O optimizaciji programskoga kôda iteriranoga Ritzova postupka

Fresl, Krešimir; Šamec, Elizabeta; Gidak, Petra

DOI: https://doi.org/ 10.5592/CO/YODA.2019.1.2

O optimizaciji programskoga kôda iteriranoga Ritzova postupka

Krešimir Fresl, Elizabeta Šamec, Petra Gidak
Sveučilište u Zagrebu, Građevinski fakultet

Sažetak

U radu su prikazana tri primjera optimizacije kôda u realizaciji iteriranoga Ritzova postupka (IRM) i njegove inačice istovrijedne metodi konjugiranih gradijenata (IRM–CG) u programskom jeziku C++. Nakon opisa osnovnih shema pohranjivanja rijetko popunjenih matrica, uključujući posebnosti simetričnih matrica, prikazano je izračunavanje umnoška rijetko popunjene matrice i vektora, izračunavanje umnoška rijetko popunjene i pune matrice te ispitivanje zadovoljenja uvjeta za prekid iteracije. Optimizacije su provedene radi ublažavanja „uskih grla" u algoritmima IRM-a i IRM–CG-a.

Ključne riječi: iterirani Ritzov postupak, metoda konjugiranih gradijenata, rijetko popunjena matrica, optimizacija programskoga kôda

On programme code opimisation for Iterated Ritz Method

Abstract

In the paper, three code optimisation examples in implementation of Iterated Ritz Method (IRM) and nonrecursive equivalent of conjugate gradient method (IRM–CG, as a special case of IRM) are presented in programming language C++. After describing the basic storage schemes for sparse matrices, including particularities of symmetric matrices, calculation of sparse matrix and vector product, calculation of sparse and full matrix product and testing the condition for terminating the iteration are shown. Optimisation was carried out to mitigate bottlenecks in IRM and IRM–CG algorithms.

Key words: Iterated Ritz Method, conjugate gradient method, sparse matrix, programme code optimisation

1. Uvod

U radovima [2], [3], [5] autori su pokazali da je iterirani Ritzov postupak (IRP ili IRM, od engl. *Iterated Ritz Method*) vrlo učinkovit za rješavanje izrazito velikih sustava linearnih jednadžbi

$$\mathbf{Ax} = \mathbf{b}. \tag{1}$$

U rješavanju malih sustava brzina toliko ne dolazi do izražaja, budući da je i primjena manje učinkovitih postupaka dovoljno kratkotrajna.
Sustavi nelinearnih jednadžbi rješavaju se iteracijskim postupcima. U većini se postupaka te jednadžbe na neki način zamjenjuju ili aproksimiraju linearnima. Primjerice, interpretiramo li Newton–Raphsonov postupak geometrijski, reći ćemo da se hiperplohe, koje su grafovi nelinearnih funkcija, zamjenjuju dirnim hiperravninama. Kako su ti postupci iteracijski, u svakomu koraku postupka nastaje sustav linearnih jednadžbi koji treba riješiti. Slično tome, pri nalaženju oblika kabelskih mreža i konstrukcija od platna nelinearne se jednadžbe ravnoteže uvođenjem gustoća sila i gustoća naprezanja linearizira ju [12], [10]. Primijenimo li metodu gustoća sila iteracijski [6], [14], sustavi linearnih jednadžbi rješavat će se veći broj puta. Brzina rješavanja sustava linearnih jednadžbi ponovno će stoga dobiti značajnu ulogu.
U sljedećemu je odjeljku prikazana programska realizacija algoritama IRM i njegove inačice istovrijedne metodi konjugiranih gradijenata u jeziku C++ uz primjenu matrične biblioteke Eigen. U odjeljcima 4., 5. i 6. opisani su primjeri optimizacije kôda za izračunavanje umnoška rijetko popunjene matrice i vektora, za izračunavanje umnoška rijetko i gusto popunjene matrice te za kriterij prekida iteracije. Odjeljak 3. svojevrstan je uvod u odjeljke 4. i 5.; u njemu su opisane osnovne sheme pohranjivanja rijetko popunjenih matrica.

