Introductie Computergebruik: Computeralgebra

Wieb Bosma

November/December 2001

Inhoudsopgave

Voorwoord

Dit is een versie van het begin van een handleiding voor het gebruik van Maple voor eerstejaars wiskundestudenten. Het beoogt een eerste snelle indruk te geven van wat een computeralgebrapakket als Maple vermag, en hoe je er zelf mee aan de slag kunt.

Hoofdstuk 1

Inleiding

1.1 Computeralgebra

Een *computeralgebra*systeem is een pakket van programma's waarmee je op een computer 'algebra' kunt doen. In de praktijk moet je het begrip algebra ruim opvatten, maar het gebruik in deze context is vooral bedoeld om te benadrukken dat we algebraïsche manipulaties willen doen op wiskundige objecten, en niet zozeer numerieke berekeningen. Als synoniem wordt wel de uitdrukking *symbolisch rekenen* gebruikt. Om de gedachten te bepalen: computeralgebra legt de nadruk op het 'formele' rekenen met *discrete* objecten (zoals gehele getallen, en breuken, maar vooral abstractere objecten als polynomen, elementen in abstracte groepen, ringen, enz.) dan op bijvoorbeeld het rekenen met reële of complexe getallen.

In de praktijk hebben de meeste computeralgebrapakketten *ook* allerhande faciliteiten om met (functies van) reële getallen te werken, simpelweg omdat dat in veel toepassingen onontbeerlijk is. Wij zullen die aspecten ook beslist niet willen negeren; aanvankelijk zullen we het pakket dat we gaan gebruiken vooral zien als een (zeer) veredelde calculator. Maar hoe verder je komt, hoe meer behoefte er ook gaat komen aan de meer symbolische kant. Toch nog één eenvoudige illustratie voor het verschil tussen computeralgebra en numerieke wiskunde. Wanneer de reële functie f gedefinieerd is door $f(x) = \sin^{-1}(x)$ interesseert computer algebra zich bijvoorbeeld voor de vraag naar een primitieve functie $\int f(x)\mathrm{d}x$ van f, terwijl de numericus geïnteresseerd zou kunnen zijn in de waarde van $\int_{1/2}^{2/3} f(x)\mathrm{d}x$. Precieser gezegd: computer algebra houdt zich hier bezig met het zoeken naar een algoritme om een primitieve te vinden, terwijl de numerieke wiskunde een goede methode probeert te vinden om een bepaalde integraal nauwkeurig te berekenen. Natuurlijk, als een primitieve eenvoudig gevonden kan worden zal de numericus daar zo mogelijk dankbaar gebruik van maken; maar als dat niet zo is, is het bepalen van de waarde nog steeds een interessant probleem.

Voorbeeld

In Maple kun je de beide vragen als volgt oplossen, bijvoorbeeld.

```
> int(arcsin(x), x);
                                                    2
                          x arcsin(x) + sqrt(1 - x )

> evalf(int(arcsin(x), x=1/2..2/3));

                                .1040163052
```

1.2 Wat valt er te leren?

Wanneer je voor het eerst een computeralgebrapakket wilt gebruiken om een wiskundig probleem op te lossen, doet zich een aantal nieuwe problemen voor.

1. de keuze van het pakket;
2. het opstarten (en afsluiten), en het invoeren van commando's en data;
3. het leren (gebruiken) van de taal van het pakket;

4. het vinden van de juiste commando's en datastrukturen;
5. het vertalen van je probleem van wiskunde naar de programmeertaal.

We zullen op die vragen in deze inleiding in de aangegeven volgorde kort proberen in te gaan.

1.3 Maple

De eerste vraag, welk pakket gebruikt gaat worden, is voorlopig beantwoord door ons: we gebruiken Maple in dit eerste jaar. Maple is een van de grote computeralgebrapakketten, naast bijvoorbeeld Mathematica. Deze twee zijn de meest gebruikte algemene systemen. Van de twee is Mathematica (waarvoor in het Mathematisch Instituut een oude versie te gebruiken is) aanzienlijk uitgebreider wat betreft toepassingsgebieden, maar richt Maple zich wat meer op de wiskundige gebruiker. Er zijn nog andere 'grote' systemen, zoals Reduce en Macsyma, die ook al een lange historie hebben. Bovendien zijn er heel veel andere systemen waarvan sommige meer numeriek gericht zijn (Matlab), andere meer algebraïsch (Magma, Axioma), en er zijn heel veel kleinere systemen die zich richten op een beperkter (wiskundig) gebied.

Eén van de redenen waarom Maple gebruikt wordt in het eerste jaar in Nijmegen is dat het een redelijk groot gebied van de wiskunde bestrijkt, en daarom aan vrij veel wiskundeafdelingen gebruikt wordt; dat betekent weer dat de kans relatief groot is dat je later te maken krijgt met Maple en gebruikers ervan.

Een omvangrijk computeralgebrapakket als Maple bestaat uit diverse componenten. In de eerste plaats is er natuurlijk de kern van het 'programma' (eigenlijk heel samenstelsel van programma's en bibliotheken) dat het uiteindelijke rekenwerk voor je doet. Daar bovenop is een 'interface' gebouwd, die het mogelijk maakt je wiskundige problemen in te voeren. Naast een grafische component — hoe ziet de omgeving waarin je werkt met Maple eruit, hoe ziet invoer en uitvoer er uit — is een belangrijk onderdeel daarvan de programmeertaal, de taal waarin je met de computer communiceert. Die taal is van hogere orde dan een gewone programmeertaal zoals C++, C, Pascal, enzovoorts, in de zin dat je allerhande geavanceerde objecten en constructies ter beschikking hebt zonder ze zelf te hoeven implementeren (bijvoorbeeld getallen van willekeurige lengte). Dat is een groot voordeel; maar omdat alle systemen er tamelijk verschillende 'talen' op na houden, heeft het als nadeel dat je voor elk pakket de problemen 3 en 4 uit het vorige hoofdstukje opnieuw moet oplossen. Voor Maple geldt bovendien nog dat de taal (in tegenstelling tot die van Mathematica bijvoorbeeld) niet erg 'netjes' is; veel dingen zijn niet erg intuïtief en voor de hand liggend, soms is het nodig om iets van de interne organisatie te weten om vreemde bij-effecten te verklaren, en met veel objecten zijn operaties toegestaan zonder dat duidelijk is of dat wiskundig gezien zinnig is. Dat maakt het belangrijk dat je zelf goed weet waarmee je bezig bent!

