PCSE Documentation

Release 5.5

Allard de Wit

Mar 01, 2023

Contents

PCSE (Python Crop Simulation Environment) is a Python package for building crop simulation models, in particular the crop models developed in Wageningen (Netherlands). PCSE provides the environment to implement crop simulation models, the tools for reading ancillary data (weather, soil, agromanagement) and the components for simulating biophysical processes such as phenology, respiration and evapotranspiration. PCSE also includes implementations of the WOFOST LINGRA and LINTUL3 crop and grassland simulation models which have been widely used around the world. For example, WOFOST has been implemented in the MARS crop yield forecasting system which is used operationally for crop monitoring and yield prediction in Europe and beyond.

Originally, models developed in Wageningen were often written using FORTRAN or the FORTRAN Simulation Environment (FSE). Both are very good tools, but they have become somewhat outdated and are difficult to integrate with many of the great tools that are available nowadays (XML, databases, web, etc). Like so many other software packages, PCSE was developed to facilitate my own research work. I wanted something that was more easy to work with, more interactive and more flexible while still implementing the sound computational approach of FSE. For this reason PCSE was developed in Python hsa become an important programming language for scientific purposes.

Traditionally, crop simulation models in Wageningen have been provided including the full source code. PCSE is no exception and its source code is open and licensed under the European Union Public License. PCSE runs on Python 2.7+ and 3.2+ and has a decent test coverage of the implementation of the biophysical processes.

CHAPTER 1

What's new

1.1 An overview of new features and fixes

1.1.1 What's new in PCSE 5.5

PCSE 5.5 has the following new features:

- WOFOST version 8.0 (beta) has been included which has variants for potential (PP), water-limited (WLP) and nutrient + water-limited (NWLP) production. Note that dynamics for N/P/K are included in all model variants but for the PP and WLP variants the supply of N/P/K is assumed to be unlimited. Note that this a beta version because testing of the N/P/K limited growth against experimental data has so far been limited. Nevertheless, the dynamics for N/P/K are based on well known principles from other models and rely on the concept of dilution curves that define the maximum, critical and residual N/P/K concentration in the crop.
- A full implementation of the LINGRA and LINGRA-N grassland simulation models are now included. This model allows to make estimates of productivity of rye grass.
- WOFOST 7.1 has been upgraded to 7.2, this is mainly to be consistent with the updated system description for WOFOST at https://wofost.readthedocs.io. Old code that relies on importing WOFOST 7.1 will keep working though.
- The WOFOST 7.2 phenology module can now be imported as a standalone model. This is useful when calibration is limited to phenology as it greatly increases the model performance.
- The FAO Water Requirement Satisfaction Index is included as a model.

1.1.2 What's new in PCSE 5.4

PCSE 5.4 has the following new features:

- PCSE is now fully compatible with python3 (>3.4) while still remaining compatibility with python 2.7.14
- The NASAPOWERWeatherDataProvider has been upgraded to take the new API into account

1.1.3 What's new in PCSE 5.3

PCSE 5.3 has the following new features:

- The WOFOST crop parameters have been reorganized into a new data structure and file format (e.g. YAML) and are available from github. PCSE 5.3 provides the *YAMLCropDataProvider* to read the new parameters files. The YAMLCropDataProvider works together with the AgroManager for specifying parameter sets for crop rotations.
- A new *CGMSEngine* that mimics the behaviour of the classic CGMS. This means the engine can be run up till a specified date. When maturity or harvest is reached, the value of all state variables will be retained and kept constant until the specified date is reached.
- Caching was added to the CGMS weather data providers, this is particularly useful for repeated runs as the weather data only have to be retrieved once from the CGMS database.

Some bugs have been fixed:

- The NASA POWER database moved from http:// to https:// so an update of the NASAPowerWeatherDataProvider was needed.
- When running crop rotations it was found that python did not garbage collect the crop simulation objects quick enough. This is now fixed with an explicit call to the garbage collector.

1.1.4 What's new in PCSE 5.2

PCSE version 5.2 brings the following new features:

- The LINTUL3 model has been implemented in PCSE. LINTUL3 is a simple crop growth model for simulating growth conditions under water-limited and nitrogen-limited conditions.
- A new module for N/P/K limitations in WOFOST was implemented allowing to simulate the impact of N/P/K limitations on crop growth in WOFOST.
- A new *AgroManager* which greatly enhances the way that AgroManagement can be handled in PCSE. The new agromanager can elegantly combine cropping calendars, timed events and state events also within rotations over several cropping campaigns. The AgroManager uses a new format based on YAML to store agromanagement definitions.
- The water-limited production simulation with WOFOST now supports irrigation using the new AgroManager. An example notebook has been added to explain the different irrigation options.
- Support for reading input data from a CGMS8 and CGMS14 database

Changes in 5.2.5:

- Bug fixes in agromanager causing problems with crop_end_type="earliest" or "harvest"
- Caching was added to the CGMS weather data providers
- Added CGMSEngine that mimics behaviour of the classic CGMS: after the cropping season is over, a call to _run() will increase the DAY, but the internal state variables do not change anymore, although they are kept available and can be queried and stored in OUTPUT.

1.1.5 What's new in PCSE 5.1

PCSE version 5.1 brings the following new features:

- Support for reading input data (weather, soil, crop parameters) from a CGMS12 database. CGMS is the acronym for Crop Growth Monitoring System and was developed by Alterra in cooperation with the MARS unit of the Joint Research Centre for crop monitoring and yield forecasting in Europe. It uses a database structure for storing weather data and model simulation results which can be read by PCSE. See the MARSwiki for the database definition.
- The ExcelWeatherDataProvider: Before PCSE 5.2 the only file-based format for weather data was the CABO weather format read by the *CABOWeatherDataProvider*. Althought the format is well documented, creating CABO weather files is a bit cumbersome as for each year a new file has to be created and mistakes are easily made. Therefore, the *ExcelWeatherDataProvider* was created that reads its input from a Microsoft Excel file. See here for an example of an Excel weather file: `downloads/nl1.xlsx`.

CHAPTER 2

Crop models Available in PCSE

2.1 Models available in PCSE

The following table lists the models that are available in PCSE and can be imported from *pcse.models* package.

Model name	Description
Wofost72_PP	An implementation of WOFOST 7.2 for potential production scenarios.
Wofost72_WLP_FD	An implementation of WOFOST 7.2 for water-limited production scenarios with freely draining soils.
Wofost80_PP_beta	An implementation of WOFOST 8.0 for potential production scenarios including N/P/K dynamics
Wofost80_WLP_FD_beta	An implementation of WOFOST 8.0 for water-limited production scenarios including N/P/K dynamics for freely draining soils.
Wofost80_NWLP_FD_beta	An implementation of WOFOST 8.0 for water-limited and nutrient-limited production scenarios including N/P/K dynamics for freely draining soils.
LINGRA_PP	A LINGRA implementation for simulating potential production scenarios.
LINGRA_WLP_FD	A LINGRA implementation for simulating water-limited production scenarios with freely draining soils.
LINTUL3	An implementation of the LINTUL3 model for production scenarios under water-limited and nitrogen- limited production scenarios.
FAO_WRSI	An implementation of the Water Requirement Satisfaction Index model. This re-uses components from WOFOST to create a simpler approach which computes water requirements and water availability.
Wofost72_Phenology	The phenology modules from WOFOST 7.2 as a standalone model. This is purely for convenience as in some cases running the phenology is sufficient and this is much faster then running the full WOFOST model.

CHAPTER 3

User guide

3.1 User Guide

3.1.1 Background of PCSE

Crop models in Wageningen

The *Python Crop Simulation Environment* was developed because of a need to re-implement crop simulation models that were developed in Wageningen. Many of the Wageningen crop simulation models were originally developed in FORTRAN77 or using the *FORTRAN Simulation Translator (FST)*. Although this approach has yielded high quality models with high numerical performance, the inherent limitations of models written in FORTRAN is also becoming increasingly evident:

- The structure of the models is often rather monolithic and the different parts are very tightly coupled. Replacing parts of the model with another simulation approach is not easy.
- The models rely on file-based I/O which is difficult to change. For example, interfacing with databases is complicated in FORTRAN.
- In general, with low-level languages like FORTRAN, simple things already take many lines of code and mistakes are easily made, particularly by agronomists and crop scientist that have limited experience in developing or adapting software.

To overcome many of the limitations above, the Python Crop Simulation Environment (PCSE) was developed. It provides an environment for developing simulation models as well as a number of implementations of crop simulation models. PCSE is written in pure Python code which makes it more flexible, easier to modify and extensible allowing easy interfacing with databases, graphical user interfaces, visualization tools and numerical/statistical packages. PCSE has several interesting features:

- Implementation in pure Python. The core system has a small number of dependencies outside the Python standard library. However many data providers require certain packages to be installed. Most of these can be automatically installed from the Python Package Index (PyPI) (*SQLAlchemy*, *PyYAML*, *xlrd*, *openpyxl*, *requests*) and in processing of the output of models is most easily done with *pandas* DataFrames.

- Modular design allowing you to add or change components relatively quickly with a simple but powerful approach to communicate variables between modules.
- Similar to FST, it enforces good model design by explicitly separating parameters, rate variables and state variables. Moreover PCSE takes care of the module initialization, calculation of rates of changes, updating of state variables and actions needed to finalize the simulation.
- Input/Output is completely separated from the simulation model itself. Therefore PCSE models can easily read from and write to text files, databases and scientific formats such as HDF or NetCDF. Moreover, PCSE models can be easily embedded in, for example, docker containers to build a web API around a crop model.
- Built-in testing of program modules ensuring integrity of the system

Why Python

PCSE was first and foremost developed from a scientific need, to be able to quickly adapt models and test ideas. In science, Python is quickly becoming a tool for implementing algorithms, visualization and explorative analysis due to its clear syntax and ease of use. An additional advantage is that the C implementation of Python can be easily interfaced with routines written in FORTRAN and therefore many FORTRAN routines can be reused by simulation models written with PCSE.

Many packages exist for numeric analysis (e.g. NumPy, SciPy), visualisation (e.g. MatPlotLib, Chaco), distributed computing (e.g. IPython, pyMPI) and interfacing with databases (e.g. SQLAlchemy). Moreover, for statistical analyses an interface with R-project can be established through Rpy or Rserve. Finally, Python is an Open Source interpreted programming language that runs on almost any hardware and operating system.

Given the above considerations, it was quickly recognized that Python was a good choice. Although, PCSE was developed for scientific purposes, it has already been implemented for tasks in production environments and has been embedded in container-based web services.

History of PCSE

Up until version 4.1, PCSE was called "PyWOFOST" as its primary goal was to provide a Python implementation of the WOFOST crop simulation model. However, as the system has grown it has become evident that the system can be used to implement, extend or hybridize (crop) simulation models. Therefore, the name "PyWOFOST" became too narrow and the name Python Crop Simulation Environment was selected in analog with the FORTRAN Simulation Environment (FSE).

Limitations of PCSE

PCSE also has its limitations, in fact there are several:

- Speed: flexibility comes a at a price; PCSE is considerably slower than equivalent models written in FORTRAN or another compiled language.
- The simulation approach in PCSE is currently limited to rectangular (Euler) integration with a fixed daily time-step. Although the internal time-step of modules can be made more fine-grained if needed.
- No graphical user interface. However the lack of a user interface is partly compensated by using PCSE with the pandas package and the Jupyter notebook. PCSE output can be easily converted

to a pandas *DataFrame* which can be used to display charts in an Jupyter notebook. See also my collection of notebooks with examples using PCSE

License

The source code of PCSE is made available under the European Union Public License (EUPL), Version 1.2 or as soon they will be approved by the European Commission - subsequent versions of the EUPL (the “Licence”). You may not use this work except in compliance with the Licence. You may obtain a copy of the Licence at: https://joinup.ec.europa.eu/community/eupl/og_page/eupl

The PCSE package contains some modules that have been taken and/or modified from other open source projects:

- the *pydispatch* module obtained from http://pydispatcher.sourceforge.net/ which is distributed under a BSD style license.
- The *traitlets* module which was taken and adapted from the *IPython* project (https://ipython.org/) which are distributed under a BSD style license. A PCSE specific version of *traitlets* was created and is available here

See the project pages of both projects for exact license terms.

3.1.2 Installing PCSE

Requirements and dependencies

PCSE is being developed on Ubuntu Linux 18.04 and Windows 10 using python 3.7 and python 3.8 As Python is a platform independent language, PCSE works equally well on Linux, Windows or Mac OSX. Before installing PCSE, Python itself must be installed on your system which we will demonstrate below. PCSE has a number of dependencies on other python packages which are the following:

```
- SQLAlchemy>=0.8.0
- PyYAML>=3.11
- xlrd>=0.9.3
- openpyxl>=3.0
- requests>=2.0.0
- pandas>=0.20
- traitlets-pcse==5.0.0.dev
```

The last package in the list is a modified version of the traitlets package which provides some additional functionality used by PCSE.

Setting up your python environment

A convenient way to set up your python environment for PCSE is through the Anaconda python distribution. In the present PCSE Documentation all examples of installing and using PCSE refer to the Windows 10 platform.

First, we suggest you download and install the MiniConda python distribution which provides a minimum python environment that we will use to bootstrap a dedicated environment for PCSE. For the rest of this guide we will assume that you use Windows 10 and install the 64bit miniconda for python 3 (`Miniconda3-latest-Windows-x86_64.exe`). The environment that we will create contains not only the dependencies for PCSE, it also includes many other useful packages such as IPython, **'Pandas'_** and the Jupyter notebook. These packages will be used in the Getting Started section as well.

After installing MiniConda you should open a command box and check that conda is installed properly:

```
(py3_pcse) C:\>conda info

     active environment : py3_pcse
    active env location : C:\data\Miniconda3\envs\py3_pcse
            shell level : 3
       user config file : C:\Users\wit015\.condarc
 populated config files : C:\Users\wit015\.condarc
          conda version : 4.9.2
    conda-build version : not installed
         python version : 3.8.5.final.0
       virtual packages : __win=0=0
                          __archspec=1=x86_64
       base environment : C:\data\Miniconda3  (writable)
           channel URLs : https://conda.anaconda.org/conda-forge/win-64
                          https://conda.anaconda.org/conda-forge/noarch
                          https://repo.anaconda.com/pkgs/main/win-64
                          https://repo.anaconda.com/pkgs/main/noarch
                          https://repo.anaconda.com/pkgs/r/win-64
                          https://repo.anaconda.com/pkgs/r/noarch
                          https://repo.anaconda.com/pkgs/msys2/win-64
                          https://repo.anaconda.com/pkgs/msys2/noarch
          package cache : C:\data\Miniconda3\pkgs
                          C:\Users\wit015\.conda\pkgs
                          ␣
↪C:\Users\wit015\AppData\Local\conda\conda\pkgs
       envs directories : C:\data\Miniconda3\envs
                          C:\Users\wit015\.conda\envs
                          ␣
↪C:\Users\wit015\AppData\Local\conda\conda\envs
               platform : win-64
             user-agent : conda/4.9.2 requests/2.24.0 CPython/3.8.5␣
↪Windows/10 Windows/10.0.18362
          administrator : False
             netrc file : None
           offline mode : False
```

Now we will use a Conda environment file to recreate the python environment that we use to develop and run PCSE. First you should download the conda environment file which comes in two flavours, an environment for running PCSE on python 3 (`downloads/py3_pcse.yml`) and one for python 2 (`downloads/py2_pcse.yml`). It is strongly recommended to use the python 3 version as python 2 is not maintained anymore. Both environments include the Jupyter notebook and IPython which are needed for running the *getting started* section and the example notebooks. Save the environment file on a temporary location such as `d:\temp\make_env\`. We will now create a dedicated virtual environment using the command `conda env create` and tell conda to use the environment file for python3 with the option `-f p3_pcse.yml` as show below:

```
(C:\Miniconda3) D:\temp\make_env>conda env create -f py3_pcse.yml
Fetching package metadata .............
Solving package specifications: .
intel-openmp-2 100% |###############################| Time: 0:00:00   6.39␣
↪MB/s
```

(continues on next page)

(continued from previous page)

```
... Lots of output here

Installing collected packages: traitlets-pcse
Successfully installed traitlets-pcse-5.0.0.dev0
#
# To activate this environment, use:
# > activate py3_pcse
#
# To deactivate an active environment, use:
# > deactivate
#
# * for power-users using bash, you must source
#
```

You can then activate your environment (note the addition of `(py3_pcse)` on your command prompt):

```
D:\temp\make_env>activate py3_pcse
Deactivating environment "C:\Miniconda3"...
Activating environment "C:\Miniconda3\envs\py3_pcse"...

(py3_pcse) D:\temp\make_env>
```

Installing PCSE

The easiest way to install PCSE is through the python package index (PyPI). Installing from PyPI is mostly useful if you are interested in using the functionality provided by PCSE in your own scripts, but are not interested in modifying or contributing to PCSE itself. Installing from PyPI is done using the package installer *pip* which searches the python package index for a package, downloads and installs it into your python environment (example below for PCSE 5.4):

```
(py3_pcse) D:\temp\make_env>pip install pcse

Collecting pcse
  Downloading https://files.pythonhosted.org/packages/8c/92/
↪d4444cce1c58e5a96f4d6dc9c0e042722f2136df24a2750352e7eb4ab053/PCSE-5.4.0.
↪tar.gz (791kB)
    100% |¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦| 798kB 1.6MB/s
Requirement already satisfied: numpy>=1.6.0 in c:\miniconda3\envs\py3_
↪pcse\lib\site-packages (from pcse) (1.15.1)
Requirement already satisfied: SQLAlchemy>=0.8.0 in c:\miniconda3\envs\py3_
↪pcse\lib\site-packages (from pcse) (1.2.11)
Requirement already satisfied: PyYAML>=3.11 in c:\miniconda3\envs\py3_
↪pcse\lib\site-packages (from pcse) (3.13)
Requirement already satisfied: xlrd>=0.9.3 in c:\miniconda3\envs\py3_
↪pcse\lib\site-packages (from pcse) (1.1.0)
Requirement already satisfied: xlwt>=1.0.0 in c:\miniconda3\envs\py3_
↪pcse\lib\site-packages (from pcse) (1.3.0)
Requirement already satisfied: requests>=2.0.0 in c:\miniconda3\envs\py3_
↪pcse\lib\site-packages (from pcse) (2.19.1)
Requirement already satisfied: pandas>=0.20 in c:\miniconda3\envs\py3_
↪pcse\lib\site-packages (from pcse) (0.23.4)
Requirement already satisfied: traitlets-pcse==5.0.0.dev in␣
↪c:\miniconda3\envs\py3_pcse\lib\site-packages (from pcse) (5.0.0.dev0)
```

(continues on next page)

(continued from previous page)

```
Requirement already satisfied: chardet<3.1.0,>=3.0.2 in␣
↪c:\miniconda3\envs\py3_pcse\lib\site-packages (from requests>=2.0.0->
↪pcse) (3.0.4)
Requirement already satisfied: idna<2.8,>=2.5 in c:\miniconda3\envs\py3_
↪pcse\lib\site-packages (from requests>=2.0.0->pcse) (2.7)
Requirement already satisfied: certifi>=2017.4.17 in␣
↪c:\miniconda3\envs\py3_pcse\lib\site-packages (from requests>=2.0.0->
↪pcse) (2018.8.24)
Requirement already satisfied: urllib3<1.24,>=1.21.1 in␣
↪c:\miniconda3\envs\py3_pcse\lib\site-packages (from requests>=2.0.0->
↪pcse) (1.23)
Requirement already satisfied: python-dateutil>=2.5.0 in␣
↪c:\miniconda3\envs\py3_pcse\lib\site-packages (from pandas>=0.20->pcse)␣
↪(2.7.3)
Requirement already satisfied: pytz>=2011k in c:\miniconda3\envs\py3_
↪pcse\lib\site-packages (from pandas>=0.20->pcse) (2018.5)
Requirement already satisfied: six in c:\miniconda3\envs\py3_pcse\lib\site-
↪packages (from traitlets-pcse==5.0.0.dev->pcse) (1.11.0)
Requirement already satisfied: decorator in c:\miniconda3\envs\py3_
↪pcse\lib\site-packages (from traitlets-pcse==5.0.0.dev->pcse) (4.3.0)
Requirement already satisfied: ipython-genutils in c:\miniconda3\envs\py3_
↪pcse\lib\site-packages (from traitlets-pcse==5.0.0.dev->pcse) (0.2.0)
Building wheels for collected packages: pcse
  Running setup.py bdist_wheel for pcse ... done
  Stored in directory:␣
↪C:\Users\wit015\AppData\Local\pip\Cache\wheels\2f\e6\2c\3952ff951dffea5ab2483892edcb7f
Successfully built pcse
twisted 18.7.0 requires PyHamcrest>=1.9.0, which is not installed.
mkl-random 1.0.1 requires cython, which is not installed.
mkl-fft 1.0.4 requires cython, which is not installed.
Installing collected packages: pcse
Successfully installed pcse-5.4.0
```

If you are wondering what the difference between *pip* and *conda* are than have a look here

If you want to develop with or contribute to PCSE, than you should fork the PCSE repository on GitHub and get a local copy of PCSE using *git clone*. See the help on github and for Windows/Mac users the GitHub Desktop application.

Testing PCSE

To guarantee its integrity, the PCSE package includes a limited number of internal tests that are installed automatically with PCSE. In addition, the PCSE git repository has a large number of the tests in the *test* folder which do a more thorough job in testing but will take a long time to complete (e.g. an hour or more). The internal tests present users with a quick way to ensure that the output produced by the different components matches with the expected outputs. While the full test suite is useful for developers only.

Test data for the internal tests can be found in the *pcse.tests.test_data* package as well as in an SQLite database (pcse.db). This database can be found under *.pcse* in your home folder and will be automatically created when importing PCSE for the first time. When you delete the database file manually it will be recreated next time you import PCSE.

For running the internal tests of the PCSE package we need to start python and import pcse:

```
(py3_pcse) D:\temp\make_env>python
Python 3.6.5 (default, Aug 14 2018, 19:12:50) [MSC v.1900 32 bit (Intel)]␣
↪:: Anaconda, Inc. on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import pcse
Building PCSE demo database at: C:\Users\wit015\.pcse\pcse.db ... OK
>>>
```

Next, the tests can be executed by calling the *test()* function at the top of the package:

```
.. code-block:: doscon
```

```
>>> pcse.test()
runTest (pcse.tests.test_abioticdamage.Test_FROSTOL) ... ok
runTest (pcse.tests.test_partitioning.Test_DVS_Partitioning) ...␣
↪ok
runTest (pcse.tests.test_evapotranspiration.Test_
↪PotentialEvapotranspiration) ... ok
runTest (pcse.tests.test_evapotranspiration.Test_
↪WaterLimitedEvapotranspiration1) ... ok
runTest (pcse.tests.test_evapotranspiration.Test_
↪WaterLimitedEvapotranspiration2) ... ok
runTest (pcse.tests.test_respiration.Test_
↪WOFOSTMaintenanceRespiration) ... ok
runTest (pcse.tests.test_penmanmonteith.Test_PenmanMonteith1) ...␣
↪ok
runTest (pcse.tests.test_penmanmonteith.Test_PenmanMonteith2) ...␣
↪ok
runTest (pcse.tests.test_penmanmonteith.Test_PenmanMonteith3) ...␣
↪ok
runTest (pcse.tests.test_penmanmonteith.Test_PenmanMonteith4) ...␣
↪ok
runTest (pcse.tests.test_agromanager.TestAgroManager1) ... ok
runTest (pcse.tests.test_agromanager.TestAgroManager2) ... ok
runTest (pcse.tests.test_agromanager.TestAgroManager3) ... ok
runTest (pcse.tests.test_agromanager.TestAgroManager4) ... ok
runTest (pcse.tests.test_agromanager.TestAgroManager5) ... ok
runTest (pcse.tests.test_agromanager.TestAgroManager6) ... ok
runTest (pcse.tests.test_agromanager.TestAgroManager7) ... ok
runTest (pcse.tests.test_agromanager.TestAgroManager8) ... ok
runTest (pcse.tests.test_wofost.TestWaterlimitedPotato) ... ok
runTest (pcse.tests.test_wofost.TestPotentialSunflower) ... ok
runTest (pcse.tests.test_wofost.TestWaterlimitedWinterRapeseed) ..
↪. ok
runTest (pcse.tests.test_wofost.TestPotentialSpringBarley) ... ok
runTest (pcse.tests.test_wofost.TestPotentialGrainMaize) ... ok
runTest (pcse.tests.test_wofost.TestWaterlimitedSpringBarley) ...␣
↪ok
runTest (pcse.tests.test_wofost.TestPotentialWinterRapeseed) ...␣
↪ok
runTest (pcse.tests.test_wofost.TestPotentialWinterWheat) ... ok
runTest (pcse.tests.test_wofost.TestWaterlimitedSunflower) ... ok
runTest (pcse.tests.test_wofost.TestWaterlimitedWinterWheat) ...␣
↪ok
runTest (pcse.tests.test_wofost.TestWaterlimitedGrainMaize) ... ok
runTest (pcse.tests.test_wofost.TestPotentialPotato) ... ok
```

(continues on next page)

(continued from previous page)

```
runTest (pcse.tests.test_wofost80.TestWOFOST80_Potential_
↪WinterWheat) ... ok
runTest (pcse.tests.test_wofost80.TestWOFOST80_WaterLimited_
↪WinterWheat) ... ok
```

Ran 32 tests in 39.809s

OK

If the model output matches the expected output the test will report 'OK', otherwise an error will be produced with a detailed traceback on where the problem occurred. Note that the results may deviate from the output above when tests were added or removed.

Moreover, SQLAlchemy may complain with a warning that can be safely ignored:

```
C:\Miniconda3\envs\py3_pcse\lib\site-packages\sqlalchemy\sql\sqltypes.
↪py:603: SAWarning:
Dialect sqlite+pysqlite does *not* support Decimal objects natively, and␣
↪SQLAlchemy must
convert from floating point - rounding errors and other issues may occur.␣
↪Please consider
storing Decimal numbers as strings or integers on this platform for␣
↪lossless storage.
```

3.1.3 Getting started

This guide will help you install PCSE as well as provide some examples to get you started with modelling. The examples are currently focused on applying the WOFOST and LINTUL3 crop simulation models, although other crop simulation models may become available within PCSE in the future.

3.1.4 An interactive PCSE/WOFOST session

The easiest way to demonstrate PCSE is to import WOFOST from PCSE and run it from an interactive Python session. We will be using the *start_wofost()* script that connects to a the demo database which contains meteorologic data, soil data, crop data and management data for a grid location in South-Spain.

Let's start a WOFOST object for modelling winter-wheat (crop=1) on a location in South-Spain (grid 31031) for the year 2000 under water-limited conditions for a freely draining soil (mode='wlp'):

```
>>> wofost_object = pcse.start_wofost(grid=31031, crop=1, year=2000, mode=
↪'wlp')
>>> type(wofost_object)
<class 'pcse.models.Wofost72_WLP_FD'>
```

You have just successfully initialized a PCSE/WOFOST object in the Python interpreter, which is in its initial state and waiting to do some simulation. We can now advance the model state for example with 1 day:

```
>>> wofost_object.run()
```

Advancing the crop simulation with only 1 day, is often not so useful so the number of days to simulate can be specified as well:

```
>>> wofost_object.run(days=10)
```

Retrieving information about the calculated model states or rates can be done with the *get_variable()* method on a PCSE object. For example, to retrieve the leaf area index value in the current model state you can do:

```
>>> wofost_object.get_variable('LAI')
0.28708095263317146
>>> wofost_object.run(days=25)
>>> wofost_object.get_variable('LAI')
1.5281215808337203
```

Showing that after 11 days the LAI value is 0.287. When we increase time with another 25 days, the LAI increases to 1.528. The *get_variable* method can retrieve any state or rate variable that is defined somewhere in the model. Finally, we can finish the crop season by letting it run until the model terminates because the crop reaches maturity or the harvest date:

```
>>> wofost_object.run_till_terminate()
```

Next we retrieve the simulation results at each time-step ('output') of the simulation:

```
>>> output = wofost_object.get_output()
```

We can now use the pandas package to turn the simulation output into a dataframe which is much easier to handle and can be exported to different file types. For example an excel file which should look like this `downloads/wofost_results.xls`:

```
>>> import pandas as pd
>>> df = pd.DataFrame(output)
>>> df.to_excel("wofost_results.xls")
```

Finally, we can retrieve the results at the end of the crop cycle (summary results) and have a look at these as well:

```
>>> summary_output = wofost_object.get_summary_output()
>>> msg = "Reached maturity at {DOM} with total biomass {TAGP} kg/ha "\
"and a yield of {TWSO} kg/ha."
>>> print(msg.format(**summary_output[0]))
Reached maturity at 2000-05-31 with total biomass 15261.7521735 kg/ha and␣
↪a yield of 7179.80460783 kg/ha.

>>> summary_output
[{'CTRAT': 22.457536342947606,
  'DOA': datetime.date(2000, 3, 28),
  'DOE': datetime.date(2000, 1, 1),
  'DOH': None,
  'DOM': datetime.date(2000, 5, 31),
  'DOS': None,
  'DOV': None,
  'DVS': 2.01745939841335,
  'LAIMAX': 6.132711275237731,
  'RD': 60.0,
  'TAGP': 15261.752173534584,
  'TWLV': 3029.3693107257263,
  'TWRT': 1546.990661062695,
```

(continues on next page)

(continued from previous page)

```
'TWSO': 7179.8046078262705,
'TWST': 5052.578254982587}]
```

Running PCSE/WOFOST with custom input data

For running PCSE/WOFOST (and PCSE models in general) with your own data sources you need three different types of inputs:

1. Model parameters that parameterize the different model components. These parameters usually consist of a set of crop parameters (or multiple sets in case of crop rotations), a set of soil parameters and a set of site parameters. The latter provide ancillary parameters that are specific for a location.
2. Driving variables represented by weather data which can be derived from various sources.
3. Agromanagement actions which specify the farm activities that will take place on the field that is simulated by PCSE.

For the second example we will run a simulation for sugar beet in Wageningen (Netherlands) and we will read the input data step by step from several different sources instead of using the pre-configured *start_wofost()* script. For the example we will assume that data files are in the directory *D:\userdata\pcse_examples* and all the parameter files needed can be found by unpacking this zip file `downloads/quickstart_part2.zip`.

First we will import the necessary modules and define the data directory:

```
>>> import os
>>> import pcse
>>> import matplotlib.pyplot as plt
>>> data_dir = r'D:\userdata\pcse_examples'
```

Crop parameters

The crop parameters consist of parameter names and the corresponding parameter values that are needed to parameterize the components of the crop simulation model. These are crop-specific values regarding phenology, assimilation, respiration, biomass partitioning, etc. The parameter file for sugar beet is taken from the crop files in the WOFOST Control Centre.

The crop parameters for many models in Wageningen are often provided in the CABO format that could be read with the TTUTIL FORTRAN library. PCSE tries to be backward compatible as much as possible and provides the *CABOFileReader* for reading parameter files in CABO format. the CABOFileReader returns a dictionary with the parameter name/value pairs:

```
>>> from pcse.fileinput import CABOFileReader
>>> cropfile = os.path.join(data_dir, 'sug0601.crop')
>>> cropdata = CABOFileReader(cropfile)
>>> print(cropdata)
```

Printing the cropdata dictionary gives you a listing of the header and all parameters and their values.