2. Realizacija u programskom jeziku C++

Autori algoritma IRM programski su kôd razvili u programskom jeziku `Fortran 90` (podrobnije, primjerice, u odjeljku 8. u [2]). Programska je podloga ovoga rada realizacija algoritma u kôdu napisanomu u jeziku C++ [13] uz primjenu biblioteke matričnih razreda i funkcija `Eigen` [8]. U oblikovanju i programskoj realizaciji biblioteke primijenjene su neke napredne sastavnice jezika C++, ponajprije razredi–predlošci (engl. *class templates*), nasljeđivanje razreda (engl. *inheritance*) i njihove međuigre (primjerice, nasljeđivanje razreda kojemu je izvedeni razred parametar–predložak, nazvano *curiously recurring template pattern*) te preopterećivanje operatora (engl. *operator overloading*) uz predloške izrazâ (engl. *expression templates*) [15].
Sažetosti i čitljivosti kôda radi uvest ćemo nazive za tipove podataka:

```
typedef int index_t;    // cijeli brojevi
typedef double scalar_t;    // pseudorealni brojevi
typedef Matrix<scalar_t, Dynamic, 1> vector_t;    // vektori
typedef Matrix<scalar_t, Dynamic, Dynamic, ColMajor> dense_matrix_t;
                                    // gusto popunjene matrice
typedef SparseMatrix<scalar_t, ColMajor> sparse_matrix_t;
                                    // rijetko popunjene matrice
```

Primjena biblioteke Eigen omogućava gotovo doslovan prijepis pseudokôda (iz, primjerice, [4]) u programski kôd:

```
1   vector_t solve (sparse_matrix_t const& A,   // matrica sustava
2                   vector_t const& b,    // vektor slobodnih članova
3                   vector_t const& x0,    // pretpostavka rješenja
4                   scalar_t tolerance,    // točnost
5                   index_t max_steps    // dopušteni broj koraka
6                  ) {
7     index_t i = 0;
8     vector_t x = x0;
9     vector_t r = b - A * x;    // rezidual
10    scalar_t q = r.dot (r) / r.dot (A * r);
11    vector_t p = q * r;    // prirast rješenja: najstrmiji silazak
12    while (r.norm() >= tolerance * b.norm() && i <= max_steps) {
13      x += p;    // poboljšana aproksimacija rješenja
14      r = b - A * x;
15      dense_matrix_t F = [φ_0 φ_1 ··· φ_{m-1}];    // baza potprostora
16      dense_matrix_t FTAF = F.transpose() * A * F;
17                                      // matrica maloga sustava
18      vector_t FTr = F.transpose() * r;
19                                  // slobodni članovi maloga sustava
20      LLT< Ref<dense_matrix_t> > llt (FTAF);
21                                      // rastav prema Choleskomu
22      a = llt.solve (FTr);    // uvrštavanje unazad
23      p = F * a;    // prirast rješenja
24      ++i;
25    }
26    return x;
27  }
```

(Ipak, redak 15. nije programski kôd, nego sažeta zamjena za nekoliko zamršenijih i nečitljivijih redaka u kojima se oblikuju bazni vektori potprostora i svrstavaju u matricu F.)

Poseban slučaj, ali i inačica iteriranoga Ritzova postupka postupak je koji su autori nazvali nerekurzivnom istovrijednicom postupka konjugiranih gradijenata, s akronimom IRM–CG [4]. Potprostor pretraživanja razapinju (samo) dva vektora: trenutačni rezidual i prirast rješenja u prethodnomu koraku. S pomoću nekoliko algebarskih akrobacija, podrobnije opisanih u [4], autori su umnožak triju matrica u retku 16. zamijenili umnoškom matrice i vektora te izbjegli množenje matrice i vektora u retku 14., tako da petlja, kao i u klasičnoj metodi konjugiranih gradijenata, sadrži samo jedno množenje matrice i vektora:

```
1    vector_t solve (sparse_matrix_t const& A,
2                    vector_t const& b,
3                    vector_t const& x0,
4                    scalar_t tolerance,
5                    index_t max_steps
6                   ) {
7      index_t i = 0;
8      vector_t x = x0;
9      vector_t r = b - A * x;
10     vector_t alpha = A * r;
11     scalar_t q = r.dot (r) / r.dot (alpha);
12     vector_t p = q * r;
13     vector_t beta = A * p;
14     while (r.norm() >= tolerance * b.norm() && i <= max_steps) {
15       x += p;
16       r -= beta;
17       alpha = A * r;    // jedino množenje matrice i vektora
18       scalar_t FTAF00 = r.dot (alpha);
19       scalar_t FTAF11 = p.dot (beta);
20       scalar_t FTAF01 = r.dot (beta);
21       scalar_t FTr0 = r.dot (r);
22       scalar_t detFTAF = FTAF00 * FTAF11 - FTAF01 * FTAF01;
23       scalar_t a0 = FTr0 * FTAF11 / detFTAF;
24       scalar_t a1 = -FTr0 * FTAF01 / detFTAF;
25       p = a0 * r + a1 * p;
26       beta = a0 * alpha + a1 * beta;
27       ++i;
28     }
29     return x;
30   }
```

