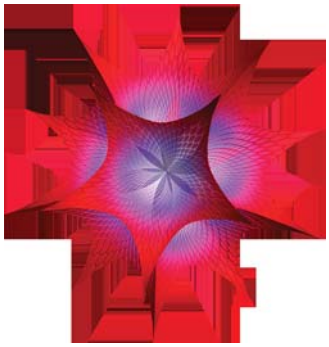
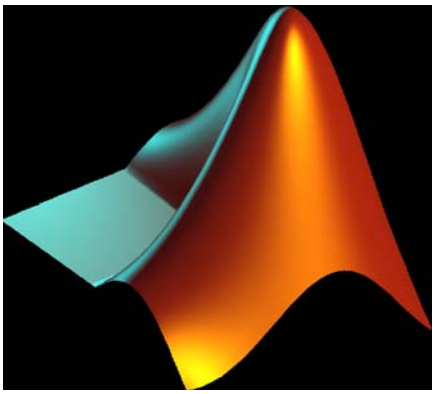
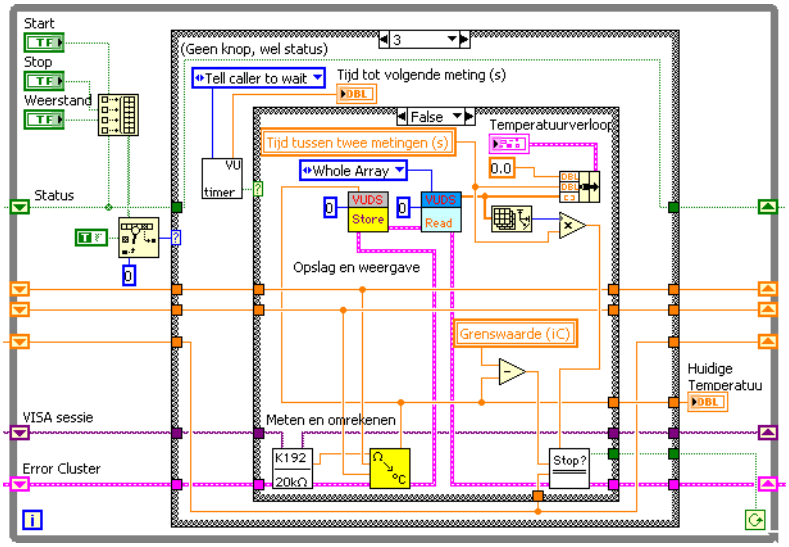
Modelleren, Programmeren en Simuleren
Najaar 2018







dr. Ivo H.M. van Stokkum
drs. Jan M. Mulder
prof. dr. Joost Hulshof
dr. Hans J.W. Spoelder (†2003)

Faculteit der Bètawetenschappen
Afdeling Natuurkunde en Sterrenkunde
vrije Universiteit *amsterdam*

Inhoud Module 1 van Modelleren, Programmeren en Simuleren

Practicumopdrachten afgewisseld met leesteksten

Studiewijzer Modelleren, Programmeren en Simuleren

Docenten

Ivo van Stokkum, ivo@few.vu.nl, Jan Mulder, janm@few.vu.nl, Joost Hulshof, jhulshof@few.vu.nl

Leerdoel en inhoud

Natuurkundestudenten vertrouwd maken met de basisprincipes van modelleren, programmeren en simuleren aan de hand van een aantal case studies. Doel van het praktische gedeelte is om de theoretische kennis in de natuurkundige praktijk te oefenen. De gebruikte omgevingen hierbij zijn "pen en papier", Mathematica, Matlab en LabVIEW. Onder andere komen aan de orde:

- Het gebruik van niet lineaire differentiaalvergelijkingen voor het modelleren en simuleren van cellulaire processen
- Kwalitatieve analyse van niet lineaire differentiaalvergelijkingen
- Structuur van belangrijke omgevingen (Problem Solving Environments)
- Datatypen, nauwkeurigheid en datastructuren
- Algoritmen en fysische toepassingen
- Klassieke architecturen
- Simulatie van een ideaal gas
- Elementaire onderdelen van een geautomatiseerd meetsysteem
- Het formuleren van specificaties voor een eenvoudig geautomatiseerd meetsysteem en het ontwikkelen van een programma uitgaande van deze specificaties.

Vorm

Het vak bestaat uit drie modulen. Module 1 wordt gedoceerd door Joost Hulshof. Module 2 wordt gedoceerd door Ivo van Stokkum. Module 3 is getiteld LabVIEW en Experiment-automatisering, en wordt gedoceerd door Jan Mulder. Voor modulen 2 en 3 is er gedurende vijf weken een dagdeel geroosterd. Voor module 3 zijn er elke week twee bijeenkomsten van 2 uur. In elke bijeenkomst geeft de docent afwisselend college of begeleidt practica. Wekelijks zullen de actuele collegestof en practicumopgaven op Canvas worden aangegeven.

Het dictaat van Module 2 bestaat uit een college en een practicum deel. In het college deel staan sheets die tijdens het college worden besproken, en verdiepende teksten, die deels zullen worden behandeld op de colleges. Het is noodzakelijk om voor en na de colleges de sheets en teksten door te nemen, en eventuele vragen (bv over onbekende begrippen) te noteren. Deze

vragen kunnen dan tijdens het volgende college gesteld worden. Maak tijdens het college zelf aantekeningen (b.v. in je dictaat), want niet alles wat de docent vertelt staat in het dictaat.
In het practicum deel van het dictaat staan de opdrachten voor achtereenvolgens de Mathematica en Matlab practica. Uitleg over de werking van Matlab is verweven met de opdrachten. Het tijdschema staat in de checklist op de website van Module 2.
Het dictaat van Module 3 bestaat uit zes hoofdstukken met opdrachten. Het tijdschema staat in Appendix E. Ook hier bestudeer je de behandelde stof, en stelt eventuele vragen tijdens het volgende college/practicum. Het dictaat van Module 1 staat op Canvas.

Tijd

6 ECTS komt overeen met 168 uur. Het aantal contacturen is 66. Dit betekent dat je zelf buiten de contacturen elke week minstens 8 uur aan het vak dient te besteden, voor het bestuderen van de stof, het afmaken van de opdrachten, het opzoeken van begrippen op internet, en voor het schrijven van de eindverslagen.

Inleveropdrachten

Op Canvas staat welke practicumopgaven van Module 2 ingeleverd dienen te worden. De **zelfgemaakte** practicumopgaven lever je in via Canvas, het gemiddelde cijfer hiervoor bepaalt 30% van het eindcijfer van Module 2. In Module 3 worden de gemaakte opdrachten uit hoofdstuk 1-3 tijdens het practicum individueel nabesproken.
Samenwerken tijdens de practica van module 2 en 3 is toegestaan en dient in het verslag duidelijk vermeld te worden. Plagiaat (klakkeloos overnemen van uitwerkingen van anderen zonder vermelding) wordt echter gemeld aan de examencommissie en bestraft. Ook het beschikbaar stellen van je uitwerkingen aan een ander is riskant, want als die ander het dan onvermeld inlevert, word je allebei bestraft.

Toetsing en eindcijfer

Elke module wordt afgerond met een eindopdracht. In de vijfde week dient het verslag van de eindopdracht van Module 2 te worden ingeleverd en nabesproken met de docent. Bovendien worden tijdens deze nabespreking vragen gesteld over de gehele stof van Module 2. In de zevende week dient het verslag en het programma van de eindopdracht van Module 3 te worden ingeleverd en nabesproken met de docent.
Het eindcijfer is het gemiddelde van de cijfers voor de drie modulen.

Interpretatie van een plaatje

“Een plaatje zegt soms meer dan duizend woorden” is een bekend spreekwoord. Dit geldt vaak ook voor wetenschappelijke plaatjes of grafieken. Hier volgt een aantal vragen die je jezelf kunt stellen bij het bekijken en interpreteren van grafieken.

Wat staat er langs de assen (grootheden/eenheden)?

Wat zijn het voor assen (lineair/logaritmisch/…)?

Bedenk bij een logaritmische as dat $^{10}\log(2)\cong 0.3$ en $^{10}\log(3)\cong 0.5$ etc.

Zijn het ruwe data, is het een modelberekening of een wiskundige functie, of een combinatie?

Als het combinatie is van ruwe data en een modelberekening of gefitte functie, hoe goed is de overeenkomst ?

Als het gaat om een fit aan ruwe data, kun je iets zeggen over residuen of meetfouten ?

Als er daarbij parameters worden geschat, kun je die makkelijk terugzien in het plaatje (helling, asafsnede, vervaltijd, …) ?

Wat voor verband(en) zie je in het plaatje (lineair, exponentieel, macht of wortel, …) ?

Zitten er bijzondere punten in (minima, maxima, knik, afwijking,gat,…)?

Wat is de boodschap of betekenis van het plaatje ?

Als je de relevante vragen voor jezelf hebt beantwoord:

Verzin een pakkende titel.

Schrijf een onderschrift (caption), of toelichting.

In de practica van dit vak zal regelmatig gevraagd worden om een onderschrift voor een plaatje te schrijven of een plaatje uit te leggen. Dit is een belangrijke academische vaardigheid die ook terugkomt bij het mondeling.

Functionele verbanden

Een cruciale vraag die je jezelf dient te stellen bij een plaatje waarbij y-waarden als functie van x-waarden staan uitgezet is:

Wat voor verband(en) zie je in het plaatje (lineair, exponentieel, macht of wortel, …) ?

Daarmee hangt samen de vraag:

Wat zijn het voor assen (lineair/logaritmisch/…)?

NB je kunt alleen een logaritme nemen als het argument positief is. In onderstaande tabel staat aangegeven wanneer het kiezen van een logaritmische as een eenvoudiger plaatje oplevert.

Overzicht van veelvoorkomende verbanden

verband	formule	natuurlijke logaritme	handig plaatje	helling	asafsnede
lineair evenredig	$y = a\,x$		lineair-lineair	a	0
	$y = a\,x$	log(y)=log(a)+log(x)	log-log	1	log(a)
omgekeerd evenredig	$y = \frac{a}{x}$	log(y)=log(a)-log(x)	log-log	-1	log(a)
lineair plus constante	$y = a\,x+b$		lineair-lineair	a	b
exponentieel	$y = b\,\exp(ax)$	$\log(y) = ax+\log(b)$	lineair-log	a	log(b)
	$y = 2^x$	$\log(y) = x \cdot \log(2)$	lineair-log	log(2)	0
	$y = 10^x$	$\log(y) = x \cdot \log(10)$	lineair-log	log(10)	0
logaritmisch	$y = a \cdot \log(x)$		log-lineair	a	0
macht	$y = b \cdot x^a$	$\log(y) = a \cdot \log(x) + \log(b)$	log-log	a	log(b)
	$y = x^a$	$\log(y) = a \cdot \log(x)$	log-log	a	0
wortel	$y = \sqrt[a]{x}$	$\log(y) = (\log(x))/a$	log-log	1/a	0
kwadratisch polynoom	$y = ax^2 + bx + c$		lineair-lineair		

Merk op dat wanneer je een lineair verband ziet in een log-log plaatje je nog steeds heel goed naar de helling moet kijken voor het onderliggende verband tussen x en y.

Wanneer je in plaats van een natuurlijke logaritme een logaritme met grondtal 10 neemt, wat een veelvoorkomende optie is in programma's als Excel, moet je de kolom *helling* in de table voor een lineair-log plaatje delen door $\log(10)$.

Informatica

- is de studie en de wetenschap van de theoretische fundamenten van informatie,
- de theoretische informatica,
- en de implementatie en toepassing in computersystemen.

Bron: Wikipedia

Subdisciplines

- Fundamentele informatica (**datastructuren, algoritmen**, formele talen-, berekenbaarheids-, complexiteits-, grafentheorie)
- Informatie- en gegevensbeheer (databanken, data mining)
- **Technisch-wetenschappelijk rekenen** (computergraphics, bio-informatica, wetenschappelijk programmeren, computeraritmetiek)
- Kunstmatige intelligentie (toegepaste logica, optimalisatietechnieken, neurale netwerken, kennisrepresentatie)
- Software engineering (softwareontwikkeling, softwaremetrieken, ICT-project management, formele specificatietechnieken, software-re-engineering, verificatie van programmacorrectheid)
- Computernetwerken en communicatiesystemen
- Mens-computerinteractie

Bron: Wikipedia

30 Core Technologies of Computing

- **algorithms**
- artificial intelligence
- compilers
- *computational science*
- **computer architecture**
- data mining
- data security
- **data structures**
- databases
- decision support systems
- distributed computation
- e-commerce
- graphics
- human computer interaction
- information retrieval
- management info systems
- natural language processing
- networks
- operating systems
- parallel computation
- programming languages
- real-time systems
- robots
- *scientific computation*
- software engineering
- supercomputers
- virtual reality
- vision
- visualization
- workflow

bold: bouwstenen, italic: science toepassingen
bron: Denning (2004) Great Principles in Computing Curricula, SIGCSE'04, 336

Tools (hulpmiddelen)

- In Toegepaste Informatica worden veelal hulpmiddelen gebruikt die speciaal zijn toegesneden op een bepaald toepassingsgebied:
  Problem Solving Environments
- In dit vak bestuderen we een aantal veelgebruikte Problem Solving Environments (PSEs) aan de hand van case studies
- Daarbij komen drie bouwstenen aan de orde die essentieel zijn voor de gehele informatica, dus ook voor PSEs: datastructuren, algoritmen, computerarchitecturen

Problem Solving Environments

Low level: taal

C(++), Java, Fortran, Python, HTML/JavaScript,
Visual Basic for Applications (VBA), ...

Mid level: Integrated Development Environment

Visual Studio, Code Warrior, Net Beans, ...

High level: “programma”, PSE

Mathematica, Matlab, R, LabVIEW, Excel, ...

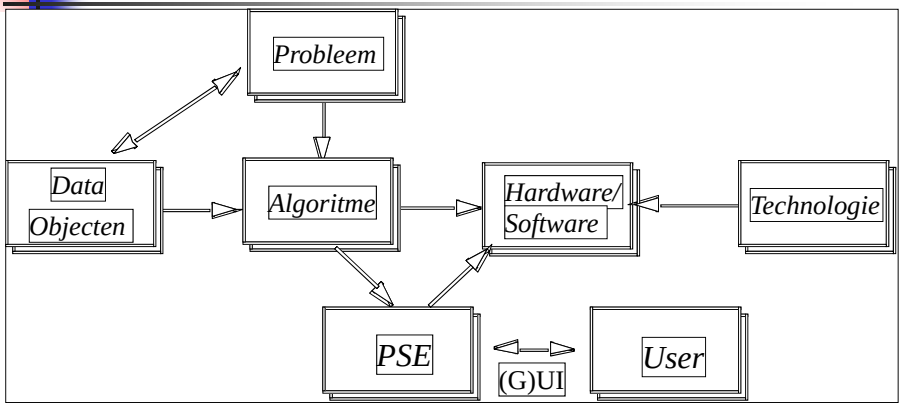
Low vs high level

- Low level:
  (Matlab print statement, C-achtige syntax)
  sprintf(‘de integer n is %d \n’,n)
  (vertaling: sprintf=string print formatted,
  %d=decimaal integer format, \n=newline)
- High level:
  (Mathematica)
  DSolve[{x’’[t]==-a x[t]},x,t]
  lost analytisch een differentiaalvergelijking op

Indeling

- Inleiding
- Basisbegrippen
- Userinterface
- Case Studies
  - C/Fortran,
  - Excel,
  - Mathematica,
  - HTML/JavaScript,
  - LabVIEW.

Problem Solving Environment (PSE) definitie



***Definitie**: Een PSE is een applicatiedomein specifieke omgeving waarin (oplossingen voor) problemen **eenvoudig** genoteerd en **efficiënt** bepaald kunnen worden.*

Enige voorbeelden PSE

Applicatie Gebied	*Voorbeeld*	*Mogelijkheden*
Internet	Mozilla/Firefox, Internet Explorer	HTML viewing E-mail, ftp
Administratief	MS-Office, OpenOffice, Lotus Notes, ...	Tekstverwerking, Presentatie, Spreadsheet, Database, Netwerking
Animatie	3D Studio, Data Explorer, Data Visualizer	Rendering, Modellering, Animatie,
Wiskunde	*Mathematica*, Maple	Numerieke berekeningen, Graphics, Formulemanipulatie
Interfacing/ Process Controle	LabVIEW, HP-VEE	Instrument controle, Data acquisitie, Graphics
Wetenschappelijk rekenen	Fortran77, Fortran90, C, C++, Java,Visual Studio	Numeriek rekenwerk, enige debug faciliteiten.
Dataverwerking	R, Splus, Matlab, SPSS	signaal- en dataverwerking, statistische analyse, graphics, …

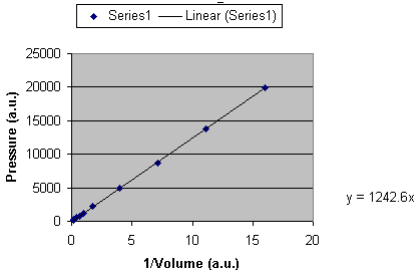
C/Fortran Language

```
/*
 * check if a particle is in a box
 */
int    InBox(Particle *p, Position ll, Position ur)
{
      if((p->new.x > ll.x) && (p->new.x < ur.x) &&
        (p->new.y > ll.y) && (p->new.y < ur.y)    )
            return 1;
      else
            return 0;
}
```

```
/*
 * Step a particle over a timestep
 */
void   Step(Particle *p, double timestep)
{
      p->new.x = p->current.x + timestep*p->vx;
      p->new.y = p->current.y + timestep*p->vy;
}
```

Low level
moeilijk leesbaar

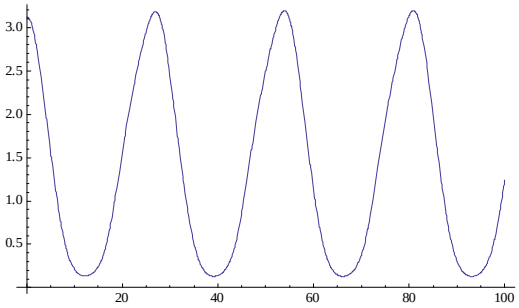
uit C case study:
Simulation ideal gas



Mathematica

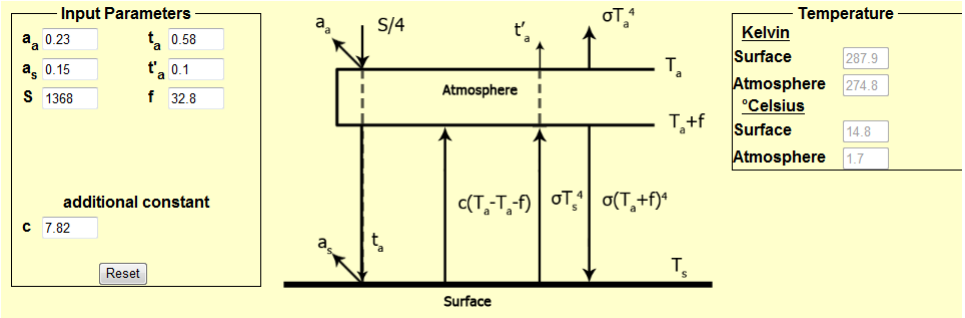
```
In[1]:= myfig1c[Kd_: 1, p_: 2, ε_: 10, S_: 1] :=
 Module[{values2}, values2 = {k1 → 1, k2 → 1, ET → 1, Km → 1};
  f1crule = NDSolve[{y'[t] == k1 S Kd^p / (Kd^p + y[t - ε]^p) - k2 ET y[t] / (y[t] + Km),
       y[t /; t < -50] == 2} /. values2, y, {t, 0, 100}] // Flatten; yfig1c = y[t] /. f1crule;
  Plot[yfig1c, {t, 0, 100}]]

In[2]:= myfig1c[]
```

Out[2]=

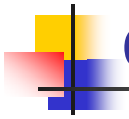


High level Case study: Biochemical oscillator

HTML/JavaScript

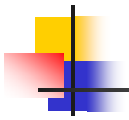


Case study: Simple Greenhouse Effect model
http://www.nat.vu.nl/envphysexp



Criteria voor juiste keuze PSE

- Eenvoud van de notatie
- Efficiency van de oplossing (rekentijd)
- Onderhoudbaarheid van de oplossing
- Robuustheid en stabiliteit van de oplossing
- Leercurve voor het gebruik



Basisbegrippen

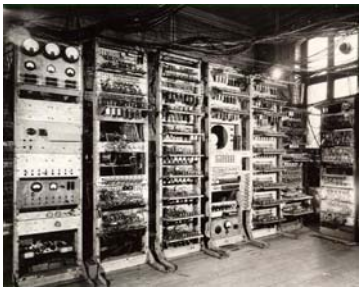
Talen, algoritme

Taal

- De taal dient om het probleem, en de oplossing, te kunnen formuleren
- De taal moet een kleine *cognitieve afstand* tot het probleem hebben voor de mens om effectief te kunnen zijn

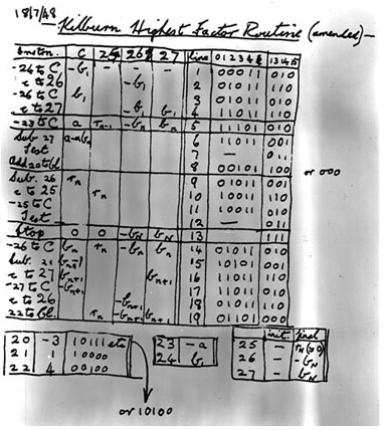
Taal: Eerste Generatie

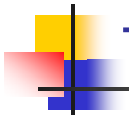
- Begonnen op de Mark I van Universiteit van Manchester
- Machine specifiek en weinig informatief.
- Vooruitgang t.o.v. van de ‘hardwired’ programma’s



Taal: Eerste Generatie

Eerste generatie programma





Tweede generatie: Assembler

Vertaler van symbolische notatie naar specifieke waarden. Notie van Operaties en Operanden.

Voordeel
grotere mate van leesbaarheid en efficiency

Nadeel
Nog steeds afhankelijk van de hardware

Zie bv cs.uwp.edu/staff/fossum/Classes/CS440/Notes/assemblers.html

Voorbeeld

```
        load    r1,#0
        load    r2,#60
loop    cmp     r1,r2
        bne     ready
        inc     r1
        br      loop
ready   hlt
```

Problemen tot zover

- Machine afhankelijkheid
- Gebrek aan bruikbare data structuren bv. $\boldsymbol{x = A^{-1}b}$ vereist getallen, vectoren, (inverse) matrices
- gebrek aan bondige, leesbare notatie (bv. Inverse, matrix vector produkt, vector assignatie)
- Uitsluitend toepasbaar op numerieke problemen

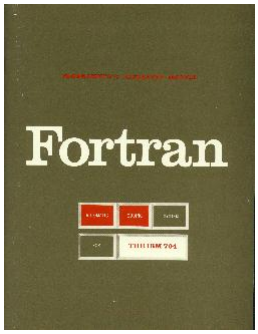
Doorbraak: 3de generatie

Concept: geautomatiseerde (formule) vertaler (Fortran I)

Voordeel: oplossing van machine afhankelijkheid (taak compiler), hoog abstractie niveau

Eerste implementatie:
John Backus (IBM) 1954

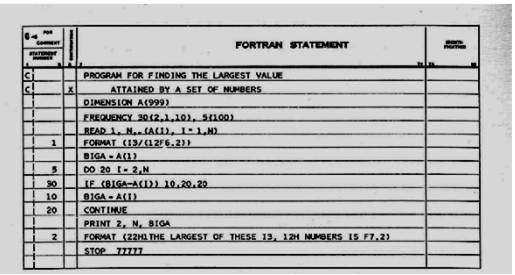
Machine: IBM 704



Efficiency dominant

[Our development group] had one primary fear. After working long and hard to produce a good translator program, an important application might promptly turn up which would confirm the views of the sceptics:
.... its [translated] program would run at half the speed of the hand-coded version. It was felt that such an occurrence, or several of them, would almost completely block acceptance of the system

Backus, 1964



		FORTRAN STATEMENT
C		PROGRAM FOR FINDING THE LARGEST VALUE
C	X	ATTAINED BY A SET OF NUMBERS
		DIMENSION A(999)
		FREQUENCY 30(2,1,10), 5(100)
		READ 1, N,.(A(I), I = 1,N)
	1	FORMAT (I3/(12F6.2))
		BIGA = A(1)
	5	DO 20 I = 2,N
	30	IF (BIGA-A(I)) 10,20,20
	10	BIGA = A(I)
	20	CONTINUE
		PRINT 2, N, BIGA
	2	FORMAT (22H1THE LARGEST OF THESE I3, 12H NUMBERS IS F7.2)
		STOP 77777

Tweede doorbraak

- Koppeling van data en functies: object georiënteerde talen.

- Matrix (voorbeeld)

optelling	elementsgewijs
vermenigvuldiging	rij maal kolom
inversie	regel van Kramer
deling	niet gedefinieerd

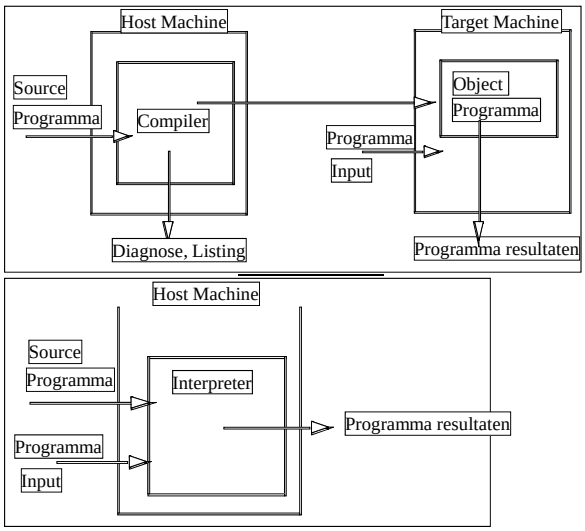
Efficiency

Omgeving	Type	Relatieve Snelheid
C	Compiler	1.
C++	Compiler	0.5
Lisp	Interpreter	0.1
Mathematica Matlab	Compiled functions	~0.5
Mathematica Matlab	Interpreter	0.0003

Dedicated talen

- Wetenschappelijke (functies) (Fortran)
- Systeem programmeertalen (C)
- Commando talen (sh, csh, ksh, perl)
- Procedurele taal
- Niet-procedurele taal
- Data flow (LabView, DX, Excel)
- Object georiënteerde taal (C++, F90, Java)
- Real-time taal (Ada)

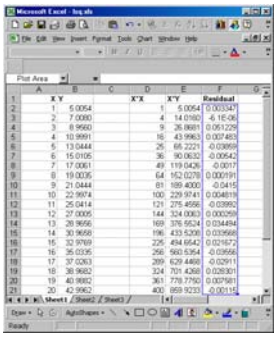
Verwerken van talen



Data flow & Control flow

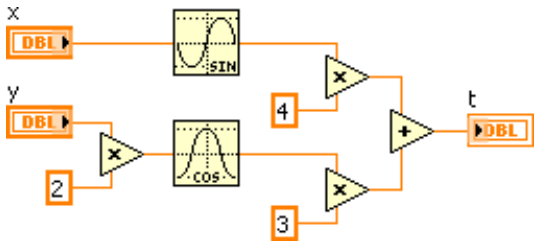
$$a = (\sum_{i=1}^{N} y_i \sum_{i=1}^{N} x_i^2 - \sum_{i=1}^{N} x_i \sum_{i=1}^{N} x_i y_i) / (N \sum_{i=1}^{N} x_i^2 - (\sum_{i=1}^{N} x_i)^2)$$

$$b = (N \sum_{i=1}^{N} x_i y_i - \sum_{i=1}^{N} x_i \sum_{i=1}^{N} y_i) / (N \sum_{i=1}^{N} x_i^2 - (\sum_{i=1}^{N} x_i)^2)$$



```
sx = 0; sy = 0; sxx = 0; sxy = 0;
for(i=0; i<; i++) {
        sx += x[i];
        sy += y[i];
        sxx += x[i]*x[i];
        sxy += x[i]*y[i];
}
a = (st*sxx - sx*sxy)/(N*sxx - sx*sx);
b = (N*sxy - sx*sy)/(N*sxx - sx*sx);
```

LabVIEW: data flow



t=4 sin(x)+3 cos(2 y)
Graphical Language G

Inhoud

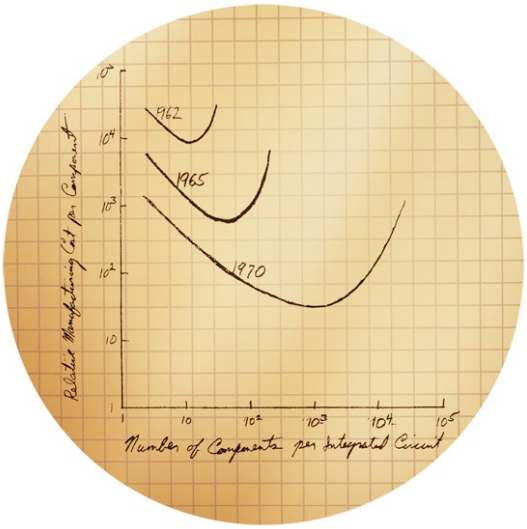
- Inleiding
- Informatie
  uitdaging: berekening eiwitvouwing
- Communicatie
- Computers en wij
- En verder …

Definities

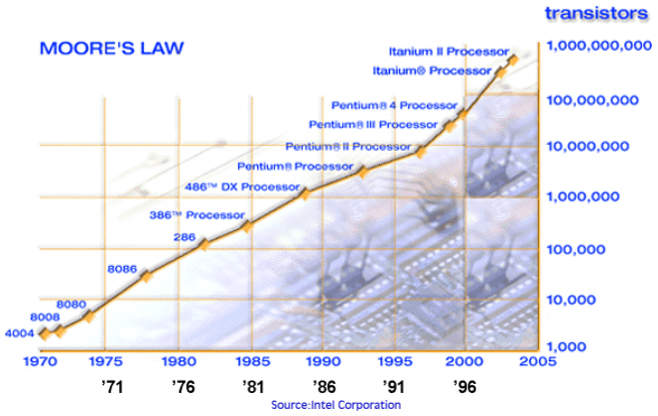
- Informatie
  'gebeurtenis in een omringende wereld'
- Communicatie
  'overdracht van informatie'
- Computers
  '(autonome) Informatie verwerkende systemen'
- Wij
  'autonome informatie verwerkende systemen die communiceren'

Technologie

Trends

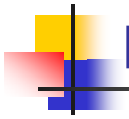


Inleiding



Exponentiële groei

http://www.extremetech.com/extreme/210872-extremetech-explains-what-is-moores-law
http://www.extremetech.com/extreme/203490-moores-law-is-dead-long-live-moores-law



Moore's Law Marching On

PC Magazine, August 19, 2002

It happens like clockwork. Every odd year, Intel introduces a new manufacturing process; soon after, it produces desktop CPUs that are far smaller and faster than ever before, proving once again that company co-founder Gordon Moore was dead right: The number of transistors on CPUs doubles every 18 months or so. In 1999, the company introduced its 0.18µ process, a means of building chips whose smallest features are only 0.18 millionths of a meter wide. In 2001, it introduced the 0.13µ process. This week, it announced its new 90-nanometer process, slated for use with mainstream processors by the end of 2003. Surpassing even its name, the process can etch devices into silicon that are only 0.05 millionths of a meter wide.

...

Intel Drives Moore's Law Forward with 65 Nanometer Process Technology [August 30, 2004]

Fall 2006 Conroe 65 nm processor, uses less power

45nm and 32nm, due in 2007 and 2009 ??? Indeed, they made it!

