Implementacije bazičnih algebarskih rutina u jezicima visoke razine

Kvastek, Ivan

SVEUČILIŠTE U ZAGREBU
PRIRODOSLOVNO-MATEMATIČKI FAKULTET
FIZIČKI ODSJEK

Ivan Kvastek

IMPLEMENTACIJE BAZIČNIH ALGEBARSKIH RUTINA U JEZICIMA VISOKE RAZINE

Diplomski rad

Zagreb, 2018.

SVEUČILIŠTE U ZAGREBU
PRIRODOSLOVNO-MATEMATIČKI FAKULTET
FIZIČKI ODSJEK

SMJER: PROFESOR FIZIKE I INFORMATIKE

IVAN KVASTEK

Diplomski rad

Implementacije bazičnih algebarskih rutina u jezicima visoke razine

Voditelj diplomskog rada: doc. dr. sc. Željko Skoko

Ocjena diplomskog rada: _____

Povjerenstvo: 1. _____

2. _____

3. _____

Datum polaganja: _____

Zagreb, 2018.

Ovom se prigodom želim zahvaliti svojim roditeljima na potpori i radosti koju su mi pružili u svakom smislu, na smijehu i suzama koje smo zajedno prošli, a najviše na toplini i razumijevanju.
Posebno mjesto zauzima Nives kojoj sam zahvalan što je bila racionalan glas i pokretač, kako kroz studij tako i izvan njega, moj vlastiti nuklearni reaktor.
Zahvaljujem se svojim prijateljima koji su samnom proživljavali sve uspone i padove. Bili smo si najveća podrška u svakom koraku.
Zahvaljujem se mentoru Željku Skoki na pokazanoj odlučnosti i velikoj dobroti. Hvala i dr. Ivici Martinjaku na vođenju kroz ovaj završni proces studiranja i znanju koje mi je nesebično predao. Hvala cijelom kadru ovog fakulteta na ovako divnom studiju.

Sažetak

U današnjem *tehnološkom dobu* znanost i industrija vape za računalnom snagom i bore se s barijerama, raznim tehnološkim ograničenjima i ostalim ograničavajućim faktorima. U toj se borbi javlja sve veća potreba za optimizacijom postojećeg *hardware-a* i *software-a*, a sve je manji fokus na čekanju "novog" tehnološkog otkrića i daljnjeg razvoja čvrstog dijela računala. U toj potrebi za optimizacijom konstantno su se pojavljivala nova rješenja i nova poboljšanja, a jedno od takvih novih rješenja je bio skup *bazičnih algebarskih rutina* - BLAS.

U ovom radu, nakon kratkog uvoda u algoritme i programske jezike, dano je nekoliko primjera algoritama i njihove implementacije u odabranim programskim jezicima više razine: *Fortranu* i *C-u*. Opisan je rad s matricama i determinantama, te je dan primjer računanja determinante i njegova implementacija koristeći odabrane programske jezike. Drugi, ujedno i glavni, dio rada obuhvaća osnove BLAS-a, njegove razine i raščlambu određene rutine treće razine. Prikazana je implementacija BLAS-a pomoću softverskog rješenja *LAPACK* koji implementira spomenutu rutinu. Na primjeru računanja determinante matrice proizvoljne dimenzije, koristeći algoritam napisan bez korištenja bazičnih algebarskih rutina i sa njima, će se pokazati ubrzanje izvršavanja našeg primjera. Na kraju rada je navedeno nekoliko primjera korištenja programskih jezika viših razina i navedena je mogućnost korištenja BLAS-a u sklopu nastave.

Implementation of basic algebraic routines in high-level programming languages

Abstract

In today's *technological age* science and industry are struggling with computer strength, barriers, various technological limitations and other limiting factors. In this struggle there is an increasing need for optimization of existing *hardware* and *software*. There is less focus on waiting for a "new" technological breakthrough and further development of a solid part of the computer. In this need for optimization, new solutions and new improvements were constantly emerging, and one of these new solutions was a set of *basic algebraic routines* - BLAS.

This paper contains short introduction to algorithms and programming languages and several examples of algorithms, their implementation in selected higher-level programming languages: *Fortran* and *C*. Work with matrices and determinants is described, and it is given an example of calculating determinants and their implementation using selected programming languages. The second, and also the main, part of the paper encompasses the basics of BLAS, its level and the breakdown of certain third-level routines. BLAS is shown using the *LAPACK* software solution that implements the mentioned routine. In the example of calculating the matrix determinant of an arbitrary dimension there will be shown the acceleration of execution of our example by using an algorithm written eith with or without basic algebraic routines. At the end of this paper are few examples of using higher level programming languages and BLAS that can be used as part of curriculum.

Sadržaj

1 Uvod

Neformalno, *algoritam* je bilo koja dobro definirana računska procedura koja za *ulaznu jedinicu* uzima neku vrijednost (skup vrijednosti) i daje neku vrijednost (skup vrijednosti) kao *izlaznu jedinicu*. Stoga je algoritam slijed računskih koraka koji transformira ulazne podatke u izlazne podatke [1].

Algoritam možemo također promatrati kao alat ili metodu kojom ćemo riješiti određeni problem s kojim smo se susreli. Algoritam opisuje određene korake kojim rješavamo navedeni problem i neovisan je o sustavu koji ga pokreće, kao što je računalna arhitektura i/ili operativni sustav. U mnogim slučajevima, a posebno u znanosti, računalni programi koriste algebarske strukture kao što su primjerice matrice i vektori. Da bi mogli napisati efikasan i brz algoritam, ili bolje iskoristiti postojeći, potrebno je proučavati i samu strukturu podataka.

Izrada kompleksnih računalnih programa zahtjeva iznimnu pažnju kod razumijevanja i definiranja problema kojeg program treba riješiti. To možemo postići rastavljanjem problema na manje podzadatke koji se lako mogu implementirati, prilagoditi i izmijeniti.

Kod računalnih programa iznimno je bitno da je algoritam, koji je implementiran za rješavanje problema, brz i efikasan (da algoritam završi s radom u praktičnom vremenu). Sam algoritam može biti određene, konačne, kompleksnosti i vremenske složenosti, te kako bi se dodatno ubrzao rad računalnog programa, potrebne su druge metode ubrzanja. Najbolji primjer takve metode je softversko rješenje koje implementira *BLAS* (*bazične algebarske rutine*). Jedno od softverskih rješenja je LAPACK koje je korišteno u ovom radu da bi se pokazala praktična primjena biblioteke i kako bi se pokazalo značajno ubrzanje računalnog programa koji vrši određene matematičke operacije.

2 Algoritmi

Kada su se u dvanaestom stoljeću preveli radovi arapskih matematičara na latinski, dekadski brojevni sustav se raširio diljem Europe zamijenivši rimske brojeve. Takav sustav je bio prikladan za izvršavanje aritmetičkih operacija i doveo je do novih metoda računanja. Iako su znakovi za brojeve preuzeti od Indijaca, brojevi su postali poznati kao arapski brojevi. Riječ *algoritam* potječe od iskrivljenog imena $AL - KHW\bar{A}RIZM\bar{I}$, autora najstarijeg poznatog djela o algebri s preve polovice devetog stoljeća.

Algoritam je konačan niz jednoznačno definiranih uputa za rješavanje nekog problema. Opće prihvaćena svojstva algoritama su:

- Konačnost. (Moraju završiti nakon konačnog broja koraka.)
- Dobra definiranost. (Svaki korak sadrži jednoznačne i jasne upute)
- Ulazni podatci (*eng. input*). (Specificirani su skupovi objekata nad kojima se algoritam može izvršiti.)
- Izlazni podatci (*eng. output*). (Željeni rezultat.)
- Funkcionalnost. (Moraju biti izvedivi u konačnom, prihvatljivom, vremenu.)

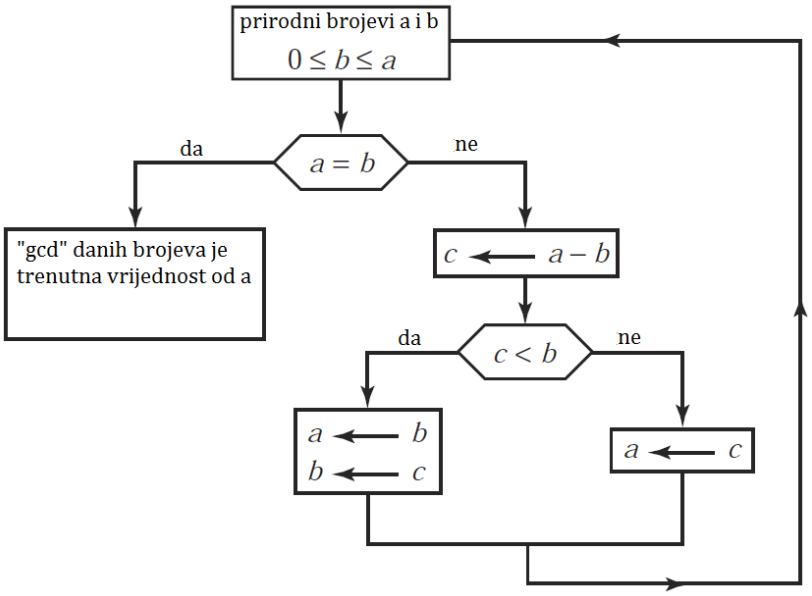
2.1 Euklidov algoritam

Jedan od osnovnih algoritama, koji se uči u školi, je Euklidov algoritam za nalaženje najvećeg zajedničkog djelitelja dvaju prirodnih brojeva (gcd[1]). Algoritam se izvršava u nizu koraka tako da je izlazni podatak svakog koraka, ulazni podatak idućeg. Izvedbu algoritma ćemo opisati na idući način:
Neka su zadana dva prirodna broja a i b.

- Vrijedi li $a = b$?
  - Ako da, najveći zajednički djelitelj danih brojeva je vrijednost od a.
  - Ako ne, $c = a - b$.
- Vrijedi li $c < b$?
  - Ako da, zamijeni vrijednost od a s vrijednošću od b i zamijeni vrijednost od b s vrijednošću od c. Idi na početak.
  - Ako ne, zamijeni vrijednost od a sa vrijednošću od c. Idi na početak.

Koraci algoritma su prikazani na dijagramu toka (slika 2.1).

[1] gcd - eng. greatest common divisor (najveći zajednički djelitelj)



Slika 2.1: Dijagram toka Euklidovog algoritma za nalaženje najvećeg zajedničkog djelitelja.

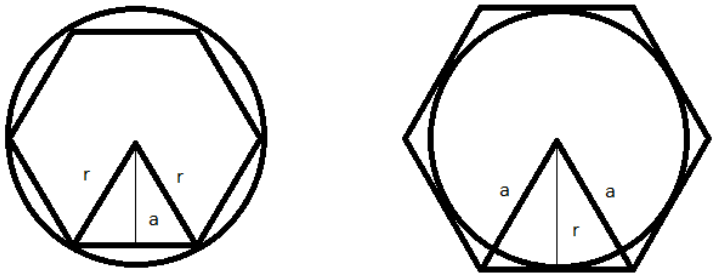
2.2 *Izračun broja* π

Interes za brojem π je oduvijek zaokupljao velike umove svijeta, primjerice se Arhimedova metoda izračuna broja π smatra među prvim algoritmima. Arhimed je u trećem stoljeću prije nove ere osmislio način kako da izračuna aproksimaciju broja π. Ako se upiše mnogokut unutar jedinične kružnice sa zajedničkim središtem, odnosno njegovi vrhovi leže na kružnici, te se opiše kružnica sa mnogokutom čije su stranice tangente na kružnicu, i ako se izračunaju površine tih mnogokuta, dobit će se donja i gornja granica vrijednosti broja π. Razlog dobivanja navedenih vrijednosti je veći opseg kružnice od dužine stranica bilo kojeg unutarnjeg mnogokuta odnosno manji od bilo kojeg vanjskog mnogokuta. Arhimed je započeo sa šesterokutima, a kasnije udvostučavanjem broja stranica, došao do devedeset-šesterokuta čime je dobivao sve preciznije vrijednosti i došao do procjene

$$3 + \frac{10}{71} \leq \pi \leq 3 + \frac{1}{7}.$$

Očito je da ova metoda uključuje iteracije, ali koliko god se broj stranica mnogokuta poveća dobit ćemo samo aproksimaciju broja π, stoga imamo proces koji nije konačan. Ono što imamo je algoritam koji može izračunati π proizvoljne preciznosti. Ukoliko želimo preciznost do desete decimale, algoritam će nakon određenog broja koraka dati željeni rezultat.

Između Euklidovog algoritma za nalaženje najvećeg zajedničkog djelitelja i algoritma za aproksimaciju broja π postoji očita razlika. Algoritmi kao što su Euklidov algoritam se često nazivaju *diskretnim algoritmima*, i uspoređuju se sa *numeričkim algoritmima* koji se koriste za dobivanje brojeva koji nisu cijeli brojevi [7].



Slika 2.2: Skica Arhimedove metode izračuna broja π pomoću izračuna površina upisanogi opisanog pravilnog šesterokuta.

Često se algoritmi mogu klasificirati prema više kriterija, prema zadaćama koje obavljaju ili prema određenom području znanosti koje ih istražuje, odnosno zbog kojeg je samo područje nastalo [2]. Spomenimo neke od tipova algoritama, prema tim klasifikacijama:

- Algoritmi *sortiranja*. Neki od algoritama za sortiranje su sortiranje umetanjem, bubble sort, selection sort, quick sort, mergesort i dr.

- Algoritmi *pretraživanja*. Neki od algoritama pretraživanja su linearno ili slijedno pretraživanje, binarno pretraživanje, binarna stabla i dr.

- *Algoritmi za obradu nizova znakova* (string). Metode koje su doprinijele razvoju tehnika kompresije podataka i kriptografiji.

- *Geometrijski algoritmi*. To je skup metoda za rješavanja problema koji uključuju linije i točke. Na primjer, algoritmi za pronalazak sjecišta nekih geometrijskih objekata, za rješavanje problema najbliže točke, multidimenzionalno pretraživanje i drugi. Neke od ovih metoda su lijepo upotpunili elementarnije metode sortiranja i pretraživanja.

- *Matematički algoritmi*. Uključuju fundamentalne metode iz *aritmetike* i *numeričke analize*. Neki od problema koji spadaju pod ove algoritme su aritmetika polinoma i matrica, generatori slučajnih brojeva, integracija i drugi.

Kao jedan od bitnih aspekata kod razvoja algoritama, njihovog razumijevanja i primjene, je analiziranje algoritama koje se svelo na predviđanje koliko je resursa potrebno za algoritam. U određenim slučajevima kao najbitnije stavke koje želimo mjeriti ističu se resursi, kao što je memorija, komunikacijska propusnost[2] ili broj

[2]Komunijske propusnosti između centralne procesorske jedinice i memorije ili nekog drugog podsistema arhitekture koji nam je bitan ili predstavlja usko grlo čime se bitno utječe na performanse algoritma.

logičkih vrata[3], no najčešće je to vrijeme računanja. Analiziranjem nekoliko algoritama kandidata može se identificirati najefikasniji algoritam za određeni problem. Takve analize mogu dati nekoliko mogućih kandidata koji ispunjavaju željene ciljeve i rezultate, ali se u pravilu algoritmi slabijih i manje zadovoljavajućih rezultata u postupku odbacuju.

Prije nego što možemo analizirati algoritam, procijeniti njegovo vrijeme izvođenja, te ga usporediti s drugim algoritmima, potreban nam je odgovarajući model implementacije algoritma i resursa. Općenito se uzima model računala koji je jednoprocesorski *RAM model* (*random-access machine*)[4] [1].

2.3 Vremenska složenost algoritama i O-notacija

Istaknuli smo kako je jedan od najvećih izazova zapravo određivanje potrebnih resursa za izvršenje željenih zadataka kao što su memorija, propusnost i vrijeme računanja. Naročito važan resurs je vrijeme, odnosno (uzimajući u obzir hardver) broj koraka potrebnih algoritmu prilikom stvarnog izvršenja zadataka. Pri tome, posebno je bitan odnos vremena potrebnog da se algoritam izvrši i povećanja ulaza za taj zadatak. Na primjer, pitanje je koliko vremena je potrebno za faktoriziranje cijelog broja s n znamenki naspram cijelog broja s $2n$ znamenki?

Drugi je aspekt također već spomenuta memorija računala. Uvijek se postavlja pitanje koliko će biti potrebno memorije za izvršenje algoritama te koliko je moguća redukcija potreba.

Svrha analize složenosti programa je određivanje vremena izračuna i pronalaženje najučinkovitijih algoritama. Pretpostavimo li da sve operacije troše jednako mnogo procesorskog vremena, da je računalo sekvencijalno jednoprocesorsko i da je fiksno vrijeme dohvata sadržaja memorijske lokacije, neformalno možemo definirati *vremensku složenost algoritama*, kao maksimalan broj operacija potrebnih za izvršavanje nekog algoritma.

Tri su slučaja koja razmatramo pri analizi složenosti određenog algoritma:

- ponašanje algoritma u *najboljem* slučaju
- ponašanje algoritma u *najgorem* slučaju
- *prosječno* ponašanje.

Prosječno ponašanje, odnosno prosječno vrijeme izvršavanja, definiramo kao računanje vremenske složenosti, to jest kao očekivani broj potrebnih operacija za izvršenje algoritma. Takav pristup u praksi je realističniji od ostala dva, ali je teži za procjenu.

[3]Logičke operacije u algoritmu (*and, or, xor...*). Prilikom analiziranja logičkih vrata algoritma mogu se otkriti i riješiti eventualni problemi na primjer smanjenjem broja logičkih vrata ili pojednostavljenjem cjelokupne sheme.

[4]RAM model računala je takav model računala koje izvršava instrukciju jednu po jednu u nizu, bez drugih procesa koji mogu smetati izvršavanju algoritma.

Razlog tome je što se za njegovu procjenu zahtjeva poznavanje raspodjele vjerojatnosti ulaznih podataka. Određivanje prosječnog ponašanja algoritma zahtjeva da ulazni podaci testova često sadrže specijalne i granične slučajeve. Na taj način se testira ponašanje algoritma u svim mogućim slučajevima [23].

2.3.1 O-notacija

U računarstvu jedna od najčešće korištenih notacija za klasifikaciju algoritama, s obzirom na stupanj vremenske složenosti u ovisnosti o rastu ulaza, je *O-notacija*[5]. O-notacija karakterizira algoritme sukladno njihovom stupnju rasta vremenske ili prostorne složenosti ovisno o ulazu, odnosno procjenjuje se trajanje izvršavanja algoritma kao red veličine određen temeljem broja podataka n pomnoženo s nekom konstantom c.

Tipično se O-notacija određuje za neku funkciju *f* korištenjem dvaju pravila:

- Ako se $f(n)$ sastoji od sume nekoliko članova te ako je neki od tih članova najveći, onda se taj član zadrži, a ostali se odbacuju.
- Ako je $f(n)$ produkt nekoliko faktora, sve konstante se odbacuju (članovi produkta koji ne ovise o n).

