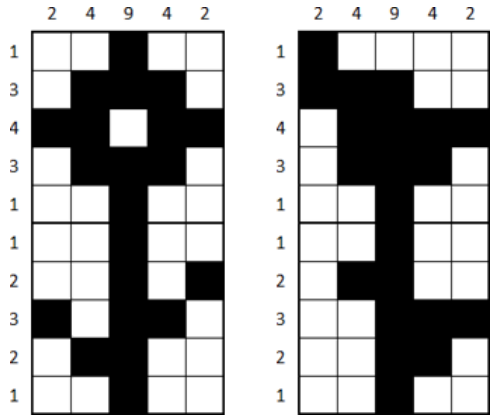
Universiteit Utrecht

Optimalisatie van een algoritme in de binaire tomografie.



Mariëlle Boot

Opleiding: Wiskunde
Faculteit: Wiskunde
11 januari 2019

Begeleider:
Dr. T. van Leeuwen

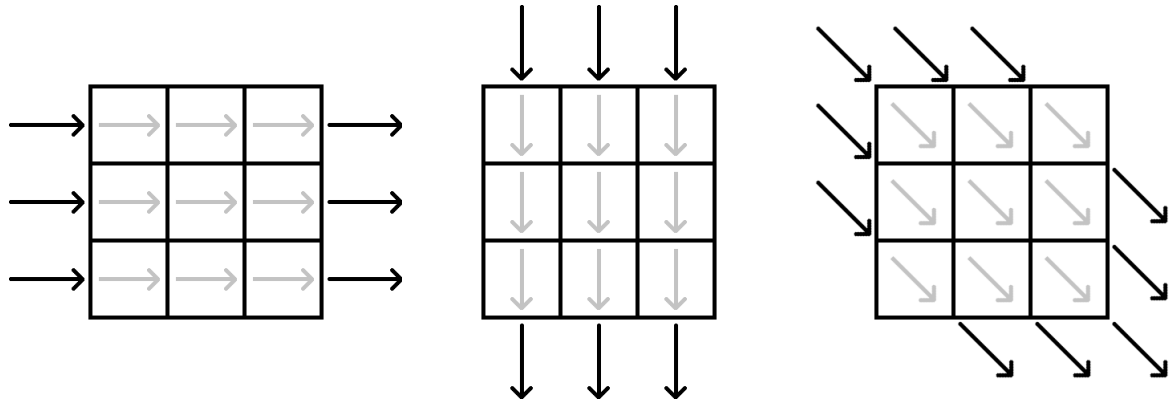
Inhoudsopgave

1 Introductie

In deze scriptie gaan we kijken naar oplossingsalgoritmes in de binaire tomografie. Om een beeld te krijgen van wat binaire tomografie inhoudt, zullen we ons in hoofdstuk 2 eerst verdiepen in wat tomografie nou in het algemeen inhoudt.
De wiskundige achtergrond van de tomografie, bevindt zich binnen de lineaire algebra. De belangrijkste begrippen en methodes voor het oplossen van een probleem binnen de tomografie, staan beschreven in hoofdstuk 3.
Als we deze achtergrondkennis hebben behandeld, gaan we ons verdiepen in de binaire tomografie. We beschrijven daarom in hoofdstuk 4 hoe we van een basis algoritme hebben ontwikkeld om alle oplossingen te genereren en hoe we deze uiteindelijk hebben geoptimaliseerd.

2 Tomografie

In de tomografie houdt men zich bezig met het reconstrueren van een tweedimensionaal beeld van een driedimensionaal voorwerp. Het woord *tomografie* is afgeleid uit het oud-Grieks van de woorden voor *snede* en *schrijven*.
Bij een CT-scan gaan bundels röntgenstraling in een rechte lijn door een bepaald lichaamsdeel. Aan de andere zijde wordt de bundel opgevangen en wordt de resterende straling opgemeten. Als de bundel door bijvoorbeeld bot of weefsel is gegaan, zal de resterende straling minder hoog zijn dan voor deze het lichaam door ging. Dit proces wordt vervolgens meerdere malen herhaald, totdat er genoeg informatie is verzameld om het beeld te reconstrueren.
Dit beeld is een tweedimensionale doorsnede van het lichaamsdeel. Als er meerdere doorsneden worden gemaakt, die steeds net iets opgeschoven is ten opzichte van de vorige, kun je ze samenvoegen tot een reconstructie van het driedimensionale lichaamsdeel.
We kunnen het proces van het reconstrueren van één zo'n doorsnede als volgt laten zien. Stel het plaatje wat we willen reconstrueren bestaat uit 3x3 pixels. Vervolgens worden er van links naar rechts, van boven naar beneden en diagonaal metingen gedaan (zie figuur 1).



(a) Van links naar rechts (b) Van boven naar beneden (c) Diagonaal

Figuur 1: Voorbeeld van metingen in een 3x3 grid (1)

2.1 Wet van Beer-Lambert

Als de straal door een vakje gaat, zat de intensiteit van de straal bij het binnenkomen groter of gelijk zijn dan wanneer de straal het vakje verlaat. Om de intensiteit van de straal te bepalen op het moment dat de straal het vakje verlaat, gebruiken we de wet van Beer-Lambert:

$$I_{\text{uit}} = I_0 \cdot e^{-(\Delta x \cdot \mu)}.$$

Hierbij noemen we I_0 de intensiteit van de straal als deze het vakje binnenkomt en I_{uit} de intensiteit van de straal als deze het vakje verlaat. Verder definiëren we Δx_i als de lengte van vakje x_i en μ_i als de absorptiecoëfficiënt[1] van het vakje x_i.
Als een straal door een aantal vakjes gaat, noteren we deze als de vakjes x_i met $i \in I$. De indices worden dus bepaald door de verzameling I. We definiëren

$$w_i = \begin{cases} \Delta x_i & \text{als } i \in I \\ 0 & \text{anders.} \end{cases}$$

[1]Dit is de mate waarin de straal wordt tegengehouden.

Als de straal het object verlaat, geeft ons dat de volgende vergelijking:

$$I_{\text{uit}} = I_0 \cdot e^{-\left(\sum\limits_{i \in I} \Delta x_i \cdot \mu_i\right)}.$$

En met de invoering van w_i kunnen we dit dus veralgemeniseren naar

$$\begin{aligned} I_{\text{uit}} &= I_0 \cdot e^{-\left(\sum\limits_{i} w_i \cdot \mu_i\right)} \\ \frac{I_{\text{uit}}}{I_0} &= e^{-\left(\sum\limits_{i} w_i \cdot \mu_i\right)} \\ \log\left(\frac{I_{\text{uit}}}{I_0}\right) &= \log\left(e^{-\left(\sum\limits_{i} w_i \cdot \mu_i\right)}\right) \\ \log\left(\frac{I_{\text{uit}}}{I_0}\right) &= -\sum_i w_i \cdot \mu_i. \end{aligned}$$

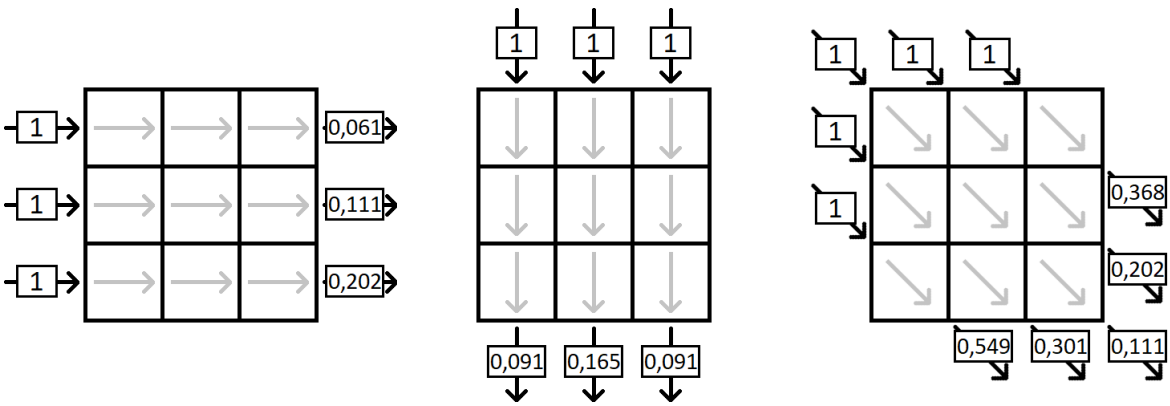
We zien dus dat het reconstrueren van een afbeelding van een CT-scan is dus niets anders dan het oplossen van stelsels vergelijkingen.

2.2 Uitwerking voorbeeld

Als we nu naar het voorbeeld kijken, kunnen we dus het stelsel van vergelijkingen gaan opstellen. In ons voorbeeld nemen we voor $\Delta x_i = 1$ voor alle i, ongeacht hoe de straal door een vakje is gegaan. Immers, als de straal recht door een vakje gaat is de afstand korter dan wanneer de straal schuin door een vakje gaat. Er geldt dan voor iedere vergelijking:

$$\log\left(\frac{I_{\text{uit}}}{I_0}\right) = -\sum_{i \in I} \Delta x_i \cdot \mu_i.$$

Als voorbeeld bekijken we onderstaande metingen.

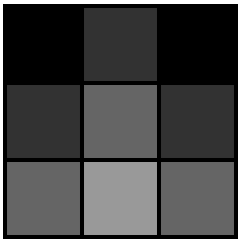


(a) Van links naar rechts (b) Van boven naar beneden (c) Diagonaal

Figuur 2: Voorbeeld van metingen in een 3x3 grid (2)

Als we de stelsels vergelijkingen dan oplossen, kunnen we de volgende afbeelding reconstrueren met de bijbehorende absorptiecoëfficiënten[2]

[2]Omdat we I_{uit} hebben afgerond, zijn deze uitkomsten niet exact.