Analyst prediction in 2009: Moore's Law 'will break down in 2014'

- Moore's law has a hidden economic layer: each new scale of miniaturisation needs massive investment in new manufacturing plants.
- "At [18nm] the industry will start getting to the point where … costs will be so high, that the value of their lifetime productivity can never justify it."
- http://www.computeractive.co.uk/pcw/news/1920602/moores-law-break-2014

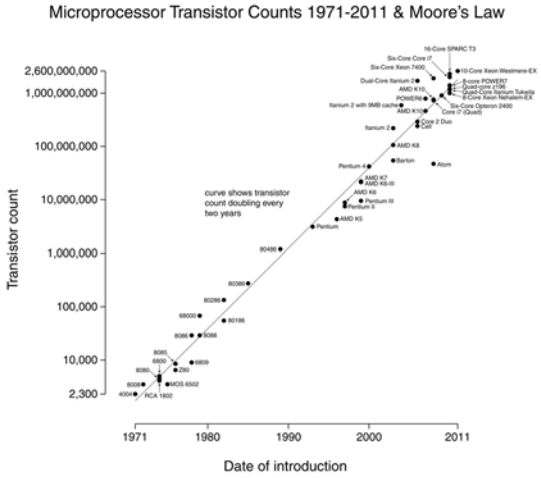
Engineers vs Economists

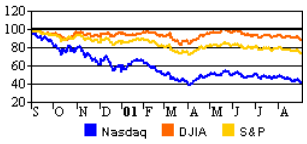
- http://www.extremetech.com/computing/184946-14nm-7nm-5nm-how-low-can-cmos-go-it-depends-if-you-ask-the-engineers-or-the-economists
- http://www.extremetech.com/computing/190845-intel-forges-ahead-to-7nm-without-the-use-of-euv-lasers/2

Dec 2011: Intel kondigt 14 nm circuits aan

- http://www.nordichardware.com/news/69-cpu-chipset/44802-intels-pat-bliemer-14nm-running-in-the-lab-exclusive.html
- Het Fab 42-complex komt in de plaats Chandler in Arizona te staan en gaat 5 miljard dollar kosten. In 2013 moet Fab 42 klaar zijn voor gebruik.
- http://en.wikipedia.org/wiki/14_nanometer
- http://en.wikipedia.org/wiki/Intel_Tick-Tock
- http://tweakers.net/nieuws/78540/intel-heeft-eerste-14nm-chips-werkend-in-testlabs.html
- http://arstechnica.com/gadgets/2013/10/intels-next-generation-broadwell-cpu=s-delayed-due-to-yield-problems/
- http://www.brightsideofnews.com/news/2013/10/16/intele28099s-broadwell-delayed-until-2014-due-to-a-defect.aspx
- http://www.nordichardware.com/Science-Technology/microlitography-with-extreme-uv-breathes-new-lift-into-moores-law-in-2015.html
- **http://www.extremetech.com/computing/174832-intel-cancels-14nm-fab-42-in-arizona-but-its-nothing-to-worry-about**

Groei die door blijft gaan … ?





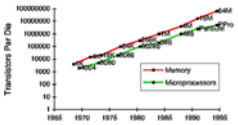
Inleiding

- Ontwikkeling gaat al ruim 40 jaar exponentieel maar grenzen aan de groei komen in zicht
- Met de huidige technologie in dit decennium in de problemen
- Nauwe correlatie met de economie

De Toekomst


Gordon Moore, Intel co-founder

The Future of Moore's Law



http://www-ics.ee.ic.ac.uk/surp98/report/sp24/index.html

Russian Roulette …

Microprocessor development is a high-risk, slow-motion gamble, requiring multiyear lead times and bet-the-company investments in research and equipment.
Robert Palmer, former CEO of Digital Equipment and an AMD board member, once likened the process to Russian roulette, but with a twist: "You put a gun to your head and pull the trigger," he says, "and four years later you find out if you blew your brains out."

http://money.cnn.com/magazines/business2/business2_archive/2002/11/01/331620/

Moore's law until 2020?

- http://news.techworld.com/operating-systems/3576581/moores-law-is-dead-says-gordon-moore/
- http://spectrum.ieee.org/semiconductors/devices/the-status-of-moores-law-its-complicated
- http://www.extremetech.com/computing/165331-intels-former-chief-architect-moores-law-will-be-dead-within-a-decade
- http://www.extremetech.com/computing/178529-this-is-what-the-death-of-moores-law-looks-like-euv-paused-indefinitely-450mm-wafers-halted-and-no-path-beyond-14nm

Moore in 2002 on new applications enabled by increasing chip capacity:

Difficult question. In 1980 I would have missed the PC, in 1990 the Internet.
Speech recognition by computers would profoundly change the way people work with machines and interact with one another.
The power of a network of powerful machines that couple humans together, I think is going to have an impact that is far beyond my comprehension.

Untrained speech-recognition in noisy environments

4 Oct 2011
Siri
Iphone 4S

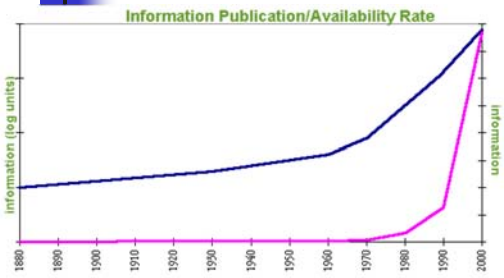
http://en.wikipedia.org/wiki/Siri_(software)
http://en.wikipedia.org/wiki/Google_Now

Informatie

Gebeurtenis in een
omringende wereld

http://www.sims.berkeley.edu/research/projects/how-much-info-2003/

Informatie verdubbelt elke 3 jaar



De laatste decennia is er een sterke steiging van de hoeveelheid informatie, zowel gedrukt als in electronische vorm

The New York Times ON THE WEB

1 NY Times on Sunday = evenveel informatie als een mens in de middeleeuwen in een heel leven tot zich nam

http://en.wikipedia.org/wiki/Big_data

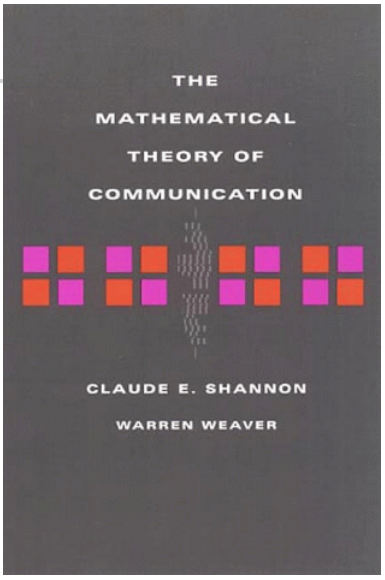
As of 2013, the World Wide Web is estimated to have reached 4 Zettabytes. http://en.wikipedia.org/wiki/Zettabyte

Informatie

- Shannon
  "The mathematical theory of communication", 1949

- In woorden

  "Informatie en waarschijnlijkheid hebben een omgekeerd evenredige relatie"



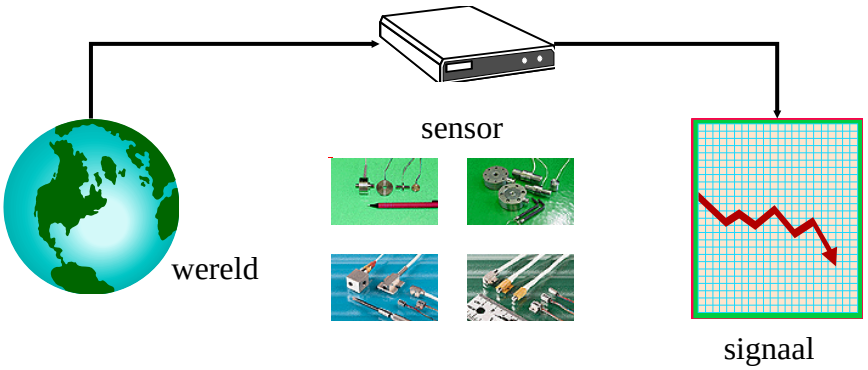
Informatie

$$I_i = \log_2(1/p_i) = -\log_2(p_i)$$

$$I_{gemiddeld} = \sum_i p_i I_i$$

bv: dobbelsteen: 2.58 bits per uitkomst

Informatie



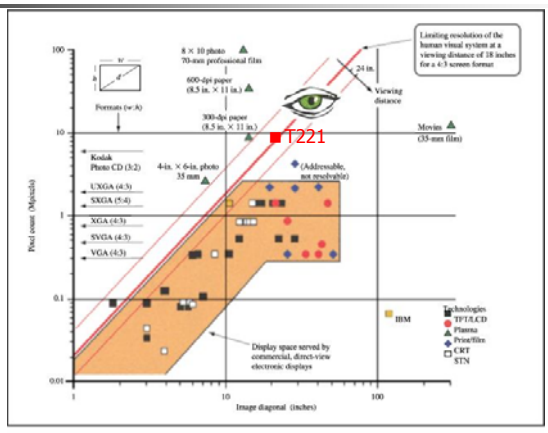
NB.: zonder sensor is informatie overdracht onmogelijk

Informatie

	Informatie (bits)	Receptoren	Zenuw aansluiting
Gezicht	$3 \cdot 10^5$	10^7 (kleur) 10^8 (schemer)	$1 - 2 \cdot 10^5$
Gehoor	$2 - 5 \cdot 10^4$	$1\text{-}3 \cdot 10^4$	$1 - 2 \cdot 10^4$
Tast	$2 \cdot 10^5$	$5 \cdot 10^5$ (druk) $3 \cdot 10^5$ (pijn)	$1 \cdot 10^4$
Warmte	$2 \cdot 10^5$	$1 \cdot 10^4$ (hitte) $3 \cdot 10^5$ (kou)	$1 \cdot 10^4$
Reuk	10 - 100	10^7	$2 \cdot 10^3$
Smaak	10	10^7	$2 \cdot 10^3$

Bron: J.H. van Bemmel en M.A. Mussen (Eds) Handbook of Medical Informatics, Springer 1977

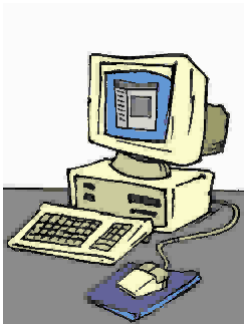
Informatie

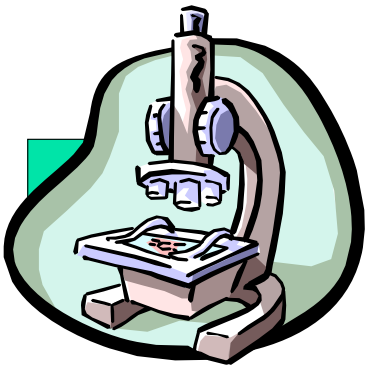


A view of display space showing existing electronic, paper, and film displays.

http://en.wikipedia.org/wiki/4K_resolution http://en.wikipedia.org/wiki/Retina_Display

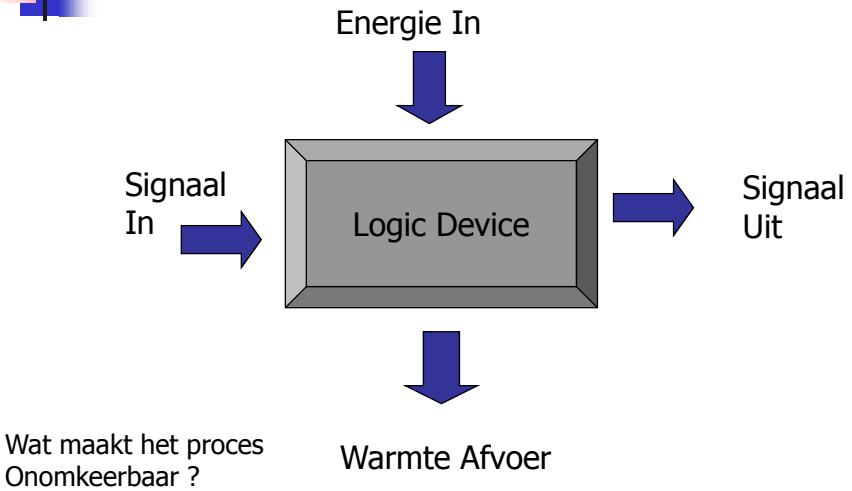
Informatie





Second law of thermodynamics:
No process is possible whose sole result is the absorption of heat from a reservoir and the conversion of this heat into work

Informatie

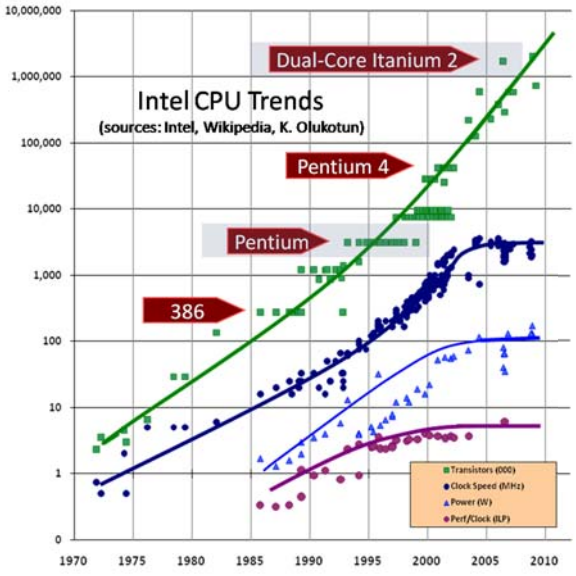


Wat maakt het proces Onomkeerbaar ?

Informatie

- Consequentie:
  Een beslissing nemen kost 'warmte' en hoe meer beslissingen (clock frequentie) hoe meer warmte.
- Nu
  Chip met paar miljard transistoren
  ≈100 Watt (vergelijk mens 80 Watt)
  Rond 2017 (Moore in 1997) problemen met huidige technologie
- Nog haalbaar
  Factor 10^5 tot fysische limiet

Warmte beperkt klokfrequentie



3 GHz Clock Speed (f)

100 W Power (P)
$P=CV^2f$
V: voltage
C:capacitance

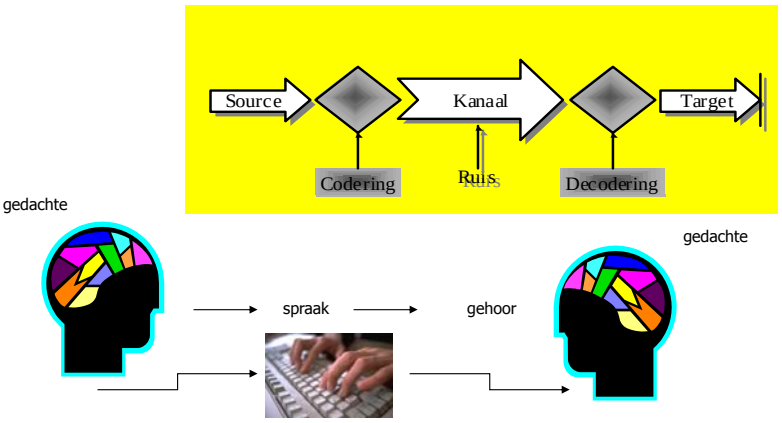
http://gotw.ca/publications/concurrency-ddj.htm

Communicatie

- Eindige *snelheid*
  Kan de informatie invalideren
- Eindige *bandbreedte*
  Niet alles kan overgedragen worden
- *Opstarttijd*
  Zelfde effect
- *Ruis*
  Verminking van de informatie

Communicatie

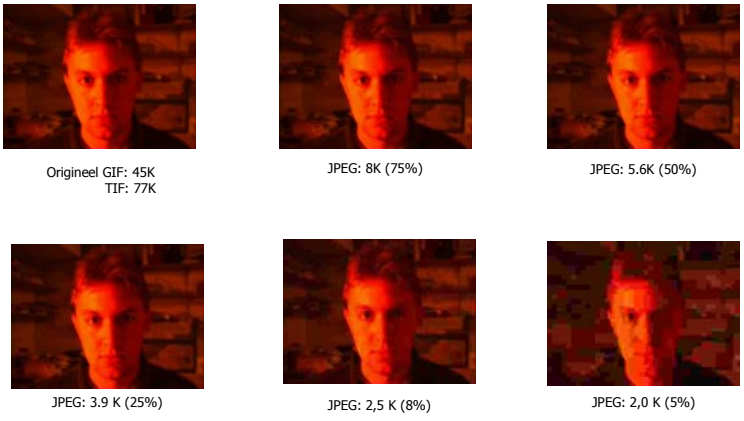
Model volgens Shannon



Communicatie

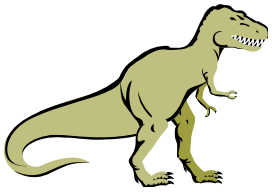
- Discreet versus analoog
  (beperk de invloed van ruis)
- switched versus continu
  (maak efficiënt gebruik van capaciteit)
- Gecomprimeerd versus normaal
  (maak gebruik van voorkennis)

Compressie



gebruik voorkennis over de informatiestructuur

Communicatie



Probleem:
Te traag van staart naar kop

Hersenen:
0.5 - 2.0 m/s (20 - 40 msec)

speciaal: 100 m/s

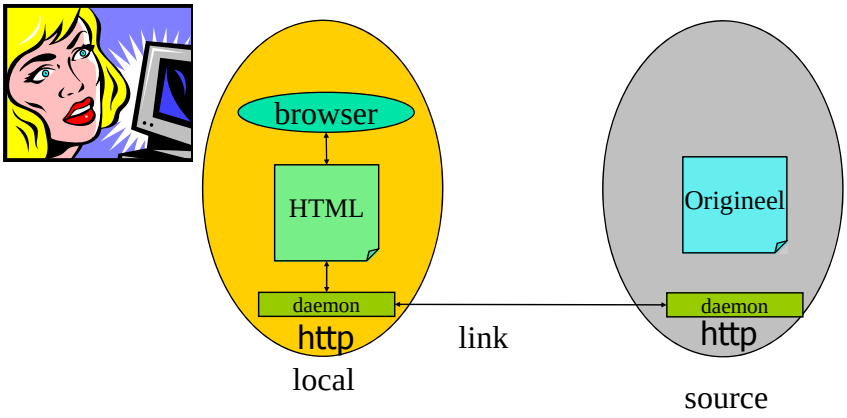
Communicatie

- Netwerk technologie
  (verbinding)
- HTML
  (ontwikkeling door CERN)
  (taal)
- Browser
  (Mosaic NCSA)
  (viewer)

Internet

History:
http://www.netvalley.com/intval1.html

Communicatie

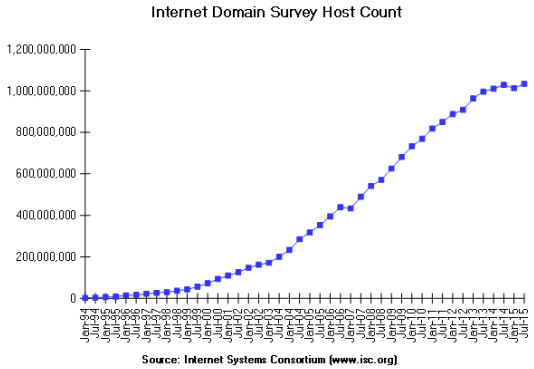


Communicatie

GIF	Image	Any
JPEG	Image	Any
XMB	Image	Unix
TIFF	Image	Any
PICT	Image	Any
Rasterfile	Image	SUN
MPEG	Movie	Any
Avi	Movie	Microsoft
QuickTime	Movie	Apple
AU	Audio	SUN
WAV	Audio	Microsoft
MP3	Audio	Fraunhofer Inst.
PostScript	Document	Any
Acrobat	Document	Any
….	……	….

Communicatie

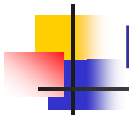
van exponentiële naar lineaire groei Internet naar...



https://www.isc.org/network/survey/

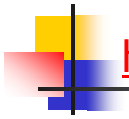
Communicatie

http://www.webfoundation.org/programs/challenges/



Future of the Web

- The W3C mission is to lead the World Wide Web to its full potential by developing protocols and guidelines that ensure the long-term growth of the Web. http://www.w3.org/Consortium/mission.html



http://www.webfoundation.org/vision/

- We envision a world where all people are empowered by the Web. Everyone — regardless of language, ability, location, gender, age or income — will be able to communicate and collaborate, create valued content, and access the information that they need to improve their lives and communities. The creativity of the billions of new Web users will be unleashed. The Web's capabilities will multiply, and play an increasingly vital role in reducing poverty and conflict, improving healthcare and education, reversing global warming, spreading good governance and addressing all challenges, local and global.

Computers & Wij

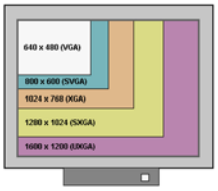
(Autonome)
Informatie verwerkende systemen

- Uitdaging
  Communiceer de informatie tussen Computers en Mensen op een voor de **mens** natuurlijke manier

  vb. Userinterface
  Visualisatie
  Virtual Reality

Computers & Wij

Virtual reality



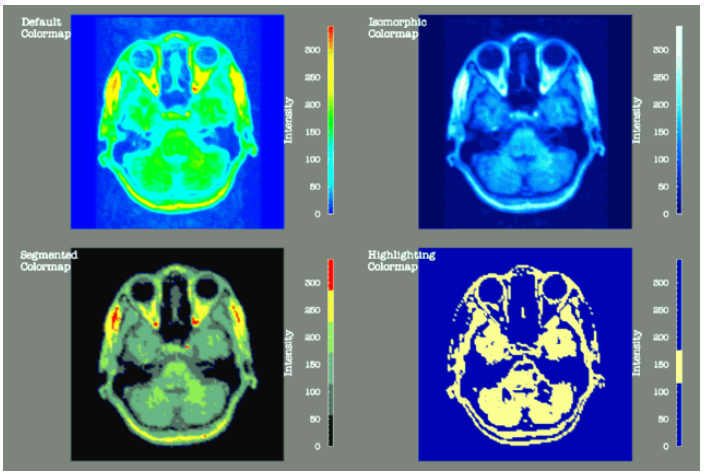
Standard display

Visuele Informatie Overdracht

Augmented reality

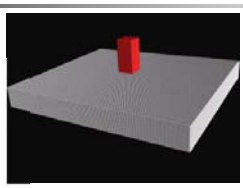
tiled display

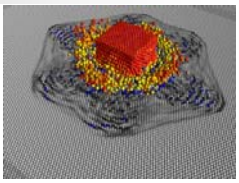
Computers & Wij

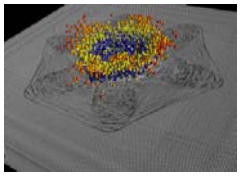


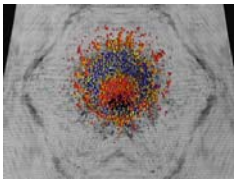
Computers & Wij

Simulatie
dode materie





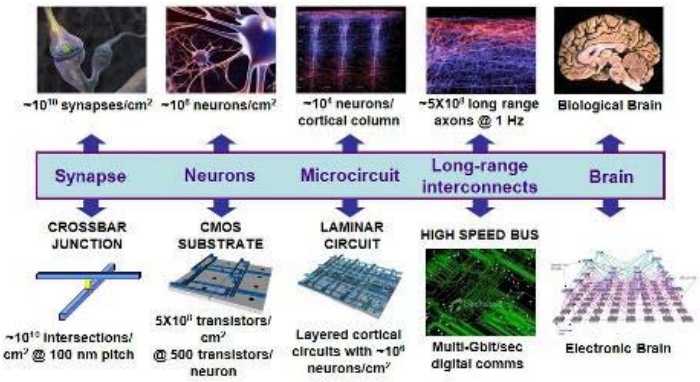




Computers & Wij

Simulatie
levende materie

EU flagship
€1.2 billion



The Rise of the Thinking Machine
http://www.hpcwire.com/hpcwire/2011-08-25/the_rise_of_the_thinking_machine.html
http://www.hpcwire.com/hpcwire/2013-10-16/supercomputing_essential_to_human_brain_project.html
http://www.humanbrainproject.eu/

Ontwikkelingen

- **Connectiviteit**
  Mobiel, omnipresent, internet
- **Groei**
  hoe lang blijft exponentiële groei doorgaan ?
- **User interface**
  Auditief, tactiel domein (VR)
- **Miniaturisatie**
  Toenemende integratie met mens

Internet pagina's die je voor dit vak dient te lezen

Over Informatica
http://nl.wikipedia.org/wiki/Informatica
http://en.wikipedia.org/wiki/Computer_science

Over een geavanceerd hulpmiddel, de IDE
http://nl.wikipedia.org/wiki/Integrated_development_environment

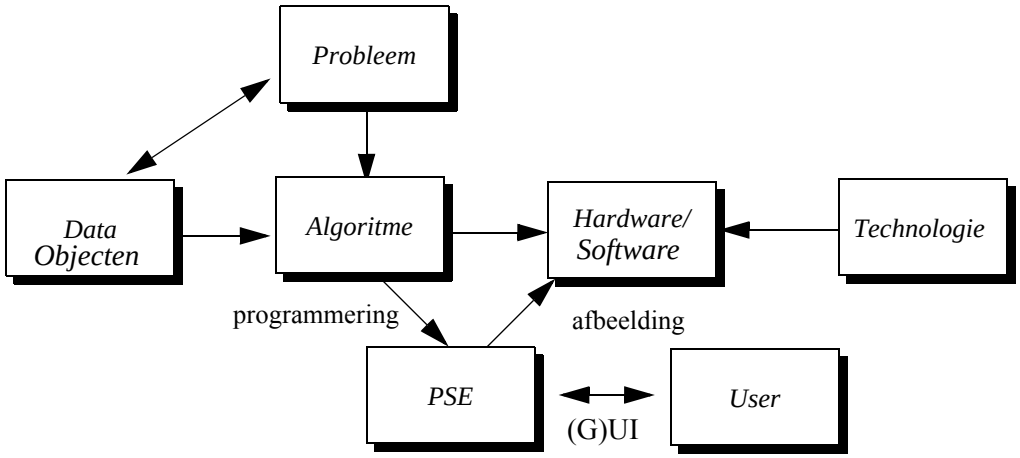
Over Problem Solving Environments (PSE)
http://research.cs.vt.edu/pse/intro.html

Over de gebruikte Problem Solving Environments
http://en.wikipedia.org/wiki/Mathematica
http://en.wikipedia.org/wiki/Matlab
http://en.wikipedia.org/wiki/LabVIEW

Hoofdstuk 2: Datatypen en Datastructuren

2.1 Inleiding

In dit hoofdstuk zullen wij enige datatypen en datastructuren behandelen. Zoals veel software concepten is het idee van datatypen en datastructuur met name ontwikkeld in de 70er jaren voortbouwend op de ervaringen die opgedaan waren met de 3e generatietalen C, Fortran en Cobol. *Onder een datatype verstaan wij een klasse van data objecten tesamen met een verzameling van operaties om ze te construeren, te manipuleren en te verwijderen.* Elke taal of omgeving heeft een verzameling van datatypen die in de taal ingebouwd zijn. Daarenboven is het in sommige omgevingen mogelijk om zelf nieuwe types te definiëren. In Figuur 1 heb-



Figuur 1 Samenhang van de componenten die van belang zijn bij het gebruik van een programmeeromgeving (of PSE).

ben wij de relatie tussen datatypen en omgevingen aangegeven.
Een datatype kan op twee verschillende niveau's bestudeerd worden: *specificatie* en *implementatie.* De basis van de specificatie wordt gevormd door:

- de *attributen* van het type,
- de *waarden* die het object kan aannemen,
- de *operaties* die gedefinieerd zijn voor de manipulatie van de data.

Als voorbeeld kan men denken aan een array. Tot de attributen zou men het aantal dimensies, de subscript range van elke dimensie, en het type van de componenten rekenen; de waarden spreken voor zich, terwijl tot de operaties het indiceren van de elementen, de operaties op de elementen en de (on)mogelijkheden om attributen te veranderen worden gerekend.
Implementatie van een datatype omvat:

- de specifieke *representatie* die gebruikt wordt in een computer om het desbetreffende datatype tijdens executie van een programma op te slaan,
- de wijze waarop de *operaties* die de types manipuleren geïmplementeerd zijn in termen van specifieke algoritmen.

Onder een datastructuur verstaan wij een verzameling (of aggregaat) van data objecten. Voor datastructuren zal degene die ze definieert zelf moeten zorg dragen voor de implementationele aspecten. Wat de specificatie betreft zjin met name de operaties van belang, de rest volgt logisch uit de definitie van de elementen.

Een data object van een bepaald datatype noemen wij een *instantatie*. Instantaties hebben een *naam* die het mogelijk maakt om ze te onderscheiden. Naar een instantatie met een naam zullen wij kortheidshalve refereren als naar een *variabele*.

Een variabele heeft een bepaalde 'levensduur' ook wel aangeduid als scope. De start wordt gekenmerkt door de creatie van de (ruimte voor de) variabele en een eventuele initialisatie, terwijl het einde gemarkeerd wordt door de destructie van de variabele. Beschouw ter illustratie de volgende twee voorbeelden (zie Fig-

```
function out=solver(a,b,c)
d=discriminant(a,b,c);
...
```

```
>> d=discriminant(1,2,-2)
```

Figuur 2 Illustratie van het begrip 'scope' van een variabele (in dit geval j). Links is **d** *een locale variabele binnen de solver functie, rechts is* **d** *een globale variabele in het command window van Matlab.*

uur 2). In het linker geval wordt de variabele *d* 'gemaakt' op het moment dat de functie *solver* aangeroepen wordt en weer opgeruimd op het moment dat de functie beëindigd wordt**.** Het maken en verwijderen van de variabele gebeurt overigens automatisch voor de gebruiker, wij komen hier nog op terug. In het rechter geval is er sprake van een globale variabele: na het invoeren van de commandoregel wordt deze gemaakt en bij het einde van het programma (hier Matlab) wordt hij verwijderd.

Bij de operaties die op een datatype uitgevoerd kunnen worden verdient het aanbeveling om een onderscheid te maken tussen die operaties die het type behouden en die welke het type veranderen. De laatste categorie wordt meestal aangeduid als een *cast*. Het casten van een variabele is maar in een zeer beperkt aantal gevallen een goed gedefinieerde en omkeerbare operatie. Als voorbeeld kan met denken aan de omzetting van een niet-geheel getal naar een geheel getal. Voor alle gevallen waarin de fractie niet gelijk aan nul is, is deze operatie onomkeerbaar.

Een rigide introductie van datatypen en -structuren zou te ver voeren. Hier willen wij ons concentreren op een aantal types en structuren die van belang zijn voor N&S omgevingen. Wij zullen opmerkingen in dit hoofdstuk illustreren met voorbeelden uit de eerder aangegeven omgevingen als Matlab en Mathematica.

