Lågrangmatriskomplettering: En jämförelse av två algoritmer

Low-Rank Matrix Completion: a Comparison between two Algorithms

Examensarbete för kandidatexamen i matematik vid Göteborgs universitet
Kandidatarbete inom civilingenjörsutbildningen vid Chalmers

Erik Andersson
Douglas Fernstedt
Ellen Gustafsson

Institutionen för Matematiska vetenskaper
CHALMERS TEKNISKA HÖGSKOLA
GÖTEBORGS UNIVERSITET
Göteborg, Sverige 2025

Lågrangmatriskomplettering: En jämförelse av två algoritmer

Kandidatarbete i matematik inom civilingenjörsprogrammet Teknisk matematik vid Chalmers
Erik Andersson

Examensarbete för kandidatexamen i matematik vid Göteborgs universitet
Douglas Fernstedt

Examensarbete för kandidatexamen i matematik inom Matematikprogrammet vid Göteborgs universitet
Ellen Gustafsson

Handledare: Andrii Dmytryshyn Matematiska vetenskaper

Institutionen för Matematiska vetenskaper
CHALMERS TEKNISKA HÖGSKOLA
GÖTEBORGS UNIVERSITET
Göteborg, Sverige 2025

Förord

I detta arbete har vi undersökt två algoritmer för lågrangmatriskomplettering. Vår frågeställning har tagits fram med hjälp av vår handledare Andrii Dmytryshyn som även har varit med oss hela vägen genom projektet och väglett oss. Vi vill tacka honom för hans engagemang och givande insikter.

Under arbetets gång har vi strävat efter att alla gruppmedlemmar ska medverka lika mycket och att alla ska prova på alla olika delar. Alla delar av rapporten har setts över av samtliga gruppmedlemmar genom omgångar av revidering och återkoppling. De huvudsakliga författarna för varje del nämns i tabell 1 nedan. Vi har haft veckovisa möten med gruppen för att planera nästföljande steg och diskutera idéer. Varje vecka har vi fört loggbok över vad varje person har bidragit med och vad som bestämts under träffen.

Avsnitt	**Huvudförfattare**
Förord	Ellen
Populärvetenskaplig presentation	Ellen
Sammandrag och abstract	Erik
Inledning	
Bakgrund	Douglas
Syfte och frågeställningar	Alla
Avgränsningar	Douglas
Samhälleliga och etiska aspekter	Erik
Teoretisk bakgrund	
Optimeringsteori	Douglas
Lågrangproblemet	Douglas
Matrisdekomposition	Douglas, Erik
LRMC-algoritmer	Ellen
Metod	
Databaser	Douglas
Utvärdering av algoritmerna	Douglas
Resultat	
RMSE över tid för olika δ och μ	Ellen
RMSE och rang över τ	Erik
RMSE över r NIHT	Erik
SVD-metoder på slumpgenererade matriser	Douglas
Netflix-databas	Douglas
Diskussion och slutsats	Alla
Referenser	Alla
Appendix 1 - Teori	
Numeriska SVD-metoder	Douglas, Erik
Appendix 2 - Källkod	
NIHT-algoritm	Ellen
SVT-algoritm	Erik
Numeriska SVD-metoder	Douglas
Datagenerering & Behandling av Netflix-databasen	Erik
Tester	Alla

Tabell 1: Bidragsrapport

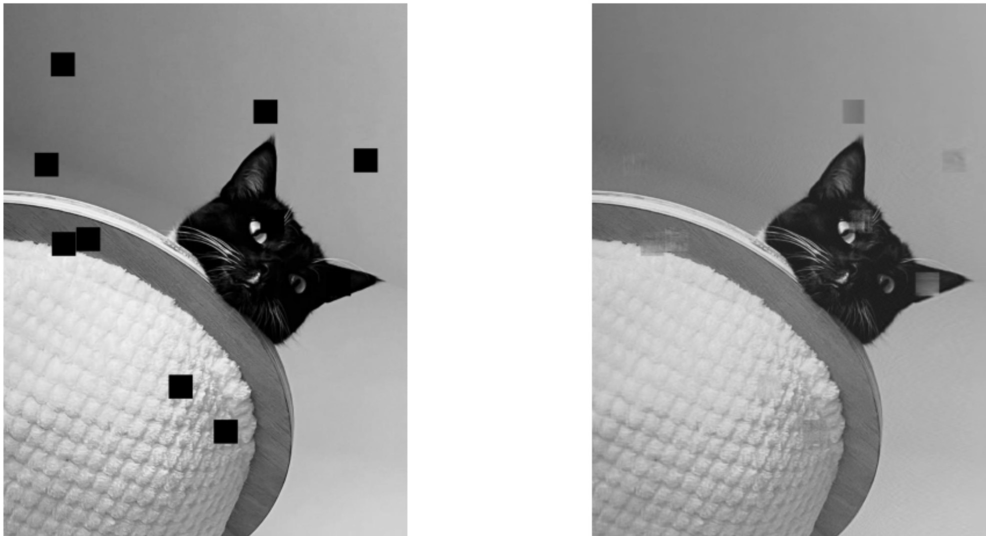
Populärvetenskaplig presentation

Lågrangmatriskomplettering är ett intressant problem, både inom akademisk forskning och industrin. Det handlar om att fylla i saknade värden i stora och ofullständiga datamängder. Användningen av lågrangmatriskomplettering kan illustreras med en av de mest kända tillämpningarna, det så kallade Netflix-problemet. Film- och TV-serie leverantören Netflix har som mål att rekommendera filmer till användare baserat på vilka filmer som de tidigare har tyckt om. Vi kan ställa upp detta i en tabell, där raderna representerar användare, kolumnerna filmer och de ifyllda värdena betyg som användaren har satt.

	Film 1	Film 2	Film 3	Film 4
Användare 1	5		4	
Användare 2		3	2	
Användare 3	4	1		5
Användare 4	5		4	

Netflix vill utifrån de givna betygen i tabellen förutse vad de saknade betygen ska vara, så att de kan rekommendera filmer som användaren sannolikt kommer att gilla. Med massvis av användare och filmer kan detta verka omöjligt, men i och med att det finns underliggande mönster i hur användare lägger sina betyg kan vi lösa problemet. En individuell användares betyg kan antagligen förklaras av ett begränsat antal preferenser, som en favoritgenre eller skådespelare.

Matriskomplettering har många andra tillämpningar inom forskning och datavetenskap. Det kan till exempel handla om att fylla i saknade pixlar i bildbehandling, rekonstruera brusiga signaler i signalbehandling och som tidigare nämnt algoritmer för rekommendationer. Bilden nedan visar hur lågrangmatriskompletterings-algoritmer kan användas för att fylla i inkompletta bilder. Den högra bilden nedan saknar vissa stycken och till höger har dessa stycken fyllts i.



Figur 1: Bild med stycken borttagna till vänster. Återskapad bild till höger.
Bild använd med tillåtelse från fotograf Selma Velagic.

Lågrangmatriskomplettering har också en roll i maskininlärning. Djupa neurala nätverk, som utgör grunden för moderna AI-system, kan integrera matriskompletterings metoder för att förfina träningsprestanda och datakvalitet [1]. Genom att fylla i saknade värden i träningsdata kan dessa metoder bidra till mer robusta modeller.

Forskning inom detta område är även spännande ur en akademisk synvinkel då det förenar linjär algebra, optimering och sannolikhetslära. Problemet är både utmanande och praktiskt relevant, då det ställer höga krav på algoritmutveckling och beräkningsoptimering för att kunna skalas upp till större datamängder. Därav är detta ett område med stor potential för både teoretiska framsteg och verkliga tillämpningar inom datavetenskap och artificiell intelligens.

I rapporten har två olika algoritmer för lågrangmatriskomplettering undersökts. Vi har implementerat dessa två algoritmer i Python. Undersökningen kommer att ske på en verklig databas samt en slumpmässigt generad datamängd. Vi har sedan jämfört algoritmernas prestation i träffsäkerhet och beräkningstid. Målet med arbetet är att hitta styrkor och svagheter i algoritmerna, samt undersöka hur algoritmerna skiljer sig åt med avseende på olika datamängder. Rapporten kommer fram till att en algoritm var bättre än den andra i våra tester.

Sammandrag

Lågrangmatriskomplettering innefattar algoritmer som fyller ut saknade värden i en matris under antagandet att den kompletta matrisen är av låg rang. Rapporten har undersökt två olika algoritmer för långrangmatriskomplettering, *singular value thresholding* (SVT) och *normalized iterative hard thresholding* (NIHT), på slumpmässigt genererad data och ett urval av databasen *Netflix prize data*. Rapportens syfte är att bestämma vilken av dessa två algoritmer som lämpar sig bättre för komplettering av Netflix-datan och slumpmässigt genererad data.

För att mäta detta undersöktes hur nära algoritmerna konvergerar till de kompletta matriserna i termer av bland annat RMSE samt hur lång tid det tar för de olika algoritmerna att köra givet olika parameterval. Eftersom både NIHT och SVT använder sig av *singulärvärdesdekomposition* som steg i algoritmen undersöktes även hur olika numeriska metoder för att beräkna dessa påverkar precisionen och tiden det tar att köra algoritmerna.

Rapporten visade att SVT var snabbare och gav högre precision än NIHT när det kommer till att komplettera Netflix-databasen. Däremot visar NIHT god precision att komplettera slumpmässigt genererad data och kan även göra det snabbare än SVT om en tillräckligt god uppskattning av rangen anges i förväg. Testerna visade även att NIHT kan ge bättre resultat om vissa parametrar i algoritmen justeras, vilket kan vara av intresse för vidare forskning.

Nyckelord - Lågrangmatriskomplettering, normalized iterative hard thresholding, singular value thresholding, singulärvärdesdekomposition

Abstract

Low-rank matrix completion encapsulates algorithms that assign values to missing entries in matrices under the assumption that the rank of the complete matrices should be low. In the thesis, two algorithms for low-rank matrix completion were examined, namely *singular value thresholding* (SVT) and *normalized iterative hard thresholding* (NIHT). They were tested on randomly generated matrices as well as a selection of entries from the *Netflix prize data* database. The purpose of the thesis is to determine which algorithm is better suited for completing the Netflix database as well as randomly generated data.

To measure this, the algorithms were compared in terms of how well they converge to the desired result. This was then measured using RMSE, among other metrics, and compared in terms of how long the runtime for each algorithm was. The algorithm's individual runtimes and precision values were also measured against their parameters. Since both NIHT and SVT make use of the *singular value decomposition* in each iteration of the algorithm, the effect on the runtime and precision from different numerical methods for evaluating the singular value decomposition was measured.

The results show that SVT performed better when it came to both runtime and precision than NIHT when applied to the Netflix prize data. However, NIHT showed high precision in completing the randomly generated data and can even do so faster than SVT if the rank can be estimated accurately. The tests also showed that NIHT can yield better results if certain parameters in the algorithm are adjusted, which may be of interest in future research endeavours.

Keywords - Low-rank matrix completion, Normalized iterative hard thresholding, Singular value thresholding, Singular value decomposition

Innehåll

1 Inledning **1**
1.1 Bakgrund . . . 1
1.2 Syfte och frågeställningar . . . 1
1.3 Avgränsningar . . . 2
1.4 Samhälleliga och etiska aspekter . . . 2
1.5 Förkortningar . . . 3

2 Teoretisk bakgrund **3**
2.1 Optimeringsteori . . . 3
2.2 Lågrangproblemet . . . 4
2.3 Matrisdekomposition . . . 4
2.3.1 Singulärvärdesdekomposition . . . 4
2.3.2 SVD-metoder . . . 5
2.4 LRMC-algoritmer . . . 5
2.4.1 Singulärvärdesnormer . . . 6
2.4.2 Singular value thresholding . . . 6
2.4.3 Normalized iterative hard thresholding . . . 7

3 Metod **9**
3.1 Databaser . . . 9
3.1.1 Slumpmässiga matriser . . . 9
3.1.2 Netflix . . . 10
3.2 Utvärdering av algoritmerna . . . 10

4 Resultat **10**
4.1 Parametrar . . . 10
4.1.1 RMSE över tid för olika δ och μ . . . 10
4.1.2 RMSE och rang över τ . . . 11
4.1.3 RMSE över r NIHT . . . 12
4.2 SVD-metoder på slumpgenererade matriser . . . 13
4.2.1 SVT . . . 13
4.2.2 NIHT . . . 15
4.3 Netflix-databasen . . . 17
4.3.1 SVT . . . 17
4.3.2 NIHT . . . 17

5 Diskussion och slutsats **20**

6 AI användning **22**

A Appendix 1 – Teori **i**
A.1 Numeriska SVD-metoder . . . i
A.1.1 QR faktorisering . . . ii
A.1.2 Randomized SVD (RSVD) . . . ii

B Appendix 2 – Källkod **iv**
B.1 LRMC-algoritmer . . . iv
B.1.1 NIHT-algoritm . . . iv
B.1.2 SVT-algoritm . . . iv
B.2 Numeriska SVD-metoder . . . vi
B.3 Datagenerering . . . vii
B.4 Behandling av Netflix-databasen . . . ix
B.5 Tester . . . ix
B.5.1 RMSE över tid för olika δ och μ . . . ix
B.5.2 RMSE och rang vs tau SVT . . . xiii

B.5.3 RMSE vs r NIHT xv
B.5.4 SVT för olika SVD-metoder på slumpgenererade matriser xviii
B.5.5 NIHT för olika SVD-metoder på slumpgenererade matriser xxvii
B.5.6 SVD-metoder på Netflix-databasen xxxvii

1 Inledning

1.1 Bakgrund

Lågrangmatriskomplettering, som i resten av artikeln kommer benämnas med förkortningen LRMC från engelska översättningen *low-rank matrix completion*, är ett problem som är välstuderat och mycket aktuellt i dagens samhälle med flera olika tillämpningar inom rekommendationssystem, bildåterskapning och maskininlärning [2]. Syftet med kompletteringen är att fylla ut en datamängd som har saknade element med hjälp av värden som är observerade. Matriser används för att strukturera och organisera datamängden. Det är omöjligt att veta vad de okända elementen exakt ska anta för värde, problemet blir då obestämt om det inte finns mer information om matrisstrukturen [2].

I många fall kan rangen av en matris antas vara låg. Till exempel är det möjligt att anta att Netflix-databas matrisen är av låg rang, eftersom det är sannolikt att endast ett fåtal faktorer som påverkar vilka typer av serier eller filmer som är intressanta. LRMC kan därför användas för att komplettera Netflix-datbasen i syfte att skapa ett rekommendationssystem för filmer. Ett annat exempel på tillämpning är att återskapa eller identifiera geometriska objekt från sekvenser av bilder, vilket är speciellt användbart inom datorseende. De enskilda bilderna kan innehålla delar av objektet men om bilderna överlappas är det möjligt att forma en matris med lågrangstruktur. Genom att anta att alla värden, även de okända, härstammar ifrån en matris med lågrang förenklar problemet och lösningen verkar vara den rätta [2].

Tidigare forskning har huvudsakligen delat upp problemet på två olika sätt: konvexa metoder och icke-konvexa metoder. De konvexa metoderna bygger ofta på *nukleärnormsminimering* (NNM), som innebär att minimera matrisens singulära värden utifrån de observerade värdena. Rapporten kommer att undersöka en populär konvex algoritm som kallas *singular value thresholding* (SVT). SVT innehar starka teoretiska garantier för att konvergera till den optimala lösningen av NNM problemet under rätt förutsättningar [2]. Icke-konvexa metoder försöker istället lösa LRMC problemet genom att optimera direkt på matrisens rang. Metoden minimerar inte på någon norm utan istället extraherar en matris approximation av en viss rang. Rapporten kommer att undersöka en algoritm som kallas *normalized iterative hard thresholding* (NIHT). Icke-konvexa metoder har inte samma teoretiska garantier för att minimera rangen hos en matris. Dock kan resultatet av LRMC vara samma eller bättre som en konvex metod [3].

Rapporten kommer att använda sig av *singulärvärdesdekomposition* (SVD) för både NIHT och SVT. Detta är den vanligaste matrisdekompositionsmetoden och används för att extrahera information ur en matris. Forskning har också utforskat möjligheten att utnyttja slumpmässiga matrisdekompositioner för LRMC. Detta bygger på att approximera den ursprungliga matrisen med en ny mindre matris som är enklare att räkna på. Detta sätt har visat sig vara väldigt effektivt för att approximera stora matriser med hög kvalité. Rapporten kommer att använda sig av slumpmässig SVD (RSVD) och utforska detta ytligt [4].

1.2 Syfte och frågeställningar

Syftet med arbetet är att jämföra resultat och prestanda för SVT och NIHT. Jämförelsen kommer beakta hur olika SVD och parametrar påverkar approximationskvalité och körtid. Specifikt kommer följande frågeställningar att beaktas:

1. Hur påverkar olika SVD-metoder kvaliteten och körningstiden av LRMC-algoritmerna?
2. Hur skiljer sig prestationen mellan SVT och NIHT när det gäller precision, matrisstorlek och konvergenshastighet?
3. Hur påverkar de valda LRMC-algoritmernas parametrar och matrisstorlek dess resultat i termer av precision och konvergenshastighet?

1.3 Avgränsningar

Datamängderna som används kommer vara rimligt stor för att körningar inte ska ta för lång tid. Undersökningen består endast av två LRMC algoritmer som nämnts ovan. Rapporten kommer att fokusera på numeriska simuleringar och tester utan att bevisa satser formellt. Det kommer inte att genomföras någon teoretisk konvergens- eller komplexitetsanalys. Endast empiriska resultat kommer redovisas, teoretisk analys finns i [2] och [3].

1.4 Samhälleliga och etiska aspekter

Företag och andra organisationer har tillgång till en stor mängd data som ofta används för att kartlägga och analysera olika användare. Att kunna kartlägga stora grupper av användare genom att hitta gemensamma beteenden är mycket attraktivt för att till exempel skapa rekommendationssystem eller rikta marknadsföring. LRMC kan användas för dessa syften.

I rapporten behandlar vi en riktig datamängd, specifikt Netflix-databasen. Denna databas består av betyg för filmrecensioner från ett antal användare och för ett antal filmer. Den gavs ut som en del av tävlingen *Netflix prize* som handlade om att göra en mer robust rekommendationsalgoritm för Netflix. Detta anser vi är nödvändigt för att skapa en välformad slutsats om vilken algoritm som lämpar sig bättre för att komplettera data, då det kan finnas mönster i riktig data som slumpmässigt genererad data inte visar.

Netflix-databasen är dock kopplad till vissa etiska problem. Trots att alla användare i databasen är anonyma går det att koppla vissa användare till riktiga personer med hjälp av andra offentliga datamängder, till exempel *Internet Movie Database* (IMDb). Även om alla filmrecensioner på IMDb är offentliga betyder det inte att alla betyg i Netflix-databasen är det. Detta innebär att vissa betygsättningar som folk vill hålla hemliga kan och har kopplats till individerna och gjorts offentliga utan deras samtycke. Av denna anledning blev Netflix stämda under anklagelsen att filmrecensioner är persondata, men processen avslutades med att Netflix gick med på att stänga ned sin andra tävling [5]. Ändå finns Netflix-databasen publicerad offentligt på internet idag [6].

På grund av detta har det beslutats att ingen data från Netflix-databasen ska skrivas i rapporten. Det enda rapporten kommer utforska är hur väl SVT och NIHT kan komplettera Netflix-databasen och det enda som kommer rapporteras är hur väl de lyckas. Det går att argumentera för att det går att identifiera ännu fler användare om man kompletterar databasen, därför ser vi till att inte heller rapportera den kompletterade datan, utan enbart RMSE eller liknande värden för att mäta hur hög precision algoritmerna har.

Slutligen bör LRMC algoritmer inte användas för att komplettera värden som kan ha signifikant påverkan för människors liv, som till exempel medicinsk data. Detta är eftersom LRMC är byggt på att förenkla datamängder till att vara beroende på få parametrar, vilket kan förstärka existerande bias i databasen.

1.5 Förkortningar

Relevanta förkortningar för rapporten listas nedan och förklaras löpande i texten.

- *LRMC* - Lågrangmatriskomplettering
- *SVT* - Singular Value Thresholding
- *NIHT* - Normalized Iterative Hard Thresholding
- *SVD* - Singulärvärdesdekomposition
- *NNM* - Nukleärnormsminimering
- *GK* - Golub-Kahan SVD
- *GK-DC* - Golub-Kahan Divide-and-counquer SVD
- *SVDS* - Implicitly Restarted Arnoldi Method SVD
- *RSVD* - Randomized SVD
- *RMSE* - Root mean square error

2 Teoretisk bakgrund

2.1 Optimeringsteori

LRMC problemet är definierat som ett matematiskt optimeringsproblem. Optimeringsproblemet kan formuleras enligt [7, s. 127] som

$$\begin{array}{lll} \text{minimera} & f_0(x) & \\ \text{under bivillkor} & f_i(x) \leq b_i, & i = 1, \dots, m, \\ & h_i(x) = 0, & i = 1, \dots, p. \end{array} \tag{2.1}$$

där $f_0(x) : \mathbb{R}^n \to \mathbb{R}$ är funktionen vi vill minimera genom att finna $x = (x_1, \dots, x_n)$ som ger det lägsta värdet och $f_i(x) : \mathbb{R}^n \to \mathbb{R}$ är bivillkoren med $b_1, \dots, b_i$ som är begränsande konstanta värden för villkoren. Bivillkoren kan tolkas som begränsningar som medför att antal lösningar minskas. Lösningen till minimeringen tolkas som det ”billigaste” alternativet av alla möjliga lösningar som möter bivillkoren [7, s. 127]. Funktionen $h_i(x) = 0$ kallar vi en likhetsbivillkor och är endast ett bivillkor med likhetstecken.

Ett icke-konvext optimeringsproblem är ett problem som inte är konvext. Det följer naturligt att definiera det konvexa problemet. Ett optimeringsproblem (2.1) är konvext enligt [7, s. 136-137] om:

$$f_j(\alpha x + \beta y) \leq \alpha f_j(x) + \beta f_j(y) \tag{2.2}$$

för $j = 0, \dots, m$ där $x, y \in \mathbb{R}^n$ och $\alpha, \beta \in \mathbb{R}$ med $\alpha + \beta = 1 : \alpha, \beta \geq 0$.

Ekvation (2.2) är definitionen av en konvex funktion. Det betyder att funktionen vi minimerar och bivillkoren måste vara konvexa funktioner för att hela problemet ska vara konvext. I [7, s. 137] nämns ytterligare ett krav om vi använder oss av ett likhetsvillkor $h_i(x) = 0$, att denna funktion måste vara affin. En affin funktion definieras som en funktion som bevarar raka linjer och parallellism. Rapporten kommer använda sig av likhetsvillkoret att de kända värdena i datamängderna ska bevaras. Villkoret är trivialt affint eftersom vi återanvänder kända värden utan att applicera någon transformation.

2.2 Lågrangproblemet

Det finns några grundläggande teoretiska begrepp som är viktiga att komma ihåg. En matris M är ett sätt att representera information genom rader och kolonner. Vi noterar en matris $M \in \mathbb{R}^{m\times n}$ vilket innebär att den har m rader, n kolonner och är reell. Observerade element noteras M_{ij} för $(i,j) \in \Omega$, där Ω är positionerna av alla observerade värden. Rangen av en matris noteras av $\text{rang}(M)$. Det är ett heltal som beskriver antalet linjärt oberoende rader eller kolonner. Notera att rad- och kolumnrangen är alltid samma [8, s. 23]. En lågrangmatris är en matris där $\text{rang}(M) \ll \min(m,n)$, vilket innebär att rangen av matrisen är betydligt mindre än det minsta av rad eller kolonn dimensionen.

