UNIVERZA V LJUBLJANI
FAKULTETA ZA MATEMATIKO IN FIZIKO

Matematika – 1. stopnja

Matej Marinko

Hitro množenje matrik

Delo diplomskega seminarja

Mentor: doc. dr. Klemen Šivic

Ljubljana, 2020

Kazalo

Hitro množenje matrik

Povzetek

Množenje matrik je v linearni algebri preprosta operacija, ki se pogosto pojavlja v rešitvah najrazličnejših problemov. Prav zato je bilo v iskanje hitrih algoritmov za množenje matrik vloženega že veliko dela. V diplomskem delu definiramo problem iskanja zgornje meje eksponenta matričnega množenja in razvijemo teorijo ranga in mejnega ranga bilinearnih preslikav. Predstavimo več algoritmov za hitro množenje matrik, ki slonijo na tej teoriji. Izbrane algoritme tudi implementiramo, jih primerjamo med seboj in ocenimo njihovo uporabnost v praksi.

Fast Matrix Multiplication

Abstract

Matrix multiplication is one of the most basic operations in linear algebra and thus very common in various scientific disciplines. Consequently, the computation complexity of matrix multiplication has been extensively studied. In this work, we define a problem of finding the upper bound for the exponent of matrix multiplication and present the theory of rank and border rank of bilinear maps. We describe multiple fast matrix multiplication algorithms based on this theory. In the end, we implement some selected algorithms, compare them, and discuss their value in practical applications.

Math. Subj. Class. (2010): 68Q17, 68Q25, 15A69

Ključne besede: množenje matrik, eksponent matričnega množenja, rang tenzorjev, mejni rang

Keywords: matrix multiplication, exponent of matrix multiplication, tensor rank, border rank

1. Uvod

Množenje dveh matrik je ena najpreprostejših operacij v linearni algebri in se pogosto pojavlja v najrazličnejših algoritmih. Formula za produkt dveh matrik je preprosta, pozna jo vsak matematik. Prav zaradi njene preprostosti je bilo veliko presenečenje, ko je Volker Strassen leta 1969 objavil algoritem, s katerim lahko velike matrike množimo hitreje kot z običajno formulo. Kmalu zatem se je raziskovanje tega področja močno razširilo, odkritih je bilo še več drugih algoritmov, ki so asimptotično še hitrejši.

Diplomska naloga je sestavljena iz dveh glavnih delov. V prvem delu si najprej podrobno ogledamo Strassenov algoritem, nato definiramo eksponent matričnega množenja in razvijemo teorijo ranga bilinearne preslikave, na kateri temeljijo vsi nadaljnji algoritmi. Ves čas se vračamo nazaj na Strassenov algoritem, ki nam služi kot zgled za boljše razumevanje definicij. Proučevanje lastnosti ranga nas pripelje do izreka, s pomočjo katerega lahko z lahkoto izračunamo časovno zahtevnost nekaterih algoritmov za matrično množenje. Zatem vpeljemo pojem mejnega ranga bilinearne preslikave in pokažemo, da ima podobne lastnosti kot rang. Tudi v primeru mejnega ranga dokažemo podoben, splošnejši izrek, s pomočjo katerega lahko analiziramo še nekatere druge algoritme. Na koncu prvega dela navedemo še nadaljnje pomembne mejnike v tej teoriji.

V drugem delu se ukvarjamo z implementacijo navedenih algoritmov. Najprej si ogledamo različne načine, kako lahko algoritme, ki delujejo samo na matrikah določene velikosti, razširimo na poljubne velikosti. Implementiranim algoritmom izmerimo čas izvajanja in ga primerjamo s pričakovanimi časovnimi zahtevnostmi algoritmov.

2. Strassenov algoritem

Preden definiramo naš problem, se spomnimo definicije O-notacije, s katero bomo merili število operacij množenja matrik.

Definicija 2.1. Naj bosta $f, g\colon \mathbb{N} \to \mathbb{N}$ funkciji. Potem označimo, da je $f = O(g)$ natanko tedaj, ko obstajata $c > 0$ in $n_0 \in \mathbb{N}$, da za vse $n \geq n_0$ velja $f(n) \leq cg(n)$.

Če je $f(n) = O(g(n))$, smo, ko gre n proti neskončno, navzgor omejili rast funkcije f z rastjo funkcije g. V tem diplomskem delu bo f funkcija, ki bo štela število operacij množenja dveh matrik. Omejevali jo bomo s funkcijami oblike $g(n) = n^\tau$ za neki $\tau > 0$.

Recimo, da za kvadratni $n \times n$ matriki, A in B, nad poljem F želimo izračunati njun produkt $C = AB$. Najpreprosteje je, da vsak element v matriki C določimo po standardni formuli:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}.$$

Za izračun vsakega člena matrike C potrebujemo n množenj in $n-1$ seštevanj. Vseh členov v matriki C je n^2, torej za izračun celotnega produkta dveh $n \times n$ matrik potrebujemo $O(n^3)$ operacij.

Strassen je leta 1969 v [13] pokazal, da lahko izračunamo produkt dveh $n \times n$ matrik z $O(n^{\log_2 7})$ operacijami. Algoritem temelji na spretnem množenju matrik velikosti 2×2.

Naj bosta A in B matriki:

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}, \quad B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}.$$

Najprej izračunamo naslednjih sedem produktov:

$$\begin{aligned} p_1 &= (a_{11} + a_{22})(b_{11} + b_{22}) \\ p_2 &= (a_{21} + a_{22})b_{11} \\ p_3 &= a_{11}(b_{12} - b_{22}) \\ p_4 &= a_{22}(-b_{11} + b_{21}) \\ p_5 &= (a_{11} + a_{12})b_{22} \\ p_6 &= (-a_{11} + a_{21})(b_{11} + b_{12}) \\ p_7 &= (a_{12} - a_{22})(b_{21} + b_{22}) \end{aligned}$$

Sedaj lahko produkt matrik A in B izračunamo na sledeč način:

$$C = AB = \begin{bmatrix} p_1 + p_4 - p_5 + p_7 & p_3 + p_5 \\ p_2 + p_4 & p_1 - p_2 + p_3 + p_6 \end{bmatrix}.$$

Ni težko preveriti, da je dobljen rezultat pravilen, recimo v prvi vrstici in drugem stolpcu:

$$p_3 + p_5 = a_{11}(b_{12} - b_{22}) + (a_{11} + a_{12})b_{22} = a_{11}b_{12} + a_{12}b_{22}.$$

Podobno za ostala tri polja. Če množimo matriki na tak način, potrebujemo za izračun le 7 namesto 8 množenj, vendar pa pridobimo veliko dodatnih seštevanj. Natančneje, opraviti moramo 18 namesto 4 seštevanj. Če privzamemo, da je množenje enako zahtevno kot seštevanje, pri matrikah velikosti 2×2 ne pridobimo ničesar. Ta algoritem ne deluje samo na številih, temveč tudi na matrikah. Elementi matrik A in B so lahko nove matrike, postopek pa lahko rekurzivno ponovimo. Množenje matrik pa ni enako zahtevno kot seštevanje matrik. Za izračun vsote dveh $n \times n$ matrik potrebujemo n^2 operacij, za izračun produkta pa dosti več (če bi množili podmatrike na klasičen način, bi potrebovali $O(n^3)$ operacij). S pomočjo naslednje leme bomo izračunali časovno zahtevnost Strassenovega algoritma.

Lema 2.2. *Produkt dveh matrik velikosti $2^l \times 2^l$ lahko izračunamo s 7^l množenji in $6 \cdot (7^l - 4^l)$ seštevanji (oziroma odštevanji).*

Dokaz. Dokazovali bomo z indukcijo po l.

Ko je $l = 1$, je to ravno primer za 2×2 matrike. Vemo že, da v tem primeru potrebujemo $7 = 7^1$ množenj in $18 = 6 \cdot (7^1 - 4^1)$ seštevanj.

Indukcijski korak: predpostavimo, da lema velja za vsa števila do vključno $l - 1$. Dokazujemo, da velja za l. Obe $2^l \times 2^l$ matriki si predstavljamo kot bločni 2×2 matriki, katerih elementi so $2^{l-1} \times 2^{l-1}$ matrike. Produkt teh dveh matrik lahko izračunamo s 7 množenji in 18 seštevanji njunih elementov. Za seštevanje dveh podmatrik velikosti $2^{l-1} \times 2^{l-1}$ potrebujemo $(2^{l-1})^2$ operacij, za množenje dveh podmatrik pa po indukcijski predpostavki potrebujemo 7^{l-1} množenj in $6 \cdot (7^{l-1} - 4^{l-1})$ seštevanj. Skupaj torej potrebujemo $7 \cdot 7^{l-1} = 7^l$ množenj in

$$7 \cdot 6 \cdot (7^{l-1} - 4^{l-1}) + 18 \cdot (2^{l-1})^2 = 6 \cdot (7^l - 7 \cdot 4^{l-1} + 3 \cdot 4^{l-1}) = 6 \cdot (7^l - 4^l)$$

seštevanj. □

Posledica 2.3. *Produkt dveh $n \times n$ matrik lahko izračunamo z $O(n^{\log_2 7})$ aritmetičnimi operacijami.*

Dokaz. Najprej predpostavimo, da je $n = 2^l$ za neki $l \in \mathbb{N}$. Po lemi potrebujemo za izračun produkta 7^l množenj in $6 \cdot (7^l - 4^l)$ seštevanj, skupaj torej največ

$$7^l + 6 \cdot (7^l - 4^l) = 7 \cdot 7^l - 6 \cdot 4^l \leq 7 \cdot 7^l = 7 \cdot 2^{l \log_2 7} = 7n^{\log_2 7} \text{ operacij.}$$

Naj bo zdaj n poljuben. Matriki A in B lahko razširimo do velikosti naslednje potence števila 2 tako, da jima dodamo ničelne stolpce in vrstice. Na ta način vsaki matriki dodamo manj kot $3n^2$ ničel. Predpostavimo, da nas dodajanje ene ničle "stane" eno aritmetično operacijo. Potem za oba koraka skupaj, dodajanje ničel in računanje produkta, potrebujemo največ $3n^2 + 3n^2 + 7n^{\log_2 7} = O(n^{\log_2 7})$ operacij. □

3. Bilinearne preslikave

Do algoritmov z manjšo časovno zahtevnostjo bomo prišli s pomočjo pojma ranga bilinearne presilkave. Najprej pa bomo ponovili nekaj definicij iz linearne algebre.

Definicija 3.1. Naj bodo U, V in W vektorski prostori nad poljem F. Potem je $\phi : U \times V \to W$ *bilinearna preslikava*, če sta za vse $u_0 \in U$ in vse $v_0 \in V$ preslikavi $v \mapsto \phi(u_0, v)$ in $u \mapsto \phi(u, v_0)$ linearni.

Primer 3.2. Preverimo, da je množenje matrik bilinearna preslikava.

$$\phi : F^{n \times k} \times F^{k \times m} \to F^{n \times m}$$
$$(A, B) \mapsto AB$$

Za prvi faktor velja:

$$\phi(\alpha A, B) = \alpha AB = \alpha \phi(A, B)$$
$$\phi(A_1 + A_2, B) = (A_1 + A_2)B = A_1B + A_2B = \phi(A_1, B) + \phi(A_2, B)$$

Torej je ϕ linearna v prvem faktorju. Podobno preverimo za drugi faktor. Sledi, da je ϕ bilinearna. ♢

Definicija 3.3. Naj bo V vektorski prostor nad poljem F. Linearno preslikavo $f : V \to F$ imenujemo *linearni funkcional*, vektorski prostor vseh linearnih preslikav iz V v F pa imenujemo *dualni prostor* prostora V in ga označimo z V^*.

Definicija 3.4. Naj bo $\phi : U \times V \to W$ bilinearna preslikava nad poljem F. Naj bodo $f_1, \ldots, f_r \in U^*$ in $g_1, \ldots g_r \in V^*$ linearni funkcionali in elementi $w_1, \ldots, w_r \in W$ takšni, da za vse pare $(u, v) \in U \times V$ velja:

$$\phi(u, v) = \sum_{i=1}^{r} f_i(u) g_i(v) w_i.$$

Potem $3r$-terico $(f_1, g_1, w_1; \ldots; f_r, g_r, w_r)$ imenujemo *bilinearni izračun* oziroma *bilinearni algoritem* za ϕ dolžine r.

Bilinearni izračun za poljubno preslikavo obstaja (glej opombo 6.2), ni pa enolično določen. Trivialni primer je, da bilinearnemu izračunu $\sum_{i=1}^{r} f_i(u) g_i(v) w_i$ dodamo še $w_{r+1} = 0$ ter poljubna f_{r+1}, g_{r+1} in tako dobimo nov bilinearni izračun dolžine $r+1$. Zanimala nas bo dolžina najkrajšega možnega bilinearnega izračuna za preslikavo množenja matrik.

Definicija 3.5. Dolžino najkrajšega bilinearnega izračuna za ϕ imenujemo *rang* ϕ. Označimo jo z $R(\phi)$.

Opomba 3.6. Takšna definicija ranga je sorodna definiciji ranga matrike. Vemo, da lahko matriko ranga 1 zapišemo kot produkt stolpca in vrstice. Izkaže se, da lahko to še razširimo, saj lahko rang matrike A alternativno definiramo kot najmanjši tak r, da lahko A zapišemo kot vsoto:

$$A = \sum_{i=1}^{r} v_i u_i^T.$$

Za poljuben vektor w je $u_i^T w$ skalar, zato je preslikava $f_i(w)$, definirana s predpisom $f_i(w) = u_i^T w$, linearni funkcional. Ko matriko A uporabimo na vektorju, dobimo:

$$Aw = \left(\sum_{i=1}^{r} v_i u_i^T \right) w = \sum_{i=1}^{r} v_i (u_i^T w) = \sum_{i=1}^{r} f_i(w) v_i,$$

kar je zelo podobno zgornji definiciji bilinearnega izračuna. Tako kot smo linearni preslikavi priredili matriko, bomo pozneje bilinearni preslikavi priredili tenzor, katerega rang bo enak rangu bilinearne preslikave.

Preslikavo množenja $n \times k$ matrike s $k \times m$ matriko bomo označevali $\langle n, k, m \rangle$. Izkazalo se bo, da lahko rang te preslikave služi kot merilo časovne zahtevnosti algoritma množenja matrik.

Primer 3.7. Strassenov algoritem lahko zapišemo tudi na način, predstavljen v definiciji 3.4. Definiramo:

$$\begin{aligned}
f_1(A) &= a_{11} + a_{22} & g_1(B) &= b_{11} + b_{22} \\
f_2(A) &= a_{21} + a_{22} & g_2(B) &= b_{11} \\
f_3(A) &= a_{11} & g_3(B) &= b_{12} - b_{22} \\
f_4(A) &= a_{22} & g_4(B) &= -b_{11} + b_{21} \\
f_5(A) &= a_{11} + a_{12} & g_5(B) &= b_{22} \\
f_6(A) &= -a_{11} + a_{21} & g_6(B) &= b_{11} + b_{12} \\
f_7(A) &= a_{12} - a_{22} & g_7(B) &= b_{21} + b_{22}
\end{aligned}$$

$$W_1 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \; W_2 = \begin{bmatrix} 0 & 0 \\ 1 & -1 \end{bmatrix}, \; W_3 = \begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix}, \; W_4 = \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix}$$

$$W_5 = \begin{bmatrix} -1 & 1 \\ 0 & 0 \end{bmatrix}, \; W_6 = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}, \; W_7 = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$$

Vidimo, da se produkti $f_i(A)g_i(B)$ ujemajo z definicijami p_i iz poglavja 2, matrike W_i pa so izbrane tako, da je vsota

$$\sum_{i=1}^{7} f_i(A) g_i(B) W_i$$

enaka produktu matrik A in B. Od tod sledi

$$R(\langle 2, 2, 2 \rangle) \leq 7,$$

torej, da je rang množenja 2×2 matrik največ 7. Da se pokazati tudi, da je rang točno 7 [16]. ♢

Definicija 3.8. Naj bo $\phi\colon U \times V \to W$ bilinearna preslikava. Množico vseh elementov iz U, ki jih ϕ slika v 0, ne glede na vrednost drugega argumenta, torej množico $\{u \in U \mid \phi(u,v) = 0\ \forall v \in V\}$, imenujemo *levo jedro preslikave* ϕ.

Podobno definiramo tudi *desno jedro preslikave* ϕ kot množico vseh elementov iz V, ki jih ϕ slika v 0, ne glede na vrednost prvega argumenta, oziroma množico $\{v \in V \mid \phi(u,v) = 0\ \forall u \in U\}$.

Definicija 3.9. Bilinearna preslikava ϕ je *zgoščena*, če je

(1) njeno levo jedro trivialno,
(2) njeno desno jedro trivialno in
(3) linearna ogrinjača $\phi(U,V)$ je enaka celotnemu prostoru W.

Lema 3.10. *Za rang zgoščene bilinearne preslikave* $\phi\colon U \times V \to W$ *velja:*

$$R(\phi) \geq \max\{\dim U, \dim V, \dim W\}.$$

Dokaz. Recimo, da je rang $r = R(\phi)$ strogo manjši od dimenzije U. Po definiciji je

$$\phi(u,v) = \sum_{i=1}^{r} f_i(u) g_i(v) w_i \tag{1}$$

za primerno izbrane f_i, g_i in w_i. Vemo, da je $\dim U = \dim U^*$. Posledično je $r < \dim U^*$, torej funkcionali $f_1, f_2, \ldots, f_r$ gotovo ne tvorijo ogrodja prostora U^*. Množico $\{f_1, f_2, \ldots, f_r\}$ dopolnimo do baze $\mathcal{B}^*$ prostora U^*. Pri tem smo dodali vsaj en funkcional f_0. Obstaja baza $\mathcal{B}$ prostora U, katere dualna baza je $\mathcal{B}^*$. Po definiciji dualne baze za vsak element $u_\kappa \in \mathcal{B}$ obstaja natanko en element $f_\kappa \in \mathcal{B}^*$, tako da $f_\kappa(u_\kappa) \neq 0$. Zato obstaja neničelni vektor $u_0 \in U$, tako da je $f_0(u_0) \neq 0$ in $f_i(u_0) = 0$ za vse $i > 0$. Vektor u_0 vstavimo v enačbo (1) in dobimo:

$$\phi(u_0, v) = \sum_{i=1}^{r} f_i(u_0) g_i(v) w_i = \sum_{i=1}^{r} 0 \cdot g_i(v) w_i = 0.$$

Levo jedro ϕ ni trivialno, zato ϕ ni zgoščena. Z enakimi argumenti pokažemo tudi, da je $R(\phi) \geq \dim V$.

Recimo, da je $r = R(\phi)$ manjši od dimenzije W. Za poljubna $u \in U$ in $v \in V$ je $\phi(u,v)$ linearna kombinacija vektorjev $w_1, w_2, \ldots w_r$. Od tod sledi, da je dimenzija linearne ogrinjače slike ϕ največ $r < \dim W$, torej linearna ogrinjača slike ϕ ni enaka celotnemu prostoru W in zato preslikava ϕ ni zgoščena.

Pokazali smo, da je $R(\phi) \geq \dim U, \dim V, \dim W$, očitno neenakost velja tudi za maksimum. □

Primer 3.11. Preverimo, da je bilinearna preslikava $\langle n,n,n\rangle$, množenje dveh $n \times n$ matrik, zgoščena.

$$\phi\colon F^{n\times n} \times F^{n\times n} \to F^{n\times n}, \quad \phi(A,B) = AB$$

Naj bo $A \in F^{n\times n}$ poljubna matrika. Naj velja $AB = 0$ za vse matrike $B \in F^{n\times n}$. Za B lahko vzamemo $B = I$, in dobimo $AI = A = 0$. S tem smo dokazali prvo točko. Podobno dokažemo točko (2). Pokazati moramo še, da je linearna ogrinjača slike ϕ cel prostor $F^{n\times n}$. Vzemimo matriko $C \in F^{n\times n}$. Očitno velja $\phi(C,I) = CI = C$, torej je C v sliki ϕ. Torej je množenje kvadratnih matrik zgoščena preslikava. Pokazati se da, da je zgoščena tudi preslikava $\langle n,k,m\rangle$ [4, trditev 14.41].

V preslikavi $\langle n,n,n\rangle$ so prostori enaki $U = V = W = F^{n\times n}$. Lema 3.10 nam pove, da je rang preslikave $\langle n,n,n\rangle$ vsaj n^2. ♢

4. Eksponent matričnega množenja

Definicija 4.1. Označimo z $M(n)$ število operacij, ki jih potrebujemo za množenje dveh $n \times n$ matrik. *Eksponent matričnega množenja* $\omega = \omega(F)$ nad poljem F definiramo kot

$$\omega(F) = \inf \{\tau \in \mathbb{R} \mid M(n) = O(n^\tau)\}.$$

Opomba 4.2. Kot število operacij $M(n)$ razumemo najmanjše možno število operacij seštevanja, odštevanja in množenja iz polja F, s katerimi lahko izračunamo produkt dveh $n \times n$ matrik. Operacijo deljenja pri tem izpustimo, v [2, poglavje 4.1] je pokazano, da v primeru, ko je polje F neskončno, to lahko naredimo, ne da spremenimo rezultat. Formalna definicija $M(n)$ je v [4, poglavje 15.1].

Če upoštevamo, kaj pomeni O-notacija, zgornja definicija pove, da lahko za vsak $\tau > \omega$ navzgor omejimo $M(n)$ (ki je funkcija v spremenljivki n) s funkcijo $c \cdot n^\tau$ za neko dovolj veliko konstanto c. Naš cilj je čim bolj omejiti eksponent matričnega množenja. Iz običajne formule za matrično množenje sledi, da je $\omega \leq 3$. Poskušali bomo najti še boljšo zgornjo mejo za ω. Strassenov algoritem pokaže, da je $\omega \leq \log_2 7$.