2.2 Datatypen

2.2.1 Booleans

Logische variabelen worden ook wel Booleans genoemd, naar de logicus http://en.wikipedia.org/wiki/George_Boole. Een logische variabele kan True (T, of 1) of False (F, of 0) zijn. Belangrijke bewerkingen op logische variabelen p en q zijn negatie (Not, of **!** of $\neg p$), conjunctie (And, of **&**, of $p \wedge q$) en disjunctie (Or, of || , of $p \vee q$). Het resultaat van zulke bewerkingen wordt weergegeven met behulp van een waarheidstabel, zie tabel 1.

p	q	$\neg p$	$p \wedge q$	$p \vee q$
T	T	F	T	T
T	F	F	F	T
F	T	T	F	T
F	F	T	F	F

tabel 1 Waarheidstabel

2.2.2 Getallen stelsels

Binnen de informatica bedient men zich niet al te vaak van het decimale stelsel, dit is meestal beperkt tot die gevallen waarin communicatie met de (menselijke) gebruiker noodzakelijk is. Om verschillende redenen zijn binair (2-), octaal (8-) en hexadecimaal (16-tallig) stelsels populairder. In ons (arabisch) stelsel geldt dat 'positie' 'waarde' is, derhalve kan men algemeen een getal noteren als

$$d_{n-1}d_{n-2}.....d_0 \tag{1}$$

en de numerieke waarde bepalen als

$$\text{waarde} = \sum_{i=0}^{n-1} d_i r^i \tag{2}$$

waarbij men *r de radix* noemt en er geldt dat

$$\max(d) + 1 = r \tag{3}$$

Binair

Voor het binaire (2-tallige) systeem impliceert dit dat $d = \{0, 1\}$ en $r = 2$. Een binaire eenheid noemt men een *bit.* Voor een serie van *n*-bits, zoals boven aangegeven geldt dat zij maximaal 2^n verschillende mogelijkheden kunnen representeren. Het bit dat correspondeert met de hoogste tweemacht (d_{n-1}) noemt men het *most-significant* bit, omgekeerd noemt men het laagste bit het *least-significant bit.*

Octaal

Bij het octale stelsel geldt dat $d = \{0, .., 7\}$ en $r = 8$. Het digit bij het octale stelsel heeft geen aparte naam en kan gedacht worden als een triplet van bits. Teneinde octale getallen te kunnen onderscheiden is in C gedefinieerd dat de reeks met een *0* begint. Dus zou in C gelden dat

$$017 = 15_{10}$$
$$17 = 17_{10}$$

Net als bij het binaire systeem kan men het aantal mogelijkheden eenvoudig bepalen, dit is 8^n.

Hexadecimaal

Tot slot wordt er ook nog gerekend in en gebruik gemaakt van het hexadecimale systeem. Hiervoor zijn de numerieke tekens onvoldoende vandaar dat hiervoor geldt $d=\{0, 1,...,9,A,...,F\}$ en $r = 16$. In vele toepassingen (onder andere C) wordt de conventie gehanteerd dat een hexadecimaal getal begint met *0x*, dus

$$0x17 = 23_{10}$$
$$0xF7 = 247$$

Een kwartet van bits is nodig en voldoende om één hexadecimale eenheid te coderen. Het zal duidelijk zijn dat ons decimale stelsel niet direct in een van deze representaties uit te drukken is, althans niet maximaal efficiënt.

Mathematica heeft een hoop functionaliteit ondergebracht in de functie *BaseForm* die het mogelijk maakt om waarden in willekeurige stelsels te representeren.

2.2.3 Enumeraties

Bij enumeraties wordt er een correspondentie gelegd tussen de elementen van een verzameling en getallen.

In[2]:= BaseForm[123456, 2]
Out[2]//BaseForm= 11110001001000000_2

In[3]:= BaseForm[123456, 8]
Out[3]//BaseForm= 361100_8

In[4]:= BaseForm[123456, 16]
Out[4]//BaseForm= $1e240_{16}$

Figuur 3 Gebruik van de functie BaseForm in Mathematica.

2.2.3.1 Characters

Characters, in goed nederlands de letters van het alfabet vermeerderd met interpunctietekens, leestekens en cijfers, vormen het voorbeeld van een geënumereerd type. De meest gebruikte conventie van de afbeelding van getallen naar characters is de zogenaamde *ASCII* code, weergegeven in figuur 4 op pagina 49. Het betreft hier een 7-bits code zodat er 128 mogelijke characters gerepresenteerd kunnen worden. Naast deze AS-

```
000 NUL001 SOH002 STX003 ETX004 EOT005 ENQ006 ACK007 BEL
010 BS 011 HT 012 NL 013 VT 014 NP 015 CR 016 SO 017 SI
020 DLE021 DC1022 DC2023 DC3024 DC4025 NAK026 SYN027 ETB
030 CAN031 EM 032 SUB033 ESC034 FS 035 GS 036 RS 037 US
040 SP 041  ! 042  " 043  # 044  $ 045  % 046  & 047  '
050  ( 051  ) 052  * 053  + 054  , 055  - 056  . 057  /
060  0 061  1 062  2 063  3 064  4 065  5 066  6 067  7
070  8 071  9 072  : 073  ; 074  < 075  = 076  > 077  ?
100  @ 101  A 102  B 103  C 104  D 105  E 106  F 107  G
110  H 111  I 112  J 113  K 114  L 115  M 116  N 117  O
120  P 121  Q 122  R 123  S 124  T 125  U 126  V 127  W
130  X 131  Y 132  Z 133  [ 134  \ 135  ] 136  ^ 137  _
140 ’  141  a 142  b 143  c 144  d 145  e 146  f 147 g
150  h 151  i 152  j 153  k 154  l 155  m 156  n 157  o
160  p 161  q 162  r 163  s 164  t 165  u 166  v 167  w
170  x 171  y 172  z 173  { 174  | 175  } 176  ~ 177 DEL
```

Figuur 4 ASCII tabel voor 7-bits code. De oneven kolommen geven de (octale) waarde, de even kolommen het ASCII teken.

CII code kan men bij IBM systemen ook de EBCDIC code aantreffen. Dit is een 8-bits code die dus 256 mogelijkheden biedt om characters te representeren. Overbrengen van een ASCII file naar een EBCDIC machine, en omgekeerd, zal duidelijk in een catastrofe resulteren.
Ook voor het representeren van getallen kunnen de characters gebruikt worden. De voor- en nadelen zijn apert: de informatie is voor iedereen leesbaar en met minimale moeite overdraagbaar naar andere systemen (iets wat niet onderschat moet worden), operaties op de getallen zijn echter niet direct mogelijk omdat er eerst een conversie uitgevoerd moet worden naar een intrinsiek formaat. Voor het representeren van een niet-geheel getal met *n* significante cijfers zijn *7+n* characters nodig, zoals duidelijk is uit Figuur 5. Ge-

$\pm.123456\mathrm{E} \pm 000$

Figuur 5 ASCII representatie van getal, let op de overhead van 7 posities.

bruikt men een ASCII representatie om getallen weg te schrijven dan moet men bedenken dat er altijd

voldoende decimalen weggeschreven moeten worden om afrondingsfouten te voorkomen.

Binnen de PSE's die ten tonele gevoerd zijn in de vorige paragraaf zijn de characters en hun manipulatie bijzaak, er zijn slechts rudimentaire middelen aanwezig. Voor allerlei toepassingen zijn wij meer in aggregaten van characters, **strings**, geïnteresseerd.
Binnen Matlab en Mathematica zijn hiervoor een aantal *intrinsic* functions gedefinieerd. De meeste operaties op strings hebben betrekking op het zoeken naar substrings binnen een string of het sorteren van strings op volgorde.

2.2.3.2 Integers

Heeltallige grootheden duidt men aan met de naam *integers*. Van oudsher vormden zij bij numerieke toepassingen een speciale klasse omdat rekenen met deze grootheden meestal sneller was dan met niet-heeltallige grootheden. Een ander voordeel van gehele getallen is dat zij exact gerepresenteerd kunnen worden, alleen het aantal elementen is afhankelijk van grootte van de representatie. Mathematisch corresponderen zij met Z.
Op machine niveau worden de gehele getallen meestal gerepresenteerd in een 2-complements notatie wat impliceert dat indien er *N* bits ter beschikking zijn een interval van

$$[-2^{N-1}, 2^{N-1} - 1] \tag{4}$$

bestreken wordt, een niet symmetrisch interval. De gebruiker moet zich helaas bewust zijn van de mogelijke grootte van *N*, er bestaat een grootste positieve getal[1] en een kleinste negatieve getal dat gerepresenteerd kan worden. Aangezien *N* kan verschillen op verschillende architecturen is het uiterst onverstandig om er vanuit te gaan dat *N* constant is. Dit temeer daar in de meeste gevallen er een soort cyclische afbeelding gehanteerd wordt, d.w.z. het grootste positieve getal plus één is gelijk aan het kleinst negatieve getal (een waarschuwing hiervan wordt niet gegeven). Dit betekent dat de standaard associatieve en distributieve eigenschappen voor gehele getallen

$$\begin{aligned} a + (b - c) &= (a + b) - c \\ a \times (b - c) &= a \times b - a \times c \end{aligned} \tag{5}$$

niet hoeven te werken. Zij zullen in elk geval niet werken in dit voorbeeld:

(grootste_positieve_getal + 1) - 2 = *ongedefinieerd*

terwijl

(grootste_positieve_getal - 2) + 1 = *gedefinieerd*

Bij C en Fortran77 is de gebruiker het slechtste af: hier moet hij in principe de representatie die gebruikt wordt voor het gehele getal[2] omzetten in een bereik. Bij Fortran90 kan er met behulp van een intrinsic functie (*select_int_kind*) bepaald worden of een bepaald bereik representeerbaar is op de machine waarop men werkt.
Voor de gehele getallen kent Mathematica geen methode om te bepalen wat het grootst mogelijke getal is.
Het voordeel van de 'oneindige nauwkeurigheid' waarmee gewerkt wordt binnen de gehele getallen moet niet onderschat worden. Het voornaamste probleem zit met name in het (te) kleine bereik van de getallen op 32-bits systemen.

[1] Voor 32 bits systemen is dit 2147483647 en -2147483648 (in woorden: iets meer dan 2 miljard)
[2] C kent maar liefst drie verschillende formaten voor gehele getallen.

2.2.4 Floating Point

In ons dagelijks taalgebruik duiden wij alle grootheden waarin een zogenaamde decimale punt voorkomt aan als niet-gehele of floating point getallen, dit laatste ter onderscheid van de hiervoor behandelde fixed point getallen. Wiskundig, en ook natuurkundig, gesproken drukken wij ons daarmee slordig uit. Correcter ware het om te spreken over de rationele getallen, dwz. de getallen die als quotiënt van twee gehele getallen geschreven kunnen worden, en de reële getallen. Beide hebben, voor computers, hun specifieke problemen: de eerst genoemde kunnen uitgedrukt worden in zoveel decimalen als men maar wil, terwijl voor de tweede groep in principe een oneindig aantal decimalen nodig is voor de juiste representatie. Het zal duidelijk zijn dat hier de implementatie en specificatie die in sektie 2.1 genoemd zijn hand in hand gaan met de hoeveelheid ruimte die ter beschikking staat voor de representatie.
In de meeste toepassingen wordt één en ander minder dramatisch doordat fysische grootheden meestal met een eindige nauwkeurigheid bekend zijn. Als voorbeeld: als een straal maar met 1% nauwkeurigheid gemeten kan worden, heeft het weinig zin om voor een oppervlakte berekening π in 100 decimalen in te voeren.
Voor de rationele getallen geldt dat, eigenlijk, voor het behoud van nauwkeurigheid, alle bewerkingen hier uitgevoerd moeten worden op de quotiënten van de gehele getallen. Bij de meeste derde generatie talen is het zo dat de gebruiker zich vastlegt op een bepaald aantal decimalen bij de keuze van het datatype. Dit aantal decimalen wordt op zijn beurt gedefinieerd door de hardware van het systeem. De overweging hiervoor is tweeledig: enerzijds ruimte, de definitie van deze talen stamt uit de tijd dat geheugen duur was, en anderzijds snelheid: hoe meer decimalen hoe trager de berekening.

Mathematica heeft vermoedelijk de meest natuurlijke en volledige implementatie gemaakt van de niet-gehele getallen. Ten eerste is het mogelijk om niet gehele getallen te benaderen als rationeel getal. In het onderstaande wordt dit toegelicht.

```
In[1]:= N[Pi, 10]
Out[1] = 3.141592654
In[2] := Rationalize[N[Pi, 10], 0]
Out[2] = 245850922/78256779
```

De functie *N* vraagt Mathematica om een expressie te evalueren met een bepaalde precisie, in dit geval 10 decimalen. *Rationalize* benadert dit zo nauwkeurig mogelijk (optie *0).* Men kan met de rekenmachine nagaan dat dit een goede benadering is. Meer decimalen waren overigens ook opvraagbaar geweest:

```
In[3]:= N[Pi, 100]
Out[3] = 3.14159265358979323846264338327950288419716939937510582097494459230781640\
      6286208998628034825342117068
```

Een ander voorbeeld is voorbeeld $\Gamma(z) = \int_o^\infty t^{z-1} e^{-t} dt$

```
In[1] := N[Gamma[1/7], 30]
6.54806294024782443771409334 94
```

Maar

```
In[2] := N[Gamma[.142857], 30]
6.548069828798543
```

zal maar een standaard aantal decimalen geven omdat het argument (.142857) van de gamma functie niet met voldoende nauwkeurigheid gegeven wordt, immers 1/7 is in principe oneindig nauwkeurig uit te rekenen. Anders geformuleerd: Mathematica maakt onderscheid tussen rationele getallen en reële getallen.
Bij Fortran90 kan men op een vergelijkbare manier vragen om een representatie met een minimaal aantal significante decimalen (*select_real_kind)*. Fortran77 en C staan geen specificatie toe van het aantal decimalen, het gedeclareerde type dient door de gebruiker zelf vertaald te worden in een nauwkeurigheid, terwijl er maar beperkte mogelijkheden zijn.

Teneinde een en ander te begrijpen is het noodzakelijk om iets dieper in te gaan op de representatie van de niet-gehele getallen. In het algemeen kan een floating point getal genoteerd worden als

$$x = mr^e \tag{6}$$

waarin m de mantisse is,
r de radix van de notatie,
e de exponent.

De in (6) gedefinieerde notatie is *niet* uniek immers $x = mr^e = (m/r)r^{e+1} = x$. Uniciteit kan gewaarborgd worden door te eisen dat de floating point representatie *genormaliseerd* wordt, i.e.

$$\frac{1}{r} \le |m| < 1 \tag{7}$$

Deze normalisatie eis is echter niet van invloed op het getal 0, zodat daar een aparte representatie voor gekozen moet worden.
Voor dataverwerkende systemen is een radix r van 2 standaard, wij zullen de afleiding echter geven voor het algemene geval van r. Zijn er voor de notatie van de mantisse N_m posities m_i gereserveerd dan geldt

$$0 \le m_i \le r - 1 \qquad i = 0, ..,(N_m - 1) \tag{8}$$

De waarde m van de mantisse wordt dan gegeven door

$$m = r^{-1} + \sum_{i=0}^{N_m - 1} m_i r^{-i-2} \tag{9}$$

Hierbij wordt uitgegaan van een bitnummering van links naar rechts.
In geval van een genormaliseerde floating point gelden voor de mantisse de volgende relaties:

$$max|m| = r^{-1} + \sum_{i=0}^{N_m - 1} \frac{max(m_i)}{r^{i+2}} = r^{-1} + \frac{r-1}{r^2} \sum_{i=0}^{N_m - 1} (r^{-1})^i = 1 - r^{-N_m - 1} \tag{10}$$

$$min|m| = r^{-1} \tag{11}$$

Voorts, als geldt dat e_{max} de grootste toegestane waarde van de exponent is en e_{min} de kleinste toegestane waarde is van de exponent, dan geldt dat

$$max|x| = r^{e_{max}}(1 - r^{-N_m - 1}) \tag{12}$$

$$min|x| = r^{(e_{min} - 1)} \tag{13}$$

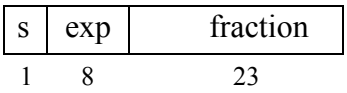
Vrijwel alle systemen conformeren zich aan de zogenaamde IEEE 754 floating point standaard. In Figuur 6 wordt een overzicht gegeven van het bereik van deze standaard
Een mogelijke machinerepresentatie is gegeven in Figuur 7 . De afbeelding van 'single' en 'double' precision ANSI grootheden op de elementen van de taal C is niet vastgelegd maar hangt samen met de woordgrootte van de computer waarop men werkt. De meeste mini systemen, zoals de IBM RS/6000 en de SUN,

ANSI/IEEE Std 754-1985 standaard

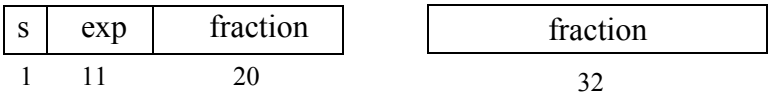
Formaat	*Data lengte*	*Implementatie*
Single-precision	32-bits	verplicht
Single-precision extended	> 42 bits	aanbevolen
Double-precision	64-bits	aanbevolen
Double-precision extended	> 78 bits	aanbevolen

Figuur 6 *Definitie van de lengte van representaties volgens de ANSI/IEEE std 754-1985 standaard voor floating points.*

Single precision floating-point



Double precision floating-point



Figuur 7 *Schematische representaties van single en double precision floating point formaten volgens de ANSI/IEEE Std 754 1985 standaard.*

hebben een 32 bits woord, een supercomputer als de Cray daarentegen heeft een 64-bits woord. Dat betekent dat *float f* op de Cray een 64-bits eenheid declareert en op de SUN een 32 bits eenheid. Het zal duidelijk zijn

Floating point parameters

	Single	*Double*
e_{max}	+127	+1023
e_{min}	-126	-1022
exponent bias	+127	+1023
exponent width in bits	8	11
format width in bits	32	64
Maximum positive value	$3.4*10^{38}$	$1.8*10^{308}$
Minimum positive value	$1.4*10^{-45}$	$4.9*10^{-324}$

Figuur 8 *Floating point format parameters voor ANSI/IEEE Std 754-1985 standaard*

dat dit tot verwarring kan leiden en het werken met een gebruiker gespecificeerde nauwkeurigheid veel logischer is. Dit is in talen als C niet mogelijk, Mathematica staat het wel toe. Veel problemen kunnen voorkomen worden door gebruik te maken in C van include files, zoals <*values.h*>[3], waarin systeemparameters als nauwkeurigheid en grootte van de representatie gedefinieerd staan.

Het eindige aantal bits dat men ter beschikking heeft voor het representeren van niet gehele getallen betekent dat men altijd werkt met een benadering. De eindigheid van de reeks introduceert een benaderings-

[3] Dit file is specifiek voor de SUN implementatie van Unix.

fout. Laat $\hat{x}$ de benadering zijn van de te representeren waarde x en laat voorts de mantisse van de representatie N_m bits groot zijn. Dan geldt

$$|m - \hat{m}| \leq \frac{1}{2} r^{-N_m - 1} \tag{14}$$

Derhalve wordt de relatieve fout in de representatie gegeven door

$$\frac{|\hat{x} - x|}{|x|} = \frac{|\hat{m} - m|}{|m|} \leq \frac{1}{2} r^{-N_m} \tag{15}$$

aangezien geldt dat $1/r \leq |m| < 1$.
De *grootte* van de relatieve fout hangt samen met de (hardware) representatie, d.w.z. het aantal bits dat voor de mantisse gereserveerd is. In dit verband is men meestal geneigd om de grootheid r^{1-N_m} de machine *nauwkeurigheid* ε_m te noemen. Voor een 32-bits machine met IEEE-754 floating point betekent dit dat in enkelvoudige precisie er (slechts) met zes significante decimalen gewerkt wordt, terwijl in dubbele precisie er 14 decimalen beschikbaar zijn.

De onnauwkeurigheid die is geïntroduceerd met de representatie zal zich ook voortplanten tijdens bewerkingen. Hij kan dramatische proporties aannemen bij het aftrekken van twee getallen. Met name als beide getallen ongeveer gelijk aan elkaar zijn. Dit is als volgt in te zien.

Zij x_1 en x_2 twee niet gehele getallen, en zij $\hat{x}_1$ en $\hat{x}_2$ hun benaderde representatie. Dan geldt

$$\hat{x}_1 = x_1(1 + \varepsilon_1) \tag{16}$$

$$\text{en} \quad \hat{x}_2 = x_2(1 + \varepsilon_2) \tag{17}$$

waarbij geldt dat $|\varepsilon_i| \leq \varepsilon_m$. Dan geldt:

$$\Delta\hat{x} = \hat{x}_1 - \hat{x}_2 = x_1(1 + \varepsilon_1) - x_2(1 + \varepsilon_2) = (x_1 - x_2)(1 + \eta) \tag{18}$$

De relatieve fout in $\Delta\hat{x}$ wordt dan gegeven door

$$\eta = \frac{|\varepsilon_1 x_1 - \varepsilon_2 x_2|}{|x_1 - x_2|} \tag{19}$$

$$= \frac{|\varepsilon_2(x_1 - x_2) + x_1(\varepsilon_1 - \varepsilon_2)|}{|x_1 - x_2|} \tag{20}$$

$$\leq \varepsilon_m \left(1 + 2\frac{|x_1|}{|x_1 - x_2|} \right) \tag{21}$$

Uit (21) blijkt dat als $|x_1 - x_2|$ klein is ten opzichte van $|x_1|$ dat wil zeggen $|x_1| \approx |x_2|$ de fout in $\Delta\hat{x}$ *niet* beperkt is tot de orde ε_m. Het dient benadrukt te worden dat dit *niet* komt door de fout in het aftrekken van $\hat{x}_1$ en $\hat{x}_2$ maar veeleer door de initiële fouten die geïntroduceerd zijn bij het afronden en manifest worden doordat een aantal significante decimalen wegvallen. Dit wordt *cancellation error* genoemd.

2.2.5 Voorbeeld: Numerieke bepaling van afgeleide

Als illustratie van de beperkte nauwkeurigheid waarmee gerekend kan worden op computers kan het probleem van het numeriek bepalen van een afgeleide dienen[4]. Zij $f(x)$ een functie dan geldt volgens het theorema van Taylor

$$f(x+h) = f(x) + hf'(x) + \frac{1}{2}h^2 f''(x) + \frac{1}{6}h^3 f'''(x) + ... \tag{22}$$

Aannemende dat h klein is levert de bekende benadering voor de afgeleide

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \tag{23}$$

Enige overweging maakt duidelijk dat het niet eenvoudig is om een juiste waarde voor h te kiezen: wordt hij te klein gekozen dan zal door de cancellation error het resultaat onbetrouwbaar worden terwijl een te grote waarde van h de betrouwbaarheid van de eerste orde benadering discutabel maakt. Dit kan men als volgt inzien.
De mathematische fout, geïntroduceerd door de eerste orde benadering, is ongeveer gelijk aan (gebruik vergelijking (22))

$$hf'(x) = f(x+h) - f(x) - \frac{1}{2}h^2 f''(x) - \frac{1}{6}h^3 f'''(x) + \ldots \tag{24}$$

$$f'(x) = \frac{f(x+h) - f(x)}{h} - \frac{1}{2}hf''(x) - \frac{1}{6}h^2 f'''(x) + \ldots \tag{25}$$

Voor kleine h is de factor 1/2 niet van belang en ook de volgende term met h^2 is dan verwaarloosbaar. Voor een fout nemen we altijd de absolute waarde:

$$\left| f'(x) - \frac{f(x+h) - f(x)}{h} \right| = \left| -\frac{1}{2}hf''(x) - \frac{1}{6}h^2 f'''(x) + \ldots \right| \tag{26}$$

$$e_t \approx |hf''(x)| \tag{27}$$

De mathematische fout, geïntroduceerd door de eerste orde benadering, is dus evenredig met h, waarbij de evenredigheidsfactor afhangt van de tweede afgeleide $f''(x)$.
De afrondingsfout die gemaakt wordt in de juiste representatie van het differentiequotiënt is ongeveer gelijk aan

$$e_r \approx \varepsilon_f \left| \frac{f(x)}{h} \right| \tag{28}$$

waarbij ε_f de machine nauwkeurigheid voorstelt. De optimale waarde van h wordt dan eenvoudig bepaald door te stellen dat

$$\frac{\partial}{\partial h}(\varepsilon_t + \varepsilon_r) = 0 \tag{29}$$

[4] Press, W.H., Teukolsky, S.A. (1991) Computers in Physics 5,68-69.

Invullen van de voorgaande benaderingsformules en uitwerken levert

$$\frac{\partial}{\partial h}\left(h|f''(x)| + \frac{\varepsilon_f|f(x)|}{h}\right) = |f'(x)| - \frac{\varepsilon_f|f(x)|}{h^2} = 0 \tag{30}$$

wat resulteert in

$$h \sim \sqrt{\varepsilon_f(f/f')} \tag{31}$$

Het quotiënt $(f/f')^{1/2}$ noemt men meestal de 'curvature scale'. Bij gebrek aan meer informatie wordt deze grootheid geëvalueerd op het punt x. Met deze keuze van *h* wordt de totale minimale relatieve fout in de afgeleide gegeven door

$$\varepsilon_{total}/|f'| \sim \sqrt{\varepsilon_f}((ff')/f'^2)^{1/2} \sim \sqrt{\varepsilon_f} \tag{32}$$

Voor een 32-bits machine waarop in enkelvoudige precisie gewerkt wordt, d.w.z. met 6 significante decimalen, betekent dit dat met een optimale keuze van *h* dit differentiequotiënt een nauwkeurigheid van .1% in de bepaling van de afgeleide toelaat.
Indien een hogere nauwkeurigheid vereist is moet men bijvoorbeeld een hogere orde benadering gebruiken. Dit is eenvoudig in te zien door meer termen van de Taylor reeks mee te nemen:

$$f(x+h) = f(x) + hf'(x) + \frac{1}{2}h^2f''(x) + \frac{1}{6}h^3f'''(x) + ... \tag{33}$$

$$f(x-h) = f(x) - hf'(x) + \frac{1}{2}h^2f''(x) - \frac{1}{6}h^3f'''(x) + ... \tag{34}$$

Wanneer vergelijking (34) wordt afgetrokken van vergelijking (33), en de termen hoger dan derde orde worden verwaarloosd, vallen de tweede orde termen tegen elkaar weg, en dat geeft de benadering

$$f(x+h) - f(x-h) = 2hf'(x) + \frac{1}{3}h^3f'''(x) \tag{35}$$

Links en rechts delen door *(2h)* geeft dan

$$f'(x) \approx (f(x+h) - f(x-h))/(2h) \tag{36}$$

waarbij een mathematische fout gemaakt wordt van ongeveer $\varepsilon_t \sim h^2f'''(x)$. Een soortgelijke afleiding als hierboven gegeven

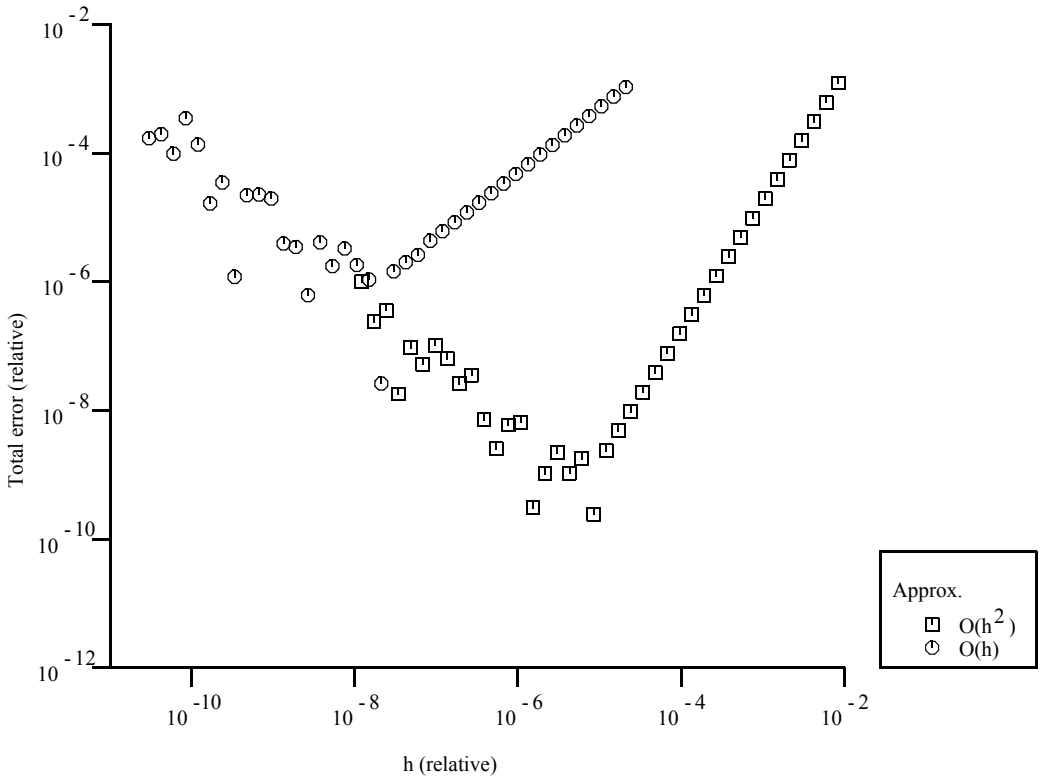
$$\frac{\partial}{\partial h}\left(h^2|f'''(x)| + \frac{\varepsilon_f|f(x)|}{h}\right) = 2h|f'''(x)| - \frac{\varepsilon_f|f(x)|}{h^2} = 0 \tag{37}$$

zal dan laten zien dat

$$h \sim (\varepsilon_f(f/f''))^{1/3} \sim (\varepsilon_f)^{1/3}x_c \tag{38}$$

waardoor de totale fout gegeven wordt door

$$\varepsilon_{total}/|f'| \sim \varepsilon_f^{2/3} \tag{39}$$



Figuur 9 *Afhankelijkheid van de totale fout die in de berekening van een numerieke afgeleide gemaakt wordt als functie van de waarde van h voor eerste (cirkels) en tweede orde (vierkanten) benadering.*

In Figuur 9 is grafisch het resultaat weergegeven voor een berekening van een afgeleide met een 64-bits woord corresponderend met ongeveer 15 decimalen. Duidelijk zichtbaar is de optimale waarde van *h* voor de bepaling van de afgeleide, en het verschuiven van deze waarde als men van een eerste orde naar een tweede orde benadering gaat. Het feit dat de linker flank van de grafiek veroorzaakt wordt door de cancellation van significante decimalen kan bijvoorbeeld met Mathematica goed aangetoond worden.

Een kanttekening: bij het bepalen van de afgeleide van een gemeten curve spelen ook de meetfouten een rol. De datapunten worden eerst gefit met een analytische functie (bv. polynoom) waarvan vervolgens de afgeleide wordt uitgerekend.

2.2.6 Rekenen in perspectief

De twee benadering van formele talen (Mathematica) en derde generatie talen (C, Fortran e.d.) zijn wezenlijk verschillend. Bij de eerste wordt het moment van numerieke representatie zo lang mogelijk vermeden en dientengevolge veel exacter gewerkt. Bij de derde generatie gaat het puur om een approximatie van het antwoord. In 'double precision' komt dat overeen met ongeveer 15 significante decimalen, voldoende voor de meeste, maar niet alle toepassingen. De cancellation error die in de vorige paragraaf aangegeven is kan in principe roet in het eten gooien.
Een verhoging van de nauwkeurigheid van representatie is niet logisch noodzakelijk een oplossing. Ten eerste moeten de betrokken parameters in voldoende nauwkeurigheid bekend zijn, ten tweede moet er

ondersteuning zijn voor het formaat. In toenemende mate moet er dus bij het manipuleren van formules nagedacht wordt of het gaat om numerieke, analytische of gemengde problemen.