Det övergripande LRMC problemet kan definieras enligt [2], [3] på följande sätt. Lått $X \in \mathbb{R}^{m\times n}$ och lös

$$\min_X \text{rang}(X) \quad \text{under bivillkoret} \quad X_{ij} = M_{ij} \text{ för alla } (i,j) \in \Omega \tag{2.3}$$

där målet är att fylla i de saknade elementen i X. Vi låter X först anta de värden vi vet från M och sedan handlar problemet om att fylla i resten. Problemet är NP-svårt [2], vilket innebär att det inte finns någon känd lösning som kan uttryckas i polynom tid. Det beror delvis på att rangen är ett diskret mått. Att minimera rangen direkt hade inneburit att utforska alla möjliga kombinationer av rader och kolumner, vilket är ett kombinatoriskt problem [9, s. 4]. Problem (2.3) är icke-konvex. Vi kan visa det med ett motexempel.

Motexempel: Antag först att rangfunktionen är konvex och uppfyller ekvation (2.2). Tag M_1, M_2 som två godtyckliga matriser med respektive rang r_1, r_2. Då får vi:

$$\text{rang}(\alpha M_1 + \beta M_2) \leq \alpha\text{rang}(M_1) + \beta\text{rang}(M_2).$$

Tag nu följande matriser:

$$M_1 = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}, \quad M_2 = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}.$$

Vi ser att $\text{rang}(M_1) = \text{rang}(M_2) = 1$. Välj sedan $\alpha = \beta = 0.5$ och olikheten blir nu:

$$\text{rang}(0.5 \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} + 0.5 \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}) \leq 0.5 \cdot 1 + 0.5 \cdot 1 = 1 \Longleftrightarrow \text{rang} \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix} = 2 \nleq 1.$$

Eftersom rang funktion inte uppfyller (2.2) för alla matriser och konstanter så är den icke konvex. ■

En viktig egenskap hos ett konvext problem är att någon lokal optimal punkt också är en global optimal punkt [7, s. 138]. Rangproblemet i (2.3) saknar i sin grundform garantier för att finna minimumet. Det är fortfarande möjligt att använda sig direkt av den icke-konvexa formuleringen, vilket NIHT gör. SVT omformulerar problemet till ett konvext optimeringsproblem istället.

2.3 Matrisdekomposition

Matrisdekompositioner används för att extrahera information ifrån en matris. Specifikt kommer vi vara intresserade av matrisens singulära värden med tillhörande singulära vektorer. Algoritmerna kommer att använda sig utav olika typer av SVD-metoder.

2.3.1 Singulärvärdesdekomposition

En SVD kan appliceras på alla reella och komplexa matriser $M \in \mathbb{R}^{m\times n}$ [8, Kap. 7]:

$$M = U\Sigma V^T \tag{2.4}$$

- U: $m \times m$ ortogonal matris där kolonnerna utgör en ortonormal bas för kolonnrummet till MM^T. Kolonnvektorer kallas även vänster singulära vektorer till M.

- Σ: $m \times n$ diagonal matris. Elementen är icke-negativa reella tal som kallas de singulära värdena till M. Mer specifikt är de singulära värdena roten ur egenvärdena till $M^T M$ och MM^T. Dessa värden är ordnade från vänster till höger i fallande ordning sådant att $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_r > 0 \quad$ där $\text{rang}(M) = r$.

- V: $n \times n$ ortogonal matris där kolonnerna utgör en ortonormal bas över radrummet till $M^T M$. Kolonnvektorerna kallas även höger singulära vektorer till M.

I projektet kommer en alternativ definition av SVD att användas som kallas ”kompakt singulärvärdesdekomposition”. Den kompakta versionen väljer istället ut de r första höger- och vänstersingulära vektorerna tillsammans med de r första singulära värdena.

$$M = U_r \Sigma_r V_r^T$$

- U_r: $m \times r$ matris som består av de första r kolonnerna av U.

- Σ_r: $r \times r$ matris definierad enligt $\Sigma_r = \text{diag}(\sigma_1, \sigma_2, ..., \sigma_r)$.

- V_r: $r \times n$ matris som består av de första r kolonnerna av V.

Det resulterar i en mer minneseffektiv SVD som rapporten kommer att använda i nästan alla SVD beräkningar. Det är också möjligt att skriva om SVD enligt [8, s. 289]:

$$M = U\Sigma V^T = U_r \Sigma_r V_r^T = u_1 \sigma_1 v_1^T + \ldots + u_r \sigma_r v_r^T, \tag{2.5}$$

där u_r, v_r är vänster- respektive högersingulära vektorer med tillhörande singulärvärde σ_r. Varje term ökar rangen av matrisen med ett och tillför mer information om strukturen av matrisen. Därmed är vi intresserade av de första singulära värdena eftersom de har störst värde för summan i (2.5) [8, s. 302]. LRMC vill reducera rangen i matrisen, behålla de kända värdena och komplettera de okända elementen.

Ett viktigt resultat för SVD är Eckart-Mirsky-Young satsen som finns beskriven i [8, s. 302]. Om en SVD beräknas på en matris $M = U\Sigma V^T$ så är den bästa rang-k approximationen $M_k = U_k \Sigma_k V_k^T = u_1 \sigma_1 v_1^T + \ldots + u_k \sigma_k v_k^T$, med hänsyn till alla normer som använder sig av singulära värden. Matematiskt kan det formuleras

$$\|M - M_k\| \leq \|M - B\| \tag{2.6}$$

där $\|\cdot\|$ är någon singulärvärdesnorm och B är godtycklig matris med $\text{rang}(B) \leq k$. Resultatet medför att vi vet att den närmaste approximerade matrisen av M, med rang-k, är M_k.

2.3.2 SVD-metoder

Det största beräkningssteget i SVD är att beräkna egenvärden och egenvektorer. Beräkning av egenvärden kan göras för små matriser genom att lösa dess karaktäristiska ekvation och sedan ta fram egenvektorer. Detta blir extremt ineffektivt när storleken ökar. Rapporten kommer att använda sig av metoderna *Golub-Kahan* (GK), *Golub-Kahan Divide-and-conquer* (GK-CD), *implicitly restarted Arnoldi method* (IRAM eller SVDS) och *randomized SVD* (RSVD). Detaljerna och implemeteringen av ovannämnda algoritmerna är utanför omfattningen av denna rapport, men kortfattad information kan hittas i appendix kapitel A.1.

2.4 LRMC-algoritmer

Denna rapport kommer att utforska två algoritmer för LRMC. En löser en konvex omformulering av originalproblemet medan den andra löser ett icke-konvext problem.

2.4.1 Singulärvärdesnormer

Rapporten kommer att behandla tre former av *singulärvärdesnormer*, det vill säga matrisnormer som enbart använder sig av dess singulära värden. De tre normerna är *Frobeniusnormen*, *nukleärnormen* och *2-normen*. Frobeniusnormen av en godtycklig matris X kan beräknas på flera sätt men kan definieras som

$$\|X\|_F = \sqrt{\sum_{i=1}^{r} \sigma_i^2}$$

där σ_i noterar matrisens singulära värden. 2-normen är definierad som $\|X\|_2 = \max(\sigma_1, ..., \sigma_r)$ och nukleärnormen är definierad som $\|X\|_* = \sum_{i=1}^{r} \sigma_i$ [8, s. 302].

2.4.2 Singular value thresholding

Singular value thresholding är en metod för LRMC som optimerar det konvexa NNM-problemet [9]. För en gles matris $M \in \mathbb{R}^{m \times n}$ definieras problemet:

$$\begin{aligned} &\text{minimera} && \|X\|_* \\ &\text{under bivillkor} && X_{ij} = M_{ij}, \quad (i,j) \in \Omega. \end{aligned} \tag{2.7}$$

Vi kan byta ut det grundläggande lågrangproblemet (2.3) med denna formulering, eftersom problemen har samma unika lösning under vissa förutsättningar [2].

Innan algoritmen presenteras behöver ett centralt steg introduceras. Steget benämns som *singular value shrinkage operator* [2]. För en godtycklig matris X och $\tau \geq 0$ definieras singular value thresholding operatorn:

$$\begin{aligned} &S_\tau(X) := U S_\tau(\Sigma) V^T \\ &S_\tau(\Sigma) = \mathrm{diag}(\max(0, \sigma_i - \tau)), \quad i = 1, \ldots, r. \end{aligned}$$

Operatorn verkar på specifikt på matrisens singulära värdena $\sigma_i \in \Sigma$ genom att beräkna en SVD enligt (2.4). De singulära värdena är alltid positiva och i fallande ordning. Parametern τ väljs till ett fast värde och indikerar hur mycket varje σ_i ska krympas. Denna typ av operator kallas också *thresholding operator*, specifikt är det *soft thresholding* eftersom att operatorn krymper de singulära värdena mot noll istället för strikt sätta ett förbestämt antal singulära värden till noll [2].

I vår implementation av SVT använder vi oss också av en maskeringsoperator för att kunna välja ut och jämföra de kända värdena. Vi definierar operatorn enligt

$$\mathcal{P}(X) = \Pi \odot X, \quad \Pi_{ij} = \begin{cases} 1, & \text{om } (i,j) \in \Omega \\ 0, & \text{annars} \end{cases} \tag{2.8}$$

för en godtycklig matris X där $\odot$ noterar Hadamardprodukten, det vill säga elementvis multiplicering av matriser. Utifrån optimeringsproblemet (2.7) formuleras SVT-algoritmen utifrån följande induktiva algoritm [2]

$$\begin{cases} X_k = S_\tau(Y_{k-1}), \\ Y_k = Y_{k-1} + \delta \cdot P(M - X_k). \end{cases}$$

Algoritmen börjar med en matris Y_0, utför soft thresholding, adderar differensen mellan resultatet och den inmatade matriser viktat med steglängden δ. Den repeterar processen tills differensen mellan resultatet X_k och den inmatade matrisen är tillräckligt liten. Den fullständiga algoritmen är som följande:

Algoritm 1 Singular value thresholding

Indata: Observerad matris $M \in \mathbb{R}^{m \times n}$, maskeringsmatris Π, steglängd δ, parameter τ, max antal iterationer max_iter, ökning ℓ och tolerans ϵ.
Utdata: Approximerad matris X_k.
1: Låt $k_0 = \lceil \tau / (\delta \|\mathcal{P}(M)\|_2) \rceil$
2: Låt $Y_0 = k_0 \delta \mathcal{P}(M)$
3: Sätt $r_0 = 0$, $r_{\max} = \min(m, n)$ och $k = 1$
Upprepa:
4: Sätt $s_k = \min(r_{k-1} + 1, r_{max} - 1)$
5: Beräkna de s_k största singulärvärdena Y_{k-1}
6: Sätt σ_{s_k} till det minsta singulärvärdet
Medan: $\sigma_{s_k} > \tau$ och $s_k < r_{max}$
7: Beräkna de s_k största singulärvärdena Y_{k-1}
8: Sätt σ_{s_k} till det minsta singulärvärdet
9: Låt $s_k = s_k + \ell$
10: Utför Soft Thresholding $X_k = S_\tau(Y_{k-1})$ fram till singulärvärde $\max(s_k - \ell, 1)$
11: Sätt $r_k = \max\{j : \sigma_j > \tau\}$
12: Sätt $Y = Y + \delta \cdot \mathcal{P}(M - X_k)$
13: Sätt $k = k + 1$
Tills: $k >$ max_iter eller $\frac{\|\mathcal{P}(X_k - M)\|_F}{\|\mathcal{P}(M)\|_F} < \epsilon$
Returnera: X_k

Algoritmen tar in en gles matris M och en maskeringsmatris Π där de kända värdena i Π representeras med ettor och de okända med nollor. Ökningen ℓ ger oss möjligheten att hoppa över iterationer av den inre loopen för en snabbare beräkningstid. Snabbstarten k_0 medför att Y_0 inte startar från en nollmatris vilket snabbar på de initiala iterationerna. Steglängden δ bestämmer storleken på uppdateringen av matrisen i steg 12, därigenom är δ avgörande i konvergensen av algoritmen. Sats 4.2 i [2] anger att med $0 < \delta < 2$ är SVT garanterad att konvergera. I [2] används:

$$\delta = 1.2 \frac{m \cdot n}{|\Omega|}. \tag{2.9}$$

Detta kan användas för en matris $M \in \mathbb{R}^{m \times n}$ med $|\Omega|$ stycken kända världen. Notera att detta värde kan överstiga 2, men det motiveras heuristiskt i [2] att algoritmen konvergerar ändå.

Steg 7-9 i algoritm 1 som väljer rangen som ska användas i soft thresholding-steget kan väljas bort beroende på vilken matrisdekompositionsmetod som används. I SVDS och RSVD väljs antalet singulärvärden som beräknas vilket gör loopen i steg 7-9 nödvändig, däremot beräknar GK och GK-DC alla singulära värden vilket gör loopen överflödig.

En nackdel med SVT-algoritmen är att relativt många parametrar behöver bestämmas vilket kräver extra arbete för att justera i implementeringar. Dock kräver SVT inte en ranguppskattning som indata.

2.4.3 Normalized iterative hard thresholding

Istället för att optimera över den konvexa formuleringen av lågrangproblemet kan vi välja att direkt optimera över det icke-konvexa problemet (2.3). Metoderna som gör detta använder sig vanligtvis av en *hard thresholding operator* [3], som definieras enligt

$$H_r(X) = U \Sigma_r V^T, \quad \text{där} \quad \Sigma_r(i,i) := \begin{cases} \Sigma(i,i) & \text{if } i \leq r \\ 0 & \text{if } i > r. \end{cases} \quad \text{och} \quad X = U \Sigma V^T.$$

Hard thresholding behåller de r största singulära värdena till skillnad från metoden soft thresholding som används i SVT-algoritmen där vi jämnar ut värdena med en konstant τ.

Normalized iterative hard thresholding är en matriskompletteringsmetod som använder sig av hard thresholding för att lösa ett icke-konvext problem. Metoden bygger på *iterative hard thresholding* (IHT) som är en algoritm inom signalbehandling som sedan har modifierats till användning inom LRMC [3]. I signalbehandling används den linjära avbildningen $\mathcal{A}(\cdot)$ för att välja ut observerade element. Algoritmen som rapporten använder sig av är baserad på algoritmen i [3] men har modifierats från att använda $\mathcal{A}(\cdot)$ operatorn till att använda $\mathcal{P}(\cdot)$ operatorn som definierat i (2.8) för att bättre passa till rapportens syfte.

NIHT utgår ifrån följande induktiva algoritm:

$$X_{k+1} = H_r(X_k + \mu \cdot \mathcal{P}(M - X_k)).$$

Algoritmen börjar med en matris X_0. Differensen mellan X_k och den inmatade matrisen adderas till X_k viktat med steglängd μ, sedan appliceras hard thresholding operatorn på resultatet. Processen repeteras tills differensen mellan X_k och den inmatade matrisen är tillräckligt liten.

Algoritm 2 Normalized iterative hard thresholding

Indata: Observerad matris $M \in \mathbb{R}^{m\times n}$, maskeringsmatris Π, rang r, max antal iterationer max_iter, tolerans ϵ.
Utdata: Approximerad matris X_k.
1: Utför hard thresholding $X_0 = H_r(\mathcal{P}(M))$
2: Låt U_0 var de r största singulära vektorerna av X_0
3: Sätt $k = 1$
Upprepa:
4: Beräkna steglängd:
$\mu = \frac{\|U_k U_k^T \mathcal{P}(X_k - M)\|_F^2}{\|\mathcal{P}(U_k U_k^T \mathcal{P}(X_k - M))\|_F^2}$.
5: Uppdatera matrisen:
$X_{k+1} = H_r(X_k + \mu \cdot \mathcal{P}(M - X_k))$.
6: Låt U_k var de r största singulära vektorerna av X_{k+1}
7: Sätt $k = k + 1$
Tills: $k >$ max_iter eller $\frac{\|\mathcal{P}(M - X_{k+1})\|_F}{\|\mathcal{P}(M)\|_F} < \epsilon$
Returnera: X_k

Skillnaden mellan IHT och NIHT är hur steglängden μ sätts. I IHT sätts μ till någon konstant, medan NIHT beräknar μ i varje iteration baserad på hur nära den approximerade matrisen är till M. Med denna modifikation kan vi garantera konvergens givet att vi har tillräckligt med observerade element i matrisen M [10].

NIHT kräver, till skillnad från SVT, en uppskattning av matrisrangen r. Rangen är av stor betydelse för kvaliteten för den ifyllda matrisen. Det finns metoder för att hitta den optimala rangen r som baseras på *restricted isometry constants* (RIC). Vi definierar RIC med en modifierad version av definitionen i [3]:

Definition 2.1 (Restricted isometry constants)**.** Låt $\mathcal{P}(\cdot)$ vara en linjär avbildning $\mathbb{R}^{m\times n} \to \mathbb{R}^{m\times n}$, som definierad i (2.8). För varje heltal $1 \leq r \leq \min(m, n)$ gäller att restricted isometry constant, R_r av $\mathcal{A}(\cdot)$ definieras som det minsta tal så att

$$(1 - R_r)\|X\|_F^2 \leq \|\mathcal{P}(X)\|_F^2 \leq (1 + R_r)\|X\|_F^2$$

håller för godtycklig matris X med $\text{rang}(X) \leq r$.

Ett lågt RIC innebär att matrisens norm är välbevarad under operatorn $\mathcal{P}(\cdot)$. För en matris M med $|\Omega| = p$ och rang r, kan man genom IHT återskapa matriser för vilka $\mathcal{P}(\cdot)$ har $R_{2r} \leq 1/3$ och $\mu = 1/(1 + R_{2r})$ [3]. NIHT är bevisat att kunna återskapa samma matris med $R_{3r} < 1/5$ och adaptiv steglängd [3]. Därav har NIHT större garantier av konvergens än IHT och är ett bättre val av algoritm.

Utöver steglängdsekvationen

$$\mu_1 = \frac{\|U_i U_i^T \mathcal{P}(X_i - M)\|_F^2}{\|\mathcal{P}(U_i U_i^T \mathcal{P}(X_i - M))\|_F^2},$$

som används i algoritm 3 nämner [3] två andra sätt att beräkna μ

$$\mu_2 = \frac{\|\mathcal{P}(X_i - M)V_i V_i^T\|_F^2}{\|\mathcal{P}(\mathcal{P}(X_i - M)V_i V_i^T)\|_F^2},$$

$$\mu_3 = \frac{\|U_i U_i^T \mathcal{P}(X_i - M)V_i V_i^T\|_F^2}{\|\mathcal{P}(U_i U_i^T \mathcal{P}(X_i - M)V_i V_i^T)\|_F^2}.$$

Vilken av dessa ekvationer som ger bäst resultat i form av beräkningstid och precision varierar beroende på formen av matrisen M. Enligt heuristiska tester i [3] är μ_1 och μ_2 likvärdiga för kvadratiska matriser men μ_3 är bättre än de andra två för rektangulära matriser. RIC testet för konvergens fungerar för samtliga steglängdsberäkningar.

3 Metod

Som tidigare nämnt kommer rapporten jämföra SVT mot NIHT. Jämförelsen kommer att använda sig av slumpgenererade matriser och en riktig datamängd.

3.1 Databaser

De slumpgenererande datamängderna kommer att ha ”snälla” egenskaper för att ge bild över hur algoritmerna presterar under kontrollerade förutsättningar. Den riktiga databasen kommer att vara ett mer verkligt sätt att se skillnader.

3.1.1 Slumpmässiga matriser

De slumpmässiga matriserna kommer att vara kvadratiska och genereras med en bestämd rang som är hälften av sidlängden. Matriserna kommer att först genereras med värden ifrån en kontinuerlig uniform fördelning på intervallet $[0, 1]$. Sedan kommer varje element i matrisen att standardiseras med z-poäng. Låt x_{ij} vara ett element i en slumpmässigt genererad kvadratisk matris $M \in \mathbb{R}^{n \times n}$.

$$z_i = \frac{x_{ij} - \mu}{\sigma}, \quad \text{där} \quad \mu = \frac{1}{n^2}\sum_{i,j} x_{ij} \quad \text{och} \quad \sigma = \sqrt{\frac{1}{n^2}\sum_{i,j}(x_{ij} - \mu)^2}.$$

Efter standardiseringen kommer matrisens medelvärde vara 0 och standardavvikelse 1. Det teoretiska medelvärdet och standardavvikelse för en uniform fördelning är $\mu = (1-0)/2 = 0.5, \quad \sigma = (1-0)/\sqrt{12} = 1/2\sqrt{3}$. Varje element i matrisen har ett värde som följer:

$$z_i = \frac{x_i - 0.5}{1/2\sqrt{3}} = 2\sqrt{3}(x_i - 0.5) \quad \text{där} \quad x_i \in [0, 1] \implies z_i \in [-\sqrt{3}, \sqrt{3}].$$

Teoretiskt kommer matrisens element att anta värden mellan $[-\sqrt{3}, \sqrt{3}]$. Att begränsa värdena i matrisen medför att vi minskar risken för att extremvärden påverkar resultatet. Att begränsa matrisens element minskar även algoritmernas körningstid.

Efter standardisering utförs en SVD på matrisen och ett antal singulära värden sätts till noll för att bestämma rangen. Detta gör att värdena inte längre är garanterade att vara inom intervallet ovan, men de kommer inte vara mycket annorlunda enligt (2.6), eftersom den resulterande matrisen kommer vara den bästa lågrangapproximationen till den begränsade matrisen.

Rapporten kommer att utforska dessa slumpmässiga matriser i tre olika storlekar: 10×10, 100×100 och 1000×1000. Efter matriserna genereras tas vissa värden bort uniformt slumpmässigt. Rapporten kommer att använda flera olika slumpgenererade matriser för samma tester och titta på spridning eller beräkna medelvärde av samtliga matriser.

3.1.2 Netflix

Den ursprungliga Netflix-databasen innehåller betygsättningar ifrån 480189 stycken användare på 17700 olika filmer [6]. Den är för stor för att rapportens tester ska kunna utföras inom en rimlig tidsram. För att konstruera databasen, som kommer att användas i rapporten, gjordes ett urval av raderna. Mer specifikt valdes alla betygsättningar från 100 av användarna som recenserat flest filmer. Efter urvalet hade matrisen 28% saknade element, vilket är ganska lite. Därmed tog vi slumpmässigt bort fler element från denna datamängd. Det resulterade i en matris med cirka 64% saknade element, som användes för att testa NIHT och SVT.

3.2 Utvärdering av algoritmerna

För de slumpmässiga matriserna vet vi vad de kända värdena faktiskt är, eftersom de genereras utan saknade värden från början. Då är det möjligt att jämföra den approximerade matrisen med en känd originalmatris. Detta tillvägagångssätt kommer också att tillämpas på Netflix databasen. Rapporten kommer att använda RMSE för att göra detta. Vi kommer också att mäta hur både Netflix-databasen och de slumpmässigt genererade matriserna konvergerar mot de observerade värdena. Vi kommer att kalla detta för felet.

$$\text{Fel} = \frac{\|\mathcal{P}(M - X_k)\|_F}{\|\mathcal{P}(M)\|_F}, \quad \text{där} \quad \text{RMSE} = \sqrt{\frac{\sum |M_{ij} - X_{ij}|^2}{mn}}.$$

M är originalmatrisen i fallet med slumpmässiga matriser. För Netflix-databasen är M matrisen med endast 28% saknade värden. X_k är resultatet av den valda algoritmen. Felet är samma formel som vi använder för toleransen i algoritmerna. RMSE beräknas på hela matrisen och ger ett ungefärligt mått på avståndet mellan den approximerade matrisen och den ursprungliga. Testerna kommer att mäta fel och RMSE över tid.