In tegenstelling tot een rekenmachine zijn de grote systemen in staat uitvoer in diverse vormen af te leveren, vooral wanneer je ze gebruikt in een 'grafische' omgeving met 'vensters', bijvoorbeeld een X-achtige omgeving onder UNIX (zoals wij doen). Antwoorden kunnen dan niet alleen uit platte tekst bestaan, maar bijvoorbeeld uit mooi getypesette formules, maar ook uit plaatjes in diverse formaten (zodat ze rechtstreeks naar een printer kunnen), of in HTML of TEX-achtige vorm. Bovendien is het mogelijk om geïntegreerde documenten te maken of gebruiken, waar tekst en te activeren commando's door elkaar gebruikt kunnen worden ('worksheets' in Maple, 'notebooks' in Mathematica, etc.). Bovendien hebben deze systemen uitgebreide 'help'-systemen ingebouwd.

1.4 Vertalen

Het probleem om een wiskundige opgave te vertalen naar een voor de computer oplosbaar probleem heeft natuurlijk vele facetten. Om te beginnen zou je eigenlijk al van de vraag af willen laten hangen welk pakket je kiest, omdat soms voor specifieke vragen speciale software is ontwikkeld. Is, zoals voor ons, die keuze eenmaal bepaald, dan moet je de wiskundige objecten in Maple-objecten omzetten, je afvragen op welke manier je probleem in principe opgelost zou kunnen worden, en dan de juiste commando's toepassen om dat door Maple uit te laten voeren. Kortom alle andere vragen uit het rijtje komen aan bod. Bovendien moet je, als je een antwoord krijgt, dat antwoord weer weten te interpreteren in de context waarmee je begon. Een belangrijke regel is dat computeralgebrapakketten, vooral bij meer geavanceerde problemen, wel eens foute of incomplete antwoorden kunnen opleveren. Het is dus van belang om na te gaan dat het gevonden antwoord zinnig is. Bovendien komt het vaak voor dat je langer op een antwoord moet wachten dan je lief is. In dat geval verdient het aanbeveling te zoeken naar een manier om Maple te helpen, door een eenvoudigere methode te kiezen, of het probleem in stapjes op te lossen zodat je kunt zien waar Maple moeite mee heeft. Soms zal dan blijken dat je vraag niet goed (genoeg) gesteld was. Het kan ook zijn dat je beter een numerieke dan een symbolische methode kunt proberen, enzovoorts. Natuurlijk weet je in het begin voor veel van de moeilijkheden die zich voordoen geen oplossing.

1.5 Doel

De bedoeling is dat je na deze inleidende werkcolleges een aantal eenvoudige taken met Maple kunt uitvoeren, zoals het doen van sommige standaardberekeningen die in Calculus of Lineaire Algebra voorkomen, of het maken van een plotje bij verzamelde data. Bovendien is het de bedoeling dat je tegen die tijd zo goed de weg kent dat je zonder begeleiding in staat bent steeds ingewikkeldere programma's te schrijven.

In de volgende hoofdstukken worden heel veel commando's alleen maar kort genoemd, zonder precieze uitleg over het gebruik. De bedoeling is dat de naam je voldoende houvast geeft om met de online help verder uit te vinden hoe de functie van dienst kan zijn.

Voorbeeld

In het tentamen Lineaire Algebra 1 kwam als opgave voor het oplossen van het lineaire stelsel vergelijkingen:

$$\begin{aligned} x_1 + x_2 + x_3 &= 2 \\ -x_1 + x_2 + 2x_3 &= -1 \\ x_1 \quad\;\; + x_3 &= 0. \end{aligned}$$

Dat is met Maple ook heel eenvoudig; maar het probleem is om de juiste commando's te vinden.

```
> with(linalg):
> A := matrix(3, 3, [[1, 1, 1],[-1, 1, 2], [1,0,1]]):
> v := vector(3, [2, -1, 0]);
                              v := [2, -1, 0]
> linsolve(A, v);
                              [1, 2, -1]
```

Vervolgens werd gevraagd voor welke waarden van α het stelsel

$$\begin{aligned} x_1 + x_2 + x_3 &= 2 \\ -x_1 + x_2 + 2x_3 &= -1 \\ x_1 + \alpha x_2 + x_3 &= 0. \end{aligned}$$

geen oplossingen heeft. Ook daarmee heeft Maple geen moeite:


```
> with(linalg):
> A := matrix(3, 3, [[1, 1, 1],[-1, 1, 2], [1,a,1]]):
> v := vector(3, [2, -1, 0]);
                              v := [2, -1, 0]
> linsolve(A, v);
                              linsolve(A, v);
```

waarbij uit de output duidelijk blijkt dat er alleen geen oplossing is als $\alpha = 1$.

Voorbeeld

In het Analyseboek wordt bij Hoofdstuk 4 gevraagd een N te bepalen zodat voor alle $n \geq N$ geldt

$$2^{-n} + 3^{-n} + 4^{-n} < \frac{1}{365}.$$

In Maple kan dat bijvoorbeeld zo:

```
> n := 0:
> while (2^(-n)+3^(-n)+4^(-n) >= 1/365) do
>  n := n+1:
> od:
> n;
                              9
```

Hoofdstuk 2

De eerste stappen

2.1 Begin en eind

2.1.1 Begin

Om Maple te activeren kun je vanuit een shell het commando `xmaple` uitvoeren, of je kunt onder de *math* tools (die je window-manager vast ergens vertoont) Maple selecteren. Als je alleen een gewoon terminalscherm tot je beschikking hebt kun je ook met `maple` nog een beperkte 'platte-tekst'-versie opstarten; we zullen er hier van uitgaan dat je de X-versie gebruikt.

Als je zo een Maple-sessie hebt opgestart, verschijnt na enige tijd op je scherm een Maple-venster, waarin in ieder geval een rijtje menu's zichtbaar zal zijn, met daaronder een balk met knoppen, en verder een ander venster met de prompt `>`. Dat is het 'werkblad' (worksheet) waarin commando's kunnen worden ingevoerd.

Een ander uiterst belangrijk hulpmiddel is de help-faciliteit. Er is een speciaal `Help` menu (helemaal rechtsboven), en je kunt hulp krijgen over functies door `?`*functie* te typen. Dan moet je natuurlijk wel eerst weten dat *functie* een geldig commando is! In veel gevallen kun je ook verder door onderstreepte woorden aan te klikken.

Eén manier om Maple te gebruiken, is als een calculator, waar je het antwoord op een vraag direkt hoopt terug te krijgen — dat is *interactief* gebruik. De kunst is dan de juiste commando's te vinden om het antwoord te vinden.