2.3 Datastructuren

2.3.1 Introductie

Gebruikergedefinieerde groeperingen van datatypen noemt men datastructuren. De reden om te komen tot een dergelijke groepering is dat voor een specifieke toepassing een aantal gegevens logisch samenhangen. Als voorbeeld kan men denken aan een spectrum. De parameters die men hiermee kan associëren zijn aangegeven in Figuur 10. In principe is het goed mogelijk om al deze parameters los te definiëren maar een identificatie van een spectrum heeft weinig zin als hij niet onlosmakelijk verbonden is met de data waar het op slaat. Reden dus om deze data bij elkaar te groeperen.

```
                                                typedef  struct    spectrum {
start golflengte, increment golflengte, einde           float     start, step, stop;
golflengte                                              int       npoints;
aantal meetpunten                                       float     *scan;
meetpunten                                              char      *identifier;
identificatie.                                  } SPECTRUM;
```

Figuur 10 Voorbeeld van een gebruiker gedefinieerde datastructuur in de taal C

Een Matlab script waarin een soortgelijke datastructuur wordt gemaakt staat in Figuur 11 . Merk op dat de

```
start=1;
stop=100;
step=1;
times=start:step:stop;
npoints=length(times);
data=exp(-times/50);
noisydata=normrnd(data,0.02);
plot(times,noisydata,'s',times,data);
mytrace=struct('start',start,'step',step,'stop',stop,'npoints',npoints,'scan',noisydata,'identifi-
er','noisy exponential decay');
mytrace.identifier
```

Figuur 11 Matlab script waarin een exponentiële decay wordt gesimuleerd, met additieve ruis. Het resultaat wordt in een eigen datastructuur opgeslagen.

laatste statement gebruikt wordt om de inhoud van een deel van de struct te zien. In dit geval is dat de string 'noisy exponential decay'. In *Mathematica* kan een inhomogene lijst gebruikt worden om samenhangende objecten te groeperen, zie Figuur 12 . Merk op dat hier het zesde element uit de lijst gebruikt wordt als label voor de plot.

Een tweede (wiskundig) voorbeeld van een datastructuur is overbekend, namelijk dat van de matrices en vectoren. De coordinaten van een punt of de elementen van een matrix horen logisch bij elkaar en dienen als zodanig ook gegroepeerd te kunnen worden in een omgeving wil deze bruikbaar zijn om eenvoudig problemen op te kunnen lossen. Dit vereist twee zaken.

- Ten eerste dient er een manier te zijn om data te kunnen structureren. Hoewel dit voor de handliggend klinkt is in Fortran77 een dergelijke mogelijkheid bijvoorbeeld afwezig.
- Ten tweede dient er een datatype aanwezig te zijn dat kan wijzen naar een dergelijke structuur, oftewel

```
start=1;
stop=100;
step=0.5;
times=Range[start,stop,step];
data=Exp[-times/50];
npoints=Length[times];
noisydata=data+RandomReal[NormalDistribution[0,0.02],npoints];
mytrace={start,stop,step,npoints,noisydata,"noisy exponential decay"};
p1=Plot[Exp[-t/50],{t,start,stop},PlotRange->{0,1},PlotLabel->mytrace[[6]]];
p2=ListPlot[Transpose[{times,noisydata}]];
Show[p1,p2]
```

Figuur 12 Mathematica script waarin een exponentiële decay wordt gesimuleerd, met additieve ruis. Het resultaat wordt in een eigen datastructuur, een inhomogene lijst, opgeslagen.

de omgeving dient een pointer te ondersteunen.

Dit tweede punt is fundamenteel om goed te begrijpen. Data, van wat voor een vorm dan ook, worden in de computer opgeslagen en corresponderen dus met een bepaalde hoeveelheid geheugen. Wanneer meerdere procedures deze data nodig hebben zijn er twee mogelijkheden: er kunnen evenveel copieën gemaakt worden als er verzoeken zijn om de data te hebben en diegenen die de data nodig hebben kunnen de plaats door krijgen waar de data staan. Ter illustratie (het gaat hier niet om details van de taal C) is een C vertaling aangegeven in Figuur 13. In het linkergeval wordt een copie meegegeven. Algemeen refereert men naar

```
void    foo(float x)
{
        x = sqrt(x);
}

main()
{
        float    val = 20.;

        foo(val);
}
```

```
void    foo(float *x)
{
        *x = sqrt(*x);
}

main()
{
        float    val = 20.;

        foo(&val);
}
```

*Figuur 13 Verschil in C van zogenaamde argument passing. In het linker geval wordt aan 'call by value' gedaan, in het rechter geval 'call by name'. De bewerking links binnen foo heeft **geen** effect in het hoofdprogramma main, de bewerking rechts binnen foo heeft **wel** effect in het hoofdprogramma main!*

deze methode als *call by value*. In het rechter geval wordt de plaats waar de waarde opgeslagen ligt meegegeven dit noemt men *call by name*. De wijzer naar de plek wordt aangeduid met de naam *pointer.* Men zou geneigd kunnen zijn om een pointer te identificeren met een fysiek adres van een geheugenplaats maar dat is niet correct. De verschillende omgevingen die wij tot nu toe de revue hebben laten passeren hebben verschillende implementaties van pointers: bij Mathematica zijn ze eigenlijk *niet* manifest aanwezig, in C is de standaard overdracht bij functies in de vorm van *call by value*, terwijl in Fortran77 alles volgens *call by name* overgedragen wordt. Een tweede voordeel van het werken met pointers is de efficiency. In plaats van alle data te moeten copiëren volstaat het nu om één variabele, de pointer, te laten passeren. In Matlab zijn variabelen binnen een functie (net als in C) lokaal, tenzij ze expliciet als global zijn gedeclareerd in het command window en in de functie. De output variabelen staan in een lijst voor de naam van de functie, de input variabelen zitten in de argument list van de functie. In Mathematica worden de variabelen die lokaal zijn expliciet opgesomd in de lijst die het eerste argument is van Module[{},expr], alle andere variabelen zijn globaal. De output van Module is de laatste expressie. Enkele voorbeelden met pseudocode vind je in

Figuur 14 .

```
function out=doeiets(in)
tmp=iets met in
out is iets met tmp

function [out1,out2]=doeiets2(in1,in2,in3)
global teller
tmp=iets met in2
while test iets met in1
        teller=teller+1
        out1 is iets met tmp,in2,in3
end
out2 is iets met out1 en in1
```

```
f[in_]:=Module[{tmp},tmp=expr met in;
out=expr met tmp;out]

f2[in1_,in2_]:=Module[{tmp}, tmp=expr met
in2;out1=expr met tmp;
While[test iets met in1,
teller=teller+1;out1 is iets met tmp,in2];
{out1,out2}]
```

Figuur 14 Voorbeelden van functies in Matlab (links) en Mathematica (rechts) met input en output argumenten, locale variabelen (genaamd tmp) en globale variabelen (genaamd teller).

Voorts karakteriseert men een datastructuur meestal naar de manier waarop de grootte vastgelegd moet worden als: *statisch*, *semi-statisch* en *dynamisch.*. De eerste kwalificatie betekent dat de grootte van de structuur (in compilatie tijd) vastgelegd is, de laatste kwalificatie betekent dat de structuur vastgelegd wordt tijdens het draaien van de applicatie, terwijl semistatisch betekent dat de gebruiker gesuggereerd wordt dat de structuur dynamisch is. Stilzwijgend hebben we hiervan al gebruik gemaakt in het voorbeeld in Figuur 10. De plaats waar de punten ingevuld moeten worden (*scan)* en de plaats waar de identificatie ingezet moet worden (*identifier)* zijn alleen als pointers gedefinieerd, dat wil zeggen (kunnen) wijzen naar een plaats waarvan de grootte nog niet vastligt. Een Matlab struct en een *Mathematica* lijst zijn beide dynamisch.

2.3.2 Maskering

Een veelvoorkomend probleem in omgevingen is maskering van namen. Wanneer in bv Matlab een variabele de naam *plot* heeft, wordt daardoor de Matlab-functie *plot()* gemaskeerd. In *Mathematica* beginnen namen van functies met een hoofdletter, en als een gebruiker zelf de namen niet met een hoofdletter begint, kan dit probleem niet optreden. Wanneer in *Mathematica* een functie aangeroepen wordt die nog niet beschikbaar is (omdat het package niet is ingelezen met Needs of <<), wordt een object met die naam aangemaakt, en is de functie effectief gemaskeerd wanneer daarna het package alsnog wordt ingelezen. In beide gevallen is het probleem op te lossen door het object met de maskerende naam te verwijderen.

Bouwstenen voor PSE

Datatypen
en
Datastructuren

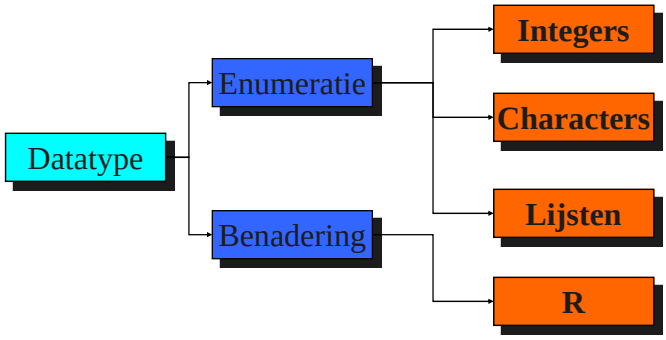
Definitie

- **Datatype**
  Klasse van **dataobjecten** tesamen met operaties om ze te **construeren**, te **manipuleren** en te **verwijderen**.
  Een datatype omvat een specificatie en een implementatie.

- **Specificatie**
  attributen, waarden range, operaties.

- **Implementatie**
  representatie en operatie.

Datatypes



Enige voorbeelden .....

Datatypes

- Getallen stelsels
  binair, octaal, hexadecimaal , ...
  (Mathematica: BaseForm)
- Characters/Lijsten
  ASCII, EBCDIC, willekeurige samenhangende types
- **R**
  Intrinsiek niet representeerbare grootheid.
- HTML
  Tekst, images, geluid, video, ...

Integers

- Gehele getallen: N bits
  - $[-2^{N-1}, 2^{N-1}-1]$
    Let op het interval is niet symmetrisch (2^N +1 combinaties)
- Associatieve en distributieve relaties hoeven niet te gelden:
  - (grootste_positieve_getal + 1) – 2 = ongedefinieerd
  - (grootste_positieve_getal - 2) + 1 = gedefinieerd
- Attributen:
  - +, -, *, /, ...

Integers (cont)

- C *short, int, long*
  Excel *convert on display*
  JavaScript *one type fits all*
  Mathematica *one type fits all*
  - C
    Geen fout, maar wel een grens aan het grootste getal, daarboven vreemde effecten
    (bv: 32 bits 2^32 –1 mogelijkheden)
  - Mathematica
    schijnbaar niet beperkt in grootte representatie.
  - Excel
    Convert voor display naar type

Characters/Strings

- ASCII character set
  (7 of 8 bits code, maakt verschil)
- C — weinig support behalve macro's
  vage notie van characters
  (<ctype.h> ToLower, IsAscii, ...)
- Excel — Enige support voor strings
- Mathematica — ToCharacterCode["..."]
  StringLength, String....
- JavaScript — Classe String en standaard een
  Flexibele interpretatie van een en ander.

Rationele Getallen
(scaled integers/fixed point)

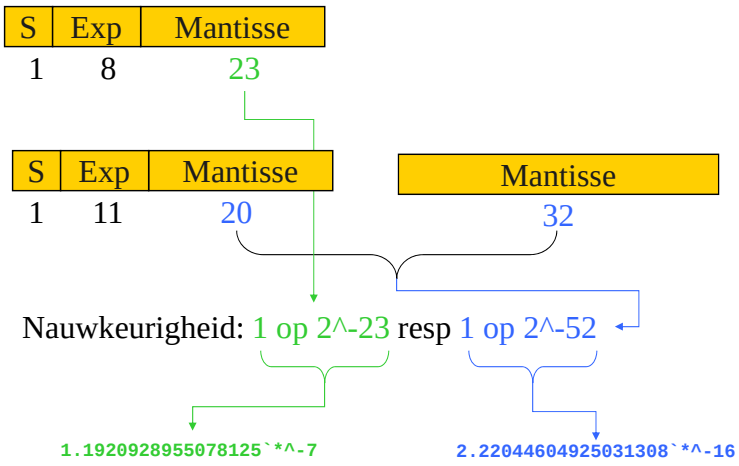
- C — Geen support
  JavaScript — Geen Support
  Mathematica — Intrinsiek
- Voordeel
  Geen afrondingsfouten
- Nadeel
  Schaling is probleem van gebruiker.

Floating Point Getallen

- Floating point getallen
- Altijd een benadering van de echte waarde!
- Notatie $x = m\ r^{e}$

  m = mantisse, r = radix, e = exponent
  Normalisatie: 1/r <= |m| < 1
- Er is een maximale nauwkeurigheid
  (machine nauwkeurigheid, hardware)
- Er is een grootste getal en een kleinste getal.

FP formaten (hardware)

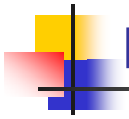


FP karakteristieken

Floating Point Values		
	Single	Double
E_{max}	+127	+1023
E_{min}	-126	-1022
Exponent bias	+127	+1023
Exponent size	8	11
Size (bits)	32	64
Max pos. value	$3.4\ 10^{38}$	$1.8\ 10^{308}$
Min pos. value	$1.4\ 10^{-45}$	$4.9\ 10^{-324}$

Standaardisatie Floating Point

IEEE 754 Standaard		
Formaat	Data Lengte	Implementatie
Single precision	32-bits	Verplicht
Single extended	> 42 bits	Aanbevolen
Double precisie	64 bits	Aanbevolen
Double extended	> 78 bits	Aanbevolen



Mantisse van 23 bits

- Eerste bit 2^{-1}, tweede bit 2^{-2}, ..., laatste bit 2^{-23}
- Vb mantisse van a is
  1111100000 1111100000111
- Dit betekent: $2^{-1}+2^{-2}+2^{-3}+2^{-4}+2^{-5}+2^{-11}+2^{-12}+2^{-13}+2^{-14}+2^{-15}+2^{-21}+2^{-22}+2^{-23}$
- Wat is nu het getal dat in de computer het dichtst bij a ligt ?

Afrondingsfout

- Begrensd in gevallen van:
  optellen (behalve bij getallen met tegengesteld teken), vermenigvuldigen, delen, ...

$$\frac{|\tilde{x} - x|}{|x|} = \frac{|\tilde{m} - m|}{|m|} \leq \frac{1}{2} r^{-N}$$

- Onbegrensd in geval van:
  aftrekken ! (cancellation error)

$$Error = \varepsilon_m \left(1 + \frac{|x_1|}{|x_1 - x_2|}\right)$$

Voorbeeld cancellation error bij mantisse van 23 bits

- mantisse van a is
1111100000 1111100000111
- mantisse van b is
1111100000 1111100000110
- de exponent van b is gelijk aan de exponent van a
- mantisse van a-b is
0000000000 0000000000001
- Hoeveel bits ken je nu van het getal a-b?

Afleiding

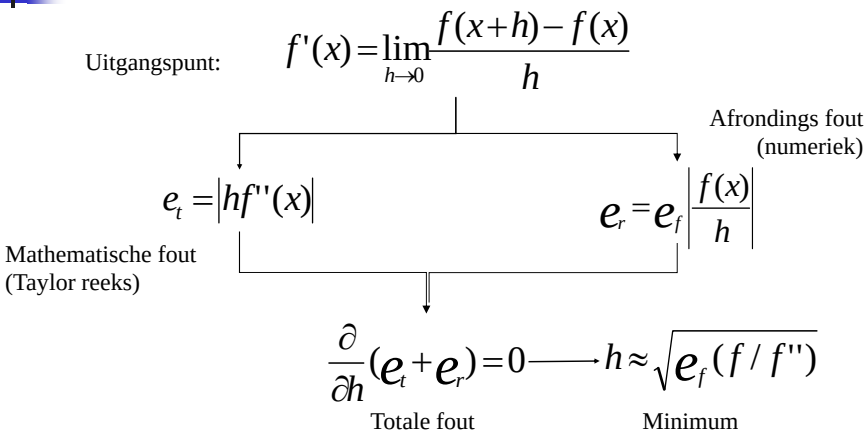
$$\hat{x}_1 = x_1(1+\varepsilon_1)$$
$$\hat{x}_2 = x_2(1+\varepsilon_2)$$
$$\Delta\hat{x} = \hat{x}_1 - \hat{x}_2 = x_1(1+\varepsilon_1) - x_2(1+\varepsilon_2)$$
$$= (x_1 - x_2)(1+\eta)$$
$$\eta = \frac{|\varepsilon_1 x_1 - \varepsilon_2 x_2|}{|x_1 - x_2|} \longrightarrow \eta = \frac{|\varepsilon_2(x_1 - x_2) + x_1(\varepsilon_1 - \varepsilon_2)|}{|x_1 - x_2|}$$
$$\leq \varepsilon\left(1 + 2\frac{|x_1|}{|x_1 - x_2|}\right)$$

Dus:

- **Afronding**
  er bestaat een *b* zodanig dat geldt:
  a + b = a
  maar: dit is een procentuele fout !

- **Cancellation**
  altijd als je het niet kunt gebruiken !

Voorbeeld: afgeleide



Voorbeeld: afgeleide (cont)

$$e_{totaal} / |f'| \approx \sqrt{e_f((ff'')/f'^2)} \approx \sqrt{e_f}$$

- Er bestaat een optimale waarde voor *h*
- Voor single precision maar 3 (!) decimalen nauwkeurig.
- Hogere nauwkeurigheid: betere benadering en/of meer significante digits.

Voorbeeld: afgeleide (3)

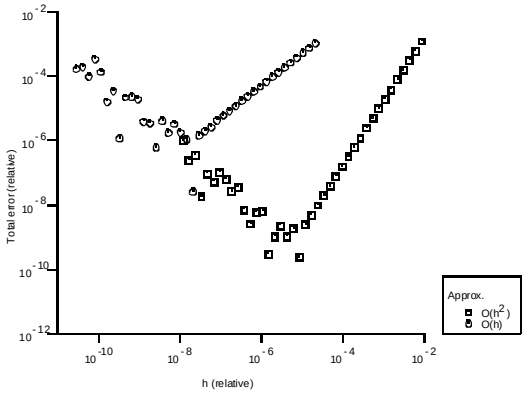
$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2} f''(x) + \frac{h^3}{6} f'''(x) + h.o.$$

$$f(x-h) = f(x) - hf'(x) + \frac{h^2}{2} f''(x) - \frac{h^3}{6} f'''(x) + h.o.$$

$$f(x+h) - f(x-h) = 2hf'(x) + \frac{h^3}{3} f'''(x) + h.o.$$

Conclusie: je wint in mathematische nauwkeurigheid als je meer functiewaarden meeneemt (maar je moet ook meer rekenen)

Voorbeeld: afgeleide (4)



Grafische representatie van nauwkeurigheid

Onderschrift bij figuur: vragen

- Wat staat er langs de assen (grootheden/eenheden)?
- Wat zijn het voor assen (lineair/logaritmisch/...)?
- Wat voor verband(en) zie je in het plaatje (lineair, exponentieel, macht of wortel, ...) ?
- Zitten er bijzondere punten in (minima, maxima, knik, afwijking,gat,...)?
- ...
- Wat is de boodschap of betekenis van het plaatje ?
- Verzin een pakkende titel, en schrijf een onderschrift

Structuren

Data Structuren

- Moeten vertaling van probleem naar algoritme mogelijk maken.
- Kunnen voorkennis van een specifiek probleem veld gebruiken
- Er zijn een aantal ‘universele’ structuren in PSE’s.

- Vector/Array
- (Gelinkte lijst) Lijst
- Stack
- Boom

Vector/Array

- Vertaling van ‘mathematische’ begrip,
- Samenhangend gebied met – uniforme - elementen,
- Basis bouwsteen voor lineaire algebra (routines),
- Speciale klasse voor Arrays met veel nul-elementen (sparse).

Enige implementaties

- C/Fortran

  Alleen numeriek met linearisatie van elementen. Probleem is scheiding van *data* en *specificatie*.

- JavaScript

  Alleen numerieke representaties. Object georiënteerde notie van vectoren en arrays, dus samenhangende *data* en *specificatie.* Automatische schakeling tussen *sparse* en *dense.*

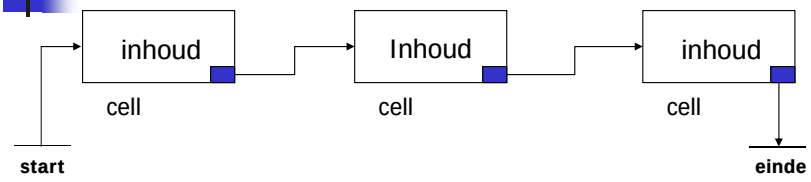
- Mathematica

  Zowel numerieke als symbolische representaties. Ook operaties zijn eveneens numeriek als symbolisch.

- Excel

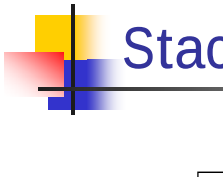
  Zeer beperkte implementatie van het begrip matrix. Alleen numerieke operaties en de grootte moet van te voren bekend zijn

Lijst

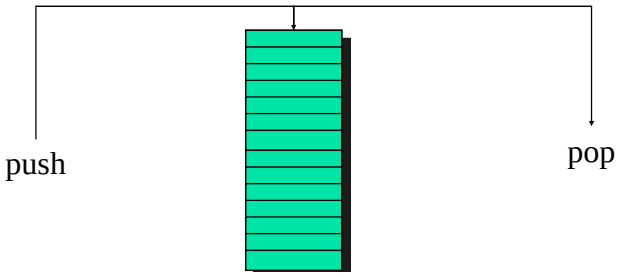


Grafische voorbeeld van een enkelvoudig gelinkte lijst.

- Cellen hoeven niet homogeen te zijn,
- Geheugen allocatie hoeft niet homogeen te zijn,
- Aantal operaties op lijsten zijn heel efficiënt,
- Lijsten zijn een niet-statische structuur,



Stack

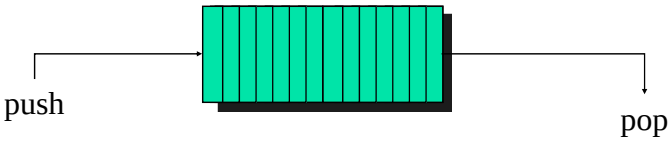


LIFO: Last In First Out Omkering van de volgorde

Stack

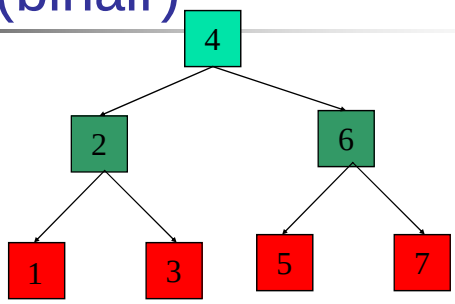
- Gebruik
  - Opslaan van tussentijdse resultaten

Queue



FIFO: First In First Out Behoud van de volgorde

Boom (binair)



.....

Hoofdstuk 3: Algoritmen

3.1 Inleiding

Het woord algoritme is afgeleid van Al-Khwarizmi, een Perzisch wiskundige uit de 9e eeuw, die ook de vader van de algebra wordt genoemd. Onder een algoritme verstaan we een verzameling van rekenregels die dienen om een probleem op te lossen. De 'bouwstenen' waarop een algoritme werkt zijn de datastructuren die we in het vorige hoofdstuk behandeld hebben. Er zijn twee redenen om na te denken over een algoritme. De eerste is als oefening om te zien hoe de oplossing geformuleerd moet worden. De tweede reden is om na te denken over de efficiency.
Het hoeft niet zo te zijn dat voor elk probleem waarvoor men weet dat er een oplossing bestaat er ook een oplossing geformuleerd kan worden. Als voorbeeld kan men denken aan het schaakspel. Volgens de spel theorie gaat het hier om een zogenaamd 'nul som' spel met volledige informatie. In dat soort gevallen moet er een optimale strategie (algoritme) zijn om het spel te spelen. Het vinden van dat algoritme is bepaald problematisch. In het algemeen zijn algoritmen nauw gekoppeld aan de gebruikte datastructuren, we zullen hier nog uitgebreid op terugkomen.

Het idee van algoritmen is oud. Euclides bijvoorbeeld heeft een efficiënte manier bedacht om de grootste gemene deler te kunnen berekenen terwijl Paus Gregorius een aantal regels gegeven heeft voor het vaststellen van schrikkeljaren.

Laten we beginnen met het probleem van het vinden van de 'grootste gemene deler' van twee getallen *u* en *v*. Een voor de hand liggende manier (algoritme) is:

1) kies het minimum van *u* en *v*
2) stel de test voor de *ggd* gelijk aan dit minimum
3) test of zowel *u* als *v* modulo *ggd* nul is, zo ja stop
4) zo nee verminder de *ggd* met één en ga door bij 3.

Een implementatie van dit algoritme in Matlab wordt gegeven in Figuur 1 (a). Het is eenvoudig in te zien,

(a)

```
function t=ggd1(u,v)
t=min(u,v);
while(mod(u,t) || mod(v,t))
      t=t-1;
end
```

(b)

```
function out=ggd(u,v)
if (v==0)
   out=u;
else
   out=ggd(v,mod(u,v));
end
```

Figuur 1 *Niet recursieve en recursieve programmering van het bepalen van de grootste gemene deler in* Matlab.

althans in dit geval, dat het algoritme het correcte antwoord op zal leveren. Voor verschillende argumenten *u* en *v* zal de tijd die het programma nodig heeft om de ggd te vinden verschillen. Een serie metingen is gegeven in tabel 1 op pagina 78, kolom a. De tijd is substantieel voor (redelijk) grote getallen. Deze tabel heeft een tweeledig doel: (i) zien of de gemeten tijden kloppen met de theoretische verwachting en (ii) dienen als basis om tijden te voorspellen voor nieuwe input parameters. Het lijkt in het bovenstaande of er een lineair verband is tussen de grootte van *u* en de tijd die nodig is voor het bepalen van de *ggd*. Theoretisch is het lastig om te voorspellen wat het verband zal zijn omdat dat afhangt van de gevonden waarde voor de *ggd*.

$$t_{ggd} \leq t_{\text{loop}} + 2min(u, v)(t_{\text{mod}} + t_{--}) \quad (1)$$

u	*v*	*time* a	b
1234567890	435268	1.2	$2\ 10^{-6}$
123456789	435268	.9	$2\ 10^{-6}$
12345678	435268	.7	$2\ 10^{-6}$
1234567	435268	.5	$2\ 10^{-6}$
123456	435268	.1	$2\ 10^{-6}$

tabel 1 Efficiency van de algoritmen gegeven in Figuur 1 voor verschillende waarden.

Voor het eerste geval als de *ggd* 1 is kan men dan een bovengrens uitrekenen. Neem voor het *min* even een getal in de grootte orde van 1000000, en bedenk dat het nemen van een modulo al gauw een aantal micro seconden kost en het wordt duidelijk dat dit algoritme weliswaar correcte antwoorden oplevert, maar niet bepaald snel is.

Gelukkig bestaat er voor dit probleem al zo'n kleine 2000 jaar een efficiënte oplossing in de vorm van het algoritme van Euclides:

Gegeven twee gehele getallen u en v, en gegeven dat u groter is dan v, dan is de grootste gemene deler van u en v hetzelfde als de grootste gemene deler van v en (u-v).

Ook van dit algoritme is het mogelijk om te laten zien dat het correct is, hoewel dat iets meer moeite kost. Het verschil met de vorige methode zit hem met name in de reductie van de mogelijkheden, bij het eerste gaat die lineair, hier met het verschil van de argumenten. Formeel kan men laten zien dat het aantal stappen voor de Euclides benadering ongeveer gegeven wordt door $((12\ln(2))/\pi^2)\ln(v)$
Wat implementatie betreft verschillende de beide procedures: de tweede is gebaseerd op de methode van de *recursie,* de eerste niet. Overigens is de recursie niet noodzakelijk, maar wel eenvoudig te programmeren. Meet men het snelheidsverschil van beide methodes dan vindt men ongeveer een factor 1000000 (zie kolom b van tabel 1) en een andere afhankelijkheid van de input parameters namelijk vrijwel niet! Dus twee functioneel equivalente methodes, immers het antwoord blijft hetzelfde, leveren bij implementatie in dezelfde omgeving (bv Matlab) een snelheidsverschil van 10^6. Dat snelheidsverschil moet weer in het licht geplaatst worden van de factor 2 per 18 maanden uit de wet van Moore, en is dus relevant.

Laten wij tot slot een voorbeeld in iets minder detail bekijken, het zoeken in een verzameling. Neem als voorbeeld het geval waarbij er *N* namen in een bestand staan. Als men een bepaalde naam wil zoeken zullen er gemiddeld *N/2* operaties nodig zijn om het gezochte element te vinden. Als de namen opgeslagen zijn in de vorm van een binaire boom, en wij aannemen dat de boom goed gebalanceerd is, dan reduceert het aantal operaties tot *LogN*. Het nadeel is dat er meer tijd besteed moet worden aan het opbouwen van de verzameling. Voor toepassingen van bijvoorbeeld de PTT, waar de veranderingen klein zijn ten opzichte van het aantal malen dat het bestand geraadpleegd zal worden loont het duidelijk om een geavanceerdere datastructuur te hanteren.

In dit hoofdstuk zullen wij aan de hand van voorbeelden een aantal vaak voorkomende types algoritmen behandelen. Daarbij zullen wij natuurlijk letten op de gevolgde methode maar ook letten op de zaken die de implementatie van een algoritme bepalen:

- de *orde*, dat wil zeggen de relatie tussen de executietijd en de input parameter van het algoritme,
- de *betrouwbaarheid*, dat wil zeggen de vraag of de code gecontroleerd is op correcte werking,
- de *robuustheid* van de code, dat wil zeggen de vraag of de code onder 'alle' omstandigheden een correct antwoord oplevert,

- de *portabiliteit*, dat wil zeggen de vraag hoe systeemspecifiek een bepaald stuk code is
- de *onderhoudbaarheid*, dat wil zeggen de mate waarin noodzakelijke veranderingen aangebracht kunnen worden in de code.

In dit bestek zullen wij ons met name op de orde en in mindere mate op de robuustheid concentreren. In globale zin zij overigens opgemerkt dat aan de hier aangegeven criteria meestal door officiële bibliotheken als ESSL, IMSL, NAG e.d. optimaal voldaan is, terwijl dat voor eigen ontwikkelde software in veel mindere mate geldt. In commerciële PSE's als Matlab en *Mathematica* is uitstekende rekensoftware aanwezig, en zijn al veel typische problemen opgelost met behulp van algemene functies, toolboxes (Matlab) of packages (*Mathematica)*. Public domain PSE's zoals **R** (voor statistiek) of Octave (tegenhanger van Matlab) maken gebruik van public domain bibliotheken zoals LAPACK (Linear Algebra PACKage). De aanbeveling is dan ook om waar mogelijk deze software te gebruiken.

3.2 Orde van een algoritme

De orde, ook wel aangeduid als de complexiteit, van een algoritme definiëren we als de relatie tussen een eigenschap (executietijd, geheugenbeslag) en een karakteristieke parameter van het probleem. Voor de orde heeft men een vaste karakterisatie waarvan wij hier enige elementen zullen toelichten:

Constant In principe is dit het meest optimale gedrag dat men kan hebben: de executie tijd is onafhankelijk van de input parameters.

Log N In vrijwel alle praktische toepassingen is dit de beste benadering van optimaal gedrag. Een voorbeeld van een dergelijke orde is het zoeken van een item in een binaire boom. Voorwaarde voor het halen van de orde is dan wel dat de boom gebalanceerd is. Dit voorbeeld illustreert ook dat voor de orde er een afweging plaats kan vinden van geheugen ruimte versus executie snelheid: het opbouwen van een binaire boom kost meer ruimte dan het neerzetten van de elementen in een (lineair) array. Dit verlies wordt goed gemaakt door de verhoogde zoeksnelheid.
Log N algoritmen vertonen een grote schaalbaarheid dat wil zeggen de karakteristieke parameter *N* kan variëren over een groot gebied.

