RA

DUPLICAAT

MATHEMATISCH CENTRUM
2e BOERHAAVESTRAAT 49
AMSTERDAM

REKENAFDELING

Cursus

Programmeren voor Automatische Rekenmachines

onder leiding van

T.J. Dekker

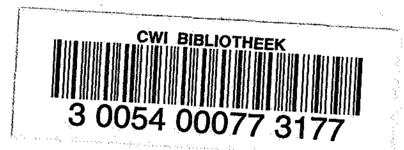
E.W. Dijkstra

Prof.Dr Ir A. van Wijngaarden

November 1957

RA

BIBLIOTHEEK MATHEMATISCH CENTRUM AMSTERDAM



Inhoud

		pag.
1.	Algemene inleiding over automatische rekenmachines	1
2.	Het woord	7
3.	Het getal	13
4.	De opdracht	18
5.	Blokschema's	20
6.	Subroutines I	30
7.	Een uitgewerkte voorbeeld	37
8.	Subroutines II	46
9.	Subroutines III	58
10.	Snelheid	67
11.	Schaling, controle en flexibiliteit	75
12.	De administratieve subroutine I	83
13.	De administratieve subroutine II	87
14.	Superprogramma's	96

1. Algemene inleiding over automatische rekenmachines

De oudste rekenmachines, die ontworpen zijn, hadden tot taak de mensen enig slavenwerk, nl. het rekenen met papier en potlood, uit handen te nemen. Het is bekend, dat met de nieuwste machines dit ideaal meer dan ingehaald is: tegenwoordig kan men amper spreken van "werk uit handen nemen": de problemen, waarbij de machtigste der moderne machines pas goed tot hun recht komen, zijn dusdanig omvangrijk, dat men er zonder deze rekenapparatuur nooit aan zou zijn begonnen! Er worden problemen mee aangepakt, die vroeger de meest drieste niet eens als "numeriek probleem" zou durven te beschouwen; en inderdaad, naarmate de methoden, waarop de machines hun resultaten afleveren, geraffineerder worden, raakt het numeriek karakter althans voor de naïeve bezoeker, ernstig op de achtegrond. Maar wij, die ons bezig hebben te houden met de interne organisatie, d.w.z. hoe de brug geslagen moet worden tussen een gegeven probleem en een gegeven machine, doen er goed aan, deze machines, ontdaan van alle spectaculaire luister, onderdeel voor onderdeel onder de loupe te nemen; het zal duidelijk worden, dat zij in hun functionele samenhang een rekenaar, uitgerust met een tafelrekenmachine, vrij aardig nabootsen en dus zijn werk over kunnen nemen.

DE ONDERDELEN

Ruwweg gesproken bestaat een machine uit vijf onderdelen: de invoer, de uitvoer, het rekenorgaan, het geheugen en de besturing. Even ruw wordt de algemene gang van een berekening weergegeven door:

het opnemen van de noodzakelijke gegevens (via de invoer naar het geheugen);

het verwerken van (= rekenen met) deze gegevens (van het geheugen via het rekenorgaan terug naar het geheugen);

het afleveren van de resultaten (van het geheugen naar de uitvoer).

De besturing zorgt ervoor, dat deze dingen geschieden, bijv. dat als er opgeteld moet worden, het rekenorgaan inderdaad ook gaat optellen, het zorgt ervoor, dat het juiste addendum op het juiste tijdstip op de juiste plaats aanwezig is, etc.

Omdat, zoals boven bleek, het geheugen de meest centrale plaats inneemt, zullen wij eerst onze aandacht hieraan schenken. De functie van het geheugen is dus ten eerste het onthouden van de noodzakelijke gegevens: men realisere zich, dat deze informatie tweeërlei is: numerieke informatie (waarden van parameters, coëfficiënten, desnoods complete matrices) en "logische informatie", d.w.z. specificatie, wàt er met al deze getallen gebeuren moet. Ten tweede onthoudt het geheugen informatie, die niet van de invoer afkomstig is, maar door de berekening gevormd is: de tussenresultaten, die niet naar de uitvoer gaan, en tenslotte de werkelijke uitkomsten, die wel naar de uitvoer gaan. Als enige informatie naar het geheugen gestuurd wordt om onthouden te worden, zegt men, "dat het in het geheugen is geschreven". Echter heeft het slechts zin, om informatie - denk aan tussenresultaten van de berekening - in het geheugen te schrijven, als deze informatie ook weer uit het geheugen gelezen kan worden; dit is inderdaad mogelijk. Het moet echter eveneens mogelijk zijn te specificeren wèlk getal in het geheugen - dat er immers honderden, ja duizenden bevatten kan! - gelezen moet worden. Dit zou niet mogelijk zijn, als de getallen in het geheugen geborgen werden gelijk knikkers in een zak; daarom is het geheugen ingedeeld in genummerde vakjes, adressen genaamd. Voor ieder getal, dat door het geheugen onthouden moet worden, wordt een bepaald adres gekozen: dit adres wordt gespecificeerd als het betroffen getal in het geheugen geschreven wordt en dit adres fungeert verder als "naam van het getal": iedere keer, dat de berekening dit getal weer nodig heeft, wordt er onder opgave van het betroffen adres in het geheugen gelezen.

De inhoud van een adres kan onbeperkt vaak gelezen worden en wordt pas vernietigd - vergeten! - als er een nieuw getal in geschreven wordt. Een geheugen dat wel "onthouden", maar niet "vergeten" kan, zou vol raken! Omdat "vergeten" alleen noodzakelijk is om nieuwe geheugenruimte vrij te maken, pleegt "vergeten" geen aparte operatie van de machine te zijn. Het is voldoende, als het schrijven de oude inhoud uitwist. De wijze waarop getallen in de adressen gerepresenteerd worden, verschilt van machine tot machine: b.v. sommige machines werken intern

tientallig, andere tweetallig. Tot zover de numerieke informatie, maar wij herinneren ons, dat het geheugen behalve de getallen, waarmede gerekend wordt, ook onthouden moet, wat er gerekend moet worden. Hiervoor wordt nu hetzelfde geheugen gebruikt, nadat voor de verschillende operaties die door het rekenorgaan verricht kunnen worden (zie onder) een nummercode is gekozen (b.v. 0 = optelling, 1 = aftrekking, 18 = vermenigvuldiging), om aan te geven, wat er met het betroffen getal gebeuren moet; met wèlk getal deze operatie uitgevoerd moet worden (dus b.v. welk getal opgeteld moet worden) wordt gespecificeerd door het adres aan te geven, waar dit getal in het geheugen te vinden is. Maar dit adres, het nummer van het vakje, is al een uit cijfers opgebouwd getal. Men kan zich dus misschien voorstellen, dat het mogelijk moet zijn, om de gang van de berekening vast te leggen in een geheugen, dat in eerste instantie alleen numerieke informatie herbergt. Zoals de informatie, die met de invoer wordt ingebracht tweeërlei is, is ook het contact met het geheugen bij werkenede machine tweeërlei: eerst moet een opdracht uit het geheugen gelezen worden; bij de uitvoering van deze opdracht worden als regel een of meer getallen van of naar het geheugen getransporteerd. Als de opdracht voltooid is, komt het signaal, dat de volgende opdracht aan de beurt is, deze wordt gelezen, etc. En zo worden de opdrachten aan elkaar geregen tot zij samen de berekening volbrengen.

Een belangrijk begrip is "de snelheid van het geheugen", die afhangt van de gemiddelde tijd, die nodig is om in een adres te schrijven of eruit te lezen ("acces-time"); hierbij is inbegrepen de eventuele wachttijd, die ontstaat, als elk adres niet continu, maar periodiek beschikbaar is: hoe kleiner deze periode is, hoe sneller het geheugen. De gebruiker moet zich dan rekenschap geven van de snelheid van de geheugens, als de machine geheugens van verschillende snelheid bezit: als de machine over een snel geheugen beschikt, dan is dat - uit financiële overwegingen - meestal vrij klein. Om van

de anders knellende beperkingen van een klein geheugen bevrijd te zijn, beschikt de machine dan vaak over een groot, langzaam geheugen. Door op het juiste moment bepaalde informatie uit het langzame geheugen "naar voren te halen" en in het snelle te copiëren, kan de tijdsduur van een berekening aanmerkelijk bekort worden.

Een tweede indeling is in "dood geheugen" en "levend geheugen". Het geheugen, zoals het boven beschreven is, is het zg. levend geheugen, d.w.z. de inhoud ervan kan, omdat erin geschreven kan worden, door de machine worden gewijzigd. Er is eveneens emplooi voor een geheugen, waarvan de inhoud niet door de machine gewijzigd kan worden: geheugen met standaard informatie, zg. dood geheugen.
(In zekere zin is de geponste band, zie onder bij de invoer, te beschouwen als dood geheugen, dat tegen geringe prijs schier onbeperkt uitgebreid kan worden.)

Als voorbeelden van geheugens noemen wij: kwik vertragingslijn, nikkel vertragingslijn, statische magnetische vertragingslijn, Williams-buizen, magnetische trommel, ferrietkernen geheugen, magnetische band (geponste band en kaarten (maar één keer te beschrijven)).

HET REKENORGAAN

Het is mogelijk, dat het rekenorgaan uitsluitend opereert op de inhoud van adressen: in geval van een optelling worden twee getallen uit het geheugen gelezen en de som wordt op een aangegeven adres geschreven (meer-adres code);

In een andere uitvoering wordt deze bewerking in drieën gesplitst door invoering van een soort totaal register: eerst wordt de andere term erbij opgeteld (de som wordt in het totaal register achtergelaten) en tenslotte wordt de inhoud van het totaal register op het adres, waar het antwoord moet komen te staan, gecopieerd (één-adres code). Het aantal registers van het rekenorgaan verschilt van machine tot machine. Als het rekenorgaan over een paar de beschikking heeft, pleegt dit vergeleken met één totaal register, de efficiency zeer te verhogen, omdat veel transport vermeden kan worden. Twee of drie registers is een veel voorkomend aantal.

(Men kan de registers beschouwen als "speciale hoekjes van het levend geheugen".)

De arithmetische bewerkingen waartoe de rekenorganen in staat zijn, verschillen: naast optellen en aftrekken, komen voor het nemen van de absolute waarde, vermenigvuldigen (niet altijd!), delen (niet altijd!); in oudere machines eveneens meer gezochte bewerkingen als worteltrekken, machtsverheffen.

INVOER

Hoewel waarschijnlijk alle machines een bedieningspaneel bezitten, waarmee (met schakelaars of telefoonschijven) het mogelijk is elk gewenst getal op een willekeurig adres in te vullen, is dit nimmer de gebruikelijke manier van invoer van gegevens, omdat hier teveel machinetijd mee gemoeid is. Als een bepaald probleem vaker op de machine gezet moet worden, heeft men bovendien liever de noodzakelijke informatie op een duurzaam medium, en wel zo, dat deze informatie zonder menselijke tussenkomst (foutenbron!) door de machine kan worden opgenomen.

De meest gebruikte methoden zijn ponskaarten, ponsband (telexband met vijf of zeven gaatjes) en magnetische band. Alle gegevens worden in een of andere (afgesproken) code geponst, resp. gemagnetiseerd; de keuze van deze code wordt bepaald door overwegingen, waar wij nu niet op in zullen gaan.

Machtig worden deze invoerfaciliteiten eerst dan, als de machine zelf in staat is, zijn eigen invoermedium te beschrijven: als de invoer via telexband geschiedt, is het gewenst, dat de machine ook banden kan ponsen.

UITVOER

Er zijn zoveel verschillende vormen van uitvoer, aangepast aan speciale behoeften, dat wij slechts de meest algemene zullen noemen.

Ten eerste de reeds vermelde uitvoer: schrijven op het eigen invoermedium. Dit is daarom zo belangrijk, omdat men dan in het invoermedium een schier onbeperkte uitbreiding van het geheugen gevonden heeft - zij het met vrij lange access-tijden en soms beperkte "opzoekmogelijkheid".

Ten tweede: typen op papier. Dit is van belang, omdat hier het resultaat in leesbare, reproduceerbare vorm wordt afgeleverd. Verreschrijvers of door de machine bediende electrische schrijfmachines zijn veel voorkomende verwezenlijkingen. Beide zijn langzaam (verreschrijver 7 kar/sec, schrijfmachine 10 kar/sec). Veel sneller zijn de lijnschrijvers (tot vele honderden kar/sec).

2. Het woord

In het volgende zal niet naar overdadige algemeenheid worden gestreefd. Het is dus mogelijk (zelfs zeker), dat er machines zijn of zullen zijn, waarvoor het onderstaande te beperkt is geformuleerd. Evenwel is het gemakkelijker zich bij de beschrijving aan iets concreets vast te klampen. Later kan men dan, na de eerste begripsmoeilijkheden te hebben overwonnen, altijd nog mogelijke generalisaties onder ogen zien. Weinigen hebben moderne algebra geleerd zonder al lang vertrouwd te zijn met de klassieke algebra. Waar wij de puntjes op de i zetten zullen wij dat altijd doen aan de hand van de machines ARMAC en X-1, waarvan enerzijds een aantal eigenschappen representatief is voor een zeer grote klasse machines, zodat wij niet al te zeer hoeven te specialiseren, terwijl anderzijds het voordeel aanwezig is eventueel gemaakte programma's op de machine te onderzoeken op hun correctheid.

In het geheugen en in de rekenkundige registers van de machine bevinden zich "woorden". Een woord is een rij cijfers - op een doodenkele uitzondering na tweetallige cijfers in verband met het feit, dat de physische elementen, waaruit snelle en betrouwbare machines kunnen worden opgebouwd, zodanig zijn, dat ze in twee toestanden kunnen verkeren, welke toestanden we de namen 0 en 1 kunnen geven. Welke betekenis aan een dergelijke cijferrij zal worden toegekend, wordt daarmede geenszins gesuggereerd en inderdaad kan een woord op verschillende wijzen worden geïnterpreteerd al naar gelang de machine ervan gebruik maakt.

Allereerst kan een woord geïnterpreteerd worden als "getal". In het eenvoudigste geval, nl. bij de zg. tweetallige of wel binaire machines wordt dit getal verkregen door de cijferrij te interpreteren als de normale tweetallige schrijfwijze van het getal (over details, zoals interpretatie van teken en plaats van de komma, enz. wordt later gesproken). Bij andere, zg. tientallige machines, worden de cijfers van het woord eerst gegroepeerd gedacht in groepjes van b.v. vier cijfers. Deze groepjes worden geacht de tientallige cijfers 0,1,...,9 te representeren en de aldus verkregen rij tientallige cijfers wordt nu weer als op de normale tientallige wijze geschreven getal geïnterpreteerd.

Een tweede interpretatie van een woord is een "opdracht" of algemener een aantal, b.v. twee opdrachten, een zg. opdrachten-koppel. Een opdracht is een hoeveelheid informatie, welke de machine leert welke handeling door haar moet worden verricht. Zij bestaat uit een functie-gedeelte en een numeriek gedeelte, d.w.z. enige cijfers vormen samen op een als boven beschreven wijze een getal, dat het nummer is van een van de functies waartoe de machine in staat is, b.v. optellen, vermenigvuldigen, terwijl de overige cijfers een of meer getallen representeren, die een nadere specificatie geven van de opdracht, b.v. het adres waarop het getal zich bevindt dat opgeteld moet worden bij dat wat zich op een afgesproken plaats bevindt. Het aantal functies waartoe de meeste machines in staat zijn is tamelijk gering, zodat meestal slechts een klein aantal, b.v. 5, tweetallige cijfers voldoende is om het functiegedeelte van de opdracht te specificeren. Voorts is het aantal adressen in het geheugen ook tamelijk beperkt, zodat b.v. 12 cijfers voldoende zijn voor het numerieke gedeelte. Dan zijn dus 17 cijfers voldoende voor de gehele opdracht. Anderzijds zijn er met het oog op het gemakkelijker verwerken van getallen met hoge precisie vrij veel cijfers nodig. b.v. ca 30 om een getal te specificeren. Als woordlengte kiest men dan dus b.v. 34 cijfers en er kunnen twee opdrachten binnen één volledig woord worden geborgen, wat van duidelijk belang is met het oog op de beschikbare geheugen ruimte. Een opdrachtlengte die juist even groter is dan de halve woordlengte is blijkbaar hoogst ongeschikt.
De hierboven gegeven getallen gelden voor de ARMAC. Wenst men een machtiger opdrachtcode en een groter geheugen, dan stijgen zowel het benodigde aantal cijfers in het functiegedeelte als in het adresgedeelte. De opdrachtlengte wordt dan zo groot, dat slechts één opdracht per woord geborgen wordt, Bij de X-1 b.v. heeft men 12 functiecijfers en 15 adrescijfers bij een woordlengte van 27 cijfers.

Een derde interpretatie van een woord is die van een "code", d.w.z. volgens een of ander afgesproken systeem wordt een bepaalde hoeveelheid informatie afgebeeld op de cijferrij. Soms kan de machine direct deze informatie interpreteren ter nadere specificatie van een opdracht (dit was het geval bij de oude relais-

machine ARRA), maar meestal is het aan speciale programma's overgelaten deze informatie te hanteren. Een eenvoudig voorbeeld van zulk een code is het volgende. Stel, men wenst een programma de beschikking te geven over een lijst van priemgetallen. Het meest voor de hand ligt in het geheugen een pijst van priemgetallen op te nemen. Als N het grootste priemgetal uit de lijst is, moeten wij zo ongeveer N/log N getallen bergen. Men kan echter ook een aantal woorden aan elkaar gebreid denken tot een grote cijferrij en deze als afbeelding van de natuurlijke getallen zien. Door een cijfer 0 dan wel 1 te kiezen al naar gelang het ermee corresponderende getal al dan niet priem is, verkrijgen wij nu ook een getrouwe afbeelding van alle priemgetallen tot N. Het ligt trouwens voor de hand alle even getallen over te slaan (eventueel ook 3- en 5-vouden), zodat men N/2 cijfers moet bergen en als n het aantal cijfers per woord voorstelt heeft men N/2n woorden nodig. Is b.v. n = 34, dan heeft men N/68 woorden nodig, wat bijzonder goed afsteekt tegenover N/log N. De machine moet natuurlijk uit de woorden de afzonderlijke cijfers kunnen isoleren of er althans op opereren. Indien zij dit alleen kan langs zuiver arithmetische weg, dan is er niet veel reden om deze woorden niet als getallen te beschouwen. Evenwel kunnen de meeste machines ook operaties op de cijfers van een woord uitvoeren (colleren en schuiven) en indien het programma daarvan gebruik maakt is het woord niet als getal maar als codewoord te beschouwen.

Het is nu nog niet duidelijk hoe de machine kan weten op welke wijze een woord geïnterpreteerd moet worden. Dit wordt verklaard uit het volgende functionele schema, dat geldt voor de meeste machines voor zover het de programmeur betreft, hoewel het technisch bezien niet altijd geheel correct behoeft te zijn. De machine voert achtereenvolgens "handelingen" uit, welke handelingen bestaan uit een aantal deelhandelingen, die volgens een vast patroon in twee fasen verlopen.

Fase 1

De besturing B zoekt een (half) woord in het geheugen op het adres aangegeven in een speciaal register, de opdrachtteller T. Zij plaatst dit (half) woord in een ander register, het opdrachtregister R. Dit woord wordt nu geïnterpreteerd als opdracht.

De besturing B hoogt vervolgens de inhoud van T op met ($\frac{1}{2}$)1 en gaat over in fase 2.

Fase 2

De besturing "voert opdracht in R uit". Daartoe beschouwt B het functiegedeelte f van de opdracht, dat in het F-gedeelte van R is terechtgekomen en het numerieke gedeelte n van de opdracht, dat in het N-gedeelte van R is terecht gekomen. Aan de hand van f wordt n geïnterpreteerd.

a) Bij de zg. arithmetische opdrachten, d.w.z. voor een zeker aantal waarden van f, als optellen, schoon inlezen, vermenigvuldigen, wordt opnieuw contact met het geheugen opgenomen. De besturing zoekt nu het woord in het geheugen op het adres aangegeven door n. Dit woord wordt als getal geïnterpreteerd. Het wordt naar het rekenorgaan gebracht en daar verwerkt op de door f aangegeven wijze, b.v. opgeteld bij wat zich al in de accumulator bevindt.

b) Bij de schrijfopdrachten wordt omgekeerd een getal uit het rekenorgaan (nader gespecificeerd door f) geschreven in het geheugen op het adres aangegeven door n.

c) Bij de zg. absolute arithmetische opdrachten wordt geen contact meer opgenomen met het geheugen. Er geschiedt een of andere operatie in het rekenorgaan gespecificeerd door f en n samen; n is nu geen adres maar een codewoord of een getal.

d) Bij de zg. absolute sprongopdrachten wordt de inhoud van N (dus n) getransporteerd naar T; n is nu dus weer als adres geïnterpreteerd, maar er is geen contact met het geheugen opgenomen. Na afloop van een van deze 4 deelhandelingen gaat de besturing over in fase 1.

Men ziet uit dit schema, dat zolang geen sprongopdracht wordt ontmoet de besturing een rij opdrachten uitvoert die zich op opeenvolgende adressen (evt. "halve" adressen) bevinden en wel in de daardoor gedefinieerde volgorde. Men zegt: de besturing voert een "programma" uit. Af en toe ontmoet de besturing een sprongopdracht. Het programma wordt nu voortgezet met de opdrachtenrij die begint op het adres gespecificeerd in de

sprongopdracht. Dit kan een nieuw stuk programma zijn, maar ook het zojuist uitgevoerde stuk programma. Een deelprogramma kan dus meerdere malen worden uitgevoerd. Dat is ook wel nodig omdat anders de machine binnen korte tijd het gehele programma dat zich in het geheugen, waar slechts voor een eindig aantal opdrachten plaats is, zou hebben uitgevoerd en dus slechts korte tijd automatisch zou kunnen werken. Zo'n herhaling van een programma houdt overigens in het geheel niet in, dat nu precies hetzelfde rekenproces zou worden uitgevoerd. Een van de handelingen van het programma kan immers zijn het ophogen van een of meer fundamentele parameters, die een rol spelen in de berekening. B.v. kan zo'n programma, dat steeds herhaald wordt f(x) uitrekenen en uittypen voor equidistante waarden van x, dus b.v. een tabel van x^2 produceren. Ja zelfs behoeven de opdrachten van het programma na een doorloping ervan niet meer dezelfde te zijn als ervoor! Immers kan men een opdracht of opdrachtenkoppel, dat toch ook maar een woord is en dan ergens op een adres in het geheugen staat evenals de getallen met behulp van andere opdrachten naar het rekenorgaan sturen, daar wijzigen, b.v. door het numerieke gedeelte ervan met 1 te verhogen en weer op de oorspronkelijke plaats neerschrijven. Zo'n opdracht noemen wij een "variabele opdracht". De besturing heeft er geen weet van, dat het de rij cijfers die het als getal naar het rekenorgaan stuurt, daar laat verwerken en weer terugbergt in het geheugen, kortweg, die het "manipuleert" of "verwerkt" in een later stadium van de berekening zal ontmoeten als opdracht die het moet "gehoorzamen". Deze consequentie echter van het opbergen van opdrachten en getallen in één enkel geheugen is van fundamenteel belang voor het karakter van de machine en is een van de voorwaarden van een werkelijk automatische machine.
Het is nu dus duidelijk, dat de machine een bepaald programma een aantal keren kan doorlopen en daarbij zelfs steeds wat anders kan uitrekenen maar nog niet hoe het een dergelijke cyclus opdrachten kan verlaten om een geheel ander deel van het totale programma uit te gaan voeren. Dit is met opdrachten van het besproken type weliswaar logisch mogelijk, maar verre van evident en zeer onpractisch. Vandaar dat nog een tweede element belangrijk is om een machine tot een volledige automatische te stempelen.

Dit is nl. het aanwezig zijn van minstens één sprongopdracht waarvan de preciese werking afhangt van het al dan niet vervuld zijn van een conditie, b.v. als aan de conditie voldaan is, fungeert de opdracht als gewone sprongopdracht, als aan de conditie niet voldaan is, fungeert de opdracht als skipopdracht, d.w.z. fase 2 van de besturingscyclus bestaat uitsluitend in de overgang naar fase 1. Wat die conditie is, varieert nogal. Vaak is het postief zijn van een getal in een van de registers van het rekenorgaan de conditie, bij de ARMAC en de X-1 is dit de inhoud van het conditieregister C, dat overigens slechts een enkel (teken)cijfer bevat. Dit register ontvangt zijn inhoud als nevenproduct van bepaalde opdrachten, die er b.v. het teken van een getal in kunnen laten onthouden.

3. Het getal

Wij zullen nu nog ingaan op de juiste samenhang van de grootte van een getal en de cijferrij die het getal representeert. Voorlopig is het voldoende alleen gehele getallen te beschouwen. Voorts zullen wij ons beperken tot machines die in het tweetallig stelsel werken, omdat de uitbreiding tot een algemeen talstelsel achteraf gemakkelijker geschiedt. Stel de cijferrij heeft n binaire cijfer $d_{n-1}, d_{n-2}, \ldots, d_1, d_0$. Het is duidelijk, dat er slechts $N = 2^n$ verschillende cijferrijen zijn en dat dus ook slechts hoogstens N getallen onderscheiden kunnen worden. Op een of andere wijze moet men zich dus beperken wat betreft het aantal getallen, terwijl voorts om practische redenen zowel positieve als negatieve getallen kunnen worden voorgesteld. Het ligt voor de hand, in verband met deze laatste opmerking, om alvast één cijfer, b.v. d_{n-1} te reserveren als tekencijfer, b.v. $d_{n-1} = 0$ definieert een positief getal en $d_{n-1} = 1$ definieert een negatief getal, terwijl de resterende cijferrij $d_{n-2}, \ldots, d_1, d_0$ op een of andere wijze een positief getal, nl. de absolute waarde van het getal definieert.
Dit stelsel sluit geheel aan bij onze normale schrijfwijze van gehele getallen. Er zijn inderdaad machines, waarbij dit stelsel wordt toegepast, maar er zijn technische nadelen aan verbonden. Een aantrekkelijker systeem is om alle getallen, die op een veelvoud van een ander getal $M \leqslant N$ na aan elkaar gelijk zijn, door één enkele cijferrij voor te stellen. Men werkt dus met een getalklasse modulo M in plaats van met getallen. Hierdoor is enerzijds een passende beperking bereikt, terwijl er ook anderzijds geen moeilijkheid meer is wat betreft positieve en negatieve getallen. Is b.v. $M = 100$, dan wordt er geen onderscheid gemaakt tussen de getallen +43, -57, +8143 en -357.

Er zijn nu nog twee vrijheden, nl. de keuze van $M \leqslant N$ en de juiste definitie van de koppeling van getalklassen en cijferrij. Wij beginnen met het laatste en definiëren dat een getal D, congruent d modulo M $(0 \leqslant d < N)$ wordt voorgesteld door:

$$d = \sum_{k=0}^{n-1} d_k 2^k \quad .$$

Bij gegeven d ($0 \leqslant d < N$) liggen de cijfers d_k ondubbelzinnig vast. Als $M = N$, legt D de cijfers d_k ook ondubbelzinnig vast, als $M < N$ kunnen bij één getal D meerdere representaties behoren, omdat er dan onder de getallen $0,1,2,\ldots,N-1$ zich meerdere getallen d kunnen bevinden, die met D congruent zijn modulo M.

Er zijn twee interessante waarden voor M, nl. $M = N = 2^n$ en $M = N-1 = 2^n-1$. Als $M = N$ is er zoals reeds gezegd geen dubbelzinnigheid omtrent de representatie van D. Als men twee getallen op de gewone wijze bij elkaar optelt en een eventuele overdracht naar de niet bestaande plaats d_n onderdrukt, krijgt men het juiste resultaat, aangezien dan toch hoogstens precies N zou worden onderdrukt, hetgeen juist de modulus is. Als de som van twee getallen N is, zijn zij te beschouwen als elkaars tegengestelde, immers $N \equiv 0$ (mod. N). Om uit de gegeven representatie van D die van -D te verkrijgen verandert men eerst alle énen in nullen en alle nullen in énen (dan completeren zij elkaar tot een rij énen, d.w.z. tot N-1) en telt er dan nog 1 bij op. Het omdraaien van het teken van een getal vereist dus een rekenkundige bewerking, nl. die optelling.

De tweede mogelijkheid, nl. $M = N-1$, het zg. inversensysteem, o.a. gebruikt in ARMAC en X-1 leidt allereerst tot één dubbelzinnigheid in notatie. Immers de rij nullen stelt 0 voor, maar de rij énen stelt N-1, dus ook 0 voor. Als men twee getallen optelt en er treedt een overdracht naar de niet bestaande plaats d_n op, dan moet men deze overdracht bij d_0 optellen. Zodoende laat men immers N weg, maar telt 1 bij en vermindert dus de som met de modulus N-1. Deze overdracht noemt men de ringoverdracht (en around carry). Als de som van twee getallen N-1 is, zijn zij te beschouwen als elkaars tegengestelde, immers $N-1 \equiv 0$ mod. N-1). Om uit de gegeven representatie van D die van -D te verkrijgen, verandert men dus alle nullen in énen en alle énen in nullen.

Het voordeel van het gebruik van het inversensysteem is, dat het omkeren van het teken van een getal, wat een veelvuldig voorkomende operatie is, een zuiver administratieve operatie is. Wij zullen ons hier verder toe beperken, hoewel het systeem der

echte complementen ($M = N$) op vrijwel gelijke wijze verder kan worden behandeld.

Zolang men zich beperkt tot optellen en aftrekken (dus optellen van het inverse!) behoeft men zich verder niet uit te spreken over de kwestie welk getal uit de getalklasse voorgesteld door de cijferrij, men werkelijk bedoelt. Zodra men echter de vermenigvuldiging introduceert, rijzen de moeilijkheden. Als product van twee getallen van n cijfers verwacht men nl. een getal van 2n cijfers, maar men zou natuurlijk het product ook alleen modulo M kennen, dus alleen aan de minst significante n cijfers zou men betekenis kunnen hechten. Theoretisch onmogelijk wordt het helemaal bij de deling. Daarom moeten wij een conventie vastleggen, welke uit iedere getalklasse één enkel getal als echte "waarde" van de cijferrij vastlegt. Om practische redenen moeten zowel positieve als negatieve getallen voorkomen. De conventie is deze, dat altijd dat getal zal worden bedoeld, dat in absolute waarde zo klein mogelijk is. Dit betekent, dat als $d_{n-1} = 0$ is, $D = d$ wordt geïnterpreteerd, terwijl als $d_{n-1} = 1$ is, geldt $D = d-M$, of in formule:

$$D = \sum_{k=0}^{n-2} (d_k - d_{n-1})2^k .$$

het cijfer d_{n-1} fungeert dus toch weer als tekencijfer, want $d_{n-1} = 0$ wijst op een positief getal, $d_{n-1} = 1$ op een negatief getal. Alleen in tegenstelling met de primitieve schrijfwijze: tekencijfer, absolute waarde, wordt nu een negatief getal verkregen door niet alleen het tekencijfer, maar ook alle andere cijfers te inverteren. Als b.v. $n = 5$ stelt 00011 het getal 3 voor en 11100 het getal -3. De twee representaties van nul, nl. de rij nullen resp. de rij énen hebben nu ook een betekenis, nl. die van +0 resp. -0. De nul kan dus zowel bij de positieve getallen worden geteld, als bij de negatieve en onderscheidingen als b.v. > 0; $\geq +0$; ≥ 0; ≤ 0; ≤ -0; < 0 zijn alle verschillend.

