a$
    else
      $a_2 \leftarrow a$
    end if
  end while
  return a

Opazimo lahko pogoj $|f(a)| < \epsilon$, pogoj omogoča uporabniku, da spremeni natančnost ϵ in s tem omeji število iteraciji, za $\epsilon = 0$ je natančno enaka natančnosti plavajočih vejic.

Algoritem vsako iteracijo skrči interval (a_1, a_2), tako da obdrži pogoj $f(a_1) \cdot f(a_2) < 0$, kar lahko opazim, tako da algoritem zamenja mejo in-

tervala glede na predznak nove točke a. S tem zagotovimo, da je ničla še vedno v intervalu. Nova točka a predstavlja razpolovišče intervala (a_1, a_2). Interval se krči, dokler a ni v ϵ okolici oz. enak nič.

Primer 1 *Poišči ničlo polinoma* $p(x) = \frac{4}{9}x^8+16x^7-\frac{1}{2}x^6+\frac{411}{10}x^5+\frac{3}{2}x^4-7x^2+2$ *s pomočjo bisekcije. Izvorna koda je na voljo v prilogi.*



Slika 2.1: Potek algoritma biskecije

Hitro lahko opazimo, da bisekcija deluje kot opisano. Rdeči + predstavljajo pridobljen a. Opazimo, da se ta razpolavljaj, tako da velikost interval konvergira proti 0. Seveda je za to potrebna tudi pravilna izbira prvotnega intervala.

2.1.2 Newtonova metoda

Najpogostejša in najbolj poznana metoda za iskanje ničel se imenuje Newtonova oz. tangentna metoda. Metoda je v resnici le poseben primer navadne iteracije. Navadna iteracija je preprosto metoda, kjer približek x_r zapišemo v oblik $x_{r+1} = \mathcal{I}(x_r)$, $\mathcal{I}$ predstavlja iteracijsko funkcijo. Iteracijska funkcija mora izpolnjevati določene pogoje, ki so opisani v [**?**]. Tangenta metoda

uporablja tangento, ki se dotika funkcije v toči $(x, p(x))$. Ničla tangente nato predstavlja približek ničle funkcije. Matematično gledano Newtonova metoda izhaja iz Taylorjevega izreka. Naj bo x_r tangenti približek (oz. prvotni približek, če je iteracija prva) in h zamik, ki ga opravimo v iteraciji. Funkcijo razvijemo v Taylorjevo vrsto:

$$f(x_r + h) = f(x_r) + \frac{f'(x_r)}{1!}h + \frac{f''(x_r)}{2!}h^2 + \mathcal{O}(h^3) \tag{2.2}$$

Zanemarimo višje člene vrste, tako da dobimo tangento (približek z linearno funkcijo). Ovrednotili bi radi ničlo tangente torej $f(x_r + h) = 0$

$$f(x_r + h) \approx f(x_r) + f'(x_r)h \xrightarrow{f(x_r+h)=0} f(x_r) + f'(x_r)h \approx 0 \tag{2.3}$$

Izpostavimo h, da dobimo nov zamik iteracije

$$h = -\frac{f(x_r)}{f'(x_r)} \tag{2.4}$$

Sedaj lahko zapišemo zamik v obliki algoritma naravne iteracije

$$x_{r+1} = \mathcal{I}(x_r) = x_r + h = x_r - \frac{f(x_r)}{f'(x_r)} \tag{2.5}$$

