Monty Python en het Holy Grondwater-model

Mark Bakker

Inleiding

De meeste computerprogramma's voor grondwatermodellering zijn geschreven in FORTRAN, C of Pascal. Menig geohydroloog heeft daar ook wel eens zelf in zitten programmeren en maar weinigen zullen daar plezier aan hebben beleefd. MATLAB leek een aardige verbetering te zijn. De MATLAB-taal is eenvoudig en flexibel, als men met matrices rekent is het verbluffend snel, het is productief om met de interactieve 'prompt' te werken en de grafische mogelijkheden zijn fantastisch (zie ook 'Snelle oudjes gaan MATLAB' door Kees Maas en Theo Olsthoorn in Stromingen 3 (1997), nr 4). Maar er kleven ook nadelen aan MATLAB. Alles is een matrix en dat is dan weer lastig, vooral als je een iets ingewikkelder computerprogramma zit te schrijven. Zo gauw je een do-loopje gebruikt, gaat de rekensnelheid met sprongen achteruit. Verder is object-georiënteerd programmeren in MATLAB erg onhandig en is MATLAB relatief duur.

In dit artikel wordt een pleidooi gehouden voor een vrij nieuwe programmeertaal, namelijk Python. Python is een krachtige programmeertaal met een zeer eenvoudige syntaxis; een Python-programma is gemiddeld zo'n tien keer korter dan een C++ programma. In Python kunnen getallen gedefinieerd worden als een matrix, maar ook als andere datatypen (verschillende handige datatypen zijn ingebouwd, andere kunnen gedefinieerd worden). Python kan ook interactief gebruikt worden, net als MATLAB, en je kunt er volledig object-georiënteerd in programmeren als je dat wilt. Het is eenvoudig om functies van een bestaand programma, geschreven in een andere computertaal, met Python te gebruiken. Verder is het gratis, open-source, en het werkt onder alle besturingssystemen (dus ook Linux, Unix, en Apple's MacOS). Doordat het open-source is, wordt er door velen aan gewerkt en zijn er vele 'packages' beschikbaar (vergelijkbaar met toolboxes in MATLAB, maar dan gratis). Python is begonnen bij het Centrum voor Wiskunde en Informatica in Amsterdam en wordt ontwikkeld onder leiding van Guido van Rossum, die inmiddels in Amerika woont. Python is vernoemd naar het legendarische Monty Python's Flying Circus; op de Python-boeken van uitgever O'Reilly prijkt een Python-slang.

Python voor (geo)hydrologen

Op de Python-webpagina staan een stuk of wat leerboeken, waar geïnteresseerden zich op uit kunnen leven (sterk aan te raden). Hier worden een paar voorbeelden voor geohydrolo-

Mark Bakker is werkzaam bij Department of Biological and Agricultural Engineering, University of Georgia, Athens, GA 30602, USA, mbakker@engr.uga.edu.

gen gegeven en een paar verschillen tussen MATLAB en Python besproken. De voorbeelden kunnen zo met Python nagedaan worden, tenminste, als je Python downloadt van www.python.org (wel aan het begin van het voorbeeld beginnen, want het voorbeeld loopt door). Wat installatietips worden aan het einde van dit artikel gegeven.

Basisprincipes

Variabelen kunnen gedefinieerd worden zoals in MATLAB. Type bijvoorbeeld bij de Pythonprompt ('>>>' is de Pythonprompt):

```
>>> Q=3.2
>>> b=1.6
>>> Q/b
2.0
```

Vrijwel alle functies moeten geïmporteerd worden voordat ze kunnen worden gebruikt. In MATLAB hoeft dat niet, omdat daar functies op bestandsnaam gezocht worden. In Python zitten vele functies in één bestand, maar je hoeft ze natuurlijk maar één keer te importeren en dan kun je ze zo vaak gebruiken als je wilt. Er zijn vele *packages* met handige functies. De voor de geohydroloog meest interessante package is wellicht het *Numeric* package (van http://sourceforge.net/projects/numpy). In *Numeric* zitten bijvoorbeeld vele standaard wiskundige functies. Laten we *Numeric* eens importeren en wat functies aanroepen:

```
>>> from Numeric import *
>>> sqrt(Q)
1.7888543819998319
>>> c=cos(b)
>>> c
-0.029199522301288815
```