(a) De oorspronkelijke afbeelding

1	0,8	1
0,8	0,6	0,8
0,6	0,4	0,6

(b) De absorptiecoëfficiënt per vakje

Figuur 3: Voorbeeld van metingen in een 3x3 grid (3)

Voor nu nemen we aan dat dit stelsel vergelijkingen inderdaad oplosbaar is en dat deze oplossing uniek is.

2.3 Lineaire vergelijkingen

We weten nu dat het reconstrueren van een afbeelding van $n \times n$ in de tomografie gebeurt door middel van het oplossen van lineaire vergelijkingen. We definiëren het vakje op de i-de rij en in de j-de kolom door μ_k met $k = n(i-1)+j$. Een afbeelding van $n \times n$ ziet er dan als volgt uit:

μ_1	...	μ_n
μ_{n+1}	...	...
...	...	μ_{n^2}

Figuur 4: Afbeelding met n^2 vakjes

De lineaire vergelijkingen die we uit figuur 2(a) kunnen halen zijn dan

$$\sum_{i\in\{1,2,3\}} \mu_i = \mu_1 + \mu_2 + \mu_3 = -\log\left(\frac{0,061}{1}\right) = 2.8$$

$$\sum_{i\in\{4,5,6\}} \mu_i = \mu_4 + \mu_5 + \mu_6 = -\log\left(\frac{0,111}{1}\right) = 2.2$$

$$\sum_{i\in\{7,8,9\}} \mu_i = \mu_7 + \mu_8 + \mu_9 = -\log\left(\frac{0,202}{1}\right) = 1.6$$

Dit kunnen we vervolgens in matrixvorm schrijven.

$$\left(\begin{array}{ccccccccc|c} 1 & 1 & 1 & . & . & . & . & . & . & 2.8 \\ . & . & . & 1 & 1 & 1 & . & . & . & 2.2 \\ . & . & . & . & . & . & 1 & 1 & 1 & 1.6 \end{array}\right)$$

We kunnen nu dus zeggen dat de lineaire vergelijkingen uit de figuren 2(a), 2(b) en 2(c) de volgende matrix weergeven:

$$\left(\begin{array}{ccccccccc|c} 1 & 1 & 1 & . & . & . & . & . & . & 2.8 \\ . & . & . & 1 & 1 & 1 & . & . & . & 2.2 \\ . & . & . & . & . & . & 1 & 1 & 1 & 1.6 \\ 1 & . & . & 1 & . & . & 1 & . & . & 2.4 \\ . & 1 & . & . & 1 & . & . & 1 & . & 1.8 \\ . & . & 1 & . & . & 1 & . & . & 1 & 2.4 \\ . & . & . & . & . & . & 1 & . & . & 0.6 \\ . & . & . & 1 & . & . & . & 1 & . & 1.2 \\ 1 & . & . & . & 1 & . & . & . & 1 & 2.2 \\ . & 1 & . & . & . & 1 & . & . & . & 1.6 \\ . & . & 1 & . & . & . & . & . & . & 1 \end{array}\right)$$

We zien dus dat de eerste kolom wordt weergegeven door de variabele μ_1, de tweede door μ_2 enzovoorts. We noteren dit ook wel als $A\boldsymbol{\mu} = \mathbf{b}$, waarbij A een matrix van $m \times n$ is, $\boldsymbol{\mu}$ de vector van alle variabelen en $\mathbf{b}$ de vector met alle uitkomsten, waarbij geldt dat $\boldsymbol{\mu} \in \mathbb{R}^n$ en $\mathbf{b} \in \mathbb{R}^m$. In vorm van $A\boldsymbol{\mu} = \mathbf{b}$ geldt voor het voorbeeld dus:

$$\left(\begin{array}{ccccccccc} 1 & 1 & 1 & . & . & . & . & . & . \\ . & . & . & 1 & 1 & 1 & . & . & . \\ . & . & . & . & . & . & 1 & 1 & 1 \\ 1 & . & . & 1 & . & . & 1 & . & . \\ . & 1 & . & . & 1 & . & . & 1 & . \\ . & . & 1 & . & . & 1 & . & . & 1 \\ . & . & . & . & . & . & 1 & . & . \\ . & . & . & 1 & . & . & . & 1 & . \\ 1 & . & . & . & 1 & . & . & . & 1 \\ . & 1 & . & . & . & 1 & . & . & . \\ . & . & 1 & . & . & . & . & . & . \end{array}\right) \cdot \begin{pmatrix} \mu_1 \\ \mu_2 \\ \mu_3 \\ \mu_4 \\ \mu_5 \\ \mu_6 \\ \mu_7 \\ \mu_8 \\ \mu_9 \end{pmatrix} = \begin{pmatrix} 2.8 \\ 2.2 \\ 1.6 \\ 2.4 \\ 1.8 \\ 2.4 \\ 0.6 \\ 1.2 \\ 2.2 \\ 1.6 \\ 1 \end{pmatrix}.$$

3 Oplosbaarheid lineaire vergelijkingen

Vanaf nu gaan we ervan uit dat de vector van variabelen wordt weergegeven door de vector $\mathbf{x}$. Dus vanaf nu wordt een stelsel lineaire vergelijkingen weergegeven als

$$A\mathbf{x} = \mathbf{b}.$$

Nu we weten dat we bezig zijn het met het oplossen van stelsels vergelijkingen, zijn we geïnteresseerd naar het feit of een stelsel een oplossing heeft en of deze uniek is.

3.1 Oplosbaarheid

Voordat we de uitkomst bepalen, kunnen we al iets zeggen over de vorm van deze uitkomst. Over deze oplosbaarheid kunnen we het volgende stellen:

1. Het stelsel van lineaire vergelijkingen heeft een unieke oplossing;
2. Het stelsel van lineaire vergelijkingen heeft geen oplossing;
3. Het stelsel van lineaire vergelijkingen heeft oneindig veel oplossingen.

In het eerste geval geldt dat alle kolomvectoren a_i met $i \in \{1, ..., n\}$ lineair onafhankelijk van elkaar zijn, waarbij de oplossingsvector $\mathbf{b}$ in de kolomruimte van A ligt en de nulruimte van A leeg is. In het tweede geval geldt dat er kolomvectoren a_i van A zijn, die lineair afhankelijk van elkaar zijn. Ook dan ligt de oplossingsvector $\mathbf{b}$ in de kolomruimte van A, maar is de nulruimte niet leeg. En in het laatste geval kunnen we niets over de lineaire afhankelijkheid zeggen, omdat we met een inconsistent stelsel vergelijkingen te maken hebben. Zo'n inconsistent stelsel noemen we ook wel een strijdig stelsel.
Er geldt dat een stelsel vergelijkingen lineair onafhankelijk is, als voor alle kolomvectoren a_i en constante λ_i met $i \in \{1, ..., n\}$, geldt dat als

$$\sum_{i=1}^{n} \lambda_i a_i = 0,$$

dan voor alle i geldt dat $\lambda_i = 0$ en dit de enige oplossing is.
Als er een vector $\boldsymbol{\lambda}$ gevonden wordt waarvoor geldt dat $\boldsymbol{\lambda} \neq \mathbf{0}$, dan geldt dat het stelsel lineair afhankelijk is. Alle oplossingen $\boldsymbol{\lambda} \neq \mathbf{0}$ worden ook wel de nulruimte van de matrix genoemd.

Als we weten dat een stelsel vergelijkingen een oplossing heeft, kunnen we deze gaan bepalen. Dit kan op verschillende manieren, die we hierna kort behandelen.

3.1.1 Rang

Verder kunnen we de mate van oplosbaarheid van $A\mathbf{x} = \mathbf{b}$ bepalen door de rang van A. De rang van een matrix A is gelijk aan het maximale aantal onafhankelijke kolom- of rijvectoren van de matrix A. Als het maximale aantal onafhankelijke kolomvectoren gelijk is aan het maximale aantal onafhankelijke rijvectoren, dan noemen we de matrix A ook wel van volledige rang. Hierbij hoeft A dus geen vierkante matrix te zijn.
Als geldt dat $\text{rang}(A) = \text{rang}(A_b)$, waarbij A_b de matrix A weergeeft die is aangevuld met vector $\mathbf{b}$, dan geldt dat het stelsel oplosbaar is. Immers, er geldt dan dat $\mathbf{b}$ een lineaire combinatie is van de kolomvectoren van A en dus geldt dat $\mathbf{b}$ in de oplossingsruimte van A ligt.

