



Australia's National
Science Agency

AusFarm Tutorial

CSIRO Agriculture and Food

Workshop training manual



Version 2.0

Feb 2020

Enquiries should be addressed to:

Neville Herrmann

CSIRO Agriculture & Food

GPO Box 1700

Canberra ACT 2601

grazplan@csiro.au

Copyright and Disclaimer

© 2020 CSIRO, To the extent permitted by law, all rights are reserved and no part of this publication covered by copyright may be reproduced or copied in any form or by any means except with the written permission of CSIRO.

Important Disclaimer

CSIRO advises that the information contained in this publication comprises general statements based on scientific research. The reader is advised and needs to be aware that such information may be incomplete or unable to be used in any specific situation. No reliance or actions must therefore be made on that information without seeking prior expert professional, scientific and technical advice. To the extent permitted by law, CSIRO (including its employees and consultants) excludes all liability to any person for any consequences, including but not limited to all losses, damages, costs, expenses and any other compensation, arising directly or indirectly from using this publication (in part or in whole) and any information or material contained in it.

Contents

1.	Introduction	5
1.1	AusFarm.....	5
2.	Using AusFarm	7
2.1	The Simulation Process	8
3.	Components, Modules and Systems	10
3.1	Components	10
3.2	Modules.....	10
3.3	Systems.....	11
4.	Getting Started	12
4.1	Installing AusFarm	12
4.2	Running AusFarm	12
4.3	The Main Window	13
4.4	A Tour of the Simulation Window	13
4.5	Upper pane	14
4.6	Left-hand pane	15
4.7	Right-hand pane	15
4.8	Running a Simulation.....	17
4.9	The Results Window	19
4.10	Generating a chart.....	19
4.11	Generating a table	21
4.12	Data treatments	22
5.	Configuring a new simulation	25
5.1	Configuring and Initialising Modules.....	25
5.2	The Notes and Logging tabs	28
5.3	Sequencing the simulation	29
5.4	Selecting and Storing Outputs.....	31
5.5	Comparing initial values of components.....	36
6.	Specifying Management	39

6.1	Variables	39
6.2	Events	41
6.3	Discovering the Variables and Events for Modules	43
7.	Contents of Manager Scripts	46
7.1	Time specifiers.....	46
7.2	Rules	47
7.3	Definition statements.....	59
7.4	Examples of complete statements	61
7.5	Management Events Summary	62
8.	Writing management scripts	64
8.1	Using the script editor	64
9.	Introduction to Livestock Management and the Stock component	68
9.1	Stock component.....	68
10.	Simulation Analyses.....	70
10.1	Setting up analyses.....	70
10.2	Using Generic modules as factors	73
11.	Configuring Reports.....	76
11.1	Report Variables	77
12.	Using Repositories	78
12.1	Getting module data from a repository	78
12.2	Copying module data to a repository.....	79
12.3	Copying module data from simulation to simulation	79
13.	Using APSoil soil data.....	80

1. Introduction

1.1 AusFarm

AusFarm is a software tool that allows problems to be analysed with simulation models of physical and biological systems. AusFarm is highly generic, but it has been built primarily to assist decision-making in agricultural enterprises at scales ranging from paddocks to whole landscapes.

With the complexity of farming systems having multiple enterprises increasing there is an opportunity for modelling tools to represent them and provide insights into how these systems can be adapted to improve efficiency and reduce environmental impact. AusFarm can model cropping systems and can also model grazing systems interacting with cropping enterprises.

Simulations in AusFarm have the following features:

Modularity

Instead of a single program that contains the entire "AusFarm model", simulations in AusFarm are built up from smaller elements known as components that describe parts of a biophysical system.

For example, the standard AusFarm distribution includes one component that handles weather data, and another that describes the dynamics of grazing ruminants.

Separating the parts of a model that are closely related into sub-models has advantages during model development, for software maintenance, and in the deployment of up-to-date versions of models. It also means that models from groups other than CSIRO Plant Industry can be used within AusFarm.

Configurability

Once a simulation model is decomposed into components, it becomes natural to arrange the sub-models in configurations that reflect a range of different real-world situations.

The practical advantages are that an AusFarm user can put together the simplest model required to analyse a given problem and can use multiple copies of a model component within a simulation (for example to represent the flows of soil water in each of several paddocks).