Zoals je in het voorbeeld kunt zien, wordt in Python een antwoord alleen naar het scherm geschreven als het niet in een variabele wordt opgeslagen. *Numeric* maakt het ook mogelijk om matrices te definiëren (het kost alleen iets meer typewerk). Maar pas op, sommige matrix-manipulaties werken net andersom als in MATLAB. Een matrix heet een 'array' in *Numeric:*

```
>>> r=array([1,2,3])
>>> Q/(2.0*pi)*log(r)
array([ 0.        ,  0.35301696,  0.55951864])
```

Nog even iets over indices. Die beginnen in Python op 0, en als je een gedeelte van een vector wilt hebben en de begin- en eindwaarden van de index opgeeft, dan stopt Python vóór de eindwaarde:

```
>>> r[0]
1
>>> r[0:2]
array([1, 2])
>>> r[0:len(r)]
array([1, 2, 3])
>>>
```

Je kunt natuurlijk ook een tweedimensionale matrix maken:

```
>>> A=array([[1,2],[3,4]])
>>> A
array([[1, 2],
       [3, 4]])
```

Het vermenigvuldigen van matrices is standaard term bij term vermenigvuldiging (net andersom als bij MATLAB), en als matrixvermenigvuldiging bedoeld wordt, dan moet die functie apart aangeroepen worden:

```
>>> A*A          # term bij term
array([[ 1,  4],
       [ 9, 16]])
>>> dot(A,A)     # matrix vermenigvuldiging
array([[ 7, 10],
       [15, 22]])
```

Het teken '#' gaat vooraf aan commentaar. Het standaard term bij term vermenigvuldiging heeft ook een handigheidje, dat MATLAB niet heeft. Je kunt twee matrices, die niet dezelfde vorm hebben, wel met elkaar vermenigvuldigen; *Numeric* noemt dit 'broadcasting'. Bijvoorbeeld

```
>>> v=array([1,2])
>>> v*A
array([[1, 4],
       [3, 8]])
```

Hier werd de eerste kolom van A met 1 vermenigvuldigd, en de tweede met 2.

Een groot aantal functies voor matrices zijn ook beschikbaar in het MLab package (dat met *Numeric* mee komt). De functienamen zijn zelfs hetzelfde als in MATLAB, bijvoorbeeld het uitrekenen van de eigenwaarden en -vectoren van een matrix:

```
>>> from MLab import *
>>> [x,v]=eig(A)
>>> x
array([ 5.37228132, -0.37228132])
>>> v
```

```
array([[ 0.41597356,  0.90937671],
       [-0.82456484,  0.56576746]])
```

Let op dat de eigenwaarden en -vectoren wel andersom dan in MATLAB teruggeleverd worden. Een ander handig package is het onlangs uitgebrachte *Scipy* package (van http://www.scipy.org), want het heeft een groot aantal speciale functies, bijvoorbeeld gemodificeerde Besselfuncties (handig voor een put in semi -spanningswater):

```
>>> from scipy.special import *
>>> k0(r)
array([ 0.42102444,  0.11389387,  0.0347395 ])
```

Functies in Python

Laten we nu eens een eigen Python-functie schrijven. De syntaxis van een functie in Python is zoals gezegd supereenvoudig. Er komt geen 'end' voor in een Python-functie, ook niet na bijvoorbeeld een 'if'-constructie. De structuur van de functies wordt gedefinieerd door consequent in te springen. Verder moet er na een opdracht die op de volgende regel verder gaat, een dubbele punt geplaatst worden (zoals een 'if'-constructie, of de functie zelf); daarna moet er natuurlijk wel ingesprongen worden. Python-functies heten 'def'. Hier is een functie die de stijghoogte van een Thiemput uitrekent. Je kunt deze in een bestandje opslaan en importeren, maar je kunt hem ook gewoon bij de prompt typen, dan zit hij in het Python-geheugen en kun je hem aanroepen (let op hoe er consequent wordt ingesprongen):

```
>>> def Thiem(x,y,x0,y0,Q,T):
        rsq = (x-x0)**2 + (y-y0)**2
        head = Q / (4*pi*T) * log(rsq)
        return head

>>> Thiem(4,2,0,0,200,20)
2.3839279975801984
```

