# Bepaling van Grondwaterstroming met Iteratieve Oplosmethoden

Albert-Jan Yzelman en Hanno Mulder

30 juni 2005

# Inhoudsopgave

# 1 Inleiding

Dit verslag gaat over de bepaling van grondwaterstroming, maar de modellering en oplossingsmethoden die we zullen bespreken zijn toepasbaar op tal van andere fysische problemen. Om te kunnen beginnen, zal het fysisch probleem eerst moeten worden gemodelleerd als wiskundig probleem. Vervolgens kunnen er wiskundige technieken toegepast worden om tot een efficiënte oplosmethode te komen, die dan in C++ wordt geimplementeerd.
We zullen zien dat het probleem zich zal toespitsen op het oplossen van het stelsel vergelijkingen

$$Ax = b, \tag{1}$$

waarin $A$ een bekende $n$ x $n$ matrix is, $b$ een bekende $n$-vector en $x$ een onbekende $n$-vector die bepaald moet worden. Een groot deel van dit verslag zal gaan over het oplossen van dit stelsel vergelijkingen. Dit moet namelijk liefst zo snel mogelijk gebeuren. Daartoe maken we gebruik van iteratieve methoden, die voor grote stelsels veel sneller zijn dan de directe methoden.
De gebruikte methoden zullen theoretisch besproken en in de praktijk getest worden.

In hoofdstuk 2 wordt er een vraagstelling geformuleerd, waar zowel een theoretisch als een experimenteel antwoord op geformuleerd zal worden. In de hoofdstukken 3, 4 en 5 wordt de benodigde theorie behandeld, waarna in hoofdstuk 6 het theoretische antwoord zal geformuleerd worden in de vorm van een hypothese. Deze hypothese zal uitgebreid worden getoetst aan de hand van testproblemen die daadwerkelijk zullen worden opgelost, in hoofdstuk 7. Uiteindelijk vindt u in hoofdstuk 8 de getrokken conclusies. Ook vindt u daar enige aanbevelingen om het wellicht beter te doen.

Dit verslag is gemaakt naar aanleiding van het vak Practicum Computational Science. Er zal daarom veelvuldig gebruik worden gemaakt van de practicumhandleiding [1].

# 2 Vraagstelling

Aan de hand van de hierna beschreven problemen, zullen we methodes zoeken die zo snel en goed mogelijk een oplossing van het probleem genereren. Het hoeft niet zo te zijn dat er een oplosmethode is die beter is dan alle andere; waarschijnlijk hangt dit van het probleem af.
De explicitiete deelvragen die we willen stellen en zullen beantwoorden zijn:

-Hoe modelleren we zo efficient mogelijk?

-Hoe kunnen we het beste preconditioneren? (Zie hoofdstuk 5.4)

-Welke iteratieve oplosmethoden passen het best bij welk probleem?

-Zijn er nog verdere versnellingen mogelijk?

## 2.1 Probleem 1

We beschouwen een grondgebied ter grootte $[0, 3000]$ bij $[0, 3000]$. Er is een lagere doorlaatbaarheid van water ($5m^3/(\text{dag } m^2)$) in het gebied $[1350, 1950]$ bij $[1200, 3000]$; bijvoorbeeld omdat daar voornamelijk zand ligt. Het gebied daarbuiten heeft een veel hogere doorlaatbaarheid (van $40m^3/(\text{dag } m^2)$); daar ligt bijvoorbeeld klei. Over de gehele westrand van het gebied ($x = 0$) kan er geen water in- of uit stromen, behalve bij $1900 \leq y \leq 2100$: daar is een uitstroomsnelheid aanwezig die evenredig is met het verschil tussen de inkomende druk en een referentiedruk (Robin-randvoorwaarde). Dus, eenvoudiger gezegd, hoe meer de binnenkomende druk afwijkt van een bepaalde referentiedruk, hoe meer water er uitstroomt. Voor de referentiedruk nemen we hier 400 meter.
Op de oostrand ($x = 3000$) hebben we voor $400 \leq y \leq 600$ een vaststaande waterdruk van 200 meter. De rest van de oostrand is wederom ondoordringbaar voor water, net zoals de gehele noord- ($y = 3000$) en zuidrand ($y = 0$).
In het gebied staan verder twee pompen opgesteld. Eentje staat op de coordinaten $(2400, 1800)$ en pompt $1200m^3$ per dag weg. De andere staat op $(1550, 600)$ en pompt dezelfde hoeveelheid water per dag weg. Verder stroomt er ook nog een rivier op de lijn $x = 900$ die $0,7m^3/(\text{dag } m)$ water toevoegt. Het probleem is nu om te berekenen wat de grondwaterdruk is op punten binnen het gebied.
Dit probleem is equivalent aan *Testprobleem V* uit [1].
Voor het visuele resultaat zie Appendix A.

## 2.2 Probleem 2

In dit probleem gaan we kijken hoe gif zich verspreidt binnen een gegeven gebied. We gaan hier uit van een een gebied ter grootte van $[0, 3000]$ bij $[0, 1500]$. De west- en oostrand hebben een constante grondwaterdruk van 200 meter. Aan de noord- en zuidkant hebben we een Robin-voorwaarde met een referentiedruk van 0 meter. Op $(1000, 700)$ staat een pomp die $1200m^3/(\text{dag } m)$ weghaalt. Ook hebben we een rivier die van $(2500, 0)$ naar $(1000, 1500)$ stroomt, en $0,24m^3/(\text{dag } m)$ water toevoegt. Aan de westkant van deze rivier heeft de grond een doorlaatbaarheid van $60m^3/(\text{dag } m^2)$, aan de oostkant is dit 40. Voor $|x - 2500| < 300$ en $y > 1000$ hebben we echter een doorlaatbaarheid van 1.
Het gif wordt op $(1900, 900)$ met 240 gram per dag het grondwater in geloosd. Op de noord- en zuidrand hebben we voor het vergif een Robin randvoorwaarde met evenredigheidsconstante $0,5$. De oost- en westrand zijn ondoordringbaar voor het vergif. Het probleem is nu om de gifconcentratie in het gebied te berekenen.
Dit probleem is equivalent aan *Testprobleem IV* uit [1].
Zie ter illustratie weer Appendix A.

# 3 Fysisch Probleem

## 3.1 Grondwaterstroming

Het fysisch probleem wat we willen oplossen is de stroming van grondwater door een poreuze grondlaag. We zullen ons richten op het 2-dimensionale probleem, hoewel een grondlaag in principe natuurlijk 3-dimensionaal is. Dit wil zeggen dat we aannemen dat het water alleen in de $x$ en $y$ richting stroomt en niet in de $z$ richting. Verder gaan we uit van een stationaire stroming: een stroming die gedurende de tijd niet verandert.
Om tot een methode te komen om de grondwaterstroming te bepalen zullen we gebruik maken van een wiskundig model.

### 3.1.1 Wiskundig model

Een belangrijke aanname die we doen is dat er geen massa verloren gaat tijdens de stroming. Dus de instroom in een gebied is gelijk aan de hoeveelheid uitstroom. Op grond daarvan kan stroming in een deel $D$ van het $xy$-vlak beschreven worden door de differentiaalvergelijking

$$-\nabla \cdot (K \nabla \phi) = Q. \tag{2}$$

Hierin is:

- $K = K(x, y)$ een 2 x 2 matrix met entries $k_{ij}(x, y)$: de doorlaatbaarheidscoëfficiënten. Deze coëfficiënten geven het aantal kubieke meter water per dag dat door een oppervlakte ($m^2$) stroomt. Dit geeft als eenheid $m^3/(\text{dag } m^2)$
- $\phi = \phi(x, y)$ de grondwaterdruk op plaats $(x, y)$. Deze wordt gemeten in meters.
- $Q = Q(x, y)$ de brondterm, die aangeeft hoeveel water er op plaats $(x, y)$ wordt toegevoegd of weggehaald.

Daarnaast kunnen we op elke plaats $(x, y)$ de snelheid van het grondwater aangeven met $u(x, y)$ en $v(x, y)$. Waarbij $u$ de snelheid is in de $x$-richting en $v$ de snelheid in de $y$-richting. Dit snelheidsveld word gegeven in $(m^3/\text{dag } m^2)$ en wordt gegeven door

$$(u, v) := -K \nabla \phi. \tag{3}$$

In een gebied bevindt zich vaak meer dan alleen maar 1 soort grond. Met behulp van het bovenstaande model kunnen we onder andere de volgende dingen modelleren:

- Verschillende grondsoorten hebben verschillende doorlaatbaarheidscoëfficiënten, en kunnen dus worden gemodelleerd door de matrix $K$.
- Obstakels, als bijvoorbeeld stenen, kunnen eveneens gemodelleerd worden door de doorlaatbaarheidscoëfficiënten, dus in de matrix $K$.
- Rivieren voegen water toe, en kunnen derhalve gemodelleerd worden door de bronterm $Q$.
- Pompen kunnen water wegnemen, en kunnen dus ook gemodelleerd worden door de bronterm $Q$.

Zie Figuur 1 voor een voorbeeld van een gebied $D$.

### 3.1.2 Randwaarden

De stroming op de rand van $D$ ($\partial D$) vraagt extra specificaties. We kunnen de rand opsplitsen in twee disjuncte stukken: $\partial D = \Gamma_0 \cup \Gamma_1$, zie ook Figuur 1.
Op $\Gamma_0$ hebben we te maken met een Dirichlet randvoorwaarde die de druk van het grondwater specificeert:

$$\phi = \phi_0. \tag{4}$$

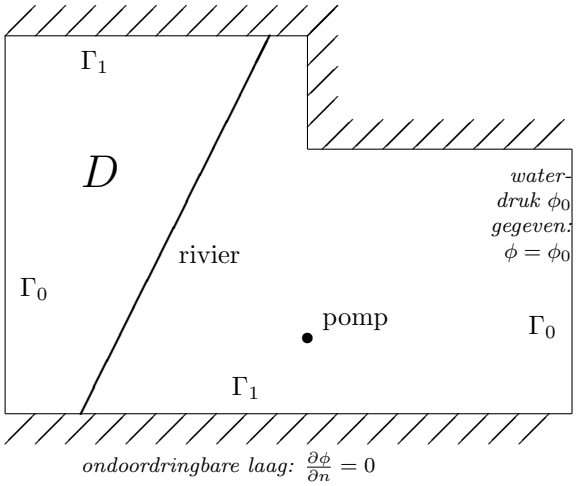



FIG.1: *Grondwaterstroming op een gebied D met rivier en pomp. Op de west – en de oost rand ($\Gamma_0$) Dirichlet randvoorwaarde, op de noord – en zuid rand ($\Gamma_1$) Neumann randvoorwaarde.*

Op $\Gamma_1$ hebben we een Neumann randvoorwaarde die de uitstroomsnelheid van het grondwater specificeert:

$$-(K\nabla\phi)\cdot n = \phi_0. \tag{5}$$

In beide randvoorwaarden is $\phi_0$ een gegeven functie, en $n$ is de normaal vector.
Tenslotte kunnen we nog een derde randvoorwaarde gebruiken, de Robin randvoorwaarde:

$$-(K\nabla\phi)\cdot n = \gamma(\phi - \phi_\infty). \tag{6}$$

Hierin is $\phi_\infty$ een referentiedruk. De uitstroomsnelheid uit $D$ is nu evenredig met het drukverschil tussen de druk $\phi$ en de referentiedruk. Verder is $\gamma$ een evenredigheidsconstante.
Aan de hand van gegeven randvoorwaarden, doorlaatbaarheidscoëfficiënten en brontermen kunnen we nu de grondwaterdruk bepalen.

## 3.2 Bodemverontreiniging

Een aan (2) verwante vergelijking beschrijft de verspreiding van een in water oplosbare stof via het grondwater. Definiëren we $\psi$ als de concentratie van de stof (in gr/$m^3$), dan geldt voor $\psi$:

$$-\nabla\cdot(A\nabla\psi) + \nabla\cdot(V\psi) + c\psi = f. \tag{7}$$

Hierin

- geeft $A$ de doorlaatbaarheid van de grondlaag met betrekking tot de stof;
- is $-A\nabla\psi$ de snelheid waarmee de stof zich verspreid in het grondwater door diffusie;
- is $f$ de bronterm;
- geeft $V = (u, v)$ het snelheidsveld van het grondwater;
- is $c$ een constante.

De concentratie van de stof op een plaats $(x, y)$ kan op verschillende manieren veranderen:

- Toevoeging van de stof van buitenaf kan gemodelleerd worden door de bronterm.

- Diffusie wordt gemodelleerd door de diffusie term $-A\nabla\psi$.
- Convectie (meestroming met het grondwater) wordt gemodelleerd door de convectie term $\nabla \cdot (V\psi)$.
- Natuurlijke afbraak wordt gemodelleerd door de term $c\psi$. Omdat dit een evenredigheid tussen de afbraak en de concentratie definieert, noemen we $c$ wel de evenredigheidsconstante. Deze hangt af van de grondsoort.

Door van (7) de convectie- en de natuurlijke afbraak term buiten beschouwing te laten, krijgen we (2): de vergelijking voor de grondwaterstroming. Met daarbij opgemerkt dat $A$ in het algemeen niet gelijk is aan $K$. We kunnen dus met (7) zowel de grondwaterstroming als de stofverspreiding modelleren. In de volgende hoofdstukken zal daarom steeds het oplossen van (7) centraal staan.

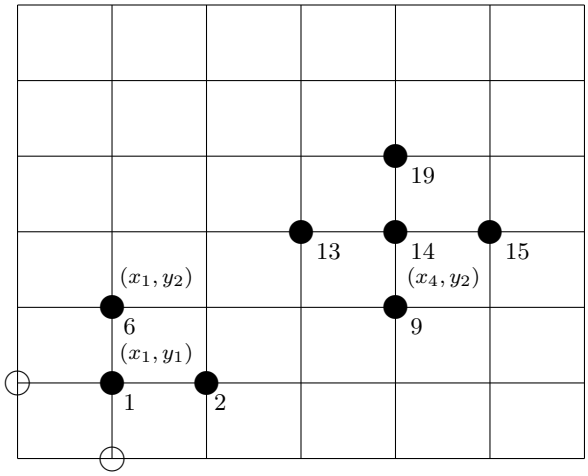



FIG.2: *Het rooster voor een rechthoekig gebied. De punten • behoren tot $D$, de punten ∘ liggen op de rand $\partial D$ van $D$.*

# 4 Discretisatie

In dit hoofdstuk wordt er een begin gemaakt met het oplosproces. Dit aan de hand van de methode van eindige differenties.
Om (voor ons) niet interessante details te vermijden gaan we uit van een rechthoekig gebied $D$

$$D = (0, X) \text{ x } (0, Y). \tag{8}$$

De algemene vergelijking (7) kunnen we in ons 2-dimensionaal probleem als volgt uitwerken:

$$-\frac{\partial}{\partial x}(a\partial\frac{\psi}{\partial x}) - \frac{\partial}{\partial y}(b\frac{\partial\psi}{\partial y}) + \frac{\partial(u\psi)}{\partial x} + \frac{\partial(v\psi)}{\partial y} + c\psi = f. \tag{9}$$

De randvoorwaarden (4) en (5) kunnen we samenvoegen in één vergelijking en verder uitwerken als

$$-\mu(a\frac{\partial\psi}{\partial x}, b\frac{\partial\psi}{\partial y}) \cdot n + (1-\mu)\psi = \psi_0. \tag{10}$$

We krijgen nu (4), dus de randvoorwaarde op $\Gamma_0$ door $\mu = 0$ te nemen, $\mu = 1$ geeft (5) en daarmee de randvoorwaarde op $\Gamma_1$ (voor de Robin randvoorwaarde (6) geldt: $\mu = \frac{1}{1-\gamma}$).

## 4.1 Discretisatie PDV

We willen de grondwaterdruk bepalen in het gebied $D$. Daartoe leggen we een rooster over $D$, zie Figuur 2.

