|  |  |
| --- | --- |
| Biostatica  Door Alistair Vardy en Mark Schrauwen | Matlab Wk1.2 |

Inhoudsopgave

[Versiebeheer 2](#_Toc492989501)

[1 Inleiding 3](#_Toc492989502)

[2 Netjes programmeren 4](#_Toc492989503)

[2.1 Opdrachten en vragen 7](#_Toc492989504)

[3 Debuggen en gebruik maken van breakpoints 8](#_Toc492989505)

[3.1 Bugs 8](#_Toc492989506)

[3.2 Syntax fouten 8](#_Toc492989507)

[3.3 Programmeerfouten 8](#_Toc492989508)

[3.3.1 Voorbeeld van een programmeerfout 9](#_Toc492989509)

[3.4 Debuggen 9](#_Toc492989510)

[3.4.1 Breakpoints 9](#_Toc492989511)

[3.4.2 Voorbeeld 9](#_Toc492989512)

[3.5 Opdrachten en vragen 13](#_Toc492989513)

[3.6 Antwoorden 14](#_Toc492989514)

[4 Standaard Matlab functies 15](#_Toc492989515)

[4.1 Wat is een functie? 15](#_Toc492989516)

[4.2 Het aanroepen van een functie 16](#_Toc492989517)

[4.2.1 Functies zonder input 16](#_Toc492989518)

[4.2.2 Andere voorbeelden 16](#_Toc492989519)

[4.3 Het vinden van standaardfuncties 18](#_Toc492989520)

[4.4 Vragen en opdrachten 19](#_Toc492989521)

[4.5 Antwoorden 20](#_Toc492989522)

[5 Meer over functies en zelf een functie schrijven 21](#_Toc492989523)

[5.1 Inputs en outputs 21](#_Toc492989524)

[5.2 Hoe zoek je hulp bij het gebruik van functies? 21](#_Toc492989525)

[5.2.1 Matlab documentatie i.p.v. help 22](#_Toc492989526)

[5.3 Zelf een functie schrijven 22](#_Toc492989527)

[5.4 De opbouw van een zelfgeschreven functie 23](#_Toc492989528)

[5.5 Nog een voorbeeld: de oppervlakte van een cirkel 31](#_Toc492989529)

[5.6 Vragen en opdrachten 34](#_Toc492989530)

[5.7 Antwoorden 35](#_Toc492989531)

# Versiebeheer

|  |  |  |  |
| --- | --- | --- | --- |
| Versie | Datum | Beschrijving | Door |
| 0.0 | 11-07-2017 | Eerste versie | Alistair Vardy |
| 0.1 | 28-08-2017 | Toevoegen subscripts en kleine verbeteringen. | Mark Schrauwen |
| 0.2 | 29-08-2017 | Opmerkingen van Denice Vis en Timothy Roos verwerkt. | Mark Schrauwen |
| 0.3 | 11-09-2017 | Aanvullingen: Inleiding. Zoeken van functies. Voorbeelden. Meer uniformiteit met andere readers. | Mark Schrauwen |
| 0.4 | 12-09-2017 | Aanvullingen: oefeningen. Uitleg opbouw van een functie en zelf een functie schrijven. | Mark Schrauwen |
| 0.5 | 15-09-2017 | Afronden maken van zelfgeschreven functie. Aanvulling opgaves. | Mark Schrauwen |

# Inleiding

Begin deze week heb je voor het eerste kennisgemaakt met Matlab. Je hebt geleerd wat variabelen zijn en je hebt geleerd wat operatoren zijn. Zoals je hebt gemerkt bevat deze cursus van maar 5 lesweken heel veel informatie. Je zult in deze beperkte tijd dus veel kennis moeten eigen maken. Zorg dat je niet gaat achterlopen en dat je begrijpt wat er in deze readers is uitgelegd. Het is onvermijdelijk dat je de readers gaat lezen.

In deze reader gaan we in op de volgende vragen:

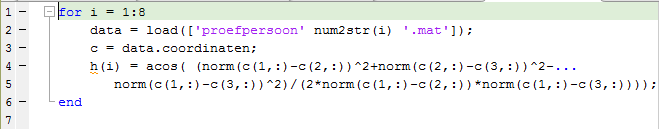
1. Hoe programmeer je netjes?
2. Wat zijn functies?
3. Waarom zijn functies handig?
4. Hoe gebruik je functies?
5. Hoe maak je functies aan?

Aan het einde van deze week worden er weekeindopdrachten uitgedeeld. Deze weekopdrachten staan op blackboard of krijg je via e-mail. Via blackboard moet je de gemaakte opdrachten weer opsturen. Dat kan tot en met de woensdag daarop tot 23:59u. Daarna is inleveren niet meer mogelijk en krijg je een onvoldoende voor de weekopdracht.

**Zie je een fout? Of heb je een aanbeveling dan horen we dat graag! Stuur dan een e-mail naar** [**mjschrau@hhs.nl**](mailto:mjschrau@hhs.nl) **en wij passen het dan z.s.m. aan.**

# Netjes programmeren

We bespreken twee programma’s. Ze doen hetzelfde; het berekenen van de kniehoek voor een gegeven boven- en onderbeenlengte voor een aantal verschillende proefpersonen, en er zijn duidelijke verschillen tussen deze lengtes (zie Figuur 1). Laat je niet afleiden door de complexiteit van de code, dit zal gaandeweg steeds duidelijker worden.



Figuur 1: een for-loop, maak je niet druk over de complexiteit van de code. Daar gaat dit voorbeeld niet over.

We kunnen aan deze code nauwelijks zien wat het programma doet. Stel je voor dat hier ergens een fout in zou zitten. Het is dan buitengewoon lastig om te achterhalen wat er mis is gegaan. Als je deze code met ander persoon deelt dan is de kans groot dat hij/zij er niets van begrijpt.

Er zijn twee manieren van programmeren:

1. netjes
2. ‘quick and dirty’[[1]](#footnote-1).

Dat laatste is niet aan te raden.

Waarom niet?

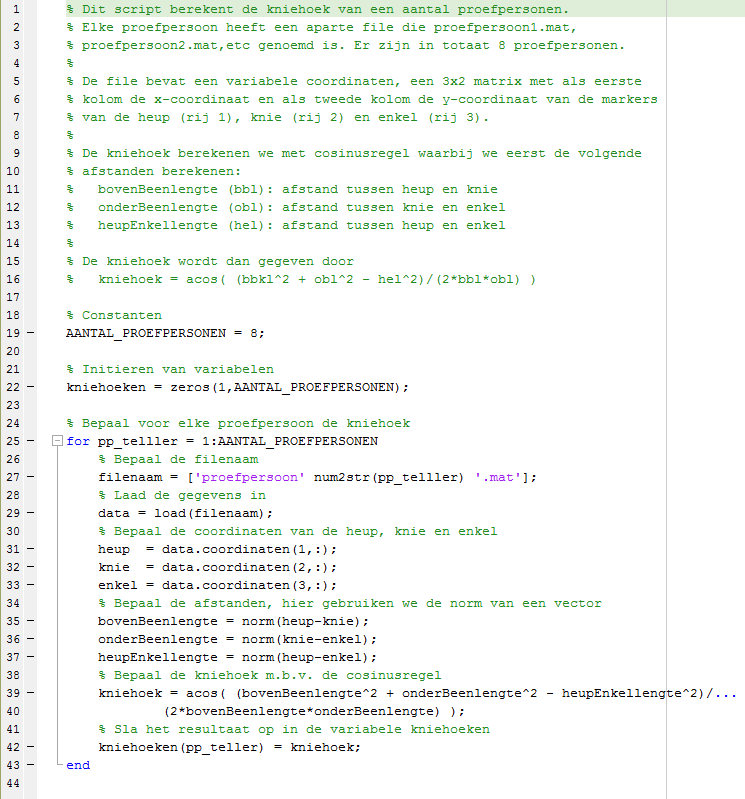
Quick and dirty programmeren is soms onvermijdelijk. Deadlines moeten worden gehaald, of je hebt slechts iets gepland. Maar quick and dirty is veel te vaak de modus operandi. Het nadeel van deze manier van programmeren is dat je niet rekening houdt met jezelf en met een ander. Stel je hebt programma in Figuur 1 geschreven. Op het moment van schrijven begrijp je het helemaal.