Primjerice, ako ustanovimo da je za izvršavanje nekog algoritma uz *n* ulaznih podatak podataka potrebno $2n^3 + 5n + 77$, koraka kažemo da je složenost $\mathcal{O}(n) = n^3$. Zanima nas kako vrijeme izvršavanja algoritma raste ako *n* raste. Ako je $n = 1$ tada je član $2n^3 = 2$, $5n = 5$ i u tom slučaju konstanta $c = 77$ doprinosi najviše rastu, no ako uzmemo da je $n = 10$, tada je član $2n^3 = 2000$, $5n = 50$ i očito je da najveći doprinos daje prvi član pa se ostali čanovi mogu, bez velikih pogrešaka, odbaciti te nam prvi član određuje složenost. Također, O-notacija se može prikazati primjerima algoritama na računalu koji su nezavisni od njega, programskog jezika i prevoditelja[6]. Slijede ti primjeri:

i)
```
x += y;
```
$\mathcal{O}(n) = 1$

ii)
```
for(i = 1; i<=n; i++) {
    x += y;
}
```
$\mathcal{O}(n) = n$

[5] eng. "Big O notation"

[6] Prevoditelj odnosno jezični prevoditelj ili programski prevoditelj (*eng. compiler*) jest računalni program koji prevodi program napisan u izvornom programskom jeziku u program u željenom, najčešće strojnom, jeziku.

iii)
```
for(i = 1; i<=n; i++) {
    for(j = 1; j<=n; j++) {
        x += y;
    }
}
```

$\mathcal{O}(n) = n^2$

Općenito, O-notacije možemo navesti po vrstama složenosti gdje je c pozitivna konstanta i n raste neograničeno:

- $\mathcal{O}(n) = 1$ - *konstantna složenost*
- $\mathcal{O}(n) = log\ n$ - *logaritamska složenost*
- $\mathcal{O}(n) = n$ - *linearna složenost*
- $\mathcal{O}(n) = n\ log\ n$ - *linearno-logaritamska složenost*
- $\mathcal{O}(n) = n^c$ - *polinomialna složenost*
- $\mathcal{O}(n) = c^n$ - *eksponencijalna složenost*
- $\mathcal{O}(n) = n!$ - *faktorijelna složenost*

Vrste složenosti su poredane od najmanje složenosti prema najvišoj, odnosno:

$$\mathcal{O}(n) = 1 < \mathcal{O}(n) = log\ n < \mathcal{O}(n) = n < \mathcal{O}(n) = n\ log\ n$$

$$< \mathcal{O}(n) = n^c < \mathcal{O}(n) = c^n < \mathcal{O}(n) = n!$$

2.4 Računalne implementacije algoritama

Za implementaciju algoritama u ovome radu koristimo dva programska jezika više razine: Fortran 95 i C.

Računala mogu izvršiti određene jednostavne instrukcije kao što su zbrajanje i pomicanje znamenaka, čijim kombinacijama se može realizirati množenje, dijeljenje i oduzimanje. Navedene jednostavne instrukcije bile su temelj prvih računala. Razvojem računala usavršavao se i jezik za komuniciranje s njima, pa razlikujemo četiri generacije:

1. Prva generacija – strojni jezici
2. Druga generacija – simbolički (asemblerski) jezici
3. Treća generacija – jezici za programiranje visoke razine
4. Četvrta generacija – jezici četvrte generacije (objektno orijentirani programski jezici i jezici posebne namjene) [3].

Strojni jezik je najniža razina prevedenog programa. Sačinjen je od niza nula i jedinica kao reprezentacija određene instrukcije kojeg izravno obrađuje računalna središnja jedinica (CPU). Kako je taj jezik specifičan za središnju procesorsku jedinicu, i u ovisnosti je o hardveru, programiranje na toj razini karakterizira sklonost pogreškama i težina programiranja. Program koji programer napiše za jedno računalo određenih hardverskih karakteristika, u mnogim slučajevima neće biti kompatibilno s računalom različitih karakteristika.

Prvi korak prema takozvanim „jezicima više razine“ bio je uvođenje *simboličkog* (*asemblerskog* ili mnemoničkog) *jezika*. U tom jeziku programer koristi mnemonička imena i znakove za operacije i operande.

Primjer na slici 2.3[7] pokazuje dio izvršnog koda (strojni jezik) i njemu ekvivalentan izvorni, asemblerski kod. Radi se o asembleru za Intelov mikroprocesor 8086 [4].

Strojni jezik	Ekvivalentan asemblerski kôd
00011110	PUSH DS
00101011	SUB AX,AX
11000000	
10111000	PUSH AX
10111000	MOV AX,MYDATA
00000001	
10001110	
11011000	MOV DS,AX

Slika 2.3: Primjer koda u asemblerskom jeziku i ekvivalent u strojnom jeziku.

Uvođenjem asemblerskih jezika znatno se olakšao posao programerima, no i dalje je programer trebao poznavati arhitekturu računala za koji piše program jer u asemblerskom jeziku programer ima potpunu kontrolu nad komponentama računala (registri, zastavice, razine memorije,...). Kompleksne operacije se svode na niz operacija koje upotrebljavaju samo primitivne tipove podataka i treba se voditi računa o tome kako i gdje su u memoriji postavljeni podaci, te kako im se pristupa.

Kako bi se takve probleme riješilo i pojednostavnilo pisanje programa, razvijeni su jezici visoke razine. U takvim jezicima se omogućuje programeru da piše algoritme u prirodnoj notaciji u kojoj se ne treba obraćati pozornost na veliki broj detalja vezanih uz specifično računalo odnosno njegovu arhitekturu. Primjerice, ugodnije je pisati $A = B + C$ nego niz asemblerskih instrukcija. Takve anotacije objedinjuju veći broj instrukcija te se tako programeru omogućilo da se koncentrira na sami problem algoritma, a ne na arhitekturu računala i iskoristivost njegove memorije (za što se pobrinuo kompajler) te se time unaprjeđuju algoritmi i omogućava uvođenje kompleksnijih struktura podataka.

[7] Slika 2.3 je preuzeta iz [4] na stranici 9.

Takvi jezici podvrgnuti su standardima s propisanom sintaksom i semantikom i time je u velikoj mjeri postignuta neovisnost o karakteristikama računala i operativnog sustava na kojem programiramo. Iako se time rješava problem ovisnosti o računalu ponekad je potrebno poznavati računalo i njegove hardverske karakteristike kako bi znali optimizirati naše programe i algoritme. Više o tome se može naći u četvrtom poglavlju ovoga rada.

2.4.1 Programski jezik Fortran

Fortran, čije ime dolazi od *Formula translation system* što znači sustav za prevođenje formula), je jezik za programiranje prvenstveno namijenjen za rješavanje numeričkih problema.

Prva verzija Fortrana (poznata kao Fortran I) je bila implementirana na računalima IBM 704, 1956. godine. Dvije godine kasnije pojavio se Fortran II kao proširenje Fortrana I. Između 1958. i 1963. godine Fortran je bio implementiran na nekoliko vrsta računala. Naredna verzija Fortrana (Fortran III) je i dalje sadržavala osobine koje su ovisile o specifičnoj arhitekturi računala što je ujedno i razlog njegovog ne objavljivanja.

Tokom 1962. godine bio je razvijen Fortran IV za računala IBM 7090/7094. Iste je godine američko udruženje za standardizaciju osnovalo "*Komitet za definiranje standardne verzije Fortrana*" čiji je cilj bio omogućiti prenošenje programa s jednog računala na drugo. Rezultat rada Komiteta bile su dvije verzije Fortrana, objavljene 1966. godine. Jedna je poznata kao „Osnovni Fortran" i slična je Fortranu II, dok je druga verzija, „Fortran" slična Fortranu IV.

Konačno je 1977. godine definirana nova, značajno unaprijeđena, verzija Fortrana poznata kao "Fortran 77" koju je bilo moguće implementirati na mnogim računalima. Pored osnovnih naredbi naslijeđenih od prethodnih verzija, treba istaknuti najaznačajnije karakteristike: osobine prilagođene za takozvano strukturno programiranje, operacije i funkcije nad nizovima znakova te mogućnosti rada s datotekama.

Nakon Fortrana 77 (kraće F77) objavljeno je nekoliko, podosta proširenih, verzija:

- Verzije F90 i F95 donose velike promjene u koncepciji jezika: slobodna forma, modularnost, rekurzivnost, „priprema" za objektno orijentirano programiranje, operacije nad jednodimenzionalnim poljima, dinamičku alokaciju memorije, pokazivače itd.
- F2003 donosi pravo objektno orijentirano programiranje, standardizaciju aritmetike (IEEE standard), napredno upravljanje podacima, interoperabilnost sa C-om, CLI (command line interface) parametre.
- F2008 donosi paralelizam, „memory management" i dr.

Za potrebe ovog rada koristimo *gfortran – GNU Fortran compiler* koji je dio *GCC-a* (*GNU Compiler Collection*) na linux operativnom sustavu. Gfortran je ime za GNU

Fortran projekt, kojim se razvija besplatni Fortran 95/2003/2008 kompajler za GCC, *GNU Compiler kolekciju* [6].

2.4.2 Programski jezik C

Programski jezik C je programski jezik opće namjene. Usko je vezan za UNIX operativni sustav pošto su sam operativni sustav i većina programa koje pokreće napisani u C-u. Međutim, jezik nije vezan uz nijedan operativni sustav niti računalo. Iako je prozvan ”programskim jezikom operativnih sustava” (jer je koristan za pisanje kompajlera i operativnih sustava) jednako je korišten u svrhu pisanja značajnog broja programa u raznim domenama [5].

Mnoge važne ideje programskog jezika C proizlaze iz jezika BCPL[8] indirektno, preko programskog jezika B[9]. C jezik je razvio *Dennis Ritchie* 1970-tih godina u *Bell Telephone Laboratories, Inc*. Programski jezik C pruža temeljne konstrukcije za kontrolu toka za dobro strukturirane programe: grupiranje naredbi, donošenje odluka (*if-else*), odabir jednog od niza slučajeva (*switch*), petlje s ispitivanjem o prestanku na početku (*while, for*) ili na kraju (*do*) i rani izlaz iz petlje (*break*).

Neke od važnijih karakteristika C-a su da funkcije mogu vratiti vrijednosti osnovnih tipova, struktura, skupova ili pokazivača. Također, svaka se funkcija može pozvati rekurzivno. Lokalne varijable su tipično „automatske“, ili stvorene iznova svakim pozivanjem. Definicije funkcija se ne mogu gnijezditi, ali se varijable mogu deklarirati u blok-strukturi. Funkcije u C programu mogu postojati u odvojenim izvornim datotekama koje su kompajlirane odvojeno. Vidljivost varijabli kroz program je definirana tako da varijable mogu biti vidljive unutar funkcije, izvan nje, i to na dva načina: da su vidljive samo unutar izvorne datoteke ili mogu biti vidljive cijelom programu. Također, pretprocesorski korak kod programskog jezika C obavlja makro supstituciju na programski tekst, uključivanje drugih izvornih datoteka i uvjetno kompajliranje.

S druge strane C nema operacije koje barataju direktno s nizovima, listama ili stringovima. Ne postoje operacije koje manipuliraju čitavim stringom ili nizom iako se strukture mogu kopirati u cijelosti. Također, ne postoji način alociranja spremišta u memoriji kao što su stog ili heap, niti posjeduje ”*sakupljače smeća*[10]”. Na kraju, *C* nema mogućnosti za manipuliranjem ulaza/izlaza (ne postoje READ/WRITE funkcije). Navedeni mehanizmi više razine se moraju pozvati sa eksplicitno definiranim funkcijama. Većina implementacija C-a imaju predefinirane standardne funkcije koje se posebno pozivaju prilikom pisanja programa. Također, C pruža jednodretvenu kontrolu toka programa sa petljama, grupiranjima, potprogramima, ali ne posjeduje

[8]Programski jezik BCPL(BasicCombinedProgrammingLanguage)je razvio *Martin Richards* profesor na *Sveučilištu Cambridge* (do umirovljenja 2007. godine)

[9]Programski jezik B je napisao Ken Thompson 1970. godine za prvi UNIX sustav na računalu DEC PDP-7 [5]

[10]skupljanje smeća (eng. garbage collection) - oblik automatskog upravljanja memorijom. Sakljupač smeća pokušava vratiti zauzetu memoriju od objekata koje računalni program više ne rabi.

mogućnost izvršavanja paralelnih operacija i sinkronizaciju procesa [5].

3 Numerička analiza

Numerička analiza je proučavanje algoritama za rješavanje problema kontinuirane matematike koji uključuju realne ili kompleksne varijable.

Računla imaju veliku ulogu u ovom području. Postoji česta miskoncepcija da znanstvenici generiraju i implementiraju formule, zatim umetnu brojeve u te formule i puste računalo da "izbaci" željeni rezultat. Stvarnost je daleko drugačija. U velikom broju slučajeva posao se ne bi mogao izvršiti s matematičkim formulama jer se većina matematičkih problema ne može riješiti u konačnom slijedu osnovnih operacija. Ono što se događa jest da algoritmi brzo konvergiraju rješenju koje je približno točno na nekoliko decimala, ili na nekoliko stotina decimala. Za znanstvenu ili inženjersku/industijsku primjenu to se može smatrati točnim rezultatom [7]. Koliko je odnos točnog i približnog (aproksimativnog) rješenja kompleksan možemo prikazati s elementarnim primjerom. Recimo da imamo jedan polinom reda 4

$$p(z) = c_0 + c_1 z + c_2 z^2 + c_3 z^3 + c_4 z^4$$

i drugi reda 5,

$$q(z) = d_0 + d_1 z + d_2 z^2 + d_3 z^3 + d_4 z^4 + d_5 z^5.$$

Poznato je da postoji eksplicitna formula za traženje nultočki polinoma p (otkrio ju je Ferrari oko 1540. godine) te je iznimno komplicirana i nepogodna za pamćenje, ali ne postoji takva formula za q (pokazali Ruffini i Abel više od 250 godina kasnije). U određenom filozofskom smislu potraga za rješenjima se znatno razlikuje dok se u praksi nimalo ne razlikuju. Ako znanstvenik želi znati rješenja tih polinoma, obratiti će se računalu i njegovim alatima, te će dobiti rješenje preciznosti do 16 decimala u manje od milisekunde. Je li računalo koristilo eksplicitnu formulu za računanje? Za q sigurno nije, za p možda je, a možda i nije. "Većinu vremena, korisnik niti zna niti ga zanima, i vjerojatno niti jedan na stotinu matematičara ne bi mogao ispisati formule za korijene polinoma p iz glave." [7]

Tri primjera koja se u principu mogu riješiti s konačnim brojem elementarnih operacija, kao što je bilo moguće za p:

i Linearne jednadžbe: riješiti sustav n linearnih jednadžbi sa n nepoznanica

ii Linearno programiranje: minimizirati linearnu funkciju n varijabli koja podliježe pod m linearnih ograničenja

iii Problem trgovačkog putnika: pronaći najkraći put između n gradova.

Pet primjera koji se, kao ni problem q polinoma, ne može riješiti na taj način:

iv Pronaći svojstvenu vrijednost $n \times n$ matrice

v Minimizirati funkciju s nekoliko varijabli

vi Izračunati integral

vii Riješiti običan diferencijalni račun

viii Riješiti parcijalni diferencijalni račun.

Postavlja se pitanje jesu li primjeri *i – iii* lakši od primjera *iv – viii*? Primjerice problem trgovačkog putnika je jako težak ako je, recimo, *n* reda veličine 100 ili 1000. Problemi *vi* i *vii* su u određenoj mjeri lagani, ako je integral jednodimenzionalan. Problemi od *i – iv* su lagani ako je n malen, ali ako je *n* velik, recimo milijun, onda ti problemi postaju jednako teški i toliko im vremena treba da se riješe da se može reći da se niti oni ne mogu riješiti u konačnom broju koraka pa se onda pribjegava aproksimativnom (ali brzom) rješavanju problema. Time se ukazuje na potrebu za numeričkim izračunima.

3.1 Rad s matricama

Rad s matricama predstavlja jedan od najbitnijih segmenata u računarstvu, iz čega proizlazi veliki interes za efikasnim algoritmima koji se koriste za rad s matricama.

Matrica je matematički objekt koji se sastoji od (realnih ili kompleksnih) brojeva koji su raspoređeni u retke i stupce. Zapisuje se u obliku pravokutne sheme, a brojeve od koji se sastoji zovemo *elementima matrice*. Matrica $A = [a_{ij}]$ sa *m* redaka, *n* stupaca i s elementima a_{ij} zapisuje se kao

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & a_{m3} & \cdots & a_{mn} \end{pmatrix}.$$

Matrice su pravokutni nizovi brojeva. Na primjer:

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}.$$

A je 2×3 matrica $A = (a_{ij})$, gdje je a_{ij} za $i = 1, 2$ i $j = 1, 2, 3$ element matrice u retku *i* i stupcu *j*. Odnosno niz brojeva $a_{i1}, a_{i2}, a_{i3}, \ldots, a_{in}$ zovemo *i*-ti redak, a niz brojeva $a_{1i}, a_{2i}, a_{3i}, \ldots, a_{mi}$ poredanih jedan ispod drugog, *j*-ti stupac matrice A. [10]

Transponirana matrica A je A^T, dobivena izmjenom redaka i stupaca od A. Za našu matricu A to je:

$$A^T = \begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix}.$$

Vektor je jednodimenzionalni niz brojeva:

$$x = \begin{pmatrix} 2 \\ 3 \\ 5 \end{pmatrix}.$$

Kao standardni vektor bi bio *vektor stupac*, dok bi se *vektor redak* dobio transponiranjem. *Jedinični vektor* je vektor čiji je *i*-ti član jednak 1 a ostali članovi su 0.

Nul matrica je matrica čiji su svi elementi jednaki 0.

Kvadratna matrica je matrica dimenzija $n \times n$ te se takav tip matrica često pojavljuje. Nekoliko specijalnih slučajeva kvadratne matrice:

- *Dijagonalna matrica* - njeni članovi $a_{ij} = 0$ kada $i \neq j$:

$$diag(a_{11}, a_{22}, \ldots, a_{nn}) = \begin{pmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn}) \end{pmatrix}.$$

- *Jedinična matrica* - $n \times n$ jedinična matrica I je dijagonalna matrica čiji su elementi na dijagonali jednaki 1.

- *Trodijagonalna matrica* T - je matrica čiji elementi $t_{ij} = 0$ ako $|i - j| > 1$. Odnosno vrijednosti se pojavljuju samo na glavnoj dijagonali matrice, neposredno iznad i neposredno ispod glavne dijagonale.

- *Gornje trokutasta matrica* U - je matrica gdje je $u_{ij} = 0$ ako $i > j$. Sve vrijednosti ispod dijagonale su jednake 0.

- *Donje trokutasta matrica* L - je matrica gdje je $l_{ij} = 0$ ako $i < j$. Sve vrijednosti iznad dijagonale su jednake 0.

- *Permutacijska matrica* P - je matrica koja ima točno jednu 1 u svakom retku i 0 za sve ostale elemente. Takva matrica se zove permutacijska matrica zato što množenjem vektora x sa takvom matricom ima efekt permutacije (preraspodjele) elemenata od x.

- *Simetrična matrica* A - je matrica koja zadovoljava uvjet $A = A^T$.

Elementi matrice ili vektora su brojevi iz nekog brojevnog sustava, primjerice realni brojevi, kompleksni brojevi ili cijeli brojevi cjelobrojno dijeljeni (*modulo*) sa prostim brojevima. Brojevni sustavi definiraju kako se zbrajaju i množe brojevi. Možemo proširiti te definicije kako bismo obuhvatili operacije zbrajanja i množenja matrica.

Zbrajanje matrica. Ako su $A = (a_{ij})$ i $B = (b_{ij})$ $m \times n$ matrice onda je suma tih matrica $C = (c_{ij}) = A + B$ definirana kao

$$c_{ij} = a_{ij} + b_{ij}$$

za $i = 1, 2, \ldots, m$ i $j = 1, 2, \ldots, n$. Odnosno zbrajanje matrica je izračunata zbjrajanjem odgovarajućih elemenata matrica istih dimenzija. Nul matrica je identitet za matrično zbrajanje:

$$A + 0 = A$$

$$= 0 + A$$

Ako je λ broj i $A = (a_{ij})$ matrica, onda je $\lambda A = (\lambda a_{ij})$ skalarni produkt od A dobiven množenjem svakog lementa matrice sa elementom λ. Kao specijalni slučaj definiramo negativnu matricu $A = (a_{ij})$ da je $-1 \cdot A = -A$ tako da je *i*-ti i *j*-ti element od $-A$ jednak $-a_{ij}$. Vrijedi

$$A + (-A) = 0$$

$$= 0 + A$$

Ovom definicijom možemo definirati *oduzimanje matrica* kao zbrajanje negativne matrice.