`FTAF00, FTAF11 i FTAF01` komponente su matrice `FTAF` (redak 16. kôda za IRM); komponenta `FTAF10` jednaka je komponenti `FTAF01`. `FTr0` prva je komponenta vektora `FTr` (redak 18. kôda za IRM); druga je komponenta jednaka nuli. Redci 22., 23. i 24. prijevod su u kôd izrazâ za rješenje sustava dviju jednadžbi primjenom determinanata (što je za tako mali sustav optimalan način rješavanja).

3. Rijetko popunjene matrice

Razmjerno je mali broj komponenata matrica, struktura kojih ocrtava povezanost čvorova u mrežama kabelâ, konačnih razlika ili konačnih elemenata, različit od nule, pa se u memoriju računala pohranjuju samo te, od nule različite komponente. Takve se matrice nazivaju rijetko popunjenima.

Nekoliko je shema pohranjivanja rijetko popunjenih matrica [11]. Najjednostavniji je koordinatni zapis (engl. *coordinate format*) u kojemu se pohranjuju trojke brojeva — vrijednost od nule različite komponente te indeksi retka i stupca u kojima se ona nalazi. Struktura podataka najčešće sadrži tri niza: niz vrijednostî od nule različitih komponenata (u bilo kojemu poretku), niz indeksâ redaka (u poretku koji odgovara poretku vrijednostî) i niz indeksâ stupaca (također u odgovarajućemu poretku). Primjerice, mogući je koordinatni zapis matrice

$$\begin{bmatrix} 10. & 0. & -2. & -3. & 0. \\ 0. & 40. & 0. & -5. & 0. \\ 2. & 0. & 60. & 0. & 0. \\ 3. & 0. & 0. & 80. & 0. \\ 0. & 0. & 7. & 0. & 90. \end{bmatrix}$$

prikazan na slici 1. Niz sadrži vrijednosti komponenata, niz pripadajuće indekse redaka, a niz pripadajuće indekse stupaca (u skladu s jezikom C++ indeksiranja počinju nulom). Duljine sva tri niza jednake su broju od nule različitih komponenata matrice.

A	10.	40.	2.	60.	-5.	3.	80.	7.	-3.	90.	-2.
I	0	1	2	2	1	3	3	4	0	4	0
J	0	1	0	2	3	0	3	2	3	4	2

Slika 1. Koordinatni zapis rijetko popunjene matrice

U mnogim algoritmima treba pristupiti svim od nule različitim komponentama određenoga retka ili stupca. Komponente nizova u koordinatnomu zapisu svrstavaju se stoga po redcima ili po stupcima (slika 2: svrstavanje po stupcima). Time se smanjuje broj pretraživanja — treba naći prvu komponentu retka ili stupca; ostale uzastopno slijede. Komponente mogu biti poredane i u svakomu retku ili stupcu (slika 3).

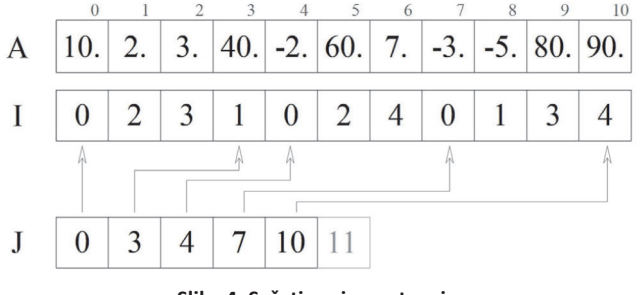
A	10.	3.	2.	40.	60.	-2.	7.	80.	-5.	-3.	90.
I	0	3	2	1	2	0	4	3	1	0	4
J	0	0	0	1	2	2	2	3	3	3	4

Slika 2. Koordinatni zapis sa svrstavanjem po stupcima

A	10.	2.	3.	40.	-2.	60.	7.	-3.	-5.	80.	90.
I	0	2	3	1	0	2	4	0	1	3	4
J	0	0	0	1	2	2	2	3	3	3	4

Slika 3. Koordinatni zapis sa svrstavanjem po stupcima i s poretkom u svakomu stupcu

