Handelshøjskolen i København
Statistikgruppen
Erhvervsøkonomi-matematik-studiets 4. semester
2003

C++-program til løsning af LP-problemer vha. simplex-baseret metode

Lene Hansen
leha01ad

Morten Høgholm
moho01ab

Opponenter:
Casper Rolf Garrelts, caga01ab
Jesper Jensen, jeje01ae

Vejleder:
H. C. Pedersen

Afleveret d. 5. maj 2003
Behandlet d. 22. maj 2003

Indhold

Kapitel 1

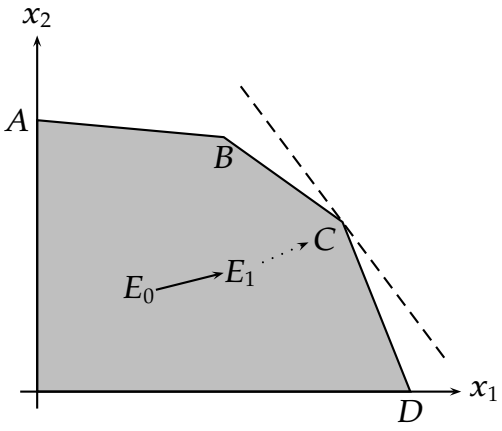
Indledning

Der findes mange forskellige typer af økonomiske problemstillinger og tilhørende løsningsmetoder. En af dem er lineære programmeringsproblemer, hvor vi har valgt at se nærmere på to af de tilhørende løsningsmetoder.

Den første metode er simplex-algoritmen udviklet af G. B. Dantzig i 1947, som er standardmetoden til løsning af LP-problemer.

I 1984 beviste Karmarkar, at man også kunne bruge iterationsmetoder – de såkaldte indre punkts-algoritmer – til løsning af LP-problemer. En sådan er *primal-dual feasible-interior-point*-metoden (herefter »Interior Point«), som vi vil stifte nærmere bekendtskab med, da den har fået ry for at være én de mest effektive af denne række af nye algoritmer.

Baggrunden for valget af disse to løsningsmodeller er foretaget med tanke på beregningsgangene, der adskiller sig markant fra hinanden. Simplex-metoden bevæger sig systematisk gennem de brugbare basisløsninger, der udgør hjørnerne af det n-dimesionale polyeder, linjerne danner. Under denne gennemgang øges kriterieværdien langsomt indtil den optimale løsning opnås. Derimod starter indre punkts-algoritmerne i et vilkårligt punkt i det mulige løsningsområde, og med udgangspunkt i Newtons metode til løsning af ikkelineære ligninger konvergerer den mod den optimale løsning. På figur 1.1 ses basisløsninger A, B, C og D som simplex undersøger én efter en, for blot at stoppe allerede ved C, som er den optimale løsning. Interior Point-metoderne starter i en mulig løsning, fx E_0 og beregner et nyt punkt E_1, der ligger nærmere den optimale løsning. Sådan bliver det ved, indtil en optimal løsning er fundet.



Figur 1.1: Forskellen mellem simplex og Interior Point

Da beregningsgangene er forskellige, åbner det muligheden for at undersøge, om man i visse situationer med fordel kan anvende den ene metode fremfor den anden, eventuelt fordi den ene af metoderne simpelthen ikke kan løse problemet, eller hvis der er åbenlys forskel på tidsforbruget ved løsning af et problem.

Men hvordan vil vi så gribe denne problemstilling an?

Efter en implementering af de to løsningsmetoder i C++, foretages en række testkørsler, hvor følgende ønskes belyst:

- Er beregningsgangen rigtig?
- Har henholdsvis simplex og Interior Point nogle svagheder – specialtilfælde?
- Hvordan udvikler antallet af iterationer sig for de to metoder, når problemets størrelse varieres?
- Hvilken indflydelse har problemets størrelse på tidsforbruget?
- Hvor god er præcisionen, når problemets størrelse øges?
- Hvilken betydning har det for resultaterne, når forskellige parametre såsom tolerancer ændres?

Ud fra resultaterne af testkørslerne skulle det så være muligt at sammenligne de to metoder og give et bud på, hvornår man med fordel kan anvende den ene metode fremfor den anden.

Vi har været nødt til at afgrænse implementeringen i C++ undervejs, for at det ikke skulle blive for omfattende.

Simplex-metoden er afgrænset på en sådan måde, at henholdsvis simplex og dual simplex er tilgængelig i beregningsgangen. Derimod er metoder som revideret simplex, fractional cut og branch-and-bound ikke implementeret, da de ikke er nødvendige for at opnå brugbare resultater med simplex.

Derudover vil der for begge løsningsmodeller højst sandsynlig være specialtilfælde, hvor der ikke kan findes en løsning. I disse tilfælde har vi valgt at angive et løsningsforslag istedet for at implementere det, selvom det sagtens kunne lade sig gøre – men igen vil det blive alt for omfattende.

Kapitel 2

Teoretisk baggrund

Herefter følger det matematiske grundlag får henholdsvis simplex- og Interior Point-metoden. Gennemgangen af simplex-metoden vil være kort, da den anses for at være velkendt fra undervisningen. Derimod vil Interior Point-metoden blive gennemgået mere omfattende, da metoden formodentlig er ukendt for de fleste.

2.1 Simplex-metoden

Simplex-metoden anvendes til at løse lineære programmeringsproblemer af typen:

$$\begin{aligned}
&\max \quad c_1x_1 + \cdots + c_nx_n \\
&\text{under bibetingelserne} \\
&a_{11}x_1 + \cdots + a_{1n}x_n \leq b_1 \\
&\quad \vdots \qquad\qquad \vdots \qquad \vdots \\
&a_{m1}x_1 + \cdots + a_{mn}x_n \leq b_m \\
&x_1, x_2, \ldots, x_n \geq 0
\end{aligned}$$

Problemet omformes til noget brugbart for simplex-metoden ved at omskrive problemet til kanonisk form. Dette gøres ved at omskrive uligheder til ligninger ved at tilføje slack-variable. Dermed bliver problemet følgende:

$$\begin{aligned}
&\max \quad c_1x_1 + \cdots + c_nx_n \\
&\text{under bibetingelserne} \\
&a_{11}x_1 + \cdots + a_{1n}x_n + x_{n+1} = b_1 \\
&\quad \vdots \qquad\qquad \vdots \qquad\qquad \vdots \\
&a_{m1}x_1 + \cdots + a_{mn}x_n + x_{n+m} = b_m \\
&x_1, x_2, \ldots, x_{n+m} \geq 0
\end{aligned}$$

Ud fra ovenstående formulering kan det tilhørende simplex-tableau så opstilles:

	c_1	$\cdots$	c_n	0	$\cdots$	0
b_1	a_{11}	$\cdots$	$a_{1,n-m}$	1	$\cdots$	0
$\vdots$	$\vdots$		$\vdots$	$\vdots$		$\vdots$
b_m	a_{m1}	$\cdots$	$a_{m,n-m}$	0	$\cdots$	1

Med udgangspunkt i simplex-tableauet kan der nu findes basisløsninger til problemet. Proceduren er følgende (Keiding, 2002):

1. Følgende skal være opfyldt for simplex-tableauet:
   a) Alle elementer i b-søjlen skal være positive.
   b) Slack-variable skal tilsammen give en basis bestående af enhedsvektorer, og det tilhørende element til basissøjlen i c-rækken skal være 0.
2. Dernæst skal en søjle udvælges. Dette sker ud fra følgende:
   a) Der skal være positive koefficienter i c-rækken.
   b) Den søjle med den største positive koefficient i c-rækken udvælges.
   c) Er der flere søjler med samme koefficient i c-rækken vælges den længst til venstre.
3. Herefter skal et pivotelement i søjlen udvælges:
   a) Kun de koefficienter i søjlen, som er positive, er relevante.
   b) Forholdet b_i/a_{ij} beregnes, og den koefficient med det mindste forhold udvælges til pivotelement.
4. Pivotsteppet udføres:
   a) Ved rækkeoperation ændres pivotelement til 1.
   b) Der skaffes 0 over og under pivotelement ved rækkeoperationer.

En optimal løsning er nået, når:

- Alle koefficienter i c-rækken er negative eller 0.
- Alle elementer i b-søjlen er positive eller 0.

2.1.1 Dual simplex

I tilfælde, hvor der optræder negative værdier i b-søjlen, må man anvende dual simplex. Dual simplex minder meget om simplex – beregningsgangen tager bare udgangspunkt i b-søjlen istedet for c-rækken:

1. Af de negative elementer i b-søjlen vælges den række med den mindste værdi.
2. Forholdet c_i/a_{ij} i den aktuelle række beregnes, hvis både c_i og a_{ij} er negative.
3. Pivotelementet bliver den koefficient, hvor brøken c_i/a_{ij} er mindst.
4. Herefter udføres pivotsteppet på samme måde som ved simplex.

Hvis der ønskes yderligere informationer om henholdsvis simplex og dual simplex henvises til (Keiding, 2002, kapitel 2 og 3).

2.2 Interior-point metoden

Interior Point løser generelt lineære programmeringsproblemer af typen

$$\begin{aligned}
&\min \quad c_1x_1 + \cdots + c_nx_n \\
&\text{under bibetingelserne} \\
&a_{11}x_1 + \cdots + a_{1n}x_n \leq b_1 \\
&\quad \vdots \qquad\qquad \vdots \qquad \vdots \\
&a_{m1}x_1 + \cdots + a_{mn}x_n \leq b_m \\
&x_1, x_2, \ldots, x_n \geq 0
\end{aligned}$$

Bemærk, at i modsætning til simplex-metoden løser denne metode minimeringsproblemer.

Baggrunden for den valgte indre-punkts metode er Newtons metode – måske bedre kendt som Lagrange-metoden – som anvendes til at løse ikke-lineære optimeringsproblemer (Sydsæter, 2000, kapitel 14).

Den generelle formulering af et ikke-lineært optimeringsproblem er:

$$\begin{aligned}
&\max/\min \quad f(x_1, \ldots, x_n) \\
&\text{under bibetingelserne} \\
&g_1(x_1, \ldots, x_n) = 0 \\
&\quad \vdots \qquad \vdots \qquad \vdots \\
&g_m(x_1, \ldots, x_n) = 0
\end{aligned}$$

2.2.1 Det primale problem

Pga. forskellen i bibetingelserne (ligninger/uligheder) kan metoden ikke anvendes direkte på det aktuelle problem. Man er derfor nødt til at omformulere problemet, så der udelukkende gælder lighedstegn i bibetingelserne, og dette kan klares på følgende måde:

$$\begin{aligned}
&\min \quad c_1x_1 + \cdots + c_nx_n - \mu \sum_{j=1}^{n} \ln(x_{ij}) \\
&\text{under bibetingelserne} \\
&a_{11}x_1 + \cdots + a_{1n}x_n = b_1 \\
&\quad \vdots \qquad\qquad \vdots \qquad \vdots \\
&a_{m1}x_1 + \cdots + a_{mn}x_n = b_m \\
&x_1, x_2, \ldots, x_n \geq 0
\end{aligned}$$

Man vælger at trække en logaritmisk funktion fra, da det udelukker alle ikkepositive løsninger. Dermed kan der ses bort fra bibetingelserne $x_1, x_2, \ldots, x_n \geq 0$, da de automatisk vil være opfyldt.

En optimal løsning til ovenstående problem vil ikke nødvendigvis være en optimal løsning til det oprindelige problem. Det viser sig dog, at hvis μ vælges tilstrækkelig lille, vil det være en god approximation til en løsning til det oprindelige problem.

Herefter defineres den tilhørende Lagrange-funktion:

$$\mathcal{L}(\boldsymbol{x}, \boldsymbol{y}) = \boldsymbol{c}^T\boldsymbol{x} - \mu \sum_{j=1}^{n} \ln(x_j) - \boldsymbol{y}^T(A\boldsymbol{x} - \boldsymbol{b})$$

hvor $\boldsymbol{y}$ svarer til Lagrange-multiplikatorerne. Efterfølgende kan 1. ordens betingelserne findes:

$$\begin{aligned}
\nabla_x \mathcal{L}(\boldsymbol{x}, \boldsymbol{y}) &= \boldsymbol{c} - \mu X^{-1}\boldsymbol{e} - A^T\boldsymbol{y} = 0 \\
\nabla_y \mathcal{L}(\boldsymbol{x}, \boldsymbol{y}) &= \boldsymbol{b} - A\boldsymbol{x} = 0 \\
\boldsymbol{x} &> 0
\end{aligned}$$

Inden udledningen fortsættes, skal der gøres opmærksom på følgende notation:

$$X := \operatorname{diag}(\boldsymbol{x}) = \begin{pmatrix} x_1 & 0 & \cdots & 0 \\ 0 & x_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & x_n \end{pmatrix}$$

Når der er tale om en diagonalmatrix, kan den inverse matrix findes meget let, hvis alle diagonalelementerne er forskellige fra 0:

$$X^{-1} := \begin{pmatrix} x_1^{-1} & 0 & \cdots & 0 \\ 0 & x_2^{-1} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & x_n^{-1} \end{pmatrix}$$

Herefter fortsættes der, og vektoren $\boldsymbol{s} = \mu X^{-1}\boldsymbol{e}$ indføres. Hvis udtrykket for $\boldsymbol{s}$ omskrives, fås $X\boldsymbol{s} = \mu\boldsymbol{e}$. Dermed kan 1. ordens betingelserne nu skrives på følgende måde:

$$\begin{aligned} A^T\boldsymbol{y} + \boldsymbol{s} &= \boldsymbol{c} \\ A\boldsymbol{x} &= \boldsymbol{b} \\ X\boldsymbol{s} &= \mu\boldsymbol{e} \\ \boldsymbol{x} &> 0 \\ \boldsymbol{s} &> 0 \end{aligned}$$

Dette resultat vil vi benytte senere hen.

2.2.2 Det duale problem

Det viser sig, at man kan udføre de samme beregninger for det duale problem:

$$\max \quad b_1y_1 + \cdots + b_my_m + \mu\sum_{j=1}^{n}\ln(s_j)$$

under bibetingelserne

$$\begin{aligned} &a_{11}y_1 + \cdots + a_{m1}y_1 + s_1 = c_1 \\ &\quad\vdots \qquad\qquad \vdots \qquad\qquad \vdots \\ &a_{1n}y_n + \cdots + a_{mn}y_n + s_n = c_n \end{aligned}$$

Hvis man gør dette, vil man opdage, at man får et sæt 1. ordens betingelser, som efter omordning ligner de 1. ordens betingelser, som beregningerne resulterede i ved det primale problem.

2.2.3 Udledning af primal-dual algoritmen

I de to foregående afsnit er det lykkedes at omskrive henholdsvis det primale og det duale problem til ikke-lineære optimeringsproblemer, som nu kan løses ved hjælp af Newtons metode.

Vi indfører størrelserne $\gamma > 0$ og $\mu := x^T s/n$, hvor n er antallet af bibetingelser, og derpå definerer vi funktionen F_γ:

$$F_\gamma(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{s}) := \begin{pmatrix} A\boldsymbol{x} - \boldsymbol{b} \\ A^T\boldsymbol{y} + \boldsymbol{s} - \boldsymbol{c} \\ X\boldsymbol{s} - \gamma\mu\boldsymbol{e} \end{pmatrix}$$

Funktionen F_γ sættes lig 0

$$F_\gamma(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{s}) = 0$$

hvilket ikke umiddelbart er til at løse. Imidlertid ved vi, at F_γ kan beskrives ved Taylor-polynomiet

$$f(\boldsymbol{x} + \boldsymbol{d}_x) = f(\boldsymbol{x}) + \nabla f(\boldsymbol{x})\boldsymbol{d}_x$$

for $\boldsymbol{d}_x$ tilstrækkelig tæt på 0. Denne ligning vil vi følgelig også gerne finde rødder for:

$$f(\boldsymbol{x}) + \nabla f(\boldsymbol{x})\boldsymbol{d}_x = 0$$

Efter indsættelse af F_γ får vi så udtrykket

$$\nabla F_\gamma(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{s}) \begin{pmatrix} \boldsymbol{d}_x \\ \boldsymbol{d}_y \\ \boldsymbol{d}_s \end{pmatrix} = -F_\gamma(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{s})$$

hvilket giver de to identiske udtryk for F_γ.

Vi beregner nu gradienten af F_γ.

$$\nabla F_\gamma(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{s}) = \begin{pmatrix} A & 0 & 0 \\ 0 & A^T & I \\ S & 0 & X \end{pmatrix}$$

Ud fra ligningssystemet:

$$\nabla F_\gamma(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{s}) \begin{pmatrix} \boldsymbol{d}_x \\ \boldsymbol{d}_y \\ \boldsymbol{d}_s \end{pmatrix} = -F_\gamma(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{s})$$

fås følgende:

$$\begin{pmatrix} A & 0 & 0 \\ 0 & A^T & I \\ S & 0 & X \end{pmatrix} \begin{pmatrix} \boldsymbol{d}_x \\ \boldsymbol{d}_y \\ \boldsymbol{d}_s \end{pmatrix} = \begin{pmatrix} \boldsymbol{r}_P \\ \boldsymbol{r}_D \\ \gamma\mu\boldsymbol{e} - X\boldsymbol{s} \end{pmatrix}$$

hvor $\boldsymbol{r}_P$ og $\boldsymbol{r}_D$ er henholdsvis residualer til det primale og det duale problem. De er defineret som

$$\boldsymbol{r}_P := \boldsymbol{b} - A\boldsymbol{x}$$
$$\boldsymbol{r}_D := \boldsymbol{c} - A^T\boldsymbol{y} - \boldsymbol{s}$$

Hermed er vi faktisk nået til det første trin i algoritmen, som går ud på at løse dette ligningssystem. Når det er gjort, opdateres variablene, og algoritmen gentages.

$$\begin{pmatrix} \boldsymbol{x}^+ \\ \boldsymbol{y}^+ \\ \boldsymbol{s}^+ \end{pmatrix} = \begin{pmatrix} \boldsymbol{x} \\ \boldsymbol{y} \\ \boldsymbol{s} \end{pmatrix} + \alpha \begin{pmatrix} \boldsymbol{d}_x \\ \boldsymbol{d}_y \\ \boldsymbol{d}_s \end{pmatrix}$$

Betydningen af faktoren α er forklaret nærmere i afsnit 2.2.5 på næste side.

2.2.4 Dualitetsspænd

Vi ved ud fra dualitetssætningens hovedtilfælde (Fuglede et al., 1999), at hvis henholdsvis det primale og det duale problem har en optimal løsning, så har de samme optimale værdi: $\sup(P) = \inf(D) \in \mathbb{R}$.

Når man anvender den omtalte indre punkts metode, får man som sagt en løsning, som er en approximation til den optimale løsning pga. omskrivningen af det oprindelige problem. Dermed kan funktionsværdien for henholdsvis det primale og det duale problem afvige fra hinanden – det såkaldte dualitetsspænd.

Hvis man beregner det dualitetsspænd, man kan forvente, når man anvender denne metode, får man følgende, hvor $(\boldsymbol{x}(\mu), \boldsymbol{y}(\mu), \boldsymbol{s}(\mu))$ er en løsning:

$$\begin{aligned} \boldsymbol{c}^T\boldsymbol{x}(\mu) - \boldsymbol{b}^T\boldsymbol{y}(\mu) &= \boldsymbol{x}(\mu)^T\boldsymbol{s}(\mu) \\ &= \boldsymbol{e}^T X(\mu)\boldsymbol{s}(\mu) \\ &= \boldsymbol{e}^T(\mu\boldsymbol{e}) \\ &= \mu n \end{aligned}$$

n er antallet af variable – inklusive slack-variable.

For at opnå det bedst mulige resultat er man selvfølgelig interesseret i, at dualitetsspændet bliver lig 0. Det er her, at vi indfører det ene af stopkriterierne i beregningsgangen. Det er nemlig defineret som $\boldsymbol{x}^T\boldsymbol{s}$ – dualitetsspændet. Så ved at vælge et stopkriterium tæt på 0, opnår man, at dualitetsspændet bliver mindst muligt, og derved får man løsninger til henholdsvis det primale og det duale problem, som afviger meget lidt fra den optimale løsning til det oprindelige problem.

2.2.5 Algoritmen

For at gøre metoden endnu mere tilgængelig, vil algoritmen nu blive gennemgået trin for trin (Andersen, 1998, p. 54–55).

1. $\boldsymbol{x}^0$, $\boldsymbol{y}^0$ og $\boldsymbol{s}^0$ vælges, så $\boldsymbol{x}^0$ er en mulig løsning og $\boldsymbol{x}^0, \boldsymbol{s}^0 > 0$. Antallet af elementer i $\boldsymbol{x}^0$ og $\boldsymbol{s}^0$ er lig antallet af variable (inkl. slackvariable), og antallet af elementer i $\boldsymbol{y}^0$ er lig antallet af bibetingelser. Desuden vælges tolerancerne $\varepsilon_G, \varepsilon_P, \varepsilon_D > 0$.

2. $k := 0$

3. Residualerne $\boldsymbol{r}_P^k$ og $\boldsymbol{r}_D^k$ og konstanten μ^k beregnes.

$$\begin{aligned}\boldsymbol{r}_P^k &= \boldsymbol{b} - A\boldsymbol{x}^k \\ \boldsymbol{r}_D^k &= \boldsymbol{c} - A^T\boldsymbol{y}^k - \boldsymbol{s}^k \\ \mu^k &= (\boldsymbol{x}^k)^T\boldsymbol{s}^k/n\end{aligned}$$

4. Hvis stopkriterierne

$$(\boldsymbol{x}^k)^T\boldsymbol{s}^k \leq \varepsilon_G \quad \|\boldsymbol{r}_P^k\| \leq \varepsilon_P \quad \|\boldsymbol{r}_D^k\| \leq \varepsilon_D$$

   er opfyldt, stoppes algoritmen. Ellers fortsættes.