Er zij nog eens op gewezen, dat deze verenging van getalvoorstelling alleen strikt noodzakelijk is, als vermenigvuldiging en deling een rol spelen en overigens alleen slechts een gemakkelijke conventie is. Als men b.v. een rij getallen bij elkaar

optelt, waarvan men op een of andere wijze weet, dat hun som ligt tussen $+\frac{1}{2}N$ en $-\frac{1}{2}N$, dan levert de machine het antwoord correct af, ook al vallen allerlei partiële sommen buiten deze capaciteit.

Alle getallen D voldoen nu dus verder aan

$$-2^{n-1} + 1 \leqslant D \leqslant 2^{n-1} - 1 .$$

Dit heeft een ander merkwaardig resultaat. Immers het product van twee getallen is nu in absolute waarde kleiner dan 2^{2n-2}, dus als wij het met dezelfde conventies voorstelden, zouden wij slechts 2n-1 cijfers nodig hebben, dus één cijfer minder dan met twee normale woordlengten overeenkomt. Dit past niet goed in ons algemene schema van woorden met vaste lengte en daarom wordt kunstmatig een extra cijfer toegevoegd en wel als volgt. Een "dubbele-lengte getal" E wordt voorgesteld door de cijferrij $e_0, e_1, \ldots, e_{2n-1}$, met de nevenconditie $e_{n-1} = e_{2n-1}$. Deze rij wordt gesplitst gedacht in de rij $e_0, e_1, \ldots, e_{n-1}$ en de rij $e_n, e_{n+1}, \ldots, e_{2n-1}$, waaraan op de normale wijze de getallen E_0 en E_1 worden toegevoegd, die dus dank zij de nevenconditie hetzelfde teken hebben.

Tenslotte wordt gedefinieerd:

$$E = E_1 . 2^{n-1} + E_0 .$$

De getallen E_0 en E_1 zijn dus de "cijfers" van E in het "2^{n-1}-tallig" stelsel, waarbij de cijfers overigens zelf van een (zelfde) teken zijn voorzien. Natuurlijk kan men deze gedachtengang nu direct overdragen op getallen van willekeurige grootte. Het grote voordeel van dit systeem is, dat men ondanks het feit, dat de woordlengte vast is, toch weer met getallen van willekeurige grootte kan werken, terwijl alle brokstukken van het grote getal weer normale getallen zijn. Afgezien van de eindige grootte van het geheugen van de machine zijn er geen grenzen aan de grootte van de getallen waarmee de machine kan werken.

De beperking dat de cijfers $E_0, E_1, \ldots$ hetzelfde teken hebben is niet noodzakelijk. Het is alleen een gemakkelijke conventie, waaraan men goed doet zich zo veel mogelijk te houden, zoals later duidelijk zal worden.

Tot zover de behandeling van het gehele getal. Evenals op een gewone tafelrekenmachine is voor het begrip van de plaats van de komma bij breuken geen speciale apparatuur noodzakelijk. Deze kommaplaats kan men zelf interpreteren waar men wenst. Het eenvoudigste is het meestal (b.v. voor ARMAC en X-1) de komma te denken tussen het tekencijfer d_{n-1} en het meest significante echte cijfer d_{n-2}, d.w.z. men denkt zich het gehele getal gedeeld door 2^{n-1}. Aangezien het gehele getal in absolute waarde juist kleiner was dan 2^{n-1}, betekent dit, dat breuk in absolute waarde juist kleiner is dan 1. De kleinste echte positieve breuk is $p = 2^{-n+1}$. Dit is een maat voor de precisie of onderscheidingsvermogen. We noemen haar de peuter.

Het is goed om enkele schrijfwijzen te hebben, waarmee men de interpretatie van het woord kan aangeven. Zijn x en y gehele getallen, dan zij:

(x) De inhoud van adres x, gewoon een woord zonder nadere interpretatie;

[x] de inhoud van adres x, als geheel getal geïnterpreteerd,

{x} de inhoud van adres x, als breuk geïnterpreteerd;
$\{x\} = [x] . 2^{-n+1}$;

[x,y] de inhoud van adressen x en y, als dubbele lengte geheel getal geïnterpreteerd, waarbij
$[x,y] = [x]\ 2^{n-1} + [y]$;

$[x,y\} = [x] + \{y\}$;

$\{x,y\} = \{x\} + \{y\} . 2^{-n+1}$.

Natuurlijk gaat het niet aan steeds nieuwe haakjes te bedenken om getallen van meer dan dubbele lengte weer te geven. Evenwel blijkt het erg prettig om de meest voorkomende gevallen van enkele en dubbele lengte te onderscheiden, omdat dit een belangrijke rol speelt in de ingebouwde arithmetiek van de machine.

4. De opdracht

De preciese samenhang tussen de cijferrij en het woord als opdracht(en) geïnterpreteerd en de betekenis van die opdracht(en) is zo sterk verschillend bij verschillende machines, dat het nauwelijks mogelijk is een verantwoord overzicht te geven. Anderzijds is het uit ervaring gebleken, dat iemand, eenmaal met de "opdrachtencode" van een bepaalde machine vertrouwd, snel die van een andere machine leert en vooral beter de merites ervan begrijpt. Daarom zullen wij ons hier voorlopig beperken tot de code van een der Nederlandse machines en wel van de ARMAC. Voor bijzonderheden van deze code verwijzen wij naar het rapport MR 25.

Een rij opdrachten vormt een programma. Een aantal eenvoudige kleine programma's volgen hier.

Voorbeelden:

1. Als $x = [3]$, $y = [4]$, gevraagd: $x + y \Rightarrow t$ als $t = [5]$.

2,3 of 10,3 of 3,4 etc.
0,4 8,4 1,3
4,5 12,5 5,5

Waarin verschillen deze programma's?

2. Evenzo, gevraagd: $2x - y \Rightarrow t$

2,3 of 2,3
0,3 24,673
1,4 1,4
4,5 4,5 .

3. Evenzo, gevraagd: $xy \Rightarrow t$.

10,3
18,4
12,5

Onder welke beperkingen is dit programma juist?

4. Als $u = \{6\}$, $v = \{7\}$, $w = \{8\}$, gevraagd: $uv \Rightarrow w$.

10,6 of met afronding, b.v. 10,6
18,7 18,7
4,8 26,992
4,8
8,8
12,8 .

5. Evenzo, $|u| \Rightarrow w$.

```
              1024 a   2,6
                   b   4,8
              1025 a  14,1026→
                   b   5,8
a 1025 → 1026 a   ...
                   b   ...
```

6. Evenzo, $\max(x,y) \Rightarrow t$.

```
              1024 a   10,3
                   b    9,4
              1025 a   12,5  of  28,994
                   b   10,3
              1026 a   14,1027→
                   b   10,4
a 1026 → 1027 a   12,5
                   b    ...
```

Opgaven:

1. Schrijf de bovenstaande programma's in de programmeurscode zoals beschreven in MR 25. Adres 1024 worde voorgesteld door 0A0.
   Laat de volgende programma's beginnen op 0A0 en schrijf ze in de programmeurscode.
2. $16x - 5y + 29 \Rightarrow t$ (geen vermenigvuldigingen gebruiken!).
3. $-(S) \Rightarrow (S)$.
4. Als $x = 2n + 1$, gevraagd: $n \Rightarrow t$, als
   a) teken x onbekend,
   b) x positief.
5. Als $x \geq 0$, gevraagd: $4n + 1$ bepaald door $x \leq 4n + 1 < x + 4$.
6. Gegeven is dat hetzij $t = 13$, hetzij $t = -27$. Gevraagd een programma, dat t door de andere waarde vervangt.
7. Gegeven $0 < u < 1$. Gevraagd: n en $2^n u$, zodat $\frac{1}{2} \leq u < 1$.
8. Gegeven $0 < u < 1$. Gevraagd: n en $4^n u$, zodat $\frac{1}{4} \leq u < 1$.

5. Blokschema's

Als de machine aan een bepaald probleem rekent, bevat het geheugen een programma van honderden, soms duizenden opdrachten. Deze opdrachten bepalen volkomen, wàt er in onderscheiden gevallen gebeure, zij leggen de berekening volledig vast, en wel in de "taal" die voor de machine "direct verstaanbaar" is.
Voor de geïnteresseerde, die het programma inkijkt, zelfs voor de programmeur, die het programma opgesteld heeft, houdt deze "verstaanbaarheid" echter niet over: het lezen van een programma van de opdrachten alleen, zonder een blik te slaan in de explicatie, die er gelukkig doorgaans wel naast staat, vergt erkend veel geduld en doorzettingsvermogen. Dit is wel te verklaren. Eén van de redenen, dat men gauw in de veelheid van opdrachten omkomt, is zeker, dat er dikwijls veel opdrachten nodig zijn, om te berekenen, wat de lezer als één logisch geheel beschouwt (b.v. x^5 of $\sin x$). Een tweede oorzaak, waardoor de structuur van het programma neigt schuil te gaan, is daarin gelegen, dat het programma belast is met veel irrelevante informatie: de plaats, waar alle opdrachten en getallen staan, ligt in een feitelijk programma vast, terwijl voor dezelfde gang van de berekening het geheugen best anders ingedeeld had kunnen zijn en alles net zo goed op andere adressen had kunnen staan. Het belangrijkste is echter wel, dat in het programma *wel* staat, wat er *gebeurt*, maar *niet*, wat dit nu allemaal *behelst*: er staat, onder welke omstandigheden een conditionele besturingsverplaatsing gehoorzaamd wordt, wat echter de zin van deze speciale omstandigheden is, wordt aan de intelligente lezer overgelaten ... . Kortom er is een behoefte aan overzichtelijke weergave van programma's, een notatie, waarin, dank zij verzwijging van de bijkomstigheden, de essentialia niet verdrinken en waar tevens de *functie* van de stukken programma (en van de "voorzorgen") in aangegeven is.
Een dergelijke notatie bewijst onmisbare diensten, niet alleen bij het bestuderen en bespreken, maar ook bij het maken van een programma; de notatie der zg. *blokschema's* (= flow diagrams) voorziet in deze behoefte. Dat deze notatie geen direct gebruik maakt van de opdrachtencode van de betroffen machine, verzekert een niet te verwerpen algemeenheid van de blokschema's.

Anderzijds is de notatie iets minder efficiënt bij uitgekookte, "getructe" programma's: bij standaardprogramma's (b.v. voor het berekenen van de logarithme of de tweede machtswortel), bij welker opstelling men het onderste uit de kan wil halen, zal men immers niet schromen, geraffineerd van de - vaak onbedoelde! - speciale eigenschappen van de machine gebruik te maken. Het antwoord op de vraag "waarom nu juist zo?" is hier doorgaans niet te duidelijk zichtbaar, nochtans bewijst ook hier, zoals wij hopen te illustreren, de methode goede diensten.

Ten eerste worden ten dienste van de overzichtelijkheid de opdrachten in zg. blokken ingedeeld, d.w.z. groepen opdrachten, die een onafscheidelijk geheel vormen, onafscheidelijk in die zin, dat bij een rondgang(etje) van het werkend programma deze opdrachten of allemaal, of geen van allen gehoorzaamd worden. Anders beschouwd: de besturing doorloopt het programma langs bepaalde routes (één-richtingsverkeer!), overal waar routes samenvloeien of splitsen, is een zg. knooppunt. Een splitsing is in eerste instantie altijd tweevoudig (conditionele besturingsverplaatsing), anderzijds kunnen in een knooppunt door samenvloeiing willekeurig veel routes samenkomen (men kan van vele punten in de machine de besturing er naar toe laten springen). Een blok nu loopt in eerste instantie van knooppunt tot knooppunt: een blok heeft dus één ingang en één uitgang. De uitgang is enkel- of meervoudig (enkelvoudig als de ingang alleen een knooppunt door splitsing is), de uitgang kan enkel- òf tweevoudig zijn (tweevoudig, als de uitgang een knooppunt door samenvloeiing is).

Men tekent blokken door rechthoeken: wij spreken af, dat de ingang van elk blok bovenaan, de uitgang onderaan getekend wordt; deze conventie maakt pijltjes in de connecties tussen de blokken overbodig. Als de uitgang tweevoudig is, staat er boven een vraag en wij spreken af, dat de besturing het blok via de uitgang rechts onder verlaat, als het antwoord op de vraag bevestigend luidt, anders via de uitgang links onder.

Een stukje blokschema kan er dus als in fig. 1 uitzien. In de blokken (waarin nu alleen een nummer staat) wordt de handeling omschreven (zie onder), de pijltjes in de verbindingen zijn hier dit keer wel getekend.

De besturing komt - uit een extern stuk programma - links boven in blok 1; blok 1 is een cyclus, want het kan enige malen herhaald worden; bij het verlaten van de cyclus doorloopt de besturing blok 2, waarna aan de hand van een of ander criterium uitgemaakt wordt of blok 3 overgeslagen wordt of niet. Na blok 4 wordt gekeken, of de besturing weer naar blok 1 gestuurd moet worden, of dat dit stuk programma verlaten kan worden (Blok 1,2,3 en 4 kunnen b.v. samen een iteratie-schema zijn, en blok 1 een of ander cyclusje.)

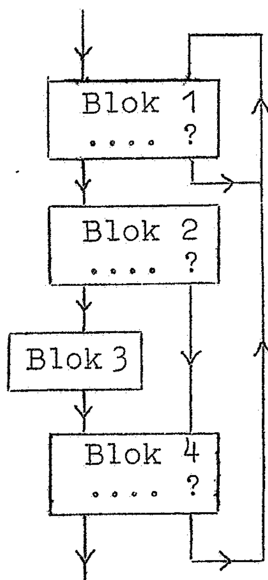


fig. 1

Als werkelijk voorbeeld volgt het blokschema, waarmede getest wordt of x = 0 is, in een machine, waarin de directe decisie gemaakt wordt op het non-negatief zijn (fig. 2 en fig.3). Hierbij is p (fig. 3 het kleinste postieve getal, dat de machine hanteren kan.

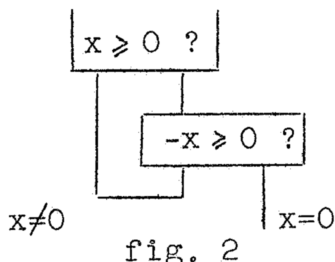


fig. 2

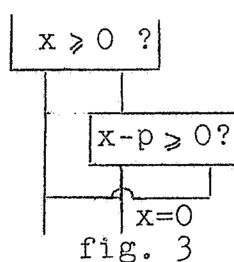


fig. 3

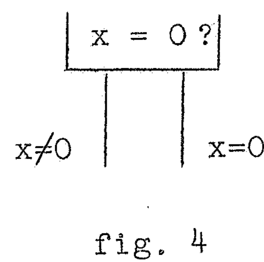


fig. 4

Opmerking

Ook al heeft de machine geen directe "nultest", toch zal dan de programmeur fig. 2 of fig. 3 in het blokschema vaak comprimeren tot de notatie als in fig. 4. Hetzelfde kan hij doen met fig. 1: als dit de (bekende standaard- of elders precies beschreven) operatie A bewerkstelligt, kan dit in de vorm van fig. 5 gecomprimeerd worden. Het verkort aangevende blok moet wel dezelfde aansluitingen met het externe programma hebben! Hiermede is het begrip blok uitgebreid: om het aantal blokken te verminderen, worden enige "minder belangrijke" knooppunten verwaarloosd.

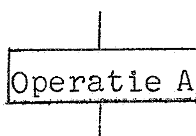


fig. 5

De notatie van de decisie's is hiermede aangegeven, echter moet ook voor de arithmetische bewerkingen in elk blok een duidelijke notatie zijn. Een notatie, die slechts voor één uitleg vatbaar is, en nauw aansluit bij de organisatie van vele machines, ontlenen wij aan H. Rutishauser: het "gerichte gelijkteken": $\Rightarrow$ (lees "vervangt"), dat gebruikt wordt ter definitie van een grootheid.

Voorbeelden:

$a + b \Rightarrow c$, d.w.z. van nu af aan wordt tot nader aankondiging met c bedoeld a + b; dit kan verwezenlijkt worden, doordat er een geheugenplaats gereserveerd wordt voor c, waarin door de opdrachten in dit blok de som a + b geplaatst wordt.
Het programma mag samenvloeien met een ander stuk (route), waarin c door iets anders vervangen is; welke c gebruikt wordt, hangt dus af van de voorgeschiedenis. In fig. 6 wordt $(a+b)^{10} \Rightarrow d$ uitgerekend, afhankelijk van de ingang (dus van vorige decisies). Het is bij het gerichte gelijkteken toegestaan, dat links en rechts hetzelfde argument voorkomt: $-p \Rightarrow p$, betekent, dat terwijl p al een of andere waarde had, bij deze route p van teken verwisseld moet worden. Fig. 7 illustrere deze notatie in het blokschema, dat met behulp van de decisie op non-negatief de operatie $|p| \Rightarrow p$ bewerkstelligt, dus p door zijn absolute waarde vervangt.

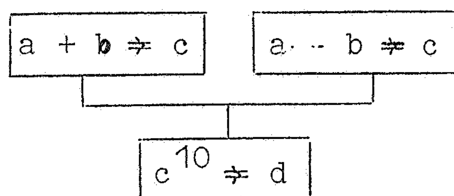


fig. 6

Thans volgt als voorbeeld de berekening van de tweedemachtswortel uit een echte breuk (d.w.z. absoluut kleiner dan 1).

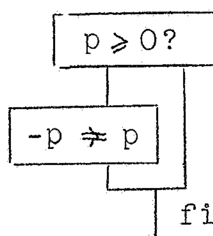


fig. 7

Blokschema $|x|^{\frac{1}{2}} \Rightarrow y$; iteratief (met gebruikmaking van de deling).

Hier wordt gebruik gemaakt van de iteratie-formule (2e orde)

$$y_{n+1} = \tfrac{1}{2}(y_n + \frac{|x|}{y_n}) = y_n + c_n, \text{ met } c_n = \tfrac{1}{2}(\frac{|x|}{y_n} - y_n)$$

$y_0 = \frac{1}{2} + \frac{1}{2}x$ (òf een waarde, waarvan we mogen verwachten, dat hij als startwaarde y_0 gunstiger is: in vele gevallen immers worden vele wortels getrokken, maar uit een continu veranderend

argument $|x|$; maar dan is het antwoord, dat de vorige worteltrekking heeft achtergelaten, waarschijnlijk de beste startwaarde, waar we over beschikken. De test of de vorig afgeleverde y als y_0 bruikbaar is, is of deze y groter is dan $|x|$, omdat anders de eerste deling een quotiënt buiten de capaciteit af zou leveren.

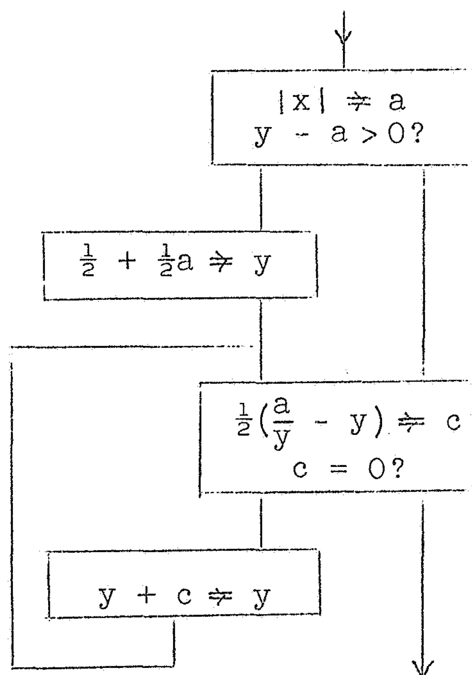


fig. 8 Blokschema $|x|^{\frac{1}{2}} \Rrightarrow y$.

Het blokschema is voor een machine, die werkt met vaste decimale (c.q. binale) punt, terwijl de capaciteit ligt tussen +1 en -1; zodra $c_n = 0$ is, wordt er niet meer geïtereerd; hierbij voert de machine een iteratie teveel uit; anderzijds duren iteraties in verband met de eenvoud van de test als de expliciete nultest is ingebouwd - korter. Het blokschema is in fig. 8 weergegeven.

Laten wij als volgend voorbeeld beschouwen de opgave, 150 getallen op te tellen, dus

$a_0 + a_1 + \ldots + a_{149} \Rrightarrow z$.

In "primitieve vorm" wordt de berekening weergegeven door het schema in fig. 9 of fig. 10.

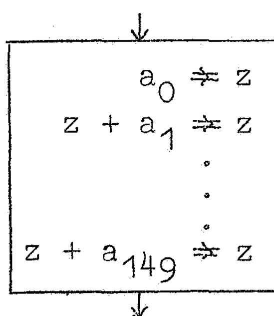


fig. 9

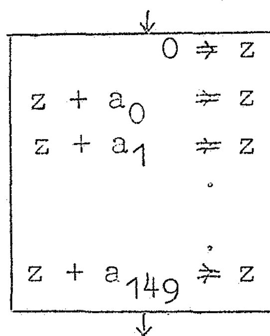


fig. 10

Als de berekening in een dergelijke vorm is weergegeven, spreken wij van een zg. verloopschema. Deze term is ontleend aan het feit, dat hierin van boven naar beneden is weergegeven

in welke vorm de berekening in de tijd verloopt, d.w.z. in welke volgorde (en eventueel onder welke voorwaarden) de arithmetische operaties uitgevoerd (kunnen) worden.

Het verloopschema is doorgaans niet gelijk aan het blokschema. Het blokschema bevat nl. een weergave van de manier, waarop de uit te voeren handelingen in de machine geformuleerd zijn; een blokschema kan b.v. cycli bevatten, dit in tegenstelling tot een verloopschema, dat wij altijd "van boven naar beneden" lezen, d.w.z. in de richting van de voortschrijdende tijd, die, zoals bekend, niet teruggezet kan worden.

Het verloopschema in fig. 10 bevat een operatie meer dan dat in fig. 9. Het voordeel is echter, dat alle a_i, inclusief a_0, nu gelijkelijk behandeld worden. Dit bergt in zich de mogelijkheid het uit te voeren proces compacter te formuleren. Deze formulering is weergegeven in het blokschema in fig. 11. (Aan dit blokschema "beantwoordt" het verloopschema in fig. 10)

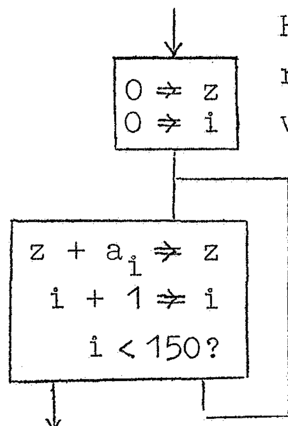


fig. 11

Het verloopschema in fig. 10 is in verticale richting als het ware "opgerimpeld" of "opgevouwen", waarbij de 150 practisch identieke regels alle zijn afgebeeld op de ene regel ("de algemene gedaante")

$$z + a_i \Rightarrow z$$

uit het blokschema.

We zien hier dan bij een verloopschema in elk stadium van de berekening op het papier de regel is aan te wijzen, die vastlegt, hoever de berekening is gevorderd. In het blokschema is dit niet meer het geval: hoever de berekening is gevorderd is nu vastgelegd in een speciaal daartoe ingevoerde hulpgrootheid i. (De hulpgrootheid i nummert tot in welk "laagje" van het opgerimpelde verloopschema de berekening is gevorderd.) Wij zouden natuurlijk ook het verloopschema van fig. 9 of fig. 10 als blokschema op hebben kunnen vatten. We zouden dan een zg. "gestrekt programma" gekregen hebben. Dit gestrekte programma heeft een goede kans sneller te zijn dan het cyclische, omdat het gemanipuleer met de hulpgrootheid i overbodig is.

(De informatie, die in het cyclische programma door i wordt vastgelegd wordt in het gestrekte programma de facto in de opdrachtteller T onthouden!)

Het gestrekte programma heeft echter twee aanzienlijke nadelen: ten eerste eist het veel geheugenruimte om het programma op te bergen; ten tweede - en dat is veel ernstiger - is de lengte van het programma afhankelijk van het aantal termen (hier = 150); het is daardoor minder flexibel. We zien, dat het blokschema in fig. 11 makkelijk aangepast kan worden aan een variabel aantal termen.

Het blokschema in fig. 11 gaat er wel vanuit, dat bij elke i het getal a_i in het geheugen vindbaar is: dit wordt als regel verwezenlijkt door de a_i zo ordelijk mogelijk (b.v. op successieve adressen) in het geheugen op te bergen.

Samenvattend: in het verloopschema is aangegeven, wat de machine moet rekenen, in het blokschema hoe de machine moet rekenen.

Als volgend voorbeeld kiezen wij de berekening van een n^{de}-graads polynoom en kiezen n = 3. Fig. 12 geeft het verloopschema, fig. 13 het blokschema.

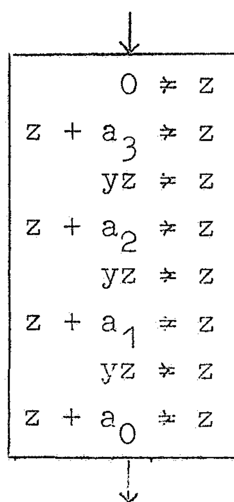


fig. 12
Verloopschema $\sum_{i=0}^{3} a_i y^i = z$

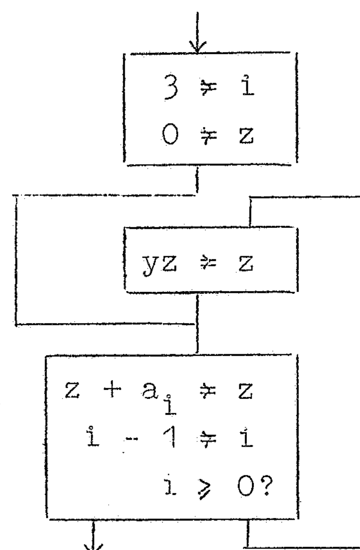


fig. 13
Blokschema $\sum_{i=0}^{3} a_i y^i = z$

Door de eerste regel uit fig. 13 door $n \Rightarrow i$ te vervangen krijgen we het blokschema voor de berekening van het n^e-graads polynoom.

Opmerking:

De berekening begint bij de coëfficiënt van de hoogste macht, omdat dan het aantal benodigde vermenigvuldigingen minimaal is.

Het volgende voorbeeld laat zien, dat in een verloopschema wel wegen mogen splitsen en samenvloeien.

Gevraagd: $\max(a_0, a_1, a_2, a_3) \Rightarrow z$.

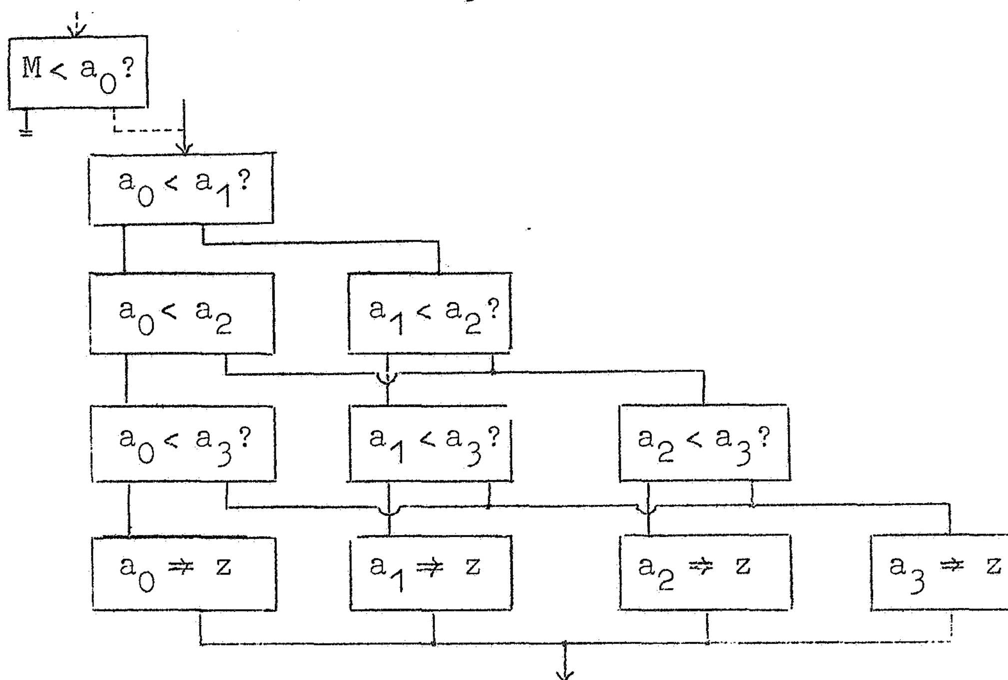


fig. 14

In fig. 14 is een verloopschema gegeven. Het met stippellijn verbonden blokje links boven is overbodig, maar onschadelijk als $M < a_0$ is, dus zeker als $M <$ alle a_i is. We zullen in het volgende aannemen, dat a priori een voldoende kleine M bekend is (b.v. als alle a_i positief zijn, is voor M elk negatief getal acceptabel). De blokken, die in fig. 14 op dezelfde regel staan, vertonen zoveel overeenkomst, dat we proberen kunnen, het verloopschema "met verticale plooitjes op te vouwen". Het resultaat staat in fig. 15. Het bovenste blok is weer overbodig, maar onder dezelfde voorwaarden onschadelijk. De laatste regel uit het verloopschema is (nagenoeg) overbodig geworden. Het opvouwen impliceert het invoeren van een nieuwe veranderlijke: reeds tijdens het zoeken van het maximum wordt de waarde van z gebruikt om in elke regel de verschillende kolommen van het verloopschema te onderscheiden.

In fig. 16 is het schema nogmaals opgevouwen, nu met horizontale plooitjes. Dit opvouwen impliceert het invoeren van nog een nieuwe hulpgrootheid, nl. i. Pas van het laatste blokschema is de omvang onafhankelijk van het aantal te onderzoeken getallen. Nu is het duidelijk, waarom M ingevoerd moest worden.