Množenje matrica definiramo na sljedeći način. Uzmimo dvije matrice A i B koje su kompatibilne[11]. Ako je matrica $A = (a_i j)$ dimenzija $m \times n$ i ako je $B = (b_{jk})$ dimenzija $n \times p$ tada je njihov produkt $C = AB$ $m \times p$ matrica $C = (c_{ik})$ za koju vrijedi:

$$c_{ik} = \sum_{(j=1)}^{n} a_{ij} b_{jk}$$

za $i = 1, 2, \ldots, m$ i $j = 1, 2, \ldots, p$.

Jedinične matrice su identitet za množenje matrica:

$$I_m A = A I_n = A$$

za bilo koju $m \times n$ matricu A.

Množenje matrice sa nul matricom daje 0:

$$A0 = 0.$$

Množenje matrica je asocijativno ako su matrice A,B i C kompatibilne:

$$(AB)C = A(BC).$$

Množenje matrica je distributivno:

$$A(B + C) = AB + AC,$$

$$(B + C)D = BD + CD.$$

[11]Kompatibilne matrice su one čiji je broj redaka jedne matrice jednak broju stupaca druge matrice. Generalno se kod množenja matrica podrazumijeva da se radi o kompatibilnim matricama.

Množenje $n \times n$ matrica nije komutativno, osim ako je $n = 1$. Na primjer ako je matrica $A = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}$ i ako je $B = \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}$ onda je:

$$AB = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix},$$

a

$$BA = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}.$$

3.2 Determinanta matrice

Determinantu matrice $A = [a_{ij}] \in \mathbb{R}^{(n\times n)}$ ili $\mathbb{C}^{n\times n}$ definiramo kao polinom

$$det(A) = \sum_{p\in\Pi_n} \pi(p) a_{1p(1)} a_{2p(2)} a_{3p(3)} \dots a_{np(n)}$$

gdje $p(1), p(2), \dots, p(n)$ prolaze $n!$ mogućih permutacija brojeva $1, 2, \dots, n$ [8]. Inverzija u permutaciji p je svaki par $(p(i), p(j))$ za koji vrijedi $p(i) > p(j)$ kad je $i < j$. $I(p)$ je ukupni broj inverzija u permutaciji p. Permutacija p je parna odnosno neparna ako je $I(p)$ redom paran odnosno neparan broj. Parnost je funkcija $\pi : \Pi_n \to \{-1, 1\}$ definirana s $\pi(p) = (-1)^{I(p)}$ [10].

Determinanta $det(A)$ je reda n ako je A reda n. Umjesto $det(A)$ još se koristi oznaka $detA$, $|A|$, ili $\begin{vmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{vmatrix}$ [10].

Ako je $n = 3$ imamo $3! = 6$ permutacija, koje možemo zapisati na sljedeći način:

$$p_1 = \begin{pmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \end{pmatrix}, p_2 = \begin{pmatrix} 1 & 2 & 3 \\ 1 & 3 & 2 \end{pmatrix}, p_3 = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 1 & 3 \end{pmatrix}, p_4 = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \end{pmatrix},$$
$$p_5 = \begin{pmatrix} 1 & 2 & 3 \\ 3 & 1 & 2 \end{pmatrix}, p_6 = \begin{pmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \end{pmatrix}.$$

Dobivamo

$$\begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix} = (-1)^0 a_{11}a_{22}a_{33} + (-1)^1 a_{11}a_{23}a_{32} + (-1)^1 a_{12}a_{21}a_{33} + (-1)^2 a_{12}a_{23}a_{31}$$
$$+(-1)^2 a_{13}a_{21}a_{32} + (-1)^3 a_{13}a_{22}a_{31}$$
$$= a_{11}a_{22}a_{33} + a_{12}a_{23}a_{31} + a_{13}a_{21}a_{32} - a_{13}a_{22}a_{31} - a_{11}a_{23}a_{32} - a_{12}a_{21}a_{33}.$$

Može se vidjeti da računanje determinante matrica velikih dimenzija (primjerice već matrica reda $n = 10$) zahtjeva popriličan broj koraka i operacija koji značajno

raste povećanjem reda matrice. Postoje razne metode kojima se determinanta brže i lakše izračuna. Jednu od tih metoda ćemo koristiti i opisati u ovom radu.

Svojstva determinanti kvadratne matrice A:

- Ako je bilo koji redak ili stupac jednak 0 onda je $det(A) = 0$
- Determinanta od A je pomnožena sa λ ako su svi elementi retka ili stupca pomnoženi sa λ
- Determinanta od A se ne mijenja ako se neki redak (ili stupac) zbroji sa drugim retkom (ili stupcem)
- Determinanta od A je jednaka determinanti od A^T
- Determinanta od A se množi sa -1 ako bilo koja dva retka (ili stupca) zamijene mjesta
- Za sve kvadratne matrice A i B vrijedi: $det(AB) = det(A)det(B)$.

Vrijedi i da je matrica A dimenzija $n \times n$ singularna ako i samo ako je $det(A) = 0$.

Determinanta se može izračunati i njezinim razvijanjem po bilo kojem retku ili stupcu koristeći *Laplaceov razvoj determinante*:

po *i*-tom stupcu:

$$det(A) = \sum_{k=1}^{n} (-1)^{k+1} a_{ki} det(A_{ki})$$

po *i*-tom retku:

$$det(A) = \sum_{k=1}^{n} (-1)^{k+1} a_{ik} det(A_{ik}).$$

3.3 Dijagonalizacija i dekompozicija

Kako bismo pokazali važnost numeričke linearne algebre za temu ovoga rada, proći ćemo, kronološki, kroz nekoliko bitnijih metoda rješavanja matematičkih problema. Kao početnu točku uzmimo Gaussovu eliminaciju. Gaussova eliminacija je proces koji može riješiti n linearnih jednadžbi sa n nepoznanica koristeći se brojem aritmetičkih operacija reda veličine n^3. Odnosno rješava jednadžbu oblika $Ax = b$ gdje je A matrica dimenzija $n \times n$ a x i b su stupičasti vektori dimenzija n. Gaussova eliminacija je proces koji se gotovo svakodnevno pokreće na računalima diljem svijeta prilikom rješavanja sustava linearnih jednadžbi. Iako je n reda veličine 1000, vrijeme potrebno za računanje je ispod sekunde na normalnom modernom računalu. Ideja je potekla od kineskih učenjaka prije oko 2000 godina, a kasniji pridonosioci su bili Lagrange, Gauss i Jacobi. Moderan način opisivanja takvih algoritama je, predstavljen tek 1930-ih godina.

Pretpostavimo da, recimo, α puta red matrice A oduzmemo sa drugim redom. Ova operacija se može interpretirati kao množenje matrice A na lijevoj strani sa donjetrokutasom matricom M_1 koja sadrži identitet sa elementom različitim od nule $m_{21} = -\alpha$. Daljnje analogne operacije nad redovima odgovaraju daljnjim množenjima na lijevoj strani sa donjetrokutastim matricama M_j. Ako k koraka pretvori matricu A u gornjetrokutastu matricu U tada imamo $MA = U$ sa $M = M_k \dots M_2 M_1$ ili ako postavimo $L = M^{-1}$ dobivamo

$$A = LU.$$

Gdje je L jedinična donja trokutasta matrica, odnosno, donjetrokutasta matrica čiji su svi elementi na dijagonali jednaki 1. Pošto U predstavlja ciljanu strukturu i L sadrži operacije koje su bile potrebne da se dođe do toga, možemo reći da je Gaussova eliminacija proces donje-trokutaste "gornje-trokutizacije", te matrica A ima *LU faktorizaciju* ako postoji jedinična donja trokutasta matrica L i gornja trokutasta matrica U.

Primjer LU dekompozicije matrice:
Imamo matricu $A = \begin{pmatrix} 2 & 5 \\ 1 & 5 \end{pmatrix}$.
Koristit ćemo Gaussovu eliminaciju kako bismo dobili gornjetrokutastu matricu U. Primjenom elementarnih operacija nad redovima matrice A $R_2 - \frac{1}{2}R_1 \to R_2$ dobivamo matricu U:

$$U = \begin{pmatrix} 2 & 5 \\ 0 & \frac{5}{2} \end{pmatrix}.$$

Naša odgovarajuća matrica L imat će jedinice na svojoj dijagonali:

$$L = \begin{pmatrix} 1 & 0 \\ * & 1 \end{pmatrix},$$

te se sadržaj ispod dijagonale dobiva inverznim operacijama koje su primijenjene na U. U našem slučaju to je $R_2 + \frac{1}{2}R_1 \to R_2$:

$$L = \begin{pmatrix} 1 & 0 \\ \frac{1}{2} & 1 \end{pmatrix}.$$

Odnosno LU dekompozicija matrice A je:

$$A = \begin{pmatrix} 2 & 5 \\ 1 & 5 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ \frac{1}{2} & 1 \end{pmatrix} \begin{pmatrix} 2 & 5 \\ 0 & \frac{5}{2} \end{pmatrix} = LU.$$

Također, mnogi su algoritmi numeričke linearne algebre bazirani na dobivanju matrice kao produkt matrica posebnih svojstava.

Ovako postavljenim okvirom, lakše se može opisati sljedeći algoritam koji se treba uzeti u obzir: Gaussova eliminacija sa pivotnim elementom. Nema svaka matrica LU faktorizaciju te je time, gore opisana, gaussova eliminacija nestabilna, odnosno time se povećava greška zaokruživanja sa potencijalno velikim iznosima. Stabilnost

se može postići zamjenom redaka tijekom eliminacije tako da se što više elemenata nalazi na dijagonali. Taj proces je poznat kao zakretanje (eng. *pivoting*). Pošto je zakretanje proces koji se vrši na redovima matrice ono ponovno odgovara množenju matrice sa lijeva.

Faktorizacija matrice odgovara Gaussovoj eliminaciji sa pivotnim elementom ("*eng. with pivoting*") je

$$PA = LU$$

gdje je U gornje trokutasta, L je jedinična donje trokutasta i P je permutacijska matrica, tj. matrica identiteta sa permutiranim redovima.

Ako su permutacije izabrane tako da dovedu najveći unos ispod dijagonale u k-ti stupac (na mjesto(k, k)) prije k-tog koraka eliminacije onda matrica L ima dodatno svojstvo , a to je da $|l_{ij}| \leq 1$ za sve i i j.

U praksi eliminacija sa zakretanjem čini Gaussovu metodu gotovo savršeno stabilnom, i rutinski se izvodi u skoro svim računalnim programima koji trebaju riješiti linearni sustav jednadžbi. Za neke rubne slučajeve, to jest određene specijalne matrice, Gaussova eliminacija i dalje nestabilna pa i u slučajevima kada se uvede metoda zakretanja [7].

Također vrijedi spomenuti algoritme bazirane na ortogonalnim ili unitarnim matricama gdje vrijedi: $Q^{-1} = Q^T$ za realne i $Q^{-1} = Q^*$ za kompleksne matrice gdje Q^* označava konjugirano transponiranu matricu.
Ako je A $m \times n$ matrica sa $m \geq n$, *QR faktorizacija* matrice A je

$$A = QR$$

gdje je Q sačinjena od ortonormiranih stupaca, a matrica R je gornje-trokutasta.

Iz *QR faktorizacije* je proizašla bogata kolekcija algebarskih algoritama. QR faktorizacija se može sama koristiti kako bi se riješio problem najmanjih kvadrata i da bi se konstruirale ortonormirane baze. Značajnija je njena uporaba kao jedan od koraka u drugim algoritmima. Konkretno, jedan od centralnih problema numeričke linearne algebre je određivanje svojstvenih vrijednosti i svojstvenih vektora kvadratne matrice A. Ako matrica A ima potpun skup svojstvenih vektora, onda formiranjem matrice X čiji stupci su ti svojstveni vektori i formiranjem dijagonalne matrice D čiji su elementi dijagonale odgovarajuće svojstvene vrijednosti, dobivamo

$$AX = XD$$

i stoga, jer je X nesingularna,

$$A = XDX^{-1}$$

je "*dekompozicija svojstvene vrijednosti*".

QR algoritam je jedan od iznimnih uspjeha numeričke analize, a njegov veliki utjecaj potvrđuje se kroz široko korištene softverske proizvode. Algoritmi i analize bazirane na njemu su u 1960-ima doveli do koda u programskim jezicima *Agol* i

Fortran,a kasnije do programske (”softverske”) biblioteke *EISPACK*[12] i njegovog nasljednika *LAPACK-a*[13]. Navedene metode su ugrađene u numeričke biblioteke opće namjere, kao što su NAG, ISML i kolekcija ”numerički recepti”, te u programska okruženja (alate) za rješavanje problema, kao što su MATLAB, Maple i Mathematica [7].

Obratimo pažnju i na EISPACK-ovog ”srodnika” LINPACK koji se koristi za rješavanje linearnih sustava i jednadžbi. LINPACK je postao originalna baza svih alata za mjerenje performansi računala (eng. *benchmark*) kojeg koriste svi proizvođači računala. Razlog ulaska superračunala na listu TOP500 (koja se od 1993 godine osvježava dva puta godišnje) je efikasnog rješavanja određenih problema s matricom $Ax = b$ dimenzija od 100 do par milijuna [7].

Dekompoziciju svojstvenih vrijednosti poznaju svi matematičari, ali razvoj numeričke linearne algebre je predstavio *SVD*[14]. Beltrami, Jordan i Sylvester su otkrili SVD na kraju devetnaestog stoljeća, a proslavili su ju Golub i ostali numerički analitičari oko 1965. godine.

Dosadašnji teoretski prikaz se odnosio na ”klasičnu” linearnu algebru koja se rodila u razdoblju od 1950 do 1975. Poslije tog razdoblja se pojavljuje potpuno novi skup alata: metode za probleme velikih razmjera baziranih na Krylovim potprostornim ponavljanjima (iteracije). Krylove potrpostorne iteracije proizašle su zajedno s konjugiranim gradijentom i Lanczosovim iteracijama koje su objavljene u 1952. godini. Tih godina, zbog nedovoljne jačine računala prilikom rješavanje problema velikih razmjera, navedena metoda nije mogla biti kompetitivna. Dobivaju na važnosti 1970-ih godina zbog rada Reida, Paigea, a pogotovo van der Vorsta i Meijerinka koji su proslavili ideju prekondicioniranja.

Također, jedan od najvećih neriješenih problema u numeričkoj analizi je: može li proizvoljna matrica dimenzija $n \times n$ biti invertirana u $O(n^\alpha)$ operacija za svaku $\alpha > 2$? (Problem rješavanja sustava $Ax = b$ ili izračunavanja umnoška matrica AB su ekvivalentni.) Gaussova eliminacija ima vrijednost $\alpha = 3$ te autor navodi da se eksponent smanjuje sve do 2.376 za određene rekurzivne (iako nepraktične) algoritme koje su objavili Coppersmith i Winograd 1990. godine [7].

3.3.1 Primjer implementacije algoritama za računanje determinante

U nastavku primjenjujemo *LU algoritam* dekompozicije matrice odnosno jednu od metoda (*Croutova metoda* dekompozicije) primjene algoritma čija će implementacija biti detaljno prikazana. Taj je algoritam jedan od mnogobrojnih načina da se dođe do izračuna determinante *n-dimenzionalne* matrice. Navedeni primjeri implementacija

[12]EISPACK (eng. EIgenSystem PACKage) - programski paket algoritama sustava svojstvenih vrijednosti i vektora

[13]LAPACK (eng. Linear Algebra PACKage) - programski paket algoritama linearne algebre (algebarskih rutina). [9]

[14]SVD (eng. singular value decomposition) - dekompozicija jedinične vrijednosti

algoritma, koji su opisani u ovom poglavlju, su cjelovito sadržani u dodatku.

U implementaciji algoritma *LU dekompozicije* matrice, u jezicima C i Fortranu, nema puno razlike osim u anotacijama i samoj sintaksi te nekoliko tehničkih detalja potrebnih za osiguranje rada algoritama sa zadanim vrijednostima. Usporedno ću prikazati implementacije tog algoritma kao jednog od mnogih koji se mogu koristiti kako bi se izračunala determinanta matrice n-tog retka. Algoritmi koji se koristi su numerička izvedba LU dekompozicije. Konkretno se koristi *Croutova metoda dekompozicije matrice*.

Jako je zanimljivo kako Croutova metoda dekompozicije matrice dosta jednostavno rješava problem $N^2 + N$ jednadžbi za sve α i β elemente razmještanjem tih jednadžbi određenim redoslijedom. Redoslijed je sljedeći:

- Postavi $\alpha_{ii} = 1$ (elemente dijagonale donje trokutaste matrice) za svaki $i = 1, \ldots, N$
- Za svaki $j = 1, 2, \ldots, N$ obavi dvije procedure: Prvo izračunaj β_{ij}

$$\alpha_{ij}\beta_{ij} = a_{ij} - \sum_{k=1}^{i-1} \alpha_{ik}\beta_{kj}. \tag{3.1}$$

  Zatim izračunaj

$$\alpha_{ij} = \frac{1}{\beta_{jj}}(a_{ij} - \sum_{k=1}^{i-1} \alpha_{ik}\beta_{kj}). \tag{3.2}$$

  Budi siguran da su obje procedure obavljene prije nego što ideš na idući *j* [14].

Također, Croutov algoritam koristi djelomično pivotiranje.

Algoritam, izveden u Fortranu, koji računa LU dekompoziciju smo nazvali `ludcmp(a,n,np,indx,d)`.

```
subroutine ludcmp(a,n,np,indx,d)
    integer n,np,indx(n),NMAX
    real TINY,a(np,np)
    parameter (NMAX=5000,TINY=1.0e-20)
    integer i,imax,j,k
    double precision aamax,dum,sum,vv(NMAX)
    real*16 d
```

NMAX nam određuje najveći očekivani *n* i mali broj TINY nam služi kako bi zamijenio nulu u algoritmu u slučaju singularne matrice. Naša matrica je označena s varijablom *a* i dimenzijama (np, np) gdje je u našem slučaju $np = n$. Matrica *a* i varijabla *n* su ulazne varijable, a izlazna varijabla nam je, ista varijabla *a* koja vraća matricu oblika:

$$\begin{pmatrix} \beta_{11} & \beta_{12} & \cdots & \beta_{1n} \\ \alpha_{21} & \beta_{22} & \cdots & \beta_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_{n1} & \cdots & \alpha_{n(n-1)} & \beta_{nn} \end{pmatrix}.$$

Također, izlazna varijabla je `indx(n)` – vektor koji bilježi efekt permutacije redova a izvršen je parcijalnim pivotiranjem. Varijabla d na izlazu daje ± 1 ovisno o broju promjena redova, je li broj bio paran $(+)$ ili neparan $(-)$. U našem slučaju služi i kao varijabla u koju se sprema izračunata vrijednost determinante, dok se za predznak više ne treba brinuti jer je osiguran.