5. Herefter vælges $\gamma \in [0;1[$.

6. Derefter løses ligningssystemet

$$\begin{pmatrix} A & 0 & 0 \\ 0 & A^T & I \\ S^k & 0 & X^k \end{pmatrix} \begin{pmatrix} \boldsymbol{d}_x \\ \boldsymbol{d}_y \\ \boldsymbol{d}_s \end{pmatrix} = \begin{pmatrix} \boldsymbol{r}_P^k \\ \boldsymbol{r}_D^k \\ \gamma\mu^k\boldsymbol{e} - X^k\boldsymbol{s}^k \end{pmatrix}$$

   hvor henholdsvis $\boldsymbol{d}_y$, $\boldsymbol{d}_s$ og $\boldsymbol{d}_x$ bestemmes ud fra følgende formler

$$\begin{aligned} AX^k(S^k)^{-1}A^T\boldsymbol{d}_y &= \boldsymbol{b} + A(S^k)^{-1}(X^k\boldsymbol{r}_D^k - \gamma\mu^k\boldsymbol{e}) \\ \boldsymbol{d}_s &= \boldsymbol{r}_D^k - A^T\boldsymbol{d}_y \\ \boldsymbol{d}_x &= -\boldsymbol{x}^k + (S^k)^{-1}(\gamma\mu^k\boldsymbol{e} - X^k\boldsymbol{d}_s)\end{aligned}$$

7. Herefter bestemmes α^{max} ud fra følgende fremgangsmåde:

$$\begin{aligned}\alpha_P^{max} &= \min(-\boldsymbol{x}_j^k/(\boldsymbol{d}_x)_j \mid (\boldsymbol{d}_x)_j < 0) \\ \alpha_D^{max} &= \min(-\boldsymbol{s}_j^k/(\boldsymbol{d}_s)_j \mid (\boldsymbol{d}_s)_j < 0)\end{aligned}$$

Ud fra disse to værdier kan α^{max} bestemmes

$$\alpha^{max} = \min(\alpha_P^{max}, \alpha_D^{max})$$

Faktoren α har betydning for størrelsen af det skridt, vi tager hen mod løsningen. Hvis $\alpha > 1$ kan vi ikke være sikre på konvergens (Andersen, 1998, p. 53). Vi bestemmer da α som

$$\alpha := \min(\theta\alpha^{max}, 1), \quad \text{hvor} \quad \theta = 1 - \gamma$$

8. Til sidst opdateres $\boldsymbol{x}$, $\boldsymbol{y}$ og $\boldsymbol{s}$ på følgende måde:

$$\begin{aligned}\boldsymbol{x}^{k+1} &:= \boldsymbol{x}^k + \alpha\boldsymbol{d}_x \\ \boldsymbol{y}^{k+1} &:= \boldsymbol{y}^k + \alpha\boldsymbol{d}_y \\ \boldsymbol{s}^{k+1} &:= \boldsymbol{s}^k + \alpha\boldsymbol{d}_s\end{aligned}$$

9. k opdateres, $k = k + 1$
10. Algoritmen gentages med de opdaterede variable.

Efterfølgende vil et enkelt gennemløb af algoritmen blive udført ud fra et eksempel.

Eksempel 2.1 Vi betragter LP-problemet

$$\begin{aligned}&\min \quad 2x_1 - 3x_2 \\ &\text{under bibetingelserne} \\ &x_1 + x_2 \leq 5 \\ &-3x_1 - x_2 \leq -3 \\ &x_1, x_2 \geq 0\end{aligned}$$

som efter omdannelse til kanonisk form, giver os følgende to vektorer og en matrix:

$$\boldsymbol{c} = \begin{pmatrix} 2 \\ -3 \\ 0 \\ 0 \end{pmatrix} \quad A = \begin{pmatrix} 1 & 1 & 1 & 0 \\ -3 & -1 & 0 & 1 \end{pmatrix} \quad \boldsymbol{b} = \begin{pmatrix} 5 \\ -3 \end{pmatrix}$$

Først vælges $\boldsymbol{x}$, $\boldsymbol{y}$ og $\boldsymbol{s}$.

$$\boldsymbol{x} = \begin{pmatrix} 2 \\ 1 \\ 1 \\ 2 \end{pmatrix} \quad \boldsymbol{y} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \quad \boldsymbol{s} = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix}$$

Tolerancerne fastsættes.

$$\varepsilon_G = 1 \times 10^{-8} \quad \varepsilon_P = 1 \times 10^{-8} \quad \varepsilon_D = 1 \times 10^{-8}$$

Herefter bestemmes residualerne $\boldsymbol{r}_P$ og $\boldsymbol{r}_D$ og μ bestemmes.

$$\begin{aligned}
\boldsymbol{r}_P &= \boldsymbol{b} - A\boldsymbol{x} \\
&= \begin{pmatrix} 5 \\ -3 \end{pmatrix} - \begin{pmatrix} 1 & 1 & 1 & 0 \\ -3 & -1 & 0 & 1 \end{pmatrix} \begin{pmatrix} 2 \\ 1 \\ 1 \\ 2 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \end{pmatrix} \\
\boldsymbol{r}_D &= \boldsymbol{c} - A^T\boldsymbol{y} - \boldsymbol{s} \\
&= \begin{pmatrix} 2 \\ -3 \\ 0 \\ 0 \end{pmatrix} - \begin{pmatrix} 1 & -3 \\ 1 & -1 \\ 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \end{pmatrix} - \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix} = \begin{pmatrix} 1 \\ -5 \\ -3 \\ -4 \end{pmatrix} \\
\mu &= \boldsymbol{x}^T\boldsymbol{s}/n = \begin{pmatrix} 2 & 1 & 1 & 2 \end{pmatrix} \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix} /4 = 3.75
\end{aligned}$$

Det er oplagt, at ingen af stopkriterierne er opfyldt, så vi går videre. Vi sætter $\gamma = 0.1$, og beregner $\boldsymbol{d}_y$ ud fra formlen:

$$AXS^{-1}A^T\boldsymbol{d}_y = \boldsymbol{b} + AS^{-1}(X\boldsymbol{r}_D - \gamma\mu\boldsymbol{e})$$

Først beregnes venstre side:

$$\begin{aligned}
L &= AXS^{-1}A^T \\
&= \begin{pmatrix} 1 & 1 & 1 & 0 \\ -3 & -1 & 0 & 1 \end{pmatrix} \begin{pmatrix} 2 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 2 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{1}{2} & 0 & 0 \\ 0 & 0 & \frac{1}{3} & 0 \\ 0 & 0 & 0 & \frac{1}{4} \end{pmatrix} \begin{pmatrix} 1 & -3 \\ 1 & -1 \\ 1 & 0 \\ 0 & 1 \end{pmatrix} \\
&= \begin{pmatrix} 2 & 1 & 1 & 0 \\ -6 & -1 & 0 & 2 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{1}{2} & 0 & 0 \\ 0 & 0 & \frac{1}{3} & 0 \\ 0 & 0 & 0 & \frac{1}{4} \end{pmatrix} \begin{pmatrix} 1 & -3 \\ 1 & -1 \\ 1 & 0 \\ 0 & 1 \end{pmatrix} \\
&= \begin{pmatrix} 2 & \frac{1}{2} & \frac{1}{3} & 0 \\ -6 & -\frac{1}{2} & 0 & \frac{1}{2} \end{pmatrix} \begin{pmatrix} 1 & -3 \\ 1 & -1 \\ 1 & 0 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 2.8333 & -6.5 \\ -6.5 & 19 \end{pmatrix}
\end{aligned}$$

Derefter beregnes højre side $\boldsymbol{R}$:

$$\boldsymbol{R} = \boldsymbol{b} + AS^{-1}(X\boldsymbol{r}_D - \gamma\mu\boldsymbol{e})$$

$$AS^{-1} = \begin{pmatrix} 1 & 1 & 1 & 0 \\ -3 & -1 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{1}{2} & 0 & 0 \\ 0 & 0 & \frac{1}{3} & 0 \\ 0 & 0 & 0 & \frac{1}{4} \end{pmatrix} = \begin{pmatrix} 1 & \frac{1}{2} & \frac{1}{3} & 0 \\ -3 & -\frac{1}{2} & 0 & \frac{1}{4} \end{pmatrix}$$

$$X\boldsymbol{r}_D - \gamma\mu\boldsymbol{e} = \begin{pmatrix} 2 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 2 \end{pmatrix} \begin{pmatrix} 1 \\ -5 \\ -3 \\ -4 \end{pmatrix} - 0.1 \times 3.75 \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$

$$= \begin{pmatrix} 2 \\ -5 \\ -3 \\ -8 \end{pmatrix} - \begin{pmatrix} 0.375 \\ 0.375 \\ 0.375 \\ 0.375 \end{pmatrix} = \begin{pmatrix} 1.625 \\ -5.375 \\ -3.375 \\ -8.375 \end{pmatrix}$$

$$\boldsymbol{R} = \begin{pmatrix} 5 \\ -3 \end{pmatrix} + \begin{pmatrix} 1 & \frac{1}{2} & \frac{1}{3} & 0 \\ -3 & -\frac{1}{2} & 0 & \frac{1}{4} \end{pmatrix} \begin{pmatrix} 1.625 \\ -5.375 \\ -3.375 \\ -8.375 \end{pmatrix}$$

$$= \begin{pmatrix} 5 \\ -3 \end{pmatrix} + \begin{pmatrix} -2.1875 \\ -4.28125 \end{pmatrix} = \begin{pmatrix} 2.8125 \\ -7.28123 \end{pmatrix}$$

Det gør os nu i stand til at beregne differentialerne i Newton-metoden. Først bestemmes $\boldsymbol{d}_y$:

$$\boldsymbol{d}_y = L^{-1}\boldsymbol{R} = \begin{pmatrix} 1.6403 & 0.5612 \\ 0.5612 & 0.2446 \end{pmatrix} \begin{pmatrix} 2.8125 \\ -7.2813 \end{pmatrix} = \begin{pmatrix} 0.5274 \\ -0.2028 \end{pmatrix}$$

Efter vi har bestemt $\boldsymbol{d}_y$ kan vi beregne de to andre differentialer. Først $\boldsymbol{d}_s$, der skal bruges til bestemmelse af $\boldsymbol{d}_x$:

$$\boldsymbol{d}_s = \boldsymbol{r}_D - A^T\boldsymbol{d}_y$$

$$= \begin{pmatrix} 1 \\ -5 \\ -3 \\ -4 \end{pmatrix} - \begin{pmatrix} 1 & -3 \\ 1 & -1 \\ 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 0.5274 \\ -0.2028 \end{pmatrix} = \begin{pmatrix} -0.1358 \\ -5.7302 \\ -3.5274 \\ -3.7972 \end{pmatrix}$$

Og endelig det sidste differentiale $\boldsymbol{d}_x$:

$$\boldsymbol{d}_x = -\boldsymbol{x} + S^{-1}(\gamma\mu\boldsymbol{e} - X\boldsymbol{d}_s)$$

$$\gamma\mu\boldsymbol{e} - X\boldsymbol{d}_s = 0.1 \times 3.75 \begin{pmatrix}1\\1\\1\\1\end{pmatrix} - \begin{pmatrix}2&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&2\end{pmatrix}\begin{pmatrix}-0.1358\\-5.7302\\-3.5274\\-3.7972\end{pmatrix}$$

$$= \begin{pmatrix}0.375\\0.375\\0.375\\0.375\end{pmatrix} - \begin{pmatrix}-0.2716\\-5.7302\\-3.5274\\-7.5944\end{pmatrix} = \begin{pmatrix}0.6466\\6.1052\\3.9024\\-7.2194\end{pmatrix}$$

$$\boldsymbol{d}_x = \begin{pmatrix}-2\\-1\\-1\\-2\end{pmatrix} + \begin{pmatrix}1&0&0&0\\0&\frac{1}{2}&0&0\\0&0&\frac{1}{3}&0\\0&0&0&\frac{1}{4}\end{pmatrix}\begin{pmatrix}0.6466\\6.1052\\3.9024\\-7.2194\end{pmatrix}$$

$$= \begin{pmatrix}-2\\-1\\-1\\-2\end{pmatrix} + \begin{pmatrix}0.6466\\3.0526\\1.3008\\1.9924\end{pmatrix} = \begin{pmatrix}-1.3534\\2.0526\\0.3008\\-0.0076\end{pmatrix}$$

Vi kan nu bestemme henholdsvis α_P^{max} og α_D^{max}.

$$\alpha_P^{max} = \min\left(\frac{2}{1.3534}, \text{ϟ}, \text{ϟ}, \frac{2}{0.0076}\right) = \frac{2}{1.3534} = 1.4777$$

$$\alpha_D^{max} = \min\left(\frac{1}{0.1358}, \frac{2}{5.7302}, \frac{3}{3.5274}, \frac{4}{3.7972}\right) = \frac{2}{5.7302} = 0.3490$$

Dermed bliver α^{max}:

$$\alpha^{max} = \min(\alpha_P^{max}, \alpha_D^{max}) = \alpha_D^{max} = 0.3490$$

Herefter bestemmes α:

$$\alpha = \min(\theta\alpha^{max}, 1) = \min\big((1 - 0.1) \times 0.3490, 1\big) = 0.3141$$

Til sidst opdateres variablene:

$$\boldsymbol{x}^+ = \boldsymbol{x} + \alpha\boldsymbol{d}_x = \begin{pmatrix}2\\1\\1\\2\end{pmatrix} + \alpha\begin{pmatrix}-1.3534\\2.0526\\0.3008\\-0.0076\end{pmatrix} = \begin{pmatrix}1.5749\\1.6448\\1.0945\\1.9976\end{pmatrix}$$

$$\boldsymbol{s}^+ = \boldsymbol{s} + \alpha\boldsymbol{d}_s = \begin{pmatrix}1\\2\\3\\4\end{pmatrix} + \alpha\begin{pmatrix}-0.1358\\-5.7302\\-3.5274\\-3.7972\end{pmatrix} = \begin{pmatrix}0.9573\\0.2\\1.8919\\2.8072\end{pmatrix}$$

$$\boldsymbol{y}^+ = \boldsymbol{y} + \alpha\boldsymbol{d}_y = \begin{pmatrix}0\\0\end{pmatrix} + \alpha\begin{pmatrix}0.5274\\-0.2028\end{pmatrix} = \begin{pmatrix}0.1657\\-0.0637\end{pmatrix}$$

Herefter gentages algoritmen på tilsvarende måde, indtil stopkriterierne er opfyldt.

Kapitel 3

Testkørsler & analyse

På baggrund af den teoretiske gennemgang var det muligt at gennemføre implementeringen i C++, som vi dog ikke vil gennemgå her. Læseren henvises til bilag A på side 35, hvor koden er gennemgået minutiøst.

Allerede her bør det nævnes, at den måde, vi konstruerer programmet på, har visse mangler. Vore arrays er nemlig statiske, hvilke C++ har en øvre størrelsesgrænse på. Der meldes fejl, når et array indeholder mere end 10 000 pladser, hvilket i vores implementering betyder, at der maksimalt kan være 140 variable, hvoraf halvdelen er slack-variable. Årsagen er de arrays, der indeholder $n(2 \times n + 1)$ pladser såsom simplextableauet. For $n = 70$ giver det 9 870 pladser, mens det for $n = 71$ giver 10 153 pladser. Dette er også grunden til at vi må forfalde til ekstrapolationer af måledata.

3.1 Interne test

Ved de interne test af programmet har vi benyttet eksempler med kendte løsninger fra Keiding (2002) og Andersen (1998).

Eksempel 3.1 Betragt LP-problemet (Keiding, 2002, p. 36–37)

$$\begin{aligned} &\min \quad 2x_1 + x_2 \\ &\text{under bibetingelserne} \\ &3x_1 + x_2 \geq 3 \\ &4x_1 + 3x_2 \geq 6 \\ &x_1 + 2x_2 \leq 3 \end{aligned} \tag{3.1}$$

Dette problem har optimal løsning

$$\boldsymbol{x}^* = \begin{pmatrix} 3/5 \\ 6/5 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

Simplexalgoritmen finder denne løsning efter to iterationer, mens Interior Point-algoritmen med $\gamma = 0.05$, $\boldsymbol{x}^0 = (½, ½, 1, 1, 1)^T$ efter otte iterationer finder løsningen

$$\boldsymbol{x} = \begin{pmatrix} 0.60000168 \\ 1.19999996 \\ 2.416 \times 10^{-6} \\ 8.561 \times 10^{-7} \\ 1.560 \times 10^{-6} \end{pmatrix}$$

Tilsvarende problemer løses også korrekt af programmet, hvorfor vi koncentrerede os om eksterne test istedet.

3.2 Eksterne test

Til testkørslerne er det vigtigt, at vi kan få fat i nærmest vilkårligt store problemer, hvis løsning vi vel at mærke kender. Derfor vil det være praktisk, hvis vi kan konstruere et problem med disse egenskaber.

Vort formål med dette konstruerede problem er selvfølgelig, at vi vil sikre os, at vi både kan benytte simplex såvel som Interior Point på det. Derfor må vi sørge for, at der findes både en mulig såvel som en optimal løsning, da Interior Point skal have en mulig løsning for at komme i gang. Følgende problem med n ligninger og n bibetingelser med hver n led opfylder vore krav.

$$\begin{aligned} &\max \quad (n(n+1)+1)x_1 + (n(n+1)+4)x_2 + \cdots + (n(n+4)-2)x_n \\ &\text{under bibetingelserne} \\ &\quad x_1 + 3x_2 + \cdots + (2n-1)x_n \le n(n+1) \\ &2x_1 + 4x_2 + \cdots + 2nx_n \le n(n+2) \\ &3x_1 + 5x_2 + \cdots + (2n+1)x_n \le n(n+3) \\ &\quad \vdots \qquad\qquad \vdots \qquad\qquad \vdots \\ &nx_1 + (n+2)x_2 + \cdots + (3n-2)x_n \le 2n^2 \end{aligned} \tag{3.2}$$

Efter påførsel af slack-variable $x_{n+1}, \ldots, x_{2n}$ kan vi prøve at se, om vi kan gætte en mulig løsning. Som start kan vi sætte $x_1, \ldots, x_n$ lig $\boldsymbol{e}$, og vi ved, at summen σ_i af koefficienterne til de første n variable i den i'te

bibetingelse er lig skalarproduktet $\boldsymbol{e}(a_{i1}, \ldots, a_{in})^T$. Vi får dermed

$$\sigma_i = \sum_{k=0}^{n-1}(i + 2k) = ni + 2\frac{n(n-1)}{2} = ni + n^2 - n,$$

og $b_i = n^2 + ni$, hvilket samlet giver os

$$x_{n+i} = b_i - \sigma_i = n^2 + ni - (ni + n^2 - n) = n$$

Altså kan vi med sikkerhed konstatere, at der altid findes en mulig løsning $\boldsymbol{x}$ i vektoren

$$\boldsymbol{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \\ x_{n+1} \\ \vdots \\ x_{2n} \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \\ n \\ \vdots \\ n \end{pmatrix}$$

Derudover har vi brug for en optimal løsning, og ikke helt overraskende viser det, at der findes en sådan.

$$\boldsymbol{x}^* = \begin{pmatrix} x_1^* \\ x_2^* \\ \vdots \\ x_n^* \\ x_{n+1}^* \\ x_{n+2}^* \\ \vdots \\ x_{2n-1}^* \\ x_{2n}^* \end{pmatrix} = \begin{pmatrix} 2n \\ 0 \\ \vdots \\ 0 \\ n(n-1) \\ n(n-2) \\ \vdots \\ n(n-(n-1)) \\ 0 \end{pmatrix} = \begin{pmatrix} 2n \\ 0 \\ \vdots \\ 0 \\ n(n-1) \\ n(n-2) \\ \vdots \\ n \\ 0 \end{pmatrix}$$

Problemet er selvfølgelig konstrueret på en sådan måde, at vi var sikre på at få denne løsning.

Eksempel 3.2 Vi tager lige et hurtigt taleksempel med $n = 3$:

$$\begin{aligned} &\max \quad 13x_1 + 16x_2 + 19x_3 \\ &\text{under bibetingelserne} \\ &\quad x_1 + 3x_2 + 5x_3 \leq 12 \\ &2x_1 + 4x_2 + 6x_3 \leq 15 \\ &3x_1 + 5x_2 + 7x_3 \leq 18, \end{aligned}$$

som efter omdannelse har udseendet

$$\begin{aligned} &\max \quad 13x_1 + 16x_2 + 19x_3 \\ &\text{under bibetingelserne} \\ &\quad x_1 + 3x_2 + 5x_3 + x_4 \qquad\quad = 12 \\ &2x_1 + 4x_2 + 6x_3 \qquad + x_5 \qquad = 15 \\ &3x_1 + 5x_2 + 7x_3 \qquad\quad + x_6 = 18 \end{aligned}$$

Jf. det ovenstående eksisterer der en mulig løsning

$$\boldsymbol{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 3 \\ 3 \\ 3 \end{pmatrix}$$

og optimal løsning

$$\boldsymbol{x}^* = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{pmatrix} = \begin{pmatrix} 2 \times 3 \\ 0 \\ 0 \\ 3(3-1) \\ 3(3-2) \\ 0 \end{pmatrix} = \begin{pmatrix} 6 \\ 0 \\ 0 \\ 6 \\ 3 \\ 0 \end{pmatrix}$$

Ovenstående problem har som nævnt den fordel, at vi både kan forudsige en mulig løsning samt den optimale løsning, og det er jo sådan set meget godt. Problemet er stadig åbent for manipulationer: Vi kan sagtens gange alle rækker igennem med en faktor ξ, idet løsningsmængden er uforandret. Vi skal senere se situationer, hvor det viser sig værdifuldt.

3.3 Præcision af algoritmerne

Vi vil i det følgende beskæftige os med præcisionen af de to metoder, hvilke parametre, der spiller ind, og hvilken betydning disse parametre har.

3.3.1 Simplex

Kigger vi nærmere på, hvad simplexalgoritmen egentlig foretager sig, så handler det om at løse ligningssystemer vha. Gauss-Jordan. Det største problem ved denne metode er, at evt. løsninger numerisk mindre end den valgte tolerance forsvinder. Tilsvarende kan vi støde på andre problemer:

Eksempel 3.3 LP-problemet

$$\begin{aligned} &\max \quad x_1 + x_2 \\ &\text{under bibetingelsen} \\ &10^8 x_1 + 10^{-8} x_2 \leq 1 \end{aligned}$$

med tolerance $\varepsilon = 10^{-14}$ vil fortælle os, at der ingen løsning er, selv om vi med simple midler kan se, at løsningen er $\boldsymbol{x}^* = (0, 10^8, 0)^T$. Årsagen er, at vi efter det første trin i algoritmen har tableauet

-10^{-8}	0	1	-10^{-8}
10^{-8}	1	10^{-16}	10^{-8}

Vores valgte tolerance medfører, at 10^{-16} udrenses, hvorfor simplexalgoritmen stopper brat.