Soil parameters

The soildata dictionary provides the parameter name/value pairs related to the soil type and soil physical properties. The number of parameters is variable depending on the soil water balance type that is used for the simulation. For this example, we will use the water balance for freely draining soils and use the soil file for medium fine sand: *ec3.soil*. This file is also taken from the soil files in the WOFOST Control Centre

```
>>> soilfile = os.path.join(data_dir, 'ec3.soil')
>>> soildata = CABOFileReader(soilfile)
```

Site parameters

The site parameters provide ancillary parameters that are not related to the crop or the soil. Examples are the initial conditions of the water balance such as the initial soil moisture content (WAV) and the initial and maximum surface storage (SSI, SSMAX). Also the atmospheric CO2 concentration is a typical site parameter. For the moment, we can define these parameters directly on the Python commandline as a simple python dictionary. However, it is more convenient to use the WOFOST71SiteDataProvider that documents the site parameters and provides sensible defaults:

```
>>> from pcse.util import WOFOST71SiteDataProvider
>>> sitedata = WOFOST71SiteDataProvider(WAV=100, CO2=360)
>>> print(sitedata)
{'SMLIM': 0.4, 'NOTINF': 0, 'CO2': 360.0, 'SSI': 0.0, 'SSMAX': 0.0, 'IFUNRN
↪': 0, 'WAV': 100.0}
```

Finally, we need to pack the different sets of parameters into one variable using the *ParameterProvider*. This is needed because PCSE expects one variable that contains all parameter values. Using this approach has the additional advantage that parameters value can be easily overridden in case of running multiple simulations with slightly different parameter values:

```
>>> from pcse.base import ParameterProvider
>>> parameters = ParameterProvider(cropdata=cropdata, soildata=soildata,␣
↪sitedata=sitedata)
```

AgroManagement

The agromanagement inputs provide the start date of the agricultural campaign, the start_date/start_type of the crop simulation, the end_date/end_type of the crop simulation and the maximum duration of the crop simulation. The latter is included to avoid unrealistically long simulations for example as a results of a too high temperature sum requirement.

The agromanagement inputs are defined with a special syntax called YAML which allows to easily create more complex structures which is needed for defining the agromanagement. The agromanagement file for sugar beet in Wageningen *sugarbeet_calendar.agro* can be read with the *YAMLAgroManagementReader*:

```
>>> from pcse.fileinput import YAMLAgroManagementReader
>>> agromanagement_file = os.path.join(data_dir, 'sugarbeet_calendar.agro')
>>> agromanagement = YAMLAgroManagementReader(agromanagement_file)
```

(continues on next page)

(continued from previous page)

```
>>> print(agromanagement)
 !!python/object/new:pcse.fileinput.yaml_agro_loader.
↪YAMLAgroManagementReader
 listitems:
 - 2000-01-01:
     CropCalendar:
       crop_name: sugarbeet
       variety_name: sugar_beet_601
       crop_start_date: 2000-04-05
       crop_start_type: emergence
       crop_end_date: 2000-10-20
       crop_end_type: harvest
       max_duration: 300
     StateEvents: null
     TimedEvents: null
```

Daily weather observations

Daily weather variables are needed for running the simulation. There are several data providers in PCSE for reading weather data, see the section on *weather data providers* to get an overview.

For this example we will use the weather data from the NASA Power database which provides global weather data with a spatial resolution of 0.5 degree (~50 km). We will retrieve the data from the Power database for the location of Wageningen. Note that it can take around 30 seconds to retrieve the weather data from the NASA Power server the first time:

```
>>> from pcse.db import NASAPowerWeatherDataProvider
>>> wdp = NASAPowerWeatherDataProvider(latitude=52, longitude=5)
>>> print(wdp)
Weather data provided by: NASAPowerWeatherDataProvider
--------Description---------
NASA/POWER SRB/FLASHFlux/MERRA2/GEOS 5.12.4 (FP-IT) 0.5 x 0.5 Degree Daily␣
↪Averaged Data
----Site characteristics----
Elevation:    4.7
Latitude:  52.000
Longitude:  5.000
Data available for 1983-07-01 - 2018-09-16
Number of missing days: 8
```

Importing, initializing and running a PCSE model

Internally, PCSE uses a simulation *engine* to run a crop simulation. This engine takes a configuration file that specifies the components for the crop, the soil and the agromanagement that need to be used for the simulation. So any PCSE model can be started by importing the *engine* and initializing it with a given configuration file and the corresponding parameters, weather data and agromanagement.

However, as many users of PCSE only need a particular configuration (for example the WOFOST model for potential production), preconfigured Engines are provided in *pcse.models*. For the sugarbeet example we will import the WOFOST model for water-limited simulation under freely draining soil conditions:

```
>>> from pcse.models import Wofost71_WLP_FD
>>> wofsim = Wofost71_WLP_FD(parameters, wdp, agromanagement)
```

We can then run the simulation and show some final results such as the anthesis and harvest dates (DOA, DOH), total biomass (TAGP) and maximum LAI (LAIMAX). Next, we retrieve the time series of daily simulation output using the *get_output()* method on the WOFOST object:

```
>>> wofsim.run_till_terminate()
>>> output = wofsim.get_output()
>>> len(output)
294
```

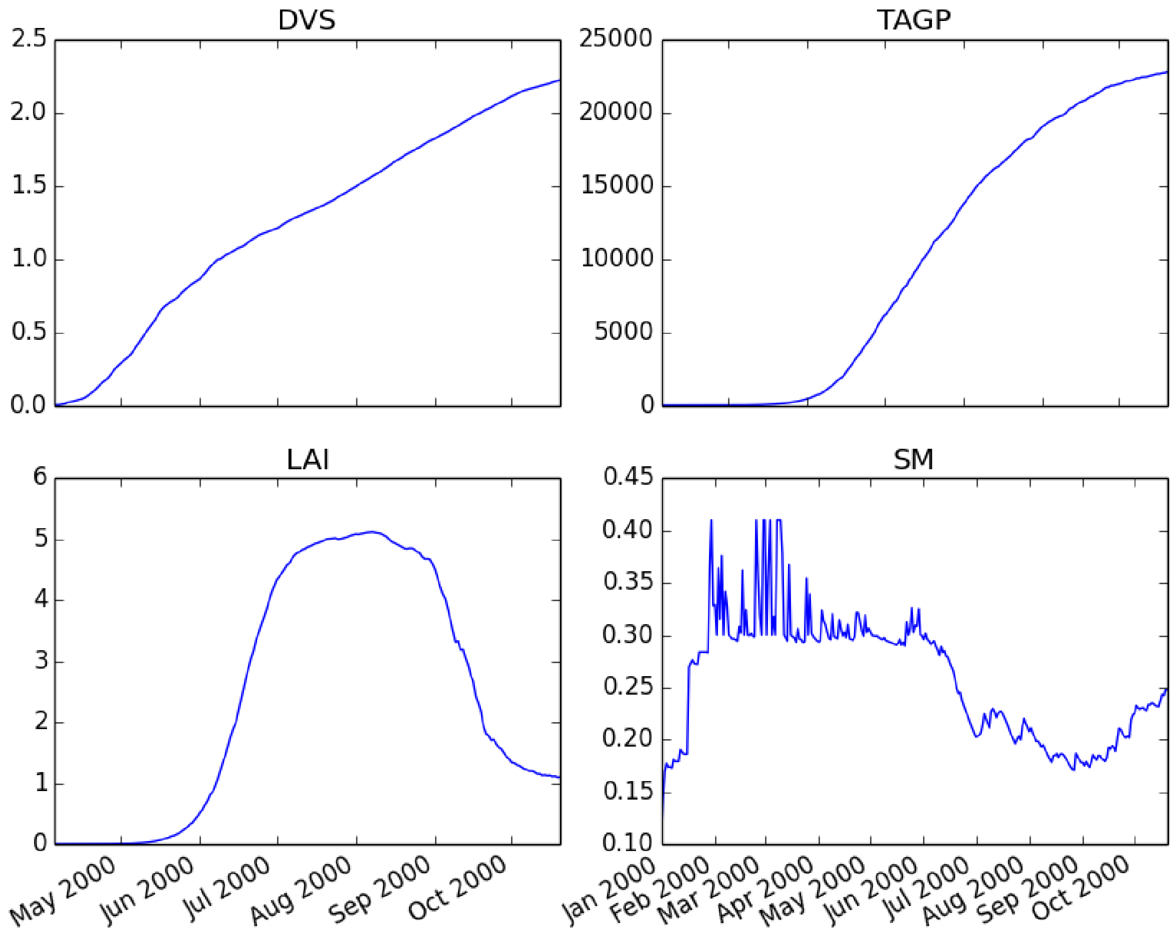
As the output is returned as a list of dictionaries, we need to unpack these variables from the list of output:

```
>>> varnames = ["day", "DVS", "TAGP", "LAI", "SM"]
>>> tmp = {}
>>> for var in varnames:
>>>     tmp[var] = [t[var] for t in output]
```

Finally, we can generate some figures of WOFOST variables such as the development (DVS), total biomass (TAGP), leaf area index (LAI) and root-zone soil moisture (SM) using the MatPlotLib plotting package:

```
>>> day = tmp.pop("day")
>>> fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(10,8))
>>> for var, ax in zip(["DVS", "TAGP", "LAI", "SM"], axes.flatten()):
>>>     ax.plot_date(day, tmp[var], 'b-')
>>>     ax.set_title(var)
>>> fig.autofmt_xdate()
>>> fig.savefig('sugarbeet.png')
```

This should generate a figure of the simulation results as shown below. The complete Python script for this examples can be downloaded here `downloads/quickstart_demo2.py`



Running a simulation with PCSE/LINTUL3

The LINTUL model (Light INTerception and UtiLisation) is a simple generic crop model, which simulates dry matter production as the result of light interception and utilization with a constant light use efficiency. In PCSE the LINTUL family of models has been implemented including the LINTUL3 model which is used for simulation of crop production under water-limited and nitrogen-limited conditions.

For the third example, we will use LINTUL3 for simulating spring-wheat in the Netherlands under water-limited and nitrogen-limited conditions. We will again assume that data files are in the directory *D:\userdata\pcse_examples* and all the parameter files needed can be found by unpacking this zip file `downloads/quickstart_part3.zip`. Note that this guide is also available as an IPython notebook: `downloads/running_LINTUL3.ipynb`.

First we will import the necessary modules and define the data directory. We also assume that you have the matplotlib, **'pandas'**_ and PyYAML packages installed on your system.:

```
>>> import os
>>> import pcse
>>> import matplotlib.pyplot as plt
>>> import pandas as pd
>>> import yaml
>>> data_dir = r'D:\userdata\pcse_examples'
```

Similar to the previous example, for running the PCSE/LINTUL3 model we need to define the tree types of inputs (parameters, weather data and agromanagement).

Reading model parameters

Model parameters can be easily read from the input files using the *PCSEFileReader* as we have seen in the previous example:

```
>>> from pcse.fileinput import PCSEFileReader
>>> crop = PCSEFileReader(os.path.join(data_dir, "lintul3_springwheat.crop
↪"))
>>> soil = PCSEFileReader(os.path.join(data_dir, "lintul3_springwheat.soil
↪"))
>>> site = PCSEFileReader(os.path.join(data_dir, "lintul3_springwheat.site
↪"))
```

However, PCSE models expect a single set of parameters and therefore they need to be combined using the *ParameterProvider*:

```
>>> from pcse.base import ParameterProvider
>>> parameterprovider = ParameterProvider(soildata=soil, cropdata=crop,␣
↪sitedata=site)
```

Reading weather data

For reading weather data we will use the ExcelWeatherDataProvider. This WeatherDataProvider uses nearly the same file format as is used for the CABO weather files but stores its data in an MicroSoft Excel file which makes the weather files easier to create and update:

```
>>> from pcse.fileinput import ExcelWeatherDataProvider
>>> weatherdataprovider = ExcelWeatherDataProvider(os.path.join(data_dir,
↪"nl1.xlsx"))
>>> print(weatherdataprovider)
Weather data provided by: ExcelWeatherDataProvider
--------Description---------
Weather data for:
Country: Netherlands
Station: Wageningen, Location Haarweg
Description: Observed data from Station Haarweg in Wageningen
Source: Meteorology and Air Quality Group, Wageningen University
Contact: Peter Uithol
----Site characteristics----
Elevation:    7.0
Latitude:  51.970
Longitude:  5.670
Data available for 2004-01-02 - 2008-12-31
Number of missing days: 32
```

Defining agromanagement

Defining agromanagement needs a bit more explanation because agromanagement is a relatively complex piece of PCSE. The agromanagement definition for PCSE is written in a format called YAML and for the current example looks like this:

```
Version: 1.0.0
AgroManagement:
- 2006-01-01:
    CropCalendar:
        crop_name: wheat
        variety_name: spring-wheat
        crop_start_date: 2006-03-31
        crop_start_type: emergence
        crop_end_date: 2006-08-20
        crop_end_type: earliest
        max_duration: 300
    TimedEvents:
    -   event_signal: apply_n
        name:  Nitrogen application table
        comment: All nitrogen amounts in g N m-2
        events_table:
        - 2006-04-10: {amount: 10, recovery: 0.7}
        - 2006-05-05: {amount:  5, recovery: 0.7}
    StateEvents: null
```

The agromanagement definition starts with *Version:* indicating the version number of the agromanagement file while the actual definition starts after the label *AgroManagement:*. Next a date must be provided which sets the start date of the campaign (and the start date of the simulation). Each campaign is defined by zero or one CropCalendars and zero or more TimedEvents and/or StateEvents. The CropCalendar defines the crop name, variety_name, date of sowing, date of harvesting, etc. while the Timed/StateEvents define actions that are either connected to a date or to a model state.

In the current example, the campaign starts on 2006-01-01, there is a crop calendar for spring-wheat starting on 2006-03-31 with a harvest date of 2006-08-20 or earlier if the crop reaches maturity before this date. Next there are timed events defined for applying N fertilizer at 2006-04-10 and 2006-05-05. The current example has no state events. For a thorough description of all possibilities see the section on AgroManagement in the Reference Guide (Chapter 3).

Loading the agromanagement definition must by done with the YAMLAgroManagementReader:

```
>>> from pcse.fileinput import YAMLAgroManagementReader
>>> agromanagement = YAMLAgroManagementReader(os.path.join(data_dir,
↪"lintul3_springwheat.amgt"))
>>> print(agromanagement)
!!python/object/new:pcse.fileinput.yaml_agro_loader.
↪YAMLAgroManagementReader
listitems:
- 2006-01-01:
    CropCalendar:
      crop_end_date: 2006-10-20
      crop_end_type: earliest
      crop_name: wheat
      variety_name: spring-wheat
      crop_start_date: 2006-03-31
      crop_start_type: emergence
      max_duration: 300
    StateEvents: null
    TimedEvents:
    - comment: All nitrogen amounts in g N m-2
      event_signal: apply_n
      events_table:
```

(continues on next page)

(continued from previous page)

```
    - 2006-04-10:
        amount: 10
        recovery: 0.7
    - 2006-05-05:
        amount: 5
        recovery: 0.7
    name: Nitrogen application table
```

Starting and running the LINTUL3 model

We have now all parameters, weather data and agromanagement information available to start the LINTUL3 model:

```
>>> from pcse.models import LINTUL3
>>> lintul3 = LINTUL3(parameterprovider, weatherdataprovider,␣
↪agromanagement)
>>> lintul3.run_till_terminate()
```

Next, we can easily get the output from the model using the get_output() method and turn it into a pandas DataFrame:

```
>>> output = lintul3.get_output()
>>> df = pd.DataFrame(output).set_index("day")
>>> df.tail()
                 DVS       LAI     NUPTT       TAGBM     TGROWTH  TIRRIG  \
day
2006-07-28  1.931748  0.384372  4.705356  560.213626  626.053663       0
2006-07-29  1.953592  0.368403  4.705356  560.213626  626.053663       0
2006-07-30  1.974029  0.353715  4.705356  560.213626  626.053663       0
2006-07-31  1.995291  0.339133  4.705356  560.213626  626.053663       0
2006-08-01  2.014272  0.326169  4.705356  560.213626  626.053663       0

               TNSOIL  TRAIN  TRAN  TRANRF  TRUNOF      TTRAN        WC  \
day
2006-07-28  11.794644  375.4     0       0       0  71.142104  0.198576
2006-07-29  11.794644  376.3     0       0       0  71.142104  0.197346
2006-07-30  11.794644  376.3     0       0       0  71.142104  0.196293
2006-07-31  11.794644  381.6     0       0       0  71.142104  0.198484
2006-08-01  11.794644  381.7     0       0       0  71.142104  0.197384

                 WLVD       WLVG        WRT         WSO         WST
day
2006-07-28  88.548865  17.687197  16.649830  184.991591  268.985974
2006-07-29  89.284828  16.951234  16.150335  184.991591  268.985974
2006-07-30  89.962276  16.273785  15.665825  184.991591  268.985974
2006-07-31  90.635216  15.600845  15.195850  184.991591  268.985974
2006-08-01  91.233828  15.002234  14.739974  184.991591  268.985974
```

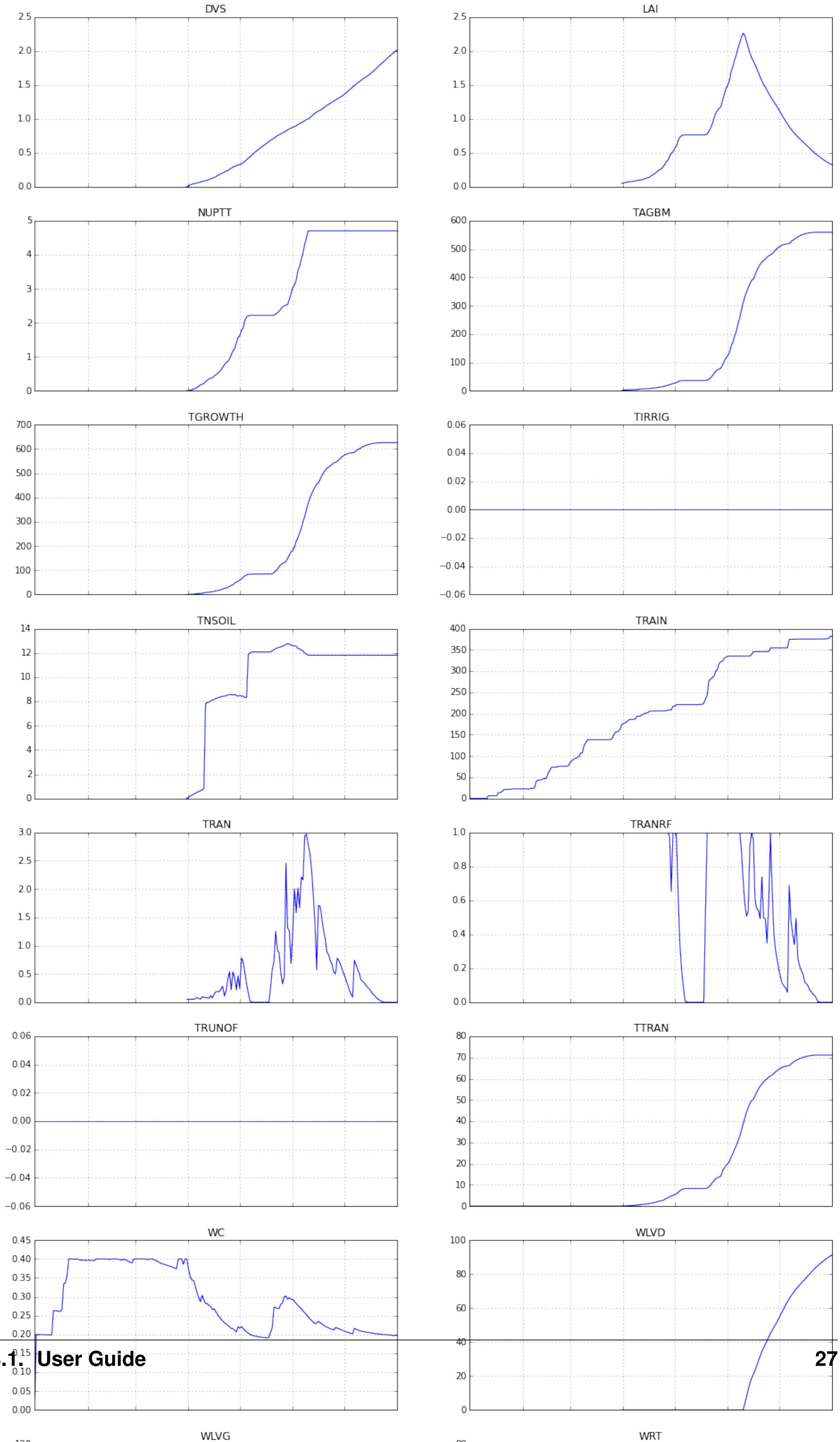
Finally, we can visualize the results from the pandas DataFrame with a few commands if your environment supports plotting:

```
>>> fig, axes = plt.subplots(nrows=9, ncols=2, figsize=(16,40))
>>> for key, axis in zip(df.columns, axes.flatten()):
```

(continues on next page)

(continued from previous page)

```
>>>     df[key].plot(ax=axis, title=key)
>>> fig.autofmt_xdate()
>>> fig.savefig(os.path.join(data_dir, "lintul3_springwheat.png"))
```


DVS
LAI
NUPTT
TAGBM
TGROWTH
TIRRIG
TNSOIL
TRAIN
TRAN
TRANRF
TRUNOF
TTRAN
WC
WLVD
WLVG
WRT

3.1.5 Advanced topics

Many more examples plus demonstrations of advanced topics are available as Jupyter notebooks at https://github.com/ajwdewit/pcse_notebooks

CHAPTER 4

Reference guide

4.1 Reference Guide

4.1.1 An overview of PCSE

The Python Crop Simulation Environment builds on the heritage provided by the earlier approaches developed in Wageningen, notably the Fortran Simulation Environment. The FSE manual (van Kraalingen, 1995) provides a very good overview on the principles of Euler integration and its application to crop simulation models. Therefore, we will not discuss this in detail here.

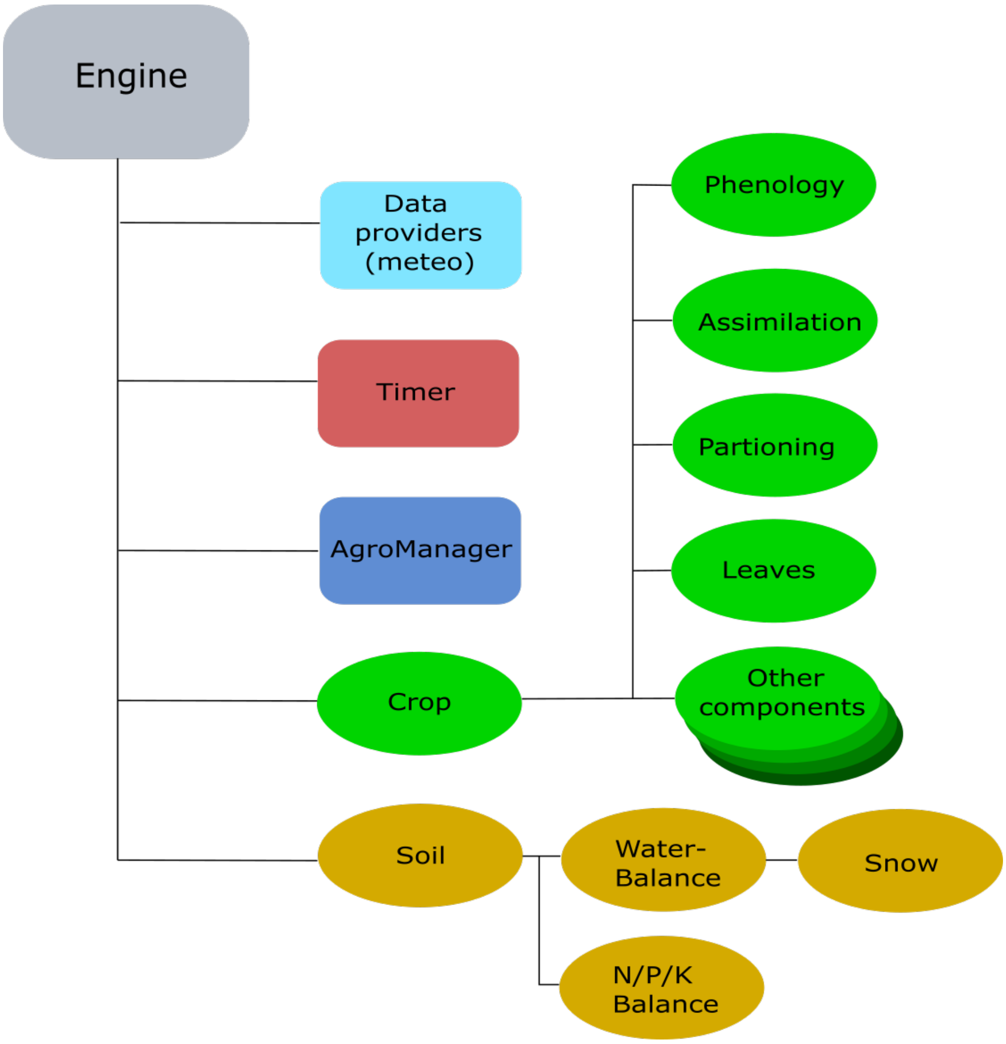
Nevertheless, PCSE also tries to improve on these approaches by separating the simulation logic into a number of distinct components that play a role in the implementation of (crop) simulation models:

1. The dynamic part of the simulation is taken care of by a dedicated simulation *Engine* which handles the initialization, the ordering of rate/state updates for the soil and plant modules as well as keeping track of time, retrieving weather data and calling the agromanager module.
2. Solving the differential equations for soil/plant system and updating the model state is deferred to SimulationObjects that implement (bio)physical processes such as phenological development or CO_2 assimilation.
3. An AgroManager module is included which takes care of signalling agricultural management actions such as sowing, harvesting, irrigation, etc.
4. Communication between PCSE components is implemented by either exporting variables into a shared state object or by implementing signals that can be broadcasted and received by any PCSE object.
5. Several tools are available for providing weather data and reading parameter values from files or databases.

Next, an overview of the different components in PCSE will be provided.

4.1.2 The Engine

The PCSE Engine provides the environment where the simulation takes place. The engine takes care of reading the model configuration, initializing model components, driving the simulation forward by calling the SimulationObjects, calling the agromanagement unit, keeping track of time, providing the weather data needed and storing the model variables during the simulation for later output. The Engine itself is generic and can be used for any model that is defined in PCSE. The overall structure of the engine can be found in the figure below which shows the different elements that are called by the Engine.



Continuous simulation in PCSE

To implement continuous simulation, the engine uses the same approach as FSE: Euler integration with a fixed time step of one day. The following figure shows the principle of continuous simulation and the execution order of various steps.

The steps in the process cycle that are shown in the figure above are implemented in the simulation *Engine* which is completely separated from the model logic itself. Moreover, it demonstrates that before the simulation can start the engine has to be initialized which involves several steps:

1. The model configuration must be loaded;
2. The AgroManager module must be initialized and called to determine the first and last of the simulation sequence;

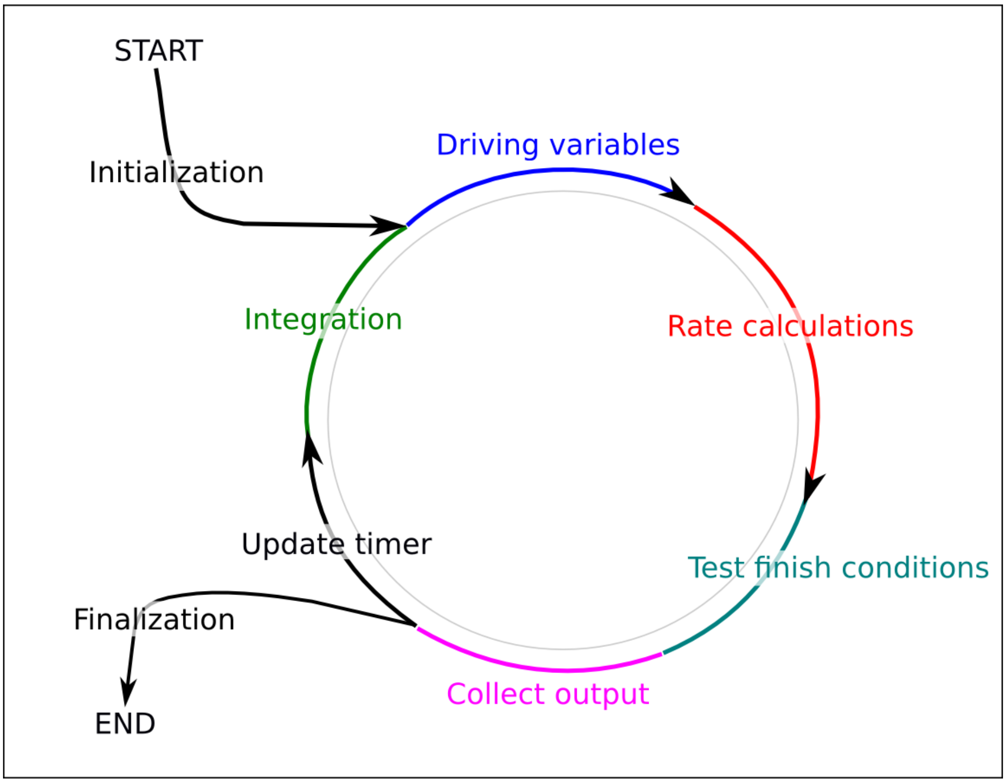


Fig. 1: Order of calculations for continuous simulation using Euler integration (after Van Kraalingen, 1995).

3. The timer must be initialized with the first and last day of the simulation sequence;
4. The soil component specified in the model configuration must be initialized.
5. The weather variables must be retrieved for the starting day;
6. The AgroManager must be called to trigger any management events that are scheduled for the starting day.
7. The initial rates of change based on the initial states and driving variables must be calculated;
8. Finally, output can be collected to save the initial states and rates of the simulation.

The next cycle in the simulation will now start with an update of the timer to the next time step (e.g. day). Next, the rates of change of the previous day will be integrated onto the state variables and the driving variables for the current day will be retrieved. Finally, the rates of change will be recalculated based on the new driving variables and updated model states and so forth.

The simulation loop will terminate when some finish condition has been reached. Usually, the *AgroManager* module will encounter the end of the agricultural campaign and will broadcast a terminate signal that terminates the entire simulation.

Input needed by the Engine

To start the Engine four inputs are needed:

1. A weather data provider that provides the Engine with the daily values of weather variables. See the section on *Weather data providers* for an overview of the different options for providing weather data.

2. A set of parameters that is needed to parameterize the SimulationObjects that simulate the soil and crop processes. Model parameters can be retrieved from different sources like files or databases. PCSE uses three sets of model parameters: crop parameters, soil parameters and site parameters. The latter present an ancillary set of parameters that are not related to the soil or the crop. The atmospheric CO2 concentration is a typical example of a site parameter. Despite having three sets of parameters, all parameters are encapsulated using a *ParameterProvider* that provides a uniform interface to access the different parameter sets. See the section on *Data providers for parameter values* for an overview.
3. Agromanagement information that is needed to schedule agromanagement actions that are taking place during the simulation. See the sections on *The AgroManager* and *Data providers for agromanagement* for a detailed overview.
4. A configuration file that tells the Engine the details of the simulation such as the components to use for the simulation of the crop, the soil and the agromanagement. Moreover, the results that should be stored as final and intermediate outputs and some other details.