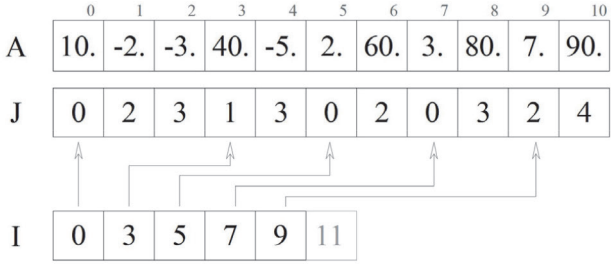
Još se učinkovitiji pristup svim od nule različitim komponentama retka ili stupca ostvaruje s pomoću sažetih zapisa. U sažetomu zapisu po stupcima (engl. *compressed sparse column format*, kratica *CSC*) nizovi A i I sadrže, kao i u koordinatnomu zapisu, vrijednosti komponenata matrice i indekse njihovih redaka, svrstane po stupcima, pa su njihove duljine jednake broju od nule različitih komponenata. Niz J pak sadrži indekse početnih komponenata pojedinih stupaca u nizovima A i I (slika 4).



Slika 4. Sažeti zapis po stupcima

Dok u koordinatnomu zapisu sa svrstavanjem po stupcima treba pretražiti niz kako bi se pronašla početna komponenta stupca *j*, u sažetomu zapisu po stupcima njezin je položaj *J*[*j*]. Broj od nule različitih komponenata stupca *j* izračunava se prema izrazu *J*[*j*+1] - *J*[*j*]. Kako bi se mogao izračunati broj komponenata u zadnjemu stupcu, nizu *J* dodaje se indeks zamišljenih komponenata neposredno iza zadnjih komponenata nizova A i I (sivo na slici 4.); pri indeksiranju koje počinje nulom taj je broj ujedno i broj od nule različitih komponenata matrice. Duljina je niza za jedan veća od broja stupaca matrice.

Sažeti zapis po redcima (engl. *compressed sparse row format*, *CSR*) prikazan je slici 5. Komponente pojedinih redaka ili stupaca u sažetim zapisima mogu (kao na slikama 4. i 5.), ali i ne moraju biti poredane.



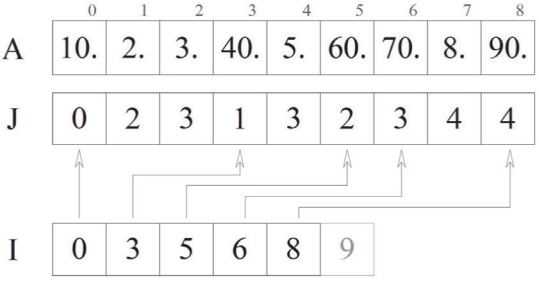
Slika 5. Sažeti zapis po redcima

Ako je matrica simetrična, dovoljno je pohraniti od nule različite komponente na dijagonali i ispod nje, u strogo donjemu trokutastom dijelu. Primjerice, pohraniti treba samo deblje otisnute (engl. *bold*) komponente matrice

$$\begin{bmatrix} \mathbf{10.} & 0. & 2. & 3. & 0. \\ 0. & \mathbf{40.} & 0. & 5. & 0. \\ \mathbf{2.} & 0. & \mathbf{60.} & 0. & 0. \\ \mathbf{3.} & \mathbf{5.} & 0. & \mathbf{70.} & 8. \\ 0. & 0. & 0. & \mathbf{8.} & \mathbf{90.} \end{bmatrix};$$

sažeti je zapis po stupcima prikazan na slici 6.

Pohraniti se, naravno, mogu i samo komponente gornjega trokutaskog dijela — na dijagonali i iznad nje. Pohranjivanjem dijela simetrične matrice štedi se memorijski prostor, a može se, ovisno o načinu i redoslijedu izračunavanja vrijednostî komponenta, skratiti i vrijeme izvođenja programa.