Eftersom att NIHT behöver en specifik rang inmatning kommer tester på NIHT att köras över flera olika rang. Varje rang kommer att kräva ett visst antal iterationer. Tiden för hela NIHT körning kommer vara en ackumulerad tid av alla förgående ranginmatningar. Detta behöver inte SVT ta hänsyn till.

4 Resultat

Python kommer att användas för att implementera algoritmerna och att producera grafer. Speciellt kommer funktioner ifrån Scipy, Numpy och Matplotlib vara relevanta. Funktionerna kommer att använda beräkningsmetoder ifrån LAPACK [11] och ARPACK [12] för olika SVD-metoder.

4.1 Parametrar

Rapporten börjar med att undersöka olika parameterval på algoritmerna genom att använda slumpgenererade matriser.

4.1.1 RMSE över tid för olika δ och μ

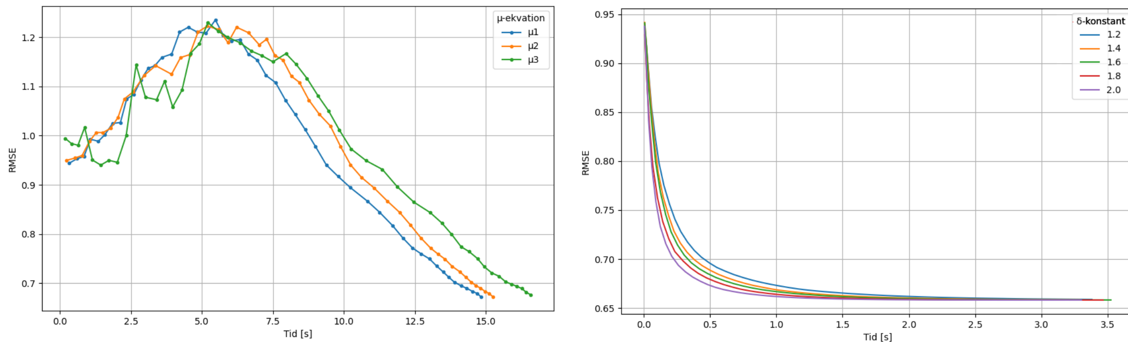
I figur 2 och 3 visas hur RMSE ändras över tid i respektive algoritm för olika μ respektive δ. Testerna visar medelvärdena för 10 stycken slumpmässiga 100×100 matriser som definierade i kapitel 3.1.1. Varje punkt i figur 2 visar när en rang är klar.

Testerna jämför hur olika värden på δ i SVT och μ NIHT presterar. De tre olika sätt att beräkna steglängden μ som beskrivs i kapitel 2.4.3 beräknas i figur 2 och värden på konstanten i (2.9) varierar mellan 1 och 2 i figur 3. SVT beräknas med $\tau = 30$ och $\ell = 5$.

Figur 2 visar att steglängd μ_3 ger större RMSE än de två andra steglängderna vilket stämmer överens med resultaten i [3]. Ekvation μ_1 har lite bättre resultat än de andra, vilket motiverar

valet av detta μ i algoritmen.

Figur 3 visar på snarlika resultat för δ-konstanterna. De högre värdena på δ går ner i RMSE aningen snabbare vilket är väntat då δ i algoritmen avgör storleken på förändringen av matrisen i varje iteration. Viktigt att notera i dessa tester är att resultaten av algoritmen kan variera beroende på form, antal kända element, storlek på värden och fördelningen av matrisens element.



Figur 2: (NIHT) RMSE vs. tid för 10st 100×100 matriser enligt kapitel 3.1.1, för 3 varianter av μ

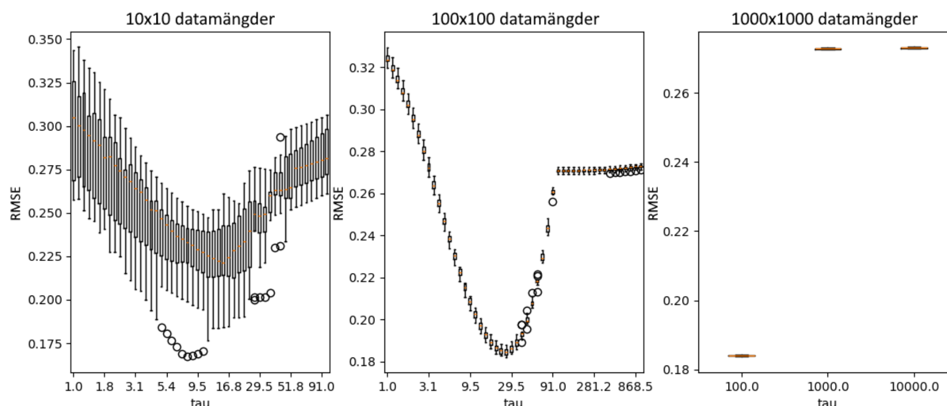
Figur 3: (SVT) RMSE vs. tid för 10st 100×100 matriser enligt kapitel 3.1.1, för 5 värden av δ

4.1.2 RMSE och rang över τ

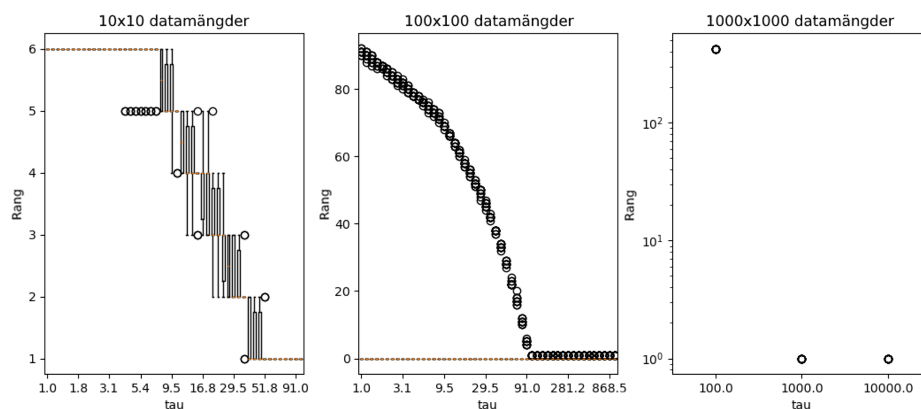
I figur 4 och figur 5 visas hur det valda värdet av τ påverkar rangen av resultatet av SVT-algoritmen, samt den totala RMSE:n mellan resultatet och originaldatan.

Alla tre delfigurer representerar resultat för SVT applicerat på 10 oberoende slumpgenererade matriser enligt metoden i kapitel 3.1.1. Alla 10 matriser i varje storleksordning har rang $r = n/2$ där n är antal kolonner eller rader i matrisen. 40% av elementen i varje matris togs bort och därefter kördes SVT-algoritmen på varje matris. De resulterande matriserna från alla 10 körningar av algoritmen jämfördes mot de kompletta originalmatriserna för att beräkna RMSE:n. Därefter upprepades testet för olika värden på τ. För varje storleksordning och för varje τ illustrerades alla 10 oberoende värden av RMSE i ett lådagram för att visa både medelvärdet och spridningen av RMSE:n.

Förutom RMSE:n av resultaten från SVT-algoritmen illustrerades även rangen av resultatet i ett lådagram på samma sätt. Varje körning av SVT-algoritmen kördes med SVDS-algoritmen för att beräkna singulära värden, 100 max-iterationer, ökning $l = 5$ och tolerans 10^{-2}.



Figur 4: (SVT) RMSE vs. τ för 10 slumpmässigt genererade matriser enligt kapitel 3.1.1

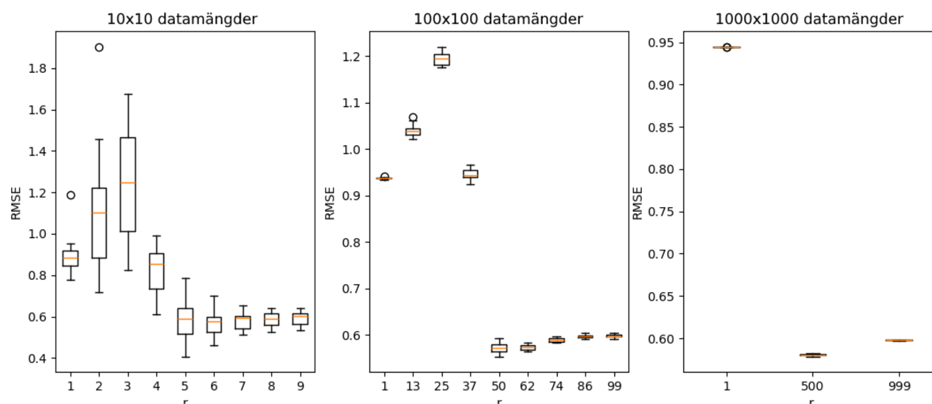


Figur 5: (SVT) Rang vs. τ för 10 slumpmässigt genererade matriser enligt kapitel 3.1.1

Figur 4 visar en trend i de två första storleksordningarna att RMSE:n minimeras kring $\tau = 3\sqrt{n}$. Figur 5 visar även att rangen av de kompletterade matriserna blir nära den sanna rangen för samma värde av τ. I den tredje storleksordningen producerades mycket färre datapunkter på grund av beräkningstiden för SVT-algoritmen på stora matriser. Bilden visar ett minsta värde för RMSE vid $\tau = 100$, vilket är nära $\tau = 3\sqrt{1000} \approx 94,87$, men det är inte säkert att ett annat mindre värde inte kan hittas för ett annat valt intervall av värden på τ. Figur 5 visar tydligt att rangen på de kompletterade matriserna avtar med ökande τ. Figur 4 visar också att spridningen av RMSE:n avtar med storleken på matrisen.

4.1.3 RMSE över r NIHT

I figur 6 visas hur valet av den uppskattade rangen r påverkar RMSE:n för NIHT algoritmen. På samma sätt som i 4 och 5 konstruerades 10 oberoende slumpmässigt genererade matriser i tre olika storleksordningar, med 50% rang och 40% av elementen borttagna. RMSE:n av de NIHT-kompletterade matriserna illustrerades som ett lådagram över alla 10 datamängder för olika värden på r och för de tre storleksordningarna.



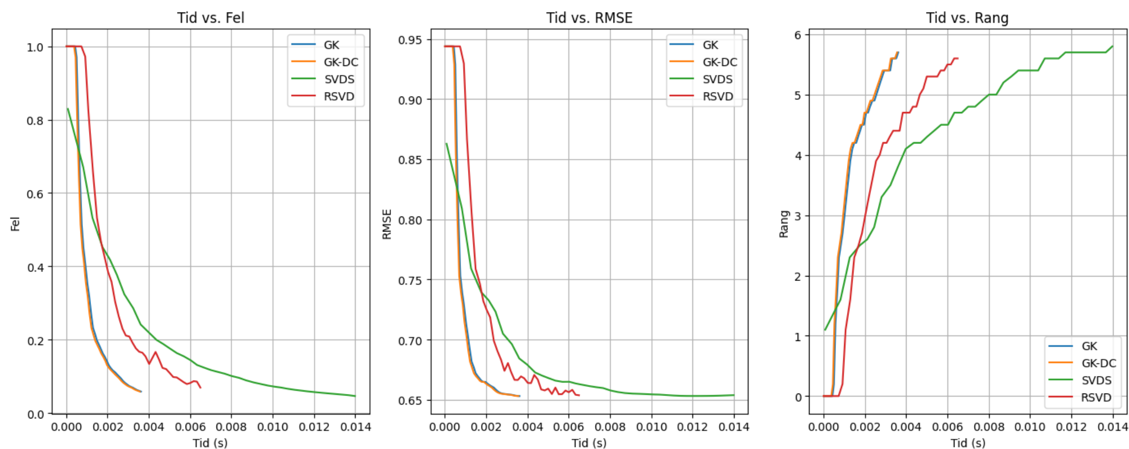
Figur 6: (NIHT) RMSE vs. r för 10 slumpmässigt genererade matriser enligt kapitel 3.1.1

Figuren visar att RMSE:n är beroende på det valda värdet av r. Det resultat som är bäst hittas då r är samma som den sanna rangen. Spridningen i RMSE mellan datamängderna minskar även med r. Slutligen visar figuren att RMSE:n blir högre om rangen underskattas än om den överskattas. Dock ökar RMSE när r överstiger den faktiska rangen, vilket är ett resultat av överanpassning.

4.2 SVD-metoder på slumpgenererade matriser

Olika SVD-metoder kommer användas för SVT och NIHT på slumpgenererade matriser. För NIHT kommer endast μ_1 att användas. Testerna för SVT och NIHT kommer använda samma parametrar för att få en bra jämförelse, om inget annat nämns. RSVD är den enda metoden som använder sig av exponentiering och översampling.

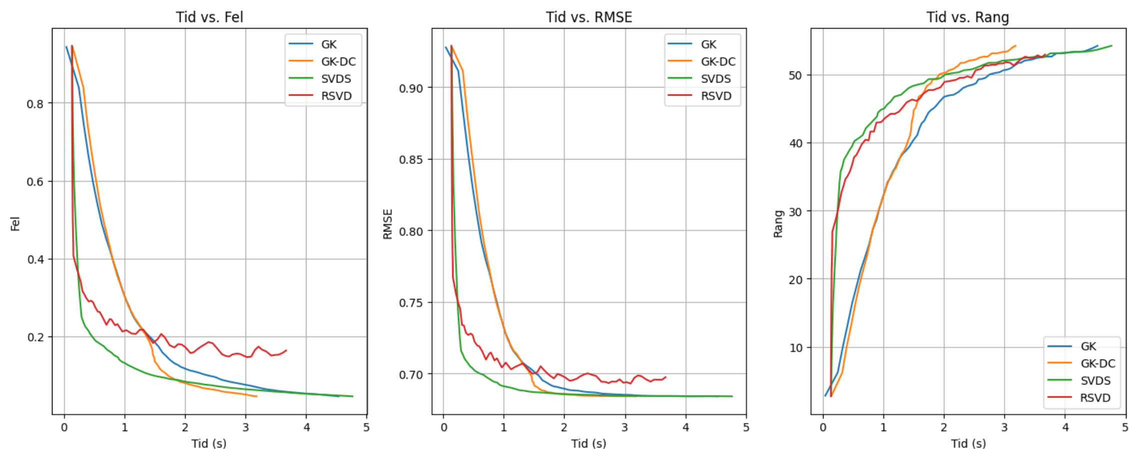
4.2.1 SVT



Figur 7: (SVT) Medelvärde av resultatet för 10 slumpmässigt genererade 10×10 matriser med rang $= 5$ och 50% borttagna element. Snabbstart användes enbart för SVDS metoden. RSVD använder exponentiering $= 1$ och översampling $= 2$.

Samtliga metoder använder sig av $\delta = 0.6n^2/m$, $\tau = 3n$, en tolerans på 10^{-4} och 40 max iterationer. För SVDS och RSVD användes $\ell = 1$. Det är tydligt att GK och GK-DC presterar bättre än de övriga, prestationen är också identisk. Över algoritmens körtid sjunker felet och RMSE snabbare och stabilare gentemot RSVD och SVDS. GK och GK-DC hittar också rangen av matrisen väldigt fort. Det är tydligt att RSVD och SVDS inte fungerar tillräckligt bra på rapportens 10×10.

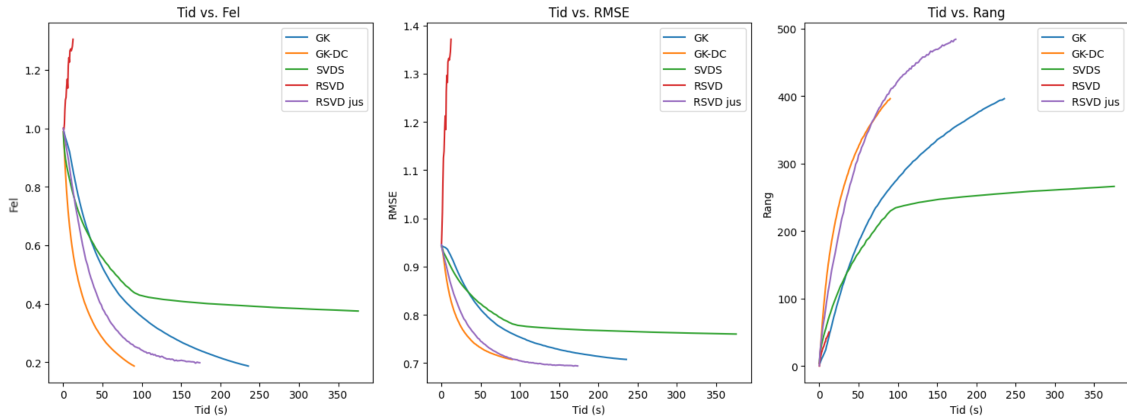
Notera att SVDS behöver använda en snabbstart för SVT-algoritmen. Eftersom att SVT utan snabbstart börjar med en nollmatris. Ett Krylovdelrum i (A.1) kan inte skapas på en slumpmässig noll-vektor.



Figur 8: (SVT) Medelvärde av resultatet för 10 slumpmässigt genererade 100×100 matriser med rang $= 50$ och 50% borttagna element. RSVD använder exponentiering $= 4$, översampling $= 10$.

Samtliga metoder använder $\delta = 1.0n^2/m$, $\tau = 3n$, tolerans på 10^{-4} och 70 max-iterationer. För RSVD och SVDS användes $\ell = 5$. För alla grafer noterar vi att SVDS och RSVD har en väldigt snabb konvergenshastighet i början av iterationerna. SVDS mynnar sedan ut stabilt medans RSVD konvergerar, men till ett högre fel och RMSE. Detta gäller även fast högre exponentiering och översampling har använts.

En annan notering är att GK och GK-DC är väldigt lika för dessa matriser också. Metoderna, förutom RSVD, konvergerar mot samma RMSE i slutet av iterationerna. GK-DC har avvikande punkt vid sekund 1.5 där den avviker från GK:s linje och beräknar felet och RMSE snabbare. Detta kan förklaras ifrån graf tre, där GK-DC beräknar fler singulära värden och genom detta konvergerar snabbare.



Parameter	GK	GK-DC	SVDS	RSVD	RSVD (justerad)
Delta	$1.2 \times n^2/m$	$1.2 \times n^2/m$	$1.2 \times n^2/m$	$1.2 \times n^2/m$	$0.1 \times n^2/m$
Tolerans	10^{-4}	10^{-4}	10^{-4}	10^{-4}	10^{-4}
Tau	$5 \times n$	$5 \times n$	$5 \times n$	$5 \times n$	$0.2 \times n$
Max iterationer	100	100	45	25	100
Ökning	-	-	5	5	5
Snabbstart	Ja	Ja	Ja	Ja	Ja
Exponentiering	-	-	-	4	4
Översampling	-	-	-	10	10

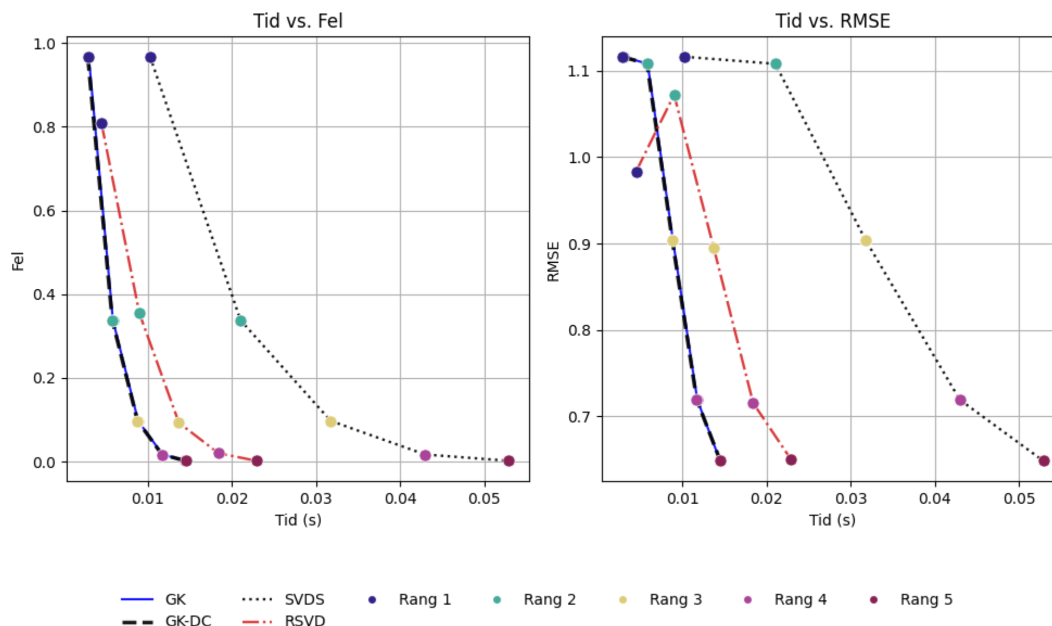
Figur 9: (SVT) Medelvärde av resultatet för 3 slumpmässigt genererade 1000×1000 matriser med rang $= 500$ och 50% borttagna element.

Det är tydligt att RSVD inte hanterade samma parametrar som de andra metoderna väl. Den började divergera tidigt och max-iterationerna sänktes. Figuren visar även en implementation av RSVD med justerade parametrar som fungerar mycket bättre. Detta illustrerar att vid justeringar av parametrarna är det möjligt att få ett bättre resultat.

SVDS har endast 45 iterationer i detta test. Det är eftersom beräkningstiden ökar markant när vi ökar rang inmatningen i metoden. Om SVDS hade fått 100 iterationer, som de andra metoderna, hade algoritmens körtid varit extremt mycket större gentemot de andra, men den hade nått ungefär samma resultat. Detta beror på att storleken av Krylovdelrummet, definierade i (A.1), blir väldigt stort och beräkningarna tar lång tid.

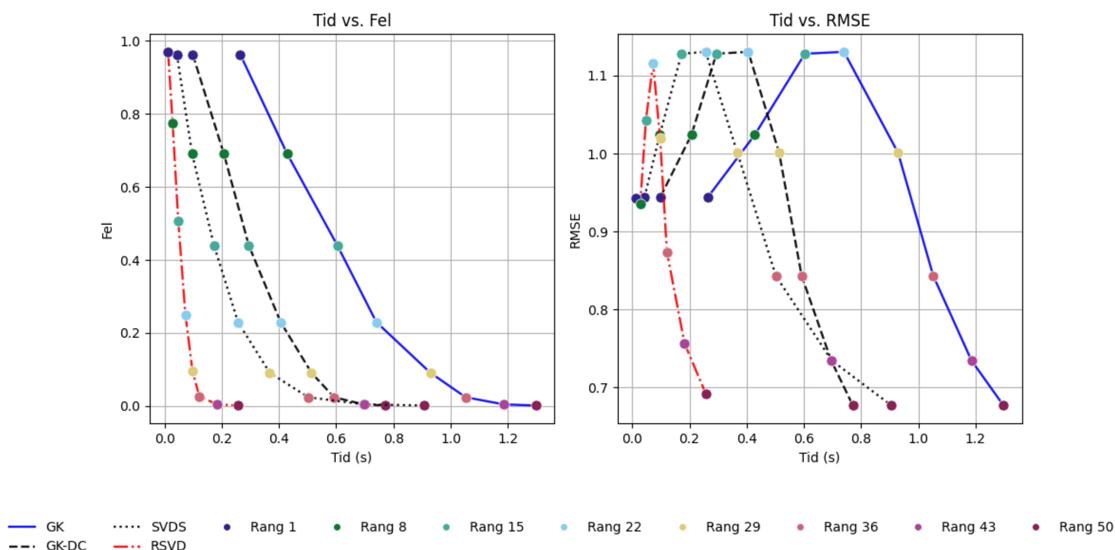
Enligt den tredje bilden slutar GK-DC på ungefär rang 400 och den justerade RSVD på rang 500. Även fast RSVD använder betydligt fler singulära värden når båda metoderna ungefär samma RMSE och fel.