Engine configuration files

The engine needs a configuration file that specifies which components should be used for simulation and additional information. This is most easily explained by an example such as the configuration file for the WOFOST 7.2 model for potential crop production:

```
# -*- coding: utf-8 -*-
# Copyright (c) 2004-2021 Wageningen Environmental Research
# Allard de Wit (allard.dewit@wur.nl), August 2021
"""PCSE configuration file for WOFOST 7.2 Potential Production simulation

This configuration file defines the soil and crop components that
should be used for potential production simulation.
"""

from pcse.soil.classic_waterbalance import WaterbalancePP
from pcse.crop.wofost7 import Wofost
from pcse.agromanager import AgroManager

# Module to be used for water balance
SOIL = WaterbalancePP

# Module to be used for the crop simulation itself
CROP = Wofost

# Module to use for AgroManagement actions
AGROMANAGEMENT = AgroManager

# variables to save at OUTPUT signals
# Set to an empty list if you do not want any OUTPUT
OUTPUT_VARS = ["DVS","LAI","TAGP", "TWSO", "TWLV", "TWST",
               "TWRT", "TRA", "RD", "SM", "WWLOW"]
# interval for OUTPUT signals, either "daily"|"dekadal"|"monthly"|"weekly"
# For daily output you change the number of days between successive
# outputs using OUTPUT_INTERVAL_DAYS. For dekadal and monthly
# output this is ignored.
OUTPUT_INTERVAL = "daily"
OUTPUT_INTERVAL_DAYS = 1
```

(continues on next page)

(continued from previous page)

```
# Weekday: Monday is 0 and Sunday is 6
OUTPUT_WEEKDAY = 0

# Summary variables to save at CROP_FINISH signals
# Set to an empty list if you do not want any SUMMARY_OUTPUT
SUMMARY_OUTPUT_VARS = ["DVS","LAIMAX","TAGP", "TWSO", "TWLV", "TWST",
                       "TWRT", "CTRAT", "RD", "DOS", "DOE", "DOA",
                       "DOM", "DOH", "DOV", "CEVST"]

# Summary variables to save at TERMINATE signals
# Set to an empty list if you do not want any TERMINAL_OUTPUT
TERMINAL_OUTPUT_VARS = []
```

As you can see, the configuration file is written in plain python code. First of all, it defines the placeholders *SOIL*, *CROP* and *AGROMANAGEMENT* that define the components that should be used for the simulation of these processes. These placeholders simply point to the modules that were imported at the start of the configuration file.

Note: Modules in configuration files must be imported using fully qualified names and relative imports cannot be used.

The second part is for defining the variables (*OUTPUT_VARS*) that should be stored during the model run (during OUTPUT signals) and the details of the regular output interval. Next, summary output *SUMMARY_OUTPUT_VARS* can be defined that will be generated at the end of each crop cycle. Finally, output can be collected at the end of the entire simulation (*TERMINAL_OUTPUT_VARS*).

Note: Model configuration files for models that are included in the PCSE package reside in the 'conf/' folder inside the package. When the Engine is started with the name of a configuration file, it searches this folder to locate the file. This implies that if you want the start the Engine with your own (modified) configuration file, you *must* specify it as an absolute path otherwise the Engine will not find it.

The relationship between models and the engine

Models are treated together with the Engine, because models are simply pre-configured Engines. Any model can be started by starting the Engine with the appropriate configuration file. The only difference is that models can have methods that deal with specific characteristics of a model. This kind of functionality cannot be implemented in the Engine because the model details are not known beforehand.

4.1.3 SimulationObjects

PCSE uses SimulationObjects to group parts of the crop simulation model that form a logical entity into separate program code sections. In this way the crop simulation model is grouped into sections that implement certain biophysical processes such as phenology, assimilation, respiration, etc. Simulation objects can be grouped to form components that perform the simulation of an entire crop or a soil profile.

This approach has several advantages:

1. Model code with a certain purpose is grouped together, making it easier to read, understand and maintain.

2. A SimulationObject contains only parameters, rate and state variables that are needed. In contrast, with monolythic code it is often unclear (at first glance at least) what biophysical process they belong to.
3. Isolation of process implementations creates less dependencies, but more importantly, dependencies are evident from the code which makes it easier to modify individual SimulationObjects.
4. SimulationObjects can be tested individually by comparing output vs the expected output (e.g. unit testing).
5. SimulationObjects can be exchanged for other objects with the same purpose but a different biophysical approach. For example, the WOFOST assimilation approach could be easily replaced by a more simple Light Use Efficiency or Water Use Efficiency approach, by replacing the SimulationObject that handles the CO_2 assimilation.

Characteristics of SimulationObjects

Each SimulationObject is defined in the same way and has a couple of standard sections and methods which facilitates understanding and readability. Each SimulationObject has parameters to define the mathematical relationships, it has state variables to define the state of the system and it has rate variables that describe the rate of change from one time step to the next. Moreover, a SimulationObject may contain other SimulationObjects that together form a logical structure. Finally, the SimulationObject must implement separate code sections for initialization, rate calculation and integration of the rates of change. A finalization step which is called at the end of the simulation can be added optionally.

The skeleton of a SimulationObject looks like this:

```
class CropProcess(SimulationObject):

    class Parameters(ParamTemplate):
        PAR1 = Float()
        # more parameters defined here

    class StateVariables(StatesTemplate):
        STATE1 = Float()
        # more state variables defined here

    class RateVariables(RatesTemplate):
        RATE1 = Float()
        # more rate variables defined here

    def initialize(self, day, kiosk, parametervalues):
        """Initializes the SimulationObject with given parametervalues."""
        self.params = self.Parameters(parametervalues)
        self.rates = self.RateVariables(kiosk)
        self.states = self.StateVariables(kiosk, STATE1=0., publish=[
↪"STATE1"])

    @prepare_rates
    def calc_rates(self, day, drv):
        """Calculate the rates of change given the current states and␣
↪driving
        variables (drv)."""

        # simple example of rate calculation using rainfall (drv.RAIN)
```

(continues on next page)

(continued from previous page)

```
        self.rates.RATE1 = self.params.PAR1 * drv.RAIN

    @prepare_states
    def integrate(self, day, delt):
        """Integrate the rates of change on the current state variables
        multiplied by the time-step
        """
        self.states.STATE1 += self.rates.RATE1 * delt

    @prepare_states
    def finalize(self, day):
        """do some final calculations when the simulation is finishing."""
```

The strict separation of program logic was copied from the Fortran Simulation Environment (FSE, Rappoldt and Van Kraalingen 1996 and Van Kraalingen 1995) and is critical to ensure that the simulation results are correct. The different calculations types (integration, driving variables and rate calculations) should be strictly separated. In other words, first all states should be updated, subsequently all driving variables should be calculated, after which all rates of change should be calculated. If this rule is not applied rigorously, some rates may pertain to states at the current time whereas others will pertain to states from the previous time step. Compared to the FSE system and the FORTRAN implementation of WOFOST, the *initialize()*, *calc_rates()*, *integrate()* and *finalize()* sections match with the *ITASK* numbers 1, 2, 3, 4.

A complicating factor that arises when using modular code is how to arrange the communication between SimulationObjects. For example, the *evapotranspiration* SimulationObject will need information about the leaf area index from the *leaf_dynamics* SimulationObject to calculate the crop transpiration values. In PCSE the communication between SimulationObjects is taken care of by the so-called *VariableKiosk*. The metaphore kiosk is used because the SimulationObjects publish their rate and/or state variables (or a subset) into the kiosk, other SimulationObjects can subsequently request the variable value from the kiosk without any knowledge about the SimulationObject that published it. Therefore, the VariableKiosk is shared by all SimulationObjects and must be provided when SimulationObjects initialize.

See the section on *Exchanging data between model components* for a detailed description of the variable kiosk and other ways to communicate between model components.

Simulation Parameters

Usually SimulationObjects have one or more parameters which should be defined as a subclass of the *ParamTemplate* class. Although parameters can be specified as part of the SimulationObject definition directly, subclassing them from *ParamTemplate* has a few advantages. First of all, parameters must be initialized and a missing parameter will lead to an exception being raised with a clear message. Secondly, parameters are initialized as read-only attributes which cannot be changed during the simulation. So occasionally overwriting a parameter value is impossible this way.

The model parameters are initialized by the calling the Parameters class definition and providing a dictionary with key/value pairs to define the parameters.

State/Rate variables

The definitions for state and rate variables share many properties. Definitions of rate and state variables should be defined as attributes of a class that inherit from *RatesTemplate* and *StatesTemplate* respectively.

Names of rate and state variables that are defined this way **must** be unique across all model components and a duplicate variable name somewhere across the model composition will lead to an exception.

Both class instances need the VariableKiosk as its first input parameter which is needed to register the variables defined. Moreover, variables can be published with the *publish* keyword as is done in the example above for *STATE1*. Publishing a variable means that it will be available in the VariableKiosk and can be retrieved by other components based on the name of the variables. The main difference between a rates and a states class is that the states class requires you to provide the initial value of the state as a keyword parameter in the call. Failing to provide the initial value will lead to an exception being raised.

Instances of objects containing rate and state variables are read-only by default. In order to change the value of a rate or state, the instance must be unlocked. For this purpose the decorators *@prepare_rates* and *@prepare_states* are being placed in front of the calls to *calc_rates()* and *integrate()* which take care of unlocking and locking the states and rates instances. Using this approach rate variables can only be changed during the call where the rates are calculated, states variables are read-only at that stage. Similarly, state variables can only be changed during the state update while the rates of change are locked. This mechanism ensures that rate/state updates are carried out in the correct order.

Finally, instances of RatesTemplate have one additional method, called *zerofy()* while instances of StatesTemplate have one additional method called *touch()*. Calling *zerofy()* is normally done by the Engine and explicitly sets all rates of change to zero. Calling *touch()* on a states object is only useful when the states variables do not need to be updated, but you do want to be sure that any published state variables will remain available in the VariableKiosk.

4.1.4 The AgroManager

Agromanagement is an intricate part of PCSE which is needed for simulating the processes that are happening on agriculture fields. In order for crops to grow, farmers must first plow the fields and sow the crop. Next, they have to do proper management including irrigation, weeding, nutrient application, pest control and finally harvesting. All these actions have to be scheduled at specific dates, connected to certain crop stages or in dependence of soil and weather conditions. Moreover specific parameters such as the amount of irrigation or nutrients must be provided as well.

In previous versions of WOFOST, the options for agromanagement were limited to sowing and harvesting. On the one had this was because agromanagement was often assumed to be optimal and thus there was little need for detailed agromanagement. On the other hand, implementing agromanagement is relatively complex because agromanagement consists of events that are happening once rather than continuously. As such, it does not fit well in the traditional simulation cycle, see *Continuous simulation in PCSE*.

Also from a technical point of view implementing such events through the traditional function calls for rate calculation and state updates is not attractive. For example, for indicating a nutrient application event several additional parameters would have to be passed: e.g. the type of nutrient, the amount and its efficiency. This has several drawbacks, first of all, only a limited number of SimulationObjects will actually do something with this information while for all other objects, the information is of no use. Second, nutrient application will usually happen only once or twice in the growing cycle. So for a 200-day growing cycle there will be 198 days where the parameters do not carry any information. Nevertheless, they would still be present in the function call, thereby decreasing the computational efficiency and the readability of the code. Therefore, PCSE uses a very different approach for agromanagement events which is based on signals (see *Broadcasting signals*).

Defining agromanagement in PCSE

Defining the agromanagement in PCSE is not very complicated and first starts with defining a sequence of campaigns. Campaigns start on a prescribed calendar date and finalize when the next campaign starts. Each campaign is characterized by zero or one crop calendar, zero or more timed events and zero or more state events. The crop calendar specifies the timing of the crop (sowing, harvesting) while the timed and state events can be used to specify management actions that are either dependent on time (a specific date) or a certain model state variable such as crop development stage. Crop calendars and event definitions are only valid for the campaign in which they are defined.

The data format used for defining the agromanagement in PCSE is called YAML. YAML is a versatile format optimized for readability by humans while still having the power of XML. However, the agromanagement definition in PCSE is by no means tied to YAML and can be read from a database as well.

The structure of the data needed as input for the AgroManager is most easily understood with an example (below). The example definition consists of three campaigns, the first starting on 1999-08-01, the second starting on 2000-09-01 and the last campaign starting on 2001-03-01. The first campaign consists of a crop calendar for winter-wheat starting with sowing at the given crop_start_date. During the campaign there are timed events for irrigation at 2000-05-25 and 2000-06-30. Moreover, there are state events for fertilizer application (event_signal: apply_npk) given by development stage (DVS) at DVS 0.3, 0.6 and 1.12.

The second campaign has no crop calendar, timed events or state events. This means that this is a period of bare soil with only the water balance running. The third campaign is for fodder maize sown at 2001-04-15 with two series of timed events (one for irrigation and one for N/P/K application) and no state events. The end date of the simulation in this case will be 2001-11-01 (2001-04-15 + 200 days).

An example of an agromanagement definition file:

```
AgroManagement:
- 1999-08-01:
    CropCalendar:
        crop_name: wheat
        variety_name: winter-wheat
        crop_start_date: 1999-09-15
        crop_start_type: sowing
        crop_end_date:
        crop_end_type: maturity
        max_duration: 300
    TimedEvents:
    -   event_signal: irrigate
        name:  Timed irrigation events
        comment: All irrigation amounts in cm
        events_table:
        - 2000-05-25: {amount: 3.0, efficiency=0.7}
        - 2000-06-30: {amount: 2.5, efficiency=0.7}
    StateEvents:
    -   event_signal: apply_npk
        event_state: DVS
        zero_condition: rising
        name: DVS-based N/P/K application table
        comment: all fertilizer amounts in kg/ha
        events_table:
        - 0.3: {N_amount : 1, P_amount: 3, K_amount: 4, N_recovery=0.7, P_
↪recovery=0.7, K_recovery=0.7}
```

(continues on next page)

(continued from previous page)

```
        - 0.6: {N_amount: 11, P_amount: 13, K_amount: 14, N_recovery=0.7,␣
↪P_recovery=0.7, K_recovery=0.7}
        - 1.12: {N_amount: 21, P_amount: 23, K_amount: 24, N_recovery=0.7,␣
↪P_recovery=0.7, K_recovery=0.7}
- 2000-09-01:
    CropCalendar:
    TimedEvents:
    StateEvents
- 2001-03-01:
    CropCalendar:
        crop_name: maize
        variety_name: fodder-maize
        crop_start_date: 2001-04-15
        crop_start_type: sowing
        crop_end_date:
        crop_end_type: maturity
        max_duration: 200
    TimedEvents:
    -   event_signal: irrigate
        name:  Timed irrigation events
        comment: All irrigation amounts in cm
        events_table:
        - 2001-06-01: {amount: 2.0, efficiency=0.7}
        - 2001-07-21: {amount: 5.0, efficiency=0.7}
        - 2001-08-18: {amount: 3.0, efficiency=0.7}
        - 2001-09-19: {amount: 2.5, efficiency=0.7}
    -   event_signal: apply_npk
        name:  Timed N/P/K application table
        comment: All fertilizer amounts in kg/ha
        events_table:
        - 2001-05-25: {N_amount : 50, P_amount: 25, K_amount: 22, N_
↪recovery=0.7, P_recovery=0.7, K_recovery=0.7}
        - 2001-07-05: {N_amount : 70, P_amount: 35, K_amount: 32, N_
↪recovery=0.7, P_recovery=0.7, K_recovery=0.7}
    StateEvents:
```

Crop calendars

The crop calendar definition will be passed on to a *CropCalendar* object which is responsible for storing, checking, starting and ending the crop cycle during a PCSE simulation. At each time step the instance of *CropCalendar* is called and at the dates defined by its parameters it initiates the appropriate actions:

- sowing/emergence: A *crop_start* signal is dispatched including the parameters needed to start the new crop simulation object (crop_name, variety_name, crop_start_type and crop_end_type)
- maturity/harvest: the crop cycle is ended by dispatching a *crop_finish* signal with the appropriate parameters.

For a detailed description of a crop calendar see the code documentation on the CropCalendar in the section on *Agromanagement*.

Timed events

Timed events are management actions that are occurring on specific dates. As simulations in PCSE run on daily time steps it is easy to schedule actions on dates. Timed events are characterized by an event signal, a name and comment that can be used to describe the event and finally an events table that lists the dates for the events and the parameters that should be passed onward.

Note that when multiple events are connected to the same date, the order in which they trigger is undetermined.

For a detailed description of a timed events see the code documentation on the TimedEventsDispatcher in the section on *Agromanagement*.

State events

State events are management actions that are tied to certain model states. Examples are actions such as nutrient application that should be executed at certain crop stages, or irrigation application that should occur only when the soil is dry. PCSE has a flexible definition of state events and an event can be connected to any variable that is defined within PCSE.

Each state event is defined by an *event_signal*, an *event_state* (e.g. the model state that triggers the event) and a *zero condition*. Moreover, an optional name and an optional comment can be provided. Finally the events_table specifies at which model state values the event occurs. The *events_table* is a list which provides for each state the parameters that should be dispatched with the given event_signal.

Managing state events is more complicated than timed events because PCSE cannot determine beforehand at which time step these events will trigger. For finding the time step at which a state event occurs PCSE uses the concept of *zero-crossing*. This means that a state event is triggered when (*model_state* - *event_state*) equals or crosses zero. The *zero_condition* defines how this crossing should take place. The value of *zero_condition* can be:

- *rising*: the event is triggered when (*model_state* - *event_state*) goes from a negative value towards zero or a positive value.
- *falling*: the event is triggered when (*model_state* - *event_state*) goes from a positive value towards zero or a negative value.
- *either*: the event is triggered when (*model_state* - *event_state*) crosses or reaches zero from any direction.

Note that when multiple events are connected to the same state value, the order in which they trigger is undetermined.

For a detailed description of a state events see the code documentation on the StateEventsDispatcher in the section on *Agromanagement*.

Finding the start and end date of a simulation

The agromanager has the task to find the start and end date of the simulation based on the agromanagement definition that has been provided to the Engine. Getting the start date from the agromanagement definition is straightforward as this is represented by the start date of the first campaign. However, getting the end date is more complicated because there are several possibilities. The first option is to explicitly define the end date of the simulation by adding a 'trailing empty campaign' to the agromanagement definition. An example of an agromanagement definition with a 'trailing empty campaigns' (YAML format) is given below. This example will run the simulation until 2001-01-01:

```
Version: 1.0.0
AgroManagement:
- 1999-08-01:
    CropCalendar:
        crop_name: wheat
        variety_name: winter-wheat
        crop_start_date: 1999-09-15
        crop_start_type: sowing
        crop_end_date:
        crop_end_type: maturity
        max_duration: 300
    TimedEvents:
    StateEvents:
- 2001-01-01:
```

The second option is that there is no trailing empty campaign and in that case the end date of the simulation is retrieved from the crop calendar and/or the timed events that are scheduled. In the example below, the end date will be 2000-08-05 as this is the harvest date and there are no timed events scheduled after this date:

```
Version: 1.0.0
AgroManagement:
- 1999-09-01:
    CropCalendar:
        crop_name: wheat
        variety_name: winter-wheat
        crop_start_date: 1999-10-01
        crop_start_type: sowing
        crop_end_date: 2000-08-05
        crop_end_type: harvest
        max_duration: 330
    TimedEvents:
    -   event_signal: irrigate
        name:  Timed irrigation events
        comment: All irrigation amounts in cm
        events_table:
        - 2000-05-01: {amount: 2, efficiency: 0.7}
        - 2000-06-21: {amount: 5, efficiency: 0.7}
        - 2000-07-18: {amount: 3, efficiency: 0.7}
    StateEvents:
```

In the case that there is no harvest date provided and the crop runs till maturity, the end date from the crop calendar will be estimated as the *crop_start_date* plus the *max_duration*.

Note that in an agromanagement definition where the last campaign contains a definition for state events, a trailing empty campaign *must* be provided because otherwise the end date cannot be determined. The following campaign definition is valid (though silly) but there is no way to determine the end date of the simulation. Therefore, this definition will lead to an error:

```
Version: 1.0
AgroManagement:
- 2001-01-01:
    CropCalendar:
    TimedEvents:
    StateEvents:
    -   event_signal: irrigate
```

(continues on next page)

(continued from previous page)

```
        event_state: SM
        zero_condition: falling
        name: irrigation scheduling on volumetric soil moisture content
        comment: all irrigation amounts in cm
        events_table:
        - 0.25: {amount: 2, efficiency: 0.7}
```

4.1.5 Exchanging data between model components

A complicating factor when dealing with modular code is how to exchange model states or other data between the different components. PCSE implements two basic methods for exchanging variables:

1. The VariableKiosk which is primarily used to exchange state/rate variables between model components and where updates of the state/rate variables are needed at each cycle in the simulation process.
2. The use of signals that can be broadcasted and received by any PCSE object and which is primarily used to broadcast information as a response to events that are happening during the model simulation.

The VariableKiosk

The VariableKiosk is an essential component in PCSE and it is created when the Engine starts. Nearly all objects in PCSE receive a reference to the VariableKiosk and it has many functions which may not be clear or appreciated at first glance.

First of all, the VariableKiosk registers *all* state and rate variables which are defined as attributes of a StateVariables or RateVariables class. By doing so, it also ensures that names are unique; there cannot be two state/rate variables with the same name within the component hierarchy of a single Engine. This uniqueness is enforced to avoid name conflicts between components that would affect the publishing of variables or the retrieval of variables. For example, *engine.get_variable("LAI")* will retrieve the leaf area index of the crop. However, if there would be two variables named "LAI" it would be unclear which one is retrieved. It would not even be guaranteed that it is the same variable between function calls or model runs.

Second, the VariableKiosk takes care of exchanging state and rate variables between model components. Variables that are published by the RateVariables and StateVariables object will become available in the VariableKiosk the moment when the variable gets a value assigned. Within the PCSE internals, published variables have a trigger connected to them that copies their value into the VariableKiosk. The VariableKiosk should therefore not be regarded as a shared state object but rather as a cache that contains copies of variable name/value pairs. Moreover, the updating of variables in the kiosk is protected. Only the SimulationObject that registers and publishes a variable can change its value in the Kiosk. All other SimulationObjects can query its value, but cannot alter it. Therefore it is impossible for two processes to manipulate the same variable through the VariableKiosk.

A potential danger with having copies of variables in the kiosk is that copies do not reflect the actual value anymore, for example due to a missing state update. In such case the value of the state is "lagging" in the kiosk which is a potential simulation error. To avoid such problems, the kiosk regularly 'flushes' its content. After a flush, the variables remain registered in the kiosk, but their values become undefined. The flushing of variables is taken care of by the engine and is done separately for rate and state variables. After the update of all states, all rate variables are flushed; when the rate calculation step is finished, all state variables in the kiosk are flushed. On the one hand, this procedure helps to enforce that calculations are done in the right order. On the other hand it also implies that in order to keep a state variable available in the kiosk its value *must* be updated with the corresponding rate, even if that rate is zero!

The last important function embodied by the VariableKiosk is as the sender ID of signals that are broadcasted by objects in PCSE. Each signal that is broadcasted has a sender ID and zero or more receivers. Each instance of a PCSE simulation object is configured to listen only to signals that have their own VariableKiosk as sender ID. Since the VariableKiosk is unique to each instance of an Engine, this ensures that two engines that are active in the same PCSE session, will not 'listen' to each others signals but only to their own signals. This principle becomes critical when running ensembles of models (e.g Engines) where the broadcasting of signals of the various ensemble members should not interfere between members.

In practice, a user of PCSE hardly needs to deal with the VariableKiosk; variables can be published by indicating them with the *publish=[<var1>,<var2>,. . .]* keyword when initializing rate/state variables, while retrieving values from the VariableKiosk works through the normal dictionary look up. For more details on the VariableKiosk see the description in the *Base classes* section.

Broadcasting signals

The second mechanism in PCSE for passing around information is by broadcasting signals as a result of events. This is very similar to the way a user interface toolkit works and where event handlers are connected to certain events like mouse clicks or buttons being pressed. Instead, events in PCSE are related to management actions from the AgroManager, output signals from the timer module, the termination of the simulation, etc.

Signals in PCSE are defined in the *signals* module which can be easily imported by any module that needs access to signals. Signals are simply defined as strings but any hashable object type would do. Most of the work for dealing with signals is in setting up a receiver. A receiver is usually a method on a SimulationObject that will be called when the signal is broadcasted. This method will then be connected to the signal during the initialization of the object. This is easy to describe with an example:

```
mysignal = "My first signal"

class MySimObj(SimulationObject):

    def initialize(self, day, kiosk):
        self._connect_signal(self.handle_mysignal, mysignal)

    def handle_mysignal(self, arg1, arg2):
        print "Value of arg1, arg2: %s, %s" % (arg1, arg2)

    def send_mysignal(self):
        self._send_signal(signal=mysignal, arg2="A", arg1=2.5)
```

In the example above, the *initialize()* section connects the *handle_mysignal()* method to signals of type *mysignal* having two arguments *arg1* and *arg2*. When the object is initialized and the *send_mysignal()* is called the handler will print out the values of its two arguments:

```
>>> from pcse.base import VariableKiosk
>>> from datetime import date
>>> d = date(2000,1,1)
>>> v = VariableKiosk()
>>> obj = MySimObj(d, v)
```

(continues on next page)

(continued from previous page)

```
>>> obj.send_mysignal()
Value of arg1, arg2: 2.5, A
>>>
```

Note that the methods for receiving signals *_connect_signal()* and sending signals *_send_signal()* are available because of subclassing *SimulationObject*. Both methods are highly flexible regarding the arguments and keyword arguments that can be passed on with the signal. For more details have a look at the documentation in the *Signals* module and the documentation of the PyDispatcher package which is used to provide this functionality.

4.1.6 Data providers in PCSE

PCSE needs to receive inputs on weather, parameter values and agromanagement in order to carry out the simulation. To obtain the required inputs several data providers have been written that read these inputs from a variety of sources. Nevertheless, care has been taken to avoid dependencies on a particular database and file format. As a consequence there is no direct coupling between PCSE and a particular file format or database. This ensures that a variety of data sources can be used, ranging from simple files, relational databases and internet resources.

Weather data in PCSE

Required weather variables

To run the crop simulation, the engine needs meteorological variables that drive the processes that are being simulated. PCSE requires the following daily meteorological variables:

Name	Description	Unit
TMAX	Daily maximum temperature	$^{\circ}C$
TMIN	Daily minimum temperature	$^{\circ}C$
VAP	Mean daily vapour pressure	hPa
WIND	Mean daily wind speed at 2 m above ground level	$msec^{-1}$
RAIN	Precipitation (rainfall or water equivalent in case of snow or hail).	$cmday^{-1}$
IRRAD	Daily global radiation	$Jm^{-2}day^{-1}$
SNOWDEPTH	Depth of snow cover (optional)	cm

The snow depth is an optional meteorological variable and is only used for estimating the impact of frost damage on the crop (if enabled). Snow depth can also be simulated by the *SnowMAUS* module if observations are not available on a daily basis. Furthermore there are some meteorological variables which are derived from the previous ones:

Name	Description	Unit
E0	Penman potential evaporation for a free water surface	$cmday^{-1}$
ES0	Penman potential evaporation for a bare soil surface	$cmday^{-1}$
ET0	Penman or Penman-Monteith potential evaporation for a reference crop canopy	$cmday^{-1}$
TEMP	Mean daily temperature (TMIN + TMAX)/2	$^{\circ}C$
DTEMP	Mean daytime temperature (TEMP + TMAX)/2	$^{\circ}C$
TMINRA	The 7-day running average of TMIN	$^{\circ}C$

How weather data is used in PCSE

To provide the simulation Engine with weather data PCSE uses the concept of a *WeatherDataProvider* which can retrieve its weather data from various sources but provides a single interface to the Engine for retrieving the data. This principle can be most easily explained with an example based on weather data files provided in the Getting Started section `downloads/quickstart_part3.zip`. In this example we will read the weather data from an Excel file *nl1.xlsx* using the ExcelWeatherDataProvider:

```
>>> import pcse
>>> from pcse.fileinput import ExcelWeatherDataProvider
>>> wdp = ExcelWeatherDataProvider('nl1.xlsx')
```

We can simply *print()* the weather data provider to get an overview of its contents:

```
>>> print(wdp)
Weather data provided by: ExcelWeatherDataProvider
--------Description---------
Weather data for:
Country: Netherlands
Station: Wageningen, Location Haarweg
Description: Observed data from Station Haarweg in Wageningen
Source: Meteorology and Air Quality Group, Wageningen University
Contact: Peter Uithol
----Site characteristics----
Elevation:    7.0
Latitude:  51.970
Longitude:  5.670
Data available for 2004-01-02 - 2008-12-31
Number of missing days: 32
```

Moreover, we can call the weather dataproviders with a date object to retrieve a *WeatherDataContainer* for that date:

```
>>> from datetime import date
>>> day = date(2006,7,3)
>>> wdc = wdp(day)
```

Again, we can print the WeatherDataContainer to reveal its contents:

```
>>> print(wdc)
Weather data for 2006-07-03 (DAY)
IRRAD:  29290000.00  J/m2/day
 TMIN:        17.20   Celsius
 TMAX:        29.60   Celsius
  VAP:        12.80       hPa
 RAIN:         0.00    cm/day
   E0:         0.77    cm/day
  ES0:         0.69    cm/day
  ET0:         0.72    cm/day
 WIND:         2.90     m/sec
Latitude  (LAT):    51.97 degr.
Longitude (LON):     5.67 degr.
Elevation (ELEV):    7.0 m.
```

While individual weather elements can be accessed through the standard dotted python notation:

```
>>> print(wdc.TMAX)
29.6
```

Finally, for convenience the WeatherDataProvider can also be called with a string representing a date. This string can in the format YYYYMMDD or YYYYDDD:

```
>>> print wdp("20060703")
Weather data for 2006-07-03 (DAY)
IRRAD:  29290000.00  J/m2/day
 TMIN:        17.20   Celsius
 TMAX:        29.60   Celsius
  VAP:        12.80       hPa
 RAIN:         0.00    cm/day
   E0:         0.77    cm/day
  ES0:         0.69    cm/day
  ET0:         0.72    cm/day
 WIND:         2.90     m/sec
Latitude  (LAT):    51.97 degr.
Longitude (LON):     5.67 degr.
Elevation (ELEV):    7.0 m.
```

or in the format YYYYDDD:

```
>>> print wdp("2006183")
Weather data for 2006-07-03 (DAY)
IRRAD:  29290000.00  J/m2/day
 TMIN:        17.20   Celsius
 TMAX:        29.60   Celsius
  VAP:        12.80       hPa
 RAIN:         0.00    cm/day
   E0:         0.77    cm/day
  ES0:         0.69    cm/day
  ET0:         0.72    cm/day
 WIND:         2.90     m/sec
Latitude  (LAT):    51.97 degr.
Longitude (LON):     5.67 degr.
Elevation (ELEV):    7.0 m.
```

Weather data providers available in PCSE

PCSE provides several weather data providers out of the box. First of all, it includes file-based weather data providers that use an input file on disk to retrieve data. The *CABOWeatherDataProvider* and the *ExcelWeatherDataProvider* use the structure as defined by the CABO Weather System. The ExcelWeatherDataProvider has the advantage that data can be stored in an Excel file which is easier to handle than the ASCII files of the CABOWeatherDataProvider. Furthermore, a weather data provider is available that uses a simple CSV data format, *CSVWeatherDataProvider*.

Second, there is a set of WeatherDataProviders that derive the weather data from the database tables implemented in the different versions of the European Crop Growth Monitoring System including a *CGMS8* database, a *CGMS12* database and a *CGMS14* database.

Finally, there is the global weather data provided by the agroclimatology from the NASA Power database at a resolution of 1x1 degree. PCSE provides the *NASAPowerWeatherDataProvider* which retrieves the NASA Power data from the internet for a given latitude and longitude.

Data providers for crop parameter values

PCSE has a specific data provider for crop parameters: the YAMLCropDataprovider. The difference with the generic data providers is that this data provider can read and store the parameter sets for multiple crops while the generic data providers only can hold a single set. This crop data providers is therefore suitable for running crop rotations with different crop types as the data provider can switch the active crop.