Door de werkbladen is het verschil tussen een interactieve sessie en het programmeren met behulp van bestanden vervaagd: je kunt namelijk navigeren in het werkblad, fouten verbeteren en de code opnieuw laten executeren. Je moet daarbij in de gaten houden dat door het navigeren in het venster de volgorde van de uitgevoerde commando's helemaal niet meer overeen hoeft te komen met de volgorde waarin ze onder elkaar staan!
Werkbladen hoeven trouwens helemaal niet uitsluitend uit Maple-commando's te bestaan: er kunnen hele stukken gewone tekst tussen gevoegd worden. Op de knoppenbalk kun je allerhande hulpmiddelen daartoe terugvinden. In het begin is het niet zo belangrijk een dergelijk geïntegreerd document te kunnen maken, maar ze worden wel vaak gebruikt om je nieuwe stof eigen te laten maken.

Opgave

Start Maple. Zoek hulp omtrent werkbladen, en sluit dat document na enig neuzen.

`xmaple` `Help` `?` `>`

2.1.2 Menu's en Eind

Onder het `File` menu zie je de `Exit` optie die je in staat stelt deze Maple-sessie weer af te sluiten. (Je kunt datzelfde ook wat minder elegant bereiken door het `xmaple` proces te `killen`.)

Andere belangrijke elementen zijn:

- de `stop` knop op de knoppenbalk, die (soms) helpt om een te lange berekening te onderbreken;
- de `Open` optie onder het `File` menu, waarmee je al bestaande documenten in Maple kunt openen (bijvoorbeeld een eerder geschreven programma of een worksheet);
- de `Save` optie onder hetzelfde `File` menu, waarmee je je huidige document als worksheet kunt opslaan;
- de `Output Display` optie onder het `Options` menu, waarmee je de output er anders kunt laten uitzien.
- de `Close` optie onder het menu dat bij het werkblad hoort. Hetzelfde kun je ook bereiken door `quit;` in dat werkblad te typen.

Bij niet-interactief gebruik schrijf je een kort of lang programma en lees je dat in. Daartoe kun je naast de menu-opties onder andere het commando `read("`*filenaam*`")` *gebruiken. Let op de quotes, die ervoor zorgen dat Maple het argument als 'naam' van een file leest. Dat is vooral van belang als de naam speciale karakters als* `.` *en* `/` *bevat (die Maple zonder quotes verkeerd interpreteert). Je kunt een pad aangeven (op de gebruikelijke UNIX wijze). Om een sessie op te slaan (uit een werkblad) kun je het commando* `save("`*filenaam*`")` *gebruiken. Door de filenaam van* `.m` *te voorzien slaat Maple een file op (of leest een file) in speciaal 'internal format', waarna de file niet voor de gewone mens leesbaar is. Ook is er de* `.mws` *uitgang voor Maple werkbladen. Voorts zijn er* `writeto` *en* `appendto` *die uitvoer voortaan naar een te specificeren file in plaats van naar het scherm sturen; ook de input wordt daarheen ge-echood.*

Opgave

In `/vol/wiskalg` is een speciale directory `/mapleonderwijs/wiskunde` waarin een subdirectory `/compgeb` met een voorbeeldwerkblad `vbd.mws` staat. Lees dat in; voer de (rode) opdrachten uit (door op de regel een `<enter>` te geven. Maak aanpassingen zodat de uitvoer er net zo uit komt te zien als in de tekst van voorbeeld 1.1 boven, en schrijf het resultaat naar een bestand in je eigen home-directory.

`Exit` `stop` `quit` `Close` `read` `save` `writeto` `appendto`

2.2 Basisbegrippen

2.2.1 Gehele getallen, breuken

Het is makkelijk om gehele getallen, breuken, en reële getallen te maken en ermee te rekenen, met de operatoren `+, -, *, /, ^` die de gebruikelijke betekenis hebben. Wel moet je soms wat meer haakjes zetten dan je zou hopen, vooral omdat je niet altijd weet in welke volgorde Maple operaties uitvoert (maar zie `operators[precedence]`) voor de regels!), en sommige dingen niet goed gedefinieerd zijn (volgens Maple):

```
> 2^3^4;
`^` unexpected
>  (3/4)^(-3);
                                    64
                                    --
                                    27
```

Rationale getallen maak je dus met `/`. Teller en noemer (na simplificatie!) vind je terug met `numer` en `denom`.

Elke opdracht moet afgesloten worden met `;` of met `:` (waarmee output wordt onderdrukt). Nuttige functies voor gehele getallen zijn `iquo`, `irem` waarmee je voor gehele a en b gehele getallen q (quotiënt) en r (rest) kunt vinden zodat $a = qb + r$, en $0 \leq |r| < |b|$. Ook `igcd` voor de grootste gemene deler van gehele getallen is handig, evenals de uitgebreide versie `igcdex` die bij gehele a, b niet alleen de ggd d vindt, maar ook gehele getallen u, v met $ua + vb = d$. Maple rekent met gehele getallen waarvan de lengte alleen begrensd is door de geheugenruimte (en de tijd nodig om ze uit te rekenen). Met `mod` kun je rekenen modulo een getal m.

Gelijkheden en ongelijkheden maak je met `=`, `<>`, `>`, `>=`, enzovoorts (voor 'is gelijk', 'is ongelijk', 'is groter dan', en 'is groter dan of gelijk aan').

Een opdracht mag meerdere regels beslaan. Er mogen ook meerdere opdrachten op een enkele regel. Als Maple een fout meldt kun je altijd terug gaan met de muis en verbeteringen aanbrengen.
Het is belangrijk om te weten dat `=` *en* `:=` *in Maple geheel verschillende dingen zijn: met* `=` *maak je een vergelijking, bijvoorbeeld met een variabele, die je vervolgens zou kunnen proberen op te lossen. Met* `:=` *ken je een waarde toe aan een variabele — je geeft als het ware een naam (links) aan een waarde (rechts) zoals in* `x := 3*2-1;` *Daarover meer in 2.2.3. Je kunt van een vergelijking ook nagaan of hij 'geldt' (bijvoorbeeld als links en rechts getallen staan) door* `evalb` *toe te passen ('evalueer tot Boolese waarde', dus waar of onwaar).*

Opgave

Vind de grootste gemene deler d van $a = 1234567890$ en $b = 987654321$, alsmede u, v zodat $d = ua + vb$, en ga na dat de gelijkheid geldt.