In elk van deze roosterpunten gaan we de grondwaterdruk bepalen.
Defineer

$$h_x := \frac{X}{n_x + 1}, \quad h_y := \frac{Y}{n_y + 1}, \tag{11}$$

met $n_x, n_y \in \mathbb{N}$, dan kunnen we het rooster $R$ over $D$ vastleggen door

$$R := \{(x_i, y_j) : i, j \in \mathbb{N}, \quad i \leq n_x, \ j \leq ny\}, \tag{12}$$

met $(x_i, y_j) := (ih_x, jh_y)$. Zie ter illustratie Figuur 2.
Nu kunnen we de gevraagde functie $\psi$ bekijken in alle roosterpunten $(x_i, y_j)$, waarbij we $\psi_{i,j} := \psi(x_i, y_j)$ invoeren. In al deze roosterpunten bekijken we de eindige differentie van $\frac{\partial\psi}{\partial x}$. Deze wordt gegeven door

$$\frac{\partial\psi_{i,j}}{\partial x} := \frac{\psi_{i+1,j} - \psi_{i-1,j}}{2h_x}. \tag{13}$$

Uiteraard hadden we dit ook anders kunnen doen:

$$\frac{\partial \psi_{i,j}}{\partial x} := \frac{\psi_{i+\frac{1}{2},j} - \psi_{i-\frac{1}{2},j}}{h_x}. \tag{14}$$

De laatste vergelijking is uiteraard een betere benadering dan (13).
We willen zowel (13) als (14) gebruiken. Daarom voeren we de volgende notatie in:

- $\partial_x^1$ voor (13),
- $\partial_x^{\frac{1}{2}}$ voor (14).

Voor de $y$-richting voeren we de volstrekt analoge definities $\partial_y^1$ en $\partial_y^{\frac{1}{2}}$ in.
Interessant wordt het als we $\frac{\partial \psi}{\partial x}$ benaderen in het tussenpunt $(x_{i-\frac{1}{2}}, y_j)$ met (14):

$$\partial_x^{\frac{1}{2}} \psi_{i-\frac{1}{2},j} = \frac{\psi_{i,j} - \psi_{i-1,j}}{h_x}. \tag{15}$$

We zien dat we, door de waarden in de tussenpunten te benaderen met een eindige differentie, we alleen de punten $\psi_{i,j}$ hoeven te benaderen, terwijl we toch gebruik maken van de tussenpunten.
Uiteindelijk discretiseren we (9) in alle roosterpunten als

$$-\partial_x^{\frac{1}{2}}(a\partial_x^{\frac{1}{2}}\psi) - \partial_y^{\frac{1}{2}}(a\partial_y^{\frac{1}{2}}\psi) + \partial_x^1 u\psi + \partial_y^1 v\psi + c\psi = f + \delta. \tag{16}$$

Hierin is $\delta(x_i, y_j)$ de discretisatiefout; bij het benaderen van de afgeleide met een eindige differentie maak je namelijk fouten, zie daarvoor paragraaf 4.4.
Zoals al gezegd hebben we door de dubbele differentie benadering $\partial_x^{\frac{1}{2}}$ alleen te maken met functiewaarden $\psi_{i,j}$. De functies $a$ en $b$ moeten we uiteraard wel in de tussenpunten berekenen.

Negeren we de discretisatiefouten $\delta_{i,j}$, dan kunnen we de discretisatie (16) in een roosterpunt uitschrijven en zo uitkomen op een lineaire vergelijking

$$\alpha_{i,j}^{centraal}\psi_{i,j} + \alpha_{i,j}^{west}\psi_{i-1,j} + \alpha_{i,j}^{oost}\psi_{i+1,j} + \alpha_{i,j}^{zuid}\psi_{i,j-1} + \alpha_{i,j}^{noord}\psi_{i,j+1} = \hat{f}_{i,j}. \tag{17}$$

Wanneer we dit doen voor alle roosterpunten, krijgen we een stelsel lineaire vergelijkingen met als onbekenden de functiewaarden $\psi_{i,j}$. Omdat elk roosterpunt correspondeert met een vergelijking is dit een stelsel met $n := n_x n_y$ vergelijkingen en $n$ onbekenden.
De functies $\alpha$ kunnen gezien worden als koppelingscoëfficiënten die een functiewaarde koppelen aan zijn 'buren'. Het volgende stencil maakt dit duidelijk:

$$\begin{bmatrix} 0 & \alpha_{i,j}^n & 0 \\ \alpha_{i,j}^w & \alpha_{i,j}^c & \alpha_{i,j}^o \\ 0 & \alpha_{i,j}^z & 0 \end{bmatrix}, \tag{18}$$

waarbij een stencil 1 vergelijking representeert.

De functies $\alpha$ hangen af van de functies $a, b, u, c$ en $v$, en de functie $\hat{f}$ hangt af van $f$. We werken de verschillende termen uit (16), volledig uit voor $\psi_{i,j}$. Met (14) volgt

$$-\partial_x^{\frac{1}{2}}(a\partial_x^{\frac{1}{2}}\psi_{i,j}) = -\partial_x^{\frac{1}{2}}(a\frac{\psi_{i+\frac{1}{2},j} - \psi_{i-\frac{1}{2},j}}{h_x}).$$

Dit kunnen we met behulp van (15) uitwerken tot

$$\partial_x^{\frac{1}{2}}(a\partial_x^{\frac{1}{2}}\psi_{i,j}) = \frac{a_{i+\frac{1}{2},j}\frac{\psi_{i+1,j}-\psi_{i,j}}{h_x} - a_{i-\frac{1}{2},j}\frac{\psi_{i,j}-\psi_{i-1,j}}{h_x}}{h_x}.$$

Vereenvoudigen van deze vreselijke vergelijking lijdt tot

$$-\partial_x^{\frac{1}{2}}(a\partial_x^{\frac{1}{2}}\psi_{i,j}) = -\frac{a_{i+\frac{1}{2},j}\psi_{i+1,j} - (a_{i+\frac{1}{2},j} + a_{i-\frac{1}{2},j})\psi_{i,j} + a_{i-\frac{1}{2},j}\psi_{i-1,j}}{h_x^2}. \tag{19}$$

Hetzelfde kunnen we doen voor de tweede term uit (16), wat leidt tot

$$-\partial_y^{\frac{1}{2}}(b\partial_y^{\frac{1}{2}}\psi_{i,j}) = -\frac{b_{i,j+\frac{1}{2}}\psi_{i,j+1} - (b_{i,j+\frac{1}{2}} + b_{i,j-\frac{1}{2}})\psi_{i,j} + b_{i,j-\frac{1}{2}}\psi_{i,j-1}}{h_y^2}. \tag{20}$$

De derde term is eenvoudig met behulp van (16) uit te schrijven als

$$\partial_x^1 u\psi_{i,j} = \frac{u_{i+1,j}\psi_{i+1,j} - u_{i-1,j}\psi_{i-1,j}}{2h_x}, \tag{21}$$

en evenzo volgt voor de vierde term:

$$\partial_y^1 v\psi_{i,j} = \frac{v_{i,j+1}\psi_{i,j+1} - v_{i,j-1}\psi_{i,j-1}}{2h_y}. \tag{22}$$

De vijde term is uiteraard eenvoudig, maar voor de volledigheid geven we ook deze exact aan:

$$c\psi_{i,j} = c_{i,j}\psi_{i,j}. \tag{23}$$

Uit deze 5 vergelijkingen kunnen we nu de functies $\alpha$ afleiden door alle coëfficiënten van de $\psi$'s op te tellen.

$$\begin{aligned}
\alpha_{i,j}^c &= \tfrac{a_{i+\frac{1}{2},j}+a_{i-\frac{1}{2},j}}{h_x^2} + \tfrac{b_{i,j+\frac{1}{2}}+b_{i,j-\frac{1}{2}}}{h_y^2} + c_{i,j} \\
\alpha_{i,j}^w &= -\tfrac{u_{i-1,j}}{2h_x} - \tfrac{a_{i-\frac{1}{2},j}}{h_x^2} \\
\alpha_{i,j}^o &= \tfrac{u_{i+1,j}}{2h_x} - \tfrac{a_{i+\frac{1}{2},j}}{h_x^2} \\
\alpha_{i,j}^z &= -\tfrac{v_{i,j-1}}{2h_y} - \tfrac{b_{i,j-\frac{1}{2}}}{h_y^2} \\
\alpha_{i,j}^n &= \tfrac{v_{i,j+1}}{2h_y} - \tfrac{b_{i,j+\frac{1}{2}}}{h_y^2}
\end{aligned} \tag{24}$$

De functie $\hat{f}_{i,j}$ tenslotte is (voorlopig althans) gelijk aan de bronterm $f_{i,j}$.

## 4.2 Eliminatie randwaarden

De tot nu toe behandelde discretisatietheorie kunnen we helaas niet zondermeer toepassen op punten $(x_i, y_j)$ dicht bij $\partial D$. We zouden dan voor de eindige differentie functiewaarden $\psi_{i,j}$ nodig hebben die buiten $D$ vallen. De rand $\partial D$ beschouwen we als niet behorend tot $D$. Om deze randwaarden te discretiseren moeten we de normaal vector specificeren. De normaal vector is de vector ter lengte 1, die t.o.v. $D$ naar buiten wijst en in $(x_i, y_j)$ loodrecht staat op de rand. Dit definieert de normaalvectoren:

- westrand: $(-1, 0)^T$,
- oostrand: $(1, 0)^T$,
- zuidrand: $(0, -1)^T$,
- noordrand: $(0, 1)^T$.

Met deze normaalvectoren kunnen we de randwaarden als volgt discretiseren (respectievelijk west-, oost-, zuid- en noordrand):

$$\begin{aligned}
\mu_{0,j}(a\partial_x^{\frac{1}{2}})_{\frac{1}{2},j} + (1-\mu_{0,j})\psi_{0,j} &= \psi_0(x_0, y_j) + \delta_{0,j}, \\
-\mu_{n_x+1,j}(a\partial_x^{\frac{1}{2}})_{n_x+\frac{1}{2},j} + (1-\mu_{n_x+1,j})\psi_{n_x+1,j} &= \psi_0(x_{n_x+1}, y_j) + \delta_{n_x+1,j}, \\
\mu_{i,0}(b\partial_y^{\frac{1}{2}})_{i,\frac{1}{2}} + (1-\mu_{i,0})\psi_{i,0} &= \psi_0(x_i, y_0) + \delta_{i,0}, \\
-\mu_{i,n_y+1}(b\partial_y^{\frac{1}{2}})_{i,n_y+\frac{1}{2}} + (1-\mu_{i,n_y+1})\psi_{i,n_y+1} &= \psi_0(x_i, y_{n_y+1}) + \delta_{i,n_y+1},
\end{aligned} \tag{25}$$

Hierin zijn de $\delta$'s weer de discretisatiefouten.
Ter illustratie werken we de discretisatie voor de westrand verder uit:

$$\mu_{0,j}(a_{\frac{1}{2},j}\frac{\psi_{1,j}-\psi_{0,j}}{hx}) + (1-\mu_{0,j})\psi_{0,j} = \psi_0(0, y_j),$$

wat na herschikking per roosterpunt

$$\psi_{0,j}(1-\mu_{0,j} - \frac{a_{\frac{1}{2},j}\mu_{0,j}}{h_x}) = \psi_0(0,y_j) - \frac{\mu_{0,j}a_{\frac{1}{2},j}\psi_{1,j}}{h_x}$$

oplevert. Definiëren we

$$\begin{aligned}
\text{teller} &:= \mu_{0,j}a_{\frac{1}{2},j} \\
\text{noemer} &:= h_x(\mu_{0,j} - 1 + \text{teller}),
\end{aligned}$$

dan krijgen we de volgende uitdrukking voor $\psi_{0,j}$:

$$\psi_{0,j} = \frac{\psi_0(0,y_j)}{\text{noemer}} - \psi_{1,j}\frac{\text{teller}}{\text{noemer}}. \tag{26}$$

Beschouwen we nu de discretisatie in $(x_1, y_j)$ (zie (16)). Deze discretisatie maakt gebruik van de term $\alpha^w_{1,j}\psi_{0,j}$. Met (26) kunnen we deze term schrijven als

$$\alpha^w_{1,j}\psi_{0,j} = \alpha^w_{1,j}\frac{\psi_0(0,y_j)}{\text{noemer}} - \psi_{1,j}\frac{\text{teller}}{\text{noemer}}. \tag{27}$$

We kunnen de term $\psi_{0,j}$ dus elimineren. Dit kan expliciet gedaan worden door de volgende 2 bewerkingen:

$$\begin{aligned}
\alpha^c_{1,j} &= \alpha^c_{1,j} + \alpha^w_{1,j}\tfrac{\text{teller}}{\text{noemer}} \\
\hat{f}_{1,j} &= \hat{f}_{1,j} + \alpha^w_{1,j}\tfrac{\psi_0(0,y_j)}{\text{noemer}}.
\end{aligned} \tag{28}$$

Op dezelde manier kunnen we ook de andere randwaarden elimineren. De rand $\partial D$ wordt dus verder niet meegenomen in het op te lossen probleem. Wie er toch in geïnteresseerd is kan eenvoudigweg terugrekenen om de waarden van $\psi$ op $\partial D$ te bepalen.

## 4.3 Pompen en Rivieren

Het effect van pompen en rivieren kunnen we ook modelleren met het in hoofdstuk 3 gepresenteerde model. Stel we hebben te maken met een pomp in het roosterpunt $P = (x_p, y_p)$. Deze pomp pompt $p$ $m^3$/dag aan grondwater weg. Dit geeft in (9) een bijdrage aan de bronterm $f$. In deze bronterm moeten we dan aangeven hoeveel $m^3$ grondwater er per dag per $m^2$ wordt weggepompt.
Stel we hebben een rechthoek $B_P$ met $P$ als middelpunt en afmetingen $h_x$ x $h_y$. Bij het midden $P$ wordt $p$ water per dag weggepompt. Door deze hoeveelheid te verdelen over de gehele rechthoek, komen we uit op $p/(h_xh_y)$ per roosterpunt. We kunnen het effect van de pomp dus modelleren door op roosterpunt $P = (x_p, y_p)$, $p/(h_xh_y)$ bij de bronterm op te tellen.

Een rivier kunnen we beschrijven door de punten $(x, y)$ waarvoor geldt: $ax + by = c$, met $a$, $b$ en $c$ bekende constanten. Kanaal is wellicht een betere naam voor een dergelijke rivier, maar we zullen het rivier blijven noemen. Een rivier voegt $q$ $m^3/$ dag $m$ water toe aan het grondwater.
We nemen weer de zelfde rechthoek $B_P$. Stel dat precies door het middelpunt $P$ een rivier stroomt die $q$ $m^3$ water per dag per $m$ toevoegt. Als deze rivier in $B_P$ $l$ meter lang is, bedraagt de bijdrage van de rivier in het roosterpunt $P$ aan het grondwater $ql/(h_x h_y)$. Dus door deze hoeveelheid op te tellen bij de bronterm in $P$, kunnen we het effect van een rivier modelleren.