The most basic use is to call YAMLCropDataProvider with no parameters. It will than pull the crop parameters from the github repository at https://github.com/ajwdewit/WOFOST_crop_parameters:

```
>>> from pcse.fileinput import YAMLCropDataProvider
>>> p = YAMLCropDataProvider()
>>> print(p)
YAMLCropDataProvider - crop and variety not set: no activate crop␣
↪parameter set!
```

All crops and varieties have been loaded from the github repository, however no active crop has been set. Therefore, we can activate a particular crop and variety:

```
>>> p.set_active_crop('wheat', 'Winter_wheat_101')
>>> print(p)
YAMLCropDataProvider - current active crop 'wheat' with variety 'Winter_
↪wheat_101'
Available crop parameters:
 {'DTSMTB': [0.0, 0.0, 30.0, 30.0, 45.0, 30.0], 'NLAI_NPK': 1.0, 'NRESIDLV
↪': 0.004,
 'KCRIT_FR': 1.0, 'RDRLV_NPK': 0.05, 'TCPT': 10, 'DEPNR': 4.5, 'KMAXRT_FR
↪': 0.5,
 ...
 ...
 'TSUM2': 1194, 'TSUM1': 543, 'TSUMEM': 120}
```

In practice it is usually not necessary to activate a crop parameter set manually because the AgroManager can handle this. Defining an agromanagement definition with the proper *crop_name* and *variety_name* will automatically activate the crop/variety during the model simulation:

```
AgroManagement:
- 1999-08-01:
    CropCalendar:
        crop_name: wheat
        variety_name: Winter_wheat_101
        crop_start_date: 1999-09-15
        crop_start_type: sowing
        crop_end_date:
        crop_end_type: maturity
        max_duration: 300
    TimedEvents:
    StateEvents:
```

Additionally, it is possible to load YAML parameter files from your local file system:

```
>>> p = YAMLCropDataProvider(fpath=r"D:\UserData\sources\WOFOST_crop_
↪parameters")
>>> print(p)
YAMLCropDataProvider - crop and variety not set: no activate crop␣
↪parameter set!
```

(continues on next page)

(continued from previous page)

Finally, it is possible to pull data from your fork of my github repository by specifying the URL to that repository:

```
>>> p = YAMLCropDataProvider(repository="https://raw.githubusercontent.com/
↪<your_account>/WOFOST_crop_parameters/master/")
```

Note that this URL should point to the location where the raw files can be found. In case of github, these URLs start with *https://raw.githubusercontent*, for other systems (e.g. gitlab) check the manual.

To increase performance of loading parameters, the YAMLCropDataProvider will create a cache file that can be restored much quicker compared to loading the YAML files. When reading YAML files from the local file system, care is taken to ensure that the cache file is re-created when updates to the local YAML are made. However, it should be stressed that this is *not* possible when parameters are retrieved from a URL and there is a risk that parameters are loaded from an outdated cache file. In that case use *force_reload=True* to force loading the parameters from the URL.

Generic data providers for parameters

PCSE provides several modules for retrieving parameter values for use in simulation models. The general concept that is used by all data providers for parameters is that they return a python dictionary object with the parameter names and values as key/value pairs. This concept is independent of the source where the parameters come from, either a file, a relational database or an internet source. It also means that parameters can be easily defined or changed on the command prompt, which is useful when iterating over loops and changing parameter files at each iteration. For example when showing the impact of a change in a crop parameter one could easily do:

```
>>> from pcse.fileinput import CABOFileReader
>>> import numpy as np
>>> cropfile = os.path.join(data_dir, 'sug0601.crop')
>>> cropdata = CABOFileReader(cropfile)
>>> TSUM1_values = np.arange(800, 1200, 25)
>>> for tsum1 in TSUM1_values:
        cropdata["TSUM1"] = tsum1
        # code needed to run the simulation goes here
```

PCSE provides two file-based data providers for reading parameters. The first one is the *CABOFileReader* which reads parameter file in the CABO format that was used to write parameter files for models in FORTRAN or FST. A more versatile reader is the *PCSEFileReader* which uses the python language itself as its syntax. This also implies that all the python syntax features can be used in PCSE parameter files.

Finally, several data providers exist for retrieving crop, soil and site parameter values from the database of the Crop Growth Monitoring System including data providers for a *CGMS8*, *CGMS12* and *CGMS14/CGMS18* databases.

As described earlier, PCSE needs parameters to define the soil, the crop and and additional ancillary class of parameters called 'site'. Nevertheless, the different modules in PCSE have different needs, some need access to crop parameters only, but some need to combine parameter values from different sets. For example, the root dynamics module computes the maximum root depth as the minimum of the crop maximum root depth (a crop parameter) and the soil maximum root depth (a soil parameter).

The facilitate accessing different parameters from different parameter sets, all parameters are combined using a *ParameterProvider* object which provides unified access to all available parameters. Moreover, parameters from different sources can be easily combined in the ParameterProvider given that each parameter set uses the basic key/value pair principles for accessing names and values:

```
>>> import os
>>> import sqlalchemy as sa
>>> from pcse.fileinput import CABOFileReader, PCSEFileReader
>>> from pcse.base import ParameterProvider
>>> from pcse.db.pcse import fetch_sitedata
>>> import pcse.settings

# Retrieve crop data from a CABO file
>>> cropfile = os.path.join(data_dir, 'sug0601.crop')
>>> crop = CABOFileReader(cropfile)

# Retrieve soildata from a PCSE file
>>> soilfile = os.path.join(data_dir, 'lintul3_springwheat.soil')
>>> soil = PCSEFileReader(soilfile)

# Retrieve site data from the PCSE demo DB
>>> db_location = os.path.join(pcse.settings.PCSE_USER_HOME, "pcse.db")
>>> db_engine = sa.create_engine("sqlite:///" + db_location)
>>> db_metadata = sa.MetaData(db_engine)
>>> site = fetch_sitedata(db_metadata, grid=31031, year=2000)

# Combine everything into one ParameterProvider object and print some␣
↪values
>>> parprov = ParameterProvider(sitedata=site, soildata=soil,␣
↪cropdata=crop)
>>> print(parprov["AMAXTB"]) # maximum leaf assimilation rate
[0.0, 22.5, 1.0, 45.0, 1.13, 45.0, 1.8, 36.0, 2.0, 36.0]
>>> print(parprov["DRATE"])  # maximum soil drainage rate
30.0
>>> print(parprov["WAV"])  # site-specific initial soil water amount
10.0
```

Data providers for agromanagement

Similar to weather and parameter values, there are several data providers for agromanagement. The structure of the inputs for agromanagement is more complex compared to parameter values or weather variables.

The most comprehensive way to define agromanagement in PCSE is to use the YAML structure that was described in the section above on *defining agromanagement*. For reading this datastructure the *YAMLAgroManagementReader* module is available which can be provided directly as input into the Engine.

For reading Agromanagement input from a CGMS database see the sections on the database tools CGMS. Note that the support for defining agromanagement in CGMS databases is limited to crop calendars only. The CGMS database has no support for defining state and timed events yet.

4.1.7 Global PCSE settings

PCSE has a number of settings that define some global PCSE behaviour. An example of a global setting is the PCSE_USER_HOME variable which is used to define the home folder of the user. The settings are stored in two files: 1) *default_settings.py* which can be found in the PCSE installation folder under *settings/* and should not be changed. 2) *user_settings.py* which can be found in the *.pcse* folder in the user home directory. Under Windows this is typically *c:\users\<username>.pcse* while under Linux systems this is typically '/home/<username>/.pcse'.

Changing the PCSE global settings can be done by editing the file *user_settings.py*, uncommenting the entries that should be changed and changing its value. Note that dependencies in the configuration file should be respected as the default settings and user settings are parsed separately.

Adding PCSE global settings can be done by adding new entries to the *user_settings.py* file. Note that settings should be defined as ALL_CAPS. Variable names in the settings file that start with '_' will be ignored, while any other variable names will generate a warning and be neglected.

If the user settings file is corrupted and PCSE fails to start, then the best option is to delete the *user_settings.py* file from the *.pcse* folder in the user home directory. The next time PCSE starts, the *user_settings.py* will be regenerated from the default settings with all settings commented out.

Within PCSE all settings can be easily accessed by importing the settings module:

```
>>> import pcse.settings
>>> pcse.settings.PCSE_USER_HOME
'C:\\Users\\wit015\\.pcse'
>>> pcse.settings.METEO_CACHE_DIR
'C:\\Users\\wit015\\.pcse\\meteo_cache'
```

CHAPTER 5

Code documentation

5.1 Code documentation

5.1.1 How to read

The API documentation provides a description of the interface and internals of all SimulationObjects, AncillaryObjects and utility routines available in the PCSE source distribution. All SimulationObjects and AncillaryObjects are described using the same structure:

1. A short description of the object
2. The positional parameters and keywords specified in the interface.
3. A table specifying the simulation parameters needed for the simulation
4. A table specifying the state variables of the SimulationObject
5. A table specifying the rate variables of the SimulationObject
6. Signals sent or received by the SimulationObject
7. External dependencies on state/rate variables of other SimulationObjects.
8. The exceptions that are raised under which conditions.

One or more of these sections may be excluded when they are not appropriate for the SimulationObject that is described.

The table specifying the simulation parameters has the following columns:

1. The name of the parameter.
2. A description of the parameter.
3. The type of the parameter. This is provided as a three-character code with the following interpretation. The first character indicates of the parameter is a scalar **(S)** or table **(T)** parameter. The second and third character indicate whether this parameter should be present in the timerdata '**Ti**', cropdata '**Cr**', soildata '**So**' or sitedata '**Si**' dictionary.

4. The physical unit of the parameter.

The tables specifying state/rate variables have the following columns:

1. The name of the variable.
2. A description of the variable.
3. Whether the variable is published in the kiosk or not: Y|N
4. The physical unit of the variable.

Finally, all public methods of all objects are described as well.

5.1.2 Engine and models

The PCSE Engine provides the environment where SimulationObjects are 'living'. The engine takes care of reading the model configuration, initializing model components (e.g. groups of SimulationObjects), driving the simulation forward by calling the SimulationObjects, calling the agromanagement unit, keeping track of time and providing the weather data needed.

Models are treated together with the Engine, because models are simply pre-configured Engines. Any model can be started by starting the Engine with the appropriate configuration file. The only difference is that models can have methods that deal with specific characteristics of a model. This kind of functionality cannot be implemented in the Engine because the model details are not known beforehand.

class pcse.engine.**CGMSEngine**(*parameterprovider*, *weatherdataprovider*, *agromanagement*, *config=None*)

Engine to mimic CGMS behaviour.

The original CGMS did not terminate when the crop cycles was finished but instead continued with its simulation cycle but without altering the crop and soil components. This had the effect that after the crop cycle finished, all state variables were kept at the same value while the day counter increased. This behaviour is useful for two reasons:

1. CGMS generally produces dekadal output and when a day-of-maturity or day-of-harvest does not coincide with a dekad boundary the final simulation values remain available and are stored at the next dekad.
2. When aggregating spatial simulations with variability in day-of-maturity or day-of-harvest it ensures that records are available in the database tables. So GroupBy clauses in SQL queries produce the right results when computing spatial averages.

The difference with the Engine are:

1. Crop rotations are not supported
2. After a CROP_FINISH signal, the engine will continue, updating the timer but the soil, crop and agromanagement will not execute their simulation cycles. As a consequence, all state variables will retain their value.
3. TERMINATE signals have no effect.
4. CROP_FINISH signals will never remove the CROP SimulationObject.
5. run() and run_till_terminate() are not supported, only run_till() is supported.

run(*days=1*)

Advances the system state with given number of days

run_till(*rday*)
Runs the system until rday is reached.

run_till_terminate()
Runs the system until a terminate signal is sent.

class pcse.engine.**Engine**(*parameterprovider*, *weatherdataprovider*, *agromanagement*, *config=None*)
Simulation engine for simulating the combined soil/crop system.

Parameters

- **parameterprovider** – A *ParameterProvider* object providing model parameters as key/value pairs. The parameterprovider encapsulates the different parameter sets for crop, soil and site parameters.
- **weatherdataprovider** – An instance of a WeatherDataProvider that can return weather data in a WeatherDataContainer for a given date.
- **agromanagement** – AgroManagement data. The data format is described in the section on agronomic management.
- **config** – A string describing the model configuration file to use. By only giving a filename PCSE assumes it to be located in the 'conf/' folder in the main PCSE folder. If you want to provide you own configuration file, specify it as an absolute or a relative path (e.g. with a leading '.')

Engine handles the actual simulation of the combined soil- crop system. The central part of the *Engine* is the soil water balance which is continuously simulating during the entire run. In contrast, *CropSimulation* objects are only initialized after receiving a "CROP_START" signal from the AgroManagement unit. From that point onward, the combined soil-crop is simulated including the interactions between the soil and crop such as root growth and transpiration.

Similarly, the crop simulation is finalized when receiving a "CROP_FINISH" signal. At that moment the *finalize()* section on the cropsimulation is executed. Moreover, the "CROP_FINISH" signal can specify that the crop simulation object should be deleted from the hierarchy. The latter is useful for further extensions of PCSE for running crop rotations.

Finally, the entire simulation is terminated when a "TERMINATE" signal is received. At that point, the *finalize()* section on the water balance is executed and the simulation stops.

Signals handled by Engine:

***Engine* handles the following signals:**

- CROP_START: Starts an instance of *CropSimulation* for simulating crop growth. See the *_on_CROP_START* handler for details.
- CROP_FINISH: Runs the *finalize()* section an instance of *CropSimulation* and optionally deletes the cropsimulation instance. See the *_on_CROP_FINISH* handler for details.
- TERMINATE: Runs the *finalize()* section on the waterbalance module and terminates the entire simulation. See the *_on_TERMINATE* handler for details.
- OUTPUT: Preserves a copy of the value of selected state/rate variables during simulation for later use. See the *_on_OUTPUT* handler for details.

- SUMMARY_OUTPUT: Preserves a copy of the value of selected state/rate variables for later use. Summary output is usually requested only at the end of the crop simulation. See the *_on_SUMMARY_OUTPUT* handler for details.

get_output()
Returns the variables have have been stored during the simulation.

If no output is stored an empty list is returned. Otherwise, the output is returned as a list of dictionaries in chronological order. Each dictionary is a set of stored model variables for a certain date.

get_summary_output()
Returns the summary variables have have been stored during the simulation.

get_terminal_output()
Returns the terminal output variables have have been stored during the simulation.

run(*days=1*)
Advances the system state with given number of days

run_till(*rday*)
Runs the system until rday is reached.

run_till_terminate()
Runs the system until a terminate signal is sent.

set_variable(*varname*, *value*)
Sets the value of the specified state or rate variable.

Parameters

- **varname** – Name of the variable to be updated (string).
- **value** – Value that it should be updated to (float)

Returns a dict containing the increments of the variables that were updated (new - old). If the call was unsuccessful in finding the class method (see below) it will return an empty dict.

Note that 'setting' a variable (e.g. updating a model state) is much more complex than just *getting* a variable, because often some other internal variables (checksums, related state variables) must be updated as well. As there is no generic rule to 'set' a variable it is up to the model designer to implement the appropriate code to do the update.

The implementation of *set_variable()* works as follows. First it will recursively search for a class method on the simulationobjects with the name *_set_variable_<varname>* (case sensitive). If the method is found, it will be called by providing the value as input.

So for updating the crop leaf area index (varname 'LAI') to value '5.0', the call will be: *set_variable('LAI', 5.0)*. Internally, this call will search for a class method *_set_variable_LAI* which will be executed with the value '5.0' as input.

class pcse.models.**ALCEPAS**(*parameterprovider*, *weatherdataprovider*, *agromanagement*)
ALCEPAS Onion growth model.

class pcse.models.**FAO_WRSI**(*parameterprovider*, *weatherdataprovider*, *agromanagement*)
Convenience class for computing actual crop water use using the Water Requirements Satisfaction Index with a (modified) FAO WRSI approach.

Parameters

- **parameterprovider** – A ParameterProvider instance providing all parameter values
- **weatherdataprovider** – A WeatherDataProvider object
- **agromanagement** – Agromanagement data

class pcse.models.**LINGRA_NWLP_FD** (*parameterprovider*, *weatherdataprovider*, *agromanagement*)

class pcse.models.**LINGRA_PP** (*parameterprovider*, *weatherdataprovider*, *agromanagement*)

class pcse.models.**LINGRA_WLP_FD** (*parameterprovider*, *weatherdataprovider*, *agromanagement*)

class pcse.models.**LINTUL3** (*parameterprovider*, *weatherdataprovider*, *agromanagement*)

The LINTUL model (Light INTerception and UtiLisation) is a simple general crop model, which simulates dry matter production as the result of light interception and utilization with a constant light use efficiency.

LINTUL3 simulates crop growth under water-limited and nitrogen-limited conditions

Parameters

- **parameterprovider** – A *ParameterProvider* object providing model parameters as key/value pairs. The parameterprovider encapsulates the different parameter sets for crop, soil and site parameters.
- **weatherdataprovider** – An instance of a WeatherDataProvider that can return weather data in a WeatherDataContainer for a given date.
- **agromanagement** – AgroManagement data. The data format is described in the section on agronomic management.

pcse.models.**Wofost71_PP**

alias of *pcse.models.Wofost72_PP*

pcse.models.**Wofost71_WLP_FD**

alias of *pcse.models.Wofost72_WLP_FD*

class pcse.models.**Wofost72_PP** (*parameterprovider*, *weatherdataprovider*, *agromanagement*)

Convenience class for running WOFOST7.2 Potential Production.

Parameters

- **parameterprovider** – A ParameterProvider instance providing all parameter values
- **weatherdataprovider** – A WeatherDataProvider object
- **agromanagement** – Agromanagement data

class pcse.models.**Wofost72_Phenology** (*parameterprovider*, *weatherdataprovider*, *agromanagement*)

Convenience class for running WOFOST7.2 phenology only.

Parameters

- **`parameterprovider`** – A ParameterProvider instance providing all parameter values
- **`weatherdataprovider`** – A WeatherDataProvider object
- **`agromanagement`** – Agromanagement data

class `pcse.models.`**`Wofost72_WLP_FD`**(*parameterprovider*, *weatherdataprovider*, *agromanagement*)

Convenience class for running WOFOST7.2 water-limited production.

Parameters

- **`parameterprovider`** – A ParameterProvider instance providing all parameter values
- **`weatherdataprovider`** – A WeatherDataProvider object
- **`agromanagement`** – Agromanagement data

class `pcse.models.`**`Wofost80_NWLP_FD_beta`**(*parameterprovider*, *weatherdataprovider*, *agromanagement*)

Convenience class for running WOFOST8.0 nutrient and water-limited production

Parameters

- **`parameterprovider`** – A ParameterProvider instance providing all parameter values
- **`weatherdataprovider`** – A WeatherDataProvider object
- **`agromanagement`** – Agromanagement data

class `pcse.models.`**`Wofost80_PP_beta`**(*parameterprovider*, *weatherdataprovider*, *agromanagement*)

Convenience class for running WOFOST8.0 potential production (includes NPK dynamics)

Parameters

- **`parameterprovider`** – A ParameterProvider instance providing all parameter values
- **`weatherdataprovider`** – A WeatherDataProvider object
- **`agromanagement`** – Agromanagement data

class `pcse.models.`**`Wofost80_WLP_FD_beta`**(*parameterprovider*, *weatherdataprovider*, *agromanagement*)

Convenience class for running WOFOST8.0 water-limited production (includes NPK dynamics)

Parameters

- **`parameterprovider`** – A ParameterProvider instance providing all parameter values
- **`weatherdataprovider`** – A WeatherDataProvider object
- **`agromanagement`** – Agromanagement data

5.1.3 Agromanagement modules

The routines below implement the agromanagement system in PCSE including crop calendars, rotations, state and timed events. For reading agromanagement data from a file or a database structure see the sections on the *reading file input* and the *database tools*.

class pcse.agromanager.**AgroManager**(*kiosk*, **args*, ***kwargs*)

Class for continuous AgroManagement actions including crop rotations and events.

See also the documentation for the classes *CropCalendar*, *TimedEventDispatcher* and *StateEventDispatcher*.

The AgroManager takes care of executing agromanagent actions that typically occur on agricultural fields including planting and harvesting of the crop, as well as management actions such as fertilizer application, irrigation, mowing and spraying.

The agromanagement during the simulation is implemented as a sequence of campaigns. Campaigns start on a prescribed calendar date and finalize when the next campaign starts. The simulation ends either explicitly by provided a trailing empty campaign or by deriving the end date from the crop calendar and timed events in the last campaign. See also the section below on *end_date* property.

Each campaign is characterized by zero or one crop calendar, zero or more timed events and zero or more state events. The structure of the data needed as input for AgroManager is most easily understood with the example (in YAML) below. The definition consists of three campaigns, the first starting on 1999-08-01, the second starting on 2000-09-01 and the last campaign starting on 2001-03-01. The first campaign consists of a crop calendar for winter-wheat starting with sowing at the given crop_start_date. During the campaign there are timed events for irrigation at 2000-05-25 and 2000-06-30. Moreover, there are state events for fertilizer application (event_signal: apply_npk) given by development stage (DVS) at DVS 0.3, 0.6 and 1.12.

The second campaign has no crop calendar, timed events or state events. This means that this is a period of bare soil with only the water balance running. The third campaign is for fodder maize sown at 2001-04-15 with two series of timed events (one for irrigation and one for N/P/K application) and no state events. The end date of the simulation in this case will be 2001-11-01 (2001-04-15 + 200 days).

An example of an agromanagement definition file:

```
AgroManagement:
- 1999-08-01:
    CropCalendar:
        crop_name: wheat
        variety_name: winter-wheat
        crop_start_date: 1999-09-15
        crop_start_type: sowing
        crop_end_date:
        crop_end_type: maturity
        max_duration: 300
    TimedEvents:
    -   event_signal: irrigate
        name:  Timed irrigation events
        comment: All irrigation amounts in cm
        events_table:
        - 2000-05-25: {irrigation_amount: 3.0}
        - 2000-06-30: {irrigation_amount: 2.5}
```

(continues on next page)

(continued from previous page)

```
    StateEvents:
    -   event_signal: apply_npk
        event_state: DVS
        zero_condition: rising
        name: DVS-based N/P/K application table
        comment: all fertilizer amounts in kg/ha
        events_table:
        - 0.3: {N_amount : 1, P_amount: 3, K_amount: 4}
        - 0.6: {N_amount: 11, P_amount: 13, K_amount: 14}
        - 1.12: {N_amount: 21, P_amount: 23, K_amount: 24}
- 2000-09-01:
    CropCalendar:
    TimedEvents:
    StateEvents
- 2001-03-01:
    CropCalendar:
        crop_name: maize
        variety_name: fodder-maize
        crop_start_date: 2001-04-15
        crop_start_type: sowing
        crop_end_date:
        crop_end_type: maturity
        max_duration: 200
    TimedEvents:
    -   event_signal: irrigate
        name:  Timed irrigation events
        comment: All irrigation amounts in cm
        events_table:
        - 2001-06-01: {irrigation_amount: 2.0}
        - 2001-07-21: {irrigation_amount: 5.0}
        - 2001-08-18: {irrigation_amount: 3.0}
        - 2001-09-19: {irrigation_amount: 2.5}
    -   event_signal: apply_npk
        name:  Timed N/P/K application table
        comment: All fertilizer amounts in kg/ha
        events_table:
        - 2001-05-25: {N_amount : 50, P_amount: 25, K_amount: 22}
        - 2001-07-05: {N_amount : 70, P_amount: 35, K_amount: 32}
    StateEvents:
```

end_date

Retrieves the end date of the agromanagement sequence, e.g. the last simulation date.

Returns a date object

Getting the last simulation date is more complicated because there are two options.

1. Adding an explicit trailing empty campaign

The first option is to explicitly define the end date of the simulation by adding a 'trailing empty campaign' to the agromanagement definition. An example of an agromanagement definition with a 'trailing empty campaigns' (YAML format) is given below. This example will run the simulation until 2001-01-01:

```
Version: 1.0
AgroManagement:
```

(continues on next page)

(continued from previous page)

```
- 1999-08-01:
    CropCalendar:
        crop_name: winter-wheat
        variety_name: winter-wheat
        crop_start_date: 1999-09-15
        crop_start_type: sowing
        crop_end_date:
        crop_end_type: maturity
        max_duration: 300
    TimedEvents:
    StateEvents:
- 2001-01-01:
```

Note that in configurations where the last campaign contains a definition for state events, a trailing empty campaign *must* be provided because the end date cannot be determined. The following campaign definition will therefore lead to an error:

```
Version: 1.0
AgroManagement:
- 2001-01-01:
    CropCalendar:
        crop_name: maize
        variety_name: fodder-maize
        crop_start_date: 2001-04-15
        crop_start_type: sowing
        crop_end_date:
        crop_end_type: maturity
        max_duration: 200
    TimedEvents:
    StateEvents:
    -   event_signal: apply_npk
        event_state: DVS
        zero_condition: rising
        name: DVS-based N/P/K application table
        comment: all fertilizer amounts in kg/ha
        events_table:
        - 0.3: {N_amount : 1, P_amount: 3, K_amount: 4}
        - 0.6: {N_amount: 11, P_amount: 13, K_amount: 14}
        - 1.12: {N_amount: 21, P_amount: 23, K_amount: 24}
```

2. Without an explicit trailing campaign

The second option is that there is no trailing empty campaign and in that case the end date of the simulation is retrieved from the crop calendar and/or the timed events that are scheduled. In the example below, the end date will be 2000-08-05 as this is the harvest date and there are no timed events scheduled after this date:

```
Version: 1.0
AgroManagement:
- 1999-09-01:
    CropCalendar:
        crop_name: wheat
        variety_name: winter-wheat
        crop_start_date: 1999-10-01
        crop_start_type: sowing
```

(continues on next page)

(continued from previous page)

```
        crop_end_date: 2000-08-05
        crop_end_type: harvest
        max_duration: 330
    TimedEvents:
    -   event_signal: irrigate
        name:  Timed irrigation events
        comment: All irrigation amounts in cm
        events_table:
        - 2000-05-01: {irrigation_amount: 2, efficiency: 0.7}
        - 2000-06-21: {irrigation_amount: 5, efficiency: 0.7}
        - 2000-07-18: {irrigation_amount: 3, efficiency: 0.7}
    StateEvents:
```

In the case that there is no harvest date provided and the crop runs till maturity, the end date from the crop calendar will be estimated as the crop_start_date plus the max_duration.

initialize (*kiosk*, *agromanagement*)
Initialize the AgroManager.

Parameters

- **kiosk** – A PCSE variable Kiosk
- **agromanagement** – the agromanagement definition, see the example above in YAML.

ndays_in_crop_cycle
Returns the number of days of the current cropping cycle.

Returns zero if no crop cycle is active.

start_date
Retrieves the start date of the agromanagement sequence, e.g. the first simulation date

Returns a date object

class pcse.agromanager.**CropCalendar** (*kiosk*, *crop_name=None*, *variety_name=None*, *crop_start_date=None*, *crop_start_type=None*, *crop_end_date=None*, *crop_end_type=None*, *max_duration=None*)
A crop calendar for managing the crop cycle.

A *CropCalendar* object is responsible for storing, checking, starting and ending the crop cycle. The crop calendar is initialized by providing the parameters needed for defining the crop cycle. At each time step the instance of *CropCalendar* is called and at dates defined by its parameters it initiates the appropriate actions:

- sowing/emergence: A *crop_start* signal is dispatched including the parameters needed to start the new crop simulation object
- maturity/harvest: the crop cycle is ended by dispatching a *crop_finish* signal with the appropriate parameters.

Parameters

- **kiosk** – The PCSE VariableKiosk instance

- **crop_name** – String identifying the crop
- **variety_name** – String identifying the variety
- **crop_start_date** – Start date of the crop simulation
- **crop_start_type** – Start type of the crop simulation ('sowing', 'emergence')
- **crop_end_date** – End date of the crop simulation
- **crop_end_type** – End type of the crop simulation ('harvest', 'maturity', 'earliest')
- **max_duration** – Integer describing the maximum duration of the crop cycle

Returns A CropCalendar Instance

get_end_date()

Return the end date of the crop cycle.

This is either given as the harvest date or calculated as crop_start_date + max_duration

Returns a date object

get_start_date()

Returns the start date of the cycle. This is always self.crop_start_date

Returns the start date

validate(*campaign_start_date*, *next_campaign_start_date*)

Validate the crop calendar internally and against the interval for the agricultural campaign.

Parameters

- **campaign_start_date** – start date of this campaign
- **next_campaign_start_date** – start date of the next campaign

class pcse.agromanager.**TimedEventsDispatcher**(*kiosk*, *event_signal*, *name*, *comment*, *events_table*)

Takes care handling events that are connected to a date.

Events are handled by dispatching a signal (taken from the *signals* module) and providing the relevant parameters with the signal. TimedEvents can be most easily understood when looking at the definition in the agromanagement file. The following section (in YAML) provides the definition of two instances of TimedEventsDispatchers:

```
TimedEvents:
-   event_signal: irrigate
    name:  Timed irrigation events
    comment: All irrigation amounts in mm
    events_table:
    - 2000-01-01: {irrigation_amount: 20}
    - 2000-01-21: {irrigation_amount: 50}
    - 2000-03-18: {irrigation_amount: 30}
    - 2000-03-19: {irrigation_amount: 25}
-   event_signal: apply_npk
    name:  Timed N/P/K application table
    comment: All fertilizer amounts in kg/ha
```

(continues on next page)

(continued from previous page)

```
    events_table:
    - 2000-01-10: {N_amount : 10, P_amount: 5, K_amount: 2}
    - 2000-01-31: {N_amount : 30, P_amount: 15, K_amount: 12}
    - 2000-03-25: {N_amount : 50, P_amount: 25, K_amount: 22}
    - 2000-04-05: {N_amount : 70, P_amount: 35, K_amount: 32}
```

Each TimedEventDispatcher is defined by an *event_signal*, an optional name, an optional comment and the events_table. The events_table is list which provides for each date the parameters that should be dispatched with the given event_signal.

get_end_date()
Returns the last date for which a timed event is given

validate(*campaign_start_date*, *next_campaign_start_date*)
Validates the timed events given the campaign window

Parameters

- **campaign_start_date** – Start date of the campaign
- **next_campaign_start_date** – Start date of the next campaign, can be None

class pcse.agromanager.**StateEventsDispatcher**(*kiosk*, *event_signal*, *event_state*, *zero_condition*, *name*, *comment*, *events_table*)
Takes care handling events that are connected to a model state variable.

Events are handled by dispatching a signal (taken from the *signals* module) and providing the relevant parameters with the signal. StateEvents can be most easily understood when looking at the definition in the agromanagement file. The following section (in YAML) provides the definition of two instances of StateEventsDispatchers:

```
StateEvents:
-   event_signal: apply_npk
    event_state: DVS
    zero_condition: rising
    name: DVS-based N/P/K application table
    comment: all fertilizer amounts in kg/ha
    events_table:
    - 0.3: {N_amount : 1, P_amount: 3, K_amount: 4}
    - 0.6: {N_amount: 11, P_amount: 13, K_amount: 14}
    - 1.12: {N_amount: 21, P_amount: 23, K_amount: 24}
-   event_signal: irrigate
    event_state: SM
    zero_condition: falling
    name: Soil moisture driven irrigation scheduling
    comment: all irrigation amounts in cm of water
    events_table:
    - 0.15: {irrigation_amount: 20}
```

Each StateEventDispatcher is defined by an *event_signal*, an *event_state* (e.g. the model state that triggers the event) and a *zero condition*. Moreover, an optional name and an optional comment can be provided. Finally the events_table specifies at which model state values the event occurs. The events_table is a list which provides for each state the parameters that should be dispatched with the given event_signal.