Bedenk je nu eens dat je twee weken later naar dat programma gaat kijken. Je hebt er twee weken niet aan gedacht. Denk je dat je het dan nog steeds scherp hebt?

Wat heb jij twee weken geleden op dit tijdstip gedaan?

Wat was je aan het doen?

Netjes programmeren houdt in dat je rekening houdt met de toekomstige lezer van de code. Dat kan jij zelf zijn of een ander persoon. **Bekijk nu het volgende programma:**



Figuur 2: een nette versie van hetzelfde programma

Dit programma is goed leesbaar en het is duidelijk wat de bedoeling is. Wat maakt dit programma nou zo veel beter? Hieronder een lijst met goede programmeereigenschappen:

1. **Maak gebruik van commentaar**

Hiermee geef je de lezer informatie over het programma en hoe verschillende tussenstappen uitgevoerd worden. Dit is essentieel als je met meerdere mensen samenwerkt of als jouw code hergebruikt zal worden. Bedenk ook dat je soms na een aantal weken je eigen code moet lezen en dan is het handig als je het nog snapt op basis van jouw eerder geschreven commentaar.

1. **Geef variabelen intuïtieve namen**

Vooral als je programma groter wordt, is het netjes benoemen van variabelen belangrijk. Hoewel het lijkt alsof het programmeren hiermee langer duurt, zal je meer tijd kwijt zijn met het oplossen van problemen (debuggen) als je hierop beknibbelt. Daarnaast wordt het programma ook veel leesbaarder, iets dat in punt 1 ook is aangestipt. Er zijn meerdere systemen voor naamgeving mogelijk. Welke je gebruikt maakt niet uit maar wees consequent. Hieronder volgt een voorbeeldsysteem waarbij in commentaar de eenheden wordt vermeld.

* Constanten (zoals de zwaartekrachtversnelling) worden vaak met hoofdletters aangegeven:
  + ZWAARTEKRACHTVERSNELLING = 9.81; % m/s^2
* Variabelen. Begin met een kleine letter en geef de eerste letter van een eventueel volgend woord een hoofdletter (dit wordt CamelCasing genoemd):
  + onderBeenlengte = 0.6; % m
  + bovenBeenlengte = 0.5; % m
  + kniehoekSnelheid = 0.1; % rad/sec

1. **Maak gebruik van lege regels**

Dit geeft meer overzicht, niets meer, niets minder

Geen van de bovenstaande programmeereigenschappen maakt het programma trager of slechter. De voordelen zijn dat het programma beter leesbaar wordt en er minder kans op programmeerfouten ontstaan. Bovendien kun je het programma na twee weken/maanden veel sneller begrijpen omdat je het sneller kunt lezen. Ook anderen zullen er blij mee zijn. Het is dus winst voor de programmeur en de collega’s van die programmeur.

## Opdrachten en vragen

Voor dit onderdeel zijn er maar twee opdrachten:

1. Zorg dat je bovenstaand hoofdstuk heb gelezen en begrepen.
2. Pas het netjes programmeren altijd toe!

# Debuggen en gebruik maken van breakpoints

## Bugs

Als we programmeren maken we altijd fouten. Dit is helaas onvermijdelijk. Om fouten op te sporen heeft Matlab een aantal hulpmiddelen. Voordat we beginnen is het belangrijk je te realiseren dat er twee soorten fouten zijn:

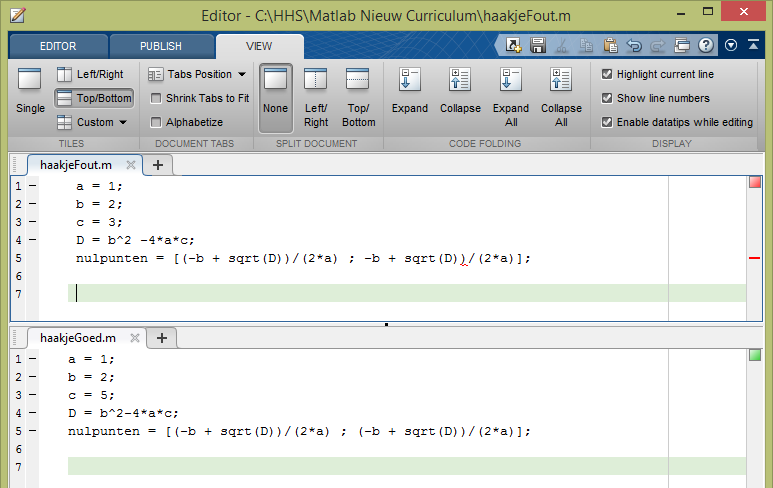
1. Syntax fouten
2. Programmeerfouten

## Syntax fouten

Syntax fouten zijn fouten in de manier waarop je een commando invoert. Matlab zal je vertellen dat er iets mis is. Bijvoorbeeld:

* Het vergeten van een haakje
* De naam van een variabelen verkeerd spellen
* Een niet bestaande functie aanroepen

Als je het programma probeert te runnen, zal Matlab ook een foutmelding geven. De Matlab editor helpt je hier ook bij (zie Figuur 3). Als je ergens een haakje bent vergeten of een andere fout hebt gemaakt geeft Matlab dit aan met rode en oranje kleurtjes aan de rechterkant van de code:

**

Figuur 3: een voorbeeld met haakjes. Aan de rechterkant van het bovenste script zie je rode blokjes. Matlab geeft hier aan dat er iets fout gaat. Dit helpt de programmeur snel fouten te herkennen.

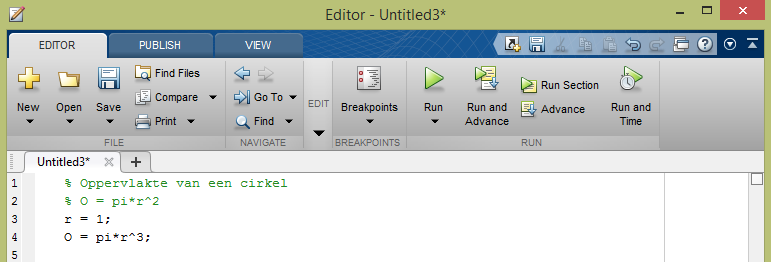
## Programmeerfouten

Programmeerfouten zorgen dat het programma iets anders doet dan je zou willen. Hierbij kan het zelfs gebeuren dat er geen foutmelding komt. Dat betekent dat jij, de programmeur, denkt dat jouw programma werkt, terwijl in werkelijkheid klopt er niets van. Dat is hetzelfde als denken dat je naar huis reist, maar aan het einde van je reis aan de andere kant van het land bent. Je bent dan van A naar B gereisd, maar had bij C moeten uitkomen.

Misschien heb je het nu nog niet door, maar dit type fouten zijn de ergste fouten die je kunt krijgen als programmeur en ze zijn onvermijdelijk! Sterker nog, je zult straks voorbeeldprogramma’s zien waar jij de fouten uit moet gaan halen.

### Voorbeeld van een programmeerfout

Hieronder is een voorbeeld gegeven waarbij de code op zich werkt, maar er is een fout in de gebruikte formule voor de oppervlakte van een cirkel.



Figuur 4: een programmeerfout

Deze laatste situatie is bijzonder vervelend. Gelukkig biedt Matlab een aantal oplossingen om dit soort problemen (en de voorgaande problemen) doeltreffend aan te pakken.