4.2.2 NIHT



Figur 10: (NIHT) Medelvärde av resultatet för 10 slumpmässigt genererade 10×10 matriser med rang $= 5$ och 50% borttagna element. RSVD använder exponentiering $= 1$, översampling $= 2$.

Samtliga metoder använder sig av steglängden μ_1, tolerans på 10^{-4} och 30 max-iterationer. Figur 10 visar att GK och GK-DC presterar identiskt och snabbast med avseende på fel och RMSE, samtidigt som RSVD och SVDS går saktare, precis som i figur 7. Alla fyra metoder ökar i RMSE till tredje rangpunkten innan den minskar, RSVD visar detta tydligast.

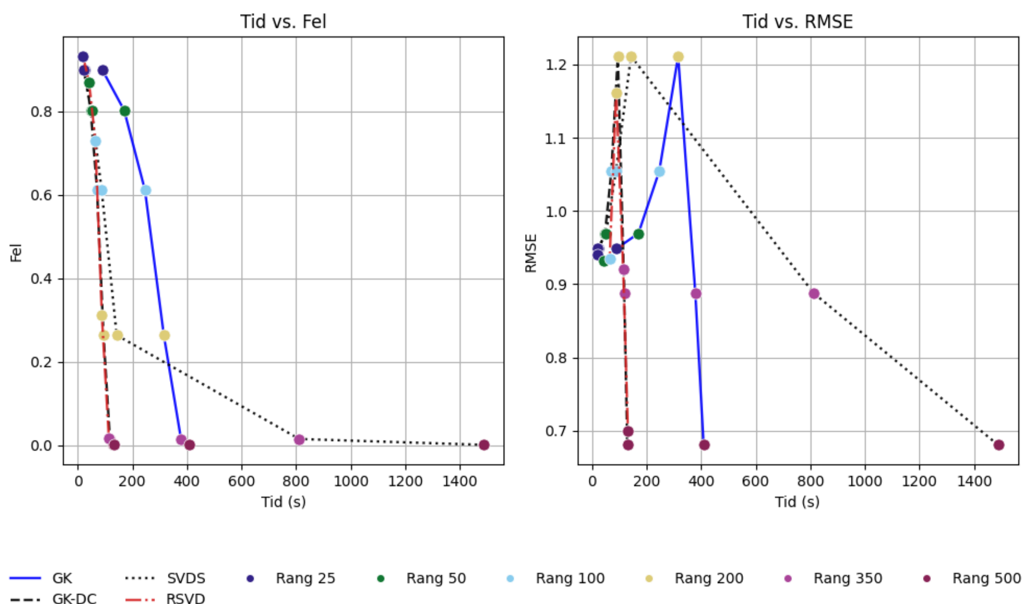


Figur 11: (NIHT) Medelvärde av resultatet för 10 slumpmässigt genererade 100x100 matriser med rang $= 50$ och 50% borttagna element. RSVD använder exponentiering $= 1$, översampling $= 2$.

Samtliga metoder använder sig av steglängd μ_1, tolerans på 10^{-4} och 30 max-iterationer. I den andra bilden är den initiala ökningen i RMSE väldigt tydlig för alla metoder. Dock hanterar RSVD

detta snabbast och har snabbast konvergenshastighet för både fel och RMSE. En tydlig skillnad kan ses mellan GK och GK-DC i denna figur gällande hastigheten av iterationerna för ranginmatning, GK-DC hanterar detta mycket snabbare.

SVDS börjar snabbare gentemot GK-DC men utefter att rang parametern ökar får de liknande sluttid för felet. Att det går saktare för högre rang inmatningar beror på att Krylovdelrummets storlek ökar för varje rang. RSVD är den metod som når högst RMSE vid sista rangpunkten.



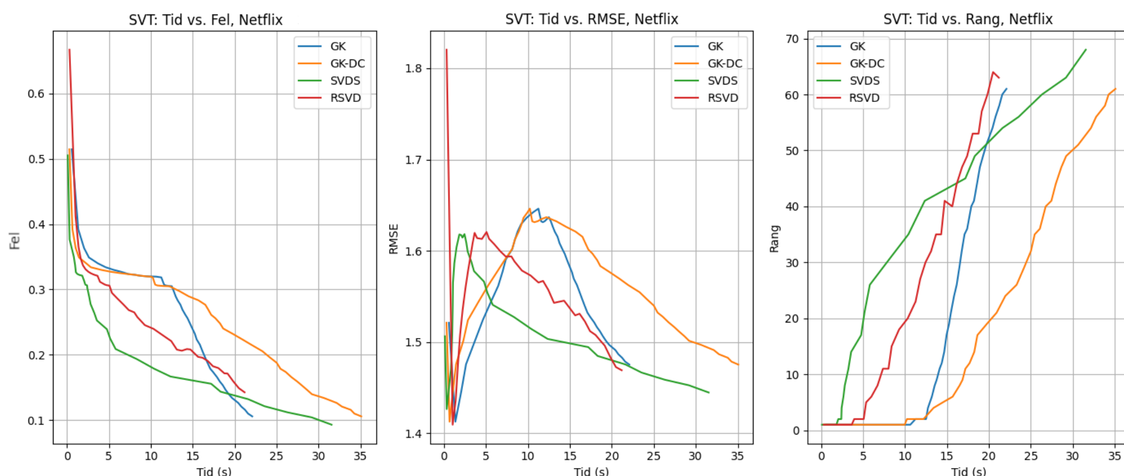
Figur 12: (NIHT) Medelvärde av resultatet för 3 slumpmässigt genererade 1000×1000 matriser med rang $= 500$ och 50% borttagna element. RSVD använder exponentiering $= 1$, översampling $= 2$

Samtliga metoder använder sig av steglängd μ_1, tolerans på 10^{-4} och 50 max-iterationer. RSVD och GK-DC har väldigt liknande mönster gällande fel och RMSE i figur 12. Båda har också liknande initial ökning i RMSE. Rangpunkterna i båda graferna för RSVD och GK-DC är också förhållandevis nära varandra, vilket indikerar ett liknande beteende. I båda graferna blir det ännu tydligare att SVDS tar betydligt längre tid vid högre ranginmatningar, men samma fel och RMSE uppnås.

4.3 Netflix-databasen

Här testas algoritmerna på Netflix-databasen med olika typer av SVD-metoder. Rapporten börjar med att undersöka SVT och sedan NIHT. Här försöker vi att ta fram de bästa parametrarna för SVT och NIHT med olika SVD-metoder. I slutet ställs SVT mot NIHT i samma graf.

4.3.1 SVT



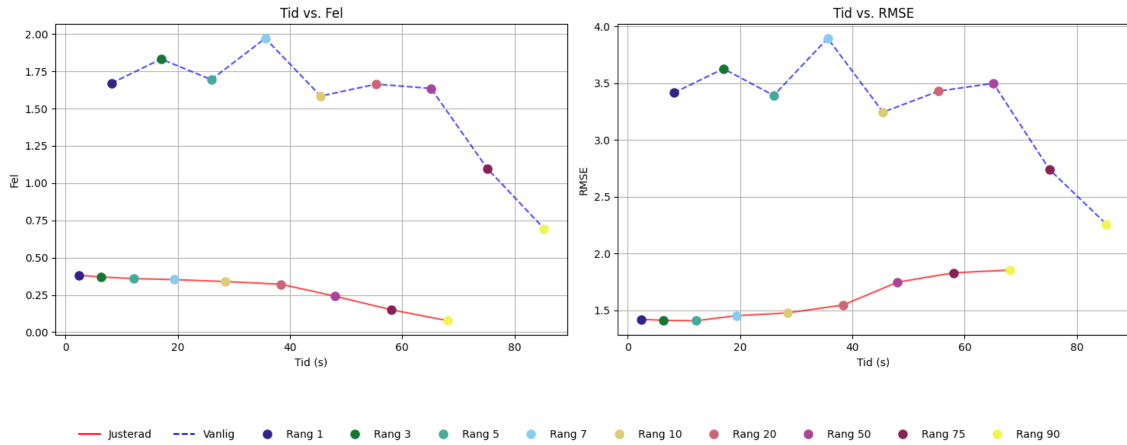
Parameter	GK	GK-DC	SVDS	RSVD
Delta	$0.6 \times \frac{n_1 n_2}{m}$	$0.6 \times \frac{n_1 n_2}{m}$	$0.6 \times \frac{n_1 n_2}{m}$	$0.5 \times \frac{n_1 n_2}{m}$
Tolerans	10^{-4}	10^{-4}	10^{-4}	10^{-4}
Tau	$4\sqrt{n_1 n_2}$	$4\sqrt{n_1 n_2}$	$2.7\sqrt{n_1 n_2}$	$2.7\sqrt{n_1 n_2}$
Max iterationer	45	45	28	33
Ökning	-	-	5	5
Snabbstart	Ja	Ja	Ja	Ja
Exponentiering	-	-	-	2
Översampling	-	-	-	10

Figur 13: (SVT) Netflix-databasen (storlek 100×4711 med cirka 64% saknade element) och parametrar.

Med SVT-algoritmen på Netflix-databasen presterar GK och SVDS bäst gentemot de andra metoderna. RSVD når inte samma fel som GK. GK-DC har långsammare körtid men når nästan samma fel och RMSE som GK.

4.3.2 NIHT

När vi först körde tester på Netflix-databasen var resultaten inte tillfredsställande. Felet och RMSE för NIHT med μ_1 konvergerande inte och oscillerade ofta fram och tillbaka. Olika justeringar på μ_1 prövades och den bästa av dessa presteras nedan.



Figur 14: (NIHT) Fel och RMSE vs. tid för steglängder μ_1 och justerat μ_1. Netflix-databas, max 40 iterationer, tolerans 10^{-4}, SVD-metod för båda är GK.

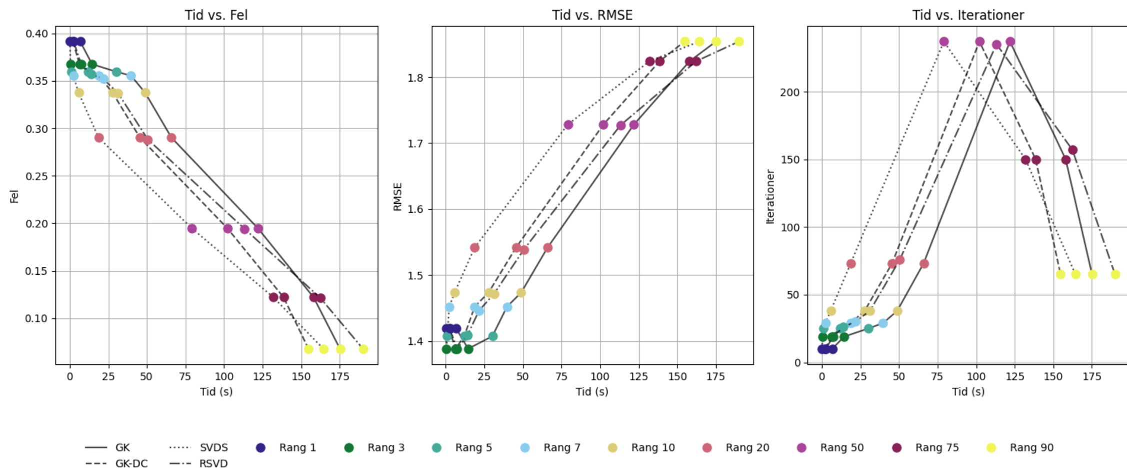
Det är tydligt att NIHT med den justerade μ_1 presterar betydligt mycket bättre. Felet konvergerar mot 0 och är mer stabilare överlag. För RMSE är den justerade också stadigare i sin kurva. RMSE sänks vid högre rang inmatningar för den ojusterade μ_1 men överlag är den väldigt ostabil. Felet och RMSE är bättre vid alla rangpunkter på den justerade μ_1.

Justeringen gjordes med en adaptiv term som multipliceras med μ_1. Den definieras enligt:

$$\mathrm{r_{jus}} = \frac{b}{\mathrm{rang}^s \times d^i}$$

b är basvärde för r_{jus}, s begränsar ranginmatningen, d är en minskningsfaktor för rang termen och i är nuvarande iteration. Värden för konstanterna valdes till $b = 0.11, s = 0.4, d = 0.98$.

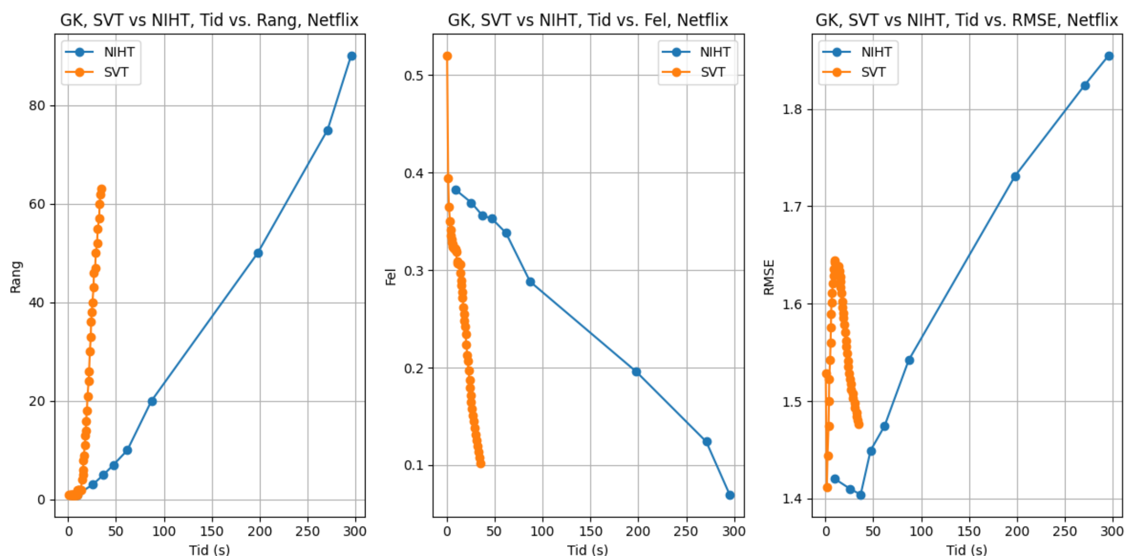
Den justerande termen kommer att stabilisera μ_1 för större rang inmatningar och flera iterationer. I samband med denna term infördes två nya stopp krav på algoritmen. Det första är att om RMSE ökar två gånger i rad stoppas algoritmen och nästa rang påbörjas. Det andra är att om RMSE är samma värde, med fem decimalers precision, på fem iterationer i rad stoppas iterationer för nuvarande rang och nästa påbörjas.



Figur 15: (NIHT) Netflix-databasen (100×4711 matris med 64% saknade element) och parametrar. RSVD använder exponentiering $= 4$, översampling $= 10$

Samtliga metoder använder sig av steglängd $\mu_1 \cdot r_{jus}$, tolerans på 10^{-4} och 10^3 max-iterationer. De adaptiva iterationerna ökar tills rangen är 50 och minskar mot slutet. Det beror på att RMSE tenderar att stabiliseras tidigt under iterationer för de större ranginmatningarna. Att antal iterationer är få i början beror på att RMSE brukar kunna divergera tidigt och genom det avbryta körningen för rangen.

Ifrån figur 15 är det inte stor skillnad på SVD-metod för NIHT. Det slutar i denna bild med att GK-DC är snabbast med avseende på tid och RMSE.



Figur 16: (SVT vs. NIHT) Netflix-databasen, storlek 100x4711, saknade element cirka 64%. Det är samma parametrar som i figure 13 för SVT och figur 15 NIHT för GK.

Det är tydligt att när SVT ställs emot NIHT på urvalet av Netflix-databasen är SVT märkbart bättre. NIHT kräver väldigt många iterationer för varje rang, vilket påverkar sluttiden av körningen. SVT har också möjligheten att beräkna rangen av matrisen mycket snabbare. För vår implementation av NIHT måste varje rang matas in och itereras, detta är inbyggt i SVT. Det gäller också att RMSE divergerar för NIHT genom att öka värdet på r, även om resultatet konvergerar för de kända värdena.

5 Diskussion och slutsats

Rapporten har nu undersökt två LRMC-algoritmer. En stor skillnad mellan NIHT- och SVT-algoritmen är att SVT approximerar rangen medan NIHT kräver att man anger rangen exakt. Av denna anledning går det inte att direkt jämföra hur lång tid det tar för algoritmerna att köra. För att lösa detta gjorde vi ett naivt test för NIHT där vi körde algoritmen för ett intervall av värden på r. SVT-algoritmen kan använda sig av tidigare information i varje iteration för att approximera rangen, vilket den naiva lösningen inte kan. Därför är detta inte en perfekt lösning för att jämföra algoritmerna, men en mer rättvis jämförelse än att bara testa NIHT för ett valt värde av r.

Genom testerna i figur 2, 3, 4 och 5 är det möjligt att få en ungefärlig bild av vilka parametrar som fungerar väl för algoritmerna. Även fast de används som referenspunkt i våra tester kan de behöva justeras utefter matrisens storlek och egenskaper. För de slumpmässigt genererade matriserna konvergerade både SVT och NIHT algoritmerna väl med parametrar nära de som föreslogs i våra tester.

För både SVT och NIHT påverkas resultaten av val av SVD-metod. Figur 7 och 10 visar att GK samt GK-DC metoderna fungerar snabbast av de valda metoderna för små matriser. Däremot presterar GK inte lika väl som GK-DC för stora matriser, som visat i figur 9 och 12. Dessa figurer visar även att SVDS presterar dåligt för stora matriser jämfört med alla andra metoder. Detta är troligtvis på grund av hur lång tid SVDS algoritmen tar att köra för höga ranger, vilket i sin tur betyder att SVDS kan prestera väl för stora matriser med väldigt låg rang. Figur 8 visar att SVDS och RSVD presterar snabbast i testerna med matriser med storlek nära 100×100, och att dessa metoder konvergerar snabbare än resterande metoder. Vi ser dock även att RSVD leder till mer instabilitet och lägre precision, speciellt om matriserna är små eller om parametrarna är suboptimala. För Netflix-databasen ser vi i figur 15 och 13 att SVT påverkas mer av den valda SVD-metoden än NIHT. Det beror på att SVT-algoritmen beräknar SVD fler gånger än NIHT. Vi ser även att SVDS ger bäst resultat vilket stämmer överens med testerna på de slumpgenererade matriserna i figur 8 och 11 eftersom Netflix-matrisen har samma maximala teoretiska rang som 100×100 matriserna.

På Netflix-databasen är det tydligt utifrån figur 16 att SVT presterar bättre med hänsyn till fel, RMSE och tid. Ett viktigt resultat är också att RMSE inte stabiliserar sig när rangen ökar för NIHT. Detta gäller även fast antal iterationer ökas med högre rang. En möjlig anledning till detta är att NIHT algoritmen konvergerar mot ett oönskat minimum, som konsekvens av att det är en icke-konvex algoritm. SVT-algoritmen löser ett konvext problem, vilket innebär att varje lokalt minimum den hittar är ett globalt minimum. Detta innebär att även om SVT kan ge olika lösningar finns det fortfarande bara ett minimum av nukleärnormen som algoritmen kan konvergera till, givet fixt τ. NIHT löser dock ett icke-konvext problem, vilket betyder att den kan hitta oönskade lokala minimum, vilket algoritmen kan konvergera mot om exempelvis steglängden väljs på ett visst sätt. I figur 14 ser vi att NIHT algoritmen ger två olika resultat för varje r beroende på vald μ.

Resultaten visar att NIHT presterar mycket sämre när det kommer till att komplettera Netflix-databasen, men den producerar fortfarande en lågrangmatris med bevarade kända värden. Om det enbart är det man bryr sig om och redan vet vilken rang man behöver är NIHT bättre lämpad än SVT, eftersom körtiden för NIHT är mycket snabbare än för SVT med bara en ranginmatning. NIHT är ursprungligen en algoritm för tillämpning i signalbehandling och kanske presterar bättre i den kontexten. Resultatet för Netflix-databasen tyder på att NIHT inte fungerade väl i detta fall. Det är dock möjligt att resultatet hade sett annorlunda ut för andra matriser.

Till vidare forskning hade det varit intressant att undersöka valet av μ på NIHT. Eventuellt är det intressant att formulera en algoritm för att adaptivt bestämma rangen till NIHT. För varje ny iteration av rang i SVT med SVDS hade det varit möjligt att spara tidigare rangberäkningar och återanvända dessa. Detta nämns som en potentiell förbättring i [2] och det kan vara av intresse att se tidsskillnaden mellan resultaten.

Referenser

[1] E. J. Candes och Y. Plan, “Matrix Completion With Noise”, *Proceedings of the IEEE*, årg. 98, nr 6, s. 925–936, 2010. URL: `http://dx.doi.org/10.1109/JPROC.2009.2035722`.

[2] J.-F. Cai, E. J. Candès och Z. Shen, “A singular value thresholding algorithm for matrix completion”, *SIAM J. Optim.*, årg. 20, nr 4, s. 1956–1982, 2010, ISSN: 1052-6234,1095-7189. URL: `https://doi.org/10.1137/080738970`.

[3] J. Tanner och K. Wei, “Normalized iterative hard thresholding for matrix completion”, *SIAM J. Sci. Comput.*, årg. 35, nr 5, S104–S125, 2013, ISSN: 1064-8275,1095-7197. URL: `https://doi.org/10.1137/120876459`.

[4] N. Halko, P. G. Martinsson och J. A. Tropp, “Finding structure with randomness: probabilistic algorithms for constructing approximate matrix decompositions”, *SIAM Rev.*, årg. 53, nr 2, s. 217–288, 2011, ISSN: 0036-1445,1095-7200. URL: `https://doi.org/10.1137/090771806`.

[5] M. Kearns och A. Roth, *The Ethical Algorithm: The Science of Socially Aware Algorithm Design.* Oxford University Press, 2020, ISBN: 9780190948207. URL: `https://books.google.se/books?id=QmmtDwAAQBAJ`.

[6] Netflix, *Netflix Prize Data*, `https://www.kaggle.com/datasets/netflix-inc/netflix-prize-data`, Accessed: 2025-05-06, 2009.

[7] S. Boyd och L. Vandenberghe, *Convex optimization.* Cambridge University Press, Cambridge, 2004, s. xiv+716, ISBN: 0-521-83378-7. URL: `https://doi.org/10.1017/CBO9780511804441`.

[8] G. Strang, *Introduction to linear algebra*, Sixth edition. Wellesley, MA: Wellesley-Cambridge Press, 2023, ISBN: 9781733146678.

[9] E. J. Candès och B. Recht, “Exact matrix completion via convex optimization”, *Found. Comput. Math.*, årg. 9, nr 6, s. 717–772, 2009, ISSN: 1615-3375,1615-3383. URL: `https://doi.org/10.1007/s10208-009-9045-5`.

[10] T. Blumensath och M. E. Davies, “Normalized Iterative Hard Thresholding: Guaranteed Stability and Performance”, *IEEE Journal of Selected Topics in Signal Processing*, årg. 4, nr 2, s. 298–309, 2010. URL: `https://doi.org/10.1109/JSTSP.2010.2042411`.

