NTI Gymnasiet - Nacka Strand
Teknikprogrammet-Teknikvetenskap
Gymnasiearbete 100p
HT 2023 - VT 2024

Framtagande av en potentiell kortleksblandare

Proof of Concept

Namn: Eduards Abisevs
E-post: eduards.abisevs@elev.ga.ntig.se
Namn: Leo Altebro
E-post: leo.altebro@elev.ga.ntig.se
Handledare: Niclas Lund
E-post: niclas.lund@ga.ntig.se
Examinator: Elias Lindqvist
E-post: elias.lindqvist@ga.ntig.se

Typsatt med LaTeX.
Kompilerad 17 maj 2024.

Abstract

This study presents a comprehensive analysis of three card shuffling algorithms: Riffle Shuffle, Pile Shuffle (in three variations) and Wheel Fisher-Yates Shuffle. The objective is to assess their statistical randomness, efficiency, and practical applicability in the design of a potential card shuffler machine. Large-scale simulations were performed to generate extensive shuffling datasets, with the classical poker test and STDMean test applied to assess their randomness. The practical feasibility was theorized, supported by conceptual diagrams. The results did not favor a singular method as superior in all aspects. Instead, the Pile Shuffle with 10 bins and 4 iterations emerged as the most balanced choice, offering optimal randomness and practicality for construction. While the Wheel Fisher-Yates Shuffle presents a more complex design, it uniquely facilitates the shuffling of multiple decks in a single iteration. The Riffle Shuffle, while traditional, requires further investigation to optimize its implementation.

Acknowledgments

Special thanks to our mentor, Elias Lindqvist, for his invaluable guidance and dedication to our development as writers. His expertise, passion for grammar, and rigorous approach to research and citation have profoundly impacted us, pushing our boundaries to new heights and instilling a meticulous and analytical mindset. His countless hours of feedback and support have been crucial to our growth and the success of this project.

Innehåll

Ordlista

p-värde Den uppmäta sannolikheten att nollhypotesen är sann

ApEn Approximate entropy

bibliotek Samling av färdigt byggt kod för programmerings språk Python

crate Samling av färdigt byggt kod för programmerings språk Rust

df Frihetsgrader (Degrees of freedom)

GSR Gilberts-Shanons-Reeds

matris Data i två dimensioner som en tabell

MP Medelvärde av Position

ns Nanosekunder

PK Position av Kort

PRNG Pseudoslumptalsgenerator

SD Standardavvikelse (eng. Standard deviation)

Signifikansnivå Sannolikheten att, vid hypotesprövning, förkasta nollhypotesen trots att den är sann

SOC Shuffle-O-MatiC

STDMean Standardavviklse-och medelvärde (eng. Standard deviation and Mean)

TRNG Äkta slumptalsgenerator

1 Inledning

1.1 Introduktion till ämnet

Spelbranschen är en industri som omsätter miljardbelopp och precis som alla industrier utvecklas den med resten av världen. Industrier över hela världen har som mål att hitta nya sätt att effektivisera och sänka kostnad på det arbete som utförs för att bedriva vinst, och spelbranschen har följt denna långvariga trend med till exempel online kasinon. Inom spelbranschen är kortspel vanligt förekommande, att försöka automatisera dem är därför ett logiskt steg att ta, men för en stor industri vill man vara extra säker på att allt utförs på bästa sätt. Alla sätt att blanda kort är nämligen inte lika bra, vilken sorteringsmetod som används kan ha påverkan på resultatet. Teknologins inflytande inom spelbranschen har ökar kraftigt med digitaliseringen, de största delarna av branschen bedrivs mer och mer av maskiner och algoritmer som får stor potentiellt inflytande på spelresultat, därför är det bäst att börja tänka på möjliga problem så snart som möjligt. Redan nu finns det blandningsmaskiner för kortlekar som vi inte vet någonting om; varken hur de funkar eller hur bra de är. Allt som vi vet är de är certifierade av tredjepartsföretag.

1.2 Syfte

Syftet med denna undersökning är att ta fram den hypotetiskt bästa kortblandaren utifrån två faktorer: 1) hur slumpmässigt den algoritm som den byggs efter kan blanda kort; 2) till vilken grad den potentiella maskinen byggd efter algoritmen skulle fungera i verkligheten. Med resultaten förväntas variationen i slumpmässighet för olika blandningsmetoder kunna visas upp samt kunna använda de resultaten för att komma fram till den hypotetiskt definitiva maskinen, något som är viktigt då spelbranschen handlar mycket om chans. Det är därför viktigt att se till att allt funkar på bästa sätt. I den här vetenskapliga rapporten kommer en jämförelse av hur effektivt framtagna blandningsmetoder kan fungera i en hypotetisk blandningsmaskin att utföras. Samtidigt som man kan utforska hur en dator kan göra slumpmässiga sekvenser på ett effektivt sätt med tanken att den ska tillämpas till en potentiell kortblandare. Detta är viktigt för att spelbranschen är en stor industri som handlar mycket om tur, att se till att de slumpmässiga resultaten är framtagna på bästa möjliga sätt är en essentiell del. Vi söker med hjälp av data svaret på frågan:

Utifrån slumpmässighet och effektivitet, vilken kortblandningsmetod är bäst för en potentiell spelkortsblandare som uppfyller följande:

- Fysiskt tillämpning: Hur pass väl den kan framställas i verkligheten
- Effektivitet: Utifrån mjukvaras och hårdvaras perspektiv
- Kortblandnings slumpmässighet

2 Teori

2.1 Slumpmässighet

Beräkningsteori är en del av matematik vars syfte är grundat i hur och om problem kan bli lösta på olika beräkningssätt. Beräkningsteori har flera olika grenar. En gren handlar om vad som går att bevisa inom matematik angående om nummer och funktioner är beräkneliga eller inte. Med beräknelig menas något som kan beräknas, det vill säga värderas, uppfattas eller förutses, när det kommer till matematik syftar det på att bestämma något via matematiska modeller eller processer, alltså att kalkylera.

Slumpmässighet är en egenskap som anger att något sker utan klart mönster. När något är helt slumpmässigt är det i princip omöjligt att förutse. Slumpmässighet inom matematik är byggd beräkningsteori och den delas upp i två beroende på om det gäller slumpmässighet av en bestämd mängd eller en oändlig mängd av objekt (Terwijn, 2016, s. 49–66).

2.2 Slumptalsgeneratorer

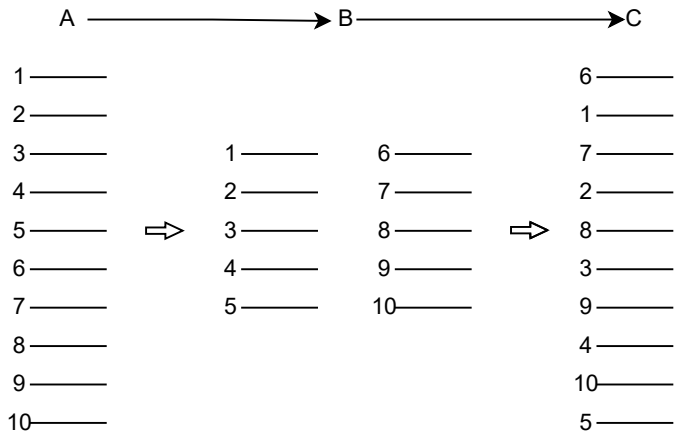
Som det nämndes i sektion 2.1 finns det olika typer av slump. Följaktligen finns det olika typer av slumptalsgeneratorer. Dem kan indelas i två typer icke-deterministiska och deterministiska så kallade *äkta slumptalsgeneratorer* (true random number generators - TRNG) respektive *pseudo slumptalsgeneratorer* (psuedo random number generators - PRNG). Icke-deterministiska processer använder sig av naturliga fenomen e.g. termisk natur, kosmisk strålning eller radioaktivt sönderfall. Därför behöver denna metod speciell utrustning. Deterministiska processer är när en slumptal algoritm simulerar en slumpmässigt händelse. Men den behöver ett start data som sedan utför ett kedja av operationer. Därför deterministiska behöver slumptalsgenerator ett frö (seed) att utgå ifrån. På så sätt kan exakt samma sekvens av slumpmässiga sekvenser återskapas om fröet är känd (Hannes, 2011).

Det finns olika områden när TRNG eller PRNG är lämpligast att använda. Enligt *random.org* sammanfattar dem dessa till följande PRNG vid simulationer och TRNG vid kasino spel (Random.org, 2024).

2.3 Bakgrund av blandingsmetoder

2.3.1 Riffle Shuffle

Riffle Shuffle är den vanligaste kortblandningsmetoden. Riffle Shuffle är utförda genom att dela upp en kortlek i två delar och sedan interfoliera dem för att få ut en blandad kortlek. Riffle Shuffle har undersökts över en lång tidsperiod. För matematiska syften angående Riffle Shuffle finns det en modell, Gilbert-Shannon-Reeds-modellen. Gilbert-Shannon-Reeds-modellen är en matematisk modell vars resultat är lika dem från Riffle Shuffle utförd av en människa. Modellen utvecklades av Gilbert och Shannon (Gilbert, 1955), och senare oberoende av Reeds 1981 i ett verk som inte publicerades (Diaconis, 2003, s. 77–79). Mycket tidigare forskning om Riffle Shuffles egenskaper bygger på Gilbert-Shannon-Reeds-modellen till exempel Bayer och Diaconis (1992), en studie vars resultat verkar medföra att Riffle Shuffle borde göras sju gånger för att få en helt slumpmässig blandning.

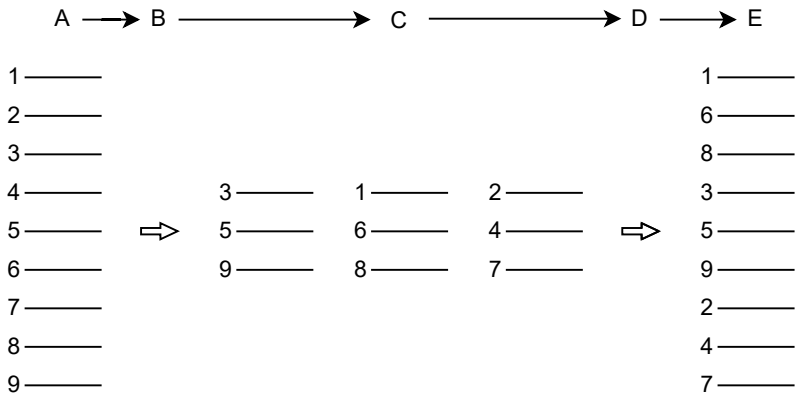


Figur 1: Steg i processen för Riffle Shuffle. Illustrationen visar de iterativa stegen från den ursprungliga högen (A), uppdelning i två högar (B), infoga dem två högarna med varandra för att bilda en ny hög. Pilar indikerar riktningen för blandningsprocessen.

2.3.2 Pile Shuffle

Pile Shuffle är en kortblandningsmetod som genomförs med fysiska kort. Processen utgår från att ett kort från kortleken läggs i en av flera olika högar. Processen försätter tills alla kort från kortleken har flyttats till en av högarna. Sedan läggs alla högar ihop i en ny kortlek. Flöde av en sådan metod visas i Figur 2.

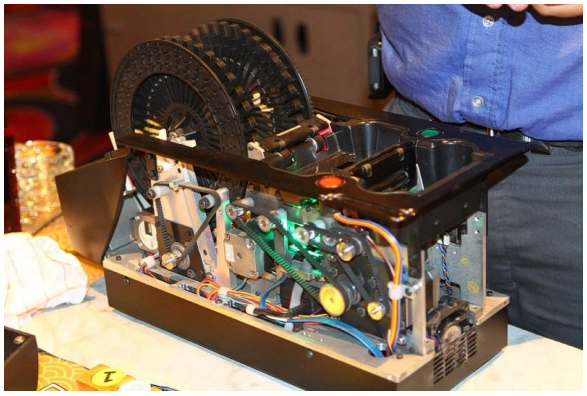
Att hitta en kortleksblandare som hade implementerat en Pile Shuffle metoden var inte enkelt uppdrag. Men det hittades en kortleksblandare som lyckades med det. Källan hittades på YouTube och var skapad av 3DprintedLife (2021). Blandningsmaskinen kallas Shuffle-o-matic (SOC). I denna YouTube video visas det upp hur en sådan maskin fungerar och hur kan denna 3D printas. Där meddelas det att kortmatningsmekanismen var svårt att implementera p.g.a. hur tunna spelkort är.



Figur 2: Steg i processen för Pile Shuffle. Illustrationen visar de iterativa stegen från den ursprungliga högen (A), genom uppdelning i högar (B), temporära högar (C), omarrangering av temporära högar (D), och tillbaka till en enda hög (E). Pilar indikerar riktningen för blandningsprocessen.

2.3.3 Inblick i en professionell kortleksblandare

Information om professionella kortleksblandare är begränsat, speciellt när det gäller vilka algoritmer som dessa använder. Det har uppfattats att detta beror på säkerhetsskäl d.v.s. att den ska inte vara tillgänglig till allmänheten. På detta sätt säkerställer och minimerar utvecklarna av dessa maskiner risken att obehöriga parter hackar dem. Men under litteraturforskning fasen hittades en gammal YouTube-video som presenterar hur en professionell kortblandningsmaskin ser ut och fungerar från insidan (Yan, 2016). En bild av en likadan blandningsmaskin hittades, se Figur 3. Ett tydligt mönster som båda maskinerna har är att utvecklarna valde att ha full kontroll av enskilda korts position när dem blandas. Det betyder att denna blandningsmaskin skulle teoretiskt sätt kunna använda vilken algoritm som helst, samt generera en riggad kortlek. Med riggad kortlek menas att ordningen är definierad i förväg. Motiveringar till detta skulle inkludera skäl av vinst men detta är inte bevisat på ett eller ett annat sätt.



Figur 3: Inblick i en isärtagen kortleksblandare.

2.4 Det klassiska pokertestet

Ett klassiskt pokertest används för att avgöra slumpmässighet i numeriska sekvenser, oftast för att testa slumptalsgeneratorer. Testet utförs genom att 3 till 5 nummer väljs ut ur en sekvens och placeras i en av sju kategorier beroende på mönstret som talen har. Mönstren är baserade på händer i poker vilket är varför testet kallas pokertest (Abdel-Rehim, Ismail och Morsy, 2014). De olika mönster som letas efter i talen visas i Tabell 1. Antalet mönster i varje kategori räknas för att få en distribution, som då jämförs med en distribution som stämmer överens med sannolikheten att få de olika mönstren. Om det totala antalet mönster är n och sannolikheten för en pokerhand i är p_i, därför borde antalet mönster i den kategori vara

Tabell 1: Vanliga kategorier till pokertest

Pokerhand	Mönster
Femtal	AAAAA
Fyrtal	AAAAB
Kåk	AAABB
Tretal	AAABC
Tvåpar	AABBC
Par	AABCD
Inget mönster	ABCDE

$$e_i = p_i * n$$

Där e_i är antalet mönster som borde matcha pokerhand i. För att bestämma om resultaten kan komma från en rättvis kortlek jämförs de resultat som fåtts av att plocka ut nummer mot de resultat som fås av sannolikheten via ett chi-två-test.