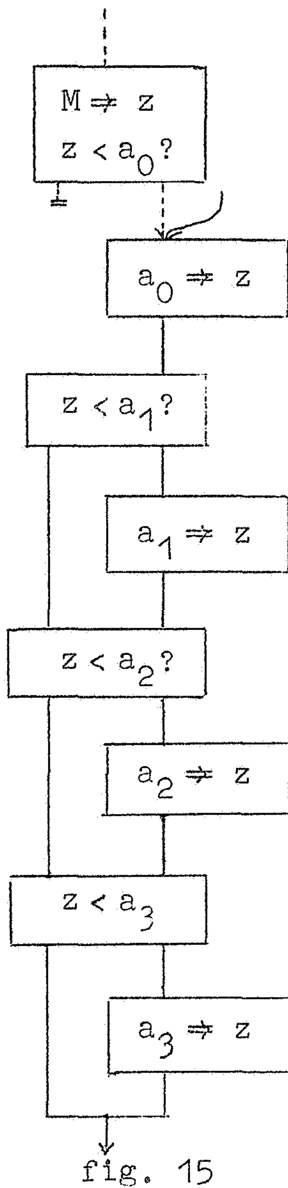


fig. 15

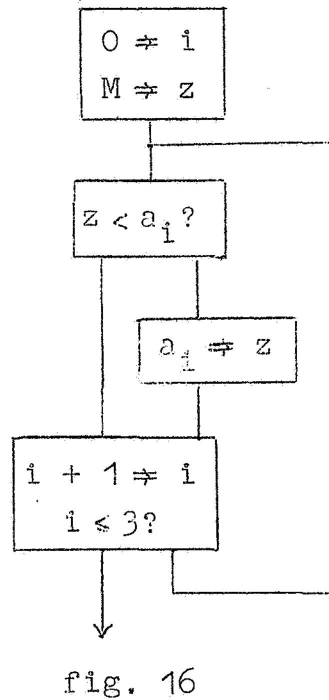


fig. 16

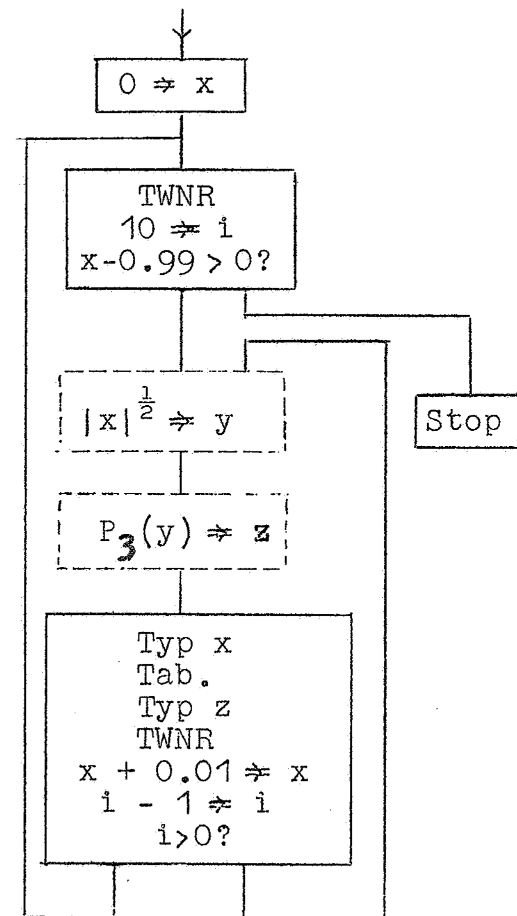


fig. 17

Stel, dat gevraagd is een 3de-graads polynoom te tabelleren, met als argument $x^{\frac{1}{2}}$, voor x = 0 (0,01)0.99. Om de 10 regels is een extra regel blank gewenst. We maken gebruik van de typ-signalen TWNR (= Terug Wagen, Nieuwe Regel) en Tab = (Tabuleert) en een typprogramma. We kunnen de beide behandelde blok-schema's incorporeren. De index i telt de regels in een "blokje" van 10. In fig. 17 is het blok-schema weergegeven. De gecomprimeerde blokken zijn door stippellijnen aangegeven. Men realisere zich, dat (b.v. als de machine in het tweetallig stelsel werkt), voor de operatie Typ mogelijkerwijs meer dan één opdracht - een stuk programma dus - nodig is.

6. Subroutines I

Het is de taak van de programmeur om een bepaald reken-proces op te bouwen met behulp van de opdrachtcode van de betroffen machine. Hij zou echter nodeloos zwaar belast worden, als hij iedere keer weer elk programma op moest bouwen uit deze minuscule bouwsteentjes, omdat haast elke berekening gesplitst kan worden in grotere onderdelen, waarvan sommige een zo algemene functie hebben, dat soortgelijke rijen opdrachten zeker in andere programma's voorkomen. Als voorbeeld noemen wij het berekenen van de (co)sinus, het trekken van de tweede- en hogere machtswortel, delen (als de machine niet over een direct deling beschikt), het vermenigvuldigen van twee complexe getallen (d.w.z. het berekenen van een reëel en een imaginair deel, als van beide factoren reëel en imaginair deel gegeven zijn), het berekenen van de logarithme, van de e-macht (al of niet voor een complex argument), het berekenen van een integraal, b.v. volgens Simpson (waarbij de integrand nader gespecificeerd dient te worden) enz. enz.

Voor dergelijke problemen zijn geschikte numerieke procedures uitgezocht en geprogrammeerd: deze programma'tjes worden standaard-subroutines of kortweg subroutines genoemd. Een subroutine is nooit een zelfstandig programma, maar moet in een "echt" programma (het zg. hoofdprogramma) geïncorporeerd worden. Om na te gaan, hoe dit incorporeren het soepelste verloopt, vergelijken wij twee machines, W en X, die alleen daarin verschillen, dat machine W wel in staat is, direct een tweede machtswortel te trekken, terwijl dit met machine X door een of ander (b.v. iteratief) proces moet gebeuren: wij nemen aan, dat een dergelijk proces voor machine X geprogrammeerd is in de vorm van een standaardsubroutine. Machine X kan elk programma voor machine W zonder meer overnemen, zolang de worteltrek-opdracht er niet in voorkomt. Als deze worteltrekopdracht er wel in voorkomt, zouden we het programma voor machine W aan machine X aan kunnen passen, door overal, waar deze opdracht voorkomt, de standaardsubroutine voor de worteltrekking in te lassen. Als het programma veel worteltrekkingen op verschillende plaatsen in het programma bevat, wordt het programma in de versie voor machine X wel heel veel langer, misschien wel zoveel langer,

dat de benodigde programmaruimte de capaciteit van het geheugen overschrijdt. Maar onze oplossing is in dit opzicht ook duidelijk verkwistend geweest: we hebben het geheugen halfvol geschreven met duplicaten van hetzelfde programma'tje, terwijl men zich af zou kunnen vragen, of één keer niet genoeg was. Dit is inderdaad het geval, maar niet zonder enige extra voorziening. Laat de worteltreksubroutine (in enkelvoud!) op een bepaalde plaats in het geheugen staan; iedere keer, dat in het programma een wortel getrokken moet worden, springt de besturing naar de standaardsubroutine. De informatie, die "de subroutine meekrijgt" is nu tweeërlei: ten eerste het getal, waaruit de wortel getrokken moet worden, ten tweede, welke worteltrekking uit het programma uitgevoerd wordt, m.a.w. waar na afloop van de worteltrekking de berekening voortgezet moet worden. Conclusie: aan het einde van de worteltrekking ontmoet de besturing een sprongopdracht met variabel adres. In het hoofdprogramma staat een sprong naar de subroutine, die (samen met al, wat er speciaal staat ter verstrekking van de noodzakelijke informatie aan de subroutine) de aanroep wordt genoemd; het gebruikelijke arrangement is, dat de besturing na het doorlopen van de subroutine, springt naar de eerste opdracht, volgend op de aanroep. Zo wordt in de ARMAC aan de subroutine in één van beide arithmetische registers (om precies te zijn, in de lage helft van het A-register) de zg. koppelopdracht (= linkorder) meegegeven, d.w.z. de inconditionele sprongopdracht naar de eerste opdracht na de aanroep. Aan het begin van de subroutine wordt de inhoud van A (= koppelopdracht) op een aan het einde van de subroutine opengelaten adres geschreven, zodat na de voltooiing van de berekening in de subroutine, de besturing "weer heengaat, vanwaar zij gekomen was" (Het effect van de opdracht, die de koppelopdracht in A plaatst, is afhankelijk van de plaats, waarop hij staat, want de opdrachtteller - de teller, die de plaats bijhoudt, waarvandaan op dat ogenblik de opdrachten gehaald worden - wordt uitgelezen.)

Het meegeven van de koppelopdracht is in de ARMAC verbonden aan de subroutinesprongen f = 22 en f = 23 (zie MR 25, p. 23,2A). Wat precies geschiedt is het volgende:

22, n als a-opdracht op adres x: 0/0; 7/x ⇒ <A> & n ⇒ (T),
22, n als b-opdracht op adres x: 0/0 6/x+1 ⇒ <A> & n ⇒ (T),
23, n als a-opdracht op adres x: 0/0; 7/x ⇒ <A> & $n+\frac{1}{2}$ ⇒ (T),
23, n als b-opdracht op adres x: 0/0; 6/x+1 ⇒ <A> & $n+\frac{1}{2}$ ⇒ (T).

Voor machine X hebben we ons doel bereikt: het programma wordt iets langer, omdat de subroutine voor de worteltrekking ergens in het geheugen moet staan, maar aan de andere kan beschikken we nu over een machine, die logisch even machtig is als machine W. De aanroep van de subroutine fungeert als uitbreiding van de opdrachtencode! Tevens zal men de subroutine volgens een of andere conventie vastleggen op het (cq. een) invoermedium van de machine; voor ARMAC hoeft men een subroutine, die men wil gebruiken, niet meer te ponsen; men gebruikt de standaard-band of een - automatisch geponste - copie. Op deze wijze reduceert men het ponswerk en daarmede een foutenbron. Voor het geval, dat de subroutine inderdaad de worteltrekking betreft is het bij ARMAC al heel eenvoudig, omdat deze behoort tot de subroutines, die zich permanent in de geblokkeerde kanalen bevinden (vgl. MR 25, p. 33 e.v.).

Een laatste voordeel van standaardsubroutines is, dat men vertrouwen kan, dat ze goed zijn: het zijn op al hun verrichtingen geteste stukken programma, ingebracht met behulp van banden uit de bibliotheek. Deze wetenschap verlicht de localisatie van fouten in een programma aanzienlijk: men weet, waar men niet hoeft te zoeken.

Een subroutine is een blok met in het algemeen een veelvoudige uitgang, nl. met een multipliciteit gelijk aan het aantal punten in het programma, vanwaar hij wordt aangeroepen. De normale blokschemanotatie (max. dubbele uitgang!) met de subroutine als gecomprimeerd blok, is dus niet aan het gebruik van subroutines aangepast. Veeleer worden in blokschema's subroutines ingelast, *als uitbreiding van de opdrachtencode*, en zo vaak genoteerd, als ze worden aangeroepen. Men duidt dan aan, dat het hier een subroutine betreft door het blok met stippellijnen te omgeven. Dit zullen we illustreren door de *gecontroleerde* berekening van sin x en cos x. In dit voorbeeld verwaarlozen we alle problemen wat betreft de capaciteit van de registers en de reductie van argumenten modulo 2π, etc.(fig. 1).

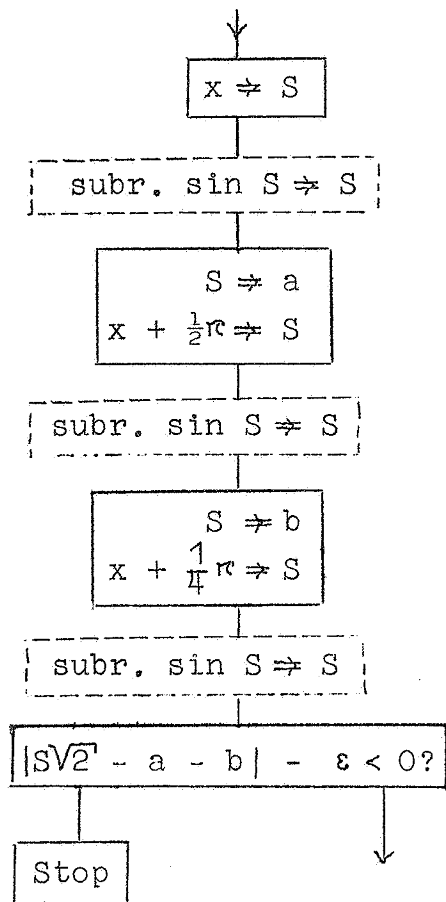


fig. 1 $\sin x \Rightarrow a$
$\cos x \Rightarrow b$
(gecontroleerd)

De berekening mag niet gecontroleerd worden door verificatie van de identiteit $\cos^2 x + \sin^2 x = 1$: als nl. b.v. sin x dicht bij nul ligt, worden door kwadratering de cijfers van sin x niet in de controle betrokken; het teken ontsnapt aan deze controle altijd. Een betere controle is de verificatie van de relatie:
$\sin(x+\frac{1}{4}\pi)=\frac{1}{2}\sqrt{2}(\sin x + \cos x)$.
Als $\sin S \Rightarrow S$ de functie van de sinussubroutine omschrijft, kan die, na optelling van $\frac{1}{2}\pi$ ook voor de berekening van cos x gebruikt worden.
In de normale procedure zal de subroutine een of ander codenummer (catalogus-nummer) bezitten, dat in het blokschema wordt vermeld.

De zojuist gebruikte sinussubroutine behoort tot een grote groep subroutines van hetzelfde type, wat betreft de hoeveelheid informatie, die erdoor verwerkt en afgeleverd wordt: uitgerekend wordt één (reële) functie van één (reëel) argument: $f(x) \Rightarrow y$. Voor de ARMAC geldt als conventie, dat hiervoor het S-register gebruikt wordt, zodat hun algemene gedaante luidt $f(S) \Rightarrow S$. Onder deze groep vallen b.v.:

1. de sinus en de cosinus;
2. de arcsinus;
3. de exponentiële functie (grondtal b.v. e, 2 of 10);
4. de logarithme (grondtal b.v. e, 2 of 10);
5. de n^{de}-machtswortel;
6. het n^{de}-machtspolynoom met bepaalde coëfficiënten;
7. het interpoleren in een bepaalde tafel.

De laatste twee voorbeelden vallen strikt genomen slechts ten dele onder deze groep: b.v. de routine voor het vijfdegraads polynoom

$$\sum_{i=0}^{5} a_i x^i = y$$

gebruikt behalve het in S meegegeven argument x bovendien de 6 coëfficiënten a_i. Omdat echter deze subroutine in één probleem doorgaans gebruikt zal worden voor vele waarden van het argument x, maar met dezelfde coëfficiënten a_i, valt, zolang het programma de coëfficiënten a_i niet wijzigt, deze routine onder het bovengenoemde type. De vijfdegraads polynoom-subroutine bestaat uit een stukje programma, benevens 6 voor de coëfficiënten gereserveerde adressen. Bij het inlezen van de subroutine worden deze coëfficiënten ter plaatse ingevuld: in de bibliotheek bevindt zich een bandje ter berekening van het *algemene* polynoom van de vijfde graad; door toevoeging van een bandje met de coëfficiënten brengt de gebruiker de subroutine ter berekening van een *bepaald* polynoom van de vijfde graad in. De interpolatie-subroutine gebruikt nog meer vaste informatie. Behalve, dat de gehele tafel in het geheugen moet staan (b.v. voor met constant interval opklimmende waarden van het argument op achtereenvolgende geheugenplaatsen), moet dan b.v. nog gespecificeerd zijn: het interval in het argument, de argumentwaarde voor het begin van de tafel, en het adres, waar de bijbehorende functiewaarde geborgen staat. Weer geldt, dat in de bibliotheek van standaard-subroutines zich het bandje met de algemene interpolatie-subroutine bevindt, waar tijdens het inlezen een bandje met parameters aan toegevoegd wordt. En zo is dan de subroutine *ter interpolatie in een bepaalde tafel* ingelezen. Men noemt deze parameters de zg. *vaste parameters*, in tegenstelling tot de zg. *programma-parameters*, d.w.z. voor de subroutine vereiste informatie, die van aanroep tot aanroep wisselt, en dus "door het programma" iedere keer meegegeven wordt. (Een voorbeeld van een enkele programma-parameter is het in S meegegeven argument in de boven gegeven voorbeelden van $f(S) \Rightarrow S$).

De bovengenoemde parameters vertegenwoordigen uitersten: het is eveneens mogelijk, dat een of andere parameter in de loop van de hele berekening af en toe eens wijzigt, denk b.v. aan het interval in de onafhankelijke veranderlijke, waarmede een stapsgewijze integratie wordt uitgevoerd. Doordat een speciaal hiervoor gereserveerd adres het interval bevat - onthoudt - is de integratie op een bepaald interval "ingesteld", bevindt de integratie-routine zich in een bepaalde toestand. Hetzelfde doet zich b.v. voor bij de subroutine, die, door tussen het typen van de getallen, volgens een bepaald patroon Tab- of TWNR-signalen te geven, de zg. pagina-indeling regelt: hier wordt, aan de hand van één of meer getallen onthouden, op welke plaats van de pagina, op welk punt van het Tab- en TWNR-patroon men is. Dergelijke routines hebben verschillende ingangen, m.a.w. verschillende aanroepen. Men kan de besturing op verschillende plaatsen laten inspringen, op deze manier verschillende functies onderscheidend. Bij de routines, die zich in verschillende toestanden kunnen bevinden, zullen naast aanroepen, waarbij de heersende toestand zich doet gelden, één of meer aanroepen voorkomen, waarmee de instelling op een bepaalde toestand bewerkstelligd wordt.
Verschillende ingangen zijn echter niet beperkt tot zg. administratieve subroutines (een type waartoe de twee zojuist genoemde onder vallen); de cosinus-subroutine berekent cosinus S, door $\frac{1}{2}\pi$ bij het argument op te tellen, en er dan de sinus van te berekenen. Deze routine kent tevens een sinus-aanroep, waarbij de aanvankelijke optelling van $\frac{1}{2}\pi$ achterwege blijft. Op deze wijze bespaart men òf het inlezen van twee routines, òf - andere mogelijkheid - de optelling van $\frac{1}{2}\pi$ door het hoofdprogramma.

Een indeling, die om der wille van de volledigheid gegeven wordt, is naar de zg. orde: de eenvoudige subroutines (als sinus of logarithme), waar geen aanroep van een andere subroutine in voorkomt, is een subroutine van de nulde orde. De subroutine echter, die op zijn beurt "subroutnes" aanroept, is van de orde n+1, als n de hoogste orde van de hulpsubroutines is. Een algemene integratie-subroutine is dus altijd van hogere orde, omdat de berekening van de (speciale) integrand als subroutine zal zijn geprogrammeerd en deze subroutine door de integratie

wordt aangeroepen. Bij een normaal programma is de hiërarchie van subroutines eenduidig.

In dit laatste voorbeeld is een nieuw element ingevoegd, de berekening van de integrand met behulp van een subroutine: dit zal over het algemeen geen standaardsubroutine zijn! De programmeur maakt zijn eigen subroutines, waarvan de functie te speciaal is, om ze standaard-subroutines te noemen. Deze procedure komt de overzichtelijkheid en de flexibiliteit van het programma zeer ten goede, maar daarover later.

In het voorafgaande is niet gepoogd een sluitende classificering van subroutines te geven. Het is meer bedoeld als een overzicht in vogelvlucht, voordat we enige speciale subroutines in detail gaan beschouwen.

7. Een uitgewerkt voorbeeld

Gevraagd te onderzoeken, op welke wijze(n) het gehele non-negatieve getal G geschreven kan worden als de som van twee quadraten. Precieser: gevraagd op te lossen

$$x^2 + y^2 = G,$$

als G een non-negatief getal is, met de nevencondities:

1: x en y zijn gehele getallen,
2: x en y voldoen aan $0 \leq x \leq y$.

(Als de eerste conditie niet gesteld was, had het probleem oneindig veel oplossingen; als de tweede conditie niet gesteld was, zou over het algemeen een oplossing in 8 niet essentieel verschillende vormen gevonden worden, nl. met negatieve tekens en verwisseling van x en y. Wanneer zijn er zo minder dan 8 gelijkwaardige oplossingen, en hoeveel?)

De oplossingen worden gevonden door dom systematisch zoeken, met volledige verwaarlozing van shortcuts geleverd door de getallentheorie. Bij een gekozen waarde van y ligt x vast: het is de vraag, of deze $x = \sqrt{G - y^2}$ een geheel getal is. Dit wordt onderzocht door bij die y enige succesieve x-waarden te proberen, d.w.z. te testen, of voor die bepaalde x en y aan $G - x^2 - y^2 = 0$ voldaan is. Zo ja, dan hebben we een oplossing gevonden, zo nee, dan weten we, dat bij deze y geen gehele x bestaat, zodra $G - x^2 - y^2$ van teken is gewisseld, en we gaan de volgende y onderzoeken. We systematiseren dit onderzoek, door met x = 0 en een te grote y te beginnen, d.w.z. $y^2 = x^2 + y^2 > G$. Zolang $x^2 + y^2 > G$, trekken we 1 van y af; als $x^2 + y^2 < G$, tellen we 1 bij x op. Zo tasten we het stukje cirkelboog $x^2 + y^2 = G$ af (zie fig. 1, waar G = 74) door het "trapje", dat door deze afspraak de cirkelboog zovaak mogelijk snijdt, zodat alle roosterpunten op de cirkelboog inderdaad gevonden worden (gekenmerkt door $x^2 + y^2 = G$). Hoe we dan door moeten gaan, specificeert de boven gemaakte afspraak niet; we maken de keuze (voor de motivering, zie later) dat dan eerst x met 1 vermeerderd wordt.

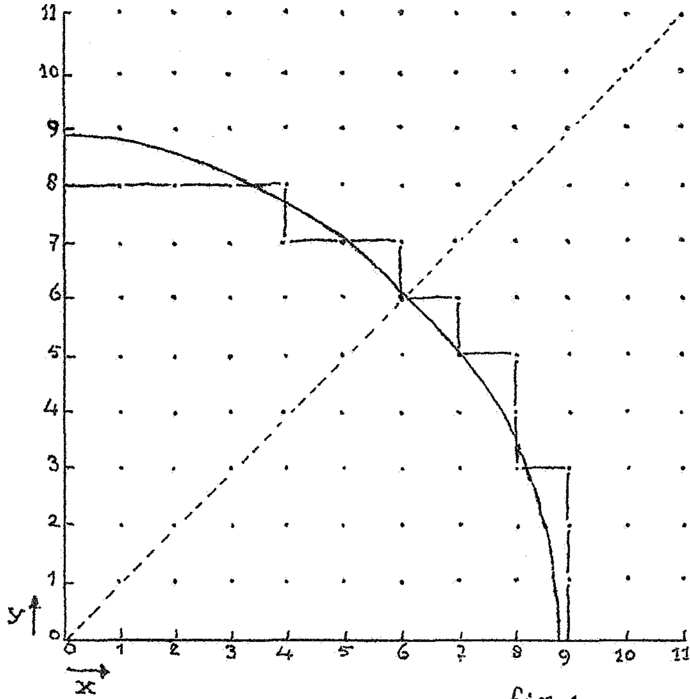


fig. 1

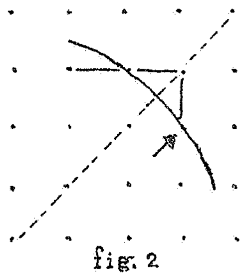


fig. 2

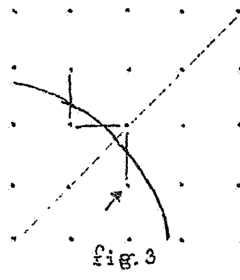


fig. 3

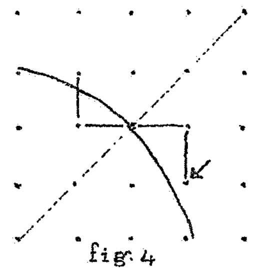


fig. 4

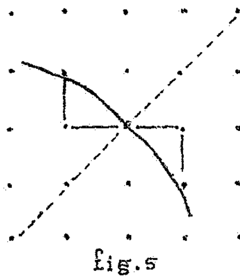


fig. 5

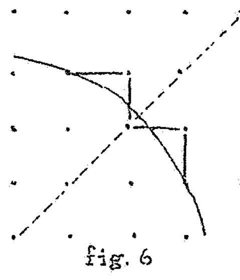


fig. 6

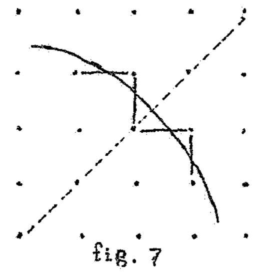


fig. 7

Ons "aftasten" moet nog in twee opzichten gecompleteerd worden:

1: voor het begin: de contructie van een geschikte y bij $x = 0$;

2: voor het einde: een test, zodat door de oplossingen aan de ongelijkheid $x \leq y$ voldaan wordt. We zoeken een startwaarde door y van nul met sprongen te vermeerderen, totdat $y^2 - G > 0$ ($y^2 - G = 0$, wat zich alleen voordoet, als G van de gedaante $G = n^2 s^2$ is, laten we schieten. Dit is in tijd gemeten een, zij het heel klein, efficiencyverlies.).

Om te onderzoeken, hoe we het beste het einde kunnen testen, bekijken we, op hoeveel verschillende manieren de cirkel $x^2 + y^2 = G$ bij de lijn $x - y = 0$ door en langs de roosterpunten kan gaan.

In fig. 2 t/m fig. 7 zijn zes opeenvolgende gevallen van gedrag in de buurt van het snijpunt getekend. (Gestippeld is de route aangegeven, die gevolgd zou zijn, als na een rooster punt op de cirkel eerst y zou zijn afgelaagd). Omdat (fig. 4) op de lijn $x = y$ nog een oplossing kan liggen, moet het zoeken naar roosterpunten pas ophouden, als $x > y$. Zodra door een nieuw punt aan deze ongelijkheid voldaan is, wordt dit punt niet meer onderzocht en is de berekening klaar. Inspectie toont aan, dat dit "alarm" niet tê laat komt, als het gegeven wordt bij de in fig. 2 t/m 7 met pijltjes aangegeven punten. Dank zij onze extra-afspraak, dat na een oplossing eerst x opgehoogd wordt, worden deze punten alle bereikt na aflaging van y. Dit betekent, dat de test op het einde hoeft te geschieden na aflaging van y, d.w.z. slechts bij 30% van het totale aantal onderzochte roosterpunten. Dit is dan ook de reden, waarom de extra-afspraak juist zo gemaakt werd.

In fig. 8 is de berekening in blokschema weergegeven. Hier wordt met "TWNR" de inloopsaanroep van de standaardtypcontrole bedoeld, de Tab en TWNR na het typen is verzwegen; n, het aantal getallen per regel, is drie. Bij fig. 8 staat het onderschrift "Het logische blokschema", omdat de arithmetische operaties hierin slechts vaag zijn aangeduid. Het is kennelijk niet de bedoeling, dat per rondgang door het cyclusje twee vermenigvuldigingen (nl. voor x^2 en y^2) uitgevoerd worden:

van deze twee quadraten is x^2 resp. y^2 de vorige keer al berekend, terwijl $(y+1)^2$ resp. $(x-1)^2$ toen gevormd werd, als nu y^2 resp. x^2 moet worden berekend: we gaan de successieve waarde van $T = G - x^2 - y^2$ berekenen, door steeds bij de vorige T de wijziging (met inbegrip van teken) op te tellen.

Hiertoe gaan we met differenties werken, elke (nieuwe) x, resp. y karakteriserend door de laatst gebruikte differentie van x^2, resp. y^2. De drie ophogingen worden dan:

1: tijdens $y + s \Rightarrow y$.

Het verschil met het vorige quadraat:

$a \equiv y^2 - (y-s)^2 = 2sy - s^2$

dus $0 \Rightarrow y$ wordt $-s^2 \Rightarrow a$

en $y + s \Rightarrow y$ wordt $a + 2s^2 \Rightarrow a$

2: tijdens $y - 1 \Rightarrow y$.

Het verschil met het vorige quadraat:

$b \equiv y^2 - (y+1)^2 = -2y - 1$

De overgang van a naar b als voor y karakteristieke grootheid impliceert de substitutie

$$\frac{-(a + s^2)}{s} - 1 \Rightarrow b$$

waarna $y - 1 \Rightarrow y$ wordt $b + 2 \Rightarrow b$.

3: tijdens $x + 1 \Rightarrow x$.

Het verschil met het vorige quadraat:

$c \equiv x^2 - (x-1)^2 = 2x - 1$

dus $0 \Rightarrow x$ wordt $-1 \Rightarrow c$

en $x + 1 \Rightarrow x$ wordt $c + 2 \Rightarrow c$

Bij de operatie "typ x" moet x uit c berekend worden: $\frac{1}{2}(c + 1) \Rightarrow x$ evenzo bij "typ y" met gebruikmaking van b: $-\frac{1}{2}(b + 1) \Rightarrow y$.

Hier schuilt de reden, waarom we bij het constateren van de beginwaarde van y het geval, dat we op een roosterpunt terecht kwamen, hebben laten schieten: dan is y nl. nog door a bepaald. Voor een relatief zeldzaam geval zou het programma merkbaar langer worden.

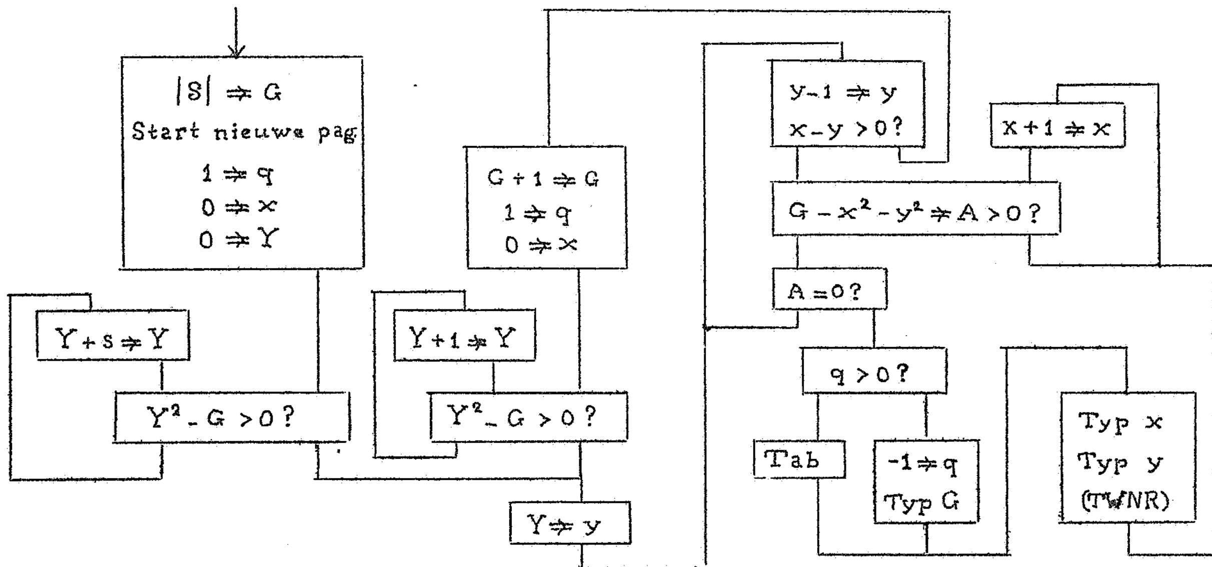


fig.8. Het logische blokschema.