`; :   + - * / ^   = <> >   iquo irem   mod   igcd igcdex   numer denom   evalb`

2.2.2 Reële getallen

Reële getallen kun je invoeren door decimalen `0-9` en een decimale `.` te gebruiken; je krijgt ze natuurlijk ook als resultaat van bepaalde uitdrukkingen. Maple evalueert het resultaat van een 'uitdrukking' niet altijd (dat wil zeggen, hij voert de berekening niet zomaar helemaal uit), maar je kunt forceren dat dat wel gebeurt met `evalf`. Je kunt daarmee dus ook bijvoorbeeld een rationaal getal omzetten in een reëel getal. Er zijn veel reële functies ingebouwd, zoals `abs` voor absolute waarde, de goniometrische functies `sin`, `cos`, `tan`, en hun inversen zoals `arcsin`. Verder `Sqrt` voor de wortel, exponentiële functies `exp` (voor e^x), `ln`, `log10` en `log[b]` voor natuurlijke,basis-10 en basis-b logaritmen.

Natuurlijk wordt er maar met eindig veel decimalen achter de komma gerekend; dat aantal kun je wijzigen met `Digits` (en is aan het begin meestal 10).

Maple is ook heel redelijk in staat *limieten* van functies in punten te bepalen, met `limit`. Daarbij kan probleemloos van oneindig gebruik gemaakt worden met `infinity`, en je kunt ook aangeven van welke kant je wilt benaderen.

```
> limit( 1/x^3, x=infinity );
                                    0

> limit( 1/x^3, x=0 );
                                    undefined

> limit( 1/x^3, x=0, left );
                                    -infinity
```

Maple kent een aantal reële en complexe constanten, zoals `I` *als basis voor de complexe getallen, en* `Pi` *voor π. Je kunt zelf ervoor zorgen dat Maple namen herkent (en gebruikt) met behulp van* `alias`.
Het is belangrijk het verschil in te zien tussen `17.0` *(reëel getal) en* `17` *(geheel getal). Een functie als* `evalf` *kan rationale getallen in reële getallen converteren; je kunt met* `whattype` *zien wat het type van een object is. Ook is er* `type(`x`, `t`)` *die een Boolese waarde geeft, namelijk of x van type t is.*

Opgave

Ga na dat met de oplossing die in voorbeeld 1.5 gevonden werd aan de gevraagde vergelijking wordt voldaan.

Opgave

Geef het getal e zijn gebruikelijke wiskundige naam in Maple met `alias`, en bepaal met behulp hiervan 50 decimalen van $\sin(e^{-2\pi i})$ waar $i^2 = -1$. Achterhaal ook de types van e, π, en i.

`.   evalf   abs   limit   Sqrt   sin cos arcsin   exp log ln`

2.2.3 Variabelen

Om werkelijk te kunnen rekenen met objecten zul je ze namen moeten geven: met `:=` kun je een *waarde* aan een *variabele* toekennen. De huidige waarde van de variabele `a` kun je met `print(a);` of gewoon `a;` zien. Met `lprint` wordt alles gewoon op een regel geplaatst.

Er is een klein lijstje van *gereserveerde woorden* in Maple, die een betekenis hebben in de programmeertaal en daarom niet als naam van een variabele gebruikt mogen worden, zoals `for` en `end` en `if`. Namen van variabelen mogen overigens uit willekeurig lange woorden van letters en cijfers bestaan (maar het eerste karakter mag geen cijfer zijn); Maple maakt onderscheid tussen hoofdletters en kleine letters.

Een speciale variabele is `%`. De waarde hiervan is steeds het resultaat van de voorgaande opdracht, ook al heb je dat resultaat zelf ook al aan een variabele toegewezen. Dit maakt het vooral gemakkelijk om, als je bent vergeten het resultaat van een berekening een naam te geven, zonder het opnieuw uit te rekenen toch aan dat resultaat te refereren. Met `%%` en `%%%` kun je een paar stappen terug.

Door `a := 'a';` *kun je alle toewijzingen aan de variabele* `a` *weer ongedaan maken. Na een toewijzing verwijst de variabele naar de waarde; deze waarde mag zelf ook weer een (uitdrukking in) andere variabele zijn, enzovoorts. Het is belangrijk om te weten bij toekenning aan een variabele, zeg* `y`*, de waarde die is van de uitdrukking waarnaar verwezen wordt; als die uitdrukking zelf een variabele is, zeg* `x`*, waaraan nog geen waarde is toegekend, verandert de waarde van y mee met die van x!*

```
> y := x;  x := 2;  z := x;
> x := 5;  x, y, z;
                                  y := x
                                  x := 2
                                  z := 2
                                  x := 5
                                  5, 5, 2
```

Het verschil is dat de variabele `y` *steeds naar* `x` *blijft verwijzen, en dat* `z` *verwijst naar de waarde van* `x` *op het moment dat* `z` *werd ingevoerd.*

Opgave *Voorspel wat het resultaat van* `x := 'x':  x, y, z;` *na het voorgaande is.*

Met de functie `eval` *kun je kijken waarnaar variabelen verwijzen; het tweede argument vertelt hoe veel stappen je de evaluatie wilt uitvoeren.*

```
> eval(y, 1); eval(y, 2); eval(z, 1);
                                  x   5    2
```

Quotes *zorgen dat de uitdrukking die er tussen staat niet geëvalueerd wordt. Daarom is het resultaat van* `'x'` *de variabele zelf, zodat* `x := 'x'` *alle toewijzingen aan* `x` *ongedaan maakt! Overigens kun je met* backquotes `` ` ` `` *ervoor zorgen dat een opeenvolging van karakters als enkel symbool gebruikt kan worden, bijvoorbeeld als variabele naam; dit is zelfs mogelijk als er spaties in zitten of het een gereserveerd woord is. Dus* `` `for` `` *kan als variabelenaam gebruikt worden!* Dubbele quotes *worden gebruikt om een string in Maple te maken: dat is alleen maar een rijtje karakters dat nergens naar verwijst — die kun je bijvoorbeeld gebruiken om ergens tekst af te drukken. Met* `.` *kun je strings achter elkaar plakken (concateneren).*

Opgave

Probeer het alias uit opgave van section 2.2.2 weer ongedaan te maken.

`' ` "   eval   :=   print lprint   %   .`

2.2.4 Functies

Een functie kun je met `->` definiëren; om bijvoorbeeld met `f` de functie aan te duiden die aan een argument `x` de derde macht ervan toevoegt, doe je `f := x -> x^3;` Je kunt de functie dan *toepassen* op een argument a door `f(`a`);` Met `solve` kun je proberen een argument te vinden waarvoor de functie een gegeven waarde aanneemt, dus bijvoorbeeld `solve( f(x)=27, x );`

Het maakt bij het definiëren van `f` *niet uit of je aan* `x` *al een waarde had toegekend of niet, want Maple neemt hier een nieuwe 'lokale' variabele.*
Dit is een van de punten waar Maple je alle vrijheid laat door niet te specificeren wat het domein (of beeld) van de functie is. Je kunt er dus van alles in stoppen.