For finding the time step at which a state event occurs PCSE uses the concept of *zero-crossing*. This means that a state event is triggered when (*model_state* - *event_state*) equals or crosses zero. The *zero_condition* defines how this crossing should take place. The value of *zero_condition* can be:

- ***rising*: the event is triggered when (*model_state* - *event_state*) goes from a negative value towards** zero or a positive value.
- ***falling*: the event is triggered when (*model_state* - *event_state*) goes from a positive value towards** zero or a negative value.
- ***either*: the event is triggered when (*model_state* - *event_state*) crosses or reaches zero from any** direction.

The impact of the zero_condition can be illustrated using the example definitions above. The development stage of the crop (DVS) only increases from 0 at emergence to 2 at maturity. A StateEvent set on the DVS (first example) will therefore logically have a zero_condition 'rising' although 'either' could be used as well. A DVS-based event will not occur with zero_condition set to 'falling' as the value of DVS will not decrease.

The soil moisture (SM) however can both increase and decrease. A StateEvent for applying irrigation (second example) will therefore be specified with a zero_condition 'falling' because the event must be triggered when the soil moisture level reaches or crosses the minimum level specified by the events_table. Note that if we set the zero_condition to 'either' the event would probably occur again the next time-step because the irrigation amount increase the soil moisture and (*model_state* - *event_state*) crosses zero again but from the other direction.

5.1.4 The Timer

class pcse.timer.**Timer**(*kiosk*, **args*, ***kwargs*)

This class implements a basic timer for use with the WOFOST crop model.

This object implements a simple timer that increments the current time with a fixed time-step of one day at each call and returns its value. Moreover, it generates OUTPUT signals in daily, dekadal or monthly time-steps that can be caught in order to store the state of the simulation for later use.

Initializing the timer:

```
timer = Timer(start_date, kiosk, final_date, mconf)
CurrentDate = timer()
```

Signals sent or handled:

- "OUTPUT": sent when the condition for generating output is True which depends on the output type and interval.

initialize(*kiosk*, *start_date*, *end_date*, *mconf*)

Parameters

- **day** – Start date of the simulation
- **kiosk** – Variable kiosk of the PCSE instance
- **end_date** – Final date of the simulation. For example, this date represents (START_DATE + MAX_DURATION) for a single cropping season. This

date is *not* the harvest date because signalling harvest is taken care of by the *AgroManagement* module.

- **mconf** – A ConfigurationLoader object, the timer needs access to the configuration attributes mconf.OUTPUT_INTERVAL, mconf.OUTPUT_VARS and mconf.OUTPUT_INTERVAL_DAYS

5.1.5 The waterbalance

The PCSE distribution provides several waterbalance modules:

1. WaterbalancePP which is used for simulation under non-water-limited production
2. WaterbalanceFD which is used for simulation of water-limited production under conditions of freely draining soils
3. The *SnowMAUS* for simulation the build-up and melting of the snow cover.
4. A multi-layer waterbalance implementing simulations for potential conditions, water-limited free drainage conditions and water-limited groundwater conditions (in case of shallow ground water tables). This waterbalance is in a prototype stage and not yet usable, although the source code is available in PCSE.

class pcse.soil.**WaterbalancePP**(*day*, *kiosk*, **args*, ***kwargs*)

Fake waterbalance for simulation under potential production.

Keeps the soil moisture content at field capacity and only accumulates crop transpiration and soil evaporation rates through the course of the simulation

class pcse.soil.**WaterbalanceFD**(*day*, *kiosk*, **args*, ***kwargs*)

Waterbalance for freely draining soils under water-limited production.

The purpose of the soil water balance calculations is to estimate the daily value of the soil moisture content. The soil moisture content influences soil moisture uptake and crop transpiration.

The dynamic calculations are carried out in two sections, one for the calculation of rates of change per timestep (= 1 day) and one for the calculation of summation variables and state variables. The water balance is driven by rainfall, possibly buffered as surface storage, and evapotranspiration. The processes considered are infiltration, soil water retention, percolation (here conceived as downward water flow from rooted zone to second layer), and the loss of water beyond the maximum root zone.

The textural profile of the soil is conceived as homogeneous. Initially the soil profile consists of two layers, the actually rooted soil and the soil immediately below the rooted zone until the maximum rooting depth is reached by roots(soil and crop dependent). The extension of the root zone from the initial rooting depth to maximum rooting depth is described in Root_Dynamics class. From the moment that the maximum rooting depth is reached the soil profile may be described as a one layer system depending if the roots are able to penetrate the entire profile. If not a non-rooted part remains at the bottom of the profile.

The class WaterbalanceFD is derived from WATFD.FOR in WOFOST7.1 with the exception that the depth of the soil is now completely determined by the maximum soil depth (RDMSOL) and not by the minimum of soil depth and crop maximum rooting depth (RDMCR).

Simulation parameters:

Name	Description	Type	Unit
SMFCF	Field capacity of the soil	SSo	•
SM0	Porosity of the soil	SSo	•
SMW	Wilting point of the soil	SSo	•
CRAIRC	Soil critical air content (waterlogging)	SSo	•
SOPE	maximum percolation rate root zone	SSo	$cmday^{-1}$
KSUB	maximum percolation rate subsoil	SSo	$cmday^{-1}$
RDMSOL	Soil rootable depth	SSo	cm
IFUNRN	Indicates whether non-infiltrating fraction of rain is a function of storm size (1) or not (0)	SSi	•
SSMAX	Maximum surface storage	SSi	cm
SSI	Initial surface storage	SSi	cm
WAV	Initial amount of water in total soil profile	SSi	cm
NOTINF	Maximum fraction of rain not-infiltrating into the soil	SSi	•
SMLIM	Initial maximum moisture content in initial rooting depth zone.	SSi	•

State variables:

Name	Description	Pbl	Unit
SM	Volumetric moisture content in root zone	Y	•
SS	Surface storage (layer of water on surface)	N	cm
SSI	Initial urface storage	N	cm
W	Amount of water in root zone	N	cm
WI	Initial amount of water in the root zone	N	cm
WLOW	Amount of water in the subsoil (between current rooting depth and maximum rootable depth)	N	cm
WLOWI	Initial amount of water in the subsoil		cm
WWLOW	Total amount of water in the soil profile WWLOW = WLOW + W	N	cm
WTRAT	Total water lost as transpiration as calculated by the water balance. This can be different from the CTRAT variable which only counts transpiration for a crop cycle.	N	cm
EVST	Total evaporation from the soil surface	N	cm
EVWT	Total evaporation from a water surface	N	cm
TSR	Total surface runoff	N	cm
RAINT	Total amount of rainfall (eff + non-eff)	N	cm
WDRT	Amount of water added to root zone by increase of root growth	N	cm
TOTINF	Total amount of infiltration	N	cm
TOTIRR	Total amount of effective irrigation	N	cm
PERCT	Total amount of water percolating from rooted zone to subsoil	N	cm
LOSST	Total amount of water lost to deeper soil	N	cm
DSOS	Days since oxygen stress, accumulates the number of con-	Y	•

Rate variables:

External dependencies:

Name	Description	Provided by	Unit
TRA	Crop transpiration rate	Evapotranspiration	$cmday^{-1}$
EVSMX	Maximum evaporation rate from a soil surface below the crop canopy	Evapotranspiration	$cmday^{-1}$
EVWMX	Maximum evaporation rate from a water surface below the crop canopy	Evapotranspiration	$cmday^{-1}$
RD	Rooting depth	Root_dynamics	cm

Exceptions raised:

A WaterbalanceError is raised when the waterbalance is not closing at the end of the simulation cycle (e.g water has "leaked" away).

class pcse.soil.**SnowMAUS**(*day*, *kiosk*, **args*, ***kwargs*)

Simple snow accumulation model for agrometeorological applications.

This is an implementation of the SnowMAUS model which describes the accumulation and melt of snow due to precipitation, snowmelt and sublimation. The SnowMAUS model is designed to keep track of the thickness of the layer of water that is present as snow on the surface, e.g. the Snow Water Equivalent Depth (state variable SWEDEPTH [cm]). Conversion of the SWEDEPTH to the actual snow depth (state variable SNOWDEPTH [cm]) is done by dividing the SWEDEPTH with the snow density in [cm_water/cm_snow].

Snow density is taken as a fixed value despite the fact that the snow density is known to vary with the type of snowfall, the temperature and the age of the snow pack. However, more complicated algorithms for snow density would not be consistent with the simplicy of SnowMAUS.

A drawback of the current implementation is that there is no link to the water balance yet.

Reference: M. Trnka, E. Kocmánková, J. Balek, J. Eitzinger, F. Ruget, H. Formayer, P. Hlavinka, A. Schaumberger, V. Horáková, M. Možný, Z. Žalud, Simple snow cover model for agrometeorological applications, Agricultural and Forest Meteorology, Volume 150, Issues 7–8, 15 July 2010, Pages 1115-1127, ISSN 0168-1923

http://dx.doi.org/10.1016/j.agrformet.2010.04.012

Simulation parameters: (provide in crop, soil and sitedata dictionary)

Name	Description	Type	Unit
TMINACCU1	Upper critical minimum temperature for snow accumulation.	SSi	$°C$
TMINACCU2	Lower critical minimum temperature for snow accumulation	SSi	$°C$
TMINCRIT	Critical minimum temperature for snow melt	SSi	$°C$
TMAXCRIT	Critical maximum temperature for snow melt	SSi	$°C$
RMELT	Melting rate per day per degree Celcius above the critical minimum temperature.	SSi	$cm°C^{-1}day^{-1}$
SCTHRESHOLD	Snow water equivalent above which the sublimation is taken into account.	SSi	cm
SNOWDENSITY	Density of snow	SSi	cm/cm
SWEDEPTHI	Initial depth of layer of water present as snow on the soil surface	SSi	cm

State variables:

Name	Description	Pbl	Unit
SWEDEPTH	Depth of layer of water present as snow on the surface	N	cm
SNOWDEPTH	Depth of snow present on the surface.	Y	cm

Rate variables:

Name	Description	Pbl	Unit
RSNOWACCUM	Rate of snow accumulation	N	$cmday^{-1}$
RSNOWSUBLIM	Rate of snow sublimation	N	$cmday^{-1}$
RSNOWMELT	Rate of snow melting	N	$cmday^{-1}$

5.1.6 Crop simulation processes for WOFOST

Phenology

class pcse.crop.phenology.**DVS_Phenology**(*day*, *kiosk*, **args*, ***kwargs*)

Implements the algorithms for phenologic development in WOFOST.

Phenologic development in WOFOST is expresses using a unitless scale which takes the values 0 at emergence, 1 at Anthesis (flowering) and 2 at maturity. This type of phenological development is mainly representative for cereal crops. All other crops that are simulated with WOFOST are forced into this scheme as well, although this may not be appropriate for all crops. For example, for potatoes development stage 1 represents the start of tuber formation rather than flowering.

Phenological development is mainly governed by temperature and can be modified by the effects of day length and vernalization during the period before Anthesis. After Anthesis, only temperature influences the development rate.

Simulation parameters

Name	Description	Type	Unit
TSUMEM	Temperature sum from sowing to emergence	SCr	$^{\circ}C$ day
TBASEM	Base temperature for emergence	SCr	$^{\circ}C$
TEFFMX	Maximum effective temperature for emergence	SCr	$^{\circ}C$
TSUM1	Temperature sum from emergence to anthesis	SCr	$^{\circ}C$ day
TSUM2	Temperature sum from anthesis to maturity	SCr	$^{\circ}C$ day
IDSL	Switch for phenological development options temperature only (IDSL=0), including daylength (IDSL=1) and including vernalization (IDSL>=2)	SCr SCr	•
DLO	Optimal daylength for phenological development	SCr	hr
DLC	Critical daylength for phenological development	SCr	hr
DVSI	Initial development stage at emergence. Usually this is zero, but it can be higher for crops that are transplanted (e.g. paddy rice)	SCr	•
DVSEND	Final development stage	SCr	•
DTSMTB	Daily increase in temperature sum as a function of daily mean temperature.	TCr	$^{\circ}C$

State variables

Name	Description	Pbl	Unit
DVS	Development stage	Y	•
TSUM	Temperature sum	N	$^{\circ}C$ day
TSUME	Temperature sum for emergence	N	$^{\circ}C$ day
DOS	Day of sowing	N	•
DOE	Day of emergence	N	•
DOA	Day of Anthesis	N	•
DOM	Day of maturity	N	•
DOH	Day of harvest	N	•
STAGE	Current phenological stage, can take the folowing values: *emerging|vegetative|reproductive|mature*	N	•

Rate variables

Name	Description	Pbl	Unit
DTSUME	Increase in temperature sum for emergence	N	$^{\circ}C$
DTSUM	Increase in temperature sum for anthesis or maturity	N	$^{\circ}C$
DVR	Development rate	Y	day^{-1}

External dependencies:

None

Signals sent or handled

DVS_Phenology sends the *crop_finish* signal when maturity is reached and the *end_type* is 'maturity' or 'earliest'.

class pcse.crop.phenology.**Vernalisation**(*day*, *kiosk*, **args*, ***kwargs*)

Modification of phenological development due to vernalisation.

The vernalization approach here is based on the work of Lenny van Bussel (2011), which in turn is based on Wang and Engel (1998). The basic principle is that winter wheat needs a certain number of days with temperatures within an optimum temperature range to complete its vernalisation requirement. Until the vernalisation requirement is fulfilled, the crop development is delayed.

The rate of vernalization (VERNR) is defined by the temperature response function VERNRTB.

Within the optimal temperature range 1 day is added to the vernalisation state (VERN). The reduction on the phenological development is calculated from the base and saturated vernalisation requirements (VERNBASE and VERNSAT). The reduction factor (VERNFAC) is scaled linearly between VERNBASE and VERNSAT.

A critical development stage (VERNDVS) is used to stop the effect of vernalisation when this DVS is reached. This is done to improve model stability in order to avoid that Anthesis is never reached due to a somewhat too high VERNSAT. Nevertheless, a warning is written to the log file, if this happens.

- Van Bussel, 2011. From field to globe: Upscaling of crop growth modelling. Wageningen PhD thesis. http://edepot.wur.nl/180295
- Wang and Engel, 1998. Simulation of phenological development of wheat crops. Agric. Systems 58:1 pp 1-24

Simulation parameters (provide in cropdata dictionary)

Name	Description	Type	Unit
VERNSAT	Saturated vernalisation requirements	SCr	days
VERNBASE	Base vernalisation requirements	SCr	days
VERNRTB	Rate of vernalisation as a function of daily mean temperature.	TCr	•
VERNDVS	Critical development stage after which the effect of vernalisation is halted	SCr	•

State variables

Name	Description	Pbl	Unit
VERN	Vernalisation state	N	days
DOV	Day when vernalisation requirements are fulfilled.	N	•
ISVERNALISED	Flag indicated that vernalisation requirement has been reached	Y	•

Rate variables

Name	Description	Pbl	Unit
VERNR	Rate of vernalisation	N	•
VERNFAC	Reduction factor on development rate due to vernalisation effect.	Y	•

External dependencies:

Name	Description	Provided by	Unit
DVS	Development Stage Used only to determine if the critical development stage for vernalisation (VERNDVS) is reached.	Phenology	•

Partitioning

class pcse.crop.partitioning.**DVS_Partitioning**(*day*, *kiosk*, **args*, ***kwargs*)

Class for assimilate partioning based on development stage (*DVS*).

DVS_partioning calculates the partitioning of the assimilates to roots, stems, leaves and storage organs using fixed partitioning tables as a function of crop development stage. The available assimilates are first split into below-ground and abovegrond using the values in FRTB. In a second stage they are split into leaves (*FLTB*), stems (*FSTB*) and storage organs (*FOTB*).

Since the partitioning fractions are derived from the state variable *DVS* they are regarded state variables as well.

Simulation parameters (To be provided in cropdata dictionary):

Name	Description	Type	Unit
FRTB	Partitioning to roots as a function of development stage.	TCr	•
FSTB	Partitioning to stems as a function of development stage.	TCr	•
FLTB	Partitioning to leaves as a function of development stage.	TCr	•
FOTB	Partitioning to storage organs as a function of development stage.	TCr	•

State variables

Name	Description	Pbl	Unit
FR	Fraction partitioned to roots.	Y	•
FS	Fraction partitioned to stems.	Y	•
FL	Fraction partitioned to leaves.	Y	•
FO	Fraction partitioned to storage orgains	Y	•

Rate variables

None

Signals send or handled

None

External dependencies:

Name	Description	Provided by	Unit
DVS	Crop development stage	DVS_Phenology	•

Exceptions raised

A PartitioningError is raised if the partitioning coefficients to leaves, stems and storage organs on a given day do not add up to '1'.

CO_2 Assimilation

`class pcse.crop.assimilation.WOFOST_Assimilation(day, kiosk, *args, **kwargs)`

Class implementing a WOFOST/SUCROS style assimilation routine.

WOFOST calculates the daily gross CO_2 assimilation rate of a crop from the absorbed radiation and the photosynthesis-light response curve of individual leaves. This response is dependent on temperature and leaf age. The absorbed radiation is calculated from the total incoming radiation and the leaf area. Daily gross CO_2 assimilation is obtained by integrating the assimilation rates over the leaf layers and over the day.

Simulation parameters

Name	Description	Type	Unit
AMAXTB	Max. leaf CO_2 assim. rate as a function of of DVS	TCr	kg $ha^{-1}hr^{-1}$
EFFTB	Light use effic. single leaf as a function of daily mean temperature	TCr	kg $ha^{-1}hr^{-1}/(J\ m^{-2}sec^{-1})$
KDIFTB	Extinction coefficient for diffuse visible as function of DVS	TCr	•
TMPFTB	Reduction factor of AMAX as function of daily mean temperature.	TCr	•
TMNFTB	Reduction factor of AMAX as function of daily minimum temperature.	TCr	•

State and rate variables

WOFOST_Assimilation returns the potential gross assimilation rate 'PGASS' directly from the *__call__()* method, but also includes it as a rate variable.

Rate variables:

Name	Description	Pbl	Unit
PGASS	Potential assimilation rate	N	kg CH_2O $ha^{-1}day^{-1}$

Signals sent or handled

None

External dependencies:

Name	Description	Provided by	Unit
DVS	Crop development stage	DVS_Phenology	•
LAI	Leaf area index	Leaf_dynamics	•

Maintenance respiration

class pcse.crop.respiration.**WOFOST_Maintenance_Respiration**(*day*, *kiosk*, **args*, ***kwargs*)

Maintenance respiration in WOFOST

WOFOST calculates the maintenance respiration as proportional to the dry weights of the plant organs to be maintained, where each plant organ can be assigned a different maintenance coefficient. Multiplying organ weight with the maintenance coeffients yields the relative maintenance respiration (*RMRES*) which is than corrected for senescence (parameter *RFSETB*). Finally, the actual maintenance respiration rate is calculated using the daily mean temperature, assuming a relative increase for each 10 degrees increase in temperature as defined by *Q10*.

Simulation parameters: (To be provided in cropdata dictionary):

Name	Description	Type	Unit
Q10	Relative increase in maintenance repiration rate with each 10 degrees increase in temperature	SCr	•
RMR	Relative maintenance respiration rate for roots	SCr	kg CH_2O $kg^{-1}d^{-1}$
RMS	Relative maintenance respiration rate for stems	SCr	kg CH_2O $kg^{-1}d^{-1}$
RML	Relative maintenance respiration rate for leaves	SCr	kg CH_2O $kg^{-1}d^{-1}$
RMO	Relative maintenance respiration rate for storage organs	SCr	kg CH_2O $kg^{-1}d^{-1}$

State and rate variables:

***WOFOSTMaintenanceRespiration* returns the potential maintenance respiration PMRES** directly from the *__call__()* method, but also includes it as a rate variable within the object.

Rate variables:

Name	Description	Pbl	Unit
PMRES	Potential maintenance respiration rate	N	kg CH_2O $ha^{-1}day^{-1}$

Signals send or handled

None

External dependencies:

Name	Description	Provided by	Unit
DVS	Crop development stage	DVS_Phenology	•
WRT	Dry weight of living roots	WOFOST_Root_Dynamics	$kg\ ha^{-1}$
WST	Dry weight of living stems	WOFOST_Stem_Dynamics	$kg\ ha^{-1}$
WLV	Dry weight of living leaves	WOFOST_Leaf_Dynamics	$kg\ ha^{-1}$
WSO	Dry weight of living storage organs	WOFOST_Storage_Organ_Dynamics	$kg\ ha^{-1}$

Evapotranspiration

class pcse.crop.evapotranspiration.**Evapotranspiration**(*day*, *kiosk*, **args*, ***kwargs*)

Calculation of potential evaporation (water and soil) rates and actual crop transpiration rate.

Simulation parameters:

Name	Description	Type	Unit
CFET	Correction factor for potential transpiration rate.	SCr	•
DEPNR	Dependency number for crop sensitivity to soil moisture stress.	SCr	•
KDIFTB	Extinction coefficient for diffuse visible as function of DVS.	TCr	•
IOX	Switch oxygen stress on (1) or off (0)	SCr	•
IAIRDU	Switch airducts on (1) or off (0)	SCr	•
CRAIRC	Critical air content for root aeration	SSo	•
SM0	Soil porosity	SSo	•
SMW	Volumetric soil moisture content at wilting point	SSo	•
SMCFC	Volumetric soil moisture content at field capacity	SSo	•
SM0	Soil porosity	SSo	•

State variables

Note that these state variables are only assigned after finalize() has been run.

Name	Description	Pbl	Unit
IDWST	Nr of days with water stress.	N	•
IDOST	Nr of days with oxygen stress.	N	•

Rate variables

Name	Description	Pbl	Unit
EVWMX	Maximum evaporation rate from an open water surface.	Y	cm day^{-1}
EVSMX	Maximum evaporation rate from a wet soil surface.	Y	cm day^{-1}
TRAMX	Maximum transpiration rate from the plant canopy	Y	cm day^{-1}
TRA	Actual transpiration rate from the plant canopy	Y	cm day^{-1}
IDOS	Indicates oxygen stress on this day (True|False)	N	•
IDWS	Indicates water stress on this day (True|False)	N	•
RFWS	Reduction factor for water stress	N	•
RFOS	Reduction factor for oxygen stress	N	•
RFTRA	Reduction factor for transpiration (wat & ox)	Y	•

Signals send or handled

None

External dependencies:

Name	Description	Provided by	Unit
DVS	Crop development stage	DVS_Phenology	•
LAI	Leaf area index	Leaf_dynamics	•
SM	Volumetric soil moisture content	Waterbalance	•

`pcse.crop.evapotranspiration.`**`SWEAF`**`(`*ET0*, *DEPNR*`)`

Calculates the Soil Water Easily Available Fraction (SWEAF).

Parameters

- **`ET0`** – The evapotranpiration from a reference crop.

- **DEPNR** – The crop dependency number.

The fraction of easily available soil water between field capacity and wilting point is a function of the potential evapotranspiration rate (for a closed canopy) in cm/day, ET0, and the crop group number, DEPNR (from 1 (=drought-sensitive) to 5 (=drought-resistent)). The function SWEAF describes this relationship given in tabular form by Doorenbos & Kassam (1979) and by Van Keulen & Wolf (1986; p.108, table 20) http://edepot.wur.nl/168025.

Leaf dynamics

class pcse.crop.leaf_dynamics.**WOFOST_Leaf_Dynamics**(*day, kiosk, *args, **kwargs*)

Leaf dynamics for the WOFOST crop model.

Implementation of biomass partitioning to leaves, growth and senenscence of leaves. WOFOST keeps track of the biomass that has been partitioned to the leaves for each day (variable *LV*), which is called a leaf class). For each leaf class the leaf age (variable 'LVAGE') and specific leaf area (variable *SLA*) are also registered. Total living leaf biomass is calculated by summing the biomass values for all leaf classes. Similarly, leaf area is calculated by summing leaf biomass times specific leaf area (*LV* * *SLA*).

Senescense of the leaves can occur as a result of physiological age, drought stress or self-shading.

Simulation parameters (provide in cropdata dictionary)

Name	Description	Type	Unit
RGR-LAI	Maximum relative increase in LAI.	SCr	ha ha-1 d-1
SPAN	Life span of leaves growing at 35 Celsius	SCr	day
TBASE	Lower threshold temp. for ageing of leaves	SCr	$°C$
PERDL	Max. relative death rate of leaves due to water stress	SCr	
TDWI	Initial total crop dry weight	SCr	kg ha^{-1}
KDIFTB	Extinction coefficient for diffuse visible light as function of DVS	TCr	
SLATB	Specific leaf area as a function of DVS	TCr	ha kg^{-1}

State variables

Name	Description	Pbl	Unit
LV	Leaf biomass per leaf class	N	kg ha^{-1}
SLA	Specific leaf area per leaf class	N	ha kg^{-1}
LVAGE	Leaf age per leaf class	N	day
LVSUM	Sum of LV	N	kg ha^{-1}
LAIEM	LAI at emergence	N	•
LASUM	Total leaf area as sum of LV*SLA, not including stem and pod area	N N	•
LAIEXP	LAI value under theoretical exponential growth	N	•
LAIMAX	Maximum LAI reached during growth cycle	N	•
LAI	Leaf area index, including stem and pod area	Y	•
WLV	Dry weight of living leaves	Y	kg ha^{-1}
DWLV	Dry weight of dead leaves	N	kg ha^{-1}
TWLV	Dry weight of total leaves (living + dead)	Y	kg ha^{-1}

Rate variables

Name	Description	Pbl	Unit
GRLV	Growth rate leaves	N	kg $ha^{-1} day^{-1}$
DSLV1	Death rate leaves due to water stress	N	kg $ha^{-1} day^{-1}$
DSLV2	Death rate leaves due to self-shading	N	kg $ha^{-1} day^{-1}$
DSLV3	Death rate leaves due to frost kill	N	kg $ha^{-1} day^{-1}$
DSLV	Maximum of DLSV1, DSLV2, DSLV3	N	kg $ha^{-1} day^{-1}$
DALV	Death rate leaves due to aging.	N	kg $ha^{-1} day^{-1}$
DRLV	Death rate leaves as a combination of DSLV and DALV	N	kg $ha^{-1} day^{-1}$
SLAT	Specific leaf area for current time step, adjusted for source/sink limited leaf expansion rate.	N	ha kg^{-1}
FYSAGE	Increase in physiological leaf age	N	•
GLAIEX	Sink-limited leaf expansion rate (exponential curve)	N	ha $ha^{-1} day^{-1}$
GLASOL	Source-limited leaf expansion rate (biomass increase)	N	ha $ha^{-1} day^{-1}$

External dependencies:

Name	Description	Provided by	Unit
DVS	Crop development stage	DVS_Phenology	•
FL	Fraction biomass to leaves	DVS_Partitioning	•
FR	Fraction biomass to roots	DVS_Partitioning	•
SAI	Stem area index	WOFOST_Stem_Dynamics	•
PAI	Pod area index	WOFOST_Storage_Organ_Dynamics	•
TRA	Transpiration rate	Evapotranspiration	cm day^{-1}
TRAMX	Maximum transpiration rate	Evapotranspiration	cm day^{-1}
ADMI	Above-ground dry matter increase	CropSimulation	kg $ha^{-1} day^{-1}$
RF_FROST	Reduction factor frost kill	FROSTOL	•

Root dynamics

class pcse.crop.root_dynamics.**WOFOST_Root_Dynamics**(*day*, *kiosk*, **args*, ***kwargs*)

Root biomass dynamics and rooting depth.

Root growth and root biomass dynamics in WOFOST are separate processes, with the only exception that root growth stops when no more biomass is sent to the root system.

Root biomass increase results from the assimilates partitioned to the root system. Root death is defined as the current root biomass multiplied by a relative death rate (*RDRRTB*). The latter as a function of the development stage (*DVS*).

Increase in root depth is a simple linear expansion over time until the maximum rooting depth (*RDM*) is reached.

Simulation parameters

Name	Description	Type	Unit
RDI	Initial rooting depth	SCr	cm
RRI	Daily increase in rooting depth	SCr	cm day^{-1}
RDMCR	Maximum rooting depth of the crop	SCR	cm
RDMSOL	Maximum rooting depth of the soil	SSo	cm
TDWI	Initial total crop dry weight	SCr	kg ha^{-1}
IAIRDU	Presence of air ducts in the root (1) or not (0)	SCr	•
RDRRTB	Relative death rate of roots as a function of development stage	TCr	•

State variables

Name	Description	Pbl	Unit
RD	Current rooting depth	Y	cm
RDM	Maximum attainable rooting depth at the minimum of the soil and crop maximum rooting depth	N	cm
WRT	Weight of living roots	Y	kg ha^{-1}
DWRT	Weight of dead roots	N	kg ha^{-1}
TWRT	Total weight of roots	Y	kg ha^{-1}

Rate variables

Name	Description	Pbl	Unit
RR	Growth rate root depth	N	cm
GRRT	Growth rate root biomass	N	kg ha^{-1}day^{-1}
DRRT	Death rate root biomass	N	kg ha^{-1}day^{-1}
GWRT	Net change in root biomass	N	kg ha^{-1}day^{-1}

Signals send or handled

None

External dependencies:

Name	Description	Provided by	Unit
DVS	Crop development stage	DVS_Phenology	•
DMI	Total dry matter increase	CropSimulation	kg $ha^{-1}day^{-1}$
FR	Fraction biomass to roots	DVS_Partitioning	•

Stem dynamics

class pcse.crop.stem_dynamics.**WOFOST_Stem_Dynamics**(*day*, *kiosk*, **args*, ***kwargs*)

Implementation of stem biomass dynamics.

Stem biomass increase results from the assimilates partitioned to the stem system. Stem death is defined as the current stem biomass multiplied by a relative death rate (*RDRSTB*). The latter as a function of the development stage (*DVS*).

Stems are green elements of the plant canopy and can as such contribute to the total photosynthetic active area. This is expressed as the Stem Area Index which is obtained by multiplying stem biomass with the Specific Stem Area (SSATB), which is a function of DVS.

Simulation parameters:

Name	Description	Type	Unit
TDWI	Initial total crop dry weight	SCr	kg ha^{-1}
RDRSTB	Relative death rate of stems as a function of development stage	TCr	•
SSATB	Specific Stem Area as a function of development stage	TCr	ha kg^{-1}

State variables

Name	Description	Pbl	Unit
SAI	Stem Area Index	Y	•
WST	Weight of living stems	Y	kg ha^{-1}
DWST	Weight of dead stems	N	kg ha^{-1}
TWST	Total weight of stems	Y	kg ha^{-1}

Rate variables

Name	Description	Pbl	Unit
GRST	Growth rate stem biomass	N	kg $ha^{-1} day^{-1}$
DRST	Death rate stem biomass	N	kg $ha^{-1} day^{-1}$
GWST	Net change in stem biomass	N	kg $ha^{-1} day^{-1}$

Signals send or handled

None

External dependencies:

Name	Description	Provided by	Unit
DVS	Crop development stage	DVS_Phenology	•
ADMI	Above-ground dry matter increase	CropSimulation	kg $ha^{-1} day^{-1}$
FR	Fraction biomass to roots	DVS_Partitioning	•
FS	Fraction biomass to stems	DVS_Partitioning	•

Storage organ dynamics

class pcse.crop.storage_organ_dynamics.**WOFOST_Storage_Organ_Dynamics**(*day*, *kiosk*, **args*, ***kwargs*)

Implementation of storage organ dynamics.

Storage organs are the most simple component of the plant in WOFOST and consist of a static pool of biomass. Growth of the storage organs is the result of assimilate partitioning. Death of storage organs is not implemented and the corresponding rate variable (DRSO) is always set to zero.

Pods are green elements of the plant canopy and can as such contribute to the total photosynthetic active area. This is expressed as the Pod Area Index which is obtained by multiplying pod biomass with a fixed Specific Pod Area (SPA).