V zgornji definiciji štejemo operacije, ki jih potrebujemo za množenje dveh $n \times n$ matrik. Definirajmo še $\tilde{\omega}$, ki na podoben način meri rang matričnega množenja:

$$\tilde{\omega}(F) = \inf \{\tau \in \mathbb{R} \mid R(\langle n, n, n \rangle) = O(n^\tau)\}.$$

Naslednja trditev pove, da lahko za zgornjo mejo ω namesto štetja operacij gledamo rang matričnega množenja. Dokaz bomo izpustili, narejen je v [4, poglavje 15.1].

Trditev 4.3. *Za poljubno polje F velja $\omega(F) = \tilde{\omega}(F)$.*

S to trditvijo se problem iskanja čim nižje zgornje meje za eksponent matričnega množenja prevede na iskanje čim nižje meje za rang bilinearne preslikave, s čimer se bomo ukvarjali v naslednjih poglavjih. Poglejmo pa si, kako lahko že zdaj uporabimo trditev, da pokažemo spodnjo mejo za ω.

Primer 4.4. V primeru 3.11 smo ugotovili, da je $R(\langle n, n, n \rangle) \geq n^2$. Torej za vsak $\tau \in \mathbb{R}$, kjer je $R(\langle n, n, n \rangle) = O(n^\tau)$, velja, da je $\tau \geq 2$. Po definiciji $\tilde{\omega}$ je $\tilde{\omega} \geq 2$. Po trditvi 4.3 je tudi $\omega \geq 2$.

Bolj običajen argument v teoriji časovne zahtevnosti bi bil naslednji: Če želimo $n \times n$ matriko pomnožiti z drugo $n \times n$ matriko, moramo gotovo vsaj prebrati njunih n^2 elementov. Zgornji dokaz smo naredili, ker smo se v definiciji $M(n)$ omejili samo na štetje operacij znotraj polja F in zato, da na preprostem primeru vidimo delovanje trditve. ♢

5. Tenzor bilinearne preslikave

Da bomo lahko pokazali določene lastnosti ranga, bomo bilinearnim preslikavam priredili tenzorje, na katerih bomo imeli podobno definicijo ranga.

Naj bodo U, V in W vektorski prostori nad poljem F z izbranimi bazami. Naj bo dimenzija prostora U enaka n, dimenzija prostora V enaka k in dimenzija prostora W enaka m.

Spomnimo se dejstva iz linearne algebre. Naj bo $\varphi\colon U \to W$ linearna preslikava. Za vsak bazni vektor u_i prostora U lahko zapišemo

$$\varphi(u_i) = \sum_{j=1}^{m} a_{ji} w_j,$$

kjer so w_j bazni vektorji prostora W. Koeficiente a_{ji} lahko zberemo v matriko $A \in F^{m\times n}$. Podobno lahko naredimo z bilinearnimi preslikavami. Naj bo $\phi\colon U \times V \to W$ bilinearna preslikava, $\{u_1, \ldots, u_n\}$ baza prostora U, $\{v_1, \ldots, v_k\}$ baza prostora V in $\{w_1, \ldots, w_m\}$ baza prostora W. Potem lahko za vsak par baznih vektorjev (u_i, v_j) zapišemo:

$$\phi(u_i, v_j) = \sum_{l=1}^{m} t_{ijl} w_l.$$

Koeficiente t_{ijl} spet lahko zberemo skupaj. V tem primeru dobimo trirazsežno strukturo t, ki jo bomo imenovali *tenzor, pripadajoč preslikavi* ϕ. Seveda tenzor ni enoličen. Takoj lahko opazimo, da je odvisen od izbire baz v posameznih prostorih.

Tenzor bo za nas vedno tridimenzionalen, predstavljamo si ga lahko kot "škatlo s števili". Ker bomo s tenzorji delali zelo konkretno, nismo navedli formalne definicije. Lahko si predstavljamo, da je tenzor trirazsežna tabela, matrika pa dvorazsežna tabela. To razmišljanje nas privede do rezine tenzorja.

Definicija 5.1. Naj bo $t \in F^{n\times k\times m}$ tenzor. Potem za vsak $l = 1, \ldots, m$ matriko

$$A_l = \begin{bmatrix} t_{11l} & \ldots & t_{1kl} \\ \vdots & & \vdots \\ t_{n1l} & \ldots & t_{nkl} \end{bmatrix} \in F^{n\times k}$$

imenujemo *rezina tenzorja* t. Rezine tenzorja dobimo tudi, če namesto tretjega indeksa fiksiramo prvi ali drugi indeks.

Lahko si predstavljamo, da je trirazsežni tenzor sestavljen iz rezin, ki jih zložimo eno za drugo. Tako kot nismo formalno definirali tenzorjev, tudi ne bomo formalno definirali tenzorskega produkta in se bomo zadovoljili s sledečo definicijo.

Definicija 5.2. Naj bo U vektorski prostor dimenzije n, V prostor dimenzije k in W prostor dimenzije m. *Tenzorski produkt* dveh vektorjev $u = (u_1, \ldots u_n) \in U$ in $v = (v_1, \ldots v_k) \in V$ je matrika

$$u \otimes v = \begin{bmatrix} u_1v_1 & u_1v_2 & \ldots & u_1v_k \\ u_2v_1 & u_2v_2 & \ldots & u_2v_k \\ \vdots & \vdots & & \vdots \\ u_nv_1 & u_nv_2 & \ldots & u_nv_k \end{bmatrix}.$$

Tenzorski produkt treh vektorjev, $u \in U, v \in V$ in $w = (w_1, \ldots w_m) \in W$ označimo $t = u \otimes v \otimes w$ in imenujemo *tenzor ranga* 1, njegove komponente pa so enake

$$t_{ijl} = u_i v_j w_l.$$

Že v opombi 3.6 smo se spomnili, da lahko vsako matriko ranga 1 zapišemo kot produkt stolpca in vrstice. V zgornji definiciji pa lahko opazimo, da je produkt uv^T ravno enak tenzorskemu produktu $u \otimes v$. Torej za matriko A ranga 1 lahko najdemo neka vektorja u in v, da je

$$A = uv^T = u \otimes v.$$

Tenzorji ranga 1 so analogni matrikam ranga 1. Tako kot smo v opombi 3.6 definirali rang matrike, bomo definirali tudi rang tenzorja.

Definicija 5.3. *Rang* tenzorja t je najmanjše tako število r, da lahko t zapišemo kot vsoto r tenzorjev ranga 1, torej da velja:

$$t = \sum_{i=1}^{r} u_i \otimes v_i \otimes w_i$$

za neke vektorje u_i, v_i, w_i. Rang tenzorja t označimo z $R(t) = r$.

Definicija ranga tenzorja je precej podobna definiciji ranga bilinearne preslikave, uporabljamo celo enako oznako. Naslednja trditev pove, da sta oba ranga med seboj povezana.

Trditev 5.4. *Rang tenzorja, ki pripada bilinearni preslikavi, je enak rangu te preslikave.*

Dokaz. Naj bo $\phi\colon U \times V \to W$ bilinearna preslikava ranga r. Naj bo $\{u_1, \ldots, u_n\}$ baza prostora U, $\{v_1, \ldots, v_k\}$ baza prostora V in $\{\tilde{w}_1, \ldots, \tilde{w}_m\}$ baza prostora W. Poljubna vektorja $u \in U$ ter $v \in V$ lahko razvijemo po bazi:

$$u = \alpha_1 u_1 + \cdots + \alpha_n u_n, \quad v = \beta_1 v_1 + \cdots + \beta_k v_k.$$

Elementi tenzorja t, ki pripada ϕ, se po definiciji pojavijo v naslednji vsoti:

$$\phi(u,v) = \phi(\alpha_1 u_1 + \cdots + \alpha_n u_n, \beta_1 v_1 + \cdots + \beta_k v_k) = \sum_{i=1}^{n}\sum_{j=1}^{k}\sum_{l=1}^{m} t_{ijl}\alpha_i\beta_j\tilde{w}_l. \tag{2}$$

Bilinearni izračun za vsako bilinearno preslikavo obstaja, zato lahko ϕ zapišemo kot vsoto

$$\phi(u,v) = \sum_{\kappa=1}^{r} f_\kappa(u) g_\kappa(v) w_\kappa.$$

Izberimo funkcionale $\{\tilde{f}_1, \ldots, \tilde{f}_n\}$ tako, da je vrednost $\tilde{f}_i(u_j) = \delta_{ij}$, kjer je δ_{ij} Kroneckerjeva delta. Iz linearne algebre vemo, da je ta množica dualna baza prostora U^*. To pomeni, da za poljubni vektor $u \in U$, ki ga, kot zgoraj, razvijemo po bazi prostora U, velja:

$$\tilde{f}_i(u) = \tilde{f}_i(\alpha_1 u_1 + \cdots + \alpha_n u_n) = \alpha_i.$$

Na enak način pridemo do baze $\{\tilde{g}_1, \ldots, \tilde{g}_k\}$ prostora V^*. Funkcionale f_κ razvijemo po bazi U^*, g_κ po bazi V^* in vektorje w_κ po bazi W. Potem lahko ϕ zapišemo v naslednji obliki

$$\phi(u,v) = \sum_{\kappa=1}^{r}\left(\sum_{i=1}^{n}\sum_{j=1}^{k}\sum_{l=1}^{m} \tilde{t}_{\kappa ijl}\tilde{f}_i(u)\tilde{g}_j(v)\tilde{w}_l\right), \tag{3}$$

kjer so $\tilde{t}_{\kappa ijl}$ koeficienti, dobljeni pri razvoju f_κ, g_κ in w_κ po bazah prostorov U^*, V^* in W. Vsote še zamenjamo in vsoto členov $\tilde{t}_{\kappa ijl}$ po κ poimenujemo $\tilde{t}_{ijl}$.

$$\phi(u,v) = \sum_{i=1}^{n}\sum_{j=1}^{k}\sum_{l=1}^{m}\left(\sum_{\kappa=1}^{r}\tilde{t}_{\kappa ijl}\right)\tilde{f}_i(u)\tilde{g}_j(v)\tilde{w}_l = \sum_{i=1}^{n}\sum_{j=1}^{k}\sum_{l=1}^{m}\tilde{t}_{ijl}\tilde{f}_i(u)\tilde{g}_j(v)\tilde{w}_l.$$

Koeficiente zberemo v tenzor $\tilde{t}$ in želimo pokazati, da je ta enak tenzorju t. Vektorja u in v lahko, kot zgoraj, razvijemo po ustreznih bazah in vstavimo v enačbo.

$$\phi(u,v) = \phi(\alpha_1 u_1 + \cdots + \alpha_n u_n, \beta_1 v_1 + \cdots + \beta_m v_n) = \sum_{i=1}^{n}\sum_{j=1}^{k}\sum_{l=1}^{m} \tilde{t}_{ijl}\alpha_i\beta_j\tilde{w}_l. \tag{4}$$

Enačbi (2) in (4) sta enaki ter veljata za poljubna vektorja $u \in U$ in $v \in V$. Namesto u in v v enačbi vstavimo bazna vektorja u_i in v_j.

$$\phi(u_i, v_j) = \sum_{l=1}^{m} t_{ijl}\tilde{w}_l = \sum_{l=1}^{m} \tilde{t}_{ijl}\tilde{w}_l.$$

Ker so vektorji $\tilde{w}_l$ med seboj neodvisni, velja $\tilde{t}_{ijl} = t_{ijl}$. Dokazujemo, da je $R(t) = R(\phi)$. S tem, ko smo pokazali, da je $t = \tilde{t}$, smo ugotovili, da lahko to pokažemo v dualnih prostorih U^* in V^*. Na funkcije f_κ in g_κ lahko gledamo kot vektorje, razvite po bazah $\{\tilde{f}_1, \ldots, \tilde{f}_n\}$ in $\{\tilde{g}_1, \ldots, \tilde{g}_k\}$, w_κ pa razvijemo po bazi $\{\tilde{w}_1, \ldots, \tilde{w}_m\}$. Po definiciji tenzorskega produkta so komponente produkta $f_\kappa \otimes g_\kappa \otimes w_\kappa$ enake

$$(f_\kappa \otimes g_\kappa \otimes w_\kappa)_{ijl} = \tilde{t}_{\kappa ijl},$$

zato ta tenzorski produkt ustreza bilinearni preslikavi

$$(u,v) \mapsto \sum_{\kappa=1}^{r}\sum_{i=1}^{n}\sum_{j=1}^{k}\sum_{l=1}^{m} \tilde{t}_{\kappa ijl}\tilde{f}_i(u)\tilde{g}_j(v)\tilde{w}_l.$$

Ta izraz opazimo v enačbi (3), iz katere sledi, da je v tej bazi $t = \tilde{t} = \sum_{\kappa=1}^{r} f_\kappa \otimes g_\kappa \otimes w_\kappa$. S tem, ko smo tenzor t napisali z vsoto r tenzorjev ranga 1, smo pokazali, da je $R(t) \leq r$.

Neenakost v drugo smer pokažemo tako, da zgornji dokaz "preberemo nazaj". Spet izberemo baze prostorov U, V in W. Kot prej ustrezno izberemo bazi dualnih prostorov U^* in V^*. Recimo, da lahko tenzor t, ki pripada preslikavi ϕ, zapišemo kot vsoto $R(t)$ tenzorjev ranga 1, torej $t = \sum_{\lambda=1}^{R(t)} u_\lambda \otimes v_\lambda \otimes w_\lambda$. V prostorih U^*, V^* in W izberemo ustrezne baze in definiramo tenzor $\tilde{t}$ tako, da je $t = \tilde{t}$ in $\tilde{t} = \sum_{\lambda=1}^{R(t)} f_\lambda \otimes g_\lambda \otimes w_\lambda$. Sedaj lahko preslikavo ϕ zapišemo kot vsoto $\phi(u,v) = \sum_{\lambda=1}^{R(t)} f_\lambda(u) g_\lambda(v) w_\lambda$, zato je $R(\phi) \leq R(t)$. Sledi, da je $R(t) = R(\phi)$. □

Posledica 5.5. *Rang tenzorja ni odvisen od izbire baze prostorov U, V in W.*

Dokaz. Naj bo ϕ poljubna bilinearna preslikava. Naj bosta t in t' tenzorja, ki pripadata ϕ pri dveh različnih izborih baz prostorov U, V in W. Trditev 5.4 pove, da je $R(t) = R(\phi)$ in $R(t') = R(\phi)$, zato je $R(t) = R(t')$. □

Opomba 5.6. Rang matrike lahko izračunamo razmeroma preprosto. Lahko bi pričakovali, da bomo sedaj le še izračunali rang tenzorja in tako dobili rang bilinearne preslikave. Izkaže se, da je določanje ranga tenzorja NP težak problem, zato bomo rang še vedno lahko le omejevali navzgor.

Primer 5.7. Poglejmo si, kakšen je tenzor množenja 2×2 matrik. Naj bo ϕ bilinearna preslikava, ki zmnoži 2 matriki.

$$\phi\colon F^{2\times 2} \times F^{2\times 2} \to F^{2\times 2}$$

V vseh prostorih $F^{2\times 2}$ izberemo standardno bazo $\{E_{11}, E_{12}, E_{21}, E_{22}\}$, kjer je E_{ij} matrika, ki ima povsod ničle, le na mestu (i,j) enico. Poglejmo, kako se med seboj

množijo nekateri izmed parov baznih elementov:

$$\begin{aligned}
&\phi(E_{11},E_{11}) = E_{11}E_{11} = E_{11} && \phi(E_{11},E_{12}) = E_{11}E_{12} = E_{12}\\
&\phi(E_{11},E_{21}) = E_{11}E_{21} = 0 && \phi(E_{11},E_{22}) = E_{11}E_{22} = 0\\
&\phi(E_{12},E_{11}) = E_{12}E_{11} = 0 && \phi(E_{12},E_{12}) = E_{12}E_{12} = 0\\
&\phi(E_{12},E_{21}) = E_{12}E_{21} = E_{11} && \phi(E_{12},E_{22}) = E_{12}E_{22} = E_{12}\\
&\dots && \dots
\end{aligned}$$

Tenzor, prirejen ϕ, bo velikosti $4 \times 4 \times 4$, saj so vsi prostori $F^{2\times 2}$ 4-dimenzionalni. Vidimo tudi, da so vsi koeficienti enaki bodisi 1 bodisi 0. Produkt dveh matrik $A = \left[\begin{smallmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{smallmatrix}\right]$ in $B = \left[\begin{smallmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{smallmatrix}\right]$ sedaj lahko zapišemo kot naslednjo vsoto:

$$AB = \sum_{i_1,j_1=1}^{2} \sum_{i_2,j_2=1}^{2} \sum_{i_3,j_3=1}^{2} t_{i_1j_1,i_2j_2,i_3j_3} a_{i_1j_1} b_{i_2j_2} E_{i_3j_3}$$

Spodaj je zapisan tenzor t, vsaka rezina posebej. Iz tenzorja lahko preberemo npr. koeficient pred $a_{11}b_{11}E_{11}$, ki je enak 1.

(5)

$$E_{11}: \begin{array}{c} \\ b_{11}\\ b_{12}\\ b_{21}\\ b_{22} \end{array} \begin{array}{c} \begin{array}{cccc} a_{11} & a_{12} & a_{21} & a_{22} \end{array} \\ \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \end{array}, \quad E_{12}: \begin{array}{c} \\ b_{11}\\ b_{12}\\ b_{21}\\ b_{22} \end{array} \begin{array}{c} \begin{array}{cccc} a_{11} & a_{12} & a_{21} & a_{22} \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \end{array}$$

$$E_{21}: \begin{array}{c} \\ b_{11}\\ b_{12}\\ b_{21}\\ b_{22} \end{array} \begin{array}{c} \begin{array}{cccc} a_{11} & a_{12} & a_{21} & a_{22} \end{array} \\ \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \end{array}, \quad E_{22}: \begin{array}{c} \\ b_{11}\\ b_{12}\\ b_{21}\\ b_{22} \end{array} \begin{array}{c} \begin{array}{cccc} a_{11} & a_{12} & a_{21} & a_{22} \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \end{array}$$

Vemo že, da je $R(\phi) \leq 7$. Trditev 5.4 pove, da je rang tenzorja t enak rangu preslikave ϕ, torej je tudi $R(t) \leq 7$. Poleg tega dokaz te trditve da tudi postopek, kako dobimo tenzorje ranga 1. V primeru 3.7 smo že definirali funkcije $f_1, \dots, f_7$ in $g_1, \dots, g_7$ in matrike W_i. Matrike W_i vektoriziramo:

$$W_1 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \longrightarrow (1,0,0,1), \quad W_2 = \begin{bmatrix} 0 & 0 \\ 1 & -1 \end{bmatrix} \longrightarrow (0,0,1,-1), \quad \dots$$

Podobno lahko vektorje priredimo funkcijam f_i in g_i:

$$f_1(A) = a_{11} + a_{22} \longrightarrow (1,0,0,1), \quad f_2(A) = a_{21} + a_{22} \longrightarrow (0,0,1,1), \quad \dots$$

Na ta način iz $f_i g_i W_i$ dobimo naslednjih 7 tenzorjev ranga 1:

$$\begin{aligned}
s_1 &= (1,0,0,1) \otimes (1,0,0,1) \otimes (1,0,0,1) \\
s_2 &= (0,0,1,1) \otimes (1,0,0,0) \otimes (0,0,1,-1) \\
s_3 &= (1,0,0,0) \otimes (0,1,0,-1) \otimes (0,1,0,1) \\
s_4 &= (0,0,0,1) \otimes (-1,0,1,0) \otimes (1,0,1,0) \\
s_5 &= (1,1,0,0) \otimes (0,0,0,1) \otimes (-1,1,0,0) \\
s_6 &= (-1,0,1,0) \otimes (1,1,0,0) \otimes (0,0,0,1) \\
s_7 &= (0,1,0,-1) \otimes (0,0,1,1) \otimes (1,0,0,0)
\end{aligned}$$

Brez težav bi lahko preverili, da velja $t = \sum_{i=1}^{7} s_i$. ♢

Primer 5.8. Poglejmo si še tenzor preslikave $\langle n,k,m \rangle$, tj. tenzor preslikave množenja $n \times k$ matrike s $k \times m$ matriko. Najprej si ponovno poglejmo tenzor matričnega množenja (5) iz prejšnjega primera. V rezini, označeni z E_{11}, sta neničelna le koeficienta pred $a_{11}b_{11}$ in $a_{12}b_{21}$. Element $(AB)_{11}$ produkta 2×2 matrik A in B pa se izraža po formuli

$$(AB)_{11} = a_{11}b_{11} + a_{12}b_{21}.$$

Splošneje, v tenzorju t, ki pripada preslikavi $\langle n,k,m \rangle$, so neničelni ravno koeficienti trojic, ki se pojavijo skupaj v formuli za izračun elementa produkta matrik $A \in F^{n \times k}$ in $B \in F^{k \times m}$:

$$(AB)_{ij} = \sum_{l=1}^{k} a_{il}b_{lj}.$$

To pomeni, da mora biti drugi indeks pri a enak prvemu indeksu pri b, prvi indeks pri a enak prvemu indeksu pri produktu in drugi indeks pri b enak drugemu indeksu pri produktu. Koeficiente tenzorja t lahko zapišemo takole:

$$t_{i_1j_1,i_2j_2,i_3j_3} = \begin{cases} 1 & j_1 = i_2 \text{ in } i_1 = i_3 \text{ in } j_2 = j_3 \\ 0 & \text{sicer} \end{cases} = \delta_{j_1i_2}\delta_{i_1i_3}\delta_{j_2j_3}.$$

Tenzor smo indeksirali z dvojnimi indeksi zaradi bolj enostavnega zapisa. Na matriko A v standardni bazi lahko namesto kot na tabelo števil velikosti $n \times k$ gledamo kot na vektor dolžine $n \cdot k$. Elemente matrike $a_{i_1j_1}$ smo oštevilčevali z indeksoma $i_1 = 1,\ldots,n$ in $j_1 = 1,\ldots,k$. Ko matriko vektoriziramo, bi načeloma potrebovali en sam indeks, ki šteje od 1 do $n \cdot k$, vendar bo v nadaljevanju lažje, če indeksa i_1j_1 združimo skupaj v "dvojni indeks". Tako i_1j_1 štejeta po prvi dimenziji tenzorja, tj. po vektorizirani matriki A, i_2j_2 po drugi dimenziji, tj. po vektorizirani matriki B in i_3j_3 po tretji dimenziji, po vektorizirani matriki produkta. Tak zapis bomo v nadaljevanju še večkrat uporabili. ♢

6. Lastnosti ranga

V tenzorski notaciji bomo precej lažje pokazali nekatere lastnosti ranga, ki jih bomo potrebovali pozneje, saj so določene operacije na tenzorjih veliko preprostejše, kot na bilineranih preslikavah.