Slika 6. Sažeti zapis po stupcima donjega trokutastog dijela simetrične matrice

4. Množenje matrice i vektora

Po broju operacija (množenja i zbrajanja brojeva), a time i vremenski, u algoritmima IRM–CG i IRM najzahtjevniji su postupci množenje matrice i vektora i množenje matrica. Za smanjenje vremena rješavanja velikih sustava jednadžbi treba svakako pokušati skratiti trajanje izračunavanja umnoška triju matrica u retku 16. i izračunavanja umnoška matrice i vektora u retku 14. kôda za IRM te matrice i vektora u retku 17. kôda za IRM–CG, jer se ta izračunavanja ponavljaju u petljama `while`. želimo li pak algoritme IRM i IRM–CG upotrijebiti u iteracijskoj primjeni metode gustoća sila ili u nekom iteracijskom postupku rješavanja sustava nelinearnih jednadžbi, važnima postaju i množenja matrice i vektora izvan te petlje (redci 9. i 10. u IRM te redci 9., 10. i 13. u IRM–CG), jer se sustavi linearnih jednadžbi rješavaju u svakomu koraku vanjske petlje [6], [14].
U većini udžbenika linearne algebre (primjerice [9]) umnožak matrice **A** i vektora **r** definiran je izrazom

$$\alpha_i = \sum_{j=0}^{n-1} a_{i,j}\, r_j = \mathbf{a}_{i,\cdot} \cdot \mathbf{r} \quad \text{za } i \in [0, m-1]; \tag{2}$$

i-ta komponenta rezultata $\boldsymbol{\alpha}$ skalarni je umnožak *i*-toga retka matrice $\mathbf{a}_i$ i vektora **r**, pa se matricom "prolazi po redcima".

Vektori **r** i $\boldsymbol{\alpha}$ pohranjeni su kao gusto popunjeni. Ako je matrica **A** pohranjena kao gusto popunjena (kraće: kao puna) matrica, s vrijednostima svih komponenata u jednomu nizu, njezinim se komponentama u tom nizu pristupa neposredno, indeksiranjem, pa su položaji komponenata *i*-toga retka pri pohranjivanju po redcima $i \cdot n + j$, a $i + j \cdot m$ pri pohranjivanju po stupcima (indeksiranja počinju nulom). I ako je matrica **A** rijetko popunjena i pohranjena sažetim zapisom po redcima, pristup je komponentama traženoga retka neposredan: položaj je vrijednosti (u nizu A) i indeksa stupca (u nizu) prve od nule različite komponente u *i*-tomu retku *I*[*i*]. Položaji vrijednostî i indeksa stupaca ostalih od nule različitih komponenata toga retka u

nizovima A i I slijede uzastopno, do položaja *I*[*i*+1]-1. "Doslovni" je "prijevod" množenja u programski kôd:

```
for (int i = 0; i < m; ++i) {
  alpha[i] = 0;
  for (int j = I[i]; j < I[i + 1]; ++j)
    alpha[j] +=  A[j] * r[J[j]];
}
```

Primjenom biblioteke `Eigen` to se može napisati kao

```
for (int i = 0; i < A.outerSize(); ++i) {
  alpha[i] = 0;
  for (typename sparse_matrix_t::InnerIterator j (A, i); j; ++j)
    alpha[i] += j.value() * r[j.index()];
}
```

Ako je `sparse_matrix_t` rijetko popunjena matrica pohranjena sažetim zapisom po redcima, to jest, uz

```
typedef SparseMatrix<scalar_t, RowMajor> sparse_matrix_t;
```

Funkcija `.outerSize()` daje broj njezinih redaka, a ako je matrica pohranjena po stupcima,

```
typedef SparseMatrix<scalar_t, ColMajor> sparse_matrix_t;
```

funkcija `.outerSize()` daje broj stupaca .InnerIterator je tip podataka jedinka kojega (j) ima ulogu kazaljke koja u matrici pohranjenoj po redcima prolazi redom svim od nule različitim komponentama određenoga retka, pa funkcija `.value()` daje vrijednost komponente, a funkcija `.index()` indeks stupca u kojemu se komponenta nalazi. Ako je matrica pohranjena po stupcima, kazaljka prolazi svim od nule različitim komponentama određenoga stupca, pa funkcija `.index()` daje indeks retka u kojemu je komponenta. Naravno, ako je matrica pohranjena po stupcima, navedeni kôd nema smisla, barem ne kao množenje matrice i vektora. Osim ako... Vratit ćemo se na to.
Ako je matrica pohranjena sažetim zapisom po stupcima, pristup je komponentama nekoga retka znatno sporiji, jer komponente toga retka treba tražiti u pojedinim stupcima (štoviše, stupac će se pretražiti i ako se u njemu ne nalazi ni jedna komponenta retka). Stoga će i množenje matrice i vektora biti znatno sporije nego uz matricu pohranjenu po redcima. No, umnožak matrice **A** i vektora **r** može se definirati i izrazom