N Een grote hoeveelheid algoritmen heeft een lineaire karakteristiek. Alle vector operaties, zoals het sommeren van een vector, het uitrekenen van een in-produkt, het bepalen van de mediaan van een verzameling hebben dit lineaire gedrag. Het is het beste wat men kan krijgen indien N inputs verwerkt moeten worden. Binnen de fysica is het uitrekenen van een potentiaal veld een orde N probleem, immers $V(r) = \sum_{i=1}^{N} q_i / r_i$

N log N Deze orde is typerend voor de zogenaamde 'verdeel-en-heers' benadering, dat wil zeggen het opbreken van een probleem in deelproblemen en die naderhand weer combineren. Het klassieke voorbeeld van een *NlogN* afhankelijkheid is de zogenaamde Fast Fourier Transformatie (FFT) . Ook de meeste sorteeroperaties zijn van deze orde.

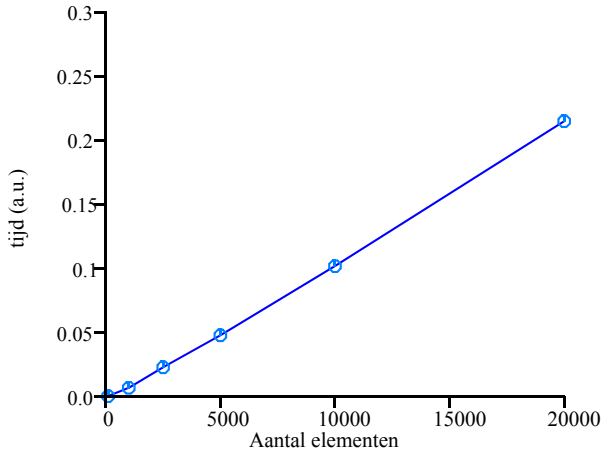
N^2 Alle problemen die betrekking hebben op paarinteracties vertonen een kwadratische afhankelijkheid. De wisselwerking van een deeltje in een elektrisch veld is hier een duidelijk voorbeeld van: $F_i = 1/4\pi\varepsilon_0 \sum_{j=1}^{N} q_i q_j / r_{ij}^2$.
Verdubbelt men het aantal deeltjes dan verviervoudigt (!) de rekentijd. Ook alle matrix/vector bewerkingen - zoals het vermenigvuldigen van een matrix met een vector - vallen in deze klasse. Met de kwadratische afhankelijkheid eindigt ongeveer de klasse van problemen die in praktische gevallen berekenbaar zijn.

N^3 Matrix/matrix operaties vallen onder de derde orde problemen. Daartoe behoren in eerste instantie ook de methoden om een stelsel vergelijkingen Ax=b op te lossen.

en hoger Hogere orde problemen zijn niet interessant om te overwegen aangezien zij bij de minste variatie

uit de grenzen van de berekenbaarheid lopen.

Voor het bepalen van de orde van een algoritme bestaan twee methoden: een theoretische en een praktische. De laatstgenoemde berust eenvoudig op het meten van de executietijd als functie van de grootte van het probleem dat men aanpakt. Beschouw als voorbeeld een sorteerprogramma (de Unix routine *qsort*) dat zo-



Figuur 2 *Tijdsafhankelijkheid van de Unix procedure qsort als functie van het aantal elementen dat aangeboden wordt aan qsort.*

genaamd 'in place' - dat wil zeggen zonder gebruik te maken van een tussenvector - een serie getallen sorteert. Met een programma dat voor een variërend aantal elementen de tijd meet, nodig voor het sorteren van de getallen, is Figuur 2 bepaald. Horizontaal staat de grootte van de te sorteren vector uit, vertikaal een maat voor de tijd nodig voor het sorteren. De orde blijkt lineair te zijn, gemiddeld is 10 microseconde per element nodig. Als men in de literatuur kijkt naar voorspellingen over dit soort algoritmen dan vindt men dat dit lineaire gedrag klopt. De waarde van deze orde bepaling is dat het nu mogelijk is om te voorspellen hoeveel tijd een bepaald probleem gaat kosten.

Voor het geval de source code of een formele representatie van het onderliggende algoritme bekend is kan men door de zogenaamde 'operation count' de orde bepalen. Als voorbeeld kan men denken aan het produkt van twee matrices:

$$a_{ij} = \sum_{k=1}^{N} b_{ik} c_{kj} \qquad (2)$$

De zo gedefinieerde produktmatrix zal N vermenigvuldigingen en N optellingen vergen per element, dus overall een N^3 afhankelijkheid vertonen. Zonder dus een uitspraak te doen over de absolute tijden kan men nu zeggen dat als het probleem 10 maal zo groot wordt, de rekentijd met een factor 1000 zal toenemen. Meestal is een dergelijke toename van de rekentijd prohibitief voor praktische toepassingen.

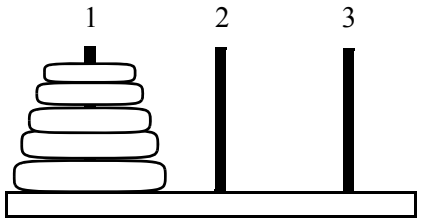
In de rest van dit hoofdstuk zullen wij kijken naar een aantal karakteristieke technieken en algoritmen en die illustreren aan de hand van een aantal applicaties.

3.3 Recursieve algoritmen

Een essentiële bouwsteen voor het implementeren van algoritmes is recursie. Recursie hangt onlosmakelijk samen met de *verdeel-en-heers* strategie. Binnen verdeel-en-heers wordt een probleem opgesplitst in twee of meer deel problemen met dezelfde structuur maar van een lagere orde. De oplossingen van de deel problemen worden vervolgens gecombineerd tot een totale oplossing. Recursie kan ook gedefinieerd worden als het *uitstellen van werk tot een oplossing mogelijk is.* Het klassieke voorbeeld voor het laatste geval is de mathematische definitie van het begrip *n!* als

$$n! = \begin{array}{ll} 1 & n \le 1 \\ n(n-1)! & n \ge 2 \end{array} \qquad (3)$$

De evaluatie van *n!* wordt net zolang uitgesteld tot er een mogelijkheid voor berekening bestaat. Daartoe worden twee regels gehanteerd: een reductie en een terminatie. Het is niet vanzelfsprekend dat herhaald toepassen van de reductie leidt tot een situatie waarin de terminatie gegarandeerd is. Als men in (3) de tweede regel vervangt door *n(n+1)!* zal de reductie voortdurend toegepast kunnen worden maar nooit leiden tot terminatie, en dus nooit een antwoord geven. Voor recursief programmeren is *n!* niet een aansprekend voorbeeld omdat het ook niet recursief opgelost kan worden. Er is echter een klasse van problemen die alleen maar met behulp van recursie tot een goed einde gebracht kunnen worden. De meest notoire vertegenwoordiger hiervan is het spel *de torens van Hanoi*. Het verhaal gaat dat na de schepping van de wereld een koperen plaat met daarop 3 diamanten naalden aan boeddhistische priesters gegeven werd. Op de eerste naald zaten 64 gouden schijven met een afnemende diameter. De priesters hadden de taak om alle schijven van de eerste naar de derde naald te brengen op voorwaarde dat er maar één schijf tegelijk verplaatst werd en dat er nooit een grotere schijf op een kleinere geplaatst werd. Aan de priesters werd verteld dat ze elke dag zo



Figuur 3 Torens van Hanoi met 5 schijven

één schijf mochten verplaatsen en dat als het probleem van de Torens opgelost zou zijn dat het einde zou aankondigen van de wereld.

Laten wij voor het oplossen van het probleem eerst een paar eenvoudige gevallen onderzoeken. Het geval van twee schijven is het meest eenvoudige geval:

- breng de top schijf naar de tijdelijke opslag (2)
- breng de onderste schijf naar de uiteindelijke plek (3)
- breng de top schijf naar de uiteindelijke plek (3)

Voor de generalisatie naar *n* schijven hoeven we alleen de *top schijf* te veranderen in *top*. In plaats van één gaat het dan om *n-1*. Hoe de oplossing werkt voor aantallen groter dan 2 is moeilijk voor te stellen maar het moet volgens dit schema gaan. In Figuur 4 wordt de Matlab code gegeven. De elegantie is hopelijk indrukwekkend. Het is een uitdaging om te proberen om dit probleem niet-recursief op te lossen, door middel van een zelfgebouwde stack data structuur. Daarover kan veel gevonden worden op het internet.
Men zou kunnen concluderen dat recursie een ideale manier van oplossen is voor elk probleem maar dat is ten onrechte want we hebben nog niet gekeken naar de hoeveelheid werk die verricht moet worden. Dat is bij de torens van Hanoi eenvoudig in te zien. Elke aanroep van MoveDisk genereert in principe weer 2 nieu-

```
function MoveDisk(ndisks, from, to, tmp)
if (ndisks > 0)
        MoveDisk(ndisks-1, from, tmp, to);
        sprintf("Move a disk from %d -> %d\n", from, to)
        MoveDisk(ndisks-1, tmp, to, from);
end
```

Figuur 4 *Matlab programma voor het oplossen van het probleem van de torens van Hanoi.*

we aanroepen. Dus voor een hoogste niveau van 64 wordt het totale aantal aanroepen van MoveDisk gegeven door

$$\text{calls} = \sum_{k=0}^{63} 2^k = 2^{64} - 1 \approx 1.6 \times 10^{19} \tag{4}$$

Voor een state-of-the-art computer met een klok frequentie van 1 *ns*, betekent dit dat er grootte orde van 10^{10} seconden gerekend moet worden wat neerkomt op ongeveer 1000 jaar. De monniken die 1 schijf per dag mochten verplaatsen waren dus nog wel even bezig.
Het geheim van recursie is het uitstellen van de oplossing maar wil dat werken dan moet er aan twee voorwaarden voldaan zijn. Ten eerste moet elke aanroep van de recursieve functie *uniek* zijn. Dat geldt triviaal voor de derde generatie talen maar bijvoorbeeld niet voor een aantal Basic interpreters waarin het begrip functie met locale variabelen onbekend is. De tweede voorwaarde is dat er genoeg ruimte moet zijn om alle oplossingen die uitgesteld worden op te slaan. Nemen wij weer het bovenstaande geval van de torens van Hanoi en gaan wij uit van 4 bytes per toestand, dan is er al een astronomische hoeveelheid geheugen nodig om dit probleem op te slaan. Intern zal de computer onveranderlijk recursie afbeelden op een stack. Een tekort aan ruimte voor de recursie zal dan ook de mededeling *'stack overflow'* genereren.

Is recursie altijd de oplossing? Nee, in een aantal gevallen is het voordeliger om een niet-recursieve oplossing te gebruiken. Het eerste voorbeeld is de berekening van de faculteit. Het gebruik van de recursie is hier

```
function out=fac(n)
out=1;
for i=1:n
        out=out*i;
end
```

```
function    out=fac( n)
        if (n > 1)
            out = n*fac(n-1);
        else
            out=1;
        end
```

Figuur 5 *Niet-recursieve en recursieve implementatie van de faculteitsberekening in* Matlab.

volstrekt overbodig omdat er een lineaire call structuur van de functie is. In zo'n geval geeft recursie alleen meer overhead.

Een tweede voorbeeld waarin recursie desastreus is, is het berekenen van een Fibonacci reeks, die gedefinieerd wordt door de recurrente betrekking

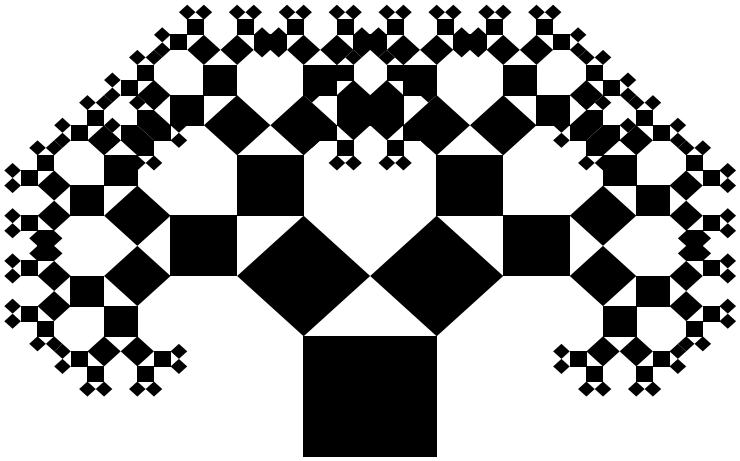
$$F_0 = a \qquad F_1 = b \qquad F_n = F_{n-1} + F_{n-2} \tag{5}$$

Een recursieve implementatie van de Fibonacci reeks is gauw gegeven. De verspilling zit in het feit dat iedere tak zijn eigen evaluatie gaat doen. Bijvoorbeeld voor F_7 is het nodig om F_6 *en* F_5 uit te rekenen. F_6 gaat de aanroep van F_5 en F_4 initiëren maar in de andere tak waren die al een keer berekend. Ook hier hebben

wij weer een exponentiële afhankelijkheid voor de orde maar dan wel met overbodige berekeningen. Dat scheelt astronomisch in de evaluatie snelheid. Voor *n=27* neemt een recursieve implementatie op een PC vele seconden in beslag terwijl de niet-recursieve implementatie in de grootte orde van microseconden zit. De algemene overweging is dat als een probleem een eenvoudige structuur heeft en identieke aanroepen van een functie genereert, evaluatie op een stack dus een recursie, niet echt voordelig is. In andere gevallen is recursie te overwegen.

3.3.1 Boom van Pythagoras

Teneinde recursie in Mathematica te illustreren zullen wij een ander voorbeeld nemen dat meestal aangegeven wordt met de naam *boom van Pythagoras*. Het gaat hier om een recursief figuur dat ontstaat door herhaald op een vierkant twee nieuwe vierkanten te zetten en dit te herhalen zolang de grootte van het vierkant groter is dan een bepaalde drempelwaarde. In Figuur 6 staat een voorbeeld dat het begrip recursief figuur direct duidelijk maakt. Terzijde zij opgemerkt dat dit soort figuren toestaat dat er 'oneindig' veel ingezoomd wordt op plekken omdat er steeds meer detail gegenereerd kan worden. Bij dit figuur is dat overigens niet zo veel interessant nieuw detail. Dergelijke eigenschappen vindt men meer algemeen bij de zogenaamde *fractals*[1]. Een dergelijk figuur is eenvoudig te tekenen met behulp van recursie. Het enige ingrediënt dat men nodig heeft is de mogelijkheid om een vierkant te kunnen tekenen. Daarnaast is er enige basale wiskunde nodig om gegeven de lowerleft coördinaten van een vierkant uit te rekenen waar de coördinaten van de volgende twee vierkanten terecht gaan komen. Bij het algoritme gaan wij van de situatie uit dat de zijde van het vierkant gegeven wordt door een lengte *L* terwijl het vierkant zelf in het algemeen onder



Figuur 6 *Voorbeeld van een recursief figuur: boom van Pythagoras.*

een hoek α staat.

Het Mathematica programma dat voor het genereren van de boom gebruikt kan worden staat in Figuur 7. Voor een goed begrip is het nodig om te weten dat *Module* in Mathematica de mogelijkheid geeft om een functie, in de zin van een derde generatie taal te schrijven. In de eerste lijst, d.w.z. tussen *{* en *}*, worden de namen van de locale variabelen aangegeven. Daarna volgen de aanroepen van de benodigde functies. In het bovenstaande voorbeeld worden twee regels, *Boom*, toegevoegd. De scheiding tussen die twee wordt uitgevoerd op grond van de actuele grootte van *L*, die na de */;* gespecificeerd wordt. Bij de recursieve aanroep wordt de *plot* output opgevangen in drie variabelen die uiteindelijk weer samengevoegd worden in een lijst. Wijs zelf in de figuur de plek van *a, b* en *c* aan. Het uiteindelijke figuur wordt zichtbaar gemaakt door het Mathematica commando *Graphics[Boom[.....]]*. In omgevingen als Mathematica is het gebruikelijk om recursie te programmeren als een verzameling van regels waarvan men het aan de omgeving over laat om die toe te passen. Kritisch punt is daarbij wel, in het geval dat meerdere regels van toepassing zijn, in welke volgorde gekeken wordt hoe de regels toegepast gaan worden. Meestal neemt men daarvoor de volgorde

[1] voor een inleiding zie H. Lauwerier (1990) Een wereld van fractals. Aramith. Lauwerier definieert een fractal als een ingewikkelde meetkundige figuur, waarin men een zekere mate van zelfgelijkvormigheid kan aantreffen.

```
Rect[x_, y_, L_, α_] :=
    Polygon[ {{x, y},
        {x+L Cos[α], y+L Sin[α]},
        {x+L Cos[α]- L Sin[α], y+L Sin[α] + L Cos[α]},
        {x- L Sin[α], y + L Cos[α]},
        {x, y}} ]

Boom[x_, y_, L_ /; L > 2 , α_] :=
    Module[{sa, ca, sa1, ca1, a, b, c}, sa = Sin[α]; ca = Cos[α];
                    sa1= Sin[α+ π/4]; ca1 = Cos[α + π/4];
        a = Boom[x-L*sa, y+L*ca, L/Sqrt[2], α + π/4];
        b = Boom[x-L*sa+L/Sqrt[2]*ca1, y+L*ca+L/Sqrt[2]*sa1,
            L/Sqrt[2], α - π/4];
        c = Rect[x, y, L, α];
        {a, b, c}
    ]

Boom[x_, y_, L_ /; L <= 2 , α_] :=
    Module[ {},  Rect[x, y, L, α]]

Graphics[Boom[0,0,10,0]]
```

Figuur 7 Implementatie van de Boom van Pythagoras in Mathematica

waarin zij ingevoerd zijn.

3.4 Random getallen

In veel toepassingen is het van belang om te kunnen beschikken over willekeurige getallen. De toepassingen zijn legio lopend van de cryptografie, en simulatieprogramma's tot en met spelletjes. Een formele definitie is op zijn plaats. *Een random generator heet uniform in een bepaald interval als alle waarden in dat interval een zelfde waarschijnlijkheid hebben om voor te komen*. Voor de volledigheid: onder een random generator verstaan wij het stuk programma dat de random getallen produceert. Uitgaande van een uniforme random generator kan men ook generatoren met andere verdelingen produceren, bijvoorbeeld Gaussisch door gebruik te maken van de centrale limiet stelling.
Met dergelijke procedures kan men willekeurige systemen nabootsen. Beschouw daarom een uniforme generator op het interval *[0, 1)*. Een binaire gebeurtenis kan men dan nadoen door uit die verdeling een getal te trekken en te testen of het resultaat groter of gelijk is aan 0.5. Meer algemeen, wil men een kans *p* op een bepaalde gebeurtenis hebben dan moet er getest worden of een random trekking tussen, bijvoorbeeld, *[0, p]* ligt.
Toch moet er enig voorbehoud gemaakt worden bij dit begrip random. Informatieverwerkende systemen zijn deterministisch vandaar dat het in principe onmogelijk is om op zulke systemen 'echte' random getallen te genereren. De beste benadering die men kan krijgen is een serie van getallen die pseudo-random zijn: ze lijken willekeurig, maar kunnen op elk willekeurig ogenblik exact gereproduceerd worden!

De meest bekende methode om random getallen te genereren is geïntroduceerd door D. Lehmer en heet de zogenaamde *linear congruential method.* In meer formele zin kan men deze methode noteren als

$$x_{N+1} = (ax_N + c) \bmod m \tag{6}$$

waarin men *m* de module noemt, *a* de multiplier, *c* het increment, en x_0 de startwaarde (of seed). Deze methode zal dus getallen in het interval *[0, m-1]* genereren. Het is eenvoudig te zien dat een dergelijke reeks de-

terministisch is: neemt men dezelfde waarden dan zal men dezelfde serie random getallen krijgen. Voor een gebruiker zijn meestal alle waarden verborgen behalve de eerste waarde die gegeven moet worden. Deze wordt de *seed* genoemd van de reeks. De andere constanten, *a, c* en *m,* zijn meestal onveranderbaar. In vele handboeken kan men de karakteristieken van bepaalde keuzen van deze getallen vinden. In tabel 2 zijn een aantal waarden opgenomen[2]. Bij de meeste random generatoren vindt er nog een conversie plaats van een

Constants for Quick and Dirty Random Generators		
m	**a**	**c**
6075	106	1283
14406	967	3041
29282	419	6173
53125	171	11213
134456	281	28411
233280	9301	49297
714025	4096	150889

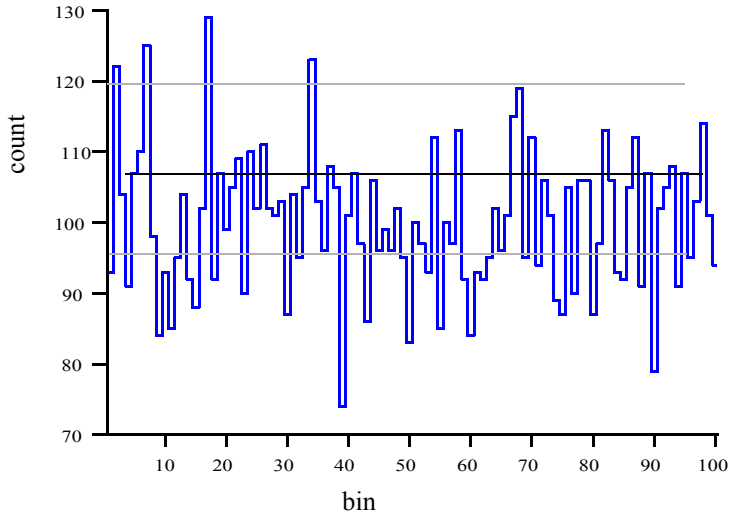
tabel 2Enige startwaarden voor een randomgenerator gedefinieerd in (6)

geheel getal naar een niet geheel getal in het interval *[0, 1]*.
De keuze van de parameters *a, c* en *m* is niet triviaal. Neemt men voor de range *m* 10, en alle andere parameters de waarde 7 dan levert dit de volgende reeks op:

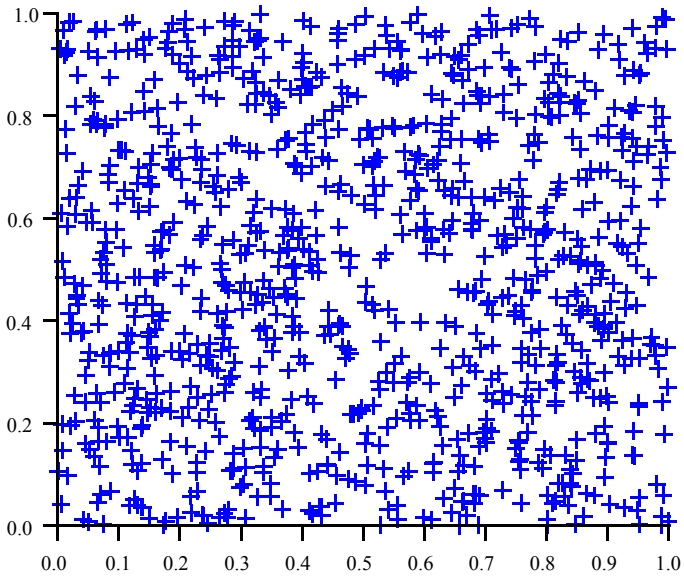
7, 6, 9, 0, 7, 6, 9, 0, ...

Van de 10 mogelijke getallen komen er maar 4 verschillende voor en de reeks herhaalt zich na deze 4. Dit noemt men de periode van een random reeks en formeel definieert men deze als het aantal elementen in de pseudo-random reeks voordat er herhaling optreedt. Aan het bovenstaande geval ziet men dat de periode makkelijk kleiner kan zijn dan de maximaal haalbare. Met name in het werk van Knuth[3] wordt uitgebreid ingegaan op de voorwaarden voor grote periodes. Een goede random generator heeft een lange periode maar dat is zeker niet de enige voorwaarde. Om dat te bedenken neemt men maar bijvoorbeeld *m* 2^31 en *a* en *c* gelijk aan 1. De random reeks zal dan de maximale periode hebben, maar volstrekt voorspelbaar oplopen.
De kwaliteit van een random generator wordt dus bepaald door zijn periode en door de vraag hoe accuraat hij het willekeurig gedrag benadert, de kansverdelingsfunctie. Voor een uniforme random generator op het interval [a, b] verwacht men dat het gemiddelde (a+b)/2 is en dat in een histogram uitgezet, alle bins van het histogram even waarschijnlijk zijn, waarbij de statistisch fout van $\sqrt{N}$ aangenomen wordt. Een voorbeeld van een dergelijke grafiek voor de standaard random generator wordt gegeven in Figuur 8: na 10.000 trekkingen wordt het gemiddelde van 100 redelijk goed benaderd binnen het interval van 10. Voldoende is een dergelijke test overigens niet.
Een mogelijke andere manier om de generator te controleren is door de getallen als pseudocoördinaten te gebruiken op een twee (of drie) dimensionaal vlak. Een voorbeeld hiervan is gegeven in Figuur 9. Aangezien er zich weinig structuur in het vlak aftekent lijkt het gerechtvaardigd om te stellen dat ook wat dit betreft de random generator voldoet. Merk op dat we hier te maken hebben met een toepassing van visualisatie om snel inzicht te krijgen in de (mogelijke) structuur van data, geprint zou het een ondoenlijke zaak zijn om

[2] De tabel is ontleend aan Press et al. Numerical Recipes
[3] The art of computer programming. Zie ook Taylor, C. (1991) Computers in Physics 5, 16-21.



Figuur 8 *Histogram van een uniforme random generator na 10000 samples.*



Figuur 9 *Representatie van de output van een random generator in een twee-dimensionaal vlak.*

deze structuur te onderzoeken.

Dan nog hoeft men niet verzekerd te zijn het juiste gedrag. Om dat te illustreren is het goed om een random generator die in de 70'er jaren op veel systemen gebruikt werkt nader te bekijken.

$$I_{n+1} = 65539 I_n \qquad mod(2^{31}) \tag{7}$$

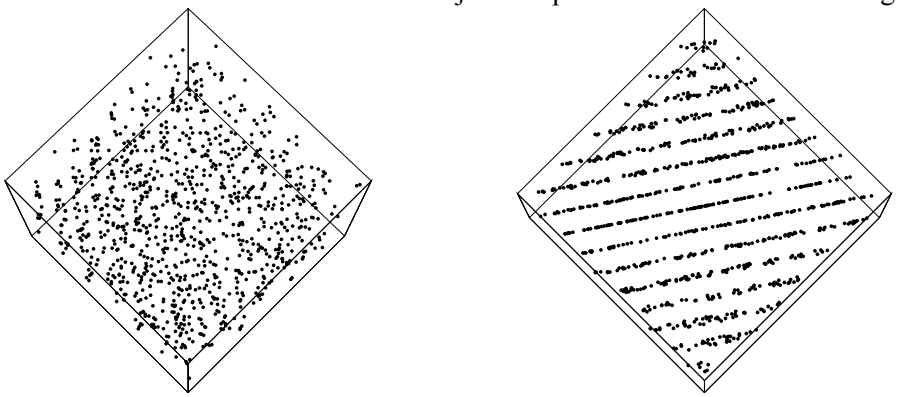
De zo gemaakte generator is weliswaar erg snel maar heeft een duidelijke tekortkoming. Het probleem is dat $65539 = 2^{16} + 3$ zodat geldt dat

$$I_{n+2} = (2^{16} + 3)(2^{16} + 3) I_n \tag{8}$$

wat aanleiding geeft tot de recurrente betrekking

$$I_{n+2} = 6 I_{n+1} - 9 I_n \tag{9}$$

Kortom, als wij de getallen die op deze manier geproduceerd worden in tripletten groeperen dan zullen zij in vlakken liggen. Dat is op zich niet erg want het zal gelden voor elke random generator van dit type. Het probleem ligt meer in het aantal vlakken, dit is hoogstens $\sqrt[3]{m}$. Voor de bovenstaande kan men, letterlijk, laten zien dat het zeer veel kleiner is. Daartoe hebben wij een implementatie van de random generator ge-

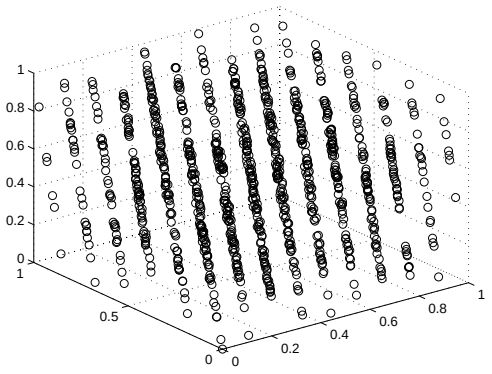


```
Ran[]:=Module[{},MySeed=Mod[MySeed*65539,2^31]]
MySeed=1;
ts = Table[ Table[Ran[], {3}], {1000}];
ListPointPlot3D[ts, BoxRatios -> {1, 1, 1}, ViewPoint -> {1, 1, 6}]
ListPointPlot3D[ts, BoxRatios -> {1, 1, 1}, ViewPoint -> {-1, 1, 15}]
```

Figuur 10 Grafische analyse van random getallen die volgens de lineair congruentiële methode (7) gegenereerd zijn. De viewpoints van de twee opnames zijn in de Mathematica code te lezen.

maakt in Mathematica (de functie *Ran* in Figuur 10) en in Matlab (zie Figuur 11). In Figuur 10 bekijken we vervolgens de tot tripletten geordende getallen vanuit twee verschillende gezichtspunten. Vanuit de willekeurige hoek (1, 1, 6) ziet alles er in orde uit maar vanuit de speciale hoek (-1, 1, 15) (denk aan relatie (9)) ziet men duidelijk het zeer beperkte aantal (ongeveer 16, terwijl $\sqrt[3]{2^{31}} \approx 1290$) vlakken dat er door de random getallen gevormd wordt. Zo'n slechte random generator kan leiden tot artefacten in simulaties!

```
global seed
seed=1;
points=mypoints;
scatter3(points(:,1),points(:,2),points(:,3),'k')
```




```
function points=mypoints
npoints=1000;
points=zeros(npoints,3);
for i=1:npoints
   for j=1:3
         points(i,j)=myran;
   end
end


function out=myran
global seed
seed=mod(seed*65539,2^31);
out=seed/2^31;
```

Figuur 11 Matlab code voor simulatie en grafische analyse van random getallen die volgens de lineair congruentiële methode (7) gegenereerd zijn. Links de code in het command window, rechts de gebruikte functies..

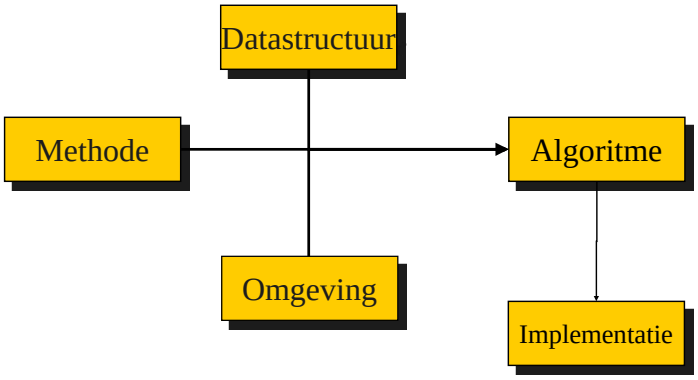
Algoritmen

Loops
Recursie
Monte Carlo

Overzicht

- *Inleiding*
- *Criteria*
- *Algoritmen*
  Recursie

Inleiding



Probleem: vind priemgetallen

- Verzin zelf een strategie, formuleer je eigen algoritme
- Klassiek algoritme:
  zeef van Erathostenes
  http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes

Probleem: vind de grootste gemene deler

- voorbeeld I: 13 en 31
- voorbeeld II: 1001 en 182
- Verzin zelf een strategie, formuleer je eigen algoritme

Voorbeeld

Methode

- Grootste gemene deler (u, v)
  - **kies het minimum van u en v**
  - **stel de test *t* voor de ggd gelijk aan dit minimum**
  - **test of zowel u als v modulo *t* nul is, zo ja stop**
  - **zo nee verminder *t* met één en ga door bij 3**

Matlab

```
function t=ggd1(u,v)
t=min(u,v);
while(mod(u,t) || mod(v,t))
        t=t-1;
end
```

Voorbeeld (2)

Snellere Methode

Grootste gemene deler (u, v)

Gegeven twee gehele getallen u en v, en gegeven dat u groter is dan v, dan is de grootste gemene deler van u en v hetzelfde als de grootste gemene deler van v en (u-v).