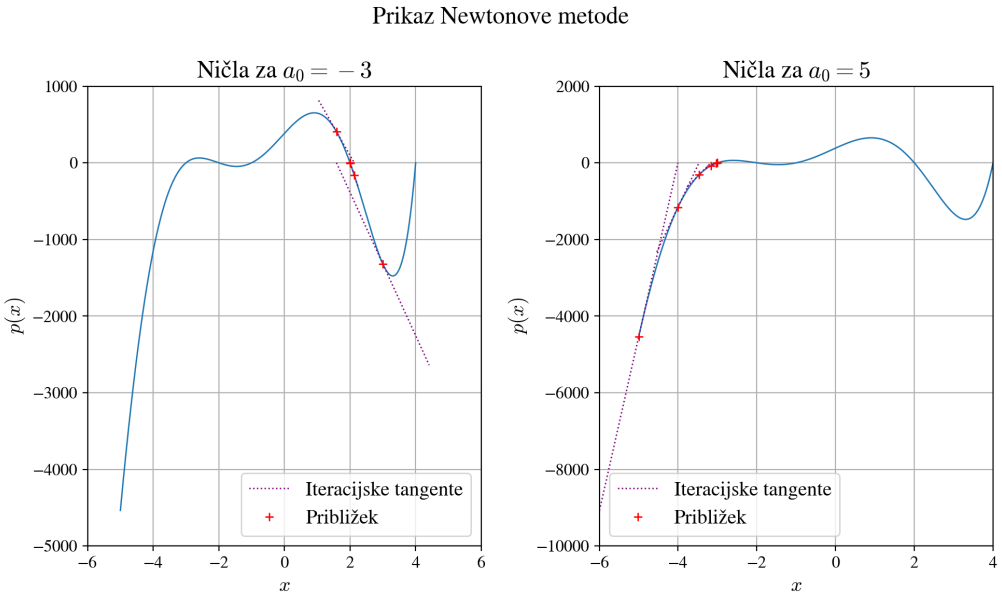
Preprosto algoritem ovrednoti nov zamik iteracije glede na ničlo tangente prejšne iteracije.

Algoritem 2 Newton-Raphsonova metoda

```
procedure DiskretniOdvod(x, h)          ▷ Najdi diskretni odvod f̃'(x)
    return (p(x+h)−p(x))/h              ▷ h predstavlja natančnost diskretnega odvoda
end procedure

Require: x_r
procedure NajdiNičlo
    while |f(x_r)| > ϵ do               ▷ Iteriraj dokler ni p(x_r), v okolici ničle
        d ← f'(x_r)                     ▷ Odvjaja diskretno ali analtično, če je možno
        x_r ← x_r − f(x_r)/d
    end while
    return x_r
end procedure
```

Primer 2 *Najdi ničlo polinoma* $p(x) = (x+3)(x+2)(x+8)(x+1)(x-2)(x-4)$



Slika 2.2: Potek algoritma Newtonove metode

Za dva različna približka nam algoritem vrne najbližjo ničlo. Algoritem konvergira veliko hitreje kot bisekcija, obenem je ničla tudi natančnejša, a več o tem kasneje. Opazimo lahko da se nov približek označen z rdečim + nahaja rano tam kjer tangenta seka ničlo, kar potrdi pravilno implementacijo metode.

2.2 Posebne metode

Veliko smo že povedali o metodah za iskanje ničel, nič pa še nismo povedali o metodah iskanje več ničel polinoma. Iskanje več ničel, kot smo že omenili, predstavlja velike probleme. Večino iterativnih algoritmov je bolj ali manj neuporabnih, ker je konvergenca teh slaba v okolicah, kjer se ničla ne nahaja. Omenili smo eno od tehniki izločanja ničel, ampak kot smo že povedali je konvergenca tovrstnih metod slaba. Zato se ponavadi uporabljajo posebni algoritmi in metode, ki delajo le za iskanje ničel polinomov. Prva metoda je

zelo preprosta obenem pa tudi (skoraj) vedno konvergira. Polinome zapišemo v obliki pridružene matrike (*Companion matrix*) [**?**]. Za poljuben polinom $p(x)$, lahko z njegovimi koeficienti sestavimo sledečo matriko

$$C(p) = \begin{bmatrix} 0 & 0 & \dots & 0 & -a_0 \\ 1 & 0 & \dots & 0 & -a_1 \\ 0 & 1 & \dots & 0 & -a_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & 1 & -a_{n-1} \end{bmatrix} \tag{2.6}$$