Het *samenstellen* van functies gaat met `@`, en herhaald toepassen van dezelfde functie met `@@`.

```
> f := x -> x^3;
> g := x -> sin(x);
> h := x -> 1/x;
> (f@g@h)(2);
                                              3
                                      sin(1/2)
```

Functies van meer veranderlijken kunnen net zo met `->` gemaakt worden, maar let wel op de haakjes: `cirkel := (x, y) -> x^2 + y^2;` Het resultaat is misschien niet precies wat je had verwacht, omdat Maple de waarde van sin(2) niet heeft uitgerekend.

Het is ook mogelijk om een functie te definiëren middels een uitdrukking waar een variabele in voorkomt, zoals `f := x^3;` *waarna evaluatie met* `eval` *plaatsvindt* `eval( f, x=2 );` *Met* `unapply` *kun je dan daarvan alsnog een functie maken,* `g := unapply(f, x);`
Ingewikkelder functies kun je ook programmeren als procedure met `proc`*, zoals we later zullen zien.*

Opgave

Vind alle y-coördinaten van punten op de cirkel om de oorsprong met straal 5. Probeer ook alle punten daarop te vinden waarvoor de x-coördinaat 3 is.

`->      eval      solve      @      @@`

2.2.5 Polynomen

Het creëren van polynomen is heel eenvoudig in Maple, omdat je eenvoudigweg een nieuwe onbekende x kunt invoeren, om daarmee polynomiale uitdrukkingen te maken. Zulke polynomen mogen ook in meerdere veranderlijken zijn. Niets belet je overigens om vervolgens ook te delen door zulke polynomen, en zo maak je *rationale functies*. Natuurlijk kun ook de andere gewone aritmetische operatoren loslaten op polynomen, maar je moet er daarbij op bedacht zijn dat Maple niet automatisch termen bij elkaar neemt en coëfficiënten combineert: daartoe moet je eerst `simplify` toepassen. Net zo werkt Maple niet automatisch producten uit (want dat is niet altijd een verbetering, denk aan $(x+1)^{1000}$), en moet je om dat te bereiken `expand` toepassen. Met `subs` kun je in een polynoom(vergelijking) variabelen door andere vervangen, desgewenst simultaan meerdere. Het oplossen van polynomiale vergelijkingen kan gebeuren met `solve` of `fsolve`. In het laatste geval gebeurt dit numeriek (en worden alle reële oplossingen gevonden, in het eerste geval gebeurt het symbolisch, waar mogelijk.

Differentiëren en integreren gaat met `diff` en `int` (en afkoring voor `integrate`). Met `factor` kunnen polynomen (met rationale coëfficiënten) in irreducibele factoren ontbonden worden. Voor polynomen met coëfficiënten modulo een priemgetal zijn `Factor` (voor ontbinden) en `msolve` voor het oplossen van vergelijkingen gegeven. Met `Roots` kan naar nulpunten gezocht worden.

> *Het gebruik van* `subs` *dient met enige omzichtigheid te gebeuren. Niet alleen bepaalt de syntax in welke volgorde de substituties gebeuren, maar het is ook mogelijk om constanten te substitueren door anderen; omdat elk voorkomen van de constante in de uitdrukkingen dan wordt vervangen, en het (vooral bij* 1*) niet altijd duidelijk is waar deze voorkomt, kan dit tot onbedoelde gevolgen leiden.*
> *Om een polynoom f in meerdere variabelen (zeg x en y) op te vatten als een polynoom x met coëfficiënten polynomen in y, gebruik je* `collect( f, x );` *(waarop variaties zijn om ook nog meer variabelen verder te ordenen. Overigens kun je met* `coeff` *toegang krijgen tot de coëfficiënten. Ook* `sort` *ordent de termen van polynomen.*
> *Voor rationale functies zorgt* `normal` *ervoor dat sommen van quotiënten op één noemer gebracht worden, in 'standaardvorm' zodat teller en noemer geen factor meer gemeen hebben. Toegang tot teller en noemer wordt overigens net als voor breuken door* `numer` *en* `denom` *verschaft. Soms is het ook handig om omgekeerd* breuksplitsing *toe te passen op een rationale functie r teneinde een som van niet verder vereenvoudigbare quotiënten te krijgen; daarbij helpt* `convert( r, parfrac, x );` *als x de variabele is.*
> *Met* `solve` *kunnen ook niet-polynomiale vergelijkingen (soms) worden opgelost; als het niet lukt, kan soms informatie verkregen worden door* `infolevel[solve]` *te vergroten.*

Opgave

Laat Maple de nulpunten van het algemene kwadratische polynoom ax^2+bx+c vinden. Lukt het ook met derdegraadspolynomen?

`msolve @ @@ diff int Factor Roots simplify normal collect sort`

2.2.6 Matrices

Om met vectoren en matrices te kunnen rekenen, moet eerst de bibliotheek `linalg` geladen worden, door middel van `with(linalg);`. Een vector van lengte n kan dan simpelweg gecreëerd worden met `v := vector(n);` waarna door indexering de vector te vullen is: `v[1] := 3;` enzovoorts. Deze stappen kunnen ook in één gedaan worden: met `vector(3, [1,2,3])` bijvoorbeeld. Bij matrices gaat het net zo, maar dan in twee dimensies, dus bijvoorbeeld voor een 2×2 matrix `matrix(2,2,[[1,2],[3,4]]);`

Op matrices en vectoren kunnen optelling (met `+`, scalaire vermenigvuldiging (met `*`) en vermenigvuldiging (met `&*` omdat de vermenigvuldiging niet commutatief is) uitgevoerd worden. Omdat het resultaat weer niet automatisch wordt uitgewerkt moet `evalm` gebruikt worden om de resulterende matrix te vinden.

Andere belangrijke operaties bepalen determinant (`det`), rang (`rank`), spoor (`trace`), inverse (`inverse`), en getransponeerde (`transpose`). Ook eigenwaarden en eigenvectoren (`eigenvals` en `eigenvects`) en `charpoly` voor karakteristiek polynoom zijn heel belangrijk.

Eén van de allerbelangrijkste toepassingen van matrices is in het oplossen van stelsels lineaire vergelijkingen. Daarvoor kun je `linsolve` gebruiken; we zagen er al een voorbeeld van in 1.5.