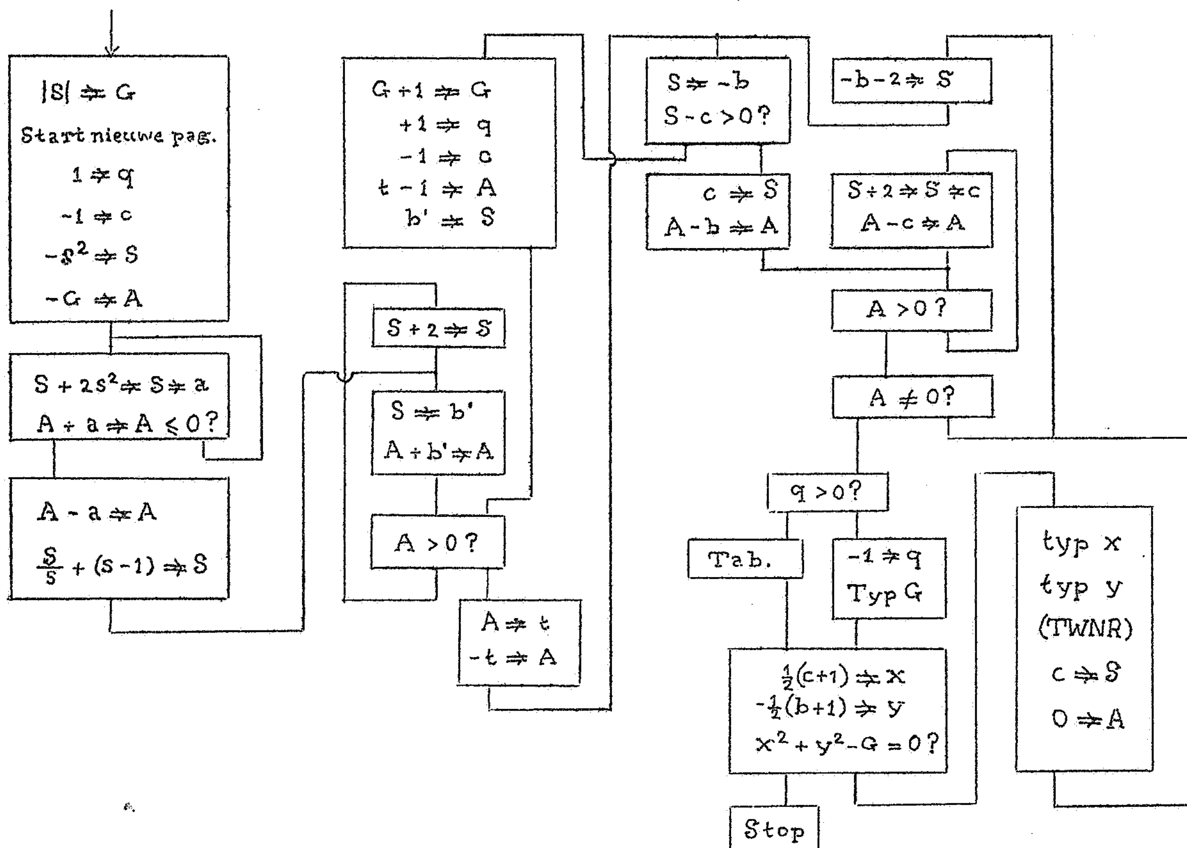


fig.9. Het arithmetisch blokschema

Deze gang van zaken wordt nu weergegeven in het arithmetisch blokschema (fig. 9). Omdat dit een klein blokschema is, kunnen we ons veroorloven, hier al gedetailleerd aan te geven, in welke registers de berekening zich afspeelt. (Uit snelheidsoverweging willen we hier nl. het aantal operaties, zoveel als makkelijk is, beperken. Een arithmetisch ingewikkeld blokschema zou door een zo gedetaileerde beschrijving er alleen maar onduidelijker op worden.)

Als regel reserveren we het A-register voor $T = G - x^2 - y^2$. Omdat we tijdens de cyclus $y + s \Rightarrow y$ moeten testen, of $y^2 - G = x^2 + y^2 - G = -T > 0$? plaatsen we hier $-T$ in A. Tijdens het typen is A bezet, maar dan weten we, dat $A = 0$ is. Tijdens de cyclus $y + s \Rightarrow y$ is S voor a aangewezen. Omdat "in de rechterhelft" de cyclus $x + 1 \Rightarrow x$ het meest doorlopen wordt, reserveren we daar het S-register voor c; zodra we in de cyclus $y - 1 \Rightarrow y$ komen, voeren we de daar aangegeven bewerkingen in S uit, d.w.z. wijziging van b, en uitvoering van de test. De oorspronkelijke voorwaarde luidde:

klaar		als	$x - y > 0$
	dus	"	$x - y - 1 \geq 0$
	"	"	$2x - 2y - 2 = (2x - 1) + (-2y - 1) \geq 0$
	"	"	$b + c \geq 0$
	"	"	$-b - c \leq 0$, als
doorgaan		als	$-b - c > 0$

En hieruit volgt, dat we bij de wijziging van y dus -b in S moeten vormen om er meteen c van af te kunnen trekken.
Voor s kiezen we 16; de deling door s is dan als schuifoperatie uit te voeren.

Het bijgaande ARMAC-programma werd aan de hand van het voorgaande gemaakt door wijlen de heer M.J. Sterk.

Het programma moet gestart worden op adres aOA1 met het eerst te splitsen getal $G = [S]$.

Voorbeeld splitsing in twee quadraten

		RD RA	O	A	O		kanaal Ao
b7 ⇒	0	11	30	X	0		-b ⇒ S
		25	2	X	12		-(b+2) ⇒ S
b30 →	1	13	30	X	0		-S ⇒ b
		9	31	X	0		S - c ⇒ S
	2	29	34	X	30		S ⩽ 0?
		15	22	A	0	→	klaar met deze G
	3	10	31	X	0		c ⇒ S
		1	30	X	0		A - b ⇒ A
	4	6	6	A	0	⇒	
a22;b6 ⇒		24	2	X	12		S + 2 ⇒ S
	5	12	31	X	0		S ⇒ c
		1	31	X	0		A - c ⇒ A
a4 →	6	28	34	X	20		A > 0?
		15	4	A	0	→	x ophogen
	7	28	0	X	0		A ≠ 0?
		14	0	A	0	→	y aflagen
	8	2	28	X	0		q ⇒ A splitsing gevonden
		28	34	X	20		A ⩾ +0?
	9	15	10	A	0	→	1ste splitsing van G
		22	6	X	26	⇒)	"acht getal geteld getypt"
	10	6	12	A	0	⇒	bij volgende splitsingen van G
a9 ⇒		5	28	X	0		-1 ⇒ c
	11	10	27	X	0		G ⇒ S
		22	0	X	28	⇒)	typ G
a10 →	12	10	31	X	0		c ⇒ S
		24	1	X	12		S + 1 ⇒ S
	13	26	1	X	30		½S ⇒ S
		12	26	X	0		S ⇒ x
	14	18	26	X	0		x^2 ⇒ S
		24	34	X	22		A ⇄ S
	15	11	30	X	0		-b ⇒ S
		25	1	X	12		S - 1 ⇒ S

		16	26	1	X	30		$\frac{1}{2}S \Rightarrow S$
			13	25	X	0		$-S \Rightarrow y$
		17	17	25	X	0		$x^2 + y^2 \Rightarrow S$
			9	27	X	0		$S - G \Rightarrow S$
		18	28	0	X	8		$S \neq 0$?
			26	16	X	0		zo ja, stop (fout!)
		19	10	26	X	0		typ x
			22	1	X	28	⇒)	
		20	10	25	X	0		typ y
			22	1	X	28	⇒)	
		21	10	31	X	0		
			27	0	X	4		
		22	7	4	A	0	⇒	
b2	⇒		26	1	X	12		$+1 \Rightarrow S$
		23	12	28	X	0		$+1 \Rightarrow q$
			13	31	X	0		$-1 \Rightarrow c$
		24	8	27	X	0		
			12	27	X	0		$G + 1 \Rightarrow G$
		25	27	1	X	4		$-1 \Rightarrow A$
			0	24	X	0		$t - 1 \Rightarrow A$ (nieuwe $y^2 - G$)
		26	10	29	X	0		
			7	28	A	0	⇒	
a29	⇒	27	24	2	X	12		
b8A1	→		12	29	X	0		
		28	0	29	X	0		
b26	→		29	34	X	20		$A \leqslant -0$?
		29	14	27	A	0	→	
			4	24	X	0		$A \Rightarrow t$
		30	3	24	X	0		$-t \Rightarrow A$
			6	1	A	0	⇒	
		31		RX1				
				RC				

Voorbeeld Splitsing in twee quadraten

	RD RA	0	A	1			Voorponsing:
⇨ 0	12	27	X	0		Start	
	15	1	A	1	→	$\lvert(S)\rvert \Rightarrow G$.	RFA 1024 Xo
1	13	27	X	0			
b0 →	22	0	X	26	⇒	Start nieuwe pag	Typ- en tab-bandje
2	26	1	X	12		$i \Rightarrow S$	RG
	12	28	X	0		$+1 \Rightarrow q$	RA 4077 Xo
3	13	31	X	0		$-1 \Rightarrow c$	+3
	11	9	A	1		$-256 \Rightarrow S$	+0
4	3	27	X	0		$-G \Rightarrow A$	+0
b6 →	8	10	A	1		$S + 512 \Rightarrow S$	RT
5	12	29	X	0		$S \Rightarrow a'$	G10 F5 S4 J1 XT
	0	29	X	0		$A + a' \Rightarrow A$	G5 F4 J1 XS
6	29	34	X	20		$A \leqslant -0$?	RC
	15	4	A	1	→		blank ⌋ bis
7	1	29	X	0		$A - a' \Rightarrow A$ (laatste keer te niet)	
	26	4	X	30		$\frac{1}{16}S \Rightarrow S$	RJ 6 22 X 25
8	25	15	X	12		$S - 15 \Rightarrow S$	+5
	7	27	A	0	⇒		+14
		RG					+6
		+ 256					+6
		+ 512					RJ 6 27 X 31
		RC					
		⋮					

8. Subroutines II

Thans is aan de orde de bespreking van enkele subroutines ter berekening van zg. elementaire functies. Het gaat hier doorgaans om functies van één onafhankelijk veranderlijke, een "argument", zoals wij zeggen (dat wij steeds in S zullen meegeven), terwijl ook het antwoord als regel uit één getal zal bestaan (dat dan, ter wille van de uniformiteit, in S achtergelaten zal worden). Wij zullen onze aandacht speciaal schenken aan die gevallen, waar de herleiding tot de directe operaties der machine (optelling, vermenigvuldiging etc.) niet evident is, dus met voorbijgaan van die functies, waar als b.v. bij de polynoomberekening, de werkelijke berekening practisch geheel in het wiskundig formalisme ligt opgesloten. Een voorbeeld hiervan hebben we al gezien, nl. een iteratieve methode ter berekening van de vierkantswortel; voor de volledigheid nemen we ook deze op in ons overzicht.

De tweedemachtswortel

Eerste methode (iteratief, met deling)

De iteratie geschiedt volgens:

$$\tfrac{1}{2}\left(\frac{a}{x_n} + x_n\right) = \tfrac{1}{2}\left(\frac{a}{x_n} - x_n\right) + x_n = x_{n+1}\ , \quad \lim x_n = a^{\frac{1}{2}}\ .$$

Bewijs: Stel $x_n = a^{\frac{1}{2}}(1 + d_n)$ d.w.z. d_n is de zg. relatieve fout van de n^{de} schatting x_n; dan is

$$\frac{a}{x_n} = \frac{a^{\frac{1}{2}}}{1 + d_n} = a^{\frac{1}{2}}(1 - d_n + d_n^2 - \ldots)\ ,$$

waaruit wij vinden:

$$x_{n+1} = \tfrac{1}{2}\left(x_n + \frac{a}{x_n}\right) = a^{\frac{1}{2}}(1 + \tfrac{1}{2}d_n^2 - \ldots\ ) = a^{\frac{1}{2}}(1 + d_{n+1})\ .$$

Omdat in eerste benadering geldt $d_{n+1} = \frac{1}{2}d_n^2$ noemt men dit iteratieschema quadratisch convergent. Per iteratiestap moet een deling uitgevoerd worden. De eerste schatting x_0 kan verschillend gemaakt worden: essentieel is dat de eerste deling geen capaciteitsoverschrijding ten gevolge heeft. Verfijning van de constructie van x_0 is de meest effectieve versnelling van dit procédé, dat mits de machine beschikt

over een snelle ingebouwde deling, als regel te verkiezen is boven de volgende methodes.

Tweede methode (iteratief, zonder deling)

Als de machine niet over een ingebouwde deling beschikt - en deze dus (zie later) geprogrammeerd moet worden, is uit tijdsoverweging de eerste methode minder aantrekkelijk. Zonder delingen zijn er iteratieschema's voor de reciproke wortel $a^{-\frac{1}{2}}$, b.v.

$$b_n(\tfrac{3}{2} - \tfrac{1}{2}ab_n^2) = b_{n+1} \quad \text{(quadratisch)},\ b = \lim b_n = a^{-\frac{1}{2}}$$

$$b_n\left\{\tfrac{15}{8} - \tfrac{10}{8}(ab_n^2) + \tfrac{3}{8}(ab_n^2)^2\right\} = b_{n+1} \quad \text{(cubisch)},\ b = \lim b_n = a^{-\frac{1}{2}}$$

Men bewijst deze formules door $b_n = a^{-\frac{1}{2}}(1 + d_n)$ te stellen en in de uitdrukkingen voor b_{n+1} te substitueren; tevens dient d_0 voldoende klein te zijn, om convergentie van d_n naar nul te verzekeren.

Door het iteratieschema vooraf te laten gaan door

$$|x| \Rightarrow a$$

door het af te sluiten met de vermenigvuldiging

$$ab \Rightarrow y$$

is de opgave

$$|x|^{\frac{1}{2}} \Rightarrow y$$

volbracht.

Echter: essentieel wordt volgens deze methodes eerst de reciproke van ons antwoord uitgerekend. Dit heeft enige directe gevolgen: ten eerste is het onmogelijk, om op deze manier de wortel uit het getal nul te trekken; ten tweede is het, zeker in een machine met vaste komma (capaciteit tussen -1 en +1) gewenst, dat a zo dicht mogelijk bij +1 ligt (als a dicht bij 0 ligt, wordt immers $a^{-\frac{1}{2}}$ verschrikkelijk groot!) In de hieronder uitgewerkte cubische iteratie wordt dit bereikt door de eerste substitutie te vervangen door

$$|x|\,2^n \Rightarrow a \quad (n \geq 0,\ \text{bepaald door } \tfrac{1}{2} \leq a < 1)$$

Deze "normering" van a wordt na afloop in rekening gebracht ("gecompenseerd") door

$$ab.2^{-\frac{1}{2}n} \Rightarrow y$$

Door het probleem te herleiden tot een breuk in het afgeperkte gebied tussen $\frac{1}{2}$ en 1 is tweeërlei winst geboekt: ten eerste is het nu mogelijk eenvoudig en snel (d.w.z. zonder vermenigvuldiging) een goede beginschatting te contrueren, ten tweede kan de iteratieformule nu zo herschreven worden, dat alle tussenresultaten in absolute waarde < 1 zijn; dit laatste zou zonder het gegeven $\frac{1}{2} \leq a < 1$ heel lastig zijn geweest. Wij geven nu de herschreven iteratie:

Beginschatting:

$$0.9510555 - a \Rightarrow c_0$$

(in plaats van het getal b_n hanteert de machine $c_n = b_n - 1$: de beginschatting b_0, die overeenkomt met de boven gegeven c_0 benadert $a^{-\frac{1}{2}}$ met een relatieve fout, kleiner dan 0.049 mits $\frac{1}{2} \leq a < 1$. Aan deze voorwaarde is voldaan.)

Iteratieschema

$$ac_n \Rightarrow z_n \ ,$$

$$c_n z_n + 2z_n + a - 1 \Rightarrow a_n \ ,$$

$$\frac{3}{8} q_n^2 - \frac{1}{2} q_n \Rightarrow t_n \ ,$$

$$c_n t_n + c_n + t_n \Rightarrow c_{n+1} \ , \quad c = \lim c_n = a^{-\frac{1}{2}} - 1.$$

Voltooiing: $ac + a = y$, waarmede de opgave $a^{\frac{1}{2}} \Rightarrow y$ volbracht is.

Opmerking:

Met de notatie $ac + a$ is bedoeld, dat het getal a bij het product ac wordt opgeteld; dit ter onderscheiding van $a(c + 1)$. De factor $(c + 1)$ zou nl. de capaciteit overschrijden. (Er is aangenomen, dat bij de vorming van een gedurige som der partiële sommen wel de capaciteit mogen overschrijden, mits het eindantwoord gegarandeerd weer binnen de capaciteit valt.)

Voor de rechtvaardiging en motivering van juist deze formules wordt verwezen naar E.W. Dijkstra, Iterative processes for the computation of elementary functions, Nachrichtentechnische Fachberichte, Band 4, 1956, p. 177-178, Vieweg, Braunschweig. Voorlopig kan de lezer controleren, dat ze equivalent zijn met de formule voor de cubische iteratie en dat op deze wijze geen capaciteitsoverschrijding optreedt.

Een laatste opmerking: dank zij de relatief al heel goede beginschatting is na twee iteraties de relatieve fout $|d_2| < 0{,}6 \times 10^{-10}$. Als deze precisie vereist en voldoende is, kan overwogen worden, om niet door te itereren tot het verschil $c_{n+1} - c_n$ klein genoeg is, maar om altijd twee iteratiestappen uit te voeren, zodat dan alle testen overbodig zijn. Het gebied voor a, waarin één iteratie al genoeg zou zijn, is nl. zo klein, dat het waarschijnlijk inefficiënt is, om voor ieder geval de testen uit te voeren, ter versnelling van slechts één op de zoveel.

Deze methode is in blokschema weergegeven in fig. 1. De standaard-subroutine voor de worteltrekking in de ARMAC gebruikt deze methode. Zie MR 26, kanaal X27.

Derde methode (repetitief, zonder deling)

De tweedemachtswortel uit een echte breuk kan eveneens zonder delingen berekend worden met een zg. repetitief proces; het quadratisch convergerende laten wij hieronder volgen. Wederom wordt in eerste instantie de reciproke wortel uitgerekend.

De relatie

$$\frac{1}{(1 - c_n)^{\frac{1}{2}}} = (1 + \tfrac{1}{2}c_n)\,\frac{1}{(1 - c_{n+1})^{\frac{1}{2}}}$$

wordt opgevat als definitievergelijking van c_{n+1}; oplossing van c_{n+1} geeft

$$c_n^2\left(\tfrac{3}{4} + \tfrac{1}{4}c_n\right) = c_{n+1} .$$

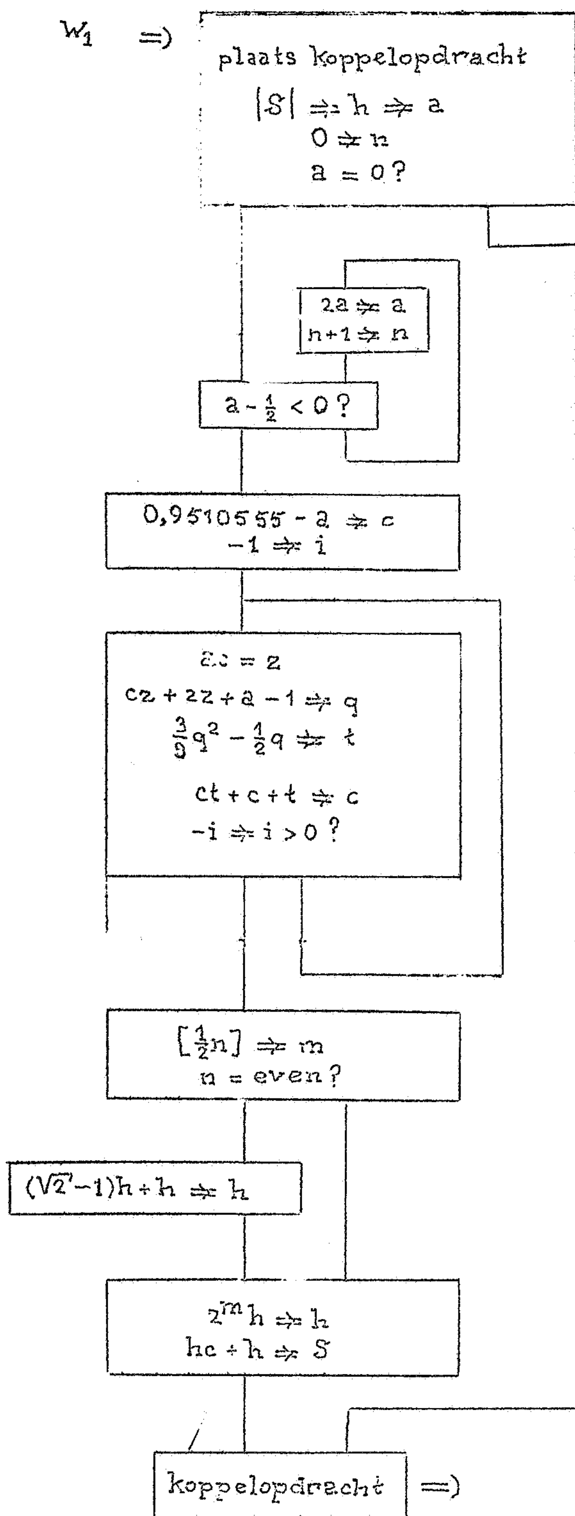


fig. 1. Blokschema $|S|^{\frac{1}{2}} \Rightarrow S$, d.w.z. worteltrekking uit een echte breuk (iteratief).

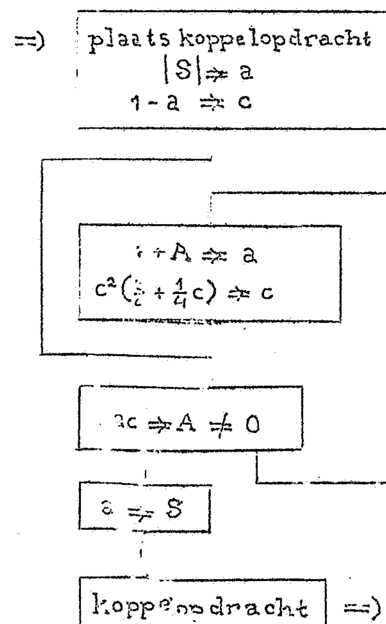


fig. 2. $|S|^{\cdot} \Rightarrow S$; repetitief

Vatten wij deze relatie op als recurrente betrekking, dan geldt voor $|c_0| < 1$ zeker $\lim c_n = 0$, m.a.w.

$$\lim \frac{1}{(1 - c_n)^{\frac{1}{2}}} = 1$$

Door echter de formule in zijn eerste gedaante herhaald toe te passen, vindt men, als $c_0 = 1 - a$ $(0 < a \leqslant 1)$, na vermenigvuldiging van beide leden met a:

$$a^{\frac{1}{2}} = \frac{a}{\sqrt{1 - c_0}} = a(1+\tfrac{1}{2}c_0) \frac{1}{\sqrt{1 - c_1}} = a(1+\tfrac{1}{2}c_0)(1+\tfrac{1}{2}c_1)(1+\tfrac{1}{2}c_2) \ \ldots$$

Hiermede is de wortel als oneindig, quadratisch convergerend product geschreven; zonder delingen kan de wortel hiermede getrokken worden.

In formule luidt dit proces

na de startwaarden:

$$a \Rightarrow a_0 \qquad 1 - a \Rightarrow c_0$$

de repetitie:

$$a_n + \tfrac{1}{2}a_n c_n \Rightarrow a_{n+1} \ , \qquad c_n^2(\tfrac{3}{4} + \tfrac{1}{4}c_n) \Rightarrow c_{n+1} \ ,$$

dan is $\lim a_n = a^{\frac{1}{2}}$.

In eenvoudige vorm is deze berekening in fig. 2 weergegeven. Hoewel $c_0 = 1$ als $a = 0$, hoeft voor dit uitzonderingsgeval geen extra voorzorg genomen te worden. Door weer a dichtbij +1 te nemen, wordt de convergentie versneld. (Het verschil tussen het iteratieve en het repetitieve proces is, dat bij het iteratieve proces het argument in de formule van de cyclus voorkomt, terwijl de beginschatting - betrekkelijk - willekeurig is, terwijl bij het repetitieve proces het argument zich juist in de startwaarden doet gelden, via de startwaarden "wordt medegedeeld". Een repetitief proces is dus niet als het iteratieve te versnellen door een listige "eerste schatting".)

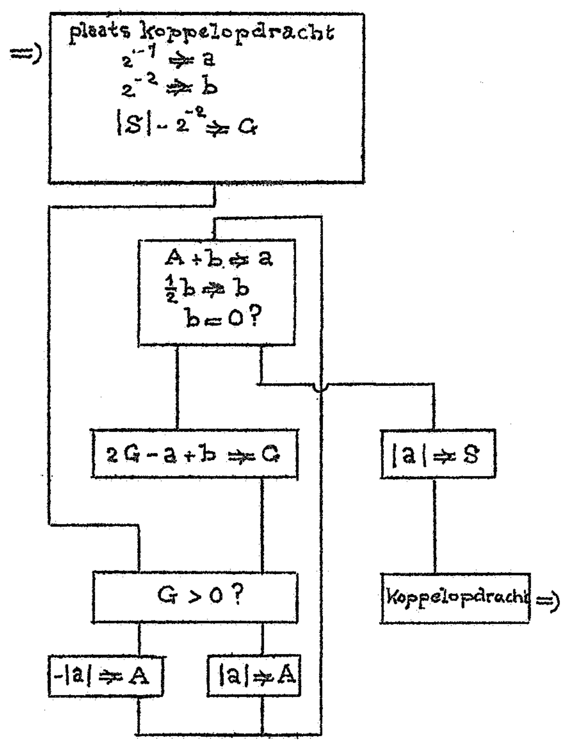


fig. 3. De geprogrammeerde worteltrekking

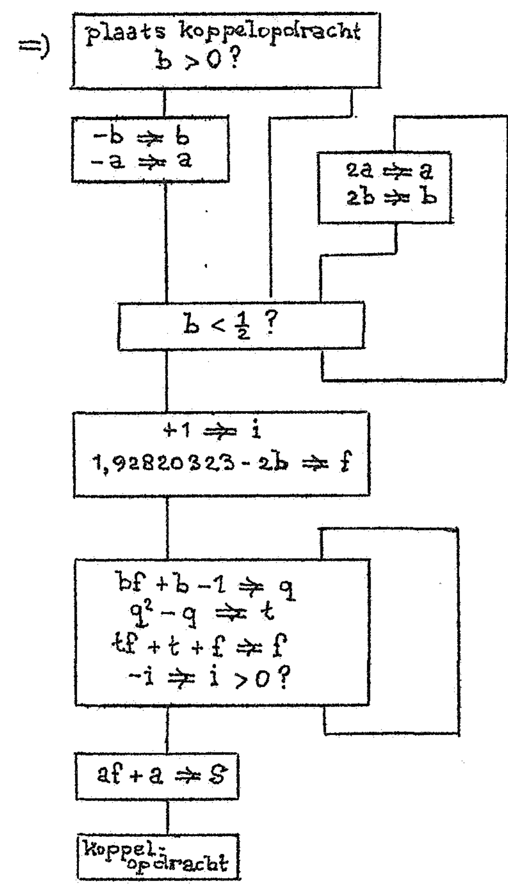


fig. 4. De iteratieve deling.

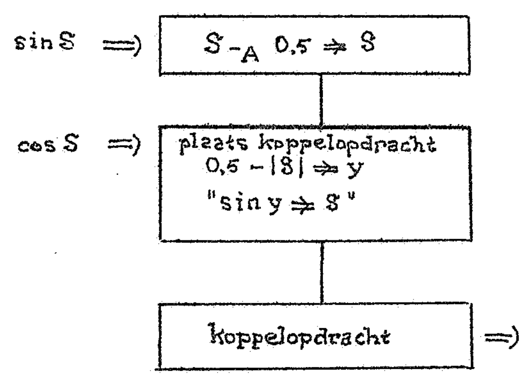


fig. 5. De herleiding bij sin x en cos x van het argument in het 1ste of 4de quadrant.

Vierde methode (de "geprogrammeerde" worteltrekking)

In fig. 3 is het blokschema weergegeven van de zg. geprogrammeerde worteltrekking. Dit is een versie van de "schoolworteltrekking", met dat verschil, dat de wortel niet decimaal voor decimaal, maar binaal voor binaal wordt (De methode is geïnspireerd op een binaire machine). De wortel wordt opgebouwd als $\Sigma d_i 2^{-i}$ met $d_i = \pm 1$ (en niet $d_i = 0$ of 1). De halveringen van b (zie fig. 3) worden niet afgerond uitgevoerd: op een gegeven ogenblik is b (in de precisie van de machine)=0; dit is een indicatie voor het einde. Fig. 3 geeft een blokschema voor een machine waar het nemen van een absolute waarde een directe operatie is. (Het blokschema is onverfijnd, wat betreft het einde: als men eisen van afronding stelt, of deze methode op gehele getallen loslaat en voor de kleinste positieve rest geïnteresseerd is, moet het einde geraffineerder).

De geprogrammeerde worteltrekking convergeert lineair: de bepaling van elke binaal duurt even lang als de vorige. Men past evenwel dergelijke methodes toe als ook deling en vermenigvuldiging niet zijn ingebouwd en analoog "geprogrammeerd" moeten worden. Men mag aannemen, dat met de opdrachten, die dan wel in de machine zitten, het interne cyclusje heel compact geprogrammeerd kan worden.

De deling

Indien de machine niet beschikt over een ingebouwde deling, moet ook dit proces geprogrammeerd worden. De onderstaande methodes hebben betrekking op de deling van twee breuken, waarvan we aannemen dat het quotiënt in absolute waarde kleiner is dan 1.