3.1.2 Inverse Matrix

Er geldt voor een $n \times n$ matrix A dat als zijn inverse A^{-1} bestaat, dat het stelsel $A\mathbf{x} = \mathbf{b}$ met $\mathbf{x}, \mathbf{b} \in \mathbb{R}^n$ dan precies één oplossing heeft. Deze oplossing wordt gegeven door

$$\mathbf{x} = A^{-1}\mathbf{b}.$$

We maken hierbij gebruik van de identiteitsregels van matrixvermenigvuldiging:

$$AA^{-1} = A^{-1}A = I$$

Dan zien we inderdaad dat we de unieke oplossing van x kunnen vinden:

$$\begin{aligned} A\mathbf{x} &= \mathbf{b} \\ \mathbf{x} &= A^{-1}\mathbf{b}. \end{aligned}$$

3.1.3 Determinant

Ook met behulp van de determinant van een $n \times n$ matrix kunnen we informatie over de oplosbaarheid verkrijgen.
De determinant van de $n \times n$-matrix A kan worden bepaald door gebruik te maken van Leibniz formule voor determinanten. Deze formule heet de formule van Leibniz of de formule van Laplace en wordt gegeven door

$$\det(A) = \sum_{\sigma \in S_n} \operatorname{sgn}(\sigma) \prod a_{i,\sigma(i)}.$$

Hierbij geeft σ een permutatie van de getallen $1, 2, ..., n$ weer en bepaald de functie $\operatorname{sgn}(\sigma)$ het teken van de permutatie. Er geldt dat

$$\operatorname{sgn}(\sigma) = \begin{cases} 1 & \text{als } \sigma \text{ een even permutatie is} \\ -1 & \text{als } \sigma \text{ een oneven permutatie is.} \end{cases}$$

Een andere manier van het bepalen van de determinant, gebeurt aan de hand van de eigenwaarden. We berekenen de eigenwaarden door het stelsel

$$|A - \boldsymbol{\lambda} I| = 0$$

op te lossen voor de constante λ. Voor een $n \times n$ matrix geldt dat je bij het oplossen n oplossingen voor λ vindt. Hierbij is het mogelijk dat een oplossing vaker voorkomt.
Er geldt nu dat

$$\det(A) = \lambda_1 \cdot \lambda_2 \cdot ... \cdot \lambda_n.$$

Aan de hand van de uitkomst van de determinant, kunnen we het volgende zeggen:

- Als $\det(A) \neq 0$, dan heeft de matrix A één unieke oplossing;
- Als $\det(A) = 0$, dan heeft de matrix A óf geen oplossingen óf meerdere oplossingen.

3.2 Over- en onderbepaalde stelsels

Zoals we in het vorige hoofdstuk hebben gezien, hebben niet alle stelsels altijd een unieke oplossing. We onderscheiden daarin twee verschillende soorten stelsels:

- Overbepaalde stelsels: stelsels met meer vergelijkingen dan onbekenden;
- Onderbepaalde stelsels: stelsels met meer onbekenden dan vergelijkingen.

We merken hierbij op dat het over- of onderbepaald zijn van een stelsel niets zegt over de uniciteit van de oplossing. Een over- of onderbepaald stelsel kan wel degelijk een unieke oplossing hebben. Een over- of onderbepaald stelsel heeft namelijk een unieke oplossing als de oplossingsvector $\mathbf{b}$ in de kolomruimte van A zit en de matrix A van volledige rang is.
In het vervolg gaan we ervan uit dat de over- en onderbepaalde stelsels die we gaan bekijken, geen unieke oplossing hebben. Om voor deze over- en onderbepaalde stelsels toch één oplossing te bepalen, maken we gebruik van de kleinste kwadraten methode om de 'beste' oplossing bepalen. De kleinste kwadraten oplossing van het stelsel $A\mathbf{x} = \mathbf{b}$ wordt gegeven door

$$\min_{\mathbf{x}} \|A\mathbf{x} - \mathbf{b}\|_2^2.$$

We gebruiken hierbij de notatie van de Euclidische norm die de lengte van de vector $\mathbf{x}$ weergeeft. Er geldt

$$\|\mathbf{x}\|_2 = \sqrt{|x_1|^2 + |x_2|^2 + ... + |x_n|^2}.$$

Verder maken we gebruik van de notatie van kwadraatsom van $\mathbf{x}$, gegeven door

$$\|\mathbf{x}\|_2^2 = \sum_n |x_n|^2 = \mathbf{x}^t\mathbf{x}.$$

Dit geeft de mate van afwijking aan. Hierbij geldt dat $\mathbf{x}^t$ de getransponeerde vector van $\mathbf{x}$ weergeeft.

3.2.1 Overbepaald stelsel

In het geval van een overbepaald stelsel, kijken we naar een stelsel in vorm van

$$A\mathbf{x} = \mathbf{b},$$

waarbij A een $m \times n$ matrix met $m > n$. Omdat $A\mathbf{x} = \mathbf{b}$ geen exacte oplossing heeft als $\mathbf{b}$ niet in de kolomruimte van A zit, gaan we er nu vanuit dat $\mathbf{b}$ niet in de kolomruimte zit maar we wel de 'beste' oplossing willen bepalen. Voor de 'beste' oplossing, bekijken we de mate van afwijking tussen het linkerdeel van de vergelijking en het rechterdeel. We bekijken dus

$$f(x) = \|\mathbf{b} - A\mathbf{x}\|_2^2.$$

Omdat we de 'beste' oplossing zoeken, willen we

$$\min_{\mathbf{x}} f(\mathbf{x}) = \min_{\mathbf{x}} \|\mathbf{b} - A\mathbf{x}\|_2^2.$$

Dit kunnen we bepalen door de gradiënt van $f(x)$ te nemen en deze gelijk te stellen aan nul, dus $\nabla f(x) = \mathbf{0}$.

We beginnen eerst met het uitwerken van $\|\mathbf{b} - A\mathbf{x}\|_2^2$.

$$\begin{aligned} f(\mathbf{x}) &= \|\mathbf{b} - A\mathbf{x}\|_2^2 \\ &= (\mathbf{b} - A\mathbf{x})^t(\mathbf{b} - A\mathbf{x}) \\ &= \mathbf{b}^t(\mathbf{b} - A\mathbf{x}) - \mathbf{x}^t A^t(\mathbf{b} - A\mathbf{x}) \\ &= \mathbf{b}^t\mathbf{b} - \mathbf{b}^t A\mathbf{x} - \mathbf{x}^t A^t\mathbf{b} + \mathbf{x}^t A^t A\mathbf{x} \\ &= \mathbf{b}^t\mathbf{b} - 2\mathbf{b}^t A\mathbf{x} + \mathbf{x}^t A^t A\mathbf{x} \end{aligned}$$

Vervolgens kunnen we de gradiënt nemen. We maken daarbij gebruik van het feit dat $\nabla(\mathbf{x}^t A\mathbf{x}) = 2A\mathbf{x}$.

$$\begin{aligned} \nabla(f(\mathbf{x})) &= \mathbf{0} - 2\mathbf{b}^t A + 2A^t A\mathbf{x} \\ &= 2A^t A\mathbf{x} - 2\mathbf{b}^t A \end{aligned}$$

Dit kunnen we vervolgens gelijkstellen aan nul.

$$\begin{aligned} 2A^t A\mathbf{x} - 2\mathbf{b}^t A &= \mathbf{0} \\ 2A^t A\mathbf{x} &= 2\mathbf{b}^t A \\ A^t A\mathbf{x} &= \mathbf{b}^t A \\ A^t A\mathbf{x} &= A^t\mathbf{b} \end{aligned}$$

Als we nu aannemen dat $A^t A$ ons een inverteerbare matrix geeft, geldt dat we de oplossing kunnen vinden door

$$\begin{aligned} A^t A\mathbf{x} &= A^t\mathbf{b} \\ (A^t A)^{-1}A^t A\mathbf{x} &= (A^t A)^{-1}A^t\mathbf{b} \\ \mathbf{x} &= (A^t A)^{-1}A^t\mathbf{b}. \end{aligned}$$

We zien nu dus dat de unieke oplossing wordt gegeven door $\mathbf{x} = (A^t A)^{-1}(A^t\mathbf{b})$. We noemen $A^\dagger = (A^t A)^{-1}A^t$ ook wel de pseudo-inverse genoemd.
We zeggen ook wel dat $A^\dagger$ de linker inverse is als geldt dat

$$A^\dagger A = I$$

en de rechter inverse is als

$$AA^\dagger = I.$$

Er geldt nu dat voor een overbepaald stelsel met volledige rang, dat er een linker inverse bestaat en voor een onderbepaald stelsel met volledige rang, dat er een rechter inverse bestaat.
Met deze gegevens hadden we de afleiding ook direct kunnen doen. Voor het stelsel $A\mathbf{x} = \mathbf{b}$ vermenigvuldigen we beide kanten met de linker inverse. Immers, we hebben een overbepaald stelsel met volledige rang. We zien dan dat de unieke oplossing voor het kleinste kwadraten methode inderdaad wordt gegeven door

$$\begin{aligned} A\mathbf{x} &= \mathbf{b} \\ A^\dagger A\mathbf{x} &= A^\dagger\mathbf{b} \\ \mathbf{x} &= (A^t A)^{-1}A^t\mathbf{b}. \end{aligned}$$

3.2.2 Onderbepaald stelsel

In het geval van een onderbepaald stelsel, kijken we naar een stelsel in vorm van

$$A\mathbf{x} = \mathbf{b},$$

waarbij A een $m \times n$ matrix met $m < n$.
In dit geval kan het zijn dat de matrix een unieke oplossing heeft, maar in het algemeen geldt dat we meerdere oplossingen hebben. We gaan van het laatste geval uit en willen we nu ook weer de 'beste' oplossing bepalen. Dit gaan we doen door de waarden van $\mathbf{x}$ te vinden, waarbij de norm zo klein mogelijk is, oftewel:

$$\min_{\mathbf{x}} \|\mathbf{x}\|_2^2.$$

Omdat we weten dat het stelsel volledige rang heeft, weten we dat de vector $\mathbf{x}$ in de rijruimte van A ligt en dat er een vector $\mathbf{y}$ bestaat zodat $\mathbf{x} = A^t\mathbf{y}$ een oplossing geeft.
Deze vergelijking kunnen we vervolgens invullen in het stelsel $A\mathbf{x} = \mathbf{b}$.

$$\begin{aligned} A\mathbf{x} &= \mathbf{b} \\ AA^t\mathbf{y} &= \mathbf{b} \\ \mathbf{y} &= (AA^t)^{-1}\mathbf{b} \\ (A^t)^{-1}\mathbf{x} &= (AA^t)^{-1}\mathbf{b} \\ \mathbf{x} &= A^t(AA^t)^{-1}\mathbf{b} \\ \mathbf{x} &= A^\dagger\mathbf{b}. \end{aligned}$$

Omdat er van alle oplossingen maar eentje het dichtste bij $\mathbf{0}$ zit, weten we dat deze oplossing uniek is. We merken op dat we in dit geval gebruik hebben gemaakt van de rechter inverse.