> *Op rijen van een matrix kunnen elementaire operaties worden uitgevoerd met* `addrow`*,* `mulrow` *en* `swaprow`*, en idem voor kolommen.*
> *Naast* `with` *is* `readlib` *te gebruiken voor het inlezen van functies uit de bibliotheken, en met* `share` *heeft men ook toegang tot meegeleverde procedures die door andere gebruikers zijn geleverd.*

Opgave

Controleer dat de oplossing van het stelsel die we vonden in voorbeeld 1.5 voldoet.

`with readlib matrix vector &* det rank transpose inverse eigenvector`

Hoofdstuk 3

Gecompliceerdere bewerkingen

Om ingewikkelder opdrachten in Maple uit te kunnen voeren, zullen we gebruik moeten maken van de programmeerfaciliteiten. We bespreken in dit hoofdstuk kort 3 facetten hiervan.

In de eerste plaats zullen we zien hoe objecten op diverse manieren bij elkaar gegroepeerd kunnen worden. Vervolgens bekijken we een paar mogelijkheden die de taal biedt tot herhaling van opdrachten en tot het maken van selekties. Tenslotte bezien we hoe procedures er uitzien.

Eén van de opvallendste aspecten van Maple met betrekking tot de samengestelde datastructuren waarin we objecten kunnen groeperen is dat dat groeperen kan zonder dat er op het type van de objecten gelet hoeft te worden. Zo kan een verzameling in Maple moeiteloos bestaan uit gehele getallen, polynomen, en verzamelingen van matrices.

3.1 Samengestelde Datastructuren

3.1.1 Expressierij

De *expressierij* (expseq) is het basistype voor samengestelde datatypes; zo'n expressierij ontstaat door Maple expressies (door komma's gescheiden) achter elkaar te zetten. Zo'n opsomming is de eerste manier om een expressierij te maken; een tweede gebruikt `seq` en is vooral handig wanneer je een formule voor de k-de term van een rij hebt: `s := seq( 2^k+1, k=1..5 )`. Ten derde kun je met de functie `op` sommige objecten in Maple in hun *operanden* ontleden en het levert dan een expressierij af. Daarnaast zijn er standaardoperaties in Maple (zoals `solve`) die als resultaat een expressierij (van oplossingen) kunnen geven. De lege expressierij wordt met `null` gemaakt.

Het j-de element van een expressierij `A` wordt verkregen via indicering: `A[`j`]`. Ook kan een deelrij zo verkregen worden, via een *range*: `A[1..4]` bijvoorbeeld voor de eerste 4 elementen.

In `seq(`e`, j=`l`..`r`)` *mag e een willekeurige expressie zijn (die niet altijd volledig geëvalueerd wordt) waar j in voor mag komen; de rij bestaat uit $e(l), \ldots, e(r)$, waar de (gehele) waarden uit de range van l tot $r \geq l$ doorlopen worden. Het is ook mogelijk* `seq(`e`, i=`E`)` *te gebruiken, waar i dan de operanden van de expressie E doorloopt. Dat zijn de elementen van de expressierij die* `op(`E`)` *geeft — is E een verzameling of lijst (zie 3.1.2) dan zijn dat de elementen daarvan. Maar* `op` *geeft bijvoorbeeld van een polynoom de expressierij van de termen ervan. Je kunt* `op` *niet op een expressierij toepassen.*
Met `seq` *en de concatenatie-operator* `.` *kun je bijvoorbeeld een rij variabelennamen maken:* `seq(x.i, i=1..3)` *produceert* `x1, x2, x3`*. Een speciale verkorte vorm hiervan is* `x.(1..3)`.
Tenslotte noemen we nog de mogelijkheid om met `sum` *waarden gegeven door een expressierij te sommeren. Er is een inerte variant* `Sum` *die de som niet evalueert — dat kan vervolgens met* `value` *bepaald worden. Iets soortgelijks bestaat voor integralen (*`int` *en* `Int`*). Als voorbeeld van de som van de eerste 10 priemgetallen:*

```
sum( 'ithprime(i)', 'i'=1..10 )
```

Opgaven

Maak de expressierij bestaande uit paren $[i, j]$ met $1 \leq i < j \leq 10$. Verklaar het resultaat van `seq(i^2, i = x+y)`. Probeer de som $\sum_{n=1}^{\infty} 1/n^2$ te bepalen.

`expseq` `seq( )` `op` `..` `.` `sum Sum` `value`

3.1.2 Verzameling en Lijst

Grof gezegd zijn verzamelingen en lijsten expressierijtjes waarom je haakjes zet: voor een verzameling de gebruikelijke accolades { } en voor een lijst de vierkante haken `[ ]`. Het grootste verschil tussen beide is dat in een verzameling (*set*) elementen niet vaker dan één keer voorkomen, en dat de volgorde waarin elementen staan niet vastligt. In een lijst (*list*) zijn de elementen lineair geördend, en op verschillende plaatsen in de lijst kunnen dezelfde elementen staan.

De lege verzameling is { } en de lege lijst `[ ]`. Verder kunnen verzamelingen en lijsten gevuld worden met expressierijen, bijvoorbeeld: `A := [ seq( 2*i, i=-3..3 )]` enzovoorts. Het toepassen van `op` op een lijst of verzameling is als het weghalen van de haken `[ ]` of { }. De functie `nops` is heel nuttig om het *aantal* operanden te tellen, en geeft van een verzameling of lijst dus de cardinaliteit.

Om een element uit een lijst te selecteren, kun je indexeren met `[ ]`, of `op` gebruiken; `L[3]` en `op(3, L)` leveren beide het derde element van de lijst `L` op. Ook kun je weer met een *range* l`..`r een aantal elementen selecteren, maar let op dat het resultaat een expressierij is, niet een deellijst. Hoewel de volgorde van elementen in een verzameling niet a priori vastligt kun je toch dezelfde mechanismen gebruiken om toegang tot elementen van een verzameling te krijgen; oplettendheid is dan geboden.

Het selecteren van een deelverzameling of deellijst die aan een criterium voldoet dat zich door een Maple-expressie laat uitdrukken kan gebeuren met `select(` C`,` V `)`; dit geeft een verzameling van $v \in V$ waarvoor $C(v)$ geldt.

Om te checken of een element x in een lijst of verzameling S zit, det men `member(` x`,` L `)`, hetgeen een Boolese waarde oplevert.

Het *concateneren* (achter elkaar plakken) van lijsten gaat het makkelijkst met `op`, door `[ op(`L`), op(`M`) ]` om L en M te concateneren.