Definicija 6.1. Naj bo $\pi \in S_3$ permutacija elementov $\{1,2,3\}$. *Permutacija dimenzij tenzorja* $t \in F^{n \times k \times m}$ je nov tenzor πt. Permutacija dimenzij tenzorja ranga 1 je enaka:

$$\pi(a_1 \otimes a_2 \otimes a_3) = a_{\pi^{-1}(1)} \otimes a_{\pi^{-1}(2)} \otimes a_{\pi^{-1}(3)}.$$

Permutacijo poljubnega tenzorja t pa izračunamo tako, da ga zapišemo kot neko vsoto tenzorjev ranga 1, $t = \sum_{i=1}^{s} a_{i1} \otimes a_{i2} \otimes a_{i3}$. Permutacijo dimenzij potem izračunamo na vsakem tenzorju ranga 1 posebej:

$$\pi t = \sum_{i=1}^{s} a_{i\pi^{-1}(1)} \otimes a_{i\pi^{-1}(2)} \otimes a_{i\pi^{-1}(3)}.$$

Opomba 6.2. Bilinearni izračun za poljubno preslikavo ϕ obstaja, vsak tenzor lahko zapišemo kot vsoto tenzorjev ranga 1. Recimo da je $t \in F^{n \times k \times m}$ tenzor. Tenzorski produkt standardnih baznih vektorjev $e_1 \in F^n, e_1 \in F^k$ in $e_1 \in F^m$ je tenzor, ki ima povsod same ničle, razen na mestu $(1,1,1)$, kjer ima vrednost 1. Podobno lahko naredimo tudi za ostala polja in tenzor t zapišemo kot vsoto:

$$t = \sum_{i=1}^{n} \sum_{j=1}^{k} \sum_{l=1}^{m} t_{ijl} e_i \otimes e_j \otimes e_l.$$

Opomba 6.3. Permutacija dimenzij tenzorja je dobro definirana. Denimo, da smo tenzor $t \in F^{m_1 \times m_2 \times m_2}$ zapisali kot vsoto tenzorjev ranga 1 na dva različna načina.

$$t = \sum_{i=1}^{n_1} a_{i1} \otimes a_{i2} \otimes a_{i3} = \sum_{j=1}^{n_2} b_{j1} \otimes b_{j2} \otimes b_{j3}.$$

Pokazati moramo, da je vrednost πt enaka, ne glede na to, na kateri vsoti delamo permutacijo elementov. Označimo elemente vektorjev v tenzorskih produktih na sledeč način: $a_{i1} = (a_{i11}, a_{i12}, \ldots, a_{i1m_1})$, $a_{i2} = (a_{i21}, a_{i22}, \ldots, a_{i2m_2})$, $a_{i3} = (a_{i31}, a_{i32}, \ldots, a_{i3m_3})$, analogno za b_{j1}, b_{j2}, b_{j3}. Vsak element tenzorja t se s temi elementi izraža na sledeč način:

$$t_{h_1 h_2 h_3} = \sum_{i=1}^{n_1} a_{i1h_1} a_{i2h_2} a_{i3h_3} = \sum_{j=1}^{n_2} b_{j1h_1} b_{j2h_2} b_{j3h_3}, \tag{6}$$

za vse indekse $h_1 = 1, \ldots, m_1$, $h_2 = 1, \ldots, m_2$ in $h_3 = 1, \ldots, m_3$. Vse, kar moramo narediti, da dobimo tenzor πt, je, da ustrezno permutiramo komponente vektorjev a_{i1}, a_{i2} in a_{i3} ter pripadajoče indekse. Dobimo:

$$(\pi t)_{k_1 k_2 k_3} = \sum_{i=1}^{n_1} a_{i\pi^{-1}(1)k_1} a_{i\pi^{-1}(2)k_2} a_{i\pi^{-1}(3)k_3} \tag{7}$$

oziroma v primeru vektorjev b_{i1}, b_{i2} in b_{i3}:

$$(\pi t)_{k_1 k_2 k_3} = \sum_{j=1}^{n_2} b_{j\pi^{-1}(1)k_1} b_{j\pi^{-1}(2)k_2} b_{j\pi^{-1}(3)k_3}, \tag{8}$$

kjer indeks k_1 teče od 1 do $m_{\pi^{-1}(1)}$, k_2 od 1 do $m_{\pi^{-1}(2)}$ in k_3 od 1 do $m_{\pi^{-1}(3)}$. Iz enačbe (6) sledi, da sta vsoti (7) in (8) enaki. Ker enakost velja za vse indekse k_1, k_2, k_3, velja tudi za celoten tenzor πt.

Analogija permutacije dimenzij tenzorja v dveh dimenzijah je transponiranje matrike. Vemo, da je rang transponiranke enak rangu prvotne matrike, zato naslednja lema ni presenetljiva.

Lema 6.4. *Permutacija dimenzij tenzorja ohranja rang.*

Dokaz. Dokazujemo, da je $R(t) = R(\pi t)$. Tenzor t lahko zapišemo kot vsoto $t = \sum_{i=1}^{r} t_i$, kjer so t_i tenzorji ranga 1 in r rang tenzorja t. Po definiciji permutacije dimenzij tenzorja je $\pi t = \sum_{i=1}^{r} \pi t_i$, torej je rang $R(\pi t) \leq R(t)$. Sklep ponovimo za tenzor πt in permutacijo π^{-1}. Tako dobimo zvezo $R(\pi^{-1}\pi t) \leq R(\pi t) \leq R(t)$, očitno pa je $\pi^{-1}\pi t = t$, torej velja enakost iz leme. □

Lema 6.5. *Permutacija rezin tenzorja ohranja rang.*

Dokaz. Naj bo t tenzor, ki pripada bilinearni preslikavi $\phi\colon U \times V \to W$, kjer je W prostor dimenzije m, $\{w_1, \ldots, w_m\}$ baza prostora W in $\sigma \in S_m$ permutacija. Naj bo t' nov tenzor, ki je po komponentah enak:

$$t'_{ijl} = t_{ij\sigma(l)}.$$

Če namesto baze $\{w_1, \ldots, w_m\}$ prostora W izberemo bazo $\{w_{\sigma(1)}, \ldots, w_{\sigma(m)}\}$, opazimo, da tenzor t' prav tako pripada preslikavi ϕ. Po trditvi 5.4 imata tenzorja t in t' enak rang. Torej permutacija rezin v tretji dimenziji ohranja rang. Za permutacijo rezin v prvi ali drugi lahko naredimo podoben dokaz, ali pa upoštevamo lemo 6.4, iz katere to direktno sledi. □

Lema nam pove, da za rang ni pomembno, v kakšem vrstnem redu so urejene posamezne rezine.

Poglejmo si preslikavo $\langle n, k, m\rangle$, tj. preslikavo množenja $n \times k$ matrike s $k \times m$ matriko. Tenzor t, ki pripada tej preslikavi, je iz prostora $F^{(n\times k)\times(k\times m)\times(n\times m)}$. Iz primera 5.8 vemo, da je t po komponentah enak

$$t_{i_1j_1,i_2j_2,i_3j_3} = \delta_{j_1i_2}\delta_{i_1i_3}\delta_{j_2j_3}.$$

Vsako od treh dimenzij tenzorja označujemo z dvojnimi indeksi, lahko bi sicer matriko vektorizirali in označevali z indeksi od 1 do kn (oziroma km ali mn), vendar nam bo označevanje z dvojnimi indeksi poenostavilo delo. Preuredimo rezine glede na tretjo dimenzijo na način, kot da bi "transponirali matriko", torej zamenjamo indeksa i_3 in j_3. Dobimo tenzor $t' \in F^{(n\times k)\times(k\times m)\times(m\times n)}$:

$$t'_{i_1j_1,i_2j_2,i_3j_3} = \delta_{j_1i_2}\delta_{i_1j_3}\delta_{j_2i_3}.$$

Po lemi 6.5 vemo, da je rang t' enak rangu t. Vzemimo permutacijo $\pi = \left(\begin{smallmatrix}1&2&3\\2&3&1\end{smallmatrix}\right) = (1\ 2\ 3)$ in naj bo $t'' = \pi t'$. Upoštevamo formulo za permutacijo dimenzij in dobimo $t'' \in F^{(m\times n)\times(n\times k)\times(k\times m)}$. Če želimo dobiti zapis po komponentah, to naredimo tako kot v opombi 6.3, med seboj "permutiramo" indekse tenzorja t', (i_1j_1, i_2j_2, j_3i_3).

$$\begin{aligned}(i_1, j_1) &\rightsquigarrow (i_{\pi^{-1}(1)}, j_{\pi^{-1}(1)}) = (i_3, j_3)\\ (i_2, j_2) &\rightsquigarrow (i_{\pi^{-1}(2)}, j_{\pi^{-1}(2)}) = (i_1, j_1)\\ (j_3, i_3) &\rightsquigarrow (j_{\pi^{-1}(3)}, i_{\pi^{-1}(3)}) = (j_2, i_2).\end{aligned}$$

Dobimo:

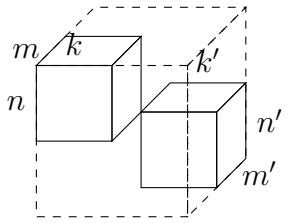
$$t''_{i_1j_1,i_2j_2,i_3j_3} = \delta_{j_3i_1}\delta_{i_3j_2}\delta_{j_1i_2}.$$

Zamenjamo oznake: $(a_1, b_1) = (i_3, j_3)$, $(a_2, b_2) = (i_1, j_1)$ in $(a_3, b_3) = (i_2, j_2)$.

$$t''_{a_1b_1,a_2b_2,a_3b_3} = \delta_{b_1a_2}\delta_{a_1b_3}\delta_{b_2a_3}.$$

Podobno kot prej preuredimo rezine glede na tretjo dimenzijo tako, da zamenjamo indeksa a_3 in b_3. Dobimo tenzor $t''' \in F^{(m\times n)\times(n\times k)\times(m\times k)}$, ki je po komponentah enak:

$$t'''_{a_1b_1,a_2b_2,a_3b_3} = \delta_{b_1a_2}\delta_{a_1a_3}\delta_{b_2b_3}.$$



SLIKA 1. Direktna vsota dveh tenzorjev za boljšo ponazoritev definicije.

Iz formule po komponentah in iz dimenzij tenzorja t''' opazimo, da je t''' tenzor, ki pripada preslikavi $\langle m, n, k\rangle$. Postopek bi lahko ponovili še enkrat in izračunali še $\pi^2 t'$, mu permutirali rezine ter dobili tenzor, ki pripada preslikavi $\langle k, m, n\rangle$. Po lemah 6.5 in 6.4 imajo tenzorji t, t', t'' in t''' enak rang. Ker imajo bilinearne preslikave enak rang kot tenzorji, ki jim pripadajo, je $R(\langle n, k, m\rangle) = R(\langle m, n, k\rangle) = R(\langle k, m, n\rangle)$. S tem smo dokazali naslednjo trditev.

Trditev 6.6. *Rang preslikave matričnega množenja $\langle n, k, m\rangle$ se ohranja, ko cikl̆no permutiramo n, k in m.*

$$R(\langle n, k, m\rangle) = R(\langle m, n, k\rangle) = R(\langle k, m, n\rangle).$$

Definirali bomo direktno vsoto in tenzorski produkt dveh tenzorjev, pozneje bomo videli, kje se to naravno pojavlja v rekurzivnih algoritmih za množenje matrik.

Definicija 6.7. Naj bosta $t \in F^{n\times k\times m}$ in $t' \in F^{n'\times k'\times m'}$ tenzorja. *Direktna vsota* t in t' je nov tenzor $t \oplus t' \in F^{(n+n')\times(k+k')\times(m+m')}$, katerega komponente so enake:

$$(t \oplus t')_{ijl} = \begin{cases} t_{ijl} & \text{če } i \le n,\ j \le k,\ l \le m \\ t'_{i-n,j-k,l-m} & \text{če } i > n,\ j > k,\ l > m \\ 0 & \text{sicer.} \end{cases}$$

Lema 6.8. *Rang direktne vsote dveh tenzorjev je manjši ali enak vsoti rangov teh dveh tenzorjev.*

Dokaz. Naj bo rang $t \in F^{n\times k\times m}$ enak r in rang $t' \in F^{n'\times k'\times m'}$ enak r'. Dokazujemo, da je $R(t \oplus t') \le R(t) + R(t')$. Vemo, da lahko oba tenzorja zapišemo kot vsoti:

$$t = \sum_{i=1}^{r} u_i \otimes v_i \otimes w_i, \quad t' = \sum_{i=1}^{r'} u'_i \otimes v'_i \otimes w'_i.$$

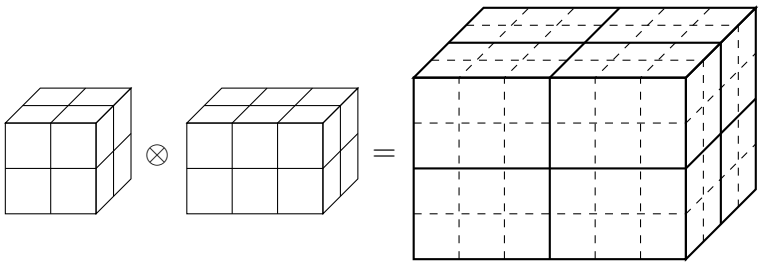
Vektorje u_i popravimo tako, da jim na koncu dodamo ničle, ter jih tako dopolnimo do dimenzije $n + n'$. Podobno dodamo ničle na začetek vektorjev u'_i, ter jih tako dopolnimo do dimenzije $n + n'$.

$$\hat{u}_i = (\underbrace{u_{i1}, \ldots, u_{in}}_{u_i}, \underbrace{0, \ldots, 0}_{n' \text{ ničel}}), \quad \hat{u}'_i = (\underbrace{0, \ldots, 0}_{n \text{ ničel}}, \underbrace{u'_{i1}, \ldots, u'_{in'}}_{u'_i})$$

Podobno dopolnimo tudi vektorje v_i, v'_i in w_i, w'_i. Direktna vsota $t \oplus t'$ se izraža kot:

$$t \oplus t' = \sum_{i=1}^{r} \hat{u}_i \otimes \hat{v}_i \otimes \hat{w}_i + \sum_{i=1}^{r'} \hat{u}'_i \otimes \hat{v}'_i \otimes \hat{w}'_i.$$

Tenzor $t \oplus t'$ smo zapisali kot vsoto $r + r'$ tenzorjev ranga 1, torej je njegov rang kvečjemu manjši. □



SLIKA 2. Tenzorski produkt dveh tenzorjev za boljšo ponazoritev definicije.

Opomba 6.9. Rang direktne vsote tenzorjev je lahko strogo manjši od vsote rangov posameznih tenzorjev. Primer je objavljen v [15].

Definicija 6.10. Naj bosta $t \in F^{n\times k\times m}$ in $t' \in F^{n'\times k'\times m'}$ tenzorja. *Tenzorski produkt* t in t' je nov tenzor $t \otimes t' \in F^{nn'\times kk'\times mm'}$, katerega komponente so enake:

$$(t \otimes t')_{ii',jj',ll'} = t_{ijl}t'_{i'j'l'}.$$

Opomba 6.11. Ponovno smo uporabili dva indeksa, npr. ii' za štetje od 1 do nn'. Dvojni indeks ii' pravzaprav pomeni $n'(i-1)+i'$, šteje po vektorizirani matriki velikosti $n \times n'$. V našem primeru bo večina formul precej enostavnejših v zapisu z dvojnimi indeksi. Za boljšo predstavo kako izgleda tenzorski produkt dveh tenzorjev, si poglejmo analogijo v dveh dimenzijah. Tenzorski produkt oziroma *Kroneckerjev produkt* matrik $A \in F^{n\times m}$ in $B \in F^{n'\times m'}$ je nova matrika $A \otimes B \in F^{nn'\times mm'}$, ki je po komponentah enaka:

$$(A \otimes B)_{ii',jj'} = a_{ij}b_{i'j'}.$$

Takšna definicija je analogna definiciji v treh dimenzijah, vendar lahko v tem primeru produkt zapišemo kot naslednjo bločno matriko:

$$A \otimes B = \begin{bmatrix} a_{11}B & \cdots & a_{1k}B \\ \vdots & & \vdots \\ a_{n1}B & \cdots & a_{nk}B \end{bmatrix}.$$

Indeksa i in j nam povesta, v katerem bloku se element nahaja, indeksa i' in j' pa natančno pozicijo v bloku. Zelo podobno je v primeru tenzorjev, samo da so bloki trirazsežni, glej sliko 2.

Lema 6.12. *Rang tenzorskega produkta dveh tenzorjev je manjši ali enak produktu rangov teh dveh tenzorjev.*

Dokaz. Kot prej naj bosta $t \in F^{n\times k\times m}$ in $t' \in F^{n'\times k'\times m'}$ tenzorja. Dokazujemo, da je $R(t \otimes t') \leq R(t)R(t')$. Ponovno tenzorja zapišemo kot vsoti $t = \sum_{i=1}^{r} u_i \otimes v_i \otimes w_i$ in $t' = \sum_{i=1}^{r'} u'_i \otimes v'_i \otimes w'_i$. Vemo, da je tenzorski produkt vektorjev $u_i \otimes u'_j$ matrika

$$u_i \otimes u'_j = \begin{bmatrix} u_{i1}u'_{j1} & u_{i1}u'_{j2} & \dots & u_{i1}u'_{jn'} \\ u_{i2}u'_{j1} & u_{i2}u'_{j2} & \dots & u_{i2}u'_{jn'} \\ \vdots & \vdots & & \vdots \\ u_{in}u'_{j1} & u_{in}u'_{j2} & \dots & u_{in}u'_{jn'} \end{bmatrix} \in F^{n\times n'};$$

za naše potrebe si to matriko predstavljamo vektorizirano, bomo pa za lažjo predstavo ohranili indeksiranje z dvema indeksoma. Podobno naredimo tudi za pare $v_i \otimes v'_j$ in $w_i \otimes w'_j$. Poskusimo izračunati $(u_i \otimes u'_j) \otimes (v_i \otimes v'_j) \otimes (w_i \otimes w'_j)$. Kot smo

že omenili, si $(u_i \otimes u'_j)$ in ostala dva člena predstavljamo kot vektorje, torej je izraz tenzor ranga 1, element $F^{nn' \times kk' \times mm'}$, njegove koeficiente pa lahko izračunamo po definiciji za tenzorski produkt treh vektorjev.

$$\begin{aligned}&\big((u_i \otimes u'_j) \otimes (v_i \otimes v'_j) \otimes (w_i \otimes w'_j)\big)_{xx',yy',zz'} \\ &\quad = (u_i \otimes u'_j)_{xx'} \cdot (v_i \otimes v'_j)_{yy'} \cdot (w_i \otimes w'_j)_{zz'} = u_{ix} u'_{jx'} v_{iy} v'_{jy'} w_{iz} w'_{jz'},\end{aligned}$$

kjer so indeksi $1 \le x \le n$, $1 \le x' \le n'$, $1 \le y \le k$, $1 \le y' \le k'$, $1 \le z \le m$ in $1 \le z' \le m'$. Sedaj lahko začnemo z računanjem.

$$\begin{aligned}(t \otimes t')_{xx',yy',zz'} &= \left(\left(\sum_{i=1}^{r} u_i \otimes v_i \otimes w_i\right) \otimes \left(\sum_{j=1}^{r'} u'_j \otimes v'_j \otimes w'_j\right)\right)_{xx',yy',zz'} \\ &= \left(\sum_{i=1}^{r}\sum_{j=1}^{r'} (u_i \otimes v_i \otimes w_i) \otimes (u'_j \otimes v'_j \otimes w'_j)\right)_{xx',yy',zz'}\end{aligned}$$

Namesto komponente vsote tenzorjev lahko gledamo vsoto komponent. Upoštevamo še definicijo produkta tenzorjev.

$$\begin{aligned}&= \sum_{i=1}^{r}\sum_{j=1}^{r'} (u_i \otimes v_i \otimes w_i)_{xyz} \cdot (u'_j \otimes v'_j \otimes w'_j)_{x'y'z'} \\ &= \sum_{i=1}^{r}\sum_{i=1}^{r'} u_{ix} v_{iy} w_{iz} \cdot u'_{jx'} v'_{jy'} w'_{jz'} = \sum_{i=1}^{r}\sum_{i=1}^{r'} u_{ix} u'_{jx'} v_{iy} v'_{jy'} w_{iz} w'_{jz'}\end{aligned}$$

V vsoti prepoznamo vrednost izraza, ki smo ga imeli zgoraj.

$$= \left(\sum_{i=1}^{r}\sum_{i=1}^{r'} (u_i \otimes u'_j) \otimes (v_i \otimes v'_j) \otimes (w_i \otimes w'_j)\right)_{xx',yy',zz'}$$

Na ta način smo tenzor $t \otimes t'$ zapisali z vsoto $r \cdot r'$ tenzorjev ranga 1. □