Eerste methode (iteratief)

De iteratieve processen ter berekening van de reciproke b^{-1}, die in aanmerking komen, zijn:

$$c_n(2 - bc_n) = c_{n+1} \qquad \text{(quadratisch)} \qquad c = \lim c_n = b^{-1}$$

$$c_n\{3 - 3(bc_n) + (bc_n)^2\} = c_{n+1} \quad \text{(cubisch)} \qquad c = \lim c_n = b^{-1}$$

Opmerking:

Het bewijs van deze formules levert men gemakkelijk, door $c_n = b^{-1}(1 + d_n)$ in de linkerleden te substitueren: d_0 moet voldoende klein zijn, om convergentie te verzekeren. In de volgende uitwerking wordt $|d_0|$ heel veel kleiner gekozen, om snelle convergentie te garanderen.
Hier doen zich, bij de aanpassing aan een machine met vaste komma dezelfde moeilijkheden voor, als bij de iteratieschema's voor de reciproke wortel; zij worden op dezelfde wijze opgelost. Men past de iteratie alleen toe voor $\frac{1}{2} \leqslant b < 1$ (moet dus elk geval hierop herleiden door teller en noemer met $\pm 2^n$ te vermenigvuldigen) en vindt voor het cubische proces het volgende rekenschema:

Beginschatting:

$$1{,}92820323 - 2b \Rightarrow f_0$$

(in plaats van c_n manipuleert de machine $f_n = c_n - 1$; de beginschatting, die met de boven gegeven f_0 overeenkomt, benadert b^{-1} met een relatieve fout, kleiner dan 0,072, mits $\frac{1}{2} \leqslant b < 1$).

Iteratieschema

$$bf_n + b - 1 \Rightarrow q_n \ ,$$

$$q_n^2 - q_n \Rightarrow t_n \ ,$$

$$t_n f_n + t_n + f_n \Rightarrow f_{n+1} \ , \qquad f = \lim f_n = b^{-1} - 1.$$

Voltooiing: $af + a = y$, waarnede de opgave $\frac{a}{b} \Rightarrow y$ volbracht is.

Het blokschema in fig. 4 is zo uitgevoerd, dat steeds tweemaal geïtereerd wordt; de relatieve fout is dan (mathematisch!) maximaal $0.5.10^{-10}$, voldoende voor de precisie van ARMAC. Zie MR 26, kanaal X 31.

Tweede methode (Repetitief)

Analoog aan het repetitieve proces voor de worteltrekking kan de reciproke b^{-1} uitgerekend worden als b.v. quadratisch convergerend product: men vindt dan

$$(1 - c)^{-1} = (1 + c)(1 + c^2)(1 + c^4)(1 + c^8) \ldots$$

De deling $a.b^{-1} = y$ kan dan in de volgende stappen worden ontleend:

1e. Wissel a en b van teken, als b negatief is.
2e. Normeer $a.2^n \Rightarrow a$ en $b.2^n \Rightarrow b$, zodat $\frac{1}{2} \leq b < 1$ $(n \geq 0)$.
3e. Zet beginwaarde $a \Rightarrow y_0$,
$1 - b \Rightarrow c_0$.
4e. Repeteer volgens $y_n + y_n c_n \Rightarrow y_{n+1}$ en $c_n^2 \Rightarrow c_{n+1}$ tot $y_n c_n = 0$.
Omdat $\lim y_n = y = ab^{-1}$ is hiermede het probleem opgelost.

Opmerking 1

De tweede stap is niet essentieel noodzakelijk. De convergentie kan er (bij kleine b) aanzienlijk mee versneld worden.

Opmerking 2

De nauwkeurigheid, waarmede het antwoord wordt afgeleverd is niet ideaal. Door de formules wat anders op te schrijven, is dit te verbeteren.

Het opstellen van het blokschema laten wij aan de lezer over.

Derde methode (de "geprogrammeerde" deling)

In twee gevallen bestaat de mogelijkheid dat een geprogrammeerde versie van de "schooldeling" de voorkeur verdient. Of als de machine niet over een snelle (d.w.z. ingebouwde) vermenigvuldiging beschikt (anders zijn de eerste twee methodes sneller) of als men gehele getallen op elkaar wil delen en in de rest geïnteresseerd is.

Opmerking 1

Als normeren (d.w.z. bepalen hoeveel plaatsen een getal naar links geschoven worde, opdat het "kopcijfer" - het hoogste significante cijfer - tegen de komma aanstaat) een snelle operatie is, kan men hiermede de plaats van het hoogst mogelijke quotiëntcijfer localiseren.

Opmerking 2

Men moet in elke cijferpositie "aftrekken tot het niet meer kan", d.w.z. men trekt een keer te veel af, om tekenwisseling te constateren. In plaats van die aftrekking ongedaan te maken, kan men ook de volgende keer gaan optellen. Dit versnelt het proces aanzienlijk; bij een tweetallige machine heeft dit tot gevolg, dat er in elke cijferpositie steeds één keer wordt afgetrokken of opgeteld, afhankelijk van het teken van de "partiële rest": het aantal keren hoeft niet meer geteld te worden. Voor een programma zie MR 26, kanaal X30.

De sinus en de cosinus

Voor sin x en cos x bestaan snel convergerende machtreeksen. Men kan met de laagste macht beginnen en zolang volgende termen erbij uitrekenen, tot men een term gevonden heeft, waarvan de bijdrage verwaarloosbaar is. Op deze wijze spendeert men twee vermenigvuldigingen per term. Is echter van te voren bekend, welk aantal termen voldoende is, dan kan men de normale methode voor de polynoomberekening toepassen, beginnend bij de hoogste macht; deze wijze van werken kost één vermenigvuldiging per term. Om die reden is het voordeliger om een vast aantal - d.w.z. in elk geval genoeg - termen mee te nemen: men berekent sin x en cos x met behulp van een polynoombenadering voor een bepaald interval van de x (In de practijk zijn de coëfficiënten van deze polynomen niet exact de laagste coëfficiënten van de Taylor-ontwikkeling, maar zijn ze wat gewijzigd: dit ter verhoging van de precisie). Algemeen geldt, dat de graad van het benaderende polynoom toeneemt met de vereiste precisie, èn met de grootte van het interval voor x, waarin de benadering moet gelden. Daarom wordt, indien dit elegant mogelijk is, gebruik makend van de analytische eigenschappen van de te benaderen functie, het interval van x, waarin de polynoombenadering toegepast wordt, gereduceerd. Zo zal men sin x met een polynoom berekenen voor $-\frac{1}{2}\pi \leq x \leq \frac{1}{2}\pi$; in dit interval doorloopt immers sin x al zijn mogelijke waarden, de sinusberekening voor x buiten dit interval wordt op dit geval herleid, evenals de berekening van $\cos x = \sin(x + \frac{1}{2}\pi)$. Het zijn deze herhalingen binnen het interval, waar wij nu onze aandacht aan zullen ..

schenken. De volgende herleiding is gebonden aan de logische constructie van de machine, nl. het gedrag bij capaciteitsoverschrijding door optelling (en aftrekking). In ARMAC werkt de optelling Modulo 2 (exact $2 - 2^{-33}$, hier zien wij nu van af); de inhoud van het register wordt geïnterpreteerd als liggend tussen -1 en +1. De optelling 0,7 + 0,4 geeft 1,1 - 2 = -0,9. Omdat wij van deze optelling expliciet gebruik zullen maken, voorzien wij in deze beschouwing de + en - tekens ter herinnering van de index A, dus b.v. $0{,}7 +_A 0{,}4 = -0{,}9$. Omdat de periode van de functie sin x gelijk is aan 2π, ligt het nu voor de hand, om het argument x uit te drukken in eenheden π: zoals automatisch het antwoord tussen -1 en +1 geïnterpreteerd wordt, wordt nu de som van een aantal hoeken automatisch tussen $-\pi$ en $+\pi$ geïnterpreteerd: de reductie modulo 2π geschiedt nu zonder extra voorzorgen. Voor de sinusberekening wordt nu gebruik gemaakt (argumenten in π radiaal uitgedrukt) van de relatie

$$\sin x = \sin y, \quad \text{als } 0{,}5 - |x -_A 0{,}05| \Rightarrow y$$

(en niet van $\sin x = \sin y$, als $|x +_A 0{,}5| - 0{,}5 \Rightarrow y$ in verband met de cosinusberekening, zie onder). Ter verificatie: als $x = 0{,}9$, dan $x -_A 0{,}5 = 0{,}4$ en $y = 0{,}1$; inderdaad is $\sin 0{,}9 = \sin 0{,}1$; als $x = -0{,}8$, dan $x -_A 0{,}5 = +0{,}7$ en $y = -0{,}2$. Aan $y \leqslant 0{,}5$ is kennelijk voldaan. Omdat $\cos x = \sin(x +_A 0{,}5)$ kunnen we de cosinus berekenen door optelling van 0,5; deze optelling effectueren we echter door de aftrekking van 0,5, waar de sinusberekening mee begint, over te slaan. In rudimentaire vorm is de gecombineerde subroutine $\sin\{S\}\pi \Rightarrow \{S\}$ en $\cos\{S\}\pi \Rightarrow \{S\}$ in fig. 6 weergegeven. De laatste regel "$\sin y \Rightarrow S$" staat voor de polynoombenadering; omdat dit polynoom oneven is bespaart men vermenigvuldigingen door eerst y^2 uit te rekenen, dan een polynoom in y^2, dat tenslotte met y vermenigvuldigd wordt.

Zo wordt voor ARMAC de volgende benadering gebruikt:

$$\sin \pi x = c_1 x + c_3 x^3 + c_5 x^5 + c_7 x^7 + c_9 x^9 + c_{11} x^{11} .$$

Voor $-\frac{1}{2} < x \leqslant \frac{1}{2}$. Voor de coëfficiënten zie Cecil Hastings, Approximations for digital computers, Princeton Univ. Press, 1955. Zie verder MR 26, kanaal X29.

9. Subroutines III

Als voorbeeld van de goniometrische functies hebben we enige aandacht gewijd aan de sinus en de cosinus. Thans is het inverse probleem aan de orde: de cyclometrische functies, d.w.z. gegeven de waarde, die de bepaalde goniometrische functie heeft, wordt gevraagd, voor welke hoek de betroffen goniometrische functie deze gegeven waarde aanneemt. Wij zullen als voorbeeld de arctangens (d.w.z. de inverse functie van de tangens) behandelen.

Arctan $\frac{y}{x}$

In fig. 1 is (volgens definitie van tan φ)

$$\tan\varphi = z, \quad \text{als } z = \frac{y}{x} .$$

In woorden is dus φ een hoek, waarvan de tangens gelijk is aan z, wat in formule ook uitgedrukt kan worden (volgens de definitie van de cyclometrische functies) door

$$\arctan z = \varphi$$

Indien we echter,door de "pijl" naar R een halve slag om de oorsprong te draaien, ($180^\circ=$)π radiaal optellen, wisselen zowel x, als y van teken; hun quotiënt blijft daarbij echter gelijk; dus geldt

$$\tan(\varphi + \pi) = \tan\varphi$$

algemeen (tevens draaiingen naar de andere kant uitvoeren)

$$\tan(\varphi + n\pi) = \tan\varphi \quad , \text{ voor elke gehele n.}$$

Dit is niet anders, dan het welbekende feit, dat de tangens van een hoek niet verandert door optelling of aftrekking van een geheel aantal malen π. In de taal van de arctan z betekent dit, dat deze functie door z niet eenduidig bepaald is, maar slechts op een veelvoud van π na. In de wiskunde is deze dubbelzinnigheid (beter "oneindigveelzinnigheid") opgeheven doordat een speciale keuze is gemaakt.

$\varphi = \arctan z$ is bepaald door $\tan\varphi = z$ en $-\frac{1}{2}\pi < \varphi < \frac{1}{2}\pi$ waar het geval $\varphi = \pm\frac{1}{2}$ hier buiten beschouwing gelaten is, omdat daar "$z = \pm\infty$" is.

(De situatie is analoog aan die bij de vierkantswortel van x, gedefinieerd als dat getal, dat in het quadraat gebracht, x oplevert: -2 is even goed $\sqrt{4}$ als +2; we spreken echter af, dat we met het symbool $\sqrt{x}$ de positieve wortel bedoelen dus in formule $y = \sqrt{x}$ is bepaald door $y^2 = x$ én $|y| = y$.)

Voor automatisch rekenen is bij de arctangens echter een andere "bij-afspraak" wenselijk en mogelijk. Daartoe beschouwen we de overgang tussen cartesiaanse coördinaten (x en y) en poolcoördinaten (r en φ) in het platte vlak (fig. 2). De positie van een punt P kan men specificeren door de projecties op de zg. coördinaat-assen; zo verkrijgt men de cartesiaanse coördinaten x en y. Een andere, veel gebruikte wijze is, dat men specificeert de zg. poolcoördinaten:

1. r = de afstand van P tot de oorsprong (= de lengte van de zg. "voerstraal" OP).
2. φ = de hoek, tussen de voerstraal OP en de richting van de positieve x-as, precieser: de hoek, waarover men de positieve x-as (in de richting tegen de wijzers van de klok in positief gemeten) draaien moet, opdat hij samenvalle met de richting OP.

Omdat r hier per definitie positief is, is φ bepaald op een veelvoud van 2π na (met uitzondering van het geval $r = 0$, waar φ onbepaald is).
Als r en φ gegeven zijn, volgen x en y (zie fig. 2) uit de betrekkingen:

$$x = r \cos \varphi \quad \text{en} \quad y = r \sin \varphi$$

Als x en y gegeven zijn, kunnen we omgekeerd r en φ berekenen. Volgens de stelling van Pythagoras geldt voor de voerstraal

$$r = \sqrt{x^2 + y^2}.$$

Voor φ geldt wel $\tan \varphi = \frac{y}{x}$; echter $\varphi = \arctan \frac{y}{x}$ is niet zonder meer juist! Volgens boven gegeven conventie geldt $-\frac{1}{2}\pi < \varphi < \frac{1}{2}\pi$, d.w.z. we vinden alleen punten in het rechter-halfvlak (P_1' in plaats van P_1). Daarom definiëren wij de arctan anders, als functie van twee veranderlijken, nl.

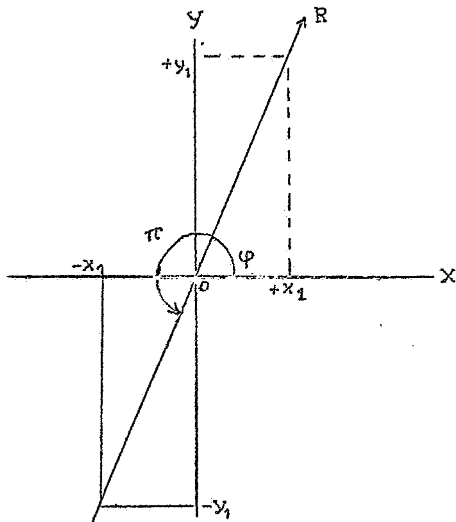


fig 1. $\tan\varphi = \frac{y_1}{x_1} = \frac{-y_1}{-x_1} = \tan(\varphi+\pi)$.

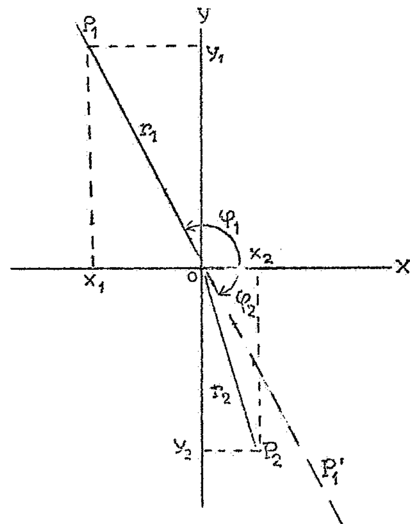


fig. 2. Cartesiaanse coördinaten en poolcoördinaten.

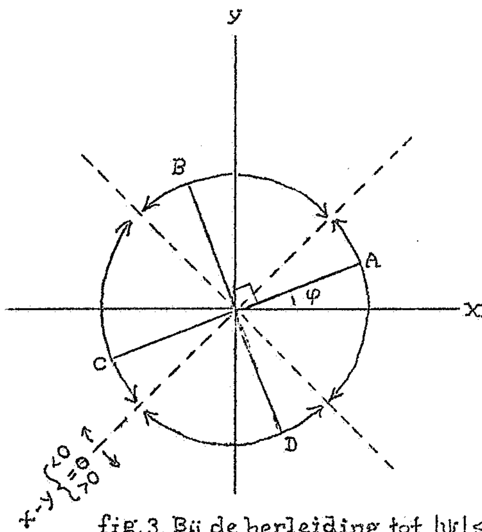


fig. 3. Bij de herleiding tot $|\psi| \leq \frac{1}{4}\pi$.

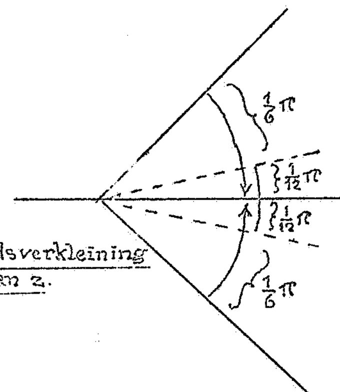


fig. 5. Gebiedsverkleining voor de arctan z.

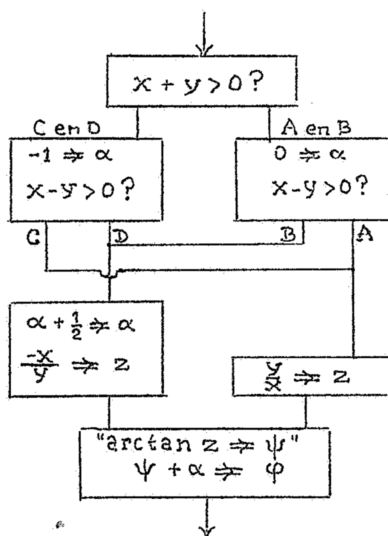


fig. 4. Het "grove" blokschema: $\arctan \frac{y}{x} \Rightarrow \varphi$.

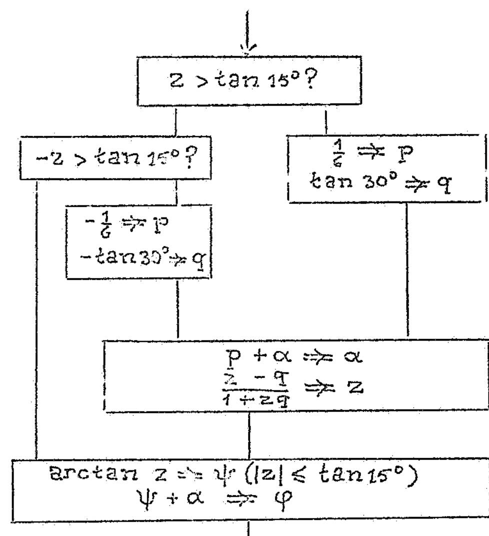


fig. 6. De uitwerking van het laatste blok in fig. 4 (zie fig. 5).

$$\varphi = \arctan \frac{y}{x} \quad \text{als } \tan\varphi = \frac{y}{x}, \text{ en } \begin{cases} 0 < \varphi < \pi \text{ als } y > 0 \\ -\pi < \varphi < 0 \text{ als } y < 0 . \end{cases}$$

Aldus gedefinieerd loopt φ "helemaal rond". φ is positief in het bovenhalfvlak, negatief in het onderhalfvlak.
(Opmerking: Bij complexe getallen vinden wij exact dezelfde situatie terug, na de terminologieverandering:

x - reëel deel
y - imaginair deel
r - modulus
φ - argument

Ook de modulus is per definitie non-negatief; het argument is op een veelvoud van 2π na bepaald.)

Het ligt in een machine met vaste komma bovendien voor de hand om niet arctan z te vragen, maar arctan $\frac{y}{x}$ (y en x afzonderlijk gegeven); immers in de helft van de gevallen zou z de capaciteit overschrijden ($|z| \to \infty$ als $\varphi \to \pm\frac{1}{2}\pi$!). De berekening wordt uitgevoerd door slechts $\varphi = \arctan z$ voor kleine $|z|$ (dus $|\varphi|$ dicht bij 0) met een polynoombenadering van niet te hoge graad te berekenen en alle andere gevallen hierop te herleiden. Dit herleiden betekent;

1. bepalen, hoe uit x en y een geschikte z volgt,
2. tevens bepalen hoe uit $\varphi = \arctan z$ (klassieke def.) onze arctan $\frac{y}{x}$ (spec. def.) gevonden wordt.

Dit geschiedt in twee stappen: eerst wordt de berekening herleid tot die van een hoek tussen $-\frac{1}{4}\pi$ en $\frac{1}{4}\pi$ (de bijbehorende tangens loopt van -1 tot +1). We maken gebruik van

$$\tan(\varphi \pm \pi) = \tan\varphi \quad \text{en} \quad \tan(\varphi \pm \tfrac{1}{2}\pi) = \frac{-1}{\tan\varphi} .$$

In fig. 3 is de volledige 2π door de stippellijnen verdeeld in 4 (gedraaide) quadranten. We beschouwen het over 90° draaibare kruis met de richtingen A, B, C en D; de hoek, die het "been A" met de positieve X-as maakt, noemen wij ψ ($|\psi| \leq \frac{1}{4}\pi$), $\tan\psi = z$. Bij een gegeven richting φ kan het kruis altijd zo gedraaid worden, dat precies één van de benen A, B, C of D met deze richting samenvalt. (Het grensgeval, dat φ langs een van

de stippellijnen valt, laten wij even buiten beschouwing).
De situatie splitst zich dus in vier gevallen (we noteren van nu af aan hoeken in eenheden π; optelling van hoeken geschiedt dus zonder extra voorzorg modulo 2π)

Geval A: $\varphi = \psi \rightarrow \tan\psi = \tan\varphi$ of $z = \frac{y}{x}$

Geval B: $\varphi = \psi + \frac{1}{2} \rightarrow \tan\psi = \frac{-1}{\tan\varphi}$ of $z = \frac{-x}{y}$

Geval C: $\varphi = \psi \pm 1 \rightarrow \tan\psi = \tan\varphi$ of $z = \frac{y}{x}$

Geval D: $\varphi = \psi - \frac{1}{2} \rightarrow \tan\psi = \frac{-1}{\tan\varphi}$ of $z = \frac{-x}{y}$.

Welk van deze vier gevallen zich voordoet, onderzoekt men door te kijken, aan welke kant van de ene, en tevens aan welke kant van de andere stippellijn men zich bevindt. Omdat de punten op de stippellijnen door $x + y = 0$, resp. $x - y = 0$ gekarakteriseerd zijn, komt dit neer op het testen van de tekens van $x + y$ resp. $x - y$. Tot zover wordt de berekening in beeld gebracht door het blokschema, fig. 4.

Opmerking 1:

Als de richting exact langs een stippellijn ligt, geldt bij het blokschema in fig. 4:

tussen A en B	als B	resumerend: "zo dicht mogelijk bij de negatieve x-as".
tussen B en C	als C	
tussen C en D	als C	
tussen D en A	als D	

Opmerking 2:

De testen op de tekens van $x + y$ en $x - y$ moeten voorzichtig uitgevoerd worden, in verband met mogelijke capaciteitsoverschrijdingen.

We hadden "het klokje rond" een arctangens gedefinieerd. In fig. 4 is de analyse gegeven, hoe deze samenhangt met de normale (klassieke) arctan z, en wel voor $|z| \leq 1$. Voor dit gebied wordt de gezochte ψ (in eenheden van π radiaal) gegeven door

$$\psi = \pi^{-1}(z - \frac{z^3}{3} + \frac{z^5}{5} - \frac{z^7}{7} + \ldots) .$$

Voor de operatie "arctan z ⇏ψ" is deze reeks niet zonder meer bruikbaar, omdat we voor grote z een onwaarschijnlijk groot aantal termen mee zouden moeten nemen, om de gewenste precisie te bereiken. Voor kleine z is de formule evenwel bruikbaar; dank zij de additieformule voor de tangens:

$$\tan(\alpha + \beta) = \frac{\tan\alpha + \tan\beta}{1 - \tan\alpha . \tan\beta}$$

kunnen we het gebied voor z, waarin we de polynoomberekening werkelijk toepassen b.v. als volgt beperken (fig. 5).
De hoek van $-\frac{1}{4}\pi$ tot $+\frac{1}{4}\pi$ wordt verdeeld in drie gelijke stukken, elk dus van $\frac{1}{6}\pi$; ligt de nog uit te rekenen hoek (waar we de tangens van kennen) in het bovenste part, dan berekenen we de tangens van de hoek, die $\frac{1}{6}\pi$ kleiner is: $|z|$ is dan klein, we berekenen arctan z met de machtreeks, en tellen bij de uitkomst weer $\frac{1}{6}\pi$ op; deze $\frac{1}{6}\pi$ wordt bij α, het toekomstig addendum opgeteld: de benadering voor arctan z hoeft uitsluitend in het middelste part toegepast te worden. Ligt de toekomstige hoek in het onderste part, dan wordt mutatis mutandis gelijkelijk gehandeld. In fig. 6 is het blokschema voor deze herleiding gegeven. Voor de uitvoering in de ARMAC, zie MR 27, subroutine SC1.

De e-macht

Uit herhaald toepassen van $a^2 - b^2 = (a + b)(a - b)$ vindt men

$$1 - e^{-x} = (1 + e^{\frac{-x}{2}})(1 + e^{\frac{-x}{2^2}})(1 + e^{\frac{-x}{2^3}})\ldots(1 + e^{\frac{-x}{2^n}})(1 - e^{\frac{-x}{2^n}})$$

of na inlassen van n factoren 2 en n factoren $\frac{1}{2}$:

$$1 - e^{-x} = \frac{(1+e^{\frac{-x}{2}})}{2} \cdot \frac{(1+e^{\frac{-x}{2^2}})}{2} \cdot \frac{(1+e^{\frac{-x}{2^3}})}{2} \cdot \ldots \cdot \underbrace{\frac{(1+e^{\frac{-x}{2^n}})}{2} \cdot \underbrace{(1-e^{\frac{-x}{2^n}})2^n}_{u_n}}_{u_{n-1}}$$

Definiëren wij $u_n = (1 - e^{\frac{-x}{2^n}})2^n$, dan is gemakkelijk af te leiden, dat geldt:

$$u_{n-1} = u_n - \frac{u_n^2}{2^{n+1}} .$$

Tevens volgt uit de wiskundige definitie van de e-macht, dat $\lim_{n=\infty} u_n = x$. In de precisie van de ARMAC is deze limiet bereikt bij $n = 32$. Kiest men nu $u_{32} = x$ (als we niet afronden, is $u_{31} = x$ voldoende) en rekent men volgens bovenstaande recurrente betrekking u_{31}, dan u_{30}, etc., uit, dan vindt men uiteindelijk:

$$u_0 = 1 - e^{-x}, \text{ dus } e^{-x} = 1 - u_0 .$$

Dit procédé is in een binaire machine geschikter dan in een decimale. Het heeft het voordeel, dat de benodigde geheugenruimte klein is; het proces is echter door het grote aantal vermenigvuldigingen niet snel.
Een snellere methode vindt men door gebruikmaking van

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \frac{x^5}{5!} + \ldots \quad (n! = n(n-1)(n-2)\ldots 3,2,1)$$

Omdat de coëfficiënten in deze machtreeks al heel snel naar nul gaan, hoeft men van deze reeks niet te veel termen mee te nemen, om de benodigde precisie te bereiken; hoe groot dit aantal is, wordt bepaald door de maximale waarde van $|x|$.
Het gebied voor x, waarin we een polynoombenadering toepassen, is weer te verkleinen, dank zij de analytische eigenschappen van de functie.

Omdat we van ons antwoord eisen, dat het binnen de capaciteit ligt, kan e^x alleen uitgerekend worden voor negatieve x, dus $-1 < x < 0$. Dit interval verdelen we in tweeën, van $-1 < x < -\frac{1}{2}$ en $-\frac{1}{2} \leq x < 0$. Als aan de laatste ongelijkheden voldaan is, berekenen we e^x met een polynoombenadering; als x in het eerste interval ligt, bedenken we, dat

$$e^x = e^{x+\frac{1}{2}-\frac{1}{2}} = 2^{-\frac{1}{2}}.e^{(x+\frac{1}{2})} .$$

Nu ligt de exponent $(x + \frac{1}{2})$ in het tweede interval en we kunnen deze e-macht met de polynoombenadering berekenen. Na afloop vermenigvuldigen we met de constante $e^{-\frac{1}{2}}$. Omdat in de polynoombenadering aanzienlijk minder termen meegenomen hoeven te worden,

is dit programma aanzienlijk sneller.

Opmerking:

De subroutine-bibliotheek voor binaire machines bevat vaak routines ter berekening van exponentiële functies met grondtal 2 in plaats van e; deze routines bevatten niets essentieel nieuws. Zie verder MR 27, subroutine Se2.

De logarithme met grondtal 2: $\log_2 x$

Als aan $\frac{1}{2} < x < 1$ voldaan is, is $\log_2 x$ een echte breuk. Als alleen maar bekend is $0 < x < 1$ kan $\log_2 x$ een geheel gedeelte hebben. Om de gehele gedeelte te bepalen begint men x zo vaak te verdubbelen, totdat het resultaat minstens $\frac{1}{2}$ is. De logarithme kan als gemengd getal in twee registers worden afgeleverd. Dus

$$y = 2^n x, \text{ waar } \tfrac{1}{2} \leqslant y < 1 .$$

Kennelijk geldt dan $\log_2 x = -n + \log_2 y$. In het volgende beperken wij ons tot het berekenen van $\log_2 x$ als aan $\frac{1}{2} \leqslant x < 1$ voldaan is.

Eerste methode (binaal voor binaal)

Start met $x = x_0$, $0 = a_0$ en vervolg met de recurrente

betrekkingen: $$\begin{cases} \text{als } x_n \geqslant \frac{1}{2}: x_n^2 = x_{n+1};\ a_n = a_{n+1} , \\ \text{als } x_n < \frac{1}{2}: (2x_n)^2 = x_{n+1};\ a_n - 2^{-n} = a_{n+1} . \end{cases}$$

Als $\frac{1}{2} \leqslant x < 1$, dan is $\lim a_n = \log_2 x$.
Dit proces heeft weer het nadeel van lineaire convergentie; voor ARMAC-precisie zou het 33 vermenigvuldigingen kosten. De charme is de geringe geheugenruimte, die het programma inneemt.

Tweede methode (polynoombenadering met deling)

Wij weten dat $\log \frac{1}{x} = -\log x$; de logarithme is dus een functie, die van teken wisselt, als men x door de reciproke vervangt. Een eenvoudige functie met deze eigenschap is

$$y(x) = \frac{x - 1}{x + 1} .$$

Tevens heeft deze functie, evenals log x, een nulpunt voor x = 1. Wij mogen dus hopen op de mogelijkheid de logarithme in-de buurt van x = 1 te benaderen door een polynoom in y van oneven graad. Deze hoop blijkt bij narekening gerechtvaardigd; met een polynoom van de 7^e of de 9^e graad kan de gebruikelijke precisie bereikt worden.