2.4.1 Chi-två-test

Inom statistik används ett *goodness-of-fit* test för att mäta hur pass väl fördelningen för den data som observerats stämmer överens med fördelningen som data förväntas ha utifrån en model. Karl Pearsons chi-två-test kan användas som *goodness-of-fit* test. Det använder chitvåfördeling. I testet jämförs ett värde χ^2 med ett kritiskt värde för att bestämma om den observerade följer den förväntade fördelningen. χ^2 beräknas enligt formeln

$$\chi^2 = \sum_{i=1}^{k} \frac{(O_i - E_i)^2}{E_i}$$

Där O_i är observerade frekvensen för en kategori av data i och E_i är förväntade frekvensen för kategorin i. Det kritiska värdet fås av antalet frihetsgrader(kategorier - parametrar) och signifikansnivån som bestämts för testet. Såsom ett viktigt aspekt i chi-två-test är att data mängderna som testas inte kan vara för små. Det rekommenderas att O_i inte är mindre en 5 (National Institute of Standards and Technology, 2023).

2.5 Programmeringsverktyg

Inom ramen för denna studie är användningen av programmeringsverktyg som Python och Rust av stor betydelse. Denna sektion inkluderar beskrivningar av alla verktyg som ska användas. Tyngden betonas av den tidigare forsknings ytterst utmärkta arbete inom dataanalys verktyg som publicerats till öppen källkod för alla att använda. För att utan dessa verktyg skulle denna studie inte vara möjlig att utföra.

Python är ett objektorienterat, dynamiskt programmeringsspråk med fokus på läsbarhet och enkel syntax (Van Rossum och Drake, 2009). Python har etablerat sig som ett fundamentalt språk inom datavetenskap p.g.a. sitt stora ekosystem av kraftfulla bibliotek.

NumPy är utvecklat av Travis Oliphant och ett globalt team av bidragsgivare och är standardbiblioteket för vektoriserad aritmetisk beräkning i Python. Dess effektiva hanteringsmetoder på stora datamängder och matematiska operationer gör det till förstahandsval för dataanalys vid vetenskaplig forskning (Harris m. fl., 2020).

SciPy, en stor samling av öppen källkod-programvara för matematik och vetenskap. Det är resultatet av samarbete mellan hundratals forskare och bidrar till att göra komplexa matematiska beräkningar tillgängliga och effektiva (Virtanen m. fl., 2020). E.g. funktioner för utförandet av chi-två-test.

Numba är ett kompilatorbibliotek som översätter Python-kod till snabb maskinkod som körs på egen tråd. Detta projekt, som drivs av Anaconda, Inc., möjliggör betydande hastighetsförbättringar för dataintensiva implementationer förutsatt att det används fundamentala datatyper och operationer (Lam, Pitrou och Seibert, 2015).

Matplotlib, skapat av John D. Hunter och nu underhållet av en stor utvecklargemenskap, är det ledande biblioteket för datavisualisering i Python. Det gör det möjligt att enkelt skapa en mängd olika grafik och plotter, vilket är kritiskt för analys och presentation av data Hunter (2007).

Rust är ett kompilerat statisk programmeringsspråk som fokuserar på minnessäkerhet, samt minneseffektivitet och parallellism. Rust är känt för

sina avancerade funktioner som ägarskapssystemet (ownership), vilket hjälper till att förhindra minnesläckor och tillåter säker minneshantering utan en skräpsamlare (garbage collector) (N. D. Matsakis och Klock II, 2014). P.g.a. dessa egenskaper har det ökats popularitet av användning av Rust i inbyggda system som ett motkandidat till programmeringsspråk C, detta undersöktes på djupet av Sharma m. fl. (2023).

Rand crate är för pseudoslumptal generering med enkel användning via *Rng* trait. Samt säker och snabb behandling av fröet med *thread_rng*, den använder algoritm ChaCha20 (The Rand Project Developers, 2022).

Rayon crate är för att konvertera sekventiella iterationer i Rust till parallellt och säker exekvering (N. Matsakis och Stone, 2022).

Genom att använda dessa verktyg har denna studie kunnat utföra omfattande simuleringar och dataanalyser på ett effektivt sätt. Deras tillgänglighet och prestanda har varit avgörande för studiens framgång.

3 Metod

Metoden för denna studie kan indelas i tre huvudområden: (i) Framtagande av blandningsalgoritmerna; (ii) Simulering av kortblandningsprocesser; (iii) Statistisk analys av insamlad data.

Inledningsvis kommer framtagande av blandningsalgoritmerna fokusera på utvecklingen av specifika kortblandningsmetoder. Detta inkluderar både etablerade metoder som Riffle Shuffle (2.3.1), samt nyare, innovativa tillvägagångssätt som Pile Shuffle (2.3.2). Såsom anpassning av design av en professionell kortleksblandare (2.3.3). Vilka valts ut baserat på deras potential i en potentiell kortleksblandare. Dessa algoritmer är kritiska för bedömningen av kortblandningsmetodens slumpmässighet och effektivitet, vilket är kärnan i studiens frågeställning.

Målsättningen är att djupgående undersöka kortblandningsmetodernas fysiskt tillämplighet, dess effektivitet och slumpmässighet. Att utföra simulationer valdes eftersom att blandningsalgoritmerna (processmässigt) skulle vara snarlika till dess potentiella kortleksblandare. Därmed skulle det genereras mer tillförlitliga testresultat. Det är därför simulationsdelen innefattar en kvantitativ metodik för att reducera den inneboende slumpvariationen i de simulerade blandningsalgoritmerna. Det åstadkoms genom att generera en bestämd mängd av kortleksblandningar, där datamängderna uppfyller kravet ifrån statistiska metoder. Dessa statistiska analysmetoder omfattar det klassiska pokertestet (2.4), med resultat given av chi-två-testet (2.4.1). Det ingår även standardavvikelse- och medelvärdestest (STDMean), vilket ger ett närmare inblick i korts variation i specifika positioner i kortleken. Tillsammans dessa tester ger en tillfredsställande indikation på blandningsmetodens slumpmässighet.

Genom att välja denna metodik syftar studien till att utföra en omfattande kvantitativ analys som inte enbart bedömer algoritmernas teoretiska effektivitet men även dess praktiska tillämplighet i en potentiell kortleksblandare. Detta tillvägagångssätt möjliggör en detaljerad utvärdering av varje algoritm, dess implementering och slutliga prestanda, vilket är avgörande för att uppnå studiens mål. Mer detaljerad beskrivning av specifika algoritmerna, simuleringar av kortblandningar och statistiska utvärderingar presenteras i kommande underrubriker.

För intresserade parter finns det källkod för algoritmernas implementation, programmet som användes för simulation, samt implementationen av analysmetoderna på Github se Bilaga A.

3.1 Testmiljö

I valet av testmiljö prioriterades operativsystemets kompatibilitet med de utvecklingsverktyg som användes. Linux som operativsystem valdes på grund av dess robusta stöd för programmeringsmiljöer och breda stöd för mjukvaruutvecklingsverktyg. Dessutom erbjuder operativsystem Linux bättre kontroll över systemsresurser, vilket är avgörande för att uppnå följdriktiga och tillförlitliga testresultat. Se Tabell 2 för testmiljöns specifikationer.

Typ	Specifikation
Processor	AMD Ryzen 5 3600 6 cores / 12 threads 3.6 GHz
RAM	15.93 GB
Hårddisk	KINGSTON SA400S3, 447 GB
Operativsystem	`Arch Linux x86_64,` `Linux kernel 6.6.9-arch1-1`

Tabell 2: Testmiljö med Linux som operativsystem.

3.2 Framtagande av blandningsalgoritmerna

I detta avsnitt presenteras processen av framtagande och detaljerna kring varje algoritms kod och dess möjliga implementering i en potentiell kortleksblandare.

Algoritmerna implementerades i programmeringsspråket Rust version 1.73.0. Rust valdes p.g.a. dess höga abstraktioner med närmare tillgång till systemresurser (för detaljer se avsnitt 2.5). Eftersom att i en fysisk maskin skulle blandningsalgoritmen köras i ett inbyggd miljö det vill säga i ett system med begränsade resurser. Därmed skapas på så sätt implicit en mer likvärdig miljö till en potentiell kortblandningsmaskins miljö för efterföljande simulationer.

Kompromissen som togs i implementation av algoritmerna är att pseudoslumptalsgeneratorn (PRNG) som används i denna studie kommer ifrån Rand crate som har inbyggt implementation av ChaCha20 (se sektion 2.2 för detaljer). Som tidigare nämnts behöver ChaCha20 mer systemresurser och därför är inte tillgängligt i ett inbyggt system. Men valet av att använda denna togs för att i simulerings miljö där PRNG behövs successivt i små tids intervaller kommer den att generera bättre pseudoslumptal än en PRNG som är anpassad till inbyggda system. Dessutom kommer den att generera mer trovärdiga kortblandningar som är mindre influerade av dess underliggande PRNG implementation.

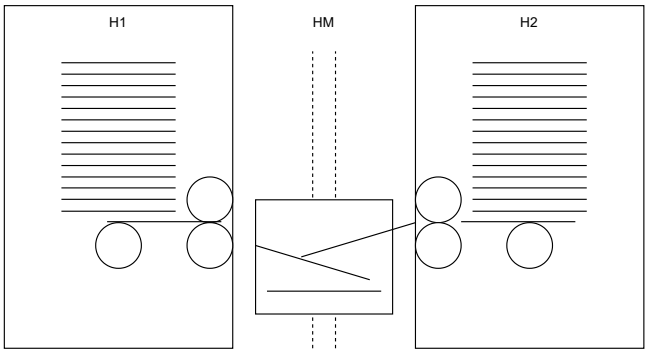
3.2.1 Anpassning av Riffle Shuffle

Första blandningsmetoden som anpassades var den matematiska Gilbert-Shannon–Reeds modell till en praktisk Rust kod (för djupgående bakgrund om modellen se sektion 2.3.1). För enkelhetens skull ska denna metod betecknas GSR Riffle Shuffle eller enbart GSR. Motiveringen av att utforska denna var p.g.a. dess lockande egenskaper som hur den matematiska modellen avspeglar människans kortblandningsprocess. Därför anpassades modellen för att utforska om den kan vara ett lämpligt kandidat till den potentiella kortleksblandaren.

Den matematiska formeln för GSR distribution är anpassad på rad 6 se Algoritm 1. För att inte distrahera med tekniska utmaningar finns implementationen i Rust i Bilaga B.

Algoritm 1 GSR Riffle Shuffle pseudokod

1: $H1 \leftarrow OD$ ▷ Alla kort från OD (Original Deck) är placerade i $H1$
2: $H2 \leftarrow \frac{H1}{2}$ ▷ $H2$ får hälften av kort (i sekvensiell ordining)
3: $HM \leftarrow tomlista$ ▷ Mittersa högen
4: **while** $H1$ eller $H2$ inte är tomma **do**
5: $r \leftarrow$ slumpmässigt tal mellan 0.0 och 1.0 ▷ PRNG för att få ett slumpmässigt tal
6: **if** $r \leq$ (längden av $H1$)/(längden av $H1 +$ längden av $H2$) **then**
7: flytta ett kort från $H1$ till HM
8: **else**
9: flytta ett kort från $H2$ till HM
10: **end if**
11: **end while**
12: flytta alla kort från HM tillbaka till $H1$



Figur 4: Ett illustrativt blandningsprocess diagram av Riffle Shuffle fundamentala mekaniska komponenter till den potentiella kortleksblandaren. Med 3 stycken högar $H1$, HM respektive $H2$. Där sträckor representerar kort, cirkelformen representerar en roterande kortmatningsmekanism och mittersta högen är en höjbar mekanism i y-led.

Potentiell fysisk implementation: Anpassnings metodiken av GSR inledes med att det skissades upp en potentiell fysisk implementation av Riffle Shuffle utifrån algoritmen, se Figur 4. Detta är ett simplifierad version. För att t.ex. det borde finnas ett mekanism att flytta kortet från HM till $H1$ och $H2$. komplikationer med denna design kan inkludera tidsplanerings problem d.v.s. att kortet från $H1$ träffar kortet från $H2$ på vägen till HM.

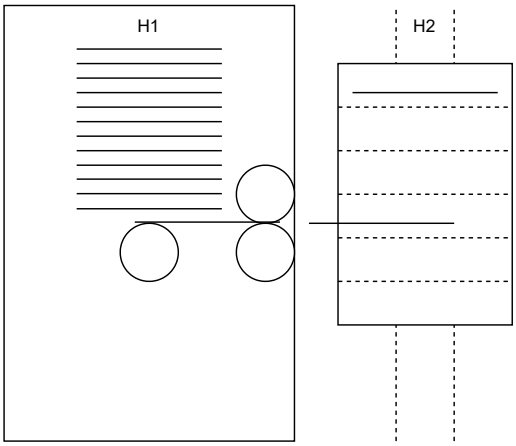
3.2.2 Anpassning av Pile Shuffle

Den andra blandningsmetoden som anpassades var Pile Shuffle. För det första är den metoden designmässig annorlunda till Riffle Shuffle. Den har endast en roterande kortmatningsmekansim samt enbart två stora delar, se Figur 5. Inspirationen för den potentiella designen kom ifrån 3DprintedLife (2021). Abbreviation till denna metod ska vara SOC Pile Shuffle eller enbart SOC. Denna metod har två varierande inställningar som inkluderar 1) antal fack n och 2) maximalt antal kort per fack M. I denna studie utforskades både SOC samt två till variationer av denna Six Pile Shuffle respektive Ten Pile Shuffle. Med följande inställningar: **SOC Pile Shuffle** med $n = 8$ och $M = 10$. Sedan utforskades hur påverkas slumpmässighet med färre fack därmed fysiska designen kan vara mindre. Detta blev till algoritmen som betecknas till **Six Pile Shuffle** med $n = 6$ och $M = 10$. Sist undersöktes om vad sker om inställningar blir till $n = 10$ och $M = 10$ denna algoritm kallas till **Ten Pile Shuffle**.