Primer 6.13. Naj bo t tenzor, ki pripada matričnemu množenju $\langle n, k, m\rangle$ in t' tenzor, ki pripada $\langle n', k', m'\rangle$. Kako izgleda tenzorski produkt $t \otimes t'$? Vemo, da je t element $F^{nk \times km \times nm}$ in t' element $F^{n'k' \times k'm' \times n'm'}$, torej bo produkt $t \otimes t'$ element $F^{nkn'k' \times kmk'm' \times nmn'm'}$, vsak element produkta bomo namesto z dvema indeksoma za lažji zapis označili z štirimi indeksi. Tenzorski produkt je po komponentah enak:

$$\begin{aligned}(t \otimes t')_{i_1 j_1 i'_1 j'_1, i_2 j_2 i'_2 j'_2, i_3 j_3 i'_3 j'_3} &= (\delta_{j_1 i_2} \delta_{i_1 i_3} \delta_{j_2 j_3})(\delta_{j'_1 i'_2} \delta_{i'_1 i'_3} \delta_{j'_2 j'_3}) \\ &= (\delta_{j_1 i_2} \delta_{j'_1 i'_2})(\delta_{i_1 i_3} \delta_{i'_1 i'_3})(\delta_{j_2 j_3} \delta_{j'_2 j'_3})\end{aligned}$$

Izraz $(\delta_{j_1 i_2} \delta_{j'_1 i'_2})$ je enak 1 le v primeru, ko je $j_1 = i_2$ in $j'_1 = i'_2$, kar je natanko tedaj, ko sta enaka para $(j_1, j'_1) = (i_2, i'_2)$. Podobno v preostalih dveh primerih.

$$= \delta_{(j_1,j'_1),(i_2,i'_2)} \delta_{(i_1,i'_1),(i_3,i'_3)} \delta_{(j_2,j'_2),(j_3,j'_3)}.$$

Par (j_1, j'_1) ima kk' različnih možnih vrednosti, lahko si predstavljamo, da ta par šteje od 1 do kk'.

$$(1,1), (1,2), \ldots, (1,k'),\ (2,1), (2,2), \ldots, (2,k'),\ (3,1), \ldots, (k,k')$$

Indeksa j_1 in j'_1 združimo skupaj. Naj bo $\tilde{j}_1 := (j_1 - 1)k' + j'_1$, ki nam prav tako šteje od 1 do kk'. Na podoben način združimo tudi par (i_1, i'_1) v $\tilde{i}_1 := (i_1 - 1)n' + i'_1$,

ki teče od 1 do nn', (i_2, i_2') v $\tilde{i}_2$ ki teče od 1 do kk' itd. Tenzor $(t \otimes t')$ lahko sedaj indeksiramo s spremenjenimi indeksi, 12-terica

$$(i_1, j_1, i_1', j_1';\ i_2, j_2, i_2', j_2';\ i_3, j_3, i_3', j_3')$$

prešteje enake možnosti kot, šesterica $(\tilde{i}_1, \tilde{j}_1;\ \tilde{i}_2, \tilde{j}_2;\ \tilde{i}_3, \tilde{j}_3)$. Indekse $\tilde{i}_1$, $\tilde{j}_1$, itd. smo definirali tako, da velja:

$$(j_1, j_1') = (i_2, i_2') \text{ natanko tedaj, ko je } \tilde{j}_1 = \tilde{i}_2,$$

zato je $\delta_{(j_1,j_1'),(i_2,i_2')} = \delta_{\tilde{j}_1\tilde{i}_2}$, podobno za ostala dva člena. Ko vse to zložimo skupaj, dobimo:

$$(t \otimes t')_{\tilde{i}_1\tilde{j}_1,\tilde{i}_2\tilde{j}_2,\tilde{i}_3\tilde{j}_3} = \delta_{\tilde{j}_1\tilde{i}_2}\delta_{\tilde{i}_1\tilde{i}_3}\delta_{\tilde{j}_2\tilde{j}_3}.$$

Če upoštevamo še vse meje, vidimo, da je $t \otimes t'$ ravno tenzor, prirejen preslikavi $\langle nn', kk', mm' \rangle$, matričnemu množenju $nn' \times kk'$ matrik s $kk' \times mm'$ matrikami. Vidimo torej, da je tenzorski produkt dveh tenzorjev matričnih množenj tenzor večjega matričnega množenja. To dejstvo bomo s pridom izkoristili v naslednjem izreku. ♢

Opomba 6.14. Rang preslikave matričnega množenja narašča z velikostjo matrik. Če je $n \leq m$, potem je $R(\langle n, n, n \rangle) \leq R(\langle m, m, m \rangle)$.

Recimo, da je $R(\langle m, m, m \rangle) = r$. Produkt dveh poljubnih $m \times m$ matrik A in B lahko po definiciji ranga izračunamo z vsoto

$$\sum_{i=1}^{r} f_i(A) g_i(B) W_i$$

za neke funkcionale $f_i \colon F^{m \times m} \to F$, $g_i \colon F^{m \times m} \to F$ in matrike $W_i \in F^{m \times m}$. Z $\tilde{A} \in F^{n \times n}$ označimo podmatriko matrike A, v kateri je prvih n stolpcev in prvih n vrstic matrike A. Podobno definiramo matrike $\tilde{B}$ in $\tilde{W}_i$. Definiramo še funkcionale $\tilde{f}_i \colon F^{m \times m} \to F$ in $\tilde{g}_i \colon F^{m \times m} \to F$ tako, da $n \times n$ matriko z ničlami razširimo do $m \times m$ matrike in na novi matriki uporabimo ustrezne funkcionale:

$$\tilde{f}_i\left(\begin{bmatrix} c_{11} & \cdots & c_{1n} \\ \vdots & & \vdots \\ c_{n1} & \cdots & c_{nn} \end{bmatrix}\right) = f_i\left(\begin{bmatrix} c_{11} & \cdots & c_{1n} & 0 & \ldots & 0 \\ \vdots & & \vdots & \vdots & & \vdots \\ c_{n1} & \cdots & c_{nn} & 0 & \ldots & 0 \\ 0 & \ldots & 0 & 0 & \ldots & 0 \\ \vdots & & \vdots & \vdots & & \vdots \\ 0 & \ldots & 0 & 0 & \ldots & 0 \end{bmatrix}\right).$$

Dobimo naslednjo enakost:

$$\tilde{A}\tilde{B} = \sum_{i=1}^{r} \tilde{f}_i(\tilde{A}) \tilde{g}_i(\tilde{B}) \tilde{W}_i.$$

Ker smo izbrali poljubni matriki A in B, to velja za vse matrike $\tilde{A}, \tilde{B} \in F^{n \times n}$. Našli smo bilinearni izračun produkta dveh $n \times n$ matrik dolžine r, zato je $R(\langle n, n, n \rangle) \leq r$.

Izrek 6.15. *Naj bo rang $R(\langle n, k, m \rangle) \leq r$. Potem za eksponent matričnega množenja ω velja:*

$$\omega \leq 3 \cdot \log_{nkm}(r).$$

Dokaz. Predpostavimo $R(\langle n,k,m\rangle) \leq r$. Naj bo t tenzor, ki pripada tej preslikavi. Iz trditve 6.6 sledi še $R(\langle m,n,k\rangle) \leq r$ in $R(\langle k,m,n\rangle) \leq r$. S t' in t'' označimo tenzorja teh dveh preslikav in s s tenzor

$$s = t \otimes t' \otimes t''.$$

Iz primera 6.13 vemo, da je $t \otimes t'$ tenzor matričnega množenja $\langle nm, kn, mk\rangle$. Za tenzorski produkt $(t \otimes t') \otimes t''$ sklep ponovimo. Tenzor s je torej tenzor preslikave $\langle nmk, knm, mkn\rangle = \langle nkm, nkm, nkm\rangle$. S pomočjo leme 6.12 lahko omejimo njegov rang:

$$R(s) = R((t \otimes t') \otimes t'') \leq R(t \otimes t')R(t'') \leq R(t)R(t')R(t'') \leq r^3.$$

Poglejmo si še range tenzorjev $s \otimes s$, $s \otimes s \otimes s$, $s \otimes s \otimes s \otimes s$, ... S podobnim sklepom kot zgoraj pridemo do ugotovitve, da je tenzorski produkt i faktorjev tenzorja s tenzor, ki pripada preslikavi $\langle (nkm)^i, (nkm)^i, (nkm)^i\rangle$.

$$R(\langle (nkm)^i, (nkm)^i, (nkm)^i\rangle) = R(\underbrace{s \otimes \cdots \otimes s}_{i-\text{krat}}) \leq R(s)^i \leq r^{3i}$$

Uporabimo še trik, da je $r = (nkm)^{\log_{nkm} r}$.

$$= \left((nkm)^{\log_{nkm} r}\right)^{3i} = \left(mkn^i\right)^{3 \cdot \log_{nkm} r}.$$

To velja za vse $i \in \mathbb{N}$. Če zamenjamo $(nkm)^i$ z N dobimo:

$$R(\langle N^i, N^i, N^i\rangle) \leq (N^i)^{3 \cdot \log_N r}.$$

Neenakost velja samo za potence števila N, če želimo omejiti eksponent matričnega množenja, moramo za vse $n_0 \in \mathbb{N}$ pokazati

$$R(\langle n_0, n_0, n_0\rangle) = O(n_0^{3 \cdot \log_N r}).$$

Poljubno število $n_0 \in \mathbb{N}$ lahko ujamemo med N^i in N^{i+1} za ustrezen i, upoštevamo še opombo 6.14.

$$R(\langle n_0, n_0, n_0\rangle) \leq R(\langle N^{i+1}, N^{i+1}, N^{i+1}\rangle) \leq (N^{i+1})^{3 \cdot \log_N r} = N^{3 \cdot \log_N r}(N^i)^{3 \cdot \log_N r}.$$

$N^{3 \cdot \log_N r}$ je konstanta, neodvisna od n_0, zato je

$$R(\langle n_0, n_0, n_0\rangle) = O((N^i)^{3 \cdot \log_N r}) = O(n_0^{3 \cdot \log_N r}),$$

torej je $\omega \leq 3 \cdot \log_{nkm} r$. □

Primer 6.16. Oglejmo si, kako bi izrek uporabili v primeru Strassenovega algoritma. Vemo, da je $R(\langle 2,2,2\rangle) \leq 7$. Izrek pravi, da je

$$\omega \leq 3 \cdot \log_{2 \cdot 2 \cdot 2}(7) = 3 \cdot \log_8(7) \approx 2{,}81,$$

kar je enako dobra omejitev, kot smo jo dobili v poglavju 2.

Ali lahko naredimo kaj podobnega tudi za večje matrike? V [6] je Laderman pokazal, kako tenzor preslikave $\langle 3,3,3\rangle$, ki je velikosti $9 \times 9 \times 9$, zapišemo kot vsoto 23 tenzorjev ranga 1. Vstavimo v formulo iz izreka in dobimo:

$$\omega \leq 3 \cdot \log_{3 \cdot 3 \cdot 3}(23) = 3 \cdot \log_{27}(23) \approx 2{,}85.$$

Torej bi lahko na podlagi teh 23 členov vsote razvili podoben algoritem kot za množenje 2×2 matrik, vendar pa bi bil asimptotično počasnejši.

V [11] je objavljena tabela 40 členov, s katerimi lahko izvedemo množenje $\langle 3,3,6\rangle$, torej množenje 3×3 matrike s 3×6 matriko. Naša formula nam pove:

$$\omega \leq 3 \cdot \log_{3 \cdot 3 \cdot 6}(40) \approx 2{,}7743,$$

kar predstavlja manjše izboljšanje. Ker pa matrike niso več kvadratne, rekurzivna zveza v tem primeru ni več tako preprosta. V članku je predlagan algoritem na 54×54 matrikah, ki jih večkrat bločno razdelimo na bloke velikosti 3×3 in 3×6. $\diamondsuit$

7. Mejni rang

Naš glavni cilj je čim bolj omejiti eksponent matričnega množenja in kot že vemo, lahko to storimo tako, da poskušamo čim bolj znižati rang matričnega množenja. Poleg tega, da je to težek problem, se izkaže, da ranga na ta način ne moremo znižati toliko, kot bi si želeli. V tem poglavju bomo vpeljali posplošitev ranga, ki jo bomo imenovali *mejni rang*. Za mejni rang bodo še vedno veljale nekatere lastnosti običajnega ranga. Pokazali bomo, da lahko število množenj še zmanjšamo, če se zadovoljimo s poljubno dobrim približkom točnega rezultata. S pomočjo mejnega ranga bomo nazaj omejili običajni rang in tako dobili posplošitev izreka 6.15. Sledili bomo večinoma [2, 5. poglavje].

Primer 7.1. Poglejmo si tenzor $t \in \mathbb{R}^{2\times 2\times 2}$ z rezinama

$$k_1 : \begin{matrix} & i_1 & i_2 \\ j_1 & 1 & 0 \\ j_2 & 0 & 0 \end{matrix}, \quad k_2 : \begin{matrix} & i_1 & i_2 \\ j_1 & 0 & 1 \\ j_2 & 1 & 0 \end{matrix}.$$

Rang t je očitno največ 3, saj ima t le 3 neničelne elemente na mestih $i_1j_1k_1$, $i_1j_2k_2$ in $i_2j_1k_2$ in ga zato lahko zapišemo kot vsoto

$$t = e_1 \otimes e_1 \otimes e_1 + e_1 \otimes e_2 \otimes e_2 + e_2 \otimes e_1 \otimes e_2,$$

kjer so e_1, e_2, e_3 standardni bazni vektorji. Da rang ni manjši ali enak 2, pa najlažje pokažemo tako, da poskusimo t zapisati kot vsoto dveh tenzorjev ranga 1

$$t = (u_1, u_2) \otimes (v_1, v_2) \otimes (w_1, w_2) + (u_1', u_2') \otimes (v_1', v_2') \otimes (w_1', w_2')$$

in tako dobimo sistem osmih enačb (ena enačba za vsak element tenzorja) z dvanajstimi neznankami, ki ga vstavimo v svoj priljubljeni program za reševanje sistemov (npr. Mathematica) in vidimo, da nima nobene rešitve. S tem dokažemo, da je rang tenzorja t enak 3.

Poglejmo si naslednji tenzor ranga 2:

$$t_\varepsilon = (1, \varepsilon) \otimes (1, \varepsilon) \otimes \left(0, \frac{1}{\varepsilon}\right) + (1, 0) \otimes (1, 0) \otimes \left(1, -\frac{1}{\varepsilon}\right).$$

Njegovi rezini sta enaki

$$k_1 : \begin{matrix} & i_1 & i_2 \\ j_1 & 1 & 0 \\ j_2 & 0 & 0 \end{matrix}, \quad k_2 : \begin{matrix} & i_1 & i_2 \\ j_1 & 0 & 1 \\ j_2 & 1 & \varepsilon \end{matrix}.$$

Ker sta t in t_ε tenzorja nad realnimi števili, lahko govorimo o konvergenci. Na $2 \times 2 \times 2$ tenzorjih vpeljemo normo:

$$||t|| = \sqrt{\sum_{i=1}^{2}\sum_{j=1}^{2}\sum_{l=1}^{2} t_{ijl}^2},$$

ki je posplošitev Frobeniusove norme za matrike. Ko gre ε proti 0, tenzor t_ε konvergira proti t. Tenzor ranga 3 smo tako poljubno dobro aproksimirali s tenzorjem ranga 2. $\diamondsuit$

Zgled pove, da lahko v nekaterih primerih za neki tenzor najdemo "bližnji" tenzor, katerega rang je manjši. Poskusili bomo spremeniti definicijo ranga tako, da bomo vključili tudi možni manjši rang "bližnjih" tenzorjev. V definiciji mejnega ranga bomo uporabili namesto konstantnih vektorjev u_i, v_i, w_i vektorje nad kolobarjem polinomov spremenljivke ε, torej $u_i(\varepsilon) \in F[\varepsilon]^n$, $v_i(\varepsilon) \in F[\varepsilon]^k$ in $w_i(\varepsilon) \in F[\varepsilon]^m$. Kaj v praksi vzamemo za ε, je odvisno od polja F. V primeru $F = \mathbb{R}$ lahko za ε preprosto vzamemo neko dovolj majhno pozitivno realno število.

Definicija 7.2. Naj bo $t \in F^{n\times k\times m}$ tenzor in $h \in \mathbb{N}$. Naj bodo $u_i(\varepsilon) \in F[\varepsilon]^n, v_i(\varepsilon) \in F[\varepsilon]^k, w_i(\varepsilon) \in F[\varepsilon]^m$ za vse $i = 1, \ldots, r$. Če velja

$$\sum_{i=1}^{r} u_i(\varepsilon) \otimes v_i(\varepsilon) \otimes w_i(\varepsilon) = \varepsilon^h t + O(\varepsilon^{h+1}),$$

potem vsoti $\sum_{i=1}^{r} u_i(\varepsilon) \otimes v_i(\varepsilon) \otimes w_i(\varepsilon)$ rečemo *približni izračun reda h* za tenzor t. Dolžina tega približnega izračuna je enaka r.

Približni rang reda h označimo z R_h in definiramo kot

$$R_h(t) := \min\{r \mid \text{obstaja približni izračun reda } h \text{ za } t \text{ dolžine } r\}.$$

Mejni rang tenzorja t je najmanjša vrednost $R_h(t)$, med vsemi $h \in \mathbb{N}$, označimo ga z $\underline{R}(t)$. Če zapišemo s formulo, je torej mejni rang enak:

$$\underline{R}(t) = \min_{h\in\mathbb{N}} R_h(t).$$

Že poimenovanje in oznaka namigujeta, da imata približni in mejni rang podobne lastnosti kot običajni rang. Približni izračun tenzorja je na nek način posplošitev bilineranega izračuna iz poglavja 3, približni rang pa je posplošitev ranga tenzorja. Pokazali bomo, da za $R_h(t)$ veljajo zelo podobne lastnosti kot za rang. Zadnja točka naslednje leme pove, da je v primeru, ko je $h = 0$, to ravno običajen rang.

Lema 7.3. *Za $R_h(t)$ iz zgornje definicije velja:*

(1) *Zaporedje $R_0(t)$, $R_1(t)$, $R_2(t)$, ... je padajoče.*
(2) $\underline{R}(t) = \lim_{h\to\infty} R_h(t)$.
(3) *Za izračun $R_h(t)$ zadostuje, da se pri izbiri $u_i(\varepsilon), v_i(\varepsilon), w_i(\varepsilon)$ omejimo na polinome stopnje največ h.*
(4) $R_0(t) = R(t)$.

Dokaz. (1) Pokažimo neenakost $R_h(t) \geq R_{h+1}(t)$ za nek $h \in \mathbb{N}$. Recimo, da je $R_h(t) = r$. Potem po definiciji obstajajo $u_i(\varepsilon) \in F[\varepsilon]^n, v_i(\varepsilon) \in F[\varepsilon]^k, w_i(\varepsilon) \in F[\varepsilon]^m$, da je vsota njihovih tenzorskih produktov približni izračun reda h za tenzor t:

$$\sum_{i=1}^{r} u_i(\varepsilon) \otimes v_i(\varepsilon) \otimes w_i(\varepsilon) = \varepsilon^h t + O(\varepsilon^{h+1}).$$

Poglejmo si vsoto tenzorskih produktov vektorjev $u_i'(\varepsilon) := \varepsilon u_i(\varepsilon)$, $v_i'(\varepsilon) = v_i(\varepsilon)$ in $w_i'(\varepsilon) = w_i(\varepsilon)$:

$$\sum_{i=1}^{r} u_i'(\varepsilon) \otimes v_i'(\varepsilon) \otimes w_i'(\varepsilon) = \varepsilon \sum_{i=1}^{r} u_i(\varepsilon) \otimes v_i(\varepsilon) \otimes w_i(\varepsilon) = \varepsilon^{h+1} t + O(\varepsilon^{h+2}).$$

Našli smo približni izračun reda $h + 1$ za t, dolžina katerega je enaka r. S tem smo pokazali, da je $R_{h+1}(t) \leq r = R_h(t)$.

(2) Mejni rang $\underline{R}(t)$ je po definiciji enak $\underline{R}(t) = \min_{h\in\mathbb{N}} R_h(t)$, iz točke (1) pa vemo, da je zaporedje $R_0(t)$, $R_1(t)$, $R_2(t)$, ... padajoče. Dokaz sledi.

(3) Naj bo $R_h(t) = r$, torej obstajajo $u_i(\varepsilon) \in F[\varepsilon]^n, v_i(\varepsilon) \in F[\varepsilon]^k, w_i(\varepsilon) \in F[\varepsilon]^m$ da je $\sum_{i=1}^{r} u_i(\varepsilon) \otimes v_i(\varepsilon) \otimes w_i(\varepsilon) = \varepsilon^h t + O(\varepsilon^{h+1})$. Recimo, da ima prvi element v vektorju $u_1(\varepsilon)$, ki je polinom, stopnjo vsaj $h+1$, torej, da je

$$u_{11}(\varepsilon) = \alpha_0 + \alpha_1\varepsilon + \cdots + \alpha_h\varepsilon^h + \alpha_{h+1}\varepsilon^{h+1} + \cdots,$$

kjer je $u_1(\varepsilon) = (u_{11}(\varepsilon), \ldots, u_{1n}(\varepsilon)) \in F[\varepsilon]^n$. Ko polinom $u_{11}(\varepsilon)$ pomnožimo s polinomi iz $v_1(\varepsilon)$ in $w_1(\varepsilon)$, se mu stopnja kvečjemu poveča, zmanjšati se ne more. Koeficient α_{h+1} torej stoji ob ε^{h+1} ali višji potenci. To pomeni, da na desni strani enačbe iz definicije α_{h+1} ne more nastopati pri členu $\varepsilon^h t$, ampak mora biti v delu $O(\varepsilon^{h+1})$. Če polinom $u_{11}(\varepsilon)$ popravimo tako, da α_{h+1} nastavimo na 0, bo enakost iz definicije še vedno veljala (zaradi O notacije, v resnici to ni zares enakost). Z istim argumentom lahko na 0 nastavimo tudi vse koeficiente pri višjih potencah ϵ. Enak postopek ponovimo še na ostalih polinomih višjih stopenj. Na ta način smo našli vektorje polinomov stopnje največ h, za katere enakost še vedno velja.