## Debuggen

Het oplossen van fouten wordt ook wel ‘debuggen’ genoemd. De term bug komt oorspronkelijk komt deze term uit de tijd van de enorme kasten van computers die soms kortsluiting veroorzaakte doordat er een insect (bug) geplet tussen de printplaten (rekeneenheden) zat. Deze term wordt nu gebruik voor een foutje in een stuk code.

### Breakpoints

Als we willen weten waar een fout zit, is het handig om het programma even op een bepaalde plek stil te zetten en dan stapje voor stapje verder te gaan. Matlab heeft hiervoor ‘breakpoints’. Hoewel het mogelijk is om vanuit het Command Window line te debuggen, beschrijven we in deze reader alleen het debuggen vanuit de editor. Het plaatsen van een breakpoint kun je in de editor doen door naast het regelnummer te klikken:

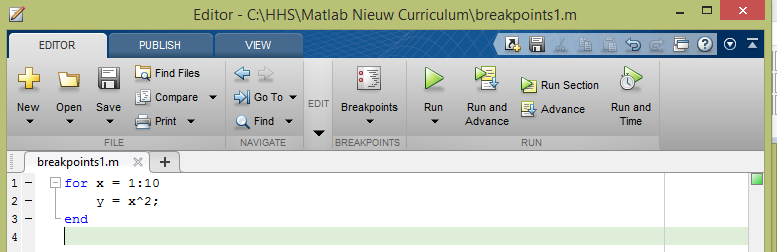
### Voorbeeld

Je hebt de functie . Als je een overzicht wilt hebben van het gedrag van deze functie, kun je bijvoorbeeld x laten lopen van 1 t/m 10 en voor iedere x een y-waarde berekenen. In Matlab programmeer je dat met behulp van een lus (of loop) zo:

for x = 1:10

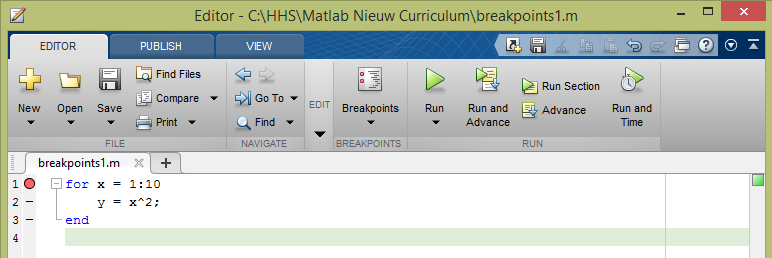
y = x^2;

end

****

Figuur 5: breakpoint voorbeeld, er is nog geen breakpoint geplaatst.

We kunnen een zogenaamd break point zetten waar we willen dat het programma stopt tijdens het uitvoeren. Dit doen we door op het streepje rechts van het regelnummer te klikken. Als het breakpunt wil verwijderen kun je nog een keer klikken:

****

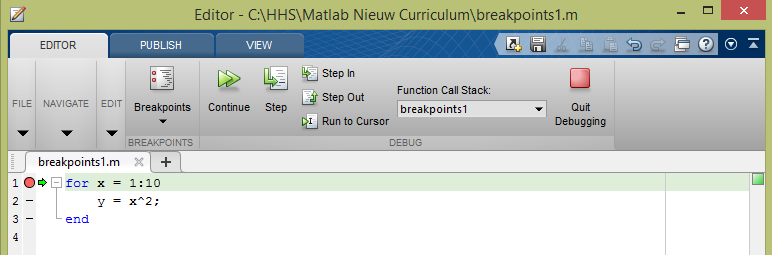
Figuur 6: nu is er een breakpoint geplaatst

Schrijf de code uit Figuur 5 over (het maakt niet uit dat je het nu nog niet begrijpt).

Ze een breakpoint (het zou er nu uit moeten zien als in Figuur 6)

Run het programma

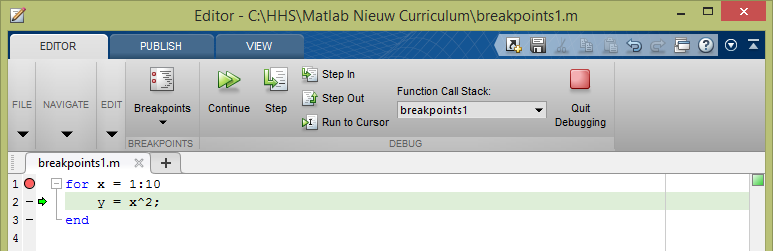
Als we het programma runnen dan zien we een groene pijl die aangeeft waar het programma zich nu bevindt. Het programma is nu gepauzeerd. De editor ziet er nu zo uit:



Figuur 7: runnen van een programma met een breakpoint

De groene pijl geeft aan dat het programma bij regel 1 is gepauzeerd. Rechtsboven zien we ook de rode knop “Quit debugging”[[2]](#footnote-2) waarmee je het debug proces kan beëindigen.

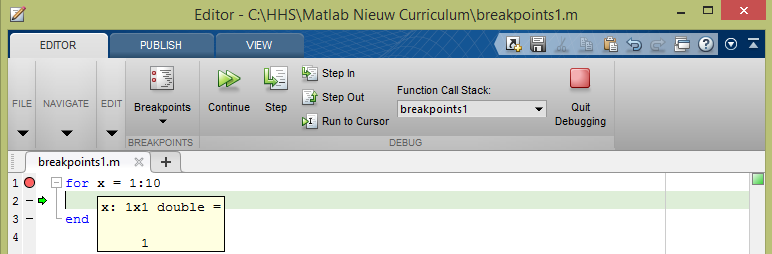
Door op de F10 toets te drukken kunnen we regel voor regel door het programma. Het volgende figuur geeft aan wat er dan gebeurt:



Figuur 8: de breakpoint staat nu op de volgende regel.

We kunnen nu ook informatie krijgen over de inhoud van de variabelen. Dit kan op twee manieren:

1. Als we de muis boven een variabele houden, dan zien we de inhoud daarvan:



Figuur 9: extra informatie als je met de muis op de regel gaat staan.

Let wel, je kunt alleen informatie zien die in de vorige stappen is uitgevoerd. Het programma is gepauzeerd op de regel waar het groene pijltje staat en deze regel is nog niet uitgevoerd.

1. In de Command Window zie je niet de normale Matlab prompt (het >> teken waar je commando’s achter typt) maar een aangepaste prompt: K>>. Hier kun je ook de inhoud van variabelen opvragen.



Figuur 10: de inhoud van een variabele terwijl je aan het debuggen bent.

Dit proces van stap voor stap door het programma lopen, wordt **debuggen** genoemd. Je kunt door het inspecteren van de waarden erachter komen of er programmeerfouten in staan.

## Opdrachten en vragen

1. Zie Tabel 1. Kopieer deze code in Matlab. Op welke regel kun je het beste een breakpoint geplaatst worden als we willen weten wat er in de variabele *vector* staat?
2. Bekijk wat er in de variabele *vector* staat.

|  |
| --- |
| 1. clear all; 2. close all; 3. clc; 4. vector = randn(1,11); 5. vecKwad = vector .^ vector; |

1. We nemen even aan dat je nog niet weet wat de code in regel 1 van Tabel 1 doet. Op welke

regel kun je nu het beste een breakpoint zetten?

Tabel 1: code voorbeeld vraag 1

1. Zie de code in Tabel 2. Het is niet de bedoeling dat je elke regel code gaat begrijpen. Dat is voor nu te ingewikkeld. Wij verwachten dat je in 99% van de gevallen code begrijpt en nu hoeft dat een keer niet. Op welke regel(s) moet je een breakpoint zetten om te zien wat de functie round() doet?
2. Zoek m.b.v. een breakpoint uit wat er in regel 10 in de variabele *trunced* staat.
3. Wat is het verschil tussen de waarde van *output* en *trunced*?
4. Zet een breakpoint op regel 4 en verander de waarde van de variabele naar de waarde 5. Onderzoek door gebruik te maken van breakpoints wat het programma doet.