We zien dus dat we bij een over- en onderbepaald stelsel gebruik kunnen maken van de kleinste kwadraten methode in combinatie met de pseudo-inverse, om de 'beste' $\mathbf{x}$ te bepalen. De voorwaarde hiervoor was, dat de matrix A van volledige rang is.
We kunnen ons vervolgens af gaan vragen hoe het werkt met matrices die geen volledige rang hebben.

3.2.3 Onvolledige rang

We gaan er nu dus vanuit dat we een matrix A van $m \times n$ hebben, met $m \neq n$ en $\text{Rang}(A) < \min(m, n)$. Als A een onvolledige rang heeft, geldt dat A^{-1} niet bestaat. We noemen zo'n matrix ook wel een singuliere matrix.
Zo'n probleem kunnen we oplossen met de singuliere waarden decompositie. Voor deze methode geldt dat we de matrix A kunnen schrijven als

$$A = U\Sigma V^t.$$

Hierbij gelden de volgende eigenschappen:

- U is een orthogonale matrix van $m \times m$, waarbij geldt dat $U^tU = UU^t = I$;

- Σ is een $m \times n$ matrix gegeven door

$$\Sigma = \begin{pmatrix} \sigma_1 & & & \\ & \sigma_2 & & \\ & & \ddots & \\ & & & \sigma_r \\ & & & \end{pmatrix}$$

  waarbij r de rang is van A;

- V is een orthogonale matrix van $n \times n$, waarbij geldt dat $V^tV = VV^t = I$.

De matrix U en de matrix V Het bepalen van de singuliere waarden decompositie, gaat als volgt. Allereerst bepalen we de $m \times m$ matrix A^tA. Van deze matrix kunnen we vervolgens de eigenwaarden berekenen. We nemen hier de positieve tweedemachtswortel en deze waarden geven vervolgens de singuliere waarden in de matrix Σ weer. Er geldt dat

$$\sigma_1 = \sqrt{\lambda_1} \geq \sigma_2 = \sqrt{\lambda_2} \geq ... \geq \sigma_r = \sqrt{\lambda_r}.$$

Alle andere elementen in de matrix Σ zijn gelijk aan 0.
We kunnen vervolgens matrix V en matrix u bepalen. De eigenvectoren van A^tA zijn de kolommen van de vector V en de eigenvectoren van AA^t.
Omdat we weer de 'beste' oplossing willen bepalen, hebben we weer te maken met een minimaliseerprobleem. We schrijven $\|\mathbf{b} - A\mathbf{x}\| = \|\mathbf{z} - \Sigma\mathbf{y}\|$. We bepalen de $\mathbf{z}$ door $\mathbf{z} = U^t\mathbf{b}$ uit te rekenen. Verder geldt dan voor $\mathbf{y}$:

$$y_i = \begin{cases} \frac{z_i}{\sigma_i} & \text{voor } i = 1, ..., r \\ 0 & \text{voor } i = r+1, ..., m \end{cases}$$

We kunnen nu de unieke oplossing van $\mathbf{x}$ bepalen door het stelsel $\mathbf{x} = V\mathbf{y}$ uit te rekenen. We krijgen dan als oplossing:

$$\mathbf{x} = \sum_{i=1}^{r} \frac{(\mathbf{u}_i)^t\mathbf{b}}{\sigma_i}\mathbf{v}_i.$$

Ondanks dat we ervan uit gaan dat matrix A geen volledige rang heeft, kunnen we met de singuliere waarden methode toch een 'beste' oplossing vinden.
Voor een overbepaald stelsel geldt dat deze 'beste' oplossing de oplossing is waarbij $\mathbf{x}$ zo dicht mogelijk bij $\mathbf{b}$ ligt en voor een onderbepaald stelsel geldt dat deze 'beste' oplossing de oplossing is waarbij de norm van $\mathbf{x}$ zo klein mogelijk is.

4 Binaire Tomografie

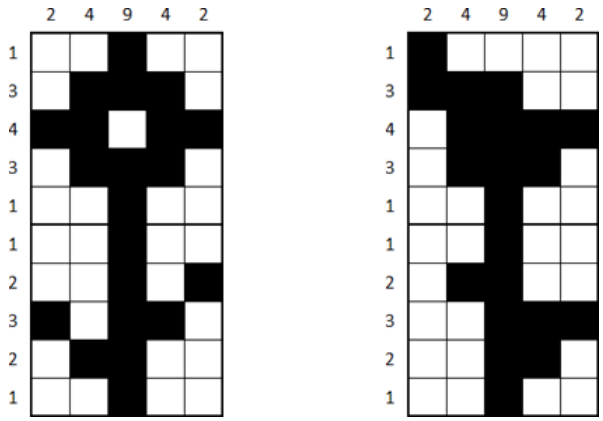
In de tomografie bestaan de pixels of vakjes van een afbeelding uit een bepaalde waarde $x \in [0, 1]$. Hierbij representeert de waarde 0 een wit vakje, de waarde 1 een zwart vakje en elke waarde tussen 0 en 1 een grijstint. Verder werd de waarde I_{uit} bepaald door de wet van Beer-Lambert toe te passen.
We maken nu de overstap naar binaire tomografie. In de binaire tomografie, geldt dat $x \in \{0, 1\}$. Oftewel, ieder vakje heeft of waarde 0 of waarde 1.
De waarde I_{uit} gaan we in de binaire tomografie niet meer bepalen aan de hand van de wet van Beer-Lambert, maar gaan we bepalen door de som van de waarden die de straal tegen is gekomen. Er geldt dus vanaf nu:

$$I_{\text{uit}} = \sum_{i \in I} x_i.$$

Hierbij is x_i het i-de vakje, op dezelfde manier bepaald als in figuur 4 maar dan met $\mu_i = x_i$. We versimpelen hier het probleem door aan te nemen dat de lengte van de straal door een vakje niet meer uitmaakt.

4.1 Oplosbaarheid

In het geval dat de pixels of vakjes van een afbeelding alle waarden tussen 0 en 1, inclusief 0 en 1, kunnen aannemen, zijn er oneindig veel mogelijkheden voor de waarde van ieder vakje. Het aantal mogelijke verschillende plaatjes zijn er dan dus ook oneindig veel.
In het geval dat ieder vakje maar 2 mogelijke waarden heeft, namelijk 0 en 1, zien we dat het aantal mogelijke verschillende afbeeldingen van $n \times n$ wordt gegeven door 2^{n^2}. Vanaf nu definiëren we $N = n^2$.
We nemen verder aan dat de data van alle rijen en alle kolommen bekend zijn, dus dat we van alle rijen de som weten en van alle kolommen de som weten. Omdat voor $n > 2$ dan geldt dat we meer onbekenden hebben dan vergelijkingen, volgt hieruit dat we vanaf $n > 2$ te maken hebben met een onderbepaald stelsel. Er geldt dan in het algemeen dat bij de verkregen data, meerdere mogelijke afbeeldingen zijn te genereren. Hieronder staat een voorbeeld van twee afbeeldingen die dezelfde dataset van rijsommen en kolomsommen heeft.



Figuur 5: Twee verschillende afbeeldingen behorende bij dezelfde dataset

4.2 Algoritme

We hebben gezien dat een stelsel van lineaire vergelijkingen één unieke, geen of meerdere oplossingen heeft. In het eerste geval is het bepalen van de oplossing niet zo moeilijk, net zoals in het tweede geval. Immers, als we kunnen bepalen dat een stelsel geen oplossing heeft, zijn we al meteen klaar. Dit kunnen we makkelijk zien, als de totale kolomsom niet gelijk is aan de totale rijsom, hebben we inderdaad een strijdig stelsel en dus geen oplossingen.
Als een stelsel echter meerdere oplossingen heeft, kunnen we tevreden zijn met een willekeurige oplossing die voldoet of kunnen we juist alle mogelijke oplossingen willen bepalen. We nemen nu aan dat we inderdaad alle mogelijke oplossingen willen bepalen, zodat we aan de hand van een dataset alle mogelijke afbeeldingen kunnen genereren.
We gaan een programma schrijven die voor ons alle mogelijke binaire afbeeldingen in matrixvorm weergeeft, gegeven een bepaalde set aan vergelijkingen. We beginnen met het genereren van alle mogelijke afbeeldingen zonder via het brute force principe. Deze gaan we vervolgens uitbreiden, zodat we ook lineaire vergelijkingen in kunnen voeren. Deze geeft, tevens via brute force, alleen alle afbeeldingen die aan de ingevoerde restricties voldoet.
Omdat we dan in principe een werkend algoritme hebben, zouden we het kunnen zien als een geslaagd programma. Maar gezien de tijdscomplexiteit (welke na iedere methode wordt onderbouwd), willen we dit algoritme gaan optimaliseren.

4.2.1 Brute force (zonder lineaire vergelijkingen)
Voor de code: zie Appendix A.

In eerste instantie bepalen we alle mogelijkheden van een afbeelding van $n \times n$ vakjes. Er wordt eerst gevraagd om n in te voeren, de grootte van de matrix. Vervolgens worden alle mogelijke combinaties van matrices gegenereerd en afgedrukt op het scherm.
Bij een gegeven n, wordt het totaal van aantal vakjes in de afbeelding gegeven door $N = n^2$. Omdat alle afbeeldingen uniek zijn en bestaan uit alleen maar nullen en enen, bepalen we de afbeeldingen door alle rijen achter elkaar te plakken.
Input:

```
Hoeveel rijen/kolommen heeft de matrix?
>>>2
```

Output:

```
[[0  0]
 [0  0]]
[[0  0]
 [0  1]]
[[0  0]
 [1  0]]
[[0  0]
 [1  1]]
[[0  1]
 [0  0]]
[[0  1]
 [0  1]]
[[0  1]
 [1  0]]
[[0  1]
 [1  1]]
```

```
[[1  0]
 [0  0]]
[[1  0]
 [0  1]]
[[1  0]
 [1  0]]
[[1  0]
 [1  1]]
[[1  1]
 [0  0]]
[[1  1]
 [0  1]]
[[1  1]
 [1  0]]
[[1  1]
 [1  1]]
```

Voor de matrix $\begin{pmatrix} x_1 & x_2 \\ x_3 & x_4 \end{pmatrix}$ geldt dat we dit dus representeren als $x_1 \quad x_2 \quad x_3 \quad x_4$.