(4) Točka (3) pove, da se lahko v primeru, ko je $h = 0$, omejimo samo na konstantne polinome. Dobimo ravno definicijo ranga. □

Opomba 7.4. Podobno kot smo to naredili pri rangu, lahko približni in mejni rang definiramo tudi za bilinearne preslikave. Pokažemo lahko, da je mejni rang bilinearne preslikave enak mejnemu rangu tenzorja, ki pripada tej preslikavi.

Opremljeni z lemo lahko pokažemo nekatere lastnosti R_h, ki so zelo podobne lastnostim običajnega ranga.

Lema 7.5. *Permutacija dimenzij tenzorja ohranja približni rang.*

Dokaz te leme je enak kot dokaz leme 6.4 pri običajnem rangu, zato tu ne dokaza ne bomo naredili. Bomo pa dokazali posplošitev leme 6.8.

Lema 7.6. *Za direktno vsoto tenzorjev $t \in F^{n\times k\times m}$ in $t' \in F^{n'\times k'\times m'}$ ter $h, h' \in \mathbb{N}$ velja:*

$$R_{\max\{h,h'\}}(t \oplus t') \leq R_h(t) + R_{h'}(t').$$

Dokaz. Naj bosta t in t' tenzorja. Obstajata približna izračuna tenzorjev teh dveh tenzorjev, torej obstajajo $u_i(\varepsilon) \in F[\varepsilon]^n, v_i(\varepsilon) \in F[\varepsilon]^k, w_i(\varepsilon) \in F[\varepsilon]^m$ in $u'_i(\varepsilon) \in F[\varepsilon]^{n'}, v'_i(\varepsilon) \in F[\varepsilon]^{k'}, w'_i(\varepsilon) \in F[\varepsilon]^m$ da velja

$$\sum_{i=1}^{r} u_i(\varepsilon)\otimes v_i(\varepsilon)\otimes w_i(\varepsilon) = \varepsilon^h t + O(\varepsilon^{h+1}) \text{ in } \sum_{i=1}^{r'} u'_i(\varepsilon)\otimes v'_i(\varepsilon)\otimes w'_i(\varepsilon) = \varepsilon^{h'} t' + O(\varepsilon^{h'+1}).$$

Brez škode za splošnost lahko predpostavimo, da je $h \geq h'$. Vektorje u'_i za indekse $i = 1, \ldots, r'$ pomnožimo z $\varepsilon^{h-h'}$. Druga enačba se spremeni v:

$$\sum_{i=1}^{r'} (\varepsilon^{h-h'} u'_i(\varepsilon)) \otimes v'_i(\varepsilon) \otimes w'_i(\varepsilon) = \varepsilon^{h'+h-h'} t' + O(\varepsilon^{h'+h-h'+1}) = \varepsilon^h t' + O(\varepsilon^{h+1}).$$

V obeh enačbah na desni strani je pred t in pred t' enak člen ε^h. Od tod naprej je dokaz enak kot dokaz leme 6.8. Vektorje $u_i(\varepsilon)$, $v_i(\varepsilon)$, $w_i(\varepsilon)$ in $u'_i(\varepsilon)$, $v'_i(\varepsilon)$, $w'_i(\varepsilon)$ z ničlami razširimo do ustreznih dimenzij in dobimo:

$$\sum_{i=1}^{r} \hat{u}_i(\varepsilon) \otimes \hat{v}_i(\varepsilon) \otimes \hat{w}_i(\varepsilon) + \sum_{i=1}^{r'} (\varepsilon^{h-h'} \hat{u'}_i(\varepsilon)) \otimes \hat{v'}_i(\varepsilon) \otimes \hat{w'}_i(\varepsilon) = \varepsilon^h (t \oplus t') + O(\varepsilon^{h+1}).$$

S tem smo našli nek približen izračun tenzorja $t \oplus t'$ dolžine $r + r'$, torej je po definiciji $R_h(t \otimes t') \leq r + r'$. □

Prav tako obstaja analogija leme 6.12 za tenzorske produkte tenzorjev. Dokaz bomo naredili bolj površno, saj je zelo podoben dokazu leme za običajen rang.

Lema 7.7. *Za tenzorski produkt tenzorjev $t \in F^{n\times k\times m}$ in $t' \in F^{n'\times k'\times m'}$ ter $h, h' \in \mathbb{N}$ velja:*

$$R_{h+h'}(t \otimes t') \leq R_h(t) \cdot R_{h'}(t').$$

Dokaz (skica). Naj bo $R_h(t) = r$ in $R_{h'}(t') = r'$. Kot v prejšnjem dokazu lahko po definiciji R_h zapišemo:

$$T = \sum_{i=1}^{r} u_i(\varepsilon) \otimes v_i(\varepsilon) \otimes w_i(\varepsilon) = \varepsilon^h t + O(\varepsilon^{h+1}).$$

Člen $O(\varepsilon^{h+1})$ lahko zapišemo kot $\varepsilon^{h+1}s$, kjer je s nek tenzor. Podobno naredimo za tenzor t', dobimo

$$T = \varepsilon^h t + \varepsilon^{h+1}s \quad \text{in} \quad T' = \varepsilon^{h'}t' + \varepsilon^{h'+1}s'.$$

Tenzorski produkt $T \otimes T'$ bi lahko izračunali tako, kot smo to naredili v dokazu leme 6.12. Za tenzorski produkt ni težko po členih preveriti distributivnosti, zato lahko tenzorski produkt izračunamo tudi takole:

$$\begin{aligned} T \otimes T' &= (\varepsilon^h t + \varepsilon^{h+1}s) \otimes (\varepsilon^{h'}t' + \varepsilon^{h'+1}s') = \\ &= \varepsilon^{h+h'} t \otimes t' + \varepsilon^{h+h'+1} t \otimes s' + \varepsilon^{h+1+h'} s \otimes t' + \varepsilon^{h+1+h'+1} s \otimes s' = \\ &= \varepsilon^{h+h'} t \otimes t' + O(\varepsilon^{h+h'+1}). \end{aligned}$$

Dobili smo nek približen izračun tenzorja $t \otimes t'$ reda $h + h'$, torej je $R_{h+h'}(t \otimes t') \leq r \cdot r'$. □

Posledica zgornjih lem je analogija trditve 6.6, kjer rang zamenjamo z R_h. Dokaz je podoben dokazu trditve 6.6, zato ga ne bomo naredili.

Trditev 7.8. *Za preslikavo matričnega množenja $\langle n, k, m\rangle$ in $h \in \mathbb{N}$ velja:*

$$R_h(\langle n, k, m\rangle) = R_h(\langle m, n, k\rangle) = R_h(\langle k, m, n\rangle).$$

Naslednja lema bo razkrila, zakaj je bilo sploh smiselno vpeljati približne izračune in mejni rang. Za vsak $h \in \mathbb{N}$ vemo, da je $R(t) = R_0(t) \geq R_h(t)$. Pokazali bomo še neenakost v drugo smer, rang tenzorja $R(t)$ lahko omejimo z nekim večkratnikom $R_h(t)$.

Lema 7.9. *Za poljuben tenzor t in $h \in \mathbb{N}$ velja naslednja neenakost:*

$$R(t) \leq \binom{h+2}{2} R_h(t).$$

Dokaz. Naj bo t tenzor in $R_h(t) = r$. Po definiciji $R_h(t)$ obstajajo taki vektorji $u_i(\varepsilon) \in F[\varepsilon]^n, v_i(\varepsilon) \in F[\varepsilon]^k, w_i(\varepsilon) \in F[\varepsilon]^m$, da velja:

$$\sum_{i=1}^{r} u_i(\varepsilon) \otimes v_i(\varepsilon) \otimes w_i(\varepsilon) = \varepsilon^h t + O(\varepsilon^{h+1}). \tag{9}$$

Zaradi leme 7.3 lahko predpostavimo, da so elementi vektorjev $u_i(\varepsilon)$, $v_i(\varepsilon)$, $w_i(\varepsilon)$ polinomi stopnje največ h. Zato lahko vektor $u_i(\varepsilon)$ zapišemo kot vsoto

$$u_i(\varepsilon) = \sum_{j_u=0}^{h} \varepsilon^{j_u} u_{ij_u},$$

kjer so $u_{i1}, u_{i2}, \ldots, u_{ih}$ vektorji iz F^n. Podobno lahko zapišemo tudi kot $v_i(\varepsilon) = \sum_{j_v=0}^{h} \varepsilon^{j_v} v_{ij_v}$ in $w_i(\varepsilon) = \sum_{j_w=0}^{h} \varepsilon^{j_w} w_{ij_w}$. Izračunajmo produkt $u_i(\varepsilon) \otimes v_i(\varepsilon) \otimes w_i(\varepsilon)$:

$$\begin{aligned} u_i(\varepsilon) \otimes v_i(\varepsilon) \otimes w_i(\varepsilon) &= \left(\sum_{j_u=0}^{h} \varepsilon^{j_u} u_{ij_u}\right) \otimes \left(\sum_{j_v=0}^{h} \varepsilon^{j_v} v_{ij_v}\right) \otimes \left(\sum_{j_w=0}^{h} \varepsilon^{j_w} w_{ij_w}\right) \\ &= \sum_{j_u=0}^{h} \sum_{j_v=0}^{h} \sum_{j_w=0}^{h} \varepsilon^{j_u+j_v+j_w} u_{ij_u} \otimes v_{ij_v} \otimes w_{ij_w}. \end{aligned}$$

Dobljeno vstavimo v enačbo (9):

$$\sum_{i=1}^{r} \sum_{j_u=0}^{h} \sum_{j_v=0}^{h} \sum_{j_w=0}^{h} \varepsilon^{j_u+j_v+j_w} u_{ij_u} \otimes v_{ij_v} \otimes w_{ij_w} = \varepsilon^h t + O(\varepsilon^{h+1}).$$

S pomočjo te enačbe bi radi omejili rang tenzorja t. Pri nekem indeksu i nas zanimajo vse možne kombinacije indeksov j_u, j_v, j_w, da bo $j_u + j_v + j_w = h$. Če vključimo samo te indekse, bomo dobili samo potence ε^h, člen $O(\varepsilon^{h+1})$ pa bo enak 0.

$$\sum_{i=1}^{r} \sum_{j_u+j_v+j_w=h} \varepsilon^h (u_{ij_u} \otimes v_{ij_v} \otimes w_{ij_w}) = \varepsilon^h t. \tag{10}$$

Enakost v tem primeru velja tudi, ko odstranimo ε^h.

Število možnih kombinacij trojic (j_u, j_v, j_w), kjer je $j_u + j_v + j_w = h$, znamo izračunati iz diskretne matematike. Lahko si predstavljamo, da želimo h kroglic razdeliti v tri škatle, pri čemer med škatlami ločimo, med kroglicami pa ne. Število vseh možnih razdelitev je enako $\binom{h+3-1}{3-1}$ (eden od načinov izpeljave: na $h+3-1$ mest razdelimo $3-1$ palčk in h kroglic, vsaka razporeditev določa eno od razporeditev po škatlah).

Število vseh členov v vsoti iz enačbe (10) je torej enako $r\binom{h+3-1}{3-1} = r\binom{h+2}{2}$. Po definiciji ranga je $R(t) \leq r\binom{h+2}{2}$. □

Opremljeni z lemami lahko sedaj pokažemo, da izrek 6.15 velja tudi za mejni rang. Večina dokaza je podobna dokazu izreka iz prejšnjega poglavja, zato bomo nekatere korake naredili malo hitreje.

Izrek 7.10. *Naj bo mejni rang $\underline{R}(\langle n, k, m\rangle) \leq r$. Potem za eksponent matričnega množenja ω velja:*

$$\omega \leq 3 \cdot \log_{nkm}(r).$$

Dokaz. Po definiciji mejnega ranga obstaja naravno število h, da je $\underline{R}(\langle n, k, m\rangle) = R_h(\langle n, k, m\rangle)$. Po predpostavki velja $R_h(\langle n, k, m\rangle) \leq r$, iz trditve 7.8 sledi še $R_h(\langle m, n, k\rangle) \leq r$ in $R_h(\langle k, m, n\rangle) \leq r$. Označimo s t tenzor preslikave $\langle n, k, m\rangle$, s t' tenzor $\langle m, n, k\rangle$, s t'' tenzor $\langle k, m, n\rangle$ in s s tenzor $s = t \otimes t' \otimes t''$. Vemo, da je s tenzor preslikave $\langle nkm, nkm, nkm\rangle$. S pomočjo leme 7.7 lahko omejimo $R_{3h}(s)$.

$$R_{3h}(s) = R_{3h}((t \otimes t') \otimes t'') \leq R_h(t) R_h(t') R_h(t'') \leq r^3.$$

Spomnimo se, da je tenzorski produkt i faktorjev tenzorja s tenzor, ki pripada preslikavi $\langle (nkm)^i, (nkm)^i, (nkm)^i\rangle$. Računamo:

$$\begin{aligned} R_{3ih}(\langle (nkm)^i, (nkm)^i, (nkm)^i\rangle) &= R_{3ih}(\underbrace{s \otimes \cdots \otimes s}_{i-\text{krat}}) \leq R_{3h}(s)^i \leq r^{3i} \\ &= \left((nkm)^{\log_{nkm} r}\right)^{3i} = \left(mkn^i\right)^{3 \cdot \log_{nkm} r}. \end{aligned}$$

Tu smo predpostavili, da lahko R_h definiramo tudi za bilinearne preslikave, ne le za tenzorje; glej opombo 7.4. Zgornja neenačba velja za vse $i \in \mathbb{N}$. Če zamenjamo nkm z N dobimo:

$$R_{3ih}(\langle N^i, N^i, N^i\rangle) \leq (N^i)^{3\cdot\log_N r}.$$

Z uporabo leme 7.9 lahko z dobljenim omejimo običajen rang:

$$R(\langle N^i, N^i, N^i\rangle) \leq \binom{3ih+2}{2} R_{3ih}(\langle N^i, N^i, N^i\rangle) \leq \binom{3ih+2}{2} (N^i)^{3\cdot\log_N r}.$$

Označimo $M = N^i$ in dobimo:

$$R(\langle M, M, M\rangle) \leq \frac{(3h\log_N M + 2)(3h\log_N M + 1)}{2} M^{3\cdot\log_N r}.$$

Vemo, da $\log_N M$ narašča počasneje kot M^ε za vsak $\varepsilon > 0$, enako velja tudi za $(\log_N M)^2$, torej v O notaciji velja:

$$R(\langle M, M, M\rangle) = O(M^{3\cdot\log_N r+\varepsilon}),$$

za vsak $\varepsilon > 0$. Podobno kot v izreku za običajen rang ostala naravna števila ujamemo med N^i in N^{i+1}. Lahko bi dokazali analog opombe 6.14 za približni rang. Sledi, da je $\omega \leq 3\log_N r + \varepsilon$ za vsak $\varepsilon > 0$, zato je $\omega \leq 3\log_N r$. □

Pogledali si bomo dva primera, v katerih bomo s pomočjo izreka poskusili znižati eksponent matričnega množenja.

Primer 7.11. Bini, Capovani, Lotti in Romani so v [3] prikazali algoritem, ki deluje s pomočjo približnega računanja. Osnovna ideja izhaja iz množenja 2×2 matrik, kjer izračunamo vse člene, razen člena spodaj desno.

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}. \tag{11}$$

Vemo, da po običajni formuli za množenje dveh 2×2 matrik potrebujemo dve množenji za vsak člen c_{ij}, torej v tem primeru 6 množenj za izračun c_{11}, c_{12}, in c_{21}. Da se pokazati tudi, da samo s petimi množenji tega ni mogoče narediti. Pokazali bomo, kako lahko s pomočjo približnega računanja to naredimo samo s petimi produkti.

$$\begin{aligned} p_1 &= (a_{12} + \varepsilon a_{22})b_{21} \\ p_2 &= a_{11}(b_{11} + \varepsilon b_{12}) \\ p_3 &= a_{12}(b_{11} + b_{21} + \varepsilon b_{22}) \\ p_4 &= (a_{11} + a_{12} + \varepsilon a_{21})b_{11} \\ p_5 &= (a_{12} + \varepsilon a_{21})(b_{11} + \varepsilon b_{22}) \end{aligned}$$

S temi produkti lahko izračunamo člene c_{ij}:

$$\begin{aligned} \varepsilon p_2 + \varepsilon p_1 &= \varepsilon(a_{11}b_{11} + a_{12}b_{21}) + O(\varepsilon^2) = \varepsilon c_{11} + O(\varepsilon^2) \\ p_2 - p_4 + p_5 &= \varepsilon(a_{11}b_{12} + a_{12}b_{22}) + O(\varepsilon^2) = \varepsilon c_{12} + O(\varepsilon^2) \\ p_1 - p_3 + p_5 &= \varepsilon(a_{21}b_{11} + a_{22}b_{21}) + O(\varepsilon^2) = \varepsilon c_{21} + O(\varepsilon^2), \end{aligned}$$

kjer smo v $O(\varepsilon^2)$ skupaj zbrali člene, pri katerih je potenca ε^2. Sedaj naredimo simetrično podobno matriko, le da sedaj odstranimo zgornji levi člen:

$$\begin{bmatrix} & c_{13} \\ c_{22} & c_{23} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{12} & b_{13} \\ b_{22} & b_{23} \end{bmatrix}. \tag{12}$$

Tudi v tem primeru lahko s približnim računanjem s samo petimi produkti izračunamo c_{13}, c_{22}, c_{23}. Matrike v enačbi (12) so indeksirane drugače kot v enačbi (11) zato, da lahko ti dve združimo skupaj in dobimo:

$$\begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{bmatrix}.$$

Z običajnim računanjem bi za izračun produkta 2×2 matrike z 2×3 matriko potrebovali 12 množenj. Pokazali smo pa, da lahko s približnimi izračuni to naredimo z 10 množenji, 5 množenj za c_{11}, c_{12}, c_{21} ter 5 za c_{13}, c_{22}, c_{23}, z drugimi besedami:

$$R_1(\langle 2, 2, 3\rangle) \leq 10.$$

Poskusimo uporabiti trditev 7.8. Sledi, da je $R_1(\langle 3, 2, 2\rangle) \leq 10$ in $R_1(\langle 2, 3, 2\rangle) \leq 10$. Zaradi leme 7.7 velja

$$R_3(\langle 2 \cdot 3 \cdot 2, 2 \cdot 2 \cdot 3, 3 \cdot 2 \cdot 2\rangle) = R_3(\langle 12, 12, 12\rangle) \leq 10^3.$$

Uporabimo lemo 7.9 in dobimo, da je

$$R(\langle 12, 12, 12\rangle) \leq \binom{3+2}{2} \cdot 10^3 = 10^4.$$

Rang matričnega množenja 12×12 je torej manjši kot 10 000. Na žalost s tem nismo dosegli ničesar, saj za izračun produkta dveh 12×12 matrik z običajno formulo potrebujemo samo $12^3 = 1728 < 10^4$ množenj.

Ta pot ni dala rezultata, ker smo se omejili na matrike velikosti 12×12, ta algoritem pa se izplača samo za dovolj velike velikosti matrik. Namesto tega se splača, da najprej uporabimo lemo 7.7 za $\underline{R}(\langle 12^i, 12^i, 12^i\rangle)$ in šele nato ocenimo rang po lemi 7.9. Lahko pa preprosto uporabimo izrek in dobimo, da je

$$\omega \leq 3 \log_{2 \cdot 2 \cdot 3} 10 = 3 \log_{12} 10 \approx 2{,}7799. \qquad \diamondsuit$$

Primer 7.12. Če dve 3×3 matriki zmnožimo po običajni formuli za množenje matrik, potrebujemo 27 množenj. V primeru 6.16 smo povedali, da lahko to naredimo že s 23 množenji, torej da je $R(\langle 3, 3, 3\rangle) \leq 23$. Če gledamo približna množenja, pa lahko to naredimo že z 21 množenji (katerih 21 produktov potrebujemo, je zapisano v [10]). Torej je $\underline{R}(\langle 3, 3, 3\rangle) \leq 21$. Uporabimo izrek in dobimo, da je

$$\omega \leq 3 \log_{3 \cdot 3 \cdot 3} 21 = 3 \log_{27} 21 \approx 2{,}7712. \qquad \diamondsuit$$

8. Nižje meje za eksponent matričnega množenja

Strassenov algoritem iz leta 1969 je prvi izboljšal Pan, ki je opazil, da je problem množenja dveh matrik ekvivalenten problemu računanja sledi produkta treh matrik in na ta način prišel do algoritma za množenje 70×70 matrik, s katerim je pokazal, da je $\omega \leq 2{,}7801$ [12, poglavje 1.2].

Naslednji algoritem, ki je postavil malenkost nižjo zgornjo mejo za ω, je bil Binijev algoritem iz primera 7.11. Pomemben del Binijevega algoritma je, da ne izračunamo celotnega produkta dveh 2×2 matrik, ampak samo tri od štirih elementov produkta. Dva takšna delna izračuna nato združimo v večji tenzor matričnega množenja. To idejo je leta 1981 še razširil Schönhage v [10] in na ta način pokazal, da je $\omega \leq 2{,}548$. V istem članku je predstavil še metodo, ki prav tako nadaljuje že predstavljeno teorijo o mejnem rangu, s katero je prišel do meje $\omega \leq 2{,}522$. Leto pozneje je mejo znižal Romani [14], ki je že prej sodeloval pri odkritju Binijevega algoritma. Nova zgornja meja je bila $\omega \leq 2{,}517$. Istega leta sta Schönhagejevo metodo še izboljšala

tudi Coppersmith in Winograd, ki sta pokazala, da je $\omega \leq 2{,}496$ [12, poglavje 1.2]. Leta 1986 je Strassen mejo še znižal, eksponent matričnega množenja je omejil z 2,479 [12, poglavje 1.2]. Coppersmith in Winograd sta izboljšala Strassenovo metodo in leta 1989 omejila ω z 2,376 [12, poglavje 1.2]. Algoritem sta še malo izboljšala Vassilevska Williams leta 2013 in Le Gall leta 2014 [17]. Trenutna najnižja zgornja meja je $\omega \leq 2{,}3728639$ [17], natančna vrednost pa je še odprt problem.