Petlja kojom idemo po stupcima matrice pomoću Croutove metode:

```
do j=1,n
        do i=1,j-1
            sum=a(i,j)

            do k=1,i-1
                sum=sum-a(i,k)*a(k,j)
            enddo

            a(i,j)=sum
        enddo

        aamax=0.

            do i=j,n
                sum=a(i,j)

                do k=1,j-1
                    sum=sum-a(i,k)*a(k,j)
                enddo

                a(i,j)=sum
                dum=vv(i)*abs(sum)
                if (dum.ge.aamax) then
                    imax=i
                    aamax=dum
                endif
            enddo

            if (j.ne.imax)then

                do k=1,n
                    dum=a(imax,k)
                    a(imax,k)=a(j,k)
                    a(j,k)=dum
                enddo
```

```
                d=-d
                vv(imax)=vv(j)
            endif
            indx(j)=imax
            if(a(j,j).eq.0.)a(j,j)=TINY
            if(j.ne.n)then
                dum=1./a(j,j)

                do i=j+1,n
                    a(i,j)=a(i,j)*dum
                enddo

            endif

    enddo
```

Prva petlja unutar velike petlje nam predstavlja jednadžbu (3.1) izuzev $i = j$.

```
do i=1,j-1
  sum=a(i,j)

       do k=1,i-1
        sum=sum-a(i,k)*a(k,j)
       enddo

       a(i,j)=sum
enddo
```

Dio jednadžbe gdje je *i=j* jednadžbe (3.1) i $i = j + 1 + \cdots + N$ jednadžbe (3.2) predstavlja druga petlja:

```
do i=j,n
    sum=a(i,j)

    do k=1,j-1
       sum=sum-a(i,k)*a(k,j)
    enddo

    a(i,j)=sum
    dum=vv(i)*abs(sum)
    if (dum.ge.aamax) then
        imax=i
```

```
        aamax=dum
    endif
enddo
```

Te u slučaju potrebe za zamjenom redova i promjenom predznaka varijable d:

```
if (j.ne.imax)then

    do k=1,n
        dum=a(imax,k)
        a(imax,k)=a(j,k)
        a(j,k)=dum
    enddo

    d=-d
    vv(imax)=vv(j)
endif
```

završno, dijeljenje s pivotnim elementom:

```
if(j.ne.n)then
    dum=1./a(j,j)
    do i=j+1,n
     a(i,j)=a(i,j)*dum
    enddo
endif
```

Algoritam izveden u programskom jeziku C:

```
void crout(double **A,double **L, double **U, int n) {
    int i, j, k;
    double sum = 0;

    for (i = 0; i < n; i++) {
        U[i][i] = 1;
    }

    for (j = 0; j < n; j++) {
        for (i = j; i < n; i++) {
            sum = 0;
            for (k = 0; k < j; k++) {
                sum = sum + L[i][k] * U[k][j];
            }
            L[i][j] = A[i][j] - sum;
```

```
        }
    }
```

U ovom primjeru algoritma također imamo dijelove koj predstavljaju korake našeg algoritma koji smo opisali korištenjem programskog jezika Fortran. Primjerice, gore navedenom petljom opisujemo implementaciju prve jednadžbe (3.1), dok se druga jednadžba (3.2) može opisati idućom petljom:

```
for (i = j; i < n; i++) {
    sum = 0;
    for(k = 0; k < j; k++) {
        sum = sum + L[j][k] * U[k][i];
    }
    if (L[j][j] == 0) {
        printf("det(L) je blizu 0! Ne može se dijeliti sa 0...\n");
        exit(EXIT_FAILURE);
    }
    U[j][i] = (A[j][i] - sum) / L[j][j];
}
```

Algoritam se razlikuje u tome što ovaj algoritam vraća matrice L i U (donje odnosno gornje trokutastu matricu) dok je algoritam napisan u fortranu vraćao izmijenjenu matricu sačinjenu od te dvije matrice.

Nakon što se izvela dekompozicija matrice, gdje je taj dio izveden kao procedura, u glavnom programu se izvršilo računanje determinante matrice:

Fortran:

```
do j=1,n
        DET=DET*a(j,j)
enddo
```

C:

```
for (i = 1; i < n; i++){
    dijagL = dijagL*L[i][i];
    dijagU = dijagU*U[i][i];
}
```

Motivaciju i inspiraciju za opisane algoritme smo pronašli u raznim izvorima među kojima su najznačajniji "*Numerical Recipies in Fortran 77*" [14] i "*Introduction to algorithms*" [1].

4 Bazične algebarske rutine (BLAS)

BLAS (eng. *Basic Linear Algebra Subprograms*) odnosno bazične algebarske rutine su sučelje za često korištene fundamentalne operacije linearne algebre.

Konceptualno, sučelje BLAS-a podržava prijenosne implementacije aplikacija (softwerskih rješenja) visokih performansi čiji je zadatak intenzivno baviti se računima s matricama i vektorima. Razvojni programeri, koji žele koristiti rutine, fokusiraju se na račun u smislu operacija koje pruža BLAS, puštajući time stručnjake da se bave optimizacijom softvera na razini arhitekture [11].

Softverska izvedba BLAS-a je biblioteka rutina pisanih u FORTRANU 77 koje efikasno obavljaju osnovne algebarske operacije (kao npr. skalarni produkt). Efikasnost tih rutina značajno ovisi o njihovoj usklađenosti s arhitekturom na kojoj se izvršavaju. Stoga je uputno koristiti ATLAS (Automatically Tuned Linear Algebra Software) koji automatski optimizira BLAS rutine na danoj platformi.

LAPACK je biblioteka rutina pisanih u FORTRANU 77 namijenjenih rješavanju linearnih algebarskih sustava, problema najmanjih kvadrata, računanju svojstvenih i singularnih vrijednosti matrice itd [12].

U ovome radu koristili smo automatski optimizirane (ATLAS[15]) biblioteke BLAS i LAPACK. ATLAS pruža sučelja za programske jezike C i Fortran77 za učinkovitu prijenosnu implementaciju BLAS-a i nekoliko rutina iz LAPACK-a [13].

4.1 Osnovna funkcionalnost

Sučelje BLAS-a je organizirano u tri razine, ovisno o tome radi li se o operacijama vektora nad vektorima (Level-1 BLAS), vektora nad matricama (Level-2 BLAS) ili matrica nad matricama (Level-3 BLAS).

Sve razine BLAS-ovog sučelja će biti objašnjene s *Choleskyjevom faktorizacijom* $n \times n$ matrice A.

4.1.1 Operacije vektora nad vektorima (Level-1 BLAS)

Kada je A konačna simetrična pozitivna matrica (svojstvo koje je potrebno da se algoritam izvrši), njena Cholesky faktorizacija je dana s donjetrokutastom matricom L takvom da vrijedi $A = LL^T$. Faktorizacija Choleskog je trokutasta faktorizacija pozitivno definitne matrice. Kažemo da je simetrična $n \times n$ matrica *pozitivno definitna* ako za sve $x \in \mathbb{R}^n$ različite od 0 vrijedi

$$x^T Ax > 0.$$

Dijagonalni elementi pozitivno definitne matrice su nužno pozitivni. Pokazuje se da pozitivna definitnost sustava osigurava egzistenciju LU faktoriziranja bez pivotiranja.

[15]Vodiče za instalaciju i pomoć pri korištenju programa ATLAS sam pročitao sa službene stranice (zaključno s 3.9.2016.)

Ako se dobije da je matrica pozitivno definitna, njezin prvi element na glavnoj dijagonali je pozitivan što znači da se postupak eliminacije može nastaviti, te se time dokazuje postojanost faktorizacije matrice A u oblik LL^T, gdje je L donjetrokutasta. Algoritam za tu operaciju se može izvesti na ovaj način: Podjelom

$$A \to \left(\begin{array}{c|c} \alpha_{11} & * \\ \hline a_{21} & A_{22} \end{array}\right) \text{ i } L \to \left(\begin{array}{c|c} \lambda_{11} & 0 \\ \hline l_{21} & L_{22} \end{array}\right),$$

gdje su α_{11} i λ_{11} skalari, a_{21} i l_{21} su vektori, A_{22} je simetrična matrica, L_{22} je donje trokutasta i $*$ označava simetrični dio matrice A koji nije korišten. Tada

$$\begin{aligned}\left(\begin{array}{c|c} \alpha_{11} & * \\ \hline a_{21} & A_{22} \end{array}\right) &= \left(\begin{array}{c|c} \lambda_{11} & 0 \\ \hline l_{21} & L_{22} \end{array}\right)\left(\begin{array}{c|c} \lambda_{11} & 0 \\ \hline l_{21} & L_{22} \end{array}\right)^T \\ &= \left(\begin{array}{c|c} \lambda_{11}^2 & * \\ \hline \lambda_{11} l_{21} & l_{21} l_{21}^T + L_{22} L_{22}^T \end{array}\right).\end{aligned}$$

Iz ovih relacija proizlazi algoritam za računanje matrice A:

- $\alpha_{11} \leftarrow \sqrt{\alpha_{11}}$
- $a_{21} \leftarrow {}^{a_{21}}/_{\alpha_{11}}$
- $A_{22} \leftarrow -a_{21}a_{21}^T + A_{22}$, ažurirajući samo donje-trokutasti dio matrice A (to se zove simetrično rang-1 ažuriranje)
- Nastavi sa prepisivanjem A_{22} sa L_{22} gdje $A_{22} = L_{22}L_{22}^T$.

Jednostavan kod u Fortranu izgleda ovako:

```
do j=1, n
    A( j,j ) = sqrt( A( j,j ) )
    do i=j+1,n
        A( i,j ) = A( i,j ) / A( j,j )
    enddo
    do k=j+1,n
        do i=k,n
            A( i,k ) = A( i,k ) - A( i,j ) * A( k,j )
        enddo
    enddo
enddo
```

Često susretane operacije s vektorima su umnožak vektora sa skalarom $x \leftarrow \alpha x$, skalarni umnožak $\alpha \leftarrow x^T y$ i vektorsko zbrajanje sa umnoškom vektora i skalara $y \leftarrow \alpha x + y$, gdje su *x* i *y* vektori odgovarajuće duljine i α je skalar. Ova zadnja operacija je još poznata i kao *axpy*[16] alfa puta *x* plus *y*.

Cholesky faktorizacija napisana u smislu takvih operacija izgleda ovako:

[16] axpy - eng. alpha times x plus y

```
do j=1, n
    A( j,j ) = sqrt( A( j,j ) )
    call dscal( n-j, 1.0d00/A( j,j ),
            A( j+1, j ), 1 )
    do k=j+1,n
        call daxpy( n-k+1, -A( k,j ),
                A(k,j), 1, A( k, k ), 1 )
    enddo
enddo
```

U ovom primjeru programa su korištene operacije `dscal` i `daxpy`. U pozivima tih operacija prvi znak nam govori da se radi o računanju s vrijednostima dvostruke preciznosti (`d` - double precision). Poziv `dscal` izvršava računanje $a_{21} \leftarrow {}^{a_{21}}/_{\alpha_{11}}$ dok je petlja

```
do i=k,n
    A( i,k ) = A( i,k ) - A( i,j ) * A( k,j )
enddo
```

zamijenjena pozivom

```
call daxpy( n-k+1, -A( k,j ), A(k,j), 1, A( k, k ), 1 )
```

Ako operacije koje su podržane s `dscal` i `daxpy` postignu visoke performanse, na ciljanoj arhitekturi, onda će također i implementacija Choleskyjeve faktorizacije postignuti visoke performanse jer najviše računanja obavi pozivom tih operacija [11].

Općenito, opis za pozive rutina *BLAS-a prve razine*[17] prikazane su na ovaj način (konkretno na našem primjeru funkcije koja implementira operaciju $y = \alpha x + y$):

```
_axpy( n, alpha, x, incx, y, incy )
```

gdje znak "_" određuje tip podatka (dane mogućnosti su prikazane u tablici 4.1). Ostatak naziva rutine označava operaciju: `axpy` - alfa puta x plus y. `n` označava broj elemenata vektora *x* i *y*, *alpha* je skalar α, dok `x` i `y` označavaju memorijske lokacije gdje su spremljeni prvi elementi vektora *x* i *y*.Parametri `incx` i `incy` su jednaki iznosu povećanja kojim se traže elementi vektora *x* i *y* po memoriji računala. Najčešće korištene rutine prve razine BLAS-a, uz prethodno opisanu rutinu, se nalaze u tablici 4.2 [11].

Prvo BLAS sučelje je predstavljeno 1970-ih kada su vektorska superračunala bila široko korištena za računalnu znanost. Takva računala su postizala performanse blizu maksimuma svojih mogućnosti pod uvjetom da je većina računanja bila zadana u vektorskim operacijama i ukoliko je bio ostvaren zajednički pristup memoriji. To sučelje se sada naziva Level-1 BLAS [11].

[17] Level-1 BLAS

s	Realna vrijednost jednostruke preciznosti (eng. single precision)
d	Realna vrijednost dvostruke preciznosti (eng. double precision)
c	Kompleksna vrijednost jednostruke preciznosti (eng. single precision complex)
z	Kompleksna vrijednost dvostruke preciznosti (eng. double precision complex)

Tablica 4.1: Prikaz tipova podataka koji se koriste za BLAS rutinu _axpy

Rutine/funkcije	Operacije
_swap	$x \leftrightarrow y$
_scal	$x \leftarrow \alpha x$
_copy	$y \leftarrow x$
_axpy	$y \leftarrow \alpha x + y$
_dot	$x^T y$
_nrm2	$\|x\|_2$
_asum	$\|re(x)\|_1 + \|im(x)\|_1$
i _max	$min(k) : \|re(x_k)\| + \|im(x_k)\| = max(\|re(x_i)\| + \|im(x_i)\|)$

Tablica 4.2: Lista nekih BLAS rutina prve razine

4.1.2 Operacije s vektorima i matricama (Level-2 BLAS)

Iduća razina BLAS-a su operacije s vektorima i matricama. Najjednostavniji primjer za takve operacije je umnožak vektora i matrica: $y = Ax$ gdje su *x* i *y* vektori, a A je matrica. Još jedan primjer je $A_{22} = -a_{21}a_{21}^T + A_{22}$ (ažuriranje matricom ranga 1) u Cholesky faktorizaciji. Ova se operacija može prikazati ovako:

```
do j=1, n
        A( j,j ) = sqrt( A( j,j ) )
        call dscal( n-j, 1.0d00 / A( j,j ), A( j+1, j ), 1 )
        call dsyr( 'Lower triangular',n-j, -1.0d00,
        A( j+1,j ), 1, A( j+1,j+1 ),lda )
enddo
```

`Dsyr` je rutina koja implementira operaciju ažuriranja matricom ranga 1 dvostruke preciznosti. Čitljivost koda se poboljšala uvođenjem rutina koje implementiraju određene operacije koje se pojavljuju u algoritmu: `dscal` za $a_{21} = {}^{a_{21}}/_{\alpha_{11}}$ i `dsyr` za $A_{22} = -a_{21}a_{21}^T + A_{22}$.

Pravilo imenovanja BLAS rutina razine dva je određen na sljedeći način: _XXYY, gdje:

- "_" može poprimiti *s*, *d*, *c* i *z* vrijednosti
- XX označava oblik matrice (Tablica 4.3)

XX	Oblik matrice
ge	Generalni oblik (kvadratna)
sy	Simetrična
he	Hermitska
tr	trokutasta

Tablica 4.3: Oblici nekih od podržanih BLAS rutina

- YY označava operaciju koja se koristi (Tablica 4.4) [11]

YY	Operacija
mv	Umnožak matrice i vektora
sv	Vektor rješenja
r	Ažuriranje matricom ranga 1
r2	Ažuriranje matricom ranga 2

Tablica 4.4: BLAS operacije druge razine

Naš primjer poziva rutine druge razine

```
dsyr( uplo, n, alpha, x, incx, A, lda )
```

implementira operaciju $A = \alpha x x^T + A$ koja ažurira gornji ili donji trokutasti dio matrice A tako da se umjesto `uplo` stavi "*Lower triangular*" ili "*Upper triangular*"[18] matrica. Parametar *lda*[19] nam govori o veličini koraka kojim će se u memoriji tražiti ostali članovi matrice A. U tablici 4.5 [11] su navedene najčešće korištene rutine druge razine BLAS-a.

4.1.3 Operacije matrica s matricama (Level-3 BLAS)

Problem s vektorskim operacijama i matrica-vektor operacijama je taj da izvršavaju $O(n)$ koraka računa nad $O(n)$ podataka odnosno, $O(n^2)$ koraka računa nad $O(n^2)$ podataka. To otežava, ako ne i onemogućuje, iskorištavanje priručne memorije u doba kada brzina obrade daleko nadmašuje brzinu radne memorije, osim ako je problem relativno malen (stane u priručnu memoriju). Rješenje je da se izvrše računi u smislu

[18] eng. Lower triangular - donje trokutasta matrica; eng. Upper triangular - gornje trokutasta matrica

[19] lda (eng. the leading dimension of matrix A) - dimenzija matrice A kojom se traže prvi članovi stupca matrice u memoriji

Rutine/funkcije	Operacije
_gemv	generalna matrica - vektor (umnožak)
_symv	simetrična matrica - vektor (umnožak)
_rmv	Trokutasta matrica - vektor (umnožak)
_trsv	Vektor rješenja trokutaste matrice
_ger	Ažuriranje generalnom matricom ranga 1
_syr	Ažuriranje simetričnom matricom ranga 1
_syr2	Ažuriranje simetričnom matricom ranga 2

Tablica 4.5: Lista nekih BLAS rutina druge razine

operacija kao što je umnožak matrica s matricama. Ponovno uzimamo za primjer Cholesky faktorizaciju.

Koraci algoritma:

- $A_{11} = L_{11}$ gdje vrijedi $A_{11} = L_{11}L_{11}^T$ (Cholesky faktorizacija manje matrice)
- $A_{21} = L_{21}$ gdje vrijedi $L_{21}L_{11}^T = A_{21}$ (trokutasto rješenje sa višestrukim desnim stranama)
- $A_{22} = -L_{21}L_{21}^T + A_{22}$ koji ažurira samo donji trokutasti dio matrice A_{22}
- Nastavi s prepisivanjem $A_{22}sL_{22}$ gdje je $A_{22} = L_{22}L_{22}^T$

Kod u Fortranu koji predstavlja navedene korake Cholesky faktorizacije matrice[11]:

```
do j=1, n, nb
        jb = min( nb,n-j+1 )
        call chol( jb, A(j,j),lda )
        call dtrsm( 'Right','Lower triangular','Transpose','Nonunit diag',
                    J-JB+1,JB,1.0d00,A(j,j),lda,A(j+jb,j),lda )
        call dsyrk( 'Lower triangular','No transpose',J-JB+1,JB,-1.0d00,
                    A(j+jb,j),lda,1.0d00,A(j+jb,j+jb),lda )
enddo
```

Podrutina `chol` izvodi Cholesky faktorizaciju gdje su `dtrsm` i `dsyrk` BLAS rutine razine tri:

- dtrsm implementira $A_{21} \leftarrow L_{21}$ gdje je $L_{21}L_{11}^T = A_{21}$
- dsyrk implementira $A_{22} \leftarrow -L_{21}L_{21}^T + A_{22}$ [11]

Većina računa koji se izvodi je sada na operacijama matrica s matricama, a te operacije mogu dostići visoke performanse.

Pravilo imenovanja BLAS rutina razine tri je slično BLAS rutinama razine 2. Naš primjer poziva rutine je:

```
dsyrk( uplo, trans, n, k, alpha, A, lda, beta, C, ldc )
```

koja implementira operacije $C \leftarrow \alpha AA^T + \beta C$ odnosno $C \leftarrow \alpha A^T A + \beta C$, ovisno o tome je li za `trans` odabrano redom "*No transpose*" ili "*Transpose*". Ovisno o odabiru parametra `uplo` s "*Lower triangular*" ili "*Upper triangular*" ažurira se redom donji trokutasti dio matrice C ili donji trokutasti dio matrice C. Parametri `lda` odnosno `ldc` govore o veličini koraka kojim će se u memoriji tražiti ostali članovi matrice A odnosno C.