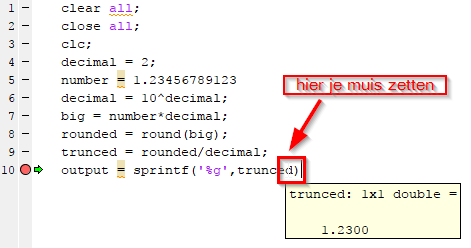
Tabel 2: code die een bewerking op decimalen uitvoert.

|  |
| --- |
| 1. clear all; 2. close all; 3. clc; 4. decimal = 2; 5. number = 1.23456789123 6. decimal = 10^decimal; 7. big = number\*decimal; 8. rounded = round(big); 9. trunced = rounded/decimal; 10. output = sprintf('%g',trunced) |

1. Wat is handig aan het gebruiken van breakpoints?

## Antwoorden

1. Op regel 5. Waarom? Omdat dan de code in regel 4 al is uitgevoerd en het resultaat van de fucntie randn(1,11) in de variabele *vector* is opgeslagen. Je wilde weten wat er in de variabele *vector* is opgeslagen.
2. Dit is een opdracht er is dus geen antwoord.
3. Op regel 1. Let op! Nu is deze regel nog niet uitgevoerd. Dat uitvoeren gebeurd pas als je op F10 hebt gedrukt, dan spring je naar de volgende regel. Tijdens die actie kun je zien wat deze regel exact doet. Als het goed is, is je opgevallen dat alle variabelen in de Workspace worden verwijderd.
4. Regel 8 of 9. Regel 9 is het meest handig als je alleen het resultaat van de functie round() wil bestuderen. Regel 8 is het meest handig als je wil bekijken wat er exact gebeurt voordat je de functie round() wil uitvoeren. In de praktijk zal je allebei de scenario’s even vaak zien voorkomen.
5. Je hebt een breakpoint op regel 10 gezet en je gaat vervolgens met de muis op de variabele *trunced* staan:



Figuur 11: Matlab geeft tijdens het debuggen aan wat de waarde is van een variabele als je met muis op de variabele staat.

1. Het verschil is te zien in de Workspace. Daar staan immers de variabele die je hebt gebruikt tijdens het uitvoeren van het programma. Het verschil is dat: output een string is en trunced een getal. Een ander verschil is dat trunced de waarde 1.2300 bevat en output de waarde ‘1.23’
2. Je hebt gezien dat je met deze code de aantal decimalen van een variabele kunt beperken. In dit geval bevat de variabele *output* de waarde ‘1.23457’ dat zijn 5 decimalen achter de komma.
3. Het handige is dat je krijgt te zien wat er in elke regel exact gebeurd. Als je jezelf afvraagt: ‘wat doet deze code exact’? Dan helpen breakpoints je om dat heel erg gemakkelijk te bepalen. Er bestaat onzes inziens geen krachtiger hulpmiddel om te leren wat een stuk code doet. Vind je het antwoord van deze vraag onvolledig? Of heb je een beter antwoord? Stuur mij een e-mail: [mjschrau@hhs.nl](mailto:mjschrau@hhs.nl). Als je een verbetering heb, zal ik dat antwoord hier plaatsen. Je mag ook andere opmerkingen over de readers/cursus sturen, die zal ik altijd z.s.m. verwerken.

# Standaard Matlab functies

We behandelen in dit hoofdstuk de volgende onderwerpen:

* Wat is een functie?
* Inputs en outputs
* Hoe gebruik je een functie?
* Hoe zoek je hulp bij het gebruik van functies
* Zelf een functie schrijven

## Wat is een functie?

Een functie in Matlab is een programma dat een bepaalde specifieke opdracht uitvoert. Dit kan bijvoorbeeld het berekenen van het gemiddelde van een rij getallen zijn. Zo zijn er een hele hoop (wiskundige) berekeningen die we kunnen doen op basis van **standaardfuncties** in Matlab. Een aantal van deze standaardfuncties staan hieronder beschreven en komen dus standaard met Matlab bijgeleverd. Dit is een greep:

* Goniometrische functies
  + sin, cos, tan – de sinus, cosinus en tangens van een hoek (in radialen)
  + asin, acos, atan – de inverse sinus, cosinus en tangens van een hoek (in radialen)
  + sind, cosd, tand – de sinus, cosinus en tangens van een hoek (in graden)
  + asind, acosd, atand – de sinus, cosinus en tangens van een hoek (in graden)
* De logaritme en exponentiele functie
  + log – de natuurlijke logaritme
  + log2, log10 – de logaritme met grondtal 2 en 10
  + exp – de e-macht
* Veelgebruikte bewerkingen
  + mean – het gemiddelde van een vector of een matrix (rijen of kolommen)
  + std – de standaarddeviatie van een vector of een matrix (rijen of kolommen)
  + sum – de som van een van een vector of een matrix (rijen of kolommen)
  + cumsum – de cumulatieve som (handig voor numeriek integreren)
  + trapz – de trapeziumregel voor integreren
  + cumtrapz – de cumulatieve trapeziumregel voor integreren
  + gradient – de numerieke afgeleide van een rij getallen of een matrix

## Het aanroepen van een functie

De hierboven beschreven standaardfuncties kun je gebruiken door ze aan te roepen. Dat aanroepen van een functie is heel erg gemakkelijk.

Typ in het Command Window: cos(pi)

Het antwoord is natuurlijk -1.

De functie cos() wordt standaard meegeleverd en kun je één argument meegeven. Dat argument is in dit geval ‘pi’[[3]](#footnote-3).

Doorgaans kun je een functie herkennen aan haakjes:

**cos ( )**

Tussen de haakjes geef je iets op aan de functie, dat was in dit geval ‘pi’. Dit noemen we een *argument* of *input* van de functie.

### Functies zonder input

Het is echter niet zo dat elke functie een input nodig heeft. Je hebt ook functies die werken zonder het geven van input. Een voorbeeld van zo’n functie is randn().

Typ in het Command Window: randn()

Herhaal dit een aantal keer

Wat valt op?

Als je de functie randn() een aantal keer uitvoert, zie je telkens een ander getal in het Command Window. Deze functie genereert random getallen voor de gebruiken en vereist dus geen input.

Je ziet dat het aanroepen van een functie heel erg gemakkelijk is, typ de naam van de functie en geef de functie een argument of waarde.

### Andere voorbeelden

Je kunt in Matlab ook de sinus berekenen van een hoek uitgedrukt in graden:

Typ in het Command Window: sind(90)

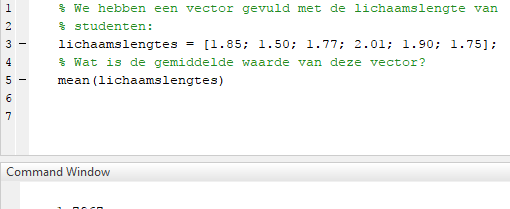
Is het resultaat wat je verwacht?

Sind() is een functie die de sinus van een getal in graden evalueert. De ‘d’ van sind() staat voor: ‘degree’. We weten uit de Wiskunde dat

Typ in het Command Window: sin(pi/2)

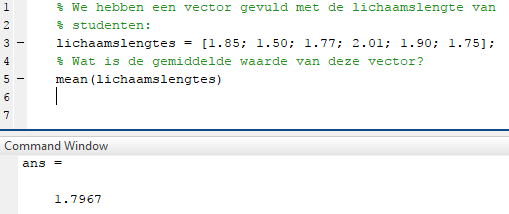
Daar komt gelukkig hetzelfde antwoord uit. In het ene geval geven we de functie het argument in graden mee. In het tweede geval geven we de functie het argument in radialen mee.