## 4.4 Foutafschatting

Door het gebruik van eindige differenties maken we fouten. In deze paragraaf willen we kort onderzoeken hoe groot deze fouten zijn.
De eindige differentie

$$\frac{\partial \psi_{i,j}}{\partial x} := \frac{\psi_{i+1,j} - \psi_{i-1,j}}{2h_x}$$

zoals al gegeven in (13) is uiteraard niet uit de lucht komen vallen. Deze benadering van de afgeleide valt af te leiden met behulp van Taylorreeksen. Beschouw de Taylorreeksen van respectievelijk $x + h_x$ en $x - h_x$:

$$\begin{aligned}\psi(x + h_x, y) &= \psi(x, y) + h_x \tfrac{\partial \psi(x,y)}{\partial x} + \mathcal{O}(h_x^2),\\ \psi(x - h_x, y) &= \psi(x, y) - h_x \tfrac{\partial \psi(x,y)}{\partial x} + \mathcal{O}(h_x^2).\end{aligned}$$

Door de laatste vergelijking af te trekken van de eerste krijgen we

$$\psi(x + h_x, y) - \psi(x - h_x, y) = 2h_x \frac{\partial \psi(x, y)}{\partial x} + \mathcal{O}(h_x^2). \tag{29}$$

Door aan beide kanten te delen door $2h$ en gebruik te maken van de ingevoerde notatie levert dit precies (13 op.
We zien dat in de gebruikte Taylorreeksen een fout van $\mathcal{O}(h_x^2)$ wordt gemaakt. De gebruikte differentie (15) is uiteraard hetzelfde, maar met een 2 keer zo kleine $h_x$. Gebruik maken van (15) levert dus een 4 keer zo kleine fout op. Voor de $y$-richting geldt uiteraard dezelfde afleiding. We kunnen dus concluderen dat we te maken hebben met een fout als gevolg van de discretisatie van $\mathcal{O}(h_x^2)$ in de $x$-richting en $\mathcal{O}(h_y^2)$ in de $y$-richting. Nauwkeuriger resultaten kunnen we dus krijgen door het verkleinen van $h_x$ en $h_y$, ofwel door het aantal roosterpunten te verhogen. Belangrijk hierbij is dat we zowel $h_x$ als $h_y$ verkleinen, en niet 1 van beide. De grootte van de fout wordt namelijk bepaald door de grootste van deze twee. Dit gezichtspunt kunnen we formuleren als: de fout (ofwel $\delta$ uit (16)) is gelijk aan $\mathcal{O}(\max\{h_x, h_y\}^2)$.
Voor de randwaarden geldt dat in het geval $\mu = 0$ (uit (10)), dus op $\Gamma_0$ dat de functiewaarden $\psi_{0,y}$ en $\psi_{n_x+1,y}$ foutloos worden berekend. Op $\Gamma_0$ maken we dus geen fouten. Op $\Gamma_1$ gebruiken we (15), wat betekent dat we een fout maken van $\mathcal{O}(h_y^2)$. Echter, we benaderen niet de afgeleide in $\psi_{i,0}$ en $\psi_{i,n_y+1}$ maar in $\psi_{i,\frac{1}{2}}$ en $\psi_{i,n_y+\frac{1}{2}}$. We maken dus een extra fout. Samen komt dit neer op een fout van $\mathcal{O}(h_y)$.
Het bovenstaande kunnen we in termen van $\delta$ samenvatten als:

$$\begin{aligned}\delta &= \mathcal{O}(h^2),\\ \delta_{0,y} &= 0,\\ \delta_{n_x+1,y} &= 0,\\ \delta_{i,0} &= \mathcal{O}(h),\\ \delta_{i,n_y+1} &= \mathcal{O}(h).\end{aligned} \tag{30}$$

## 4.5 Matrix formulering

In de vorige paragrafen hebben we voor elk roosterpunt een vergelijking opgesteld. Ook het effect van de randvoorwaarden en van pompen en rivieren is behandeld en kan nu eenvoudig in de vergelijkingen worden meegenomen. Dat betekent dat we het hele probleem hebben teruggebracht tot het oplossen van $n = n_x n_y$

onbekenden uit $n$ vergelijkingen. Een dergelijk probleem vraagt om een matrix formulering.
We kunnen de roosterpunten nummeren. We doen dit lexicografisch, d.w.z. per rij, van links naar rechts en vervolgens de rijen van onder naar boven, zie ook weer Figuur 2. Door de roosterpunten zo te nummeren, hebben we automatisch de onbekende functiewaarden $\psi_{i,j}$ genummerd. Elke functiewaarde $\psi_{i,j}$ correspondeert immers met roosterpunt $(x_i, y_j)$. Het nummer van een onbekende functiewaarde $\psi_{i,j}$ wordt expliciet gegeven door:

$$k = (i-1) \cdot n_x + j. \tag{31}$$

Nu definiëren we de vector $x$ als

$$x_k = \psi_{i,j}, \quad i = 1 \dots n_x,\ j = 1 \dots n_y. \tag{32}$$

Op dezelfde manier definiëren we de vector $b$:

$$b_k = \hat{f}_{i,j}, \quad i = 1 \dots n_x,\ j = 1 \dots n_y. \tag{33}$$

We hebben voor elk roosterpunt een vergelijking, dus hoort bij elke $x_k$ een vergelijking. Door de vector $x$ voor te vermenigvuldigen met een matrix $A$ en deze vervolgens gelijk te stellen aan $f$ kunnen we al deze vergelijkingen krijgen. In rij $k$ van matrix $A$ komt dan de vergelijking die bij roosterpunt $x_k$ hoort. Dit definieert de matrix $A$, waarin we voor het gemak de indices $i, j$ steeds hebben weggelaten (deze corresponderen met het rijnummer van de matrix):

$$A = \begin{pmatrix}
\alpha^c & \alpha^o & \dots & \alpha^n & \dots & & & & \\
\alpha^w & \alpha^c & \alpha^o & \dots & \alpha^n & \dots & & & \\
 & \ddots & \ddots & \ddots & & \ddots & & & \\
\alpha^z & \dots & \alpha^w & \alpha^c & \alpha^o & \dots & \alpha^n & \dots & \\
 & \ddots & & \ddots & \ddots & \ddots & & \ddots & \\
 & & \alpha^z & \dots & \alpha^w & \alpha^c & \alpha^o & \dots & \alpha^n \\
 & & & \ddots & & \ddots & \ddots & \ddots & \\
 & & & & \alpha^z & \dots & \alpha^w & \alpha^c & \alpha^o \\
 & & & & & \alpha^z & \dots & \alpha^w & \alpha^c
\end{pmatrix}. \tag{34}$$

In deze matrix zitten er tussen $\alpha^z$ en $\alpha^w$ en tussen $\alpha^o$ en $\alpha^n$ steeds $n_x - 1$ nullen. Verder bestaan het eerste en laatste deel van $A$ (respectievelijk zonder $\alpha^z$ en $\alpha^n$) uit precies $n_y$ rijen. Ook het middelste deel van $A$ bestaat uit blokken van $n_y$ rijen en $n_x$ kolommen. Al deze blokken zijn voor het middelste deel echter identiek. We hebben dus te maken met een matrix bestaande uit $n = n_x n_y$ entries, dus $A$ heeft afmetingen $n$ x $n$.
Nu we $A$, $x$ en $b$ hebben gedefinieerd arriveren we bij (1): $Ax = b$.

## 4.6 Matrix eigenschappen

Voor dat we oplosmethoden voor (1) gaan bestuderen, staan we eerst even stil bij een aantal eigenschappen van de matrix $A$. Enkele eigenschappen zullen belangrijk blijken voor het kiezen van de juiste oplosmethode. De matrix $A$ is een 5-diagonaal matrix. Dat betekent dat deze matrix ijl is: een matrix met overwegend nullen. Het zou een enorme geheugenverspilling zijn om al deze nullen op te slaan. Daarom maken we gebruik van een structuur in C++ die het opslaan van deze nullen overbodig maakt: de zogenaamde "row-indexed sparse storage mode". Voor specificaties, zie [2].

Een matrix is symmetrisch als geldt: $A = A^T$. Vertaald naar onze specifieke matrix $A$ betekent dat, dat $A$ symmetrisch is als voor alle $i, j$ geldt:

$$\alpha^o_{i,j} = \alpha^w_{i+1,j} \quad \text{en} \quad \alpha^n_{i,j} = \alpha^z_{i,j+1}. \tag{35}$$

We zien in (24) voor $\alpha^o_{i,j}$ en $\alpha^w_{i,j}$ respectievelijk de termen $a_{i+\frac{1}{2},j}$ en $a_{i-\frac{1}{2},j}$. Omdat $i+\frac{1}{2}$ gelijk is aan $(i+1)-\frac{1}{2}$, zien we dat voor het symmetrisch zijn van $A$, de functie $a$ geen enkele rol speelt. Eenzelfde verhaal geldt natuurlijk voor de functie $b$. En opnieuw zien we een interessant gevolg van het discretiseren in de tussenpunten!
Dit geconcludeerd hebbend, kunnen we de eisen voor het symmetrisch zijn van $A$ reduceren tot

$$u_{(i)+1,j} = -u_{(i+1)-1,j} \quad \text{en} \quad v_{i,(j)+1} = -v_{i,(j-1)+1}. \tag{36}$$

Dit leidt uiteraard tot de eis

$$u_{i+1,j} = -u_{i,j} \quad \text{en} \quad v_{i,j+1} = -v_{i,j}. \tag{37}$$

Dit wordt in het bijzonder bereikt als $u = v = 0$. En dus altijd in het grondwaterprobleem (2).
De randwaarden spelen geen enkele rol in het symmetrisch zijn van de matrix $A$, omdat bij de eliminatie alleen termen opgeteld werden bij $\alpha^c$ en bij de bronterm $\hat{f}$. Aangezien de $\alpha^c$'s de diagonaal van $A$ vormen en $\hat{f}$ de vector $b$, hebben de randwaarden geen invloed. Dit is een gevolg van de manier van discretiseren van de randwaarden.

Een matrix heet positief definiet als voor alle vectoren $x \in \mathbb{R}^n$ geldt

$$x^T A x > 0, \tag{38}$$

wat equivalent is met uitsluitend positieve eigenwaarden. Voor symmetrische matrices geldt dat de matrix in het bijzonder positief definiet is als de matrix diagonaal dominant is. Het diagonaal dominant zijn van een matrix betekent concreet:

$$\forall i : |a_{i,i}| \geq \sum_{j \neq i} |a_{i,j}| \tag{39}$$

Voor onze matrix $A$ kunnen we dit verder uitwerken als:

$$\forall i : |\alpha^c_{i,j}| \geq |\alpha^w_{i,j}| + |\alpha^o_{i,j}| + |\alpha^z_{i,j}| + |\alpha^n_{i,j}| \tag{40}$$

We hebben het over symmetrische matrices en gaan er daarom vanuit dat $u = v = 0$. Dit leidt tot

$$|a_{i+\frac{1}{2},j} + a_{i-\frac{1}{2},j} + b_{i,j+\frac{1}{2}} + b_{i,j-\frac{1}{2}} + c_{i,j}| \geq |a_{i+\frac{1}{2},j}| + |a_{i-\frac{1}{2},j}| + |b_{i,j+\frac{1}{2}}| + |b_{i,j-\frac{1}{2}}|. \tag{41}$$

De doorlaatbaarheidscoëfficiënten $a$ en $b$ en de afbraakcoëfficiënt $c$ zullen niet negatief zijn. Vanuit deze aanname kunnen we concluderen dat een symmetrische matrix $A$ voor ons probleem altijd diagonaal dominant, en dus positief definiet is. Dit onder de zeer reële aannames dat $u = v = 0$, $a \geq 0$, $b \geq 0$ en $c \geq 0$.
We hebben ook nog te maken met de geëlimineerde randwaarden. Door deze eliminatie werden er termen opgeteld bij $\alpha^c_{i,j}$. Deze termen hangen af van $a$ of $b$ en van $h_x$ of $h_y$, wat betekent dat deze termen niet negatief zijn. We kunnen de getrokken conclusie dus handhaven.

# 5 Iteratieve Oplosmethoden

## 5.1 Motivatie

Voor het oplossen van (1) zijn vele methoden bekend. We kunnen deze methoden opdelen in twee groepen: directe methoden en iteratieve methoden. Directe methoden zijn bijvoorbeeld de bereking van $A^{-1}$, berekening van een $QR$-decompositie ($Q$ een orthonormale matrix, $R$ een bovendriehoeksmatrix) of de berekening van een $LU$-decompositie (met $L$ een benedendriehoeksmatrix en $U$ een bovendriehoeksmatrix). Waarna het eigenlijke oplosproces simpel is.
Hebben we eenmaal $A^{-1}$, een $QR$-decompositie of een $LU$-decompositie, dan kunnen we voor elke willekeurige vector $b$ de vergelijking (1) efficiënt oplossen. Echter, wij zijn slechts geïnteresseerd in $Ax = b$ met 1 enkele, vaststaande $b$.
Anders gezegd: de matrix $A^{-1}$ is een afbeelding van $\mathbb{R}^n$ naar $\mathbb{R}^n$ terwijl wij slechts geïnteresseerd zijn in de restrictie van die afbeelding: $A^{-1} : V \to \mathbb{R}^n$. Waarin $V$ een deelruimte is, opgespannen door $b$: $V = \text{span}(b)$.
Directe methoden doen dus meer dan wij eigenlijk nodig hebben. Voor grote matrixdimensie $n$ kan de hoeveelheid overbodig werk enorm groot worden. Dit kunnen we verder speciferen aan de hand van de matrix $A$ voor ons probleem. Daartoe vergelijken we de kosten van $LU$-decompositie met de kosten van een matrix-vector vermenigvuldiging (MV). Iteratieve methoden maken namelijk slechts gebruik van een aantal MV's en wat andere goedkope operaties, zoals zal blijken uit de volgende paragrafen.
De matrix $A$ uit hoofdstuk 4 is ijl, het is een 5-diagonaal matrix. Bij $LU$-decompositie gaat deze ijlheid helaas verloren, we krijgen te maken met zogenaamde 'fill-in'. In plaats van 5 niet-nullen per rij, zullen $L$ en $U$ ieder ongeveer $n_x$ niet-nullen per rij bevatten. Dit gaat dus meer geheugen kosten. Bovendien kost het berekenen van de $L$- en $U$-factoren aanzienlijk meer flops dan een MV. Onderstaand overzicht geeft de kosten (in flops) en het benodigde geheugen voor $LU$-decompositie en een MV weer.

| | Geheugen | Kosten |
|---|---|---|
| $MV$ | $5n_xn_y$ | $10n_xn_y$ |
| $LU$ | $2n_x^2n_y$ | $n_x^3n_y$ |

*kosten 2-dimensionaal*

We zien dat $LU$-decompostie een factor $n_x$ aan geheugen en een factor $n_x^2$ aan flops meer kost. Genoeg motivatie om na te denken over methoden die zich beperken tot het oplossen van (1) voor een vaste $b$: de iteratieve methoden. Ter illustratie geven we nog een analogie van Tabel 1, maar dan voor het 3-dimensionale probleem. Het zal de motivatie alleen maar vergroten. We hebben dan in plaats van een 5-diagonaal matrix een 7-diagonaal matrix $A$. Met $n_z$ het aantal punten in de $z$-richting (discretisatie van de hoogte) geldt:

| | Geheugen | Kosten |
|---|---|---|
| $MV$ | $7n_xn_yn_z$ | $14n_xn_yn_z$ |
| $LU$ | $2(n_xn_y)^2n_z$ | $(n_xn_y)^3n_z$ |

*kosten 3-dimensionaal*

Men heeft experimenteel aangetoond dat voor 2-dimensionale problemen de iteratieve methoden vanaf ongeveer $n = 60000$ (denk aan een rooster van bv. 300 x 200) goedkoper zijn. Voor 3-dimensionale problemen ligt dit omslagpunt lager (vanwege de 7 i.p.v. 5 diagonalen), op ongeveer $n = 40000$ (bv. 35 x 35 x 35)!

## 5.2 De idee

Stel $A$ is een 1 x 1 matrix, $A = (\lambda)$ en $b = \beta$. Dan hebben we de vergelijking $\lambda x = \beta$. We willen deze vergelijking oplossen voor $x$. Stellen we nu $\alpha = \frac{1}{\lambda}$, dan geldt:

$$x = x + \alpha(\beta - \lambda x). \tag{42}$$

We willen echter geen delingen gebruiken (vanwege de filosofie dat vermenigvuldigen goedkoper is), en nemen $\alpha$ daarom niet gelijk aan $\frac{1}{\lambda}$, maar nemen daarvoor een willekeurige scalar. Door nu een $x_0$ te kiezen (meestal

$x_0 = 0$), kunnen we de volgende iteratie uitvoeren:

$$x_{k+1} = x_k + \alpha(\beta - \lambda x_k), \quad k = 0, 1, \ldots. \tag{43}$$

Dit is een benaderingsalgoritme welke onder de juiste condities naar het goede antwoord convergeert. We definiëren het residu als

$$r_k = \lambda x_k - \beta. \tag{44}$$

Convergentie treedt nu op als

$$r_{k+1} < r_k \quad \forall k. \tag{45}$$

We kunnen $r_{k+1}$ uitdrukken in zijn voorganger als $r_{k+1} = (1 - \alpha\lambda)r_k$. Dit betekent dat de voorwaarde voor convergentie gegeven wordt door

$$|1 - \lambda\alpha| < 1. \tag{46}$$

De analogie van bovenstaand iteratieproces voor de meerdimensionale situatie is de zogenaamde Richardson iteratie:

$$x_{k+1} = x_k + \alpha r_k, \quad \text{met} \quad r_k = b - Ax_k. \tag{47}$$

Met dit iteratieproces vermijden we het berekenen van $A^{-1}$. We maken slechts gebruik van zogenaamde AXPY's ($\alpha x + y$), DOT's ($x^T y$) en MV's ($Ax$). We kunnen het iteratieproces laten stoppen als $||r_k|| \leq tol$ ($||\cdot||$ is, tenzij anders vermeld, de Euclidische norm) waarbij *tol* een gegeven tolerantie is.
De analogie van de eis voor convergentie luidt $|I - A\alpha| < I$, wat weer equivalent is met

$$\forall\lambda : |1 - \lambda\alpha| < 1, \tag{48}$$

waarin de $\lambda$'s de eigenwaarden van de matrix $A$ zijn.
We suggereerden dat $\alpha$ een willekeurige scalar is. Maar we kunnen meer doen met deze $\alpha$, we kunnen daarmee de convergentiesnelheid beïnvloeden. De snelste convergentie krijgen we voor die $\alpha$ waarvoor $(1 - \max_\lambda |1 - \alpha\lambda)$ maximaal is. Deze $\alpha$ is natuurlijk in de praktijk onmogelijk te vinden, aangezien de $\lambda$'s onbekend zijn. Het is echter wel mogelijk om afhankelijk van $k$, in iedere stap de beste $\alpha$ te kiezen. We willen dan de $\alpha$ waarvoor $r_{k+1}$ minimaal is.
Het is makkelijk in te zien dat voor het nieuwe residue $r_{k+1}$ geldt:

$$r_{k+1} = r_k - \alpha A r_k. \tag{49}$$

De optimale $\alpha$ is nu die $\alpha$ waarvoor $||r_k - \alpha A r_k||$ minimaal is. Met $c_k = Ar_k$ is deze term minimaal als $\alpha c_k$ de orthogonale projectie is van $r_k$ op $c_k$. Dan volgt

$$\alpha c_k - r_k \perp c_k \Leftrightarrow (c_k, \alpha c_k - r_k) = 0 \Leftrightarrow \alpha c_k^T c_k = c_k^T r_k \Leftrightarrow \alpha = \frac{c_k^T r_k}{c_k^T c_k}. \tag{50}$$

De Euclidische norm van de orthogonale projectie van $r_k$ op $c_k$ is altijd kleiner dan de norm van $r_k$ zelf. Met behulp van deze optimale $\alpha$ kunnen we dus altijd convergentie afdwingen. We kunnen nog snellere convergentie afdwingen door het residu niet alleen te minimaliseren ten opzichte van de laatste correctievector $c_k$, maar ten opzichte van alle voorgaande correctievectoren. Het algoritme wat dan ontstaat heet GCR.