U tablici 4.6 su prikazane najčešće korištene BLAS operacije treće razine[11]:

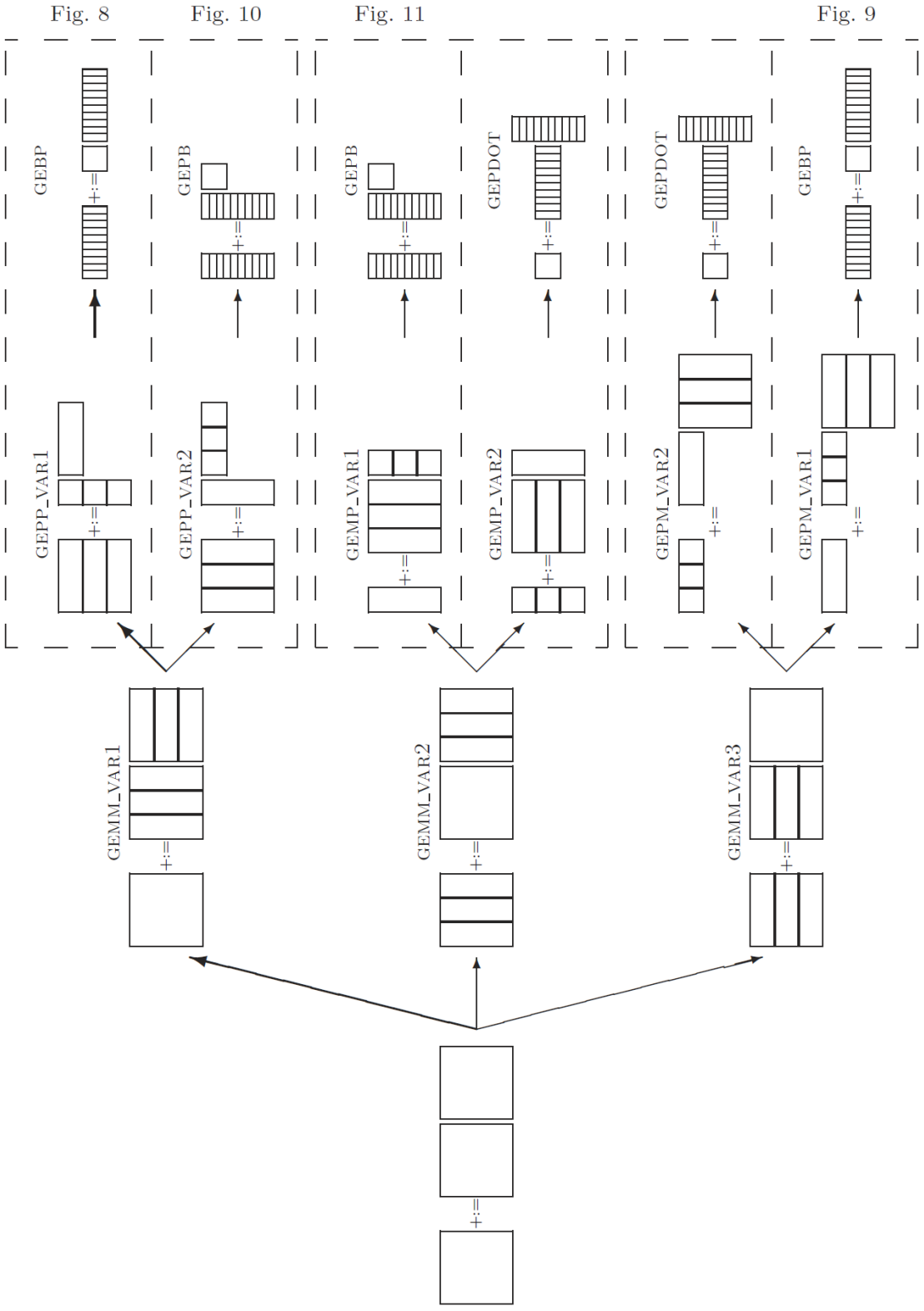
Rutine/funkcije	Operacije
_gemm	Umnožak matrica
_symm	Umnožak simetričnih matrica
_trmm	Umnožak trokutastih matrica
_trsm	Trokutasto rješenje sa višestrukim desnim stranama
_syrk	Simetrično ažuriranje ranga k
_syr2k	Simetrično ažuriranje ranga 2k

Tablica 4.6: Lista nekih BLAS rutina treće razine

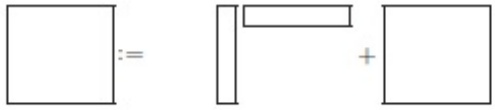
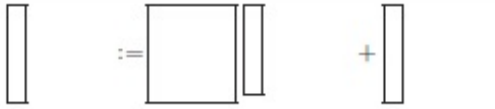
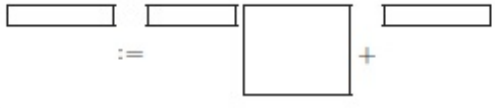
4.2 Raščlamba rada pojedinih rutina

Na slici 4.1[20] se može vidjeti kako se rutina GEMM može razložiti na specijalne slučajeve koji su prikazani na slici 4.2[21]. Prema tome „GEMM može biti razložen na više poziva GEPP, GEMP ili GEPM rutina. One same se mogu razložiti na više poziva GEBP, GEPB ili GEPDOT kernela. Ideja je takva da ako ta tri kernela najniže razine postignu visoke performanse, onda će i ostali slučajevi GEMM rutine.“ [16]

[20] Slika 4.1 je preuzeta iz [16] (str. 5.)
[21] Slika 4.2 je preuzeta iz [16] (str. 4.)



Slika 4.1: Slojeviti pristup implementaciji rutine GEMM

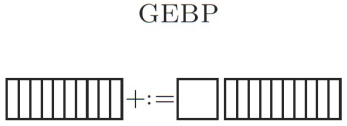
m	n	k	Illustration	Label
large	large	large		GEMM
large	large	small		GEPP
large	small	large		GEMP
small	large	large		GEPM
small	large	small		GEBP
large	small	small		GEPB
small	small	large		GEPDOT
small	small	small		GEBB

Slika 4.2: Specijalni oblici GEMM rutine za $C := AB + C$ gdje su matrice A,B i C redom dimenzija $m \times k$, $k \times n$ i $m \times n$

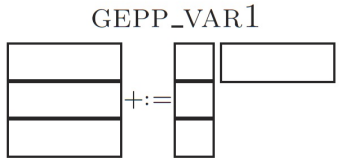
Pseudokod:

```
for p = 1:K
        for i = 1:M
                for j = 1:N
                        C_ij += A_ip B_pj
                endfor
        endfor
endfor
```

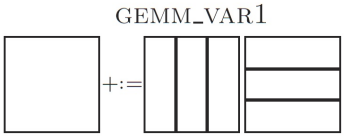
je reprezentacija jednog od scenarija slojevitog pristupa sa slike 4.1. gdje je unutarnja petlja ekvivalentna operaciji koja koristi GEBP:



druga (srednja) petlja je ekvivalentna operaciji koja koristi GEPP:



i treća (vanjska) petlja je ujedno i početni problem koji koristi GEMM:



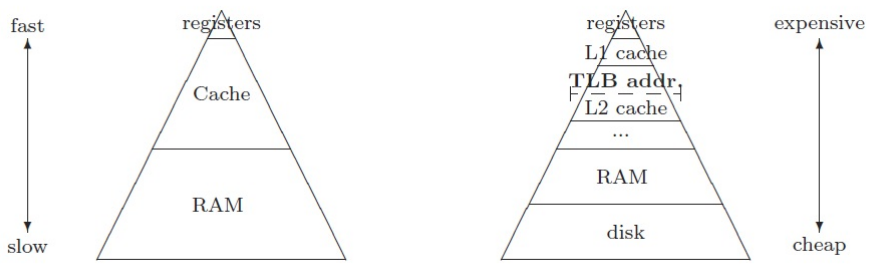
Takav algoritam nam govori da se problemi, prikazani na slici 4.1, uvijek uzimaju kao trostruko ugniježđene petlje. Kako bi ostvarili visoko-performansni GEPB, GEBP i GEPDOT (BLAS rutine razine 1 i najniži sloj slojevitog pristupa) za početak se treba analizirati "cijena" prijenosa podataka između različitih razina memorije te proučiti i pokušati iskoristiti propusnosti tih podataka [16].

Na slici 4.3[22] se može vidjeti odnos brzina, cijena i veličina raznih slojeva i vrsta memorije u računalu. Na lijevoj strani slike se prikazuje jednostavan model višeslojne memorije s "ubačenim" jednim slojem priručne memorije između RAM[23] memorije i registara. Najbitniji problemi povezani sa visoko-performansnim implementacijama GEPB, GEBP i GEPDOT rutina mogu se opisati koristeći se navedenom pojednostavljenom arhitekturom [16].

Fokusiramo se na GEBP sa matricama $A \in R^{m_c \times k_c}$, $B \in R^{k_c \times n}$ i $C \in R^{m_c \times n}$.

[22] Slika 4.3 je preuzeta iz [16] (str. 7.); eng. cache memorija – hrv. priručna memorija (L1,L2 označavaju razine); TLB addr (Translation lookaside buffer) – tip priručne memorije koji koji koristi MMU (Memory-management unit) a sprema zadnje prijenose između virtualne i fizičke memorije i može se još nazvati spremnikom adresnih translacija (address-translation cache)

[23] RAM – Random-Access Memory (hr. radna memorija)



Slika 4.3: Hijerarhijski model memorije prikazan u obliku piramide

Podijelimo B i C u particije

$$B = (B_0|B_1|\dots|B_{N-1}) \text{ i } C = (C_0|C_1|\dots|C_{N-1})$$

i postavimo pretpostavke [16]:

a) Dimenzije m_c i k_c su dovoljno male tako da zajedno A i n_r stupci iz B i C (B_j i C_j) stanu u priručnu memoriju

b) Ako su A, C_j i B_j u priručnoj memoriji onda se $C_j := AB_j + C_j$ može izračunati na vrhu performanse CPU-a

c) Ako je A u priručnoj memoriji, onda ostaje tamo sve do trenutka kada više nije potreban.

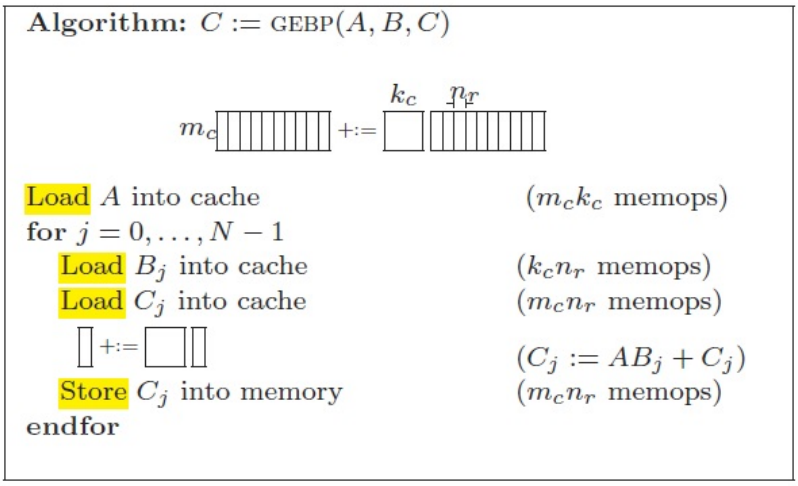
Pod navedenim pretpostavkama pristup GEBP, kao što je prikazano na slici 4.4[24], amortizira cijenu prenošenja podataka među memorijama. Ukupan omjer računanja i pomicanja podataka je:

$$\frac{2m_c k_c}{(m_c + k_c)} \tag{4.1}$$

Ulaskom u finiju analizu potrebno je odrediti razinu priručne memorije u koju se spremaju pripadajuće varijable. Pod pretpostavkama (a) – c)) prema jednadžbi 4.1, što je veći $m_c \times n_c$ to je bolje amortizirana cijena prijenosa između RAM memorije i priručne memorije. Time je određeno da je bolje spremiti A u priručnu memoriju koja je najdalje od registara (veći kapacitet).

Iduća stvar koju se treba uzeti u obzir, kod arhitekture, je TLB tablica. Tipična moderna arhitektura računala se koristi virtualnom memorijom, uz onu dodijeljenu fizičku memoriju, tako da se veličina iskoristive memorije ne ograničava fizičkom memorijom. Memorija je podijeljena u stranice (često fiksne) određene veličine. Tablica straničenja je tablica koja virtualne adrese spaja s fizičkim adresama odnosno, prati je li određena stranica u memoriji ili na disku. Problem je što je sama tablica spremljena u memoriji računala te se pristupanje njoj dodatno naplaćuje kao pristup memoriji, stoga je uvedena manja tablica TLB koja sprema informaciju zadnje

[24]Slika 4.4 preuzeta iz [16] (str.7)



Slika 4.4: Osnovna implementacija GEBP-a

korištenih stranica. Ukoliko je stranica pronađena prijenos podataka je brz, ukoliko nije, onda se tablica straničenja koristi. Ukratko, TLB je priručna memorija za tablicu straničenja.

Najznačajnija razlika između promašaja priručne memorije i TLB promašaja[25] je ta što se raznim algoritmima za pred-dobavljanje može zamaskirati promašaj priručne memorije i CPU se neće zadržavati dok se željeni podatak ne pronađe. S druge strane kod TLB promašaja nije tako, već se CPU treba zadržati sve dok se TLB ne ažurira s novom adresom. Postojanje TLB-a znači i uvođenje dodatnih pretpostavki [16]:

d) Dimenzije m_c i k_c su dovoljno malene tako da A, n_r stupci od B (B_j) i n_r stupac od C (C_j) su istovremeno adresabilni od strane TLB-a tako da se tijekom računanja $C_j := AB_j + C_j$ ne dogodi niti jedan TLB promašaj.

e) Ako je A adresiran od strane TLB-a, ostaje tako dokle god A ne postane nepotreban.

Nadalje, uz cijenu smještanja podataka u priručnu memoriju, njenu veličinu i TLB tablicu, *pakiranje* je idući problem. Fundamentalni problem je taj što je tipično varijabla A zapravo podmatrica neke veće matrice i tako nije kontinuirana u memoriji. Problem je taj što je u tom slučaju potrebno više od minimalnog broja TLB unosa. Rješenje je da se A zapakira u kontinuirani radni niz $\tilde{A}$. Parametri m_c i k_c su birani tako da $\tilde{A}$, B i C stanu u L2 priručnu memoriju i da ih TLB može adresirati.

Analiza i implementacija GEPB i GEPDOT se može izvesti na sličan način kao i za GEBP.

Implementacije GEMP, GEPM i GEPP se mogu izvesti pomoću GEPB, GEBP i GEPDOT imajući na umu prethodne optimizacije nižih slojeva rutina i kernela te daljnjim

[25]Promašaj priručne memorije i TLB promašaj – nije pronađen željeni podatak na željenom mjestu kako bi se obavila operacija (kopiranje u registre, izračun na CPU)

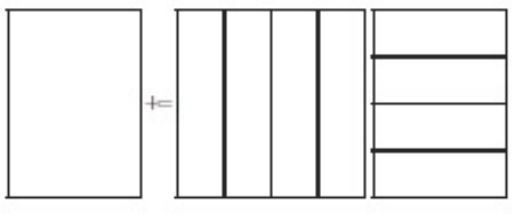
pažljivim pakiranjem particija A i B u kontinuirane radne nizove tako da i kapacitet TLB tablice i radne memoriju budu zadovoljeni.

4.2.1 Ostvarivanje visokih performansi BLAS rutina razine 3 (Level-3 BLAS)

Kao što je navedeno u poglavlju 4.1.3. BLAS rutine razine 3 su rutine koje vrše operacije matrica nad matricama. Implementacija visoko-performansnog množenja matrica s matricama (GEMM) se može iskoristiti za implementaciju drugih operacija matrica nad matricama koje su dio treće razine BLAS-a. Spomenute operacije, odnosno rutine, su SYMM, SYRK, SYR2K, TRMM i TRSM.

Ako uzmemo u obzir račun $C := AB + C$ gdje su C, A, i B redom $m \times n$, $m \times k$ i $k \times n$ matrice, te pretpostavimo da je $m = b_m M$, $n = b_n N$ i $k = b_k K$ gdje su M,N,K,b_m,b_n i b_k cijeli brojevi. Podijeljeno na način (kao na slici 4.5[26]):

$$A \rightarrow \left(\begin{array}{c|c|c|c} A_0 & A_1 & \ldots & A_{K-1} \end{array} \right) \text{ i } B \rightarrow \left(\begin{array}{c} \check{B}_0 \\ \hline \check{B}_1 \\ \hline \check{B}_{k-1} \end{array} \right),$$



Slika 4.5: Podjela A i B

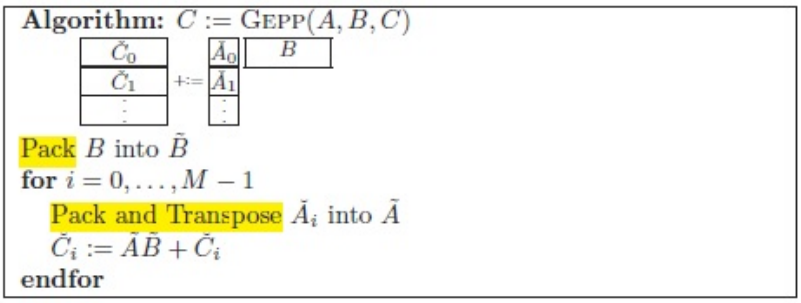
gdje A_p i $\check{B}_p$ sadrže redom b_k stupaca i redaka[27]. Tada je $C := A_0\check{B}_0 + A_1\check{B}_1 + \cdots + A_{K-1}\check{B}_{K-1} + C$.

Tipična visoko-performansna implementacija GEMM će se fokusirati na to da se ažuriranje $C := A_p\check{B}_p + C$, što ćemo zvati umnožak ploča-ploča[28] (GEPP), učini što brže je moguće. Ukupan učinak GEMM-a je u suštini jednak svakom zasebnom GEPP-u sa širinom ploče jednakoj optimalnoj veličini b_k.

Slika 4.6[29] nam prikazuje visoko-performansni algoritam za GEPP operaciju, $C := AB + C$, gdje je „*k*“ dimenzija b_k. Ovaj algoritam zahtjeva tri vrlo optimizirane komponente:

- *Pakiraj B*: Rutina pakiranja B u kontinuirani spremnik. Na određenim arhitekturama ta rutina može reorganizirati podatke za specijalizirane instrukcije korištene od strane GEBP niže opisane kernel rutine.

[26] Slika 4.5 preuzeta iz [17] (str. 2)
[27] Znak ˇ označava za naše potrebe podjelu po redcima
[28] Autori [17] navode umnožak ploča-ploča (eng. panel-panel multiplication)
[29] Slika 4.6 preuzeta iz [17] (str.2.)



Slika 4.6: Optimizirana implementacija GEPP (individualna operacija umnoška ploča-ploča $C := A_j B_j + C$)

- *Pakiraj i transponiraj* $\check{A}_i$: Rutina pakiranja $\check{A}_i$ u kontinuirani spremnik. Ova rutina često transportira matricu u svrhu poboljšanja redoslijeda pristupanja GEBP kernel rutine.

- *GEBP kernel rutina*: Ova rutina računa $\check{C}_i := \check{A}\check{B} + \check{C}_i$ koristeći pakirane spremnike. GEBP označava (eng. General block-times-panel multiply) generalni umnožak blok-puta-ploča.

Na sadašnjim arhitekturama veličina $\check{A}_i$ je određena tako da napuni polovicu L2 priručne memorije (odnosno memorija koju TLB može adresirati) [17].

Na slici 4.7[30] u grafovima su prikazane performanse i brzina (odaziv) raznih kernela korištenih za implementaciju visoko-performansnog DGEMM-a (GEMM dvostruke preciznosti) opisanog u [16] i prethodnom potpoglavlju. Klasični pristupi implementaciji treće razine BLAS-a su implementacija pomoću iterativnog pristupa (petlje) i pomoću rekurzivnog pristupa. No, alternativni pristup implementaciji je bolji i već u sebi ima implementirana rješenja za određene probleme koji se mogu pojaviti.

Na primjeru SYMM rutine alternativni pristup je primjena istog algoritma koji se koristio za GEPP kako bi se promijenila rutina koja kopira podmatrice matrice A, u obliku tako da odgovara simetričnoj prirodi matrice A. Cijela ideja iza toga je izbjegavanje nepotrebnog ponovnog kopiranja podataka u pakirane formate. Također, rutine, kao što su rutine koje služe za pakiranje u kontinuirane nizove, i kernel rutine se tretiraju kao *građevni blokovi* za biblioteke.