[11] E. Anderson, Z. Bai, C. Bischof m. fl., *LAPACK Users' Guide*, Third. Philadelphia, PA: Society for Industrial och Applied Mathematics, 1999, ISBN: 0-89871-447-8.

[12] R. B. Lehoucq, D. C. Sorensen och C. Yang, *ARPACK users' guide* (Software, Environments, and Tools). Society for Industrial och Applied Mathematics (SIAM), Philadelphia, PA, 1998, vol. 6, s. xvi+142, Solution of large-scale eigenvalue problems with implicitly restarted Arnoldi methods, ISBN: 0-89871-407-9. URL: `https://doi.org/10.1137/1.9780898719628`.

[13] G. H. Golub och C. F. Van Loan, *Matrix computations* (Johns Hopkins Studies in the Mathematical Sciences), Fourth. Johns Hopkins University Press, Baltimore, MD, 2013, s. xiv+756, ISBN: 978-1-4214-0794-4; 1-4214-0794-9; 978-1-4214-0859-0.

[14] R. P. Stanley, “An introduction to hyperplane arrangements”, i *Geometric combinatorics*, ser. IAS/Park City Math. Ser. Vol. 13, Amer. Math. Soc., Providence, RI, 2007, s. 389–496, ISBN: 978-0-8218-3736-8; 0-8218-3736-2. URL: `https://doi.org/10.1090/pcms/013/08`.

[15] Y. Saad, *Numerical methods for large eigenvalue problems* (Algorithms and Architectures for Advanced Scientific Computing). Manchester University Press, Manchester; Halsted Press [John Wiley & Sons, Inc.], New York, 1992, s. xii+346, ISBN: 0-7190-3386-1.

6 AI användning

Artificiell intelligens, i form av chattbottar, har ibland använts som inspirationskällor för informationsinhämtning. Informationen ifrån en sådan tjänst har inte använts direkt utan att den först kontrollerades mot andra källor och vår handledare. Ingen källkod eller text i rapporten har genererats av AI.

A Appendix 1 – Teori

A.1 Numeriska SVD-metoder

Golub-Kahan. Första metoden kan kallas Golub-Kahan SVD-metoden (GK-SVD). Tänk att vi ska utföra en SVD som i (2.4) på en matris M. GK-SVD använder sig av en två stegs metod. Denna kan beskrivas enligt algoritm 8.6.2, 8.6.1 och 5.4.2 i [13] med följande steg

Steg 1: Bidiagonalisering
M omvandlas först till en bidiagonal matris B. Matrisen kan illustreras enligt

$$A \xrightarrow{\text{Bidiagnolisering}} B = \begin{pmatrix} d_1 & e_1 & \cdots & \cdots & \cdots & \cdots & 0 \\ 0 & d_2 & e_2 & \cdots & \cdots & \cdots & 0 \\ 0 & 0 & d_3 & e_3 & \cdots & \cdots & 0 \\ 0 & 0 & 0 & d_4 & e_4 & \cdots & 0 \\ 0 & 0 & 0 & 0 & \ddots & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & d_{n-1} & e_{n-1} \\ 0 & 0 & 0 & 0 & 0 & \cdots & d_n \end{pmatrix}$$

B skapas genom att välja ut specifika rotationer som reflekterar enskilda vektorer till ett hyperplan. Matriserna som innehåller dessa vektorer kallas Householder matrices eller Householder reflections. Vanligtvis definieras ett hyperplan till ett n-dimensionellt vektorrum V som ett delrum med dimension $(n-1)$ [14, s. 2]. Det medför att vektorkomponenter nollas.

Steg 2: Diagonalisering
Efter bidiagnoliseringen består B av två diagonaler. Elementen $d_1, \ldots, d_n$ kallas för huvuddiagonalen, $e_1, \ldots, e_n$ kallas för superdiagonalen. I detta steg så kommer superdiagonalen att arbetas bort genom att applicera iterativa rotationer, vilket resulterar i Σ från (2.4). Rotationerna som utförs, vänster/höger, sparas och sedan multipliceras ihop för att återskapa U och V ifrån (2.4).

Divide-and-conquer. Denna metod bygger på GK SVD-metoden. Men istället för att arbeta iterativt på hela matrisen B så delar Divide-and-conquer (GK-DC) den upp i två delar och beräknar parallellt. Detta medför att beräkningshastigheten blir betydligt snabbare än SVD-GK, speciellt för större matriser [11].

Partiell SVD. I denna rapport kommer matriser som saknar värden att användas. Det finns flera olika metoder som är specifikt anpassade för att beräkna SVD på denna typen av matriser. Rapporten kommer att använda en metod som är mycket effektiv på denna typ av matriser. Den kallas Implicitly Restarted Arnoldi Method (IRAM) och är beskriven i figur 4.1 i [12, s. 44].

Innan vi introducerar IRAM måste ett begrepp introduceras. Ett Krylovdelrum definieras enligt

$$\mathcal{K}_k(A, \mathbf{v}_1) = \text{span}\{\mathbf{v}_1, A\mathbf{v}_1, A^2\mathbf{v}_1, \ldots, A^{k-1}\mathbf{v}_1\} \tag{A.1}$$

där $A, \mathbf{v}_1$ är godtycklig matris och vektor. Att konstruera ett Krylovvektorrum medför möjligheten att extrahera information ifrån flera linjärkombinationer av exponentieringar av matrisen [12, s. 48].

En överblick av metoden kan ges enligt [12] och algoritm 7.3 i [15, s. 169].

Steg 1: Arnoldi iteration
Skapa först ett Krylovdelrum, $\mathcal{K}_k(A, \mathbf{v}_1)$, utav en matris A med en slumpmässigt vald vektor $\mathbf{v}_1$ enligt (A.1). Sedan skapas ett ortogonalt delrum till $\mathcal{K}_k$ genom en Gram-Schmidt-liknande process. Mer precist så normaliseras varje vektor utefter nästföljande vektor. Först väljs en vektor $\mathbf{v}_1$ med norm 1. En ny vektor $(A\mathbf{v}_j)$ skapas och ortogonaliseras mot $\mathbf{v}_1$, vilket utförs $m + q = n$ gånger med olika vektorer. Vektorerna skapar en bas för Krylovdelrummet i (A.1).

Steg 2: Implicit omstart
Beräkna approximativ egenvärden av matrisen i förra steget, dessa värden kallas Ritz värden. Vi söker egenvärden som har större värde. Därför appliceras skiftningar med de egenvärden som inte är av intresse på matrisen, de med mindre värde. Detta komprimerar faktoriseringen till att behålla de egenvärden som är större.

Steg 3: Fortsätt Arnoldi
Efter att intressanta egenvärden har valts ut så fortsätter vi utföra Arnoldi iterationer q gånger. Nu är delrummet mindre men mer fokuserat på utvalda intressanta egenvärden. När alla q iterationer har körts så genereras de vänstersingulära vektorerna. Låt dessa betecknas med E_v.

Steg 4: SVD
Nu har approximerade egenvärden och egenvektorer genererats. Därifrån skapas en SVD. Beräkna först $AE_v \approx P$. Denna matris kan ses som en projicerad matris. Beräkna sedan en GK/GK-DC SVD på P:

$$P \approx U\Sigma V_h$$

Detta exponerar de höger singulära vektorerna V_h för den nya projicerade matrisen P. För att få högersingulära vektorer som relaterar till originalmatrisen A så beräknas,

$$V \approx V_h E_v^T$$

vilket transformerar vektorerna tillbaka. Den slutgiltiga approximerade SVD:n blir $A \approx U\Sigma V$. [12] rekommenderar att välja $q = 2k$, där k är antalet singulära värden vi vill beräkna. Detta är något rapporten utgår ifrån.

A.1.1 QR faktorisering

En annan typ av matrisfaktorisering är QR-faktorisering. En matris $A \in \mathbb{R}^{m\times n}$ kan faktoriseras till en produkt av en ortogonal matris $Q \in \mathbb{R}^{m\times m}$ och en övre triangulär matris $R \in \mathbb{R}^{n\times n}$ [13, Kap. 5.2]:

$$A = QR = \begin{pmatrix} | & | & & | \\ \mathbf{q}_1 & \mathbf{q}_2 & \cdots & \mathbf{q}_m \\ | & | & & | \end{pmatrix} \begin{pmatrix} r_{11} & r_{12} & \cdots & r_{1n} \\ 0 & r_{22} & \cdots & r_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & r_{nn} \end{pmatrix}$$

där $\mathbf{q_1}, \ldots, \mathbf{q_m}$ är ortogonala vektorer och $r_{i,j}$ är reella element. Matrisen Q har en viktig koppling till A:s kolonnrum. Om $A \in \mathbb{R}^{m\times n}$ och har full kolonn rang, då är $\text{span}(\mathbf{q_1}, \ldots, \mathbf{q_m}) = \text{span}(\mathbf{a_1}, \ldots, \mathbf{a_m})$ där $(\mathbf{a_1}, \ldots, \mathbf{a_m})$ är kolonn vektorer i A [13, s. 247]. Då spänner de alltså samma kolonnrum. Rapporten kommer att använda detta till slumpmässig SVD för att approximera en större matris kolonnrum.

A.1.2 Randomized SVD (RSVD)

Rapporten har tidigare nämnt att SVD är kostsamt beräkningsmässigt. Det är möjligt att minska beräkningstiden genom att skapa en mindre matris och beräkna dess singulära värden. Det första steget är att skapa ett lågdimensionellt underrum som är en approximation av den inmatade matrisens kolonnrum. Sedan är det andra steget att beräkna en matrisdekomposition på den reducerande matrisen [4]. Mer konkret:

Steg A: Vi beräknar en matris Q vars kolonnrum approximerar en kolonnrummet av en större matris M. Q behöver ha ortogonala kolumner sådant att

$$M \approx QQ^T M. \tag{A.2}$$

Steg B: Givet att (A.2) uppfylls beräknar vi med hjälp av Q en SVD av den stora matrisen M.

Dessa två steg är en övergripande bild av vad som utförs. Denna rapport använder sig utav algoritm 4.1, 4.3 och 5.1 från [4]. 4.1 och 4.3 beskriver två sätt att utföra steg A där enda skillnaden mellan dom är ett frivilligt steg som beskrivs nedan. 5.1 är en metod för att utföra steg B. Genom att kombinera dessa tre kan en algoritm beskrivas på följande sätt:

Algoritm 3 Slumpmässig direkt SVD (RSVD)

Indata: Matris $M \in \mathbb{R}^{m \times n}$, mål rang k, översamplingsantal p, exponentiering q
Utdata: Approximerad faktorisering $M \approx U_k \Sigma_k V_k^T$
Steg A.1: Slumpmässig projektion
1: Skapa slumpmässig matris $\mathrm{S} \in \mathbb{R}^{n \times (k+p)}$ med oberoende element ifrån $\mathcal{N}(0, 1)$.
2: Skapa urvalsmatrisen $Y = M\mathrm{S}$
Steg A.2: Exponentiering (frivilligt)
3: Sätt $i = 1$
Upprepa:
4: $Y = M(M^T Y)$
5: $i = i + 1$
6: **Tills:** $i = q$
Steg A.3: Beräkna QR-dekomposition
7: Beräkna QR-dekompositionen $Y = QR$
Steg B.1: Skapa projektionsmatris
8: Skapa projektionsmatrisen $B = Q^T M$
Steg B.2: Reducerad SVD
9: Beräkna SVD $B = \widehat{U} \Sigma V^T$
Steg B.3: Rekonstruktion
10: Hämta vänstra singulära vektorer till M: $U = Q\widehat{U}$
11: Trunkera till k: $U_k = U_{:,1:k}$, $\Sigma_k = \Sigma_{1:k,1:k}$, $V_k^T = V_{1:k,:}^T$
Returnera: U_k, Σ_k, V_k^T

Steg 1 använder sig av en slumpmässigt utvald matris. [4] beskriver flera olika sätta att välja ut denna matris. Men artikeln nämner också att valet av just en Gauss matris fungerar väl, rapporten kommer också använda det. Det frivilliga steg 2 används för att öka approximationskvaliten av matrisen med kostnaden av mer beräkningar. Steget ska användas om de singulära värdena inte avtar tillräckligt snabbt [4].

B Appendix 2 – Källkod

B.1 LRMC-algoritmer

B.1.1 NIHT-algoritm

```
import numpy as np

def niht(X_obs, mask, rank, max_iter=1000, tol=1e-6):
    X0 = np.nan_to_num(X_obs, nan=0.0)

    U_0r, S_0r, VT_0r = svd_function(X0, k=rank)
    X = U_0r @ np.diag(S_0r) @ VT_0r

    residuals = np.where(mask, X0 - X, 0)
    projected_residuals = U_0r @ (U_0r.T @ residuals)

    for i in range(max_iter):
        step_size = np.linalg.norm(projected_residuals, 'fro')**2 /
        (np.linalg.norm(mask*projected_residuals, 'fro')**2)

        X_new = X + (step_size*residuals)
        U_r, S_r, Vt_r = svd_function(X_new, k=rank)
        X_new = U_r @ np.diag(S_r) @ Vt_r

        residuals = mask*(X0-X_new)
        projection_operator = U_r@U_r.T
        projected_residuals = projection_operator@residuals

        error = np.linalg.norm(residuals, 'fro') / np.linalg.norm(mask * X0, 'fro')
        if error < tol:
            print(f"Converged at iteration {i+1}")
            break

        X = X_new

    return X
```

B.1.2 SVT-algoritm

```
import numpy as np

def svt_solve_scipy(mask, PM, delta, tolerance, tau, l, kmax, log=False):
    k0 = np.ceil(tau/(delta*np.linalg.norm(PM, ord=2)))
    Y0 = k0*delta*PM
    r0 = 0

    Y_prev = Y0
    r_prev = r0

    Xk = np.nan
    for k in range(1,kmax+1):
        full_theoretical_rank = np.min(Y_prev.shape)

        sk = min(r_prev + 1, full_theoretical_rank-1)
        singular_values = svd_function(Y_prev, sk, return_singular_vectors=False)
        sigma = singular_values[0]
```

```python
        while sigma > tau and sk < full_theoretical_rank:
            singular_values = svd_function(Y_prev, sk, return_singular_vectors=False)
            sigma = singular_values[-sk]
            sk = sk + 1
        U, singular_values, VT = svd_function(
            Y_prev,
            k=max(sk-1, 1),
            return_singular_vectors=True
        )

        soft_thresholded_s = np.maximum(singular_values - tau, 0)
        r_prev = np.count_nonzero(soft_thresholded_s)
        Xk = U@np.diag(soft_thresholded_s)@VT

        projected_difference = mask*(Xk-PM)
        err = np.linalg.norm(projected_difference, ord="fro")/np.linalg.norm(PM, ord="fro")
        if log: print(f"Loop: k={k} Error:{err}")
        if err <= tolerance:
            break
        Y_prev -= delta*projected_difference
    return Xk

def svt_2(nonsparseM, M, step_size, epsilon, tau, kmax, svd_function, increment=False, k_0=False,
    """
    M: Matris
    Step_size: Delta
    Epsilon: Fel
    Tau: Thresholding param
    Kmax: Maxiter
    k_0: Snabbstart
    mask: Projectionsmask
    comments: ...

    Returnera
    - X_opt, tid, error, rank

    """
    time_points = []
    error_points = []
    rank_points = []
    rmse_total_points = []
    start_time = time.time()

    m, n = np.shape(M)
    M = M.astype(np.float64)
    if mask is None:
        mask = np.ones_like(M, dtype=np.float64)

    unknown_values_mask = 1-mask

    Y = np.zeros_like(M, dtype=np.float64)

    if k_0 is True:
        k_0 = np.ceil(tau / (step_size * np.linalg.norm(M, ord=2)))
        Y = k_0 * step_size * M
```

```
        if comments:
            print(f"Snabbstart, k_0 värde: {k_0}")

    M_obs_norm = np.linalg.norm(mask * M, 'fro')
    if M_obs_norm < 1e-12:
        return M

    for k in range(kmax):

        iteration_start = time.time()
        U, sigma, Vt = svd_function(Y)

        sigma_thresholded = np.maximum(sigma - tau, 0)
        current_rank = np.sum(sigma_thresholded > 0)

        X_k = np.zeros_like(Y)
        for j in range(current_rank):
            u_j = U[:, j].reshape(-1, 1)
            v_j = Vt[j, :].reshape(1, -1)
            X_k += (sigma[j] - tau) * np.dot(u_j, v_j)

        residual = mask * (M - X_k)
        Y += step_size * residual
        error = np.linalg.norm(residual, 'fro') / M_obs_norm

        time_points.append(iteration_start - start_time)
        error_points.append(error)
        rank_points.append(current_rank)
        rmse_total_points.append(np.sqrt(np.mean(np.square(nonsparseM-X_k))))

        if comments:
            print(f"Iteration {k+1:3d}, rank = {current_rank:3d}, error = {error:.2e}")

        if error <= epsilon:
            if comments:
                print(f"Converged after {k+1} iterations with error {error:.2e}")
            break

    return X_k, time_points, error_points, rank_points, rmse_total_points
```

B.2 Numeriska SVD-metoder

```
#Bidiagnoliseringemetod
def golubkahan_svd(A):
    U, s, Vt = svd(A,full_matrices=False, lapack_driver="gesvd")
    return U, s, Vt

#Bidiagnolisering parallelliserat
def golubkahan_dc_svd(A):
    U, s, Vt = svd(A, full_matrices=False, lapack_driver="gesdd")
    return U, s, Vt

#Partiell svd
def iram_rayleigh_ritz(A, k, tol=0, which="LM", solver="arpack"):
    U, s, Vt = svds(A, k=k, tol=tol, which=which, solver=solver)
    return U, s, Vt
```

```python
#Slumpmässig svd
def random_svd(A, k, p, q):
    """
    Slumpmässig SVD (RSVD) algoritm

    Parametrar:
    A : matris
    k : rang input
    p : översampling antal
    q : exponentiering
    """

    m, n = A.shape

    # Steg A.1
    Omega = np.random.randn(n, k + p)
    Y = A @ Omega

    # Steg A.2
    for _ in range(q):
        Y = A @ (A.T @ Y)

    # Steg A.3
    Q, R, P = qr(Y, mode='economic', pivoting=False)


    # Steg B.1
    B = Q.T @ A

    # Steg B.2
    U_hat, S, Vt = svd(B, full_matrices=False, lapack_driver="gesdd")

    # Steg B.3
    U = Q @ U_hat

    # Trunkering
    U_k = U[:, :k]
    S_k = S[:k]
    Vt_k = Vt[:k, :]

    return U_k, S_k, Vt_k
```

B.3 Datagenerering

```python
import numpy as np

def remove_elements_uniform(img, percentage):
    rows, cols = img.shape
    holes_amount = int(img.size*percentage)
    removed_element_indeces = np.random.choice(rows*cols, holes_amount, replace=False)
    row_indeces, col_indeces = np.unravel_index(removed_element_indeces, (rows,cols))
    modified_img = img.copy().astype(np.float32)
    modified_img[row_indeces, col_indeces] = np.nan
```

```python
    return modified_img

def generate_data_set(m,n,r):
    unif = np.random.uniform(0, 1, (m,n))
    mean = np.mean(unif)
    std = np.std(unif)
    z_scored = (unif-mean)/std

    u,s,vt = np.linalg.svd(z_scored)
    s[r:] = 0
    return u @ np.diag(s) @ vt

def generate_data_and_masks(
    n_small, n_medium, n_large,
    r_small, r_medium, r_large,
    datasets_amount, sparsity_percentage
):
    small_datasets  = np.zeros((n_small,    n_small,    datasets_amount))
    medium_datasets = np.zeros((n_medium,   n_medium,   datasets_amount))
    large_datasets  = np.zeros((n_large,    n_large,    datasets_amount))

    small_sparse_datasets  = np.zeros((n_small,    n_small,    datasets_amount))
    medium_sparse_datasets = np.zeros((n_medium,   n_medium,   datasets_amount))
    large_sparse_datasets  = np.zeros((n_large,    n_large,    datasets_amount))

    print("Generating datasets...")
    for i in range(datasets_amount):
        small_datasets[:,:,i]   = generate_data_set(n_small,    n_small,    r_small)
        medium_datasets[:,:,i]  = generate_data_set(n_medium,   n_medium,   r_medium)
        large_datasets[:,:,i]   = generate_data_set(n_large,    n_large,    r_large)

    print("Adding sparisty...")
    for i in range(datasets_amount):
        small_sparse_datasets[:,:,i] = remove_elements_uniform(
            small_datasets[:,:,i],  sparsity_percentage
            )
        medium_sparse_datasets[:,:,i] = remove_elements_uniform(
            medium_datasets[:,:,i], sparsity_percentage
            )
        large_sparse_datasets[:,:,i] = remove_elements_uniform(
            large_datasets[:,:,i],  sparsity_percentage
            )
    mask_small  = 1-np.isnan(small_sparse_datasets).astype(int)
    mask_medium = 1-np.isnan(medium_sparse_datasets).astype(int)
    mask_large  = 1-np.isnan(large_sparse_datasets).astype(int)

    PM_small    = np.nan_to_num(small_sparse_datasets)
    PM_medium   = np.nan_to_num(medium_sparse_datasets)
    PM_large    = np.nan_to_num(large_sparse_datasets)
    print("Done!")

    return (small_datasets, medium_datasets, large_datasets,
            small_sparse_datasets, medium_sparse_datasets, large_sparse_datasets,
            mask_small, mask_medium, mask_large,
            PM_small, PM_medium, PM_large)
```

B.4 Behandling av Netflix-databasen

```
def load_data(users):
    print("Loading data...")
    start_time = time.time()
    data = []
    for k in range(1,5):
        with open(f'netflix_data/combined_data_{k}.txt') as f:
            raw = f.read()
            movies_raw = raw.split(':')

            movie_data = ["\n".join(s.strip().split('\n')[:-1]) for s in movies_raw[1:]]
            remove_last_column = lambda s: s.split(',')[:-1]
            for i, _ in enumerate(movie_data):
                movie_data[i] = movie_data[i].split('\n')
                for j, _ in enumerate(movie_data[i]):
                    movie_data[i][j] = remove_last_column(movie_data[i][j])
                    movie_data[i][j].append(i)
            movie_data = np.array([j for i in movie_data for j in i], dtype=int)

            data.append(movie_data)
            print(f"Finished iteration {k} of 4")
    data = np.array([j for i in data for j in i], dtype=int)
    values, counts = np.unique(data[:, 0], return_counts=True)
    most_active_users = values[(-counts).argsort()][:users]
    mask = np.isin(data[:,0], most_active_users)
    data = data[mask]

    end_time = time.time()
    print(f"Finished loading in {end_time-start_time} s.")
    return data

def format_as_matrix_csv():
    data = pd.read_csv('netflix_data/choice.csv')
    skimmed_data = data.drop_duplicates(['user_id', 'movie_id'])
    matrix = skimmed_data.pivot(columns='movie_id', index='user_id', values='rating').to_numpy()
    np.savetxt('netflix_data/netflix_data_matrix.csv', matrix, delimiter=',')
```