## 5.3 GCR

GCR (Generalized Conjugate Residuals) bepaalt de benadering $x_k$ van $x$ door het residu te minimaliseren in de zogenaamde Krylov deelruimte $\mathcal{K}_{k+1}(A; r_0)$. Deze Krylov deelruimte is gedefinieerd als

$$\mathcal{K}_{k+1}(A; r_0) := \text{span}(r_0 | Ar_0 | A^2 r_0 | \ldots | A^k r_0 |).$$

Als $x_0 = 0$ geldt nu dat $x_k$ behoort tot $\mathcal{K}_k(A; r_0)$. Het residu $r_k$, is dan gelijk aan $r_0 - Ax_k$. Deze is nu minimaal in $\mathcal{K}_{k+1}(A; r_0)$ als

$$||r_k|| = \min\{||r_0 - v|| : v \in A\mathcal{K}_k(A; r_0)\}.$$

```
Choose an x_0.
Compute r_0 = b − Ax_0.
For k = 0, 1, 2, ..., k_max do
    stop if ‖r_k‖ ≤ tol
    u_k = r_k
    compute c_k = Au_k
    for i = 0, ..., k − 1 do
        β_{i+1} = c_i^T c_k / σ_i
        u_k = u_k − β_{i+1} u_i
        c_k = c_k − β_{i+1} c_i
    end for
    σ_k = c_k^T c_k ,  α_k = c_k^T r_k / σ_k
    x_{k+1} = x_k + α_k u_k
    r_{k+1} = r_k − α_k c_k
end for
```

ALGORITME 1: *Generalized Conjugate Residuals met residuale nauwkeurigheid tol en maximaal aantal iteraties* $k_{max}$.

En dit geldt vanwege het idee van projectie precies als $r_k \perp A\mathcal{K}_k(A; r_0)$. Dit motiveert om iteratief een orthogonale basis van $A\mathcal{K}_k(A; r_0)$ te vinden, waarna $r_{k+1}$ berekend kan worden als het orthogonale complement van $r_0$ op deze deelruimte:

$$r_{k+1} = r_0 - \alpha_0 c_0 - \ldots - \alpha_k c_k = r_k - \alpha_k c_k,$$

met $\alpha$ als in (50) en $c_k = Au_k$, als $u_k$ de zoekrichting voor de benadering $x_{k+1}$ (in GCR gelijk aan $r_k$) is. Met de recursie als beschreven in 47 komen we op Algoritme 1: GCR.
GCR heeft de C in zijn naam te danken aan de eigenschap dat de residuen geconjugeerd zijn:

$$r_{k+1} \perp Ar_j \quad j = 0 \ldots k. \tag{51}$$

De G van GCR komt vanwege de toepasbaarheid van de methode op alle matrices (en niet alleen op symmetrische).

### 5.3.1 Kostenanalyse

In het begin van dit hoofdstuk hebben we beweerd met iteratieve methoden goedkoper te kunnen zijn. Nu we een iteratief algoritme hebben kunnen we de kosten voor GCR beter bestuderen.
In GCR wordt een lus van $k$ stappen doorlopen waarin gebruik wordt gemaakt 3 DOT's, 2 AXPY's, 1 ID en 1 MV en van een binnenlus. Deze binnenlus bestaat weer uit een aantal basisoperaties en wordt $\sum_{i=1}^{k-1} i = \frac{1}{2}k(k-1)$ maal uitgevoerd. Het aantal basisoperaties in de binnenlus bestaat uit 1 DOT en 2 AXPY's. Aangezien DOT's AXPY's, ID's en MV's respectievelijk $2n$, $2n$, $n$ en $10n$ flops kosten komen we uit op

$$21n + 3(k-1)n\text{flops} \tag{52}$$

aan kosten per iteratieslag. Dit is dus $\mathcal{O}(k^2 n)$ kosten voor het hele GCR algoritme. In paragraaf 5.1 hadden we al gezien dat we met $LU$-decompositie uitkomen op $\mathcal{O}(n_x^2 \cdot n)$, waarna we kunnen conluderen dat voor $n_x > k$ het raadzaam is over te stappen op iteratieve methoden. Natuurlijk weet je van tevoren nooit precies de waarde van $k$. Desalniettemin is het hoopvol dat GCR niet kwadratisch afhangt van het aantal roosterpunten.

Omdat we alle vectoren $u_i$ en $c_i$ opslaan, geldt dat de benodigde geheugenruimte voor GCR evenredig

is met $k$ (om precies te zijn $2k+2$). Elk van deze vectoren is $n$ lang. Verder moeten we natuurlijk de matrix $A$ opslaan, waarvoor het geheugengebruik $5n$ is (zie paragraaf 5.1), en de vectoren $x$, $b$ en $r$ die $n$ doubles aan geheugen nodig hebben. De benodigde geheugenruimte is dus al met al $\mathcal{O}(kn)$.
Voor de $LU$-decompositie hadden we een geheugengebruik van $2nn_x$, dus $\mathcal{O}(n_x \cdot n)$. We zien dat we voor het geheugengebruik dezelfde conclusie kunnen trekken als voor de kosten: voor $n_x > k$ zijn iteratieve methoden aantrekkelijker.

## 5.4 Preconditionering

De convergentie van een iteratieve methode hangt af van de distributie van de eigenwaarden. Een groot verschil tussen de kleinste en de grootste eigenwaarde betekent slechte convergentie. Het quotiënt van grootste- en kleinste eigenwaarde noemt men het conditiegetal van een matrix. Hoe groter het conditiegetal, hoe slechter de convergentie van de iteratieve methode zal zijn. Daarom zou het interessant zijn om dit conditiegetal te kunnen beïnvloeden. Dit is precies wat preconditionering doet: door middel van vermenigvuldigen van het systeem met een matrix $M$ hopen we de eigenwaarden zodanig te herdistribueren dat het conditiegetal kleiner wordt, en we dus betere convergentie krijgen.
Bovenstaande is een abstracte benadering van preconditionering. In de rest van dit hoofdstuk zal op een meer inzichtelijke manier worden ingegaan op preconditonering.

### 5.4.1 Impliciet preconditioneren

De verbetering van benadering $x_k$ tot $x_{k+1}$ wordt gezocht in de zoekrichting $u_k$. Derhalve is deze zoekrichting van groot belang voor de snelheid van de convergentie. De snelste convergentie krijgen we, als we als zoekrichting de oplossing van de vergelijking $Au_k = r_k$ nemen. In dat geval geldt:

$$x_{k+1} = x_k + A^{-1}r_k = x_k + A^{-1}(b - Ax_k) = A^{-1}b = x,$$

dus zijn we met het gebruik van deze zoekrichting in 1 iteratie klaar. Helaas is het oplossen van $Au_k = r_k$ net zo moeilijk als het oplossen van het originele probleem. Toch kunnen we onze zoekrichting verbeteren, door i.p.v. $A$ een andere matrix $M$ te nemen, die op $A$ lijkt en waarvoor we de vergelijking $Mu_k = r_k$ makkelijk kunnen oplossen. De zoekrichting $u_k$ wordt dan gegeven door de oplossing van de vergelijking $Mu_k = r_k$. Door de regel $u_k = r_k$ in GCR te vervangen door solve $Mu_k = r_k$ hebben we GCR gepreconditioneerd met matrix $M$. Zie Algoritme 2: PGCR.
Omdat we de preconditionering binnen de lus van GCR verwerken, spreken we van impliciet preconditioneren.

### 5.4.2 Expliciet preconditioneren

We kunnen ook buiten de GCR lus preconditioneren, wat we dan expliciet preconditioneren noemen.
Daartoe is het belangrijk te realiseren dat $r_k$, het residu van gepreconditioneerd GCR minimaal is in de Krylov ruimte $\mathcal{K}_{k+1}(AM^{-1}; r_0)$. Bekijk namelijk het volgende stelsel:

$$AM^{-1}y = b, \tag{53}$$

dan geldt: $x = M^{-1}y$, met $y$ de exacte oplossing van (53) en $x$ de exacte oplossing van (1). Passen we nu GCR toe op (53) en vermenigvuldigen we daarna $y_k$ met $M^{-1}$, dan ontstaat precies Algoritme 2. Dit achteraf vermenigvuldigen met $M^{-1}$ om terug te werken naar $x$ noemen we het post-proces, het vooraf vermenigvuldigen van $x$ met $AM^{-1}$ noemen we het pre-proces.
De in (53) voorgestelde preconditionering heet expliciet rechts preconditioneren. We kunnen ook expliciet links preconditoneren, door

$$M^{-1}Ax = M^{-1}b. \tag{54}$$

We hoeven dan niet te post-processen, omdat we nog gewoon $x$ oplossen. Een nadeel is echter dat je werkt met gepreconditioneerde residuen, waardoor je niet zeker weet of je stop-criterium wel is behaald. Anderen

```
Choose an x_0.
Compute r_0 = b − Ax_0.
For k = 0, 1, 2, ..., k_max do
    stop if ‖r_k‖ ≤ tol
    solve Mu_k = r_k
    compute c_k = Au_k
    for i = 0, ..., k − 1 do
        β_{i+1} = c_i^T c_k / σ_i
        u_k = u_k − β_{i+1} u_i
        c_k = c_k − β_{i+1} c_i
    end for
    σ_k = c_k^T c_k ,  α_k = c_k^T r_k / σ_k
    x_{k+1} = x_k + α_k u_k
    r_{k+1} = r_k − α_k c_k
end for
```

ALGORITME 2: *Preconditioned* GCR.

zien dit echter als voordeel omdat $M^{-1}$ lijkt op $A^{-1}$ en

$$A^{-1}r_k = A^{-1}(Ax_k - b) = x_k - A^{-1}b = x_k - x.$$

Men hoopt dus op deze manier convergentie te krijgen van de fout naar 0, in plaats van convergentie van het residu. En uiteindelijk is de fout natuurlijk interessanter dan het residu. Omdat $M$ slechts een benadering is van $A$, is het nog maar de vraag of dit wel werkelijk interessant is. Wij zullen van deze laatste eigenschap verder geen gebruik maken.
Tenslotte kunnen we ook nog tweezijdig preconditioneren met $M = M_1 M_2$:

$$M_1^{-1} A M_2^{-1} M_2 y = M_1^{-1} b, \quad x = M_2^{-1} y. \tag{55}$$

### 5.4.3 Preconditioneerders

De vraag dringt zich nu op welke precondtioneerders we kunnen gebruiken, ofwel: wat nemen we voor $M$? We zullen de volgende notatie gebruiken:

- $L_A$ is het strikte benedendriehoeksdeel van $A$;
- $D_A$ is de diagonaal van $A$;
- $U_A$ is het strikte bovendriehoeksdeel van $A$,

waarna geldt: $A = L_A + D_A + U_A$.

Een voor de hand liggende keus voor M, waarmee heel efficiënt gerekend kan worden is

$$M = D_A, \tag{56}$$

de zogenaamde Jacobi-preconditionering. Een andere bekende en eenvoudige optie is de Gauss-Seidel variant

$$M = L_A + D_A, \tag{57}$$

waarna een simple uitbreiding ons brengt op SOR:

$$M = \frac{1}{\omega} L_A + D_A. \tag{58}$$

We zullen ook wat ingewikkelder preconditioneerders van de vorm

$$M = (D + L_A)D^{-1}(D + U_A) \tag{59}$$

onderzoeken. Hierin is $D$ een diagonaalmatrix, die hierna verder gespecificeerd zal worden. Eerst merken we nog op dat door de structuur van $M$ in (59), de vergelijking $Mu_k = r_k$ efficiënt is op te lossen. Dit komt slechts neer op het oplossen van boven- en benedendriehoeksstelsels en diagonaalstelstels. Deze preconditioneerder kan dus worden gebruikt voor impliciete preconditionering.
Definieer

$$R := A - M = D_A - D - L_A D^{-1} U_A. \tag{60}$$

Omdat $M$ behoort te lijken op $A$ geldt $R \approx 0$, ofwel $R$ moet 'klein' zijn. $R$ kan op verschillende manieren klein zijn:

- ILU : diag($R$)=0 (ofwel diag($M$)=diag($A$))
- MILU: $R \cdot 1 = 0$ (ofwel $M \cdot 1 = A \cdot 1$), waarbij we 1 zien als de vector met op iedere coördinaat een 1, $\cdot$ staat hier voor een MV.
- RILU: combinatie van ILU en MILU, bepaald door een paramater $\omega$.

Door de structuur van $A$ (zoals uitgewerkt in paragraaf 4.5) kunnen we de diagonaal elementen $D_k := D_{kk}$ van $D$ berekenen met behulp van de volgende iteratie:

$$D_k = A_{kk} - \left( \frac{A_{k,k-1}A_{k-1,k}}{D_{k-1}} + \frac{A_{k,k-n_x}A_{k-n_x,k}}{D_{k-n_x}} \right) - \omega \left( \frac{A_{k,k-1}A_{k-1,k+n_x-1}}{D_{k-1}} + \frac{A_{k,k-n_x}A_{k-n_x,k-n_x+1}}{D_{k-n_x}} \right). \tag{61}$$

Keuze van $\omega = 0$ leidt tot ILU en met $\omega = 1$ hebben we de MILU benadering. Een 'magische' waarde is $\omega \approx 0.95$. Maar de optimale waarde hangt af van het probleem.
De ILU, MILU en RILU benaderingen zijn voorbeelden van zogenaamde incomplete decomposities. Het idee bij deze decomposities is het berekenen van de $LU$-decompositie en (een deel van) de 'fill-in' te negeren.
We zullen vanaf nu ons concentreren op diagonaal preconditionering, en impliciet- en expliciet preconditionering met (59)

### 5.4.4 Efficiënt rekenen

In deze paragraaf gaan we de preconditioneerder (59) gebruiken met behulp van tweezijdige preconditionering (55). Een combinatie van deze twee leidt tot een erg efficiënte rekenmethode.
We nemen $M$ dus als in (59). We gaan nu de tweezijdige preconditioneerder opbouwen in twee stappen:

1. Diagonale preconditionering:
$$D^{-1}Ax = D^{-1}b. \tag{62}$$
Waarna we de resulterende matrix $\tilde{A} := D^{-1}A$ verder preconditioneren als

2. $$(I + \tilde{L}_A)^{-1}\tilde{A}(I + \tilde{U}_A)^{-1}\tilde{x} = \tilde{\tilde{b}}, \tag{63}$$
met
$$y = (I + \tilde{U}_A)^{-1}\tilde{x} \quad \text{en} \quad \tilde{\tilde{b}} = (I + \tilde{L}_A)^{-1}\tilde{b}$$

In feite verdelen we de matrix $\tilde{M}^{-1}$, met $\tilde{M} = (I + \tilde{L}_A)^{-1}(I + \tilde{U}_A)^{-1}$ over de linker- en rechterkant van $A$.
We hebben nu een tweezijdig expliciet gepreconditioneerd systeem, wat we kunnen oplossen met behulp van GCR. De berekening van $c_k$ in dit algoritme kan met de verkregen preconditioneerder zeer efficiënt worden uitgevoerd.
Er geldt nu in GCR

$$c_k = (I + \tilde{L}_A)^{-1}\tilde{A}(I + \tilde{U}_A)^{-1}u_k. \tag{64}$$

Uiteraard geldt $\tilde{A} = \tilde{D}_A + \tilde{L}_A + \tilde{U}_A$, waarna we de matrix $\tilde{A}$ in (64) kunnen uitschrijven in zijn factoren. Met nog wat optellen en aftrekken van eenheidsmatrices leidt dit tot

$$(I + \tilde{L}_A)^{-1}\tilde{A}(I + \tilde{U}_A)^{-1} = (I + \tilde{L}_A)^{-1} + (I + \tilde{U}_A)^{-1} + (I + \tilde{L}_A)^{-1}(\tilde{D}_A - 2I)(I + \tilde{U}_A)^{-1}.$$

We kunnen dit nu gebruiken om tot de volgende methode te komen om $c_k$ te berekenen uit $u_k$:

$$\begin{array}{ll} \text{solve} & (I + \tilde{U}_A)^{-1}u' = u_k; \\ \text{compute} & u'' = u_k + (\tilde{D}_A - 2I)u'; \\ \text{solve} & (I + \tilde{L}_A)^{-1}u''' = u'; \\ \text{compute} & c_k = u' + u'' \end{array} \tag{65}$$

Deze methode staat bekend als de Eisenstat truuk.
Regel 1 en 3 uit (65) kosten ieder 2 AXPY's (vanwege de speciale diagonaal structuur van $A$), regel 2 kost 1 AXPY en regel 4 kost 1/2 AXPY. Aangezien een AXPY $2n$ flops kost, kost de hele Eisenstat truuk $11n$ flops. De berekening van $Au_k$ kost $10n$ flops, en het oplossen van $Mu_k = r_k$ (wat met Eisenstat kan worden overgeslagen) kost nog eens minstens $n$ flops (in geval $M = I$). We zien dat we met behulp van de Eisenstat truuk gratis (!) tweezijdig expliciet (met $M$ uit (59)) kunnen preconditoneren. Afgezien natuurlijk van de pre- en post-proces fase.