Echt mooi wordt het als we van de object-georiënteerde mogelijkheden gebruik maken. Laten we een 'class' maken die 'Put' heet en vijf attributen heeft: de *x,y*-coordinaten, het debiet, de straal van de put en de straal in het kwadraat. Verder wordt de stijghoogtefunctie (nu head genaamd) aan de class toegevoegd, en het geheel wordt in een bestand 'put.py' opgeslagen. Hier is het bestand:

```
from Numeric import *
class Put:
   def __init__(self,x0,y0,Q,rw):
      self.x0 = x0 ; self.y0 = y0
      self.Q  = Q  ; self.rw = rw ; self.rwsq = rw*rw
   def __repr__(self):
      return \
```

```
        'Put xw,yw,Q,rw: ' + str((self.x0,self.y0,self.Q,self.rw))
    def head(self,x,y,T):
        rsq = (x-self.x0)**2 + (y-self.y0)**2
        if rsq < self.rwsq:
            head = 0.0
        else:
            head = self.Q / (4*pi*T) * log(rsq/self.rwsq)
        return head
```

Op de eerste regel wordt de *Numeric* package ingelezen, hetgeen betekent dat binnen het bestand functies uit de *Numeric* package gebruikt mogen worden. In elk bestand waar de *Numeric* package gebruikt wordt, moet hij geïmporteerd worden. Daarna begint de class met het geven van zijn naam, waarna ingesprongen moet worden. De Put class heeft drie functies. De eerste, `__init__`, is de constructor (elke class moet een constructor met deze wat jammere naam hebben). Daar worden ook alle attributen gedefinieerd; een attribuut is een variabele die per instance van een object opgeslagen wordt. Voor de Put class moeten er vier opgegeven worden (de coördinaten van de put, het debiet, en de straal); een vijfde attribuut wordt uitgerekend en is de straal in het kwadraat. De tweede functie, `__repr__`, is de representatiefunctie, waarin gespecificeerd wordt hoe een instance van de class naar het scherm geschreven wordt. Als je deze functie niet definieert, dan schrijft Python een wat minder bruikbare (standaard) versie naar het scherm. Merk ook op dat de achterwaartse schuine streep, '\', aangeeft dat een regel op de volgende regel doorgaat. De derde functie is de stijghoogte die een beetje aangepast is, zodat precies in het midden van de put de stijghoogte niet min oneindig wordt, maar gelijk aan de stijghoogte op de rand van de put. Merk ook op dat alle functies in een class als eerste argument 'self' moeten hebben.

Laten we de Put class eens gebruiken. We importeren hem eerst en zullen dan twee putten invoeren, één is het spiegelbeeld van de ander. Een put kan ingevoerd worden door de constructor aan te roepen. De putten worden opgeslagen in de variabelen *w1* en *w2*. Door het typen van één van deze namen wordt de representatiefunctie aangeroepen:

```
>>> from put import *
>>> w1=Put(0,20,200,.1)
>>> w2=Put(0,-20,-200,.1)
>>> w1
Put xw,yw,Q,rw: (0, 20, 200, 0.10000000000000001)
>>> w2
Put xw,yw,Q,rw: (0, -20, -200, 0.10000000000000001)
```

Functies van een object kunnen aangeroepen worden als `instance.functie(variabelen)`, en een attribuut als `instance.attribuut`. Let wel op dat na een functie er *altijd* twee haakjes met de invoervariabelen moeten staan, ook als de functie geen invoervariabelen heeft (dan staan er twee haakjes met niks ertussen). De stijghoogte langs de x-as van put *w1* is, zoals verwacht mag worden, min de stijghoogte van put *w2*:

```
>>> w1.Q
```

```
200
>>> w1.head(0,0,20)
8.4325339895575357
>>> w2.head(0,0,20)
-8.4325339895575357
```

Nu zou het handiger zijn om een stijghoogtefunctie te maken, die de stijghoogte van alle putten bij elkaar optelt. Alle putten worden daartoe verzameld in een lijst, of 'list' zoals het datatype in Python heet. Een list is een datatype dat bestaat uit een lijst met allerlei willekeurige datatypen (erg handig). Een lijst staat tussen vierkante haken. Hier is een lijst met bijvoorbeeld een nummer, een woord, en een vector:

```
>>> lijst = [3,'oudje',array([1,2])]
>>> lijst
[3, 'oudje', array([1, 2])]
```