Simulation parameters

Name	Description	Type	Unit
TDWI	Initial total crop dry weight	SCr	kg ha^{-1}
SPA	Specific Pod Area	SCr	ha kg^{-1}

State variables

Name	Description	Pbl	Unit
PAI	Pod Area Index	Y	•
WSO	Weight of living storage organs	Y	kg ha^{-1}
DWSO	Weight of dead storage organs	N	kg ha^{-1}
TWSO	Total weight of storage organs	Y	kg ha^{-1}

Rate variables

Name	Description	Pbl	Unit
GRSO	Growth rate storage organs	N	kg $ha^{-1}day^{-1}$
DRSO	Death rate storage organs	N	kg $ha^{-1}day^{-1}$
GWSO	Net change in storage organ biomass	N	kg $ha^{-1}day^{-1}$

Signals send or handled

None

External dependencies

Name	Description	Provided by	Unit
ADMI	Above-ground dry matter increase	CropSimulation	kg $ha^{-1}day^{-1}$
FO	Fraction biomass to storage organs	DVS_Partitioning	•
FR	Fraction biomass to roots	DVS_Partitioning	•

N/P/K dynamics

class pcse.crop.npk_dynamics.**NPK_Crop_Dynamics**(*day*, *kiosk*, **args*, ***kwargs*)

Implementation of overall NPK crop dynamics.

NPK_Crop_Dynamics implements the overall logic of N/P/K book-keeping within the crop.

Simulation parameters

Name	Description	Unit
NMAXLV_TB	Maximum N concentration in leaves as function of dvs	kg N kg-1 dry biomass
PMAXLV_TB	As for P	kg P kg-1 dry biomass
KMAXLV_TB	As for K	kg K kg-1 dry biomass
NMAXRT_FR	Maximum N concentration in roots as fraction of maximum N concentration in leaves	•
PMAXRT_FR	As for P	•
KMAXRT_FR	As for K	•
NMAXST_FR	Maximum N concentration in stems as fraction of maximum N concentration in leaves	•
KMAXST_FR	As for K	•
PMAXST_FR	As for P	•
NRESIDLV	Residual N fraction in leaves	kg N kg-1 dry biomass
PRESIDLV	Residual P fraction in leaves	kg P kg-1 dry biomass
KRESIDLV	Residual K fraction in leaves	kg K kg-1 dry biomass
NRESIDRT	Residual N fraction in roots	kg N kg-1 dry biomass
PRESIDRT	Residual P fraction in roots	kg P kg-1 dry biomass
KRESIDRT	Residual K fraction in roots	kg K kg-1 dry biomass
NRESIDST	Residual N fraction in stems	kg N kg-1 dry biomass
PRESIDST	Residual P fraction in stems	kg P kg-1 dry biomass
KRESIDST	Residual K fraction in stems	kg K kg-1 dry biomass

State variables

Name	Description	Unit
NamountLV	Actual N amount in living leaves	kg N ha^{-1}
PamountLV	Actual P amount in living leaves	kg P ha^{-1}
KamountLV	Actual K amount in living leaves	kg K ha^{-1}
NamountST	Actual N amount in living stems	kg N ha^{-1}
PamountST	Actual P amount in living stems	kg P ha^{-1}
KamountST	Actual K amount in living stems	kg K ha^{-1}
NamountSO	Actual N amount in living storage organs	kg N ha^{-1}
PamountSO	Actual P amount in living storage organs	kg P ha^{-1}
KamountSO	Actual K amount in living storage organs	kg K ha^{-1}
NamountRT	Actual N amount in living roots	kg N ha^{-1}
PamountRT	Actual P amount in living roots	kg P ha^{-1}
KamountRT	Actual K amount in living roots	kg K ha^{-1}
Nuptake_T	total absorbed N amount	kg N ha^{-1}
Puptake_T	total absorbed P amount	kg P ha^{-1}
Kuptake_T	total absorbed K amount	kg K ha^{-1}
Nfix_T	total biological fixated N amount	kg N ha^{-1}

Rate variables

Name	Description	Unit
RNamountLV	Weight increase (N) in leaves	kg N ha^{-1} d^{-1}
RPamountLV	Weight increase (P) in leaves	kg P ha^{-1} d^{-1}
RKamountLV	Weight increase (K) in leaves	kg K ha^{-1} d^{-1}
RNamountST	Weight increase (N) in stems	kg N ha^{-1} d^{-1}
RPamountST	Weight increase (P) in stems	kg P ha^{-1} d^{-1}
RKamountST	Weight increase (K) in stems	kg K ha^{-1} d^{-1}
RNamountRT	Weight increase (N) in roots	kg N ha^{-1} d^{-1}
RPamountRT	Weight increase (P) in roots	kg P ha^{-1} d^{-1}
RKamountRT	Weight increase (K) in roots	kg K ha^{-1} d^{-1}
RNamountSO	Weight increase (N) in storage organs	kg N ha^{-1} d^{-1}
RPamountSO	Weight increase (P) in storage organs	kg P ha^{-1} d^{-1}
RKamountSO	Weight increase (K) in storage organs	kg K ha^{-1} d^{-1}
RNdeathLV	Rate of N loss in leaves	kg N ha^{-1} d^{-1}
RPdeathLV	as for P	kg P ha^{-1} d^{-1}
RKdeathLV	as for K	kg K ha^{-1} d^{-1}
RNdeathST	Rate of N loss in roots	kg N ha^{-1} d^{-1}
RPdeathST	as for P	kg P ha^{-1} d^{-1}
RKdeathST	as for K	kg K ha^{-1} d^{-1}
RNdeathRT	Rate of N loss in stems	kg N ha^{-1} d^{-1}
RPdeathRT	as for P	kg P ha^{-1} d^{-1}
RKdeathRT	as for K	kg K ha^{-1} d^{-1}
RNloss	N loss due to senescence	kg N ha^{-1} d^{-1}
RPloss	P loss due to senescence	kg P ha^{-1} d^{-1}
RKloss	K loss due to senescence	kg K ha^{-1} d^{-1}

Signals send or handled

None

External dependencies

Name	Description	Provided by	Unit
DVS	Crop development stage	DVS_Phenology	•
WLV	Dry weight of living leaves	WOFOST_Leaf_Dynamics	kg ha^{-1}
WRT	Dry weight of living roots	WOFOST_Root_Dynamics	kg ha^{-1}
WST	Dry weight of living stems	WOFOST_Stem_Dynamics	kg ha^{-1}
DRLV	Death rate of leaves	WOFOST_Leaf_Dynamics	kg ha^{-1} day^{-1}
DRRT	Death rate of roots	WOFOST_Root_Dynamics	kg ha^{-1} day^{-1}
DRST	Death rate of stems	WOFOST_Stem_Dynamics	kg ha^{-1} day^{-1}

class `pcse.crop.nutrients.`**`NPK_Demand_Uptake`**(*day*, *kiosk*, **args*, ***kwargs*)

Calculates the crop N/P/K demand and its uptake from the soil.

Crop N/P/K demand is calculated as the difference between the actual N/P/K concentration (kg N/P/K per kg biomass) in the vegetative plant organs (leaves, stems and roots) and the maximum N/P/K concentration for each organ. N/P/K uptake is then estimated as the minimum of supply from the soil and demand from the crop.

Nitrogen fixation (leguminous plants) is calculated by assuming that a fixed fraction of the daily N demand is supplied by nitrogen fixation. The remaining part has to be supplied by the soil.

The N/P/K demand of the storage organs is calculated in a somewhat different way because it is assumed that the demand from the storage organs is fulfilled by translocation of N/P/K from the leaves, stems and roots. So Therefore the uptake of the storage organs is calculated as the minimum of the translocatable N/P/K (supply) and the demand from the storage organs. Moreover, there is time coefficient for translocation which takes into account that there is a delay in the availability of translocatable N/P/K

Simulation parameters

Name	Description	Unit
NMAXLV_TB	Maximum N concentration in leaves as function of DVS	kg N kg-1 dry biomass
PMAXLV_TB	As for P	kg P kg-1 dry biomass
KMAXLV_TB	As for K	kg K kg-1 dry biomass
NMAXRT_FR	Maximum N concentration in roots as fraction of maximum N concentration in leaves	•
PMAXRT_FR	As for P	•
KMAXRT_FR	As for K	•
NMAXST_FR	Maximum N concentration in stems as fraction of maximum N concentration in leaves	•
PMAXST_FR	As for P	•
KMAXST_FR	As for K	•
NMAXSO	Maximum N concentration in storage organs	kg N kg-1 dry biomass
PMAXSO	As for P	kg P kg-1 dry biomass
KMAXSO	As for K	kg K kg-1 dry biomass
NCRIT_FR	Critical N concentration as fraction of maximum N concentration for vegetative plant organs as a whole (leaves + stems)	•
PCRIT_FR	As for P	•
KCRIT_FR	As for K	•
TCNT	Time coefficient for N translation to storage organs	days
TCPT	As for P	days
TCKT	As for K	days
NFIX_FR	fraction of crop nitrogen uptake by biological fixation	kg N kg-1 dry biomass
RNUPTAKEMAX	Maximum rate of N uptake	kg N ha^{-1} d^{-1}
RPUPTAKEMAX	Maximum rate of P uptake	kg N ha^{-1} d^{-1}
RKUPTAKEMAX	Maximum rate of K uptake	kg N ha^{-1} d^{-1}

State variables

Rate variables

Name	Description	Pbl	Unit
RNuptakeLV	Rate of N uptake in leaves	Y	kg N ha^{-1} d^{-1}
RNuptakeST	Rate of N uptake in stems	Y	kg N ha^{-1} d^{-1}
RNuptakeRT	Rate of N uptake in roots	Y	kg N ha^{-1} d^{-1}
RNuptakeSO	Rate of N uptake in storage organs	Y	kg N ha^{-1} d^{-1}
RPuptakeLV	Rate of P uptake in leaves	Y	kg P ha^{-1} d^{-1}
RPuptakeST	Rate of P uptake in stems	Y	kg P ha^{-1} d^{-1}
RPuptakeRT	Rate of P uptake in roots	Y	kg P ha^{-1} d^{-1}
RPuptakeSO	Rate of P uptake in storage organs	Y	kg P ha^{-1} d^{-1}
RKuptakeLV	Rate of K uptake in leaves	Y	kg K ha^{-1} d^{-1}
RKuptakeST	Rate of K uptake in stems	Y	kg K ha^{-1} d^{-1}
RKuptakeRT	Rate of K uptake in roots	Y	kg K ha^{-1} d^{-1}
RKuptakeSO	Rate of K uptake in storage organs	Y	kg K ha^{-1} d^{-1}
RNuptake	Total rate of N uptake	Y	kg N ha^{-1} d^{-1}
RPuptake	Total rate of P uptake	Y	kg P ha^{-1} d^{-1}
RKuptake	Total rate of K uptake	Y	kg K ha^{-1} d^{-1}
RNfixation	Rate of N fixation	Y	kg N ha^{-1} d^{-1}
NdemandLV	N Demand in living leaves	N	kg N ha^{-1}
NdemandST	N Demand in living stems	N	kg N ha^{-1}
NdemandRT	N Demand in living roots	N	kg N ha^{-1}
NdemandSO	N Demand in storage organs	N	kg N ha^{-1}
PdemandLV	P Demand in living leaves	N	kg P ha^{-1}
PdemandST	P Demand in living stems	N	kg P ha^{-1}
PdemandRT	P Demand in living roots	N	kg P ha^{-1}
PdemandSO	P Demand in storage organs	N	kg P ha^{-1}
KdemandLV	K Demand in living leaves	N	kg K ha^{-1}
KdemandST	K Demand in living stems	N	kg K ha^{-1}
KdemandRT	K Demand in living roots	N	kg K ha^{-1}
KdemandSO	K Demand in storage organs	N	kg K ha^{-1}
Ndemand	Total crop N demand	N	kg N ha^{-1} d^{-1}
Pdemand	Total crop P demand	N	kg P ha^{-1} d^{-1}
Kdemand	Total crop K demand	N	kg K ha^{-1} d^{-1}

Signals send or handled

None

External dependencies

Name	Description	Provided by	Unit
DVS	Crop development stage	DVS_Phenology	•
TRA	Crop transpiration	Evapotranspiration	|cm d-1|
TRAMX	Potential crop transpiration	Evapotranspiration	|cm d-1|
NAVAIL	Total available N from soil	NPK_Soil_Dynamics	kg ha^{-1}
PAVAIL	Total available P from soil	NPK_Soil_Dynamics	kg ha^{-1}
KAVAIL	Total available K from soil	NPK_Soil_Dynamics	kg ha^{-1}
Ntranslocatable	Translocatable amount of N from stems, Leaves and roots	NPK_Translocation	kg ha^{-1}
Ptranslocatable	As for P	NPK_Translocation	kg ha^{-1}
Ktranslocatable	As for K	NPK_Translocation	kg ha^{-1}

class pcse.crop.nutrients.**NPK_Stress**(*day*, *kiosk*, **args*, ***kwargs*)

Implementation of NPK stress calculation through [NPK]nutrition index.

Stress factors are calculated based on the mass concentrations of N/P/K in the leaf and stem biomass of the plant. For each pool of nutrients, four concentrations are calculated based on the biomass for leaves and stems: - the actual concentration based on the actual amount of nutrients

divided by the actual leaf and stem biomass.

- The maximum concentration, being the maximum that the plant can absorb into its leaves and stems.
- The critical concentration, being the concentration that is needed to maintain growth rates that are not limited by N/P/K. For P and K, the critical concentration is usually equal to the maximum concentration. For N, the critical concentration can be lower than the maximum concentration. This concentration is sometimes called 'optimal concentration'.
- The residual concentration which is the amount that is locked into the plant structural biomass and cannot be mobilized anymore.

The stress index (SI) is determined as a simple ratio between those concentrations according to:

$SI = (C_a - C_r)/(C_c - C_r)$

with subscript a, r and c being the actual, residual and critical concentration for the nutrient. This equation is applied in analogue to N, P and K and results in the nitrogen nutrition index (NNI), phosphorous nutrition index (PNI) and Potassium nutrition index (KNI). Next, the NPK index (NPKI) is calculated as the minimum of NNI, PNI, KNI. Finally, the reduction factor for assimilation (NPKREF) is calculated using the reduction factor for light use efficiency (NLUE_NPK).

Simulation parameters

Name	Description	Unit
NMAXLV_TB	Maximum N concentration in leaves as function of DVS	kg N kg-1 dry biomass
PMAXLV_TB	As for P	kg P kg-1 dry biomass
KMAXLV_TB	As for K	kg K kg-1 dry biomass
NMAXRT_FR	Maximum N concentration in roots as fraction of maximum N concentration in leaves	•
PMAXRT_FR	As for P	•
KMAXRT_FR	As for K	•
NMAXST_FR	Maximum N concentration in stems as fraction of maximum N concentration in leaves	•
PMAXST_FR	As for P	•
KMAXST_FR	As for K	•
NCRIT_FR	Critical N concentration as fraction of maximum N concentration for vegetative plant organs as a whole (leaves + stems)	•
PCRIT_FR	As for P	•
KCRIT_FR	As for K	•
NRESIDLV	Residual N fraction in leaves	kg N kg-1 dry biomass
PRESIDLV	Residual P fraction in leaves	kg P kg-1 dry biomass
KRESIDLV	Residual K fraction in leaves	kg K kg-1 dry biomass
NRESIDST	Residual N fraction in stems	kg N kg-1 dry biomass
PRESIDST	Residual P fraction in stems	kg P kg-1 dry biomass
KRESIDST	Residual K fraction in stems	kg K kg-1 dry biomass
NLUE_NPK	Coefficient for the reduction of RUE due to nutrient (N-P-K) stress	•

Rate variables

The rate variables here are not real rate variables in the sense that they are derived state variables and do not represent a rate. However, as they are directly used in the rate variable calculation it is logical to put them here.

Name	Description	Pbl	Unit
NNI	Nitrogen nutrition index	Y	•
PNI	Nitrogen nutrition index	N	•
KNI	Nitrogen nutrition index	N	•
NPKI	Minimum of NNI, PNI, KNI	Y	•
RFNPK	Reduction factor for CO_2 assimlation based on NPKI and the parameter NLUE_NPK	N	•

External dependencies:

Name	Description	Provided by	Unit
DVS	Crop development stage	DVS_Phenology	•
WST	Dry weight of living stems	WOFOST_Stem_Dynamics	kg ha^{-1}
WLV	Dry weight of living leaves	WOFOST_Leaf_Dynamics	kg ha^{-1}
NamountLV	Amount of N in leaves	NPK_Crop_Dynamics	kg ha^{-1}
NamountST	Amount of N in stems	NPK_Crop_Dynamics	kg ha^{-1}
PamountLV	Amount of P in leaves	NPK_Crop_Dynamics	kg ha^{-1}
PamountST	Amount of P in stems	NPK_Crop_Dynamics	kg ha^{-1}
KamountLV	Amount of K in leaves	NPK_Crop_Dynamics	kg ha^{-1}
KamountST	Amount of K in stems	NPK_Crop_Dynamics	kg ha^{-1}

class `pcse.crop.nutrients.`**`NPK_Translocation`**(*day*, *kiosk*, **args*, ***kwargs*)

Does the bookkeeping for translocation of N/P/K from the roots, leaves and stems towards the storage organs of the crop.

First the routine calculates the state of the translocatable amount of N/P/K. This translocatable amount is defined as the amount of N/P/K above the residual N/P/K amount calculated as the residual concentration times the living biomass. The residual amount is locked into the plant structural biomass and cannot be mobilized anymore. The translocatable amount is calculated for stems, roots and leaves and published as the state variables Ntranslocatable, Ptranslocatable and Ktranslocatable.

The overal translocation rate is calculated as the minimum of supply (the translocatable amount) and demand from the storage organs as calculated in the component on Demand_Uptake. The actual rate of N/P/K translocation from the different plant organs is calculated assuming that the uptake rate is distributed over roots, stems and leaves in proportion to the translocatable amount for each organ.

Simulation parameters

Name	Description	Unit
NRESIDLV	Residual N fraction in leaves	kg N kg-1 dry biomass
PRESIDLV	Residual P fraction in leaves	kg P kg-1 dry biomass
KRESIDLV	Residual K fraction in leaves	kg K kg-1 dry biomass
NRESIDST	Residual N fraction in stems	kg N kg-1 dry biomass
PRESIDST	Residual P fraction in stems	kg P kg-1 dry biomass
KRESIDST	Residual K fraction in stems	kg K kg-1 dry biomass
NPK_TRANSLRT_FR	NPK translocation from roots as a fraction of resp. total NPK amounts translocated from leaves and stems	•

State variables

Name	Description	Pbl	Unit
NtranslocatableLV	Translocatable N amount in living leaves	N	kg N ha^{-1}
PtranslocatableLV	Translocatable P amount in living leaves	N	kg P ha^{-1}
KtranslocatableLV	Translocatable K amount in living leaves	N	kg K ha^{-1}
NtranslocatableST	Translocatable N amount in living stems	N	kg N ha^{-1}
PtranslocatableST	Translocatable P amount in living stems	N	kg P ha^{-1}
KtranslocatableST	Translocatable K amount in living stems	N	kg K ha^{-1}
NtranslocatableRT	Translocatable N amount in living roots	N	kg N ha^{-1}
PtranslocatableRT	Translocatable P amount in living roots	N	kg P ha^{-1}
KtranslocatableRT	Translocatable K amount in living roots	N	kg K ha^{-1}
Ntranslocatable	Total N amount that can be translocated to the storage organs	Y	[kg N ha-1]
Ptranslocatable	Total P amount that can be translocated to the storage organs	Y	[kg P ha-1]
Ktranslocatable	Total K amount that can be translocated to the storage organs	Y	[kg K ha-1]

Rate variables

Name	Description	Pbl	Unit
RNtranslocationLV	Weight increase (N) in leaves	Y	kg $ha^{-1}day^{-1}$
RPtranslocationLV	Weight increase (P) in leaves	Y	kg $ha^{-1}day^{-1}$
RKtranslocationLV	Weight increase (K) in leaves	Y	kg $ha^{-1}day^{-1}$
RNtranslocationST	Weight increase (N) in stems	Y	kg $ha^{-1}day^{-1}$
RPtranslocationST	Weight increase (P) in stems	Y	kg $ha^{-1}day^{-1}$
RKtranslocationST	Weight increase (K) in stems	Y	kg $ha^{-1}day^{-1}$
RNtranslocationRT	Weight increase (N) in roots	Y	kg $ha^{-1}day^{-1}$
RPtranslocationRT	Weight increase (P) in roots	Y	kg $ha^{-1}day^{-1}$
RKtranslocationRT	Weight increase (K) in roots	Y	kg $ha^{-1}day^{-1}$

Signals send or handled

None

External dependencies:

Name	Description	Provided by	Unit
DVS	Crop development stage	DVS_Phenology	•
WST	Dry weight of living stems	WOFOST_Stem_Dynamics	kg ha^{-1}
WLV	Dry weight of living leaves	WOFOST_Leaf_Dynamics	kg ha^{-1}
WRT	Dry weight of living roots	WOFOST_Root_Dynamics	kg ha^{-1}
NamountLV	Amount of N in leaves	NPK_Crop_Dynamics	kg ha^{-1}
NamountST	Amount of N in stems	NPK_Crop_Dynamics	kg ha^{-1}
NamountRT	Amount of N in roots	NPK_Crop_Dynamics	kg ha^{-1}
PamountLV	Amount of P in leaves	NPK_Crop_Dynamics	kg ha^{-1}
PamountST	Amount of P in stems	NPK_Crop_Dynamics	kg ha^{-1}
PamountRT	Amount of P in roots	NPK_Crop_Dynamics	kg ha^{-1}
KamountLV	Amount of K in leaves	NPK_Crop_Dynamics	kg ha^{-1}
KamountST	Amount of K in stems	NPK_Crop_Dynamics	kg ha^{-1}
KamountRT	Amount of K in roots	NPK_Crop_Dynamics	kg ha^{-1}

Abiotic damage

class `pcse.crop.abioticdamage.`**`FROSTOL`**(*day*, *kiosk*, **args*, ***kwargs*)

Implementation of the FROSTOL model for frost damage in winter-wheat.

Parameters

- **`day`** – start date of the simulation

- **kiosk** – variable kiosk of this PCSE instance
- **parvalues** – *ParameterProvider* object providing parameters as key/value pairs

Simulation parameters

Name	Description	Type	Unit
IDSL	Switch for phenological development options temperature only (IDSL=0), including daylength (IDSL=1) and including vernalization (IDSL>=2). FROSTOL requires IDSL>=2	SCr	•
LT50C	Critical LT50 defined as the lowest LT50 value that the wheat cultivar can obtain	SCr	$°C$
FROSTOL_H	Hardening coefficient	SCr	$°C^{-1}day^{-1}$
FROSTOL_D	Dehardening coefficient	SCr	$°C^{-3}day^{-1}$
FROSTOL_S	Low temperature stress coefficient	SCr	$°C^{-1}day^{-1}$
FROSTOL_R	Respiration stress coefficient	SCr	day^{-1}
FROSTOL_SDBASE	Minimum snow depth for respiration stress	SCr	cm
FROSTOL_SDMAX	Snow depth with maximum respiration stress. Larger snow depth does not increase stress anymore.	SCr	cm
FROSTOL_KILLCF	Steepness coefficient for logistic kill function.	SCr	•
ISNOWSRC	Use prescribed snow depth from driving variables (0) or modelled snow depth through the kiosk (1)	SSi	•

State variables

Name	Description	Pbl	Unit
LT50T	Current LT50 value	N	$^{\circ}C$
LT50I	Initial LT50 value of unhardened crop	N	$^{\circ}C$
IDFST	Total number of days with frost stress	N	•

Rate variables

Name	Description	Pbl	Unit
RH	Rate of hardening	N	$^{\circ}Cday^{-1}$
RDH_TEMP	Rate of dehardening due to temperature	N	$^{\circ}Cday^{-1}$
RDH_RESP	Rate of dehardening due to respiration stress	N	$^{\circ}Cday^{-1}$
RDH_TSTR	Rate of dehardening due to temperature stress	N	$^{\circ}Cday^{-1}$
IDFS	Frost stress, yes (1) or no (0). Frost stress is defined as: RF_FROST > 0	N	•
RF_FROST	Reduction factor on leave biomass as a function of min. crown temperature and LT50T: ranges from 0 (no damage) to 1 (complete kill).	Y	•
RF_FROST_T	Total frost kill through the growing season is computed as the multiplication of the daily frost kill events, 0 means no damage, 1 means total frost kill.	N	•

External dependencies:

Name	Description	Provided by	Unit
TEMP_CROWN	Daily average crown temperature derived from calling the crown_temperature module.	CrownTemperature	$^{\circ}C$
TMIN_CROWN	Daily minimum crown temperature derived from calling the crown_temperature module.	CrownTemperature	$^{\circ}C$
ISVERNALISED	Boolean reflecting the vernalisation state of the crop.	Vernalisation i.c.m. with DVS_Phenology module	•

Reference: Anne Kari Bergjord, Helge Bonesmo, Arne Oddvar Skjelvag, 2008. Modelling the course of frost tolerance in winter wheat: I. Model development, European Journal of Agronomy, Volume 28, Issue 3, April 2008, Pages 321-330.

http://dx.doi.org/10.1016/j.eja.2007.10.002

class `pcse.crop.abioticdamage.`**`CrownTemperature`**(*day*, *kiosk*, **args*, ***kwargs*)

Implementation of a simple algorithm for estimating the crown temperature (2cm under the soil surface) under snow.

Is is based on a simple empirical equation which estimates the daily minimum, maximum and mean crown temperature as a function of daily min or max temperature and the relative snow depth (RSD):

$RSD = min(15, SD)/15$

and

$T^{crown}_{min} = T_{min} * (A + B(1 - RSD)^2)$

and

$T^{crown}_{max} = T_{max} * (A + B(1 - RSD)^2)$

and

$T^{crown}_{avg} = (T^{crown}_{max} + T^{crown}_{min})/2$

At zero snow depth crown temperature is estimated close the the air temperature. Increasing snow depth acts as a buffer damping the effect of low air temperature on the crown temperature. The maximum value of the snow depth is limited on 15cm. Typical values for A and B are 0.2 and 0.5

Note that the crown temperature is only estimated if drv.TMIN<0, otherwise the TMIN, TMAX and daily average temperature (TEMP) are returned.

Parameters

- **`day`** – day when model is initialized

- **kiosk** – VariableKiosk of this instance
- **parvalues** – *ParameterProvider* object providing parameters as key/value pairs

Returns a tuple containing minimum, maximum and daily average crown temperature.

Simulation parameters

Name	Description	Type	Unit
ISNOWSRC	Use prescribed snow depth from driving variables (0) or modelled snow depth through the kiosk (1)	SSi	•
CROWNTMPA	A parameter in equation for crown temperature	SSi	•
CROWNTMPB	B parameter in equation for crown temperature	SSi	•

Rate variables

Name	Description	Pbl	Unit
TEMP_CROWN	Daily average crown temperature	N	$^{\circ}C$
TMIN_CROWN	Daily minimum crown temperature	N	$^{\circ}C$
TMAX_CROWN	Daily maximum crown temperature	N	$^{\circ}C$

Note that the calculated crown temperatures are not real rate variables as they do not pertain to rate of change. In fact they are a *derived driving variable*. Nevertheless for calculating the frost damage they should become available during the rate calculation step and by treating them as rate variables, they can be found by a *get_variable()* call and thus be defined in the list of OUTPUT_VARS in the configuration file

External dependencies:

Name	Description	Provided by	Unit
SNOWDEPTH	Depth of snow cover.	Prescibed by driving variables or simulated by snow cover module and taken from kiosk	cm

5.1.7 Crop simulation processes for LINGRA & LINGRA-N

Implementation of the LINGRA grassland simulation model

This module provides an implementation of the LINGRA (LINtul GRAssland) simulation model for grasslands as described by Schapendonk et al. 1998 (https://doi.org/10.1016/S1161-0301(98)00027-6) for use within the Python Crop Simulation Environment.

Overall grassland model

class pcse.crop.lingra.**LINGRA**(*day*, *kiosk*, **args*, ***kwargs*)

Top level implementation of LINGRA, integrating all components

This class integrates all components from the LINGRA model and includes the main state variables related to weights of the different biomass pools, the leaf area, tiller number and leaf length. The integrated components include the implementations for source/sink limited growth, soil temperature, evapotranspiration and root dynamics. The latter two are taken from WOFOST in order to avoid duplication of code.

Compared to the original code from Schapendonk et al. (1998) several improvements have been made:

- an overall restructuring of the code, removing unneeded variables and renaming the remaining variables to have more readable names.
- A clearer implementation of sink/source limited growth including the use of reserves
- the potential leaf elongation rate as calculated by the Sink-limited growth module is now corrected for actual growth. Thereby avoiding unlimited leaf growth under water-stressed conditions which led to unrealistic results.

Simulation parameters:

Name	Description	Unit
LAIinit	Initial leaf area index	•
TillerNumberinit	Initial number of tillers	tillers/m2
WeightREinit	Initial weight of reserves	kg/ha
WeightRTinit	Initial weight of roots	kg/ha
LAIcrit	Critical LAI for death due to self-shading	•
RDRbase	Background relative death rate for roots	d-1
RDRShading	Max relative death rate of leaves due to self-shading	d-1
RDRdrought	Max relative death rate of leaves due to drought stress	d-1
SLA	Specific leaf area	ha/kg
TempBase	Base temperature for photosynthesis and development	C
PartitioningRootsTB	Partitioning fraction to roots as a function of the reduction factor for transpiration (RFTRA)	-, -
TSUMmax	Temperature sum to max development stage	C.d

Rate variables

Name	Description	Unit
dTSUM	Change in temperature sum for development	C
dLAI	Net change in Leaf Area Index	d-1
dDaysAfterHarvest	Change in Days after Harvest	•
dCuttingNumber	Change in number of cuttings (harvests)	•
dWeightLV	Net change in leaf weight	kg/ha/d
dWeightRE	Net change in reserve pool	kg/ha/d
dLeafLengthAct	Change in actual leaf length	cm/d
LVdeath	Leaf death rate	kg/ha/d
LVgrowth	Leaf growth rate	kg/ha/d
dWeightHARV	Change in harvested dry matter	kg/ha/d
dWeightRT	Net change in root weight	kg/ha/d
LVfraction	Fraction partitioned to leaves	•
RTfraction	Fraction partitioned to roots	•

State variables

Name	Description	Unit
TSUM	Temperature sum	C d
LAI	Leaf area Index	•
DaysAfterHarvest	number of days after harvest	d
CuttingNumber	number of cuttings (harvests)	•
TillerNumber	Tiller number	tillers/m2
WeightLVgreen	Weight of green leaves	kg/ha
WeightLVdead	Weight of dead leaves	kg/ha
WeightHARV	Weight of harvested dry matter	kg/ha
WeightRE	Weight of reserves	kg/ha
WeightRT	Weight of roots	kg/ha
LeafLength	Length of leaves	kg/ha
WeightABG	Total aboveground weight (harvested + current)	kg/ha
SLAINT	Integrated SLA during the season	ha/kg
DVS	Development stage	•

Signals sent or handled

Mowing of grass will take place when a *pcse.signals.mowing* event is broadcasted. This will reduce the amount of living leaf weight assuming that a certain amount of biomass will remain on the field (this is a parameter on the MOWING event).

External dependencies:

Name	Description	Provided by
RFTRA	Reduction factor for transpiration	pcse.crop.Evapotranspiration
dLeafLengthPot	Potential growth in leaf length	pcse.crop.lingra.SinkLimitedGrowth
dTillerNumber	Change in tiller number	pcse.crop.lingra.SinkLimitedGrowth

Source/Sink limited growth

class pcse.crop.lingra.**SourceLimitedGrowth**(*day*, *kiosk*, **args*, ***kwargs*)

Calculates the source-limited growth rate for grassland based on radiation and temperature as driving variables and possibly limited by soil moisture or leaf nitrogen content.The latter is based on static values for current and maximum N concentrations and is mainly there for connecting an N module in the future.