### 5.4.5 Kostenanalyse

Uiteraard willen we weten wat preconditionering kost. Diagonaal preconditionering kost in elke iteratiestap $n$ flops, vanwege het oplossen van het diagonaalstelsel.
Impliciet preconditioneren levert i.p.v. een ID, een solve op. Deze solve kost $11n$ flops. Het verschil met ongepreconditioneerd GCR bedraagt dus $10n$ flops.
Expliciet preconditioneren hebben we reeds beken bij de Eisenstat truuk. Ten opzichte van gewoon GCR kost dit helemaal niets meer.
Het extra geheugengebruik als gevolg van preconditionering bestaat uit het opslaan van de matrix $M$. Deze matrix hoeft niet perse ijl te zijn (bij (59) is dat zeker niet het geval). Het extra geheugengebruik is daarom $\mathcal{O}(n^2)$.

## 5.5 Korte recursies

Naarmate de recursie in GCR vordert, wordt het algoritme langzamer en vergt het meer geheugen. We hadden in paragraaf 5.3 al gezien dat de kosten kwadratisch en het geheugengebruik evenredig afhangen van $k$. Voor grote $k$ kan dit dus nog aardig de pan uit rijzen. Een voor de hand liggende manier om dit in te perken, is het gebruik van kortere recursies. In plaats van de residuen loodrecht te conjugeren ten opzichte van alle vorige residuen, kunnen we dit ook doen voor slechts een aantal vorige residuen. We hebben dan natuurlijk geen optimale zoekrichting meer, maar we hopen dat het nog steeds goed is.

Om kortere recursies te krijgen, vervangen we de regel
"for $i = 0, \ldots k-1$ do"door "for $i = j(k), \ldots k-1$ do". We kunnen dit op verschillende manieren invullen:

- $j(k) = l_1\lfloor k_1/l_1 \rfloor$: herstarten na $l_1$ stappen, we herstarten GCR met als $x_0$ de laatst verkregen benadering.
- $j(k) = \max(0, k - l_2)$: afkappen, alleen de laatste $l_2$ vectoren worden meegenomen in het orthogonalisatie proces.
- herstarten of afkappen, en bovendien de regel "$u_k = r_k$"vervangen door "solve $Au_k = r_k$ by $l_3$ steps of GCR": nesten. Het bepalen van de zoekrichting wordt uitgevoerd met GCR, dus GCR binnen GCR.

De methode van nesten in combinatie met herstarten wordt GMRESR($l_1, l_3$) genoemd. Een betere benaming zou GCRr($l_1, l_3$) zijn, aangezien we hier werken met GCR en niet met het populaire GMRES algoritme. Voor het GMRES algoritme zie bijvoorbeeld [3].
Bij afkappen worden steeds de laatste $l_2$ vectoren meegenomen. Bij herstarten daarentegen wordt elke keer opnieuw begonnen met 0 tot $l_1$ vectoren. Afkappen lijkt dus een snellere convergentie te leveren dan herstarten. Voor alle drie de strategieen geldt dat ze tot goedkopere stappen leiden. Echter in GCR zijn we zeker van convergentie, omdat GCR minimale residuen produceert. Met de kortere recursies zijn we deze zekerheid kwijt. De kans op snelle convergentie stijgt naarmate de waarde van $l_i$ groter wordt.
Typische waarden voor $l_1$,$l_2$ en $l_3$ zijn: $l_1 = l_2 = l_3 = 10$.

### 5.5.1 Kostenanalyse

We wilden de kosten reduceren t.o.v. GCR. Nu willen we ook de kosten van de drie strategieen vergelijken. Daartoe vergelijken we de gemiddelde hoeveelheid werk per iteratiestap. We middelen dit, omdat bij herstarten de hoeveelheid werk verschilt per iteratie.

**Herstarten** Voor de start van de binnenste lus maken we gebruik van 1 ID en 1 MV, na de binnenste lus hebben we 2 DOT's en 2 AXPY's nodig. In de binnenste lus is sprake van 1 DOT en 2 AXPY's. Deze binnenste lus wordt in totaal $\frac{1}{2}l_1(l_1+1)$ keer uitgevoerd. Gemiddeld over $l_1$ iteratieslagen wordt dit voor 1 iteratie

$$\text{solve} + 3\text{DOT} + 2\text{AXPY} + \frac{1}{2}(l_1 - 1)(\text{DOT} + 2\text{AXPY}). \tag{66}$$

**Afkappen** De buitenste lus is hetzelfde als bij herstarten, dus deze kosten zijn gelijk. Voor de binnenste lus zijn de kosten eveneens gelijk. Het verschil is dat deze binnenste lus elke keer $l_2$ maal uitgevoerd wordt. Per iteratieslag hebben we dus aan kosten:

$$\text{solve} + 3\text{DOT} + 2\text{AXPY} + l_2(\text{DOT} + 2\text{AXPY}). \tag{67}$$

**Nesten** Bij deze methode kunnen we de kosten op 2 manieren bekijken. We kunnen de kosten per buitenste iteratieslag bekijken of de totale kosten middelen over het totale aantal iteratieslagen $l_1 \cdot l_3$. De laatste optie lijkt ons het meest eerlijk, waardoor we op dezelfde kosten komen als voor herstarten.

Omgezet in flops levert dit (met $11n$ flops voor solve):

- Herstarten: $17n + 3(l_1 - 1)n$ flops;
- Afkappen: $17n + 6l_2n$ flops;
- Nesten: $17n + 3(l_1 - 1)n$ flops.

Aannemend dat $l_1 = l_2$ kunnen we concluderen dat herstarten de goedkoopste strategie is. Of dit ook de beste is, hangt af van de convergentie die, zoals beargumenteerd, naar verwachting juist bij afkappen beter zal zijn.

Het geheugengebruik wordt natuurlijk minder als we gebruik maken van kortere recursies. In plaats van $\mathcal{O}(kn)$ krijgen we voor herstarten $\mathcal{O}(l_1n)$ en voor afkappen $\mathcal{O}(l_2n)$ aan geheugengebruik. Voor nesten hebben we voor de $l_3$ extra slagen, extra vectoren nodig. Dit levert $\mathcal{O}(l_1l_3n)$.

## 5.6 Symmetrische problemen

In paragraaf 4.6 hebben we gezien wanneer de matrix $A$ symmetrisch is. Als dit het geval is, kunnen we het algoritme GCR een stuk efficiënter maken. De uitkomst van sommige rekenstappen in GCR is voor symmetrische problemen namelijk al van te voren bekend.

### 5.6.1 CR

Voor het gemak nemen we even $M = I$, ofwel we gaan uit van ongepreconditioneerd GCR. We kunnen de binnenste lus van GCR nu aanpassen. Er geldt in GCR:

$$c_k = Ar_k - \beta_1 c_0 - \ldots - \beta_k c_{k-1}, \quad \beta_{j+1} = c_j^T Ar_k/\sigma_j.$$

Vanwege de symmetrie van $A$ geldt $c_j^T Ar_k = (A^T c_j)^T r_k = (Ac_j)^T r_k$. Omdat $r_k \perp Ac_j$ is deze term gelijk aan 0 en dus geldt voor $j < k$

$$\beta_j = 0. \tag{68}$$

We hoeven dus in stap $k$ de $\beta_j$ voor $j < k$ niet meer uit te rekenen, en hebben daarom ook de $c_j$ en $u_j$ voor $j < k-1$ niet meer nodig. We hebben dus geen binnenste lus meer nodig, dit levert een behoorlijke besparing op.
Definiëren we

$$\rho_k = r_k^T Au_k,$$

dan geldt voor $\beta_k$

$$\beta_k = \frac{\rho_k}{\rho_{k-1}},$$

en voor $\alpha_k$

$$\alpha_k = \frac{\rho_k}{\sigma_k}.$$

Hierdoor kunnen we per stap nog een extra DOT besparen. Het algoritme wat we nu hebben afgeleid is Conjugate Residuals (CR), zie Algoritme 3.
Voor CR geldt in principe dat de preconditioneerder $M$ zo moet zijn dat $AM^{-1}$ symmetrisch is. Echter als $M$ symmetrisch en positief definiet is (zie weer paragraaf 4.6) kunnen we, door met een aangepast inproduct te werken, de eis $AM^{-1}$ symmetrisch vergeten. We kunnen dan een $M^{-1}$ orthogonale basis van $c_j$'s opbouwen. We nemen dus als inproduct het $M^{-1}$-inproduct

$$(x, y)_{M^{-1}} := (M^{-1}x, y). \tag{69}$$

We krijgen dan de uitdrukkingen $r_k^T M^{-1} c_k$ (i.p.v. $r_k^T c_k$) en $c_k^T M^{-1} c_k$ (i.p.v. $c_k^T c_k$).
Ten opzichte van dit inproduct is $AM^{-1}$ precies symmetrisch, dus deze eis kan vervallen.

ALGORITME 3: *Conjugate Residuals*

Choose an $\mathbf{x}_0$.
Compute $\mathbf{r} = \mathbf{b} - \mathbf{A}\mathbf{x}$.
Put $\mathbf{u} = \mathbf{c} = 0$, $\mathbf{x} = \mathbf{x}_0$, $\rho = 1$.
For $k = 0, 1, 2, \ldots, k_{\max}$ do
  stop if $\|\mathbf{r}\| \leq tol$
  solve $\mathbf{M}\mathbf{u}' = \mathbf{r}$
  compute $\mathbf{c}' = \mathbf{A}\mathbf{u}'$
  $\rho' = \rho$, $\rho = \mathbf{r}^T\mathbf{c}'$, $\beta = -\rho/\rho'$
  $\mathbf{u} = \mathbf{u}' - \beta\,\mathbf{u}$
  $\mathbf{c} = \mathbf{c}' - \beta\,\mathbf{c}$
  $\sigma = \mathbf{c}^T\mathbf{c}$, $\alpha = \rho/\sigma$
  $\mathbf{x} = \mathbf{x} + \alpha\,\mathbf{u}$
  $\mathbf{r} = \mathbf{r} - \alpha\,\mathbf{c}$
end for

### 5.6.2 CG

Als $A$ positief definiet is, kunnen we het proces nog verder versnellen. We kunnen nu het standaard inproduct vervangen door het $A^{-1}$-inproduct. Gebruiken we verder nog $A^{-1}c_k = u_k$, dan geldt:

$$\rho_k = r_k^T A^{-1} c_k = r_k^T u_k, \tag{70}$$
$$\sigma_k = c_k^T A^{-1} c_k = c_k^T u_k. \tag{71}$$

Dit scheelt dus 2 MV's in kosten. Verder geldt in geval $M = I$, dat $\rho_k = r_k^T r_k$, we krijgen dus de norm van $r_k$ cadeau, zonder extra inproduct.
Voor de berekening van $\rho_k$ hebben we niet meer $c_k$ nodig, waardoor we de AXPY voor het updaten van $c_k$ kunnen besparen. We kunnen $c_k$ dan eenvoudig berekenen volgens $c_k = Au_k$. Met deze besparingen hebben we het Conjugate Gradient (CG) algoritme, zie Algoritme 4 (met impliciete preconditionering).
Wel gelden voor CG hardere eisen. Nu moet $A$ niet alleen symmetrisch zijn, maar ook positief definiet. Ook de preconditioneerder $M$ moet symmetrisch en positief definiet zijn. Om eenzelfde redenering als bij CR, hoeft $AM^{-1}$ niet symmetrisch te zijn.
Werken met een ander inproduct betekent dat we het residu niet meer minimaliseren in de Euclidische norm. In CG minimaliseren we het residu ten opzichte van de $A^{-1}$ norm

$$||r_k||_{A^{-1}} := r_k^T A^{-1} r_k. \tag{72}$$

Het stopcriterium is echter nog steeds gedefinieerd in termen van de Euclidische norm, dit omdat we $||r_k||$ voor niets hebben, terwijl $||r_k||_{A^{-1}}$ moeilijk te berekenen is.
CG is voor symmetrische en positief definiete matrices zonder meer de meest efficiëntste.

### 5.6.3 Preconditioneren

De preconditionering (59) is symmetrisch als $A$ symmetrisch is. Er geldt dan namelijk $L_A = U_A^T$, dus $(D + L_A)^T = D + U_A$. Dus is $M$ van de vorm $B^T DB$, wat altijd symmetrisch is. Helaas hoeft $AM^{-1}$ niet symmetrisch te zijn. We willen graag zeker weten dat we CR en CG kunnen toepassen en zullen de preconditionering daarom iets aanpassen.
Ervan uitgaande dat $D$ positieve diagonaalelementen heeft, kunnen we uit al deze elementen de wortel trekken. De resulterende matrix noemen we $D^{1/2}$. Nu veranderen we de eerste stap van de preconditionering in paragraaf 5.4 als

$$D^{-1/2} A D^{-1/2} y = D^{-1/2} b, \quad x = D^{-1/2} y. \tag{73}$$

ALGORITME 4: *Conjugate Gradients.*

Choose an $\mathbf{x}_0$.
Compute $\mathbf{r} = \mathbf{b} - \mathbf{A}\mathbf{x}_0$.
Put $\mathbf{u} = 0$, $\mathbf{x} = \mathbf{x}_0$, $\rho = 1$.
For $k = 0, 1, 2, \ldots, k_{\max}$ do
    solve $\mathbf{M}\mathbf{c} = \mathbf{r}$
    $\rho' = \rho$, $\rho = \mathbf{r}^T\mathbf{c}$, $\beta = -\rho/\rho'$
    stop if $\rho \leq tol^2$
    $\mathbf{u} = \mathbf{c} - \beta\,\mathbf{u}$
    compute $\mathbf{c} = \mathbf{A}\mathbf{u}$
    $\sigma = \mathbf{c}^T\mathbf{u}$, $\alpha = \rho/\sigma$
    $\mathbf{x} = \mathbf{x} + \alpha\,\mathbf{u}$
    $\mathbf{r} = \mathbf{r} - \alpha\,\mathbf{c}$
end for

Het resulterende systeem met matrix $\tilde{A} := D^{-1/2}AD^{-1/2}$ preconditioneren we verder als in paragraaf 5.4. Dit resulteert in het systeem

$$(I + \tilde{L}_A)^{-1}\tilde{A}(I + \tilde{U}_A)^{-1}\tilde{x} = \tilde{\tilde{b}},$$

en dit is symmetrisch. De matrix is nu namelijk van de vorm $(B^T)^{-1}AB^{-1}$. En hiervoor geldt

$$(B^T)^{-1}AB^{-1} = B^{-1}A^T(B^T)^{-1} = ((B^T)^{-1}AB^{-1})^T,$$

en is dus symmetrisch. De algoritmen CG en CR kunnen nu dus met preconditionering toegepast worden.