Algoritm 2 Pile Shuffle pseudokod

Indata: En kortlek $H1$
Utdata: Blandad kortlek $H1$

$n \leftarrow num$ ▷ Antal fack
$M \leftarrow num$ ▷ Max antal kort per facket
$H2 \leftarrow$ lista av n tomma listor
for *kort* i $H1$ **do**
    loop
        $sf \leftarrow$ slumpmässigt fack index mellan 0 och $(n-1)$ ▷ slumpmässigt fack (sf)
        if längden av $H2[sf] < M$ **then**
            Lägg *kort* i $H2[sf]$
            break
        end if
    end loop
end for
$H1 \leftarrow$ sammanfoga alla facken i $H2$ ▷ Samla ihop alla fack till en kortlek



Figur 5: Ett illustrativt diagram av kortblandningsprocessen med Pile Shuffle metoden. Fundamentala mekaniska komponenter till den potentiella kortleksblandaren, där sträckor är kort, cirkelformen är roterande kortmatningsmekanismen. Med två högar, $H1$ och $H2$. Där $H1$ har plats för en kortlek medans $H2$ har n antal fack med maximalt M antal kort per fack.

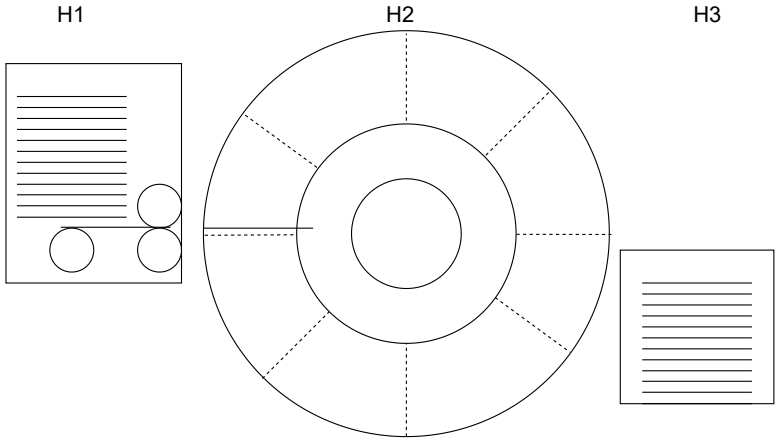
3.2.3 Anpassning av design av professionell kortleksblandare

Tredje blandningsmetoden som skapades var inspirerad av en professionell kortleksblandare. P.g.a. denna design har den fack för varje kort därmed anpassades det Fisher-Yates Shuffle. D.v.s. först blandades en lista med index med Fisher-Yates algoritmen och sedan flyttades kortet från $H1$ till $H2$ baserat på slumpmässigt index se Figur 6. Denna metod skulle teoretiskt sätt kräva endast en iteration för att blanda en kortlek.

Algoritm 3 Wheel Fisher-Yates Shuffle pseudokod

Indata: En kortlek $H1$
Utdata: Blandad kortlek $H3$

$I \leftarrow$ lista med index 0 till 51 ▷ Skapa en lista med index
FisherYatesShuffle(I) ▷ Blanda indexen slumpmässigt
$H2 \leftarrow$ ny lista med 52 platser ▷ Skapa 'wheel' med plats för varje kort
for $i \leftarrow 0$ till längden av $D-1$ **do**
    $idx \leftarrow I[i]$ ▷ Välj ett slumpmässigt index från I
    $H2[idx] \leftarrow D[i]$ ▷ Placera kortet i 'wheel' baserat på slumpindex
end for
$H3 \leftarrow H2$ ▷ Uppdatera $H3$ med den nya ordningen från 'wheel'



Figur 6: Ett illustrativt diagram av kortblandningsprocesen med Fisher-Yates metoden. Fundamentala mekaniska komponenter till den potentiella kortleksblandaren, där sträck är kort, cirkelformen är roterande kortmatningsmekanismen. Med två högar, $H1$ och $H3$. Där $H2$ har ett fack för varje kort, d.v.s. 52 stycken. Och $H3$ är plats till en blandad kortlek.

3.3 Simulering av kortblandningsprocesser

Simulations- och blandningsalgoritmerna implementerades i programmeringsspråket Rust version 1.73.0. Rust valdes p.g.a. dess robusta stöd för abstraktioner utan bekostnad.

Simulationen avspeglar hur ett fysiskt kortleksblandare skulle fungera. Därför togs det valet att alla simulationer ska ha ett och samma utgångspunkt. Därmed i analysen kan det jämföras olika metoder beroende på deras iterationer. D.v.s. ett verkligt scenario simulerades vart ett helt ny oöppnad kortlek skulle placeras i den potentiella kortleksblandare. Den ordning kallas för ett fabriksordning för en standard 52-kortlek. Där korten är ordnade efter sin färg i sekventiell ordning från två till ess (utan jokrarna). Matematiskt kan detta ordningen beskrivas som mängden $\{x \in \mathbb{N}, 0 \leq x \leq 51\}$, där x representerar en enskild kort och vart ordningen spelar roll.

För att bestämma datamängden d.v.s. antal kortlekar per en simulation. Användes det rekommendationen för chi-två-testet. Där NIST redogör för att minsta förekommande kategorin för pokertest borde inte vara mindre än 5 stycken (se sektion 2.4.1). I poker är den mest sällsynta pokerhanden Royal Flush och det finns 4 stycken av dessa i en standardkortlek. Den teoretiska sannolikheten att få 5 Royal Flushes per m kortdelningar kan approximeras på följande sätt:

$$P(\text{Kunglig färgstege}) = \frac{\binom{4}{1}}{\binom{52}{5}} = \frac{4}{2\,598\,960} \approx \frac{1}{649\,740}$$

Resultatet skalas med faktor av 5: $m = P(\text{Kunglig färgstege})^{-1} \times 5 = 3\,248\,700$. Den resulterande värden på m används som den absoluta längden av datamängden i simulationen. För att visualisera detta, låt D vara en matris med 52 kolumner (antal kort i standardkortleken) och m antal rader (längden av datamängden), och där x är en av talen ur den tidigare definierade mängden.

$$D = \begin{bmatrix} x_{0,0} & x_{0,1} & x_{0,2} & \cdots & x_{0,51} \\ x_{1,0} & x_{1,1} & x_{1,2} & \cdots & x_{1,51} \\ x_{2,0} & x_{2,1} & x_{2,2} & \cdots & x_{2,51} \\ \vdots & \vdots & \vdots & & \vdots \\ x_{m,0} & x_{m,1} & x_{m,2} & \cdots & x_{m,51} \end{bmatrix}$$

Via experimentell metodik valdes det att utföra 15 iterationer per algoritm. En iteration definieras till att kortleken blandas successiv på föregående blandning. Detta gjordes för att senare kunna jämföra hur algoritmens slumpmässighet påverkas av ökande antal iterationer. Dessutom kan detta valet motiveras av att GSRs matematiska modellen visar sig vara mest effektiv vid 7 och 11 iterationer (se sektion 2.3.1). Detta antal iterationer kunde ske p.g.a. att simulationsprocessen effektiviserades med Rayon crate. Med detta verktyg utfördes programmet i parallellt exekvering.

För att förutspå hur mycket RAM och hårddisk minne skulle eventuellt användas vid lagring och dataanalys. Räknades tyngden av vad datamängd ska väga $161MB$ vid användning av 8-bit datatypen.

$$\frac{\text{totala bytes}}{\text{megabyte (MB)}} = \frac{52 \times m}{1024 \times 1024} = \frac{52 \times 3\,248\,700}{1024 \times 1024} = \frac{168\,932\,400}{1\,048\,576} \approx 161MB$$

Detta steget behövdes för att ta ett informativt beslut senare. D.v.s. hur data ska analysers och eventuella anpassas.

Som det tidigare nämndes utfördes simulationen parallellt, varje algoritm exekverades på egen tråd, oberoende av varandra. Simulationen utfördes i följande steg: 1) Algoritmerna sparades i en lista; 2) En lista av kortlekar, motsvarande matrisen D med m antal kortlekar, initierades till fabriksordningen; 3) Algoritmerna itererades i en nästlad loop från $i = 1$ till $i = 15$; 4) Ledtidsmätaren sätts igång; 5) Denna iterationsvariabel i användes för att iterera över listan av

kortlekar och utföra blandningen i gånger; 6) Efter varje iteration i adderades ledtiden; 7) Efter avklarad iteration i medelvärde av ledtiden kalkylerades och sparades i en csv fil; 8) Resulterande blandade kortlekarna plattades ut till en endimensionell lista; 9) Denna endimensionella lista sparades som en binärfil för statistisk analys.

3.4 Statistisk analys av insamlad data.

Denna sektion handlar om hur statistiska metoder som Pokertest och STDMean test implementerades för att analyser slumpmässighet av valda och simulerade algoritmerna. Statistiska tester utfördes med Python version 3.11. Python valdes p.g.a. dess breda användning vid statistisk analys (se sektion 2.5).

Efter simulationen, genererade mängderna laddades in i Python-program och sedan återställdes tillbaka till formen av matrisen D från sektion 3.3 med hjälp av NumPy för vidare bearbetning. Analysmetoderna delar inladdad data p.g.a. optimeringsskäl som e.g. spara minne och minimera initial exekveringstid.

På grund av begränsad kunskap inom avancerad statistisk matematik, även om metoder som Approximate EntropY (ApEn) skulle kunna erbjuda ytterligare insikter i slumpmässighets analys (Delgado-Bonal och Marshak, 2019), valdes det att inte använda dem i denna studie. Därför ligger fokus på en annan gren av slumpmässighet baserad på sannolikhetslära.

3.4.1 Implementation av klassiska pokertestet

Syfte med denna metod är att utföra preliminär gallring av algoritmernas iterationer som har betydande avvikelser från dem förväntade värdena av en slumpmässigt blandningsalgoritm.

Klassiska pokertest utgår ifrån att man karaktäriserar typer av mönster till pokerhänder och kalkylerar den absolut värde av χ^2 i detalj beskrevs teorin bakom chi-två-test i sektion 2.4. Klassiska pokertestet i denna studie baseras på faktumet att simuleringar utfördes med en standard kortlek. Därför valdes det att anpassa chi-två-testet att använda alla pokerhänder. Matematiken, speciellt kombinatoriken att räkna ut sannolikheter och frekvensen av alla pokerhänder är relativt svår uppdrag därför adopterades det uträkningar ifrån studie av Armstrong (2006), se Tabell 3. Detta metodiken medföljde med mer komplex karakterisering av pokerhänder än när man använder chi-två-testet till att e.g. testa slumptalsgeneratorer. Motiveringen till komplexiteten var att i denna studie utforskas särskilt kortlekar till spel varav kombinationer av kort ofta spelar betydande roll och därför var det särskilt viktigt att anpassa metoden för slumpmässighets analys till hur den potentiella kortleksblandaren skulle fungera i drift.

Tabell 3: Namn på alla pokerhänder och dess relativ mönster. Antal_1 är teoretisk framkommande kombinationer med en standard kortlek. Antal_2 är faktoriserade kombinationer d.v.s. $\text{Antal}_2 = \text{Antal}_1 \times 1.25$

Pokerhand	Exempel mönster	Antal_1	Antal_2
Royal Flush	$A\spadesuit\ K\spadesuit\ Q\spadesuit\ J\spadesuit\ 10\spadesuit$	4	5
Färgstege	$K\spadesuit\ Q\spadesuit\ J\spadesuit\ 10\spadesuit\ 9\spadesuit$	36	45
Fyrtal	$A\spadesuit\ A\heartsuit\ A\diamondsuit\ A\clubsuit\ K\spadesuit$	624	780
Kåk	$A\spadesuit\ A\heartsuit\ A\diamondsuit\ K\clubsuit\ K\spadesuit$	3 744	4 680
Färg	$K\spadesuit\ Q\spadesuit\ J\spadesuit, 10\spadesuit\ 8\spadesuit$	5 108	6 385
Stege	$K\spadesuit\ Q\heartsuit\ J\diamondsuit\ 10\clubsuit\ 9\spadesuit$	10 200	12 750
Triss	$A\spadesuit\ A\heartsuit\ A\diamondsuit\ K\clubsuit\ Q\spadesuit$	54 912	68 640
Två par	$A\spadesuit\ A\heartsuit\ K\diamondsuit\ K\clubsuit\ Q\spadesuit$	123 552	154 440
Ett par	$A\spadesuit\ A\heartsuit, K\diamondsuit\ Q\clubsuit\ J\spadesuit$	1 098 240	1 372 800
Högt kort	$A\spadesuit\ Q\heartsuit\ J\diamondsuit\ 5\clubsuit\ 4\spadesuit$	1 302 540	1 628 175
	Summan:	2 598 960	3 248 700

Kategoriseringsprocessen till att få en hand till en pokerhand började med att utnyttja dem vektoriserad aritmetik funktionaliteterna NumPy har. Med denna funktionen valdes det på ett effektivt sätt ut fem kort i.e. en hand och kördes igenom en kategoriserings funktion om denna senare. De fem korten valdes ut på ett speciellt sätt för att göra denna likvärdig verkligheten. D.v.s. pokerspel med 2 spelare. Där valdes det kort med index 0, 2, 5, 6 och 7 (spelare-ett: 2 kort i handen, spelare-2 kort förkastas och 3 sista kort är flop) med NumPy funktionalitet kördes kategoriserings funktionen radmässigt.

Kategoriserings funktion: I standardkortlek finns det 52 kort med fyra olika färger (Spader, Hjärter, Ruter och Klöver) och 13 valörer (2, 3, 4, 5, 6, 7, 8, 9, 10, Knekt, Dam, Kung, Ess). Konvertering av tidigare definierade mängden av kort x, i.e. (0, 1, 2, ..., 51) till kategoriserade kort utfördes. För att optimera kategoriseringsproccessen användes Numba för att skapa en ny tråd, detta kunde enbart göras p.g.a. att det användes NumPy 8-bit datatyp, som definierades tidigare. Efter att en ny tråd skapades fortsattes processen på följande sätt:

$$\text{valör}(x) = x \mod 13 \tag{1}$$

$$\text{färg}(x) = \left\lfloor \frac{x}{13} \right\rfloor \tag{2}$$

Där valör(x) ger värde från 0-12 och färg(x) ger värde från 0-3. För dem intresserade av specifika kod detaljerna se Bilaga A. Fortsättningsvis gav denna funktion tillbaka ett värde från 0-9 som indikerar ett pokerhand ifrån Tabell 3, där 0 är Högt kort och 9 Royal Flush. För att denna funktionen kördes kolumnmässigt över hela datamängden var denna resulterande lista med antal observerade pokerhänder från varje rad, låt denna lista betecknas till O (Observerat frekvens). I bästa fall skulle denna lista fått likadana värdena som kolumnen Antal_2 i Tabell 3.