Stel, we willen de gemiddelde lichaamslengte bereken van een aantal studenten. We maken dan een vector met lichaamslengtes (zie Figuur 12):



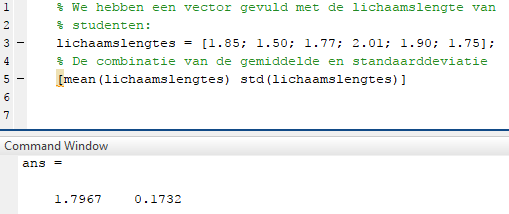
Figuur 12: een voorbeeld van een aantal lengtes in een vector.

Het berekenen van de gemiddelde lengte kan dan eenvoudig met de fucntie mean(). De mean van een vector is hetzelfde als *het gemiddelde* van een vector. In Figuur 14 is de toepassing van deze functie te zien en de uitkomst.



Figuur 13: een voorbeeld van de functie mean().

Dit voorbeeld kan worden uitgebreid met de standaard deviatie. We willen hier ook weten wat de standaarddeviatie is van de lengtes van het groepje studenten:



Figuur 14: het combineren van twee maten (gemiddelde en standaarddeviatie) in één vector.

**Typ de code over en vul de lichaamslengtes aan met jouw lichaamslengte.**

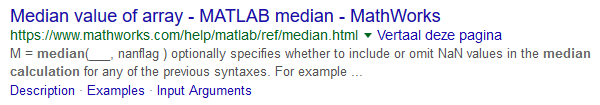
Het aanroepen en gebruiken van functies is, zoals je hebt gezien erg handig en zal ons veel tijd gaan besparen in de toekomst.

## Het vinden van Matlab standaardfuncties

Een veel voorkomende en terechte vraag van studenten is: ‘hoe vindt je een bepaalde Matlab functie’? Het antwoord is vervelend genoeg: door veel te lezen!

|  |
| --- |
| **<Een zoekmachine> is your friend.** |

Stel je wil de *mediaan* berekenen van de lijst van lichaamslengtes. Je kunt in een willekeurige zoekmachine een aantal keywords typen zoals: *Matlab calculate median*. Je krijgt dan dit resultaat:

[](https://www.google.nl/search?client=firefox-b-ab&dcr=0&q=Matlab+calculate+median&oq=Matlab+calculate+median&gs_l=psy-ab.3..0i19k1j0i8i30i19k1l3.13988.13988.0.14213.1.1.0.0.0.0.39.39.1.1.0....0...1.1.64.psy-ab..0.1.38.TNPsdLHDhsk)

Figuur 15: het zoeken van een functie.

Bekijk Figuur 15

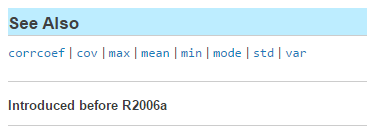
Zie je het voorbeeld van de mediaan functie?

Het voorbeeld laat zien dat de mediaan functie in Matlab: median() heet. Dat is valt te verwachten natuurlijk. Een andere manier om meer functies te vinden is om de Matlab documentatie te lezen. Verreweg de gemakkelijkste manier is om de *See Also* kopjes te lezen.

Typ in het Command Window: doc median

Er wordt nu een venster geopend met Matlab documentatie.

**Klik op *See Also*.** Dit is het resultaat:



Figuur 16: See Also informatie van de functie mean().

Je ziet in Figuur 16 dat Matlab een aantal suggesties doet voor andere functies. In dit geval zie je de mean() functie terugkomen, maar ook een max() functie en een min() functie. Heel handig zijn dit interne links naar andere documentatie pagina’s.

Klik op de link ‘max’

Je merkt dat je nu uitkomt op de max documentatie pagina van Matlab. Als je dit met regelmaat doet, kom je vanzelf bij handige functies uit voor de toekomst. Later in deze cursus gaan we je laten zien hoe je de documentatie het beste kunt benaderen en gebruiken. Je kunt hier natuurlijk nu al mee beginnen.

## Vragen en opdrachten

1. Je ziet aan het begin van de hoofdstuk een aantal standaard Matlab functies. Je moet in deze en de volgende oefeningen gebruik maken van deze functies. Maak een nieuw Matlab script aan en vul het met deze variabele: lichaamslengtes = [1.85; 1.50; 1.77; 2.01; 1.90; 1.75]; Tel m.b.v. een functie alle waardes bij elkaar op.
2. Maak de volgende sinus functie aan: met een variabele frequentie=1. Een tijdvector maak je als volgt aan: t = 0:0.1:10; In dit geval moet de tijd vector van 0 tot en met pi lopen in stapjes van 0.01.
3. Wat is de frequentie van de sinus in de voorgaande afbeelding? Hoe kun je dit zien?
4. Open een nieuw Matlab script en maak daar een vector aan die de getallen van 0 tot en met 10 bevat in stapjes van 0.1. De vector bevat nu waardes als: 0, 0.1, 0.2, 0.3 … het verschil tussen elk stapje is 0.1. Welke van deze functies kun je gebruiken om het verschil tussen elk element te bepalen: cumsum, trapz, cumtrapz, gradient? Test dit!
5. Waarom is Matlab zo handig voor een Bewegingstechnoloog?

## Antwoorden

|  |  |
| --- | --- |
|  | clear all; close all; clc;  lichaamslengtes = [1.85; 1.50; 1.77; 2.01; 1.90; 1.75];  sum(lichaamslengtes) |
|  | clear all; close all; clc;  frequentie = 1;  % Een tijdvector tot en met pi (3,1415 seconde)  t = 0:0.01:pi;  % De sinus functie  functie = sin(2\*pi\*frequentie\*t)  % Dit heb je nog niet geleerd, maar je kunt de waardes in  % de variabele functie ook plotten (lees: weergeven in een  % grafiek):  plot(t,functie) |
|  | Je kunt dit alleen goed zien als je de sinus plot in een afbeelding. Het plotten komt later in deze cursus nog aan bod. De frequentie is volgens bovenstaande plot 1 Hz. Dat is in overeenstemming met wat we hadden verwacht. Merk op dat plot() ook een functie is. |
|  | clear all; clc;  vector = 0:0.1:10;  gradient(vector)  De functie gradient() bepaalt van elk element het verschil met het opvolgende element. |
|  | Dat was één van de eerste vragen gesteld in deze cursus. Een antwoord op deze vraag is: omdat Matlab zoveel standaardfuncties bevat. Deze standaardfuncties besparen programmeurs en bewegingstechnologen veel tijd in het ontwikkelen van hun algoritmes. Dat is dus erg prettig. |

# Meer over functies en zelf een functie schrijven

Een basis vaardigheid van het programmeren in Matlab is het zelf schrijven van functies. Aan het einde van dit blok zal je dat veelvuldig hebben gedaan. Hopelijk leer je in dat proces ook dat je m.b.v. functies veel tijd kunt besparen, vooral in grotere programmeer projecten.

Deze week focussen we ons enkel op het schrijven van eenvoudige functies. Dat wil zeggen functies met één input en één output.

## Inputs en outputs

Een functie heeft een aantal eigenschappen. In plaats van een set commando’s uit te voeren, heeft een functie een aantal *inputs* en *outputs*. We geven hieronder een schematische weergave.