Voor iedere verschillende matrix geldt dat deze representatie uniek is. Verder kunnen we deze representaties zien als binaire getallen, waarbij we zien dat in een $n \times n$ matrix de representaties alle getallen van 0 tot en met 2^{N-1} aan doen.
Om brute force alle matrices te bepalen, gaan we dus andersom te werk. We bepalen alle binaire getallen van 0 tot en met $2^N - 1$ en drukken deze af op het scherm in matrix-vorm.

Tijdscomplexiteit
We hebben 2^N operaties nodig om alle mogelijke matrices te vormen. Immers, er zijn voor alle N vakjes 2 mogelijkheden. De tijdcomplexiteit van dit algoritme is dus gelijk aan $\mathcal{O}(2^N)$.

4.2.2 Brute force (met lineaire vergelijkingen) Voor de code: zie Appendix B.

Aan de hand van het brute force programma dat in 4.1 beschreven staat, willen we nu een programma schrijven die alleen de matrices bepalen die aan bepaalde restricties voldoen. Deze restricties worden gevormd door de verkregen lineaire vergelijkingen.
We formuleren eerst de methode om deze gegevens in te lezen. De in te lezen gegevens zijn de volgende:

- Het getal n: het aantal kolommen en rijen van de matrix;
- de rijsommen r_i met $i \in [1, ..., n]$;
- De kolomsommen k_i, met $i \in [1, ..., n]$

Input:

```
Hoeveel rijen/kolommen heeft de matrix?
>>>2
Vul de som van rij 1 in:
>>>1
Vul de som van rij 2 in:
>>>1
Vul de som van kolom 1 in:
>>>1
Vul de som van kolom 2 in:
>>>1
```

Hierbij slaan we alle lineaire vergelijkingen op in een lijst.
Met deze gegevens gaan we voortborduren op het al bekende brute force algoritme zoals beschreven in hoofdstuk 4.2.1. We bepalen nog steeds alle mogelijke matrices zonder restricties. Per geconstrueerde matrix gaan we vervolgens controleren of aan alle restricties wordt voldaan. We bekijken de eerste ingevoerde vergelijking en lezen deze in. We kunnen vervolgens bepalen of in de gegenereerde matrix de lineaire vergelijking klopt. Als dit waar is, dan bekijken we de volgende lineaire vergelijking, en zo voorts. Als dit niet waar is, dan zijn we klaar met deze matrix en genereren we een nieuwe matrix (tot we alle matrices hebben gegenereerd).
Als de gegenereerde matrix aan alle n lineaire vergelijkingen voldoet van de rijen en alle n vergelijkingen van de kolommen voldoet, dan drukken we deze af op het scherm en ook dan genereren we een nieuwe matrix (wederom tot we alle matrices hebben gegenereerd).
Bij bovenstaande code kunnen we dan de output bepalen.

Output:

```
[[0  1]
 [1  0]]
[[1  0]
 [0  1]]
```

Tijdscomplexiteit
Ook in dit geval hebben we 2^N operaties nodig om alle mogelijke matrices te vormen. Immers, er zijn voor alle N vakjes 2 mogelijkheden. Per gegenereerde matrix moeten ver vervolgens maximaal m keer controleren of er aan de gegeven lineaire vergelijking wordt voldaan. De tijdscomplexiteit is dus gelijk aan $O(2^N)$.

4.2.3 Optimalisatie met permutaties
Voor de code: zie Appendix C.

Het brute force zoals in 4.2.2 staat beschreven, bepaald alle mogelijke afbeeldingen en controleert of deze aan de voorwaarden voldoet. Er zijn makkelijke voorbeelden te bedenken waarbij het algoritme alle afbeeldingen vrij snel heeft gevonden, maar dan alsnog alle andere mogelijkheden langs gaat om te checken of deze voldoen. Als de som van alle rijen/kolommen bijvoorbeeld gelijk is aan 1, dan zal maar één vakje van de n^2 vakjes een 1 bevatten.
Omdat we de totale som van de rijen/kolommen kunnen bepalen, weten we hoeveel enen (en dus hoeveel nullen) er in onze afbeelding zitten. We bepalen vervolgens één zo'n binaire waarde met het aantal nullen en enen die in onze afbeelding zitten. Vervolgens maken we een lijst met alle permutaties van dit getal, waarbij gelijke waarden er worden uitgefilterd. Immers, een permutatie van '001' wordt gegeven door de eerste 0 met de tweede 0 te wisselen en geeft '001'. We weten dat alle binaire waarden van de mogelijke afbeeldingen, in onze lijst zitten. Door dezelfde test uit te voeren als in het algoritme van hoofdstuk 4.2.2, kunnen we dus bepalen of deze waarden inderdaad juiste afbeeldingen genereren.
Het volgende voorbeeld genereerd significant alle mogelijke afbeeldingen sneller.
Input:

```
Hoeveel rijen/kolommen heeft de matrix?
5
Vul de som van rij 1 in:
>>>1
Vul de som van rij 2 in:
>>>2
Vul de som van rij 3 in:
>>>0
Vul de som van rij 4 in:
>>>2
Vul de som van rij 5 in:
>>>0
Vul de som van kolom 1 in:
>>>2
Vul de som van kolom 2 in:
>>>2
Vul de som van kolom 3 in:
>>>0
Vul de som van kolom 4 in:
>>>0
Vul de som van kolom 5 in:
>>>1
```

Output:

```
[[0 0 0 0 1]
 [1 1 0 0 0]
 [0 0 0 0 0]
 [1 1 0 0 0]
 [0 0 0 0 0]]

[[0 1 0 0 0]
 [1 0 0 0 1]
 [0 0 0 0 0]
 [1 1 0 0 0]
 [0 0 0 0 0]]

[[0 1 0 0 0]
 [1 1 0 0 0]
 [0 0 0 0 0]
 [1 0 0 0 1]
 [0 0 0 0 0]]

[[1 0 0 0 0]
 [0 1 0 0 1]
 [0 0 0 0 0]
 [1 1 0 0 0]
 [0 0 0 0 0]]

[[1 0 0 0 0]
 [1 1 0 0 0]
 [0 0 0 0 0]
 [0 1 0 0 1]
 [0 0 0 0 0]]
```

Waar we met het brute force algoritme minuten bezig waren om alle afbeeldingen te genereren, kost dit algoritme ons maximaal een aantal seconden. We zullen later laten zien dat deze methode voor dit voorbeeld sneller is dan de brute force methode die beschreven staat in hoofdstuk 4.2.2.

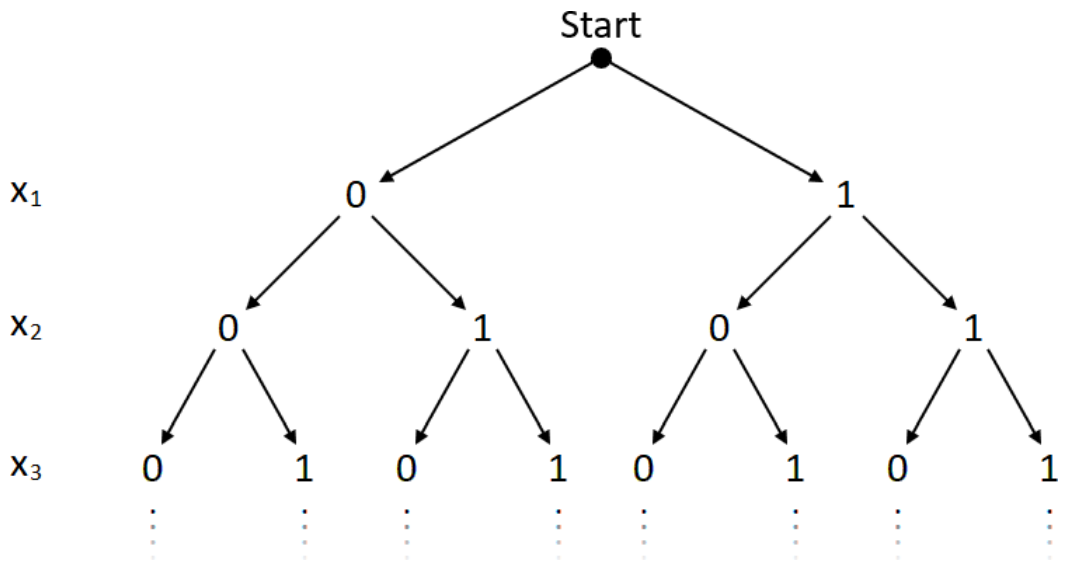
Tijdscomplexiteit

Het bepalen van alle permutaties van een getal N, kost ons $O(N!)$ tijd in de meest ongunstige gevallen. Deze ongunstige gevallen zijn de gevallen waarbij het aantal enen en het aantal nullen vrij dicht bij elkaar liggen[3]. In het geval dat je veel meer enen dan nullen hebt (of andersom), dan hoef je minder permutaties te berekenen en is het algoritme wel degelijk veel sneller dan de brute force methode zoals beschreven in hoofdstuk 4.2.2.