This routine uses a light use efficiency (LUE) approach where the LUE is adjusted for effects of temperature and radiation level. The former is need as photosynthesis has a clear temperature response. The latter is required as photosynthesis rate flattens off at higher radiation levels which leads to a lower 'apparent' light use efficiency. The parameter *LUEreductionRadiationTB* is a crude empirical correction for this effect.

Note that a reduction in growth rate due to soil moisture is obtained through the reduction factor for transpiration (RFTRA).

This module does not provide any true rate variables, but returns the computed growth rate directly to the calling routine through __call__().

Simulation parameters:

Name	Description	Unit
KDIFTB	Extinction coefficient for diffuse visible as function of DVS.	•
CO2A	Atmospheric CO2 concentration	ppm
LUEreductionSoilTempTB	Reduction function for light use efficiency as a function of soil temperature.	C, -
LUEreductionRadiationTB	Reduction function for light use efficiency as a function of radiation level.	MJ, -
LUEmax	Maximum light use efficiency.	

Rate variables

Name	Description	Unit
RF_RadiationLevel	Reduction factor for light use efficiency due to the radiation level	•
RF_RadiationLevel	Reduction factor for light use efficiency due to the radiation level	•
LUEact	The actual light use efficiency	g /(MJ PAR)

Signals send or handled

None

External dependencies:

Name	Description	Provided by
DVS	Crop development stage	pylingra.LINGRA
TemperatureSoil	Soil Temperature	pylingra.SoilTemperature
RFTRA	Reduction factor for transpiration	pcse.crop.Evapotranspiration

class pcse.crop.lingra.**SinkLimitedGrowth**(*day*, *kiosk*, **args*, ***kwargs*)

Calculates the sink-limited growth rate for grassland assuming a temperature driven maximum leaf elongation rate multiplied by the number of tillers. The conversion to growth in kg/ha dry matter is done by dividing by the specific leaf area (SLA).

Besides the sink-limited growth rate, this class also computes the change in tiller number taking into account the growth rate, death rate and number of days after defoliation due to harvest.

Simulation parameters:

Name	Description	Unit
TempBase	Base temperature for leaf development and grass phenology	C
LAICrit	Cricical leaf area beyond which leaf death due to self-shading occurs	•
SiteFillingMax	Maximum site filling for new buds	tiller/leaf-1
SLA	Specific leaf area	ha/kg
TSUMmax	Temperature sum to max development stage	C.d
TillerFormRateA0	A parameter in the equation for tiller formation rate valid up till 7 days after harvest	
TillerFormRateB0	B parameter in the equation for tiller formation rate valid up till 7 days after harvest	
TillerFormRateA8	A parameter in the equation for tiller formation rate starting from 8 days after harvest	
TillerFormRateB8	B parameter in the equation for tiller formation rate starting from 8 days after harvest	

Rate variables

Name	Description	Unit
dTillerNumber	Change in tiller number due to the radiation level	tillers/m2/d
dLeafLengthPot	Potential change in leaf length. Later on the actual change in leaf length will be computed taking source limitation into account.	cm/d
LAIGrowthSink	Growth of LAI based on sink-limited growth rate.	d-1

Signals send or handled

None

External dependencies:

Name	Description	Provided by
DVS	Crop development stage	pylingra.LINGRA
LAI	Leaf Area Index	pylingra.LINGRA
TemperatureSoil	Soil Temperature	pylingra.SoilTemperature
RF_Temperature	Reduction factor for LUE based on temperature	pylingra.SourceLimitedGrowth
TillerNumber	Actual number of tillers	pylingra.LINGRA
LVfraction	Fraction of assimilates going to leaves	pylingra.LINGRA
dWeightHARV	Change in harvested weight (indicates that a harvest took place today)	pylingra.LINGRA

Nitrogen dynamics

class pcse.crop.lingra_ndynamics.**N_Demand_Uptake**(*day*, *kiosk*, **args*, ***kwargs*)

Calculates the crop N demand and its uptake from the soil.

Crop N demand is calculated as the difference between the actual N concentration (kg N per kg biomass) in the vegetative plant organs (leaves, stems and roots) and the maximum N concentration for each organ. N uptake is then estimated as the minimum of supply from the soil and demand from the crop.

Simulation parameters

Rate variables

Name	Description	Pbl	Unit
RNuptakeLV	Rate of N uptake in leaves	Y	kg N ha^{-1} d^{-1}
RNuptakeRT	Rate of N uptake in roots	Y	kg N ha^{-1} d^{-1}
RNuptake	Total rate of N uptake	Y	kg N ha^{-1} d^{-1}
NdemandLV	Ndemand of leaves based on current growth rate and deficienties from previous time steps	N	kg N ha^{-1}
NdemandRT	N demand of roots, idem as leaves	N	kg N ha^{-1}
Ndemand	Total N demand (leaves + roots)	N	kg N ha^{-1}

Signals send or handled

None

External dependencies

Name	Description	Provided by	Unit
DVS	Crop development stage	DVS_Phenology	•
NAVAIL	Total available N from soil	NPK_Soil_Dynamics	kg ha^{-1}

class `pcse.crop.lingra_ndynamics.`**`N_Stress`**(*day*, *kiosk*, **args*, ***kwargs*)

Implementation of N stress calculation through nitrogen nutrition index.

Stress factors are calculated based on the mass concentrations of N in the vegetative biomass of the plant. For each pool of nutrients, four concentrations are calculated based on the biomass for leaves and stems: - the actual concentration based on the actual amount of nutrients

divided by the vegetative biomass.

- The maximum concentration, being the maximum that the plant can absorb into its leaves and stems.
- The critical concentration, being the concentration that is needed to maintain growth rates that are not limited by N (regulated by NCRIT_FR). For N, the critical concentration can be lower than the maximum concentration. This concentration is sometimes called 'optimal concentration'.
- The residual concentration which is the amount that is locked into the plant structural biomass and cannot be mobilized anymore.

The stress index (SI) is determined as a simple ratio between those concentrations according to:

$SI = (C_{a) - C_r)/(C_c - C_r)}$

with subscript *a*, *r* and *c* being the actual, residual and critical concentration for the nutrient. This results in the nitrogen nutrition index (NNI). Finally, the reduction factor for assimilation (RFNUTR) is calculated using the reduction factor for light use efficiency (NLUE).

Simulation parameters

Rate variables

The rate variables here are not real rate variables in the sense that they are derived state variables and do not represent a rate. However, as they are directly used in the rate variable calculation it is logical to put them here.

Name	Description	Pbl	Unit
NNI	Nitrogen nutrition index	Y	•
RFNUTR	Reduction factor for light use efficiency	Y	•

External dependencies:

Name	Description	Provided by	Unit
DVS	Crop development stage	DVS_Phenology	•
WST	Dry weight of living stems	WOFOST_Stem_Dynamics	kg ha^{-1}
WeightLVgreen	Dry weight of living leaves	WOFOST_Leaf_Dynamics	kg ha^{-1}
NamountLV	Amount of N in leaves	N_Crop_Dynamics	kg ha^{-1}

class `pcse.crop.lingra_ndynamics.`**`N_Crop_Dynamics`**(*day*, *kiosk*, **args*, ***kwargs*)

Implementation of overall N crop dynamics.

NPK_Crop_Dynamics implements the overall logic of N book-keeping within the crop.

Simulation parameters

State variables

Name	Description	Pbl	Unit
NamountLV	Actual N amount in living leaves	Y	kg N ha^{-1}
NamountRT	Actual N amount in living roots	Y	kg N ha^{-1}
Nuptake_T	total absorbed N amount	N	kg N ha^{-1}
Nlosses_T	Total N amount lost due to senescence	N	kg N ha^{-1}

Rate variables

Name	Description	Pbl	Unit
RNamountLV	Weight increase (N) in leaves	N	kg ha^{-1}day^{-1}
RNamountRT	Weight increase (N) in roots	N	kg ha^{-1}day^{-1}
RNdeathLV	Rate of N loss in leaves	N	kg ha^{-1}day^{-1}
RNdeathRT	Rate of N loss in roots	N	kg ha^{-1}day^{-1}
RNloss	N loss due to senescence	N	kg ha^{-1}day^{-1}

Signals send or handled

None

External dependencies

5.1.8 Base classes

The base classes define much of the functionality which is used “under the hood” in PCSE. Except for the *VariableKiosk* and the *WeatherDataContainer* all classes are not to be called directly but should be subclassed instead.

VariableKiosk

class pcse.base.**VariableKiosk**

VariableKiosk for registering and publishing state variables in PCSE.

No parameters are needed for instantiating the VariableKiosk. All variables that are defined within PCSE will be registered within the VariableKiosk, while usually only a small subset of those will be published with the kiosk. The value of the published variables can be retrieved with the bracket notation as the variableKiosk is essentially a (somewhat fancy) dictionary.

Registering/deregistering rate and state variables goes through the *self.register_variable()* and *self.deregister_variable()* methods while the *set_variable()* method is used to update a value of a published variable. In general, none of these methods need to be called by users directly as the logic within the *StatesTemplate* and *RatesTemplate* takes care of this.

Finally, the *variable_exists()* can be used to check if a variable is registered, while the *flush_states()* and *flush_rates()* are used to remove (flush) the values of any published state and rate variables.

example:

```
>>> import pcse
>>> from pcse.base import VariableKiosk
>>>
>>> v = VariableKiosk()
>>> id0 = 0
>>> v.register_variable(id0, "VAR1", type="S", publish=True)
>>> v.register_variable(id0, "VAR2", type="S", publish=False)
>>>
>>> id1 = 1
>>> v.register_variable(id1, "VAR3", type="R", publish=True)
>>> v.register_variable(id1, "VAR4", type="R", publish=False)
>>>
>>> v.set_variable(id0, "VAR1", 1.35)
>>> v.set_variable(id1, "VAR3", 310.56)
>>>
>>> print v
Contents of VariableKiosk:
 * Registered state variables: 2
 * Published state variables: 1 with values:
  - variable VAR1, value: 1.35
 * Registered rate variables: 2
 * Published rate variables: 1 with values:
  - variable VAR3, value: 310.56

>>> print v["VAR3"]
310.56
>>> v.set_variable(id0, "VAR3", 750.12)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "pcse/base.py", line 148, in set_variable
    raise exc.VariableKioskError(msg % varname)
pcse.exceptions.VariableKioskError: Unregistered object tried to set␣
↪the value of variable 'VAR3': access denied.
>>>
>>> v.flush_rates()
>>> print v
```

(continues on next page)

(continued from previous page)

```
Contents of VariableKiosk:
 * Registered state variables: 2
 * Published state variables: 1 with values:
  - variable VAR1, value: 1.35
 * Registered rate variables: 2
 * Published rate variables: 1 with values:
  - variable VAR3, value: undefined

>>> v.flush_states()
>>> print v
Contents of VariableKiosk:
 * Registered state variables: 2
 * Published state variables: 1 with values:
  - variable VAR1, value: undefined
 * Registered rate variables: 2
 * Published rate variables: 1 with values:
  - variable VAR3, value: undefined
```

deregister_variable(*oid*, *varname*)
Object with id(object) asks to deregister varname from kiosk

Parameters

- **oid** – Object id (from python builtin id() function) of the state/rate object registering this variable.
- **varname** – Name of the variable to be registered, e.g. "DVS"

flush_rates()
flush the values of all published rate variable from the kiosk.

flush_states()
flush the values of all state variable from the kiosk.

register_variable(*oid*, *varname*, *type*, *publish=False*)
Register a varname from object with id, with given type

Parameters

- **oid** – Object id (from python builtin id() function) of the state/rate object registering this variable.
- **varname** – Name of the variable to be registered, e.g. "DVS"
- **type** – Either "R" (rate) or "S" (state) variable, is handled automatically by the states/rates template class.
- **publish** – True if variable should be published in the kiosk, defaults to False

set_variable(*id*, *varname*, *value*)
Let object with id, set the value of variable varname

Parameters

- **id** – Object id (from python builtin id() function) of the state/rate object registering this variable.
- **varname** – Name of the variable to be updated

- **value** – Value to be assigned to the variable.

variable_exists (*varname*)

Returns True if the state/rate variable is registered in the kiosk.

Parameters **varname** – Name of the variable to be checked for registration.

Base classes for parameters, rates and states

class pcse.base.**StatesTemplate** (*kiosk=None*, *publish=None*, ***kwargs*)

Takes care of assigning initial values to state variables, registering variables in the kiosk and monitoring assignments to variables that are published.

Parameters

- **kiosk** – Instance of the VariableKiosk class. All state variables will be registered in the kiosk in order to enfore that variable names are unique across the model. Moreover, the value of variables that are published will be available through the VariableKiosk.
- **publish** – Lists the variables whose values need to be published in the VariableKiosk. Can be omitted if no variables need to be published.

Initial values for state variables can be specified as keyword when instantiating a States class.

example:

```
>>> import pcse
>>> from pcse.base import VariableKiosk, StatesTemplate
>>> from pcse.traitlets import Float, Integer, Instance
>>> from datetime import date
>>>
>>> k = VariableKiosk()
>>> class StateVariables(StatesTemplate):
...     StateA = Float()
...     StateB = Integer()
...     StateC = Instance(date)
...
>>> s1 = StateVariables(k, StateA=0., StateB=78, StateC=date(2003,7,
↪3),
...                         publish="StateC")
>>> print s1.StateA, s1.StateB, s1.StateC
0.0 78 2003-07-03
>>> print k
Contents of VariableKiosk:
 * Registered state variables: 3
 * Published state variables: 1 with values:
  - variable StateC, value: 2003-07-03
 * Registered rate variables: 0
 * Published rate variables: 0 with values:

>>>
>>> s2 = StateVariables(k, StateA=200., StateB=1240)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "pcse/base.py", line 396, in __init__
    raise exc.PCSEError(msg)
pcse.exceptions.PCSEError: Initial value for state StateC missing.
```

touch()
Re-assigns the value of each state variable, thereby updating its value in the variablekiosk if the variable is published.

class pcse.base.**RatesTemplate**(*kiosk=None*, *publish=None*)
Takes care of registering variables in the kiosk and monitoring assignments to variables that are published.

Parameters

- **kiosk** – Instance of the VariableKiosk class. All rate variables will be registered in the kiosk in order to enfore that variable names are unique across the model. Moreover, the value of variables that are published will be available through the VariableKiosk.
- **publish** – Lists the variables whose values need to be published in the VariableKiosk. Can be omitted if no variables need to be published.

For an example see the *StatesTemplate*. The only difference is that the initial value of rate variables does not need to be specified because the value will be set to zero (Int, Float variables) or False (Boolean variables).

zerofy()
Sets the values of all rate values to zero (Int, Float) or False (Boolean).

class pcse.base.**ParamTemplate**(*parvalues*)
Template for storing parameter values.

This is meant to be subclassed by the actual class where the parameters are defined.

example:

```
>>> import pcse
>>> from pcse.base import ParamTemplate
>>> from pcse.traitlets import Float
>>>
>>>
>>> class Parameters(ParamTemplate):
...     A = Float()
...     B = Float()
...     C = Float()
...
>>> parvalues = {"A" :1., "B" :-99, "C":2.45}
>>> params = Parameters(parvalues)
>>> params.A
1.0
>>> params.A; params.B; params.C
1.0
-99.0
2.4500000000000002
>>> parvalues = {"A" :1., "B" :-99}
>>> params = Parameters(parvalues)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "pcse/base.py", line 205, in __init__
    raise exc.ParameterError(msg)
pcse.exceptions.ParameterError: Value for parameter C missing.
```

Base and utility classes for weather data

class pcse.base.**WeatherDataProvider**

Base class for all weather data providers.

Support for weather ensembles in a WeatherDataProvider has to be indicated by setting the class variable *supports_ensembles = True*

Example:

```
class MyWeatherDataProviderWithEnsembles(WeatherDataProvider):
    supports_ensembles = True

    def __init__(self):
        WeatherDataProvider.__init__(self)

        # remaining initialization stuff goes here.
```

check_keydate(*key*)

Check representations of date for storage/retrieval of weather data.

The following formats are supported:

1. a date object
2. a datetime object
3. a string of the format YYYYMMDD
4. a string of the format YYYYDDD

Formats 2-4 are all converted into a date object internally.

export()

Exports the contents of the WeatherDataProvider as a list of dictionaries.

The results from export can be directly converted to a Pandas dataframe which is convenient for plotting or analyses.

class pcse.base.**WeatherDataContainer**(**args*, ***kwargs*)

Class for storing weather data elements.

Weather data elements are provided through keywords that are also the attribute names under which the variables can accessed in the WeatherDataContainer. So the keyword TMAX=15 sets an attribute TMAX with value 15.

The following keywords are compulsory:

Parameters

- **LAT** – Latitude of location (decimal degree)
- **LON** – Longitude of location (decimal degree)
- **ELEV** – Elevation of location (meters)
- **DAY** – the day of observation (python datetime.date)
- **IRRAD** – Incoming global radiaiton (J/m2/day)
- **TMIN** – Daily minimum temperature (Celsius)
- **TMAX** – Daily maximum temperature (Celsius)

- **VAP** – Daily mean vapour pressure (hPa)
- **RAIN** – Daily total rainfall (cm/day)
- **WIND** – Daily mean wind speed at 2m height (m/sec)
- **E0** – Daily evaporation rate from open water (cm/day)
- **ES0** – Daily evaporation rate from bare soil (cm/day)
- **ET0** – Daily evapotranspiration rate from reference crop (cm/day)

There are two optional keywords arguments:

Parameters

- **TEMP** – Daily mean temperature (Celsius), will otherwise be derived from (TMAX+TMIN)/2.
- **SNOWDEPTH** – Depth of snow cover (cm)

add_variable (*varname*, *value*, *unit*)

Adds an attribute <varname> with <value> and given <unit>

Parameters

- **varname** – Name of variable to be set as attribute name (string)
- **value** – value of variable (attribute) to be added.
- **unit** – string representation of the unit of the variable. Is only use for printing the contents of the WeatherDataContainer.

5.1.9 Signals defined

This module defines and describes the signals used by PCSE

Signals are used by PCSE to notify components of events such as sowing, harvest and termination. Events can be send by any SimulationObject through its *SimulationObject._send_signal()* method. Similarly, any SimulationObject can receive signals by registering a handler through the *SimulationObject._connect_signal()* method. Variables can be passed to the handler of the signal through positional or keyword arguments. However, it is highly discouraged to use positional arguments when sending signals in order to avoid conflicts between positional and keyword arguments.

An example can help to clarify how signals are used in PCSE but check also the documentation of the PyDispatcher package for more information:

```
import sys, os
import math
sys.path.append('/home/wit015/Sources/python/pcse/')
import datetime as dt

import pcse
from pcse.base import SimulationObject, VariableKiosk

mysignal = "My first signal"

class MySimObj(SimulationObject):
```

(continues on next page)

(continued from previous page)

```
    def initialize(self, day, kiosk):
        self._connect_signal(self.handle_mysignal, mysignal)

    def handle_mysignal(self, arg1, arg2):
        print "Value of arg1,2: %s, %s" % (arg1, arg2)

    def send_signal_with_exact_arguments(self):
        self._send_signal(signal=mysignal, arg2=math.pi, arg1=None)

    def send_signal_with_more_arguments(self):
        self._send_signal(signal=mysignal, arg2=math.pi, arg1=None,
                          extra_arg="extra")

    def send_signal_with_missing_arguments(self):
        self._send_signal(signal=mysignal, arg2=math.pi, extra_arg="extra")


# Create an instance of MySimObj
day = dt.date(2000,1,1)
k = VariableKiosk()
mysimobj = MySimObj(day, k)

# This sends exactly the right amount of keyword arguments
mysimobj.send_signal_with_exact_arguments()

# this sends an additional keyword argument 'extra_arg' which is ignored.
mysimobj.send_signal_with_more_arguments()

# this sends the signal with a missing 'arg1' keyword argument which the␣
↪handler
# expects and thus causes an error, raising a TypeError
try:
    mysimobj.send_signal_with_missing_arguments()
except TypeError, exc:
    print "TypeError occurred: %s" % exc
```

Saving this code as a file *test_signals.py* and importing it gives the following output:

```
>>> import test_signals
Value of arg1,2: None, 3.14159265359
Value of arg1,2: None, 3.14159265359
TypeError occurred: handle_mysignal() takes exactly 3 non-keyword␣
↪arguments (1 given)
```

Currently the following signals are used within PCSE with the following keywords.

CROP_START

Indicates that a new crop cycle will start:

self._send_signal(signal=signals.crop_start, day=<date>, crop_name=<string>, variety_name=<string>, crop_start_type=<string>, crop_end_type=<string>)

keyword arguments with *signals.crop_start*:

- day: Current date

- crop_name: a string identifying the crop
- variety_name: a string identifying the crop variety
- crop_start_type: either 'sowing' or 'emergence'
- crop_end_type: either 'maturity', 'harvest' or 'earliest'

CROP_FINISH

Indicates that the current crop cycle is finished:

```
self._send_signal(signal=signals.crop_finish, day=<date>,
                  finish_type=<string>, crop_delete=<True|False>)
```

keyword arguments with *signals.crop_finish*:

- day: Current date
- finish_type: string describing the reason for finishing the simulation, e.g. maturity, harvest, all leaves died, maximum duration reached, etc.
- crop_delete: Set to True when the CropSimulation object must be deleted from the system, for example for the implementation of crop rotations. Defaults to False.

TERMINATE

Indicates that the entire system should terminate (crop & soil water balance) and that terminal output should be collected:

```
self._send_signal(signal=signals.terminate)
```

No keyword arguments are defined for this signal

OUTPUT

Indicates that the model state should be saved for later use:

```
self._send_signal(signal=signals.output)
```

No keyword arguments are defined for this signal

SUMMARY_OUTPUT

Indicates that the model state should be saved for later use, SUMMARY_OUTPUT is only generated when a CROP_FINISH signal is received indicating that the crop simulation must finish:

```
self._send_signal(signal=signals.output)
```

No keyword arguments are defined for this signal

APPLY_NPK

Is used for application of Nitrate/Phosphate/Potassium (N/P/K) fertilizer:

```
self._send_signal(signal=signals.apply_npk, N_amount=<float>, P_amount=
↪<float>, K_amount=<float>,
                  N_recovery<float>, P_recovery=<float>, K_recovery=<float>
↪)
```

Keyword arguments with *signals.apply_npk*:

- N/P/K_amount: Amount of fertilizer in kg/ha applied on this day.
- N/P/K_recovery: Recovery fraction for the given type of fertilizer

IRRIGATE

Is used for sending irrigation events:

```
self._send_signal(signal=signals.irrigate, amount=<float>, efficiency=
↪<float>)
```

Keyword arguments with *signals.irrigate*:

- amount: Amount of irrigation in cm water applied on this day.
- efficiency: efficiency of irrigation, meaning that the total amount of water that is added to the soil reservoir equals amount * efficiency

MOWING

Is used for sending mowing events used by the LINGRA/LINGRA-N models:

```
self._send_signal(signal=signals.mowing, biomass_remaining=<float>)
```

Keyword arguments with *signals.mowing*:

- biomass_remaining: The amount of biomass remaining after mowing in kg/ha.

5.1.10 Utilities

The utilities section deals with tools for reading weather data and parameter values from files or databases.

Tools for reading input files

The file_input tools contain several classes for reading weather files, parameter files and agromanagement files.

class pcse.fileinput.**CABOFileReader**(*fname*)

Reads CABO files with model parameter definitions.

The parameter definitions of Wageningen crop models are generally written in the CABO format. This class reads the contents, parses the parameter names/values and returns them as a dictionary.

Parameters **fname** – parameter file to read and parse

Returns dictionary like object with parameter key/value pairs.

Note that this class does not yet fully support reading all features of CABO files. For example, the parsing of booleans, date/times and tabular parameters is not supported and will lead to errors.

The header of the CABO file (marked with ** at the first line) is read and can be retrieved by the get_header() method or just by a print on the returned dictionary.

Example

A parameter file 'parfile.cab' which looks like this:

```
** CROP DATA FILE for use with WOFOST Version 5.4, June 1992
**
** WHEAT, WINTER 102
** Regions: Ireland, central en southern UK (R72-R79),
**          Netherlands (not R47), northern Germany (R11-R14)
CRPNAM='Winter wheat 102, Ireland, N-U.K., Netherlands, N-Germany'
CROP_NO=99
TBASEM   = -10.0    ! lower threshold temp. for emergence [cel]
DTSMTB   =   0.00,    0.00,     ! daily increase in temp. sum
            30.00,   30.00,     ! as function of av. temp. [cel; cel␣
↪d]
            45.00,   30.00
** maximum and minimum concentrations of N, P, and K
** in storage organs        in vegetative organs [kg kg-1]
NMINSO   =   0.0110 ;       NMINVE   =   0.0030
```

Can be read with the following statements:

```
>>>fileparameters = CABOFileReader('parfile.cab')
>>>print fileparameters['CROP_NO']
99
>>>print fileparameters
** CROP DATA FILE for use with WOFOST Version 5.4, June 1992
**
** WHEAT, WINTER 102
** Regions: Ireland, central en southern UK (R72-R79),
**          Netherlands (not R47), northern Germany (R11-R14)
------------------------------------
TBASEM: -10.0 <type 'float'>
DTSMTB: [0.0, 0.0, 30.0, 30.0, 45.0, 30.0] <type 'list'>
NMINVE: 0.003 <type 'float'>
CROP_NO: 99 <type 'int'>
CRPNAM: Winter wheat 102, Ireland, N-U.K., Netherlands, N-Germany
↪<type 'str'>
NMINSO: 0.011 <type 'float'>
```

class pcse.fileinput.**CABOWeatherDataProvider**(*fname*, *fpath=None*, *ETmodel='PM'*, *distance=1*)

Reader for CABO weather files.

Parameters

- **fname** – root name of CABO weather files to read
- **fpath** – path where to find files, can be absolute or relative.
- **ETmodel** – "PM"|"P" for selecting penman-monteith or Penman method for reference evapotranspiration. Defaults to "PM".
- **distance** – maximum interpolation distance for meteorological variables, defaults to 1 day.

Returns callable like object with meteo records keyed on date.

The Wageningen crop models that are written in FORTRAN or FST often use the CABO weather system (http://edepot.wur.nl/43010) for storing and reading weather data. This class implements a reader for the CABO weather files and also implements additional features like interpolation of weather data in case of missing values, conversion of sunshine duration to global radiation estimates and calculation the reference evapotranspiration values for water, soil and plants (E0, ES0, ET0) using the Penman approach.

A difference with the old CABOWE system is that the python implementation will read and store all files (e.g. years) available for a certain station instead of loading a new file when crossing a year boundary.

Note: some conversions are done by the CABOWeaterDataProvider from the units in the CABO weather file for compatibility with WOFOST:

- vapour pressure from kPa to hPa
- radiation from kJ/m2/day to J/m2/day
- rain from mm/day to cm/day.
- all evaporation/transpiration rates are also returned in cm/day.

Example

The file 'nl1.003' provides weather data for the year 2003 for the station in Wageningen and can be found in the cabowe/ folder of the WOFOST model distribution. This file can be read using:

```
>>> weather_data = CABOWeatherDataProvider('nl1', fpath="./meteo/
↪cabowe")
>>> print weather_data(datetime.date(2003,7,26))
Weather data for 2003-07-26 (DAY)
IRRAD:  12701000.00  J/m2/day
 TMIN:        15.90   Celsius
 TMAX:        23.00   Celsius
  VAP:        16.50       hPa
 WIND:         3.00     m/sec
 RAIN:         0.12    cm/day
   E0:         0.36    cm/day
  ES0:         0.32    cm/day
  ET0:         0.31    cm/day
Latitude  (LAT):    51.97 degr.
Longitude (LON):     5.67 degr.
Elevation (ELEV):    7.0 m.
```

Alternatively the date in the print command above can be specified as string with format YYYYMMDD or YYYYDDD.

class pcse.fileinput.**PCSEFileReader**(*fname*)

Reader for parameter files in the PCSE format.

This class is a replacement for the *CABOFileReader*. The latter can be used for reading parameter files in the CABO format, however this format has rather severe limitations: it only supports string, integer, float and array parameters. There is no support for specifying parameters with dates for example (other then specifying them as a string).

The *PCSEFileReader* is a much more versatile tool for creating parameter files because it leverages the power of the python interpreter for processing parameter files through the *execfile* functionality in python. This means that anything that can be done in a python script can also be done in a PCSE parameter file.

Parameters **fname** – parameter file to read and parse

Returns dictionary object with parameter key/value pairs.

Example

Below is an example of a parameter file 'parfile.pcse'. Parameters can be defined the 'CABO'-way, but also advanced functionality can be used by importing modules, defining parameters as dates or numpy arrays and even applying function on arrays (in this case *np.sin*):

```
"""This is the header of my parameter file.

This file is derived from the following sources
* dummy file for demonstrating the PCSEFileReader
* contains examples how to leverage dates, arrays and functions, etc.
"""

import numpy as np
import datetime as dt

TSUM1 = 1100
TSUM2 = 900
DTSMTB = [ 0., 0.,
           5., 5.,
          20., 25.,
          30., 25.]
AMAXTB = np.sin(np.arange(12))
cropname = 'alfalfa'
CROP_START_DATE = dt.date(2010,5,14)
```

Can be read with the following statements:

```
>>>fileparameters = PCSEFileReader('parfile.pcse')
>>>print fileparameters['TSUM1']
1100
>>>print fileparameters['CROP_START_DATE']
2010-05-14
>>>print fileparameters
PCSE parameter file contents loaded from:
D:\UserData\pcse_examples\parfile.pw

This is the header of my parameter file.

This file is derived from the following sources
* dummy file for demonstrating the PCSEFileReader
* contains examples how to leverage dates, arrays and functions, etc.
DTSMTB: [0.0, 0.0, 5.0, 5.0, 20.0, 25.0, 30.0, 25.0] (<type 'list'>)
CROP_START_DATE: 2010-05-14 (<type 'datetime.date'>)
TSUM2: 900 (<type 'int'>)
cropname: alfalfa (<type 'str'>)
AMAXTB: [ 0.          0.84147098  0.90929743  0.14112001 -0.7568025
  -0.95892427 -0.2794155   0.6569866   0.98935825  0.41211849
  -0.54402111 -0.99999021] (<type 'numpy.ndarray'>)
TSUM1: 1100 (<type 'int'>)
```

class pcse.fileinput.**ExcelWeatherDataProvider**(*xls_fname*, *missing_snow_depth=None*, *force_reload=False*)

Reading weather data from an excel file (.xlsx only).

Parameters

- **xls_fname** – name of the Excel file to be read
- **mising_snow_depth** – the value that should use for missing SNOW_DEPTH values, the default value is *None*.
- **force_reload** – bypass the cache file, reload data from the .xlsx file and write a new cache file. Cache files are written under *$HOME/.pcse/meteo_cache*

For reading weather data from file, initially only the CABOWeatherDataProvider was available that reads its data from a text file in the CABO Weather format. Nevertheless, building CABO weather files is tedious as for each year a new file must constructed. Moreover it is rather error prone and formatting mistakes are easily leading to errors.

To simplify providing weather data to PCSE models, a new data provider was written that reads its data from simple excel files

The ExcelWeatherDataProvider assumes that records are complete and does not make an effort to interpolate data as this can be easily accomplished in Excel itself. Only SNOW_DEPTH is allowed to be missing as this parameter is usually not provided outside the winter season.

class pcse.fileinput.**CSVWeatherDataProvider**(*csv_fname*, *delimiter=', '*, *dateformat='%Y%m%d'*, *ETmodel='PM'*, *force_reload=False*)

Reading weather data from a CSV file.