### 5.6.4 Methode van Graig

In het geval $A$ niet symmetrisch is, kunnen we toch CG en CR gebruiken. We kunnen de matrix $A$ namelijk gemakkelijk symmetrisch maken door te vermenigvuldigen met $A^T$. We krijgen dan

$$AA^T y = b, \quad x = A^T y. \tag{74}$$

Aanpassing van CG levert Algoritme 5 op: Graig's methode.
Door te vermenigvuldigen met $A^T$ wordt het conditiegetal van een matrix (grootste eigenwaarde gedeeld door kleinste eigenwaarde) behoorlijk vergroot. Het spectrum van de eigenwaarden wordt dus groter, waardoor we een slechtere convergentie kunnen verwachten. Graig's methode lijkt dus niet heel erg goed te zijn.
Een interessante eigenschap van Graig's methode is, dat de fout $x - x_k$ wordt geminimaliseerd (en niet het residu). We werken immers nog steeds met de $A^{-1}$-norm. En door vermenigvulding met $A^T$ wordt dit de $(AA^T)^{-1}$-norm. En hiervoor geldt

$$||r_k||_{(AA^T)^{-1}} = ||b - Ax_k||_{(AA^T)^{-1}} = ||x - x_k||.$$

Preconditionering van Graig's methode kan alleen expliciet gebeuren.

### 5.6.5 Kostenanalyse

Omdat CG toch efficiënter is dan CR, laten we vanaf nu dat algoritme rusten. Tellen van het aantal basisoperaties voor CG en Graig levert

- CG: 1 solve + 2 DOT + 3 AXPY + 1 MV = $31n$ flops,
- Graig: 1 MV extra t.o.v. CG, dus $41n$ flops,

ALGORITME 5: *Graig's methode.*

```
Choose an x_0.
Compute r = b − Ax_0.
Put u = 0, x = x_0, ρ = 1.
For k = 0, 1, 2, ..., k_max do
      ρ' = ρ, ρ = r^T r, β = −ρ/ρ'
      stop if ρ ≤ tol^2
      compute c = A^T r
      u = c − β u
      compute c = Au
      σ = u^T u, α = ρ/σ
      x = x + α u
      r = r − α c
end for
```

aan kosten per iteratieslag. De totale kosten hangen natuurlijk af van $k$. Beide methoden kosten in totaal $\mathcal{O}(kn)$ flops.
Het geheugengebruik in CG en Graig is beduidend minder dan in GCR. We hoeven immers elke keer slechts 1 vector $u$, $r$ en $c$ te onthouden. Dit duidt op een geheugengebruik van slechts $\mathcal{O}(n)$.

## 5.7 Bi-orthogonale methoden

We hebben nu een efficiënt algoritme voor symmetrische matrices (CG), maar voor niet-symmetrische matrices hebben we nog niet een echte topper gevonden. GCR convergeert gegarandeerd, maar wordt duurder naarmate het aantal iteratiestappen toeneemt. Werken we met korte recursies, dan krijgen we mindere convergentie. Ook Graig's methode was theoretisch gezien niet echt een aanrader. Dit betekent dat we voor niet-symmetrische problemen of dure stappen moeten toestaan, of mindere convergentie. In deze paragraaf willen we nog een andere optie bestuderen.
In plaats van te streven naar minimale residuen, gaan we streven naar residuen $r_k$ die loodrecht staan op zogenaamde $k$-dimensionale schaduwruimten. Omdat bij toenemende $k$ de $r_k$'s geconstrueerd worden in steeds kleinere ruimten, verwacht men convergentie.
We zullen $x_k$ en $r_k$ gelijktijdig bijwerken met

$$x_{k+1} = x_k + v_k, \quad r_{k+1} = r_k - Av_k. \tag{75}$$

Als we aannemen dat $r_0 = b - Ax_0$ dan geldt met inductiehypothese $r_k = b - Ax_k$:

$$r_{k+1} = b - Ax_{k+1} = b - A(x_k + v_k) = r_k + v_k.$$

Dus met inductie volgt dat voor iedere $k$ $r_k = b - Ax_k$.

De schaduwruimten zijn de Krylov deelruimten $\mathcal{K}_k(A^T; \tilde{r}_0)$. Deze ruimte wordt dus voortgebracht door $A^T$ en een schaduw beginresidu $\tilde{r}_0$. We spreken dan wel over bi-orthogonaliteit. Voor het schaduwresidu kiest men meestal $\tilde{r}_0 = r_0$ of volkomen willekeurig. Het residu $r_k$ en $c_k = Au_k$ moeten dus loodrecht staan op $\mathcal{K}_k(A^T; \tilde{r}_0)$. Dit kunnen we bewerkstelligen door $\alpha_{k-1}$ en $\beta_k$ zo te bepalen dat $r_k \perp \tilde{r}_{k-1}$ en $c_k \perp \tilde{r}_{k-1}$. Met inductie volgt nu ook de loodrechtheid op de overige schaduw basisvectoren $\tilde{r}_j$ $(j < k-1)$.
Hoe moeten we de coëfficiënten $\alpha_{k-1}$ en $\beta_k$ kiezen? We weten uit het CG algoritme dat $u_k = r_k - \beta_k u_{k-1}$. Verder weten we dat $c_k = Au_k$ loodrecht staat op $\tilde{r}_{k-1}$. Dit uitwerken geeft

$$\tilde{r}_{k-1}^T A(r_k - \beta_k u_{k-1}) = 0,$$

waarna oplossen voor $\beta_k$ de volgende uitdrukking geeft:

$$\beta_k = \frac{\tilde{r}_{k-1}^T A r_k}{\tilde{r}_{k-1}^T A u_{k-1}}. \tag{76}$$

Via de bekende relaties $r_k = r_{k-1}\alpha_{k-1}c_{k-1}$ en $r_k \perp \tilde{r}_{k-1}$ krijgen we op dezelfde manier

$$\alpha_{k-1} = \frac{\tilde{r}_{k-1} r_{k-1}}{\tilde{r}_{k-1}^T c_{k-1}}. \tag{77}$$

Verder kiezen we $r_k$ zodanig dat voor zekere $\theta_{k-1} \neq 0$ geldt

$$r_k - \theta_{k-1} A^T \tilde{r}_k \in \mathcal{K}_k(A^T; \tilde{r}_0).$$

Omdat we een orthonormale basis hebben geldt $r_k \perp \tilde{r}_k - \theta_{k-1} A^T \tilde{r}_k$ en dus

$$(\tilde{r}_k - \theta_{k-1} A^T \tilde{r}_k)^T r_k = 0.$$

Waaruit volgt dat

$$\tilde{r}_{k-1}^T A r_k = (A^T \tilde{r}_k)^T r_k = \frac{1}{\theta_{k-1}} \tilde{r}_k r_k, \tag{78}$$

zodat we de teller in de berekening van $\beta$ goedkoop kunnen uitvoeren. Algoritme 6 produceert de orthogonale residuen zoals hierboven uitgewerkt.
De openliggende vraag is nu nog, hoe de schaduw basisvectoren $\tilde{r}_k$ te kiezen.

Choose an $\mathbf{x}_0$ and an $\widetilde{\mathbf{r}}_0$.
Compute $\mathbf{r}_0 = \mathbf{b} - \mathbf{A}\mathbf{x}_0$.
Put $\mathbf{u}_{-1} = 0$, $\mathbf{x} = \mathbf{x}_0$, $\sigma_{-1} = \theta_{-1} = 1$.

For $k = 0, 1, 2, \ldots$ do

stop if $\|\mathbf{r}_k\| \leq tol$

$\rho_k = \widetilde{\mathbf{r}}_k^T \mathbf{r}_k$, $\beta_k = \rho_k/(\theta_{k-1}\sigma_{k-1})$
$\mathbf{u}_k = \mathbf{r}_k - \beta_k \mathbf{u}_{k-1}$
compute $\mathbf{c}_k = \mathbf{A}\mathbf{u}_k$
$\sigma_k = \widetilde{\mathbf{r}}_k^T \mathbf{c}_k$, $\alpha_k = \rho_k/\sigma_k$
$\mathbf{x} = \mathbf{x} + \alpha_k \mathbf{u}_k$
$\mathbf{r}_{k+1} = \mathbf{r}_k - \alpha_k \mathbf{c}_k$

select a $\widetilde{\mathbf{r}}_{k+1} \in \mathcal{K}_{k+2}(\mathbf{A}^T; \widetilde{\mathbf{r}}_0)$
such that for some $\theta_k \neq 0$
$\widetilde{\mathbf{r}}_{k+1} - \theta_k \mathbf{A}^T \widetilde{\mathbf{r}}_k \in \mathcal{K}_{k+1}(\mathbf{A}^T; \widetilde{\mathbf{r}}_0)$

end for

ALGORITME 6: *Bi-orthogonal residuals.*

### 5.7.1 BiCG

In het Bi-Conjugate Gradient (BiCG) algoritme worden voor deze schaduw residuen de residuen van de schaduw vergelijking $A^T \tilde{x} = \tilde{r}_0$ gekozen. Deze kunnen op eenzelfde manier worden berekend als de echte residuen. In feite is het volkomen analoog aan CG, met $A = A^T$ en $r_0 = \tilde{r}_0$:

$$\begin{array}{rcl} \tilde{u}_k & = & \tilde{r}_k = \beta_k \tilde{u}_{k-1} \\ \tilde{c}_k & = & A^T \tilde{u}_k \\ \tilde{r}_{k+1} & = & \tilde{r}_k - \alpha_k \tilde{c}_k. \end{array} \tag{79}$$

Verder geldt $\theta_k = -\alpha_k$.
De benaderingen $\tilde{x}_k$ van de oplossing van de schaduw vergelijking hebben we niet nodig en hoeven we dus ook niet te berekenen. Wel wordt er in BiCG heel wat rekenwerk geïnvesteerd in het berekenen van de schaduw residuen, terwijl die eigenlijk niet interessant zijn. Dit kost in vergelijking met CG 2 AXPY's (updaten van de schaduw residuen en zoekrichtingen), 2 DOT's (bepalingen van $\alpha$ en $\beta$) en een MV (met $A^T$) extra.

### 5.7.2 Bi-CGSTAB

Om te bereiken dat elke matrix vermenigvuldiging het residu verkleint, gaan we op een andere manier aankijken tegen de schaduw residuen $r_k$. Het is niet moeilijk om in te zien dat

$$r_k \in \mathcal{K}_k(A^T; \tilde{r}_0) \Leftrightarrow r_k = Q_{k-1}(A^T)\tilde{r}_0,$$

voor een polynoom $Q_{k-1}$ van graad $k-1$:

$$Q_{k-1} = q_0 I + q_1 A + \ldots + q_{k-1} A^{k-1}.$$

Dan geldt voor $\rho_k$ en $\sigma_k$ met $Q_k := Q_k(A)$

$$\rho_k = \tilde{r}_k^T r_k = \tilde{r}_0^T Q_k r_k \tag{80}$$

$$\sigma_k = \tilde{r}_k^T A u_k = \tilde{r}_0^T A Q_k(A) u_k. \tag{81}$$

Nu willen we $Q_k$ zo kiezen dat $||Q_k r_k||$ kleiner is dan $||r_k||$. De berekening van $Q_k r_k$ en $Q_k u_{k-1}$ kunnen we opsplitsen in 2 delen.

Choose an $\mathbf{x}_0$ and an $\widetilde{\mathbf{r}}_0$
Compute $\mathbf{r} = \mathbf{b} - \mathbf{A}\mathbf{x}_0$.
Put $\mathbf{u} = 0$, $\mathbf{x} = \mathbf{x}_0$, $\sigma = \omega = 1$.

For $k = 0, 1, 2, \ldots, k_{\max}$ do

stop if $\|\mathbf{r}\| \leq tol$

$\rho = \widetilde{\mathbf{r}}_0^T \mathbf{r}$, $\beta = -\rho/(\omega\sigma)$
solve $\mathbf{M}\widehat{\mathbf{u}} = \mathbf{r}$
$\widehat{\mathbf{u}} = \widehat{\mathbf{u}} - \beta\,\mathbf{u}$
compute $\mathbf{c} = \mathbf{A}\widehat{\mathbf{u}}$
$\sigma = \widetilde{\mathbf{r}}_0^T \mathbf{c}$, $\alpha = \rho/\sigma$
$\widehat{\mathbf{x}} = \mathbf{x} + \alpha\,\widehat{\mathbf{u}}$
$\widehat{\mathbf{r}} = \mathbf{r} - \alpha\,\mathbf{c}$

solve $\mathbf{M}\mathbf{s}' = \widehat{\mathbf{r}}$
solve $\mathbf{M}\widehat{\mathbf{c}} = \mathbf{c}$
compute $\mathbf{s} = \mathbf{A}\mathbf{s}'$
$\omega = \mathbf{s}^T\widehat{\mathbf{r}}/\mathbf{s}^T\mathbf{s}$
$\mathbf{u} = \widehat{\mathbf{u}} - \omega\,\widehat{\mathbf{c}}$
$\mathbf{x} = \widehat{\mathbf{x}} + \omega\,\mathbf{s}'$
$\mathbf{r} = \widehat{\mathbf{r}} - \omega\,\mathbf{s}$

end for

Algoritme 7: Bi-CGSTAB.

- Eerst vermenigvuldigen we de BiCG stappen uit Algoritme 6 met $Q_k$:

$$\begin{aligned} Q_k u_k &= Q_k r_k - \beta_k Q_k u_{k-1} \\ Q_k c_k &= A Q_k u_k \\ Q_k r_{k+1} &= Q_k r_k - \alpha_k Q_k c_k. \end{aligned} \tag{82}$$

- Vervolgens hogen we de graad van $Q_k$ 1 op met

$$Q_{k+1} = (I - \omega_k A) Q_k. \tag{83}$$

  door de scalair $\omega$ te kiezen zodanig dat $||(I - \omega_k A)Q_k r_k||$ minimaal is. Nu kunnen we met $Q_{k+1}$ $u_{k+1}$ en $r_{k+1}$ berekenen.

Met de notatie $u_{k-1} = Q_k u_{k-1}$, $\hat{u}_k = Q_k u_k$, $c_k = A Q_k u_k$, $r_k = Q_k r_k$ en $\hat{r}_k = Q_k r_{k+1}$, krijgen we Algoritme 7: Bi-CG stabilized (Bi-CGSTAB).
Dit algoritme is nog niet helemaal efficiënt. Het aantal benodigde 'solves' kan naar 2 worden teruggebracht en de benodigde geheugenruimte kan worden verkleind. Wel moeten we de volgorde van het algoritme dan iets omgooien. Het resulterende algoritme is Algoritme 8.

### 5.7.3 Kostenanalyse

Voor het efficiënte Bi-CGSTAB algoritme tellen we

$$2\text{solve} + 5\text{DOT} + 5\frac{1}{2}\text{AXPY} + 2\text{MV} = 63n\text{flops}.$$

Ook Bi-CGSTAB kost dus $\mathcal{O}(kn)$ flops.
Het geheugengebruik is hoger dan bij CG, maar ook hier hoeven we niet alle voorgaande $r_k$'s, $c_k$'s en $u_k$'s bij te houden. Dus is het geheugengebruik bij Bi-CGSTAB $\mathcal{O}(n)$.

Choose an $\mathbf{x}_0$ and an $\widetilde{\mathbf{r}}_0$
Compute $\mathbf{r} = \mathbf{b} - \mathbf{A}\mathbf{x}_0$.
Put $\mathbf{u} = 0$, $\mathbf{x} = \mathbf{x}_0$, $\sigma = \omega = 1$.

