Prva domača naloga

Matej Klančar (63200136)

1 Uvod

To poročilo predstavlja rešitev prve domače naloge, ki se osredotoča na dva osrednja problema: učinkovito shranjevanje redkih matrik in iterativno reševanje linearnih sistemov.

V prvem delu je opisana implementacija podatkovnega tipa `RedkaMatrika` v jeziku Julia, vključno z osnovnimi operacijami, kot sta indeksiranje in množenje z vektorjem. V nadaljevanju je predstavljena implementacija metode SOR (Successive Over-Relaxation) za reševanje sistema $A\boldsymbol{x} = \boldsymbol{b}$.

Poročilo se zaključi z analizo, v kateri za testni primer poiščemo optimalni relaksacijski parameter ω in prikažemo njegov vpliv na hitrost konvergence.

2 Implementacija

V tem poglavju bomo podrobneje predstavili implementacijo podatkovnega tipa `RedkaMatrika` v programskem jeziku Julia. Cilj je bil ustvariti učinkovito predstavitev za matrike z velikim številom ničelnih elementov.

2.1 Podatkovna struktura `RedkaMatrika`

Osnova naše implementacije je podatkovna struktura `RedkaMatrika`. Struktura vsebuje dve polji:

- `V`: Matrika, ki hrani vrednosti neničelnih elementov. Vsaka vrstica v `V` ustreza enakoležni vrstici v redki matriki.
- `I`: Matrika, ki hrani stolpčne indekse neničelnih elementov. Struktura te matrike je enaka matriki `V`.

Element na mestu `V[i, k]` hrani vrednost neničelnega elementa v `i`-ti vrstici redke matrike, njegov stolpčni indeks pa je shranjen na mestu `I[i, k]`. Mesta, ki še niso zasedena, imajo v matriki `I` vrednost 0.

2.2 Indeksiranje

Za smiselno uporabo našega tipa je bilo treba implementirati standardni vmesnik za indeksiranje, ki ga v Julii predstavljata funkciji `getindex` in `setindex!`.

Pridobivanje elementov (`Base.getindex`)

Funkcija `getindex` omogoča branje vrednosti z običajno sintakso `T[i, j]`. Implementacija najprej poišče vrstico v matriki indeksov `T.I`, ki ustreza iskani vrstici `i`. Nato v tej vrstici s funkcijo `findfirst` poišče iskani stolpčni indeks `j`.

- Če `findfirst` vrne `nothing`, stolpec `j` ni med shranjenimi indeksi za vrstico `i`. Element je torej enak nič.
- Če `findfirst` vrne pozicijo `index`, funkcija na tej isti poziciji v matriki vrednosti `T.V` poišče in vrne shranjeno vrednost `T.V[i, index]`.

Nastavljanje elementov (`Base.setindex!`)

Funkcija `setindex!` omogoča nastavljanje vrednosti s sintakso `T[i, j] = v`. Implementacija je kompleksnejša, saj mora obravnavati več primerov.

1. **Spreminjanje obstoječega neničelnega elementa:** Najprej preverimo, ali na mestu `(i, j)` že obstaja neničelni element (z uporabo naše funkcije `getindex`). Če obstaja, poiščemo njegov indeks in posodobimo vrednost v matriki `V`.
2. **Dodajanje novega neničelnega elementa:** Če element na mestu `(i, j)` ne obstaja (je 0), moramo zanj poiskati prostor.
   - **Prosto mesto obstaja:** V `i`-ti vrstici matrike `I` poiščemo prvo prosto mesto, označeno z `0`. Na to mesto vpišemo nov stolpčni indeks `j`, na ustrezno mesto v matriki `V` pa vrednost `v`.
   - **Ni prostega mesta:** Če so vsa mesta v vrstici že zasedena, matriki `V` in `I` dinamično razširimo z dodajanjem novega stolpca. Nato novo vrednost in indeks vpišemo na novo vstavljeno mesto.

2.3 Množenje z vektorjem

Algoritem deluje po definiciji matričnega množenja. Ustvarimo nov ničelni vektor `b`. Nato z dvema gnezdenima zankama iterira čez vse elemente `(i, j)` navidezne polne matrike. V vsakem koraku izračuna produkt `T[i, j] * v[j]` in ga prištejemo `i`-ti komponenti rezultata `b[i]`.

3 Reševanje linearnih sistemov z metodo SOR

Za reševanje linearnih sistemov oblike $A\boldsymbol{x} = \boldsymbol{b}$, kjer je A naša `RedkaMatrika`, smo implementirali iterativno metodo SOR. Ta je posebej primerna za velike, redke sisteme, saj iterativno izboljšuje približek rešitve.

3.1 Algoritem metode SOR

Metoda SOR posodobi vsako komponento vektorja rešitve $\boldsymbol{x}$ v koraku $k+1$ po formuli:

$$x_i^{(k+1)} = (1-\omega)x_i^{(k)} + \frac{\omega}{A_{ii}} \left(b_i - \sum_{j<i} A_{ij} x_j^{(k+1)} - \sum_{j>i} A_{ij} x_j^{(k)} \right). \tag{1}$$

Ključna značilnost metode je, da pri izračunu nove vrednosti $x_i^{(k+1)}$ takoj uporabi že posodobljene komponente iz iste iteracije (za $j < i$). Parameter ω (omega) uravnava hitrost konvergence.