Men når det nu giver os problemer, at vi har en sådan tolerance med, kan det så overhovedet forsvares?

Svaret ligger i de afrundingsfejl, C++ laver; vi vil tit se, at der opstår værdier som ligger en lille smule ved siden af det forventede.

Eksempel 3.4 Vi lader C++ beregne følgende simple udtryk:

$$7 \times \frac{1}{7} - 1 = -5.55112 \times 10^{-17}$$

Se vi ved jo godt, at resultatet i virkeligheden er 0, men det gør C++ ikke, omend det er meget tæt på.

Eksemplet illustrerer nogle af de problemer, vi alt for ofte roder os ud i, og disse små fejl hober sig op og bliver større og større. Man kan sagtens forestille sig, hvordan den ovenstående lille fejl kunne udvikle

sig, hvis der var 1000 af dem i en matrix, og vi derpå udførte rækkeoperationer på den. Derfor må vi ty til en rensning af matricerne efter hvert beregningstrin, hvor vi leder efter værdier, der (numerisk) er meget små.

Vi har derfor valgt en tolerance på $\varepsilon = 10^{-14}$, når vi kører simplexalgoritmen, hvilket så har den konsekvens, at de numerisk største og mindste koefficienter i en bibetingelse ikke må have en forskel i størrelsesorden på mere end ε^{-1}. I eksempel 3.3 på foregående side befandt vi os i netop denne situation.

Sluttelig kan vi prøve at vurdere hvor præcis simplexalgoritmen egentlig er, når vi skruer op for antallet af variable. Efter at have ganget alle rækker med en faktor $\xi = 10^{-4}$, fås de resultater, der er vist i tabel 3.1. De viser med al tydelighed konsekvensen af en stigning i antallet af variable: Den numerisk største fejl $E_{\max}$ bliver bare større og større.

Dog er der en anden ting vi bemærker: Vort testproblem har en yderst uheldig indflydelse på antallet af beregningstrin i simplexalgoritmen, forstået på den måde, at det afhænger lineært af antallet af variable. Vi har altså uforvarende skabt et *worst-case*-tilfælde for $n = 6, 12, 18, \ldots$, men det gælder *kun* for $\xi = 10^{-4}$.

n	# trin	$E_{\max}$	n	# trin	$E_{\max}$
6	6	1.421×10^{-14}	42	43	3.863×10^{-11}
12	13	2.416×10^{-13}	48	49	5.508×10^{-11}
18	18	2.089×10^{-12}	54	55	5.014×10^{-11}
24	25	3.098×10^{-12}	60	60	1.161×10^{-10}
30	31	5.791×10^{-12}	66	66	1.093×10^{-10}
36	37	6.906×10^{-12}			

Tabel 3.1: Største fejl i simplexalgoritmen som funktion af n

3.3.2 Interior Point

Interior Point-algoritmen har en del flere parametre, vi kan variere, så vi vil indledningsvist vurdere for hvilke værdier af γ, algoritmen giver det mindste antal iterationer og bagefter prøve at finde passende tolerancer.

3.3.2.1 Ændring af iterationsskridt

Fra den teoretiske gennemgang ved vi, at et lille γ leder til en stor formindskelse af det komplementære spænd, men det kan være interessant

at undersøge hvor lille vi egentlig kan sætte γ, før det fører os ud i problemer. I tabel 3.2 ses resultaterne af en variation af γ for faste tolerancer $\varepsilon_{P,D,G} = 10^{-8}$. $E_{\max}$ angiver den numerisk største afvigelse fra den korrekte løsning.

# variable n	5	10	20	30
γ	0.9	0.9	0.9	0.9
# iterationer	249	280	312	330
$E_{\max}$	6.429×10^{-6}	4.083×10^{-6}	2.100×10^{-6}	1.496×10^{-6}
γ	0.5	0.5	0.5	0.5
# iterationer	40	45	48	50
$E_{\max}$	4.453×10^{-6}	2.197×10^{-6}	2.125×10^{-6}	1.269×10^{-6}
γ	0.1	0.1	0.1	0.1
# iterationer	15	17	19	21
$E_{\max}$	7.219×10^{-7}	7.274×10^{-7}	1.059×10^{-6}	2.362×10^{-7}
γ	0.05	0.05	0.05	0.05
# iterationer	13	15	17	19
$E_{\max}$	2.404×10^{-6}	1.125×10^{-6}	1.513×10^{-6}	1.779×10^{-7}
γ	0.01	0.01	0.01	0.01
# iterationer	12	13	15	16
$E_{\max}$	5.078×10^{-6}	8.916×10^{-7}	4.246×10^{-8}	1.422×10^{-7}
γ	0.005	0.005	0.005	0.005
# iterationer	12	13	14	15
$E_{\max}$	1.885×10^{-7}	3.827×10^{-7}	9.783×10^{-8}	8.748×10^{-7}
γ	0.001	0.001	0.001	0.001
# iterationer	12	13	13	14
$E_{\max}$	1.591×10^{-8}	1.736×10^{-8}	7.479×10^{-7}	8.803×10^{-7}
γ	0.0001	0.0001	0.0001	0.0001
# iterationer	12	17	19	24
$E_{\max}$	1.229×10^{-9}	1.318×10^{-9}	3.904×10^{-10}	7.275×10^{-7}
γ	0.00001	0.00001	0.00001	0.00001
# iterationer	18	÷	35	21
$E_{\max}$	1.696×10^{-9}	÷	2.734×10^{-11}	9.490×10^{-7}

Tabel 3.2: Ændring af γ med faste tolerancer $\varepsilon_{P,D,G} = 10^{-8}$

Resultaterne er meget interessante på flere måder. Jo tættere γ er på 1,

jo flere iterationer er det nødvendigt at bruge, men vi ser, at vi heller ikke hovedløst kan sætte γ vilkårligt tæt på 0. Faktisk sker der det ved $\gamma = 10^{-5}$, at programmet går ned for $n = 10$, mens der for $n = 20$ bruges intet mindre end 35 iterationer. Yderligere bemærkes det at programmet går ned for $n = 19$ og $n = 22$, $n = 18$ bruger 26 iterationer og $n = 21$ bruger 20 iterationer.

Det tyder alvorligt på, at det første trin i beregningerne simpelthen bliver for stort, hvor vi kan være uheldige at ramme nogle værdier, hvor algoritmen enten kører lidt i "selvsving" eller $\boldsymbol{x}$ og $\boldsymbol{s}$ indeholder værdier, der ligger under vores udrensningstolerance (eksempel 3.4 på side 19). Dette forklarer hvorfor programmet går ned, da der derfor divideres med nul. Vi ser samme tendenser ved $\gamma = 10^{-4}$, hvor der er noget større spring end forventet mellem iterationsantallet for $n = 20$ og $n = 30$. Man ser også, at den største afvigelse ligger en faktor 10^2 fra den nærmeste, så denne værdi af γ er for usikker.

I udvælgelsen af et "passende" γ er der nogle andre faktorer, vi skal tage højde for. I vort testproblem er værdierne i den oprindelige A-matrix alle mindre end 1, og det viser sig at have indflydelse på resultatet. Sætter vi vores multiplikationsfaktor $\xi = 0.1$ ser vi, at $n = 30$ løses fint med stor præcision, mens $n = 31$ får programmet til at gå ned. Derfor har vi valgt at sætte $\gamma = 0.05$ i resten af vore testkørsler, da denne værdi sikrer, at vi får et brugbart resultat uden programnedbrud. At det så medfører nogle ekstra iterationer, er en pris vi gerne betaler.

3.3.2.2 Ændring af tolerancer

Vi vil nu overveje tolerancernes (ε_P, ε_D og ε_G) betydning for præcisionen af løsningen. Ud fra de indledende testkørsler af programmet tegnede der sig et billede af, at det var ε_G, der var den udslagsgivende: Hvis det stopkriterium var opfyldt, lå normen af de to residualer langt under denne tolerance. Vi har derfor valgt at sætte

$$\varepsilon_P = \varepsilon_D = \varepsilon_G$$

Af tabel 3.3 på næste side lader det umiddelbart til at præcisionen bare bliver bedre i takt med at tolerancen formindskes, men for $n = 30$ ser vi at forskellen i præcision mellem $\varepsilon_G = 10^{-9}$ og $\varepsilon_G = 10^{-10}$ er uforandret. Endvidere kan vi konstatere, at $\varepsilon_G = 10^{-14}$ medfører $E_{\max} = 4.15 \times 10^{-11}$, $\varepsilon_G = 10^{-13}$ medfører $E_{\max} = 2.65 \times 10^{-11}$ og $\varepsilon_G = 10^{-12}$ medfører $E_{\max} = 7.77 \times 10^{-11}$, hvilket virker lidt tilfældigt. Ved $\varepsilon_{P,D,G} = 10^{-15}$ går programmet selvfølgelig ned, fordi udrensningstolerancen på 10^{-14} spiller ind.

# variable n	5	10	20	30
Tolerance ε_G	10^{-4}	10^{-4}	10^{-4}	10^{-4}
# iterationer	10	11	14	15
$E_{\max}$	5.093×10^{-3}	2.175×10^{-2}	1.764×10^{-3}	2.045×10^{-3}
Tolerance ε_G	10^{-5}	10^{-5}	10^{-5}	10^{-5}
# iterationer	11	12	15	16
$E_{\max}$	4.488×10^{-4}	1.923×10^{-3}	1.967×10^{-4}	2.447×10^{-4}
Tolerance ε_G	10^{-6}	10^{-6}	10^{-6}	10^{-6}
# iterationer	11	13	15	17
$E_{\max}$	4.488×10^{-4}	2.004×10^{-4}	1.967×10^{-4}	2.478×10^{-5}
Tolerance ε_G	10^{-7}	10^{-7}	10^{-7}	10^{-7}
# iterationer	12	14	16	18
$E_{\max}$	3.754×10^{-5}	1.670×10^{-5}	1.857×10^{-5}	2.240×10^{-6}
Tolerance ε_G	10^{-8}	10^{-8}	10^{-8}	10^{-8}
# iterationer	13	15	17	19
$E_{\max}$	2.404×10^{-6}	1.125×10^{-6}	1.513×10^{-6}	1.779×10^{-7}
Tolerance ε_G	10^{-9}	10^{-9}	10^{-9}	10^{-9}
# iterationer	14	16	18	20
$E_{\max}$	1.276×10^{-7}	6.230×10^{-8}	1.031×10^{-7}	1.190×10^{-8}
Tolerance ε_G	10^{-10}	10^{-10}	10^{-10}	10^{-10}
# iterationer	15	17	19	20
$E_{\max}$	6.400×10^{-9}	3.149×10^{-9}	5.819×10^{-9}	1.190×10^{-8}

Tabel 3.3: Ændring af tolerance ε_G med fast $\gamma = 0.05$

På baggrund af disse observationer har vi valgt at sætte $\varepsilon_G = 10^{-8}$, fordi denne værdi giver resultater, vi både kan stole på og fordi vi næppe på noget tidspunkt vil indtaste værdier med en præcision på mere end de seks cifre, som denne tolerance giver os.

3.4 Sammenligning af algoritmerne

Til sammenligningen af algoritmerne er det oplagt at vurdere parametre som præcision, kørselstid som funktion af antal variable m.v.

3.4.1 Interior Point – hastighed og iterationer

Det store problem med Interior Point-algoritmen er, at antallet af beregninger i hvert iterationstrin er større end for hele simplexalgoritmen for det samme problem. Vi så tidligere, at simplexalgoritmen sagtens kan få opgaver, der kræver at hver eneste bibetingelse køres igennem, men hvad sker der med Interior Point? Kan vi forudsige hvor mange iterationstrin I, der skal til for et givent antal variable n? Det vil vi undersøge nærmere i det følgende.

Med tolerance $\varepsilon_G = 10^{-8}$ og $\gamma = 0.05$ får vi en stribe sammenhængende værdier af I og n. De talpar (n, I), der påberåber sig mest interesse, er dem, hvor der sker et skift i antallet af interationer. Resultaterne kan ses i tabel 3.4.

n	# iterationer	n	# iterationer
1	8	9	15
2	11	13	16
3	12	17	17
4	13	22	18
6	14	28	19

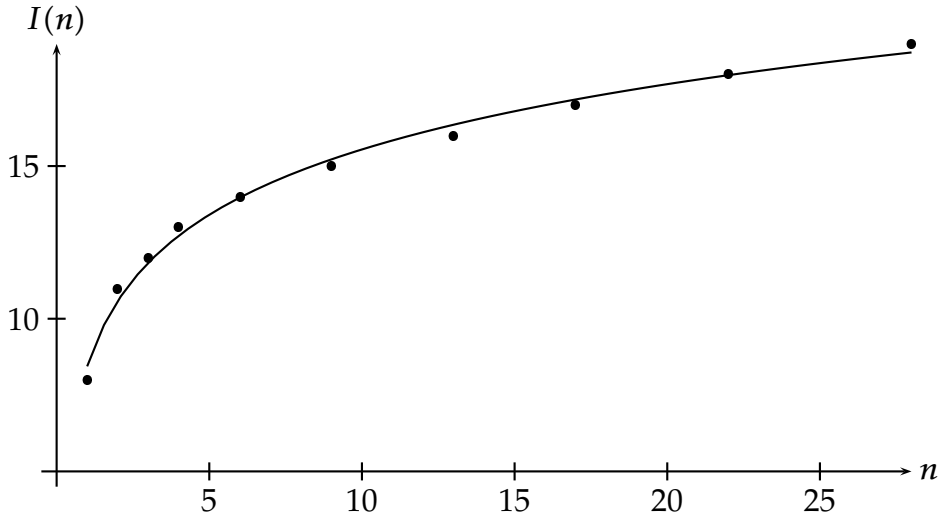
Tabel 3.4: Antal variable n og antal iterationer med $\varepsilon = 10^{-8}$ og $\gamma = 0.05$

Men et er at se resultaterne i en tabel, noget andet er at finde en sammenhæng mellem dem. Derfor er de tegnet ind i et (n, I)-diagram, hvilket ses på figur 3.1 på næste side. Man bemærker den aftagende stigning af iterationer som funktion af variable, og med en logaritmisk regression finder vi udtrykket

$$I(n) = 3.085 \ln n + 8.441, \tag{3.3}$$

der har en korrelationskoefficient $\text{corr}(n, I) = 0.9963$, hvilket må siges at være en glimrende overensstemmelse mellem model og data. Vore tidligere testkørsler taget i betragtning, må det tages for givet, at $I(n)$ også har γ og $\varepsilon_{P,D,G}$ som parametre, og man kunne sagtens forestille sig en nærmere undersøgelse af dette aspekt, hvilket i sig selv ville være en endog meget stor opgave.

Efter vi har fået styr på iterationsantallet, er vi interesseret i at få at vide hvor mange beregninger, der egentlig foretages. Derfor har vi i vort `Test_Mode` indbygget et "stopur", der holder øje med forbruget af tid, når vi fx gentager iterationsprocessen 1500 gange. En fornuftig test kunne derfor være at undersøge, hvor lang tid $T_{IP}(N)$ der går på at køre



Figur 3.1: Antal variable n og antal iterationer med $\varepsilon = 10^{-8}$ og $\gamma = 0.05$

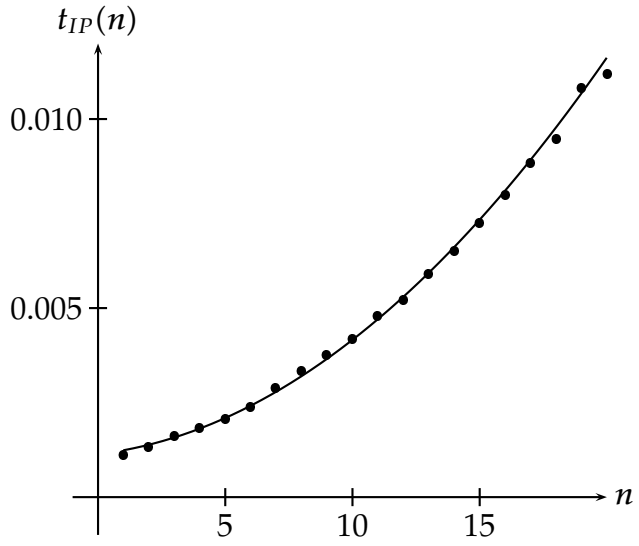
et bestemt antal gentagelser N. Dog skal man her være opmærksom på, at det tager væsentligt længere tid at køre 1500 gentagelser ved $n = 30$ end ved $n = 20$, hvorfor man bør formindske antallet af gentagelser, som n vokser.

Den information, vi får fra dette forsøg, er værdifuld i sig selv, men bliver endnu bedre, når vi sammenholder den med den foregående test. Vi har jo styr på antallet af iterationer for et givet n, som vi kan dividere ud, så vi kan beregne tidsforbruget pr. iteration. Resultatet af denne test kan ses i tabel 3.5.

n	$t_{IP}(n)$	n	$t_{IP}(n)$	n	$t_{IP}(n)$
16	1.110×10^{-3}	23	3.333×10^{-3}	30	7.263×10^{-3}
17	1.345×10^{-3}	24	3.778×10^{-3}	31	8.000×10^{-3}
18	1.627×10^{-3}	25	4.189×10^{-3}	32	8.842×10^{-3}
19	1.843×10^{-3}	26	4.778×10^{-3}	33	9.474×10^{-3}
20	2.090×10^{-3}	27	5.222×10^{-3}	34	1.084×10^{-2}
21	2.392×10^{-3}	28	5.895×10^{-3}	35	1.119×10^{-2}
22	2.889×10^{-3}	29	6.526×10^{-3}		

Tabel 3.5: Iterationstid som funktion af variable n

Igen kan det fremme overskueligheden at se det i et koordinatsystem som vist i figur 3.2 på næste side. Det ses, at punkterne tenderer en parabel, og ved nærmere undersøgelse viser det sig, at vi ved at lægge



Figur 3.2: Beregningstid t_{IP} pr. iteration for Interior Point som funktion af variable n.

parablen

$$t_{IP}(n) = 2.222 \times 10^{-5} n^2 - 5.868 \times 10^{-4} n + 4.939 \times 10^{-3}, \tag{3.4}$$

opnår en korrelationskoefficient $\text{corr}(n, t_{IP}(n)) = 0.9988$, hvilket stort set ikke kunne være bedre. Det virker også naturligt, at $t(n)$ er polynomial, idet vi ved en fordobling af n i input kvadrerer matrixstørrelserne i beregningerne.

Nu har vi altså en situation, hvor vi med Interior Point-algoritmen kan forudsige a) hvor mange interationstrin der er nødvendige for et givet antal variable n og b) hvor lang tid et iterationstrin tager for et givet n. Den samlede gennemløbstid T_{IP} for Interior Point-algoritmen må da være givet ved

$$T_{IP}(n) = I_{IP}(n) t_{IP}(n) \tag{3.5}$$

Der er nu et simpelt spørgsmål, der trænger sig på: Hvordan opfører $T(n)$ sig for store n? Vi ved at $I(n)$ er logaritmisk, så denne størrelse vil miste sin betydning for $n \to \infty$ (Fuglede et al., 1999, p. 11), og det vil således være $t(n)$, der er den toneangivende størrelse. Alt i alt kan vi altså konkludere, at der er en klar polynomial sammenhæng mellem antal variable n og iterationstiden T, hvilket Karmarkar i sin tid også forudsagde (Keiding, 2002, p. 60).

3.4.2 Simplex – hastighed og iterationer

Efter at have klarlagt Interior Point-algoritmens opførsel som en funktion af antallet af variable n, vil vi nu kigge nærmere på simplex. Det konstrurede problem (3.2) har den ubehagelige egenskab, at simplex skal køre alle rækkerne i tableauet igennem, hvilket medfører at iterationsantallet er en lineær funktion af n, dvs.

$$I_S(n) = n \tag{3.6}$$

Lige som for Interior Point-algoritmen må vi forvente en polynomial sammenhæng (af 2. grad) mellem variable og antal beregninger pr. iterationstrin, så alt i alt giver det os de data, der ses i tabel 3.6 og figur 3.3 på næste side. Ud fra dette findes følgende udtryk for den samlede gennemløbstid som funktion af n:

$$\begin{aligned} T_S(n) = {} & 1.070 \times 10^{-7} n^3 + 2.336 \times 10^{-6} n^2 \\ & - 3.930 \times 10^{-5} n + 2.027 \times 10^{-4} \end{aligned} \tag{3.7}$$

Korrelationskoefficienten er her $\text{corr}(n, T_S(n)) = 0.9999$, hvilket er glimrende. Altså har vi en polynomial sammenhæng af 3. grad mellem simplexalgoritmens køretid og antal variable! Det betyder dermed at simplexalgoritmen på et tidspunkt vil bruge længere tid på at løse et problem end Interior Point-algoritmen. Vi kan så nu prøve at beregne, hvornår det vil ske. Det er nemlig det mindste n, der opfylder uligheden

$$T_S(n) > T_{IP}(n),$$

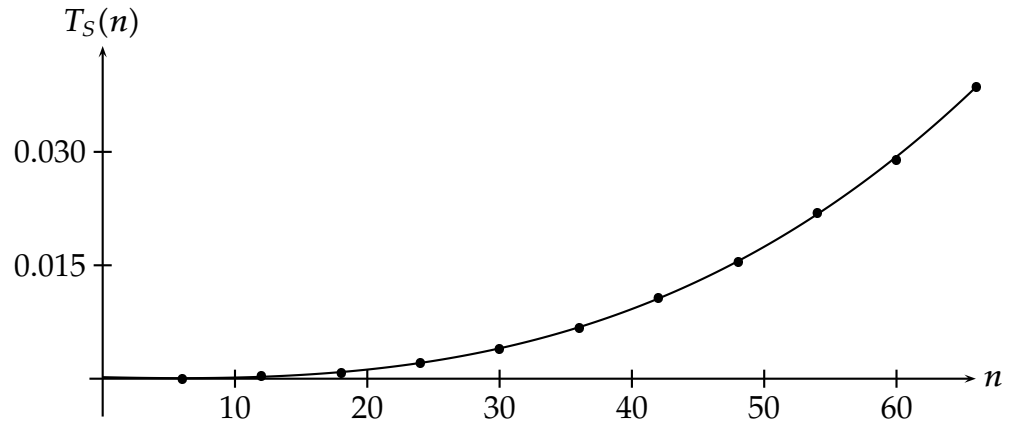
hvilket giver os, at

$$n = 7\,428 \quad \text{med} \quad T_S = 43\,981 \quad \text{og} \quad T_{IP} = 43\,979 \tag{3.8}$$

For større tolerancer og mindre γ vil man kunne opnå en formindskelse af dette antal variable.

n	$T_S(n)$	n	$T_S(n)$	n	$T_S(n)$
6	4.700×10^{-5}	30	3.900×10^{-3}	54	2.200×10^{-2}
12	3.067×10^{-4}	36	6.800×10^{-3}	60	2.900×10^{-2}
18	8.800×10^{-4}	42	1.067×10^{-2}	66	3.867×10^{-2}
24	2.100×10^{-3}	48	1.550×10^{-2}		

Tabel 3.6: Gennemløbstid T_S for simplex som funktion af variable n.