For $k = 0, 1, 2, \ldots, k_{\max}$ do

    stop if $\|\mathbf{r}\| \leq tol$
    $\rho = \widetilde{\mathbf{r}}_0^T \mathbf{r}$, $\beta = -\rho/(\omega\sigma)$
    $\mathbf{u} = \mathbf{r} - \beta\omega\,\mathbf{c}$
    solve $\mathbf{M}\mathbf{u} = \mathbf{u}$
    compute $\mathbf{c} = \mathbf{A}\widehat{\mathbf{u}}$
    $\sigma = \widetilde{\mathbf{r}}_0^T \mathbf{c}$, $\alpha = \rho/\sigma$
    $\widehat{\mathbf{x}} = \mathbf{x} + \alpha\,\widehat{\mathbf{u}}$
    $\mathbf{r} = \mathbf{r} - \alpha\,\mathbf{c}$
    solve $\mathbf{M}\mathbf{s}' = \widehat{\mathbf{r}}$
    compute $\mathbf{s} = \mathbf{A}\mathbf{s}'$
    $\omega = \mathbf{s}^T\widehat{\mathbf{r}}/\mathbf{s}^T\mathbf{s}$
    $\mathbf{x} = \widehat{\mathbf{x}} + \omega\,\mathbf{s}'$
    $\mathbf{r} = \mathbf{r} - \omega\,\mathbf{s}$

end for

Algoritme 8: efficiënt Bi-CGSTAB.

# 6 Hypothese

Nu we alle theorie hebben behandeld, kunnen we een theoretisch antwoord formuleren op onze vraagstelling: een hypothese. Deze hypothese zullen we testen in hoofdstuk 7.

De subvraag aangaande het efficiënt modelleren is uitgebreid beantwoord in hoofdstuk 3 en 4. We hebben daar gezien dat de nauwkeurigheid van de oplossing samenhangt met het aantal roosterpunten $n_x n_y$. De verwachting is dat de nauwkeurigheid omgekeerd kwadratisch evenredig is met $\max h_x, h_y$.

In hoofdstuk 5 hebben we verschillende iteratieve methoden onderzocht. Voor alle methoden hebben we de kosten in flops bepaald. Helaas hangen deze kosten altijd af van het aantal iteratieslagen $k$, dus van de snelheid van de convergentie.
Voor symmetrische problemen (die in ons probleem ook positief definiet zijn) hebben we CG als meest efficiënte methode, daar kunnen we kort in zijn.
Voor niet-symmetrische methoden ligt een en ander veel lastiger. GCR was duur, is als enige methode qua kosten kwadratisch evenredig met het aantal iteratiestappen ($\mathcal{O}(k^2 n)$ flops). Voor GCR met korte recursies kunnen we de convergentie niet meer garanderen, en Graig's methode heeft slechte convergentie eigenschappen vanwege het hoge conditiegetal, maar is dan weer goedkoop per iteratiestap. Uiteindelijk vonden we als laatste optie Bi-CGSTAB. Deze methode is efficiënt, maar convergentie is ook hier niet altijd gegarandeerd. Naar verwachting zullen Bi-CGSTAB en GCR met kort recursies het best presteren. Voor de laatste zullen we dan moeten zoeken naar optimale waarden voor $l_i$. Ervaringen van anderen duiden er op dat Bi-CGSTAB de meest efficiënte methode is. We verwachten dus dat voor probleem 1 CG het beste zal werken, en voor probleem 2 Bi-CGSTAB.

Preconditionering konden we grofweg op 3 manieren doen: diagonaal, impliciet (59) en expliciet met (59) en de Eisenstat truuk. We hebben de extra kosten van deze preconditioneringen onderzocht. Diagonaal preconditioneren was goedkoop, maar waarschijnlijk niet heel erg behulpzaam. Impliciet preconditioneren ging $10n$ flops per iteratie extra kosten. En expliciet preconditioneren met de Eisenstat truuk kan gratis. Hiermee kijken we echter alleen naar de kosten per iteratieslag. De kosten voor het pre-proces zijn voor expliciet preconditioneren dus niet meegenomen. Dit hoeft dan ook maar 1 keer te gebeuren. Voor grote $k$ kunnen we dit dus hopelijk verwaarlozen. De convergentie snelheid zal voor de impliciete en de expliciete variant nooit veel verschillen. Al met al verwachten we daarom dat explicitiet preconditioneren met de Eisentat truuk het meeste tijdwinst op zal leveren. Al zal de impliciete versie er waarschijnlijk weinig voor onderdoen, afhankelijk van de convergentiesnelheid.
Een andere vraag, die voor zowel expliciet als impliciet preconditioneren van belang is, is: wat is de optimale waarde voor $\omega$? Gebaseerd op (andermans) ervaring verwachten we als optimale waarde $\omega \approx 0.95$. Voor het symmetrische probleem 1 zal $\omega = 1$ waarschijnlijk het beste resultaat geven.

In de behandelde theorie zijn we steeds stilzwijgend uitgegaan van startvector $x_0 = 0$. Dit zal natuurlijk nooit de meest efficiënte startvector zijn. De waarden van pompen kunnen we bijvoorbeeld al ongeveer in de startvector invullen, zodat we deze niet meer iteratief hoeven te bepalen. Ook random startvectoren kunnen wel eens efficiënter zijn dan de nul-vector.
Tenslotte kunnen we de startvector ook bepalen, door het probleem eerst op te lossen op een grof grid, wat veel sneller gaat. De verkregen oplossing kunnen we dan gebruiken als startvector voor de iteratie op het fijne grid. Op die manier krijgen we een soort 2-grid algoritme.

Samenvattend stellen we de volgende hypotheses.

- Probleem 1: Modellering volgens hoofdstuk 3 en 4, discretisatiefouten van $\mathcal{O}(h^2)$;
  De beste iteratieve oplosmethode is CG;
  Preconditionering is het efficiëntst met de expliciete variant door middel van de Eisenstat truuk, $\omega \approx 0.95$.

Door aanpassen van de startvector valt er snelheidswinst te behalen.

- Probleem 2: Modellering volgens hoofdstuk 3 en 4, discretisatiefouten van $\mathcal{O}(h^2)$;
  De beste iteratieve oplosmethode is Bi-CGSTAB;
  Preconditionering is het efficiëntst met de expliciete variant door middel van de Eisenstat truuk, $\omega = 1$.
  Door aanpassen van de startvector valt er snelheidswinst te behalen.

# 7 Testresultaten

We gaan hier antwoorden proberen te geven op de laatste drie deelvragen die we eerder hebben gesteld. We zullen dit in dit hoofdstuk doen door middel van experimenten. In het volgende hoofdstuk zullen we de resultaten in detail bespreken en kijken of deze kloppen met de hypotheses uit het vorige hoofdstuk.

## 7.1 Effect van preconditioneren

We hebben gezien dat we globaal gezien op drie verschillende manieren konden preconditioneren:

-Diagonaal ($M = D_A$),

-Impliciet, en

-Expliciet met de Eisenstat truuk.

Laten we eerst probleem 1 oplossen op een 50 bij 48 grid, gebruik makend van GCR en de drie PGCR methodes. We noteren de tijdsduur en het aantal iteratieslagen:

| Preconditioneer-methode | Tijdsduur (in seconden) | Aantal iteratieslagen |
|---|---|---|
| Geen | 27 | 327 |
| Diagonaal | 20 | 277 |
| Impliciet | $0,7$ | 48 |
| Expliciet | $0,7$ | 49 |

*Grid:* 50 *bij* 48*, probleem 1, PGCR*

We zien dat deze resultaten aansluiten bij onze hypotheses. Niet preconditioneren en diagonaal preconditioneren werken inderdaad niet al te best, in vergelijking met impliciet en expliciet preconditioneren. Het verschil tussen die twee laatste was -in theorie- dat met behulp van expliciet preconditioneren de rekentijd per iteratiestap omlaag gaat, maar te maken hebben met een post- en pre-proces fase. In ons experiment was het grid waarschijnlijk te grof om dat te bevestigen. Laten we het grid dus fijner kiezen:

| Preconditioneer-methode | Tijdsduur (in seconden) | Aantal iteratieslagen |
|---|---|---|
| Impliciet | 137 | 158 |
| Expliciet | 143 | 164 |

*Grid:* 200 *bij* 198*, probleem 1, PGCR*

Deze testresultaten ondersteunen de hypothese niet; alhoewel het tijdsverschil redelijk klein is, is expliciet preconditioneren langzamer dan impliciet. Er is zelfs te bereken dat het aantal seconden per iteratieslag voor impliciet 0.1 minder is dan bij de expliciete methode. Hier is echter niet meegerekend dat het post- en preprocessen ook tijd hebben ingenomen. Aangezien dit resultaat toch vreemd is besloten we het experiment op een middelfijn grid (100 bij 98) te herhalen voor probleem 2:

| Preconditioneer-methode | Tijdsduur (in seconden) | Aantal iteratieslagen |
|---|---|---|
| Geen | 102 | 306 |
| Diagonaal | 71 | 255 |
| Impliciet | $1,5$ | 29 |
| Expliciet | $1,3$ | 29 |

*Grid:* 100 *bij* 98*, probleem 2, PGCR*

We zien dat dit experiment wel aan de verwachtingen voldoet. Dit komt waarschijnlijk omdat dit tweede probleem complexer was dan het eerste (lees: asymmetrischer); maar hierover meer in de volgende subsectie. Eerst gaan we hier nog verder in op het preconditioneren.
Uit het theoretische gedeelte weten we dat het preconditioneren gebeurt aan de hand van een matrix $M$ die geconstrueerd wordt met behulp van de zogenaamde RILU-methode. Deze methode maakt gebruik van een constante $\omega$. Variatie hiervan zou tot snellere convergentie kunnen leiden; dit gaan we nu experimenteel onderzoeken. Standaard hebben we $\omega = 0,95$ aangehouden; dit geldt in het algemeen als de meest optimale waarde.
We varieren $\omega$ rond de standaardwaarde van $0,95$:

| $\omega$ | Tijdsduur (in seconden) |
|---|---|
| $0,9$ | $1,5$ |
| $0,95$ | $1,3$ |
| $\mathbf{0,975}$ | $\mathbf{1,1}$ |
| $1,0$ | $1,4$ |

*Omega-waardes uitgezet tegen de oplosduur van probleem* 2 *op een* 100 *bij* 98 *grid (expliciete PGCR)*

Theoretisch gezien zijn $\omega$-waardes dicht in de buurt van 1 ideaal. Helaas werken onze computers met eindige preciesie en is het dankzij afrondingsfouten zo dat $\omega = 1$ bijna nooit de efficientste keuze is, dit vooral als het probleem niet symmetrisch is. Laten we dit nu onderzoeken aan de hand van probleem 1, welk symmetrisch is:

| $\omega$ | Tijdsduur (in seconden) |
|---|---|
| $0,9$ | 12 |
| $0,95$ | $9,2$ |
| $0,975$ | $6,8$ |
| $0,99$ | $5,49$ |
| $\mathbf{1,0}$ | $\mathbf{3,9}$ |

*Omega-waardes uitgezet tegen de oplosduur van probleem* 1 *op een* 100 *bij* 98 *grid (expliciete PGCR)*

We zien dat bij probleem 1 de meest optimale $\omega$-waarde van 1 wel tot het snelste resultaat leidt.

## 7.2 Verbanden tussen soort probleem en de gebruikte oplosmethode

We maken allereerst een onderscheid tussen verschillende soorten problemen:

-Symmetrische problemen, en

-Asymmetrische problemen

Probleem 1 uit hoofdstuk 2 is bijvoorbeeld een symmetrisch probleem, maar dankzij toevoeging van een vectorveld $(u, v)$ is probleem 2 dat niet. Ook kunnen we een onderscheid maken tussen de grootte van problemen; e.g. de grootte van het grid. We zullen deze factoren laten varieren en op elk op zodanige manier verkregen probleem alle oplosmethodes loslaten. We meten de efficientie van de methoden aan de hand van de oplossnelheid van de methode, d.w.z. de benodigde rekentijd voor de methode om het lineaire stelsel op te lossen.
De keuze van preconditioneerder en andere variabelen ($l_i$ bij PGCRr) zijn telkens zodanig gekozen dat het snelste resultaat bereikt wordt.

| Methode | Tijdsduur (in seconden) |
|---|---|
| GCR | 27 |
| PGCR | $0,7$ |
| PGCRr (afkappen) | $0,6$ |
| **PCG** | $\mathbf{0,2}$ |
| Graig | $1,8$ |
| **Bi-CGSTAB** | $\mathbf{0,2}$ |

*Grid:* 40 *bij* 38*, probleem* 1

| Methode | Tijdsduur (in seconden) |
|---|---|
| GCR | 458 |
| PGCR | $8,0$ |
| PGCRr (afkappen) | $5,6$ |
| **PCG** | $\mathbf{1,1}$ |
| Graig | 19 |
| Bi-CGSTAB | $1,7$ |

*Grid:* 100 *bij* 99*, probleem 1*

| Methode | Tijdsduur (in seconden) |
|---|---|
| GCR | – |
| PGCR | – |
| PGCRr | – |
| **PCG** | **210** |
| Graig | – |
| Bi-CGSTAB | 302 |

*Grid:* 500 *bij* 498*, probleem 1*
*Experimenten veel langer dan vijf minuten werden voortijdig afgebroken*

We zien dat voor probleem 1, wat een symmetrisch probleem is, dat PCG de meest efficiente methode lijkt te zijn. Dit is onafhankelijk van de grootte van het probleem; wel is het zo dat het verschil in snelheid in vergelijking met Bi-CGSTAB steeds groter wordt. Onze hypothese wordt hier dus bevestigd.
We gaan nu kijken naar het (niet-symmetrische) probleem 2. De PCG-methode werkt niet op asymmetrische problemen en wordt dus niet meegenomen in de volgende experimenten:

| Methode | Tijdsduur (in seconden) |
|---|---|
| GCR | $4,5$ |
| PGCR | $0,1$ |
| PGCRr (afkappen) | $0,08$ |
| Graig | $2,4$ |
| **Bi-CGSTAB** | $\mathbf{0,09}$ |

*Grid:* 40 *bij* 38*, probleem 2*

| Methode | Tijdsduur (in seconden) |
|---|---|
| GCR | 105 |
| PGCR | $1,3$ |
| PGCRr (afkappen) | $0,8$ |
| Graig | 52 |
| **Bi-CGSTAB** | $\mathbf{0,5}$ |

*Grid:* 100 *bij* 99*, probleem 2*

| Methode | Tijdsduur (in seconden) |
|---|---|
| GCR | – |
| PGCR | 17 |
| **PGCRr** (restart) | **11** |
| Graig | 750 |
| Bi-CGSTAB | 18, 5 |

*Grid:* 200 *bij* 198*, probleem 2*

Hier zien we dat voor een niet-symmetrisch probleem de strijd om de meest efficiente methode vooral gaat tussen Bi-CGSTAB en PGCRr. Bij PGCRr gaat het vooral om het vinden van goede herstart/afkap waardes ($l_1$,$l_2$) en/of nest-waardes ($l_3$), voordat deze methode sneller kan worden dan Bi-CGSTAB. Een direct verband tussen de $l_i$-waardes en de grootte van het probleem lijkt niet te bestaan. Herstarten is op een groter grid efficienter dan afkappen, zo bleek uit dit experiment. Nesten leidt in dit probleem niet tot een noemenswaardige snelheidswinst.

## 7.3 Verdere verbeteringen

We hebben twee ideeën voor verbetering opgedaan, welk we hier zullen toelichten en experimenteel de meerwaarde van zullen bepalen.