Opmerking:

Als boven gegeven loopt het gebied, waarin de polynoombenadering geldig is, van de grenzen $z \leq x < \frac{1}{2}$. Herleiding tot het gebied van $\frac{1}{2}$ tot 1 impliceert, dat men schrijft:

$$\frac{1}{2} + \log x = \text{oneven polynoom in } y, \text{ met } \frac{\frac{1}{2}\sqrt{2} - x}{\frac{1}{2}\sqrt{2} + x} = y.$$

Derde methode (polynoombenadering, zonder deling)

Men kan ook log x in een beperkt gebied door een polynoom in x benaderen; dit wordt dan van aanmerkelijk hogere graad. Een voor de hand liggende manier om deze graad te verlagen is verkleining van het interval, wat weer dank zij de analytische eigenschappen van de te benaderen functie mogelijk is: b.v. in het gebied $\frac{1}{2}\sqrt{2} \leq x < 1$ past men een polynoombenadering voor $\log_2 x$ toe: als $\frac{1}{2} \leq x < \frac{1}{2}\sqrt{2}$, dan vermenigvuldigt men eerst x met $\sqrt{2}$ (x ligt dan in het gebied, waar we de polynoom benadering toe mogen passen); de bij dit argument berekende logarithme $\log_2 x$ vermindert men na afloop met $\frac{1}{2}$. Zie MR 27, subroutine S12.

10. Snelheid

In het voorgaande hebben wij een overzicht gegeven van de opdrachtencode en de uitbreiding ervan door middel van subroutines. Daarmede is echter nog lang niet alles gezegd wat van belang is voor de opmaak van een programma. Immers daarbij spelen allerlei factoren een rol, zoals b.v.:

de grootte-orde van alle (!) getallen welke in de berekening optreden en het al of niet maatregelen nemen ("schalen") om alle getallen in absolute waarde kleiner dan de eenheid te maken;

kwesties betreffende de "service" die het programma biedt, b.v. of het programma gemakkelijk uitwendig in zijn loop beïnvloed kan worden op grond van decisies die de gebruiker maakt naar aanleiding van getypte resultaten, met name of het gemakkelijk gestart kan worden of herstart kan worden nadat b.v. een fout is opgetreden en in het algemeen de flexibiliteit van het programma, d.w.z. de eigenschap dat het gebruikt kan worden onder een grote verscheidenheid van omstandigheden dan wel sterk gespecialiseerd is;

de programmering van controles op de berekening en het typen;

de snelheid;

de gemakzucht van de programmeur, enz.

Wij beginnen met de behandeling van de snelheid, enerzijds omdat deze a priori van groot belang is en voorts omdat bij de bespreking van de overige factoren er steeds rekening mee dient te worden gehouden. Het lijkt wellicht op het eerste gezicht vreemd, dat de programmeur anders dan door het weglaten van overbodige opdrachten de snelheid van het programma noemenswaardig zou kunnen beïnvloeden. Immers deze snelheid is in eerste instantie een gegeven eigenschap van de speciale machine waarmee men werkt. Toch is het niet de bedoeling, dat de programmeur de snelheid van zijn programma tracht op te voeren door constructiewijzigingen in de machine aan te brengen. De situatie is nl. deze, dat bij vele machines de tijd benodigd voor het uitvoeren van een opdracht, niet alleen van de functie van die opdracht afhangt, maar ook van het adres waarop de opdracht zich bevindt en het adres gespecificeerd in de opdracht. Door hier rekening mee te houden kan vaak een veel betere tijd gescoord worden dan die

waarop men gemiddeld "recht heeft". Voorts kan vaak mirabile dictu de tijd drastisch beperkt worden door het aantal opdrachten drastisch maar op geschikte wijze te vergroten. Dit geldt zelfs voor bijna alle machines.

We zullen eerst de tijd benodigd voor het uitvoeren van een enkele opdracht eens bezien. Deze uitvoering valt meestal uiteen in vier delen, te weten:

1. het halen van de opdracht uit het geheugen;
2. het interpreteren van de opdracht;
3. het halen of wegbergen van een getal uit of in het geheugen;
4. het uitvoeren van de door de opdracht gespecificeerde bewerking.

Op de tijd benodigd voor de punten 2 en 4 valt, gegeven de machine niet af te dingen. Dit zijn tijden direct resulterend uit de intrinsieke snelheid van de machine. Wat betreft de tijd benodigd voor de punten 1 en 3 ligt de situatie echter geheel anders. Vele machines hebben een geheugen, waarin de verschillende adressen slechts op gezette tijden van een bepaalde tijdscyclus (trommelomwenteling) beschikbaar zijn. Wanneer nu geen bepaalde keuze uit adressen van tevoren is gemaakt zal zowel voor punt 1 als punt 3 gemiddeld een halve cyclus gewacht worden, d.w.z. gemiddeld totaal één cyclus. Bij de meeste opdrachten is de duur van de punten 2 en 4 zó kort met betrekking tot de cyclus, dat het uitvoeren van de opdracht dan gemiddeld ruim één cyclus kost, waarvan het grootste gedeelte van de tijd besteed is aan wachten op het geheugen. Veelal kan nu deze wachttijd sterk worden gereduceerd, wat dus een geweldige snelheidswinst ten gevolge heeft.

Allereerst is de wachttijd verbonden aan punt 1 te reduceren op twee volkomen verschillende wijzen. De eerste manier is het invoeren van een extra adres in de opdracht, hetwelk vertelt waar de volgende opdracht te vinden is. Iedere opdracht is dan een sprongopdracht. Eventueel behoeft dit extra adres slechts een correctie op het heersende adres te zijn, zodat slechts enkele cijfers ervoor ter beschikking behoeven te worden gesteld.

Het voordeel ervan is, dat de programmeur na kan gaan op welke tijd een bepaalde opdracht klaar is en nu met behulp van dit extra adres ervoor kan zorgen, dat de volgende opdracht nu direct of bijna direct bereikbaar is. Wil men deze extra moeite niet nemen, dan vult men op het tweede adres altijd +1 in, d.w.z. de volgende opdracht staat op het volgende adres. Neemt men echter de moeite van dit zg. optimum programmeren, dan kan een aanzienlijke snelheidswinst worden behaald.

De tweede manier om het wachten op het halen van de opdracht te reduceren, is niet slechts één opdracht, maar meerdere opdrachten te halen en de nog niet aan bod zijnde te bewaren in een buffergeheugen met veel kleinere wachttijd. Bij de ARMAC wordt dit systeem toegepast. In de buffer bevindt zich altijd een geheel kanaal, dus mogelijk 64 opdrachten. Pas wanneer van kanaal moet worden gewisseld, treedt een verlies op, nl. van 35 getaltijden ofwel ruim een trommelomwenteling om het nieuwe kanaal in de buffer in te lezen. De programmeur behoeft hier geen moeite te doen om de snelheidswinst te boeken.

Om de wachttijd op getallen, dus te tijd benodigd voor punt 3 te beperken is wel noodzakelijk over een eventueel zeer bescheiden snel geheugen te beschikken. Bij de ARMAC is dit geheugen ter beschikking, enerzijds weer in de buffer, anderzijds in het snelle kanaal XO. De inhoud van de buffer wordt ingelegen bij kanaalwisseling, die van XO niet, maar hierin kan met behulp van opdrachten met f = 20 gemakkelijk tamelijk grote hoeveelheden woorden uit het trommelgeheugen overvoeren (zg. kanaaltransporten en kwartkanaaltransporten), terwijl door middel van opdrachten met f = 21 deze transporten in de andere richting snel plaatsvinden.

De aanwezigheid van buffer en snel kanaal bij de ARMAC heeft tot gevolg, dat men zich meestal geen zorgen behoeft te maken over snelheid of over verstandige plaatsing van getallen op de trommel. Slechts wanneer men zo veel getallenmateriaal moet verwerken (matrices!), dat de beperktheid van het snel geheugen hinderlijk gaat worden, kan het verstandig zijn er enige aandacht aan te schenken. Matrixprogramma's om matrices van orde $\leqslant 32$ achten doorgaans het element a_{ij} geplaatst op iAj of jAi,

waarin OAO een kanaalbegin, dus een 32-voud is. De programma's kunnen dan een kolom of rij van de matrix geheel of gedeeltelijk in XO overvoeren met één enkel kanaaltransport of enkele kwartkanaaltransporten, waarna dan verder de elementen van die rij of kolom zonder wachttijden beschikbaar zijn.

Moeten elementen van een lange rij op de trommel aan een zelfde proces worden onderworpen, dan is enig tijdverlies in het algemeen niet te vermijden. Is de bewerking afgelopen, dan is nl het nu benodigde element nog niet aanwezig. Dit maakt, dat men bij vele matrixprocessen rekening moet houden met een tijdsduur gelijk aan N omwentelingen als N het aantal behandelde elementen is, ook als de behandeling per element veel korter duurt dan een omwenteling. Een kleine winst kan men boeken door de elementen van achteren af te bewerken. Dan kost iedere bewerking 31 getaltijden in plaats van 33.

Het belangrijkste is, dat men ervoor wacht geen bijzonder ongunstige situaties te scheppen. Moet men berekenen $\sum a_i b_i$, waarbij de rijen a_i en b_i beide op de trommel staan,dan is het heel dom om de adressen $)a_i($ = iAO en $)b_i($ = iBO modulo 32 congruent te kiezen, m.a.w. voor OAO en OBO beide een kanaal te kiezen. Immers voor de vorming van het product moeten a_i en b_i van de trommel gehaald worden en wanneer hun adressen met dezelfde getaltijd corresponderen kost dat halen ons onherroepelijk een extra omwenteling.

Men kan dit inconveniënt gemakkelijk verhelpen door de ene rij een weinig te verschuiven, dus b.v. door voor oAo een kanaalbegin te kiezen en voor OBO een kanaalbegin +2 te kiezen. De opdrachten

10 iAO
18 iBO

kosten dan geen onnodige wachttijd. Men lette er goed op, dat bij de ARMAC minstens 1 getaltijd verloopt tussen het aanvragen door een opdracht van een getal op de trommel en het ter beschikking komen van dat getal. Als iAO dus gehaald is, is het eerste te halen getal er een dat twee getaltijden later aan bod komt, dus iBO mits OBO twee getaltijden na OAO komt. Op zichzelf is die extra getaltijd bij aanvrage van een getal op de trommel geen ernstig verlies, omdat getallen slechts zelden van de trommel worden gehaald, maar bij voorkeur in XO of in de buffer geborgen

worden. Heeft men echter er mee te maken, dan moet men er heel goed op letten om niet onnodig tijdverlies te laten ontstaan.

Een voorbeeld van een echte berekening, waarin op tijd is gelet is de volgende berekening van autocorrelatie- en mutuele correlatiefuncties. Op iAO resp. iBO zijn gegeven rijen van betrekkelijk kleine gehele getallen a_i resp. b_i, $i = 0(1)N-1$. Gevraagd wordt te berekenen

$$\Sigma_{aa} = \Sigma a_i a_{i+h}, \quad \Sigma_{bb} = \quad b_i b_{i+h} ,$$

$$\Sigma_{ab} = \Sigma a_i b_{i+h}, \quad \Sigma_{ba} = \quad b_i a_{i+h} ,$$

waarin gesommeerd is over $i = 0(1)N-h-1$ en waarin $h = 0(1)m$. Zowel N als m kunnen heel groot zijn (b.v. $N = 800$, $m = 100$), zodat het benodigde aantal operaties enorm is en een beetje economie vereist is. Op pag. 72 en 73 staat een programma, dat uitgesneden is uit een werkelijk ARMAC-programma. Op kanaal PO bevinden zich een aantal gegevens, waarvan voor ons doel alleen die op 28(1) 31 PO van belang zijn. Het programma vangt aan op aOPO en spreekt voor zichzelf.

Maatgevend voor de tijd benodigd voor het programma is de cyclus tussen a 5 DO en a 20 DO. Als de machine nooit op het geheugen behoefde te wachten waren voor deze cylcus 79 getaltijden nodig, te weten

31 opdrachten = 31 getaltijden,
waaronder 4 vermenigvuldigingen = 48 getaltijden extra.

Door te kiezen OBO = 2 A 30 zorgen we ervoor, dat de wachttijden bij de opdrachten b 5 DO en b 7 DO tot het minimum van een getaltijd worden gereduceerd. Waar we niets aan kunnen doen is het feit, dat in het algemeen moet worden gezocht bij a 5 DO en a 7 DO. Als men alles precies narekent zal men zien, dat voor $h \equiv 6,7,\ldots,20 \pmod{32}$ de cyclus 3 omwentelingen plus 1 getaltijd dus 97 getaltijden kost, terwijl voor $h \equiv -11,-10,\ldots,5 \pmod{32}$ de cyclus 1 omwenteling langer, dus 129 getaltijden in beslag neemt. Het nuttig effect is dan
$32 \times 79 \div (15 \times 97 + 17 \times 129) = 69\%$. Men ziet duidelijk het desastreuze gevolg van het wachten, ook zelfs nu we moeite hebben gedaan om het effect te verminderen. Hadden we gekozen OBO = OA 30 dan was het nuttig effect 45% geweest.

	RD RA	0	D	0	
⇨ 0	20	0	P	0	(P0) ⇛ (X0)
	26	0	X	4	
1	4	14	X	0	} 0 ⇛ h
	4	15	X	0	
b26D0 → 2	4	10	X	0	$0 \Rrightarrow \Sigma_{aa}$
	4	11	X	0	$0 \Rrightarrow \Sigma_{bb}$
3	4	12	X	0	$0 \Rrightarrow \Sigma_{ba}$
	4	13	X	0	$0 \Rrightarrow \Sigma_{ba}$
4	2	29	X	0	
	7	17	D	0	⇛
a20D0 → (5		RX1			2iA0
					) 10iB0
6	4	6	X	0	$a_i \Rrightarrow$ (6X0)
	12	7	X	0	$b_i \Rrightarrow$ (7X0)
(7		RX1			2 2i + h A0
					) 10i + h B0
8	4	8	X	0	$a_{i+h} \Rrightarrow$ (8X0)
	12	9	X	0	$b_{i+h} \Rrightarrow$ (9X0)
9	18	7	X	0	
	8	11	X	0	
10	12	11	X	0	$\Sigma_{bb} + b_i b_{i+h} \Rrightarrow \Sigma_{bb}$
	10	6	X	0	
11	18	8	X	0	
	8	10	X	0	
12	12	10	X	0	$\Sigma_{aa} + a_i a_{i+h} \Rrightarrow \Sigma_{aa}$
	10	6	X	0	
13	18	9	X	0	
	8	12	X	0	
14	12	12	X	0	$\Sigma_{ab} + a_i b_{i+h} \Rrightarrow \Sigma_{ab}$
	10	7	X	0	
15	18	8	X	0	
	8	13	X	0	

		RD RA	O	D	O		
	16	12	13	X	0		$\Sigma_{ba} + b_i a_{i+h} = \Sigma_{ba}$
		2	5	D	0		
	17	0	31	X	0		
b4DO →		28	5	X	2		$\left.\begin{matrix} i+1 \\ 0 \end{matrix}\right\} \Rightarrow i$
	18	0	14	X	0		
		28	7	X	2		
	19	1	30	X	0		
		28	0	X	0		$(A) \neq 0?$
	20	14	5	D	0	→	
		....					Schematisch voor programma
	21	....					a, dat iets doet met de
		....					berekende sommen.
	22	2	31	X	0		
		0	14	X	0		
	23	4	14	X	0		
		26	1	X	4		
	24	0	15	X	0		
		4	15	X	0		
	25	1	28	X	0		
		29	34	X	20		$A \leqslant -0?$
	26	26	0	X	4		
		14	2	D	0	→	
	27	....					Programma als alle waarden
		....					van h zijn afgewerkt.
		RA	28	P	0		
		RG					
	28	+	m				
		RD					
	29	2	0	A	0		
		10	0	B	0		
	30	2	N	A	0		
		10	N	B	0		
	31	0	1	X	0		
		0	1	X	0		

Een verbetering kunnen we nog wel verkrijgen door twee waarden van h, welke 2 verschillen tegelijk te behandelen. Op deze wijze komt men tot een rendement van 80%. In werkelijkheid is de winst nog groter. Immers getaltijden zijn er dan 151. Dit is minder dan 2 × 79 = 158, omdat de opdrachten op 5 DO en 6 DO niet herhaald behoeven te worden terwijl ook de opdrachten van b 16 DO tot en met a 20 DO nauwelijks uitgebreid behoeven te worden. Het nuttig effect van 80% is berekend op deze 151 nuttige getaltijden, maar het was even eerlijk geweest 158 nuttige opdrachten te rekenen en in dat geval had het nuttig effect 87% bedragen. Effectief het beste criterium is het aantal getaltijden gemiddeld per term gespendeerd. Dit bedraagt in het uitgewerkte geval met 0 BO = 0 A 30: $43\frac{1}{2}$, met 0 BO = 2A 30: $28\frac{1}{2}$ en in het geval dat h en h+2 gezamenlijk behandeld worden en tevens 0 B 0= 2 A 30: $22\frac{3}{4}$.

Een gedeelte van de winst die in het laatste voorbeeld is geboekt is te danken aan het feit, dat we administratieve functies als ophogen van opdrachten, controleren of we misschien aan het einde van een berekening zijn, enz. hebben laten drukken op een grotere hoeveelheid rekenwerk dan in het uitgewerkte voorbeeld. Trouwens daar was ook al gebruik gemaakt van het zg. "uitschrijven" van een programma. Immers een veel beknopter programma had de beoogde functies ook kunnen verrichten. Doelbewust is het programma niet tot een minimum opgevouwen, omdat dit een enorme tijdsverspilling met zich mee zou brengen. Vrij algemeen geldt, dat men snelheid kan krijgen ten koste van programmaruimte: "Een lang programma duurt kort, een kort programma duurt lang". Natuurlijk zijn hieraan beperkingen. Als een programma door een beetje opvouwen binnen een kanaal gehouden kan worden, betekent dit een winst, omdat geen buffer behoeft te worden ingelezen. Daarom zij men met het uitschrijven van programma's zuinig en beperke dit tot de meest intensief gebruikte gedeelten van h t programma.

11. Schaling, controle en flexibiliteit

Bij de beschrijving der arithmetische operaties hebben wij ons consequent beperkt tot operaties op hetzij gehele getallen, hetzij echte breuken.

Er zijn problemen, waarbij deze twee soorten getallen toereikend zijn, b.v. als alle getallen in een berekening of een aantal of een waarschijnlijkheid voorstellen.

In menige berekening komen in eerste instantie natuurlijk ook gemengde getallen voor, d.w.z. getallen met een aantal cijfers voor, zowel als achter de komma. Een eenvoudig voorbeeld is b.v. een (positieve) hoek in één van de vier quadranten. In radialen uitgedrukt loopt de hoekmaat op tot 2π, in graden tot 360; in eenheden van 2π radiaal uitgedrukt wordt hij door een maatgetal $x < 1$ gekarakteriseerd. In de ARMAC-precisie komt dit overeen met een nauwkeurigheid van 0,007 sec., wat voor de meeste toepassingen ruimschoots voldoende is. Is het toereikend als de hoeken worden gerepresenteerd afgerond op het dichtsbij gelegen gehele aantal seconden, dan kan men de hoek met behulp van dit aantal vastleggen: dit gehele getal is $< 360.60.60 = 1296000$, wat de capaciteit van een woord nog lang niet overschrijdt.

Kortom in dergelijke eenvoudige gevallen zijn er legio mogelijkheden om door keuze van een geschikte eenheid, of, wat op hetzelfde neerkomt, door het vermenigvuldigen met een geschikt gekozen constante - de variabele enerzijds voldoende nauwkeurig voor te stellen, zonder dat anderzijds bij grote waarden de capaciteit van een woord overschreden wordt.

Het vermenigvuldigen van constantes en variabelen met geschikt gekozen factoren, opdat zij daardoor een hanteerbare orde van grootte krijgen, heet "het schalen van een berekening"; de ingelaste factoren heten schaalfactoren. Als deze schaalfactoren machten van tien zijn, zeggen wij, dat het probleem "decimaal geschaald" is.

In een binaire machine heeft het echter vaak voordelen te schalen met machten van twee. Als b.v. de getallen x en y variëren van 1 tot 2000, dan kunnen we, om er echte breuken van te maken ze b.v. delen door 2048 ($= 2^{11}$). Deze binaire schaling stelt de getallen iets nauwkeuriger voor dan de decimale schaling met een factor 10^{-4}.

Stel, dat het product x.y gevormd moet worden, maar dat bekend is, dat eveneens het product kleiner dan 2000 blijft. In dat geval zal men wellicht dit product eveneens vermenigvuldigd met een factor 2^{-11} willen berekenen. Dit betekent echter dat het door de machine gevormde product $(x.2^{-11}).(y.2^{-11})$ met 2^{+11} vermenigvuldigd moet worden, om het gewenste resultaat $(x.y.2^{-11})$ te vormen. Hadden we decimaal geschaald, dan had de schaling ons hier extra vermenigvuldigen gekost, terwijl we nu met een (snellere) schuifopdracht kunnen volstaan.

Algemeen kunnen we zeggen, dat het schalen met vaste schaalfactoren als in bovenstaande voorbeelden slechts mogelijk is als men een (liefst vrij nauwkeurig) idee heeft over de orde van grootte der variabelen. Het zal duidelijk zijn, dat bij gecompliceerde berekeningen het verkrijgen van dit inzicht - het vaststellen van de maximale absolute waarde - met aanzienlijke problemen gepaard kan gaan. Deze problemen kunnen onoverkomelijk zijn; het is ook heel gebruikelijk, dat men vreest of zelfs aan kan tonen, dat een probleem onmogelijk met een vaste schaling berekend kan worden. Als b.v. de verschillende voorkomende waarden van een bepaalde variabele een factor 10^8 kunnen verschillen en bij minimale absolute waarde de variabele nog voorgesteld moet kunnen worden met een relatieve precisie van minstens 10^{-4}, dan is het onmogelijk met een vaste schaalfactor deze variabele in een enkel woord van de ARMAC voor te stellen. In deze gevallen kan men overgaan tot een andere getalrepresen tatie, nl. in plaats van het getal x bergt men de breuk b en de macht m, bepaald door

$$x = b.2^m \text{ , waarbij } \tfrac{1}{2} \leq |b| < 1.$$

Bij een dergelijke notatie spreekt men van "een drijvende (binaire) komma". Het voordeel van de representatie met drijvende komma is dat met behoud van de relatieve precisie (bepaald door het aantal cijfers, dat voor de breuk b ter beschikking gesteld wordt) de variabelen een veel groter gebied kunnen bestrijken.

Er bestaan rekenmachines, waarvan het rekenorgaan direct op getallen in drijvende notatie opereren kan en het antwoord ook weer in deze vorm aflevert. Voor de ARMAC zijn deze arithmetische operaties geprogrammeerd: voor elke arithmetische bewerking op zg. drijvende getallen moet de ARMAC een stuk programma

uitvoeren, wat met een behoorlijke vertraging van de effectieve rekensnelheid gepaard gaat (vergelijk MR 27 en MR 27a).

Voorbeeld

Gevraagd $\frac{n}{1440} \Rightarrow \{7Xo\}$, voor n geheel en $n < 1440$.
We nemen aan dat [6 Xo] = n. We zouden dit kunnen doen met de extra deling

```
 2   6   X   0
26   0   X  12
 4   0   X   0
12   1   X   0
10   3   B   5   als b.v. [3 B 5] = 1440
22   0   X  30 =)
12   7   X   0
```

Als dit voor veel waarden van n berekend moet worden is dit tamelijk tijdrovend vanwege de langzame deling. Met behulp van een andere constante kan dit vlugger:

```
10   4   B   5   als b.v. {4 B 5} = 2^10/1440
18   6   X   0
26  10   X  28
12   7   X   0
```

Waarom is de factor 2^{10} in de constante bijgevoegd?

Terloops willen wij nog wijzen op de mogelijkheid "gedeeltelijk drijvend" te werken. Uit tijdsoverwegingen is het nl. gewenst zo min mogelijk drijvend te werken. Gelukkig doet zich bij moeilijk schaalbare berekeningen vaak de omstandigheid voor, dat de moeilijkheden zich tot een heel klein stukje van de berekening beperken. Dan loont het vaak de moeite vlak van te voren voor de betroffen getallen over te gaan op de drijvende notatie en deze weer ijlings te verlaten, zodra het gevaar geweken is. Een andere vorm van "gedeeltelijk drijvend" werken bestaat daaruit, dat men niet elk getal afzonderlijk met een individuele tweemacht schaalt, maar dat de variabelen groepsgewijs met een aangepaste tweemacht vermenigvuldigd worden. Per groep hoeft het programma dan maar met één macht m te manipuleren; aan de ongelijkheid $\frac{1}{2} \leq |b|$ is dan in de regel voor de individuele getallen niet meer voldaan.
Deze techniek is b.v. met vrucht toegepast bij de oplossing van lineaire stelsels (dank zij de homogene afhankelijkheid van de coëfficiënten!). De resulterende programma's behoren niet tot de eenvoudigste. Omdat schalingsmoeilijkheden over het algemeen te nauw verbonden zijn met het specifieke probleem, dat men onder

handen heeft, zullen wij echter nu niet nader op hierop ingaan. Wij volstaan tot troost met de opmerking dat de woordlengte van de ARMAC (equivalent aan bijna 10 decimalen!) vaak niet zozeer gebruikt wordt om ijselijk precies te kunnen rekenen, dan wel om met wat meer gemak schalingsmoeilijkheden op te kunnen vangen.

Onder controle vallen alle maatregelen, die de programmeur treft om er zo goed als zeker van te zijn, dat de resultaten, die de machine aflevert overeenkomen met de bedoelde resultaten.

De noodzaak van dergelijke maatregelen verschilt van machine tot machine: een apparaat dat een hoeveelheid cijfers produceert, maar zonder enige garantie, dat deze cijfers inderdaad iets voorstellen is een vrij onnut apparaat. Immers de waarde der antwoorden hangt af van de kracht der garantie, waarmee ze verstrekt worden!

Men kan programma's maken, die incidentele reken- of transportfouten van de machine detecteren. Een stukje berekening steeds tweemaal uitvoeren en de machine de twee resultaten laten vergelijken, moet onvoldoende geacht worden: de kans op compenserende fouten (in dit geval binnen b.v. 1 sec. tweemaal precies dezelfde fout) is hier niet verwaarloosbaar klein. Geraffineerdere technieken met een maximum aan zekerheid voor een minimum aan extra rekentijd zijn ontwikkeld in elk instituut, dat voor de opgave gestaan heeft met een onbetrouwbare machine betrouwbare resultaten af te leveren. Wij zullen hier niet nader op in gaan, omdat de ervaring ons geleerd heeft, dat deze technieken bij de ARMAC als overbodig beschouwd mogen worden. Dit is te danken aan de zg. ingebouwde paritycheck: aan elk half woord in het geheugen, dus aan elk zeventiental cijfers, wordt bij het schrijven in het geheugen een achttiende binaal toegevoegd, zo, dat het totale aantal énen in een dergelijk achttiental oneven is (eigenlijk dus: "imparity check"). Altijd als deze informatie gelezen wordt, wordt in elk achttiental het oneven zijn van het aantal énen gecontroleerd: is hieraan niet voldaan, dan stopt de machine. Deze controle wordt bij werkende machine één à twee duizend malen per seconde aangelegd; behalve dat deze controle ons vrijwaart voor geheugenfouten, heeft deze controle een belangrijke indicatieve functie: als de paritycheck geen alarm slaat, weet men, dat de algemene conditie van de machine goed is.

Weten we bovendien, dat het geheugen het meest critische onderdeel van de ARMAC is, dan begrijpen we, dat in geval van b.v. netstoring de ARMAC op de paritycheck stopt, voordat hij de kans gehad heeft een rekenfout te maken. Het rekenorgaan zelf is niet gecontroleerd. Fouten hierin hebben echter meer de neiging permanent te zijn; dit onderdeel wordt daarenboven zo intensief gebruikt, dat een fout van het rekenorgaan niet lang onopgemerkt pleegt te blijven.

In tegenstelling tot het verleden is nu het stadium bereikt, dat de zwakste schakel in het proces niet meer de machine, maar ----- de programmeur is! En de programmeur doet er goed aan met behulp van het programma zichzelf te controleren.

Enige voorbeelden mogen dit illustreren.
In een geregeld voorkomende standaardberekening, die geregeld op de ARMAC uitgevoerd wordt, kan op een bepaald punt capaciteitsoverschrijding optreden, afhankelijk van het getallenmateriaal, waarvan men uit gaat. Of dit op zal treden, is niet aan deze getallen "te zien"; wel kunnen we inzien, dat de kans op deze theoretisch mogelijke capaciteitsoverschrijding heel klein is. Het programma nu test, of deze capaciteitsoverschrijding optreedt, zo ja, dan stopt de machine. Hoewel de berekening inmiddels honderden malen, en met het meest uiteenlopende getallenmateriaal is uitgevoerd, is de machine nog nooit hierom gestopt. Is daarom tot nog toe deze controle overbodig geweest? Nee: wij zouden weliswaar dezelfde resultaten aan onze opdrachtgevers afgeleverd hebben, maar met dat verschil - een groot verschil! - dat wij ons aanmerkelijk minder prettig gevoeld zouden hebben.

Zo fungeert de geprogrammeerde controle als tegemoetkoming aan onze onmacht - soms ook luiheid! - om van te voren vast te stellen, of alles goed hoort te gaan.

Een tweede functie van geprogrammeerde controle's komt tegemoet aan ons gebrek aan accuratesse. Zo controleert het invoerprogramma tijdens het bandlezen in menig opzicht, of de band wel volgens de regels van de kunst geponst is: wie zondigt tegen de zg. "ponsconventies" (zie MR 25) zal meestal ervaren, dat bij het inlezen de machine stopt. Het mag voor de beginnende programmeur ontmoedigend zijn, dat het hem meestal niet eens lukt, zijn band ingelezen te krijgen, wij betreuren alleen maar, dat het invoerprogramma niet nog veel meer controleert.

Vele subroutines controleren waar mogelijk, of ze correct gebruikt worden. De subroutine Se2 voor de 2-macht accepteert, omdat het antwoord kleiner dan 1 moet zijn, alleen negatieve argumenten; de subroutine S12 voor de $_2$logarithme accepteert alleen positieve argumenten; wie deze routines met een argument van het verkeerde teken aanroept, wordt met stoppen gestraft (zie MR 27). Waar de programmeur dergelijke controles in kan lassen, zoals het in deze subroutines onder het motto "Kost zo weinig, geeft zo veel" gedaan is, raden wij hem aan, dit niet na te laten. Het zal hem bij het testen van zijn programma goede diensten bewijzen. Als het programma het test-geval correct heeft doorgerekend, maar na een paar dagen, bij heel andere waarden der variabelen, stopt de machine toch nog op een dergelijke controle, dan zal de programmeur blij zijn, zo "absurd voorzichtig" geweest te zijn.

Wat betreft de geboden flexibiliteit van programma's wijzen wij ten eerste op de herstartbaarheid. Hiermede worden (zie MR 25a) de eisen samengevat, waaraan een programma moet voldoen, om het zo makkelijk mogelijk te maken opnieuw bij het begin - tijden testen van het programma - of een eindje terug - bij een machinefout - te beginnen. Deze eisen zijn dus een consequentie van de onvolkomenheid van programmeur resp. machine.