Chi-två-test: Inställningar till testet var $\alpha = 0.05$ (Signifikansnivå), df $= 10 - 1 = 9$ (frihetsgrader), där 10 är den antal pokerhänder givna av Tabell 3. Sedan beräknades gränsvärde med SciPy biblioteks funktionen $Chi2.ppf(1-\alpha, df)$. Sedan används det *chisquare* funktionen ifrån SciPy för att beräkna p-värde och χ^2 värde. För att kalkylera denna användes tidigare genererade listan O och dem förväntade värden given av kolumnen Antal_2 i Tabell 3. Resulterande värdena på p-värde och χ^2 sparades i en csv fil för redovisningen i Resultat sektionen.

3.4.2 Implementation av STDMean testet

Syfte med denna analysmetod är att utforska hur enskild kort rör sig genom kortleken när den blandas. Därmed avgöra om algoritmen har eller inte har en tendens att blanda specifika delar av kortleken mer än dem andra. Det här är i sin tur särskilt viktigt då på positionen på det första korten har en betydande roll i spelets rättvisa. Snedvridningen kan alltså potentiell indikera logiska fel i implementeringen av en algoritm. I initiala tester användes STDMean testet för att upptäcka logiska fel. Detta hände i den första versionen av implementation av GSR Riffle Shuffle, där det på fel sätt användes indexering i iterationen, felet upptäcktes med hjälp av denna analysmetoden.

För att förstå metoden är det relevant att återbesöka konceptet med matrisen D i sektion 3.3. Därför att det utnyttjade NumPy vektoriserad aritmetik i symbios med dess inbyggda funktioner som medelvärde och standardavvikelse. Både funktionera utfärds kolumn vis på m antal rader. Denna resultatet användes sedan till att rita punktdiagram av Medelvärde av Position (MP) på y-led och Position av Kort (PK) på x-led med hjälp av matplotlib. Standardavvikelse av PK visas som vertikal symmetrisk linje ifrån PK punkten. Den teoretiska medelvärdet i en slumpmässigt blandning borde ligga runt $51 \div 2 = 25.5$ och den experimentellt testade SD ifrån PK ≈ 15.

4 Resultat

Totalt 50 simulationer utfördes. 15 simulationer per algoritm för att jämföra samma algoritm med olika antal iterationer, varje algoritm har alltså resultat som motsvarar utförsel med iterationer 1-15. De mest relevanta resultaten av simulationerna plockades ut och presenteras här. Resultat av pokertest samt medelvärde för algoritmers ledtid presenteras i tabellen nedan. Resultaten av STDMean testet visas i Bilaga E. Vilket kort som en punkt i diagrammet representerar ges av punktens position på x-axeln, medelvärde är avläst från punktens position på y-axeln och standardavvikelse är givet av den symmetriska vertikala linjen som går genom punkten.

GSR Riffle Shuffle: Pokertestet visade en varierande slumpmässighet med första giltiga resultat vid iteration 7, efter det försämrades slumpmässigheten. Speciellt iteration 9 som översteg gränsvärdet på 16.92 men efter denna iteration förbättrades resultaten igen. Ledtiden visade ett stabilt värde men efter iteration 9 minskade ledtiden med ca 15% relativt till iteration 3 (se Tabell 4a). Resultat av STDMean testet visade en stigande trend av medelvärde av position (MP) över position av kort (PK) samt dess standardavvikelse (SD) var jämnt för alla PK. Iteration 3 visade ett stigande trend (se Bilaga E 8) men vid iteration 7 normaliserades denna till det teoretiska beräknade värdet av MP = 25.5 (se Bilaga E Tabell 10).

Six Pile Shuffle: Iteration 1 och 2, pokertestet visade att χ^2 värdena låg över gränsvärden. Iteration 3 visade det första giltiga resultatet. Vid nästkommande iterationer låg gränsvärden inom normen, med ingen tydlig trend. Vid ökade iterationer ledtidenerna verkar varit konstanta med minimal variation på ca +8% mellan iteration 1 och 7 (se Tabell 4b). STDMean testet vid iteration 1 visade värdena av stor betydelse, var PK 0-2 hade låg SD värde ca 5.4 vid PK 1 och ca SD 9.0 vid PK 2 d.v.s. värden ökade i det här intervallen med SD ca 2.0 (se Bilaga E Tabell 11). Sedan dess värdena varit varierande som liknade sinusfunktion. Men vid PK 49-51 visades det exakt likadant mönster som i PK 0-2. Vid iteration 2 plattades värdena men vart tionde PK hade avvikelser i MP (Bilaga E Tabell 12). Sedan vid iteration 3, detta mönstren minskades (Bilaga E Tabell 13). Sedan vid iteration 4 plattades MP värdena helt och var vid den teoretiska värden av MP (Bilaga E Tabell 14).

SOC Pile Shuffle: Från pokertestet, första giltiga resultat visades vid iteration 2. Medans i iteration 3 överstig gränsvärden medans dem successiva iterationer låg χ^2 värdena inom den kalkylerade gränsvärden. Ledtiden från iteration 7 visade likadan variation på +8% relativt till iteration 1 (se Tabell 4c). Resultatet av STDMean testet iteration 1 visade samma mönster som i Six Pile Shuffle. För PK 0-2 och 49-51 men SD av PK 0 och 51 var med ca 2.0 större än i Six Pile Shuffle (Bilaga E Tabell 15). Medans PK från 15-36 hade MP som låg runt 25.4. Iteration 2 hade synlig avvikelse av MP för PK 0 och 51. Annars enbart PK 10 och 41 hade synlig avvikelse från den teoretiska MP (Bilaga E Tabell 16). Iteration 3 hade endast första och sista PK som hade synlig avvikelse från MP (Bilaga E Tabell 17). Medans vid iteration 4 låg alla MP på den teoretiska värden (Bilaga E Tabell 18).

Ten Pile Shuffle: Iteration 1 enligt resultatet av pokertestet visade en icke slumpmässigt iteration. Med första giltiga resultat var vid iteration 2. Dessutom iteration 2 till 5 visade stabila χ^2 värden runt 5.28 (medelvärde av dess 4 iterationer). Ledtiden hade likadan trend som föregående Pile Shuffles d.v.s. ökat värde på ca 8% från iteration 1 till 7 (se Tabell 4d). STDMean testet vid iteration 1 visade likadan SD mönster för PK men vid högre MP värdena än SOC Pile Shuffle (Bilaga E Tabell 19). Samt iteration 2 hade likadan mönster (Bilaga E Tabell 20). Medans iteration 3 hade väldigt liten avvikelse i PK 0 och 51 (Bilaga E Tabell 17). Vid iteration 4 var alla MP värdena vid den teoretiska värden (Bilaga E Tabell 18).

Wheel Fisher-Yates Shuffle: Alla iterationer var inom gränsvärden d.v.s. hade giltiga resultat enligt pokertestet. Men vid iteration 3 och 6 χ^2 värdena var relativt lägre än dem andra. Ledtiden från iteration 1 till 7 hade ett ökat värde på ca 9% (se Tabell 4e). STDMean testet visade att dess MP låg vid den beräknade värde redan vid första iteration såsom vid samtliga successiva iterationer (se Bilaga E Tabell 23).

Tabell 4: Resultatet från den klassiska pokertestet: Gränsvärde räknat ut till 16.92 vid en signifikansnivå (α) på 0.05 och med 9 frihetsgrader (df). Vart ledtiden representerar medelvärdet för en kortleksblandning per iteration.

(a) GSR Riffle Shuffle

Iteration	χ^2	p-värde	Ledtid [ns]
3	233941.28	0	1681
4	7077.30	0	1687
5	449.67	3.38	1660
6	34.46	7.42	1667
7	6.03	0.74	1654
8	8.50	0.48	1642
9	19.64	0.020	1582
10	8.88	0.45	1495
11	16.19	0.063	1432

(b) Six Pile Shuffle

Iteration	χ^2	p-värde	Ledtid [ns]
1	3854.97	0	1116
2	18.36	0.031	1124
3	12.27	0.20	1131
4	5.94	0.75	1184
5	12.34	0.19	1167
6	8.89	0.45	1168
7	4.33	0.89	1211

(c) SOC Pile Shuffle

Iteration	χ^2	p-värde	Ledtid [ns]
1	3349.24	0	1450
2	7.02	0.64	1465
3	19.11	0.024	1518
4	7.53	0.58	1441
5	8.50	0.48	1501
6	12.34	0.19	1536
7	8.36	0.50	1577

(d) Ten Pile Shuffle

Iteration	χ^2	p-värde	Ledtid [ns]
1	747.30	4.65^{-155}	1587
2	5.83	0.76	1654
3	4.83	0.85	1716
4	5.38	0.80	1625
5	5.09	0.82	1697
6	8.80	0.46	1698
7	14.46	0.11	1710

(e) Wheel Fisher-Yates Shuffle

Iteration	χ^2	p-värde	Ledtid [ns]
1	10.71	0.30	715
2	13.46	0.14	720
3	6.66	0.67	728
4	9.05	0.43	733
5	8.95	0.44	772
6	6.10	0.73	794
7	9.41	0.40	778

5 Diskussion

Diskussion om resultaten av simulationerna är uppdelat i flera underavsnitt i syfte att göra det enklare att följa tankeprocessen. I jämförelsen av blandningsmetoderna(5.1) kommer närmare undersökning av metoderna komma fram till vilken av dem som passar bäst inpå dem krav som ställdes i syftet (1.2). Felkällor (5.2) nämns efter att den mest passande blandningsmetoden tagits fram för att diskutera deras påverkan på dem tidigare slutsatserna i diskussionen. I slutsatsen (5.3) återpresenteras den blandningsmetod som undersökningen kom fram till vad det innebär och vilka förbättringar som skulle kunna göras i potentiell framtida forskning.

5.1 Jämförelse av blandningsmetoderna

Utifrån slumpmässighet: Resultaten av det klassisk pokertestet (Tabell 4) ger ut vilka av blandningsmetoderna som kan definitivt inte sägas ge slumpmässigt blandade kortlekar. Eftersom den bästa blandningsmetoden måste kunna slumpmässigt blanda en kortlek. Därför är ingen av metoderna som gav signifikanta resultat d.v.s. dessa som överstiger gränsvärde på 16.92 värda att tänka på, dem kan gallras från populationen av metoderna som behöver vidare analys. För GSR Riffle Shuffle betyder det att alla innan 7 kan gallras bort, det är ett logiskt resultat då det stämmer överens med tidigare undersökningar på GSR vars resultat också medför att 7 upprepningar av Riffle Shuffle krävs för att få en helt slumpmässig blandning (Bayer och Diaconis, 1992). Ten Pile Shuffle visade det mest säkraste indikation på hög slumpmässighet vid iterationer 2 till 5 men enligt STDMean testet var det endast med fyra iterationer som alla kort i kortleken hade tillräckligt stort variation d.v.s. ingen avvikelse från dem teoretiska värdena. SOC Pile shuffle visade också giltigt slumpmässighets värde vid iteration 2 men såsom med Ten Pile Shuffle endast iteration 4 visade helt slumpmässigt resultat från STDMean testet. Six Pile Shuffle vid pokertestet gav giltiga värdena som dem andra variationer endast iteration 5 visade giltigt resultat enligt STDMean testet.

Baserad på dess effektivitet: Eftersom den potentiella kortleksblandaren prioriterar effektiviteten av blandningen kan blandningsmetoderna bedömas utifrån mängden successiva blandningar (iterationer) dem utför. Blandningsmetoder som utför fler iterationer kommer vara något mindre effektiva än metoder av samma typ men som utför färre iterationer, d.v.s. ett lågt antal iterationer är en fördel. På grund av det så kan alla blandningsmetoderna som utför mer än tio iterationer konstateras innehava en underlägsen effektivitet utan noggrannare analys. Dessutom visades dem inte ökat slumpmässighet med över 11 iterationer. Med detta sagt iterationer från 11 till och med 15 iterationer är alltså inte inkluderade i den närmare analysen.

Algoritmerna presterade med olika slags mjuvarassnabbhet. GSR Riffle Shuffle var den näst långsammaste men ledtiden minskade vid mer iterationer. Detta kan beror på att alla algoritmerna kördes i parallellt simulation därför friades det mer resurser åt dem sista iterationer. Six Pile Shuffle enligt ledtiderna var den snabbaste detta kan bero på att algoritmen hade mindre fack att välja mellan dessutom stannade algoritm färre gånger. D.v.s. när ett annat fack är redan fullt av dem max 10 kort per fack. SOC Pile Shuffle presterade sämre än GSR. Ten Pile Shuffle var den mest långsammaste och dess ledtid ökade vid fler iterationer. Detta kunde också påverkas av faktumet på inte tillräckligt likt till den potentiella kortleksblandare metod.

Ledtiderna för Wheel Fisher-Yates proccessmässigt kunde inte simuleras på likt som den potentiella blandningsmaskinen. För att processen att simulera detta blev svårare och löstes inte. Därför dess ledtiden kan inte vara ett mått på dess effektivitet.

Dessutom ledtiderna kan vara vilse ledande. Detta kan beror på att simulerings miljö inte riktigt avspeglade hur den potentiella kortleksblandare skulle ha fungerat. Därför blandningsmetoderna effektivitet diskuteras i nästa del.

Ifrån dess fysiskt tillämplighet: Som det nämndes i sektion 3.2 dessa tre undersökta algoritmerna har väldigt annorlunda design principer därmed behöver olika antal av rörliga delar involverade i processen för att imitera dess underliggande algoritm. Dessa faktorer kommer att påverka dess fysiska tillämplighet i den potentiella blandningsmaskinen. Därför önskade faktorerna som storlek, snabbhet och verkställbarhet (hur enkelt är dess design och hur framgångsrik kan den utföra successiva iterationer utan problem) ska tänkas innan valet görs. Men för att dessa krav undersöks inte i denna studie kan inte den bästa blandningsmetod tas fram. Därför i denna studie har det kommits fram till att det finns ingen definitiv svar men snarare ett eller annat metod kan vara mer eller mindre passande. GSR visade att den behöver 7 iterationer för att nå slumpmässig blandning. För effektiviteten innebär det ett långt blandningsprocess. Dess fysiska tillämplighet är komplex. Den behöver två kortmatningsmekanismer som fungerar i symbios med varandra för att eliminera tidsrelaterade problem. Pile Shuffle har relativt lätt framtagningsprocess och har redan 3D printats av (3DprintedLife, 2021). Den är effektivt för att den kräver endast en kortmatningsmekanism, ingen tidsrelaterade problem finns för att det är endast ett kort som rör sig i taget. Men för den mest slumpmässiga resultat med minsta antal iterationer ska SOC inte användes. Istället Ten Pile Shuffle ska tillämpas för att den visade mer stabila resultat i pokertestet. Därför är denna algoritm mer pålitlig att ha högre slumpmässighet. Om kravet är att blanda flera kortlekar kan Wheel Fisher-Yates Shuffle tillverkas. Angående effektivitet kommer denna att vara långsamt p.g.a. den måste snurra för att mata in en kort i rätt position. Men den behöver endast en kortmatningsmekanism. Samt endast ett iteration för att nå slumpmässig blandning.