$a_0, a_1, ...a_k \in \vec{a}$ predstavljajo koeficiente polinoma. Obstaja več zapisov za pridruženo matriko. Zapis matrike (2) se uporablja najpogosteje. Ničle polinoma tovrstne matrike so lastne vrednosti matrike, torej

$$C(p)\vec{x} = \lambda\vec{x} \tag{2.7}$$

λ predstavlja lastno vrednost. Problem iskanja ničel se torej spremeni v problem iskanja lastnih vrednosti. Seveda pa za polinome zelo visokega reda tovrstni algoritem ni primeren, ker velja, da je časovna kompleksnost algoritmov za iskanje lastnih vrednosti razpršenih matrik (matrik z veliko ničlami) enaka $\mathcal{O}(ck^2)$. k v izrazu za kompleksnost predstavlja stopnjo polinoma, ter red matrike (matrika je reda $k \times k$).

Primer 3 *Matrika je v računalništvu le več dimenzionalni seznam. Razpršena matrika (na levi) je več dimnezionalni seznam, ki vsebuje veliko ničel, medtem ko gosta matrika (na desni) vsebuje predvsem ne ničelne elemente.*

$$\begin{bmatrix} 34 & 12 & 0 & 0 & 0 & 0 \\ 0 & 7856 & 68 & 0 & 0 & 0 \\ 0 & 0 & 456 & 54 & 0 & 0 \\ 0 & 0 & 0 & 654 & 227 & 0 \\ 0 & 0 & 0 & 0 & 45 & 23 \\ 0 & 0 & 0 & 0 & 0 & 54 \end{bmatrix} \qquad \begin{bmatrix} 34 & 12 & 32 & 1 & 32 & 43 \\ 37 & 7856 & 68 & 76 & 4 & 2 \\ 54 & 324 & 456 & 54 & 34 & 41 \\ 65 & 34 & 456 & 654 & 227 & 675 \\ 43 & 456 & 435 & 87 & 45 & 23 \\ 23 & 45 & 234 & 34 & 21 & 54 \end{bmatrix} \tag{2.8}$$

Mi se ne bomo preveč osredotočali na algoritme za iskanje lastnih vrednost in preprosto uporabili algoritem, ki ga ponuja `scipy`. Zaradi konvergence in hitrost algoritma za manjše matrike se ta metoda velikokrat uporablja za ovrednotenje rezultatov brskanja na različnih brskalnikih (Google, Bing, ...), kot primarno metodo jo pa uporablja tudi *Matlab*.

Algoritem 3 Izračun večih ničel

Require: p

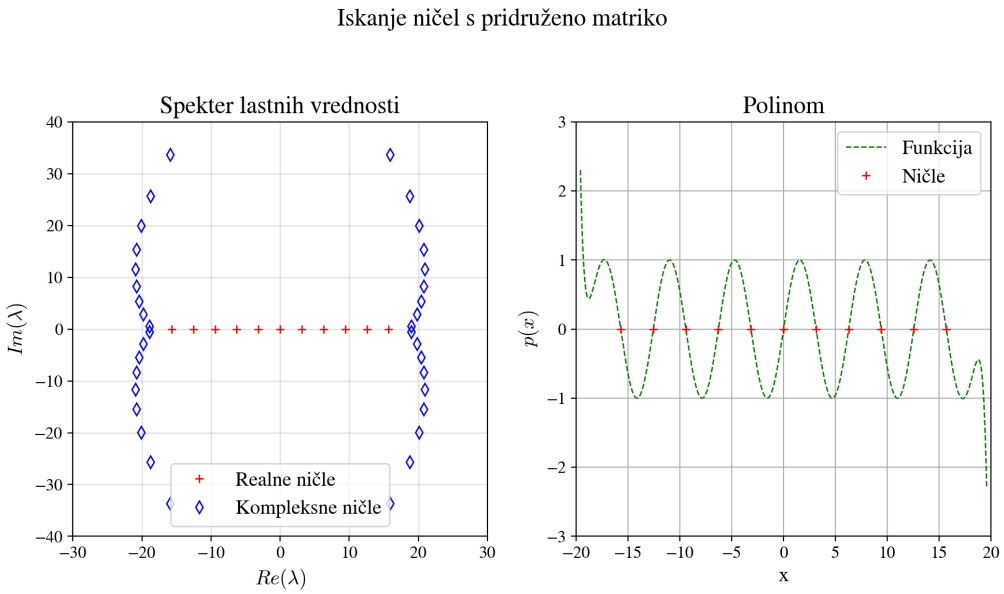
$C \leftarrow$ GenerateMatrix(p) ▷ Ustavri pridruženo matriko iz polinoma p