Vsi zgornji algoritmi v ozadju vključujejo teorijo o tenzorjih matričnega množenja in mejnem rangu. Pojavil se je tudi čisto drugačen način, ki ne uporablja tenzorjev matričnega množenja, temveč si pomaga z lastnostmi iz teorije grup [12, poglavje 5.]. Do najboljšega rezultata z uporabo tega načina so prišli Cohn, Kleinberg, Szegedy in Umans [5], ki so pokazali, da je $\omega \leq 2{,}41$.

9. Implementacija algoritmov

V prejšnjih poglavjih smo omejevali rang preslikave matričnega množenja in s tem omejili časovno zahtevnost matričnega množenja. Po definiciji O-notacije vemo, da so predstavljeni algoritmi hitrejši kot običajno matrično množenje, če so le matrike dovolj velike. Zanima nas, kaj v praksi pomeni "dovolj velike". Algoritem, ki sta ga razvila Coppersmith in Winograd je t. i. galaktični algoritem [18], to je algoritem z dobrimi asimptotičnimi lastnostmi, ki pa se ga ne splača uporabiti v praksi, saj na Zemlji nimamo dovolj velikih problemov, na katerih bi opazili razliko v primerjavi z astimptotično počasnejšimi algoritmi.

9.1. **Zapolnjevanje in luščenje.** V tem poglavju bomo vse naredili samo na primeru Strassnovega algoritma, za ostale implementirane algoritme je ideja enaka, razlika je le v številu blokov. Ideje so iz [7]. Želimo implementirati funkcijo, ki bo izračunala produkt $n \times k$ matrike s $k \times m$ matriko. V primeru klasičnega algoritma za množenje matrik je to preprosto, samo sledimo formuli. Osnovni rekurzivni Strassenov algoritem pa deluje samo na matrikah velikosti $2^i \times 2^i$:

- Če je $i = 0$, množimo dva skalarja.
- Če je $i \geq 1$, potem na matriki gledamo kot na bločni 2×2 matriki, vsak blok je velikosti $2^{i-1} \times 2^{i-1}$. Izvedemo korak Strassnovega algoritma.

Kaj naredimo, če dimenzije matrik niso potence števila 2 ali pa niso kvadratne? Recimo, da računamo produkt matrik $A \in F^{n \times k}$ in $B \in F^{k \times m}$. Najpreprosteje je, če poiščemo najbližjo potenco števila 2:

$$i = \min \left\{ j \in \mathbb{N}_0 \mid 2^j \geq \max\{n, k, m\} \right\}.$$

Matriki A in B z ničlami dopolnimo do velikosti $2^i \times 2^i$. Opazimo, da je prvih n vrstic in m stolpcev produkta dopolnjenih matrik ravno podmatrika AB.

$$\left.\begin{array}{l} \\ \\ 2^i \end{array}\right\} \underbrace{\left[\begin{array}{c|c} A & 0 \\ \hline 0 & 0 \end{array}\right]}_{2^i} \underbrace{\left[\begin{array}{c|c} B & 0 \\ \hline 0 & 0 \end{array}\right]}_{2^i} = \left[\begin{array}{c|c} AB & 0 \\ \hline 0 & 0 \end{array}\right]$$

Na dopolnjenih matrikah lahko izvedemo klasični Strassenov algoritem. Takšen način dopolnjevanja imenujemo *statično zapolnjevanje*. Že v posledici 2.3 smo pokazali, da v primeru kvadratnih matrik statično zapolnjevanje ne spremeni asimptotične časovne zahtevnosti. Če pa imamo "smolo", lahko naše množenje traja tudi precej dlje. Poglejmo si preprost primer s konkretnimi velikostmi matrik.

Primer 9.1. Če sta matriki A in B velikosti 1025×1025, potem ju bomo s statičnim zapolnjevanjem najprej dopolnili do velikosti 2048×2048. S tem smo število vrstic in stolpcev matrik A in B skoraj podvojili. Ker je časovna zahtevnost Strassenovega algoritma $O(n^{\log_2 7})$, bomo za izračun produkta dvakrat večjih matrik potrebovali približno $2^{\log_2 7} = 7$-krat toliko časa.

V primeru statičnega zapolnjevanja potrebujemo za produkt dveh 1025×1025 matrik približno 7-krat toliko časa kot za produkt dveh 1024×1024 matrik. Pri algoritmih, ki delujejo na večjih matrikah, je ta skok v času še izrazitejši, npr. Ladermanov algoritem iz primera 6.16, ki deluje na 3×3 matrikah, porabi za produkt dveh $(3^i + 1) \times (3^i + 1)$ matrik približno 23-krat več časa kot za produkt dveh $3^i \times 3^i$ matrik. ♢

Težave z velikimi skoki v času računanja, ko povečujemo velikost matrik, se znebimo z *dinamičnim zapolnjevanjem*. Namesto da matriki A in B zapolnimo do velikosti $2^i \times 2^i$, lahko samo po potrebi dodamo en stolpec oziroma vrstico:

- Če ima matrika A liho vrstic, ji dodamo še eno vrstico ničel.
- Če ima matrika A liho stolpcev, ji dodamo še en stolpec ničel.

Enako naredimo za matriko B. Ko obe matriki dopolnimo do sodih dimenzij, ju lahko razdelimo na bloke in izvedemo en korak Strassenovega algoritma. Ponavljamo rekurzivno.

Namesto da dodajamo ničelne vrstice in stolpce, lahko odvečno vrstico ali stolpec odstranimo, izvedemo Strassenov algoritem, po koncu odstranjene vrstice in stolpce dodamo nazaj in izračunamo še dodatnih nekaj produktov, ki smo jih prej izpustili. To idejo imenujemo *dinamično luščenje*.

Recimo, da so vse dimenzije n, k in m lihe. Matriki A in B razdelimo na bloke:

$$A = \begin{bmatrix} A_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \quad \text{in} \quad B = \begin{bmatrix} B_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}.$$

Blok A_{11} je velikosti $(n-1) \times (k-1)$, vektor a_{12} velikosti $(n-1) \times 1$, vrstica a_{21} velikosti $1 \times (k-1)$, a_{22} pa je skalar. Podobno za matriko B. Produkt $C = AB$ izračunamo po formuli za bločno množenje matrik.

$$\begin{bmatrix} C_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} A_{11}B_{11} + a_{12}b_{21} & A_{11}b_{12} + a_{12}b_{22} \\ a_{21}B_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{bmatrix}.$$

Produkt $A_{11}B_{11}$ izračunamo z uporabo Strassenovega algoritma. Ostali produkti pa so množenje matrike z vektorjem, skalarni produkt dveh vektorjev ali množenje vektorja s skalarjem, pri njih si s Strassenovim algoritmom ne moremo pomagati, zato jih izračunamo s klasičnim algoritmom za množenje matrik. V primeru Strassenovega algoritma se da pokazati, da dinamično luščenje potrebuje manj dodatnih operacij kot statično ali dinamično zapolnjevanje [8, poglavje 4.1].

9.2. **Implementacija.** Del diplomskega dela je tudi implementacija nekaterih predstavljenih algoritmov za množenje matrik. Za implementacijo sem izbral programski jezik C++. Čas izvajanja algoritmov je bil merjen na računalniku s procesorjem Intel® Core™ i7-6700HQ in 16 GiB spomina. Izvorna koda je vključena v prilogi,

Algoritem	Preslikava	Tip	Zahtevnost
Klasičen algoritem	/	točen	$O(n^3)$
Strassenov algoritem (poglavje 2, [13])	$\langle 2,2,2\rangle$	točen	$O(n^{2,81})$
Ladermanov algoritem (primer 6.16, [6])	$\langle 3,3,3\rangle$	točen	$O(n^{2,85})$
Binijev algoritem (primer 7.11, [3])	$\langle 2,2,3\rangle$	približen	$O(n^{2,78})$
Schönhagejev algoritem (primer 7.12, [10])	$\langle 3,3,3\rangle$	približen	$O(n^{2,77})$

TABELA 1. Implementirani algoritmi. Za vsak algoritem je navedena še bilinearna preslikava množenja matrik, na kateri temelji en rekurzivni korak, ali je algoritem točen ali približen in časovna zahtevnost algoritma.

glej [9] za celoten projekt. Namen je ugotoviti uporabnost predstavljenih algoritmov v praksi. Napisani algoritmi niso preveč optimizirani, saj je za to delo bolj pomembna berljivost izvorne kode kot sama hitrost izvajanja, zanimajo nas predvsem razlike med posameznimi algoritmi in naraščanje časa izvajanja z velikostjo matrik. Tako kot v [1] sem po nekaj poskusih s Strassenovim algoritmom namesto zapolnjevanja izbral dinamično luščenje. Tudi vse ostale algoritme sem implementiral z dinamičnim luščenjem. Implementirani algoritmi so v tabeli 1.

Implementacija Strassenovega in Ladermanovega algoritma ni zahtevna, bolj kot ne samo vstavljanje v rekurzivno formulo. Kako implementirati približne algoritme, pa ni tako zelo očitno. Približnega računanja se lahko lotimo na dva načina.

Če želimo dobiti natančen rezultat, moramo implementirati celotno množenje polinomov. Npr. po končanem koraku Binijevega algoritma, ko dobimo εC (glej primer 7.11), iz polinomov vzamemo samo koeficiente pri ε^1. V primeru Schönhagejevega algoritma naredimo podobno, upoštevamo formule iz [10], da dobimo rezultat, moramo ponekod deliti z ε in ponekod z ε^2.

Na žalost pa smo s tem načinom implementacije povečali časovno zahtevnost algoritma. Težava je v tem, da produkt dveh polinomov ni več operacija, za katero bi potrebovali konstantno časa, ampak je odvisna od stopnje teh dveh polinomov. Na začetku začnemo s konstantnimi polinomi, vendar pa se stopnja z rekurzivnimi klici povečuje. V vsakem rekurzivnem klicu Binijevega algoritma polinomu povečamo stopnjo za 1, v vsakem klicu Schönhagejevega algoritma pa za 2. Če množimo dve $n \times n$ matriki, je klicev približno $\log_2 n$ oziroma $\log_3 n$, torej za vsako seštevanje potrebujemo $O(\log n)$ in za množenje $O((\log n)^2)$ operacij (če uporabimo običajno formulo za množenje polinomov, seveda se da, tako kot matrike, tudi polinome množiti hitreje). Časovna zahtevnost celotnega eksaktnega Binijevega algoritma je $O((\log n)^2 n^{2,7799}) = O(n^{2,78})$, ker logaritem narašča počasneje kot katera koli potenca n. V praksi se čas izvajanja precej poveča, npr. če sta matriki veliki 1000×1000, potem je $(\log_3 1000)^2 \approx 40$.

Drugi način je, da si za ε izberemo neko majhno realno število, npr. $\varepsilon = 10^{-3}$. Računamo kot običajno, ko pa želimo dobiti koeficient v "polinomu" pri ε preprosto delimo z ε (ker oba algoritma delujeta tako, da so koeficienti pri nižjih potencah ε enaki 0). Na ta način ne dobimo natančnega rezultata, saj rezultat še vedno vključuje višje potence ε. Če je ε majhen, bo ta napaka majhna. S tem smo se odpovedali točnemu rezultatu, vendar pa je na ta način čas računanja produkta konstanten in ne več odvisen od stopnje polinomov.

Algoritem	Čas izvajanja [s]
Klasičen algoritem	2,33
Strassenov algoritem	1,92
Ladermanov algoritem	2,10
Binijev algoritem (točen)	59,54
Binijev algoritem (približen)	2,24
Schönhagejev algoritem (točen)	61,86
Schönhagejev algoritem (približen)	1,97

TABELA 2. Čas računanja produkta dveh naključnih 500×500 matrik.

Težava druge implementacije se pojavi pri velikih stopnjah polinomov. Realnih števil v računalniku ne moremo dobro predstaviti, ponavadi uporabimo števila s plavajočo vejico. Če računamo v dvojni natančnosti, so izračuni natančni na približno 15 decimalnih mest. Če smo izbrali npr. $\varepsilon = 10^{-3}$, potem za vrednost izraza $1 + \varepsilon^6$ dobimo:

$$1 + \varepsilon^6 = 1 + 10^{-18} = 1{,}000000000000000001 = 1{,}0.$$

Torej nam člen z ε^6 pri seštevanju izgine. Ko se to zgodi, to pomeni, da koeficienta pri ε^6 v nadaljevanju izvajanja algoritma sploh ne bomo upoštevali. Res je, da v enem koraku Binijevega algoritma zanemarimo vse člene s stopnjo 2 ali več, vendar to ne pomeni, da nikoli ne potrebujemo koeficientov pri višjih stopnjah polinomov, upoštevati moramo tudi rekurzijo. V osnovnem koraku želimo, da nam rekurzivni klic vrne koeficiente pri ε. Ker v rekurzivnem klicu spet računamo z Binijevim algoritmom in moramo izračunati tudi koeficiente pri ε, potrebujemo izračunane tudi koeficiente pri ε^2. V vsakem rekurzivnem klicu moramo izračunati eno stopnjo več. Če nam koeficient pri ε^6 izgine, to pomeni, da je izračun v petem rekurzivnem koraku napačen.

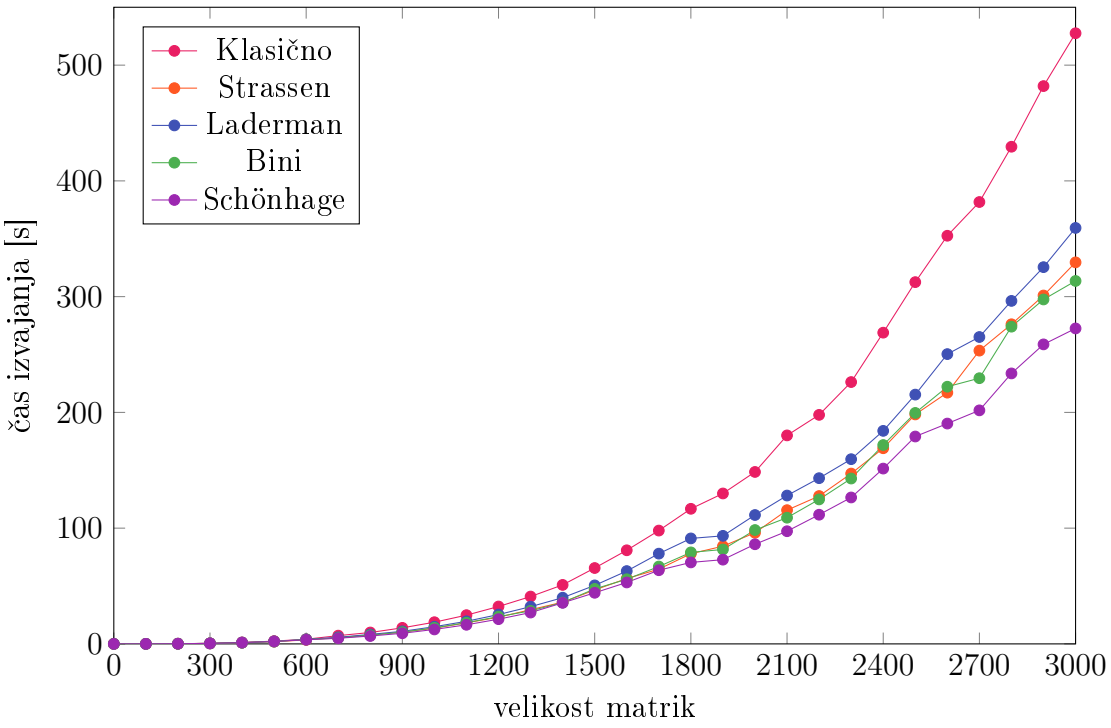
Tako v [1, poglavje 3.4] kot v [7, poglavje 3.4] osnovno implementacijo algoritmov še spremenijo. Vemo, da so predstavljeni algoritmi hitrejši za dovolj velike matrike, za manjše matrike pa porabimo manj operacij, če uporabimo klasično formulo za množenje matrik. Ko množimo matriki $A \in F^{n \times k}$ in $B \in F^{k \times m}$ in je

$$n \leq N \quad \text{ali} \quad k \leq N \quad \text{ali} \quad m \leq N,$$

se vrnemo na klasično matrično množenje. Parameter N lahko določimo eksperimentalno za vsak algoritem posebej, ampak določanje optimalnega parametra ni enostaven problem. Po precej preizkusih brez enoličnih zaključkov sem se na koncu odločil, da vsem algoritmom dodelim enak parameter $N = 200$ (v [7] je bil za Strassenov algoritem izbran parameter $N = 199$).

Primerjava algoritmov pri računanju produkta dveh naključnih 500×500 matrik je v tabeli 2. Vključuje obe možnosti implementacije Binijevega in Schönhagejevega algoritma. Takoj opazimo, da so časi eksaktnega računanja pri tej velikosti matrik neprimerljivo višji. Ker želimo algoritme primerjati še na večjih matrikah, bomo opustili obravnavo obeh točnih algoritmov.

Učinkovitost algoritmov v praksi sem primerjal s preprostim eksperimentom. Za vse $n = 100, 200, 300, \ldots, 3000$ sem generiral dve naključni $n \times n$ matriki s celimi števili med 0 in 10 ter za vsak algoritem izmeril čas množenja. Rezultati so vidni na sliki 3. Takoj opazimo, da je za velike matrike klasično množenje daleč najpočasnejše, tudi ostali algoritmi so razvrščeni približno tako, kot bi pričakovali iz



Slika 3. Čas izvajanja algoritmov v odvisnosti od velikosti matrik.

časovne zahtevnosti. Ladermanov algoritem je počasnejši od Strassenovega, Schönhagejev algoritem pa je najhitrejši. Ali to pomeni, da je Schönhagejev algoritem tudi najboljši v praksi? Kot smo že omenili, je težava približno računanje v velikih matrikah, končni rezultat ni nujno blizu pravilnega rezultata. Če želimo izbrati zanesljiv algoritem, potem je najbolj primeren Strassenov algoritem.