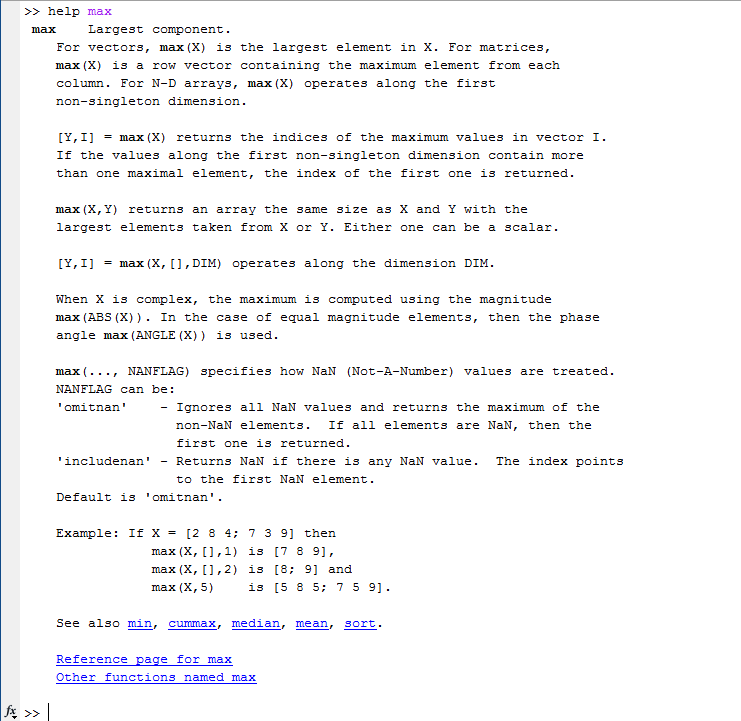
Figuur 17: een schematische weergave van een functie

## Hoe zoek je hulp bij het gebruik van functies?

Als we een onbekende functie willen gebruiken, dan kunnen we de help-tekst daarvan lezen. Stel, we willen meer informatie hebben over de standaard Matlab functie max(). Dan krijgen we deze informatie door in het Command Window het volgende te typen:

Typ in het Command Window: help max

Je krijgt dan onderstaande output in het Command Window te zien:



Figuur 18: de output van de Matlab help

### Matlab documentatie i.p.v. help

In plaats van de help functie van Matlab kun je ook wat uitleg zien in een speciaal documentatie venster van Matlab. In plaats van help mean typ je dan doc mean.

Typ in het Command Window: help mean

Typ nu in het Command Window: doc mean

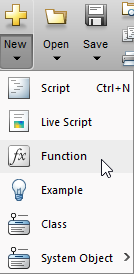
Welke van de twee manieren van informatievoorziening vind je het prettigst?

## Zelf een functie schrijven

We weten nu van alles over functies. Maar het wordt echt de hoogste tijd dat we zelf functies gaan schrijven. We weten dat een functie meerdere inputs kan hebben en een output kan geven. We gaan eerst naar eenvoudige functies kijken die één input heeft en één output geeft.

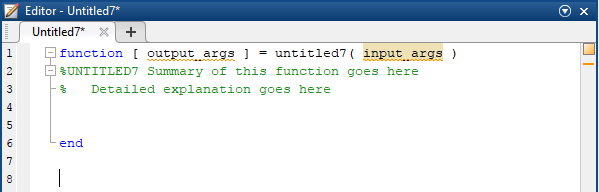
Stel we willen de vierde macht van een getal bereken en daar de waarde 2 bij optellen. We moeten dit in ons hoofdprogramma meer dan 10 x uitvoeren op verschillende plekken in de code. In dat geval is een functie heel erg handig.

We kunnen een functie in Matlab gemakkelijk aanmaken. **Ga naar het tabblad EDITOR en klik op New** (zie Figuur 19).



Figuur 19: het aanmaken van een functie script

Nu wordt het onderstaande script aangemaakt[[4]](#footnote-4):



Figuur 20: een door Matlab aangemaakt script met alle benodigdheden voor het maken van een functie.

|  |
| --- |
| Merk op dat als je een bestand wijzigt en deze wijzigingen zijn nog niet opgeslagen dan staat er een \* boven de bestandsnaam/tab (zie Figuur 20). |

## De opbouw van een zelfgeschreven functie

In Figuur 21 is de opbouw van een functie vergroot weergegeven. In het geel is de naam van een functie aangegeven.

Waarom is de naam van een functie belangrijk?

Eerder hebben we al een aantal Matlab functies gebruikt. Een voorbeeld: cos(), mean(), sind(), etc. Hoe roepen we een functie aan? Door het typen van de ***naam van de functie***. In het geval van Figuur 21 hebben we net een functie gegeneerd die nog geen logische naam heeft. Daarom heet de functie in dit voorbeeld tijdelijk: ‘*untitled7’*.

Zodadelijk als we de functie gaan opslaan, geven we het een logische naam.

|  |
| --- |
| Naam van functie  Figuur 21: de opbouw van de functie, in het geel is de naam van de functie aangegeven. |

In Figuur 22 is de input van een functie gemarkeerd. Hier komt te staan wat we aan de functie hebben meegegeven. Eerder hebben we bijvoorbeeld de sinus berekent van een hoek van 90 graden. Dat deden we met: sind(90). In dat voorbeeld is de waarde 90 de input van de functie.

|  |
| --- |
| Input van functie  Figuur 22: de opbouw van de functie, in het rood is de input van een functie aangegeven. |

Het is heel normaal dat als we iets in een functie stoppen, we er iets uit krijgen[[5]](#footnote-5). Daarom moeten we in deze *ontleding van de opbouw van een functie* ook aangeven hoe de output van een functie er uit ziet (zie Figuur 35).

Helemaal op het einde als de functie klaar is met uitvoeren, wordt de waarde naar de output van een functie geschreven. Zodoende kan een gebruiker de output verder gebruiken. Bijvoorbeeld in het voorbeeld met: sind(90) was de output: 1. Als deze functie niets had teruggegeven dan was het aanroepen er van ook niet nuttig geweest. De reden dat we in dit specifieke geval de waarde ‘1’ terugkrijgen van de functie sind() is omdat aan het einde van het uitvoeren van de functie het resultaat (de waarde 1) naar de output van de functie sind() wordt geschreven.

|  |
| --- |
| Output(s) van de functie  Figuur 23: de opbouw van de functie, in het paars is de output van een functie aangegeven. |

Waar komt nu de code te staan die de functie moet uitvoeren?

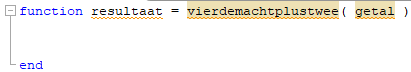
De code die een functie moet uitvoeren staat in de *body* van de functie. In Figuur 24 is dit aangegeven. Hier komt de berekening of het algoritme te staan die de functie moet gaan uitvoeren. Het is niet vreemd dat de body van een functie meer dan 50 regels code heeft.

|  |
| --- |
| De body van een functie  Figuur 24: de opbouw van de functie, in het blauw is de body van een functie aangegeven. |

Tot nu toe zijn de voorgaande voorbeelden slechts bedoeld om de opbouw van een functie aan te duiden en uit te leggen. Nu gaan we zelf een functie bouwen. Het is een eenvoudig voorbeeld. We gaan de voorgaande code ombouwen zodat de functie de volgende acties uitvoert:

Stel we willen de vierde macht van een getal bereken en daar de waarde 2 bij optellen.

Pas de functie aan zodat het overeenstemt met Figuur 25:



Figuur 25: we hebben de aangemaakte functie aangepast.

Merk op dat we de *input*, *naam* van de functie en de *output* van de functie hebben aangepast. De code is nu al een stuk leesbaarder dan voorheen.

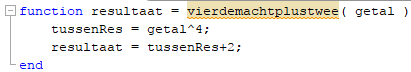
|  |
| --- |
| **We hebben nog een andere belangrijke stap uitgevoerd die niet is te zien. Namelijk we hebben het script opgeslagen onder de exacte naam: vierdemachtplustwee.m** |

Zie je waar de naam van het script mee overeenstemt?

|  |
| --- |
| **Vanaf dit punt noemen we een script met alleen een functie erin een: function file.** |

Het is in Matlab belangrijk dat de function file dezelfde naam heeft als de functie die in dat bestand staat. Dus als je een functie: hottentottententoonstelling hebt moet die functie zijn opgeslagen in een bestand met de naam: hottentottententoonstelling.m

Het is de bedoeling dat bij het uitvoeren van de functie vierdemachtplustwee de input getal tot de vierdemacht wordt berekend en dat er daarna twee bij wordt opgeteld. We vullen dan de functie als volgt aan:

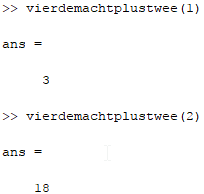


Figuur 26: we hebben de aangemaakte functie verder aangevuld

Je ziet dat we een extra variabele hebben aangemaakt genaamd tussenRes. In die variabele wordt een tussen resultaat opgeslagen. In de tweede regel code tellen we de waarde twee op bij het tussen resultaat.