Figur 3.3: Beregningstid T_S for simplex som funktion af variable n

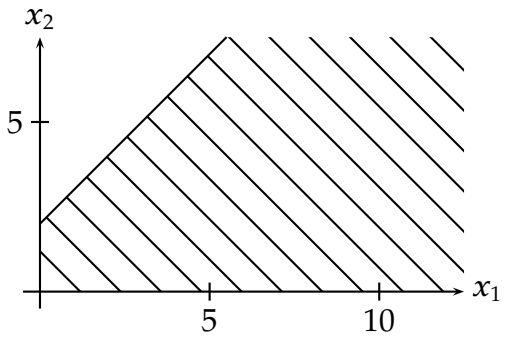
3.5 Specialtilfælde

Et problem, der specifikt kun gælder for Interior Point, er, at der skal bruges en mulig løsning som udgangspunkt, hvis man vil være sikker på at få et resultat, man kan regne med. Det er et reelt problem, da vi for meget store antal bibetingelser sådan set skal løse det tilhørende lineære ligningssystem først.

Løsningsforslag: Man kan enten finde en mulig løsning ved at løse bibetingelserne separat, men det er en meget krævende metode. En bedre mulighed er at anvende en model, der kaldes den *homogene og selv-duale metode* (Andersen, 1998, p. 60). Implementeringen af denne ligger langt ud over opgavens rammer, og er i sig selv et større projekt.

Vi vil nu belyse de specialtilfælde, hvor der kan opstå nogle problemer for simplex og Interior Point.

3.5.1 Ingen optimal løsning



Følgende problem betragtes:

$$\begin{aligned} &\max \quad x_1 - x_2 \\ &\text{under bibetingelserne} \\ &-x_1 + x_2 \leq 2 \\ &x_1, x_2 \geq 0 \end{aligned}$$

Dette maksimeringsproblem har ikke nogen optimal løsning, da

alle punkter på linjen $x_2 = 0$ opfylder betingelserne, og dermed kan x_1 gøres vilkårligt stor. Det skyldes, at løsningsmængden er en lukket, men ubegrænset mængde.

Hvis simplex skal løse dette problem, bliver start-tableauet:

	1	−1	0
2	−1	1	1

Ud fra tableauet ses det tydeligt, at simplex får problemer allerede ved første beregningstrin. Der kan ikke anvendes almindelig simplex, da der ikke findes noget pivotelement, som opfylder betingelserne. Dual simplex kan heller ikke benyttes, da der ikke er nogle negative koefficienter i b-søjlen. Dermed bliver resultatet, at simplex ikke kan finde nogen løsning.

Hvis Interior Point anvendes med den mulige løsning

$$\boldsymbol{x} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$$

som udgangspunkt, fås fx løsningen:

$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 3 \\ 0 \end{pmatrix}$$

Denne løsning er mulig, men ikke optimal.

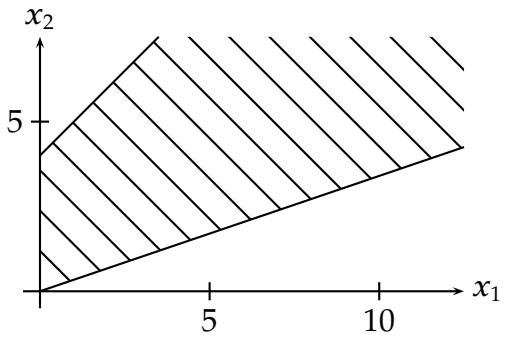
Hvis man skal sammenligne de to løsninger, vil man jo nok umiddelbart mene, at indre punkts metoden opnår den bedste løsning, da den i det mindste får et resultat.

Men på den anden side er resultatet med simplex også meget brugbart. Her får vi nemlig en oplysning, som indre punkts metoden ikke giver os. Simplex fortæller os faktisk, at den ikke kan maksimere dette problem, hvilket jo er rigtigt.

Resultatet fra indre punkts metoden kan medføre misforståelser, hvis problemets antal af variable øges, og man dermed ikke har mulighed for at overskue problemet grafisk som i dette tilfælde. Dermed kan man nemt komme til at forveksle denne løsning med en optimal løsning.

Løsningsforslag: En mulig løsning på problemet kunne være at lade Interior Point-algoritmen teste hvor store skridt, der egentlig tages. For γ tilstrækkelig lille ved vi jo, at vi ved hver iteration kommer tættere på løsningen, end vi var i forvejen, hvorfor det ville give god mening at sammenligne interationstrinnet $\boldsymbol{d}_x^k$ med det foregående $\boldsymbol{d}_x^{k-1}$. Her kunne algoritmen stoppe med en passende meddelelse til brugeren.

3.5.2 Lukket og ubegrænset løsningsmængde



Problemet er følgende:

$$\begin{aligned} &\max \quad x_1 - 2x_2 \\ &\text{under bibetingelserne} \\ &-x_1 + x_2 \leq 4 \\ &x_1 - 2x_2 \leq 0 \\ &x_1, x_2 \geq 0 \end{aligned}$$

Igen har vi en lukket, men ubegrænset løsningsmængde, og da kriterieværdien kan gøres vilkårligt stor, eksisterer der ikke nogen optimal løsning.

Det ses også ved, at alle punkter på linjen $x_2 = \frac{1}{2}x_1$, som begrænser løsningsmængden nedadtil, opfylder betingelserne, og dermed kan x_1 og x_2 gøres vilkårligt store. Så der opnås uendelig mange løsninger.

Simplex finder følgende løsning, når den løser problemet:

$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 4 \\ 0 \end{pmatrix}$$

Den finder altså det venstre endepunkt for den tidligere omtalte linje. Det stemmer også meget godt overens med den fremgangsmåde, simplex anvender. Den undersøger nemlig hjørnerne fra venstre mod højre.

I dette tilfælde er det egentlig et meget fornuftigt resultat, når man tænker nærmere over det, selvom der ikke findes nogen optimal løsning. Alle punkterne på linjen $x_2 = \frac{1}{2}x_1$ giver en kriterieværdi, som er lig 0. Så hvis man opnår den samme kriterieværdi, uanset hvor meget af x_1 og x_2 man anvender, så vil det jo være meget logisk at anvende 0 enheder af begge, da det giver de laveste omkostninger.

Det "farlige" ved denne løsning er, at man får en mulig løsning ud af mange løsninger, som er lige så anvendelige. Derved kan man være tilbøjelig til at opfatte det som en optimal løsning.

Selvom indre punkts metoden får følgende mulige løsning som startpunkt:

$$\boldsymbol{x} = \begin{pmatrix} 1 \\ 1 \\ 4 \\ 1 \end{pmatrix}$$

finder den ikke nogen løsning. Istedet vokser x_1 og x_2 mod uendelig. Det giver selvfølgelig også god mening, da x_1 og x_2 som sagt kan gøres vilkårligt store.

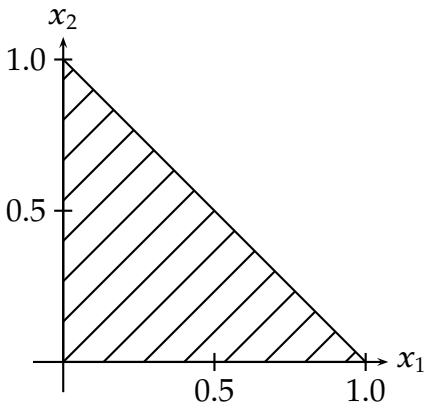
Hvis man sammenligner de to resultater, er situationen den modsatte af det forrige tilfælde. Indre punkts metodens resultat er klart at foretrække, da det ikke kan give anledning til misforståelser.

Løsningsforslag: Problemet med Interior Point er at metoden starter med en mulig løsning, og så kører den forkerte vej, fordi søgeretningen er forkert. Man kunne godt forestille sig, at en mere avanceret implementering undersøger om nogle af bibetingelserne er parallelle med kriteriefunktionen, hvilket kunne afhjælpe problemet i en vis grad.

3.5.3 Kompakt løsningsmængde

Vi ser på problemet:

$$\begin{aligned} &\min \quad -2x_1 - 2x_2 \\ &\text{under bibetingelserne} \\ &x_1 + x_2 \leq 1 \\ &x_1, x_2 \geq 0 \end{aligned}$$



I modsætning til de to forrige problemer har vi denne gang en kompakt løsningsmængde. Alligevel opnås mange løsninger, da hele linjen $x_2 = 1 - x_1$ minimerer problemet. Simplex finder følgende løsning til problemet:

$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$$

Igen giver det god mening, at den finder et højre hjørne i den mulige løsningsmængde, da simplex starter med at lede efter en optimal løsning fra venstre mod højre.

Situationen er den samme som i det forrige problem. Man skal være varsom med at opfatte den som en optimal løsning, da der er andre løsninger, som giver samme kriterieværdi.

Indre punkts metoden derimod finder denne løsning:

$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} ½ \\ ½ \\ 0 \end{pmatrix}$$

Denne løsning genkender vi som midtpunktet af løsningslinjen, og det er lige præcis hvad Interior Point-algoritmen vil gøre i tilfælde af flere optimale løsninger (Andersen, 1998, p. 59–60).

Løsningsforslag: Problemet ligger igen i, at bibetingelsen og kriteriefunktion er parallelle, som man kunne tage højde for, men det ville kræve en noget anden tilgangsvinkel til tingene. En løsning kunne være at have en præprocessor, der tjekker LP-problemet for sådanne parallelle funktioner.

Kapitel 4

Konklusion

Ud fra vore testkørsler tegner der sig hurtigt et billede – omend noget tvetydigt – af styrkeforholdet mellem simplex- og Interior Point-algoritmen. Umiddelbart fremstår simplex som sejrherre, men som vi har set i løbet af testkørslerne, vil både præcision og hastighed blive forringet væsentligt, når antallet af variable vokser. Interior Point har ikke i samme grad disse problemer, men er til gengæld en meget langsom starter – det kræver mange variable (typisk flere tusinde), før den hastighedsmæssigt er konkurrencedygtig med simplex.

For den almindelige bruger er simplex dermed klart at foretrække, men skal man løse virkeligt store problemer af fx nationaløkonomisk karakter, må vi sige, at vinden blæser i retning af en Interior Point-baseret metode.

Derudover er der flere ting i vore testkørsler, der tyder på, at man med fordel kunne køre LP-problemet gennem en præprocessor, der forholdsvis hurtigt kunne afgøre, om der fandtes enten mange eller ingen løsninger, hvilket ville kunne være værdifuldt for brugeren i sparet tid.

Bilag A

C++-koden

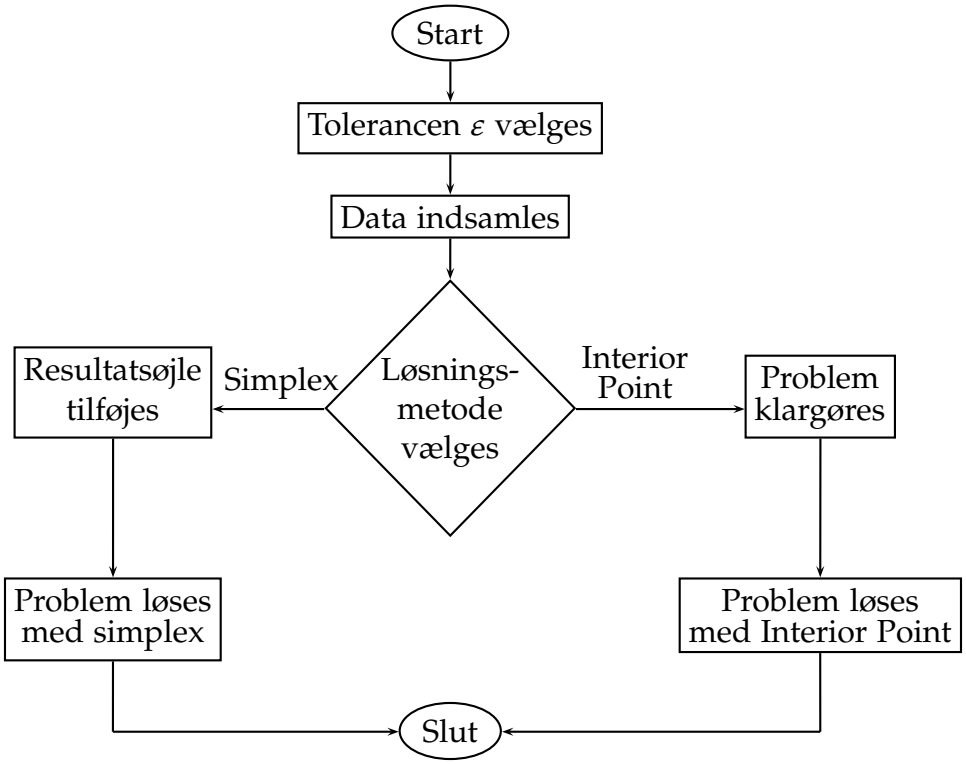
A.1 Indledende bemærkninger

Den følgende programkode er forsøgt gjort så overskuelig som overhovedet muligt vha. *Literate Programming* – en teknik, hvor kode og kommentarer er skrevet som sammenhængende dele af det samme kildedokument. Som programmør er én af fordelene, at man med simple midler kan kommentere sit program gennemgribende, således at man lettere kan forstå det, når man vender tilbage til det. For læseren er der om muligt endnu flere fordele. Eksempelvis er kildedokumentet til nærværende rapport udstyret med indstillinger, der gør at C++-udtryk som `!=`, `==`, `<=`, `&&` m.fl. automatisk skrives som $\neq$, $\equiv$, $\leq$, $\wedge$ osv. Tilsvarende med nøgleord såsom `int` og `char`, der fremhæves som **`int`** og **`char`**. Man kan også sagtens forestille sig hvor let det er at overskue et program som det nedenstående, der strækker sig over 1079 linjer. Alt i alt gøres der altså en tjeneste både for programmør og læser.

Ud over brugen af denne teknik, er der gjort stor umage med at give funktioner og variable meningsfyldte navne, sådan så funktionen `AppendColumnToMatrix` rent faktisk tilføjer en søjle(vektor) til en matrix. Dette fremmer både læseligheden af programkoden såvel som det reducerer fejlene fra programmørernes hånd. Det er trods alt nemmere at overskue, hvad der sker, når en funktion indeholder beskrivende navne som `InputMatrix` og `OutputMatrix` i stedet for de kortere `A` og `B`. Den ekstra tid brugt på indtastning af de lange navne, der i øvrigt er holdt konsekvent på engelsk, er godt givet ud.

Med disse ord kan man roligt give sig i kast med læsningen af programmet. Det er lettere, end man tror!

A.2 Hovedprogrammet



Selve vort hovedprogram er meget simpelt. Indledningsvist inkluderes filerne indeholdende funktionsprototyper og funktionsdefinitioner.

```
1 #include "header.h" Se afsnit A.7.
2 #include "funktion.cpp" Alle funktionerne.
```

Herefter kan vi gå igang med `main`. I visse situationer kan det være meget rart at få programmet til at vise mere information end ellers, fx ved fejlsøgning. Man kunne ændre i programmet hele tiden ved at indsætte ekstra `cout`'er på "passende" steder, men vi har en anden løsning. Programmet kan tage et valgfrit antal argumenter, når det køres fra kommandoprompten på en computer. Se linje 870 på side 76 for nærmere. I vores implementering har vi valgt at lade argumentet `-info` sørge for at den booleske operator `PrintInfo` sættes til sand, så vi kan bruge den i konstruktioner som vist i linje 25 på næste side. Tilsvarende sætter argumentet `-test` `Test_Mode` til sand, og vi kører i stedet testproblemet.

```
3 int main(int argc, char *argv[]){
4    XtraOps(argc, argv);
5    if(!Test_Mode)
6       ChooseTolerance(Tolerance); Tolerancen vælges.
```

Vi går så småt i gang med at definere de variable, vi har brug: Arrays, booleske operatorer (true/false) m.fl. For dem, der ikke skulle vide det, så er en boolesk operator (i al fald dem, vi benytter) simpelthen en variabel, der kan have enten sand eller falsk som værdi.

```
7   int EquationNumber=0;
8   int NumberOfColumns=0;
9   double TrueResult[MaxBigCol]={0}; Sætter alle elementerne til
        0.
10  double ResultCoef[MaxRow];
11  double CoefAndSlack[MaxRow][MaxBigCol];
12  if(Test_Mode)
13      SetupTests(MaxTestRun, NumberOfColumns,
            EquationNumber, CoefAndSlack, ResultCoef,
            TrueResult);
14  double SimplexTableau[MaxRow][MaxBigCol+1];
15  bool IP_Method=false; Til indre punkters metode.
16  bool MaxProblem=false;
```

Efter disse indledende erklæringer går vi i gang med funktionskaldene. Dataindsamlingen startes, og ud kommer der svar på, om brugeren tastede et maksimerings- eller et minimeringsproblem, antallet af bibetingelser og variable, en vektor `ResultCoef` og en matrix `CoefAndSlack`, der som navnene antyder indeholder hhv. resultatsøjlen og koefficienter plus slack-variable.

```
17  if(!Test_Mode)
18      CollectData(MaxProblem, ResultCoef, CoefAndSlack,
            EquationNumber, NumberOfColumns);
19  ChooseOneOfTwo("interior", "simplex", IP_Method); Brugeren
        vælger simplex eller indre punkters metode.
```

Til indre punkts metoden skal vi foretage et par ændringer af placeringen af visse data. Eksempelvis skal kriteriefunktionen gemmes separat, så vi kalder `PrepareIP` til dette formål.

```
20  time_t Start_Time, Stop_Time;
21  if(Test_Mode)
22      Start_Time=time(NULL); Starter tidtagning.
23  while(NumberOfTestRuns<MaxTestRun){
24      if(IP_Method){
25          if(PrintInfo)
26              cout≪endl≪"Interior Point er valgt.";
27          double CritFunc[MaxBigCol]; kriteriefunktionen.
```

```
28          int ColsInCoefAndSlack=NumberOfColumns+
               EquationNumber; Samlet antal søjler.
29          PrepareIP(MaxProblem, CoefAndSlack, ResultCoef,
               EquationNumber, ColsInCoefAndSlack, CritFunc)
               ;
30          if(PrintInfo){
31             cout≪endl≪"PrepareIP udført. EquationNumber er
                  "≪EquationNumber≪" ColsInCoefAndSlack er "≪
                  ColsInCoefAndSlack;
32             PrintVector(CritFunc, ColsInCoefAndSlack);
33             PrintVector(ResultCoef, EquationNumber);
34             PrintMatrix(CoefAndSlack, EquationNumber,
                  ColsInCoefAndSlack);
35          }
36          InteriorPoint(CoefAndSlack, ResultCoef,
               CritFunc, EquationNumber, ColsInCoefAndSlack,
                TrueResult); Løser LP-problemet.
37          if(Test_Mode){
38             EquationNumber++;
39             ColsInCoefAndSlack++;
40          }
41       }
```

Vi har valgt at løse LP-problemet vha. simplex, så vi starter med at tilføje resultatsøjlen, så vi har et komplet simplextableau. Derefter løses LP-problemet.

```
42       else{
43          AppendColumnToMatrix(ResultCoef, CoefAndSlack,
               NumberOfColumns+EquationNumber-1,
               EquationNumber, SimplexTableau);
44          if(PrintInfo)
45             PrintMatrix(SimplexTableau, EquationNumber,
                  EquationNumber+NumberOfColumns);
46          Simplex(SimplexTableau, EquationNumber,
               EquationNumber+NumberOfColumns, TrueResult);
47          if(PrintInfo)
48             PrintMatrix(SimplexTableau, EquationNumber,
                  EquationNumber+NumberOfColumns+1);
49       }
50       NumberOfTestRuns++;
51    }
52    if(Test_Mode){
```

```
53          Stop_Time=time(NULL); Stopper tidtagningen.
54          cout≪endl≪"Der er gået "≪difftime(
                Stop_Time,Start_Time)≪" sekunder";
55      }
56      return 0;  Slut på programmet.
57  }
```

A.3 Indsamling af data

Den første del af programmet går ud på, at brugeren gør sig selv og programmet klart, hvad det egentlig er, der ønskes. Til det skal der bruges:

- En funktion, der afgør hvilken type LP-problem, der er tale om.
- En funktion, der indsamler data.
- En funktion, der vurderer disse data for at se om der er ulighedstegn eller lignende.
- En funktion, der omdanner data til en anvendelig form.