$D \leftarrow$ DenseMatrix(p) ▷ Matriko **lahko** iz razpršene pretvorimo v gosto

return FindEigenValues($C \vee D$) ▷ Najdi lastne vrednosti matrike

Iskanje lastnih vrednosti matrike lahko opravimo na tako razpršenih kot gostih matrikah. Ponavadi moramo za iskanje **vseh lastnih vrednosti** uporabiti gosto matriko, ker je konvergenca algoritmov za razpršene matrike slabša. V primeru, da iščemo le nekaj lastnih vrednosti (oz. ničel) pa so algoritmi, kot so Laczosev in Arnoldijev algoritem za razpršene matrike seveda boljši [**?**]. Ko govorimo o pretvorbi iz razpršene v gosto matriko govorimo o načinu shranjevanja matrike na notranjem pomnilniku *in ne o matematičnem pomenu tovrstnih matrik*. Razpršena matrika je na notranjem pomnilniku shranjena v obliki manjšega seznama (*array*). Elementi sesznama so seznami oblike {`int x, int y, double st`}. Torej je v resnici ta seznam le $n \times 3$ matrika. V algoritmih, kjer nastopajo razpršene matrike, so tako vsi elementi pridružene matrike, ki niso v množici razpršene matrike opredeljeni z 0. To nam omogoča, da prihranimo na prostoru, ki ga matrika zasede na notranjem pomnilniku. Če uporabljamo goste matrike so pa v nasprotju s razpršenimi matrikami na notranjem pomnilniku zapisani tudi elementi s številom 0. Seveda za matrike, ki niso razpršene, razpršena matrika vzame več prostora na notranjem pomnilniku. Obenem pa so algoritmi, ki so definirani v razredih razpršenih matrik, namenjeni le razpršenim matrikam in zato so matematično hitrejši le v primeru, da so matrike res razpršene [**?**]. Nesemo pozabiti tudi, da so moderne centralne procesne enote optimizirane za množenje gostih matrika.

Primer 4 *Poišči vse ničle polnoma podanega s predpisom* $p(x) = \sum_{n=0}^{24}(-1)^n \frac{x^{2n+1}}{(2n+1)!}$ *s pomočjo pridružene matrike. Izvorna koda je v prilogi.*



Slika 2.3: Potek algoritma s pridruženo matriko

Opazimo lahko, da je vsaka lastna vrednost (Slika **??***), ki ima le realni del na spektru, tudi ničla polinoma. To tudi potrjuje pravilnost našega algoritma. Za iskanje lastnih vrednosti smo uporabili kar funkcijo* `eigs`*, ki jo ponuja* `numpy`*. Matirka je sicer razpršena, zato bi bila hitrejša metoda verjetno* `sparse.linalg.eigs`*, ki jo ponuja* `scipy`*.*

Metoda pridružene matrike je sicer res iterativna, ker se lastne vrednosti ponavadi iščejo iterativno, vendar težko rečemo, da je direktno iterativna kot prejšne metode. Za polinome so bile razvite še druge posebne iterativne metode za iskanje več ničel. Metode konvergirajo veliko hitreje, kot metoda pridružene matrike. Obenem pa konvergirajo za (skoraj) vsak začetni približek. Mi se bomo osredotočili predvsem na dve metodi Laguerreovo in Durand-Kernejevo. Obstaja še veliko metod, tudi nekaj novejši, ki so v nekaterih primerih hitrejše.