5.2 Felkällor

Som det nämndes i föregående sektion ledtiderna visar inte rätt representation av hur långt tid det tar för att utföra en blandning. Samt simulationen avspeglar inte hur långt tid denna skulle ta i verkligheten. Därför ledtiderna blev relativa men kunde inte användes för att argumentera för en eller annan kortleksblandare skulle vare mer effektiv.

I denna studie saknades det kunskaper om andra statistiska analysmetoder som ApEn som skulle ha gett inblick i andra aspekter av algoritmens slumpmässighet. Dessutom datamängd som valdes ifrån i krav pokertestet skulle ha varit högre för att uppnå mer testdata. På så sätt få att Royal Flush, matematiskt skulle ha uppkommit fler än 5 gånger. Detta skulle ha get mer pålitliga testresultat.

5.3 Slutsats

I studien undersöktes 3 olika design av blandningsmetoderna- Riffle Shuffle, Pile Shuffle (med 3 olika variationer) och Wheel Shuffle. Utifrån resultat av statistiska analysmetoder som Pokertest och STDMean testet har det kommits fram till mest passande antal iterationer. Men en definitivt bästa blandningsmetod till den potentiella kortleksblandare kunde inte tas fram. Istället varje blandningsmetod kan användes vid specifika krav, dessa krav undersöktes inte. Därför är den

enklaste metoden att tillverka Pile Shuffle med 10 fack och 10 max kort per fack. Dessutom för den mest slumpmässiga kortleksblandning ska kortleken blandas 4 gånger. Om istället kravet ligger att blanda mer en kortlek då skulle Wheel Shuffle med Fisher-Yates blandningsalgoritm passa bäst den skulle endast kräva 1 iterationer för att vara tillräckligt slumpmässig. Slutligen Riffle Shuffle kräver mer undersökningar därför att 7 iterationer är tidskrävande d.v.s. inte effektivt samt design av en sådan maskin är mest komplex av dem 3. Såsom denna metod ska behöva 2 kortmatningsmekanismer samt ska denna kunna utföra blandningar utan att få tidsrelaterade problem.

I framtida forskning skulle Riffle Shuffle undersökas närmare, andra variationer i algoritmen testas för att nå tillräckligt stort slumpmässighet med färre iterationer på så sätt dess effektivitet förbättras. För Pile Shuffle and Wheel Shuffle skulle prototyper byggas för att undersöka dess riktiga ledtider i ett fysiskt maskin. På så sätt dess fysiskt tillämplighet undersökas närmare. Samt prototyper skulle byggas i 3D modellerings program och sedan program som Gazebo simulator användas för att simulera dessa i sin fysiska form men i en simulation som simulerar vår fysiska värld på mer tillfredsställande sätt.

Bibliografi

3DprintedLife (2021). *Rigged Card Sorting Machine - ALWAYS Get The Hand You Want!* URL: https://www.youtube.com/watch?v=eMTXyl7tPEk (hämtad 2023-08-15).

Abdel-Rehim, Wael MF, Ismail A Ismail och Ehab Morsy (2014). "Implementing the classical poker approach for Testing Randomness". I: *International Journal* 4.8. URL: https://www.researchgate.net/profile/Wael-Fawaz-2/publication/281178831_Implementing_the_Classical_Poker_Approach_for_Testing_Randomness/links/55da3a9608aec156b9ae74a7/Implementing-the-Classical-Poker-Approach-for-Testing-Randomness.pdf (hämtad 2023-12-07).

Armstrong, Drew (2006). "Probability of Poker Hands". I: URL: https://www-users.cse.umn.edu/~reiner/Classes/Poker.pdf (hämtad 2023-12-09).

Bayer, Dave och Persi Diaconis (1992). "Trailing the Dovetail Shuffle to its Lair". I: *The Annals of Applied Probability* 2 (2), s. 294–313. DOI: https://doi.org/10.1214/aoap/1177005705.

Delgado-Bonal, Alfonso och Alexander Marshak (2019). "Approximate Entropy and Sample Entropy: A Comprehensive Tutorial". I: *Entropy* 21.6. ISSN: 1099-4300. URL: https://www.mdpi.com/1099-4300/21/6/541.

Diaconis, Persi (2003). "Mathematical developments from the analysis of riffle shuffling". I: *Groups, combinatorics & geometry (Durham, 2001)*, s. 73–97. URL: https://www.yumpu.com/en/document/read/16491188/persi-diaconis-department-of-statistics-stanford-university (hämtad 2024-02-09).

Gilbert, Edgar (1955). *Theory of shuffling*. Tekn. rapport. Murray Hill: Bell Laboratories Technical Memorandum.

Hannes, Salin (2011). "Analys av pseudoslumptalsalgoritmer". I: URL: https://www.csc.kth.se/utbildning/kandidatexjobb/datateknik/2011/rapport/salin_hannes_K11089.pdf (hämtad 2024-02-06).

Harris, Charles R. m. fl. (2020). "Array programming with NumPy". I: *Nature* 585, s. 357–362. DOI: 10.1038/s41586-020-2649-2.

Hunter, John D (2007). "Matplotlib: A 2D graphics environment". I: *Computing in science & engineering* 9.3, s. 90–95.

Lam, Siu Kwan, Antoine Pitrou och Stanley Seibert (2015). "Numba: A LLVM-Based Python JIT Compiler". I: LLVM '15. URL: https://doi.org/10.1145/2833157.2833162 (hämtad 2023-12-17).

Matsakis, Nicholas D och Felix S Klock II (2014). "The rust language". I: *ACM SIGAda Ada Letters*. Vol. 34. 3. ACM, s. 103–104.

Matsakis, Niko och Josh Stone (2022). *Rayon: Data-parallelism library for Rust*. Version 1.8.0. URL: https://crates.io/crates/rayon.

National Institute of Standards and Technology (2023). *Engineering Statistics Handbook - Section 1.3.5.15: Chi-Square Goodness-of-Fit Test*. URL: https://www.itl.nist.gov/div898/handbook/eda/section3/eda35f.htm (hämtad 2023-10-04).

Random.org (2024). *Introduction to Randomness and Random Numbers*. URL: https://www.random.org/randomness/ (hämtad 2024-02-08).

Sharma, Ayushi m. fl. (2023). "Rust for Embedded Systems: Current State, Challenges and Open Problems". I: *arXiv preprint arXiv:2311.05063*. URL: https://arxiv.org/pdf/2311.05063.pdf (hämtad 2023-12-09).

Terwijn, Sebastiaan A. (2016). "The Mathematical Foundations of Randomness". I: *The Challenge of Chance: A Multidisciplinary Approach from Science and the Humanities*. Utg. av Klaas Landsman och Ellen van Wolde. Cham: Springer International Publishing, s. 49–66. URL: https://doi.org/10.1007/978-3-319-26300-7_3 (hämtad 2023-04-10).

The Rand Project Developers (2022). *Rand: A Rust library for random number generation*. Version 0.8. URL: https://crates.io/crates/rand.

Van Rossum, Guido och Fred L. Drake (2009). *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace. ISBN: 1441412697.

Virtanen, Pauli m. fl. (2020). "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python". I: *Nature Methods* 17, s. 261–272. DOI: 10.1038/s41592-019-0686-2.

Yan, Kevin (16 aug. 2016). *1-8 decks Casino Full-Automatic card shuffler*. URL: https://www.youtube.com/watch?v=txl3gqIfwHM (hämtad 2024-02-06).

Bilagor

A Källkod

Länk till Github repository vart Rust och Python källkod till simulationen respektive statistisk analys är samlad, samt källkod till latex med vilken detta rapport hade skrivits länk: https://github.com/Abishevs/gymnasieArbete$_2$3 – 24

B Kod för GSR Riffle Shuffle

Kodlistning 1: Pile Shuffle skriven i Rust

```
fn shuffle(&self, deck: &mut Deck){
    let mut deck_vec: Vec<Card> = Vec::with_capacity(52);
    let mut deck_half_1: Vec<Card> = Vec::new();
    let mut deck_half_2: Vec<Card> = Vec::new();

    let mut card_count = 1;
    for &card in deck.iter() {
        if card_count <= (deck.len() / 2) {
            deck_half_1.push(card);
            card_count += 1;
        } else {
            deck_half_2.push(card);
        }
    }

    let mut rng = rand::thread_rng();
    let mut rand_id;
    while !deck_half_1.is_empty() || !deck_half_2.is_empty() {
        rand_id = rng.gen_range(0.0..=1.0);
        if rand_id <= (deck_half_1.len() as f64)/((deck_half_1.len() as f64)+(
            deck_half_2.len() as f64)){
            if let Some(card) = deck_half_1.pop(){
                deck_vec.insert(0, card);
            }

        } else {
            if let Some(card) = deck_half_2.pop() {
                deck_vec.insert(0, card);
```

```
            }
        }
    }

    let mut deck_position = 0;
    for card in deck_vec {
        deck[deck_position] = card;
        deck_position += 1;
    }
```

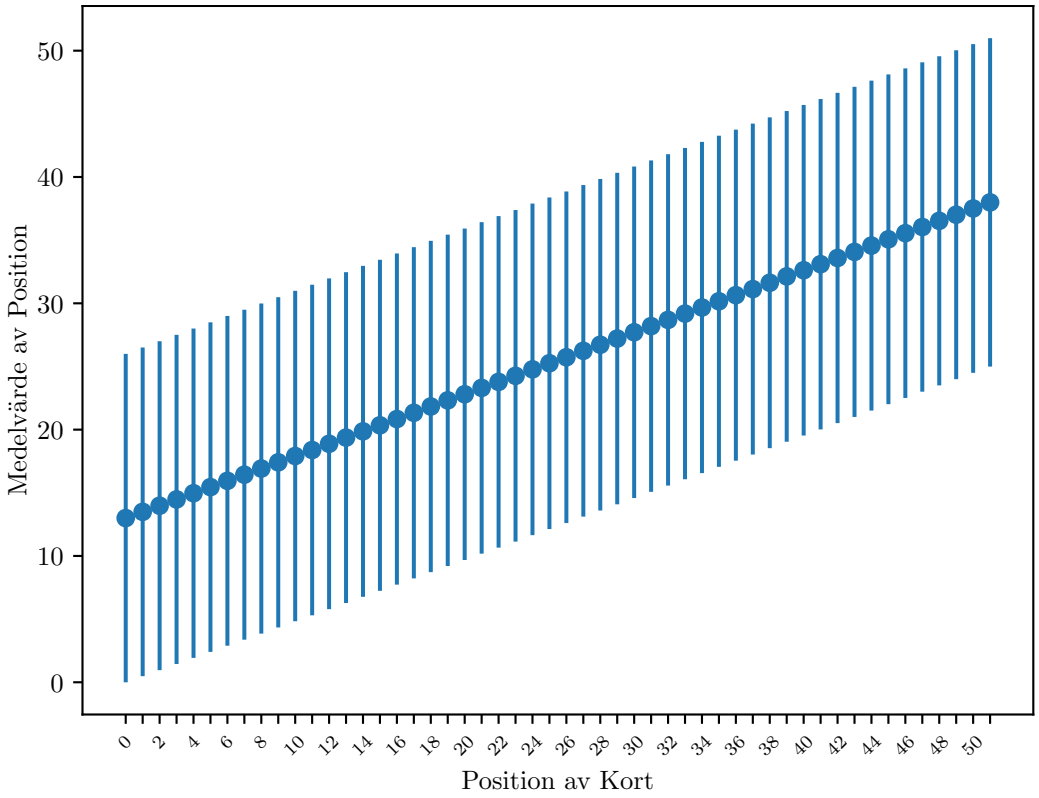
C Kod för Pile Shuffle

```
fn shuffle(&self, deck: &mut Deck){
    const BIN_COUNTS:usize = 8;
    const MAX_CARDS_PER_BIN:usize = 10;

    let mut bins: Vec<Vec<Card>> = vec![Vec::new(); BIN_COUNTS];

    let mut rng = rand::thread_rng();
    for &card in deck.iter() {
        loop {
            let random_bin = rng.gen_range(0..BIN_COUNTS);
            if bins[random_bin].len() < MAX_CARDS_PER_BIN {
                // put the cards in random bins
                bins[random_bin].push(card);
                break;
            }
        }
    }

    let mut deck_position = 0;
    // Reassemble the deck
    for bin in bins {
        for card in bin {
            deck[deck_position] = card;
            deck_position += 1;
        }
    }
}
```

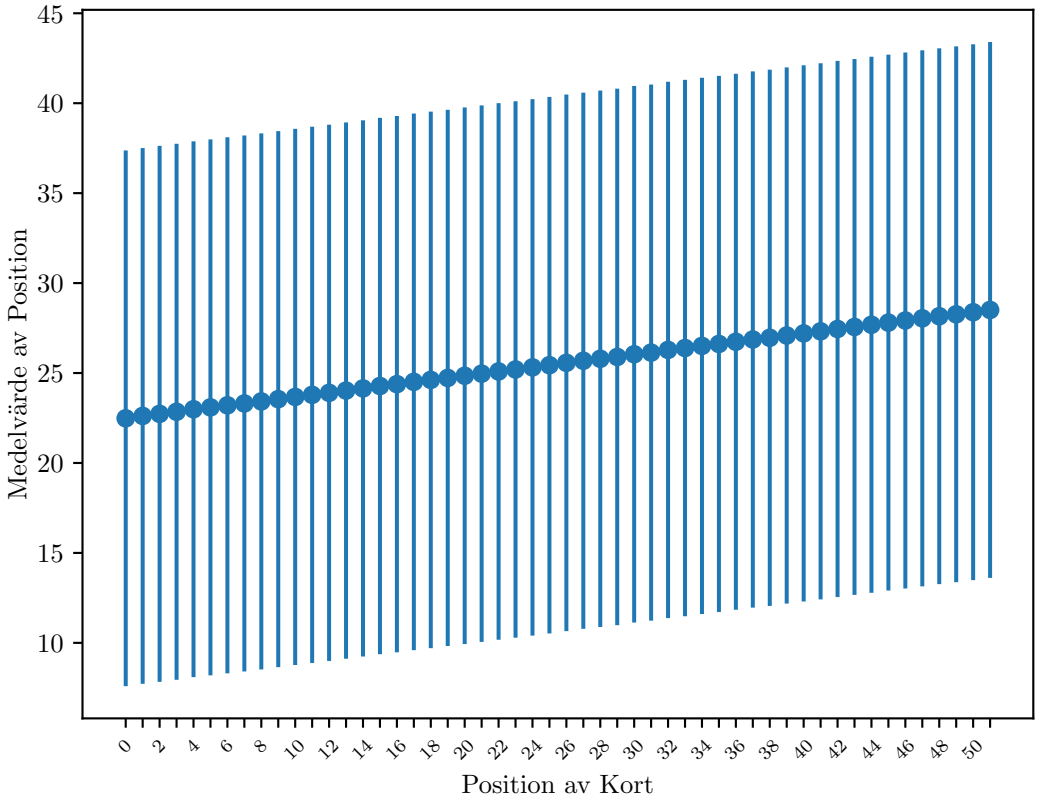
D Kod för Wheel Fisher-Yates Shuffle

```
fn shuffle(&self, deck: &mut Deck) {
    let mut rng = rand::thread_rng();

    // Run fisher yates shuffle to create random indexs.
    let mut random_indexs = (0..52).collect::<Vec<_>>();
    random_indexs.shuffle(&mut rng);

    let mut wheel_slots: Vec<Card> = vec![0; 52];
    // map index to an slot on the wheel.
    for &card in deck.iter() {
        if let Some(index) = random_indexs.pop() {
            wheel_slots[index] = card;
        }
    }

    // take all the slots from the beginning and place them in it's new deck.
    let mut deck_position = 0;
    for card in wheel_slots {
        deck[deck_position] = card;
        deck_position += 1;
    }
```