Alt dette samler vi i funktionen `CollectData`. Det bemærkes, at det ikke er nødvendigt at angive rækkeantal på arrays, hvorfor notationen `[]` er anvendt konsekvent.

```
58  void CollectData(bool &MaxProblem, double ResultCoef[],
        double CoefAndSlack[][MaxBigCol], int &EquationNumber,
         int &NumberOfColumns){
59      double Coef[MaxRow][MaxCol];
60      double SlackVars[MaxRow][MaxRow];      Altid kvadratisk
            (enhedsmatrix).
61      InitializeMatrix(Coef);
62      InitializeMatrix(SlackVars);
63      bool LessThan=false;
64      bool GreaterThan=false;
65      bool EqualTo=false;
66      bool CriterionFunction=false;
67      bool StopKeyingIn=false;
68      string buffer;
```

Først vælges om der er tale om et maksimerings- eller minimeringsproblem.

```
69      ChooseOneOfTwo("max", "min", MaxProblem);
```

Dernæst kører vi en **do-while**-løkke, der fortsætter så længe brugeren siger ja til at fortsætte.

```
70      do{
71         ReadString(buffer);
72         EndOfEquations(buffer, StopKeyingIn);
73         if(StopKeyingIn≡false){ Vi fortsætter indtastningerne.
74            CleanUpStringSpaces(buffer);
75            DeterminEquationType(buffer, ResultCoef,
                  LessThan, GreaterThan, EqualTo,
                  CriterionFunction, EquationNumber);
76            CleanUpStringOther(buffer);
```

Efter rensningen af det indtastede konverteres det til tal i en matrix. Endvidere skal vi huske, at simplexalgoritmen virker på et maksimeringsproblem, så vi ændrer fortegn i tilfælde af at brugeren har indtastet et minimeringsproblem. Endelig sættes slack-variable ind i en matrix for sig selv, og antallet af ligninger opskrives.

```
77            ReadCoef(buffer, Coef, EquationNumber,
                  NumberOfColumns);
78            SwitchSigns(MaxProblem, CriterionFunction,
                  GreaterThan, EquationNumber, Coef, ResultCoef
                  );
79            CreateSlackVars(LessThan, GreaterThan,
                  EquationNumber, SlackVars);
80            EquationNumber++;
81         }
82      }
83      while(StopKeyingIn≡false);
84      if(PrintInfo){
85         PrintMatrix(SlackVars, EquationNumber,
               EquationNumber-1);
86         getch();
87         PrintMatrix(Coef, EquationNumber, NumberOfColumns);
88         getch();
89      }
```

Derefter kopierer vi koefficienterne og slack-variablerne over i en samlet matrix `CoefAndSlack`.

```
90      CopyMatrix(Coef, EquationNumber, NumberOfColumns, 0,
            CoefAndSlack);
91      CopyMatrix(SlackVars, EquationNumber, EquationNumber,
            NumberOfColumns, CoefAndSlack);
```

```
92 }
```

De enkelte funktioner i `CollectData` gennemgås i det følgende.

A.3.1 Maksimering eller minimering?

Brugeren skal indledningsvist afgøre, om LP-problemet er et maksimerings- eller et minimeringsproblem, og til det anvender vi den booleske operator `MaxProblem`, til at gemme brugerens valg. Vi vælger dog en mere generisk løsning, da vi kan få brug for sådan en sammenligningsfunktion senere hen.

```
93 void ChooseOneOfTwo(string FirstString, string
       SecondString, bool &Choice){
94     string Answer;
95     bool CorrectInput;
```

Vi kører nu en **do-while**-løkke, som tester, om brugeren har indtastet enten `min` eller `max`.

```
96     do{
97         CorrectInput=true;
98         cout≪endl≪"Indtast "≪FirstString≪" eller "≪
               SecondString≪": ";
99         getline(cin,Answer);
```

Da man nogle gange kan komme til at trykke på caps lock, sørger vi for, at den indtastede streng konverteres til små bogstaver. Denne praksis betyder selvfølgelig, at programmøren skal indtaste strenge kun indeholdende små bogstaver! Selve konverteringen foretages let med en **for**-løkke.

```
100        int len=Answer.length();
101        for(int i=0; i≤len; i++)
102            Answer[i]=tolower(Answer[i]);
```

Vi tester derpå indtastningen.

```
103        if (Answer≡FirstString)
104            Choice=true;
105        else{
106            if(Answer≡SecondString)
107                Choice=false;
108            else
109                CorrectInput=false;
110        }
```

```
111     }
112     while(CorrectInput≡false);
113  }
```

A.3.2 Afslutning af indtastninger

Vi vil gerne tillade brugeren en nem måde at stoppe indtastningerne på, så vi tjekker om strengen er lig med `slut`. Er den det, stoppes der, ellers må strengen betragtes som indeholdende koefficienter til LP-problemet.

```
114  void EndOfEquations(string &buffer, bool &StopKeyingIn){
115     string TheEnd("slut");
116     int len=buffer.length();
117     if(PrintInfo)
118        cout≪endl≪"buffer før: "≪buffer;
119     for(int i=0; i<len; i++)
120        buffer[i]=tolower(buffer[i]);
121     if(PrintInfo)
122        cout≪endl≪"buffer efter: "≪buffer;
123     if (buffer≡TheEnd)
124        StopKeyingIn=true;
125     else
126        StopKeyingIn=false;
127  }
```

A.3.3 Indtastning af data fra tastaturet

Vi gør nu klar til dataopsamlingen: Den indtastede linje gemmes i en `string` vha. `getline`, og denne opsamlede streng kan vi så manipulere, som vi lyster.

```
128  void ReadString(string &buffer){
129     bool SatisfiedUser; Sådan nogle skulle man gerne have!
130     do{
131        SatisfiedUser=true; :-)
132        cout≪endl≪"Indtast bibetingelser/kriteriefunktion"≪endl
               ≪"(skriv slut, hvis du ikke vil indtaste flere): ";
133        getline(cin,buffer);
134        cout≪endl≪"Er du tilfreds med din indtastning?";
135        ChooseOneOfTwo("ja", "nej", SatisfiedUser);
136     }
137     while(SatisfiedUser≡false);
```

```
138 }
```

A.3.4 Rensning af data

Den overordnede strategi ved indlæsning af data fra strengen er som følger:

1. Fjern alle overflødige tegn fra strengen. Dette kan være mellemrum, plusser osv.
2. Bestem typen af det indtastede. Er der nogle ulighedstegn?
3. Indlæs koefficienterne vha. en passende omdannelse af strengen.

Tankegangen her er at vi kan skabe en kommasepareret streng indeholdende koefficienterne til LP-problemet. Altså: vi vil tillade brugeren at taste udtryk som

```
x1 - x2 + 3.14159 x3 > - 2.71828
```

og så skal programmet kunne tolke det.

Dette er ikke noget større problem, men vi skal være sikre på, at der kun indlæses de nødvendige værdier. Derfor har vi valgt en metode, der bestemmer typen af det indtastede, læser en konstant fra bibetingelserne til en resultatsøjle og sluttelig indlæser koefficienterne til x_i'erne.

Vi må følgelig ty til en todelt udrensningsproces.

A.3.4.1 Mellemrum fjernes

Efter indtastningen af strengen renser vi den først for mellemrum. Der er to primære grunde til, at vi gør det:

- Hvis brugeren indtaster fx `- 5x1`, vil en konvertering af denne streng (vha. `atof`) føre til resultatet -0.5 forsvinder!
- Hvis brugeren indtaster fx `3x1> -5`, vil konverteringen medføre resultatet »3,« som koefficient til x_1. Kommaet inkluderes!

Det er to meget gode grunde til at fjerne mellemrummene, så det gør vi.

```
139 void CleanUpStringSpaces(string &buffer){
140    int len=buffer.length();
```

Til at søge i strengen bruges `find`-operationen, som returnerer den plads, tegnet står på. Meget praktisk viser det sig, at hvis `find` *ikke* finder det søgte tegn, returneres -1. Det udnytter vi så.

```
141     int t=buffer.find (" ");
142     while(t<len∧t≠-1){
143        buffer.erase(t,1); Fjerner tegnet på t's plads.
144        t=buffer.find(" ", t+1); Finder næste mellemrum.
145     }
146  }
```

Herefter er strengen reduceret til

```
x1-x2+3.14159x3>-2.71828
```

og vi kan gå videre med typebestemmelse af den.

A.3.5 Bestemmelse af ligningstype

Når vi nu vil til at manipulere en streng, så er det jo praktisk, hvis vi ved, hvilken type det er; fx har det stor betydning, om brugeren taster $5x_1 - 3x_2 < 5$ eller $5x_1 - 3x_2 > 5$. Til netop dette formål kreerede vi jo de booleske operatorer tidligere (linje 63–66).

Herefter skal vi så have søgt strengen igennem for at se, om der er nogle af tegnene <, > eller =. Hvis det er tilfældet kaldes funktionen `SaveResultCoef`, som gemmer den tilhørende koefficient i resultatsøjlen. Når resultatet er i hus, sletter vi det fra strengen, så det ikke også bliver gemt som en koefficient senere. Hvis ingen af dem er tilstede, må det betyde, at brugeren har indtastet en kriteriefunktion.

```
147  void DeterminEquationType(string &buffer, double
        ResultCoef[MaxRow], bool &LessThan, bool &GreaterThan,
         bool &EqualTo, bool &CriterionFunction, int
        &EquationNumber){
148     CriterionFunction=false;
149     EqualTo=false;
150     GreaterThan=false;
151     LessThan=false;
152     int t=buffer.find("<");
153     if(t≠-1){
154        LessThan=true;
155        SaveResultCoef(buffer, ResultCoef, EquationNumber,
              t);
156        buffer.erase(t); Sletter koefficienten og <.
157     }
158     else{ Vi fandt intet <, så vi søger videre.
159        t=buffer.find(">");
```

```
160         if(t≠-1){
161            GreaterThan=true;
162            SaveResultCoef(buffer, ResultCoef,
                  EquationNumber, t);
163            cout≪"t="≪t;
164            buffer.erase(t); Se linje 156.
165         }
166         else{ Vi fandt intet >, så vi søger videre.
167            t=buffer.find("=");
168            if(t≠-1){
169               EqualTo=true;
170               SaveResultCoef(buffer, ResultCoef,
                     EquationNumber, t);
171               buffer.erase(t); Se linje 156.
172            }
173            else{ Vi fandt intet =, så vi har en kriteriefunktion.
174               CriterionFunction=true;
175               ResultCoef[EquationNumber]=0;
176            }
177         }
178      }
179   }
```

Vi skal nu have gemt den tilhørende koefficient i resultatsøjlen. Når funktionen `SaveResultCoef` kaldes, kender vi placeringen af enten <, > eller = ud fra `t`. Tricket er så, at vi skal have C++ til at gemme resten af strengen som en **`double`** i et passende `array`. Derfor måler vi den totale længde af den indtastede streng og gemmer tegnene, fra der hvor der blev fundet enten <, > eller =, nemlig umiddelbart *efter* `t`. Stopkriteriet er selvfølgelig enden af strengen.

```
180   void SaveResultCoef(string &buffer, double
         ResultCoef[MaxRow], int &EquationNumber, int t){
181      string Remainder;
182      int j=0;
183      int len=buffer.length(); Længden af strengen
184      t++;
```

Vi er nødt til at tage højde for fx James Bond-inspirerede brugere, som kunne finde på at taste 007 som koefficient i resultatsøjlen. Det giver nemlig problemer for `atof`-funktionen, da den vil konvertere 007 til 0, hvilket jo ikke er hensigtsmæssigt. Så derfor sørger vi for, at eventuelle nuller i starten af den aktuelle streng bliver slettet. Der er nemlig ingen

problemer med fx at konvertere `.045` eller `-0.56`, som bliver gemt som hhv. 0.045 og −0.56.

```
185     while(buffer[t]≡'o')
186        buffer.erase(t,1);
187     cout≪"buffer[t]="≪buffer[t];
188     if(PrintInfo)
189        cout≪"Længden af buffer er "≪len;
190     while(t<len){
191        Remainder[j]=buffer[t];
192        j++;
193        t++;
194     }
```

Nu er løkken færdig, så vi skal bare have konverteret strengen til en **double**. For at vi kan gøre det, skal der sættes et `NULL` på den sidste plads i strengen, hvorefter tallet gemmes i et array til resultatsøjlen.

```
195     Remainder[j]='\o';
196     ResultCoef[EquationNumber]=atof(Remainder.end());
197     if(PrintInfo){
198        if(Remainder.end()≡"o")
199            cout≪"Remainder var 'o'!";
200        cout≪"\t Remainder ="≪Remainder.end();
201        cout≪"resultatsøjle ="≪ResultCoef[EquationNumber];
202     }
203  }
```

Vor streng er nu reduceret til

```
x1-x2+3.14159x3
```

A.3.5.1 Andre tegn fjernes

Efter at have bestemt typen af strengen og fjernet et evt. konstantled kan vi nu gå videre.

```
204  void CleanUpStringOther(string &buffer){
```

Men hvordan skal vi så manipulere denne streng? Vi ved at brugeren indtaster udtryk af typen $ax_1 + bx_2 + \cdots$. Men vi skal jo kun bruge selve koefficienterne, så vi kan istedet benytte fortegnene samt x'erne som skilletegn. Derfor laver vi en søg-og-erstat på strengen.

Vi kan allerede forudse, hvilke sekvenser, vi skal søge efter: `+x`, `-x` og `x`. Implicitte ettaller skal jo indsættes.

Til at starte med tjekker vi så simpelthen det første tegn i strengen.

```
205     int t;
206     int len=buffer.length();
207     if(buffer[0]≡'x'){
208        buffer.insert(0,"1");
209        len++;
210     }
```

Vi kan nu indsætte de implicitte ettaller ved +x og -x. Tanken er at hvis vi finder fx −x skal vi indsætte et ettal mellem − og x. Ved den næste søgning er vi så nødt til at at springe to pladser frem i strengen med opdateringen t+2, for ellers vil denne nye søgning finde det samme minus eller plus som vi allerede havde behandlet, og det vil give en uendelig løkke. Stopkriterierne er, at vi skal holde os inden for strengens længde og $t \neq -1$ (se side 43).

```
211     t=buffer.find("+x");
212     while(t<len∧t≠−1){
213        buffer.insert(t+1, "1");
214        len++;
215        t=buffer.find("+x", t+2);
216     }
217     t=buffer.find("−x");
218     while(t<len∧t≠−1){
219        buffer.insert(t+1, "1");
220        len++;
221        t=buffer.find("−x", t+2);
222     }
```

Efter disse modifikationer har strengen følgende udseende:

```
1x1-1x2+3.14159x3
```

Nu er vi næsten færdige! Vi skal bare erstatte plusser, minusser og x'er med kommaer.

```
223     t=buffer.find ("−");
224     while(t<len∧t≠−1){
225        buffer.insert(t, ",");
226        len++;
227        t=buffer.find("−", t+2);
228     }
229     t=buffer.find ("+");
230     while(t<len∧t≠−1){
231        buffer.replace(t, 1, ",");
232        t=buffer.find("+", t+1);
```

```
233     }
234     t=buffer.find ("x");
235     while(t<len∧t≠−1){
236        buffer.replace(t, 1, ",");
237        t=buffer.find("x", t+1);
238     }
```

Endelig mangler vi bare at tjekke, om der stod et minus på første plads i strengen. Hvis der gjorde, blev der jo indsat et komma, som vi så skal fjerne igen.

```
239     if(buffer[0]≡',')
240        buffer.erase(0,1);
241     if(PrintInfo)
242        cout≪endl≪"Bufferen efter manipulation: "≪buffer;
243  }
```

Resultat:

```
1,1,-1,2,3.14159,3
```

Koefficienter og placering er nydeligt opdelt.

A.3.6 Konvertering af data

Når vi skal konvertere den indtastede strengs data, må vi sørge for, at de konverterede data bliver gemt på en "fornuftig" vis.

```
244  void ReadCoef(string &buffer, double Coef[][MaxCol], int
        &EquationNumber, int &NumberOfColumns){
```

Da strengen udelukkende består af tal og kommaer, kan – og vil – vi udnytte det på det groveste. Imidlertid vil vi gerne have, at `NULL` er med i strengen, hvilket den ikke er automatisk. Derfor findes først længden på strengen, som derefter anvendes, når strengen kopieres over i en **char**-pointer. Pointerens længde bliver nemlig en større end strengens længde, og dermed er der plads til `NULL`.

```
245     int len=buffer.length();
246     buffer[len]=',';
247     len++;
248     char *bufferptr=new char [len+1];
249     buffer.copy(bufferptr, len, 0);
250     bufferptr[len]='\0'; Der indsættes et |NULL| på sidste plads
```

Herefter vil vi gerne omdanne koefficienterne i buffer fra strenge til **double**. Her udnyttes, at koefficienterne står umiddelbart før hvert andet komma.

```
251     char *Tempptr=strtok(bufferptr, ",");
252     double TempCoef;
253     int TempCol, MaxTempCol=0;
254     int i=0;
255     while(Tempptr≠NULL){
256        if(i mod 2≡0) Vi udvælger hvert andet i (koefficienten til x).
257           TempCoef=atof(Tempptr);
258        if(i mod 2≡1){ Vi udvælger hvert andet i (placeringen i
              matricen).
259           TempCol=atoi(Tempptr);
260           if(TempCol>MaxTempCol)
261              MaxTempCol=TempCol; Gemmer den største værdi af
                    søjlenummeret.
262           Coef[EquationNumber][TempCol-1]=TempCoef; Gemmer
                 det hele i matricen.
263        }
264        i++;
265        Tempptr=strtok(NULL, ","); Vi springer videre til næste
              komma.
266     }
267     NumberOfColumns=MaxTempCol;
268     if(PrintInfo){
269        cout≪endl≪"i ="≪i≪"\t\t MaxTempCol ="≪MaxTempCol;
270        cout≪endl≪"Koefficienter : ";
271        for(int j=0; j<NumberOfColumns; j++)
272           cout≪endl≪Coef[EquationNumber][j]≪"\t j ="≪j≪"\
                 t Eqnum ="≪EquationNumber;
273     }
274  }
```

Efter indlæsningen af data fra strengen tjekker vi, om der var tale om et maksimerings- eller minimeringsproblem. I tilfælde af det sidste skifter vi fortegn på kriteriefunktionen, således at det nu alligevel er et maksimeringsproblem. Det samme gør sig gældende, hvis der er tale om bibetingelser af typen >.

```
275  void SwitchSigns(bool MaxProblem, bool CriterionFunction,
         bool GreaterThan, int EquationNumber, double
        Coef[][MaxCol], double ResultCoef[MaxRow]){
```

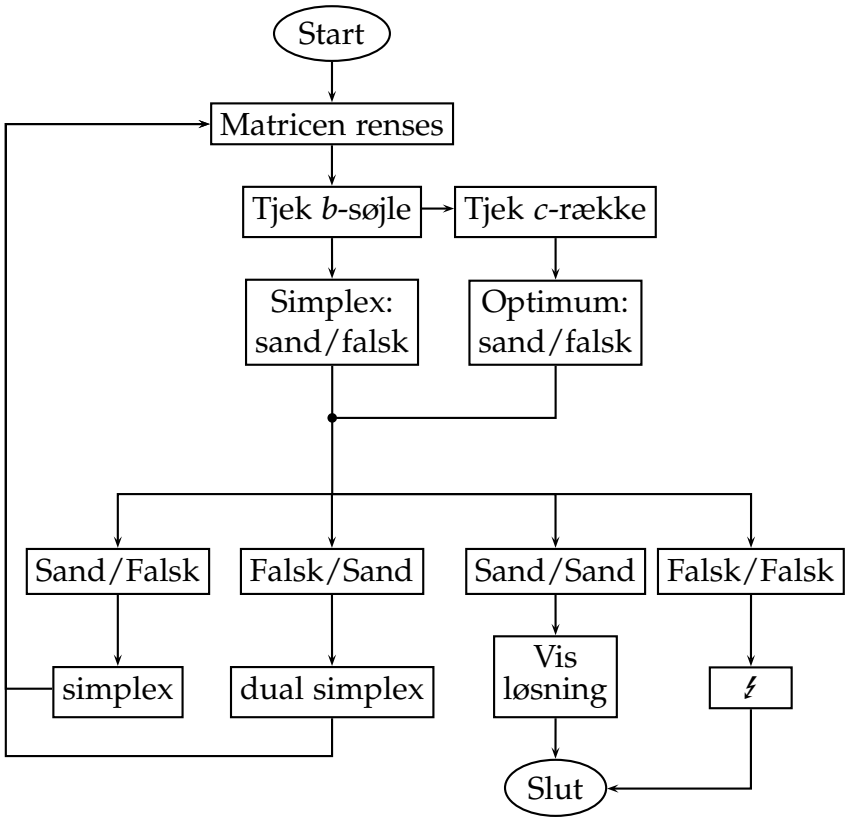
```
276     int i;
277     if(!MaxProblem)
278        if(CriterionFunction){
279           if(PrintInfo)
280              cout≪"Kriteriefunktion, min−problem.";
281           for(i=0;i<MaxCol;i++){
282              Coef[EquationNumber][i]=−
                    Coef[EquationNumber][i];
283              if(PrintInfo)
284                 cout≪endl≪"Coef[EquationNumber][i]="≪
                       Coef[EquationNumber][i];
285           }
286        }
287     if(GreaterThan){
288        for(i=0;i<MaxCol;i++){
289           Coef[EquationNumber][i]=−
                 Coef[EquationNumber][i];
290           if(PrintInfo)
291              cout≪endl≪"Coef[EquationNumber][i]="≪
                    Coef[EquationNumber][i];
292        }
293        ResultCoef[EquationNumber]=−
              ResultCoef[EquationNumber];
294        if(PrintInfo)
295           cout≪endl≪"ResultCoef[EquationNumber]="≪
                 ResultCoef[EquationNumber];
296     }
297  }
```

A.3.7 Slack-variable

Vi skal have kreeret en stribe slack-variable ud fra de informationer, bibetingelserne giver os. Det er sådan set meget nemt, idet vi blot skal sætte 1 eller 0 ind i diagonalelementet.

```
298  void CreateSlackVars(bool LessThan, bool GreaterThan, int
        EquationNumber, double SlackVars[][MaxCol]){
299        if(LessThan∨GreaterThan)
300           SlackVars[EquationNumber][EquationNumber−1]=1;
301  }
```

A.4 Simplex



Når vi skal køre den helt almindelige simplexalgoritme, skal vi som udgangspunkt sørge for, at alle koefficienterne i resultatsøjlen er ikkenegative. Er de det, kan vi gå videre med algoritmen, som inden selve Gauss-Jordan-eliminationen finder en startsøjle, hvori pivotelementet findes.

```
302 void Simplex(double InputMatrix[][MaxBigCol+1], int
       NumberOfRows, int NumberOfCols, double TrueResult[]){
303    int NumberOfIterations=-1;
304    do{
305       FoundValidSolution=false;
306       NumberOfIterations++;
307       CleanMatrix(InputMatrix, NumberOfRows, NumberOfCols
             );
308       CheckForSimplex(InputMatrix, NumberOfRows,
             NumberOfCols); Tjekker for dual-Simplex.
309       CheckForOptimum(InputMatrix, NumberOfRows,
             NumberOfCols); Tjekker for optimalitet.
```

```
310         if(PrintInfo){
311             cout≪endl≪"SingularSimplex ="≪boolalpha≪
                    SingularSimplex≪" FoundOptimum ="≪boolalpha
                    ≪FoundOptimum;
312             getch();
313         }
```