2.2.1 Laguerreova metoda

Laguerreova metoda je najpreprostejša metoda za izračun ničel polinomov. Za polno izpeljavo bi potrebovali nekaj matematičnih dokazov o občutljivosti ničel, ki jih ne bomo opisali. Izreki in moderna modificirana implementacija so opisani v [**?**]. Če zapišemo polinom v faktorizacijski obliki dobimo $p(x) = \alpha_k(x - x_1)(x - x_2)...(x - x_n)$. Definiramo dve vsoti vrst (izpeljani v [**?**]):

$$S_1(x) = \sum_{i=1}^{n} \frac{1}{x - x_i} = \frac{p'(x)}{p(x)} \tag{2.9}$$

$$S_2(x) = \sum_{i=1}^{n} \frac{1}{(x - x_i)^2} = -S_1'(x) = \frac{p'^2(x) - p(x)p''(x)}{p^2(x)} \tag{2.10}$$

S pomočjo manipulacije teh vrst [**?**] lahko nato izrazimo iteracijsko funkcijo $\mathcal{I}$.

$$x_{r+1} = x_r - \frac{n}{S_1 \pm \sqrt{(n-1)(nS_2 - S_1^2)}} \tag{2.11}$$

Zaporednje x_{r+1} nam vrne približke ničle polinoma. Obstaja zelo malo primerov, kjer ta metoda konvergira počasi zato je pogosto uporabljena kot primerna metoda za izračun ničle polinomov višjega reda. Za izračun več ničel s tovrstno metodo je potrebno spreminjat začetni približek x_0. Prednost te metode je, da za razliko od Newtonove metode, ta metoda hitreje konvergira proti ničli, in da najde skoraj vse ničle, četudi so te skupaj (to je pogojeno z natančnostjo plavajočih vejic, če so ničle preveč skupaj jih ne moramo razlikovati).

Algoritem 4 Laguerreova Metoda

procedure NajdiNičlo(x_r)
  while $|f(x_r)| > \epsilon$ **do** ▷ Iteriraj dokler ni $p(x_r)$, v okolici ničle
    $d \leftarrow f'(x_r)$ ▷ Odvjaja diskretno ali analtično, če je možno
    $d_2 \leftarrow f'(f'(x_r))$
    $S_1 \leftarrow \frac{d}{f(x_r)}$
    $S_2 \leftarrow \frac{d^2 - f(x_r)d_2}{f^2(x_r)}$
    $x_r \leftarrow x_r - \frac{n}{S_1 \pm \sqrt{(n-1)(nS_2 - S_1^2)}}$
  end while
  return x_r
end procedure

procedure VečNičel(i_{max}, x_z, h)
  $\mathbb{S} \leftarrow \{\}$
  while $k \leq n$ **do** ▷ Iteriraj doklar ni število ničel enako stopnji polinoma
    if $i > i_{max}$ **then** ▷ Meja iteraciji, če algoritem ne konvergira
      return $\mathbb{S}$
    end if
    $x_r \leftarrow x_r \pm h$ ▷ Nov začetnik približek
    $i \leftarrow i + 1$
    $x \leftarrow$ NajdiNičlo(x_r)
    if $x \notin \mathbb{S}$ **then** ▷ Poglej če je ničla že v množici ničel
      $\mathbb{S} + \{x\}$
    end if
  end while
end procedure

2.2.2 Durand–Kernerjeva metoda

Zadnja metoda, ki jo bomo opisali je Durdand-Kernerjeva metoda. Metoda je samo po sebi dokaj kompliciranа, cel matematičen postopek je opisan v [**?**]. Mi bomo metodo opisal preprosto. Za polinom $p(x) = \alpha_k(x - x_1)(x - x_2)...(x - x_k)$ iščemo popravek oz. približek Δx_i. Torej

$$p(x) = (x - (x_1 + \Delta x_1))(x - (x_2 + \Delta x_2))...(x - (x_k + \Delta x_k)) \tag{2.12}$$