B.5 Tester

B.5.1 RMSE över tid för olika δ och μ

```
import numpy as np
from scipy.sparse.linalg import svds
import time

def svt_solve_scipy_with_rmse(mask, PM, delta, tolerance, tau, l, kmax, dataset, log=False):
    k0 = np.ceil(tau / (delta * np.linalg.norm(PM, ord=2)))
    Y0 = k0 * delta * PM
    r0 = 0

    Y_prev = Y0
    r_prev = r0

    Xk = np.zeros(PM.shape)
    rmse_arr = []
    time_arr = []
```

```
    start_time = time.time()

    for k in range(1, kmax + 1):
        sk = r_prev + 1
        singular_values = svds(Y_prev, sk, return_singular_vectors=False)
        sigma, rk = singular_values[0], 0

        full_theoretical_rank = np.min(Y_prev.shape)
        while sigma > tau or sk + l >= full_theoretical_rank:
            sk = sk + l
            singular_values = svds(Y_prev, sk, return_singular_vectors=False)
            sigma, rk = singular_values[-sk], sk

        U, singular_values, VT = svds(Y_prev, k=sk, return_singular_vectors=True)

        U_capped = U[:, :rk]
        V_capped = VT[:rk, :]
        soft_thresholded_s = np.maximum(singular_values[:rk] - tau, 0)
        Xk = U_capped @ np.diag(soft_thresholded_s) @ V_capped

        rmse = np.sqrt(np.mean((Xk - dataset)**2))
        elapsed_time = time.time() - start_time
        rmse_arr.append(rmse)
        time_arr.append(elapsed_time)

        projected_difference = mask * (Xk - PM)
        err = np.linalg.norm(projected_difference, ord="fro") / np.linalg.norm(PM, ord="fro")
        if log:
            print(f"k: {k}, RMSE: {rmse:.5f}, Delta: {delta:.5f}")
        if err <= tolerance:
            rmse_arr.extend([rmse] * (kmax - k))
            time_arr.extend([elapsed_time] * (kmax - k))
            break

        Y_prev -= delta * projected_difference

    if len(rmse_arr) < kmax:
        rmse_arr.extend([rmse_arr[-1]] * (kmax - len(rmse_arr)))
        time_arr.extend([time_arr[-1]] * (kmax - len(time_arr)))

    return Xk, np.array(time_arr), np.array(rmse_arr)

def svt_rmse_over_time_multiple_deltas(n, datasets_amount, mask, PM, tau, delta_arr,
tolerance, l, kmax, datasets, log=False):
    results = {}

    for delta in delta_arr:
        print(f"\nTesting delta = {delta:.3f}")
        rmse_over_time = pd.DataFrame({'time': np.zeros(kmax, dtype=float),
        'rmse': np.zeros(kmax, dtype=float)})

        for i in range(datasets_amount):
            _, time_arr, rmse_arr = svt_solve_scipy_with_rmse(
                mask[:, :, i],
                PM[:, :, i],
```

```
                delta,
                tolerance,
                tau,
                l,
                kmax,
                datasets[:, :, i],
                log=log
            )

            df_temp = pd.DataFrame({'time': time_arr, 'rmse': rmse_arr})
            rmse_over_time += df_temp

        rmse_over_time /= datasets_amount

        results[delta] = rmse_over_time

    return results

def niht_with_rmse(mask, X0, mu, rank, dataset, kmax, tol, log=False):
    U_0r, S_0r, VT_0r = svds(X0, k=rank)
    X = U_0r @ np.diag(S_0r) @ VT_0r

    residuals = np.where(mask, X0 - X, 0)

    if mu == 1 or mu == 3:
        projected_residuals_U = U_0r @ (U_0r.T @ residuals)

    if mu == 2 or mu == 3:
        projection_operator_V = VT_0r.T @ VT_0r
        projected_residuals_V = residuals @ projection_operator_V


    for k in range(kmax):
        if mu == 1:
            step_size = np.linalg.norm(projected_residuals_U, 'fro')**2 /
            (np.linalg.norm(mask*projected_residuals_U, 'fro')**2)
        elif mu == 2:
            step_size = np.linalg.norm(projected_residuals_V, 'fro')**2 /
            (np.linalg.norm(mask*projected_residuals_V, 'fro')**2)
        elif mu == 3:
            step_size = np.linalg.norm(projected_residuals_U @ projection_operator_V, 'fro')**2 /
            (np.linalg.norm((mask*projected_residuals_U) @ projection_operator_V, 'fro')**2)
        else:
            raise ValueError("Incorrect mu. Must be 1, 2, or 3.")


        X_new = X + (step_size*residuals)
        U_r, S_r, VT_r = svds(X_new, k=rank)
        X_new = U_r @ np.diag(S_r) @ VT_r

        residuals = mask * (X0 - X_new)
        if mu == 1 or mu == 3:
            projected_residuals_U = U_r @ (U_r.T @ residuals)

        if mu == 2 or mu == 3:
            projection_operator_V = VT_r.T @ VT_r
```

```
            projected_residuals_V = residuals @ projection_operator_V

        error = np.linalg.norm(residuals, 'fro') / np.linalg.norm(mask * X0, 'fro')
        if log:
            print(f"Iteration {k+1}, Error: {error:.3f}")

        if error < tol:
            break

        X = X_new

    rmse = np.sqrt(np.mean(np.square(X_new - dataset)))
    return rmse

def calculate_rmse_over_time_multiple_ranks(datasets_amount, mask, PM, tol, kmax, datasets,
rank_range, log=False):
    results = {}

    for mu in range(1, 4):
        print(f"\nTesting mu = {mu}")
        rmse_over_time = pd.DataFrame({'time': np.zeros(kmax, dtype=float),
                                       'rmse': np.zeros(kmax, dtype=float),
                                       'err': np.zeros(kmax, dtype=float)})

        time_arr = []
        rmse_arr = []

        time_start = time.time()
        for i in range(datasets_amount):
            for r in rank_range:
                rmse = niht_with_rmse(
                    mask[:, :, i],
                    PM[:, :, i],
                    mu,
                    r,
                    datasets[:, :, i],
                    kmax,
                    tol,
                    log=log
                )

                elapsed_time = time.time() - time_start

                time_arr.append(elapsed_time)
                rmse_arr.append(rmse)

            if len(rmse_arr) < kmax:
                rmse_arr.extend([rmse_arr[-1]] * (kmax - len(rmse_arr)))
                time_arr.extend([time_arr[-1]] * (kmax - len(time_arr)))

            df_temp = pd.DataFrame({'time': time_arr, 'rmse': rmse_arr})
            rmse_over_time += df_temp

        rmse_over_time /= datasets_amount
        results[mu-1] = rmse_over_time
    return results
```

B.5.2 RMSE och rang vs tau SVT

```
def generate_svt_completed_dataset_helper(
        n, r, datasets_amount, tau_range,
        datasets, mask, PM, tolerance, l,
        kmax, log, sparsity_percentage
    ):
    svt_results_dict = {}
    for j, tau in enumerate(tau_range):
        svt_result = np.zeros((n, n, datasets_amount))
        for i in range(datasets_amount):
            svt_result[:,:,i] = svt_solve_scipy(
                mask[:,:,i],
                PM[:,:,i],
                1.2*n*n/(np.sum(mask)),
                tolerance,
                tau,
                l,
                kmax
            )
        svt_results_dict[str(tau)] = svt_result.tolist()
        if log: print(f"Iteration: {j+1} of {len(tau_range)}, TAU: {tau}")
    result = {}
    result["parameters"] = {
        'kmax':kmax,'l':l,
        'tolerance':tolerance,
        'sparsity_percentage':sparsity_percentage,
        'n': n,
        'rank': r
    }
    result["original"] = datasets.tolist()
    result["mask"] = mask.tolist()
    result["svt_completed"] = svt_results_dict

    return result

def generate_svt_completed_datasets(
        n_small,n_medium, n_large,
        r_small, r_medium, r_large,
        datasets_amount,
        sparsity_percentage,
        kmax,
        tolerance,
        l,
        log=False,
        save_dir_small=None, save_dir_medium=None, save_dir_large=None
    ):

    (small_datasets, medium_datasets, large_datasets,
    small_sparse_datasets, medium_sparse_datasets, large_sparse_datasets,
    mask_small, mask_medium, mask_large,
    PM_small, PM_medium, PM_large) = generate_data_and_masks(
        n_small, n_medium, n_large,
        r_small, r_medium, r_large,
        datasets_amount,
        sparsity_percentage
```

```
)

tau_range_small = np.geomspace(1, 10*n_small, 50)
tau_range_medium = np.geomspace(1, 10*n_medium, 50)
tau_range_large = np.geomspace(100, 10*n_large, 6)

small_result = generate_svt_completed_dataset_helper(
    n_small,
    r_small,
    datasets_amount,
    tau_range_small,
    small_datasets,
    mask_small,
    PM_small,
    tolerance,
    l,
    kmax,
    log,
    sparsity_percentage
)
if not save_dir_small is None:
    with open(save_dir_small, 'w') as f:
        json.dump(small_result, f)

medium_result = generate_svt_completed_dataset_helper(
    n_medium,
    r_medium,
    datasets_amount,
    tau_range_medium,
    medium_datasets,
    mask_medium,
    PM_medium,
    tolerance,
    l,
    kmax,
    log,
    sparsity_percentage
)
if not save_dir_medium is None:
    with open(save_dir_medium, 'w') as f:
        json.dump(medium_result, f)

large_result = generate_svt_completed_dataset_helper(
    n_large,
    r_large,
    datasets_amount,
    tau_range_large,
    large_datasets,
    mask_large,
    PM_large,
    tolerance,
    l,
    kmax,
    log,
    sparsity_percentage
)
```

```python
    if not save_dir_large is None:
        with open(save_dir_large, 'w') as f:
            json.dump(large_result, f)

    return small_result, medium_result, large_result

# DATASET STRUCTURE:
# parameters: dict {
#       kmax:                   int
#       l:                      int
#       tolerance:              float
#       sparsity_percentage:    float
#       n:                      int
#       rank:                   int
#   }
# original:         3darray
# mask:             3darray
# svt_completed:    dict{ tau: -> 3darray }

def get_r_vs_tau(dataset):
    tau_values = np.fromiter(dataset["svt_completed"].keys(), dtype=float)
    rank_matrix = np.zeros((int(dataset["parameters"]["n"]), len(tau_values)))

    for i, tau in enumerate(tau_values):
        B = np.array(dataset["svt_completed"][str(tau)])
        for j in range(B.shape[2]):
            rank_matrix[j, i] = np.linalg.matrix_rank(B[:,:,j])
    return tau_values, rank_matrix

def get_rmse_vs_tau(dataset):
    tau_values = np.fromiter(dataset["svt_completed"].keys(), dtype=float)
    A = np.array(dataset["original"])

    rmse_values = np.zeros((A.shape[2], len(tau_values)))
    for i, tau in enumerate(tau_values):
        B = np.array(dataset["svt_completed"][str(tau)])
        for j in range(A.shape[2]):
            rmse_values[j,i] = np.sqrt(np.mean((A[:,:,j]-B[:,:,j])*(A[:,:,j]-B[:,:,j])))
    return tau_values, rmse_values
```

B.5.3 RMSE vs r NIHT

```python
def generate_niht_completed_dataset_helper(
    n,
    r_true,
    datasets_amount,
    r_range, datasets,
    sparse_datasets,
    mask,
    tolerance,
    kmax,
    log,
    sparsity_percentage,
    detailed_log=False
):
```

```
    niht_results_dict = {}
    for j, r_estimate in enumerate(r_range):
        niht_result = np.zeros((n, n, datasets_amount))
        for i in range(datasets_amount):
            niht_result[:,:,i], error_arr = niht_rsvd(
                sparse_datasets[:,:,i],
                mask[:,:,i],
                r_estimate,
                kmax,
                tolerance,
                log=detailed_log
            )
        niht_results_dict[str(r_estimate)] = {
            "error_arr":error_arr,
            "result": niht_result.tolist()
        }
        if log: print(f"Iteration: {j+1} of {len(r_range)}, r: {r_estimate}")
    result = {}
    result["parameters"] = {
        'kmax':kmax,
        'tolerance':tolerance,
        'sparsity_percentage':sparsity_percentage,
        'n': n,
        'rank': r_true
    }
    result["original"] = datasets.tolist()
    result["mask"] = mask.tolist()
    result["niht_completed"] = niht_results_dict

    return result

def generate_niht_completed_datasets(
        n_small,n_medium, n_large,
        r_small, r_medium, r_large,
        datasets_amount,
        sparsity_percentage,
        kmax,
        tolerance,
        log=False,
        save_dir_small=None, save_dir_medium=None, save_dir_large=None
    ):

    (small_datasets, medium_datasets, large_datasets,
    small_sparse_datasets, medium_sparse_datasets, large_sparse_datasets,
    mask_small, mask_medium, mask_large,
    PM_small, PM_medium, PM_large) = generate_data_and_masks(
        n_small, n_medium, n_large,
        r_small, r_medium, r_large,
        datasets_amount,
        sparsity_percentage
    )

    r_range_small = np.linspace(1, n_small-1, n_small-1).astype(int)
    r_range_medium = np.linspace(1, n_medium-1, n_small-1).astype(int)
    r_range_large = np.linspace(1, n_large-1, 3).astype(int)
```

```
    small_result = generate_niht_completed_dataset_helper(
        n_small,
        r_small,
        datasets_amount,
        r_range_small,
        small_datasets,
        small_sparse_datasets,
        mask_small,
        tolerance,
        kmax,
        log,
        sparsity_percentage
    )
    if not save_dir_small is None:
        with open(save_dir_small, 'w') as f:
            json.dump(small_result, f)

    medium_result = generate_niht_completed_dataset_helper(
        n_medium,
        r_medium,
        datasets_amount,
        r_range_medium,
        medium_datasets,
        medium_sparse_datasets,
        mask_medium,
        tolerance,
        kmax,
        log,
        sparsity_percentage
    )
    if not save_dir_medium is None:
        with open(save_dir_medium, 'w') as f:
            json.dump(medium_result, f)

    large_result = generate_niht_completed_dataset_helper(
        n_large,
        r_large,
        datasets_amount,
        r_range_large,
        large_datasets,
        large_sparse_datasets,
        mask_large,
        tolerance,
        kmax,
        log,
        sparsity_percentage,
        True
    )
    if not save_dir_large is None:
        with open(save_dir_large, 'w') as f:
            json.dump(large_result, f)

    return small_result, medium_result


# DATASET STRUCTURE:
```

```
# parameters: dict {
#       kmax:                   int
#       l:                      int
#       tolerance:              float
#       sparsity_percentage:    float
#       n:                      int
#       rank:                   int
#   }
# original:         3darray
# mask:             3darray
# niht_completed:    dict{ r: -> {error_arr: 1darray, result: 3darray} }

def get_rmse_vs_r(data):
    r_values = np.fromiter(data["niht_completed"].keys(), dtype=int)
    A = np.array(data["original"])

    error_arrays = []
    rmse_values = np.zeros((A.shape[2], len(r_values)))
    for i, r in enumerate(r_values):
        entry = data["niht_completed"][str(r)]
        error_arr = np.array(entry["error_arr"])
        error_arrays.append(error_arr)
        B = np.array(entry["result"])
        for j in range(A.shape[2]):
            rmse_values[j,i] = np.sqrt(np.mean((A[:,:,j]-B[:,:,j])*(A[:,:,j]-B[:,:,j])))
    return r_values, rmse_values

def get_errorarrays_vs_r(data):
    r_values = np.fromiter(data["niht_completed"].keys(), dtype=int)

    error_arrays = []
    for r in r_values:
        entry = data["niht_completed"][str(r)]
        error_arr = np.array(entry["error_arr"])
        error_arrays.append(error_arr)
    return r_values, error_arrays
```

B.5.4 SVT för olika SVD-metoder på slumpgenererade matriser

10×10 matriser

```
time_points_gk_10matrices = []
error_points_gk_10matrices = []
rank_points_gk_10matrices = []
rmse_total_points_gk_10matrices = []

time_points_gkd_10matrices = []
error_points_gkd_10matrices = []
rank_points_gkd_10matrices = []
rmse_total_points_gkd_10matrices = []

time_points_svds_10matrices = []
error_points_svds_10matrices = []
rank_points_svds_10matrices = []
rmse_total_points_svds_10matrices = []
```

```
time_points_rsvd_10matrices = []
error_points_rsvd_10matrices = []
rank_points_rsvd_10matrices = []
rmse_total_points_rsvd_10matrices = []

number_of_matrices10x10 = 10
matrix_size = 10

for i in range(1,number_of_matrices10x10+1):
    unif_10x10 = np.random.uniform(0, 1, (matrix_size,matrix_size))

    mean = np.mean(unif_10x10)
    std = np.std(unif_10x10)
    unif_10x10 = (unif_10x10-mean)/std

    u,s,vt = np.linalg.svd(unif_10x10)
    number_of_sing = matrix_size // 2
    s[number_of_sing:] = 0
    normal_matrix_10 = u @ np.diag(s) @ vt

    flattened_matrix = normal_matrix_10.flatten()

    total_elements = flattened_matrix.size
    num_to_sparsify = total_elements // 2

    indices_to_zero = np.random.choice(total_elements, num_to_sparsify, replace=False)

    sparsified_matrix_10 = flattened_matrix.copy()
    sparsified_matrix_10[indices_to_zero] = 0

    sparsified_matrix_10 = sparsified_matrix_10.reshape(normal_matrix_10.shape)


    n1_test_10, n2_test_10 = sparsified_matrix_10.shape
    m_test_10 = np.count_nonzero(sparsified_matrix_10)


    ################################################################

    step_size_gk = 0.6* n1_test_10*n2_test_10 / m_test_10
    tolerance_gk = 10E-4
    tau_gk = 3 * n1_test_10
    max_iter_gk = 40
    bool_mask_gk = sparsified_matrix_10 != 0

    step_size_gkd = 0.6* n1_test_10*n2_test_10 / m_test_10
    tolerance_gkd = 10E-4
    tau_gkd = 3 * n1_test_10
    max_iter_gkd = 40
    bool_mask_gkd = sparsified_matrix_10 != 0

    step_size_svds = 0.6* n1_test_10*n2_test_10 / m_test_10
    tolerance_svds = 10E-4
    tau_svds =3 * n1_test_10
    max_iter_svds = 40
```

```
    increment_svds = 1
    bool_mask_svds = sparsified_matrix_10 != 0

    step_size_rsvd = 0.6* n1_test_10*n2_test_10 / m_test_10
    tolerance_rsvd = 10E-4
    tau_rsvd = 3 * n1_test_10
    max_iter_rsvd = 40
    increment_rsvd = 1
    bool_mask_rsvd = sparsified_matrix_10 != 0

    ################################################################

    X_k, time_points_gk, error_points_gk, rank_points_gk,rmse_total_points_gk =
    svt_thresholding_only(normal_matrix_10, sparsified_matrix_10,
    step_size_gk, tolerance_gk, tau_gk, max_iter_gk, svd_function=golubkahan_svd,
    increment=False, k_0=False,mask=bool_mask_gk, comments=False)

    X_k, time_points_gkd, error_points_gkd, rank_points_gkd, rmse_total_points_gkd =
    svt_thresholding_only(normal_matrix_10, sparsified_matrix_10,
     step_size_gkd, tolerance_gkd, tau_gkd, max_iter_gkd, svd_function=golubkahan_dc_svd,
     increment=False, k_0=False, mask=bool_mask_gkd, comments=False)

    X_k, time_points_svds, error_points_svds, rank_points_svds, rmse_total_points_svds =
    svt_threshholding_fixrang(normal_matrix_10, sparsified_matrix_10, step_size_svds,
    tolerance_svds, tau_svds, max_iter_svds,
    l=increment_svds, svd_function="", k_0=True, mask=bool_mask_svds, comments=False)

    X_k, time_points_rsvd, error_points_rsvd, rank_points_rsvd, rmse_total_points_rsvd =
    svt_threshholding_fixrang(normal_matrix_10, sparsified_matrix_10, step_size_rsvd,
    tolerance_rsvd, tau_svds,max_iter_rsvd,
    l=increment_rsvd, svd_function="random_svd", k_0=False, mask=bool_mask_svds, comments=True)

    ################################################################

    time_points_gk_10matrices.append(time_points_gk)
    error_points_gk_10matrices.append(error_points_gk)
    rank_points_gk_10matrices.append(rank_points_gk)
    rmse_total_points_gk_10matrices.append(rmse_total_points_gk)

    time_points_gkd_10matrices.append(time_points_gkd)
    error_points_gkd_10matrices.append(error_points_gkd)
    rank_points_gkd_10matrices.append(rank_points_gkd)
    rmse_total_points_gkd_10matrices.append(rmse_total_points_gkd)

    time_points_svds_10matrices.append(time_points_svds)
    error_points_svds_10matrices.append(error_points_svds)
    rank_points_svds_10matrices.append(rank_points_svds)
    rmse_total_points_svds_10matrices.append(rmse_total_points_svds)

    time_points_rsvd_10matrices.append(time_points_rsvd)
    error_points_rsvd_10matrices.append(error_points_rsvd)
    rank_points_rsvd_10matrices.append(rank_points_rsvd)
    rmse_total_points_rsvd_10matrices.append(rmse_total_points_rsvd)


time_points_gk_average_10 = np.sum(time_points_gk_10matrices, axis=0) / 10
```

```
error_points_gk_average_10 = np.sum(error_points_gk_10matrices, axis=0)/10
rank_points_gk_average_10 = np.sum(rank_points_gk_10matrices, axis=0) /10
rmse_total_points_gk_average_10 = np.sum(rmse_total_points_gk_10matrices, axis=0) /10

time_points_gkd_average_10 = np.sum(time_points_gkd_10matrices, axis=0) /10
error_points_gkd_average_10 = np.sum(error_points_gkd_10matrices, axis=0)/10
rank_points_gkd_average_10 = np.sum(rank_points_gkd_10matrices, axis=0) /10
rmse_total_points_gkd_average_10 = np.sum(rmse_total_points_gkd_10matrices, axis=0) /10

time_points_svds_average_10 = np.sum(time_points_svds_10matrices, axis=0) /10
error_points_svds_average_10 = np.sum(error_points_svds_10matrices, axis=0)/10
rank_points_svds_average_10 = np.sum(rank_points_svds_10matrices, axis=0) /10
rmse_total_points_svds_average_10 = np.sum(rmse_total_points_svds_10matrices, axis=0) /10

time_points_rsvd_average_10 = np.sum(time_points_rsvd_10matrices, axis=0) /10
error_points_rsvd_average_10 = np.sum(error_points_rsvd_10matrices, axis=0)/10
rank_points_rsvd_average_10 = np.sum(rank_points_rsvd_10matrices, axis=0) /10
rmse_total_points_rsvd_average_10 = np.sum(rmse_total_points_rsvd_10matrices, axis=0) /10
```

