Code documentation

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.

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*) [source]

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*) [source]

Advances the system state with given number of days

`run_till`(*rday*) [source]

Runs the system until rday is reached.

`run_till_terminate`() [source]

Runs the system until a terminate signal is sent.

class `pcse.engine.Engine`(*parameterprovider*, *weatherdataprovider*, *agromanagement*, *config=None*) [source]

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() [source]

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() [source]

Returns the summary variables have have been stored during the simulation.

get_terminal_output() [source]

Returns the terminal output variables have have been stored during the simulation.

run(*days=1*) [source]

Advances the system state with given number of days

run_till(*rday*) [source]

Runs the system until rday is reached.

run_till_terminate() [source]

Runs the system until a terminate signal is sent.

set_variable(*varname*, *value*) [source]

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*) [source]

ALCEPAS Onion growth model.

---

class **pcse.models.FAO_WRSI**(*parameterprovider*, *weatherdataprovider*, *agromanagement*) [source]

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*) [source]

---

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

---

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

---

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

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*) [source]

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*) [source]

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*) [source]

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*) [source]

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*) [source]

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*) [source]

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

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*) [source]

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}
    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:
- 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
        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*) [source]

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*) [source]

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()` [source]

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() [source]

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*) [source]

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*) [source]

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
    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() [source]

Returns the last date for which a timed event is given

validate(*campaign_start_date*, *next_campaign_start_date*) [source]

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*) [source]

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.

The Timer

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

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*) [source]

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

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*) [source]

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*) [source]

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

State variables:

Name	Description
SM	Volumetric moisture content in root zone
SS	Surface storage (layer of water on surface)
SSI	Initial urface storage
W	Amount of water in root zone
WI	Initial amount of water in the root zone
WLOW	Amount of water in the subsoil (between current rooting depth and maximum rootable depth)
WLOWI	Initial amount of water in the subsoil
WWLOW	Total amount of water in the soil profile WWLOW = WLOW + W
WTRAT	Total water lost as transpiration as calculated by the water balance. This can be different from
EVST	Total evaporation from the soil surface
EVWT	Total evaporation from a water surface
TSR	Total surface runoff
RAINT	Total amount of rainfall (eff + non-eff)
WDRT	Amount of water added to root zone by increase of root growth
TOTINF	Total amount of infiltration
TOTIRR	Total amount of effective irrigation
PERCT	Total amount of water percolating from rooted zone to subsoil
LOSST	Total amount of water lost to deeper soil
DSOS	Days since oxygen stress, accumulates the number of consecutive days of oxygen stress
WBALRT	Checksum for root zone waterbalance. Will be calculated within *finalize()*, abs(WBALRT) > 0.0
WBALTT	Checksum for total waterbalance. Will be calculated within *finalize()*, abs(WBALTT) > 0.0001 w

Rate variables:

External dependencies:

Name	Description	Provided by
TRA	Crop transpiration rate	Evapotranspiration
EVSMX	Maximum evaporation rate from a soil surface below the crop canopy	Evapotranspiration
EVWMX	Maximum evaporation rate from a water surface below the crop canopy	Evapotranspiration
RD	Rooting depth	Root_dynamics

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*) [source]

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	T
TMINACCU1	Upper critical minimum temperature for snow accumulation.	S
TMINACCU2	Lower critical minimum temperature for snow accumulation	S
TMINCRIT	Critical minimum temperature for snow melt	S
TMAXCRIT	Critical maximum temperature for snow melt	S
RMELT	Melting rate per day per degree Celcius above the critical minimum temperature.	S
SCTHRESHOLD	Snow water equivalent above which the sublimation is taken into account.	S
SNOWDENSITY	Density of snow	S
SWEDEPTHI	Initial depth of layer of water present as snow on the soil surface	S

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}

Crop simulation processes for WOFOST

Phenology

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

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
TSUMEM	Temperature sum from sowing to emergence
TBASEM	Base temperature for emergence
TEFFMX	Maximum effective temperature for emergence
TSUM1	Temperature sum from emergence to anthesis
TSUM2	Temperature sum from anthesis to maturity
IDSL	Switch for phenological development options temperature only (IDSL=0), including daylength (
DLO	Optimal daylength for phenological development
DLC	Critical daylength for phenological development
DVSI	Initial development stage at emergence. Usually this is zero, but it can be higher for crops that
DVSEND	Final development stage
DTSMTB	Daily increase in temperature sum as a function of daily mean temperature.