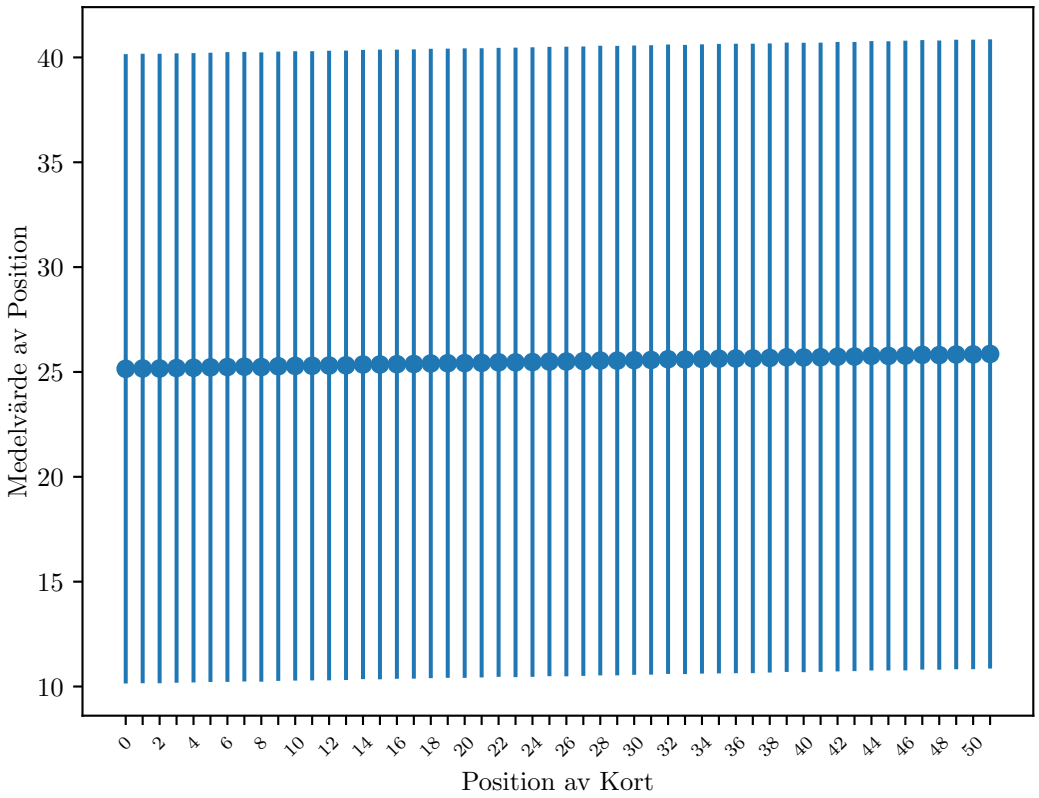
E Resultat av STDMean test



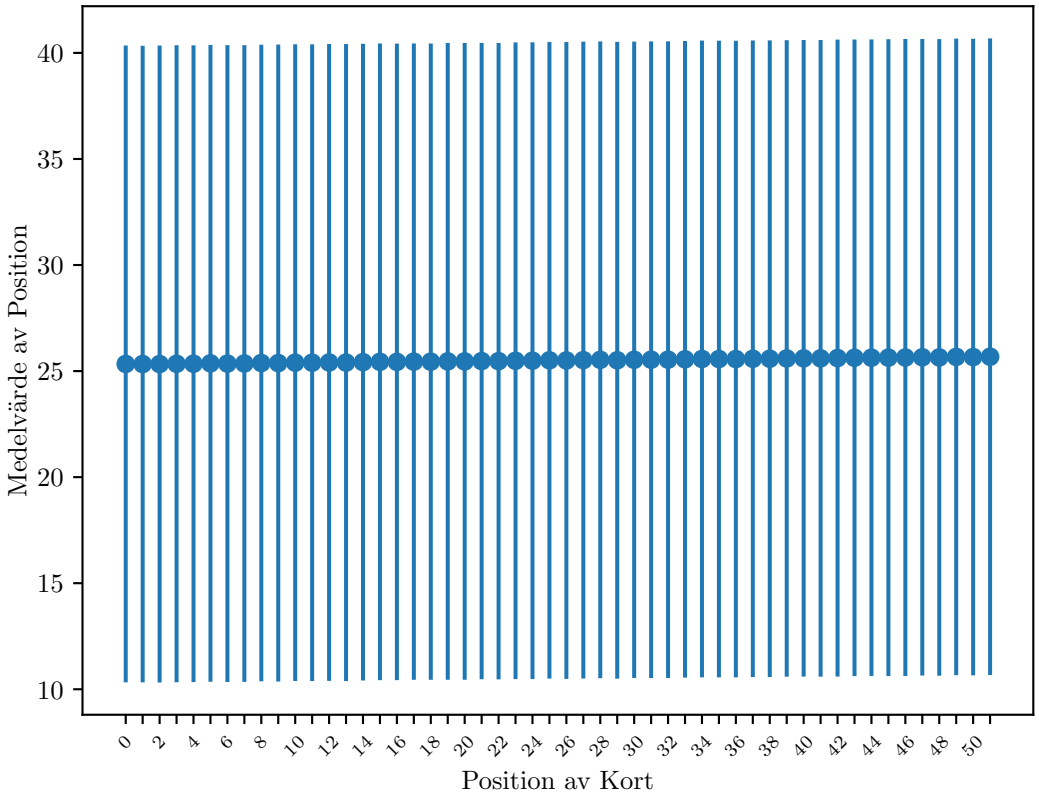
Figur 7: Resultatet från STDMean testet för GSR Riffle Shuffle med **1** iteration.



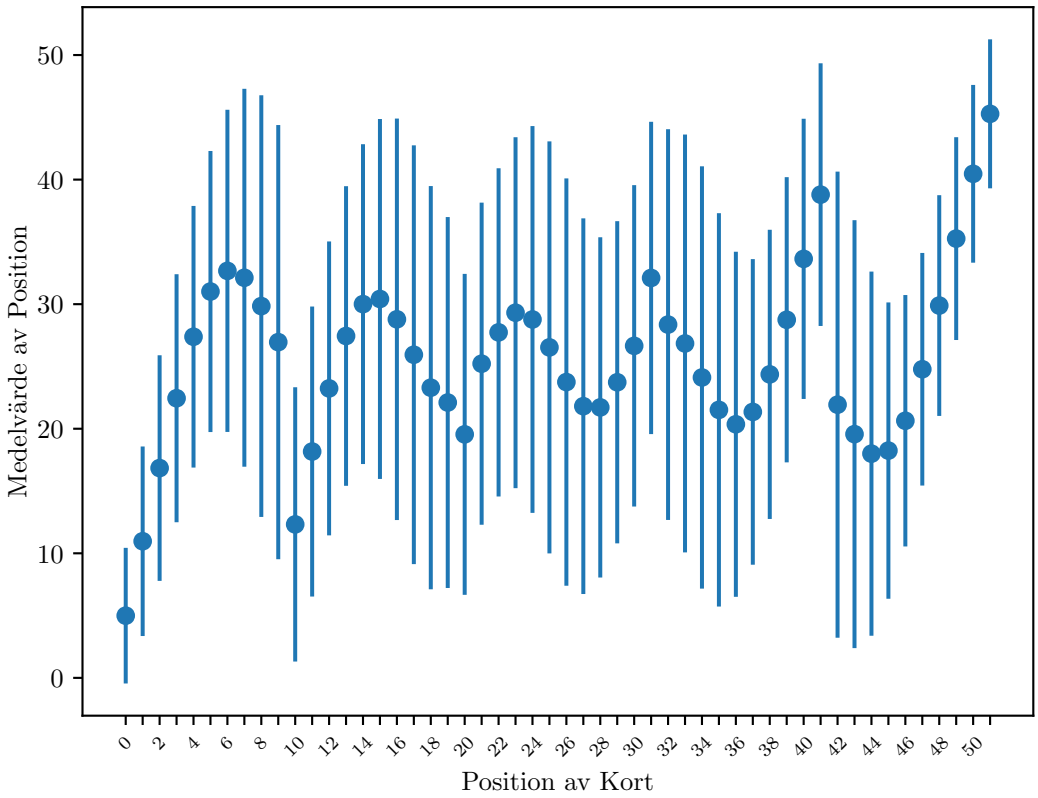
Figur 8: Resultatet från STDMean testet för GSR Riffle Shuffle med **3** iterationer.



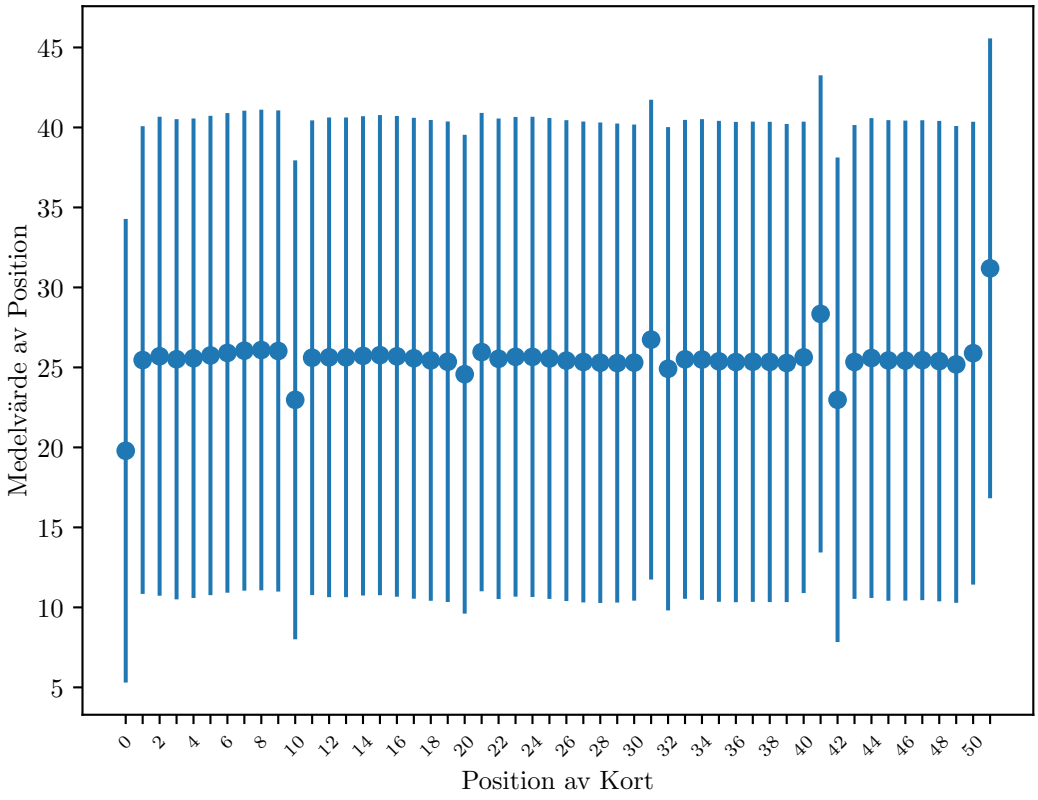
Figur 9: Resultatet från STDMean testet för GSR Riffle Shuffle med **6** iterationer.



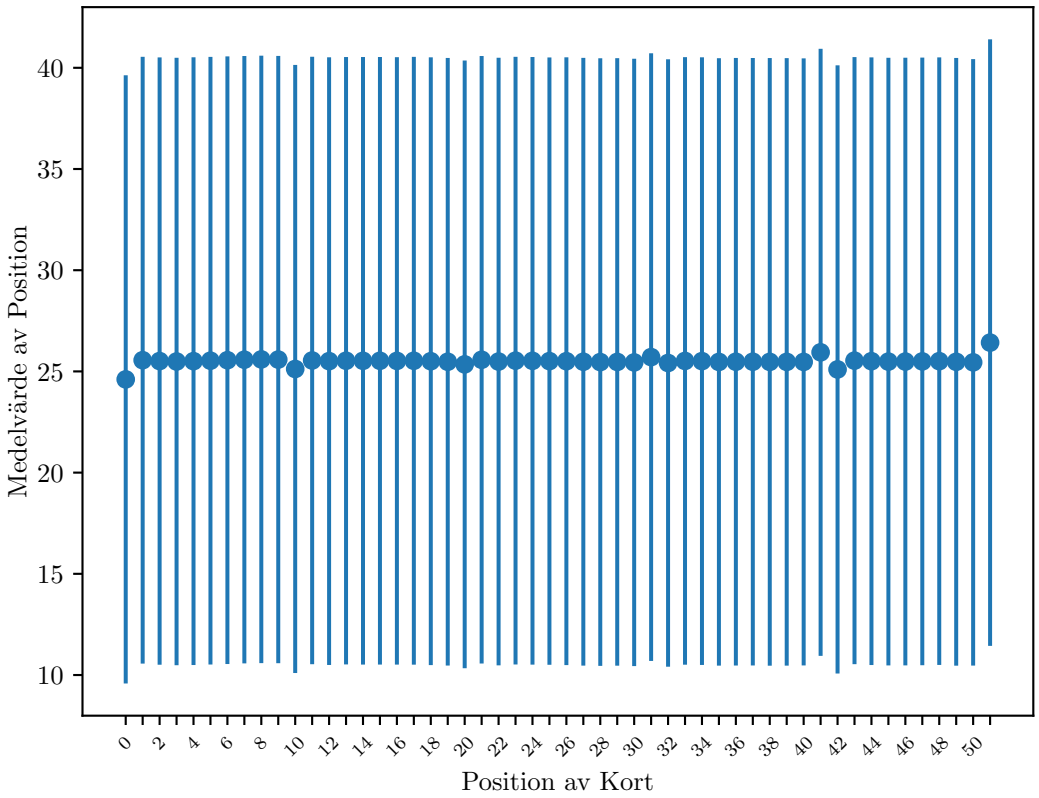
Figur 10: Resultatet från STDMean testet för GSR Riffle Shuffle med **7** iterationer.