In de laatste regel slaan we het tussen resultaat plus twee op in de variabele resultaat. Dit is alles. Waarom slaan we het resultaat van de bewerking in de variabele resultaat? De output van de functie is gedefinieerd als resultaat. Zodra we schrijven naar een variabele, binnen de functie, met exact die naam, schrijven we direct naar de output van een functie.

Nu kunnen we onze zelfgeschreven functie testen:



Figuur 27: de output van de functie vierdemachtplustwee.

Op deze manier ontwikkel je eenvoudig zelfgeschreven functie. Even controleren of de output klopt: 2^4 is 16. Dan 16 + 2 is 18. Het ziet er naar uit dat onze functie goed werkt.

## De laatste stap: het begrijpen van het gedrag van een functie

Je hebt gezien hoe een functie is ingedeeld en opgebouwd. Je hebt gezien hoe je een functie moet aanroepen. Je hebt gezien hoe je een functie bouwt.

Maar heb je begrepen wat exact het gedrag is van een functie?

In deze sectie van dit document gaan we daar naar kijken. Om het exacte gedrag te begrijpen, gaan we het meest krachtige hulpmiddel in Matlab beschikbaar bekijken om dit te gaan begrijpen: breakpoints.

Lees **de code in Figuur 28 en typ** het over in een nieuw script. Let op! Dit script moet in dezelfde folder staan als onze zelfgeschreven functie.

Zet een breakpoint zoals is te zien in **Figuur 28.**

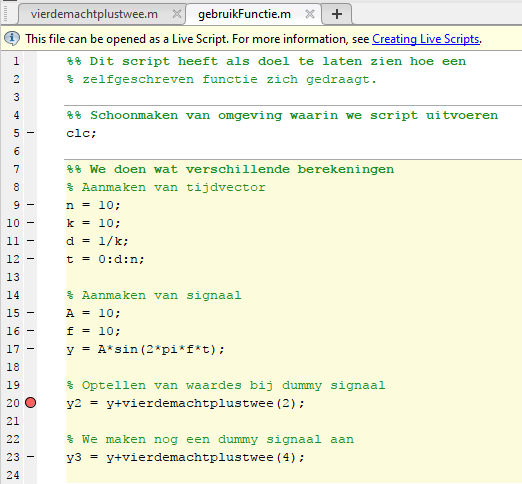
Snap je waarom we op deze regel een breakpoint plaatsen?

We zijn aan het onderzoeken hoe de functie zich gedraagt. Dus we willen dat het script stopt met runnen zodra het op de regel komt waar onze functie wordt aangeroepen.

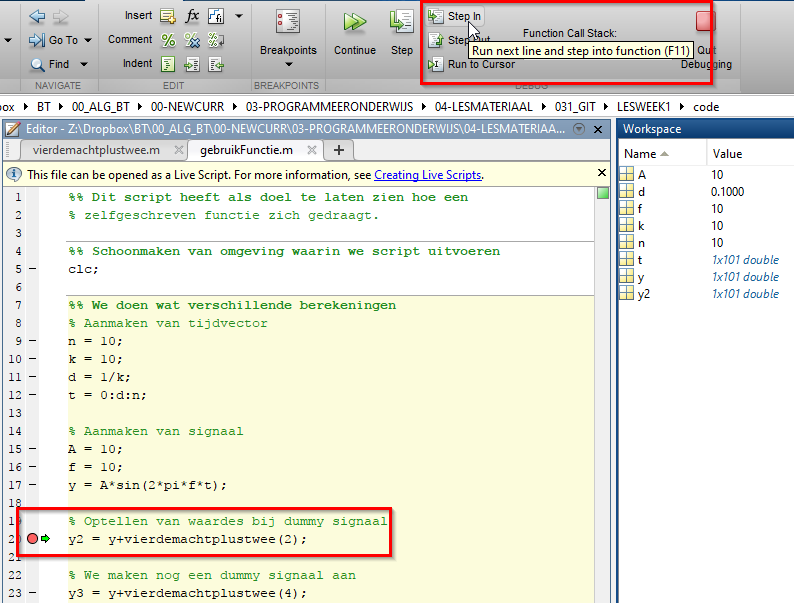
Run nu het script

Je bent nu gestopt op de regel met de breakpoint. Druk nu op F11 om in de functie gaan (Step In, zie Figuur 29).

Matlab brengt je nu naar onze zelfgeschreven functie (zie Figuur 30).



Figuur 28: het gebruiken van onze functie in een dummy omgeving.



Figuur 29: het programma is gestopt op de regel waar onze zelfgeschreven functie wordt aangeroepen. Druk nu op F11 om ' in ' de functie te gaan.



Figuur 30: Matlab heeft ons nu naar de zelfgeschreven functie gebracht. De rode pijl wijst naar de muispointer. Als je tijdens het debuggen op een variabele staat met de muispointer krijg je de inhoud van deze variabele te zien.

We zijn aan het uitzoeken wat het exacte gedrag is van een functie. Je ziet dat bij het aanroepen van een functie we de code *in* de functie gaan uitvoeren.

**Kijk eens wat er in de workspace staat in Figuur 30**

**Vergelijk dit met de Workspace in Figuur 29**

Wat valt op?

Zodra je in een functie springt, d.m.v. debuggen zie je ook welke variabele staan in de Workspace van de functie. Het script in Figuur 29 heeft een Workspace en de functie in Figuur 30 heeft zijn eigen Workspace. Het debuggen heeft ons dat late zien.

Wat is er gebeurt met de Workspace in Figuur 29?

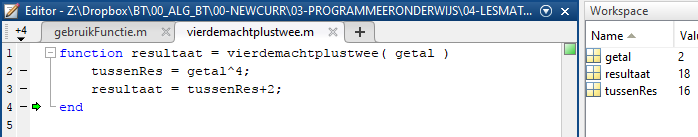
De workspace van het script in **Figuur 29** bestaat nog steeds. Het is op dit moment niet beschikbaar. Dit hebben de makers van Matlab bewust gedaan. Als je in een functie komt, heeft de functie altijd zijn eigen Workspace. Dit is namelijk veel veiliger. Een functie heeft input en output. Als alle operatie van een functie ook de variabelen in de Workspace van het script zouden kunnen aanpassen heeft de input en output van een functie weinig zin meer. Immers, variabelen zouden in dat geval direct vanuit de functie kunnen worden aangepast. Dat levert onveilig gedrag op. Het wordt dan heel lastig om bij te houden welke wijzigingen een functie heeft op de Workspace van het script.

Het is veiliger om de Workspace van een functie gescheiden te houden van het script dat het aanroept. In Matlab heeft elke functie zijn eigen Workspace.

**Druk, om een regel code van de functie uit te voeren op F10**

**Druk, om de tweede en laatste regel van de functie uit te voeren op F10**

**Druk, nogmaals op F10**

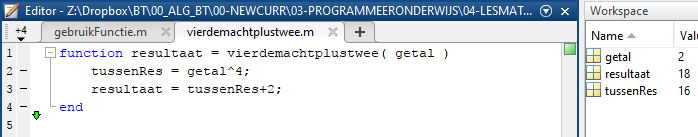


Figuur 31: We zijn tijdens het debuggen in de laatste regel van de functie aangekomen. Merk op dat de output-variabele als zijn eind waarde bevat.

We zijn nu aangekomen bij de laatste regel code van de functie. Ook deze regel wordt uitgevoerd.