Parameters

- **csv_fname** – name of the CSV file to be read
- **delimiter** – CSV delimiter
- **dateformat** – date format to be read. Default is '%Y%m%d'
- **ETmodel** – "PM"|"P" for selecting Penman-Monteith or Penman method for reference evapotranspiration. Default is 'PM'.
- **force_reload** – Ignore cache file and reload from the CSV file

The CSV file should have the following structure (sample), missing values should be added as 'NaN':

```
## Site Characteristics
Country     = 'Netherlands'
Station     = 'Wageningen, Haarweg'
Description = 'Observed data from Station Haarweg in Wageningen'
Source      = 'Meteorology and Air Quality Group, Wageningen␣
↪University'
Contact     = 'Peter Uithol'
Longitude = 5.67; Latitude = 51.97; Elevation = 7; AngstromA = 0.18;␣
↪AngstromB = 0.55; HasSunshine = False
## Daily weather observations (missing values are NaN)
DAY,IRRAD,TMIN,TMAX,VAP,WIND,RAIN,SNOWDEPTH
20040101,NaN,-0.7,1.1,0.55,3.6,0.5,NaN
20040102,3888,-7.5,0.9,0.44,3.1,0,NaN
20040103,2074,-6.8,-0.5,0.45,1.8,0,NaN
20040104,1814,-3.6,5.9,0.66,3.2,2.5,NaN
```

(continues on next page)

(continued from previous page)

```
20040105,1469,3,5.7,0.78,2.3,1.3,NaN
[...]

with
IRRAD in kJ/m2/day or hours
TMIN and TMAX in Celsius (°C)
VAP in kPa
WIND in m/sec
RAIN in mm
SNOWDEPTH in cm
```

For reading weather data from a file, initially the CABOWeatherDataProvider was available which read its data from text in the CABO weather format. Nevertheless, building CABO weather files is tedious as for each year a new file must constructed. Moreover it is rather error prone and formatting mistakes are easily leading to errors.

To simplify providing weather data to PCSE models, a new data provider has been derived from the ExcelWeatherDataProvider that reads its data from simple CSV files.

The CSVWeatherDataProvider assumes that records are complete and does not make an effort to interpolate data as this can be easily accomplished in a text editor. Only SNOWDEPTH is allowed to be missing as this parameter is usually not provided outside the winter season.

class pcse.fileinput.**YAMLAgroManagementReader**(*fname*)

Reads PCSE agromanagement files in the YAML format.

Parameters **fname** – filename of the agromanagement file. If fname is not provided as a absolute or relative path the file is assumed to be in the current working directory.

class pcse.fileinput.**YAMLCropDataProvider**(*fpath=None*, *repository=None*, *force_reload=False*)

A crop data provider for reading crop parameter sets stored in the YAML format.

param fpath full path to directory containing YAML files

param repository URL to repository containg YAML files. This url should be the *raw* content (e.g. starting with 'https://raw.githubusercontent.com')

param force_reload If set to True, the cache file is ignored and al parameters are reloaded (default False).

This crop data provider can read and store the parameter sets for multiple crops which is different from most other crop data providers that only can hold data for a single crop. This crop data providers is therefore suitable for running crop rotations with different crop types as the data provider can switch the active crop.

The most basic use is to call YAMLCropDataProvider with no parameters. It will than pull the crop parameters from my github repository at https://github.com/ajwdewit/WOFOST_crop_parameters:

```
>>> from pcse.fileinput import YAMLCropDataProvider
>>> p = YAMLCropDataProvider()
>>> print(p)
YAMLCropDataProvider - crop and variety not set: no activate crop␣
↪parameter set!
```

All crops and varieties have been loaded from the YAML file, however no activate crop has been set. Therefore, we need to activate a a particular crop and variety:

```
>>> p.set_active_crop('wheat', 'Winter_wheat_101')
>>> print(p)
YAMLCropDataProvider - current active crop 'wheat' with variety
↪'Winter_wheat_101'
Available crop parameters:
 {'DTSMTB': [0.0, 0.0, 30.0, 30.0, 45.0, 30.0], 'NLAI_NPK': 1.0,
↪'NRESIDLV': 0.004,
 'KCRIT_FR': 1.0, 'RDRLV_NPK': 0.05, 'TCPT': 10, 'DEPNR': 4.5,
↪'KMAXRT_FR': 0.5,
 ...
 ...
 'TSUM2': 1194, 'TSUM1': 543, 'TSUMEM': 120}
```

Additionally, it is possible to load YAML parameter files from your local file system:

```
>>> p = YAMLCropDataProvider(fpath=r"D:\UserData\sources\WOFOST_crop_
↪parameters")
>>> print(p)
YAMLCropDataProvider - crop and variety not set: no activate crop␣
↪parameter set!
```

Finally, it is possible to pull data from your fork of my github repository by specifying the URL to that repository:

```
>>> p = YAMLCropDataProvider(repository="https://raw.
↪githubusercontent.com/<your_account>/WOFOST_crop_parameters/master/
↪")
```

To increase performance of loading parameters, the YAMLCropDataProvider will create a cache file that can be restored much quicker compared to loading the YAML files. When reading YAML files from the local file system, care is taken to ensure that the cache file is re-created when updates to the local YAML are made. However, it should be stressed that this is *not* possible when parameters are retrieved from a URL and there is a risk that parameters are loaded from an outdated cache file. In that case use *force_reload=True* to force loading the parameters from the URL.

Simple or dummy data providers

This class of data providers can be used to provide parameter values in cases where separate files or a database is not needed or not practical. An example is the set of soil parameters for simulation of potential production conditions where the value of the parameters does not matter but nevertheless some values must be provided to the model.

class pcse.util.**DummySoilDataProvider**

This class is to provide some dummy soil parameters for potential production simulation.

Simulation of potential production levels is independent of the soil. Nevertheless, the model does not some parameter values. This data provider provides some hard coded parameter values for this situation.

class pcse.util.**WOFOST72SiteDataProvider**(***kwargs*)

Site data provider for WOFOST 7.2.

Site specific parameters for WOFOST 7.2 can be provided through this data provider as well as through a normal python dictionary. The sole purpose of implementing this data provider is that the site parameters for WOFOST are documented, checked and that sensible default values are given.

The following site specific parameter values can be set through this data provider:

```
- IFUNRN    Indicates whether non-infiltrating fraction of rain is a␣
↪function of storm size (1)
            or not (0). Default 0
- NOTINF    Maximum fraction of rain not-infiltrating into the soil␣
↪[0-1], default 0.
- SSMAX     Maximum depth of water that can be stored on the soil␣
↪surface [cm]
- SSI       Initial depth of water stored on the surface [cm]
- WAV       Initial amount of water in total soil profile [cm]
- SMLIM     Initial maximum moisture content in initial rooting depth␣
↪zone [0-1], default 0.4
```

Currently only the value for WAV is mandatory to specify.

class pcse.util.**WOFOST80SiteDataProvider**(***kwargs*)

Site data provider for WOFOST 8.0.

Site specific parameters for WOFOST 8.0 can be provided through this data provider as well as through a normal python dictionary. The sole purpose of implementing this data provider is that the site parameters for WOFOST are documented, checked and that sensible default values are given.

The following site specific parameter values can be set through this data provider:

```
- IFUNRN        Indicates whether non-infiltrating fraction of rain␣
↪is a function of
                storm size (1) or not (0). Default 0
- NOTINF        Maximum fraction of rain not-infiltrating into the␣
↪soil [0-1],
                default 0.
- SSMAX         Maximum depth of water that can be stored on the soil␣
↪surface [cm]
- SSI           Initial depth of water stored on the surface [cm]
- WAV           Initial amount of water in total soil profile [cm]
- SMLIM         Initial maximum moisture content in initial rooting␣
↪depth zone [0-1],
                default 0.4
- CO2           Atmospheric CO2 level (ppm), default 360.
- BG_N_SUPPLY   Background N supply through atmospheric deposition in␣
↪kg/ha/day. Can be
                in the order of 25 kg/ha/year in areas with high N␣
↪pollution. Default 0.0
- NSOILBASE     Base N amount available in the soil. This is often␣
↪estimated as the nutrient
                left over from the previous growth cycle (surplus␣
↪nutrients, crop residues
                or green manure).
- NSOILBASE_FR  Daily fraction of soil N coming available through␣
↪mineralization
- BG_P_SUPPLY   Background P supply in kg/ha/day. Usually this is␣
↪mainly through deposition
```

(continues on next page)

(continued from previous page)

```
                of dust and an order of magnitude smaller then N␣
↪deposition. Default 0.0
- PSOILBASE     Base P amount available in the soil.
- PSOILBASE_FR  Daily fraction of soil P coming available through␣
↪mineralization
- BG_K_SUPPLY   Background P supply in kg/ha/day. Default 0.0
- KSOILBASE     Base K amount available in the soil
- KSOILBASE_FR  Daily fraction of soil K coming available through␣
↪mineralization
- NAVAILI       Amount of N available in the pool at initialization␣
↪of the system [kg/ha]
- PAVAILI       Amount of P available in the pool at initialization␣
↪of the system [kg/ha]
- KAVAILI       Amount of K available in the pool at initialization␣
↪of the system [kg/ha]
```

Currently, the parameters for initial water availability (WAV) and initial availability of nutrients (NAVAILI, PAVAILI, KAVAILI) are mandatory to specify.

The database tools

The database tools contain functions and classes for retrieving agromanagement, parameter values and weather variables from database structures implemented for different versions of the European Crop Growth Monitoring System.

Note that the data providers only provide functionality for *reading* data, there are no tools here *writing* simulation results to a CGMS database. This was done on purpose as writing data can be a complex matter and it is our experience that this can be done more easily with dedicated database loader tools such as SQLLoader for ORACLE or the `load data infile` syntax of MySQL

The CGMS8 database

The CGMS8 tools are for reading data from a database structure that is used by CGMS executable version 9 and 10.

class `pcse.db.cgms8.`**`GridWeatherDataProvider`**(*engine*, *grid_no*, *start_date=None*, *end_date=None*, *recalc_ET=False*, *use_cache=True*)

Retrieves meteodata from the GRID_WEATHER table in a CGMS database.

Parameters

- **`metadata`** – SqlAlchemy metadata object providing DB access
- **`grid_no`** – CGMS Grid ID
- **`startdate`** – Retrieve meteo data starting with startdate (datetime.date object)
- **`enddate`** – Retrieve meteo data up to and including enddate (datetime.date object)

- **recalc_ET** – Set to True to force calculation of reference ET values. Mostly useful when values have not been calculated in the CGMS database.
- **use_cache** – Set to False to ignore read/writing a cache file.

Note that all meteodata is first retrieved from the DB and stored internally. Therefore, no DB connections are stored within the class instance. This makes that class instances can be pickled.

class pcse.db.cgms8.**SoilDataIterator**(*engine*, *grid_no*)

Soil data iterator for CGMS8.

The only difference is that in CGMS8 the table is called 'ELEMENTARY_MAPPING_UNIT' and in CGMS12 it is called 'EMU'

class pcse.db.cgms8.**CropDataProvider**(*engine*, *grid_no*, *crop_no*, *campaign_year*)

Retrieves the crop parameters for the given grid_no, crop_no and year from the tables CROP_CALENDAR, CROP_PARAMETER_VALUE and VARIETY_PARAMETER_VALUE.

Parameters

- **engine** – SqlAlchemy engine object providing DB access
- **grid_no** – Integer grid ID, maps to the GRID_NO column in the table
- **crop_no** – Integer crop ID, maps to the CROP_NO column in the table
- **campaign_year** – Integer campaign year, maps to the YEAR column in the table. The campaign year usually refers to the year of the harvest. Thus for crops crossing calendar years, the start_date can be in the previous year.

class pcse.db.cgms8.**STU_Suitability**(*engine*, *crop_no*)

Returns a set() of suitable STU's for given crop_no.

Parameters

- **engine** – SqlAlchemy engine object providing DB access
- **crop_no** – Integer crop ID, maps to the CROP_NO column in the table

class pcse.db.cgms8.**SiteDataProvider**(*engine*, *grid_no*, *crop_no*, *campaign_year*, *stu_no*)

Provides the site data from the tables INITIAL_SOIL_WATER and SITE.

Parameters

- **engine** – SqlAlchemy engine object providing DB access
- **grid_no** – Grid number (int)
- **crop_no** – Crop number (int)
- **campaign_year** – Campaign year (int)
- **stu_no** – soil typologic unit number (int)

Note that the parameter SSI (Initial surface storage) is set to zero

Moreover, the start date of the water balance is defined by the column GIVEN_STARTDATE_WATBAL. This value can be accessed as an attribute *start_date_waterbalance*.

The CGMS12 database

The CGMS12 tools are for reading data from a CGMS12 database structure that is used by CGMS executable version 11 and BioMA 2014.

Tools for reading weather data and timer, soil and site parameters from a CGMS12 compatible database.

class pcse.db.cgms12.**WeatherObsGridDataProvider**(*engine*, *grid_no*, *start_date=None*, *end_date=None*, *recalc_ET=False*, *recalc_TEMP=False*, *use_cache=True*)

Retrieves meteodata from the WEATHER_OBS_GRID table in a CGMS12 compatible database.

Parameters

- **engine** – SqlAlchemy engine object providing DB access
- **grid_no** – Grid number (int) to retrieve data for
- **start_date** – Retrieve meteo data starting with start_date (datetime.date object)
- **end_date** – Retrieve meteo data up to and including end_date (datetime.date object)
- **recalc_ET** – Set to True to force calculation of reference ET values. Mostly useful when values have not been calculated in the CGMS database.
- **recalc_TEMP** – Set to True to force calculation of daily average temperature (TEMP) from TMIN and TMAX: TEMP = (TMIN+TMAX)/2.

Note that all meteodata is first retrieved from the DB and stored internally. Therefore, no DB connections are stored within the class instance. This makes that class instances can be pickled.

If start_date and end_date are not provided then the entire time-series for the grid is retrieved.

class pcse.db.cgms12.**AgroManagementDataProvider**(*engine*, *grid_no*, *crop_no*, *campaign_year*, *campaign_start=None*)

Class for providing agromanagement data from the CROP_CALENDAR table in a CGMS12 database.

Parameters

- **engine** – SqlAlchemy engine object providing DB access
- **grid_no** – Integer grid ID, maps to the grid_no column in the table
- **crop_no** – Integer id of crop, maps to the crop_no column in the table
- **campaign_year** – Integer campaign year, maps to the YEAR column in the table. The campaign year usually refers to the year of the harvest. Thus for crops crossing calendar years, the start_date can be in the previous year.
- **campaign_start** – Optional keyword that can be used to define the start of the campaign. Note that by default the campaign_start_date is set equal

to the crop_start_date which means that the simulation starts when the crop starts. This default behaviour can be changed using this keyword. It can have multiple meanings:

- if a date object is passed, the campaign is assumed to start on this date.
- if an int/float is passed, the campaign_start_date is calculated as the crop_start_date minus the number of days provided by campaign_start.

For adjusting the campaign_start_Date, see also the *set_campaign_start_date(date)* method to update the campaign_start_date on an existing AgroManagementDataProvider.

set_campaign_start_date(*start_date*)
Updates the value for the campaign_start_date.

This is useful only when the INITIAL_SOIL_WATER table in CGMS12 defines a different campaign_start

class pcse.db.cgms12.**SoilDataIterator**(*engine*, *grid_no*)
Class for iterating over the different soils in a CGMS grid.

Instances of this class behave like a list, allowing to iterate over the soils in a CGMS grid. An example:

```
>>> soil_iterator = SoilDataIterator(engine, grid_no=15060)
>>> print(soildata)
Soil data for grid_no=15060 derived from oracle+cx_oracle://
↪cgms12eu:***@eurdas.world
  smu_no=9050131, area=625000000, stu_no=9000282 covering 50% of smu.
    Soil parameters {'SMLIM': 0.312, 'SMFCF': 0.312, 'SMW': 0.152,
↪'CRAIRC': 0.06,
                     'KSUB': 10.0, 'RDMSOL': 10.0, 'K0': 10.0, 'SOPE
↪': 10.0, 'SM0': 0.439}
  smu_no=9050131, area=625000000, stu_no=9000283 covering 50% of smu.
    Soil parameters {'SMLIM': 0.28325, 'SMFCF': 0.28325, 'SMW': 0.
↪12325, 'CRAIRC': 0.06,
                     'KSUB': 10.0, 'RDMSOL': 40.0, 'K0': 10.0, 'SOPE
↪': 10.0, 'SM0': 0.42075}
>>> for smu_no, area, stu_no, percentage, soil_par in soildata:
...     print(smu_no, area, stu_no, percentage)
...
(9050131, 625000000, 9000282, 50)
(9050131, 625000000, 9000283, 50)
```

class pcse.db.cgms12.**CropDataProvider**(*engine*, *grid_no*, *crop_no*, *campaign_year*)
Retrieves the crop parameters for the given grid_no, crop_no and year from the tables CROP_CALENDAR, CROP_PARAMETER_VALUE and VARIETY_PARAMETER_VALUE.

Parameters

- **engine** – SqlAlchemy engine object providing DB access
- **grid_no** – Integer grid ID, maps to the GRID_NO column in the table
- **crop_no** – Integer crop ID, maps to the CROP_NO column in the table
- **campaign_year** – Integer campaign year, maps to the YEAR column in the table. The campaign year usually refers to the year of the harvest. Thus for crops crossing calendar years, the start_date can be in the previous year.

class pcse.db.cgms12.**STU_Suitability**(*engine*, *crop_no*)
Returns a set() of suitable STU's for given crop_no.

Parameters

- **engine** – SqlAlchemy engine object providing DB access
- **crop_no** – Integer crop ID, maps to the CROP_NO column in the table

class pcse.db.cgms12.**SiteDataProvider**(*engine*, *grid_no*, *crop_no*, *campaign_year*, *stu_no*)
Provides the site data from the tables INITIAL_SOIL_WATER and SITE.

Parameters

- **engine** – SqlAlchemy engine object providing DB access
- **grid_no** – Grid number (int)
- **crop_no** – Crop number (int)
- **campaign_year** – Campaign year (int)
- **stu_no** – soil typologic unit number (int)

Note that the parameter SSI (Initial surface storage) is set to zero

Moreover, the start date of the water balance is defined by the column GIVEN_STARTDATE_WATBAL. This value can be accessed as an attribute *start_date_waterbalance*.

The CGMS14 database

The CGMS14 database is the database structure that is compatible with the 2015 BioMA implementation of WOFOST. Note that the CGMS14 database structure is considerably different from CGMS8 and CGMS12.

The NASA POWER database

class pcse.db.**NASAPowerWeatherDataProvider**(*latitude*, *longitude*, *force_update=False*, *ETmodel='PM'*)
WeatherDataProvider for using the NASA POWER database with PCSE

Parameters

- **latitude** – latitude to request weather data for
- **longitude** – longitude to request weather data for
- **force_update** – Set to True to force to request fresh data from POWER website.
- **ETmodel** – "PM"|"P" for selecting penman-monteith or Penman method for reference evapotranspiration. Defaults to "PM".

The NASA POWER database is a global database of daily weather data specifically designed for agrometeorological applications. The spatial resolution of the database is 0.5x0.5 degrees (as of 2018). It is derived from weather station observations in combination with satellite data for parameters like radiation.

The weather data is updated with a delay of about 3 months which makes the database unsuitable for real-time monitoring, nevertheless the POWER database is useful for many other studies and it is a major improvement compared to the monthly weather data that were used with WOFOST in the past.

For more information on the NASA POWER database see the documentation at: http://power.larc.nasa.gov/common/AgroclimatologyMethodology/Agro_Methodology_Content.html

The *NASAPowerWeatherDataProvider* retrieves the weather from the th NASA POWER API and does the necessary conversions to be compatible with PCSE. After the data has been retrieved and stored, the contents are dumped to a binary cache file. If another request is made for the same location, the cache file is loaded instead of a full request to the NASA Power server.

Cache files are used until they are older then 90 days. After 90 days the NASAPowerWeatherDataProvider will make a new request to obtain more recent data from the NASA POWER server. If this request fails it will fall back to the existing cache file. The update of the cache file can be forced by setting *force_update=True*.

Finally, note that any latitude/longitude within a 0.5x0.5 degrees grid box will yield the same weather data, e.g. there is no difference between lat/lon 5.3/52.1 and lat/lon 5.1/52.4. Nevertheless slight differences in PCSE simulations may occur due to small differences in day length.

Convenience routines

These routines are there for conveniently starting a WOFOST simulation for the demonstration and tutorials. They can serve as an example to build your own script but have no further relevance.

`pcse.start_wofost.`**`start_wofost`**(*grid=31031*, *crop=1*, *year=2000*, *mode='wlp'*, *dsn=None*)

Provides a convenient interface for starting a WOFOST instance

If started with no arguments, the routine will connnect to the demo database and initialize WOFOST for winter-wheat (cropno=1) in Spain (grid_no=31031) for the year 2000 in water-limited production (mode='wlp')

Parameters

- **`grid`** – grid number, defaults to 31031
- **`crop`** – crop number, defaults to 1 (winter-wheat in the demo database)
- **`year`** – year to start, defaults to 2000
- **`mode`** – production mode ('pp' or 'wlp'), defaults to 'wlp'
- **`dsn`** – PCSE DB as SQLAlchemy data source name defaults to *None* and in that case a connection to the demo database will be established.

example:

```
>>> import pcse
>>> wofsim = pcse.start_wofost(grid=31031, crop=1, year=2000,
...     mode='wlp')
>>>
>>> wofsim
```

(continues on next page)

(continued from previous page)

```
<pcse.models.Wofost71_WLP_FD at 0x35f2550>
>>> wofsim.run(days=300)
>>> wofsim.get_variable('tagp')
15261.752187075261
```

Miscelaneous utilities

Many miscelaneous for a variety of purposes such as the Arbitrary Function Generator (AfGen) for linear interpolation and functions for calculating Penman Penman/Monteith reference evapotranspiration, the Angstrom equation and astronomical calculations such as day length.

`pcse.util.`**`reference_ET`**(*DAY*, *LAT*, *ELEV*, *TMIN*, *TMAX*, *IRRAD*, *VAP*, *WIND*, *ANGSTA*, *ANGSTB*, *ETMODEL='PM'*, ***kwargs*)

Calculates reference evapotranspiration values E0, ES0 and ET0.

The open water (E0) and bare soil evapotranspiration (ES0) are calculated with the modified Penman approach, while the references canopy evapotranspiration is calculated with the modified Penman or the Penman-Monteith approach, the latter is the default.

Input variables:

```
DAY      -  Python datetime.date object                              -
LAT      -  Latitude of the site                                    ␣
↪degrees
ELEV     -  Elevation above sea level                                m
TMIN     -  Minimum temperature                                      C
TMAX     -  Maximum temperature                                      C
IRRAD    -  Daily shortwave radiation                              J m-
↪2 d-1
VAP      -  24 hour average vapour pressure                        hPa
WIND     -  24 hour average windspeed at 2 meter                   m/s
ANGSTA   -  Empirical constant in Angstrom formula                   -
ANGSTB   -  Empirical constant in Angstrom formula                   -
ETMODEL  -  Indicates if the canopy reference ET should             ␣
↪PM|P
            be calculated with the Penman-Monteith method
            (PM) or the modified Penman method (P)
```

Output is a tuple (E0, ES0, ET0):

```
E0       -  Penman potential evaporation from a free
            water surface [mm/d]
ES0      -  Penman potential evaporation from a moist
            bare soil surface [mm/d]
ET0      -  Penman or Penman-Monteith potential␣
↪evapotranspiration from a
            crop canopy [mm/d]
```

Note: The Penman-Monteith algorithm is valid only for a reference canopy and therefore it is not used to calculate the reference values for bare soil and open water (ES0, E0).

The background is that the Penman-Monteith model is basically a surface energy balance where the net solar radiation is partitioned over latent and sensible heat fluxes (ignoring the soil heat flux).

To estimate this partitioning, the PM method makes a connection between the surface temperature and the air temperature. However, the assumptions underlying the PM model are valid only when the surface where this partitioning takes place is the same for the latent and sensible heat fluxes.

For a crop canopy this assumption is valid because the leaves of the canopy form the surface where both latent heat flux (through stomata) and sensible heat flux (through leaf temperature) are partitioned. For a soil, this principle does not work because when the soil is drying the evaporation front will quickly disappear below the surface and therefore the assumption that the partitioning surface is the same does not hold anymore.

For water surfaces, the assumptions underlying PM do not hold because there is no direct relationship between the temperature of the water surface and the net incoming radiation as radiation is absorbed by the water column and the temperature of the water surface is co-determined by other factors (mixing, etc.). Only for a very shallow layer of water (1 cm) the PM methodology could be applied.

For bare soil and open water the Penman model is preferred. Although it partially suffers from the same problems, it is calibrated somewhat better for open water and bare soil based on its empirical wind function.

Finally, in crop simulation models the open water evaporation and bare soil evaporation only play a minor role (pre-sowing conditions and flooded rice at early stages), it is not worth investing much effort in improved estimates of reference value for E0 and ES0.

`pcse.util.`**`penman_monteith`**`(`*DAY*, *LAT*, *ELEV*, *TMIN*, *TMAX*, *AVRAD*, *VAP*, *WIND2*`)`

Calculates reference ET0 based on the Penman-Monteith model.

This routine calculates the potential evapotranspiration rate from a reference crop canopy (ET0) in mm/d. For these calculations the analysis by FAO is followed as laid down in the FAO publication Guidelines for computing crop water requirements - FAO Irrigation and drainage paper 56

Input variables:

```
DAY    -  Python datetime.date object                     -
LAT    -  Latitude of the site                         degrees
ELEV   - Elevation above sea level                        m
TMIN   - Minimum temperature                              C
TMAX   - Maximum temperature                              C
AVRAD  - Daily shortwave radiation                     J m-2 d-1
VAP    - 24 hour average vapour pressure                 hPa
WIND2  - 24 hour average windspeed at 2 meter            m/s
```

Output is:

ET0 - Penman-Monteith potential transpiration rate from a crop canopy [mm/d]

`pcse.util.`**`penman`**`(`*DAY*, *LAT*, *ELEV*, *TMIN*, *TMAX*, *AVRAD*, *VAP*, *WIND2*, *ANGSTA*, *ANGSTB*`)`

Calculates E0, ES0, ET0 based on the Penman model.

This routine calculates the potential evapo(transpi)ration rates from a free water surface (E0), a bare soil surface (ES0), and a crop canopy (ET0) in mm/d. For these calculations the analysis by Penman is followed (Frere and Popov, 1979;Penman, 1948, 1956, and 1963). Subroutines and functions called: ASTRO, LIMIT.

Input variables:

```
DAY     -  Python datetime.date object                                     ␣
↪    -
LAT     -  Latitude of the site                           degrees
ELEV    -  Elevation above sea level                         m
TMIN    -  Minimum temperature                               C
TMAX    -  Maximum temperature                               C
AVRAD   -  Daily shortwave radiation                      J m-2 d-1
VAP     -  24 hour average vapour pressure                  hPa
WIND2   -  24 hour average windspeed at 2 meter             m/s
ANGSTA  -  Empirical constant in Angstrom formula            -
ANGSTB  -  Empirical constant in Angstrom formula            -
```

Output is a tuple (E0,ES0,ET0):

```
E0      -  Penman potential evaporation from a free water surface [mm/
↪d]
ES0     -  Penman potential evaporation from a moist bare soil␣
↪surface [mm/d]
ET0     -  Penman potential transpiration from a crop canopy [mm/d]
```

pcse.util.**check_angstromAB**(*xA*, *xB*)

Routine checks validity of Angstrom coefficients.

This is the python version of the FORTRAN routine 'WSCAB' in 'weather.for'.

pcse.util.**wind10to2**(*wind10*)

Converts windspeed at 10m to windspeed at 2m using log. wind profile

pcse.util.**angstrom**(*day*, *latitude*, *ssd*, *cA*, *cB*)

Compute global radiation using the Angstrom equation.

Global radiation is derived from sunshine duration using the Angstrom equation: globrad = Angot * (cA + cB * (sunshine / daylength)

Parameters

- **day** – day of observation (date object)
- **latitude** – Latitude of the observation
- **ssd** – Observed sunshine duration
- **cA** – Angstrom A parameter
- **cB** – Angstrom B parameter

Returns the global radiation in J/m2/day

pcse.util.**doy**(*day*)

Converts a date or datetime object to day-of-year (Jan 1st = doy 1)

pcse.util.**limit**(*min*, *max*, *v*)

limits the range of v between min and max

pcse.util.**daylength**(*day*, *latitude*, *angle=-4*, *_cache={}*)

Calculates the daylength for a given day, altitude and base.

Parameters

- **day** – date/datetime object

- **latitude** – latitude of location
- **angle** – The photoperiodic daylength starts/ends when the sun is *angle* degrees under the horizon. Default is -4 degrees.

Derived from the WOFOST routine ASTRO.FOR and simplified to include only daylength calculation. Results are being cached for performance

`pcse.util.astro(day, latitude, radiation, _cache={})`
python version of ASTRO routine by Daniel van Kraalingen.

This subroutine calculates astronomic daylength, diurnal radiation characteristics such as the atmospheric transmission, diffuse radiation etc.

Parameters

- **day** – date/datetime object
- **latitude** – latitude of location
- **radiation** – daily global incoming radiation (J/m2/day)

output is a *namedtuple* in the following order and tags:

```
DAYL      Astronomical daylength (base = 0 degrees)     h
DAYLP     Astronomical daylength (base =-4 degrees)     h
SINLD     Seasonal offset of sine of solar height       -
COSLD     Amplitude of sine of solar height             -
DIFPP     Diffuse irradiation perpendicular to
          direction of light                         J m-2 s-1
ATMTR     Daily atmospheric transmission                -
DSINBE    Daily total of effective solar height         s
ANGOT     Angot radiation at top of atmosphere       J m-2 d-1
```

Authors: Daniel van Kraalingen Date : April 1991

Python version Author : Allard de Wit Date : January 2011

`pcse.util.merge_dict(d1, d2, overwrite=False)`
Merge contents of d1 and d2 and return the merged dictionary

Note:

- The dictionaries d1 and d2 are unaltered.
- If *overwrite=False* (default), a *RuntimeError* will be raised when duplicate keys exist, else any existing keys in d1 are silently overwritten by d2.

`class pcse.util.Afgen(tbl_xy, unit=None)`
Emulates the AFGEN function in WOFOST.

Parameters

- **tbl_xy** – List or array of XY value pairs describing the function the X values should be mononically increasing.
- **unit** – The interpolated values is returned with given unit assigned, defaults to None if Unum is not used.

Returns the interpolated value provided with the absicca value at which the interpolation should take place.

example:

```
>>> tbl_xy = [0,0,1,1,5,10]
>>> f =  Afgen(tbl_xy)
>>> f(0.5)
0.5
>>> f(1.5)
2.125
>>> f(5)
10.0
>>> f(6)
10.0
>>> f(-1)
0.0
```

class pcse.util.**ConfigurationLoader**(*config*)

Class for loading the model configuration from a PCSE configuration files

Parameters **config** – string given file name containing model configuration

pcse.util.**is_a_month**(*day*)

Returns True if the date is on the last day of a month.

pcse.util.**is_a_dekad**(*day*)

Returns True if the date is on a dekad boundary, i.e. the 10th, the 20th or the last day of each month

pcse.util.**is_a_week**(*day*, *weekday=0*)

Default weekday is Monday. Monday is 0 and Sunday is 6

pcse.util.**load_SQLite_dump_file**(*dump_file_name*, *SQLite_db_name*)

Build an SQLite database <SQLite_db_name> from dump file <dump_file_name>.

This document was generated on 2023-03-01/13:50.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

A

C

D

E

F

G

I

L

M

N

P

R

S

T

V

W

Y

Z