State variables

Name	Description

DVS	Development stage
TSUM	Temperature sum
TSUME	Temperature sum for emergence
DOS	Day of sowing
DOE	Day of emergence
DOA	Day of Anthesis
DOM	Day of maturity
DOH	Day of harvest
STAGE	Current phenological stage, can take the folowing values: *emerging*|*vegetative*|*reproductive*|*mat*

Rate variables

Name	Description	Pbl	Unit
DTSUME	Increase in temperature sum for emergence	N	°C
DTSUM	Increase in temperature sum for anthesis or maturity	N	°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*) [source]

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	Un
VERNSAT	Saturated vernalisation requirements	SCr	da
VERNBASE	Base vernalisation requirements	SCr	da
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
DVS	Development Stage Used only to determine if the critical development stage for vernalisation (VEI

Partitioning

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

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*) [source]

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 $ha^{-1}hr^{-}$
EFFTB	Light use effic. single leaf as a function of daily mean temperature	TCr	kg $ha^{-1}hr^{-}$
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*) [source]

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
Q10	Relative increase in maintenance repiration rate with each 10 degrees increase in temperature
RMR	Relative maintenance respiration rate for roots
RMS	Relative maintenance respiration rate for stems
RML	Relative maintenance respiration rate for leaves
RMO	Relative maintenance respiration rate for storage organs

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*) [source]

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*) [source]

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*) [source]

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
RGRLAI	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}
			-1

TWLV	Dry weight of total leaves (living + dead)	Y	kg ha

Rate variables

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

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}da
RF_FROST	Reduction factor frost kill	FROSTOL	•

Root dynamics

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

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
RD	Current rooting depth
RDM	Maximum attainable rooting depth at the minimum of the soil and crop maximum rooting depth
WRT	Weight of living roots
DWRT	Weight of dead roots
TWRT	Total weight of roots

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*) [source]

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*) [source]

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*) [source]

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
NMAXLV_TB	Maximum N concentration in leaves as function of dvs
PMAXLV_TB	As for P
KMAXLV_TB	As for K
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
PRESIDLV	Residual P fraction in leaves
KRESIDLV	Residual K fraction in leaves
NRESIDRT	Residual N fraction in roots
PRESIDRT	Residual P fraction in roots
KRESIDRT	Residual K fraction in roots
NRESIDST	Residual N fraction in stems
PRESIDST	Residual P fraction in stems
KRESIDST	Residual K fraction in stems

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*) [source]

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
NMAXLV_TB	Maximum N concentration in leaves as function of DVS
PMAXLV_TB	As for P
KMAXLV_TB	As for K
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
PMAXSO	As for P
KMAXSO	As for K
NCRIT_FR	Critical N concentration as fraction of maximum N concentration for vegetative plant or
PCRIT_FR	As for P
KCRIT_FR	As for K
TCNT	Time coefficient for N translation to storage organs
TCPT	As for P
TCKT	As for K
NFIX_FR	fraction of crop nitrogen uptake by biological fixation
RNUPTAKEMAX	Maximum rate of N uptake
RPUPTAKEMAX	Maximum rate of P uptake
RKUPTAKEMAX	Maximum rate of K uptake