De eis der flexibiliteit ligt echter veel ingewikkelder. Het in detail behandelde voorbeeld der splitsing als som van twee quadraten is een idealisatie: we wisten van begin af aan het juiste formularium èn wat we als antwoord wensten. Aan beide condities is in de practijk zelden voldaan: als resultaat van proefberekeningen is het niet ongebruikelijk, dat dan blijkt, dat men foutieve, dan wel onvolledige formules heeft gekregen. Als deze fouten achterhaald zijn - wat wijzigingen in het programma impliceert - gebeurt het vaak, dat de opdrachtgever aan de hand van de uitgetypte resultaten (of soms ook op eigen gelegenheid) tot de ontdekking komt, dat hij dat en dat ook zo graag uitgetypt zag. Als het aanvankelijke programma zo flexibel is, dat de gewenste wijzigingen kunnen worden aangebracht, zonder het hele programma om te gooien, bespaart men zich veel werk en ergernis.

Nog stringenter is de eis der flexibiliteit bij programma's van een zekere algemeenheid: zodra er de mogelijkheid bestaat, dat dit programma nog eens voor een ander gebruikt zal worden, moet men zo voorzichtig mogelijk te werk gaan.

Enige voorbeelden mogen dit toe lichten:
Een programma, dat een variabel aantal getallen verwerken moet, moet het laatste getal als zodanig kunnen identificeren. Eén van de methodes is het aantal mee te geven; een andere methode is om het einde met een label aan te geven: als alle getallen positief zijn, kan men het einde indiceren door de rij met een negatief getal af te sluiten. Dit is programma-technisch vaak heel aantrekkelijk: als het programma de getallen in volgorde afwerkt, onderzoekt het elk volgend getal eerst op zijn teken. Ook al zijn in een betroffen probleem de getallen allemaal positief, toch mag men van deze label-techniek in een algemeen programma alleen maar gebruik maken, als de getallen essentieel positief zijn! Doet men het toch, dan zal men ervaren, dat bij de eerstvolgende keer, dat er belangstelling voor dit programma is, getallen van wisselend teken verwerkt moeten worden.

Illustratief zijn in dit verband de mee- en tegenvallers bij ons eigenwaarden-programma. Hoewel het aantrekkelijk was, slechts matrices van een orde hoogstens = 32 te verwerken, zijn we in een lucide moment voor deze verleiding niet gezwicht: de eerste matrix, die ons werd aangeboden, was van de orde 33. Alle eigenwaarden waren positief, en het programma was ook alleen geschikt voor het bepalen van positieve eigenwaarden: de tweede matrix, die verwerkt moest worden ... had eigenwaarden van beiderlei teken! De eigenvectoren worden door iteratie bepaald en als regel schijnt er a priori van de eigenvectoren weinig bekend. Na een jaar kregen wij een stel matrices te verwerken waarbij voor de eigenvectoren heel fatsoenlijke schattingen bekend waren. Gelukkig was het programma inmiddels zo flexibel uit subroutines opgebouwd, dat het slechts een kleine toevoeging betekende om het programma voor de volgende eigenvector een schatting van de band te laten lezen.

Kortom: de moeilijkheid is dat veelal de wens naar voren zal komen een bestaand programma te gebruiken op een manier, die niet precies bedoeld was, voor een probleem, waarvoor het programma niet precies gemaakt was. De programmeur maakt het programma, d.w.z. een preciese formulering van het rekenschema: aan hem wordt overgelaten, rekening te houden met allerlei mogelijke nog niet geformuleerde eisen.

12. De administratieve subroutine I

De tot nu toe behandëlde subroutines dienden meestal ter berekening van speciale functies als cos x, log x of om een andere operatie op een getal toe te passen, dus bijv. typ x op een of andere manier. Het getal x werd in S meegegeven aan de subroutine. Het getal x werd in S meegegeven aan de subroutine. Het hoofdprogramma roept de subroutine aan, deze doet haar werk zonder andere subroutines nodig te hebben en keert terug.

Nu komen wij tot subroutines welker taak coördinerend is. Zij roepen op hun beurt weer andere subroutines aan, of zelfs delen van het hoofdprogramma. De algemeenste routines van deze klasse zijn zelfs superieur aan het eigenlijke hoofdprogramma; het hoofdprogramma wordt dan gedegradeerd tot een stel subroutines ten dienste van de administratieve subroutines. De reden, waarom deze dan nog als subroutine zijn uitgevoerd is om nog algemenere coördinerende routines er mee te laten werken. Uiteindelijk is het ideaal een "hoofdprogramma" als volgt in ARMAC-taal:

```
22        FO =) Ga naar hoogstgeordende administratieve
                                          subroutine
29 4096 XO      Stop.
```

De allermachtigste van deze routines hebben derhalve slechts één aanroep. De minder machtige daarentegen zijn met het oog op de flexibiliteit meestal gekenmerkt door een groot aantal aanroepmogelijkheden.

We zullen enige voorbeelden geven, te beginnen met enkele van matige orde, zodat het hoofdprogramma er op verschillende manieren mee kan spelen.

Type argument, functie, differentie

Wij nemen aan, dat het hoofdprogramma x en f(x) voor opvolgende waarden van x berekent en x, f(x) en $\nabla f(x)$ uitgetypt moeten worden. Daarbij moet bedacht worden, dat op de eerste regel natuurlijk $\nabla f(x)$ ontbreken moet. Het hoofdprogramma moet voorts de mogelijkheid hebben om via de subroutine een regel nog eens over te typen, b.v. bovenaan een nieuwe pagina. Tevens is er voor gezorgd, dat er ingangen zijn, waarbij het typen van $\nabla f(x)$ of van x wordt onderdrukt. Als het typen van $\nabla f(x)$ wordt

onderdrukt, wordt er altijd voor in de plaats een door de regelindelingsubroutine van het communicatieprogramma getelde tabulatie gegeven. De aanroepen zijn:

Normaal: 22 0 FO =) Typt x, f(x), bewaart f(x), berekent, bewaart en typt $\nabla f(x)$.

Herhaal: 22 8 FO =) Typt x, f(x); $\nabla f(x)$ zoals bewaard, laat alles ongerept.

Aanloop: 22 10 FO =) Te geven voor de rekencyclus van het hoofdprogramma. Onderdrukt eenmaal het typen van $\nabla f(x)$.

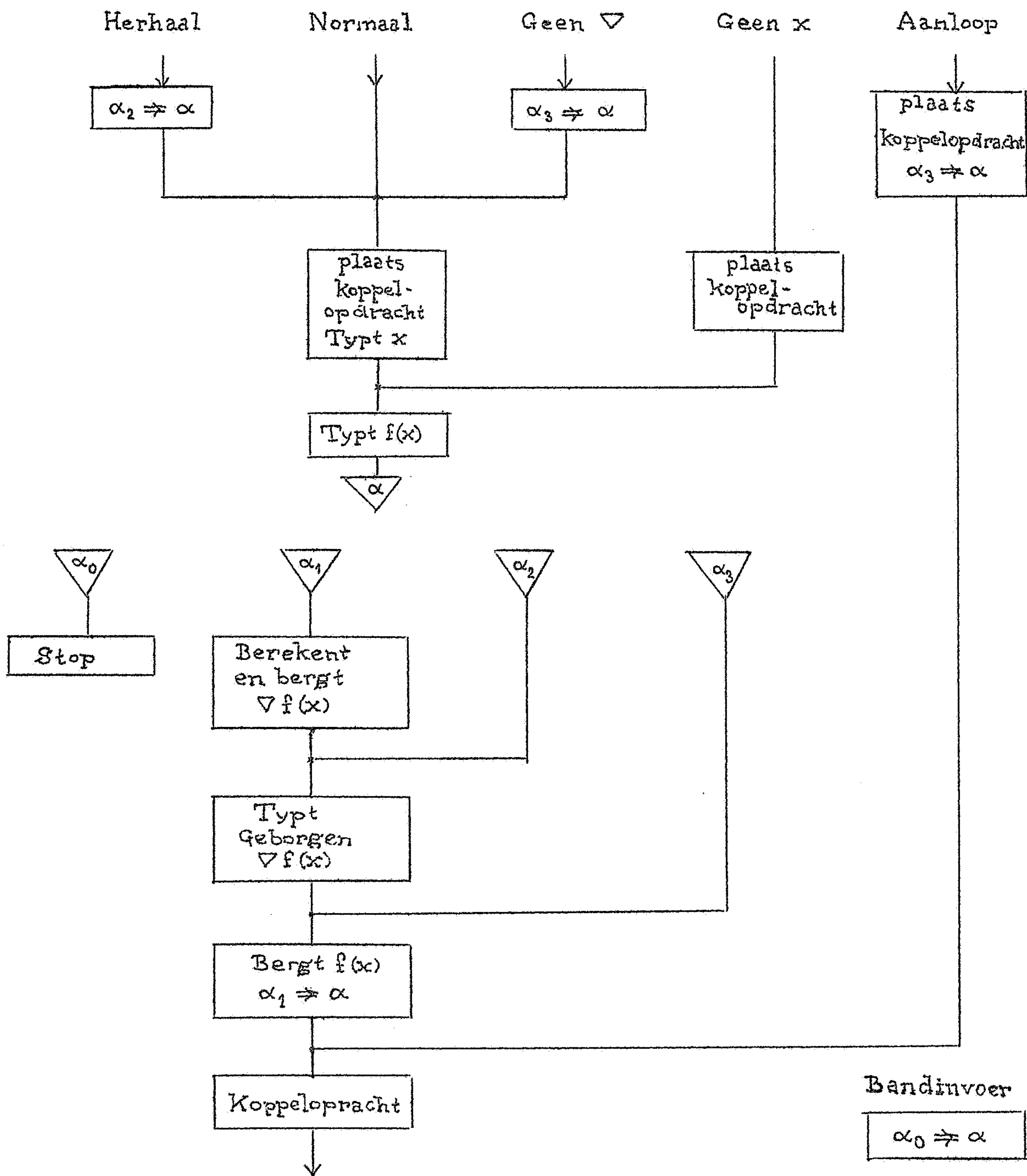
Als in plaats van 22 0 FO gebruikt wordt:

22 9 FO, dan wordt x niet getypt,
23 10 FO, dan wordt $\nabla f(x)$ niet getypt.

De typconstanten specificeren op welke wijze men x, f(x) en $\nabla f(x)$ wenst te typen, dus b.v. als geheel getal of als breuk en zonder of met teken. De adressen, waar x en f(x) te vinden zijn, zijn per definitie 6 XO en 7 XO.

Het programma, dat hieronder volgt, bevat enkele controle-elementen, alsmede een merkwaardige programmawisseling, die niet eenvoudig door een blokschema is te beschrijven. Let voorts speciaal op het geval "Aanloop"!

	RD RA	0	R	0		
Normaal ⇒ 0	4	7	R	0	plant link (16	) f(x)
a_{11}	10	6	X	0		
1	22	6	X	28	⇒ typ x (17	) ∇f(x).
b →	10	7	x	0		
(2	22	7	X	28	⇒ typ f(x) α_0	
	29	4096	X	0	) stop	
3	0	7	X	0		
	4	17	R	0	berg ∇f(x)	
b2 → 4	10	17	R	0		
	22	8	X	28	⇒ typ ∇f(x)	
b12 → 5	2	7	X	0		
	4	16	R	0	berg f(x)	
6	2	13	R	0		
	4	2	R	0	$\alpha_1 \Rightarrow \alpha$	
(7	R	X	1			
					)	
Herhaal ⇒ 8	10	15	R	0		
	6	11	R	0	⇒	
Geen x ⇒ 9	4	7	R	0	plant link	
	7	1	R	0	⇒	
Aanloop ⇒ 10	28	0	X	2	plant link!	
Geen ∇ ⇒	10	14	R	0		
b8 → 11	12	2	R	0		
	6	0	R	0	⇒	
b2 ⇒ 12	22	6	X	26	⇒ "acht getal geteld getypt"	
	6	5	R	0	⇒	
13	22	7	X	28	α_1	
	3	16	R	0		
14	22	7	X	28	α_3	
	6	12	R	0		
15	22	7	X	28	α_2	
	6	4	R	0		
	RC					


Herhaal
Normaal
Geen ▽
Geen x
Aanloop
$\alpha_2 \Rightarrow \alpha$
$\alpha_3 \Rightarrow \alpha$
plaats Koppelopdracht $\alpha_3 \Rightarrow \alpha$
plaats koppel-opdracht Typt x
plaats koppel-opdracht
Typt f(x)
α
α_0
α_1
α_2
α_3
Stop
Berekent en bergt ▽f(x)
Typt Geborgen ▽f(x)
Bergt f(x) $\alpha_1 \Rightarrow \alpha$
Koppelopracht
Bandinvoer
$\alpha_0 \Rightarrow \alpha$

13. De administratieve subroutine II

Integratie

In wetenschappelijke berekeningen treedt vaak een integraal op, dus een uitdrukking van de vorm

$$\int_a^b f(x)dx .$$

Als gegeven is, hoe f(x) uit x volgt en wat a en b zijn, staat hier een volledig bepaald getal. Natuurlijk kan men geen routine maken, die alle integralen uitrekent zonder f(x) te definiëren. We zullen derhalve aannemen, dat er een subroutine bestaat, die met (S) = x aangeroepen terugkeert met (S) = f(x). Geven we dan nog a en b op een of andere wijze, dan moet het berekenen van de integraal op een of andere wijze uniform kunnen geschieden en daarvoor kan een subroutine gemaakt worden, zodat de programmeur zich daar verder geen zorgen meer over behoeft te maken, maar het integraalbegrip kan inlijven bij zijn standaardroutines voor de machine.

Voor we de werking van deze subroutine analyseren, moeten we ons echter eerst realiseren, dat, willen we onze f(x) niet beperken tot een behalve voor het candidaatsexamen oninteressante kleine groep van functies, die in gesloten vorm geïntegreerd kunnen worden, terwijl anderzijds het niet de bedoeling is om andere dan Riemannse integralen te beschouwen, het enerzijds nodig maar anderzijds ook voldoende is om de numerieke analyse te hulp te roepen. Dit betekent, dat we niet kunnen verwachten het preciese antwoord voor de integraal te vinden, d.w.z. de op een voorgeschreven aantal decimalen (of binalen) afgeronde waarde van het exacte resultaat, maar dat we alleen kunnen eisen, dat althans zeer waarschijnlijk het gevonden antwoord van het juiste antwoord niet meer verschilt dan een opgegeven tolerantie. Deze tolerantie zal dus ook worden gespecificeerd, maar het wordt tot de taak van de subroutine geacht ervoor zorg te dragen, dat deze tolerantie niet wordt overschreden. De subroutine mag dus niet meer informatie vragen, dan iedere geschoolde rekenaar zou doen, nl. de betekenis van het symbool f(x), de waarden van a en b en tenslotte de tolerantie van het antwoord.

Waarschijnlijk zal een dergelijke subroutine met het oog op haar algemeenheid niet voor iedere functie en onder alle omstandigheden het allersnelste werken, maar zij spaart onnoemelijk veel denktijd en programmeertijd en kan natuurlijk zorgvuldig en snel geprogrammeerd worden, zodat ze toch niet zo heel gemakkelijk overtroefd zal worden met een ad hoc programma.

Men moet dus van een of andere numerieke integratieformule gebruik maken en in het te behandelen voorbeeld is de formule van SIMPSON gebruikt. Integreert men van C tot C + 4h met stappen van 2h, h of 0 (d.w.z. exact), dan geldt voor deze deelintegraal:

$$\int_C^{C+4h} f(x)dx = 4h\{F(2h) + R(2h)\}$$

$$= 4h\{\tfrac{1}{6}f(C) + \tfrac{2}{3}f(C+2h) + \tfrac{1}{6}f(C+4h) + R(2h)\};$$

$$\int_C^{C+4h} f(x)dx = 4h\{F(h) + R(h)\}$$

$$= 4h\{\tfrac{1}{12}f(C) + \tfrac{1}{3}f(C+h) + \tfrac{1}{6}f(C+2h) + \tfrac{1}{3}f(C+3h) + \tfrac{1}{12}f(C+4h) + R(h)\};$$

$$\int_C^{C+4h} f(x)dx = 4h\,F(0) .$$

Wat we graag zouden weten is F(0), maar berekenbaar zijn alleen F(2h) en F(h). Evenwel weten wij, dat bij benadering voor de overigens onbekende resttermen geldt:

$$16\,R(h) \approx R(2h)$$

en wel des te nauwkeuriger naarmate h kleiner is. Een schatting van R(h) volgt dus uit:

$$F(h) - F(2h) = R(2h) - R(h) \approx 15R(h) .$$

De procedure is dus, dat men F(2h) en F(h) beide berekent, toeziet, dat het verschil klein is en zo dit het geval is hieruit een benaderde waarde van R(h) berekent. Afgezien van pathologische gevallen vindt men zo een schatting van F(0), welke aanzienlijk beter is dan het verschil F(h) - F(2h) zou doen vermoeden. Laten wij daarom een interval h slechts als

genoegzaam klein accepteren als

$$|F(h) - F(2h)| < \Delta_{max} ,$$

waarin Δ_{max} een gegeven klein positief getal is. Ongeacht, hoe h dan ook moge variëren over het totale integratiegebied van a tot b, is

$$|b - a| \Delta_{max}$$

een pessimistische schatting van de orde van grootte van de fout welke gemaakt wordt in de volledige integraal van a tot b. Hoe kleiner fout getolereerd wordt, des te kleiner moet Δ_{max} gekozen worden, des te kleiner h wordt getolereerd en des te zekerder is het, dat de werkelijke fout aanzienlijk kleiner zal zijn.

De gedachtengang van de subroutine is nu, dat zij zelf h tracht te kiezen door de bovenstaande ongelijkheid te testen. Is daaraan niet voldaan, dan verwerpt zij h, halveert h en programmeert het opnieuw. Alleen als $|h|$ gelijk mocht zijn aan 1 peuter, dan wordt noodgedwongen dit "grove" interval geaccepteerd. Is omgekeerd aan de gelijkheid wel voldaan, dan wordt h geaccepteerd en de bijdrage tot de integraal wordt alvast verdisconteerd. Evenwel wordt daarna getest of soms

$$|F(h) - F(2h)| < \Delta_{min} ,$$

waarin Δ_{min} een kleine fractie van Δ_{max} is. Is dit nl. het geval, dan betekent het, dat h wel groter had mogen zijn. De keuze van $\Delta_{min}/\Delta_{max}$ is een strategisch probleem. Kiest men Δ_{min} zeer klein t.o.v. Δ_{max}, dan wordt h onnodig lang klein gehouden. Kiest men Δ_{min} weinig kleiner dan Δ_{max}, dan loopt men grote kans dat de verdubbelde stap 2h direct weer wordt afgekeurd, wat eveneens onnodig rekenwerk tengevolge heeft. Omdat verdubbeling van h de restterm ongeveer met 16 vermenigvuldigt, is $\Delta_{max} = 16\ \Delta_{min}$ ongeveer de kritieke verhouding. Het ARMAC-programma kiest $\Delta_{max} = 20\ \Delta_{min}$.

Zodra een integratiestap van C tot C+4h is geaccepteerd, moet de gevonden F nog met 4h worden vermenigvuldigd, om de bijdrage tot de integraal te vinden. Deze vermenigvuldiging wordt in dubbele lengte uitgevoerd.

Ten eerste is er de kwestie, dat in de omgeving van de punten waar f(x) zich wild gedraagt, h zeer klein wordt. Een deel van de integraal kan dus worden opgebouwd uit een zeer groot aantal zeer kleine bijdragen. Om de precisie niet te laten lijden is dubbele lengte rekenen dus noodzakelijk.

De integraal wordt ook afgeleverdals breuk van dubbele lengte $I'+I''.2^{-33}$. De subroutine heeft verschillende aanroepen. Allereerst is er de aanroep "Precisie", die $\{S\} = \Delta_{max}$ bergt, hieruit Δ_{min} berekent en bergt. Vervolgens is er de aanroep "Ondergrens", die $\{S\}$ = a bergt, I (= I',I") schoonmaakt en vast $\frac{1}{12}$ f(a) uitrekent en bergt als $\frac{1}{12}f_0$. Aan iedere integratie moet natuurlijk één maal het zetten van Δ_{max} en a voorafgaan, waarbij de volgorde niet ter zake doet.

De subroutine is gereed om te gaan integreren. Wil men één enkele integraal, dan roept men haar aan met $\{S\}$ = b op de ingang "Bovengrens". De subroutine bergt b, kiest h = (b-a)/4 om te beginnen al gaat aan de slag en komt uiteindelijk terug als de integraal is berekend, die wordt afgeleverd op 2 XO en 3 XO.

De subroutine rekent zoals beschreven steeds met een zo gunstig mogelijke h en houdt deze waarde van h bij wisselingen van h ook bij in een apart adres "h-suggestie". Pas wanneer de integratie bijna afgelopen is, moet in het algemeen even een klein ad hoc interval h gebruikt worden om de bovengrens b precies te bereiken. Dit uitzonderingsinterval wordt niet genoteerd, zodat na afloop van de integratie nog steeds het locaal, d.w.z. in de buurt van b gunstige interval h als h-suggestie genoteerd staat. Wil men door integreren, dan roept men de routine weer met de b-aanroep aan met de nieuwe bovengrens b in S. Omdat geen a-aanroep voorafgegaan is deze keer, wordt de integraal niet nul gemaakt en de routine integreert gewoon door, waarbij ze nu echter de h-suggestie als beginwaarde van h gebruikt en met (b-a)/4. Dat dit logisch mogelijk is komt omdat, wat we nog niet hadden verteld, de a-aanroep ook iets doet met de h-suggestie, zij valt er nl. 0 voor in. De integreer-aanroep heeft tot gevolg, dat de h-suggestie geaccepteerd wordt, tenzij ze 0 is. Dit kan nl. nooit tijdens de integratie gebeurd zijn, want het kleinste interval is 1 peuter. Dus betekent

$h_s = 0$ dat een a-aanroep is voorafgegaan en dat de suggestie niets voorstelt, waarop de routine (b-a)/4 kiest (en noteert) als h-suggestie.

In het bijgaande blokschema is te bestuderen, wat allemaal precies gemanipuleerd moet worden om te zorgen, dat de bovengrens precies bereikt wordt en niet overschreden wordt (dan hoeft de integrand niet meer te bestaan!). Een deel van de complicaties komt hieruit voort, dat b best kleiner dan a kan zijn en men ook de integratierichting voortdurend mag wijzigen, wat speciaal voor iteratieve processen belangrijk is.

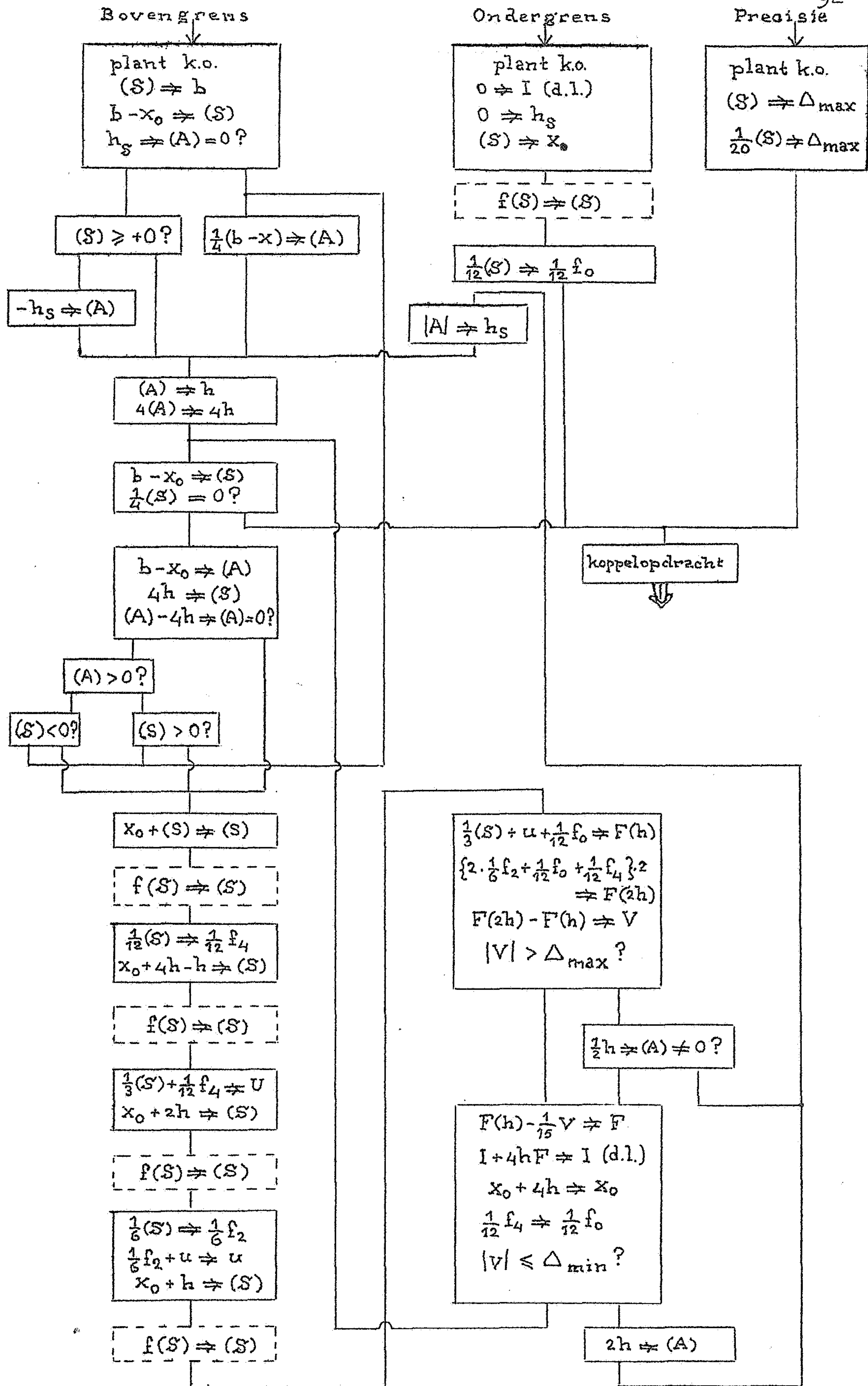
Tabellatie

Tenslotte geven wij als voorbeeld van een administratieve subroutine van in hoge mate overkoepelend karakter één, die tot taak heeft een tabel te maken van één of meer functies van één variabele. Het is er een, die volkomen in staat is te werken zonder echt hoofdprogramma en dus met het ideale hoofdprogramma bediend kan worden.

De opgave luidt precieser een functie f(x) te tabelleren voor x = a(h)b, met de restrictie, dat a en b exacte veelvouden zijn van h, wat practisch altijd het geval is. Overigens is door een kleine modificatie deze restrictie te vermijden. De functie f(x) mag een vector zijn, d.w.z. een aantal functies eventueel met differenties, enz. De pagina-indeling moet flexibel zijn maar aan hoge eisen voldoen. Daartoe is verondersteld, dat een aantal andere administratieve subroutines ter beschikking staan. Deze zullen wij eerst de revue laten passeren.

Subroutine "Functie"

Deze is in staat om bij gegeven x, die op een vaste plaats in "Tabellatie" te vinden is f(x) te berekenen. Eventuele differenties behoeven niet te worden berekend; daar wordt elders zorg voor gedragen. De subroutine kan anders moeten ageren dan normaal als zij voor de eerste keer moet rekenen, b.v. als zij lopende integralen met behulp van "Integratie" berekent. Daarom zullen wij de mogelijkheid open laten, dat zij een speciale ingang heeft voor dat geval. Wij noemen haar dan "Nulfunctie".


Bovengrens
plant k.o.
(S) ⇒ b
b - x_0 ⇒ (S)
h_S ⇒ (A) = 0 ?
(S) ≥ +0 ?
$\frac{1}{4}$(b - x) ⇒ (A)
- h_S ⇒ (A)
(A) ⇒ h
4(A) ⇒ 4h
b - x_0 ⇒ (S)
$\frac{1}{4}$(S) = 0 ?
b - x_0 ⇒ (A)
4h ⇒ (S)
(A) - 4h ⇒ (A) = 0 ?
(A) > 0 ?
(S) < 0 ?
(S) > 0 ?
x_0 + (S) ⇒ (S)
f(S) ⇒ (S)
$\frac{1}{12}$(S) ⇒ $\frac{1}{12}$ f_4
x_0 + 4h - h ⇒ (S)
f(S) ⇒ (S)
$\frac{1}{3}$(S) + $\frac{1}{12}$ f_4 ⇒ U
x_0 + 2h ⇒ (S)
f(S) ⇒ (S)
$\frac{1}{6}$(S) ⇒ $\frac{1}{6}$ f_2
$\frac{1}{6}$ f_2 + u ⇒ u
x_0 + h ⇒ (S)
f(S) ⇒ (S)
Ondergrens
plant k.o.
0 ⇒ I (d.l.)
0 ⇒ h_S
(S) ⇒ x_0
f(S) ⇒ (S)
$\frac{1}{12}$(S) ⇒ $\frac{1}{12}$ f_0
|A| ⇒ h_S
Precisie
plant k.o.
(S) ⇒ Δ_{max}
$\frac{1}{20}$(S) ⇒ Δ_{max}
koppelopdracht
$\frac{1}{3}$(S) + u + $\frac{1}{12}$ f_0 ⇒ F(h)
{2 · $\frac{1}{6}$ f_2 + $\frac{1}{12}$ f_0 + $\frac{1}{12}$ f_4}.2
⇒ F(2h)
F(2h) - F(h) ⇒ V
|V| > Δ_{max} ?
$\frac{1}{2}$h ⇒ (A) ≠ 0 ?
F(h) - $\frac{1}{15}$ V ⇒ F
I + 4hF ⇒ I (d.l.)
x_0 + 4h ⇒ x_0
$\frac{1}{12}$ f_4 ⇒ $\frac{1}{12}$ f_0
|V| ≤ Δ_{min} ?
2h ⇒ (A)

De twee inloopadressen moeten natuurlijk aan "Tabellatie" worden meegegeven. Als er geen bijzondere Nulfunctie-ingang is wordt daarvoor dan de gewone ingang meegegeven.

Subroutine "Regeltype"

Deze is in staat om bij gegeven x en f(x), die op medegedeelde plaatsen in "Functie" te vinden zijn een nette regel uit te typen eventueel voor differenties, enz. zorgdragend. Een voorbeeld daarvan hebben wij behandeld. B.v. in het geval dat differenties door "Regeltype" berekend worden moet zij speciale ingangen hebben, nl. "Nultype" voor de eerste regel en "Herhaaltype" om een regel te herhalen zonder opnieuw differenties te vormen. De drie inloopadressen moeten aan "Tabellatie" worden meegegeven. Als er geen bijzondere inlopen zijn wordt daarvoor de gewone ingang meegegeven.