Matlab

```
function out=ggd2(u,v)
if (v==0)
   out=u;
else
   out=ggd2(v,mod(u,v));
end
```

Euclides 300 v C

Efficiency ggd

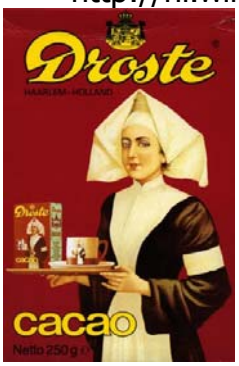
Let wel: implementatie kan een groot verschil maken voor de efficiency.

Vergelijking van de methodes

u	v	Time	
		a	b
1234567890	435268	1.2	$2\ 10^{-6}$
123456789	435268	.9	$2\ 10^{-6}$
12345678	435268	.7	$2\ 10^{-6}$
1234567	435268	.5	$2\ 10^{-6}$
123456	435268	.1	$2\ 10^{-6}$

Recursie

- Functie die zichzelf aanroept
- Droste effect
  http://nl.wikipedia.org/wiki/Droste-effect



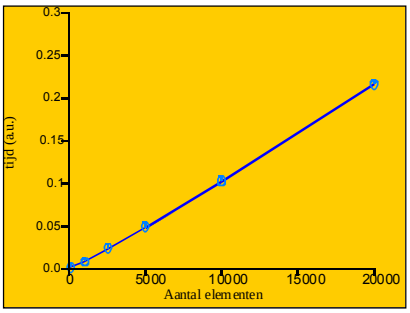
Algoritme

- **de <u>orde</u>**, dat wil zeggen de relatie tussen de executie tijd en de input parameter van het algoritme,
- **de <u>betrouwbaarheid</u>**, dat wil zeggen de vraag of de code gecontroleerd is op correcte werking,
- **de <u>robuustheid</u>** van de code, dat wil zeggen de vraag of de code onder 'alle' omstandigheden een correct antwoord oplevert,
- **de <u>portabiliteit</u>**, dat wil zeggen de vraag hoe systeemspecifiek een bepaald stuk code is
- **de <u>onderhoudbaarheid</u>**, dat wil zeggen de mate waarin noodzakelijke veranderingen aangebracht kunnen worden in de code.

Orde

- *Constant*
- *Log(N)*
- *N*
- *N Log(N)*
- N^2
- N^3
- *hoger*

de relatie tussen de executie tijd
en de input parameter van het algoritme

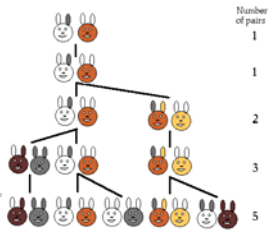


Orde is te meten en kan (soms) theoretisch bepaald worden

Recursieve Algoritmen

Fibonacci
Hanoi
Fractals

Fibonacci



- Suppose a newly-born pair of rabbits, one male, one female, are put in a field. Rabbits are able to mate at the age of one month so that at the end of its second month a female can produce another pair of rabbits. Suppose that our rabbits **never die** and that the female **always** produces one new pair (one male, one female) **every month** from the second month on. The puzzle that Fibonacci posed was...
- How many pairs will there be in one year?

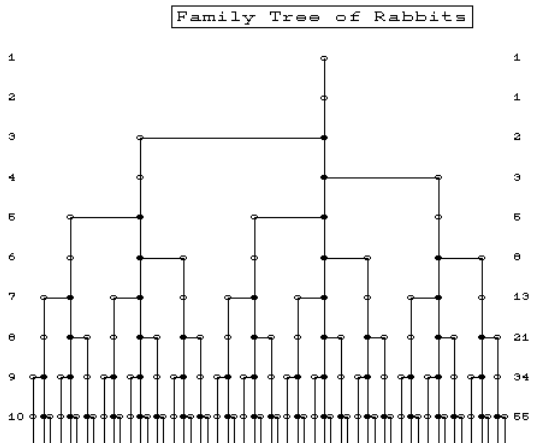
Fibonacci recursief

- Pseudocode (Matlab control structure):

```
function output=fib(n)
if (test voor n die zorgt voor terminatie
                                van de recursie)
        output is iets
else
        output is iets anders (recursie!)
end
```

Fibonacci (orde)

```
Fib[n_] := Fib[n - 1] + Fib[n - 2]
```



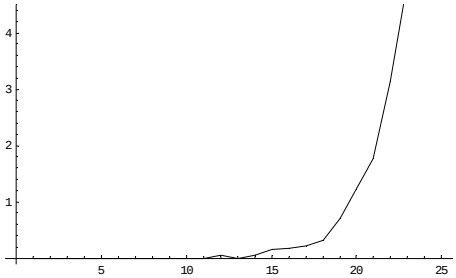
Fibonacci (methode)

- Recursief

```
Fib[n_] := Fib[n - 1] + Fib[n - 2]
```

  - Intuïtief maar niet realistisch in elke programmeer omgeving. Orde is een Fibonacci reeks

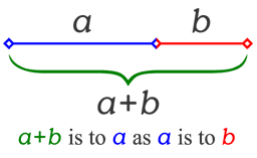
(2, 3, 5, 8, ...)

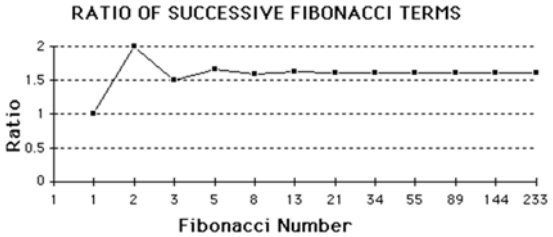


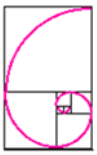
Fibonacci (methode)

- Slim recursief
  - Voorkom opnieuw uitrekenen door tussenresultaten op te slaan
  - Minder intuïtief maar veel sneller alleen het kost meer geheugen !
  - Dan voorbeeld van een constant algoritme!
- Excel
  - Makkelijke implementatie

Fibonacci en de gulden snede







$$\lim_{n\to\infty}\frac{f(n+1)}{f(n)} \to 1.618033989...$$

$$\frac{1}{x} = x - 1$$

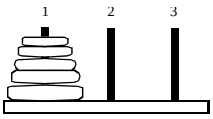
http://en.wikipedia.org/wiki/Golden_ratio

Towers of Hanoi

Definition: Given three posts (towers) and *n* disks of decreasing sizes, move the disks from one post to another one at a time without putting a larger disk on a smaller one. The minimum is 2^n-1 moves.
The "ancient legend" was written in 1884.

http://obelix.ee.duth.gr/~apostolo/TowersOfHanoi/index.html
http://www.nist.gov/dads/HTML/towersOfHanoi.html

Hanoi



- Recursief oplosbaar
  - **breng de top naar de tijdelijke opslag (2)**
  - **breng de onderste schijf naar de uiteindelijke plek (3)**
  - **breng de top naar de uiteindelijke plek (3)**

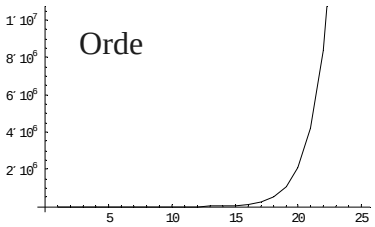
Hanoi (2)

Matlab

```
function hanoi(ndisks,from,to,tmp)
if ndisks>0
    hanoi(ndisks-1,from,tmp,to);
    sprintf('Move disk %c from %d to %d\n', ndisks+64,from,to)
    hanoi(ndisks-1,tmp,to,from);
end
```

Aantal termen voor N schijven

$$\sum_{k=0}^{N} 2^k$$

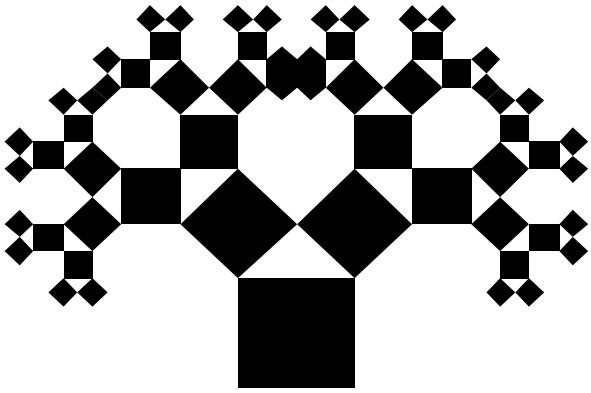


Samenvattend

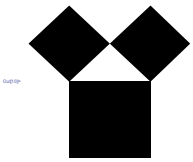
- Recursie
  - Elegante en belangrijke techniek
  - Kan in de performance soms nadelen hebben (denk aan Fibonacci)
  - Deze nadelen kunnen soms opgeheven worden door tussentijdse opslag

Recursieve figuren: Boom van Pythagoras

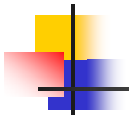
Out[7]=



Basiselement



- Recursief oplosbaar
  - Teken een naar links gekantelde kleinere boom op de linkerbovenhoek (mits groot genoeg)
  - Teken de stam
  - Teken een naar rechts gekantelde kleinere boom op de rechterbovenhoek (mits groot genoeg)

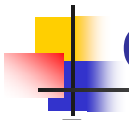


Challenge: Computing for the Life Sciences

- Gaining insight into protein science and protein folding mechanisms
- Simulating (thousands of) macromolecules
- Silicon cells
- …

Interdisciplinary: Mathematics, Physics, Chemistry, Computer Science, Biology

Method: molecular dynamics: integrating Newton's law F=m.a for all atoms where F depends on all atoms! Initialization with the help of random numbers



Counting Pairwise Interactions

- Force between two atoms depends upon their distance
- N atoms, each interacting with all N-1 other atoms, would result in N(N-1) interactions
- But now all interactions were counted twice, so divide by 2
- N(N-1)/2 which is $\approx \frac{1}{2}N^2$
- Order of the algorithm is quadratic (or N^2)

Protein folding

http://www.research.ibm.com/bluegene/
http://researchweb.watson.ibm.com/journal/sj/402/allen.html

Physical time for simulation	10^{-4} s
Typical time-step size	10^{-15} s
Number of MD time steps	10^{11}
Atoms in a typical protein and water simulation	32000
Approximate number of interactions in force calculation	10^{9}
Machine instructions per force calculation	1000
Total number of machine instructions	10^{23}

One year 60*60*24*365 ≈ 3 * 10^{7} seconds
Clock cycle 10^{-9} s
Three years * 10^{6} processors = 3 * 3 * 10^{7} * 10^{6} * 10^{9} = 10^{23} machine instructions

The computational effort required to study protein folding is enormous. Using crude workload estimates for a petaflop/second capacity machine leads to an estimate of three years to simulate 100 microseconds.

Metrische of SI prefixen

milli m 10^{-3}	micro µ 10^{-6}	nano n 10^{-9}	pico p 10^{-12}	femto f 10^{-15}	atto a 10^{-18}	zepto z 10^{-21}
kilo k 10^{3}	mega M 10^{6}	giga G 10^{9}	tera T 10^{12}	peta P 10^{15}	exa E 10^{18}	zetta Z 10^{21}

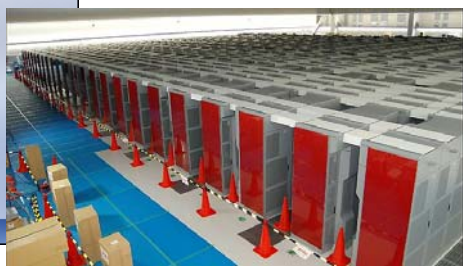
http://nl.wikipedia.org/wiki/SI-prefix

Supercomputers

- Since nov 2009 new #1 Jaguar ORNL, DOE
- Built in 4 years
- Cray XT5
- OS: Linux
- Processor
  AMD x86_64 Opteron Six Core 2600 MHz (10.4 GFlops)
- > 224000 cores
- 1.76 PF
- peak 2.33 PF
- **Nov 2010: China at #1 !**

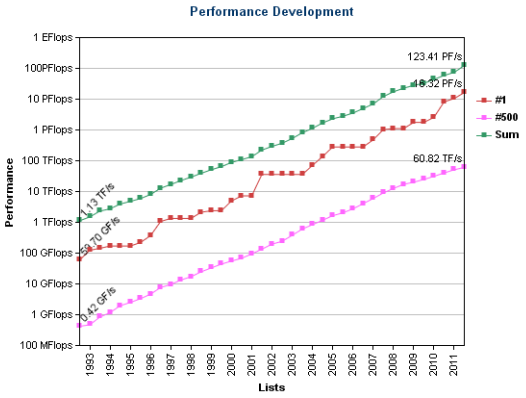
jun 2011: Japan at #1 with 8 PF

K computer
548352 cores
OS: Linux
SPARC64 VIIIfx 2.0GHz
Tofu interconnect



http://en.wikipedia.org/wiki/K_computer

jun 2012: Sequoia de grootste 16 PF



Lawrence Livermore DOE
Berkeley CA
IBM BlueGene/Q
OS: Linux
1572864 cores
Power BQC 16C 1.60 GHz
1572864 GB memory
8 MW power
Green: 2 GFLOPS/W

Supercomputers june 2013: China verdringt opnieuw de USA

- Tianhe-2 Milkyway in Guangzhou (China)
- OS: Linux
- 3120000 cores
- Processor Intel Xeon E5-2692v2 12C 2.2GHz
- 33.8 PF
- 17.8 MW

... looks like 170 large refrigerators and occupies 720 square metres

Monte Carlo

Het lot slaat toe:
Generatie van random getallen

Monte Carlo

- Evalueer het gedrag voor een aantal willekeurige gevallen.
- In de limiet van 'aantal' naar voldoende groot kan het ware gedrag bepaald worden.
- Bv Moleculaire Dynamica, Protein Folding

Monte Carlo (2)

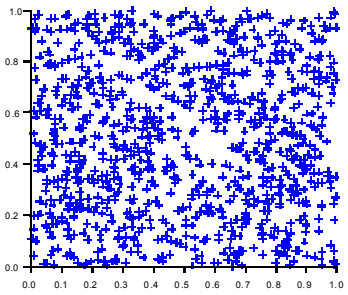
- Voorwaarde: goede random generator

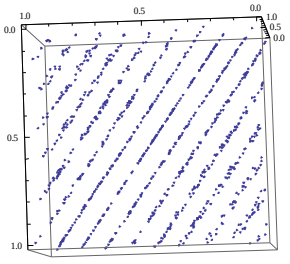
  $x_{N+1} = (a\, x_N + c) \bmod m$

  m bepaalt het bereik, x0 heet de seed
  Let op: a = c = x0 = 7, m = 10

  Geeft: 7, 6, 9, 0, 7, ...

Monte Carlo (2)





Een voorbeeld van een slechte random generator die er in 2 dimensies random uit ziet, maar in 3 dimensies beperkt blijkt tot een klein aantal vlakken

Hoofdstuk 4:Architecturen

4.1 Inleiding en Overzicht

De snelle ontwikkelingen op het gebied van informatieverwerkende systemen zijn al een aantal malen benadrukt in dit dictaat: men dient grote voorzichtigheid te betrachten met het in te veel detail bestuderen van snel verouderende systeemeigenschappen. In dit hoofdstuk zullen wij proberen de gulden middenweg te bewandelen tussen detail en globaal overzicht.

Traditioneel verdeelt men data verwerkende systemen in hardware en software. De wisselwerking tussen hardware en software is aan voortdurende verandering onderhevig. Drijvende kracht hierbij is de afweging tussen snelheid, prijs en flexibiliteit. 'Hardware' oplossingen zijn meestal het snelst, duur en moeilijk te veranderen. Software is meestal trager, vermoedelijk goedkoper[1] en in elk geval eenvoudiger te veranderen. Algoritmen en mogelijkheden die eerst verkend worden in software, worden in een laatste, produktie, fase geïmplementeerd in hardware. Een duidelijk voorbeeld hiervan zijn de zogenaamde 'multi media' instructies in de Ultra Sparc CPU van SUN die het mogelijk maken om real-time image processing te doen. Daarmee vergelijkbaar zijn de MMX (multi media extensies) instructies van Intel processoren.

Als uitgangspunt van de hardware ontwikkelingen wordt meestal het baanbrekende werk van Von Neumann genomen, reden waarom men dan ook vaak spreekt over 'de Von Neumann architectuur'. In dergelijke architecturen is het gebruikelijk om over twee eenheden te spreken 'instructies' en 'data'. Globaal gesproken definiëren de instructies de akties die uitgevoerd moeten worden op de data. Die constatering is het uitgangspunt van de classificatie van *Flynn* die in de 70er jaren opgesteld is en nog steeds gehanteerd wordt. In deze taxonomie onderscheidt men - voor de volledigheid - vier categorieën:

1) *Single Instruction Single Data (SISD)*
   Tot deze groep rekent men al die machines die doen aan zogenaamde seriële (Von Neumann) processing. Zij vormen als het ware het startpunt van de computerontwikkelingen.

2) *Single Instruction Multiple Data (SIMD)*
   Tot deze groep rekent men al die machines die in staat zijn om met 1 instructie een serie data te bewerken. Vaak rekent men de zogenaamde vectormachines tot deze groep.

3) *Multiple Instruction Single Data (MISD*)
   Weinig mensen kunnen zich wat voorstellen bij deze klasse, behalve misschien Flynn zelf, immers wat stellen veel bewerkingen op dezelfde data voor.

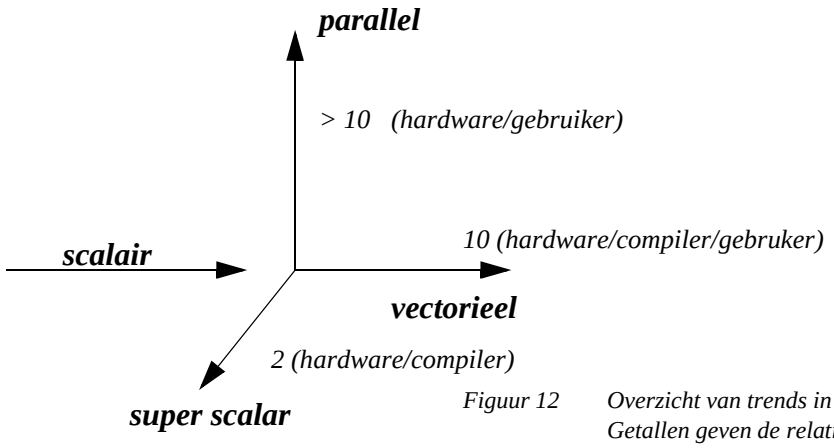
4) *Multiple Instruction Multiple Data (MIMD)*
   Multiple instructies onderstellen dat er meerdere processoren in dezelfde computer zitten die actief kunnen zijn. Vandaar dat men tot deze klasse al de systemen rekent die multiprocessor configuraties omvatten.

De hier aangegeven taxonomie kan men ook op een andere manier in kaart brengen, zoals aangegeven in Figuur 12. Uit het startpunt van de scalaire processoren zijn drie (bijna) orthogonale wegen gekomen: super scalar, vector en parallel. De trends en ontwikkelingen in dit veld zullen wij in de volgende paragrafen in kaart brengen.

4.2 Scalaire architectuur

Als uitgangspunt van de systemen zullen wij de scalaire architectuur nemen, die ook wel aangeduid wordt als de Von Neumann of de SISD. Zoals in de inleiding aangegeven beperken wij ons in eerste instantie tot twee eenheden: *instructies* en *data*, daarbij aard en type van instructies en data in het midden latend.

[1] Overwegend dat één regel foutvrije code $10 kost, is het de vraag of deze bewering helemaal waar is.



Figuur 12 *Overzicht van trends in High Performance Computing. Getallen geven de relatieve prestatie aan t.o.v. scalair.*

De Von Neumann architectuur is opgebouwd rond deze twee eenheden. Globaal onderscheidt men drie componenten:

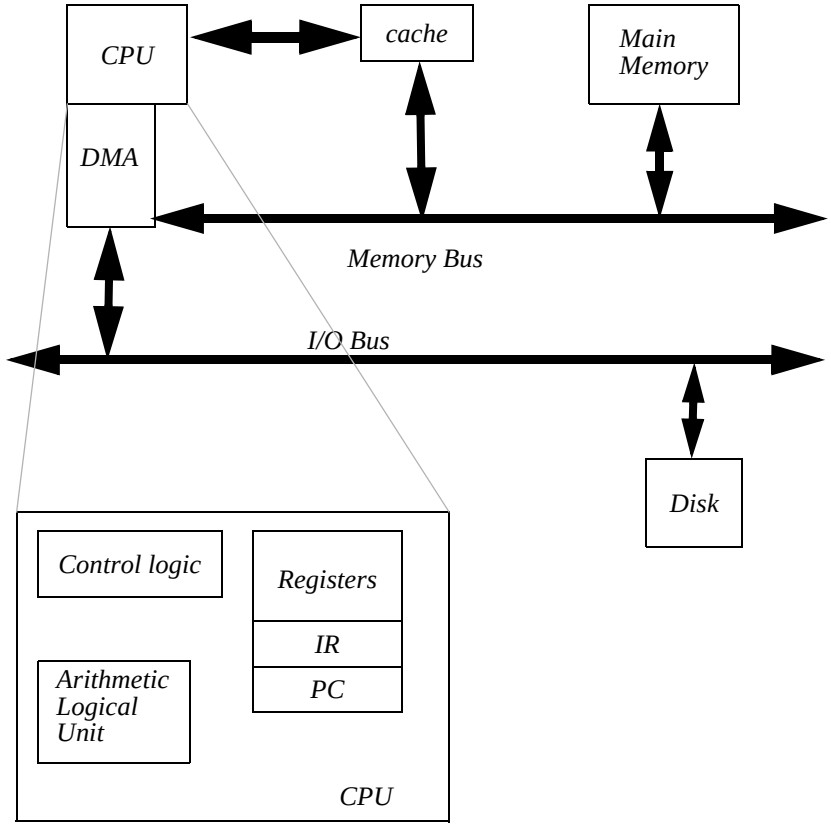
- *memory*, waar de instructies en de data opgeslagen worden,
- *CPU,* waar de instructies op de data toegepast worden en
- *I/O,* verbinding tussen CPU en memory

Gebaseerd op deze indeling doorloopt de klassieke Von Neumann machine een cyclus die uit 5 stappen bestaat, bekend als de zogenaamde Von Neumann cyclus;

1) fetch
   haal de volgende instructie op uit het geheugen
2) *decode*
   bepaal wat gedaan moet worden
3) *operand fetch*
   haal, indien nodig de *data* op waarop de instructie moet werken
4) *execute*
   voer de instructie uit
5) *store*
   schrijf de eventuele resultaten weg.

Drie van de vijf onderdelen van deze cyclus hebben te maken met het ophalen van gegevens uit het geheugen.
Het is belangrijk om twee zaken te benadrukken. Ten eerste worden bij dit schema nog geen aannames gemaakt over de aard en type van de data/instructies. De praktijk met name in de begin jaren van de computer was echter anders: men ging uitsluitend uit van numerieke informatie, data waren getallen en instructies waren rekenkundige instructies.
De tweede overweging die belangrijk is, is dat nergens in de boven aangegeven cyclus aangenomen wordt dat geheugen en CPU zich op dezelfde plaats bevinden, d.w.z. geïntegreerd zijn binnen één fysiek systeem. Dit opent de weg naar genetwerkte systemen.

In eerste instantie kan men het doorlopen van de vijf genoemde stappen zien als het uitvoeren van een instructie. De tijd die hiervoor nodig is, is in principe variabel: 40% van de cyclus is optioneel (stap 3 en 5) terwijl binnen het rekenkundige domein het uitrekenen van een wortel duidelijk complexer zal zijn dan de absolute waarde nemen van een getal. Het zal duidelijk zijn dat deze intrinsieke chaos een computer moeilijk hanteerbaar zou maken. Daarom is de tijd in een computer gediscretiseerd. De minimale tijdstap die te



Figuur 13 Principeschema van een elementair scalair computer systeem

maken is wordt aangeduid als de *cycle tijd*. In een eerste benadering zou men kunnen stellen dat in een dergelijke tijdbestek de eerder genoemde cyclus, ook voor de traagste instructie uitgevoerd zou moeten kunnen worden, of

$$t_{cycle} = t_{fetch} + t_{decode} + t_{operand} + max(t_{xeq}) + t_{store} \quad (10)$$

Naast discretisatie van de tijd is ook discretisatie van de informatie aanwezig in een dergelijk systeem: er is een minimale hoeveelheid informatie die gebruikt wordt. Deze basishoeveelheid wordt altijd aangeduid als het *woord.*

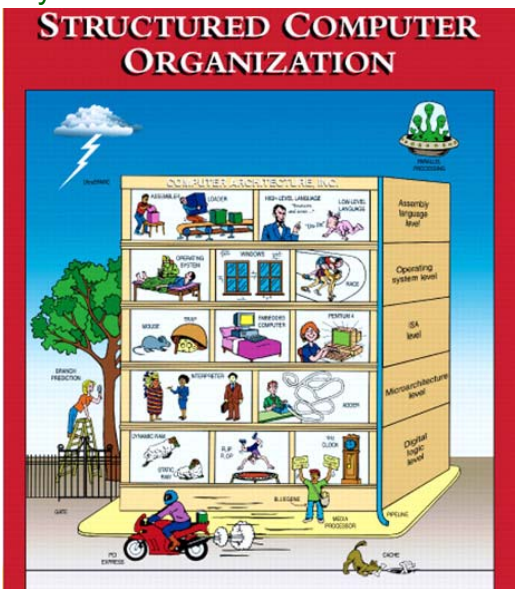
Hiermee kunnen wij een werkdefinitie geven van een 'scalair' systeem, namelijk een systeem dat (ten hoogste) per clock cyclus *één* instructie kan verwerken. De Von Neumann machine kan gezien worden als de technische realisatie van een dergelijk systeem.

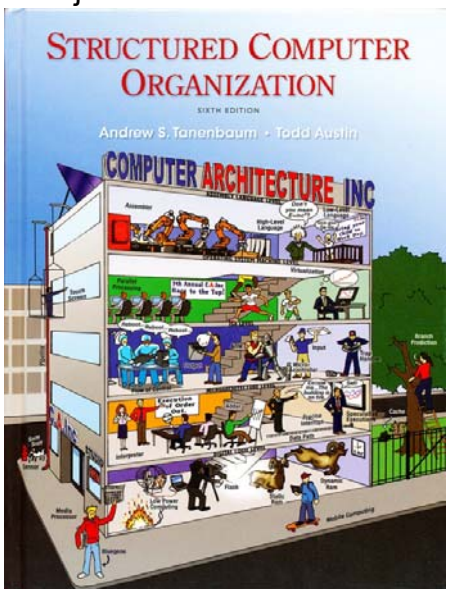
De vragen die daarbij een rol spelen zijn de volgende:

- hoe zorgt men ervoor dat de verwerkingssnelheid van de CPU gelijke tred houdt met de mogelijk om gegevens uit het geheugen te halen (adresseren van het geheugen) en naar de CPU te brengen (I/O). Men onderscheidt hierbij drie mogelijke systemen (systeemtoestanden):
  CPU bound (CPU is snelheidsbepalend, karakteristiek van zwaar rekenwerk),
  memory bound (veel geheugen is nodig),
  I/O bound (karakteristiek van allerlei transaktie systemen).
- welke mogelijkheden er zijn voor optimalisatie/snelheidsverhoging van het systeem.
- welke soorten instructie/informatie kan men eenvoudig verwerken.