4.2.4 Optimalisatie met boomstructuur

Voor de code: zie Appendix D.

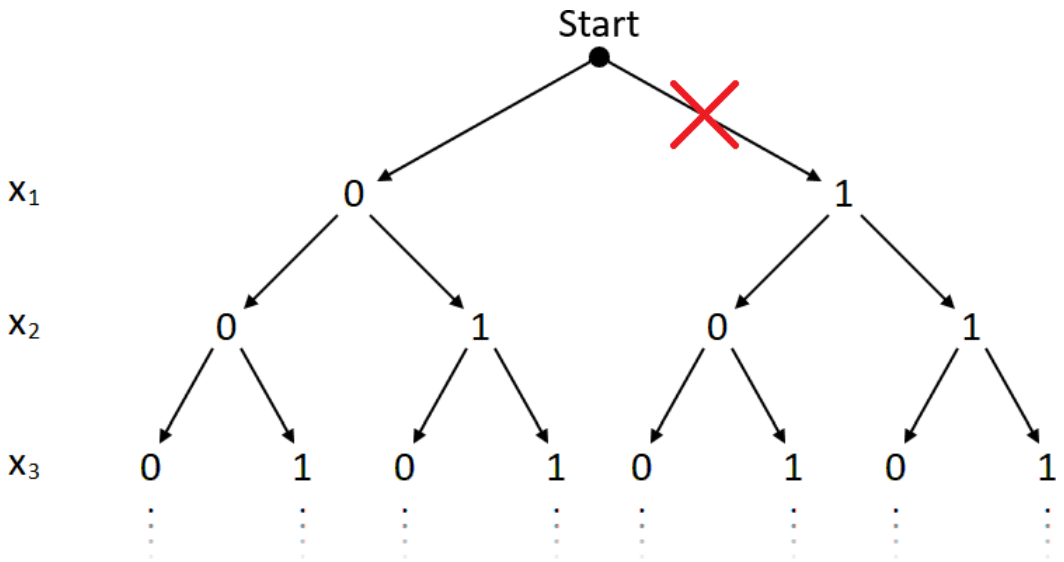
Omdat we net zagen dat het optimaliseren van ons brute force algoritme zoals beschreven in hoofdstuk 4.2.2 ons geen optimalisatie voor alle n heeft gegeven. We laten de eerste beschreven algoritmes los en gaan nu naar een andere methode kijken. We bekijken eerst het eerste vakje en vragen ons af of hier een 1 en een 0 in kan staan. Als dit het geval is, dan kiezen we een van de twee opties en gaan we naar het volgende vakje. Gegeven dat vakje 1 genoteerd wordt door x_1 en zo voorts, krijgen we de volgende boomstructuur.



Figuur 6: De boomstructuur.

[3]Voor $n = 5$ met 12 nullen en 13 enen is het algoritme al niet meer uitvoerbaar omdat het te veel geheugen gebruikt.

Als we de volledige boom hebben bepaald, zijn alle uiteinden van de takken (ook wel 'blaadjes' genoemd) alle verschillende mogelijke afbeeldingen. Het pad naar het blad geeft dan aan welke vakjes de waarde 0 en welke vakjes waarde 1 hebben.
Als geldt dat we in een vakje alleen een 0 in mogen vullen, dan kunnen we de tak naar 1 afknippen en andersom. Alles wat er dan aan deze tak hangt, wordt onbereikbaar en zal nooit tot een oplossing leiden. Het aantal mogelijke afbeeldingen slinkt daar dus mee.



Figuur 7: Het geval waar x_1 geen waarde 1 aan kan nemen.

We zien in figuur 7 dus dat de tak naar $x_1 = 1$ wordt afgeknipt en dat de boom daar mee de helft van zijn blaadjes dus zal verliezen.
In het algoritme kijken we eerst naar vakje 1, dan naar vakje 2, enzovoorts. Bij elk vakje controleren we of het al vast staat welke waarde dit vakje zal krijgen. We onderscheiden hier de volgende mogelijkheden:

- De waarde van de rij staat vast:
  - De som van de rij waarin het vakje ligt is gelijk aan 0, dus het vakje moet ook waarde 0 krijgen;
  - De som van de rij waarin het vakje ligt is gelijk aan n, dus het vakje moet waarde 1 krijgen.
- De waarde van de kolom staat vast:
  - De som van de kolom waarin het vakje ligt is gelijk aan 0, dus het vakje moet ook waarde 0 krijgen;
  - De som van de kolom waarin het vakje ligt is gelijk aan n, dus het vakje moet waarde 1 krijgen.
- De waarde van de rij vanaf dit vakje staat vast:
  - De som van de rij waarin het vakje staat is gelijk aan de al ingevulde vakjes in die rij, dus het vakje (en zijn opvolgende vakjes in dezelfde rij) moet waarde 0 krijgen;

– De som van de rij waarin het vakje ligt, is gelijk aan het nog in te vullen vakjes in die rij, dus het vakje (en zijn opvolgende vakjes in dezelfde rij) moeten waarde 1 krijgen.

Elke keer als we een vakje hebben gehad die vast staat, gaan we een vakje verder en vragen we ons weer hetzelfde af.
Als een vakje niet vast staat, dan kunnen we kiezen of deze 0 of 1 gaat aannemen. Omdat we recursief werken, zullen beide gevallen worden uitgewerkt. We nemen daarom aan dat als de keuze niet vastligt, dat we dan eerst het geval $x_i = 0$ bekijken en, als deze tak helemaal is doorgelopen, daarna $x_i = 1$ bekijken.
Zoals we net al aanhaalde, werken we recursief. Dat wil zeggen dat we in een bepaalde methode, dezelfde methode weer aanroepen. Zo construeren we de datastructuur zoals de boom die we in figuur 6 hebben weergegeven.

Tijdscomplexiteit
De tijdscomplexiteit van dit algoritme met een boomstructuur zoals wij deze hebben gebruikt, wordt gegeven door $O(2^N)$. Dit is het slechtste geval en aangezien we altijd wel takken kunnen knippen. Immers, de laatste pixels op een rij en kolom staan altijd vast. Dus zal de looptijd altijd strikt korter zijn dan $O(2^N)$. Er geldt nu dan dus dat dit voor alle n sneller is dan $O(N!)$ of $O(2^N)$. Dus we hebben inderdaad een sneller algoritme geproduceerd.

4.3 Vervolgoptimalisatie

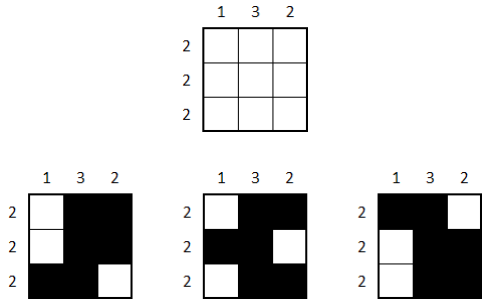
Dat het optimalisatiealgoritme uit hoofdstuk 4.2.4 werkt en gewenste resultaten laat zien wat betreft de looptijd, zullen we in het volgende hoofdstuk gaan zien. Dit betekent echter niet dat het algoritme niet efficiënter kan. Er zijn namelijk implementaties te bedenken die dit algoritme nog verder zullen optimaliseren.
We kunnen het volgende voorbeeld onder andere nog implementeren. We hebben net al geconstateerd dat hoe hoger in de boom een tak wordt afgeknipt, hoe minder mogelijkheden je overhebt om te moeten doorlopen. In het meest ideale geval is het eerste hokje dus een hokje waarvan we de waarde meteen kunnen bepalen.
Dit brengt ons op het volgende idee, wat als we nu de rijen van volgorde veranderen. Dat wil zeggen, dat we de rijen eerst op een andere manier ordenen en aan het einde weer terugzetten. Door rijen te verwisselen, zullen de kolomsommen immers niet veranderen dus kunnen we dit doen zonder dat dit effect heeft op de mogelijke afbeeldingen.
We weten dat de rijen met som n en som 0 bekend zijn. We willen de rijen dan dus zo sorteren, dat deze bovenaan staan, gevolgd door de rijen met som $n - 1$ en som 1, enzovoorts.
Er zijn uiteraard nog meer mogelijke optimalisaties, maar die laat ik aan de lezer over om te bedenken en eventueel uit te voeren.

5 Resultaten

Hieronder zullen we een aantal voorbeelden geven van de looptijden van de algoritmen die zijn beschreven in de hoofdstukken 4.2.2, 4.2.3 en 4.2.4.
We zullen bij ieder voorbeeld de data set en (een deel van) de mogelijke afbeeldingen weergeven, waarna we in een staafgrafiek en een tabel de gemeten looptijd in seconden tegen elkaar af zullen zetten.
Alle metingen zijn onder dezelfde omstandigheden genomen.

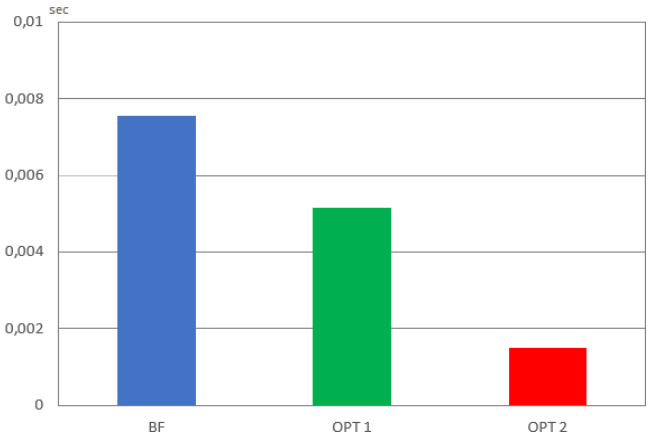
5.1 Voorbeeld met $n = 3$

We bekijken het volgende voorbeeld voor $n = 3$ en laten alle drie de algoritmen de bijbehorende mogelijke afbeeldingen bepalen.