100×100 matriser

```
time_points_gk_100matrices = []
error_points_gk_100matrices = []
rank_points_gk_100matrices = []
rmse_total_points_gk_100matrices = []

time_points_gkd_100matrices = []
error_points_gkd_100matrices = []
rank_points_gkd_100matrices = []
rmse_total_points_gkd_100matrices = []

time_points_svds_100matrices = []
error_points_svds_100matrices = []
rank_points_svds_100matrices = []
rmse_total_points_svds_100matrices = []

time_points_rsvd_100matrices = []
error_points_rsvd_100matrices = []
rank_points_rsvd_100matrices = []
rmse_total_points_rsvd_100matrices = []

time_points_rsvd_opt_100matrices = []
error_points_rsvd_opt_100matrices = []
rank_points_rsvd_opt_100matrices = []
rmse_total_points_rsvd_opt_100matrices = []

number_of_matrices100x100 = 2
matrix_size = 100
for i in range(1,number_of_matrices100x100+1):
    unif_100x100 = np.random.uniform(0, 1, (matrix_size,matrix_size))

    mean = np.mean(unif_100x100)
    std = np.std(unif_100x100)
    unif_100x100 = (unif_100x100-mean)/std

    u,s,vt = np.linalg.svd(unif_100x100)
    number_of_sing = matrix_size // 2
```

```
s[number_of_sing:] = 0
normal_matrix_100 = u @ np.diag(s) @ vt

flattened_matrix = normal_matrix_100.flatten()

total_elements = flattened_matrix.size
num_to_sparsify = total_elements // 2

indices_to_zero = np.random.choice(total_elements, num_to_sparsify, replace=False)

sparsified_matrix_100 = flattened_matrix.copy()
sparsified_matrix_100[indices_to_zero] = 0

sparsified_matrix_100 = sparsified_matrix_100.reshape(normal_matrix_100.shape)

n1_test_100, n2_test_100 = sparsified_matrix_100.shape
m_test_100 = np.count_nonzero(sparsified_matrix_100)
################################################################
step_size_gk = 1 * n1_test_100*n2_test_100 / m_test_100
tolerance_gk = 10E-4
tau_gk = 3 * n1_test_100
max_iter_gk = 70
bool_mask_gk = sparsified_matrix_100 != 0

step_size_gkd = 1 * n1_test_100*n2_test_100 / m_test_100
tolerance_gkd = 10E-4
tau_gkd = 3 * n1_test_100
max_iter_gkd = 70
bool_mask_gkd = sparsified_matrix_100 != 0

step_size_svds = 1 * n1_test_100*n2_test_100 / m_test_100
tolerance_svds = 10E-4
tau_svds = 3 * np.sqrt(n1_test_100)
max_iter_svds = 70
increment_svds = 5
bool_mask_svds = sparsified_matrix_100 != 0

step_size_rsvd = 1 * n1_test_100*n2_test_100 / m_test_100
tolerance_rsvd = 10E-4
tau_rsvd = 3 * n1_test_100
max_iter_rsvd = 70
increment_rsvd = 5
bool_mask_rsvd = sparsified_matrix_100 != 0


################################################################
X_k, time_points_gk, error_points_gk, rank_points_gk, rmse_total_points_gk =
svt_thresholding_only(normal_matrix_100, sparsified_matrix_100,
 step_size_gk, tolerance_gk, tau_gk, max_iter_gk, svd_function=golubkahan_svd,
 increment=False, k_0=True, mask=bool_mask_gk, comments=True)

X_k, time_points_gkd, error_points_gkd, rank_points_gkd, rmse_total_points_gkd =
svt_thresholding_only(normal_matrix_100, sparsified_matrix_100,
 step_size_gkd, tolerance_gkd, tau_gkd, max_iter_gkd, svd_function=golubkahan_dc_svd,
 increment=False, k_0=True, mask=bool_mask_gkd, comments=True)
```

```
    X_k, time_points_svds, error_points_svds, rank_points_svds, rmse_total_points_svds =
    svt_threshholding_fixrang(normal_matrix_100, sparsified_matrix_100, step_size_svds,
    tolerance_svds, tau_svds, max_iter_svds,
    l=increment_svds, svd_function="", k_0=True, mask=bool_mask_svds, comments=True)

    X_k, time_points_rsvd, error_points_rsvd, rank_points_rsvd, rmse_total_points_rsvd =
    svt_threshholding_fixrang(normal_matrix_100, sparsified_matrix_100, step_size_rsvd,
    tolerance_rsvd, tau_rsvd,max_iter_rsvd,
    l=increment_rsvd, svd_function="random_svd_opt", k_0=True, mask=bool_mask_svds,
    comments=True)

    ################################################################


    time_points_gk_100matrices.append(time_points_gk)
    error_points_gk_100matrices.append(error_points_gk)
    rank_points_gk_100matrices.append(rank_points_gk)
    rmse_total_points_gk_100matrices.append(rmse_total_points_gk)

    time_points_gkd_100matrices.append(time_points_gkd)
    error_points_gkd_100matrices.append(error_points_gkd)
    rank_points_gkd_100matrices.append(rank_points_gkd)
    rmse_total_points_gkd_100matrices.append(rmse_total_points_gkd)

    time_points_svds_100matrices.append(time_points_svds)
    error_points_svds_100matrices.append(error_points_svds)
    rank_points_svds_100matrices.append(rank_points_svds)
    rmse_total_points_svds_100matrices.append(rmse_total_points_svds)

    time_points_rsvd_100matrices.append(time_points_rsvd)
    error_points_rsvd_100matrices.append(error_points_rsvd)
    rank_points_rsvd_100matrices.append(rank_points_rsvd)
    rmse_total_points_rsvd_100matrices.append(rmse_total_points_rsvd)



    ################################################################

time_points_gk_average_100 = np.sum(time_points_gk_100matrices, axis=0) /10
error_points_gk_average_100 = np.sum(error_points_gk_100matrices, axis=0)/10
rank_points_gk_average_100 = np.sum(rank_points_gk_100matrices, axis=0) /10
rmse_total_points_gk_average_100 = np.sum(rmse_total_points_gk_100matrices, axis=0) /10

time_points_gkd_average_100 = np.sum(time_points_gkd_100matrices, axis=0) /10
error_points_gkd_average_100 = np.sum(error_points_gkd_100matrices, axis=0)/10
rank_points_gkd_average_100 = np.sum(rank_points_gkd_100matrices, axis=0) /10
rmse_total_points_gkd_average_100 = np.sum(rmse_total_points_gkd_100matrices, axis=0) /10

time_points_svds_average_100 = np.sum(time_points_svds_100matrices, axis=0) /10
error_points_svds_average_100 = np.sum(error_points_svds_100matrices, axis=0)/10
rank_points_svds_average_100 = np.sum(rank_points_svds_100matrices, axis=0) /10
rmse_total_points_svds_average_100 = np.sum(rmse_total_points_svds_100matrices, axis=0) /10

time_points_rsvd_average_100 = np.sum(time_points_rsvd_100matrices, axis=0) /10
error_points_rsvd_average_100 = np.sum(error_points_rsvd_100matrices, axis=0)/10
rank_points_rsvd_average_100 = np.sum(rank_points_rsvd_100matrices, axis=0) /10
```

```
rmse_total_points_rsvd_average_100 = np.sum(rmse_total_points_rsvd_100matrices, axis=0) /10
```

1000×1000 matriser

```
time_points_gk_1000matrices = []
error_points_gk_1000matrices = []
rank_points_gk_1000matrices = []
rmse_total_points_gk_1000matrices = []

time_points_gkd_1000matrices = []
error_points_gkd_1000matrices = []
rank_points_gkd_1000matrices = []
rmse_total_points_gkd_1000matrices = []

time_points_svds_1000matrices = []
error_points_svds_1000matrices = []
rank_points_svds_1000matrices = []
rmse_total_points_svds_1000matrices = []

time_points_rsvd_1000matrices = []
error_points_rsvd_1000matrices = []
rank_points_rsvd_1000matrices = []
rmse_total_points_rsvd_1000matrices = []

time_points_rsvd_opt_1000matrices = []
error_points_rsvd_opt_1000matrices = []
rank_points_rsvd_opt_1000matrices = []
rmse_total_points_rsvd_opt_1000matrices = []


number_of_matrices1000x1000 = 3
matrix_size = 1000

for i in range(1,number_of_matrices1000x1000+1):
    unif_1000x1000 = np.random.uniform(0, 1, (matrix_size,matrix_size))

    mean = np.mean(unif_1000x1000)
    std = np.std(unif_1000x1000)
    unif_1000x1000 = (unif_1000x1000-mean)/std

    u,s,vt = np.linalg.svd(unif_1000x1000)
    number_of_sing = matrix_size // 2
    s[number_of_sing:] = 0
    normal_matrix_1000 = u @ np.diag(s) @ vt

    flattened_matrix = normal_matrix_1000.flatten()

    total_elements = flattened_matrix.size
    num_to_sparsify = total_elements // 2

    indices_to_zero = np.random.choice(total_elements, num_to_sparsify, replace=False)

    sparsified_matrix_1000 = flattened_matrix.copy()
    sparsified_matrix_1000[indices_to_zero] = 0

    sparsified_matrix_1000 = sparsified_matrix_1000.reshape(normal_matrix_1000.shape)
```

```
n1_test_1000, n2_test_1000 = sparsified_matrix_1000.shape
m_test_1000 = np.count_nonzero(sparsified_matrix_1000)
################################################################
step_size_gk = 1.2* n1_test_1000*n2_test_1000 / m_test_1000
tolerance_gk = 10E-4
tau_gk = 5 * n1_test_1000
max_iter_gk = 100
bool_mask_gk = sparsified_matrix_1000 != 0

step_size_gkd = 1.2* n1_test_1000*n2_test_1000 / m_test_1000
tolerance_gkd = 10E-4
tau_gkd = 5 * n1_test_1000
max_iter_gkd = 100
bool_mask_gkd = sparsified_matrix_1000 != 0

step_size_svds = 1.2* n1_test_1000*n2_test_1000 / m_test_1000
tolerance_svds = 10E-4
tau_svds = 5 * n1_test_1000
max_iter_svds = 45
increment_svds = 5
bool_mask_svds = sparsified_matrix_1000 != 0

step_size_rsvd = 1.2* n1_test_1000*n2_test_1000 / m_test_1000
tolerance_rsvd = 10E-4
tau_rsvd = 5 * n1_test_1000
max_iter_rsvd = 25
increment_rsvd = 5
bool_mask_rsvd = sparsified_matrix_1000 != 0

step_size_rsvd_opt = 0.1* n1_test_1000*n2_test_1000 / m_test_1000
tolerance_rsvd_opt = 10E-4
tau_rsvd_opt = 0.2 * n1_test_1000
max_iter_rsvd_opt = 100
increment_rsvd_opt = 5
bool_mask_rsvd_opt = sparsified_matrix_1000 != 0
################################################################
X_k, time_points_gk, error_points_gk, rank_points_gk, rmse_total_points_gk =
svt_thresholding_only(normal_matrix_1000, sparsified_matrix_1000,
 step_size_gk, tolerance_gk, tau_gk, max_iter_gk, svd_function=golubkahan_svd,
 increment=False, k_0=True, mask=bool_mask_gk, comments=True)

X_k, time_points_gkd, error_points_gkd, rank_points_gkd, rmse_total_points_gkd =
svt_thresholding_only(normal_matrix_1000, sparsified_matrix_1000,
 step_size_gkd, tolerance_gkd, tau_gkd, max_iter_gkd, svd_function=golubkahan_dc_svd,
 increment=False, k_0=True, mask=bool_mask_gkd, comments=True)

X_k, time_points_svds, error_points_svds, rank_points_svds, rmse_total_points_svds =
svt_threshholding_fixrang(normal_matrix_1000, sparsified_matrix_1000, step_size_svds,
tolerance_svds, tau_svds, max_iter_svds,
l=increment_svds, svd_function="", k_0=True, mask=bool_mask_svds, comments=True)

X_k, time_points_rsvd, error_points_rsvd, rank_points_rsvd, rmse_total_points_rsvd =
svt_threshholding_fixrang(normal_matrix_1000, sparsified_matrix_1000, step_size_rsvd,
tolerance_rsvd, tau_rsvd,max_iter_rsvd,
l=increment_rsvd, svd_function="random_svd_opt", k_0=True, mask=bool_mask_rsvd,
comments=True)
```

```
    X_k, time_points_rsvd_opt, error_points_rsvd_opt, rank_points_rsvd_opt,
    rmse_total_points_rsvd_opt = svt_threshholding_fixrang(normal_matrix_1000,
    sparsified_matrix_1000, step_size_rsvd_opt, tolerance_rsvd_opt, tau_rsvd_opt,
    max_iter_rsvd_opt, l=increment_rsvd_opt, svd_function="random_svd_opt", k_0=True,
    mask=bool_mask_rsvd_opt, comments=True)

    ################################################################


    time_points_gk_1000matrices.append(time_points_gk)
    error_points_gk_1000matrices.append(error_points_gk)
    rank_points_gk_1000matrices.append(rank_points_gk)
    rmse_total_points_gk_1000matrices.append(rmse_total_points_gk)

    time_points_gkd_1000matrices.append(time_points_gkd)
    error_points_gkd_1000matrices.append(error_points_gkd)
    rank_points_gkd_1000matrices.append(rank_points_gkd)
    rmse_total_points_gkd_1000matrices.append(rmse_total_points_gkd)

    time_points_svds_1000matrices.append(time_points_svds)
    error_points_svds_1000matrices.append(error_points_svds)
    rank_points_svds_1000matrices.append(rank_points_svds)
    rmse_total_points_svds_1000matrices.append(rmse_total_points_svds)

    time_points_rsvd_1000matrices.append(time_points_rsvd)
    error_points_rsvd_1000matrices.append(error_points_rsvd)
    rank_points_rsvd_1000matrices.append(rank_points_rsvd)
    rmse_total_points_rsvd_1000matrices.append(rmse_total_points_rsvd)

    time_points_rsvd_opt_1000matrices.append(time_points_rsvd_opt)
    error_points_rsvd_opt_1000matrices.append(error_points_rsvd_opt)
    rank_points_rsvd_opt_1000matrices.append(rank_points_rsvd_opt)
    rmse_total_points_rsvd_opt_1000matrices.append(rmse_total_points_rsvd_opt)

    ################################################################

time_points_gk_average_1000 = np.sum(time_points_gk_1000matrices, axis=0) /3
error_points_gk_average_1000 = np.sum(error_points_gk_1000matrices, axis=0)/3
rank_points_gk_average_1000 = np.sum(rank_points_gk_1000matrices, axis=0) /3
rmse_total_points_gk_average_1000 = np.sum(rmse_total_points_gk_1000matrices, axis=0) /3

time_points_gkd_average_1000 = np.sum(time_points_gkd_1000matrices, axis=0) /3
error_points_gkd_average_1000 = np.sum(error_points_gkd_1000matrices, axis=0)/3
rank_points_gkd_average_1000 = np.sum(rank_points_gkd_1000matrices, axis=0) /3
rmse_total_points_gkd_average_1000 = np.sum(rmse_total_points_gkd_1000matrices, axis=0) /3

time_points_svds_average_1000 = np.sum(time_points_svds_1000matrices, axis=0) /3
error_points_svds_average_1000 = np.sum(error_points_svds_1000matrices, axis=0)/3
rank_points_svds_average_1000 = np.sum(rank_points_svds_1000matrices, axis=0) /3
rmse_total_points_svds_average_1000 = np.sum(rmse_total_points_svds_1000matrices, axis=0) /3

time_points_rsvd_average_1000 = np.sum(time_points_rsvd_1000matrices, axis=0) /3
error_points_rsvd_average_1000 = np.sum(error_points_rsvd_1000matrices, axis=0)/3
rank_points_rsvd_average_1000 = np.sum(rank_points_rsvd_1000matrices, axis=0) /3
rmse_total_points_rsvd_average_1000 = np.sum(rmse_total_points_rsvd_1000matrices, axis=0) /3
```

```
time_points_rsvd_opt_average_1000 = np.sum(time_points_rsvd_opt_1000matrices, axis=0) /3
error_points_rsvd_opt_average_1000 = np.sum(error_points_rsvd_opt_1000matrices, axis=0)/3
rank_points_rsvd_opt_average_1000 = np.sum(rank_points_rsvd_opt_1000matrices, axis=0) /3
rmse_total_points_rsvd_opt_average_1000 = np.sum(rmse_total_points_rsvd_opt_1000matrices,
axis=0) /3
```

B.5.5 NIHT för olika SVD-metoder på slumpgenererade matriser

10×10 matriser

```
time_points_gk_10matrices = []
error_points_gk_10matrices = []
rank_points_gk_10matrices = []
rmse_total_points_gk_10matrices = []
iterations_per_rank_gk_10matrices = []

time_points_gkd_10matrices = []
error_points_gkd_10matrices = []
rank_points_gkd_10matrices = []
rmse_total_points_gkd_10matrices = []
iterations_per_rank_gkd_10matrices = []


time_points_svds_10matrices = []
error_points_svds_10matrices = []
rank_points_svds_10matrices = []
rmse_total_points_svds_10matrices = []
iterations_per_rank_svds_10matrices = []


time_points_rsvd_10matrices = []
error_points_rsvd_10matrices = []
rank_points_rsvd_10matrices = []
rmse_total_points_rsvd_10matrices = []
iterations_per_rank_rsvd_10matrices = []


number_of_matrices10x10 = 10
matrix_size = 10

for i in range(1,number_of_matrices10x10+1):

    unif_10x10 = np.random.uniform(0, 1, (matrix_size,matrix_size))

    mean = np.mean(unif_10x10)
    std = np.std(unif_10x10)
    unif_10x10 = (unif_10x10-mean)/std

    u,s,vt = np.linalg.svd(unif_10x10)
    number_of_sing = matrix_size // 2
    s[number_of_sing:] = 0
    normal_matrix_10 = u @ np.diag(s) @ vt


    flattened_matrix = normal_matrix_10.flatten()
```

```
total_elements = flattened_matrix.size
num_to_sparsify = total_elements // 2

indices_to_zero = np.random.choice(total_elements, num_to_sparsify, replace=False)

sparsified_matrix_10 = flattened_matrix.copy()
sparsified_matrix_10[indices_to_zero] = 0

sparsified_matrix_10 = sparsified_matrix_10.reshape(normal_matrix_10.shape)

n1_test_10, n2_test_10 = sparsified_matrix_10.shape
m_test_10 = np.count_nonzero(sparsified_matrix_10)

################################################################
ranks_10 = np.arange(1,6,1)
#gk params
tolerance_gk = 10E-4
max_iter_gk = 30
bool_mask_gk = sparsified_matrix_10 != 0

tolerance_gkd = 10E-4
max_iter_gkd = 30
bool_mask_gkd = sparsified_matrix_10 != 0

tolerance_svds = 10E-4
max_iter_svds = 30
bool_mask_svds = sparsified_matrix_10 != 0

tolerance_rsvd = 10E-4
max_iter_rsvd = 30
bool_mask_rsvd = sparsified_matrix_10 != 0

################################################################
final_errors_per_r_gk_10 = []
final_time_per_r_gk_10 = []
total_duration_gk_10 = 0.0
accumulated_duration_gk_10 = []
final_rmse_per_r_gk_10 = []
iterations_per_r_gk_10 = []

final_errors_per_r_gkd_10 = []
final_time_per_r_gkd_10 = []
total_duration_gkd_10 = 0.0
accumulated_duration_gkd_10 = []
final_rmse_per_r_gkd_10 = []
iterations_per_r_gkd_10 = []

final_errors_per_r_svds_10 = []
final_time_per_r_svds_10 = []
total_duration_svds_10 = 0.0
accumulated_duration_svds_10 = []
final_rmse_per_r_svds_10 = []
iterations_per_r_svds_10 = []

final_errors_per_r_rsvd_10 = []
final_time_per_r_rsvd_10 = []
```

```
total_duration_rsvd_10 = 0.0
accumulated_duration_rsvd_10 = []
final_rmse_per_r_rsvd_10 = []
iterations_per_r_rsvd_10 = []


################################################################
for r in ranks_10:
    X_k, time_points_gk, error_points_gk, rmse_total_points_gk =
    niht(normal_matrix_10, sparsified_matrix_10, mask=bool_mask_gk, rank=r,
    tol=tolerance_gk, max_iter=max_iter_gk, svd_function="svd_gk", comments=True)

    final_errors_per_r_gk_10.append(error_points_gk[-1])
    rank_duration_gk = time_points_gk[-1]
    final_rmse_per_r_gk_10.append(rmse_total_points_gk[-1])

    final_time_per_r_gk_10.append(rank_duration_gk)
    total_duration_gk_10 += rank_duration_gk
    accumulated_duration_gk_10.append(total_duration_gk_10)


    X_k, time_points_gkd, error_points_gkd, rmse_total_points_gkd =
    niht(normal_matrix_10, sparsified_matrix_10, mask=bool_mask_gkd, rank=r,
    tol=tolerance_gkd, max_iter=max_iter_gkd,
    svd_function="svd_gk_dc", comments=True)

    final_errors_per_r_gkd_10.append(error_points_gkd[-1])
    rank_duration_gkd = time_points_gkd[-1]
    final_rmse_per_r_gkd_10.append(rmse_total_points_gkd[-1])

    final_time_per_r_gkd_10.append(rank_duration_gkd)
    total_duration_gkd_10 += rank_duration_gkd
    accumulated_duration_gkd_10.append(total_duration_gkd_10)

    X_k, time_points_svds, error_points_svds, rmse_total_points_svds =
    niht(normal_matrix_10, sparsified_matrix_10, mask=bool_mask_svds, rank=r,
    tol=tolerance_svds, max_iter=max_iter_svds,
    svd_function="svds", comments=True)

    final_errors_per_r_svds_10.append(error_points_svds[-1])
    rank_duration_svds = time_points_svds[-1]
    final_rmse_per_r_svds_10.append(rmse_total_points_svds[-1])

    final_time_per_r_svds_10.append(rank_duration_svds)
    total_duration_svds_10 += rank_duration_svds
    accumulated_duration_svds_10.append(total_duration_svds_10)


    X_k, time_points_rsvd, error_points_rsvd, rmse_total_points_rsvd=
    niht(normal_matrix_10, sparsified_matrix_10, mask=bool_mask_rsvd, rank=r,
    tol=tolerance_rsvd, max_iter=max_iter_rsvd,
    svd_function="rsvd", comments=True)

    final_errors_per_r_rsvd_10.append(error_points_rsvd[-1])
    rank_duration_rsvd = time_points_rsvd[-1]
```

```
        final_rmse_per_r_rsvd_10.append(rmse_total_points_rsvd[-1])

        final_time_per_r_rsvd_10.append(rank_duration_rsvd)
        total_duration_rsvd_10 += rank_duration_rsvd
        accumulated_duration_rsvd_10.append(total_duration_rsvd_10)


    ################################################################


    time_points_gk_10matrices.append(accumulated_duration_gk_10)
    error_points_gk_10matrices.append(final_errors_per_r_gk_10)
    rmse_total_points_gk_10matrices.append(final_rmse_per_r_gk_10)

    time_points_gkd_10matrices.append(accumulated_duration_gkd_10)
    error_points_gkd_10matrices.append(final_errors_per_r_gkd_10)
    rmse_total_points_gkd_10matrices.append(final_rmse_per_r_gkd_10)

    time_points_svds_10matrices.append(accumulated_duration_svds_10)
    error_points_svds_10matrices.append(final_errors_per_r_svds_10)
    rmse_total_points_svds_10matrices.append(final_rmse_per_r_svds_10)

    time_points_rsvd_10matrices.append(accumulated_duration_rsvd_10)
    error_points_rsvd_10matrices.append(final_errors_per_r_rsvd_10)
    rmse_total_points_rsvd_10matrices.append(final_rmse_per_r_rsvd_10)



time_points_gk_average_10 = np.sum(time_points_gk_10matrices, axis=0) /10
error_points_gk_average_10 = np.sum(error_points_gk_10matrices, axis=0)/10
rmse_total_points_gk_average_10 = np.sum(rmse_total_points_gk_10matrices, axis=0) /10

time_points_gkd_average_10 = np.sum(time_points_gkd_10matrices, axis=0) /10
error_points_gkd_average_10 = np.sum(error_points_gkd_10matrices, axis=0)/10
rmse_total_points_gkd_average_10 = np.sum(rmse_total_points_gkd_10matrices, axis=0) /10

time_points_svds_average_10 = np.sum(time_points_svds_10matrices, axis=0) /10
error_points_svds_average_10 = np.sum(error_points_svds_10matrices, axis=0)/10
rmse_total_points_svds_average_10 = np.sum(rmse_total_points_svds_10matrices, axis=0) /10

time_points_rsvd_average_10 = np.sum(time_points_rsvd_10matrices, axis=0) /10
error_points_rsvd_average_10 = np.sum(error_points_rsvd_10matrices, axis=0)/10
rmse_total_points_rsvd_average_10 = np.sum(rmse_total_points_rsvd_10matrices, axis=0) /10
```