Adapted from an introduction to: COMPUTER ARCHITECTURE

by Andrew S. Tanenbaum Emeritus Vrije Universiteit Amsterdam





1

Antikythera mechanism (≈150 BC)

analog computer to predict astronomical positions and eclipses

http://en.wikipedia.org/wiki/Antikythera_mechanism

3

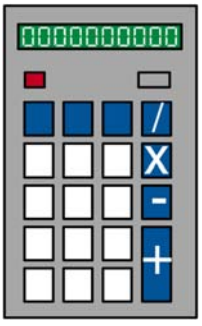
BLAISE PASCAL (1623-1662)

- Invented mechanical adding machine at 19
- Like modern computer: Broke down a lot, hard to fix
- Did much research in mathematics and hydrostatics

4

BARON GOTTFRIED WILHELM VON LEIBNIZ (1646-1716)

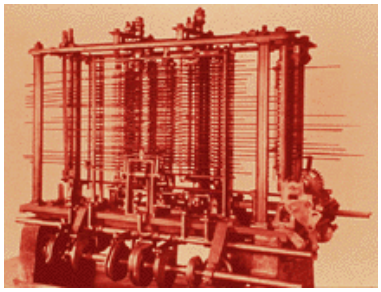
- Taught himself advanced Latin and Greek
- Entered University of Leipzig at age 14
- Co-invented calculus (independent of Newton)
- Invented binary numbers, 4-function calculator



5

CHARLES BABBAGE (1791-1871)

- Occupied Newton's chair at Cambridge (never taught)
- Built working Difference Engine (1832)
- Designed Analytical Engine, but never completed it
- Analytical Engine had processor, memory, I/O



Analytical engine

6

Enigma & the Bombe (1939-1942)

- electromechanical device used by British cryptologists to help decipher German Enigma-machine-encrypted secret messages during World War II

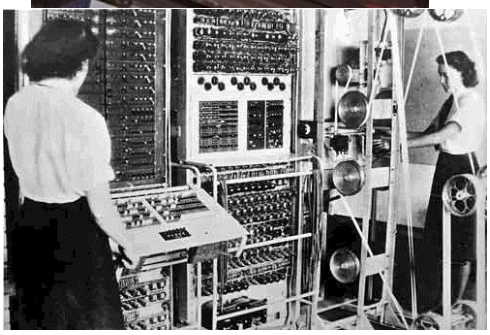
http://en.wikipedia.org/wiki/Bombe

Alan Turing (1912-1954)
father of theoretical computer science and artificial intelligence

7

Lorenz machine & Colossus (1943,1944)

- first “programmable” electronic digital computer by setting up plugs and switches to alter the wiring
- no internally stored programs
- developed for British codebreakers during World War II
- used thermionic valves (vacuum tubes)

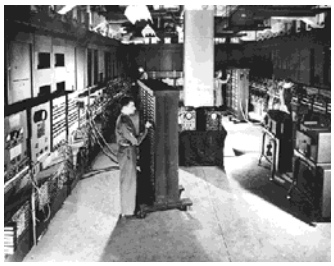


Tommy Flowers, MBE (1905-1998) British engineer

8

ENIAC (1945)

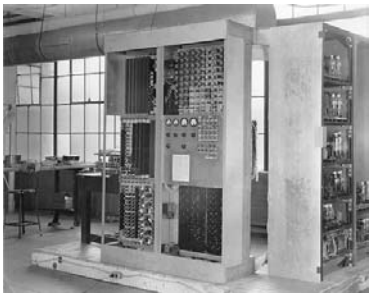
- Designed by J. Mauchley & J.P. Eckert at U. of Penn
- Funded by the Army for numerical calculations
- Contained 17,000 vacuum tubes
- Speed: 5000 additions/sec
- Programmed using jumper cables



9

EDVAC (1948)

- Designed by John von Neumann at U. of Penn.
- ***Program was stored in memory***
- Many computers used this design for decades



10

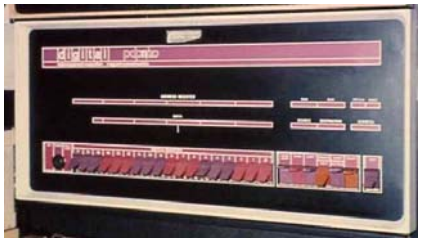
IBM SYSTEM 360 (1964)

- Family of six compatible mainframe
- Used small integrated circuits (chips)
- Used for scientific and commercial work
- Widely used for COBOL (Y2K problem)
- Made IBM dominant computer company for 20 years

11

MINICOMPUTERS (1960s)

- PDP-1 (1961) cost $100K, occupied 2 m^3
- Individual departments could now buy a computer
- Minis were used to control scientific experiments
- President of DEC (once-dominant mini vendor) said:
  - "Nobody needs a computer in his house"

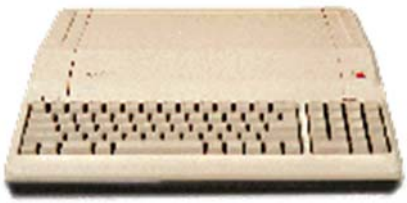


PDP-11 family had a run of 25 years

12

APPLE I (1976) AND APPLE II (1977)

- Built by Steve Jobs & Steve Wozniak in Jobs' garage
- First mass market PC (millions sold)
- Apple II: 4 KB RAM, 8 expansion slots, used color TV
- Schools and individuals could now buy a computer



13

IBM PC (1981)

- IBM wanted a game machine to compete with APPLE II
- Instead, PC was used by corporations
- Used 4.77 MHz Intel 8088 CPU, 16 KB RAM
- IBM published the design, which led to clone industry
- Current Pentium systems derive from this system

14

CURRENT SPECTRUM

Type	Price ($)	Example
Disposable computer	1	Greeting card
Embedded computer	10	Watch, car, TV
Game computer	100	Home video game
Personal computer	1K	Desktop or portable computer
Server	10K	Network server
COW	100K	Departmental minisupercomputer
Mainframe	1M	Batch data processing in a bank
Supercomputer	10M	Long-range weather prediction

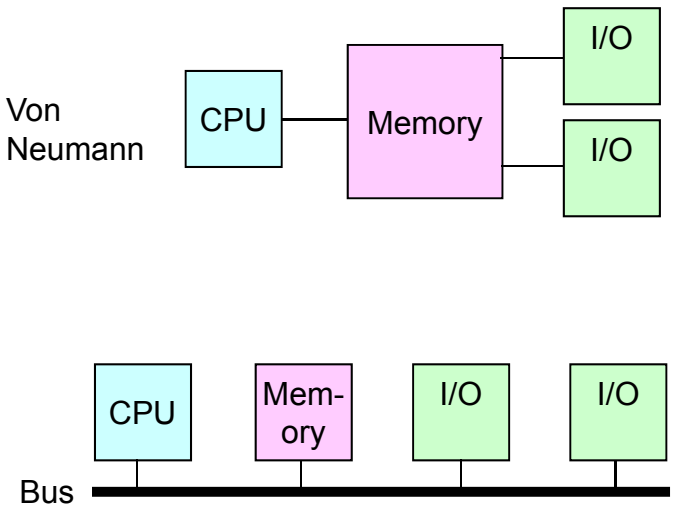
15

COMPUTER ARCHITECTURE

- What the computer looks like to the programmer
- About the hardware interface, not the implementation
- Components
  - CPU
  - Memory
  - Input/Output

16

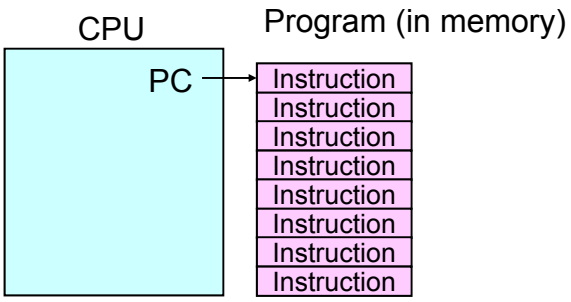
VON NEUMANN VS BUS DESIGNS



17

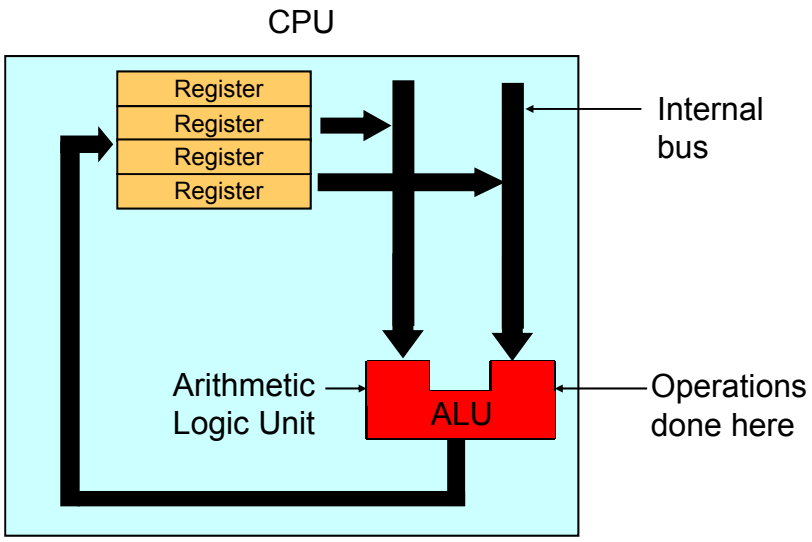
BASIC CPU OPERATION

- A program is a sequence of instructions in memory
- PC (Program Counter) points to current instruction
- CPU fetches an instruction, executes it, advances PC



18

CPU DATA PATH



19

TRENDS IN CPU ARCHITECTURE

- RISC engines
- Multiprocessors
- Cluster of Workstations (COW)

1997: 64 of the VU DAS Machines

2011:74 dual-quad CPU nodes

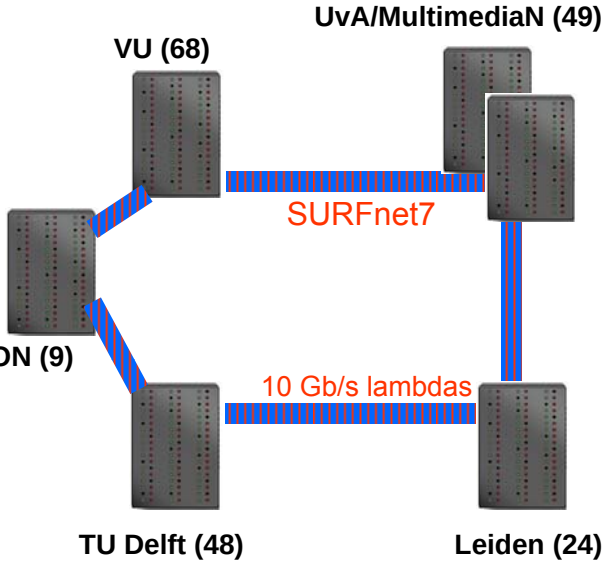
Connected to other sites:
Distributed ASCI Supercomputer (DAS)

20

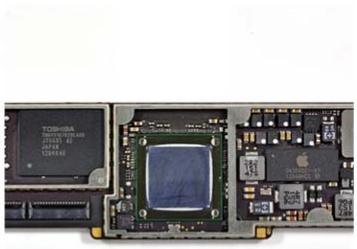
Six locations, see http://www.cs.vu.nl/das5

Testbed for clouds, diversity & Green IT

≈200 Dual 8-core Intel E5-2630v3
64 GB memory
128 TB disk
FDR Infiniband + 1Gb/s Ethernet
Various accelerators (GPUs, multicores, ….)
OS: CentOS Linux



CPU vs SoC (System on a Chip)



Conventional PC motherboard (left) vs. the main iPad 3 circuit board (right)
http://www.extremetech.com/computing/126235-soc-vs-cpu-the-battle-for-the-future-of-computing
http://www.extremetech.com/computing/190764-a8-soc-reverse-engineered-revised-cpu-new-quad-core-gpu-tsmcs-20nm-process

22

Computer Science Software pioneers (by IvS)

- Leibniz dreams of universal calculator
  http://en.wikipedia.org/wiki/Leibniz
- Boole's algebra
  wiki/Boole
- Frege's logic
  wiki/Gottlob_Frege
- Turing conceives first all-purpose computer:
  state machines
  wiki/Alan_Turing
- See also http://www.computerhistory.org/timeline/
- Further reading: Martin Davis (2000) The Universal Computer, The Road from Leibniz to Turing. Norton&Co, New York

23

Practicumopgaven Module 1 van Modelleren, Programmeren en Simuleren

Mathematica

Herhalingsopgaven

In de cursus Programmeren in Mathematica van het Natuurkunde practicum 1 B dit voorjaar hebben jullie deze opdrachten al gemaakt. Om deze vaardigheden op te frissen is het wenselijk om een klein aantal herhalingsopgaven te maken. Het gaat hier met name om het schrijven van functies en het gebruik van Module en van Map. Lees eerst de toelichting in **functies.nb**
De twee herhalingsopgaven zijn:
numderiv.nb
word_exercise.nb
Open ook het file **digits.nb** en experimenteer zelf met enkele getallen.
Data files en notebooks staan in http://www.nat.vu.nl/~ivo/MPS/Mathematica

Opgave 1 GGD algoritme

Zie **ggd_opgave.nb**.
Implementeer het GGD algoritme van Euclides in Mathematica,
en test het voor 3 gevallen:
1001 182
13 31
289587035228888704
Hoeveel stappen heb je in deze 3 gevallen nodig?
Hoe beoordeel je de geschiktheid van Mathematica als PSE voor dit probleem?

Opgave 2 Fibonacci getallen recursief met Mathematica

Schrijf een recursieve functie Fib (zonder tussentijdse opslag) zoals behandeld op het college.
Meet de rekentijd als functie van **n** voor de eerste 25 Fibonacci getallen.
Maak er een grafiek van? Wat is de orde?
Maak een verbeterde functie Fib2 met tussentijdse opslag zoals behandeld op het college.
Maak weer een grafiek van de rekentijd als functie van **n** voor de eerste 25 Fibonacci getallen.
Bestudeer ook nog het effect van Compile.

Opgave 3 Towers of Hanoi

Maak de opgave in het file **hanoi.nb**

Opgave 4 Teken een boom van Pythagoras.

Zoals beschreven in paragraaf 2.3 van het dictaat. Ga na dat je het principe begrijpt.
Beantwoord de vraag uit het dictaat: “Wijs zelf in de figuur de plek van a, b en c aan.”
Schrijf een bijschrift voor het plaatje. Onderzoek wat er gebeurt als je a, b en c uit de lijst van locale variabelen haalt, en verklaar de uitkomst die je dan krijgt.

Opgave 5 Teken een Sierpinski driehoek

Maak de opgave in het file **Sierpinski_exercise.nb**
Schrijf een bijschrift voor het plaatje.

Opgave 6 Genereren van random getallen.

Gebruik de lineair congruentiële methode met de constanten **a,c** en **m** beschreven in figuur 10 van paragraaf 3.4 van het dictaat en maak een ListPointPlot3D. Schat het aantal vlakken in deze figuren (zorg dat je genoeg punten tekent als er veel vlakken zijn), en maak daarvan een tabel. Plot het aantal vlakken als functie van **m**, en check of dit inderdaad klopt met de formule $\sqrt[3]{m}$.

Onderzoek van de volgens jou beste random generator de verdeling met behulp van een histogram.
Berekenen het gemiddelde over een aantal samples en de verdeling over het interval met behulp van een *histogram*.
Histogram[list]
Onderzoek kwalitatief wat er met de verdeling gebeurt als je tien samples gaat optellen.
Volgens de centrale limietstelling zou er een Gauss verdeling uit moeten komen. Controleer dit mbv een nieuw *histogram*.

Opgave 7 *Samenhangende* begrippen opzoeken

Informatica en de toepassingen ervan vormen een zeer dynamisch gebied. Het World wide web bevat een schat aan informatie. Zoek tien zelfgekozen begrippen op (bv. technische begrippen uit dit dictaat, of buzzwords waar je graag wat meer over wilt weten). *Kies bij voorkeur een aantal samenhangende begrippen.*
Schrijf voor elk begrip een korte samenvatting van je bevindingen (dit mag naar keuze in het Nederlands of in het Engels, ook door elkaar). Je mag daarbij ook de internet tekst citeren. Vermeld steeds je bronnen.
Het totale verslag van deze opgave dient 2 à 4 A4 te zijn.

Matlab & Simulatie

Kennismaken met Matlab

Bekijk eerst het begin van de Matlab Help, met name van Getting Started:
Introduction

- Starting and Quitting the MATLAB Program

Verlaat Matlab via File/Exit Matlab! (anders blijft de licentie hangen)

Matrices and Arrays

- Matrices and Magic Squares
- Expressions
- Working with matrices
- More About Matrices and Arrays

Het is verstandig om veel commando's zelf in te typen in het Command Window. Merk op dat je operaties zowel via het menu als via het Command Window kunt uitvoeren. Bv **Edit/Clear Command Window** (via het Edit menu) is equivalent met **clc** in het Command Window.
In het Desktop menu kun je met **Desktop/Desktop Layout/Default** je GUI herstellen.

Opgave 1 Finite difference afgeleide berekening met Matlab

De afgeleide van een sinus is een cosinus. In het college is uitgelegd dat een benadering hiervoor gevonden kan worden mbv een finite difference afgeleide.
Hier bestuderen we hoe goed die benadering is voor twee gevallen.

Maak een array:

powers=(1:64)

We werken in Matlab het meest met arrays, en maken daarbij veel gebruik van de colon operator (**:**) (dubbele punt). Merk op dat de dubbele punt een array maakt van 1 tot 64, met stappen van 1. Het is equivalent met **(1:1:64)** . Zie een van de help pagina's die je kunt vinden door in de Search optie van het Help window te zoeken op **colon**.

harr=2.^(-powers)

Merk op dat de punt voor het dakje (**.^**) essentieel is om de bewerking (hier: machtsverheffen) op alle elementen van de vector uit te voeren! Evenzo gaan we nu de afgeleide berekenen (**fd** is de afkorting voor finite difference, **1** voor eerste orde benadering, en **arr** voor array, je mag natuurlijk zelf andere namen geven):

fd1arr=abs((sin(1+harr)-sin(1))./harr-cos(1))
fd2arr=abs((sin(1+harr)-sin(1-harr))./(2.*harr)-cos(1))

Merk weer op dat de punt voor het gedeeld door teken essentieel is. MATLAB functies worden met een kleine letter geschreven, merk op dat MATLAB case sensitive is, SIN bestaat dus niet.
We gaan nu naar:

Graphics
Basic Plotting
Creating a Plot
plot(harr,fd1arr)
Multiple Data Sets in One Graph
plot(harr,fd1arr,harr,fd2arr)

Specifying Line Styles and Colors
Plotting Lines and Markers

plot(harr,fd1arr,'k:s', harr,fd2arr,'r:+')

Voor een dubbel-logaritmisch plaatje bestaat er de functie **loglog.** Probeer eens of je daar zelf achter kunt komen mbv de Search optie van het Help window.

Axis Labels and Titles

Geef nu zelf labels en een titel.

Om de lineaire verbanden op deze dubbellogaritmische schaal nader te onderzoeken is het nodig om een nieuw plaatje te tekenen (gebruik daarvoor **figure()**), van de eerste 27 punten van **fd1arr** en van de eerste 17 punten van **fd2arr**. Gebruik de Colon Operator om delen van een array te plotten. Gebruik nu uit het Tools menu de Basic Fitting interface. Probeer eerst lineair, en daarna quadratic. Vink Show equations aan. Save je plaatje als een JPEG image. Doe dit voor beide curves. Wat concludeer je uit deze equations over de verbanden? Klopt dit met de theorie uit het dictaat?

Interpreteer het plaatje en schrijf er een bijschrift voor.

Maak nu een script file via het menu File / New / Script
Zet daarin de gebruikte commando's (die in de *command history* staan) voor het gehele plaatje (dus zonder de fit), save het file als **fd.m** en executeer (=voer uit) met **fd** in het *command window* (of met het groene pijltje in de Editor). Merk op dat je nu veel output ziet verschijnen. Open het file **fd.m** opnieuw en merk op dat de = tekens geel zijn. Als je daar met je muis naar wijst, krijg je een aanwijzing. Om deze output te onderdrukken kun je een **;** (puntkomma=semicolon) toevoegen aan het einde van de regel.
Het voordeel van een script file is dat je een aantal vaste commando's makkelijk kunt executeren (=uitvoeren). Het is echter niet zo flexibel als een functie, die we in opgave 4 zullen tegenkomen.

Als je een globale variabele wilt verwijderen gebruik je bv

clear powers

Hoe beoordeel je nu de geschiktheid van Matlab en Mathematica als PSE voor dit probleem? Wat zijn de voors en tegens van elk, en waarom? Hoeveel tijd heeft het je nu gekost om kennis te maken met deze nieuwe PSE?

Opgave 2 Plotten van externe gegevens: bepalen van de wet van Moore

Importeer de rechthoekige data set **Moore.dat** mbv de **File/Import** wizard. Geef de data (Range B1:D16) daarbij de naam "moore" (ipv de default naam "data") door tweemaal in de name box te clicken.
Maak een figuur met twee y-assen (secondary axes) met behulp van

plotyy(moore(:,1), moore(:,2), moore(:,1), moore(:,3))

Gebruik voor het aanpassen van de vertical assen het **Edit/Axes** menu van de Figure GUI. Maak om te beginnen de Y-axes logarithmisch. Wat concludeer je over het verband tussen x en y?
De **Basic Fitting** mogelijkheden onder de Tools van de Figure GUI zijn beperkt tot eenvoudige functies die lineair zijn in de fit parameters. Om de wet van Moore te controleren met deze data is het dus nodig om eerst de data te transformeren, met behulp van functies als **log**, **log2** of **log10**. We kiezen hier voor een transformatie met **log2** omdat we de verdubbelingstijd τ willen vinden. Als

$$y = y(t_0)2^{(t-t_0)/\tau}$$

Dan is

$$^2\log(y) = {}^2\log(y(t_0)) + (t-t_0)/\tau$$

We kiezen $t_0 = 1970$

plotyy(moore(:,1)-1970, log2(moore(:,2)), moore(:,1)-1970, log2(moore(:,3)))

Voer nu een fit uit van de beide data sets met een lineair model. Wat is de betekenis van de geschatte parameters? Interpreteer de uitkomst van de beide fits. Wat concludeer je over de wet van Moore op grond van deze gegevens?

Opgave 3 Fibonacci getallen met Matlab

Bereken de eerste 100 Fibonacci getallen met Matlab.
Maak eerst een array van lengte 100, waarbij de eerste twee elementen overeenkomen met de eerste twee Fibonacci getallen.

fibo=(1:100)

We gaan nu verder met de Matlab Help.

Getting Started: Manipulating Matrices: Subscripts

fibo(2)=1

Getting Started: Programming with MATLAB: Flow Control

De belangrijkste control statements zijn voor ons:

conditional control

if, else, en elseif

loop control

for en while

Zoek in het vervolg alle onbekende begrippen op met de Matlab Help door een woord te selecteren en dan op de F1 toets te drukken.
We zullen de verschillende control statements één voor één tegenkomen in verschillende opgaven, we beginnen hier met de bekende **for** loop (=herhalingsconstructie)

```
for n=3:100
    fibo(n)=fibo(n-1)+fibo(n-2);
```

Merk op dat de puntkomma hier noodzakelijk is om tussentijds printen te onderdrukken

```
end
fibo(73)
fibo
```

Merk op dat in het laatste geval het hele array geprint wordt, waarbij een factor 1E20 vooraan staat!
Probeer weer om alle regels te begrijpen. Vertel in je eigen woorden hoe deze code werkt.
Om te zorgen dat een getal weggeschreven wordt geheel getal zonder cijfers achter de komma, en niet in wetenschappelijke notatie zoals 1.23456 E2 moeten we een aparte functie gebruiken zoals **fprintf** en **sprintf**, waarbij de eerste letter een afkorting is voor respectievelijk *file* en *string*, en de laatste **f** staat voor *formatted*.

sprintf('%.0f', fibo(73))

waarbij tussen de quotes het gebruikte formaat staat, zie Constructing the Formatting Operator en zie ook het dictaat. Een string is een aantal characters achter elkaar, bv een regel.
Om zonder *ans* te printen gebruik dan display ().
Bekijk de Fibonacci getallen vanaf 70 nauwkeurig. Wat valt je op?
Hoeveel Fibonacci getallen kun je met uint64 nauwkeurig berekenen?

Opgave 4 Berekening van de discriminant van een vierkantsvergelijking $ax^2 + bx + c = 0$ met Matlab

We gaan nu verder met de Matlab Help. Een functie staat in een **.m** file

Getting Started: Programming with MATLAB: Functions.

Maak ook de beide examples op het eind van de Function help pagina (waar bovenaan staat MATLAB function reference).

Development Environment: Editing and Debugging M-Files: Starting the Editor/Debugger

We gaan een simpele functie schrijven die de discriminant $D = b^2 - 4ac$ uitrekent.

Schrijf de volgende code:

function out=discriminant(a,b,c)

out= vul hier zelf iets in

Merk op dat de editor met rode kleur aangeeft als er fouten in je code zitten. Als je daar met je muis naar wijst, krijg je een aanwijzing, probeer het maar eens door achteraan de eerste regel een extra) toe te voegen.

Save en check dat er een **discriminant.m** file is aangemaakt in je Current Directory.

Test de functie in je command window met:

discriminant(1,3,-4)

Het voordeel van een functie is dat je die met verschillende inputwaarden voor **a,b,c** kunt gebruiken. Test dit. Een functie is dus veel flexibeler dan een script.

Opgave 5 Oplossen van een vierkantsvergelijking met Matlab

We gaan de **discriminant** functie nu gebruiken in een nieuwe functie.

Schrijf de volgende code:

```
function [nsol,sol1,sol2]=solvevierkantsvgl(a,b,c)
det=discriminant(a,b,c);
```

Merk op dat de puntkomma hier weer noodzakelijk is om tussentijds printen te onderdrukken.

```
if det<0
    nsol=0;
    return
elseif det==0
        nsol=1;
        sol1=-b/(2*a);
    else
```

maak de functie verder zelf af. De wortel (=square root) functie is **sqrt.**

Save en check dat er een **solvevierkantsvgl.m** file is aangemaakt in je Current Directory.

Test je functie in je command window met

```
[nsol,sol1,sol2]=solvevierkantsvgl(1,1,-2)
```

en met een aantal andere inputwaarden voor **a,b,c.**

Wanneer er nu minder dan twee oplossingen zijn treedt er een probleem op als **sol1** of **sol2** nog geen waarde hebben, en Matlab geeft error meldingen. Dit probleem kun je vermijden door voorafgaand aan de **if** test deze variabelen gelijk aan NaN te maken (NaN is Not-a-Number):

```
sol1=NaN;
```

Opgave 6 GGD algoritme met Matlab

Schrijf de volgende code:

```
function out=ggd(u,v)
if (v==0)
  out=u;
```

Merk op dat de puntkomma hier weer gebruikt wordt om tussentijds printen te onderdrukken

```
else
  out=ggd(v,mod(u,v));
```

end

Probeer eerst alle regels te begrijpen.
Check dat er een **ggd.m** file is aangemaakt in je Current Directory.
Test de functie in je command window met:
ggd(1001,182)
ggd(1000000000012,112)
ggd(100000000000012, 112)
ggd(100987651,19876582)
Breid de functie nu uit met een **teller** globale variabele die het aantal aanroepen van ggd bijhoudt. Zie

MATLAB Functions: global

Vergeet niet om de teller steeds te resetten.
Hoe beoordeel je nu de geschiktheid van Matlab en Mathematica als PSE voor dit probleem? Wat zijn de voors en tegens van elk, en waarom?

Opgave 7 Inefficiënt GGD algoritme met Matlab

Schrijf de volgende code:

```
function t=ggd1(u,v)
t=min(u,v);
while(mod(u,t) || mod(v,t))
    t=t-1;
end
```

Probeer weer eerst alle regels te begrijpen.
Test de functie in Matlab met:
ggd1(1001,182)
ggd1(1000000000012, 112)
ggd1(100000000000012, 112)
ggd1(100987651, 19876582)
Om de rekentijd van de beide algoritmen te vergelijken gebruiken we **tic** en **toc** op de volgende manier:
tic;ggd(100987651,19876582);toc
tic;ggd1(100987651,19876582);toc
Wat concludeer je?
NB Een langlopende berekening kun je afbreken met ^C (Ctrl-C, spreek uit control-C).
Zou je dit algoritme net zo snel in Mathematica kunnen programmeren? Waarom wel of niet?

Opgave 8 Oplossen van een vierkantsvergelijking met Matlab, deel 2: subfunction en struct.

In een function file kunnen meerdere functions gedefinieerd worden. De eerste functie is de **primary function**, en de volgende heten **subfunctions**. Subfunctions worden net zo gedefinieerd als functies, maar kunnen alleen maar aangeroepen worden door de andere functies in het function file, terwijl de primary function natuurlijk wel van buiten aangeroepen kan worden. Lees nogmaals de help-pagina van **function**, en bestudeer daarvan Example 2. De naam van het function file moet wel corresponderen met de naam van de primary function. Alle variabelen die gebruikt worden in een function zijn locaal, tenzij ze globaal gedeclareerd zijn natuurlijk.
Verder gaan we de output van de functie nu in een zelfgedefinieerde datastructuur onderbrengen. Deze structuur vat een aantal bij elkaar horende variabelen samen, met aparte namen. Lees daarvoor eerst de help-pagina van **struct**
Schrijf de volgende code:

```
function s=solvevierkantsvgl2(a,b,c)
```

det=detsub(a,b,c);

Merk op dat de puntkomma hier weer noodzakelijk is om tussentijds printen te onderdrukken.

```
if det<0
    nsol=0;
    s=struct('nsol',nsol);
    return
elseif det==0
        nsol=1;
        sol1=-b/(2*a);
        s=struct('nsol',nsol,'sol1',sol1);
    else
```

maak de functie verder zelf af, en voeg tenslotte een subfunction toe:

function out= detsub(a,b,c)

out= vul hier zelf iets in

Test **solvevierkantsvgl2** weer met een aantal inputwaarden voor **a,b,c** bijvoorbeeld:

s=solvevierkantsvgl2(4,1,-3

fieldnames(s)

geeft de namen van de verschillende fields. Om de value van een field van **s** te zien gebruik je de **.** operator gevolgd door de naam van het field:

s.nsol

s.sol2

Opgave 9 Oplossen van een vierkantsvergelijking met Matlab, deel 3: een command interpreter met input en text output.

Met de volgende functie file creëer je een eenvoudige command interpreter:

```
function solver
doorgaan=1;
while (doorgaan)
    a=input('geef a, a dient ongelijk aan 0 te zijn ');
    b=input('geef b ');
    c=input('geef c ');
    s=solvevierkantsvgl2(a,b,c);
    printsol(s);
    doorgaan=input('doorgaan? [1 betekent doorgaan, en 0 betekent
stoppen]');
end
```

Lees de help-pagina's van **input, sprintf** en **strvcat.**

Schrijf om uitgebreidere text output te genereren zelf een functie **printsol** waarin je de **if, elseif** control structure en de functies **sprintf** en **strvcat** gebruikt om text strings te vullen, en daarna aan elkaar te plakken. Bijvoorbeeld om een integer (=geheel getal) in een string te printen gebruik je:

sprintf('de integer n is %d \n',n)

waarbij %d het formaat is, en **d** staat voor **d**ecimaal integer. Zie verder Constructing the Formatting Operator en zie ook het dictaat.

Zorg dat alle informatie over het aantal oplossingen en de waarde van die oplossingen terecht komt in een text string met

text=strvcat(text1,text2,text3);

zodat ze onder elkaar geprint worden wanneer je het resultaat vraagt:

text

Let op, zet hier geen **;** achter **text** omdat je hier juist wel output wilt.
Test **`solver`** weer met een aantal inputwaarden voor **a,b,c**
De output dient er ongeveer zo uit te zien:

solver()
geef a, a dient ongelijk aan 0 te zijn -1
geef b 1
geef c 2

text =

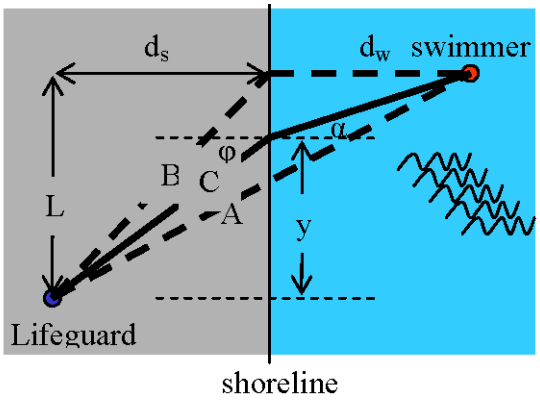
er zijn 2 oplossingen

oplossing 1 is -1.000000
oplossing 2 is 2.000000

doorgaan? [1 betekent doorgaan, en 0 betekent stoppen]

Opgave 10 Berekenen van het snelste pad.

(*taken from MatLab: An introduction with applications from A. Gilat, 3rd edition 2008*, p. 114)



A student has a summer job as a lifeguard at the beach. After spotting a swimmer in trouble he tries to deduce the path by which he can reach the swimmer in the shortest time. The path of shortest distance (path A) is obviously not best since it maximizes the time spent swimming (he can run faster than he can swim). Path B minimizes the time spent swimming but is probably not best since it is the longest (reasonable) path. Clearly the optimal path is somewhere in between paths A and B.

Consider an intermediate path C and determine the time required to reach the swimmer in terms of the running speed v_{run} = 3 m/s, swimming speed v_{swim} = 1 m/s, the distances L = 48 m, d_s = 30 m, and d_w = 42 m, and the lateral distance y at which the lifeguard enters the water. Create a vector y that ranges between path A and path B (y = 20, 21, 22, …, 48 m) and compute a time t for each y. Use MATLAB built-in function `min` to find the minimum time t_{min} and the entry point y for which it occurs. Determine the angles that correspond to the calculated value of y and investigate whether your result satisfies Snell's law of refraction:

$$\frac{\sin\phi}{\sin\alpha} = \frac{v_{run}}{v_{swim}}$$

Opgave 11 Programmeer zelf de Towers of Hanoi

In het college is de volgende code besproken:

```
function hanoi(fid,ndisks,from,to,tmp)
if ndisks>0
    hanoi(fid,ndisks-1,from,tmp,to);
    fprintf(fid,'Move disk %c from %d to %d\n',ndisks+64,from,to);
    hanoi(fid,ndisks-1,tmp,to,from);
end
```

hierbij is **fid** een file identifier die geretourneerd wordt door een aanroep in het command window van **fopen:**

fid = fopen('hanoi.txt', 'w');

Roep **hanoi** aan met ndisks gelijk aan 3, en check dat je het algoritme helemaal begrijpt.
Alvorens **hanoi** opnieuw aan te roepen moet het file eerst gesloten worden:
fclose(fid)
Wanneer het file nu opnieuw geopend wordt, wordt de oude inhoud overschreven. Lees de manual page van fopen voor andere opties, zoals append.
Roep daarna **hanoi** aan met ndisks gelijk aan 4 en 5.
Tel het aantal aanroepen van de functie MoveDisks door een globale variabele te gebruiken, en print dit aantal. Check dat dit aantal klopt met de som formule bij gebruik van verschillende ndisks waarden.
Toelichting: We noemen de kleinste disk A, de net iets grotere disk B, etc.
Voor het printen van een integer als character, zie de ASCII tabel in Figuur 4 op pagina 4 van het hoofdstuk Datatypes in het dictaat, en ook http://en.wikipedia.org/wiki/ASCII.