Figuur 8: Voorbeeld van een dataset van $n = 3$ met de bijbehorende oplossingen.

Als we dit voorbeeld door de drie algoritmen laten gaan, krijgen we de volgende gegevens.



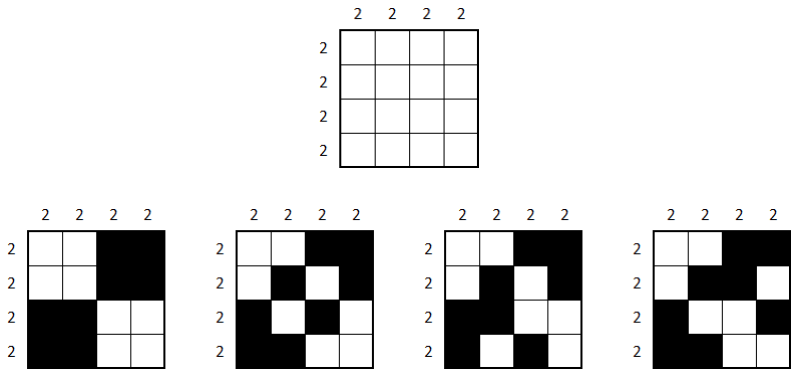
	Naam	Algoritme	Tijd in sec
	BF	Brute force	0,007566
	OPT 1	Optimalisatie (permutatie)	0,005151
	OPT 2	Optimalisatie (boomstructuur)	0,001497

Figuur 9: De looptijden van de algoritmen

We zien dat beide optimalisatiealgoritmen voor een snellere looptijd hebben gezorgd. Het optimalisatiealgoritme met de boomstructuur is $80,21\%$ sneller, terwijl het optimalisatiealgoritme met de permutaties maar $31,92\%$ sneller is. Hier zien we dus dat de optimalisatie met de permutaties voor deze dataset wel een optimalisatie is, maar significant minder effectief is dan de optimalisatie met de boomstructuur.

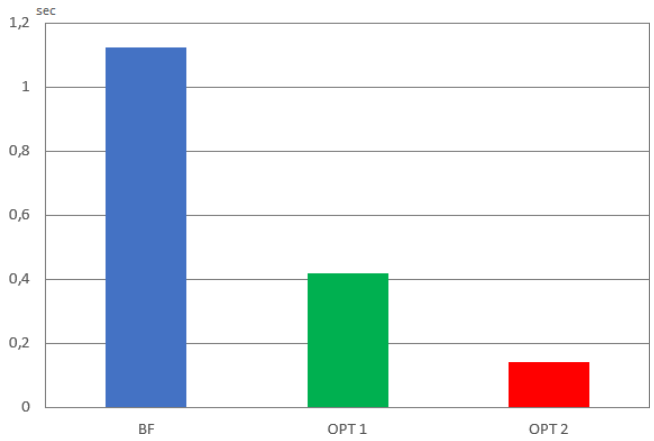
5.2 Voorbeeld met $n = 4$

We bekijken het volgende voorbeeld voor $n = 4$ en laten weer alle drie de algoritmen de bijbehorende mogelijke afbeeldingen bepalen.



Figuur 10: Voorbeeld van een dataset van $n = 4$ met de eerste 4 bijbehorende oplossingen.

We hebben hierboven de eerste 4 mogelijke oplossingen afgebeeld die de algoritmen hebben gegenereerd. Deze data heeft in totaal 90 mogelijke afbeeldingen die voldoen aan de dataset. Als we dit voorbeeld door de drie algoritmen laten gaan, krijgen we de volgende gegevens.



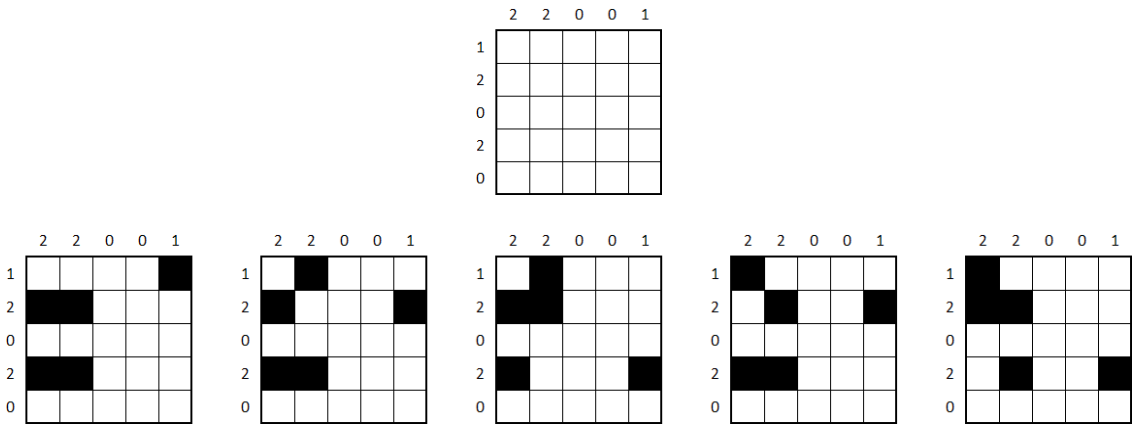
	Naam	Algoritme	Tijd in sec
	BF	Brute force	1,12438
	OPT 1	Optimalisatie (permutatie)	0,41896
	OPT 2	Optimalisatie (boomstructuur)	0,14103

Figuur 11: De looptijden van de algoritmen

We zien ook nu dat beide optimalisatiealgoritmen voor een snellere looptijd hebben gezorgd. Het optimalisatiealgoritme met de permutaties is $62,74\%$ sneller en het optimalisatiealgoritme met de boomstructuur is $87,46\%$ sneller. Hier zien we dus dat beide optimalisaties significant sneller zijn voor deze dataset, al blijft de optimalisatie met de boomstructuur effectiever.

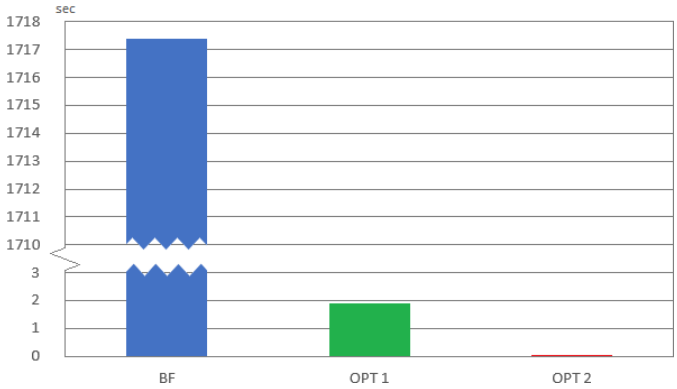
5.3 Voorbeeld met $n = 5$

We bekijken het volgende voorbeeld voor $n = 5$ en laten weer alle drie de algoritmen de bijbehorende mogelijke afbeeldingen bepalen.



Figuur 12: Voorbeeld van een dataset van $n = 5$ met de bijbehorende oplossingen.

Als we dit voorbeeld door de drie algoritmen laten gaan, krijgen we de volgende gegevens.



	Naam	Algoritme	Tijd in sec
	BF	Brute force	1717,379
	OPT 1	Optimalisatie (permutatie)	1,908
	OPT 2	Optimalisatie (boomstructuur)	0,003

Figuur 13: De looptijden van de algoritmen

We zien ook nu dat beide optimalisatiealgoritmen voor een snellere looptijd hebben gezorgd. Het optimalisatiealgoritme met de permutaties is $99,89\%$ sneller en het optimalisatiealgoritme met de boomstructuur is $99,99\%$ sneller. Hier zien we dus dat beide optimalisaties significant sneller zijn voor deze dataset en ondanks dat het verschil tussen de twee optimalisaties nog wel zichtbaar is, zijn het effectieve optimalisaties.

6 Conclusie

Allereerst kunnen we concluderen dat we voor een kleine afbeelding met veel verschil tussen het aantal zwarte en witte vakjes, we een optimalisatie hebben gevonden zoals beschreven in hoofdstuk 4.2.3 die ons sneller alle mogelijke afbeeldingen geeft dan een brute force algoritme zoals beschreven in hoofdstuk 4.2.2. Echter willen we dat het optimalisatiealgoritme ook voor grotere waarden van n sneller de mogelijke afbeeldingen geeft, maar is dit helaas niet het geval. Voor een grotere afbeelding met weinig verschil tussen het aantal nullen en enen is het algoritme zelfs fors langzamer. We kunnen hier dus uit concluderen dat we een optimalisatie hebben gevonden voor specifieke gevallen, maar dat we ons doel daar nog niet mee hebben bereikt.
Verder kunnen we concluderen dat de optimalisatie zoals beschreven in hoofdstuk 4.2.4 een veel snellere alle afbeeldingen geeft voor een hogere n, dan een brute force algoritme zoals beschreven in hoofdstuk 4.2.2. Ongeacht de grootte van n en de verhouding tussen nullen en enen, geeft dit optimalisatiealgoritme altijd sneller alle afbeeldingen dan met een brute force methode. Hieruit concluderen we dus dat we het doel hebben bereikt en een optimalisatiealgoritme hebben gevonden.
Echter betekent dit niet dat voor een grote waarde van n alle afbeeldingen in een korte van tijd worden gegeven. De optimalisatie de we hebben geschreven voldoet dus wel, maar voor een grote n is het wenselijk om dit algoritme nog verder te optimaliseren. Dat kan bijvoorbeeld door de rijen te sorteren. Als we beginnen met de rijen die bestaan uit alleen nullen of enen, zullen we sneller door de boom heen kunnen lopen. Immers, hoe hoger een tak kan worden geknipt, hoe meer mogelijkheden er wegvallen.
Al met al hebben we dus een minder effectief algoritme en een effectief algoritme ontwikkeld om alle mogelijke afbeeldingen bij een gegeven dataset te reconstrueren.