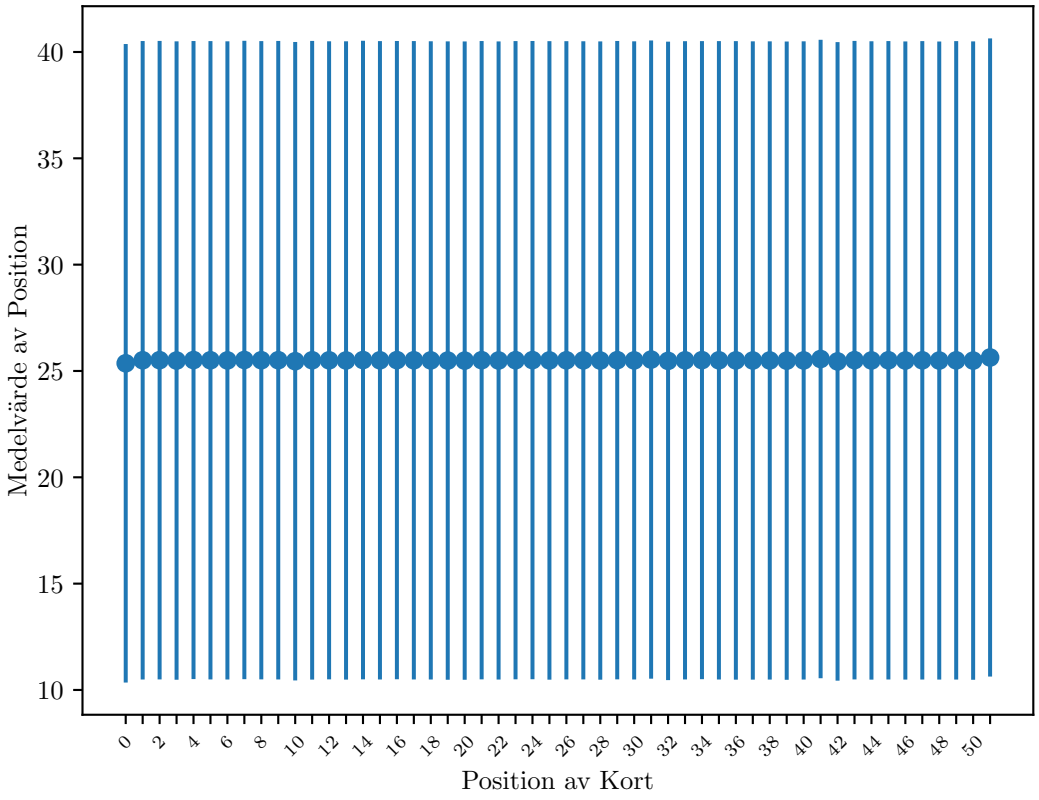
Figur 11: Resultatet från STDMean testet för Six Pile Shuffle med **1** iterationer.



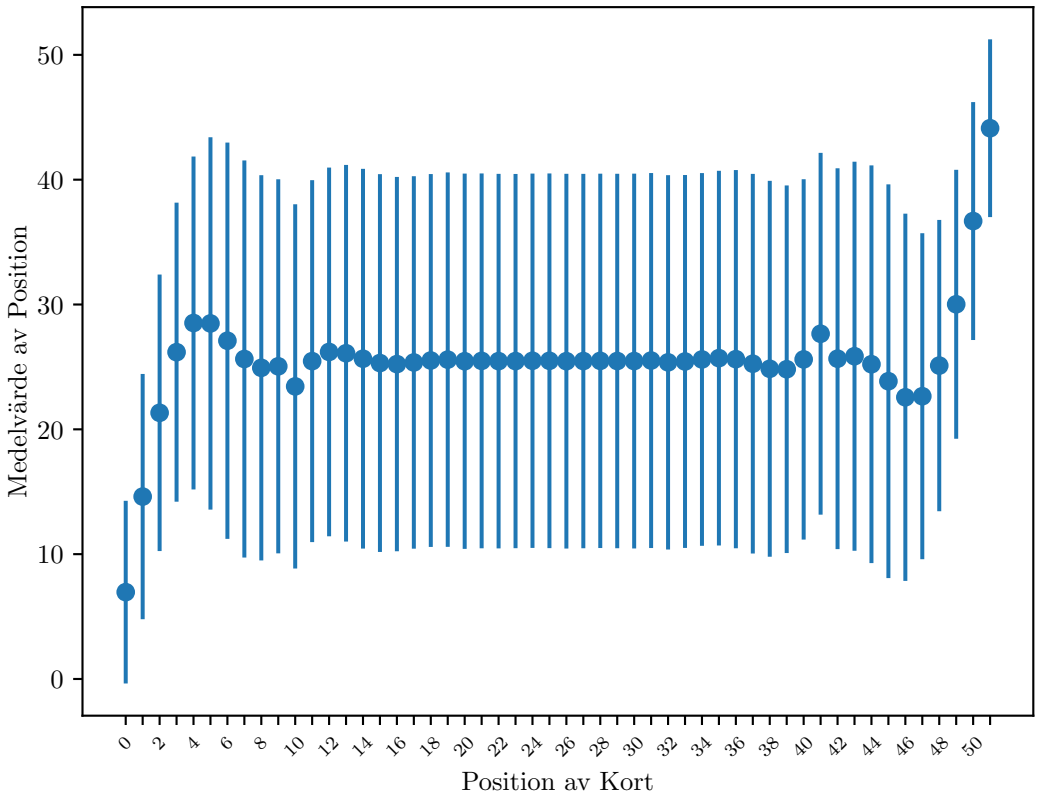
Figur 12: Resultatet från STDMean testet för Six Pile Shuffle med **2** iterationer.



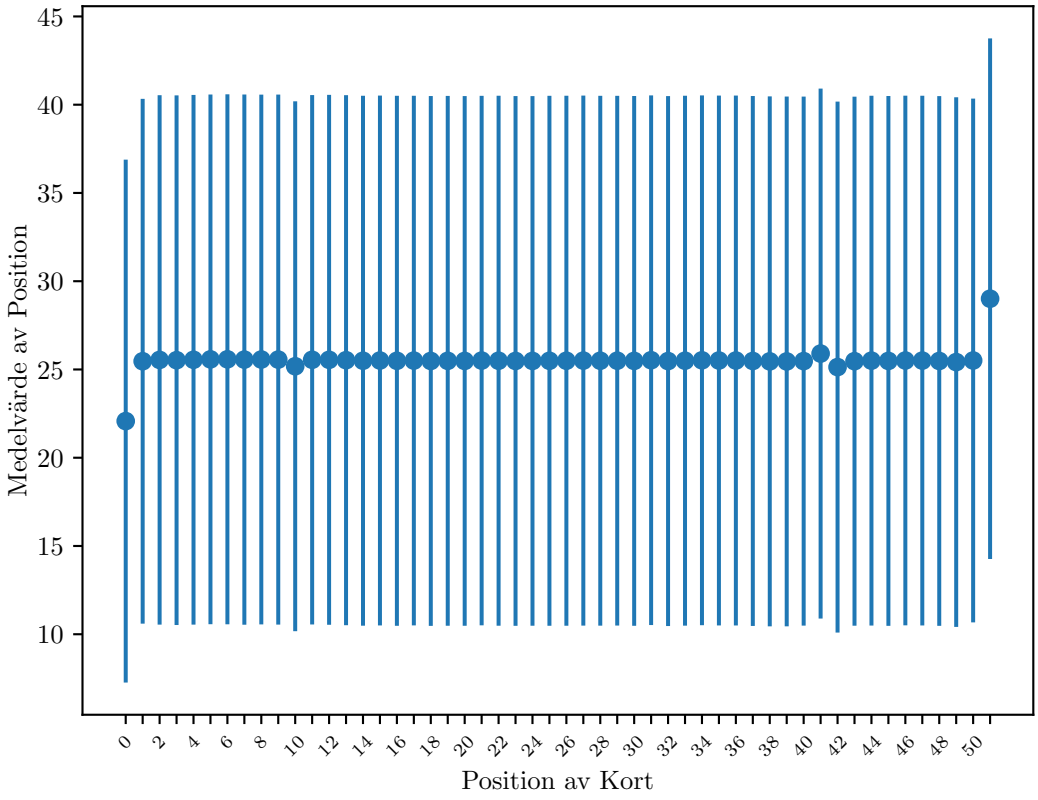
Figur 13: Resultatet från STDMean testet för Six Pile Shuffle med **3** iterationer.



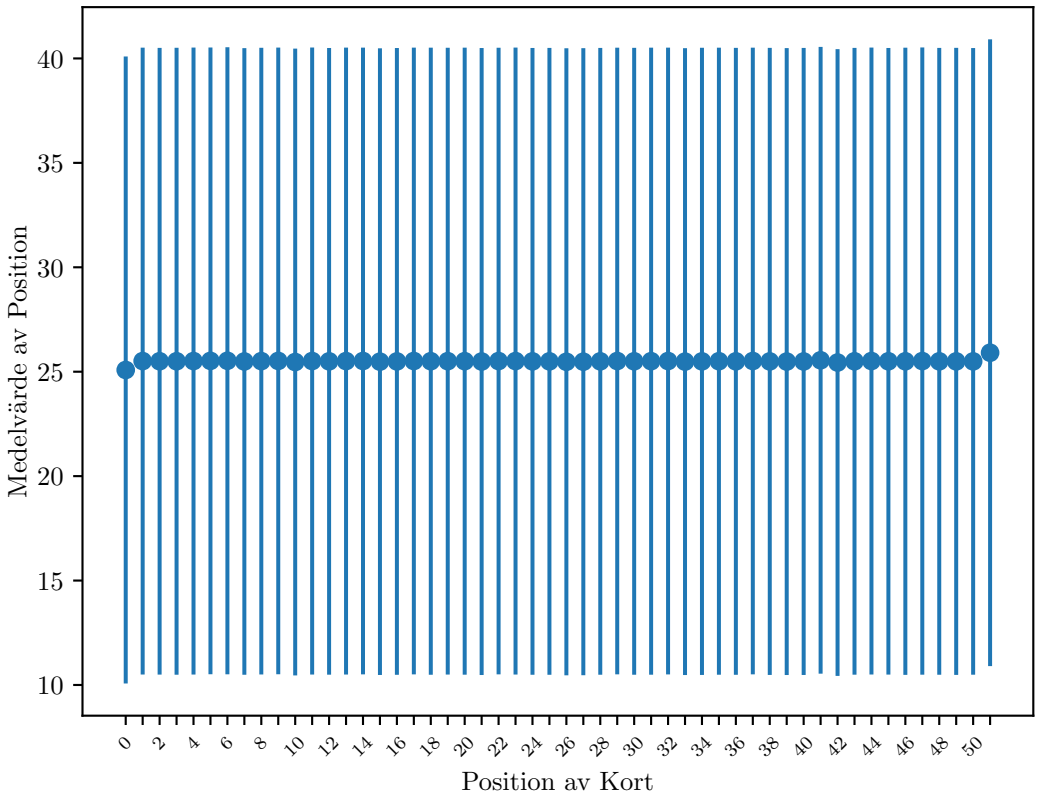
Figur 14: Resultatet från STDMean testet för Six Pile Shuffle med **4** iterationer.



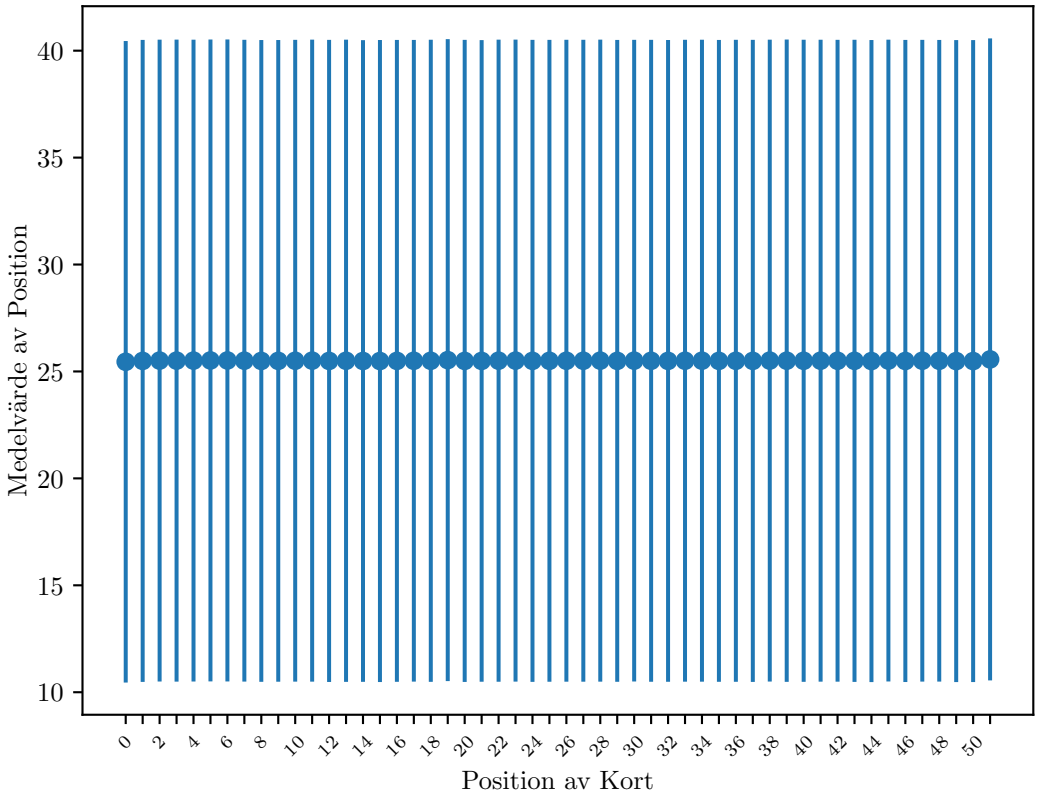
Figur 15: Resultatet från STDMean testet för SOC Pile Shuffle med **1** iteration.