Dodatek A. Priloga

Ta priloga vsebuje implementacijo algoritmov, opisano v poglavju 9. Za lepše strukturiran projekt glej [9].

```
#include <iostream>
#include <array>
#include <vector>
#include <cassert>
#include <algorithm>
#include <tuple>

template<class Scalar>
class Matrix {
public:
    // number of rows in this matrix
    unsigned int rows;
    // number of columns in this matrix
    unsigned int cols;

    // where actual matrix data is stored
    // (maybe this should be changed to C's array which is faster)
    std::vector<Scalar> data;

    // default constructor is empty matrix
    Matrix() : rows(0), cols(0), data({}) {}

    // constructs matrix of size rows x cols, filled with initial_value
    Matrix(const unsigned int _rows, const unsigned int _cols,
```

```
       const Scalar &initial_value) : rows(_rows), cols(_cols) {
    // allocate necessary space and fill
    data = std::vector<Scalar>(_rows * _cols, initial_value);
}

// constructs matrix of size rows x cols, filled with zeros
static const Matrix<Scalar>
zeros(const unsigned int rows, const unsigned int cols) {
    return Matrix(rows, cols, Scalar(0));
}

// copy matrix from 1D array
// no dimension checks are performed
Matrix(const std::vector<Scalar> &_data, const unsigned int _rows,
       const unsigned int _cols) : data(_data),
                                   rows(_rows),
                                   cols(_cols) {}

// Cast matrix of other type to this type.
template<typename OtherScalar>
Matrix(const Matrix<OtherScalar> A) {
    rows = A.rows;
    cols = A.cols;
    data = std::vector<Scalar>(A.data.begin(), A.data.end());
}

// matrix is indexed from (0, 0) to (rows-1, cols-1)
// because reference is returned, values can also be set using this
Scalar &operator[](const std::pair<unsigned int, unsigned int> location) {
    const int i = location.first;
    const int j = location.second;

    return data[i * cols + j];
}

// same as above, except this is const version
const Scalar &
operator[](const std::pair<unsigned int, unsigned int> location) const {
    const int i = location.first;
    const int j = location.second;

    return data[i * cols + j];
}

// return transposed  matrix
Matrix<Scalar> transposed() {
    std::vector<Scalar> new_data(rows * cols);
    for (unsigned int i = 0; i < rows; ++i) {
        for (unsigned int j = 0; j < cols; ++j) {
            new_data[j * rows + i] = data[i * cols + j];
        }
    }
    return Matrix<Scalar>(new_data, cols, rows);
}

// block sub-matrix of current matrix
// create new smaller matrix and copy data to it
Matrix<Scalar> subblock(std::pair<unsigned int, unsigned int> top_left,
                        std::pair<unsigned int, unsigned int> block_size) const {
    // unpack
    unsigned int start_row, start_col;
    unsigned int block_rows, block_cols;

    std::tie(start_row, start_col) = top_left;
    std::tie(block_rows, block_cols) = block_size;

    // check dimensions
    assert(start_row + block_rows <= rows);
```

```
    assert(start_col + block_cols <= cols);

    // allocate new vector and create new matrix
    std::vector<Scalar> block_data(block_rows * block_cols);
    Matrix<Scalar> block(block_data, block_rows, block_cols);

    // get iterator of newly created matrix
    auto output_iter = block.data.begin();

    for (unsigned int i = 0; i < block_rows; ++i) {
        // copy i-th row to newly created
        auto input_iter =
                data.begin() + ((start_row + i) * cols + start_col);
        for (unsigned int j = 0; j < block_cols; ++j) {
            // move value
            *output_iter++ += *input_iter++;
        }
    }
    return block;
}

// add block starting from top_left to this matrix
Matrix<Scalar> &block_add(std::pair<unsigned int, unsigned int> top_left,
                          const Matrix<Scalar> &block) {
    unsigned int start_row = top_left.first;
    unsigned int start_col = top_left.second;

    // check if dimensions are correct
    assert(start_row + block.rows <= rows &&
           start_col + block.cols <= cols);

    // get iterator of block matrix
    auto block_iter = block.data.begin();

    for (unsigned int i = 0; i < block.rows; ++i) {
        // add i-th row to our data
        auto data_iter =
                data.begin() + ((start_row + i) * cols + start_col);
        for (unsigned int j = 0; j < block.cols; ++j) {
            // add value
            *data_iter++ += *block_iter++;
        }
    }
    return *this;
}

// subtract block starting from top_left to this matrix
// exactly same as block_add except here we are subtracting
Matrix<Scalar> &
block_subtract(std::pair<unsigned int, unsigned int> top_left,
               const Matrix<Scalar> &block) {
    unsigned int start_row = top_left.first;
    unsigned int start_col = top_left.second;

    // check if dimensions are correct
    assert(start_row + block.rows <= rows &&
           start_col + block.cols <= cols);

    // get iterator of block matrix
    auto block_iter = block.data.begin();

    for (unsigned int i = 0; i < block.rows; ++i) {
        // add i-th row to our data
        auto data_iter =
                data.begin() + ((start_row + i) * cols + start_col);
        for (unsigned int j = 0; j < block.cols; ++j) {
            // subtract value
            *data_iter++ -= *block_iter++;
```

```
            }
        }
        return *this;
    }

    // add other matrix to this matrix
    // does not create new matrix, changes current one
    Matrix<Scalar> &operator+=(const Matrix<Scalar> &other) {
        // check if dimensions are correct
        assert(rows == other.rows && cols == other.cols);

        // add other's data to our data
        auto iter = other.data.begin();
        for (auto &element : data) {
            element += *iter;
            iter++;
        }
        return *this;
    }

    // subtract other matrix from this matrix
    // does not create new matrix, changes current one
    Matrix<Scalar> &operator-=(const Matrix<Scalar> &other) {
        // ensure that both matrices have same shape
        assert(rows == other.rows && cols == other.cols);

        // subtract other's data from our data
        auto iter = other.data.begin();
        for (auto &element : data) {
            element -= *iter;
            iter++;
        }
        return *this;
    }

    // multiply matrix with a scalar
    Matrix<Scalar> &operator*=(const Scalar &s) {
        // multiply all elements with a scalar value
        for (auto &element : data) {
            element *= s;
        }
        return *this;
    }

    // divide matrix by a scalar
    Matrix<Scalar> &operator/=(const Scalar &s) {
        // multiply all elements with a scalar value
        for (auto &element : data) {
            element /= s;
        }
        return *this;
    }

    // check equality by elements
    bool operator==(const Matrix<Scalar> &other) const {
        if (rows != other.rows || cols != other.cols) return false;

        // if dimensions are correct, we need to check all items
        for (unsigned int i = 0; i < rows * cols; ++i) {
            if (data[i] != other.data[i]) return false;
        }
        return true;
    }

    bool operator!=(const Matrix<Scalar> &other) const {
        return !(*this == other);
    }
};
```

```cpp
// Adds two matrices, creates new matrix object, leaves original matrices unchanged
template<class Scalar>
Matrix<Scalar> operator+(const Matrix<Scalar> &A, const Matrix<Scalar> &B) {
    Matrix<Scalar> result = A;
    result += B;
    return result;
}

// Subtracts one matrix from another, creates new matrix object, leaves original matrices unchanged
template<class Scalar>
Matrix<Scalar> operator-(const Matrix<Scalar> &A, const Matrix<Scalar> &B) {
    Matrix<Scalar> result = A;
    result -= B;
    return result;
}

// Multiply matrix with a scalar
template<class Scalar>
Matrix<Scalar> operator*(const Scalar &s, const Matrix<Scalar> &A) {
    Matrix<Scalar> result = A;
    result *= s;
    return result;
}

template<class Scalar>
Matrix<Scalar> operator==(const Matrix<Scalar> &A, const Matrix<Scalar> &B) {
    return A == B;
}

// make matrix printable
template<typename Scalar>
std::ostream &operator<<(std::ostream &os, const Matrix<Scalar> &A) {
    auto iter = A.data.begin();

    os << "(" << A.rows << " x " << A.cols << ")" << std::endl;
    for (size_t i = 0; i < A.rows; i++) {
        for (size_t j = 0; j < A.cols; j++) {
            os << *iter << '\t';
            iter++;
        }
        os << std::endl;
    }
    return os;
}

// performs dynamic peeling for matrix product <n, k, m> algorithm
// (dimensions n, k, and m mean in how many blocks we split matrices A and B)
// this function directly adds needed products to matrix C
template<typename Scalar>
void dynamic_peeling(const Matrix<Scalar> &A, const Matrix<Scalar> &B,
                     Matrix<Scalar> &C,
                     unsigned int n, unsigned int k, unsigned int m) {
    // check if matrix dimensions are valid
    assert(A.rows == C.rows && A.cols == B.rows && B.cols == C.cols);

    // how many rows and cols were included in algorithm
    unsigned int included_rows_A = (A.rows / n) * n,
            included_cols_A = (A.cols / k) * k,
            included_rows_B = (B.rows / k) * k,
            included_cols_B = (B.cols / m) * m;

    // for how many rows do we need to do dynamic peeling?
    unsigned int need_peeling_rows_A = A.rows - included_rows_A,
            need_peeling_cols_A = A.cols - included_cols_A,
            need_peeling_rows_B = B.rows - included_rows_B,
            need_peeling_cols_B = B.cols - included_cols_B;
```

```
    // add product of not included columns from A * not included rows from B
    // we will split this in 3 blocks we can easily calculate

    // first block
    // A * B = C
    // | . . 0 |   | . . . |   | 0 0 . |
    // | . . 0 | * | . . . | = | 0 0 . |
    // | . . . |   | 0 0 . |   | . . . |
    if (need_peeling_cols_A > 0) {
        Matrix<Scalar> A_extra = A.subblock({0, included_cols_A},
                                            {included_rows_A,
                                             need_peeling_cols_A});
        Matrix<Scalar> B_extra = B.subblock({included_rows_B, 0},
                                            {need_peeling_rows_B,
                                             included_cols_B});

        // calculate this product and add it to large matrix C in correct place
        Matrix<Scalar> product = multiply_classic(A_extra, B_extra);

        C.block_add({0, 0}, product);
    }

    // second block
    // A * B = C
    // | 0 0 0 |   | . . 0 |   | . . 0 |
    // | 0 0 0 | * | . . 0 | = | . . 0 |
    // | 0 0 0 |   | . . 0 |   | . . 0 |
    if (need_peeling_cols_B > 0) {
        Matrix<Scalar> B_extra = B.subblock({0, included_cols_B},
                                            {B.rows, need_peeling_cols_B});

        // calculate product and add it to product C
        Matrix<Scalar> product = multiply_classic(A, B_extra);
        C.block_add({0, included_cols_B}, product);
    }

    // third block
    // A * B = C
    // | . . . |   | 0 0 . |   | . . . |
    // | . . . | * | 0 0 . | = | . . . |
    // | 0 0 0 |   | 0 0 . |   | 0 0 . |
    if (need_peeling_rows_A > 0) {
        Matrix<Scalar> A_extra = A.subblock({included_rows_A, 0},
                                            {need_peeling_rows_A, A.cols});
        Matrix<Scalar> B_extra = B.subblock({0, 0}, {B.rows, included_cols_B});

        // calculate this product and add it to large matrix C in correct place
        Matrix<Scalar> product = multiply_classic(A_extra, B_extra);
        C.block_add({included_rows_A, 0}, product);
    }
}

// Represents polynomial in epsilon, needed for Bini's and Schonhage's algorithm.
// Because algorithm requires polynomial multiplication, which is quadratic, this is not a great idea.
template<typename Scalar>
class Polynomial {
public:
    std::vector<Scalar> a;

    // linear polynomial
    Polynomial(Scalar a0, Scalar a1) : a({a0, a1}) {};

    // Returns a constant polynomial with a constant c.
    Polynomial(Scalar a0) : a({a0}) {};

    // zero
    Polynomial() : a({Scalar(0)}) {}
```

```
// Returns a polynomial that represents epsilon
static Polynomial<Scalar> epsilon() {
    return Polynomial<Scalar>(Scalar(0), Scalar(1));
}

Polynomial<Scalar> &operator*=(const Polynomial<Scalar> &other) {
    std::vector<Scalar> product(a.size() + other.a.size() - 1, 0);

    for (size_t i = 0; i < other.a.size(); ++i) {
        for (size_t j = 0; j < a.size(); ++j) {
            product[i + j] += a[j] * other.a[i];
        }
    }

    a = product;
    return *this;
}

Polynomial<Scalar> &operator+=(const Polynomial<Scalar> &other) {
    while (a.size() < other.a.size()) {
        a.push_back(Scalar(0));
    }
    for (unsigned int i = 0; i < std::min(a.size(), other.a.size()); i++) {
        a[i] += other.a[i];
    }
    return *this;
}

Polynomial<Scalar> &operator-=(const Polynomial<Scalar> &other) {
    while (a.size() < other.a.size()) {
        a.push_back(Scalar(0));
    }
    for (unsigned int i = 0; i < std::min(a.size(), other.a.size()); i++) {
        a[i] -= other.a[i];
    }
    return *this;
}

Polynomial<Scalar> &operator/=(const Polynomial<Scalar> &other) {
    // Not actual polynomial division.
    // Needed for Bini's algorithm where we divide with epsilon at the end of a recursive step.
    // Needed for Schonhage's algorithm where we divide with epsilon and epsilon squared.
    assert(// epsilon
            (other.a.size() == 2 && other.a[0] == Scalar(0) &&
             other.a[1] == Scalar(1)) ||
            // epsilon squared
            (other.a.size() == 3 && other.a[0] == Scalar(0) &&
             other.a[1] == Scalar(0) && other.a[2] == Scalar(1))
    );

    // divide by epsilon
    if (other.a.size() == 2) {
        if (a.size() == 1) {
            a = {0};
        } else {
            for (unsigned int i = 0; i < a.size() - 1; i++) {
                a[i] = a[i + 1];
            }
            a.pop_back();
        }
    } else if (other.a.size() == 3) {
        if (a.size() <= 2) {
            a = {0};
        } else {
            for (unsigned int i = 0; i < a.size() - 2; i++) {
                a[i] = a[i + 2];
            }
            a.pop_back();
```

```
                a.pop_back();
            }
        } else {
            assert("Not implemented.");
        }

        return *this;
    }
};

template<class Scalar>
Polynomial<Scalar>
operator+(const Polynomial<Scalar> &p, const Polynomial<Scalar> &q) {
    Polynomial<Scalar> result = p;
    result += q;
    return result;
}

template<class Scalar>
Polynomial<Scalar>
operator-(const Polynomial<Scalar> &p, const Polynomial<Scalar> &q) {
    Polynomial<Scalar> result = p;
    result -= q;
    return result;
}

template<class Scalar>
Polynomial<Scalar>
operator*(const Polynomial<Scalar> &p, const Polynomial<Scalar> &q) {
    Polynomial<Scalar> result = p;
    result *= q;
    return result;
}

template<class Scalar>
Polynomial<Scalar>
operator/(const Polynomial<Scalar> &p, const Polynomial<Scalar> &q) {
    Polynomial<Scalar> result = p;
    result /= q;
    return result;
}

// function that converts polynomial matrix to scalar matrix
// (we can imagine that we sent epsilon to zero, actually we just take constant term)
// to convert in other direction just use matrix constructor
template<typename Scalar>
Matrix<Scalar> polynomial_to_scalar(const Matrix<Polynomial<Scalar>> &A) {
    std::vector<Scalar> data(A.data.size());

    for (unsigned int i = 0; i < A.data.size(); i++) {
        data[i] = (A.data[i]).a[0];
    }

    return Matrix<Scalar>(data, A.rows, A.cols);
}

// make polynomial printable
template<typename Scalar>
std::ostream &operator<<(std::ostream &os, const Polynomial<Scalar> &p) {
    for (unsigned int i = 0; i < p.a.size() - 1; i++) {
        os << p.a[i] << "e^" << i << " + ";
    }
    return os << p.a[p.a.size() - 1] << "e^" << p.a.size() - 1;
}

// finds power of 2 larger (or same as) given value
unsigned int next_power_of_2(unsigned int value) {
    unsigned int power = 1;
```

```
    unsigned int n = 2;

    while (n < value) {
        power++;
        n *= 2;

        // otherwise we have too large matrices,
        // using integers for sizes is not sufficient
        assert(power < 32);
    }

    return n;
}

// when to switch to classic matrix multiplication
const unsigned int strassen_threshold = 200;
const unsigned int laderman_threshold = 200;
const unsigned int bini_threshold = 200;
const unsigned int schonhage_threshold = 200;

// CLASSIC MULTIPLICATION
template<class Scalar>
Matrix<Scalar>
multiply_classic(const Matrix<Scalar> &A, const Matrix<Scalar> &B) {
    // check dimensions
    assert(A.cols == B.rows);

    // create new matrix
    Matrix<Scalar> C = Matrix<Scalar>::zeros(A.rows, B.cols);

    for (unsigned int i = 0; i < A.rows; ++i) {
        for (unsigned int k = 0; k < A.cols; ++k) {
            // calculate c_ij
            auto Aik = A[{i, k}];
            // this will be Bkj, but using iterators improves multiplication speed by a factor of 2
            auto iter_B = B.data.begin() + k * B.cols;
            // this will be Cij
            auto iter_C = C.data.begin() + i * C.cols;
            for (unsigned int j = 0; j < B.cols; ++j) {
                *(iter_C + j) += Aik * (*(iter_B + j));
            }
        }
    }
    return C;
}

// STRASSEN'S ALGORITHM
template<class Scalar>
Matrix<Scalar>
multiply_strassen(const Matrix<Scalar> &A, const Matrix<Scalar> &B) {
    // dimension check
    assert(A.cols == B.rows);

    // if any of the dimensions is too small, strassen's algorithm wont help
    if (std::min(A.rows, std::min(A.cols, B.cols)) <= strassen_threshold) {
        return multiply_classic(A, B);
    }

    // split into subblocks:
    // if dimensions are even, make normal strassen step
    // if dimensions are odd, round matrix size down, do a normal strassen algorithm for smaller
    // matrix and perform dynamic peeling after this step finishes
    std::pair<unsigned int, unsigned int>
            block_A = {A.rows / 2, A.cols / 2},
            block_B = {B.rows / 2, B.cols / 2},
            product_block = {A.rows / 2, B.cols / 2};

    Matrix<Scalar> A11 = A.subblock({0, 0}, block_A),
```

```
            A12 = A.subblock({0, A.cols / 2}, block_A),
            A21 = A.subblock({A.rows / 2, 0}, block_A),
            A22 = A.subblock({A.rows / 2, A.cols / 2}, block_A);

    Matrix<Scalar> B11 = B.subblock({0, 0}, block_B),
            B12 = B.subblock({0, B.cols / 2}, block_B),
            B21 = B.subblock({B.rows / 2, 0}, block_B),
            B22 = B.subblock({B.rows / 2, B.cols / 2}, block_B);

    // create larger matrix for result
    Matrix<Scalar> C = Matrix<Scalar>::zeros(A.rows, B.cols);

    // temporary matrix, here we will store products
    Matrix<Scalar> P;

    // P1 = (A11 + A22) (B11 + B22)
    P = multiply_strassen_dynamic(A11 + A22, B11 + B22);
    // add P1 to C11 and C22
    C.block_add({0, 0}, P);
    C.block_add(product_block, P);

    // P2 = (A21+ A22) B11
    P = multiply_strassen_dynamic(A21 + A22, B11);
    // add P2 to C21 and subtract from C22
    C.block_add({product_block.first, 0}, P);
    C.block_subtract(product_block, P);

    // P3 = A11 (B12 - B22)
    P = multiply_strassen_dynamic(A11, B12 - B22);
    // add P3 to C12 and C22
    C.block_add({0, product_block.second}, P);
    C.block_add(product_block, P);

    // P4 = A22 (-B11 + B21)
    P = multiply_strassen_dynamic(A22, B21 - B11);
    // add P4 to C11 and C21
    C.block_add({0, 0}, P);
    C.block_add({product_block.first, 0}, P);

    // P5 = (A11 + A12) B22
    P = multiply_strassen_dynamic(A11 + A12, B22);
    // subtract P5 from C11 and add it to C21
    C.block_subtract({0, 0}, P);
    C.block_add({0, product_block.second}, P);

    // P6 = (-A11 + A21)(B11 + B12)
    P = multiply_strassen_dynamic(A21 - A11, B11 + B12);
    // add P6 to C22
    C.block_add(product_block, P);

    // P7 = (A12 - A22)(B21 + B22)
    P = multiply_strassen_dynamic(A12 - A22, B21 + B22);
    // add P7 to C11
    C.block_add({0, 0}, P);

    // fix remaining row and column if dimensions are odd
    dynamic_peeling(A, B, C, 2, 2, 2);

    return C;
}

// LADERMAN'S ALGORITHM
template<typename Scalar>
Matrix<Scalar>
multiply_laderman(const Matrix<Scalar> &A, const Matrix<Scalar> &B) {
    // dimension check
    assert(A.cols == B.rows);
```

```
// if any of the dimensions is too small, laderman's algorithm wont help
if (std::min(A.rows, std::min(A.cols, B.cols)) <= laderman_threshold) {
    return multiply_classic(A, B);
}

// subblock sizes
std::pair<unsigned int, unsigned int>
        block_A = {A.rows / 3, A.cols / 3},
        block_B = {B.rows / 3, B.cols / 3},
        product_block = {A.rows / 3, B.cols / 3};
// block dimensions
unsigned int block_rows_A = block_A.first,
        block_cols_A = block_A.second,
        block_rows_B = block_B.first,
        block_cols_B = block_B.second,
        product_rows = product_block.first,
        product_cols = product_block.second;

// split matrices in subblocks
Matrix<Scalar> A11 = A.subblock({0, 0}, block_A),
        A12 = A.subblock({0, block_cols_A}, block_A),
        A13 = A.subblock({0, 2 * block_cols_A}, block_A),
        A21 = A.subblock({block_rows_A, 0}, block_A),
        A22 = A.subblock({block_rows_A, block_cols_A}, block_A),
        A23 = A.subblock({block_rows_A, 2 * block_cols_A}, block_A),
        A31 = A.subblock({2 * block_rows_A, 0}, block_A),
        A32 = A.subblock({2 * block_rows_A, block_cols_A}, block_A),
        A33 = A.subblock({2 * block_rows_A, 2 * block_cols_A}, block_A);

Matrix<Scalar> B11 = B.subblock({0, 0}, block_B),
        B12 = B.subblock({0, block_cols_B}, block_B),
        B13 = B.subblock({0, 2 * block_cols_B}, block_B),
        B21 = B.subblock({block_rows_B, 0}, block_B),
        B22 = B.subblock({block_rows_B, block_cols_B}, block_B),
        B23 = B.subblock({block_rows_B, 2 * block_cols_B}, block_B),
        B31 = B.subblock({2 * block_rows_B, 0}, block_B),
        B32 = B.subblock({2 * block_rows_B, block_cols_B}, block_B),
        B33 = B.subblock({2 * block_rows_B, 2 * block_cols_B}, block_B);

// create larger matrix for result
Matrix<Scalar> C = Matrix<Scalar>::zeros(A.rows, B.cols);

// temporary matrix, here we will store products
Matrix<Scalar> P;

// 23 products

// P1 = (A11 + A12 + A13 - A21 - A22 - A32) B22
P = multiply_laderman(A11 + A12 + A13 - A21 - A22 - A32 - A33, B22);
// add P1 to C12
C.block_add({0, product_cols}, P);

// P2 = (A11 - A21) (B22 - B12)
P = multiply_laderman(A11 - A21, B22 - B12);
// add P2 to C21, C22
C.block_add({product_rows, 0}, P);
C.block_add({product_rows, product_cols}, P);

// P3 = A22 (B12 - B11 + B21 - B22 - B23 - B31 + B33)
P = multiply_laderman(A22, B12 - B11 + B21 - B22 - B23 - B31 + B33);
// add P3 to C21
C.block_add({product_rows, 0}, P);

// P4 = (A21 - A11 + A22) (B11 - B12 + B22)
P = multiply_laderman(A21 - A11 + A22, B11 - B12 + B22);
// add P4 to C12, C21, C22
C.block_add({0, product_cols}, P);
C.block_add({product_rows, 0}, P);
```

```
C.block_add({product_rows, product_cols}, P);

// P5 = (A21 + A22) (B12 - B11)
P = multiply_laderman(A21 + A22, B12 - B11);
// add P5 to C12, C22
C.block_add({0, product_cols}, P);
C.block_add({product_rows, product_cols}, P);

// P6 = A11 B11
P = multiply_laderman(A11, B11);
// add P6 to C11, C12, C13, C21, C22, C31, C33
C.block_add({0, 0}, P);
C.block_add({0, product_cols}, P);
C.block_add({0, 2 * product_cols}, P);
C.block_add({product_rows, 0}, P);
C.block_add({product_rows, product_cols}, P);
C.block_add({2 * product_rows, 0}, P);
C.block_add({2 * product_rows, 2 * product_cols}, P);

// P7 = (A31 - A11 + A32) (B11 - B13 + B23)
P = multiply_laderman(A31 - A11 + A32, B11 - B13 + B23);
// add P7 to C13, C31, C33
C.block_add({0, 2 * product_cols}, P);
C.block_add({2 * product_rows, 0}, P);
C.block_add({2 * product_rows, 2 * product_cols}, P);

// P8 = (A31 - A11) (B13 - B23)
P = multiply_laderman(A31 - A11, B13 - B23);
// add P8 to C31, C33
C.block_add({2 * product_rows, 0}, P);
C.block_add({2 * product_rows, 2 * product_cols}, P);

// P9 = (A31 + A32) (B13 - B23)
P = multiply_laderman(A31 + A32, B13 - B11);
// add P9 to C13, C33
C.block_add({0, 2 * product_cols}, P);
C.block_add({2 * product_rows, 2 * product_cols}, P);

// P10 = (A11 + A12 + A13 - A22 - A23 - A31 - A32) B23
P = multiply_laderman(A11 + A12 + A13 - A22 - A23 - A31 - A32, B23);
// add P10 to C13
C.block_add({0, 2 * product_cols}, P);

// P11 = A32 (B13 - B11 + B21 - B22 - B23 - B31 + B32)
P = multiply_laderman(A32, B13 - B11 + B21 - B22 - B23 - B31 + B32);
// add P11 to C31
C.block_add({2 * product_rows, 0}, P);

// P12 = (A32 - A13 + A33) (B22 + B31 - B32)
P = multiply_laderman(A32 - A13 + A33, B22 + B31 - B32);
// add P12 to C12, C31, C32
C.block_add({0, product_cols}, P);
C.block_add({2 * product_rows, 0}, P);
C.block_add({2 * product_rows, product_cols}, P);

// P13 = (A13 - A33) (B22 - B23)
P = multiply_laderman(A13 - A33, B22 - B32);
// add P13 to C31, C32
C.block_add({2 * product_rows, 0}, P);
C.block_add({2 * product_rows, product_cols}, P);

// P14 = A13 B31
P = multiply_laderman(A13, B31);
// add P14 to C11, C12, C13, C21, C23, C31, C32
C.block_add({0, 0}, P);
C.block_add({0, product_cols}, P);
C.block_add({0, 2 * product_cols}, P);
C.block_add({product_rows, 0}, P);
```

```
    C.block_add({product_rows, 2 * product_cols}, P);
    C.block_add({2 * product_rows, 0}, P);
    C.block_add({2 * product_rows, product_cols}, P);

    // P15 = (A32 + A33) (B32 - B31)
    P = multiply_laderman(A32 + A33, B32 - B31);
    // add P15 to C12, C32
    C.block_add({0, product_cols}, P);
    C.block_add({2 * product_rows, product_cols}, P);

    // P16 = (A22 - A13 + A23) (B23 + B31 - B33)
    P = multiply_laderman(A22 - A13 + A23, B23 + B31 - B33);
    // add P16 to C13, C21, C23
    C.block_add({0, 2 * product_cols}, P);
    C.block_add({product_rows, 0}, P);
    C.block_add({product_rows, 2 * product_cols}, P);

    // P17 = (A13 - A23) (B23 - B33)
    P = multiply_laderman(A13 - A23, B23 - B33);
    // add P17 to C21, C23
    C.block_add({product_rows, 0}, P);
    C.block_add({product_rows, 2 * product_cols}, P);

    // P18 = (A22 + A23) (B33 - B31)
    P = multiply_laderman(A22 + A23, B33 - B31);
    // add P18 to C13, C23
    C.block_add({0, 2 * product_cols}, P);
    C.block_add({product_rows, 2 * product_cols}, P);

    // P19 = A12 B21
    P = multiply_laderman(A12, B21);
    // add P19 to C11
    C.block_add({0, 0}, P);

    // P20 = A23 B32
    P = multiply_laderman(A23, B32);
    // add P20 to C22
    C.block_add({product_rows, product_cols}, P);

    // P21 = A21 B13
    P = multiply_laderman(A21, B13);
    // add P21 to C23
    C.block_add({product_rows, 2 * product_cols}, P);

    // P22 = A31 B12
    P = multiply_laderman(A31, B12);
    // add P22 to C32
    C.block_add({2 * product_rows, product_cols}, P);

    // P23 = A33 B33
    P = multiply_laderman(A33, B33);
    // add P23 to C33
    C.block_add({2 * product_rows, 2 * product_cols}, P);

    // fix remaining row and column if dimensions are odd
    dynamic_peeling(A, B, C, 3, 3, 3);

    return C;
}

// BINI'S ALGORITHM
// Exact multiplication, problem is that multiplication of polynomials is not O(1) anymore.
template<typename Scalar>
Matrix<Scalar>
multiply_bini_exact(const Matrix<Scalar> &A, const Matrix<Scalar> &B) {
    // convert to polynomials
    Matrix<Polynomial<Scalar>> poly_A(A);
    Matrix<Polynomial<Scalar>> poly_B(B);
```

```
    Matrix<Polynomial<Scalar>> poly_C = multiply_bini(poly_A, poly_B,
                                                      Polynomial<Scalar>::epsilon());

    // convert back from polynomials
    return polynomial_to_scalar(poly_C);
}

// actual Bini's algorithm
template<typename Poly>
Matrix<Poly> multiply_bini(const Matrix<Poly> &A, const Matrix<Poly> &B,
                           const Poly &epsilon) {
    // check dimensions
    assert(A.cols == B.rows);

    // if matrices are too small for Bini's algorithm we have nothing to do but multiply it classicaly
    if (A.rows < 2 || A.cols < 2 || B.cols < 3) {
        return multiply_classic(A, B);
    } else if (A.rows <= bini_threshold || A.cols <= bini_threshold ||
               B.cols <= bini_threshold) {
        return multiply_classic(A, B);
    }

    // create subblocks
    // last line and up to 2 last columns may not be included, this is handled by dynamic peeling
    unsigned int block_rows_A = A.rows / 2,
            block_cols_A = A.cols / 2,
            block_rows_B = B.rows / 2,
            block_cols_B = B.cols / 3;

    std::pair<unsigned int, unsigned int>
            block_A = {block_rows_A, block_cols_A},
            block_B = {block_rows_B, block_cols_B};

    // | A11 A12 | | B11 B12 B13 |
    // | A21 A22 | | B21 B22 B23 |
    Matrix<Poly> A11 = A.subblock({0, 0}, block_A),
            A12 = A.subblock({0, block_cols_A}, block_A),
            A21 = A.subblock({block_rows_A, 0}, block_A),
            A22 = A.subblock({block_rows_A, block_cols_A}, block_A);

    Matrix<Poly> B11 = B.subblock({0, 0}, block_B),
            B12 = B.subblock({0, block_cols_B}, block_B),
            B13 = B.subblock({0, 2 * block_cols_B}, block_B),
            B21 = B.subblock({block_rows_B, 0}, block_B),
            B22 = B.subblock({block_rows_B, block_cols_B}, block_B),
            B23 = B.subblock({block_rows_B, 2 * block_cols_B}, block_B);

    // create new empty matrix for product
    // | C11 C12 C13 |
    // | C21 C22 C23 |
    Matrix<Poly> C = Matrix<Poly>::zeros(A.rows, B.cols);

    // dimensions of a block in a product
    unsigned int block_rows_C = block_rows_A, block_cols_C = block_cols_B;

    // matrix for storing products
    Matrix<Poly> P;

    // first half
    // | C11 C12 |
    // | C21 ... |

    // P1 = (A12 + e A22) B21
    P = multiply_bini(A12 + epsilon * A22, B21, epsilon);
    // Add e P1 to C11 and P1 to C21
    C.block_add({0, 0}, epsilon * P);
    C.block_add({block_rows_C, 0}, P);
```

```
// P2 = A11 (B11 + e B12)
P = multiply_bini(A11, B11 + epsilon * B12, epsilon);
// Add e P2 to C11 and P2 to C12
C.block_add({0, 0}, epsilon * P);
C.block_add({0, block_cols_C}, P);

// P3 = A12 (B11 + B21 + e B22)
P = multiply_bini(A12, B11 + B21 + epsilon * B22, epsilon);
// Subtract P3 from C21
C.block_subtract({block_rows_C, 0}, P);

// P4 = (A11 + A12 + e A21) B11
P = multiply_bini(A11 + A12 + epsilon * A21, B11, epsilon);
// Subtract P4 from C12
C.block_subtract({0, block_cols_C}, P);

// P5 = (A12 + e A21) (B11 + e B22)
P = multiply_bini(A12 + epsilon * A21, B11 + epsilon * B22, epsilon);
// Add P5 to C12 and C21
C.block_add({0, block_cols_C}, P);
C.block_add({block_rows_C, 0}, P);

// second half
// | ... C13 |
// | C22 C23 |
// how to get to formulas for P1, ... P5: transpose upper matrix and rename indices.

// transpose all blocks (we won't need B11 and B21, we do not need to transpose them).
A11 = A11.transposed();
A12 = A12.transposed();
A21 = A21.transposed();
A22 = A22.transposed();
B12 = B12.transposed();
B22 = B22.transposed();
B13 = B13.transposed();
B23 = B23.transposed();

// P1 = (B13 + e B12) A21
// After multiplying, transposed back
P = multiply_bini(B13 + epsilon * B12, A21, epsilon).transposed();
// Add e P1 to C23 and P1 to C22
C.block_add({block_rows_C, 2 * block_cols_C}, epsilon * P);
C.block_add({block_rows_C, block_cols_C}, P);

// P2 = B23 (A22 + e A12)
P = multiply_bini(B23, A22 + epsilon * A12, epsilon).transposed();
// Add e P2 to C23 and P2 to C13
C.block_add({block_rows_C, 2 * block_cols_C}, epsilon * P);
C.block_add({0, 2 * block_cols_C}, P);

// P3 = B13 (A22 + A21 + e A11)
P = multiply_bini(B13, A22 + A21 + epsilon * A11, epsilon).transposed();
// Subtract P3 from C22
C.block_subtract({block_rows_C, block_cols_C}, P);

// P4 = (B23 + B13 + e B22) A22
P = multiply_bini(B23 + B13 + epsilon * B22, A22, epsilon).transposed();
// Subtract P4 from C13
C.block_subtract({0, 2 * block_cols_C}, P);

// P5 = (B13 + e B22) (A22 + e A11)
P = multiply_bini(B13 + epsilon * B22, A22 + epsilon * A11,
                  epsilon).transposed();
// Add P5 to C13 and C22
C.block_add({0, 2 * block_cols_C}, P);
C.block_add({block_rows_C, block_cols_C}, P);
```

```
    // using bini's algorithm now we got epsilon * C, now we have to divide by epsilon.
    C /= epsilon;

    // dynamic peeling for not included rows and cols
    dynamic_peeling(A, B, C, 2, 2, 3);

    return C;
}

// SCHONHAGE'S ALGORITHM
// Exact multiplication, problem is that multiplication of polynomials is not O(1) anymore.
template<typename Scalar>
Matrix<Scalar>
multiply_schonhage_exact(const Matrix<Scalar> &A, const Matrix<Scalar> &B) {
    // convert to polynomials
    Matrix<Polynomial<Scalar>> poly_A(A);
    Matrix<Polynomial<Scalar>> poly_B(B);

    Matrix<Polynomial<Scalar>> poly_C = multiply_schonhage(poly_A, poly_B,
                                                           Polynomial<Scalar>::epsilon());

    // convert back from polynomials
    return polynomial_to_scalar(poly_C);
}

// actual Schonhage's algorithm
template<typename Poly>
Matrix<Poly> multiply_schonhage(const Matrix<Poly> &_A, const Matrix<Poly> &_B,
                                const Poly &epsilon) {
    // check dimensions
    assert(_A.cols == _B.rows);

    // if matrices are too small for Bini's algorithm we have nothing to do but multiply it classicaly
    if (_A.rows < 3 || _A.cols < 3 || _B.cols < 3) {
        return multiply_classic(_A, _B);
    } else if (_A.rows <= schonhage_threshold ||
               _A.cols <= schonhage_threshold ||
               _B.cols <= schonhage_threshold) {
        return multiply_classic(_A, _B);
    }

    // create subblocks
    // last line and up to 2 last columns may not be included, this is handled by dynamic peeling
    unsigned int block_rows_A = _A.rows / 3,
            block_cols_A = _A.cols / 3,
            block_rows_B = _B.rows / 3,
            block_cols_B = _B.cols / 3;

    std::pair<unsigned int, unsigned int>
            block_A = {block_rows_A, block_cols_A},
            block_B = {block_rows_B, block_cols_B};

    // | A11 A12 A13 | | B11 B12 B13 |
    // | A21 A22 A23 | | B21 B22 B23 |
    // | A31 A32 A33 | | B31 B32 B33 |

    // matrices in blocks
    std::vector<std::vector<Matrix<Poly>>>
            A(3, std::vector<Matrix<Poly>>(3)),
            B(3, std::vector<Matrix<Poly>>(3)),
            C(3, std::vector<Matrix<Poly>>(3));

    // subtract 1 because indices start at 1
    A[1 - 1][1 - 1] = _A.subblock({0, 0}, block_A),
    A[1 - 1][2 - 1] = _A.subblock({0, block_cols_A}, block_A),
    A[1 - 1][3 - 1] = _A.subblock({0, 2 * block_cols_A}, block_A),
    A[2 - 1][1 - 1] = _A.subblock({block_rows_A, 0}, block_A),
    A[2 - 1][2 - 1] = _A.subblock({block_rows_A, block_cols_A}, block_A),
```

```
A[2 - 1][3 - 1] = _A.subblock({block_rows_A, 2 * block_cols_A}, block_A),
A[3 - 1][1 - 1] = _A.subblock({2 * block_rows_A, 0}, block_A),
A[3 - 1][2 - 1] = _A.subblock({2 * block_rows_A, block_cols_A}, block_A),
A[3 - 1][3 - 1] = _A.subblock({2 * block_rows_A, 2 * block_cols_A},
                              block_A);

B[1 - 1][1 - 1] = _B.subblock({0, 0}, block_B),
B[1 - 1][2 - 1] = _B.subblock({0, block_cols_B}, block_B),
B[1 - 1][3 - 1] = _B.subblock({0, 2 * block_cols_B}, block_B),
B[2 - 1][1 - 1] = _B.subblock({block_rows_B, 0}, block_B),
B[2 - 1][2 - 1] = _B.subblock({block_rows_B, block_cols_B}, block_B),
B[2 - 1][3 - 1] = _B.subblock({block_rows_B, 2 * block_cols_B}, block_B),
B[3 - 1][1 - 1] = _B.subblock({2 * block_rows_B, 0}, block_B),
B[3 - 1][2 - 1] = _B.subblock({2 * block_rows_B, block_cols_B}, block_B),
B[3 - 1][3 - 1] = _B.subblock({2 * block_rows_B, 2 * block_cols_B},
                              block_B);

// create new empty matrix for product
// | C11 C12 C13 |
// | C21 C22 C23 |
// | C31 C32 C33 |
for (unsigned int i = 0; i < 3; i++) {
    for (unsigned int j = 0; j < 3; j++) {
        // fill with zeros
        C[i][j] = Matrix<Poly>::zeros(block_rows_A, block_cols_B);
    }
}

// epsilon squared
Poly epsilon2 = epsilon * epsilon;

// actual algorithm,
// see article Partial and Total Matrix Multiplication, example 2.2 for formulas
for (unsigned int i = 0; i < 3; i++) {
    // Wi
    Matrix<Poly> W = multiply_schonhage(
            A[i][0],
            B[1][i] + B[2][i],
            epsilon
    );

    // D'ji from article, we are calculating C
    // Cji = 1/epsilon^2 (Uij + Vij - Wi) + 1/epsilon (Vji - Vjj)
    // Cii = 1/epsilon^2 (Uii + Vii - Wi)
    // because we do not want to keep all matrices in memory,
    // this calculation will be split in multiple parts
    for (unsigned int j = 0; j < 3; j++) {
        Matrix<Poly> U, V;

        if (j == i) {
            // Uii
            U = multiply_schonhage(
                    A[i][0] + epsilon2 * A[i][1],
                    epsilon2 * B[0][i] + B[1][i],
                    epsilon
            );
            // Vii
            V = multiply_schonhage(
                    A[i][0] + epsilon2 * A[i][2],
                    B[2][i],
                    epsilon
            );

            Matrix<Poly> to_subtract = V;
            to_subtract /= epsilon;
            // subtract Vjj from all Cji (where i != j)
            for (unsigned int k = 0; k < 3; k++) {
                if (k != j) {
```

```
                    C[j][k] -= to_subtract;
                }
            }

        } else {
            // Uij
            U = multiply_schonhage(
                    A[i][0] + epsilon2 * A[j][1],
                    B[1][i] - epsilon * B[0][j],
                    epsilon
            );
            // Vij
            V = multiply_schonhage(
                    A[i][0] + epsilon2 * A[j][2],
                    B[2][i] + epsilon * B[0][j],
                    epsilon
            );
            // 1/epsilon Vji
            Matrix<Poly> to_add = V;
            to_add /= epsilon;
            C[i][j] += to_add;
        }

        // 1/epsilon^2 (Uij + Vij - Wi)
        // (or if i == j (Uii + Vii - Wi)
        Matrix<Poly> to_add = U + V - W;
        to_add /= epsilon2;
        C[j][i] += to_add;
    }
}

// join block from matrix C into a single block
Matrix<Poly> _C = Matrix<Poly>::zeros(_A.rows, _B.cols);

for (unsigned int i = 0; i < 3; i++) {
    for (unsigned int j = 0; j < 3; j++) {
        _C.block_add({block_rows_A * i, block_cols_B * j}, C[i][j]);
    }
}

// dynamic peeling for not included rows and cols
dynamic_peeling(_A, _B, _C, 3, 3, 3);

return _C;
}
```