100×100 matriser

```
time_points_gk_100matrices = []
error_points_gk_100matrices = []
rank_points_gk_100matrices = []
rmse_total_points_gk_100matrices = []

time_points_gkd_100matrices = []
error_points_gkd_100matrices = []
```

```
rank_points_gkd_100matrices = []
rmse_total_points_gkd_100matrices = []

time_points_svds_100matrices = []
error_points_svds_100matrices = []
rank_points_svds_100matrices = []
rmse_total_points_svds_100matrices = []

time_points_rsvd_100matrices = []
error_points_rsvd_100matrices = []
rank_points_rsvd_100matrices = []
rmse_total_points_rsvd_100matrices = []

number_of_matrices100x100 = 10
matrix_size=100
for i in range(1,number_of_matrices100x100+1):

    unif_100x100 = np.random.uniform(0, 1, (matrix_size,matrix_size))

    mean = np.mean(unif_100x100)
    std = np.std(unif_100x100)
    unif_100x100 = (unif_100x100-mean)/std

    u,s,vt = np.linalg.svd(unif_100x100)
    number_of_sing = matrix_size // 2
    s[number_of_sing:] = 0
    normal_matrix_100 = u @ np.diag(s) @ vt


    flattened_matrix = normal_matrix_100.flatten()

    total_elements = flattened_matrix.size
    num_to_sparsify = total_elements // 2

    indices_to_zero = np.random.choice(total_elements, num_to_sparsify, replace=False)

    sparsified_matrix_100 = flattened_matrix.copy()
    sparsified_matrix_100[indices_to_zero] = 0

    sparsified_matrix_100 = sparsified_matrix_100.reshape(normal_matrix_100.shape)

    n1_test_100, n2_test_100 = sparsified_matrix_100.shape
    m_test_100 = np.count_nonzero(sparsified_matrix_100)

    ################################################################
    ranks_100 = np.arange(1,52,7)
    tolerance_gk = 10E-4
    max_iter_gk = 30
    bool_mask_gk = sparsified_matrix_100 != 0

    tolerance_gkd = 10E-4
    max_iter_gkd = 30
    bool_mask_gkd = sparsified_matrix_100 != 0

    tolerance_svds = 10E-4
    max_iter_svds = 30
```

```
bool_mask_svds = sparsified_matrix_100 != 0

tolerance_rsvd = 10E-4
max_iter_rsvd = 30
bool_mask_rsvd = sparsified_matrix_100 != 0

#################################################################
final_errors_per_r_gk_100 = []
final_time_per_r_gk_100 = []
total_duration_gk_100 = 0.0
accumulated_duration_gk_100 = []
final_rmse_per_r_gk_100 = []

final_errors_per_r_gkd_100 = []
final_time_per_r_gkd_100 = []
total_duration_gkd_100 = 0.0
accumulated_duration_gkd_100 = []
final_rmse_per_r_gkd_100 = []

final_errors_per_r_svds_100 = []
final_time_per_r_svds_100 = []
total_duration_svds_100 = 0.0
accumulated_duration_svds_100 = []
final_rmse_per_r_svds_100 = []

final_errors_per_r_rsvd_100 = []
final_time_per_r_rsvd_100 = []
total_duration_rsvd_100 = 0.0
accumulated_duration_rsvd_100 = []
final_rmse_per_r_rsvd_100 = []


#################################################################
for r in ranks_100:
    X_k, time_points_gk, error_points_gk, rmse_total_points_gk =
    niht(normal_matrix_100, sparsified_matrix_100, mask=bool_mask_gk, rank=r,
    tol=tolerance_gk, max_iter=max_iter_gk, svd_function="svd_gk", comments=True)

    final_errors_per_r_gk_100.append(error_points_gk[-1])
    rank_duration_gk = time_points_gk[-1]
    final_rmse_per_r_gk_100.append(rmse_total_points_gk[-1])

    final_time_per_r_gk_100.append(rank_duration_gk)
    total_duration_gk_100 += rank_duration_gk
    accumulated_duration_gk_100.append(total_duration_gk_100)

    X_k, time_points_gkd, error_points_gkd, rmse_total_points_gkd =
    niht(normal_matrix_100, sparsified_matrix_100, mask=bool_mask_gkd, rank=r,
    tol=tolerance_gkd, max_iter=max_iter_gkd, svd_function="svd_gk_dc", comments=True)

    final_errors_per_r_gkd_100.append(error_points_gkd[-1])
    rank_duration_gkd = time_points_gkd[-1]
    final_rmse_per_r_gkd_100.append(rmse_total_points_gkd[-1])

    final_time_per_r_gkd_100.append(rank_duration_gkd)
    total_duration_gkd_100 += rank_duration_gkd
```

```
        accumulated_duration_gkd_100.append(total_duration_gkd_100)

        X_k, time_points_svds, error_points_svds, rmse_total_points_svds =
        niht(normal_matrix_100, sparsified_matrix_100, mask=bool_mask_svds, rank=r,
        tol=tolerance_svds, max_iter=max_iter_svds, svd_function="svds", comments=True)

        final_errors_per_r_svds_100.append(error_points_svds[-1])
        rank_duration_svds = time_points_svds[-1]
        final_rmse_per_r_svds_100.append(rmse_total_points_svds[-1])

        final_time_per_r_svds_100.append(rank_duration_svds)
        total_duration_svds_100 += rank_duration_svds
        accumulated_duration_svds_100.append(total_duration_svds_100)

        X_k, time_points_rsvd, error_points_rsvd, rmse_total_points_rsvd =
        niht(normal_matrix_100, sparsified_matrix_100, mask=bool_mask_rsvd, rank=r,
        tol=tolerance_rsvd, max_iter=max_iter_rsvd,svd_function="rsvd", comments=True)

        final_errors_per_r_rsvd_100.append(error_points_rsvd[-1])
        rank_duration_rsvd = time_points_rsvd[-1]
        final_rmse_per_r_rsvd_100.append(rmse_total_points_rsvd[-1])

        final_time_per_r_rsvd_100.append(rank_duration_rsvd)
        total_duration_rsvd_100 += rank_duration_rsvd
        accumulated_duration_rsvd_100.append(total_duration_rsvd_100)

    ################################################################


    time_points_gk_100matrices.append(accumulated_duration_gk_100)
    error_points_gk_100matrices.append(final_errors_per_r_gk_100)
    rmse_total_points_gk_100matrices.append(final_rmse_per_r_gk_100)

    time_points_gkd_100matrices.append(accumulated_duration_gkd_100)
    error_points_gkd_100matrices.append(final_errors_per_r_gkd_100)
    rmse_total_points_gkd_100matrices.append(final_rmse_per_r_gkd_100)

    time_points_svds_100matrices.append(accumulated_duration_svds_100)
    error_points_svds_100matrices.append(final_errors_per_r_svds_100)
    rmse_total_points_svds_100matrices.append(final_rmse_per_r_svds_100)

    time_points_rsvd_100matrices.append(accumulated_duration_rsvd_100)
    error_points_rsvd_100matrices.append(final_errors_per_r_rsvd_100)
    rmse_total_points_rsvd_100matrices.append(final_rmse_per_r_rsvd_100)


time_points_gk_average_100 = np.sum(time_points_gk_100matrices, axis=0) / 10
error_points_gk_average_100 = np.sum(error_points_gk_100matrices, axis=0)/10
rmse_total_points_gk_average_100 = np.sum(rmse_total_points_gk_100matrices, axis=0) /10

time_points_gkd_average_100 = np.sum(time_points_gkd_100matrices, axis=0) /10
error_points_gkd_average_100 = np.sum(error_points_gkd_100matrices, axis=0)/10
rmse_total_points_gkd_average_100 = np.sum(rmse_total_points_gkd_100matrices, axis=0) /10

time_points_svds_average_100 = np.sum(time_points_svds_100matrices, axis=0) /10
error_points_svds_average_100 = np.sum(error_points_svds_100matrices, axis=0)/10
```

```
rmse_total_points_svds_average_100 = np.sum(rmse_total_points_svds_100matrices, axis=0) /10

time_points_rsvd_average_100 = np.sum(time_points_rsvd_100matrices, axis=0) /10
error_points_rsvd_average_100 = np.sum(error_points_rsvd_100matrices, axis=0)/10
rmse_total_points_rsvd_average_100 = np.sum(rmse_total_points_rsvd_100matrices, axis=0) /10
```

1000×1000 matriser

```
time_points_gk_1000matrices = []
error_points_gk_1000matrices = []
rank_points_gk_1000matrices = []
rmse_total_points_gk_1000matrices = []

time_points_gkd_1000matrices = []
error_points_gkd_1000matrices = []
rank_points_gkd_1000matrices = []
rmse_total_points_gkd_1000matrices = []

time_points_svds_1000matrices = []
error_points_svds_1000matrices = []
rank_points_svds_1000matrices = []
rmse_total_points_svds_1000matrices = []

time_points_rsvd_1000matrices = []
error_points_rsvd_1000matrices = []
rank_points_rsvd_1000matrices = []
rmse_total_points_rsvd_1000matrices = []

number_of_matrices1000x1000 = 3
matrix_size = 1000

for i in range(1,number_of_matrices1000x1000+1):

    unif_1000x1000 = np.random.uniform(0, 1, (matrix_size,matrix_size))

    mean = np.mean(unif_1000x1000)
    std = np.std(unif_1000x1000)
    unif_1000x1000 = (unif_1000x1000-mean)/std

    u,s,vt = np.linalg.svd(unif_1000x1000)
    number_of_sing = matrix_size // 2
    s[number_of_sing:] = 0
    normal_matrix_1000 = u @ np.diag(s) @ vt

    flattened_matrix = normal_matrix_1000.flatten()

    total_elements = flattened_matrix.size
    num_to_sparsify = total_elements // 2

    indices_to_zero = np.random.choice(total_elements, num_to_sparsify, replace=False)

    sparsified_matrix_1000 = flattened_matrix.copy()
    sparsified_matrix_1000[indices_to_zero] = 0

    sparsified_matrix_1000 = sparsified_matrix_1000.reshape(normal_matrix_1000.shape)
```

```python
n1_test_1000, n2_test_1000 = sparsified_matrix_1000.shape
m_test_1000 = np.count_nonzero(sparsified_matrix_1000)

################################################################
ranks_1000 = [25,50,100,200,350,500]
#gk params
tolerance_gk = 10E-4
max_iter_gk = 50
bool_mask_gk = sparsified_matrix_1000 != 0

tolerance_gkd = 10E-4
max_iter_gkd = 50
bool_mask_gkd = sparsified_matrix_1000 != 0

tolerance_svds = 10E-4
max_iter_svds = 50
bool_mask_svds = sparsified_matrix_1000 != 0

tolerance_rsvd = 10E-4
max_iter_rsvd = 50
bool_mask_rsvd = sparsified_matrix_1000 != 0

################################################################
final_errors_per_r_gk_1000 = []
final_time_per_r_gk_1000 = []
total_duration_gk_1000 = 0.0
accumulated_duration_gk_1000 = []
final_rmse_per_r_gk_1000 = []

final_errors_per_r_gkd_1000 = []
final_time_per_r_gkd_1000 = []
total_duration_gkd_1000 = 0.0
accumulated_duration_gkd_1000 = []
final_rmse_per_r_gkd_1000 = []

final_errors_per_r_svds_1000 = []
final_time_per_r_svds_1000 = []
total_duration_svds_1000 = 0.0
accumulated_duration_svds_1000 = []
final_rmse_per_r_svds_1000 = []

final_errors_per_r_rsvd_1000 = []
final_time_per_r_rsvd_1000 = []
total_duration_rsvd_1000 = 0.0
accumulated_duration_rsvd_1000 = []
final_rmse_per_r_rsvd_1000 = []


################################################################
for r in ranks_1000:
    X_k, time_points_gk, error_points_gk, rmse_total_points_gk =
    niht(normal_matrix_1000, sparsified_matrix_1000, mask=bool_mask_gk, rank=r,
    tol=tolerance_gk, max_iter=max_iter_gk,svd_function="svd_gk", comments=True)

    final_errors_per_r_gk_1000.append(error_points_gk[-1])
    rank_duration_gk = time_points_gk[-1]
```

```
    final_rmse_per_r_gk_1000.append(rmse_total_points_gk[-1])

    final_time_per_r_gk_1000.append(rank_duration_gk)
    total_duration_gk_1000 += rank_duration_gk
    accumulated_duration_gk_1000.append(total_duration_gk_1000)


    X_k, time_points_gkd, error_points_gkd, rmse_total_points_gkd =
    niht(normal_matrix_1000, sparsified_matrix_1000, mask=bool_mask_gkd, rank=r,
    tol=tolerance_gkd, max_iter=max_iter_gkd,svd_function="svd_gk_dc", comments=True)

    final_errors_per_r_gkd_1000.append(error_points_gkd[-1])
    rank_duration_gkd = time_points_gkd[-1]
    final_rmse_per_r_gkd_1000.append(rmse_total_points_gkd[-1])

    final_time_per_r_gkd_1000.append(rank_duration_gkd)
    total_duration_gkd_1000 += rank_duration_gkd
    accumulated_duration_gkd_1000.append(total_duration_gkd_1000)


    X_k, time_points_svds, error_points_svds, rmse_total_points_svds =
    niht(normal_matrix_1000, sparsified_matrix_1000, mask=bool_mask_svds, rank=r,
    tol=tolerance_svds, max_iter=max_iter_svds,svd_function="svds", comments=True)

    final_errors_per_r_svds_1000.append(error_points_svds[-1])
    rank_duration_svds = time_points_svds[-1]
    final_rmse_per_r_svds_1000.append(rmse_total_points_svds[-1])

    final_time_per_r_svds_1000.append(rank_duration_svds)
    total_duration_svds_1000 += rank_duration_svds
    accumulated_duration_svds_1000.append(total_duration_svds_1000)


    X_k, time_points_rsvd, error_points_rsvd, rmse_total_points_rsvd =
    niht(normal_matrix_1000, sparsified_matrix_1000, mask=bool_mask_rsvd, rank=r,
    tol=tolerance_rsvd, max_iter=max_iter_rsvd,svd_function="rsvd", comments=True)

    final_errors_per_r_rsvd_1000.append(error_points_rsvd[-1])
    rank_duration_rsvd = time_points_rsvd[-1]
    final_rmse_per_r_rsvd_1000.append(rmse_total_points_rsvd[-1])

    final_time_per_r_rsvd_1000.append(rank_duration_rsvd)
    total_duration_rsvd_1000 += rank_duration_rsvd
    accumulated_duration_rsvd_1000.append(total_duration_rsvd_1000)

################################################################


time_points_gk_1000matrices.append(accumulated_duration_gk_1000)
error_points_gk_1000matrices.append(final_errors_per_r_gk_1000)
rmse_total_points_gk_1000matrices.append(final_rmse_per_r_gk_1000)

time_points_gkd_1000matrices.append(accumulated_duration_gkd_1000)
error_points_gkd_1000matrices.append(final_errors_per_r_gkd_1000)
rmse_total_points_gkd_1000matrices.append(final_rmse_per_r_gkd_1000)
```

```
    time_points_svds_1000matrices.append(accumulated_duration_svds_1000)
    error_points_svds_1000matrices.append(final_errors_per_r_svds_1000)
    rmse_total_points_svds_1000matrices.append(final_rmse_per_r_svds_1000)

    time_points_rsvd_1000matrices.append(accumulated_duration_rsvd_1000)
    error_points_rsvd_1000matrices.append(final_errors_per_r_rsvd_1000)
    rmse_total_points_rsvd_1000matrices.append(final_rmse_per_r_rsvd_1000)


time_points_gk_average_1000 = np.sum(time_points_gk_1000matrices, axis=0) /3
error_points_gk_average_1000 = np.sum(error_points_gk_1000matrices, axis=0)/3
rmse_total_points_gk_average_1000 = np.sum(rmse_total_points_gk_1000matrices, axis=0) /3

time_points_gkd_average_1000 = np.sum(time_points_gkd_1000matrices, axis=0) /3
error_points_gkd_average_1000 = np.sum(error_points_gkd_1000matrices, axis=0)/3
rmse_total_points_gkd_average_1000 = np.sum(rmse_total_points_gkd_1000matrices, axis=0) /3

time_points_svds_average_1000 = np.sum(time_points_svds_1000matrices, axis=0) /3
error_points_svds_average_1000 = np.sum(error_points_svds_1000matrices, axis=0)/3
rmse_total_points_svds_average_1000 = np.sum(rmse_total_points_svds_1000matrices, axis=0) /3

time_points_rsvd_average_1000 = np.sum(time_points_rsvd_1000matrices, axis=0) /3
error_points_rsvd_average_1000 = np.sum(error_points_rsvd_1000matrices, axis=0)/3
rmse_total_points_rsvd_average_1000 = np.sum(rmse_total_points_rsvd_1000matrices, axis=0) /3
.append(final_rmse_per_r_rsvd_1000)
```

B.5.6 SVD-metoder på Netflix-databasen

Modifierat μ_1

```
        b = 0.11
        d = 0.98
        s = 0.40

        rjus = b / (rank ** s) * (d ** i)
```

NIHT på Netflix

```
Tolerance_test = 10E-4
ranks = [1,3,5,7,10,20,50,75,90]
Max_iter_test = 1000
bool_mask_ntest = sparsified_matrix_netflix != 0

final_errors_per_r_gk = []
final_time_per_r_gk = []
total_duration_gk = 0.0
accumulated_duration_gk = []
rmse_tot_per_r_gk = []
list_of_iters_per_rank_gk = []

final_errors_per_r_gkd = []
final_time_per_r_gkd = []
total_duration_gkd = 0.0
accumulated_duration_gkd = []
rmse_tot_per_r_gkd = []
list_of_iters_per_rank_gkd = []

final_errors_per_r_svds = []
```

```
final_time_per_r_svds = []
total_duration_svds = 0.0
accumulated_duration_svds = []
rmse_tot_per_r_svds = []
list_of_iters_per_rank_svds = []


final_errors_per_r_rsvd = []
final_time_per_r_rsvd = []
total_duration_rsvd = 0.0
accumulated_duration_rsvd = []
rmse_tot_per_r_rsvd = []
list_of_iters_per_rank_rsvd = []


for r in ranks:

    X_k, time_pts, error_pts, rmse_tot, iters_per_rank =
    niht(matrix_real,sparsified_matrix_netflix, bool_mask_ntest,
    r, Tolerance_test, Max_iter_test, svd_function="svd_gk", comments=True)

    final_errors_per_r_gk.append(error_pts[-1])
    rank_duration_gk = time_pts[-1]
    rmse_tot_per_r_gk.append(rmse_tot[-1])

    final_time_per_r_gk.append(rank_duration_gk)
    total_duration_gk += rank_duration_gk
    accumulated_duration_gk.append(total_duration_gk)

    list_of_iters_per_rank_gk.append(iters_per_rank)


for r in ranks:

    X_k, time_pts, error_pts, rmse_tot, iters_per_rank =
    niht(matrix_real,sparsified_matrix_netflix, bool_mask_ntest,
    r, Tolerance_test, Max_iter_test, svd_function="svd_gk_dc", comments=True)

    final_errors_per_r_gkd.append(error_pts[-1])
    rank_duration_gkd = time_pts[-1]
    rmse_tot_per_r_gkd.append(rmse_tot[-1])

    final_time_per_r_gkd.append(rank_duration_gkd)
    total_duration_gkd += rank_duration_gkd
    accumulated_duration_gkd.append(total_duration_gkd)

    list_of_iters_per_rank_gkd.append(iters_per_rank)

for r in ranks:

    X_k, time_pts, error_pts, rmse_tot, iters_per_rank =
    niht(matrix_real,sparsified_matrix_netflix, bool_mask_ntest,
    r, Tolerance_test, Max_iter_test, svd_function="svds", comments=True)

    final_errors_per_r_svds.append(error_pts[-1])
    rank_duration_svds = time_pts[-1]
```

```
    rmse_tot_per_r_svds.append(rmse_tot[-1])

    final_time_per_r_svds.append(rank_duration_svds)
    total_duration_svds += rank_duration_svds
    accumulated_duration_svds.append(total_duration_svds)

    list_of_iters_per_rank_svds.append(iters_per_rank)



for r in ranks:

    X_k, time_pts, error_pts, rmse_tot, iters_per_rank=
    niht(matrix_real,sparsified_matrix_netflix, bool_mask_ntest, r, Tolerance_test,
    Max_iter_test, svd_function="rsvd", comments=True)

    final_errors_per_r_rsvd.append(error_pts[-1])
    rank_duration_rsvd = time_pts[-1]
    rmse_tot_per_r_rsvd.append(rmse_tot[-1])

    final_time_per_r_rsvd.append(rank_duration_rsvd)
    total_duration_rsvd += rank_duration_rsvd
    accumulated_duration_rsvd.append(total_duration_rsvd)

    list_of_iters_per_rank_rsvd.append(iters_per_rank)
```

SVT på Netflix

```
Step_size_gk  = 0.6 * n1_test*n2_test/m_test
Tolerance_gk = 10E-4
Tau_gk = 4 * np.sqrt(n1_test*n2_test)
Max_iter_gk = 50
Increment_gk = 5
bool_mask_gk = sparsified_matrix_netflix != 0

Step_size_gkd  = 0.6 * n1_test*n2_test/m_test
Tolerance_gkd = 10E-4
Tau_gkd = 4 * np.sqrt(n1_test*n2_test)
Max_iter_gkd = 50
Increment_gkd = 5
bool_mask_gkd = sparsified_matrix_netflix != 0

Step_size_svds  = 0.7 * n1_test*n2_test/m_test
Tolerance_svds = 10E-4
Tau_svds = 2.7 * np.sqrt(n1_test*n2_test)
Max_iter_svds = 30
Increment_svds = 5
bool_mask_svds = sparsified_matrix_netflix != 0

Step_size_rand  = 0.6 * n1_test*n2_test/m_test
Tolerance_rand = 10E-4
Tau_rand = 2.7 * np.sqrt(n1_test*n2_test)
Max_iter_rand = 40
Increment_rand = 5
bool_mask_rand = sparsified_matrix_netflix != 0
```

```
svt_matrix_real_gkd, time_gkd, errors_gkd, rank_gkd, rmse_tot_gkdc =
svt_thresholding_only(matrix_real, sparsified_matrix_netflix,
Step_size_gkd, Tolerance_gkd, Tau_gkd, Max_iter_gkd,
golubkahan_dc_svd,increment=Increment_gkd,
k_0=True, mask=bool_mask_gkd, comments=True )

svt_matrix_real_gk, time_gk, errors_gk, rank_gk, rmse_tot_gk =
svt_thresholding_only(matrix_real, sparsified_matrix_netflix,
Step_size_gk, Tolerance_gk, Tau_gk, Max_iter_gk,
golubkahan_svd,increment=Increment_gk, k_0=True, mask=bool_mask_gk, comments=True )

svt_matrix_svds, time_svds, error_svds, rank_svds, rmse_tot_svds =
svt_threshholding_fixrang(matrix_real,sparsified_matrix_netflix ,
Step_size_svds, Tolerance_svds, Tau_svds, Max_iter_svds,
Increment_svds, svd_function="", k_0=True, mask=bool_mask_svds, comments=True )

svt_matrix_rand, time_rand, error_rand, rank_rand, rmse_tot_rsvd =
svt_threshholding_fixrang(matrix_real,sparsified_matrix_netflix,
Step_size_rand, Tolerance_rand, Tau_rand, Max_iter_rand,
Increment_rand, svd_function="random_svd",k_0=True, mask=bool_mask_rand, comments=True )
```