Iz izraza izpostavimo približek Δx in zanemarimo višje člene (postopek je opisan v [**?**, **?**]). Iz izraza lahko sedaj izpostavimo Δx_n in dobimo.

$$\left.\Delta x_n = \frac{-p(x_n)}{\prod_{i=1, i\neq n}^{k} (x_n - x_i)}\right\} \quad n \textit{ predstavlja eno izmed ničlo} \tag{2.13}$$

Sedaj lahko to zapišemo v obliki iteracijske funkcije $\mathcal{I}$

$$x_i^{r+1} = x_i^r - \Delta x_i(x_r). \tag{2.14}$$

Dobimo preprosto iteracijo za izračun vseh ničel. Tovrstni algoritem nam omogoča, da imamo za vsako ničlo posebej prvotni približek, ki nato konvergira proti vsaki ničli posebej. Potemtakem ne potrebujemo več pregledovati pogoja če je ničla že v množici (seznamu) ničel. Upoštevati moramo, da morajo biti začetni x_r paroma različni sicer algoritem divergira, ker se v imenovalcu izraza za Δx_n pojavi nič. Obstajajo tudi druge izepljave algoritmov, ki omogočajo bolj stabilna konvergenco. Opazili bomo, da nekatere ničle metode za slabe začetne približke hitro divergirajo.

Algoritem 5 Durand–Kernerjeva metoda

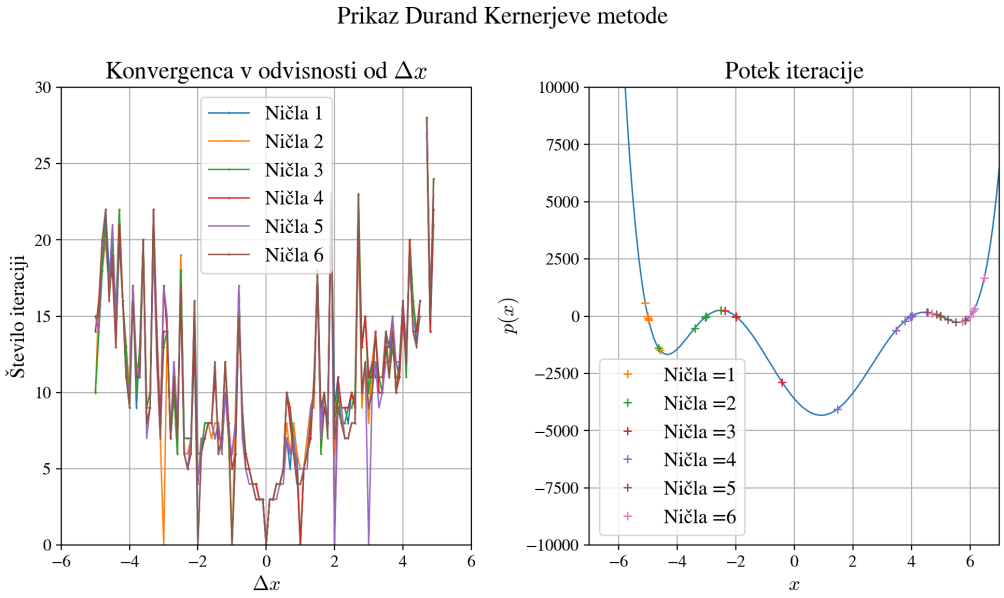
Require: $k, n_{max}, \vec{x}^1$ ▷ Red polinoma, maksimalno št. iteraciji in prvotni približek
  while $n < n_{max}$ **do**
    for $i \leftarrow 1$ to k **do** ▷ Iteriraj skozi vse ničle
      if $|p(x_i^n)| \leq \epsilon$ **then** ▷ Iteriraj dokler ni $p(x_r)$, v okolici ničle
        return x_i^n
      end if
      $p = 1$ ▷ Pripravi p za izračun produkta $\prod_{i=1, i\neq n}^{k}(x_n - x_i)$
      for $h \leftarrow 1$ to k **do**
        if $h \neq i$ **then**
          $p \leftarrow p \cdot (x_i^n - x_h^n)$
        end if
      end for
      $\Delta x_i \leftarrow \frac{f(x_i)}{p}$
      $x_i^{n+1} = x_i^n - \Delta x_i$
    end for
  end while