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}
			-1

KdemandRT	K Demand in living roots	N	kg K ha
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
DVS	Crop development stage	DVS_Phenology
TRA	Crop transpiration	Evapotranspiration
TRAMX	Potential crop transpiration	Evapotranspiration
NAVAIL	Total available N from soil	NPK_Soil_Dynamics
PAVAIL	Total available P from soil	NPK_Soil_Dynamics
KAVAIL	Total available K from soil	NPK_Soil_Dynamics
Ntranslocatable	Translocatable amount of N from stems, Leaves and roots	NPK_Translocation
Ptranslocatable	As for P	NPK_Translocation
Ktranslocatable	As for K	NPK_Translocation

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

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
NMAXLV_TB	Maximum N concentration in leaves as function of DVS
PMAXLV_TB	As for P
KMAXLV_TB	As for K
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 organ

PCRIT_FR	As for P
KCRIT_FR	As for K
NRESIDLV	Residual N fraction in leaves
PRESIDLV	Residual P fraction in leaves
KRESIDLV	Residual K fraction in leaves
NRESIDST	Residual N fraction in stems
PRESIDST	Residual P fraction in stems
KRESIDST	Residual K fraction in stems
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
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*) [source]

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
NRESIDLV	Residual N fraction in leaves
PRESIDLV	Residual P fraction in leaves
KRESIDLV	Residual K fraction in leaves

NRESIDST	Residual N fraction in stems
PRESIDST	Residual P fraction in stems
KRESIDST	Residual K fraction in stems
NPK_TRANSLRT_FR	NPK translocation from roots as a fraction of resp. total NPK amounts translocate

State variables

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

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*) [source]

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
IDSL	Switch for phenological development options temperature only (IDSL=0), including
LT50C	Critical LT50 defined as the lowest LT50 value that the wheat cultivar can obtain
FROSTOL_H	Hardening coefficient
FROSTOL_D	Dehardening coefficient
FROSTOL_S	Low temperature stress coefficient
FROSTOL_R	Respiration stress coefficient
FROSTOL_SDBASE	Minimum snow depth for respiration stress
FROSTOL_SDMAX	Snow depth with maximum respiration stress. Larger snow depth does not increase
FROSTOL_KILLCF	Steepness coefficient for logistic kill function.
ISNOWSRC	Use prescribed snow depth from driving variables (0) or modelled snow depth throu

State variables

Name	Description	Pbl	Unit
LT50T	Current LT50 value	N	°C

LT50I	Initial LT50 value of unhardened crop	N	°C
IDFST	Total number of days with frost stress	N	-

Rate variables

Name	Description
RH	Rate of hardening
RDH_TEMP	Rate of dehardening due to temperature
RDH_RESP	Rate of dehardening due to respiration stress
RDH_TSTR	Rate of dehardening due to temperature stress
IDFS	Frost stress, yes (1) or no (0). Frost stress is defined as: RF_FROST > 0
RF_FROST	Reduction factor on leave biomass as a function of min. crown temperature and LT50T: ra
RF_FROST_T	Total frost kill through the growing season is computed as the multiplication of the daily fr

External dependencies:

Name	Description
TEMP_CROWN	Daily average crown temperature derived from calling the crown_temperature module.
TMIN_CROWN	Daily minimum crown temperature derived from calling the crown_temperature module.
ISVERNALISED	Boolean reflecting the vernalisation state of the crop.

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*) [source]

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):

$$\mathrm{RSD} = \min(15, \mathrm{SD})/15$$

and

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

and

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

and

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

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
ISNOWSRC	Use prescribed snow depth from driving variables (0) or modelled snow depth through th
CROWNTMPA	A parameter in equation for crown temperature
CROWNTMPB	B parameter in equation for crown temperature

Rate variables

Name	Description	Pbl	Unit
TEMP_CROWN	Daily average crown temperature	N	°C
TMIN_CROWN	Daily minimum crown temperature	N	°C
TMAX_CROWN	Daily maximum crown temperature	N	°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
SNOWDEPTH	Depth of snow cover.	Prescibed by driving variables or simulated by snow cover mod

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*) [source]