$$\boldsymbol{\alpha} = \sum_{j=0}^{n-1} r_j \, \mathbf{a}_{\cdot,j}; \tag{3}$$

vektor $\boldsymbol{\alpha}$, rezultat, linearna je kombinacija stupaca $\mathbf{a}_{.,j}$ matrice **A**, pri čemu su koeficijenti te linearne kombinacije komponente vektora **r** [1]. Za jednostavnu i učinkovitu programsku realizaciju tako definiranoga množenja pogodno je prethodno definirati programsku funkciju koja izračunava

$$\boldsymbol{\alpha} = c\,\mathbf{a}$$

gdje je c broj, $\boldsymbol{\alpha}$ puni, a **a** rijetko popunjeni vektor (rijetko popunjeni vektor pohranjuje se u koordinatnom zapisu s pomoću dva niza — niza vrijednostî od nule različitih komponenata i niza njihovih indeksa).
Ako je matrica **A** simetrična, onda je $\mathbf{A} = \mathbf{A}^{\mathrm{T}}$, pa za matricu pohranjenu po stupcima možemo pisati

```
alpha = A.transpose() * r;
```

što ima isti učinak kao prije navedeni programski kôd koji smo za po stupcima pohranjene matrice proglasili besmislenim. Funkcija `.transpose()` ne oblikuje novu matricu; njezin je rezultat samo drugi pogled na postojeću matricu u kojemu su uloge redaka i stupaca zamijenjene.
Ako je pohranjen samo trokutasti dio simetrične matrice, nisu sve komponente pojedinih redaka (u zapisu po redcima) ili stupaca (u zapisu po stupcima) smještene uzastopno. Primjerice, ako je simetrična matrica pohranjena u zapisu po redcima kao gornja trokutasta matrica, uzastopno su smještene dijagonalna komponenta i komponente retka desno od nje. Komponente retka lijevo od nje nalaze u se prethodnim redcima kao komponente stupca s istim indeksom. Nalaženje tih komponenata usporit će množenje matrice i vektora. Pretraživanje se može izbjeći primjenom svojstva distributivnosti i kombinacijom množenjâ prema izrazima (2) i (3):

```
1   alpha.setZero();
2   for (int i = 0; i < m.outerSize(); ++i) {
3     typename sparse_matrix_t::InnerIterator j (A, i);
4     alpha[i] += j.value() * r[j.index()];
5     ++j;
6     for (; j; ++j) {
7       alpha[i] += j.value() * r[j.index()];
8       alpha[j.index()] += j.value() * r[i];
9     }
10  }
```

Svojstvo distributivnosti daje

$$\mathbf{Ar} = (\mathbf{L} + \mathbf{D} + \mathbf{U})\mathbf{r} = \mathbf{Lr} + \mathbf{Dr} + \mathbf{Ur}$$

gdje su **D**, L i **U** dijagonalna, strogo donja trokutasta i strogo gornja trokutasta matrica. Budući da je matrica **A** simetrična, $\mathbf{L} = \mathbf{U}^T$, pa je

$$\mathbf{Ar} = \mathbf{U}^T\mathbf{r} + \mathbf{Dr} + \mathbf{Ur} \tag{4}$$

Komponente umnoška **Dr** izračunavaju se u retku 4. (Pretpostavljeno je da su sve dijagonalne komponente matrice različite od nule,[1] tako da je prva pohranjena komponenta svakoga retka dijagonalna komponenta). Komponente umnoška **Ur** izračunavaju se prema izrazu (2) u retku 7. u unutarnjoj petlji for (s kazaljkom j), dok se pribrojnici izraza (3) za umnožak $\mathbf{U}^T\mathbf{r}$ izračunavaju u retku 8. u istoj petlji, pri čemu pohranjeni redci matrice **U** imaju ulogu stupaca matrice $\mathbf{U}^T$. Kako se oba izračunavanja izvršavaju u istoj petlji, vrijednosti komponente matrice **A** (`j.value()`) i indeksu njezina stupca (`j.index()`) pristupa se samo jednom.
Lako je vidjeti da programski kôd ne treba mijenjati za simetričnu matricu koja je pohranjena kao donja trokutasta matrica u zapisu po stupcima, ali je njegova interpretacija drugačija:

$$\mathbf{Ar} = \mathbf{Lr} + \mathbf{Dr} + \mathbf{L}^T\mathbf{r} \tag{5}$$

Komponente umnoška **Dr** ponovno se izračunavaju u retku 4. Pribrojnici izraza (3) za umnožak **Lr** izračunavaju u retku 8., dok se komponente umnoška $\mathbf{L}^T\mathbf{r}$ prema izrazu (2) izračunavaju u retku 7., pri čemu pohranjeni stupci matrice **L** imaju ulogu redaka matrice $\mathbf{L}^T$.
(Pisanje kôda za pohranjivanje simetrične matrice kao donje trokutaste matrice u zapisu po redcima ili kao gornje trokutaste matrice u zapisu po stupcima prepuštamo radoznalim čitateljima. Napomenut ćemo ipak da je zadaća nešto teža, jer su dijagonalne komponente sada posljednje pohranjene komponente redaka ili stupaca.)
U biblioteci `Eigen` trokutasta matrica može preuzeti ulogu simetrične (u stvari, hermitske) matrice primjenom predloška funkcije `.selfadjointView<>()`. Ako smo, primjerice, simetričnu matricu pohranili kao donju trokutastu matricu, njezin je umnožak s vektorom

```
alpha = A.selfadjointView<Lower>() * r;
```

1 U matricama sustava jednadžbi ravnoteže u metodi gustoća sila i u inačicama i izvedenicama metode pomaka dijagonalne su komponente zbrojevi gustoća sila ili krutosti elemenata priključenih u čvorove uvjete ravnoteže kojih pojedine jednadžbe izražavaju.

5. Množenje triju matrica

Redak 16. kôda za IRM izračunavanje je umnoška triju matrica, $\mathbf{F}^T\mathbf{AF}$. Matrica **A** je rijetko popunjena, dok je matrica **F** puna.
Množenje matrica nije komutativna, ali jest asocijativna operacija, pa se umnožak može izračunati kao $(\mathbf{F}^T\mathbf{A})\mathbf{F}$ ili kao $\mathbf{F}^T(\mathbf{AF})$, pri čemu su $\mathbf{F}^T\mathbf{A}$ i **AF** pune matrice. Učinkovitiji je drugi način.
Naime, u izračunavanju umnoška dviju matrica, $\mathbf{F}^T\mathbf{A}$ ili **AF**, unutarnjom petljom, u kojoj se izračunava vrijednost neke njegove komponente, upravlja prva, lijeva matrica — prolazi se redom po svim pohranjenim komponentama njezina retka. Ako je lijeva matrica puna, a desna rijetko popunjena, za mnoge komponente retka lijeve matrice ne postoje odgovarajuće komponente u stupcu desne matrice, ali se to može otkriti samo tako da ih se potraži. Ako je pak lijeva matrica rijetko popunjena, u petlji se prolazi samo po od nule različitim komponentama retka rijetko popunjene matrice i po odgovarajućim komponentama stupca desne, pune matrice. Stoga ćemo, ako je matrica **A** pohranjena po redcima, redak 16. kôda za IRM zamijeniti redcima

```
dense_matrix_t AF = A * F;
dense_matrix_t FTAF = F.transpose() * AF;
```

a ako pohranjena po stupcima, redcima

```
dense_matrix_t AF = A.transpose() * F;
dense_matrix_t FTAF = F.transpose() * AF;
```

6. Uvjet prekida iteracije

U oba se postupka petlja `while` prekida kada broj koraka dosegne zadani dopušteni broj koraka (što je osiguranje za slučaj divergencije) ili kada euklidska norma reziduala postane manja od odabranoga praga. Za prag se obično uzima umnožak zadane točnosti i norme vektora desnih strana sustava jednadžbi, pa je uvjet prekida

$$\| \mathbf{r} \| < \tau \| \mathbf{b} \|,$$

odnosno, petlja se ponavlja dok je

$$\| \mathbf{r} \| \geq \tau \| \mathbf{b} \|.$$

Izračunavanje euklidske norme vektora uključuje izračunavanje drugoga korijena:

$$\| \mathbf{r} \| = \sqrt{\mathbf{r} \cdot \mathbf{r}}.$$

Korjenovanje se može izbjeći uzme li se za uvjet prekida

$$\mathbf{r} \cdot \mathbf{r} < \tau^2 \mathbf{b} \cdot \mathbf{b}.$$

7. Umjesto zaključka

U radu smo, bez želje za izvornošću i većim znanstvenim doprinosima, opisali primjere optimizacije programskoga kôda razvijenog u jeziku C++, domišljene u pokušajima ublažavanja "uskih grla" u algoritmima IRM–CG-a i IRM-a — množenja matrice i vektora i množenja triju matrica (množenje dviju matrica može se interpretirati kao uzastopno množenje matrice i vektorâ, stupaca desne matrice). Inačice nekih od prikazanih zamisli za množenje matrice i vektora ugrađene su u biblioteku Eigen. Iako primjena takve pouzdane (dobro testirane) i optimizirane biblioteke velike izražajnosti[2] olakšava razvoj učinkovitih programa, moguće je daljnje fino ugađanje kôda. Primjerice, iako je množenje matrica asocijativno, u umnošku triju matrica trajanje proračuna može bitno ovisiti o redoslijedu množenja.

Zahvala

Rad je financirala Hrvatska zaklada za znanost projektom IP–2014–09–2899.

Literatura

[1] S. Duff, A. M. Erisman, J. K. Reid: Direct Methods for Sparse Matrices, Clarendon Press, 1986

[2] J. Dvornik, D. Lazarević: Iterirani Ritzov postupak za rješavanje sustava linearnih algebarskih jednadžbi, Građevinar, 69 (2017) 7, str. 521–535, doi:https://doi.org/10.14256/JCE.2036.2017

[3] J. Dvornik, D. Lazarević, M. Uroš, M. šavor Novak: The Iterated Ritz Method: Basis, implementation and further development, Coupled Systems Mechanics, 7 (2018) 6, pp. 755–774, doi: https://doi.org/10.12989/csm.2018.7.6.755

[4] J. Dvornik, D. Lazarević, A. Jaguljnjak Lazarević, M. Demšić: Nonrecursive equivalent of the conjugate gradient method without the need to restart, Advances in Civil Engineering, Volume 2019, Article ID 7527590, 5 pages, doi: https://doi.org/10.1155/2019/7527590

[5] J. Dvornik, D. Lazarević, A. Jaguljnjak Lazarević: Iterirani Ritzov postupak: počela, trenutačno stanje i budući razvoj, Zbornik mini–simpozija "Novi, učinkoviti iteracijski postupak proračuna konstrukcija — poopćenje suvremenih postupaka" (ovaj zbornik), Građevinski fakultet Sveučilišta u Zagrebu, Zagreb, 2019., str. 11-37

2 Različiti postupci za različite vrste matrica skriveni su iza gotovo jedinstvenoga sučelja koje često oponaša matematički simbolizam.

[6] K. Fresl, P. Gidak, R. Vrančić: Poopćene minimalne mreže u oblikovanju konstrukcija od užadi, Građevinar, 65 (2013) 8, str. 707–720

[7] P. Gidak, K. Fresl: Programming the force density method, Proceedings of the IASS–APCS Symposium 2012, From spatial structures to space structures, K. Seung Deog (ed.), Seoul, Republic of Korea, 2012, p. 197 (abstract; article on CD)

[8] G. Guennebaud, B. Jacob et al.: Eigen v3, http://eigen.tuxfamily.org, 2010, pristupljeno 27. veljače 2019.

[9] S. Kurepa: Uvod u linearnu algebru, Školska knjiga, Zagreb, 1987.

[10] B. Maurin, R. Motro: The surface stress density method as a form–finding tool for tensile membranes, Engineering Structures, 20 (1998) 8, pp. 712–719

[11] Y. Saad: Iterative Methods for Sparse Linear Systems, Second Edition, Society for Industrial and Applied Mathematics, 2003

[12] H.–J. Schek: The force density method for form finding and computation of computation of general networks, Computer Methods in Applied Mechanics and Engineering, 3 (1974) 1, pp. 115–134

[13] B. Stroustrup: The C++ Programming Language, Fourth Edition, Addison–Wesley, Upper Saddle River, NJ, 2013

[14] E. Šamec, K. Fresl, M. Baniček: Povećanje učinkovitosti iteracijske primjene metode gustoće sila, Građevinar, 69 (2017) 12, str. 1075–1084, doi: https://doi.org/10.14256/JCE.2132.2017

[15] D. Vandevoorde, N. M. Josuttis, D. Gregor: C++ Templates. The Complete Guide, Second Edition, Addison–Wesley, Boston, 2018