Performanse svih rutina treće razine BLAS-a i njihova usporedba se mogu vidjeti na slici 4.8[31]

Tipična LU faktorizacija (s ili bez pivotiranja) vrši TRSM operaciju s matricom koja kasnije postaje operand u rutini GEPP. To omogućuje da se izbjegne ponovno pakiranje podataka, ako bi se već spremio pakirani niz koji je korišten u operaciji TRSM. Prednosti takvog pristupa su ilustrirani na slici 4.9[32]. Krivulja nazvana „LAPACK“

[30] Slika 4.7 preuzeta iz [17] (str. 3)
[31] Slika 4.8 preuzeta iz [17] (str. 2)
[32] Slika 4.9 preuzeta iz [17] (str. 16)

odgovara LAPACK implementaciji LU faktorizacije s parcijalnim pivotiranjem [17] .

Na slici 4.9 krivulja nazvana "*Fused dtrsm/dgemm*"[33] spaja DTRSM i DGEMM pozive tako da se matrica "B" može iskoristiti pri čemu se veličine blokova drže stalnima. Na arhitekturi koja je korištena za implementaciju algoritama i testiranje[34], poboljšanja performansi su značajna.

Potpuno optimizirana implementacija LU faktorizacije s djelomičnim pivotiranjem, koja optimizira zamjenu više redova (DLASWP), dodaje rekurziju „faktorizaciji trenutne ploče" i povećava veličinu blokova na 192 na Pentium4 arhitekturi. Ta implementacija je označena „*Fully optimized*"[35] na slici 4.9 [17].

LAPACK rutina (Slika 4.10), koju smo koristili, poziva i DLASWP, te kao nova inačica opisivane rutine vjerojatno ima bolje performanse od testirane u radu [17].

Implementacija BLAS rutina treće razine na SMP[36] ili višejezgrenim platformama može lako naići na suvišne operacije pakiranja na drugim, različitim dretvama. Izlaganjem građevnih blokova rutina među dretvama sustava takvi problemi bi se mogli izbjeći, pritom značajno poboljšavajući performanse rada algoritma odnosno računalnog programa [17].

4.3 Implementacija BLAS-a

Za prikaz primjera implementacije BLAS i LAPACK rutina i operacija koristimo ATLAS koji je optimizirao rutine arhitekturi računala[37]. Rutinu koju smo odabrali (`dgetrf`) je ona koju možemo koristiti da bi izračunali determinantu matrice reda *n*. Za primjer je korištena LU dekompozicija matrice sa djelomičnim zamjenama stupaca (*pivoting*). Metoda je već opisana ranije u radu, a krajnji rezultat koji se dobije, među ostalim, su matrice U i L, gornje trokutasta matrica i jedinična donje trokutasta matrica. Uz pravila koja vrijede za determinante, vrijedi i da je determinanta jedinične donje trokutaste matrice L jednaka 1, "a matrice U je umnožak elemenata po dijagonalama:" [14]

$$det = \prod_{j=1}^{N} \beta_{jj}$$

te je umnožak dviju determinanti, također prema pravilu $det(AB) = det(A)det(B)$, rješenje tražene determinante matrice A.

Rutina koja je korištena je `dgetrf` iz LAPACK biblioteke. Rutina `dgetrf` je skup podrutina iz LAPACK i BLAS biblioteka (Slika 4.10[38]) na koje se ta rutina poziva, ali

[33] Hrv. spojene dtrsm/dgemm

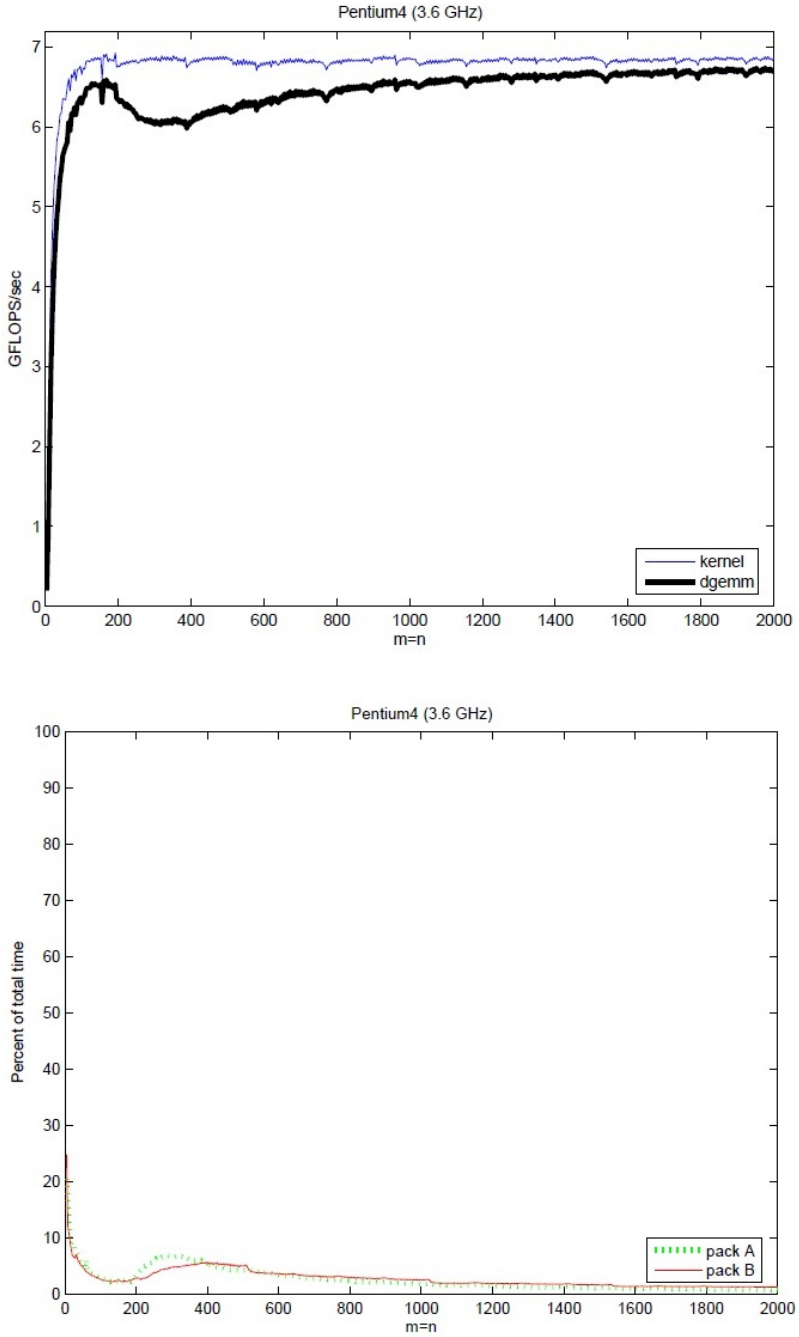
[34] Arhitektura koju smo koristili u ovom radu je vrlo slična arhitekturi korištenoj u izvornom radu [17] (Pentium 4).

[35] Hrv. Potpuno optimizirana

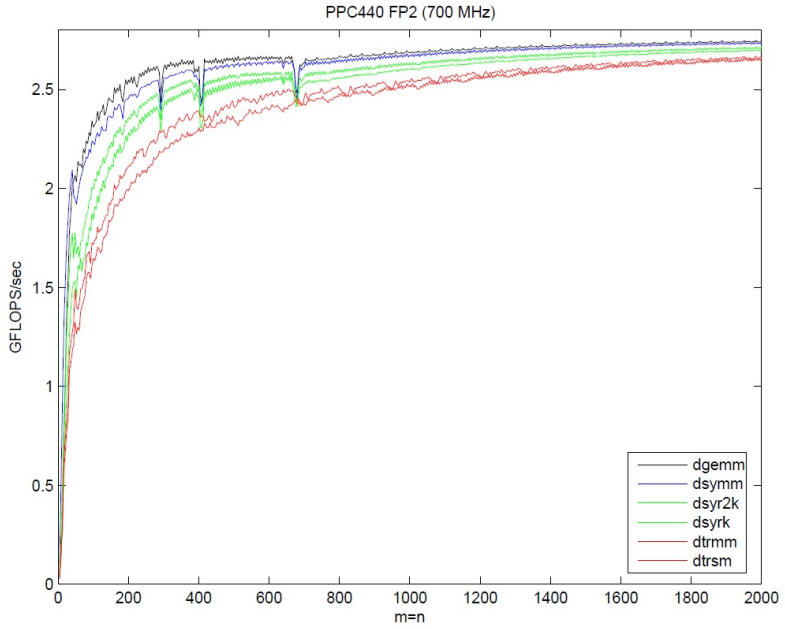
[36] SMP –(eng. Symmetric multiprocessing) Simetričan višeprocesorski sustav

[37] Intel Core 2 Duo CPU E7300 @ 2,66GHz x 2; 4 GB RAM; OS – ubuntu 16.04 64-bit

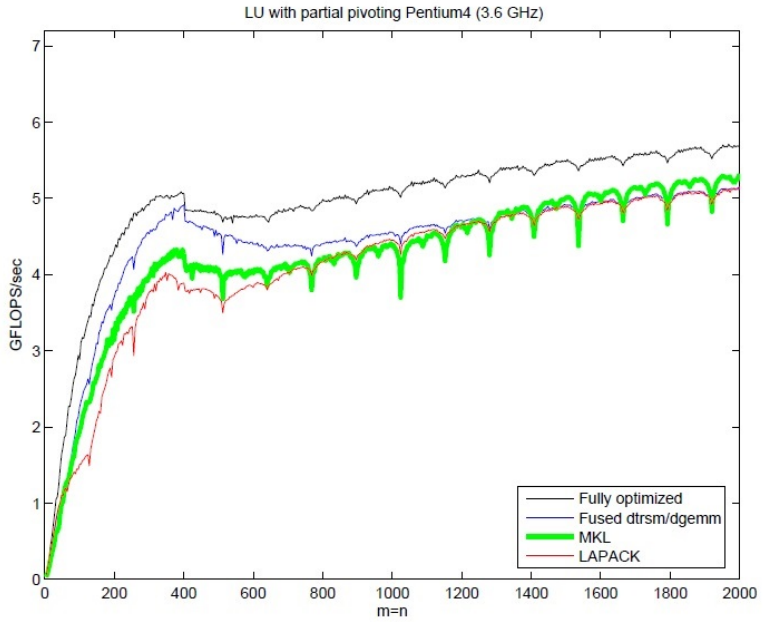
[38] Slika 4.10 preuzeta je sa stranice: http://www.netlib.org/lapack/explore-html/ (03.09.2016.)



Slika 4.7: Performanse GEMM-a. Gornji graf prikazuje performanse GEBP kernel rutine (naznačen „kernel“) i GEMM-a. Donji graf prikazuje postotak ukupnog vremena koji rutina provede u pakiranju



Slika 4.8: Performanse svih BLAS rutina treće razine (level-3 BLAS na IBM PPC440 FP2 (700 MHz) računalu).



Slika 4.9: Performanse LU faktorizacije sa djelomičnim pivotiranjem za razne razine optimizacije

je ujedno i jedna od korištenijih rutina na koju se, također, ostale rutine pozivaju (Slika 4.11[39]).

Opis rutine `dgetrf` [15]:

```
subroutine dgetrf (M,N,A,LDA,IPIV,INFO)
```

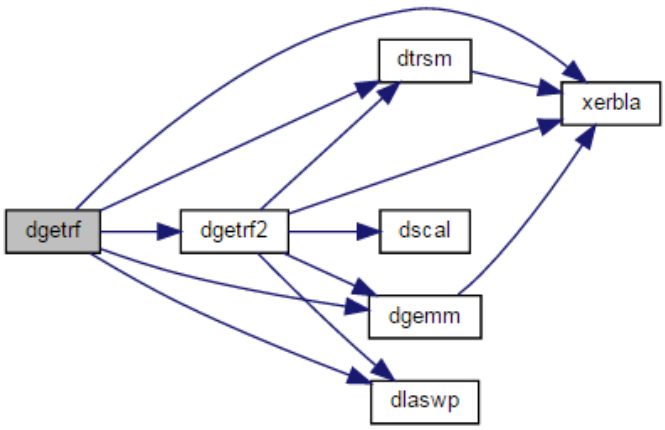
Svrha rutine je izračun LU faktorizacije generalne matrice A dimenzija $M \times N$ koristeći parcijalno pivotiranje sa zamjenama redova.

Faktorizacija ima oblik

$$A = P * L * U$$

gdje je P permutacijska matrica, L je donje torkutasta matrica sa jediničnim elementima na dijagonali (donje trapezoidna ako je $m > n$), a U je gornje trokutasta (gornje trapezoidna ako je $m < n$).

"Ova rutina je prava slika verzije BLAS rutine treće razine." [15]



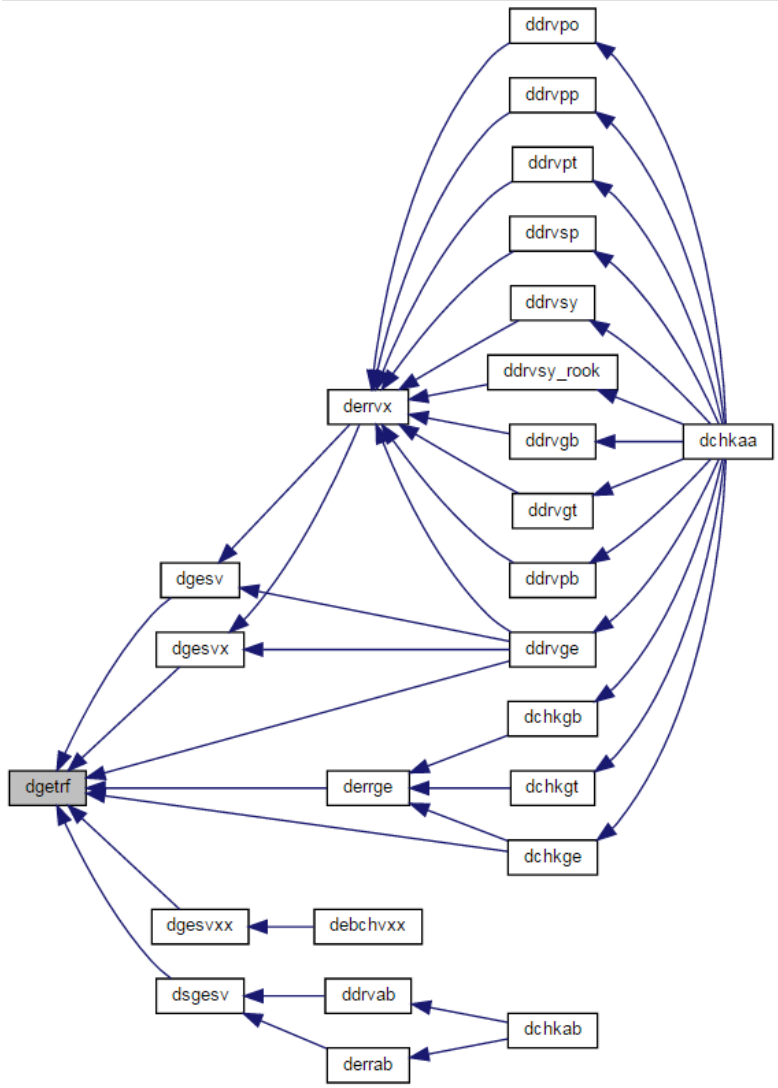
Slika 4.10: Slika prikazuje pozive rutina koje rutina dgetrf koristi da bi se izvršio traženi zadatak

U tablici 4.7 se prikazuje opis parametara:

[39]Slika 4.11 preuzeta je sa stranice: http://www.netlib.org/lapack/explore-html/ (03.09.2016.)

M (ulaz)	M je INTEGER tipa (cjelobrojan) Označava broj redova matrice A. $M \geq 0$
N (ulaz)	N je INTEGER tipa (cjelobrojan) Označava broj stupaca matrice A. $N \geq 0$
A (ulaz,izlaz)	A je niz tipa DOUBLE PRECISION (realan broj dvostruke preciznosti), dimenzija (LDA,N). Na ulaz dolazi matrica A dimenzija $M \times N$ koja se treba faktorizirati. Na izlaz se vraćaju matrice L i U od faktorizacije $A = P * L * U$; jedinični elementi matrice L se ne spremaju.
LDA (ulaz)	LDA je INTEGER tipa (cjelobrojan). Vodeća dimenzija niza A. $LDA \geq max(1, M)$.
IPIV (ulaz)	IPIV je niz INTEGER (cjelobrojan), dimenzija $(min(M, N))$ Indikator zamijene; za $1 \leq i \leq min(M, N)$, red i matrice je zamijenjen sa redom IPIV(i).
INFO (izlaz)	INFO je INTEGER tipa (cjelobrojan) $= 0$: uspješan izlaz < 0 : ako je $INFO = -i$, i-ti argument je poprimio ilegalan iznos. > 0 : ako je $INFO = i$, $U(i, i)$ je jednak 0. Faktorizacija je bila uspješna, ali matrica U je singularna i dogoditi će se dijeljenje sa nulom ako se koristi za rješavanje sustava jednadžbi.

Tablica 4.7: Opis parametara rutine `dgetrf`



Slika 4.11: Slika prikazuje sve rutine koje pozivaju (direktno ili preko poziva drugih rutina, ali su jednako ovisne o njoj) rutinu `dgetrf` kako bi izvršile svoj zadatak

Potpis rutine je jedinstven stoga se poziv sa raznih platformi, operacijskih sustava i programskih jezika vrlo malo razlikuje.

4.3.1 Implementacija BLAS rutine pomoću programskog jezika Fortran

Poziv rutine se obavlja s naredbom "`call`", zatim slijedi naziv rutine. Poziv je identičan kao poziv klasičnih podrutina, samo što je ovo vanjski poziv ("*external*"):

```
external DGETRF
...
call DGETRF(N,N,A,N,IPIV,INFO)
```

Prilikom kompajliranja programa bilo je potrebno uključiti tražene biblioteke u sam proces, pa je proces kompajliranja bio nalik ovome:[40]

gfortran naziv_programa.f90 -llapack

kao dodatak se na kraj može staviti i "*-lblas*" u slučaju da treba rutina iz BLAS biblioteke, no korištenje "*-llapack*" je bilo sasvim dovoljno jer LAPACK uključuje BLAS biblioteku.

Uz rutinu, za izračun LU faktorizacije `dgetrf`, generirali smo matricu čiji su elementi slučajni brojevi:

```
call random_number(b)
```

izračunati determinantu:

```
if (INFO.ne.0) then
        write(*,'(a,i4)') 'degtrf (INFO) vraca gresku',INFO
        stop
endif
        DET = 1d0
        do i=1,n
                if (IPIV(i).ne.i) then
                        DET = -DET * A(i,i)
                else
                        DET = DET * A(i,i)
                endif

        enddo
```

izmjeriti vrijeme izvršavanja izračuna LU faktorizacije i determinante:

[40] Kasnije u izračunima determinante matrica velikih dimenzija bilo je potrebno dodati razne zastavice, te je naredba za komplajliranje bila: *gfortran -g -fcheck=all -Wall -mcmodel=large naziv_programa.f90 (-llapack za program koji koristi BLAS rutinu)*

```
call system_clock ( t1, clock_rate, clock_max )
...
blok koda
...
call system_clock ( t2, clock_rate, clock_max )
```

te, izračun i ispis željenog vremena (izraženo u sekundama):

```
write (*,*) 'Vrijeme izvrsavanja = ',real(t2 - t1)/real(clock_rate)
```

Kod izračuna determinante `if` naredba, `if (INFO.ne.0)`, nam određuje uvjet kada će se računati determinanta. INFO služi kao zastavica uspješno ili neuspješno završene faktorizacije. Linija „`if (IPIV(i).ne.i)`“ nam označava kada je došlo do zamjene reda, odnosno kada treba mijenjati predznak determinante. Razlog tome je što je IPIV (kao što je navedeno u tablici 4.7) niz koji sprema svaku zamjenu redova na svakoj iteraciji, te u slučaju da `IPIV(i) != i` onda treba zamijeniti predznak determinante.