Voor verzamelingen heb je de operaties `union`, `intersect`, en `minus` ter beschikking om vereniging, doorsnede en verschil (wel in de eerste, niet in de tweede) van 2 te nemen.

De procedure `map` maakt het mogelijk om in één klap een functie f op een hele lijst (of verzameling) van argumenten L los te laten: `map(`f`,` L`)`.

Het criterium waaraan een deellijst of -verzameling moet voldoen kan met behulp van een `->` *procedure ter plekke gemaakt worden. Bijvoorbeeld om uit de eerste tien kwadraten degene te selecteren die tussen 20 en 75 liggen kun je*

```
select( x-> x>20 and x<75, [ seq(i^2, i=1..10) ])
```

doen.
Met `convert(`L`, set)` *kun je van een lijst een verzameling maken (of omgekeerd), maar je kunt ook weer* `op` *gebruiken: als L een lijst is doe je* { `op(`L`)` } *om er een lijst van te maken.*
Je kunt indexering met `[i]` *ook gebruiken om het i-de element van een lijst een andere waarde te geven:* `L[3] := 7;` *bijvoorbeeld.*

Opgaven

Probeer te raden wat `[ seq( j^2, j = -4..4 ) ][6]` en `{ seq( j^2, j = -4..4 ) }[6]` is.

Maak de rij van priemgetallen tussen 100 en 200 die 7 modulo 8 zijn.

`{ }` `[ ]` `nops` `select` `union` `intersect` `minus` `member` `map` `convert`

3.1.3 Arrays, Tabellen

Als laatste van de samengestelde datastructuren noemen we *arrays* en *tabellen*. Arrays zijn we al tegengekomen in de speciale vorm van matrices: dat zijn twee-dimensionale arrays. Een array kan willekeurige dimensie hebben, en de lengte in elke dimensie wordt vooraf gespecificeerd waarna met lijsten van lijsten de array gevuld kan worden. In drie dimensies bijvoorbeeld, een $2 \times 2 \times 3$ array:

```
A := array(1..2,1..2,1..3,[[[1,0,0],[-1,1,2]],[[0,1,0],[7,8,9]]]);
```

De elementen van het array zijn toegankelijk met `A[1,2,3]`, en daaraan kan ook een nieuwe waarde worden toegekend.

Een tabel is een heel algemeen datatype waar de lijst en array speciale vormen van zijn. Het is bijvoorbeeld mogelijk om een tabel te maken waarvan de inhoud niet door gehele getallen wordt geïndiceerd, maar willekeurige andere 'namen': `T := table([a=11,u=x,c=1.3]);`

Een belangrijk element van de manier waarop Maple objecten evalueert heeft betrekking op objecten als lijsten, arrays, tabellen en ook procedures, en wel het principe van de last name evaluation. Dit houdt ruwweg in dat Maple de namen die je aan zulke objecten geeft niet verder evalueert dan de laatste stap die verwijst naar zo'n object zelf. De array A zoals boven gedefinieerd heeft type 'string' en het commando `A;` *laat niet de array maar slechts de naam 'A' zien (met* `eval(A);` *rijg je pas de inhoud te zien). Het belangrijkste gevolg hiervan is, dat een toewijzing als* `B := A;` *aan B slechts de naam A toewijst. Beide namen verwijzen nu naar dezelfde array, en elke verandering in B verandert ook A en omgekeerd: er is maar 1 array! Wil je een hele kopie maken, dan moet je* `C := copy(A);` *gebruiken, waarna er 2 arrays zijn ontstaan, die aanvankelijk dezelfde inhoud hebben.*
Tabellen spelen ook een belangrijke rol omdat Maple ze gebruikt als `remember tables` *bij het opslaan van waarden van procedures die al berekend zijn onder de* `options remember`*. Die maakt het mogelijk dat waarden in een gegeven argument niet telkens opnieuw uitgerekend hoeven te worden.*

Opgave

Probeer de elementen uit de tabel T tevoorschijn te halen en te veranderen.

`array` `table` `remember`

3.2 Switchen, Iteratie, Procedures

3.2.1 Switchen

De standaardmethode om een commando te kiezen uit verschillende mogelijkheden afhankelijk van het al dan niet vervuld zijn van bepaalde voorwaarden, is door middel van de `if - then - else - fi` constructie. Na de *if* volgt een expressie die moet evalueren tot een Boolese (true/false) waarde, op grond waarvan de commando's na `then` worden uitgevoerd (true) of die na `else` (false). Deze gevallen mogen genest worden, en als er in een `else` geval niets hoeft te gebeuren mag dat helemaal weggelaten worden. Het volgende voorbeeld geeft aan hoe je het *teken* van een reëel getal kunt bepalen:

```
> if (x>0) then
>    1
> else
>    if (x<0) then
>       -1
>    else
>       0
>    fi
> fi;
```

Om onnodig nesten te voorkomen is ook de afkorting `elif` beschikbaar; je kunt bovenstaande dan zo korter en overzichtelijker schrijven:

```
> if (x>0) then
>    1
> elif (x<0) then
>    -1
> else
>    0
> fi;
```

Een dergelijke *switch* kan ook gebruikt worden in de definitie van een functie met behulp van `->`, bijvoorbeeld om het teken als functie in te voeren:

```
> teken := x -> if (x>0) then 1 elif (x<0) then -1 else 0 fi;
```

Opgaven

Onderzoek of de hier gedefinieerde functie `teken` precies hetzelfde doet als de ingebouwde `sign` functie.

Schrijf een `->` functie die aan elk paar `[x, y]` het kwadrant toevoegt waarin dat paar ligt.

`if then else fi` `elif`

3.2.2 Iteratie

Het is mogelijk om een aantal malen dezelfde opdrachten te herhalen; daartoe moeten deze dan tussen `do` en `od` ingesloten worden, en het geheel moet vooraf gegaan worden door een uitdrukking die vertelt hoe vaak dat herhalen dient te gebeuren. Dat kan op een aantal manieren.

De eerste manier is door te *itereren* over een lijst of verzameling door middel van `for` x `in` L `do - od;`

De tweede manier is een speciaal geval van itereren waarbij via `for` x `from` b `to` e `do - od;` alleen de (gehele) beginwaarde b en eindwaarde e voor de variabele x worden aangegeven en x achtereenvolgens de waarden $b, b+1, b+2, \ldots, e$ aanneemt. Zo is het resultaat van `for x from -3 to 4 do - od;` hetzelfde als `for x in [-3..4] do - od;`

Het is mogelijk de stapgrootte nog aan te passen in de `from - to -` constructie door toevoeging van `by` s waar s een niet-nul geheel getal is. Zo kan men van 4 naar -3 aftellen met `for x from 4 to -3 by -1`.