**Druk, nogmaals op F10**

Zie Figuur 32. Je ziet dat het debug-pijl omlaag wijst. Dat betekent dat Matlab gaat stoppen met het uitvoeren van de functie. Matlab weet dat de functie en de bijbehorende workspace kan ophouden te bestaan.

****

Figuur 32: het debug pijltje wijst nu naar onderen. Dat betekent dat we uit de functie gaan.

**Druk, nogmaals op F10**

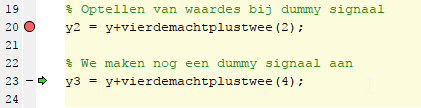
We zijn nu teruggesprongen naar het script dat onze functie heeft aangeroepen (Figuur 33):

****

Figuur 33: We zijn nu teruggesprongen naar het script dat de functie heeft aangeroepen.

Op dit exacte moment bevat de variabele y2 nog niet het resultaat van y+vierdemachtplustwee(2); Pas als we nogmaals op F10 drukken wordt het resultaat van de berekening toegekend aan de variabele y2:

**Druk, nogmaals op F10**

****

Figuur 34: het resultaat in y2 kan nu worden bekeken. Ga met je muis op de variabele y2 staan en bekijk wat er in staat.

## Nog een voorbeeld: de oppervlakte van een cirkel

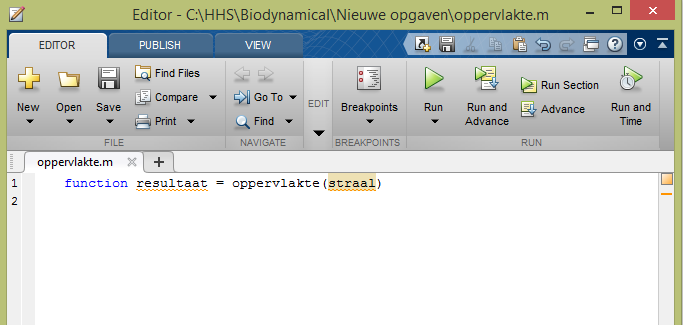
We gaan nu nog een functie maken. De uitleg is iets anders, maar behandelt hetzelfde als hiervoor. De functie heeft als input de straal van een cirkel en als output de oppervlakte van een cirkel met die straal.

Een Matlab functie begint door een nieuwe file te openen en te beginnen met een regel die het volgende aangeeft:

1. Dat het hier een Matlab functie en geen script betreft
2. Wat de inputs zijn
3. Wat de outputs zijn

Voor onze oppervlakte berekening beginnen we met

function resultaat = oppervlakte(straal)



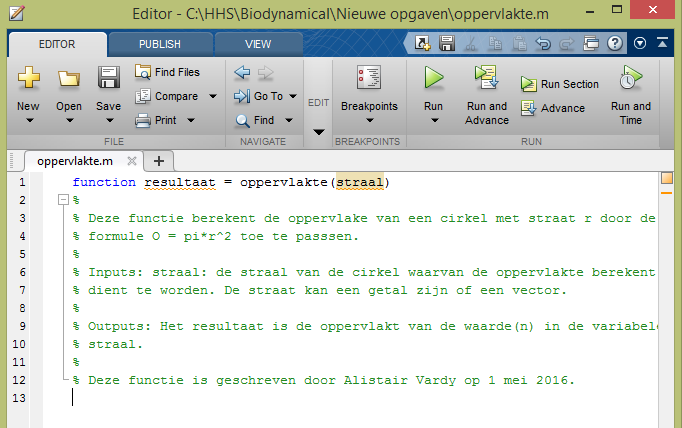
Figuur 35: een nieuwe functie voor het bereken van de oppervlakte van een cirkel.

We slaan vervolgens de file op. Matlab zal een suggestie doen, namelijk *oppervlakte.m.* Het is belangrijk dat de functienaam en de naam van de file hetzelfde zijn.

We kunnen nu regels code toevoegen om de bewerking compleet te maken. Echter, voordat we dat doen, is het altijd raadzaam op extra informatie toe te voegen.

Als je bijvoorbeeld in het Command Window “help mean” intypt, dan krijg je informatie te zien over hoe je de functie moet gebruiken. Voeg altijd deze hulp tekst toe zodat anderen eenvoudig kunnen zien hoe ze de functie moeten gebruiken. Ook voor jezelf is het goed omdat het opschrijven meteen een kader biedt voor de code die je later gaat toevoegen.

Hulp-tekst kun je als commentaar meteen achter de eerste regel toevoegen. Commentaar is tekst dat voorafgegaan wordt door een %-teken.



Figuur 36: verder aangevuld.

Voeg nu de overige code toe

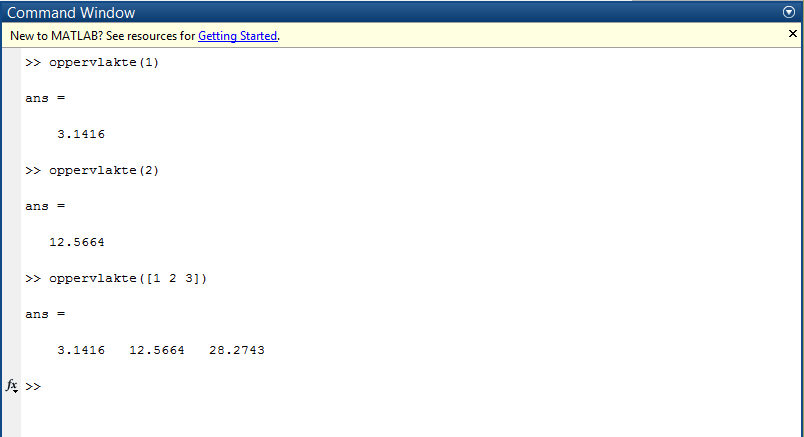
Test de functie in Matlab door de volgende commando’s uit te voeren:

oppervlakte(1);

oppervlakte(10);

oppervlakte([1 10 100]);

Hieronder volgen een aantal andere voorbeelden. Komen jouw antwoorden overeen?



Figuur 37: de output van de zelfgeschreven functie oppervlakte() in het Command Window.

Check de help tekst door help oppervlakte in te toetsen.

Je kunt, zoals in het derde voorbeeld hierboven, ook een vector van stralen invoeren. Dit werkt echter alleen als de elementen bij het kwadrateren puntsgewijs vermenigvuldigd worden. Is dit bij jouw functie het geval?

## Vragen en opdrachten

1. Hoe kun je in Matlab opzoeken hoe je de functie **round** moet gebruiken?
2. Hoe kan je ervoor zorgen dat een vermenigvuldiging of machtsverheffing tussen twee vectoren of matrices puntsgewijs wordt uitgevoerd?
3. Als je een functie maakt met de volgende regel

function resultaat = test1(a)

Hoe kun je dan het beste de filenaam noemen als je deze functie op wil slaan?

1. Hoe bereken je het gemiddelde van de vector v = [1 2 3 4 5];?
2. Bekijk de hulp tekst van de functie round en geef zonder het uit te voeren aan wat het resultaat is van het commando round(3.14159,3);
3. Welke functie wordt gebruikt voor het berekenen van de standaarddeviatie?

## Antwoorden

1. M.b.v. het commando help round
2. M.b.v. een “.” ; bijvoorbeeld straal.^2
3. test1.m
4. M.b.v. het commando mean(v)
5. 3.1220
6. Std()

1. Quick and dirty is uiteindelijk slow and cumbersome [↑](#footnote-ref-1)
2. Je kunt ook op shift+F5 drukken. [↑](#footnote-ref-2)
3. De variabele ‘pi’ is een standaard Matlab variabele. [↑](#footnote-ref-3)
4. Let op! Het nummer, in dit geval 7 kan anders zijn. [↑](#footnote-ref-4)
5. Het is echter ook heel normaal er functies zijn die geen output teruggeven. Een voorbeeld is hier van is een functie waar je nog nader kennis mee gaat maken: plot(). [↑](#footnote-ref-5)