### 7.3.1 Aanpassing startvector

We zullen nu methodes bekijken om het convergeren nog sneller te laten verlopen. We herinneren ons dat de methodes die we gebruiken voor het oplossen van onze lineaire vergelijking iteratieve methodes zijn; een bepaalde oplossingsvector wordt telkens iteratief verbeterd. Tot nu toe hebben we als startvector telkens de nul-vector genomen; maar dat kan efficiënter. Voor sommige waarden in de vector weten we namelijk de oplossing al ongeveer. Als we een pomp in een bepaald gridpunt hebben staan weten we dat de uitkomst op dat punt niet al te veel zal verschillen van wat die pomp daar bij- of wegpompt. We kunnen dus de startvector op dat gridpunt gelijkstellen aan de hoeveelheid die de pomp daar bij- of wegpompt.
Echter, in plaats daarvan kunnen we ook random een startvector kiezen. De random waarden laten we varieren tussen 0 en de maximale waarde die de pompen en rivieren aan het probleem toevoegen. Dit varieren kan uniform gebeuren (e.g., elk getal tussen 0 en de maximale waarde heeft een evengrote kans gekozen te worden), of exponentioneel (hoe groter of hoe kleiner de waarde, hoe groter de kans om gekozen te worden). Pseudo-code voor het genereren van een random startvector is te vinden in Appendix B.
Een combinatie tussen de twee net besproken methodes lijkt ook leuk; eerst kiezen we random een startvector, en daarna passen we de waardes aan aan wat we weten over pompen en rivieren. We hebben deze drie methodes uitgeprobeerd op probleem 1:

| Methode startvector | Tijd (in seconden) | Aantal slagen |
|---|---|---|
| Zonder aanpassing naar bronterm | 189 | 274 |
| Met aanpassing naar bronterm | 183 | 267 |
| Random (uniform) | 179 | 260 |
| Random (uniform) | 178 | 259 |
| Random (uniform) | 177 | 258 |
| Random (gem.) | 178 | 259 |
| Random (exp) | 171 | 255 |
| Random (exp) | 173 | 260 |
| Random (exp) | 174 | 258 |
| Random (gem.) | 173 | 259 |
| Random (omgekeerd exp) | 184 | 269 |
| Random (omgekeerd exp) | 186 | 270 |
| Random (omgekeerd exp) | 185 | 270 |
| Random (gem.) | 185 | 270 |
| **Random (exp) met bronterm** | 175 | 256 |

*Grid:* 100 *bij* 99*, probleem 1, PCG impliciet*

We zien dat we door aanpassing van de startvector een snelheidswinst van rond de 5 procent kunnen halen. Als snelste methode vonden we een exponentioneel random startvector met aanpassing naar de bronterm. Laten we kijken hoe deze methode zich gedraagt in samenwerking met Bi-CGSTAB:

- Tijd zonder aanpassing startvector: 302

- Tijd met aanpassing startvector: 255

Een behoorlijk grote winst; het aanpassen van de startvector lijkt echt de moeite waard te zijn. We merken nog wel op dat de manier van het random genereren probleemspecifieke prestaties levert. Een probleem waarvan de uitkomstwaarden nagenoeg uniform verdeeld zijn heeft logischerwijs meer profijt van een uniform verdeelde startvector, in tegenstelling tot bijvoorbeeld een exponentiele.

### 7.3.2 Twee-grid algoritme

In voorgaande zagen we dat het aanpassen van de startvector voor niet-minieme snelheidswinst kan zorgen. Aangezien we voor probleem 2 al een soort twee-fase algoritme hebben moeten maken (eerst een probleem oplossen om het grondwaterstromingsveld te berekenen, en daarna het probleem van de gifconcentratie), kwamen we op het idee om nu twee keer hetzelfde probleem op te lossen; de eerste keer op een grof grid, om met de uitkomst daarvan een startvector te maken voor het oplossen op het fijne (beoogde) grid.
We proberen het bovenstaande uit, op een 300 bij 300 grid voor probleem 1. Voor het grove grid nemen we twee keer zo weinig roosterpunten. Pseudocode voor dit algoritme is weer te vinden in Appendix B. De resultaten zijn als volgt:

| Methode | Tijd met 2 fases | Tijd zonder 2 fases |
|---|---|---|
| PCG (impliciet) | 55 | 55 |
| BiCGSTAB (impliciet) | 80 | 87 |

*Grid:* 300 *bij* 300*, probleem 1*

Deze methode werkt op een of andere manier minder goed dan het random kiezen van een startvector; wat op zich een vreemd resultaat is. We kunnen niet verklaren waarom dit zo is. Wel geeft het algoritme bij

BiCGSTAB een kleine tijdswinst. Bij PGCR is dit ook het geval:

| Tijd met 2 fases | Tijd zonder 2 fases |
|---|---|
| 102 | 104 |

*Grid:* 200 *bij* 175*, probleem 1, PGCR (impiciet)*

Wederom een zeer kleine snelheidswinst. Laten we nu bekijken wat deze 2-fasen methode doet met een niet-symmetrisch probleem:

| Grid | Tijd met 2 fases | Tijd zonder 2 fases |
|---|---|---|
| 200 bij 200 | 10 | 15 |
| 260 bij 250 | 34 | 62 |

*Probleem 2, Bi-CGSTAB (impliciet)*

Dit zijn wel resultaten om over naar huis te schrijven. Een tijdswinst van bijna 50 procent op een groot grid is niet weinig. Deze methode is wellicht nog beter te tunen door aanpassing van de factor (waarmee de grootte van het grove grid werd bepaald). Ook werkt het misschien beter als we de startvector alsnog aanpassen aan de bronterm, wanneer daar een rivier of een pomp staat. Voorlopig zijn we echter hier ook wel tevreden mee.

# 8 Conclusies & Aanbevelingen

Aan de hand van het fysisch probleem voorgesteld in hoofdstuk 3 hebben we een discretisatiemethode besproken dat een grondwaterprobleem omzet in een linear vergelijkingsstelsel $Ax = b$ dat opgelost moet worden. De manier van discretiseren zoals besproken in hoofdstuk 4 zorgt voor een symmetrische matrix $A$ zolang er geldt dat het $(V = (u, v))$-vectorveld (7) is zoals in (37). Dit symmetrisch zijn zorgt ervoor dat we zeer snelle oplosmethoden kunnen gebruiken die uitsluitend op symmetrische problemen werken (zoals PCG). Dit draagt bij aan een zo efficiënt mogelijk oplosproces. De manier waarop we de randwaarden hebben geëlimineerd dragen hier ook aan bij; de oploswaarden op de rand worden niet expliciet uitgerekend door het oplossen van het lineaire stelsel.
Preconditioneren is altijd aan te raden. Theoretisch gezien is expliciet preconditioneren het beste, echter in de praktijk zagen we dat er niet zo gek veel verschil tussen impliciet en expliciet is. Wel is het zo, dat hoe groter en hoe a-symmetrischer het probleem is, hoe interessanter expliciet preconditioneren wordt.
Een ander punt op het gebied van preconditioneren is het kiezen van de $\omega$ waarde, nodig voor het bepalen van de matrix $D$ in (59). Aan te raden is om deze waarde voor niet-symmetrische problemen experimenteel optimaal te kiezen (desnoods met behulp van een grof grid). De optimale waarde ligt altijd rond de $0,95$, en voor symmetrische problemen is de optimale $\omega$ waarde meestal 1.
Bij het oplossen van symmetrische problemen hebben we reeds zowel theoretisch als experimenteel aangetoond dat PCG de snelste op dat gebied is. We kunnen ook opmerken dat Bi-CGSTAB hier een goede tweede is; het komt niet echt in de buurt bij PCG, maar laat de andere methodes wel achter zich op grotere problemen.
Voor niet-symmetrische gevallen hangt veel af van het soort probleem, theoretisch gezien. Als de convergentiesnelheid groot is, heeft PGCRr een groot voordeel. Is dit niet zo, dan is Bi-CGSTAB wellicht in het voordeel. De uitgevoerde experimenten bevestigden dit. Echter moeten we wel opmerken dat we, om PGCRr efficiënt te gebruiken, vooraf moeten kunnen 'spelen' met het probleem om efficiente $l_i$ waardes te kunnen kiezen; bij verkeerde keuzes is het goed mogelijk dat Bi-CGSTAB minstens net zo snel is.
Verdere versnellingen zijn mogelijk door te spelen met de startvector die we gebruiken bij het opstarten van onze iteratieve methodes. Deze startvector kan bijvoorbeeld random worden gekozen, of aan de hand van een oplossing op een grover grid worden gekozen (2-grid). Voor symmetrische problemen zagen we dat de zogenaamde 2-grid methode niet zo efficient was als we hoopten; hier werkte het random kiezen heel wat beter. Voor niet-symmetrische problemen werkt de 2-grid methode wel erg goed, zoals we gezien hebben.

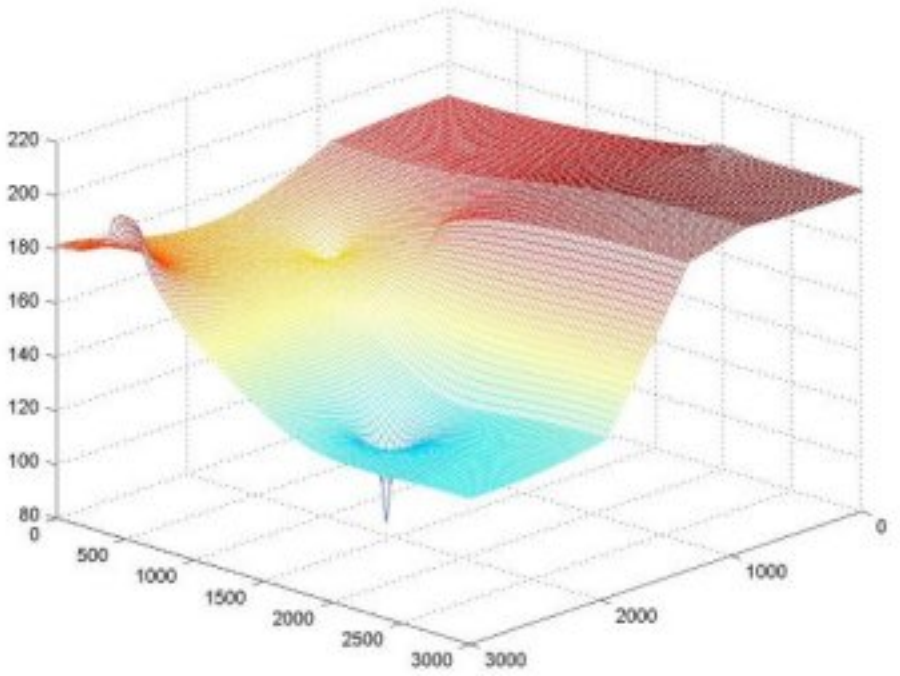
Ruimte voor verbetering zit hem in de eerste plaats vooral in de 2-grid methode. Aanpassen van de factor waardoor de grootte van het grove grid bepaald werd lijkt een interessant onderwerp voor dieper onderzoek. Ook zouden we ons kunnen afvragen of het efficiënt is om met meerdere grove grids te werken. Een andere richting van verbetering met betrekking tot het grid is gebaseerd op de vraag of het wel zo handig is om een uniforme verdeling aan te houden voor het grid. Als een grid op een stuk waar de uiteindelijke oplossing veel verandert fijner wordt gekozen dan op andere stukken, hebben we een efficiëntere methode. Probleem is natuurlijk dat je niet weet waar de meeste veranderingen optreden in het probleem; wellicht is het handig om daar weer een grof grid voor te gebruiken.
Dit zijn maar een paar interessante suggesties voor verbeteringen; het is goed mogelijk dat er op andere gebieden ook efficienter kan worden gewerkt. Het is ook mogelijk om het model dat we nu hebben verder uit te breiden, bijvoorbeeld naar een 3-dimensionaal probleem, of een tijdsafhankelijk probleem. Dit soort suggesties worden ook gedaan in [1].
Al met al hebben we middels dit verslag al onze deelvragen kunnen beantwoorden, en een leuk stukje software kunnen ontwikkelen.

# Referenties

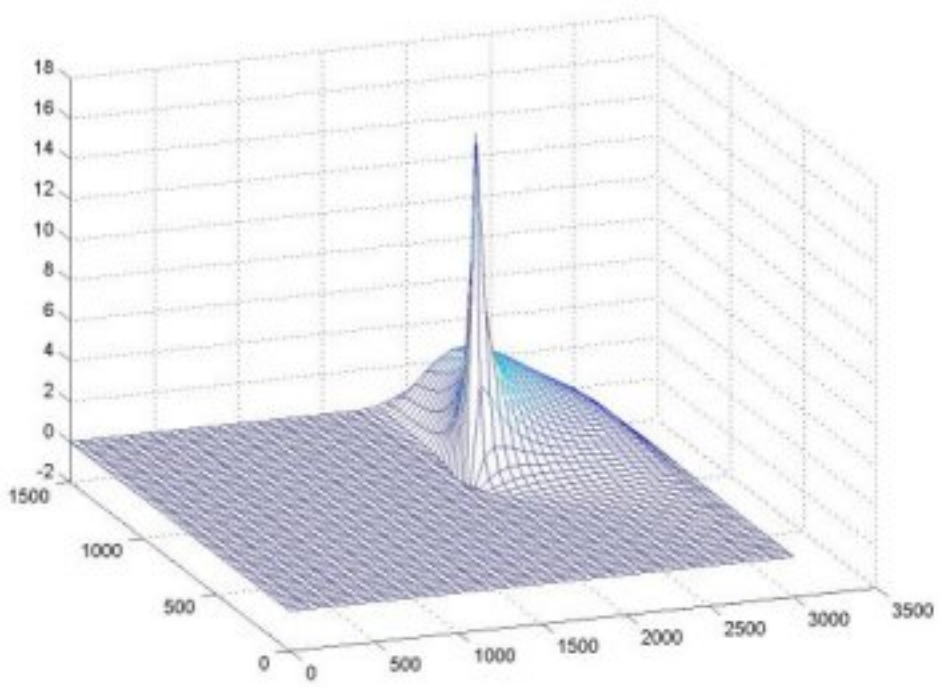

- [1] G.L.G. Sleijpen, *Compuatational Science Praktikum; Iteratieve Lineaire Oplosmethoden*, Universiteit Utrecht, mei 2002.
- [2] W.H. Press, S.A. Teukolsky, W.T. Vetterling en B.P. Flannerly, *Numerical Recipes*, Cambridge University Press, 1992.
- [3] G.H. Golub en C.F. van Loan, *Matrix Computations*, The Johns Hopkins University Press, Baltimore, 1996.

# A Visuele Resultaten

Probleem 1:



Probleem 2:


18
16
14
12
10
8
6
4
2
0
-2
1500
1000
500
0
0
500
1000
1500
2000
2500
3000
3500

# B Pseudocode

Hier behandelen we kort de 'speciale' C-code die we geschreven hebben om de ideeën in 7.3 te kunnen uitwerken. We beginnen met het bepalen van een random startvector. Deze vector noemen we $x$, en heeft als grootte $n_x * n_y + 1$. De gridpuntwaarden worden dus opgeslagen in de $x_i$ coördinaten, met $1 \leq i \leq n_x * n_y$. Het gebruikte algoritme voor uniform random vectorgeneratie ziet er als volgt uit (in psuedo code):

```
Algorithm uniform random startvector:
for i<-1 to i<=nx*ny
       rand<-uniform random getal tussen in [0,1]
       x[i]<-rand*maxval
End Algorithm
```

maxval is de maximale waarde die door pompen en rivieren worden aangenomen; deze waarde wordt in het probleem zelf gedefenieërd. Het algoritme voor de exponentiële varianten:

```
Algorithm exponential distributed random startvector:
for i<-1 to i<=nx*ny
       rand<-uniform random getal tussen in [0,1]
       x[i]<-log(100.0)/maxval
       x[i]<-log(1/(1-rand))/x[i]
       if (x[i]>maxval)
              x[i]<-maxval;    //x[i] mag niet groter zijn dan het maximum
End Algoritm
```

```
Algorithm reversed exponential distributed random startvector:
for i<-1 to i<=nx*ny
       rand<-uniform random getal tussen in [0,1]
       x[i]<-log(100.0)/maxval
       x[i]<-log(1/(1-rand))/x[i]
       if (x[i]>maxval)
              x[i]<-maxval     //x[i] mag niet groter zijn dan het maximum
       x[i]<-maxval-x[i]        //Omkeren
End Algoritm
```

Het programma heeft in zijn originele versie een zogenaamde *main*-functie, welk imperatief de stappen aanroept die nodig zijn om het totale probleem op te lossen. Om het 2-grid methode simpel te implementeren hebben we dat *main*–functie hernoemd naar *origmain*, met als parameters ($n_x$,$n_y$,$x$); $n_x$ en $n_y$ stellen het gridgrootte voor, en $x$ is de startvector. We schrijven nu een nieuwe functie main die origmain tweemaal aanroept, in overeenstemming met de 2-grid methode:

```
Algorithm main
       factor<-2
       nx<-Grid.x
       ny<-Grid.y
       x<-nieuwe vector van grootte nx*ny
       OrigineelGrid<-Grid
       Grid.x<-Grid.x/factor
       Grid.y<-Grid.y/factor
       nx<-Grid.x
       ny<-Grid.y
       y<-nieuwe vector van grootte nx*ny
       for i<-0 to i<=nx*ny+1
              y[i]<-0
```

```
      origmain(nx,ny,y)                    //y bevat nu de oplossing van het grove grid

      for i<-0 to i<=nx*ny+1
            for j<-0 to j<factor
                  x[(i*factor)+j]<-y[i]    //x bevat nu de oplossing van het grove grid,
                                               via waardenherhaling

      Grid<-OrigineelGrid
      nx<-Grid.x
      ny<-Grid.y

      origmain(nx,ny,x);                   //x bevat nu de 'echte' oplossing
End Algorithm
```

*Aan de hand van factor wordt de grootte van het grove grid bepaald*