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
LAIinit	Initial leaf area index
TillerNumberinit	Initial number of tillers
WeightREinit	Initial weight of reserves
WeightRTinit	Initial weight of roots
LAIcrit	Critical LAI for death due to self-shading
RDRbase	Background relative death rate for roots
RDRShading	Max relative death rate of leaves due to self-shading
RDRdrought	Max relative death rate of leaves due to drought stress
SLA	Specific leaf area
TempBase	Base temperature for photosynthesis and development
PartitioningRootsTB	Partitioning fraction to roots as a function of the reduction factor for transpiration (RF
TSUMmax	Temperature sum to max development stage

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*) [source]

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
KDIFTB	Extinction coefficient for diffuse visible as function of DVS.
CO2A	Atmospheric CO2 concentration
LUEreductionSoilTempTB	Reduction function for light use efficiency as a function of soil temperature.
LUEreductionRadiationTB	Reduction function for light use efficiency as a function of radiation level.
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*) [source]

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
TempBase	Base temperature for leaf development and grass phenology
LAICrit	Cricical leaf area beyond which leaf death due to self-shading occurs
SiteFillingMax	Maximum site filling for new buds
SLA	Specific leaf area
TSUMmax	Temperature sum to max development stage
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
dTillerNumber	Change in tiller number due to the radiation level
dLeafLengthPot	Potential change in leaf length. Later on the actual change in leaf length will be compute

LAIGrowthSink	Growth of LAI based on sink-limited growth rate.

Signals send or handled

None

External dependencies:

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

Nitrogen dynamics

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

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
RNuptakeLV	Rate of N uptake in leaves
RNuptakeRT	Rate of N uptake in roots

RNuptake	Total rate of N uptake
NdemandLV	Ndemand of leaves based on current growth rate and deficienties from previous time steps
NdemandRT	N demand of roots, idem as leaves
Ndemand	Total N demand (leaves + roots)

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*) [source]

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*) [source]

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

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** [source]

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
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*) [source]

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() [source]

flush the values of all published rate variable from the kiosk.

flush_states() [source]

flush the values of all state variable from the kiosk.

register_variable(*oid*, *varname*, *type*, *publish=False*) [source]

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*) [source]

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*) [source]

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*) [source]

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() [source]

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*) [source]

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() [source]

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

class **pcse.base.ParamTemplate**(*parvalues*) [source]

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** [source]

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*) [source]

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() [source]

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*) [source]

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*) [source]

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.

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):

    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.

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*) [source]

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*) [source]

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*) [source]

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*)** [source]

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*) [source]

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
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*) [source]

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*) [source]

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_parameter
```

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 [source]

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) [source]

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) [source]

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
                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*) [source]

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*) [source]

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*) [source]

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*) [source]

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*) [source]

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*) [source]

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*) [source]

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*) [source]

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*) [source]

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*) [source]

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*) [source]

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*) [source]

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'*) [source]

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*) [source]

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
<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*) [source]

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*) [source]

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*) [source]

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*) [source]

Routine checks validity of Angstrom coefficients.

This is the python version of the FORTRAN routine ‘WSCAB’ in ‘weather.for’.

pcse.util.wind10to2(*wind10*) [source]

Converts windspeed at 10m to windspeed at 2m using log. wind profile

pcse.util.angstrom(*day*, *latitude*, *ssd*, *cA*, *cB*) [source]

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*) [source]

Converts a date or datetime object to day-of-year (Jan 1st = doy 1)

pcse.util.limit(*min*, *max*, *v*) [source]

limits the range of v between min and max

pcse.util.daylength(*day*, *latitude*, *angle=-4*, *_cache={}*) [source]

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={}*) [source]

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*) [source]

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*) [source]

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*) [source]

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*) [source]

Returns True if the date is on the last day of a month.

pcse.util.is_a_dekad(*day*) [source]

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*) [source]

Default weekday is Monday. Monday is 0 and Sunday is 6

pcse.util.load_SQLite_dump_file(*dump_file_name*, *SQLite_db_name*) [source]

Build an SQLite database <SQLite_db_name> from dump file <dump_file_name>.

Previous