Opgave 12 Computereigenschappen

Bekijk via de Properties van My Computer wat de eigenschappen van je systeem zijn. Doe dit zowel op een PC tijdens een *Matlab* practicum, als op je laptop. Op het eerste tabblad General staat je processor (hoe snel is ie?), operating system, en de grootte van je memory. Onder het tabblad Hardware zit een Device Manager. Bekijk eens hoe groot je harddisk is via Windows Explorer of andere tools. Met Ctrl-Alt-Delete kun je de Windows Task Manager starten. Bekijk de verschillende tabbladen. Hoeveel Memory en CPU gebruiken applicaties als *Mathematica* of *Matlab?*

Opgave 13 Computer architectuur simulatie

Bestudeer **Reed2004Ch14.pdf**, bijgaande instructieset en het voorbeeld programma. Schrijf nu zelf een programma in assembly van minstens tien instructies, waarin je enkele getallen leest uit het Main Memory, de getallen daarna op je eigen wijze manipuleert, en wegschrijft naar het Main Memory.
Maak een screenshot van je programma, en schrijf een bijschrift (10-20 regels) bij deze screenshot, waaruit blijkt dat je begrijpt hoe een computer werkt.

P.1 I/O in Matlab

Een standaardvoorziening is het lezen van en schrijven naar *files*. Essentieel voor de input/output naar een file is een *file_identifier*, die men verkrijgt via de functie *fopen*. Er bestaat standaard al een file_identifier (*fid*) voor *standard output (fid=1)*, en *standard error (fid=2)*. De syntaxis van *fopen* wordt gegeven door

```
fid = fopen(filename, mode);
```

Filename is een character string die aangeeft welk file geopend moet worden, terwijl de character string *mode* aangeeft hoe het file geopend moet worden. Voor *mode* zijn de volgende mogelijkheden:

r	read only
w	write only, if necessary create; existing files are truncated to zero
r+	read/write
w+	read/write, if necessary create
a	append.

Indien de fopen succesvol was wordt een file_pointer groter dan nul geretourneerd anders wordt de -1 geretourneerd. De logische tegenhanger van fopen wordt gevormd door *fclose* met de volgende syntaxis:

```
fclose(fid);
```

P.1.1 Formatted output

Formatted output kan gedaan worden met behulp van het fprintf statement dat de volgende syntaxis heeft (zie ook de manual page van fprintf);

```
fprintf(fid, format, arg_0, arg_1, arg_2, .....)
```

Fid specificeert de *file_identifier* die gebruikt moet worden. *Format* specificeert hoe de printf uitgevoerd moet worden, terwijl *arg_0*, *arg_1* etc. de argument vormen die geprint moeten worden. Het is goed om er op te wijzen dat wij hier een voorbeeld zien van een functie met een variabel aantal argumenten! De *file_identifier* is of geopend volgens de hiervoor beschreven procedure of is een van de standaard geopende file_pointers (stdout en stderr).
Het *format* wordt gegeven als een string d.w.z. tussen enkele quotes. Alle tekst die hierin geplaatst is wordt direct geprint, behalve als er een **%** teken gezien wordt. Het procentteken wordt opgevat als een special. Het argument dat direct na een % teken volgt wordt geïnterpreteerd als specificatie voor de manier waarop het argument geprint moet worden. Uit de mogelijkheden die er zijn geven wij de volgende aan:

c	het volgende argument van de lijst moet worden geprint als character,
s	het volgende argument van de lijst moet worden geprint als string,
d	het volgende argument van de lijst moet worden geprint als integer,
f	het volgende argument van de lijst moet worden geprint als float,
e	het volgende argument van de lijst moet worden geprint als niet geheel getal in een exponentiële notatie.

Bij het *f* en *e* format kan nog een verdere beschrijving opgegeven worden. Een volledige omschrijving voor het printen van een float heeft de vorm %<*digit*>.<*digit*>*f* waarin het eerste digit veld aangeeft hoeveel digits er in het totaal gebruikt mogen worden voor het printen van het getal en het tweede veld aangeeft hoeveel digits er gebruikt moeten worden voor het printen van de fractie achter de decimale punt. Het eerste digit veld is optioneel. Zo zal het format *%.4f* een niet geheel getal printen met vier decimalen achter de komma. Een wat completer voorbeeld is (hierin is \n een special character, newline):

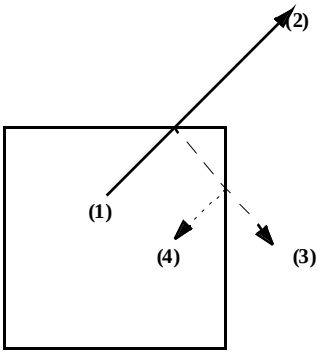
```
fprintf(fid, '%d counts are registers; average is: %.4f\n', ncount, average);
fprintf(fid, 'The sample was %s\n', 'Silver');
```

Men dient wel enige voorzichtigheid te betrachten bij het gebruik van fprintf. Het maakt de functie weinig uit wat het type is van de actuele argumenten die hij meekrijgt: alles bepalend is het formaat dat opgegeven is.

Zie verder de Matlab manual en Appendix C van Module 2

Case study Matlab

- Simuleer een ideaal gas
  - Geen interactie met deeltjes
  - Wel interactie met wanden



$$x(t+\Delta t) = x(t) + v(t)\Delta t$$

(1) Startpunt
(2) Positie zonder reflectie
(3) Eerste gecorrigeerde positie
(4) Uiteindelijke positie

Het algoritme

- Initialiseer het systeem
- Herhaal voor alle tijdstappen
  - Herhaal voor alle deeltjes
    - Bereken nieuwe positie,
    - Controleer positie en zonodig corrigeer,
    - Print indien gewenst resultaat

Implementatie

```
function prestot=gasv(nprint,ntimes,nparticles,vmax)
% schrijf zelf commentaar
x=unifrnd(0,1,[1 nparticles]);
y=unifrnd(0,1,[1 nparticles]);
vx=unifrnd(-vmax,vmax,[1 nparticles]);
vy=unifrnd(-vmax,vmax,[1 nparticles]);
pres=zeros(5,1);
prestot=zeros(ntimes/nprint,5);
iprint=0;
```

Datastructuur
&
Initialisatie

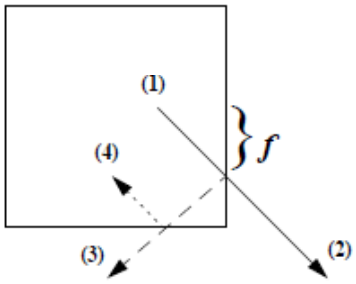
Implementatie (main loop)

```
for i=1:1:ntimes
   newx=step(x,vx,.1);
   newy=step(y,vy,.1);
   for j=1:1:nparticles
      if (~inbox(newx(j),newy(j),0,0,1,1))
[x(j),y(j),newx(j),newy(j),vx(j),vy(j),pres]= ...
correct(x(j),y(j),newx(j),newy(j),vx(j),vy(j),pres,0,0,1,1);
      end
   end
   x=newx;
   y=newy;
   if (~mod(i,nprint))
      iprint=iprint+1;
      prestot(iprint,:)=pres(:);
      prestot(iprint,5)=mean((abs(pres(1:4))));
      pres=zeros(5,1);
   end
end
```

simulatie

hulpfuncties

```
function q=step(p,v,deltat)
q=p+deltat*v;
function out=inbox(x,y,llx,lly,urx,ury)
if (x>llx&&x<urx&&y>lly&&y<ury)
   out=1;
else
   out=0;
end
function [x,y,newx,newy,vx,vy,pres]= correct(x,y,newx,newy,vx,vy,pres,llx,lly,urx,ury)
while (~inbox(newx,newy,llx,lly,urx,ury))
   if (newx>urx)
      f=(newy-y)*(urx-x)/(newx-x);
      if(y+f>lly)&&(y+f<ury)
         y=y+f;
         x=urx;
         newx=urx-(newx-urx);
         vx=-vx;
         pres(1)=pres(1)+2*vx;
      end
   end
…
```



5 De Matlab taal

C en Fortran (en hun opvolgers) kunnen gezien worden als typische voorbeelden van derde generatie programmeertalen: zij bieden recht-toe-recht-aan control flow gebaseerde constructies. Dat maakt de programma's in principe "makkelijk" leesbaar. De basis component waarmee gewerkt wordt is de zogenaamde *functionele abstractie*: de groepering van een aantal statements met specifieke functie. De fundamentele data objecten in C zijn characters, integers en floating point getallen. Van de numerieke types kan de gebruiker tot op zekere hoogte de grootte, en dus onder andere de nauwkeurigheid specificeren. Van deze basis types kunnen afgeleide types gemaakt worden met pointers, arrays, structures en functies.
C biedt flow-control constructies aan die voor gestructureerd programmeren onmisbaar zijn. Deze omvatten: het groeperen van statement (**{** .... **}**), beslissing (**if** .... **else**), herhaling met beslissing aan de top (**for**, **while**) of aan het einde (**do**) en het selecteren van een case uit een aantal mogelijkheden (**switch**). Deze inleiding pretendeert niet om alle essentiele elementen van de taal samen te vatten maar wil dienen als een eerste introductie en een oefening in het lezen van programma's.
PSE's zoals Mathematica en Matlab hebben hun eigen talen. Deze zijn deels geïnspireerd door C, maar verschillen wel onderling en van C. In elke omgeving kom je ongeveer dezelfde datastructuren, controlstructuren en *functionele abstractie* tegen. De Matlab functies worden geschreven in zogenaamde .m files, en deze worden gecompileerd, zodat simulaties efficiënt doorgerekend kunnen worden.

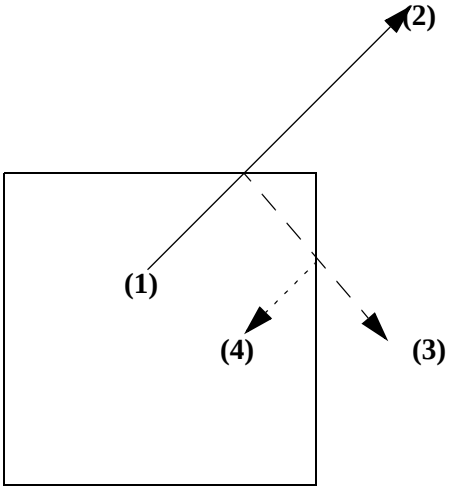
5.1 Case Study

Aan de hand van een case study zullen we proberen om de taal van Matlab nader toe te lichten. Onze case studie betreft de simulatie van een twee dimensionaal ideaal gas opgesloten in een vierkant vat. Voor dit vat willen wij een druk kunnen meten als functie van de tijd.
Om dit te kunnen doen dienen wij van de deeltjes in ons systeem, die we als puntmassa's benaderen, de positie en de snelheid te weten. Omdat het gas ideaal is zullen de deeltjes geen wisselwerking hebben. Verwaarlozen we bovendien de wrijving van de deeltjes met de wand dan kunnen we stellen dat zij een constante snelheid hebben.
Om het probleem te kunnen oplossen discretiseren we de tijd. De verplaatsing van een deeltje in het tijdsinterval wordt berekend met behulp van de formule

$$x(t+\Delta t) = x(t) + v(t)\Delta t \qquad (1)$$

Het is niet mogelijk om zonder meer deze formule te gebruiken om de posities van de deeltjes te vernieuwen want de nieuwe positie hoeft niet binnen het vat te liggen. De juiste vorm van het vat waarin de deeltjes opgesloten zijn is hierbij van belang. Voor convexe vormen zal gelden dat als begin en eindpunt in het vat liggen dat ook zal gelden voor alle punten tussen begin en eindpunt. Voor concave vormen geldt dit niet. In dat geval neemt de complexiteit van ons algoritme snel toe. Voor een vierkant is het eenvoudig te controleren of een punt *(x, y)* binnen het vierkant of er buiten ligt, een simpele test op de coordinaten is voldoende. Als begin en eindpunt binnen het vat liggen kan er geen wand gepasseerd zijn. Als het eindpunt erbuiten ligt is dat wel het geval. Dan moet de nieuwe positie gecorrigeerd worden door de botsingen met de wand in rekening te brengen. Dit proces is in Figuur 1 weergegeven. Vanuit de initiële positie (1) wordt de tentatieve positie (2) uitgerekend. Deze ligt buiten het vat. Reflectie aan de bovenwand geeft de alternatieve positie (3) die nog steeds buiten het vat ligt. Alweer reflectie aan de wand geeft uiteindelijk de correcte positie (4). Bij de botsingen met de wand wordt de snelheidscomponent omgekeerd: bij onder en bovenvlak is dat de y-component van de snelheid, bij linker- en rechtervlak is dat de x-component. Voor de eenvoud tellen wij voor de druk de botsingen op de wand.
De vertaling van dit probleem in Matlab kent een aantal stappen.

- Eerst wordt een passende datastructuur gekozen voor de representatie van de gegevens,
- Dan wordt het totale probleem opgebroken in een aantal stappen,
- Vervolgens wordt voor elke stap, indien opportuun, een functie geschreven,
- dan worden de functies gecombineerd tot de eindoplossing,
- tenslotte dient de correctheid van het programma op een bepaalde manier geverifieerd te worden.



Figuur 1 Simulatie van pad van deeltje wat botst met wanden van vat.
Initiële positie (1), tentatieve positie (2), reflectie op de bovenwand, alternatieve positie (3), reflectie op de rechterwand, correcte positie

5.1.1 Datastructuur

De datastructuur die gekozen wordt voor dit probleem is simpel: vectoren waarin voor alle gasdeeltjes een bepaalde eigenschap wordt opgeslagen. Een voorbeeld is:
x=unifrnd(0,1,[1 nparticles]);
waarin een vector voor de x-positie van alle nparticles wordt aangepakt, die wordt gevuld met uniform verdeelde randomgetallen tussen 0 en 1. Hier worden dus twee vliegen in één klap geslagen: de datastructuur wordt aangemaakt, en meteen geïnitialiseerd met de startwaarden voor de simulatie.

5.1.2 Het Stappenplan

Bij het opstellen van de stappen om de oplossing te realiseren zal van te voren duidelijk zijn dat er niet één unieke oplossing is: smaak, stijl en mode zullen in meerdere of mindere mate hun invloed doen gelden. Als vuistregel geldt dat het goed is om het probleem op te breken in een (groot) aantal kleine functies. Die kleine functies hebben het voordeel dat zij goed overzichtelijk zijn en hergebruik van code makkelijker mogelijk maken. Daarnaast stellen zij de onderliggende compiler of interpreter in staat om efficiëntere code te ontwerpen. Meestal helpt het ook om te proberen het probleem in woorden te formuleren. In ons geval zou dat tot de volgende structuur leiden:

- initialiseer het systeem, dwz kies de grootte van het vat, geef elk deeltje een plaats en een snelheid (*init*)
- herhaal voor alle tijdstappen die je wilt maken het volgende
  - herhaal voor alle deeltjes die er zijn het volgende
    - Bereken de nieuwe positie
    - Controleer of de nieuwe positie binnen het vat zit. Zo ja neem het volgende deeltje. Zo nee corrigeer de positie en de snelheid en verhoog de counter van de wand waarmee gebotst is.
  - kijk of je voldoende tijdstappen gemaakt hebt om een druk te kunnen printen
- Kijk of je voldoende stappen gemaakt hebt om te kunnen stoppen.

5.1.3 De hulpfuncties

Het bovengenoemde stappenplan leidt tot de volgende hulpfuncties

5.1.3.1 Step

Step verplaatst het deeltje door toepassing van de vergelijking (1)

```
function q=step(p,v,deltat)
q=p+deltat*v;
```

Merk op dat deze functie verschillende vector argumenten kan hebben: zowel de inputs p en v als de output q kunnen vectoren zijn. Dit wordt straks bij de aanroep van step duidelijker.

5.1.3.2 InBox

InBox controleert of een deeltje in het vierkant van het vat besloten is. Vanwege de simpele geometrische vorm is deze controle vrijwel triviaal. De logische operator *&&* voert de *and* operatie uit. De expressie die bij de *if* opgegeven wordt is dus slechts dan waar als alle sub-expressies waar zijn.

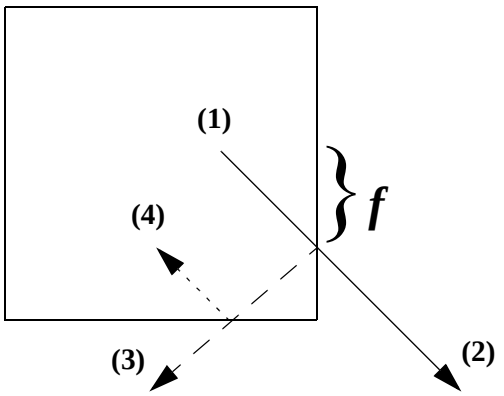
```
function out=inbox(x,y,llx,lly,urx,ury)

if (x>llx&&x<urx&&y>lly&&y<ury)
   out=1;
else
   out=0;
end
```

Voor de output wordt aangesloten bij een conventie van o.a. C dat een waarde ongelijk aan nul als *true* gezien wordt en dat nul als *false* gerekend wordt.

5.1.3.3 Correct

Correct is de meest gecompliceerde routine. Deze moet het botsen van de deeltjes met de wand in rekening brengen. In deze implementatie wordt er 'hard' vanuit gegaan dat we te maken hebben met een vierkant (of rechthoek). In het code voorbeeld geven we alleen de code voor het corrigeren van de x coördinaat. Hier wordt gebruik gemaakt van een conditionele herhaling (*while)* waarbij de voorwaarde de positie van het deeltje is. Zolang de nieuwe positie nog buiten het vierkant valt (~inbox, waarbij ~ staat voor **not**) moet er verder gecorrigeerd worden. Een nadeel van deze constructie is dat van te voren niet duidelijk is of het algoritme termineert en de *while* loop dus ooit zal eindigen. In een robuust programma zou daarop gecontroleerd moeten worden. In plaats van de *while* constructie zou men ook een recursieve benadering kunnen kiezen..

```
function [x,y,newx,newy,vx,vy,pres]= ...
          correct(x,y,newx,newy,vx,vy,pres,llx,lly,urx,ury)

while (~inbox(newx,newy,llx,lly,urx,ury))
   if (newx>urx)
      f=(newy-y)*(urx-x)/(newx-x);
      if(y+f>lly)&&(y+f<ury)
         y=y+f;
         x=urx;
         newx=urx-(newx-urx);
         vx=-vx;
         pres(1)=pres(1)+2*vx;
      end
   end

          .... similar stuff for the other coordinates ....
end
```



Cruciaal is steeds de berekening van het lijnstuk ***f***. Ga na dat je de geometrie daarvan begrijpt. Let op, zoals ook uit Figuur 1 blijkt, dat het niet voldoende is om alleen te testen of een enkel coördinaat groter is dan de waarde van het vierkant: de positie kan zo ver verwijderd zijn dat er met twee wanden een snijpunt is. In dat geval moeten de wanden wel in de juiste volgorde afgehandeld worden. Merk op dat hier eerst berekend wordt of het deeltje botst met de rechterwand. Dit wordt getest door te checken of de y-coördinaat bij het bereiken van urx binnen de doos ligt. Zo ja, dan wordt p->current het punt van de botsing op de rechterwand, en wordt p->new.x verkregen door spiegeling aan de rechterwand. Ga na of het algoritme dat hier staat alleen maar toepasbaar is op vierkanten, of dat voor een rechthoek een nadere aanpassing nodig zal zijn.

5.1.4 De main loop in de hoofdfunctie

De hierboven gedefinieerde functies dienen om een aantal akties te definiëren die de programmeur het mogelijk maken zijn uiteindelijke probleem neer te schrijven in een aantal korte functie aanroepen. Dit wordt gedaan in de hoofdfunctie *gasv.* Het hier gepresenteerde programma is vrijwel een letterlijke vertaling

```
function prestot=gasv(nprint,ntimes,nparticles,vmax)
% schrijf zelf commentaar achter de regels
x=unifrnd(0,1,[1 nparticles]);
y=unifrnd(0,1,[1 nparticles]);
vx=unifrnd(-vmax,vmax,[1 nparticles]);
vy=unifrnd(-vmax,vmax,[1 nparticles]);
pres=zeros(5,1);
prestot=zeros(ntimes/nprint,5);
iprint=0;
for i=1:1:ntimes
   newx=step(x,vx,.1);
   newy=step(y,vy,.1);
   for j=1:1:nparticles
      if (~inbox(newx(j),newy(j),0,0,1,1))
[x(j),y(j),newx(j),newy(j),vx(j),vy(j),pres]= ...
correct(x(j),y(j),newx(j),newy(j),vx(j),vy(j),pres,0,0,1,1);
      end
   end
   x=newx;
   y=newy;
   if (~mod(i,nprint))
      iprint=iprint+1;
      prestot(iprint,:)=pres(:);
      prestot(iprint,5)=mean((abs(pres(1:4)))); %calculate mean pressure over 4 walls
      pres=zeros(5,1);
   end
end
```

van het stappenplan dat in 5.1.2 als uitgangspunt neergeschreven was. Een paar specifieke constructies verdienen nog even de aandacht. De tilde '~' in de constructie *if(~inbox) ....* staat voor de logische 'not' operator en zou uitgesproken moeten worden als *'if not inbox '* . Functioneel zal de bedoeling duidelijk zijn.
Het hier beschreven programma zou men kunnen zien als een één laags oplossing. Voor complexere problemen zijn vaak (veel) meer lagen nodig. Die hoeven lang niet altijd door de gebruiker geschreven te worden maar kunnen ook geleverd worden door zogenaamde bibliotheken. Van de functie in die bibliotheken is het voldoende om te weten hoe hun *API (A*pplication *P*rogramming *I*nterface) gedefinieerd is. Een voorbeeld hiervan is de Matlab functie **unifrnd**. De gebruiker hoeft deze functie niet zelf te schrijven; het is voldoende om te weten welke argumenten die functie heeft.
Ga zelf na dat je begrijpt wat er in de output **prestot** staat.

5.1.5 Validatie

Bij de validatie wordt de correctheid van het programma geverifieerd. Het zal duidelijk zijn dat deze stap niet alleen nodig is voor Matlab-programma's. Verificatie kan op een groot aantal manieren plaats vinden. Door compilers en interpreters (ook die van Matlab of Mathematica) wordt altijd op de grammaticale correctheid van de programma's gecontroleerd. Dat is wel een nodige maar geen voldoende voorwaarde voor een goed draaiend programma. Als in het bovenstaande programma bij voorbeeld het ~vergeten wordt in de constructie *if(~inbox ...* dan zal het programma nog steeds grammaticaal correct zijn maar naar de werking verkeerde resultaten opleveren.

De validatie van dit programma zullen wij op globale fysische overwegingen doen. De beschrijving van het ideale gas moet in elk geval voldoen aan de gas wet *PV=constant*. Voorts moet de variatie in de druk op een enkele wand groter zijn dan de variatie in de gemiddelde druk. Men kan eenvoudig Matlab zelf of een ander programma als Excel gebruiken om de output te visualiseren.

```
plot(prestot(:,5));
      xlabel('Time')
      ylabel('Pressure(Pa)')
      title('Pressure')
```

Eindopdracht Module 1 van Modelleren, Programmeren en Simuleren

Doel van deze opgave is om ervaring te krijgen met een simulatie programma van een ideaal gas dat in Matlab geschreven is. Een primitieve functie is geschreven en besproken in het college, die naar hartenlust uitgebreid en onderzocht kan worden.

1. Bestudeer het programma, schrijf zelf commentaar (bekijk de help pagina's van functies die nieuw voor je zijn) en bepaal waar de volgende variabelen een waarde krijgen:

- De afmetingen van het vat dat gebruikt wordt. Hoe kun je zelf een rechthoekig vat maken?
- Hoe kun je zelf de snelheidsverdeling aanpassen?
- Bepaal ook waar de druk berekend wordt. Welke kritiek kun je fysisch hebben op de manier waarop de druk berekend wordt?
- De functie Correct is de belangrijkste functie van het programma. In deze functie worden vier verschillende gevallen onderscheiden. Laat in een tekening zien met welke omstandigheden deze gevallen corresponderen.

2. Hoe kun je de output van het programma wegschrijven naar een file?

- Maak een run met het programma en maak een plot van de druk als functie van de tijd. Het programma geeft 5 variabelen per tijdsstap: de druk op de diverse wanden en de gemiddelde druk. Vergelijk de gemiddelde waarden.
- Maak een grafiek van de spreiding van de gemiddelde druk als functie van het aantal deeltjes dat je gebruikt. Klopt dit met wat je verwacht?
- Controleer of het programma ook goed werkt voor een rechthoekige doos.
- Onderzoek wat er gebeurt als je de snelheid van de deeltjes kleiner of groter maakt ten opzichte van de tijdstap in je programma. Klopt dit met wat je verwacht?
- Bestudeer het notebook over de conversie naar fysische eenheden. Converteer je resultaten naar fysische eenheden. Kies daarvoor een bepaald volume en snelheid. Wat zijn dan de druk en temperatuur?

3. Bedenk een manier om het langzaam leeglopen van het vat te imiteren door deze code en laat grafisch het resultaat zien. Als je wilt debuggen, click op je op de regel in het .m file, en wanneer je daarna het programma runt, kun je met de muis erovergaand de waarden van de variabelen zien. Naast de Run button zitten 7 debug icoontjes.

4. Verzin zelf nog iets interessants om te testen met dit programma!
   Enkele suggesties:

- Breid het programma uit naar 3 dimensies (dus botsingen met 6 wanden)
- Gebruik een andere verdeling van de beginsnelheden (bv een normale

verdeling,
http://en.wikipedia.org/wiki/Maxwell%E2%80%93Boltzmann_distribution#Distribution_for_the_velocity_vector
)

- Onderzoek de gaswet pV=nRT
- Visualiseer de bewegingen van de gasdeeltjes (Matlab functies *scatter()* of *scatter3()*, en *pause()*)
- Zou je deze simulatie van een ideaal gas ook in Mathematica kunnen? Met of zonder For loops?
- Hoe zou je de beweging van een cylinder met een zuiger simuleren?
- Onderzoek correlaties tussen drukken op tegenoverliggende wanden
- Hoe zou je een temperatuur kunnen definiëren ?

Schrijf een leesbaar verslag van deze eindopdracht van Module 1 (minstens 3 A4, exclusief illustraties). Dit verslag (svp schriftelijk indienen, met daarbij de code) zal als eerste besproken worden op het mondeling. Als je tijdens het mondeling van Module 1 iets wilt demonstreren op je laptop, bereid het dan goed voor. Daarna zal een deel van je uitwerkingen op de laptop besproken worden. Het mondeling van Module 1 duurt 15 minuten, met een mogelijke uitloop van 5 minuten.

Conversion of gas simulation results to physical units

compute the velocity of He at 0 °C

equipartition

In[9]:= **m vx^2 == kB T**

Out[9]= $m\ vx^2 == kB\ T$

In[1]:= **Needs["PhysicalConstants`"]**

In[2]:= **Needs["Units`"]**

In[4]:= **BoltzmannConstant**

Out[4]= $\frac{1.38065 \times 10^{-23}\ \text{Joule}}{\text{Kelvin}}$

In[26]:= **T0K = ConvertTemperature[0, Celsius, Kelvin]**

Out[26]= $\frac{5463}{20}$

http : // en.wikipedia.org/wiki/Atomic_mass _unit

In[24]:= **Convert[AtomicMassUnit, Kilogram]**

Out[24]= $1.66054 \times 10^{-27}\ \text{Kilogram}$

compute the mass of He (kg)

In[10]:= **ElementData["Helium", "AtomicWeight"]**

Out[10]= 4.002602

In[16]:= **mHe = Convert[ElementData["Helium", "AtomicWeight"] AtomicMassUnit, Kilogram]**

Out[16]= $6.64648 \times 10^{-27}\ \text{Kilogram}$

In[17]:= **vxHe0K = Sqrt[(BoltzmannConstant T0K Kelvin) / mHe]**

Out[17]= $753.263 \sqrt{\frac{\text{Joule}}{\text{Kilogram}}}$

In[22]:= **vxHe0Km = Convert[vxHe0K, Meter / Second]**

Out[22]= $\frac{753.263\ \text{Meter}}{\text{Second}}$

In[19]:= **Convert[vxHe0K, Kilo Meter / Hour]**

Out[19]= $\frac{2711.75\ \text{Kilo Meter}}{\text{Hour}}$

In[20]:= **SpeedOfSound**

Out[20]= $\frac{340.292\ \text{Meter}}{\text{Second}}$

In[23]:= **vxHe0Km / SpeedOfSound**

Out[23]= 2.21358

http : // en.wikipedia.org/wiki/Molar_volume
The molar volume of an ideal gas at 1 atmosphere of pressure and at 0 °C is

In[5]:= **MolarVolume**

Out[5]= $\frac{0.022414\ \text{Meter}^3}{\text{Mole}}$

In[3]:= **AvogadroConstant**

Out[3]= $\frac{6.02214 \times 10^{23}}{\text{Mole}}$

compute the volume of 1000 particles of an ideal gas at 1 atmosphere of pressure and at 0 °C

In[25]:= **V1000 = 1000 MolarVolume / AvogadroConstant**

Out[25]= $3.72193 \times 10^{-23}\ \text{Meter}^3$

Vergelijking PSEs (vul zelf aan !)

	Matlab	Mathematica
array of vector (homogeen)	B=[1 2 3]	B={1,2,3}
lijst (inhomogeen)	A={'Hello',magic(3)}	B={1,a,Sin[x]}
element uit array of lijst	B(3) A(2)	B[[3]]
2D array of matrix	[1 2; 3 4]	{{1,2},{3,4}}
simpele functies	function out=myfunc(a) out=sin(a)/a^3	Myfunc[a_]:=Sin[a]/a^3
opmerking	in .m file met **dezelfde** naam	Interne Mathematica functies beginnen met een Hoofdletter
input argument	tussen ( )	tussen [ ]
loop	for i=1:10 expr(i) end	Table[expr[i],{i,1,10}]
if	if test1 expr1 elseif test2 expr2 else expr3 end	If[test1,expr1, If[test2,expr2,expr3]]
test	n==1	n==1
Globale variabelen	global … Workspace	??Global`*

Globale variabelen verwijderen	clear all clear global clear	`Remove["Global`*"]` Evaluation/Quit/Kernel Clear[B] B=. (Unset)
Afbreken	^C	Evaluation/Abort Evaluation
control	via Editor	Evaluation Menu
debug	via Editor	Trace[ ]
Help	Select+F1	Select+F1
Lijst maken	for … end while… end	Table
comment	%	Alt-7 (=Format/Style/Text)
don't print	expr;	expr;
Compound Expression	expr1;expr2	expr1;expr2
timing	tic;expr;toc	Timing[ ]
vermenigvuldigen	*	Spatie of *