De to tjek resulterer i fire mulige resultater, som vi tager fat i. Først en simplex, hvor der ikke blev fundet et optimum.

```
314         if(SingularSimplex∧!FoundOptimum){ Vi har en normal
                (ikkeoptimal) simplex, som vi behandler.
315             if(PrintInfo)
316                 cout≪endl≪"Du har indtastet et Simplex−problem"
                        ;
317             FindMaxCoef(InputMatrix, NumberOfCols−1);     Først
                    findes startsøjlen...
318             if(PrintInfo)
319                 cout≪endl≪"MaxCoefPosition er: "≪
                        MaxCoefPosition;
320             if(MaxCoefPosition≡−1)
321                 FoundValidSolution=true;
322             if(!FoundValidSolution){
323                 FindSmallestFraction(InputMatrix,
                        NumberOfRows, NumberOfCols−1,
                        MaxCoefPosition); ... og dernæst startrækken.
324                 if(PrintInfo)
325                     cout≪endl≪"MinFractionPosition er: "≪
                            MinFractionPosition;
326                 if(MinFractionPosition≡−1)
327                     break;
328                 DivideRow(InputMatrix, MinFractionPosition,
                        MaxCoefPosition, NumberOfCols);      Rækken
                        divideres igennem.
329                 EliminateAboveAndBelow(InputMatrix,
                        MinFractionPosition, MaxCoefPosition,
                        NumberOfRows, NumberOfCols); Vi eliminerer oven
                        over og neden under vort initialettal.
330             }
331         }
```

Dernæst kommer det tilfælde, hvor det blev konstateret, at der ikke var tale om en almindelig simplex, men hvor der var et optimum. Derfor

er der tale om en dual simplex, hvor vi skal finde den mindste koefficient

```
332         if(!SingularSimplex∧FoundOptimum){
333            if(PrintInfo)
334               cout≪endl≪"Du har fat i et dualt simplexproblem!"
                     ;
335            FindMinCoef(InputMatrix, NumberOfRows,
                  NumberOfCols);
336            if(PrintInfo)
337               cout≪endl≪"MinCoefPosition er: "≪
                     MinCoefPosition;
338            if(MinCoefPosition≡-1)
339               FoundValidSolution=true;
340            if(!FoundValidSolution){
341               DualFindSmallestFraction(InputMatrix,
                     NumberOfCols, MinCoefPosition);
342               if(PrintInfo)
343                  cout≪endl≪"DualMinFractionPosition er: "≪
                        DualMinFractionPosition;
344               DivideRow(InputMatrix, MinCoefPosition,
                     DualMinFractionPosition, NumberOfCols);
345               EliminateAboveAndBelow(InputMatrix,
                     MinCoefPosition, DualMinFractionPosition,
                     NumberOfRows, NumberOfCols);
346            }
347         }
```

Tilbage er der de sidste to tilfælde, hvor det første betyder at vi har fundet en optimal løsning, mens den anden betyder, at løsningen hverken er gyldig eller optimal.

```
348         if(SingularSimplex∧FoundOptimum){
349            FoundValidSolution=true;
350            if(PrintInfo∨(Test_Mode∧NumberOfTestRuns≡
                  MaxTestRun-1)){
351               PresentSolution(InputMatrix, NumberOfRows,
                     NumberOfCols, TrueResult);
352               cout≪endl≪"Der er brugt "≪NumberOfIterations≪
                     " iterationer.";
353            }
354         }
355         if(!SingularSimplex∧!FoundOptimum){
```

```
356            cout≪endl≪"Ingen løsning!";
357            break;
358         }
359         if(PrintInfo){
360            PrintMatrix(InputMatrix, NumberOfRows,
                  NumberOfCols);
361            getch();
362         }
363      }
364      while(!FoundValidSolution);
365   }
```

A.4.1 Præsentation af løsning

Når simplex er færdig, skal vi have tolket resultaterne i matricen. Vi leder efter initialettaller, og udskriver løsningerne.

```
366   void PresentSolution(double InputMatrix[][MaxBigCol+1],
         int NumberOfRows, int NumberOfCols, double
         TrueResult[]){
367      int i, j, k, RowPosition;
368      double ResultVector[MaxBigCol];
369      int UsedRows[MaxRow]={0};
370      for(i=0;i<NumberOfCols-1;i++){
371         k=0;
372         for(j=0;j<NumberOfRows;j++){
373            if(fabs(InputMatrix[j][i])>0){
374                  k++;
375                  RowPosition=j;
376            }
377         }
378         if(k≡1)
379            if(UsedRows[RowPosition]≡0){
380               ResultVector[i]=
                     InputMatrix[RowPosition][NumberOfCols-1]/
                     InputMatrix[RowPosition][i];
381               UsedRows[RowPosition]=1;
382            }
383            else
384               ResultVector[i]=0;
385         else
386            ResultVector[i]=0;
```

```
387     }
388     cout≪endl≪"Løsningen er : "≪endl;
389     PrintVector(ResultVector,NumberOfCols−1);
```

Hvis vi kører i Test_Mode får vi en oversigt over afvigelserne fra det korrekte resultat.

```
390     if(Test_Mode∧NumberOfTestRuns≡MaxTestRun−1){
391        double Error[MaxBigCol];
392        VectorSubtraction(TrueResult, ResultVector,
              NumberOfCols−1, Error);
393        cout≪endl≪"Fejl: ";
394        PrintVector(Error, NumberOfCols−1);
395        double Max=fabs(Error[0]);
396        for(int i=1; i<NumberOfCols−1; i++)
397           if(fabs(Error[i])>Max)
398              Max=fabs(Error[i]);
399        cout≪endl≪"Den numerisk største fejl er: "≪Max;
400     }
401  }
```

A.4.2 Tjek af simplextableauet

Vi starter med at tjekke validiteten af simplextableauet. Det gør vi ved at sikre os, at alle værdierne (undtagen den første) i resultatsøjlen er ikkenegative.

```
402  bool CheckForSimplex(double InputMatrix[][MaxBigCol+1],
        int NumberOfRows, int NumberOfCols){
403     SingularSimplex=true;
404     if(PrintInfo)
405        cout≪endl≪"[NumberOfCols]"≪NumberOfCols;
406     for(int i=1; i<NumberOfRows; i++)
407        if(InputMatrix[i][NumberOfCols−1]<0){
408           SingularSimplex=false;
409           break; Vi kan lige så godt stoppe med det samme.
410        }
411     return SingularSimplex;
412  }
```

Dernæst udfører vi kontrollen af optimalitetsbetingelserne, nemlig at øverste række i tableauet kun indeholder ikkepositive værdier.

```
413 bool CheckForOptimum(double InputMatrix[][MaxBigCol+1],
        int NumberOfRows, int NumberOfCols){
414     FoundOptimum=true;
415     if(PrintInfo)
416         cout≪endl≪"[NumberOfRows]"≪NumberOfRows;
417     for(int i=0; i<NumberOfCols-1; i++)
418         if(InputMatrix[0][i]>0){
419             FoundOptimum=false;
420             break; Vi kan lige så godt stoppe med det samme.
421         }
422     return FoundOptimum;
423 }
```

A.4.3 Udsøgning af startsøjle og -række

Udsøgningen af startsøjlen er sådan set nem nok: Tjek tallene i øverste række igennem ét efter ét, og hold styr på, hvor det største er. Dog holder vi øje med, om tallene alle er ikkepositive, for hvis de er, har vi en løsning. I dette tilfælde returneres søjlenummer −1, da det er en entydig indikation af, at der ikke fandtes et positivt resultat.

```
424 int FindMaxCoef(double InputMatrix[][MaxBigCol+1], int
        NumberOfCols){
425     double Test=InputMatrix[0][0];
426     MaxCoefPosition=0;
427     for(int i=1; i<NumberOfCols; i++)
428         if(InputMatrix[0][i]>Test){ Gem den nye værdi, hvis den
              var større end den hidtidigt største.
429             Test=InputMatrix[0][i];
430             MaxCoefPosition=i;
431         }
```

Her kommer så en fiks test: Hvis der ikke fandtes et ikkenegativt tal i rækken, returneres −1 – som jo er en ugyldig position i matricen – ellers det fundne søjlenummer.

```
432     if(Test≤0)
433         return MaxCoefPosition=-1;
434     else
435         return MaxCoefPosition; Returnerer søjlenummeret.
436 }
```

På samme vis – i al fald hypotetisk – foregår det også, når vi skal bestemme den mindste brøk, der opstår, når vi dividerer elementer fra samme række i startsøjlen og resultatsøjlen.

Ikke desto mindre opstår der her et problem, fordi vi skal holde styr på flere ting på én gang:

1. Sørg for, at nævneren er positiv.

2. Test om brøken `FracValue` er mindre end nogle af de andre brøker i søjlen.

Den første test er triviel, mens den anden kan gribes an på flere måder.

Det er ikke fordi, det er svært at sammenligne to af brøkerne, men vi skal være sikre på, at vi får fundet mindst én værdig kandidat, hvilket vi *ikke* kan være sikre på på forhånd. Derfor må vi nøjes med at lede efter en værdi til at starte med, og først når vi har fundet en værdig kandidat, kan vi bruge den til at sammenligne andre eventuelle andre kandidater med. Til det formål definerer vi den booleske operator `FirstRun`, der sørger for netop dette.

Så kan vi køre rækkerne igennem én efter én, og sluttelig returnere rækkenummeret.

```
437 int FindSmallestFraction(double InputMatrix[][MaxBigCol+1
       ], int NumberOfRows, int NumberOfCols, int
       MaxCoefPosition){
438    double Test;
439    double FracValue;
440    bool FirstRun=true;
441    MinFractionPosition=-1;
442    for(int i=1; i<NumberOfRows; i++){
443       if(InputMatrix[i][MaxCoefPosition]>0∧
             InputMatrix[i][NumberOfCols]≥0){
444          FracValue=InputMatrix[i][NumberOfCols]/
                InputMatrix[i][MaxCoefPosition];
445          if(FirstRun){
446             Test=FracValue;
447             MinFractionPosition=i;
448             FirstRun=false;
449          }
450          if(FracValue<Test){
451             Test=FracValue;
452             MinFractionPosition=i;
453          }
```

```
454         }
455     }
456     return MinFractionPosition;
457 }
```

A.4.4 Division og elimination af rækker

Vi skal genneføre en Gauss-Jordan-elimination af simplextableauet, og denne deler vi op i to trin.

Først dividerer vi den fundne række igennem omkring pivotelementet.

```
458 void DivideRow(double InputMatrix[][MaxBigCol+1], int
        MinFracPosition, int MaxCoefPosition, int NumberOfCols
        ){
459     for(int i=0; i<NumberOfCols; i++)
460         if(i≠MaxCoefPosition)
461             InputMatrix[MinFracPosition][i]/=
462               InputMatrix[MinFracPosition][MaxCoefPosition];
463     InputMatrix[MinFracPosition][MaxCoefPosition]=1; Undgår
            afrundingsfejl.
464 }
```

Dernæst skal vi have elimineret tallene over og under pivotelementet.

```
465 void EliminateAboveAndBelow(double
        InputMatrix[][MaxBigCol+1], int MinFracPosition, int
        MaxCoefPosition, int NumberOfRows, int NumberOfCols){
466     int i, j;
467     for(i=0; i<NumberOfRows; i++)
468         if(i≠MinFracPosition){
469             for(j=0; j<NumberOfCols; j++)
470                 if(j≠MaxCoefPosition)
471                     InputMatrix[i][j]-=
                            InputMatrix[i][MaxCoefPosition]*
                            InputMatrix[MinFracPosition][j];
472             InputMatrix[i][MaxCoefPosition]=0;
473
474         }
475 }
```

A.4.5 Dual simplex

Til dual simplex skal vi stort set gentage alt hvad vi gjorde ved normal simplex. Dog er der den detalje, at vi leder efter den mindste koefficient i en søjle i stedet for den største koefficient i en række.

```
476 int FindMinCoef(double InputMatrix[][MaxBigCol+1], int
       NumberOfRows, int NumberOfCols){
477    double Test=InputMatrix[1][NumberOfCols-1];
478    cout<<endl<<"Test="<<Test;
479    MinCoefPosition=1;
480    for(int i=2; i<NumberOfRows; i++)
481       if(InputMatrix[i][NumberOfCols-1]<Test){ Gem den nye
             værdi, hvis den var mindre end den hidtidigt mindste.
482          Test=InputMatrix[i][NumberOfCols-1];
483          MinCoefPosition=i;
484       }
485    return MinCoefPosition; Returnerer søjlenummeret.
486 }
```

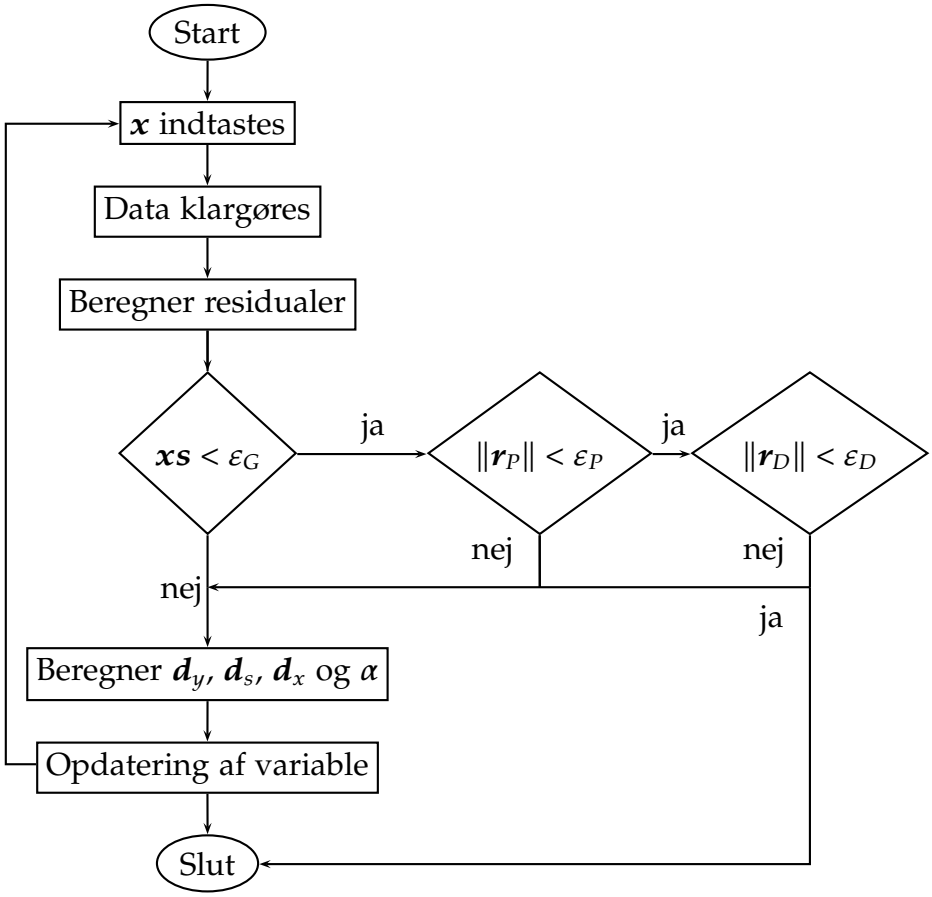
Tilsvarende skal vi også finde den mindste brøk.

```
487 int DualFindSmallestFraction(double
       InputMatrix[][MaxBigCol+1], int NumberOfCols, int
       MinCoefPosition){
488    double Test;
489    double FracValue;
490    bool FirstRun=true;
491    DualMinFractionPosition=-1;
492    for(int i=0; i<NumberOfCols-1; i++){
493       if(InputMatrix[0][i]<0∧
             InputMatrix[MinCoefPosition][i]<0){
494          FracValue=InputMatrix[0][i]/
                InputMatrix[MinCoefPosition][i];
495          if(FirstRun){
496             Test=FracValue;
497             DualMinFractionPosition=i;
498             FirstRun=false;
499          }
500          if(FracValue<Test){
501             Test=FracValue;
502             DualMinFractionPosition=i;
503          }
504       }
```

```
505     }
506     return DualMinFractionPosition;
507 }
```

Det er faktisk alt, der er krævet for at få (dual) simplex til at fungere.

A.5 Interior Point



Indre punkters metode er en markant anderledes algoritme end simplex, selv om den løser samme type problemer. Som udgangspunkt skal vi bruge en matrix indeholdende koefficienter og slack-variable til et minimeringsproblem, så vi skal have omformet vore indtastede data til en form, der passer bedre til vort formål.

A.5.1 Forberedelse af Interior Point

Vi skal først gennemgå vore data, idet vi skal have visse ting med i beregningsgangene:

- Der skal være tale om et minimeringsproblem.
- Kriteriefunktionen skal pilles ud.

Så det starter vi med.

```
508 void PrepareIP(bool MaxProblem, double
        CoefAndSlack[][MaxBigCol], double ResultCoef[], int
        &RowsInCoefAndSlack, int &ColsInCoefAndSlack, double
        CritFunc[]){
509     RowsInCoefAndSlack--;
510     ColsInCoefAndSlack--;
511     if(NumberOfTestRuns<1){
512        for(int i=0; i≤ColsInCoefAndSlack; i++){
513           CritFunc[i]=CoefAndSlack[0][i]; Gemmer kriteriefunk-
                tionen.
514           CoefAndSlack[0][i]=
                CoefAndSlack[RowsInCoefAndSlack][i]; Flytter den
                nederste bibetingelse op i øverste række.
515           CoefAndSlack[RowsInCoefAndSlack][i]=CritFunc[i];
516        }
517        double Temp=ResultCoef[0];
518        ResultCoef[0]=ResultCoef[RowsInCoefAndSlack];
              Tilsvarende for resultatsøjlen.
519        ResultCoef[RowsInCoefAndSlack]=Temp;
520     }
521     if(!MaxProblem∧NumberOfTestRuns<1)
522        for(int i=0; i≤ColsInCoefAndSlack; i++)
523           CritFunc[i]=-CritFunc[i]; Negerer kriteriefunktionen
                om nødvendigt.
524 }
```

Til testkørslerne skal vi bruge en lidt anden type.

```
525 void PrepareIP_Test(int NumberOfColumns, double x[],
        double s[]){
526     int i;
527     for(i=0; i<NumberOfColumns+1; i++)
528        x[i]=1;
529     for(i=NumberOfColumns+1; i<2*(NumberOfColumns+1); i++
           )
530        x[i]=NumberOfColumns;
531     for(i=0; i<2*(NumberOfColumns+1); i++)
532        s[i]=1;
```

```
533 }
```

A.5.2 Interior point-algoritmen

Nu er vi så klar til at køre selve algoritmen. Med hensyn til betegnelserne af de enkelte matricer og vektorer, har vi holdt os til de symbolske navne, de fik i den strengt matematiske gennemgang af algoritmen.

Indlæsningen af den mulige løsning $\boldsymbol{x}^0$ foregår ved at brugeren indtaster en række kommaseparerede tal som fx `1,2,1,.5`.

```
534 void ReadVector(double x[], int NumberOfElements){
535    string Temp;
536    cout≪endl≪"Indtast en mulig løsning til x: ";
537    getline(cin,Temp);
538    cout≪Temp;
539    CleanUpStringSpaces(Temp);
540    cout≪Temp;
541    int len=Temp.length();
542    char *TempPtr=new char [len+1];
543    Temp.copy(TempPtr,len,0);
544    TempPtr[len]='\0';
545    char *TempPtr2=strtok(TempPtr, ",");
546    cout≪"TempPtr2: "≪TempPtr2;
547    int i=0;
548    while(TempPtr2≠NULL){
549       x[i]=atof(TempPtr2);
550       cout≪"\t TempPtr2: "≪atof(TempPtr2);
551       i++;
552       TempPtr2=strtok(NULL, ",");
553    }
554    PrintVector(x,NumberOfElements);
555 }

556 void InteriorPoint(double CoefAndSlack[][MaxBigCol],
       double ResultCoef[], double CritFunc[], int
       RowsInCoefAndSlack, int ColsInCoefAndSlack, double
       TrueResult[]){
```

Først initialiseres vores startvektorer til

$$(\boldsymbol{y}^0, \boldsymbol{s}^0) = (\boldsymbol{0}, \boldsymbol{e}),$$

mens vi kreerer $\boldsymbol{d}_y$, $\boldsymbol{d}_s$ og $\boldsymbol{d}_x$ til senere brug. Brugeren skal selv indtaste en mulig løsning i $\boldsymbol{x}$.

```
557     int NumberOfIterations=0;
558     double x[MaxBigCol];
559     if(!Test_Mode)
560       ReadVector(x, ColsInCoefAndSlack);
561     double s[MaxBigCol];
562     for(int i=0; i<ColsInCoefAndSlack; i++)
563        s[i]=1.;
564     if(Test_Mode)
565        PrepareIP_Test(RowsInCoefAndSlack-1, x, s);
566     double y[MaxRow]={0};
567     double dy[MaxRow];
568     double ds[MaxRow];
569     double dx[MaxRow];
```

Vi skal på et tidspunkt i algoritmen løse en større matrixligning, så vi skal bruge nogle matricer til det formål.

```
570     double LeftResult[MaxRow][MaxRow];
571     double LeftResult_Inverse[MaxRow][MaxRow];
572     double S[MaxBigCol][MaxBigCol];
573     double S_Inverse[MaxBigCol][MaxBigCol];
574     double X[MaxBigCol][MaxBigCol];
```

Tilsvarende skal vi også bruge nogle vektorer bl.a. til $\boldsymbol{r}_D$ og $\boldsymbol{r}_P$.

```
575     double RightResult[MaxRow];
576     double rP[MaxBigCol];
577     double rD[MaxBigCol];
578     double TempVector[MaxBigCol];
579     double Result;
580     double mu;
581     double xs;
582     bool Continue=true; Til løkken omkring algoritmen.
```

Inden vi går i gang beregner vi lige A^T, og det kan vi lige så godt gøre inden løkken, da det er den samme matrix, der skal bruges hele vejen igennem.

```
583     double CoefAndSlack_T[MaxBigCol][MaxBigCol];
584     TransposeMatrix(CoefAndSlack, RowsInCoefAndSlack,
           ColsInCoefAndSlack, CoefAndSlack_T);
585     if(PrintInfo){
586        cout<<endl<<"CoefAndSlack";
587        PrintMatrix(CoefAndSlack, RowsInCoefAndSlack,
              ColsInCoefAndSlack);
```

```
588         getch();
589         cout≪endl≪"CoefAndSlack_T";
590         PrintMatrix(CoefAndSlack_T, ColsInCoefAndSlack,
                RowsInCoefAndSlack);
591         getch();
592     }
```