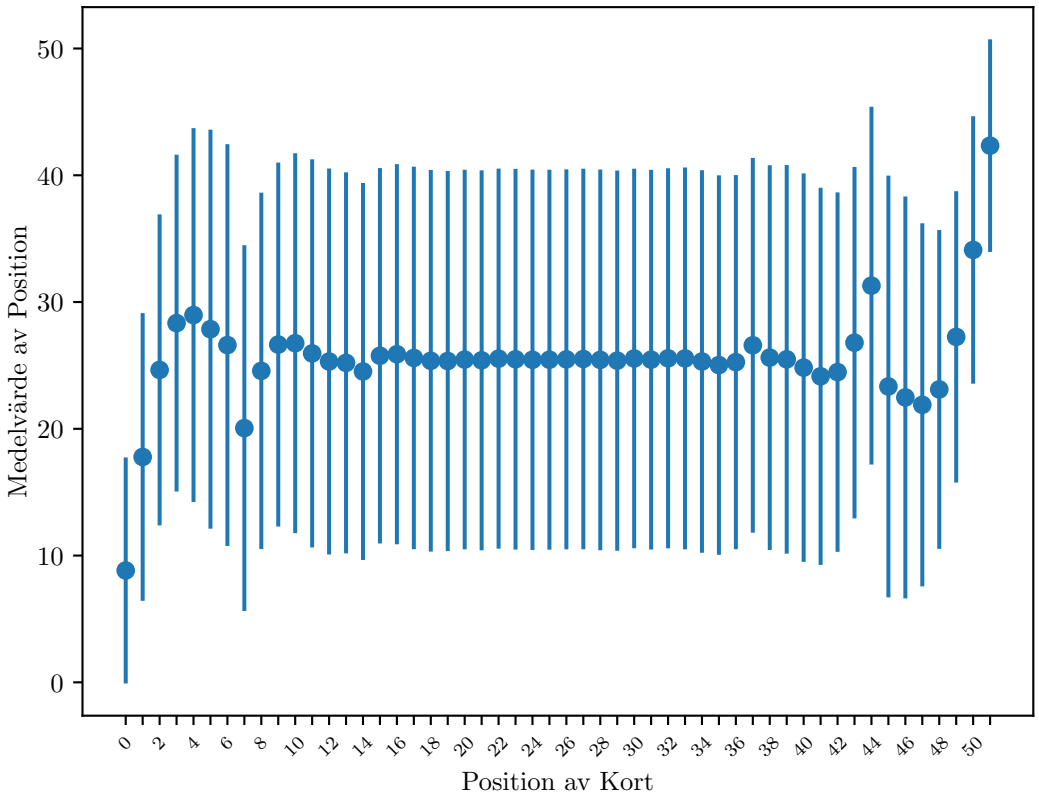
Figur 16: Resultatet från STDMean testet för SOC Pile Shuffle med **2** iterationer.



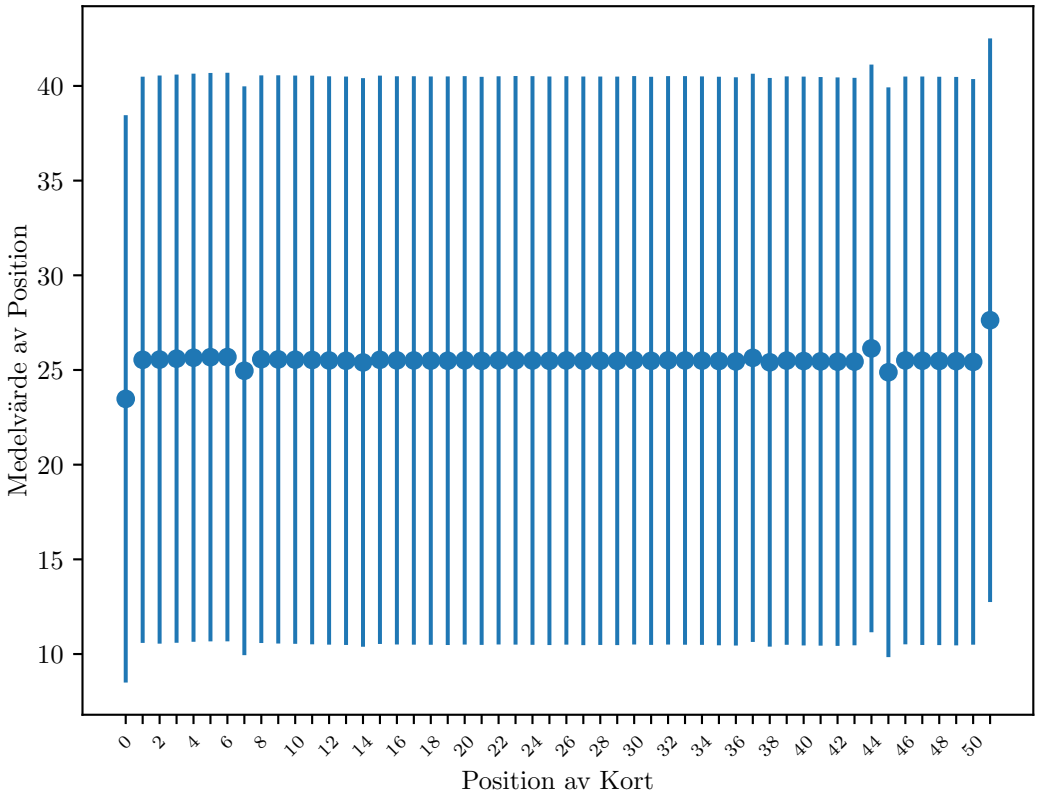
Figur 17: Resultatet från STDMean testet för SOC Pile Shuffle med **3** iterationer.



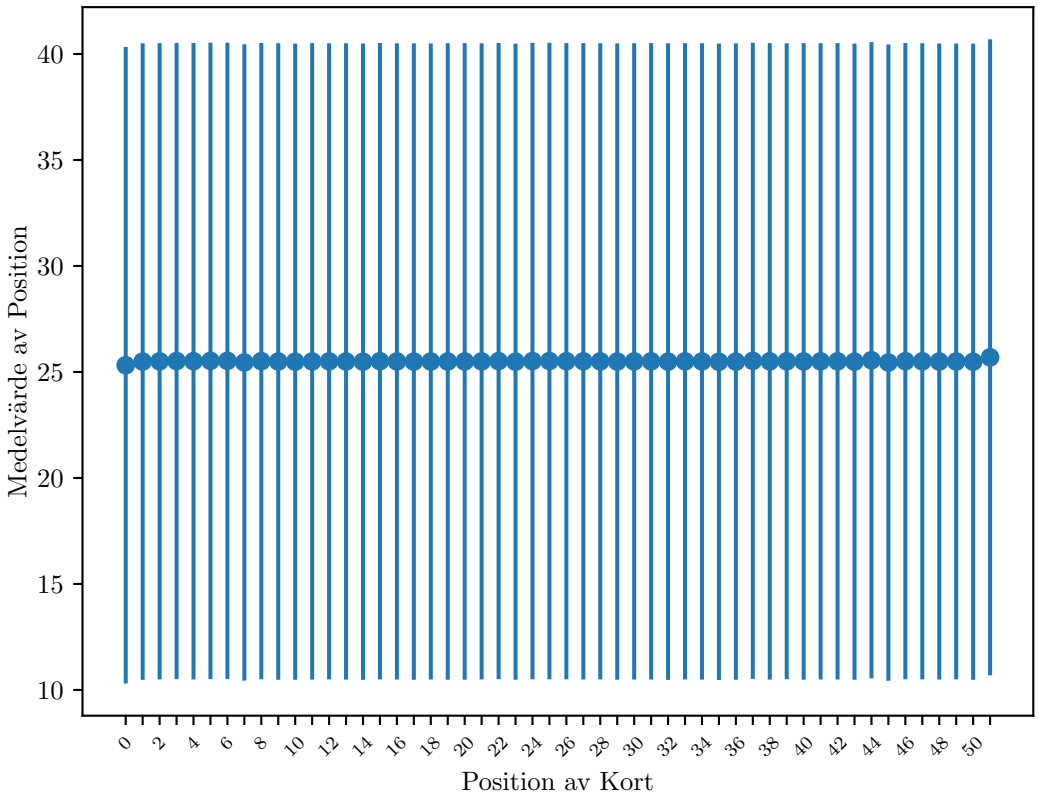
Figur 18: Resultatet från STDMean testet för SOC Pile Shuffle med **4** iterationer.



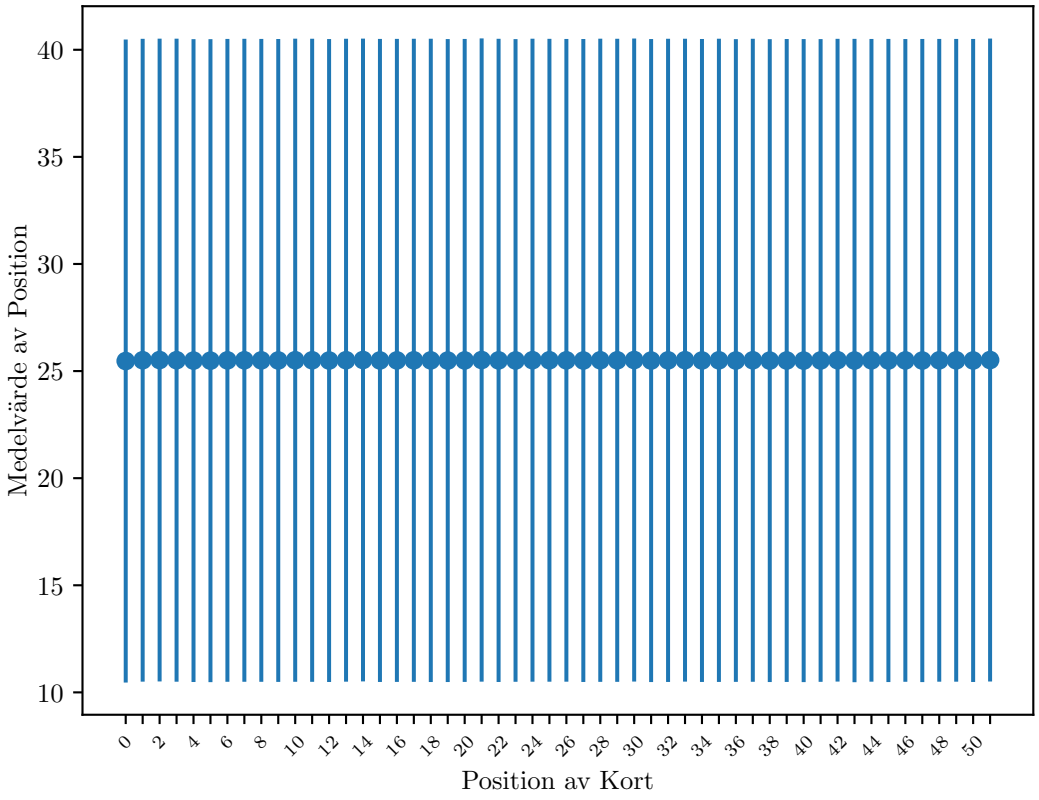
Figur 19: Resultatet från STDMean testet för Ten Pile Shuffle med **1** iteration.



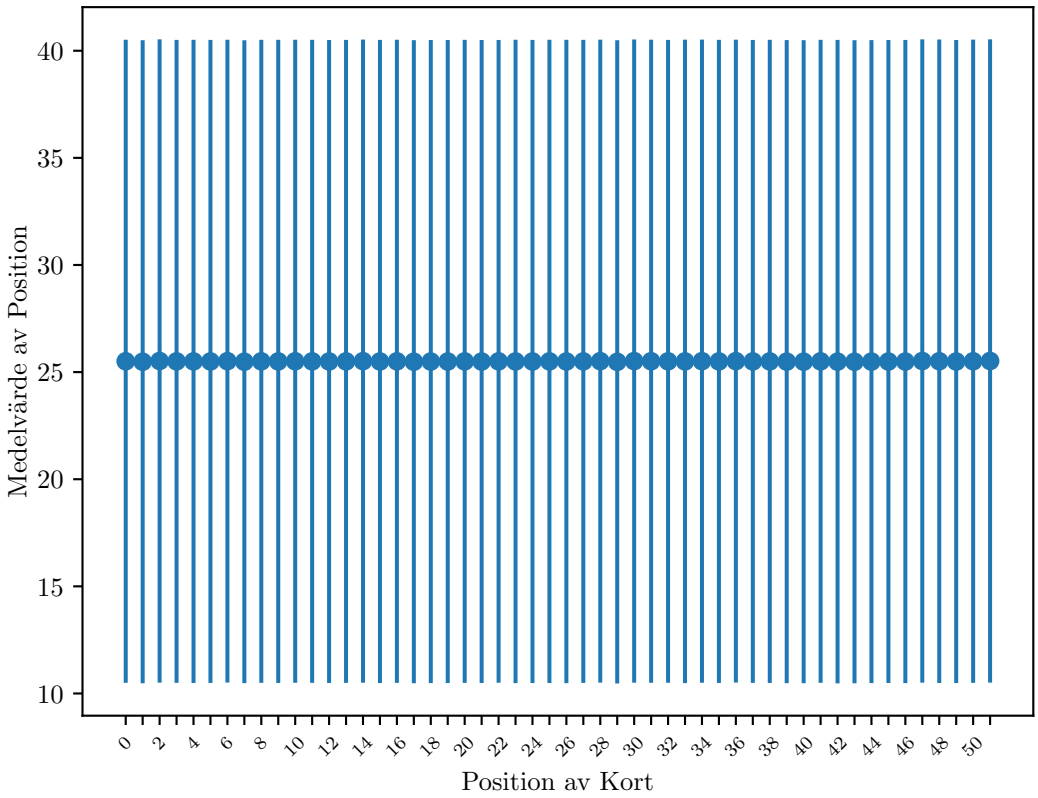
Figur 20: Resultatet från STDMean testet för Ten Pile Shuffle med **2** iterationer.



Figur 21: Resultatet från STDMean testet för Ten Pile Shuffle med **3** iterationer.



Figur 22: Resultatet från STDMean testet för Ten Pile Shuffle med **4** iterationer.



Figur 23: Resultatet från STDMean testet för Wheel Fisher-Yates shuffle med **1** iteration.