Referenties

[1] "Tomografie". *Wikipedia*, Wikimedia Foundation, 21 juli 2016, https://nl.wikipedia.org/wiki/Tomografie. Geraadpleegd op 15 november 2018.

[2] Seeram, E., *Computed Tomography: Physical Principles, Clinical Applications, and Quality Control*, Elsevier Health Sciences, 4de editie, 2016.

[3] Garrett, P., *Absorption and Transmission of light and the Beer-Lambert Law*, Lecture notes 'Elementary Physics', 2006.

[4] Beukers, F., *Lineaire Algebra*, Departement Wiskunde UU, 2014.

[5] Ascher, U.M.,& Greif, C., *A First Course on Numerical Methods*, Society for Industrial and Applied Mathematics, 5de editie , 2011.

[6] Selesnick, I., *Least Squares with Examples in Signal Processing*, Lecture notes 'Digital Signal Processing I', 2013.

[7] "Singulierewaardeontbinding". *Wikipedia*, Wikimedia Foundation, 10 mei 2018, https://nl.wikipedia.org/wiki/Singulierewaardenontbinding. Geraadpleegd op 4 januari 2018.

[8] Cormen, T.H., Leiserson, C.E., Rivest, R.L.,& Stein, C., *Introduction to Algorithms*, MIT Press Ltd, 3de editie , 2009.

Appendices

A Brute force I

```
import numpy
n = int(input('Hoeveel rijen/kolommen heeft de matrix?\n'))
spaces = n*n
X = numpy.ndarray(shape=(n,n))
for i in range(1 << n*n):
    row = 0
    start = 0
    end = n
    i_bin = bin(i)[2:].zfill(n*n)
    add = 0
    while row < n:
        for j in range(start, end):
            X[row,j] = i_bin[j+add]
        row +=1
        add = row*n
    X = X.astype(int)
    print(X)
```

B Brute force II

```
import numpy

n = int(input('Hoeveel rijen/kolommen heeft de matrix?\n'))
spaces = n*n #Het aantal vakjes in de matrix

lin_eq = [] #De lijst met alle lineaire vergelijkingen

m = 0
while (m < n): #Invoer van de rijen
    m_str = str(m+1)
    text = str(input('Vul de som van rij ' + m_str + ' in: '))
    lin_eq.append(text)
    m+=1

m = 0
while (m < n): #Invoer van de kolommen
    m_str = str(m+1)
    text = str(input('Vul de som van kolom ' + m_str + ' in: '))
    lin_eq.append(text)
    m+=1

index = 0

X = numpy.ndarray(shape=(n,n)) #"Lege" matrix (deafault waarden zijn != 0)

def check(k): #x = nr van lineaire vergelijking uit lin_eq
    total = 0 #De som van de waarden in de nog te doorlopen vakjes
    sum_k = int(lin_eq[k]) #De som die is ingevoerd

    pos = n - 1
    while pos >= 0:
        if(k < n): #Het gaat om een rij
            total += X[k, pos]
            pos-=1
        else: #Het gaat om een kolom
            total += X[pos,k-n]
            pos-=1
    if (sum_k == total):
        return True #Geef 'True' terug als het klopt
    else:
        return False #Geef 'False' terug als het niet klopt


for i in range(1 << n*n):
    row = 0
    start = 0
    end = n
    i_bin = bin(i)[2:].zfill(n*n)
    add = 0
    while row < n: #Het omschrijven van binair getal naar matrix
        for j in range(start, end):
            X[row,j] = int(i_bin[j+add])
        row +=1
        add = row*n
    X = X.astype(int)
    boolean = True
    for k in range(0, len(lin_eq)): #Check elke vergelijking
        if (boolean == True):
```

```
59                 boolean = check(k)
60             else:
61                 break
62         if (boolean == True):
63             index+=1
64             print(X)
```

C Optimalisatie I

```
import numpy
from sympy.utilities.iterables import multiset_permutations

n = int(input('Hoeveel rijen/kolommen heeft de matrix?\n'))
spaces = n*n #Het aantal vakjes in de matrix

lin_eq = [] #De lijst met alle lineaire vergelijkingen

m = 0
while (m < n): #Invoer van de rijen
    m_str = str(m+1)
    text = str(input('Vul de som van rij ' + m_str + ' in:\n'))
    lin_eq.append(int(text))
    m+=1

m = 0
while (m < n): #Invoer van de kolommen
    m_str = str(m+1)
    text = str(input('Vul de som van kolom ' + m_str + ' in:\n'))
    lin_eq.append(int(text))
    m+=1

index = 0

#Check rijsom gelijk is aan de kolomsom:
row = 0
column = 0
for i in range(0,n):
    row += lin_eq[i]
    column += lin_eq[i+n]

if(row != column):
    print("Strijdig stelsel!")
    exit()


X = numpy.ndarray(shape=(n,n)) #"Lege" matrix (deafault waarden zijn != 0)

def check(k): #x = nr van lineaire vergelijking uit lin_eq
    total = 0 #De som van de waarden in de nog te doorlopen vakjes
    sum_k = int(lin_eq[k]) #De som die is ingevoerd

    pos = n - 1
    while pos >= 0:
        if(k < n): #Het gaat om een rij
            total += X[k, pos]
            pos-=1
        else: #Het gaat om een kolom
            total += X[pos,k-n]
            pos-=1
    if (sum_k == total):
        return True #Geef 'True' terug als het klopt
    else:
        return False #Geef 'False' terug als het niet klopt

bin2 = ""
for i in range (0, row):
    bin2 += "1"
```

```
for j in range(row, n*n):
    bin2 += "0"

string = str(bin2)

var = []
for i in range(0,n):
    var.append(bin2[i])

var2 = list(multiset_permutations(string))

for i in range(0, len(var2)):
    number = ""
    for j in range(0, len(var2[i])):
        number += str(var2[i][j])

    i_bin2 = int(number, base=2)

    rows = 0
    start = 0
    i_bin = bin(i_bin2)[2:].zfill(n*n)
    end = n
    add = 0
    while rows < n: #Het omschrijven van binair getal naar matrix
        for j in range(start, end):
            X[rows,j] = int(i_bin[j+add])
        rows +=1
        add = rows*n
    X = X.astype(int)
    boolean = True
    for k in range(0, len(lin_eq)): #Check elke vergelijking
        if (boolean == True):
            boolean = check(k)
        else:
            break
    if (boolean == True):
        index+=1
        print(X)
```

D Optimalisatie II

```
import numpy

n = int(input('Hoeveel rijen/kolommen heeft de matrix?\n'))
spaces = n*n #Het aantal vakjes in de matrix

row_sums = [] #De lijst met alle rijsommen
col_sums = [] #De lijst met alle kolomsommen

m = 0
while (m < n): #Invoer van de rijen
    m_str = str(m+1)
    text = str(input('Vul de som van rij ' + m_str + ' in:\n'))
    row_sums.append(int(text))
    m+=1

m = 0
while (m < n): #Invoer van de kolommen
    m_str = str(m+1)
    text = str(input('Vul de som van kolom ' + m_str + ' in:\n'))
    col_sums.append(int(text))
    m+=1

#Check rijsom gelijk is aan de kolomsom:
row_i = 0
column_i = 0
for i in range(0,n):
    row_i += row_sums[i]
    column_i += col_sums[i]

if(row_i != column_i):
    print("Strijdig stelsel!")
    exit()

X = numpy.ndarray(shape=(n,n)) #"Lege" matrix (deafault waarden zijn != 0)
X = X.astype(int)

M = [] #Lijst met alle pixels (hokjes)

class Pixel:
  def __init__(self, number, row, col, first_time, fixed, flag):
    self.number = number
    self.row = row
    self.col = col
    self.first_time = first_time
    self.fixed = fixed
    self.flag = flag

for i in range(0, n*n):
    col = (i % n)
    row = (i - col)/n
    y = Pixel(i,row,col,False,False,False)
    M.append(y)

def CheckFixed(cur_sum, i):
    if cur_sum == row_sums[M[i].row]:
        return True
    elif cur_sum == row_sums[M[i].row] - (n - (i%n)):
        return True
```

```
    elif col_sums[i%n] == 0:
        return True
    elif col_sums[i%n] == n:
        return True
    else:
        return False

def FixedValue(cur_sum, i):
    if cur_sum == row_sums[M[i].row] or col_sums[i%n] == 0:
        return 0
    else:
        return 1

def Step(cur_sum, i):
    if (i > 0 and M[i-1].row == n-1): #Als laatste regel: check of colsum klopt!
        col_sum = 0
        for j in range(0, n):
            col_sum += X[j,M[i-1].col]
        if(col_sum == col_sums[M[i-1].col]): #Als kolomsom WEL klopt
            pass
        else: #Als kolomsom NIET klopt
            return

    if (i == n*n):
        print(X)
        return

    else:
        if(i % n == 0):
            cur_sum = 0

        fixed = CheckFixed(cur_sum, i)
        if fixed == True:
            M[i].fixed = True
            value = FixedValue(cur_sum, i) #Waarde van vakje i bepalen
            X[M[i].row,M[i].col] = value #Vakje i in X waarde geven
            cur_sum += value #Huidige som van de rij aanpassen
            Step(cur_sum, i+1)
        else:
            X[M[i].row, M[i].col] = 0
            Step(cur_sum, i+1)

            X[M[i].row, M[i].col] = 1
            Step(cur_sum+1, i+1)

#BEGIN PROGRAMMA:
i = 0
cur_sum = 0
Step(cur_sum,i)
```