Så kører vi. Først beregnes $\boldsymbol{r}_D^k$ og $\boldsymbol{r}_P^k$.

```
593     do{
594         Calculate_rD(CritFunc, CoefAndSlack_T, y, s,
                RowsInCoefAndSlack, ColsInCoefAndSlack, rD);
595         Calculate_rP(ResultCoef, CoefAndSlack, x,
                RowsInCoefAndSlack, ColsInCoefAndSlack, rP);
596         if(PrintInfo){
597             cout≪endl≪"rP:";
598             PrintVector(rP, RowsInCoefAndSlack);
599             getch();
600             cout≪endl≪"rD:";
601             PrintVector(rD, ColsInCoefAndSlack);
602             getch();
603         }
604         VectorProduct(x, s, ColsInCoefAndSlack, xs); μ bereg-
                nes som (x^k)^T s^k/n.
605         mu=xs/ColsInCoefAndSlack;
606         if(PrintInfo)
607             cout≪endl≪"x∗s er: "≪xs≪"\t mu er: "≪mu;
```

Nu skal vi afgøre om vi skal fortsætte algoritmen, så vi tester stopkriteriet.

```
608         if(fabs(xs)<Tolerance_xs){
609             if(PrintInfo)
610                 cout≪"\t xs er mindre end tolerancen";
611             if(fabs(FindNorm(rP, RowsInCoefAndSlack))<
                    Tolerance_rP){
612                 if(PrintInfo)
613                     cout≪"\t rP er mindre end tolerancen";
614                 if(fabs(FindNorm(rD, ColsInCoefAndSlack))<
                        Tolerance_rD){
615                     if(PrintInfo)
616                         cout≪"\t rD er mindre end tolerancen";
617                     Continue=false;
618                 }
```

```
619             }
620         }
```

Algoritmen starter med at vi konstruerer diagonalmatricerne X^k, S^k og $(S^k)^{-1}$.

```
621         if(Continue){
622             CreateDiagonalMatrix(x, ColsInCoefAndSlack, X);
623             CreateDiagonalMatrix(s, ColsInCoefAndSlack, S);
624             InverseDiagonalMatrix(S, ColsInCoefAndSlack,
                    S_Inverse);
```

Vi ved, at matrixligningen får udseendet

$$L\boldsymbol{d}_y = \boldsymbol{R} \iff \boldsymbol{d}_y = L^{-1}\boldsymbol{R},$$

hvor

$$L = AX^k(S^k)^{-1}A^T \quad \text{og} \quad \boldsymbol{R} = \boldsymbol{b} + A(S^k)^{-1}(X^k\boldsymbol{r}_D^k - \gamma\mu^k\boldsymbol{e}).$$

Vi udfører beregningerne.

```
625             LeftSide(CoefAndSlack, X, S_Inverse,
                    CoefAndSlack_T, RowsInCoefAndSlack,
                    ColsInCoefAndSlack, LeftResult);
626             CleanMatrix(LeftResult, RowsInCoefAndSlack,
                    ColsInCoefAndSlack);
627             InvertMatrix(LeftResult, RowsInCoefAndSlack,
                    LeftResult_Inverse);
628             RightSide(ResultCoef, CoefAndSlack, S_Inverse,
                    X, rD, mu, RowsInCoefAndSlack,
                    ColsInCoefAndSlack, RightResult);
629             CleanVector(RightResult, RowsInCoefAndSlack);
630             if(PrintInfo){
631                 cout≪"LeftSide udført";
632                 PrintMatrix(LeftResult, RowsInCoefAndSlack,
                        RowsInCoefAndSlack);
633                 cout≪"InvertMatrix udført";
634                 PrintMatrix(LeftResult_Inverse,
                        RowsInCoefAndSlack, RowsInCoefAndSlack);
635                 cout≪"RightSide udført";
636                 PrintVector(RightResult, RowsInCoefAndSlack);
637             }
638             Small_MatrixVectorProduct(LeftResult_Inverse,
                    RightResult, RowsInCoefAndSlack,
                    RowsInCoefAndSlack, dy); // d_y = L^{-1}R.
```

```
639         if(PrintInfo){
640            cout≪endl≪"dy: ";
641            PrintVector(dy, RowsInCoefAndSlack);
642         }
```

Vi beregner $\boldsymbol{d}_s^k$ som $\boldsymbol{r}_D^k - A^T\boldsymbol{d}_y^k$.

```
643         MatrixVectorProduct(CoefAndSlack_T, dy,
               ColsInCoefAndSlack, RowsInCoefAndSlack,
               TempVector);
644         VectorSubtraction(rD, TempVector,
               ColsInCoefAndSlack, ds);
645         if(PrintInfo){
646            cout≪endl≪"ds: ";
647            PrintVector(ds, ColsInCoefAndSlack);
648         }
```

Vi beregner $\boldsymbol{d}_x^k$ som $-\boldsymbol{x}^k + (S^k)^{-1}(\gamma\mu^k\boldsymbol{e} - X^k\boldsymbol{d}_s^k)$.

```
649         Calculate_dx(x, S_Inverse, mu, X, ds,
               ColsInCoefAndSlack, dx);
650         if(PrintInfo){
651            cout≪endl≪"dx: ";
652            PrintVector(dx, ColsInCoefAndSlack);
653         }
```

Vi skal nu bestemme α som beskrevet tidligere.

```
654         double alpha=FindSmallestNumber(s, ds, x, dx,
               ColsInCoefAndSlack)*(1-gamma);
655         if(alpha>1)
656            alpha=1;
657         if(PrintInfo)
658            cout≪endl≪"alpha ="≪alpha;
```

Vi kan nu endelig foretage udregningerne af $\boldsymbol{x}^{k+1}$, $\boldsymbol{y}^{k+1}$ og $\boldsymbol{s}^{k+1}$.

```
659         MultiplyVector(dx, alpha, ColsInCoefAndSlack);
660         VectorAddition(x, dx, ColsInCoefAndSlack, x);
661         MultiplyVector(ds, alpha, ColsInCoefAndSlack);
662         VectorAddition(s, ds, ColsInCoefAndSlack, s);
663         MultiplyVector(dy, alpha, RowsInCoefAndSlack);
664         VectorAddition(y, dy, RowsInCoefAndSlack, y);
665         if(PrintInfo){
666            cout≪endl≪"x: ";
667            PrintVector(x, ColsInCoefAndSlack);
```

```
668             cout≪endl≪"s: ";
669             PrintVector(s, ColsInCoefAndSlack);
670             cout≪endl≪"y: ";
671             PrintVector(y, RowsInCoefAndSlack);
672             getch();
673          }
674       }
675       NumberOfIterations++;
676    }
677    while(Continue∧NumberOfIterations<1000); Vi fortsætter så
          længe stopkriteriet tillader det.
678    if(Test_Mode∧NumberOfTestRuns≡MaxTestRun−1){
679       double Error[MaxBigCol];
680       cout≪endl≪"x: ";
681       PrintVector(x, ColsInCoefAndSlack);
682       VectorSubtraction(TrueResult, x,
             ColsInCoefAndSlack, Error);
683       cout≪endl≪"Fejl: ";
684       PrintVector(Error, ColsInCoefAndSlack);
685       double Max=fabs(Error[0]);
686       for(int i=1; i<ColsInCoefAndSlack; i++){
687          if(fabs(Error[i])>Max)
688             Max=fabs(Error[i]);
689       }
690       cout≪endl≪"Den numerisk største fejl er: "≪Max;
691       cout≪endl≪"Der er brugt "≪NumberOfIterations≪"
             iterationer.";
692    }
693 }
```

A.5.3 Funktioner i Interior Point

Her kører beregningsgangene i Interior Point-algoritmen som beskrevet ovenfor. Der er ikke noget nyt i disse, da der blot er tale om matrix- og vektorprodukter.

Først $\boldsymbol{r}_P$.

```
694 void Calculate_rP(double ResultCoef[MaxBigCol], double
       CoefAndSlack[][MaxBigCol], double x[MaxBigCol], int
       RowsInCoefAndSlack, int ColsInCoefAndSlack, double
       rP[MaxBigCol]){
```

```
695     double TempVector[MaxBigCol];
696     MatrixVectorProduct(CoefAndSlack, x,
            RowsInCoefAndSlack, ColsInCoefAndSlack, TempVector)
            ;
697     VectorSubtraction(ResultCoef, TempVector,
            RowsInCoefAndSlack, rP);
698 }
```

Så $\boldsymbol{r}_D$.

```
699 void Calculate_rD(double CritFunc[MaxBigCol], double
        CoefAndSlack_T[][MaxBigCol], double y[MaxRow], double
        s[MaxBigCol], int RowsInCoefAndSlack, int
        ColsInCoefAndSlack, double rD[MaxBigCol]){
700     double TempVector[MaxBigCol];
701     MatrixVectorProduct(CoefAndSlack_T, y,
            ColsInCoefAndSlack, RowsInCoefAndSlack, TempVector)
            ;
702     VectorSubtraction(CritFunc, TempVector,
            ColsInCoefAndSlack, TempVector);
703     VectorSubtraction(TempVector, s, ColsInCoefAndSlack,
            rD);
704 }
```

Venstresiden L.

```
705 void LeftSide(double CoefAndSlack[][MaxBigCol], double
        X[][MaxBigCol], double S_Inverse[][MaxBigCol], double
        CoefAndSlack_T[][MaxBigCol], int RowsInCoefAndSlack,
        int ColsInCoefAndSlack, double LeftResult[][MaxRow]){
706     double TempMatrix[MaxRow][MaxBigCol];
707     double TempMatrix2[MaxRow][MaxBigCol];
708     MatrixProduct(CoefAndSlack, X, RowsInCoefAndSlack,
            ColsInCoefAndSlack, ColsInCoefAndSlack, TempMatrix)
            ;
709     MatrixProduct(TempMatrix, S_Inverse,
            RowsInCoefAndSlack, ColsInCoefAndSlack,
            ColsInCoefAndSlack, TempMatrix2);
710     Small_MatrixProduct(TempMatrix2, CoefAndSlack_T,
            RowsInCoefAndSlack, ColsInCoefAndSlack,
            RowsInCoefAndSlack, LeftResult);
711 }
```

Højresiden $\boldsymbol{R}$ beregnes også lige ud ad landevejen.

```
712 void RightSide(double ResultCoef[], double
       CoefAndSlack[][MaxBigCol], double
       S_Inverse[][MaxBigCol], double X[][MaxBigCol], double
       rD[MaxBigCol], double mu, int RowsInCoefAndSlack, int
       ColsInCoefAndSlack, double RightResult[MaxRow]){
713    double TempVector[MaxBigCol];
714    double TempVector2[MaxRow];
715    double TempMatrix[MaxRow][MaxRow+MaxRow];
716    MatrixVectorProduct(X, rD, ColsInCoefAndSlack,
          ColsInCoefAndSlack, TempVector);
717    for(int i=0; i<ColsInCoefAndSlack; i++)
718       TempVector[i]-=gamma*mu;
719    MatrixProduct(CoefAndSlack, S_Inverse,
          RowsInCoefAndSlack, ColsInCoefAndSlack,
          ColsInCoefAndSlack, TempMatrix);
720    MatrixVectorProduct(TempMatrix, TempVector,
          RowsInCoefAndSlack, ColsInCoefAndSlack, TempVector2
          );
721    VectorAddition(ResultCoef, TempVector2,
          RowsInCoefAndSlack, RightResult);
722 }
```

Beregningen af $\boldsymbol{d}_x^k$ er også ligetil.

```
723 void Calculate_dx(double x[], double
       S_Inverse[][MaxBigCol], double mu, double
       X[][MaxBigCol], double ds[], int ColsInCoefAndSlack,
       double dx[MaxBigCol]){
724    double TempVector[MaxBigCol];
725    double TempVector2[MaxBigCol];
726    MatrixVectorProduct(X, ds, ColsInCoefAndSlack,
          ColsInCoefAndSlack, TempVector);
727    for(int i=0; i<ColsInCoefAndSlack; i++)
728       TempVector[i]=gamma*mu-TempVector[i];
729    MatrixVectorProduct(S_Inverse, TempVector,
          ColsInCoefAndSlack, ColsInCoefAndSlack, TempVector2
          );
730    VectorSubtraction(TempVector2, x, ColsInCoefAndSlack,
          dx);
731 }
```

A.5.3.1 Invertering af matricer

For at beregne L^{-1} tager vi igen fat i Gauss-Jordan. Vi erindrer, at for at invertere matricen L skal vi køre Gauss-Jordan-elimination på matricen

$$(L \quad I)$$

hvorved vi opnår matricen

$$(I \quad L^{-1})$$

```
732 void InvertMatrix(double LeftResult[][MaxRow], int
       RowsInCoefAndSlack, double
       LeftResult_Inverse[][MaxRow]){
733    double TempMatrix[MaxRow][MaxRow+MaxRow];
```

Vi starter med at sætte en enhedsmatrix på L og kører så Gauss-Jordan.

```
734    AppendIdentityMatrix(LeftResult, RowsInCoefAndSlack,
          TempMatrix);
735    for(int i=0; i<RowsInCoefAndSlack; i++){
736       FindZero(TempMatrix, i, RowsInCoefAndSlack);
737       DivideRow(TempMatrix, i, i,2*RowsInCoefAndSlack);
738       EliminateAboveAndBelow(TempMatrix, i, i,
             RowsInCoefAndSlack, 2*RowsInCoefAndSlack);
739    }
```

Vi afslutter med at splitte matricen op igen.

```
740    SplitMatrix(TempMatrix, RowsInCoefAndSlack,
          LeftResult_Inverse);
741 }
```

Ved en invertering skal vi som nævnt have tilføjet en enhedmatrix til den oprindelige.

```
742 void AppendIdentityMatrix(double
       SmallInputMatrix[][MaxRow], int NumberOfRows, double
       BigOutputMatrix[][MaxRow+MaxRow]){
743    for(int i=0; i<NumberOfRows; i++)
744       for(int j=0; j<NumberOfRows; j++){
745          BigOutputMatrix[i][j]=SmallInputMatrix[i][j];
746          if(i≡j)
747             BigOutputMatrix[i][j+NumberOfRows]=1;
748          else
749             BigOutputMatrix[i][j+NumberOfRows]=0;
750    }
751 }
```

Efter fuldendt Gauss-Jordan skal vi så splitte matricen op, så vi får den inverterede ud igen.

```
752 void SplitMatrix(double InputMatrix[][MaxBigCol], int
       NumberOfRows, double OutputMatrix[][MaxRow]){
753    for(int i=0; i<NumberOfRows; i++)
754       for(int j=0; j<NumberOfRows; j++)
755          OutputMatrix[i][j]=InputMatrix[i][NumberOfRows+
                j];
756 }
```

I udvælgelses af α skal vi som nævnt tidligere finde det mindste tal mindre end 1 mellem

$$\alpha_P^{\max} = \min_j\{-\boldsymbol{x}_j/(\boldsymbol{d}_x)_j \mid (\boldsymbol{d}_x)_j < 0\} \qquad \text{og}$$
$$\alpha_D^{\max} = \min_j\{-\boldsymbol{s}_j/(\boldsymbol{d}_s)_j \mid (\boldsymbol{d}_s)_j < 0\}$$

Dermed er strategien ret klar: Løb vektorerne $\boldsymbol{d}_x$ og $\boldsymbol{d}_s$ igennem, undersøg om et element er mindre end nul og regn brøken ud. Hvis den er mindre end den hidtidigt største brøk, så gemmes værdien. Sluttelig returneres den mindste af de fundne værdier.

```
757 double FindSmallestNumber(double VectorOne[MaxBigCol],
       double VectorOne2[MaxBigCol], double
       VectorTwo[MaxBigCol], double VectorTwo2[MaxBigCol],
       int NumberOfRows){
758    SmallestNumber=pow(10,308); Et absurd højt tal, som vi aldrig vil
          støde på.
759    for(int i=0; i<NumberOfRows; i++){
760       if(VectorOne2[i]<0)
761          if(-VectorOne[i]/VectorOne2[i]<SmallestNumber)
762             SmallestNumber=-VectorOne[i]/VectorOne2[i];
763       if(VectorTwo2[i]<0)
764          if(-VectorTwo[i]/VectorTwo2[i]<SmallestNumber)
765             SmallestNumber=-VectorTwo[i]/VectorTwo2[i];
766    }
767    return SmallestNumber;
768 }
```

Ganger et tal på en vektor.

```
769 void MultiplyVector(double TempVector[MaxBigCol], double
       Factor, int NumberOfRows){
770    for(int i=0; i<NumberOfRows; i++)
```

```
771         TempVector[i]*=Factor;
772  }
```

Transponerer en matrix.

```
773  void TransposeMatrix(double CoefAndSlack[][MaxBigCol],
        int RowsInCoefAndSlack, int ColsInCoefAndSlack, double
         CoefAndSlack_T[][MaxBigCol]){
774     for(int i=0; i<RowsInCoefAndSlack; i++)
775        for(int j=0; j<ColsInCoefAndSlack; j++)
776           CoefAndSlack_T[j][i]=CoefAndSlack[i][j];
777  }
```

Beregner et skalarprodukt.

```
778  void VectorProduct(double b[MaxBigCol], double
        c[MaxBigCol], int NumberOfRowsInC, double &Result){
779     Result=0;
780     for(int i=0; i<NumberOfRowsInC; i++){
781        Result+=b[i]*c[i];
782     }
783  }
```

Beregner et matrixprodukt.

```
784  void MatrixProduct(double FirstMatrix[][MaxBigCol],
        double SecondMatrix[][MaxBigCol], int
        ColsInFirstMatrix, int RowsInFirstMatrix, int
        ColsInSecondMatrix, double ResultMatrix[][MaxBigCol]){
785     for(int i=0; i<ColsInFirstMatrix; i++)
786        for(int j=0; j<ColsInSecondMatrix; j++){
787           ResultMatrix[i][j]=0;
788           for(int k=0; k<RowsInFirstMatrix; k++)
789              ResultMatrix[i][j]+=FirstMatrix[i][k]*
                    SecondMatrix[k][j];
790        }
791  }
```

Den samme funktion som ovenstående, men hvor søjleantallet i resultatmatricen er noget mindre end ellers.

```
792  void Small_MatrixProduct(double FirstMatrix[][MaxBigCol],
         double SecondMatrix[][MaxBigCol], int
        ColsInFirstMatrix, int RowsInFirstMatrix, int
        ColsInSecondMatrix, double ResultMatrix[][MaxRow]){
793     for(int i=0; i<ColsInFirstMatrix; i++)
```

```
794         for(int j=0; j<ColsInSecondMatrix; j++){
795            ResultMatrix[i][j]=0;
796            for(int k=0; k<RowsInFirstMatrix; k++)
797               ResultMatrix[i][j]+=FirstMatrix[i][k]*
                     SecondMatrix[k][j];
798         }
799  }
```

Produktet mellem en matrix og en vektor. Ud kommer en vektor.

```
800  void MatrixVectorProduct(double InputMatrix[][MaxBigCol],
        double InputVector[], int RowsInInputMatrix, int
       ColsInInputMatrix, double ResultVector[]){
801     for(int i=0; i<RowsInInputMatrix; i++){
802        ResultVector[i]=0;
803        for(int j=0; j<ColsInInputMatrix; j++)
804           ResultVector[i]+=InputMatrix[i][j]*
                 InputVector[j];
805     }
806  }
```

Det samme som ovenstående, men med en anden søjlestørrelse af matricen.

```
807  void Small_MatrixVectorProduct(double
       InputMatrix[][MaxRow], double InputVector[], int
       RowsInInputMatrix, int ColsInInputMatrix, double
       ResultVector[]){
808     for(int i=0; i<RowsInInputMatrix; i++){
809        ResultVector[i]=0;
810        for(int j=0; j<ColsInInputMatrix; j++)
811           ResultVector[i]+=InputMatrix[i][j]*
                 InputVector[j];
812     }
813  }
```

Vi laver en diagonalmatrix ($\boldsymbol{s}^k \rightarrow S^k$ og $\boldsymbol{x}^k \rightarrow X^k$).

```
814  void CreateDiagonalMatrix(double InputVector[], int
       RowsInInputVector, double DiagonalMatrix[][MaxBigCol])
       {
815     for(int i=0; i<RowsInInputVector; i++)
816        for(int j=0; j<RowsInInputVector; j++){
817           if(i≡j)
```

```
818             DiagonalMatrix[i][j]=InputVector[i]; Sætter dia-
                    gonalelementet.
819          else
820             DiagonalMatrix[i][j]=0; Alle andre pladser sættes til
                    0.
821       }
822 }
```

Subtrahering af to vektorer.

```
823 void VectorSubtraction(double FirstVector[], double
       SecondVector[], int RowsInVectors, double
       ResultVector[]){
824    for(int i=0; i<RowsInVectors; i++)
825       ResultVector[i]=FirstVector[i]-SecondVector[i];
826 }
```

Addition af to vektorer.

```
827 void VectorAddition(double FirstVector[], double
       SecondVector[], int RowsInVectors, double
       ResultVector[]){
828    for(int i=0; i<RowsInVectors; i++)
829       ResultVector[i]=FirstVector[i]+SecondVector[i];
830 }
```

Den inverse til en diagonalmatrix er let at regne ud, idet der som diagonalelementer i den nye matrix blot skal stå den reciprokke værdi af de oprindelige diagonalelementer.

```
831 void InverseDiagonalMatrix(double
       InputMatrix[][MaxBigCol], int RowsInInputMatrix,
       double InputMatrix_Inverse[][MaxBigCol]){
832    for(int i=0; i<RowsInInputMatrix; i++)
833       for(int j=0; j<RowsInInputMatrix; j++){
834          if(i≡j)
835             InputMatrix_Inverse[i][j]=1./
                    InputMatrix[i][j];
836          else
837             InputMatrix_Inverse[i][j]=InputMatrix[i][j];
838       }
839 }
```

A.5.4 Gauss-Jordan-funktioner

En Gauss-Jordan-elimination består af tre trin:

1. Der foretages rækkeoperationer således at der i den betragtede søjle findes et diagonalelement forskelligt fra nul.
2. Divider rækken igennem, således at der fremkommer et initialetal.
3. Eliminer ovenover og nedenunder.