3.2 Implementacija

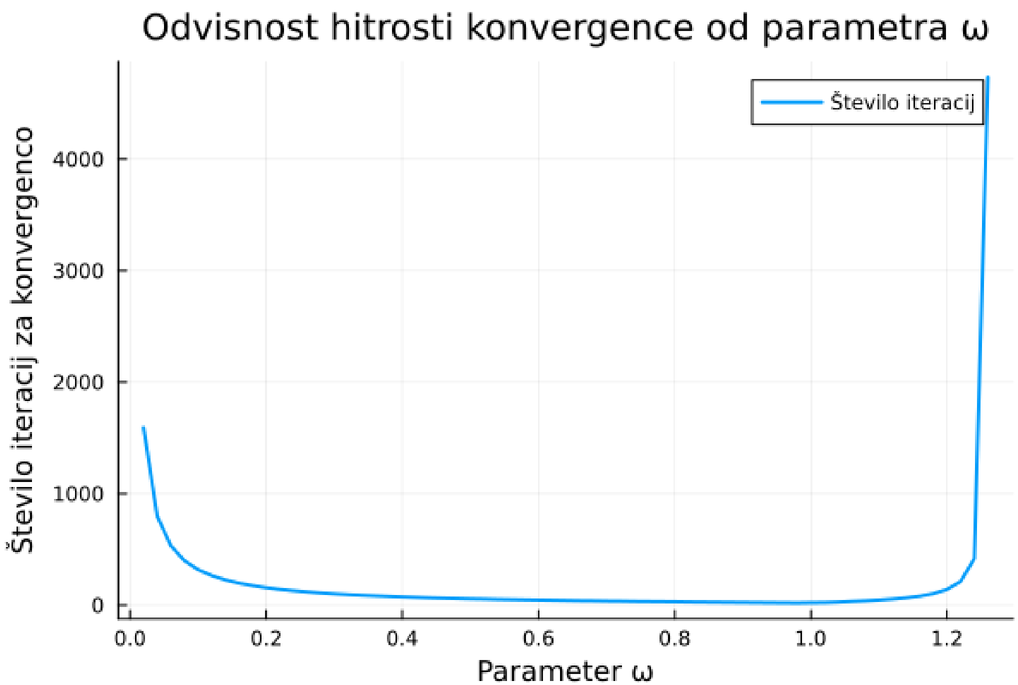
Naša implementacija je sestavljena iz dveh glavnih delov:

1. **Izvedba enega koraka iteracije:** Pomožna funkcija `sor_korak` izvede en sam prehod čez vse komponente vektorja $\boldsymbol{x}$ v skladu s formulo (1). Za vsako komponento x_i izračuna vsoto produktov $A_{ij}x_j$ za $j \neq i$ in nato posodobi vrednost x_i. Ker se vektor rešitve posodablja sproti znotraj istega koraka, so pri izračunu za x_i že uporabljene nove vrednosti za vse $j < i$.
2. **Glavna zanka in pogoj za ustavitev:** Glavna funkcija `sor` zaporedno kliče `sor_korak` in po vsaki iteraciji preverja pogoj za ustavitev. Iteracija se ustavi, ko je neskončna norma vektorja ostanka, $\|A\boldsymbol{x}^{(k)} - \boldsymbol{b}\|_\infty$, manjša od podane tolerance `tol`. To v praksi pomeni, da se ustavi, ko največja absolutna vrednost katerekoli komponente vektorja ostanka pade pod toleranČno mejo.

Funkcija sprejme matriko sistema A, vektor desnih strani $\boldsymbol{b}$, začetni približek $\boldsymbol{x_0}$, parameter ω in toleranco `tol`. V primeru uspešne konvergence vrne končni približek rešitve $\boldsymbol{x}$. Za preprečevanje neskončnega izvajanja v primeru nekonvergence je vgrajena tudi omejitev največjega števila iteracij.

4 Analiza optimalnega parametra ω

Za metodo SOR je ključnega pomena pravilna izbira parametra ω, saj ta neposredno vpliva na hitrost konvergence. Da bi poiskali optimalno vrednost za naš testni primer, smo izvedli numerično analizo. Metodo SOR smo pognali za različne vrednosti ω v intervalu $(0, 2)$ in zabeležili število iteracij, potrebnih za dosego tolerance 10^{-10}.



Slika 1: Odvisnost števila iteracij od vrednosti parametra ω.

Rezultati, prikazani na sliki 1, kažejo jasno odvisnost hitrosti konvergence od izbire ω. Ugotovili smo, da je za obravnavani sistem optimalna vrednost parametra $\omega = 0.98$. Pri tej vrednosti je metoda konvergirala v **21 iteracijah**. Za ω večji ali enak 1.28 metoda ni konvergirala.