Kot že omenjeno obstajajo tudi druge implimentacije. Najbolj poznana implimentacija je definitvno kvadratna implimetacija. V imenovalcu v tej metodi zapišem

$$\Delta x_n^{r+1} = \frac{-p(x_n)}{\prod_{i=1}^{n-1}(x_n^r - x_i^{r+1}) \prod_{i=n+1}^{k}(x_n^r - x_i^r)} \tag{2.15}$$

Tovrstna implimetacija izboljša konvergenco. Mi se bomo scier držali orginalne implimetacije, ker ima kvadratna metoda svoje probleme. Problemi se izražajo predvsem v obliki hitrih divergence ničel z slabim začetnim približkom.

Primer 5 *Poišči vse ničle polnoma poljubnega polinoma s Durand Kernerjevo metodo. Izvorna koda je v prilogi.*



Slika 2.4: Potek Durand-Kernerjevega algoritma

Δx predtsavlja razdaljo od ničle do prvotnega približka. Opazimo, da metoda deluje. Pričakovano potrebuje metoda več iteraciji gelde na odaljenost pvotnega približka od ničle. Zanimivo je mogoče dejstvo, da če so prvotni približki preveč skupaj algoritem zelo hitro divergira. Obenem hitro divergira tudi če niso prvotni približki dobro sortirani. Več o sami konvergenci bomo povedali v 4. poglavju.

Poglavje 3

Algoritmi v $\mathbb{R}^n$

Do sedaj smo omenili le algoritme, ki iščejo ničle funkciji z 1 spremenljivko. Pogosteje pa se znajdemo pri problemih funkciji, ki so odvisne od več spremenljivk. Algoritmi za iskanje ničel funkciji več spremenljiv so še posebej pomembni v optimizacijskih problemih, kjer moram ponavadi minimizirat funkcijo na katero vpliva več faktorjev. Naj bo $p : \mathbb{R}^n \to \mathbb{R}$ polinom. Predpis polinoma je

$$p(x_1, x_2, x_3, ..., x_n) = \sum_{h=0}^{x_{1s}} \alpha_{1,h} x_1^h + \sum_{h=0}^{x_{2s}} \alpha_{2,h} x_2^h + ... + \sum_{h=0}^{x_{ns}} \alpha_{n,h} x_n^h = \sum_{\ell=1}^{n} \sum_{h=0}^{x_{\ell s}} \alpha_{\ell,h} x_\ell^h \tag{3.1}$$

Obstaja le peščica algoritmov, ki lahko izračunajo ničle tovrstnih funkciji. Računalniško gledano se največkrat uporablja posplošitev bisekcije. Namesto na interval se sedaj osredotočamo na regijo v prostoru. Algoritma je le posplošitev že izpeljanega algoritma, zato ga ne bomo izpeljali še enkrat. Namesto mej intervala sedaj izberemo mejne točke, ki predstavljajo regijo, tako da ima na mejnih točka funkcija nasprotna si predznaka, nato regijo manjšamo po vseh prostorskih spremenljivkah. Obstaja tudi algoritem, ki je posledica Poincare-Mirandovega izreka [**?**], vendar je algoritem zapleten.

V optimizacijskih problemih se več ne išče ničel funkcije odvoda, temveč operatorjev, ki delujejo na funkcijo. Najbolj znana je Newtonov algoritem konjugiranega gradianta, ki ga bomo kot edinega predstavili. Obstajajo pa

seveda bolj kompleksni algoritmi kot so konjugirana vektorska metoda, Broyden–Fletcher–Goldfarb–Shanno algoritem,Nelder-Mead simpleks,...