We kunnen dus ook een lijst maken met alle putten; laten we ook de transmissiviteit opslaan in een variabele T:

```
>>> allePutten = [w1,w2]
>>> T = 20
```

Dan kunnen we nu een functie definiëren, die de stijghoogte van alle putten bij elkaar optelt:

```
>>> def stijghoogte(allePutten,x,y,T):
        head = 0.0
        for w in allePutten:
            head = head + w.head(x,y,T)
        return head

>>> stijghoogte(allePutten,0,0,T)
0.0
>>> stijghoogte(allePutten,20,10,T)
-0.76037184828498283
```

Een plaatje

Wat zou een handige programmeertaal zijn zonder grafische uitvoer? Hiervoor zijn in Python verschillende packages beschikbaar (nog niet zo mooi als in MATLAB), bijvoorbeeld *biggles* (van http://biggles.sourceforge.net). Eerst maken we een rijvector van x en een rijvector van y aan, waar we de stijghoogte uit willen rekenen, bijvoorbeeld van –25 tot +25 met een interval van 1:

```
>>> x=arange(-25.,26.,1.)
>>> y=arange(-25.,26.,1.)
```

De 'arange' functie geeft een vector. Let hierbij op dat Python stopt voordat het de maximale waarde (in dit geval 26) bereikt heeft. Bereken nu de stijghoogte op elk punt van het grid (wel eerst even een matrix aanmaken van de juiste grootte, met allemaal nullen; de toevoeging 'd' betekent dat alle nullen reële getallen zijn):

```
>>> h=zeros( (len(x),len(y)),'d')
>>> for i in range(len(x)):
      for j in range(len(y)):
            h[i,j] = stijghoogte( allePutten, x[i], y[j], T )
```

Nu kunnen we een plaatje maken. Eerst gaan we de nieuwe package *biggles* importeren en dan een plaatje aanmaken dat 'plaatje' heet:

```
>>> from biggles import *
>>> plaatje = FramedPlot()
```

Vergeet vooral de dubbele haken na FramedPlot niet (dit is immers een functie). Nu kunnen we van alles aan het plaatje toevoegen (zie ook de voorbeelden op de *biggles* website), bijvoorbeeld namen langs assen, tickmarks, enzovoort. Maar dus ook contourlijnen:

```
>>> plaatje.add( Contours( h, x, y, (-10,10), levels=19 ) )
```

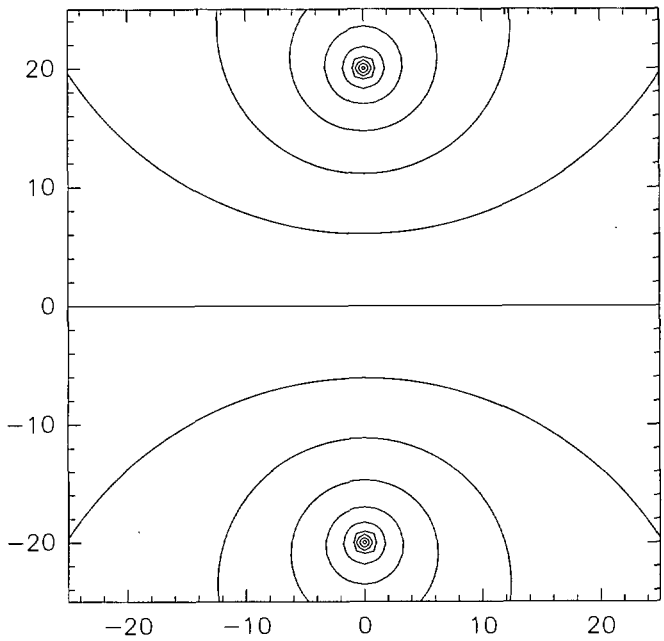
De syntaxis voor de contourlijnen is enigszins onhandig (meneer Biggles gebruikt dit vast niet vaak zelf). In bovenstaande opdracht is -10 het minimum, 10 het maximum, en wordt de stapgrootte uitgerekend als (max-min)/(levels+1). Maar gelukkig kunnen we eenvoudig een contourfunctie maken, die onze eigen favoriete syntaxis gebruikt. Om te zorgen dat de schaal langs de x- en y-as hetzelfde is, moeten we dit ook nog even opgeven:

```
>>> plaatje.xrange=(-25,25)
>>> plaatje.yrange=(-25,25)
>>> plaatje.aspect_ratio=1
```