Kao dodatnu inspiraciju prilikom kodiranja možemo navesti službene *netlib* stranice [15], NAG-ove službene stranice[41], Intelove službene stranice i forum¸[42], stranice sveučilišta *Old Dominion University* (stranice profesora Alexandera L Godunova)[43] i *Numerical recipes in fortran 77* [14].

4.3.2 Implementacija BLAS rutine pomoću programskog jezika C

Poziv *BLAS/LAPACK* rutine se ostvaruje isto kao i poziv "običnih" rutina, operacija ili metoda u programskom jeziku C uz dodatak "podcrte" na kraju naziva. Kako bi se omogućio poziv vanjske rutine, potrebno je uključiti biblioteku što je moguće ostvariti na dva načina: pomoću `#include <lapacke.h>` kako bi se mogle koristiti rutine kompajlirane za programski jezik C, odnosno prilikom kompajliranja u linux operativnom sustavu potrebno je koristiti zastavice kao što su *–llapack* ili *–lblas* kako bi GNU kompajler mogao uključiti biblioteke u proces kompajliranja. Dodatan korak koji se još može napraviti jest deklarirati našu rutinu u programu, ali taj korak nije nužan jer je rutina vanjska, odnosno deklarirana i opisana izvan našeg programa. Način implementacije ovisi o platformi na kojoj se željena biblioteka želi implementirati:

[41]NAG (eng. Numerical Algorithms Group) - http://www.nag.co.uk/content/who-we-are-and-what-we-do (1.9.20116.)

[42]INTEL - https://software.intel.com/en-us/forums/intel-math-kernel-library/topic/309460 (1.9.2016)

[43]ODU – A.Godunov (računska fizika)- http://ww2.odu.edu/ agodunov/computing/programs/ (1.9.2016.)

```
#include <lapacke.h>
...
dgetrf_(&N, &N, A[0], &N, pivotArray, &errorHandler);
```

odnosno:

```
...
#include <stdlib.h>

extern void dgetrf_(int *m,int *n,double *A,int *LDA,int *IPIV,int *INFO);
...
```

Da bi se dobila LU faktorizacija pomoću rutine `dgetrf`, trebalo je ispuniti matricu željene dimenzije s nasumičnim brojevima, što je ostvareno jednostavnim pozivom funkcije:

```
double random_br(int min, int max) {
        double x = min + ((double)rand()/(double)(RAND_MAX)) * max;
    return x;
}
```

Za proizvoljno veliku matricu, odnosno proizvoljnu dimenziju, implementacija je trebala biti nešto drugačija od običnog deklariranja dvodimenzionalnog niza:

```
A=(double **)malloc(N*sizeof(double));
for(i=0;i<N;i++)
        A[i]=(double *)malloc(N*sizeof(double));
```

Pomoću ovakvog načina rezerviranja memorije, može se odrediti proizvoljan, gotovo neograničen, broj članova polja niza, a samim time i proizvoljna dimenzija matrice jer se posebno rezervira memorija za svaka ćeliju (for petlja).

Izračun determinante je ostvaren na sljedeći način:

```
if (errorHandler!=0) {
                fprintf(stderr, "Greška!\n");
                exit(1);
        } else {
                det=1.0;
                for(i=1; i<N; i++){
                        if (pivotArray[i]!=i) {
                                det *= - A[i][i];
                        } else {
                                det *= A[i][i];
                        }
                }
        }
```

Gdje prvi `if` blok služi kao kontrola je li zastavica, koju naša rutina daje, zadovoljavajuća ili nije, odnosno je li matricu A moguće faktorizirati li ne. Nadalje, za svaki član „pivotne matrice“ provjerava se je li član jednak broju člana, te ako nije, determinanta se množi s negativom vrijednošću člana dijagonale faktorizirane matrice.

Za dobivanje vremena izvođenja operacije korištena je `clock()` funkcionalnost jezika C. Funkcija kao rezultat vraća procesorsko vrijeme[44] proteklo od početka izvođenja programa, a ako vrijeme nije dostupno, funkcija kao rezultat vraća -1. Rezultat je cjelobrojnog tipa `clock_t` koji je pomoću `typedef` deklariran u zaglavlju `ctime`. Želimo li to vrijeme pretvoriti u sekunde, treba ga podijeliti sa `CLOCKS_PER_SEC` [45] [18].

Da bi dobili željeno vrijeme izvođenja određenog dijela koda (jedne ili više linija koda, metoda i drugo), a ne izvođenja cijelog programa, koristimo `start_t` i `end_t` varijable cjelobrojnog tipa `clock_t` kako bi označili trenutke koje trebamo zapamtiti. Pozivom `clock()` funkcije dobivamo trenutan broj otkucaja sistemskog sata od početka programa te spremamo taj broj u naše varijable. Na kraju, radimo razliku dva zabilježena trenutka:

```
start_t = clock();
...
blok koda
...
end_t = clock();
total_t = (double)(end_t - start_t) / CLOCKS_PER_SEC;
```

[44] Procesorsko vrijeme (eng. CPU time) – označava clock procesora, odnosno broj otkucaja sistemskog sata. To je mjera za frekvenciju procesora, odnosno u SI sustavu se mjeri u Hz, a nama znači koliko će operacija procesor obraditi u sekundi

[45] standardno makro ime definirano u zaglavlju `ctime` - ono definira broj otkucaja sistemskog sata u sekundi za računalo na kojem radimo

Kao inspiracija za izradu koda korištene su službene *netlib* stranice [15], na kojima se nalaze sve bitne informacije za korištenje metoda ili operacija, zatim stranice praktikuma na matematičkom odsjeku PMF-a[12] te službene stranice IBM-a[46] i njihova dokumentacija.

4.4 Ubrzanje rada programa pomoću BLAS-a

Usporedba se vrši nad dvije različite implementacije, odnosno dva različita algoritma, kojima je traženi rezultat isti napisanih u dva različita programska jezika.

U tablici 4.8 se mogu vidjeti i usporediti vremena potrebna za izvršavanje algoritama napisanih u jeziku *Fortran 90* sa i bez korištenja *LAPACK* rutine.

N	$t_1[s]$	$t_2[s]$
500	0.243000001	0.045000002
1000	2.756999972	0.331000000
1500	13.069000282	0.851000011
2500	65.065002457	4.547999861

Tablica 4.8: Vremena izvršavanja programa napisanog u Fortranu. $t_1[s]$ predstavlja rezultate programa napisanog bez korištenja vanjskih biblioteka, dok $t_2[s]$ predstavlja rezultate prgrama napisanog korištenjem LAPACK rutine `dgetrf`.

U tablici 4.9 se mogu vidjeti vremena potrebna za izvršavanje algoritama napisanih u *C* jeziku i poboljšanje, u smislu smanjenja vremena potrebnim za izračun determinante n-dimenzionalne matrice, između „klasičnog“ algoritma i pristupa rješavanju determinante i korištenja *LAPACK* rutine.

N	$t_1[s]$	$t_2[s]$
500	0.250038	0.038734
1000	3.476156	0.241579
1500	13.858723	0.810967
2500	93.336057	3.679643

Tablica 4.9: Vremena izvršavanja programa napisanog u C-u. $t_1[s]$ predstavlja rezultate programa napisanog bez korištenja vanjskih biblioteka, dok $t_2[s]$ predstavlja rezultate prgrama napisanog korištenjem LAPACK rutine `dgetrf`.

Vidljivo je značajno ubrzanje računanja determinante koristeći *blas/lapack* rutine usporedno s algoritmima bez navedenih biblioteka. Primjeri matrica koje su se koristile za izračune s elementima slučajno generiranih brojeva se nalaze u dodatku.

[46] IBM Knowledge Center - https://www.ibm.com/support/knowledgecenter/SSLTBW_2.1.0/com.ibm.zos.v2r1.cbcpx01/atlasexample1.htm (01.09.2016.)

Dimenzija matrice u danom primjeru je N=10 radi lakšeg prikaza i služi isključivo kao primjer da se pokažu elementi kao svojevrstan isječak velike matrice korištene u izračunima koji su mjereni i čiji su rezultati prikazani. Prikazani rezultati vremena izvršavanja su reducirani za one dimenzije matrica za koje program nije mogao prikazati vrijednost (dobiveni rezultat je bio prikazan kao beskonačnost). Tablice s rezultatima za svaki program također su u cijelosti prikazane u dodatku, uključujući i izostavljene iznose.

5 Jezici više razine u nastavi

U nastavi informatike, pogotovo srednjih škola gimnazijskih i nekih stručnih usmjerenja, u višim razredima se može i treba uvesti programiranje u višim programskim jezicima. Programiranjem se učenika uči algoritamskom razmišljanju i novim tehnikama pristupu rješavanja problemskih zadataka, te je odabir jezika neovisan o cilju koji se želi postići. Iako se Fortran i C smatraju staromodnim i osnovnim jezicima, oni su i dalje prisutni kao neizostavni programski jezici u znanosti i raznim granama industrije. Njihova brzina izvršavanja koda i specijaliziranih programa za izračune je i dalje pri vrhu u usporedbi s mnogim novim jezicima. Sintaksa je sličnija pseudokodu nego apstraktnim mnemonicima, klasama i funkcijama. Takva sintaksa je intuitivna i pruža učenicima da dođu do vlastitog rješenja napisanog u pseudokodu, i time do samog koda. Takav, gotovo nesmetan, prelazak iz pisanja algoritama u pseudokodu u pisanje koda učenicima pruža čvrste temelje algoritamskog razmišljanja i zapisivanja algoritama koje mogu odmah testrati.

5.1 Primjer algoritma za množenje matrica

Algoritmi za množenje matrica mogu poslužiti kao dobar primjer prilikom obrade petlji i rekurzija u redovnoj nastavi informatike.

Prvi primjer je izvedba algoritma za množenje matrica pomoću petlji napisanom u programskom jeziku C.

```
void pomnoziMatricePetlja(int a[MAX][MAX],int b[MAX][MAX]) {
        int suma,i=0,j=0,k=0;
        while(i<m) {
                j=0;
                while(j<p) {
                        k=0;
                        suma=0;
                        while(k<n) {
                                suma=suma+a[i][k]*b[k][j];
                                k++;
                        }
                        c[i][j]=suma;
                        j++;
                }
                i++;
        }
}
```

Drugi primjer je izvedba algoritma pomoću rekurzije.

```
void pomnoziMatriceRekurzija(int a[MAX][MAX],int b[MAX][MAX]) {
        static int sum,i=0,j=0,k=0;
        if(i<m) {
                if(j<p) {
                        if(k<n) {
                                sum=sum+a[i][k]*b[k][j];
                                k++;
                                pomnoziMatriceRekurzija(a,b);
                        }
                c[i][j]=sum;
                sum=0;
                k=0;
                j++;
                pomnoziMatriceRekurzija(a,b);
                }
        j=0;
        i++;
        pomnoziMatriceRekurzija(a,b);
        }
}
```

Također, pomoću dva dana primjera se može usporediti brzina izvođenja dvaju algoritama za dane ulazne parametre. Na dodatnoj nastavi informatike se može objasniti rekurzivna izvedba algoritma i pokazati kako korake izvođenja takvog algoritma nije jednostavno pratiti, te da eventualna optimizacija takvog algoritma zahtjeva veliki napor i ne daje nužno optimalno rješenje. Izvedba istog algoritma u Fortranu nema značajnih razlika, ali vrijedi spomenuti kako postoji već implementirana funkcija `matmul(a,b)` za množenje matrica.

Kao nastavak na implementiranu funkciju za množenje matrica, iz prethodnog primjera, može se napraviti kratki uvod u BLAS, također na dodatnoj nastavi informatike, i predstaviti dodatno rješenje za ubrzanje izvođenja algoritama pomoću implementiranih rutina na primjeru rutine za množenje matrica `DGEMM` gdje prvi znak `D` označava da se radi o rutini koja koristi realne brojeve dvostruke preciznosti (5.1).

Programski jezik	Poziv podrutine
Fortran	`CALL DGEMM (transa, transb, l, n, m, alpha, a, lda, b, ldb, beta, c, ldc)`
C i C++	`dgemm (transa, transb, l, n, m, alpha, a, lda, b, ldb, beta, c, ldc)`

Tablica 5.1: Lista nekih BLAS rutina treće razine

Redom, varijabla `transa` je ulazna varijabla tipa *char* i označava koja forma matrice će se koristiti. Vrijednosti koje može poprimiti su *N*, *T* i *C* gdje *N* označava da se koristi matrica A, *T* označava da se koristi A^T i *C* da se koristi A^H. Varijabla `transb` je ista kao `transa` i govori koja forma matrice B će se koristiti. Našem primjeru koji možemo iskoristiti u nastavi odgovara korištenje vrijednosti *N* za obje varijable. Varijabla `l` označava broj redaka, a `n` broj stupaca matrice C. U našem slučaju odabirom vrijednosti *N* za `transa` i `transb`, `m` označava broj stupaca matrice A i broj redaka matrice B. `alpha` je skalar. Varijabla `a` je matrica A sa `l` redaka i `m` stupaca. `lda` odgovara vodećoj dimenziji od A i u našem primjeru vrijedi $lda \geq l$. `b` i `ldb` također određuju matricu B i njenu pripadajuću vodeću dimenziju gdje je $ldb \geq m$ dok `c` i `ldc` određuju matricu C i njenu vodeću dimenziju gdje je $ldc \geq l$. Izlazna varijabla je `c` u koju se sprema rezultat i odgovara $l \times n$ matrici C. S ovako postavljenim varijablama naša rutina odgovara kombinaciji množenja i zbrajanja matrica:

$$C \leftarrow \alpha AB + \beta C$$

Kada bi postavili $\beta = 0$ dobili bi:

$$C \leftarrow \alpha AB$$

što je naš željeni cilj.

Primjer izračuna umnoška matrica A i B koristeći BLAS podrutinu `DGEMM`[47]:

```
CALL DGEMM('N','N',M,N,K,ALPHA,A,M,B,K,BETA,C,M)
```

gdje su $M = 5$, $K = 5$, $N = 5$, $ALPHA = 1.0$, $BETA = 0.0$.
Matrica A:

$$A = \begin{pmatrix} 5 & 3 & 5 & 4 & 2 \\ 2 & 0 & 0 & 2 & 2 \\ 1 & 0 & 5 & 2 & 2 \\ 3 & 0 & 3 & 3 & 1 \\ 5 & 2 & 1 & 5 & 3 \end{pmatrix}.$$

Matrica B:

$$B = \begin{pmatrix} 4 & 2 & 1 & 0 & 4 \\ 3 & 4 & 5 & 3 & 0 \\ 3 & 5 & 5 & 1 & 3 \\ 3 & 3 & 5 & 5 & 3 \\ 4 & 2 & 5 & 0 & 4 \end{pmatrix}.$$

[47] Cijeli kod je sadržan u dodatku

Rezultat[48]:

$$C = \begin{pmatrix} 64 & 63 & 75 & 34 & 55 \\ 22 & 14 & 22 & 10 & 22 \\ 33 & 37 & 46 & 15 & 33 \\ 34 & 32 & 38 & 18 & 34 \\ 56 & 44 & 60 & 32 & 50 \end{pmatrix}.$$

Na isti način bi se mogle iskoristiti i rutine za operacije nad vektorima kao što su `sscal` - *umnožak vektora i skalara*, `sdot` - vektorski umnožak, `saxpy` - umnožak vektora i skalara i zbroj vektora i dr.

BLAS rutine prve razine mogli bismo iskoristiti kao pokazni primjer za množenje vektora i skalara ili zbrajanju vektora kod uvođenja vektorskog računa u nastavi fizike u prvom razredu srednjih škola. U sklopu praktične nastave fizike, BLAS bi mogao poslužiti također kao pokazni primjer za izračune rezultata eksperimentalnih mjeranja znanstvenika i njihovu obradu. Time bi se učenicima pružio uvid u stvarne potrebe znanstvenika za računalima i njihovu isprepletenost na najosnovijoj razini. Također, kada bi se na taj način pristupilo rješavanju nekog problema, i kada bi se učenike upoznalo sa svim koracima potrebnim da se dođe do željenog cilja (od postavljanja problema, eksperimentalnog dijela mjerenja i obrade rezultata te primjenom računalnih programa i biblioteka za daljnju obradu tih rezultata, primjenu kompleksnijih matematičkih i fizikalnih modela i prezentaciju rezultata), razvila bi se dodatna motivacija učenika i interes za određenim znanstvenim područjima. Učenike bi se dodatno poticalo na algoritamsko razmišljanje i zaključivanje čime bi se, uz eksperimentalni pristup nastave fizike, moglo dodatno motivirati učenike te cjeloživotnim učenjem poboljšati opće rezultate, kako u školi tako i izvan nje.

[48] Matrica C je postavljenja kao nulmatrica za ulaznu varijablu.

6 Zaključak

Numerički izračuni su neophodni u akademskom svijetu, raznim granama industrije i školstvu. Implementacija i optimizacija algoritama ključan je segment razvojnog ciklusa računalnog programa. Glavni zadatak ovog rada bio je razmotriti praktičnu primjenu BLAS-a te implemenacijom algoritma za računanje determinante n-dimentionalne matrice pokazati poboljšanje u performansama rada algoritma s BLAS-om, naspram algoritma bez BLAS-a, te pokazati pojednostavljenje pisanja kompleksnijih algoritama i računalnih programa korištenjem rutina LAPACK-a (čijim je dijelom, među ostalim implementiranim rutinama, i BLAS biblioteka rutina). Algoritmi su napisani na dva načina (koristeći rutine BLAS-a i "klasičan" način bez pomoći vanjskih biblioteka) i pomoću dva programska jezika (Fortran i C) kako bismo pokazali da gotovo ne postoji razlika u implemenaciji BLAS-a u oba programska jezika.

S tehničke strane treba istaknuti da su algoritmi, i u jednoj i u drugoj izvedbi (Fortran i C), prilikom izvođenja koristili samo jednu jezgru CPU-a. Razlog tome je što nisu korištene dodatne mogućnosti paralelizacije, niti su korištene vanjske biblioteke, niti programski dodaci da se takvo što omogući. Također, prevodioci koji su korišteni na linux operativnom sustavu, samostalno nemaju omogućenu distribuciju dretvi na višejezgrene sustave. Ukidajući navedenu prepreku, performanse izvršavanja algoritma se mogu poboljšati i samim time se može iskoristiti potpuni potencijal hardware-a koji smo imali na raspolaganju. Radi lakše reprodukcije rezultata i vjerodostojnijeg prikaza istih navedena se postavka nije mijenjala.

Također, izvršeni su izračuni determinanti za matrice većih dimenzija, od u radu prikazanih rezultata (slika za N=10000 i tablice rezultata dostupni u dodatku), u želji da se istraži kolika je maksimalna dimenzija koja je u okvirima prihvatljivih granica „čekanja" rezultata. Implementacijom BLAS biblioteka izvršavanje kompleksnijih izračuna više nije bilo ograničeno samo na vektorska super-računala, već se omogućilo izvršavanje na osobnim računalima te se, smanjenjem vremenske ovisnosti i složenosti o veličini ulaza, omogućilo pomicanje granica maksimalne veličine ulaza, a da pritom vrijeme izračuna ostane u prihvatljivom okviru. Daljnjim napretkom tehnologije BLAS će u sklopu programskih rješenja, među kojima je i LAPACK, još više doći do izražaja u poboljšanju performansi algoritama, a samim time i računalnih programa.

Vrlo značajna karakteristika BLAS-a je ta što nam je omogućeno da pametno i efikasno iskoristimo dostupne resurse računala, s gotovo maksimalnom optimizacijom algoritama koji su nam potrebni, dok s druge strane korištene rutine se mogu iznova iskoristiti. Time do izražaja dolazi BLAS-ova bitna karakteristika: *modularnost*. Proces pojednostavljenja programiranja, te izrade i primjene algoritma, je ostvaren na način da su građevni blokovi svakog algoritma sastavni dio biblioteke kao već definirane i, u smislu arhitekture, jako dobro isplanirane bazične rutine. U vidu standardizacije bazičnih rutina, BLAS omogućava jedinstven i unificiran pristup rješavanju

problema i značajno poboljšanje performansi algoritama i računanja kompleksnih računa.