Slovar strokovnih izrazov

border rang mejni rang
concise zgoščen
dynamic padding dinamično zapolnjevanje
dynamic peeling dinamično luščenje
static padding statično zapolnjevanje
tensor slice rezina tenzorja

Literatura

[1] A. Benson in G. Ballard, *A Framework for Practical Parallel Fast Matrix Multiplication*, ACM SIGPLAN Notices **50** (2015) 42–53; dostopno tudi na `arxiv.org/pdf/1409.2908.pdf`.

[2] M. Bläser, *Complexity of bilinear problems*, verzija 7. 10. 2009 [ogled 12. 5. 2020], dostopno na `www-cc.cs.uni-saarland.de/media/oldmaterial/bc.pdf`.

[3] D. Bini, M. Capovani, G. Lotti in F. Romani, $O(n^{2.7799})$ *complexity for* $n \times n$ *approximate matrix multiplication*, Information Processing Lett. **8** (1979) 234–235.

[4] P. Bürgisser, M. Clausen in M. A. Shokrollahi, *Algebraic complexity theory*, Grundlehren der mathematischen Wissenschaften **315**, Springer-Verlag, Berlin, 1996.
[5] H. Cohn, R. Kleinberg, B. Szegedy, C. Umans, *Group-theoretic algorithms for matrix multiplication*, v: Proceedings of the 46th Annual IEEE Symposium on Foundations of Computer Science, IEEE, Pittsburgh, Pennsylvania, ZDA, 2005, str. 379–388; dostopno tudi na `arxiv.org/pdf/math/0511460.pdf`.
[6] J. Laderman, *A noncommutative algorithm for multiplying* 3×3 *matrices using 23 multiplications*, Bull. Amer. Math. Soc. **82** (1976) 126–128.
[7] S. Lederman, E. Jacobson, J. Johnson, A. Tsao in T. Turnbull, *Implementation of Strassen's Algorithm for Matrix Multiplication*, v: Supercomputing '96: Proceedings of the 1996 ACM/IEEE Conference on Supercomputing, Pittsburgh, Pennsylvania, ZDA, 1996.
[8] S. Lederman, E. Jacobson, J. Johnson, A. Tsao in T. Turnbull, *Strassen's algorithm for matrix multiplication: Modeling, analysis, and implementation*, v: Supercomputing '96: Proceedings of the 1996 ACM/IEEE conference on Supercomputing, Pittsburgh, Pennsylvania, ZDA, 1996, str. 32–58.
[9] M. Marinko, *Fast Matrix Multiplication*, [7. 9. 2020], dostopno na `github.com/matejm/fast-matrix-multiplication`.
[10] A. Schönhage, *Partial and total matrix multiplication*, SIAM J. Comput. **10** (1981) 434–455.
[11] A. V. Smirnov, *The bilinear complexity and practical algorithms for matrix multiplication*, Comput. Math. Math. Phys. **53** (2013) 1781–1795; dostopno tudi na `cs.uwaterloo.ca/~eschost/Exam/Smirnov.pdf`.
[12] A. J. Stothers, *On the complexity of matrix multiplication*, doktorsko delo, University of Edinburgh, 2010; dostopno na `era.ed.ac.uk/bitstream/handle/1842/4734/Stothers2010.pdf`.
[13] V. Strassen, *Gaussian elimination is not optimal*, Numer. Math. **13** (1969) 354–356.
[14] F. Romani, *Some properties of disjoint sums of tensors related to matrix multiplication*, SIAM J. Comput. **11** (1982) 263–267.
[15] S. Yaroslav, *A counterexample to Strassen's direct sum conjecture*, Acta Math. **222** (2019) 363–379; dostopno tudi na `arxiv.org/pdf/1712.08660.pdf`.
[16] S. Winograd *On multiplication of* 2×2 *matrices*, Lin. Alg. Appl. **4** (1971) 381–388.
[17] *Matrix multiplication*, v: Wikipedia, The Free Encyclopedia, [ogled 18. 7. 2020], dostopno na `en.wikipedia.org/wiki/Matrix_multiplication`.
[18] *Galactic algorithm*, v: Wikipedia, The Free Encyclopedia, [ogled 1. 8. 2020], dostopno na `en.wikipedia.org/wiki/Galactic_algorithm`.