Tenslotte kan men een opdracht herhalen zolang aan een bepaalde conditie voldaan is door `while - do - od` . Na `while` moet dan een uitdrukking volgen die tot true/false evalueert, en zolang aan de voorwaarde is voldaan wordt het `do - od` gedeelte herhaald.

Merk op dat in de `while` conditie in het algemeen een variabele zal voorkomen (waaraan aanvankelijk al een waarde toegekend moet zijn) die in het `do - od` gedeelte zal wijzigen (anders termineert het programma nooit). Bijvoorbeeld zoiets:

```
> x := 0;
> while (x<10^6) do
>    x := (x+1)^2;
> od;
```

Maar de variabele waarover geïtereerd wordt in `for x in L` mag juist niet veranderen in het `do - od` gedeelte; hij mag wel gebruikt worden:

```
> t := 0;
> for x in [1..10] do
>    t := (t-x)^2;
> od;
```

Opgave

Maak een $4 \times 3 \times 2$ array met op plaats (i, j, k) de waarde $(-i) + (-j)^2 + (-k)^3$.

`for in` `in` `for from to` `by` `while` `do od`

3.2.3 Procedures

Voor het schrijven van serieuze programma's zal het snel nodig zijn je eigen Maple functies te kunnen definieren. Dat gaat met `f := proc - end;` waar in de plaats van `-` een klein of heel ingewikkeld complex van commando's gespecificeerd kan worden.

De te specificeren functie f heeft (in het algemeen) 1 of meer *argumenten* waarop we de functie willen toepassen, zeg de twee argumenten x, y. Zodra we de functie willen uitrekenen voor bepaalde waarden van x en y doen we dat door $f(7, 3)$ te typen, bijvoorbeeld; op dat moment geven we de waarden voor x, y door aan de variabelen in de gedefinieerde functie. We kunnen dus in de definitie die waarden dan wel gebruiken, maar de x en y zijn globale variabelen (gedefinieerd buiten f) waarvan de waarde in de definitie van f niet mogen wijzigen. Hebben we in de omschrijving van f extra variabelen nodig, *lokale variabelen* die alleen binnen de functie zichtbaar zijn, dan moeten we die *declareren*.

```
> posdif := proc(x, y)
> local z;
> if (x >= y) then  z := x-y;
> else z := y-x; fi;
> RETURN(z);
> end;
```

Ook goed is:

```
> posdif := proc(x, y) if (x >= y) then RETURN(x-y) else RETURN(y-x) fi end;
```

Tenslotte noemen we nog de mogelijkheid om waarden toe te kennen aan extra variabelen die we meegeven als argument aan de functie, bijvoorbeeld om in het bovenstaande aan te kunnen geven welke van de twee gevallen optrad. Dan moet die variabele niet geëvalueerd worden bij toepassing van de functie en daarom moeten er quotes om:

```
> posdif := proc(x, y, 'w')
> if (x >= y) then w := 1; RETURN(x-y);
> else w := 2; RETURN(y-x);
> fi end;
```

Het is mogelijk in de `proc()` *direct type-checking op te nemen om ervoor te zorgen dat de functie niet per ongeluk met foutieve argumenten wordt aangeroepen; dat kan door de argumenten te laten volgen door een type:* `proc( x :: numeric, y :: numeric, w :: name)` *bijvoorbeeld.*
Natuurlijk mag in functiedefinities naar eerder gedefinieerde functies verwezen worden; er mag zelfs naar de functie die op dat moment gedefinieerd wordt verwezen: dat is recursie. Hier is een voorbeeld, waarin bovendien de eerder genoemde remember table van Maple gebruikt wordt:

```
> fibo := proc( n :: integer ) options remember;
>    if ((n=0) or (n=1)) then RETURN(1)
>    else RETURN( fibo(n-1)+fibo(n-2)
>    fi end;
```

Het is soms belangrijk om te weten dat lokale variabelen maar 1 nivo geëvalueerd worden.

Opgave

Schrijf een functie die voor twee positieve gehele getallen x, y de grootste gemene deler d bepaalt. Maak ook een versie die tevens u, v vindt zodat $d = ux + vy$.

`proc() end`   `RETURN`   `option remember`

Hoofdstuk 4

Plaatjes

In dit hele korte hoofdstukje noemen we alleen heel kort de voornaamste twee functies voor het maken van grafieken van functies. Inmiddels zal wel volstaan kunnen worden met een verwijzing naar de on-line help voor de vele en uitgebreide faciliteiten voor het definiëren en manipuleren van de grafische objecten.

4.1 Plots

Het maken van een plaatje van de grafiek van een reële functie gaat eenvoudig met `plot(`f`(x), x=`a`..`b`)`. Hier bepaalt de range a`..`b het interval waarop f wordt afgebeeld. Ook de verticale range kan worden gespecificeerd: `plot(`f`(x), x=`a`..`b`, y=`u`..`v`)`. Let ook op de diverse `options` voor `style`, `axes`, `projection` enzovoorts. Het zal nu duidelijk zijn hoe een plaatje voor de grafiek van een reële functie in 2 variabelen te specificeren met `plot3d`.

Deze functies creëren Maple objecten; die kunnen op de gebruikelijke wijze aan variabelen worden toegekend met `:=` . Onder `plottools` zijn allerhande hulpmiddelen te vinden voor het manipuleren van plotjes: roteren, spiegelen, schalen, etc. Er zijn ook mogelijkheden op het scherm om met menu-opties het plaatje te wijzigen.

Voor sommige functies moet men uitkijken vanwege asymptoten en eventueel gebrek aan oplossend vermogen: bekijk bijvoorbeeld het plaatje van $1/\sin(x^2)$ eens (voor $-5 < x < 5$ en $-10 < y < 10$) gewoon en met de opties `style = point, numpoints=1000`.

Het is mogelijk meerdere grafieken in één plaatje te plotten, via bijvoorbeeld `plot[display]( { een, twee, drie } )`, wanneer aan die variabelen grafieken zijn toegewezen.

Opgave

Maak een lijst van 10 paren $[x, y]$, bijvoorbeeld met $x = 1, 2, \ldots 10$ en plot deze waarden. Experimenteer met de diversie opties. Speel ook met plaatjes in 3 dimensies via `plot3d` voor een paar funkties. Probeer een postscript versie van een aardig plaatje te maken en kijk hoe dat er uitziet met `gv`.

`plot` `plot3d` `plottools` `style numpoints display`