Bitno je istaknuti kako primjena BLAS-a ne mora biti samo u znanstvene svrhe i poboljšanje efikasnosti performansi algoritama i računanja, već bi se njegova iskoristivost mogla implementirati u nastavne planove srednjoškolskog obrazovanja što bi mlade dodatno motiviralo za daljnja znanstvena istraživanja.

Dodaci

Dodatak A Kod - Fortran

A.1 LU dekompozicija i izračun determinante

```
!determinanta_LU.f90
PROGRAM determinanta
    implicit none
    integer, parameter ::  n=1500
    integer i,j,indx(n)
    real*16 DET
    real a(n,n)
    integer :: t1, t2, clock_rate, clock_max

    !pozivi

    !RNG
    call random_seed

    call random_number(a)

    call system_clock ( t1, clock_rate, clock_max )
    ! LU dekompozicija
    call ludcmp(a,n,n,indx,DET)

    do j=1,n
        DET=DET*a(j,j)
    enddo

    call system_clock ( t2, clock_rate, clock_max )

    write(*,*)'d: '
    write(*,*) DET
    write(*,*)
    write(*,*)'Realno vrijeme izvrsavanja=',real(t2 - t1)/real(clock_rate)

    END
```

```
subroutine ludcmp(a,n,np,indx,d)
    integer n,np,indx(n),NMAX
    real TINY,a(np,np)
    parameter (NMAX=5000,TINY=1.0e-20)
    integer i,imax,j,k
    double precision aamax,dum,sum,vv(NMAX)
    real*16 d

    d=1.

    do i=1,n
        aamax=0.

        do j=1,n
            if (abs(a(i,j)).gt.aamax) aamax=abs(a(i,j))
        enddo

        if (aamax.eq.0.) read(*,*) !pause
        vv(i)=1./aamax
    enddo

    do j=1,n
        do i=1,j-1
            sum=a(i,j)

            do k=1,i-1
                sum=sum-a(i,k)*a(k,j)
            enddo

            a(i,j)=sum
        enddo

        aamax=0.

        do i=j,n
            sum=a(i,j)

            do k=1,j-1
                sum=sum-a(i,k)*a(k,j)
            enddo
```

```
            a(i,j)=sum
            dum=vv(i)*abs(sum)
            if (dum.ge.aamax) then
                imax=i
                aamax=dum
            endif
        enddo

        if (j.ne.imax)then

            do k=1,n
                dum=a(imax,k)
                a(imax,k)=a(j,k)
                a(j,k)=dum
            enddo

            d=-d
            vv(imax)=vv(j)

        endif
        indx(j)=imax
        if(a(j,j).eq.0.)a(j,j)=TINY

        if(j.ne.n)then
            dum=1./a(j,j)

            do i=j+1,n
                a(i,j)=a(i,j)*dum
            enddo

        endif

    enddo

    return

    end
```

A.2 Izračun determinante pomoću lapack rutine

```
!determinanta_LU_BLAS.f90
```

```
PROGRAM determinanata_blas
    implicit none
    integer i, IFAIL, INFO, j
    integer :: t1, t2, clock_rate, clock_max, g
    integer, parameter :: N=1500 ! Odabir dimenzija matrice
    double precision, allocatable :: A(:,:)
    real*16 DET
    integer IPIV(N)

    external DGETRF
    allocate( A(N, N) )

    !pozivi

    !RNG
    do  i=1,N
        do  j=1,N
            call random_number(b)
            A(i,j) = b
        end do
    end do

    call system_clock ( t1, clock_rate, clock_max )

    call DGETRF(N,N,A,N,IPIV,INFO)

    DET = 0d0
    if (INFO.ne.0) then
        write(*,'(a,i4)') 'degtrf (INFO) vraca gresku',INFO
        stop !zaustavi izvodenje programa
    endif

    do i=1,N
        if (IPIV(i).ne.i) then
            DET = -DET * A(i,i)
        else
            DET = DET * A(i,i)
        endif

    enddo
```

```
    call system_clock ( t2, clock_rate, clock_max )

    write(*,*)'d: '
    write(*,*) DET
    write(*,*)
    write(*,*)'Vrijeme izvrsavanja=',real(t2-t1)/real(clock_rate),'[s]'

    END
```

Dodatak B Kod - C

B.1 LU dekompozicija i izračun determinante

```
//determinanta_C_LU.c
#include<stdio.h>
#include<stdlib.h>
#include<time.h>


void crout(double **A,double **L, double **U, int n);

double random_br(int min, int max);

int main(){
clock_t start_t, end_t;
double total_t;
int n,i,j;
double **A;
double **L;
double **U;

printf("dimenzija matrice: ");
scanf("%d", &n);

A=(double **)malloc(n*sizeof(double));
for(i=0;i<n;i++)
    A[i]=(double *)malloc(n*sizeof(double));

U=(double **)malloc(n*sizeof(double));
for(i=0;i<n;i++)
    U[i]=(double *)malloc(n*sizeof(double));
```

```
L=(double **)malloc(n*sizeof(double));
for(i=0;i<n;i++)
    L[i]=(double *)malloc(n*sizeof(double));

srand(time(NULL));

for(i = 0; i < n; i++) {
    for(j = 0; j < n; j++){
        A[i][j] = random_br(1,9);
    }
}

start_t = clock();

crout(A,L,U,n);

long double dijagL = 1;
long double dijagU = 1;

for (i = 1; i < n; i++){
    dijagL = dijagL*L[i][i];
    dijagU = dijagU*U[i][i];
}

long double det = dijagL*dijagU;

end_t = clock();

total_t = (double)(end_t - start_t) / CLOCKS_PER_SEC;

printf("\ndeterminanta matrice A: %Le\n", det);
printf("\nvrijeme potrebno za izvršavanje programa: %f sekundi\n", total_t);

return 0;
}

double random_br(int min, int max) {
    double x = min + ((double)rand()/(double)(RAND_MAX)) * max;
    return x;
}
```

```
void crout(double **A,double **L, double **U, int n) {
    int i, j, k;
    double sum = 0;

    for (i = 0; i < n; i++) {
         U[i][i] = 1;
    }

    for (j = 0; j < n; j++) {
        for (i = j; i < n; i++) {
            sum = 0;
            for (k = 0; k < j; k++) {
                sum = sum + L[i][k] * U[k][j];
            }
            L[i][j] = A[i][j] - sum;
        }

        for (i = j; i < n; i++) {
            sum = 0;
            for(k = 0; k < j; k++) {
                sum = sum + L[j][k] * U[k][i];
            }
            if (L[j][j] == 0) {
                printf("det(L) je blizu 0! Ne može se dijeliti sa 0...\n");
                exit(EXIT_FAILURE);
            }
            U[j][i] = (A[j][i] - sum) / L[j][j];
        }
    }
}
```

B.2 Izračun determinante pomoću lapack rutina

```
//determinanta_C_LU_BLAS.c
#include <stdio.h>
#include <stddef.h>
#include <time.h>
#include <stdlib.h>

extern void dgetrf_ (int *m, int *n, double *A,
```

```
                    int *LDA, int *IPIV, int *INFO);
double random_br(int min, int max);

int main() {
    int N = 1500;
    clock_t start_t, end_t;
    double total_t;
    double **A;
    long double det=1.0;
    int pivotArray[N];
    int i,j,errorHandler;

    A = (double **)malloc(N*sizeof(double));
    for(i=0;i<N;i++)
        A[i] = (double *)malloc(N*sizeof(double));

    srand(time(NULL));

    for(i = 0; i < N; i++) {
        for(j = 0; j < N; j++) {
            A[i][j] = random_br(1,9);
        }
    }

    start_t = clock();

    //pozivanje rutine
    dgetrf_ (&N, &N, A[0], &N, pivotArray, &errorHandler);
    //printf ("dgetrf errorHandler zastavica: %d\n", errorHandler);

    if (errorHandler != 0) {
        fprintf(stderr, "Greška!\n");
        exit(1);
    } else {
        for(i=0; i < N; i++) {
            if (pivotArray[i] != i) {
                det *= -A[i][i];
            } else {
                det *= A[i][i];
            }
        }
```

```
    }

    end_t = clock();
    total_t = (double)(end_t - start_t) / CLOCKS_PER_SEC;

    printf("\nDeterminanata matrice A dimenzije N=%d iznosi: %Le",N, det);
    printf("\nvrijeme potrebno za izvrsavanje programa: %f [s]\n", total_t);

    printf("\n");

    return 0;
}

double random_br(int min, int max) {
    double x = min + ((double)rand()/(double)(RAND_MAX)) * max;
    return x;
}
```

Dodatak C Elementi matrice generirane u Fortranu pomoću fortranovog generatora pseudoslučajnih brojeva (N=10)

```
0.99756 0.56682 0.96592 0.74793 0.36739 0.48064 0.07375 0.05355 0.34708 0.34224
0.21795 0.13316 0.90052 0.38677 0.44548 0.66193 0.01610 0.65085 0.64641 0.32299
0.85569 0.40129 0.20687 0.96854 0.59840 0.67298 0.45688 0.33002 0.10038 0.75545
0.60569 0.71905 0.89733 0.65823 0.15072 0.61231 0.97866 0.99914 0.25680 0.55087
0.65905 0.55401 0.97776 0.90192 0.65792 0.72886 0.40246 0.92863 0.14784 0.67453
0.76961 0.33932 0.11582 0.61437 0.82062 0.94709 0.73113 0.49760 0.37480 0.42151
0.55290 0.99792 0.99039 0.74631 0.95376 0.09327 0.73402 0.75176 0.94685 0.70618
0.81381 0.55859 0.06170 0.48038 0.59769 0.13753 0.58740 0.51997 0.88588 0.30381
0.66966 0.66494 0.50368 0.26158 0.07656 0.10125 0.54927 0.37558 0.01515 0.79292
0.62088 0.77360 0.95358 0.11424 0.31846 0.59682 0.04815 0.11421 0.21596 0.10057
```

Dodatak D Rezultati izvođenja algoritama

D.1 Fortran

N	$t[s]$	det
500	0.243000001	1.962283142543441753230932125096OE+0297
1000	2.756999972	3.265133832068641450344642557127 1E+0744
1500	13.069000282	-2.362087484036562297081191827476E+1248
2500	65.065002457	-4.893203567307399485909512837018E+2357
5000	621.193970	Infinity
7500	2430.44702	Infinity
10000	6010.15186	Infinity
15000	23363.0117	-Infinity
20000	72409.7969	Infinity

Tablica D.1: Rezultati algoritma za izračun determinante pomoću LU dekompozicije (FORTRAN)

D.2 Fortran i BLAS/LAPACK

N	$t[s]$	det
500	4.50000018E-02	1.87495558020102167104526280699 9E-4648
1000	0.331000000	3.119849614314809858117083789434E-4201
1500	0.851000011	-2.257618936074936758627770762 07E-3697
2500	4.54799986	-4.67694823628573833639245651695 5E-2588
5000	46.6189995	2.947712225314185731666144926557 5E+0519
7500	135.938004	1.664000441413048632424242940522 2E+3912
10000	324.092010	Infinity
15000	1252.94495	-Infinity
20000	2872.40894	Infinity

Tablica D.2: Rezultati algoritma za izračun determinante pomoću BLAS/LAPACK rutina (FORTRAN)

D.3 C

N	$t[s]$	det
500	0.250038	3.140760e+773
1000	3.476156	-7.730900e+1698
1500	13.858723	-1.553594e+2680
2500	93.336057	-4.011613e+4743
5000	941.161511	inf
7500	2995.149077	inf
10000	8480.862033	-inf
15000	39512.223772	inf
20000	129861.111253	inf

Tablica D.3: Rezultati algoritma za izračun determinante pomoću LU dekompozicije (C)

D.4 C i BLAS

Znak * označava rezultat koji nedostaje radi tehničkih poteškoća[49].

N	$t[s]$	det
500	0.038734	-1.710289e+38
1000	0.241579	-2.598385e+91
1500	0.810967	-3.515266e+268
2500	3.679643	2.701345e+390
5000	32.984538	2.771135e+1082
7500	144.460411	8.768586e+1982
10000	366.892277	-1.636846e+2871
15000	1095.939918	-4.013275e+4689
20000	*	*

Tablica D.4: Rezultati algoritma za izračun determinante pomoću BLAS/LAPACK rutina (C)

[49]Program pisan u programskom jeziku C se, uz sve uložene napore da se to ne događa, zaustavljao usred izvođenja računa uz poruku *segmentation fault* što je poruka koja označava da je problem u nedostatku računalne memorije ili da su nizovi korišteni u izračunima nekonzistentni u memoriji. Takve prepreke su nam se nekoliko puta javile te smo ih uspješno uklonili u svim slučajevima osim u slučaju velikih dimenzija matrice (N=20000).

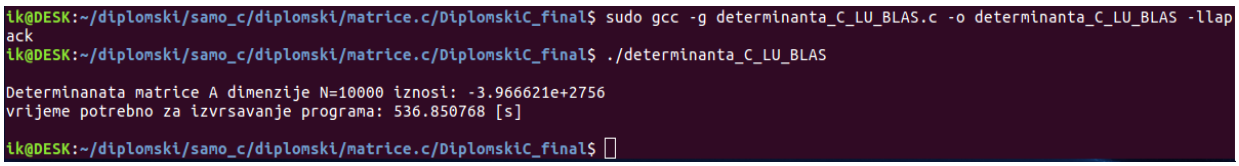
Dodatak E Množenje matrice (DGEMM)

```
PROGRAM   MAIN

IMPLICIT NONE

DOUBLE PRECISION ALPHA, BETA
INTEGER          M, K, N, I, J
PARAMETER        (M=5, K=5, N=5)
DOUBLE PRECISION A(M,K), B(K,N), C(M,N), br, intbr

PRINT *, "This example computes real matrix C=alpha*A*B+beta*C"
PRINT *, "C=A*B"
PRINT 10, " matrix A(",M," x",K, ") and matrix B(", K," x", N, ")"
10 FORMAT(a,I5,a,I5,a,I5,a,I5,a)
PRINT *, ""
ALPHA = 1.0
BETA = 0.0

do  i=1,M
 do  j=1,K
  call random_number(br)
  intbr = floor( br*6 )
  A(i,j) = intbr
 end do
end do

do  i=1,K
 do  j=1,N
  call random_number(br)
  intbr = floor( br*6 )
  B(i,j) = intbr
 end do
end do

DO I = 1, M
 DO J = 1, N
  C(I,J) = 0.0
 END DO
END DO
```

```
PRINT *, "Podrutina DGEMM "
CALL DGEMM('N','N',M,N,K,ALPHA,A,M,B,K,BETA,C,M)
PRINT *, ""

PRINT *, "Ispis matrice A "
do, i=1,N
 write(*,"(100g15.5)") ( a(i,j), j=1,N )
end do

PRINT *, "Ispis matrice B "
do, i=1,N
 write(*,"(100g15.5)") ( b(i,j), j=1,N )
end do

PRINT *, "Ispis matrice C "
do, i=1,N
 write(*,"(100g15.5)") ( c(i,j), j=1,N )
end do

STOP

END
```

Dodatak F Slika

F.1 Izračun determinate za matricu N = 10000



Slika F.1: Izračun determinante matrice N = 10000. Program napisan u programskom jeziku C koristeći LAPACK rutine

Literatura

[1] Thomas H. Cormen; Charles E. Leiserson; Ronald L. Rivest, Introduction to Algorithms. USA: MIT, 2000.

[2] Robert Sedgewick, Algorithms. 2nd ed. USA Princeton University: Addison-Wesley, 1989.

[3] Zdravko Dovedan; Mirko Smilevski; Janez Divjak Zalokar, Fortran 77 i programiranje. Treće izdanje, Zagreb, DON, 1991.

[4] M. Jurak, Programski jezik C: predavanja (ak. g. 2003/2004), `https://web.math.pmf.unizg.hr/~singer/Prog_Add/c.pdf`, 3.9.2016.

[5] Brian W. Kernighan; Dennis M. Ritchie, The C Programming Language. 2nd ed., New Jersey, AT&T Bell Laboratories, 1988.

[6] Gfortran – the GNU Fortran compiler, partof GCC, (1.8.2016.), GCC GNU, `https://gcc.gnu.org/wiki/GFortran`, 10.8.2016.

[7] TimothyGowers, ThePrincetonCompanion to Mathematics, New Jersey, Princeton University Press, 2008.

[8] Damira Keček, Osječki matematički list: Metode izračunavanja determinanti n-tog reda // Studentska rubrika, Veleučilište u Varaždinu, broj 10 (2010), str. 31-42, `http://hrcak.srce.hr/file/89410`, 3.9.2016.

[9] LAPACK - Linear Algebra PACKage: programski paket algebarskih rutina; Sveučilište Tennesse; Sveučilište Kalifornija, Berkley; Sveučilište Kolorado Denver; NAG Ltd.; `http://www.netlib.org/lapack/`, 3.9.2016.

[10] Zlatko Drmač; Vjeran Hari; Miljenko Marušić; Malden Rogina; Sanja Singer; Saša Singer, Numerička analiza, Zagreb, Sveučilište u Zagrebu, PMF - matematički odjel, 2003.

[11] Robert van de Gejin, Kazushige Goto, BLAS (Basic Linear Algebra Subprograms), 157.-164. Encyclopedia of Parallel Computing, Springer, 2011.

[12] Praktikum primjenjene matematike I - PMF - matematički odjel, (ak.god. 2006/07), `https://web.math.pmf.unizg.hr/nastava/ppm1/`, 3.9.2016.

[13] ATLAS - Automatically Tuned Linear Algebra Software, `http://math-atlas.sourceforge.net/`, 3.9.2016.

[14] William H. Press; Saul A. Teukolsky; William T. Vetterling; Brian P. Flannery, Numerical Recipes in Fortran 77: The Art of Scientific Computing (Vol 1 of Fortran Numeric Recipes), Cambrige MA, Cambridge University Press, 1997

[15] LAPACK - Linear Algebra PACKage (3.6.1), (19.6.2016), http://www.netlib.org/lapack/explore-html/index.html, 3.9.2016.

[16] Kazushige Goto; Robert A. van de Geijn, Anatomy of High-Performance Matrix Multiplication, ACM Transactions on Mathematical Software, 2008.

[17] Kazushige Goto; Robert A. van de Geijn, High-Performance Implementation of the Level-3 BLAS, 2008.

[18] Julijan Šribar; Boris Motik, Demistificirani C++ (drugo izdanje), Element, Zagreb, 2001.

[19] Recommendation of the European Parliament and of the Council of 18 December 2006 on key competences for lifelong learning (2006/962/EC), Official Journal of the European Union L 394, 30.12.2006, str. 10.–18., http://eur-lex.europa.eu/eli/reco/2006/962/oj, 01.02.2017.

[20] Jerbić - Zorc, G., Metodika nastave informatike/tehnike akademska godina 2015/2016., http://www.phy.pmf.unizg.hr/~gorjana/nastava/Informatika/nastavni%20materijali/met_inf_novo.pdf, 01.02.2017.

[21] Zakon o Hrvatskom kvalifikacijskom okviru; Ministarstvo znanosti, obrazovanja i sporta, Republika Hrvatska, Zagreb/2013., http://www.kvalifikacije.hr/fgs.axd?id=519 i http://narodne-novine.nn.hr/clanci/sluzbeni/2013_02_22_359.html, 01.02.2017.

[22] Ispitni katalog za državnu maturu u školskoj godini 2016./2017., Informatika, Nacionalni centar za vanjsko vrednovanje obrazovanja, https://www.ncvvo.hr/wp-content/uploads/2016/09/INFORMATIKA-2017.pdf, 01.02.2017.

[23] Andreja Ilić, Analiza algoritma, http://www.pmf.ni.ac.rs/pmf/predmeti/7015/vezbe/slozenost_algoritama.pdf, 01.03.2017.