3.0.1 Newtonova metoda konjugiranega gradienta

Za poljubno funkcijo $f : \mathbb{R}^n \to \mathbb{R}$ velja, da jo lahko zapišem z Taylorjevo vrsto, kot kvadratno aproksimacijo

$$f(\vec{x} - \vec{a}) \approx f(\vec{a}) + (Jf)(a)(\vec{x} - \vec{a}) + \frac{1}{2}(\vec{x} - \vec{a})^T (Hf)(\vec{a})(\vec{x} - \vec{a}) \tag{3.2}$$

H predstavlja Hesejevo matriko (matriko drugih parcialnih odvodov), J predstavlja Jacobijevo matriko (matriko prvih parcialnih odvodov). Jacobijeva matrika je v primeru skalarnega polja torej $\mathbb{R}^n \to \mathbb{R}$, le gradient. Gradient ∇f je preprosto vektor, ki kaže proti spremembi funkcije. Mi želimo izvedeti, kje ta funkcija doseže minimum, torej $\nabla f = 0$. Iz zgornje enačbe preprosto izrazimo $(Jf)(a)(x - a)$.

$$(Jf)(a)(\vec{x} - \vec{a}) = f(\vec{x} - \vec{a}) - f(\vec{a}) - \frac{1}{2}(\vec{x} - \vec{a})^T (Hf)(\vec{a})(\vec{x} - \vec{a}) \tag{3.3}$$

Sedaj primerjamo to z 0, da dobimo lokalni minimum (naj bo zamik $a = \vec{0}$)

$$f(\vec{x} - \vec{a}) - f(\vec{a}) - \frac{1}{2}(\vec{x} - \vec{a})^T (Hf)(\vec{a})(\vec{x} - \vec{a}) = 0 \xrightarrow{izepljava} (Hf)(\vec{x}) = -f(\vec{x}) \tag{3.4}$$

$(Hf)(\vec{x}) = -f(\vec{x})$ je sedaj linearni problem, ki ga rešimo z algoritmom za reševanje sistema linearnih enačb. **Ta metoda je izredno pomembna v strojnem učenju!**

Poglavje 4

Primerjava algoritmov in rezultati

4.0.1 Problemi v informatiki in računalništvu

Poglavje 5

Zaključek

Zahvala

Zahvaljujem se mentorju mag. Tomi Zebiču za pomoč in nasvete pri pisanju seminarske naloge.

Viri in literatura

[1] Floating-Point Formats — quadibloc.com. `http://www.quadibloc.com/comp/cp0201.htm`. [Dostopano 10-09-2023].

[2] Ieee standard for floating-point arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (2019), 1–84.

[3] Bohte, Z. *Numerično reševanje nelinearnih enačb*, vol. 34. Društvo matematikov, fizikov in astronomov Slovenije, 1993. 1.000 izv.

[4] Cameron, T. An effective implementation of a modified laguerre method for the roots of a polynomial. *Numerical Algorithms 82* (11 2019).

[5] Kulpa, W. The poincare-miranda theorem. *The American Mathematical Monthly 104*, 6 (June 1997), 545.

[6] Plestenjak, B. *Numerične Metode*. Fakulteta za Matematiko in Fiziko, 2010.

[7] Plestenjak, B. *Iterativne numerične metode v linearni algebri*. Fakulteta za Matematiko in Fiziko, 2021.

[8] Ramond, P. The abel–ruffini theorem: Complex but not complicated. *The American Mathematical Monthly 129*, 3 (mar 2022), 231–245.

[9] Scott, J., and Tůma, M. *An Introduction to Sparse Matrices*. 01 2023, pp. 1–18.

[10] Scott, L. R. *Numerical Analysis*. Princeton University Press, 2011.

Priloga A

Algoritmi

Biskecija

Newtonova metoda

Metoda pridružene matrike