De eerste twee opdrachten geven de minimum en maximum waarden langs de as aan. Nu is de schaalverdeling langs de assen hetzelfde, tenminste als het plaatje als vierkant gedefinieerd wordt (gelukkig is dat de standaard, dus daar hoeven we ons geen zorgen over te maken). We kunnen nu het plaatje te zien krijgen door te typen:

```
>>> plaatje.show()
```

en Python maakt een PNG-bestand van het plaatje aan, opent het programma op jouw machine dat standaard deze files weergeeft (de webbrowser bijvoorbeeld, of zoiets ergs als PhotoEditor), en voila, je ziet het onderstaande plaatje. Je kunt ook PNG- of PostScript-bestanden wegschrijven. Mocht je inderdaad je webbrowser gebruiken voor PNG-bestanden, en je hebt weer wat aan het plaatje toegevoegd en plaatje.show() getyped, dan krijg je de verandering niet altijd direct in je webbrowser te zien. Mocht dat zo zijn, klik dan Ctrl-Shift-Reload zodat de webbrowser het plaatje bijwerkt.



Snelle oudjes gaan Python

Python kan ook gebruikt worden om de voorbeelden uit het 'Snelle oudjes'-artikel uit te rekenen (dus even Stromingen 3 aflevering 4 erbij pakken). We moeten wel een functie schrijven voor de systeemmatrix en voor matrixfuncties; de MATLAB-functie 'funm' is (nog) niet standaard in Python. Deze worden opgeslagen in een filetje funm.py. Let op hoe fantastisch mooi het is dat je een functie f (bijvoorbeeld de wortel of een e-macht) gewoon door kunt geven aan een functie zonder enige poespas; Python doet dat moeiteloos:

```
from MLab import *
from LinearAlgebra import *

def funm(f,A):
    [x,v] = eig(A)
    v = transpose(v)
    rv = dot( dot ( v , diag(f(x)) ) , inverse(v) )
    return rv

def sysmat(kD,c):
    n = len(kD)
    a = 1 / (kD * c)
    b = 1 / (kD * array( list(c[1:]) + [1e30] ))
    A = diag(a+b) - diag(a[1:n],-1) - diag(b[0:n-1],1)
    return A
```

Dan kunnen we nu de systeemmatrix van bladzijde 25 aanmaken:

```
>>> from funm import *
>>> from Numeric import *
>>> kD=array([100,200,300,400],'d')
>>> c=array([500,600,700,800],'d')
>>> A=sysmat(kD,c)
```

en de gegeneraliseerde formule van Mazure toepassen zoals op bladzijde 27:

```
>>> x=500
>>> h=array([[1],[1],[1],[1]],'d')
>>> phi = dot( funm( exp, -x * funm( sqrt, A ) ), h )
>>> phi
array([[ 0.27785487],
       [ 0.52727062],
       [ 0.67255971],
       [ 0.73549258]])
```

Dit leidt tot hetzelfde resultaat als in het 'Snelle oudjes'-artikel.

Andere Python-mogelijkheden

Bovenstaande voorbeelden geven hopelijk een idee van de verbluffende mogelijkheden van Python. Python wordt doorlopend ontwikkeld en verbeterd (alleen de prijs kan niet verbeterd worden). De snelheid van Python laat misschien ook nog iets te wensen over voor grote programma's, maar Python is zeker niet langzamer dan MATLAB. Als gebruik gemaakt wordt van functies uit packages, is Python een stuk sneller. Voor de (geo)hydroloog zijn er ook nog andere zeer nuttige packages beschikbaar. Zo bevat het *Numeric* package ook een package dat *LinearAlgebra* heet. Dit package heeft mooie functies zoals `solve_linear_equations` en `linear_least_squares`, die verder geen uitleg behoeven. Er zijn ook packages voor bijvoorbeeld Fast Fourier Transforms (het *FFT* package) of om mooie grafische schillen te bouwen (het *TkInter* package).