Vi lægger ud med at søge en søjle igennem for at finde et element forskelligt fra nul. Når det lykkes, ombyttes rækkerne, således at der ikke står et nul som diagonalelement.

```
840 void FindZero(double InputMatrix[][MaxBigCol], int
        ColPosition, int NumberOfRows){
841     double Temp;
842     int ValidRow;
843     for(int i=ColPosition; i<NumberOfRows; i++)
844        if(fabs(InputMatrix[i][ColPosition])>Tolerance){
               Tjekker om elementet er mindre end vor tolerance.
845           ValidRow=i; Fandt en gyldig række.
846           break;
847        }
```

Når vi har fundet en gyldig række, skal den ombyttes med den første, vi undersøgte. Dog er der ingen grund til at foretage en ombytning, hvis vi havde held med søgningen i første hug.

```
848     if(ValidRow≠ColPosition)
849        for(int i=0; i<(2*NumberOfRows); i++){
850           Temp=InputMatrix[ValidRow][i];
851           InputMatrix[ValidRow][i]=
                 InputMatrix[ColPosition][i];
852           InputMatrix[ColPosition][i]=Temp;
853        }
854 }
```

Nu kan vi så gå videre med at dividere rækken igennem omkring pivotelementet.

```
855 void DivideRow(double InputMatrix[][MaxRow+MaxRow], int
        RowPosition, int ColPosition, int NumberOfCols){
856     for(int i=0; i<NumberOfCols; i++)
```

```
857         if(i≠ColPosition)
858            InputMatrix[RowPosition][i]=
                  InputMatrix[RowPosition][i]/
                  InputMatrix[RowPosition][ColPosition];
859      InputMatrix[RowPosition][ColPosition]=1; Undgår afrund-
            ingsfejl.
860   }
```

Sluttelig skal vi have elimineret tallene over og under pivotelementet.

```
861   void EliminateAboveAndBelow(double InputMatrix[][MaxRow+
         MaxRow], int RowPosition, int ColPosition, int
         NumberOfRows, int NumberOfCols){
862      for(int i=0; i<NumberOfRows; i++)
863         if(i≠RowPosition){
864            for(int j=0; j<NumberOfCols; j++)
865               if(j≠ColPosition)
866                  InputMatrix[i][j]−=
                        InputMatrix[i][ColPosition]*
                        InputMatrix[RowPosition][j];
867            InputMatrix[i][ColPosition]=0; Undgår afrundingsfejl.
868         }
869   }
```

A.6 Hjælpefunktioner

I det følgende beskrives nogle af de mindre funktioner, der anvendes i programmet. En del af dem bruges udelukkende til at skrive informationer på skærmen, hvorfor de ikke er beskrevet nærmere.

A.6.1 Ekstra informationer

Som et led i testkørslerne har vi valgt at lade programmet modtage et ekstra argument, nemlig strengen `-info`, som hvis aktiveret giver anledning til udskrift af en mængde informationer undervejs. `argc` tæller antallet af argumenter til programmet, mens `*argv` gemmer argumenterne på hver sin plads i en **char**-vektor.

```
870   void XtraOps(int argc, char *argv[]){
871      string InfoString("−info");
872      string TestRunString("−test");
```

```
873     if(argc>1){
874        if(argv[1]≡InfoString){
875           PrintInfo=true;
876           cout≪"PrintInfo er aktiveret...";
877        }
878        else
879           if(argv[1]≡TestRunString){
880              Test_Mode=true;
881              cout≪"Test_Mode er aktiveret...";
882           }
883           else
884              cout≪endl≪"Beklager, jeg kan ikke godkende det,
                    du har indtastet, så du får lidt hjælp istedet."≪
                    endl≪"Godkendte valgmuligheder er:"≪endl≪
                    InfoString≪": Printer information såsom \"cout
                    ≪\"i løbet af programmet."≪endl≪
                    TestRunString≪": Kører nogle testproblemer.";
885     }
886  }
```

A.6.2 Test af programmet

Til testkørslerne af programmet er der konstrueret følgende funktion:

```
887  void SetupTests(long int &MaxTestRun, int
        &NumberOfColumns, int &EquationNumber, double
        CoefAndSlack[][MaxBigCol], double ResultCoef[], double
         TrueResult[]){
888     cout≪endl≪"Indtast antal variable (n) i testproblemet: ";
889     cin≫NumberOfColumns;
890     cout≪endl≪"Indtast antal gentagelser af problemets
           løsningsmetode: ";
891     cin≫MaxTestRun;
892     cout≪endl≪"Indtast gamma: ";
893     cin≫gamma;
894     cout≪endl≪"Indtast tolerance: ";
895     cin≫Tolerance_xs;
896     Tolerance=pow(10,-14);
897     Tolerance_rD=Tolerance_xs;
898     Tolerance_rP=Tolerance_xs;
899     cout≪endl≪"Indtast multiplikationsfaktor til problemet: ";
900     double Factor;
```

```
901     cin≫Factor;
902     int OffSet=NumberOfColumns*NumberOfColumns;
903     EquationNumber=NumberOfColumns+1;
904     for(int i=1;i<EquationNumber;i++){
905        CoefAndSlack[i][i+NumberOfColumns-1]=Factor*1;
906        for(int j=0;j<NumberOfColumns;j++)
907           CoefAndSlack[i][j]=Factor*(i+2*j);
908     }
909     for(int i=0;i<NumberOfColumns;i++)
910           CoefAndSlack[0][i]=Factor*(OffSet+EquationNumber
                 +3*i);
911     for(int i=1;i<EquationNumber;i++)
912        ResultCoef[i]=Factor*(OffSet+NumberOfColumns*i);
913     for(int i=1; i<NumberOfColumns; i++)
914        TrueResult[i+NumberOfColumns-1]=NumberOfColumns*(
              NumberOfColumns-i);
915     TrueResult[0]=2.*NumberOfColumns;
916  }
```

A.6.3 Rensning af matricer

Vi skal have renset matricer og vektorer igennem for at fjerne værdier tæt på 0. Pga. den måde C++ gemmer arrays på i hukommelsen, er vi nødt til at lave en version for hvert søjleantal. C++ kan dog godt finde ud af det, når bare man kalder funktionen med argumenter (dvs. arrays) af den rigtige størrelse.

```
917  void CleanMatrix(double ArbitraryMatrix[][MaxBigCol+1],
        int NumberOfRows, int NumberOfCols){
918     for(int i=0; i<NumberOfRows; i++)
919        for(int j=0; j<NumberOfCols; j++)
920           if(fabs(ArbitraryMatrix[i][j])<Tolerance)
921              ArbitraryMatrix[i][j]=0;
922  }
923  void CleanMatrix(double ArbitraryMatrix[][MaxCol], int
        NumberOfRows, int NumberOfCols){
924     for(int i=0; i<NumberOfRows; i++)
925        for(int j=0; j<NumberOfCols; j++)
926           if(fabs(ArbitraryMatrix[i][j])<Tolerance)
927              ArbitraryMatrix[i][j]=0;
928  }
```

```
929 void CleanVector(double ArbitraryVector[], int
       NumberOfRows){
930    for(int i=0; i<NumberOfRows; i++)
931       if(fabs(ArbitraryVector[i])<Tolerance)
932          ArbitraryVector[i]=0;
933 }
```

A.6.4 Udskrivning af matricer og vektorer

```
934 void PrintMatrix(double CoefAndSlack[][MaxBigCol], int
       RowsInCoefAndSlack, int ColsInCoefAndSlack){
935    for(int i=0; i<RowsInCoefAndSlack; i++){
936       cout≪endl;
937       for(int j=0; j<ColsInCoefAndSlack; j++)
938          cout≪setprecision(2)≪setw(6)≪
                CoefAndSlack[i][j];
939    }
940 }

941 void PrintMatrix(double ArbitraryMatrix[][MaxCol], int
       NumberOfRows, int NumberOfCols){
942    for(int i=0; i<NumberOfRows; i++){
943       cout≪endl;
944       for(int j=0; j<NumberOfCols; j++)
945          cout≪setprecision(2)≪setw(6)≪
                ArbitraryMatrix[i][j];
946    }
947 }

948 void PrintMatrix(double ArbitraryBigMatrix[][MaxBigCol+1
       ], int NumberOfRows, int NumberOfCols){
949    for(int i=0; i<NumberOfRows; i++){
950       cout≪endl;
951       for(int j=0; j<NumberOfCols; j++)
952          cout≪setprecision(2)≪setw(6)≪
                ArbitraryBigMatrix[i][j];
953    }
954 }

955 void PrintVector(double c[], int NumberOfRowsInC){
956    cout≪endl;
```

```
957     for(int i=0; i<NumberOfRowsInC; i++)
958        cout≪setw(19)≪setprecision(10)≪c[i];
959  }
```

A.6.5 Inialitisering af matricer

Flere steder i programmet skal vi initialisere matricer, således at de kun indeholder nuller. Derfor laver vi en lille funktion til dette formål.

```
960  void InitializeMatrix(double ArbitraryMatrix[][MaxCol]){
961     for(int i=0; i<MaxRow; i++)
962        for(int j=0; j<MaxCol; j++)
963           ArbitraryMatrix[i][j]=0;
964  }
```

A.6.6 Kopiering af matricer

Først en lille funktion til kopiering af matricer.

```
965  void CopyMatrix(double ArbitraryMatrix[][MaxCol], int
        NumberOfRows, int NumberOfCols, int StartColumn,
        double SimplexTableau[][MaxBigCol]){
966     for(int i=0; i<NumberOfRows; i++)
967        for(int j=0; j<NumberOfCols; j++)
968           SimplexTableau[i][j+StartColumn]=
                 ArbitraryMatrix[i][j];
969  }
```

Dernæst en funktion, der tilføjer en vektor til en matrix.

```
970  void AppendColumnToMatrix(double Vector[], double
        InputMatrix[][MaxBigCol], int InsertionPlace, int
        NumberOfRows, double SimplexTableau[][MaxBigCol+1]){
971     for(int i=0;i<NumberOfRows; i++){
972        SimplexTableau[i][InsertionPlace]=Vector[i];
973        for(int j=0; j<InsertionPlace; j++)
974           SimplexTableau[i][j]=InputMatrix[i][j];
975     }
976  }
```

A.6.7 Andre små funktioner

Selve tolerancen.

```
977 void ChooseTolerance(double &Tolerance){
978    int a, b;
979    cout≪endl≪"Indtast en tolerance af formen a ∗ 10^−b, hvor a og
          b er heltal.";
980    cout≪endl≪"Indtast a: ";
981    cin≫a;
982    cout≪endl≪"Indtast b: ";
983    cin≫b;
984    Tolerance=a*pow(10,−b);
985 }
```

Beregner normen af en vektor.

```
986 double FindNorm(double Vector[MaxBigCol], int
       NumberOfRows){
987    double Norm=0;
988    for(int i=0; i<NumberOfRows; i++)
989       Norm+=pow(Vector[i],2);
990    return sqrt(Norm);
991 }
```

A.7 En hjemmestrikket header-fil

Nedenfor er listet vores egen header-fil `header.h`, som vi bruger til at lave funktionsprototyper mm. Først inkluderes de forskellige funktionsbiblioteker som fx `<iostream.h>`.

```
992 #include <iostream.h>
993 #include <conio.h>
994 #include <string.h>
995 #include <ctype.h>
996 #include <stdlib.h>
997 #include <iomanip.h>
998 #include <time.h>
```

Konstanter:

```
999  const int MaxCol=70;
1000 const int MaxBigCol=140;
1001 const int MaxRow=70;
1002 long int MaxTestRun=1;
1003 long int NumberOfTestRuns=0;
1004 bool PrintInfo=false;
```

```
1005 bool Test_Mode=false;
1006 bool SingularSimplex;
1007 bool FoundOptimum;
1008 bool FoundValidSolution;
1009 int MaxCoefPosition;
1010 int MinCoefPosition;
1011 int MinFractionPosition;
1012 int DualMinFractionPosition;
1013 double Tolerance;
1014 double gamma=0.05;
1015 double Tolerance_xs=pow(10,-6);
1016 double Tolerance_rD=pow(10,-6);
1017 double Tolerance_rP=pow(10,-6);
1018 double SmallestNumber;
```

Herefter kommer så funktionsprototyperne.

```
1019 void PresentSolution(double InputMatrix[][MaxBigCol+1],
        int NumberOfRows, int NumberOfCols, double
        TrueResult[]);
1020 void SetupTests(long int &MaxTestRun, int
        &NumberOfColumns, int &EquationNumber, double
        CoefAndSlack[][MaxBigCol], double ResultCoef[], double
         TrueResult[]);
1021 void XtraOps(int argc, char *argv[]);
1022 void ReadCoef(string &buffer, double Coef[][MaxCol], int
        &EquationNumber, int &NumberOfColumns);
1023 void DeterminEquationType(string &buffer, double
        ResultCoef[], bool &LessThan, bool &GreaterThan, bool
        &EqualTo, bool &CriterionFunction, int &EquationNumber
        );
1024 void ReadString(string &buffer);
1025 void ChooseOneOfTwo(string FirstString, string
        SecondString, bool &Choice);
1026 void SaveResultCoef(string &buffer, double ResultCoef[],
        int &EquationNumber, int t);
1027 void SwitchSigns(bool MaxProblem, bool CriterionFunction,
         bool GreaterThan, int EquationNumber, double
        Coef[][MaxCol], double ResultCoef[]);
1028 void EndOfEquations(string &buffer, bool &StopKeyingIn);
1029 void CleanUpStringOther(string &buffer);
1030 void CleanUpStringSpaces(string &buffer);
1031 void InitializeMatrix(double ArbitraryMatrix[][MaxCol]);
```

```
1032 void CreateSlackVars(bool LessThan, bool GreaterThan, int
         EquationNumber, double SlackVars[][MaxCol]);
1033 void PrintMatrix(double ArbitraryMatrix[][MaxCol], int
       NumberOfRows, int NumberOfCols);
1034 void PrintMatrix(double CoefAndSlack[][MaxBigCol], int
       RowsInCoefAndSlack, int ColsInCoefAndSlack);
1035 void PrintMatrix(double ArbitraryBigMatrix[][MaxBigCol+1
       ], int NumberOfRows, int NumberOfCols);
1036 void CopyMatrix(double ArbitraryMatrix[][MaxCol], int
       NumberOfRows, int NumberOfCols, int StartColumn,
       double SimplexTableau[][MaxBigCol]);
1037 void AppendColumnToMatrix(double Vector[], double
       InputMatrix[][MaxBigCol], int InsertionPlace, int
       NumberOfRows, double SimplexTableau[][MaxBigCol+1]);
1038 void CleanMatrix(double ArbitraryMatrix[][MaxBigCol+1],
       int NumberOfRows, int NumberOfCols);
1039 void CleanMatrix(double ArbitraryMatrix[][MaxCol], int
       NumberOfRows, int NumberOfCols);
1040 void CleanVector(double ArbitraryVector[], int
       NumberOfRows);
1041 void ChooseTolerance(double &Tolerance);
1042 void CollectData(bool &MaxProblem, double ResultCoef[],
       double CoefAndSlack[][MaxBigCol], int &EquationNumber,
        int &NumberOfColumns);
1043 void Simplex(double InputMatrix[][MaxBigCol+1], int
       NumberOfRows, int NumberOfCols, double TrueResult[]);
1044 void PrepareIP(bool MaxProblem, double
       CoefAndSlack[][MaxBigCol], double ResultCoef[], int
       &RowsInCoefAndSlack, int &ColsInCoefAndSlack, double
       CritFunc[]);
1045 void PrepareIP_Test(int NumberOfColumns, double x[],
       double s[]);
1046 void InteriorPoint(double CoefAndSlack[][MaxBigCol],
       double ResultCoef[], double CritFunc[], int
       RowsInCoefAndSlack, int ColsInCoefAndSlack, double
       TrueResult[]);
1047 int FindMaxCoef(double InputMatrix[][MaxBigCol+1], int
       NumberOfCols);
1048 int FindMinCoef(double InputMatrix[][MaxBigCol+1], int
       NumberOfRows, int NumberOfCols);
```

```
1049 int FindSmallestFraction(double InputMatrix[][MaxBigCol+1
       ], int NumberOfRows, int NumberOfCols, int
       MaxCoefPosition);
1050 int DualFindSmallestFraction(double
       InputMatrix[][MaxBigCol+1], int NumberOfCols, int
       MinCoefPosition);
1051 bool CheckForSimplex(double InputMatrix[][MaxBigCol+1],
       int NumberOfRows, int NumberOfCols);
1052 bool CheckForOptimum(double InputMatrix[][MaxBigCol+1],
       int NumberOfRows, int NumberOfCols);
1053 void DivideRow(double InputMatrix[][MaxBigCol+1], int
       MinFracPosition, int MaxCoefPosition, int NumberOfCols
       );
1054 void EliminateAboveAndBelow(double
       InputMatrix[][MaxBigCol+1], int MinFracPosition, int
       MaxCoefPosition, int NumberOfRows, int NumberOfCols);
1055 void MatrixProduct(double [][MaxBigCol], double
       B[][MaxBigCol], int RowsInCoefAndSlack, int
       ColsInCoefAndSlack, int NumberOfColsInB, double
       C[][MaxBigCol]);
1056 void Small_MatrixProduct(double
       CoefAndSlack[][MaxBigCol], double B[][MaxBigCol], int
       RowsInCoefAndSlack, int ColsInCoefAndSlack, int
       NumberOfColsInB, double C[][MaxRow]);
1057 void VectorProduct(double b[MaxBigCol], double
       c[MaxBigCol], int NumberOfRowsInC, double &Result);
1058 void PrintVector(double c[], int NumberOfRowsInC);
1059 void MatrixVectorProduct(double
       CoefAndSlack[MaxRow][MaxBigCol], double b[MaxBigCol],
       int RowsInCoefAndSlack, int ColsInCoefAndSlack, double
        ResultVector[MaxBigCol]);
1060 void Small_MatrixVectorProduct(double
       CoefAndSlack[MaxRow][MaxRow], double b[MaxRow], int
       RowsInCoefAndSlack, int ColsInCoefAndSlack, double
       ResultVector[MaxRow]);
1061 void CreateDiagonalMatrix(double b[], int
       NumberOfRowsInB, double DiagonalMatrix[][MaxBigCol]);
1062 void VectorSubtraction(double b[], double c[], int
       NumberOfRowsInB, double SubVector[]);
1063 void VectorAddition(double b[], double c[], int
       NumberOfRowsInB, double SubVector[]);
```

```
1064 void TransposeMatrix(double
        CoefAndSlack[MaxRow][MaxBigCol], int
        RowsInCoefAndSlack, int ColsInCoefAndSlack, double
        CoefAndSlack_T[MaxBigCol][MaxBigCol]);
1065 void InverseDiagonalMatrix(double
        Input[MaxBigCol][MaxBigCol], int RowsInCoefAndSlack,
        double InverseInput[MaxBigCol][MaxBigCol]);
1066 void DivideRow(double InputMatrix[MaxRow][MaxRow+MaxRow],
         int RowPosition, int ColPosition, int NumberOfCols);
1067 void EliminateAboveAndBelow(double
        InputMatrix[MaxRow][MaxRow+MaxRow], int RowPosition,
        int ColPosition, int NumberOfRows, int NumberOfCols);
1068 void FindZero(double InputMatrix[MaxRow][MaxRow+MaxRow],
        int ColPosition, int NumberOfRows);
1069 void SplitMatrix(double InputMatrix[MaxRow][MaxRow+
        MaxRow], int NumberOfRows, double
        OutputMatrix[MaxRow][MaxRow]);
1070 void AppendIdentityMatrix(double
        SmallInputMatrix[MaxRow][MaxRow], int NumberOfRows,
        double BigOutputMatrix[MaxRow][MaxRow+MaxRow]);
1071 double FindSmallestNumber(double VectorOne[MaxBigCol],
        double VectorOne2[MaxBigCol], double
        VectorTwo[MaxBigCol], double VectorTwo2[MaxBigCol],
        int NumberOfRows);
1072 void MultiplyVector(double TempVector[MaxBigCol], double
        Factor, int NumberOfRows);
1073 double FindNorm(double Vector[MaxBigCol], int
        NumberOfRows);
1074 void Calculate_rP(double ResultCoef[MaxBigCol], double
        CoefAndSlack[MaxRow][MaxBigCol], double x[MaxBigCol],
        int RowsInCoefAndSlack, int ColsInCoefAndSlack, double
         Rp[MaxBigCol]);
1075 void Calculate_rD(double CritFunc[MaxBigCol], double
        CoefAndSlack_T[MaxBigCol][MaxBigCol], double
        y[MaxRow], double s[MaxBigCol], int
        RowsInCoefAndSlack, int ColsInCoefAndSlack, double
        rD[MaxBigCol]);
1076 void LeftSide(double CoefAndSlack[MaxRow][MaxBigCol],
        double X[MaxBigCol][MaxBigCol], double
        S_Inverse[MaxBigCol][MaxBigCol], double
        CoefAndSlack_T[MaxBigCol][MaxBigCol], int
```

```
        RowsInCoefAndSlack, int ColsInCoefAndSlack, double
        LeftResult[MaxRow][MaxRow]);
1077  void InvertMatrix(double LeftResult[MaxRow][MaxRow], int
        RowsInCoefAndSlack, double
        LeftResult_Inverse[MaxRow][MaxRow]);
1078  void RightSide(double ResultCoef[MaxRow], double
        CoefAndSlack[MaxRow][MaxBigCol], double
        S_Inverse[MaxBigCol][MaxBigCol], double
        X[MaxBigCol][MaxBigCol], double rD[MaxBigCol], double
        mu, int RowsInCoefAndSlack, int ColsInCoefAndSlack,
        double RightResult[MaxRow]);
1079  void Calculate_dx(double x[MaxBigCol], double
        S_Inverse[MaxBigCol][MaxBigCol], double mu, double
        X[MaxBigCol][MaxBigCol], double ds[MaxBigCol], int
        ColsInCoefAndSlack, double dx[MaxBigCol]);
```

Litteratur

Erling D. Andersen.
Linear optimization: Theory, methods and extensions.
Department of Management, Odense University, Odense, 1998.
URL `http://erling.andersen.name`.

Harvey Deitel og Paul Deitel.
C++ How to Program.
Prentice Hall, Upper Saddle River, New Jersey, 3. udgave, 2001.

Bent Fuglede, Tage Gutmann Madsen og Gert Grubb.
Analyse og optimering.
Universitetsbogladen, Københavns Universitet, Universitetsparken 5, 2100 København Ø, 1999.

Hans Keiding.
Operationsanalyse.
Jurist- og Økonomforbundets forlag, København, 2002.

Knut Sydsæter.
Matematisk analyse Bind 1.
Gyldendal Akademisk, Oslo, 7. udgave, 2000.