Subroutine "Kop"

Deze verzorgt de bovenzijde van de pagina boven de eigenlijke tabel. Bijvoorbeeld typt zij een streep over de breedte van het papier om het later precies te kunnen afsnijden, voert dan het papier een voorgeschreven aantal regels op en typt vervolgens indicaties boven de verschillende kolommen. Als men het paginanummer boven de pagina wenst, dan is daarvoor in "Tabellatie" op een vaste plaats de grootheid p te vinden, die de pagina's telt. Onder de indicaties wordt weer het papier enkele regels opgevoerd.

Subroutine "Staart"

Deze verzorgt de onderzijde van de pagina onder de eigenlijke tabel, ook b.v. met indicaties of een paginanummer. Het is verstandig om haar ook tenslotte een streep onderaan te laten typen, opdat men de bovenzijde en onderzijde van de pagina's kan afsnijden. Als het papier wat scheefgetrokken is, kan men dan de machine stoppen na "Staart" en het papier weer recht zetten. Natuurlijk kunnen "Kop" en "Staart" weer gemeenschappelijke subroutines gebruiken, b.v. de subroutine "Streep", die een streep over het papier typt!

De gehele rest van de administratie wordt door "Tabellatie" verzorgd. Zij hoogt het argument x op van a tot en met b en ziet toe op de pagina-indeling. Allereerst zorgt zij ervoor, dat de pagina in Q blokjes van R regels wordt getypt. Van het Q+1 ste blokje wordt dan nog de eerste regel getypt, de pagina wordt met "Staart" afgewerkt en op de nieuwe pagina, die met "Kop" wordt voorbewerkt, wordt deze extra regel herhaald. Zij doet echter nog meer. Zij zorgt er nl. voor dat het argument x = 0 in de tabel op de eerste regel van een pagina getypt wordt of, als het in werkelijkheid niet voorkomt, getypt zou worden als men het interval voor x zou vergroten. Bij de eerste pagina zorgt zij er nl. voor dat na "Kop" het papier zover wordt opgevoerd, dat dit voor elkaar komt. Evenzo zorgt zij ervoor dat de laatste pagina van de tabel van dezelfde lengte wordt door zo nodig voor "Staart" het papier een passend aantal regels op te voeren. Dit zijn schijnbaar futiliteiten, maar zij zijn in werkelijkheid absoluut noodzakelijk als men goedverzorgde tabellen wil afleveren.

Het bijgaande blokschema spreekt nu voor zichzelf.

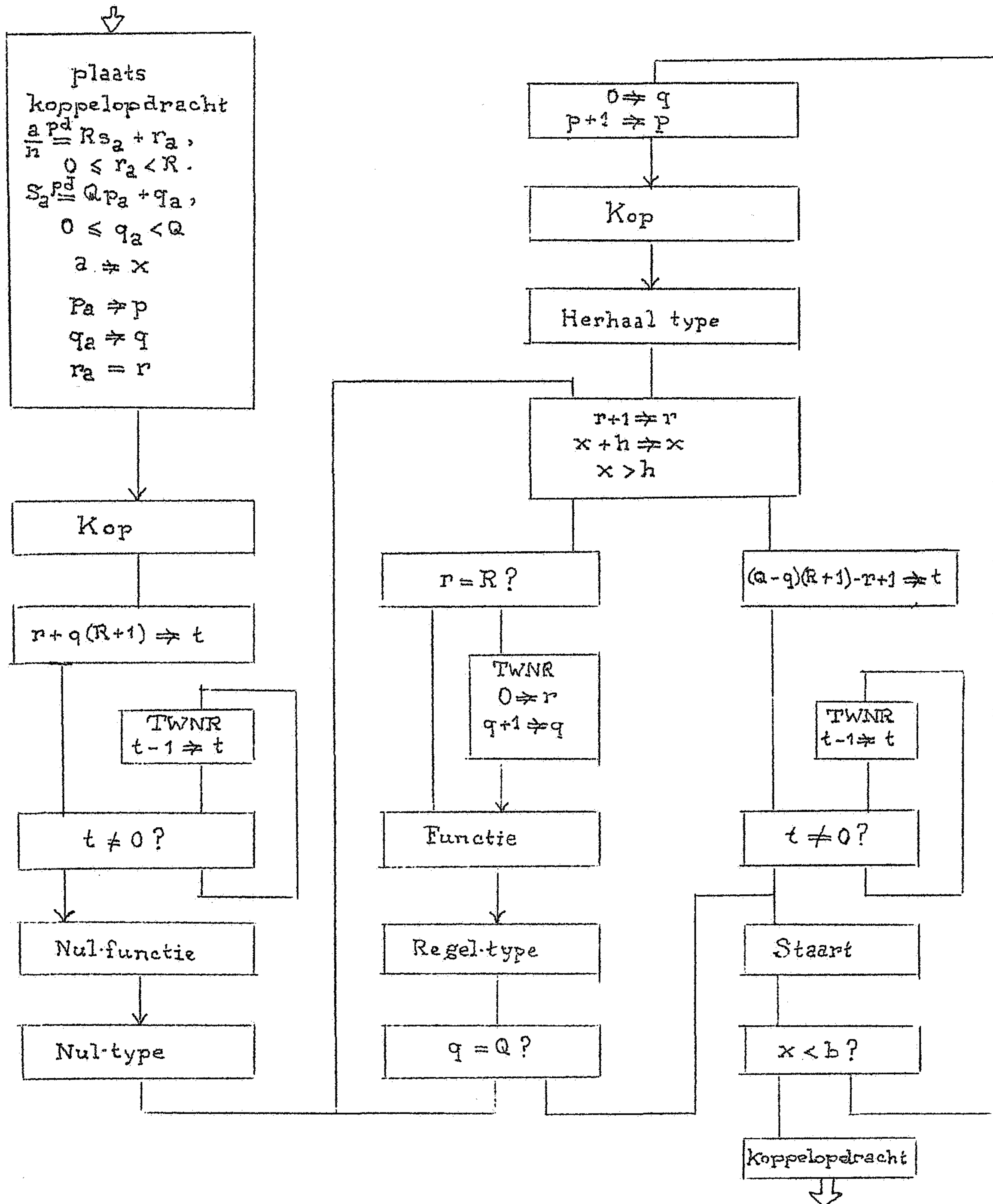
Wij sluiten hiermede de behandeling van de administratieve subroutines af. Om zich goed te realiseren welke macht de subroutines voor speciale functies en de administratieve subroutines hebben, is het aardig om het volgende vraagstuk te analyseren:

Maak een tabel in 6D van de Fresnelintegralen:

$$C(x) = \int_0^x \cos\frac{\pi}{2}t^2.dt \quad \text{en} \quad S(x) = \int_0^x \sin\frac{\pi}{2}t^2.dt$$

voor x = 0(0.001)25 met differenties.

Men zal dan zien, dat men practisch niets behoeft te programmeren!


plaats koppelopdracht
$\frac{a}{h} \overset{pd}{=} Rs_a + r_a,$
$0 \leq r_a < R.$
$S_a \overset{pd}{=} Qp_a + q_a,$
$0 \leq q_a < Q$
$a \Rightarrow x$
$p_a \Rightarrow p$
$q_a \Rightarrow q$
$r_a = r$
Kop
$r + q(R+1) \Rightarrow t$
TWNR
$t - 1 \Rightarrow t$
$t \neq 0$?
Nul-functie
Nul-type
$0 \Rightarrow q$
$p+1 \Rightarrow p$
Kop
Herhaal type
$r+1 \Rightarrow r$
$x + h \Rightarrow x$
$x > h$
$r = R$?
TWNR
$0 \Rightarrow r$
$q+1 \Rightarrow q$
Functie
Regel-type
$q = Q$?
$(Q-q)(R+1)-r+1 \Rightarrow t$
TWNR
$t-1 \Rightarrow t$
$t \neq 0$?
Staart
$x < b$?
Koppelopdracht

14. Superprogramma's

Tenslotte komen wij dan tot de behandeling van superprogramma's, wier taak het is om andere programma's te onderzoeken, te interpreteren, enz. Deze taak kan op allerlei wijzen worden verwezenlijkt. Eén van de allereenvoudigste taken, welke zo'n programma kan opknappen is een bepaald gedeelte van de geheugeninhoud uit te typen of te ponsen. Op deze wijze kan een nette copie van een programma gemaakt worden, een programma, dat overigens zelf in het geheel niet aan bod komt. Het meest practische nut van een dergelijk uitponsprogramma is, dat het een ponsband kan afleveren die heel eenvoudig door het invoerprogramma kan worden gelezen. Een programma inclusief constanten e.d. in het geheugen is immers niets anders dan een aantal cijferrijen, woorden, die wat het inbrengen betreft gerust allen als getal kunnen worden beschouwd. Pas wanneer het programma moet werken wordt het van belang te weten, wat opdrachten en wat getallen zijn. De programmeur moet het natuurlijk ook weten of beter, het is het enige wat hij weet. Hij heeft er weet nog zorg van hoe b.v. een opdrachtenkoppel er binair geschreven uitziet. Hij schrijft en ponst zijn programma volgens een code, die voor hem gemakkelijk is en heeft een invoerprogramma ter beschikking, dat deze code ontrafelt en de juiste cijferrijen in het geheugen plaatst. De taak van dit invoerprogramma is dan echter niet eenvoudig en hoe gemakkelijker en flexibeler de code is, des te ingewikkelder en des te trager is het invoerprogramma. Is het programma echter eenmaal ingelezen en op zijn deugdelijkheid getest, dan kan het uitponsprogramma een band maken, waarop de door het programma met toebehoren beslagen geheugeninhoud als rij van woorden in het tweetallig stelsel wordt gegeven. Voor de ARMAC met 34 bits per woord worden dus 7 pentades van 5 cijfers per woord geponst. Heeft men later dat programma weer nodig, dan kan een speciaal hoekje van het invoerprogramma zo'n biband zeer snel weer inlezen. Het hoeft nl. alleen telkens 7 pentades van de band te lezen, die aan elkaar te rijgen en weg te bergen. Let er wel op, dat het programma noodzakelijkerwijs op precies dezelfde plaats in het geheugen dient te worden geborgen. Bij niet zeer snelle machines is deze wijze van werken zeer aanbevelenswaardig. Het is overigens wel zielig, dat de

met zorg geponste banden van het programma slechts één maal behoeven te worden gebruikt en dan zodra de biband wordt gemaakt, weggegooid kunnen worden.

Deze allereerste taak van de nu beschouwde programma's is wel zeer summier. Immers het feit, dat het uitgeponste een programma is, komt in het geheel niet ter sprake. Het binaire uitponsprogramma kan inderdaad ook heel geschikt gebruikt worden om grote hoeveelheden getallenmateriaal uit het geheugen te redden voor later gebruik. Een speciale rol wordt vervuld, als men bij een programma, dat zeer lang werk heeft, b.v. bij het oplossen van partiële differentiaalvergelijkingen zo nu en dan de bereikte toestand laat uitponsen om, wanneer onverhoopt een fout zou ontstaan, niet gedwongen te zijn geheel opnieuw te beginnen doch slechts bij de toestand welke het laatste was vastgelegd.

Het genoemde invoerprogramma, dat wij trouwens al lang kenden is ook zo'n programma, dat iets met een ander programma "doet" behalve het alleen inlezen. Het vertaalt het nl. uit de programmeurscode in de machinecode, die een beetje of heel veel er van afwijkt. Wat dat betreft is het al een stuk intelligenter dan het binaire uitvoerprogramma, dat alleen amorphe cijferrijen kent. Wij hebben indertijd al een en ander over het invoerprogramma verteld, doch zullen nu over gaan tot enkele andere mogelijkheden om aan te tonen, dat men veel meer kan doen dan tot nu toe behandeld was. Voorop staat, dat in de eerste plaats het doel is de programmeur het leven gemakkelijk te maken, zowel wat betreft het programmeren als ook wat betreft het ponsen.

Wat dit laatste betreft kan men al of niet gebruik maken van speciale apparatuur bij het ponsen. Hoe slim het invoerprogramma ook is, het heeft toch in ieder geval de minimaal benodigde hoeveelheid informatie nodig en daarvoor is een zeker aantal pentades op de band nodig. Dit impliceert een zekere hoeveelheid aanslagen op een gewone ponsmachine, waarop alleen de pentades geponst kunnen worden. Heeft men echter een speciaal gemaakte ponsmachine ter beschikking, die op indrukken van speciale toetsen reageert met het ponsen van meerdere pentades, dan kan men ponsarbeid sparen, iets wat met programmeren niet valt te bereiken. Men heeft dan als het ware een steno-pons-

machine. De ARMAC heeft een dergelijke speciale ponsmachine niet. Een en ander kan resulteren in een aanzienlijke daling van de ponsarbeid.

Een heel andere bron van gemak is daarentegen te vinden in de vorm van het invoerprogramma zelf, eventueel dan nog in samenwerking met speciale roffels via de bandponser geïntroduceerd. Voorbeelden van speciale faciliteiten zijn het zg. drijvend programmeren, het ladderen, utility-programma's.

Het drijvend programmeren (M.V. Wilkes, The use of a "floating address" system for orders in an automatic digital computer, Proc. Cambridge Phil. Soc. 49 (1953, Part 1, 84) is een methode om het vervelende administratieve werk aan programma's drastisch te beperken. Bij de behandelde programma's werden aan de constanten, variabelen, en variabele opdrachten steeds bepaalde adressen toegewezen, b.v. x = (13 B6) of zo, terwijl wij er, als optimaal programmeren niet nagestreefd wordt, in werkelijkheid helemaal niet in geïnteresseerd zijn, waar die x nu precies in het geheugen staat. Wel moet de x op een of andere wijze gekarakteriseerd worden, b.v. door het getal 6. Alle opdrachten die met x opereren, dus het adres van x als numeriek gedeelte in de opdracht hebben,kunnen dus gekarakteriseerd worden door daar 6 voor het adres te schrijven. Waar de opdrachten zelf staan kan ons al helemaal niet schelen, alleen de adressen van de variabele opdrachten en die van de plaatsen in het programma waarnaar wordt gesprongen zijn essentieel, maar deze kunnen ook worden gekarakteriseerd door een beperkt aantal getallen. Wij zullen ons verder een machine voorstellen, waarin slechts één opdracht in een woord gaat, want bij het systeem van de ARMAC met zijn opdrachtenkoppels rijzen een aantal complicaties. Het programma wordt aan één stuk achter elkaar geschreven zonder dat men zich behoeft te bekommeren over het nummeren van de opdrachten. Achteraf mag men hier en daar gewoon opdrachten inlassen. De adressen van de constanten enz. geeft men nummers, niet noodzakelijkerwijs in volgorde. Wel moet men hier en daar natuurlijk die constanten enz. vermelden. Er is allereerst een openingsletter, b.v. P te gebruiken met de volgende betekenis. Als ergens P7 staat, dan is dit een aanwijzing voor het invoerprogramma, dat het adres,

waarop het nu gaat invullen (dat de programmeur niet weet, maar het invoerprogramma wel!) verderop zal aangeduid worden met het cijfer 7. Het invoerprogramma kan daarvan een notitie maken in een drijvend adresboekje en als verder als drijvend adres 7 wordt gebruikt daarvoor het echte adres substitueren. Tot zover is alles eenvoudig. Als een grootheid niet meer nodig is, dan kan zijn drijvend adres opnieuw gebruikt worden. Er ontstaat een moeilijkheid als het adres behorend bij het drijvend adres nog niet is gespecificeerd als het wordt gebruikt. Dit hoeft niet voor te komen bij werkruimten, constanten en variabelen, daar men daarvoor altijd tijdig adressen met P-combinatie kan reserveren, maar is nu en dan practisch onvermijdelijk bij variabele opdrachten en bij sprongopdrachten die naar voren springen. Een manier om deze moeilijkheid op te lossen werkt als volgt. Als men verwijst naar een nog niet vastgelegd drijvend adres, geeft men dit aan met een openingsletter, b.v. Q. Verder kan in het drijvend adresboekje, dat het invoerprogramma bijhoudt bij ieder adres natuurlijk ook nog een functiegedeelte F worden onthouden. Dit zal een functiegedeelte zijn, dat nooit door de programmeur met een drijvend adres voorzien kan of in elk geval mag worden gebruikt. Zulk een functiegedeelte is altijd wel te vinden, b.v. een niet bestaande opdracht. Bij de aanvang van het invoerprogramma worden in het adresboekje deze functiegedeelten F vast ingevuld. Een P-combinatie vult er dan het adres bij in. Een Q-combinatie zal dan als volgt werken. Hij zal de combinatie, die in het adresboekje bij dat drijvend adres staat, plaatsen op het in te vullen adres en in het boekje noteren het functiegedeelte dat op de band is gespecificeerd plus het echte adres waar geborgen moest worden. Dit kan zo een aantal malen gebeuren. De functie van de P-combinatie wordt nu nader zó gedefinieerd, dat hij eerst nakijkt of in het adresboekje het "onbestaanbare" functiegedeelte F is gespecificeerd. Zo ja, dan wordt het vast te leggen adres toegevoegd en de zaak is klaar. Zo neen, dan was het drijvend adres al een of meer keren met een Q-opening gebruikt en moet op die plaatsen het echte adres nog worden ingevuld. Waar dat de laatste keer gebeurd is, is te zien aan het adres in het boekje. De inhoud van dat adres wordt in het boekje geplaatst na het functiegedeelte dat

in het boekje stond te hebben gered. Dit functiegedeelte wordt geassembleerd met het echte adres en op zijn goede plaats opgeborgen. Hierna begint de analyse weer opnieuw. Men zal zien als men nagaat wat er gebeurt, dat alle adressen, waar de Q-combinatie is gebruikt bijgewerkt worden alvorens het programma verder gaat.

Een schema van de organisatie luidt b.v.:

Op band	In adresboekje op drijvend adres 9	Werkelijke adressen	Ingevuld in geheugen
-	F...	-	-
-		-	-
-		-	-
f_1Q9	f_1a_1	a_1	F...
-		-	-
-		-	-
-		-	-
-		-	-
f_2Q9	f_2a_2	a_2	f_1a_1
-		-	-
-		-	-
-		-	-
f_3Q9	f_3a_3	a_3	f_2a_2
-		-	-
-		-	-
-		-	-
P9x		a	x
	f_2a_2	a_3	f_3a
	f_1a_1	a_2	f_2a
	F...	a_1	f_1a
-	Fa	a+1	-
-		-	-
-		-	-
-		-	-

Kennelijke voordelen van het drijvend adresseren zijn gelegen in kortheid en flexibiliteit. Omdat het aantal drijvende adressen tamelijk klein is, en de drijvende adressen dus zeer kort zijn, wordt ook nog aanzienlijke ponsarbeid bespaard. Veranderingen aan het programma zijn zeer eenvoudig aan te brengen, men last eenvoudig in of schrapt opdrachten weg zonder dat omnummering noodzakelijk is. Een nadeel is, dat optimum programmering illusoir wordt en voorts dat men niet weet, waar het programma

in het geheugen precies staat. Hieraan is evenwel gemakkelijk tegemoet te komen. Het drijvend invoerprogramma kan b.v. al bij iedere P-indicatie het werkelijke adres uittypen en eventueel kan men na de invoer even het programma-uittypprogramma alles laten uittypen. Men heeft dan een nette copie.

Het is gemakkelijk kleine accessoires in te bouwen; b.v. komt het nogal eens voor, dat een sprong gemaakt moet worden naar een adres dat een zeer klein aantal plaatsen verderop staat. In plaats van de Q-faciliteit te gebruiken kan men dan b.v. een R-faciliteit benutten, waarbij R3 betekent 3 plaatsen verder. Dit spaart arbeid en tijd.

Nauw verwant met het drijvend adresseren en al of niet in samenhang ermee te gebruiken is het adresboekprogramma. Laat het programma bestaan uit een groot aantal subroutines, die elkaar gedeeltelijk kunnen gebruiken en een hoofdprogramma. Iedere subroutineband sluit met een controlecombinatie, welke zorgt, dat het adres dat volgt op het zo juist ingevulde, in volgorde genoteerd wordt in een adresboek. De ingangen van de subroutines kunnen dan worden gekenmerkt door het nummer van invoer te vermelden. Alleen dient er zorg voor gedragen te worden, dat een subroutine A, die gebruikt wordt door subroutine B ook voor B wordt ingelezen. In het bijzonder wordt het hoofd-programma het allerlaatste ingelezen. Het aanduiden van zo'n adres wordt weer door een speciale letter, b.v. S onderscheiden van de aanduiding van een normaal adres.

Een geheel andere faciliteit is b.v. het ladderen. Bij langzame machines zelfs, als geen optimum programmering wordt of kan worden toegepast, kan, zoals is gezien, desalniettemin de snelheid van een programma worden opgevoerd door het zg. strekken, d.w.z. het volledig uitschrijven van het programma in plaats van het gebruik van cycli. Dit heeft dan tengevolge, dat gedeelten van het programma bestaan uit opdrachtengroepen die herhaald worden en slechts een regelmatige verandering in hun adressen vertonen. In dat geval kan een ladder-programma benut worden, dat uit de indicaties: "hier is een opdracht, schrijf deze _k_ maal neer om de _m_ plaatsen telkenmale het adresgedeelte met _n_ verhogend of iets algemener met de volgende adresgedeelten ... , en sta dan weer klaar om op de eerste

vrije plaats weer in te vullen" voldoende weet om de opgedragen taak te vervullen. Niet alleen is dit veel eenvoudiger voor de programmeur, maar het spaart bovendien veel ponstijd en invoertijd.

Een andere mogelijkheid tenslotte om tijd te sparen ligt in het gebruik van utilityprogramma's. Het gewone invoerprogramma kan b.v. willekeurig achterelkaar gehele getallen en breuken inbrengen. Dit maakt nodig, dat openingsletters als +, -, +., -. worden geponst ter inleiding van de getallen. Moet men nu een grote hoeveelheid getallenmateriaal bestaande uit louter breuken ponsen, dan is het regelmatig aanslaan op de ponsmachine van de indicaties +. of -. een tijdrovende affaire. Speciaal bij kleine getallen betekent het een belangrijke fractie van de gehele ponstijd. Men kan dan utilityconventies invoeren, te begrijpen door een utilityprogramma, b.v. hierin bestaande, dat de indicaties +. als 1 en -. als 2 geïnterpreteerd wordt en dan verbonden gedacht met het eerste cijfer tot één karakter en dus ook door één aanslag op de ponser te ponsen.

B.v. de getallenrij

+.123456 +.3245 -.881 +.7107273 -.001 +.2 +.35 ...

wordt geponst als

11 2 3 4 5 6 13 2 4 5 28 8 1 17 1 0 7 1 7 3 20 0 1 12 13 5 ...

Men moet natuurlijk even aan zo'n conventie wennen, maar ieder die werkelijk een kilometer-band met kleine getallen vol moet ponsen, zal deze kleine moeite gaarne over hebben om zich het ponsen van enkele honderden meters te besparen.

Voor machines werkend met een tamelijk langzaam geheugen, maar waar het door optimum programmeren mogelijk is om de snelheid te verbeteren ligt het voor de hand een superprogramma te hebben, dat in staat is om een door de programmeur gemaakt programma aan het optimaliseringsproces te onderwerpen. Dit bevrijdt de programmeur van deze hoogst onaangename taak.

De functies van de superprogramma's, die tot nu toe behandeld waren, missen nog het meest fundamentele facet, dat ze kunnen hebben. Zij hanteren nl. nog steeds het objectprogramma als een stuk dood materiaal, maar begrijpen nog niet het wezenlijk levende karakter van het objectprogramma, nl. dat dit zelf, als het aan bod kwam, zou gaan manipuleren met getallen of zelfs met opdrachten, dat de opdrachten ervan niet in volgorde zouden worden afgehandeld, maar dat er al of niet conditionele sprongen in voorkomen, dat er cycli aanwezig zijn, enz. Wel waren de behandelde superprogramma's eventueel in staat een vertaling te leveren van een willekeurige code, de programmeurscode, in de machinecode, waaraan nu eenmaal niet valt te tornen, omdat zij de axiomatiek van de gegeven machine is. De vraag rijst - en wordt bevestigend beantwoord - of er superprogramma's kunnen worden ontwikkeld, die het objectprogramma wezenlijk kunnen interpreteren, d.w.z. met inachtneming van de werking van het objectprogramma. Zulke superprogramma's heten interpreterende programma's. Het objectprogramma kan in de machinecode in de machine aanwezig zijn. In dat geval zou het dus zonder meer aan het werk gezet kunnen worden. Het nut van het interpreterende programma zou dus dubieus kunnen lijken en ook inderdaad zijn, als het zich alleen beperkte tot het nabootsen van wat het objectprogramma zou doen, als het aan bod was. Het interpreterende programma kan echter behalve dit meer doen, b.v. uittypen welke opdrachten van het objectprogramma gehoorzaamd worden, en voorts welke getallen gemanipuleerd worden, wat de gewijzigde inhouden van de registers van het rekenorgaan zijn, wat de conditie is, enz. Een dergelijk programma is een prachtig hulpmiddel om fouten in een gemaakt programma op te sporen. Het is noodzakelijkerwijs langzaam, zelfs zeer langzaam, maar kan op allerlei wijzen geweldig worden versneld. Het zou nl. ook alle gebruikte subroutines gaan interpreteren, wat kennelijk niet de bedoeling is, (ook de werking van een semi-interpreterende subroutine, terwijl men daar met veel minder informatie zou kunnen volstaan). Door b.v. van te voren te vertellen aan het interpreterende programma, in welke gedeelten van het geheugen het werkelijk te interpreteren objectprogramma staat, kan het

volstaan met het interpreteren van dat gedeelte, terwijl het de subroutines vrij laat lopen (bij de koppelopdracht pikt het dan de draad weer op). Door de subroutines dan in klassen te verdelen, b.v. zij die hun resultaat in S afleveren en zij die het in een pseudo-accumulator afleveren (zoals de semi-interpreterende subroutines), kan het interpreterende programma zich na het beëindigen van de subroutines dan beperken tot mededelingen aangaande de inhoud van de werkelijk interessante registers (b.v. S of de pseudo-accumulator).

Het samenstel, machines plus interpreterend programma, van deze soort is dan in wezen te beschouwen als een nieuwe machine, een pseudo-machine, die handelt als de echte machine, maar daarbij nog voortdurend verslag uitbrengt van zijn handelingen als een neurotische patiënt aan de psychiater (nl. de programmeur).

Dat een dergelijk programma te maken is, zal wel zonder meer duidelijk zijn op grond van wat tot nu toe behandeld is. Het interpreterende programma moet zelf een pseudo-machine bijhouden in de vorm van een aantal adressen gebruikt als pseudo-opdrachtteller, pseudo-rekenregisters, pseudo-condities, enz. Er rijzen geen principiële moeilijkheden.

Naar aanleiding van de analogie patiënt-psychiater merken wij nog op, dat wij ook het objectprogramma alleen als patiënt kunnen zien. Het interpreterende programma is dan psychiater en het uitgetypte diens aantekeningen. Een hoogst vermakelijk experiment, dat wij jaren geleden op de ARRA pleegden, is het interpreterende programma in duplo in de machine te brengen, daarbij elk specimen als objectprogramma aan het andere aanwijzend. Laat men dan één van beide werken, dan verkrijgt men uitgetypt een verslag van psychiater A, die psychiater B's handelingen analyseert, als deze (B) bezig is A's handelingen te onderzoeken. Het bleek daarbij, dat het interpreteren van dit verslag de programmeur ook spoedig rijp maakt voor de psychiater.

Een tweede soort interpreterend programma, dat veel nuttiger is, werkt op een objectprogramma, dat in de machine aanwezig is, maar in een andere code dan de machinecode is gesteld. Het samenstel machine plus interpreterend programma van deze soort is te

beschouwen als een pseudo-machine van een geheel ander soort dan de oorspronkelijke, b.v. een machine die zonder meer met complexe getallen kan werken of met drijvende komma. Het stelt ons allereerst in staat om een programma met complexe getallen net zo eenvoudig te programmeren als een gewoon programma. Daarbij dient te worden bedacht, dat zelfs in een programma, dat veelvuldig werkt met drijvende complexe getallen toch nog een aanzienlijk gedeelte, nl. de gehele administratie gewoon kan geschieden. Het is een groot tijdverlies ook de administratie te laten interpreteren. Daarom is het van belang, dat men gemakkelijk om kan schakelen tussen interpreteren en niet-interpreteren. De ene omschakeling, nl. die tussen niet-interpreteren en interpreteren is eenvoudig. Het objectprogramma, dat als subject werkt, heeft dan eenvoudig een passende sprong naar het interpreterende programma, te interpreteren als: "ga me nu verder maar interpreteren". De omgekeerde overgang is principieel moeilijker. Het vereist, dat één bepaalde pseudo-opdracht in het objectprogramma door het interpreterende programma herkend en gehoorzaamd wordt, te interpreteren als: "ik wil nu weer zelf werken". Zulk een pseudo-opdracht mag dan natuurlijk niet tevens toegelaten zijn als opdracht in de te interpreteren code. Deze overgang is principieel dezelfde als optreedt bij de invoer als een bepaalde controle-combinatie op de band het invoerprogramma, dat overigens de informatie op de band manipuleert, dwingt om de gelezen opdracht uit te voeren, zodat b.v. het ingelezen programma zelf kan starten.

Een aardige toepassing van dit soort interpreterende programma's rijst bij het ontwerpen van een nieuwe machine als men zelf beschikt over een andere machine, die in een andere code werkt. Men kan deze nieuwe machine nabootsen door een interpreterend programma op de bestaande machine en daarmede dan de ontworpen programma's en subroutines voor de nieuwe machine vast controleren.

Een derde soort interpreterende programma's werkt op programma's die zich nog niet in de machine bevinden, maar werkt op programma's die zich op de band bevinden. Er zijn weer twee typen. Het eerste type maakt uit de informatie op de band een machineprogramma en plaatst dat in de machine. Het gewone invoerprogramma is het eenvoudigste voorbeeld van deze generende programma's. Door uitbreiding van het invoer-programma (ladderen, drijvende adressen hebben wij al behandeld) kunnen wij de code op de band, dus de programmeercode wille-keurig compact maken. Bijvoorbeeld kunnen wij het algebraïsche formules laten uitrafelen, recursiebetrekkingen laten vertalen, enz. Deze gevormde programma's zullen in het algemeen niet bijzonder elegant, doch eerder wat houterig zijn, maar het voornaamste is, dat ze snel te programmeren zijn met kleine kans op fouten. Het aldus gemaakte programma kan uitgeponst worden en staat dus verder op normale wijze ter beschikking.

Het tweede type van de derde soort, het compilerende programma maakt niet van de bandinformatie een programma in de machine, maar interpreteert direct de bandinformatie en voert de gevraagde handelingen uit.

In het algemeen kan men zeggen, dat een schier eindeloze hoeveelheid "kennis" in de superprogramma's kan worden opge-slagen, zodat het converseren met de machine meer het karakter krijgt van het praten tot een collega in plaats van tegen een imbeciele slaaf. Hun toepassing vereist echter grote snel-heid en geheugencapaciteit van de machine.