Het Tim-project

Enkele jaren geleden hebben de Nederlandse RIZA en de Amerikaanse EPA besloten samen de ontwikkeling van open-source, object-georiënteerde, hydrologische rekenmodellen te stimuleren. In eerste instantie heeft die samenwerking zich gericht op de ontwikkeling van analytische-elementen-programma's voor grondwatermodellering. Vooral voor ontwikkelaars van analytische-elementen-programma's is het handig om object-georiënteerd te programmeren, omdat dit de uitwisseling van gedeelten van computerprogramma's buitengewoon vergemakkelijkt. De ontwikkeling van deze programma's is, tot nu toe, gebeurd in Python, met alle voordelen van de Pythontaal. De ontwikkeling van deze programma's is het *Tim* project genoemd, naar de figuur 'Tim de Duivelskunstenaar' in de film 'Monty

Python and the Holy Grail'. Er zijn twee programma's ontwikkeld: Tim^{SL} (de éénlaagsversie), en Tim^{ML} (de meerlagenversie). Het basisontwerp voor beide programma's is gelijk, hoewel het ontwerp van Tim^{SL} veel uitgebreider is.

Tim^{SL} is nog een beta-versie (momenteel versie 0.3) en heeft zowel stationaire als tijdsafhankelijke analytische elementen. Het object-georienteerde ontwerp maakt het makkelijk om met meerdere mensen tegelijk aan Tim^{SL} te werken. Nieuwe analytische elementen kunnen zonder enige ingreep in de bestaande code van Tim^{SL} worden toegevoegd.

Tim^{ML} is momenteel beschikbaar als versie 1.0. Het wijkt af van andere meerlagen analytische-elementen-programma's in de zin dat de lek tussen lagen exact uitgerekend wordt, zonder dat daar een belegging met area elementen voor nodig is (zoals bijvoorbeeld in MLAEM).

Beide *Tim*-programma's zijn gratis beschikbaar onder een open-source licentie (je wordt zelfs aangemoedigd om er zelf wat aan toe te voegen!). Voor beide programma's zijn uitgebreide handleidingen beschikbaar.

Webpagina's en een tip voor beginners

Hier zijn eerst de webadressen. Python is beschikbaar op http://www.python.org, het *Numeric* package op http://sourceforge.net/projects/numpy, het *Scipy* package op http://www.scipy.org, en het *biggles* package op http://biggles.sourceforge.net. Vrijwel alle packages variëren per Python-versie, zorg er dus voor dat je het package voor de juiste Python-versie downloadt (alle genoemde packages zijn momenteel beschikbaar voor Python 2.1.X). Als je *biggles* wilt gebruiken, lees dan even hun installatiebeschrijving, want je moet nog drie andere files van het net halen (maar ze geven netjes aan waar). De webpagina van het *Tim* project, waar zowel Tim^{SL} en Tim^{ML} als de handleidingen gedownload kunnen worden, is www.engr.uga.edu/~mbakker/Tim.html. Het laatste nieuws voor Mac-gebruikers is dat de nieuwste versie van het besturingssysteem (MacOS 10.2) Python 2.2 automatisch installeert.

Nog een laatste tip om te beginnen. Je kunt Python starten door een 'Command Window' te openen (wat vroeger een DOS-venster was), naar de directory te gaan van waaruit je wilt werken, en 'python' te typen (tenminste, als je Python aan je pad toegevoegd hebt). Je kunt ook naar de 'Start' knop gaan, en 'IDLE', een Python-schil, openen. In IDLE worden commando's gekleurd weergegeven, je krijgt een soort ballonhelp met het geven van argumenten van functies, en andere handigheidjes (ik gebruik altijd IDLE). Let wel op dat als je vanuit IDLE werkt, je nog wel even op moet geven waar je je eigen Python-files opgeslagen hebt. Als je files bijvoorbleed in een directory 'c:\stromingen' staan, type dan

```
>>> from sys import *
>>> path.append('c:/stromingen')
```

Let op dat de achterwaartse schuine streep nu een voorwaartse is (Python werkt immers onder allerlei besturingssystemen; op niet door Bill Gates ontworpen systemen worden gelukkig geen achterwaartse strepen gebruikt). Het is natuurlijk niet handig om dit altijd eerst te moeten typen, maar gelukkig kun je je eigen favoriete directory ook aan de PYTHONPATH environment variabele toevoegen. Hoe dit moet, kun je zo vinden in de Python- of *Tim*-handleidingen.

Verantwoording

Het *Tim*-project werd financieel mogelijk gemaakt door RIZA en de USEPA. Tim^{SL} is mede ontwikkeld door Vic Kelson en Willem Jan Zaadnoordijk.