Interchangeability

Modular construction also permits substitution of one representation of a process by another, depending on the

needs of the user. This can be useful in comparing different representations of a process, or in configuring a simulation for efficient execution.

Representation of both continuous and discrete processes

Many processes in agricultural systems are fundamentally continuous in nature. Others, particularly management interventions, involve sharp changes in the state of the system, which may be thought of as instantaneous events. AusFarm can accommodate both continuous and discontinuous processes. Farm system management tasks can be customised using a Manager component which allows the use of detailed scripts containing rules to drive much of the behaviour of the simulation.

Hierarchical structure

Ecological and hence agronomic systems contain too many entities to be solved analytically by differential-equation techniques, and they have too few entities to be treated as statistical assemblages. Current ecological theory suggests that the best way to analyse this kind of complexity is to take advantage of the hierarchical organization in these systems that arises from differences in the rates of different processes. Simulations in AusFarm can be configured to capture such hierarchical structure.

Advanced reporting features

AusFarm allows the output of results in several formats including text and databases. Built into AusFarm is a reporting system that has flexibility to accommodate many variations in simulation structure and allows easy comparison of treatments in a multidimensional simulation experiment called an *Analysis*.

AusFarm has been developed by CSIRO Agriculture and Food. The standard distribution includes a set of models, also developed by CSIRO Agriculture and Food, that enable simulations of grazing enterprises located in temperate southern Australia.

2. Using AusFarm

Typically, use of AusFarm will follow these steps:

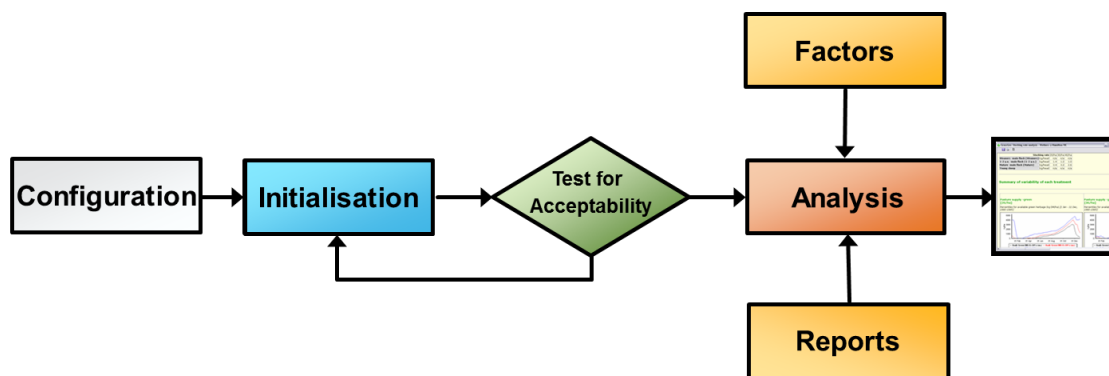
- Determine what question AusFarm is to answer, and therefore what information a simulation (or simulations) with AusFarm needs to generate. *This is the most important step in the process.*
- Construct a simulation that represents the biophysical and management system under consideration.
 - (a) Create a simulation window or open an existing simulation file that is suited to the question at hand.
 - (b) Configure the simulation to include representations of all the processes that are important in understanding the biophysical problem.
 - (c) Specify the initial conditions of the simulation: the locality for which weather data are to be used, the attributes of soils, plants and animals on the first day of the simulation, costs and prices, etc.
 - (d) If necessary (almost always), describe the management of the biophysical system by writing a *management script*.
 - (e) Select the outputs to be stored for later viewing.
- Run the simulation.
- Extract the results of the simulation as tables or charts and using these outputs to help answer the question at hand. AusFarm contains powerful facilities for summarizing simulation outputs. It may also be useful to export information from AusFarm to another program such as a spreadsheet or statistics package for further analysis.

Answering a question will often require several slightly different simulations. A set of simulations in AusFarm can take the form of a structured “simulation experiment” in which one or more inputs to the simulation are varied systematically. Such simulation experiments can require large numbers of simulation runs. AusFarm incorporates a feature called a *Simulation Analysis* where configuring similar simulations and reviewing the results comparatively is simplified and has enormous flexibility.

Note: Beyond a certain level of complexity, it becomes almost inevitable that a simulation will not work as intended the first time it is run. It is the user's responsibility to store and examine outputs from the simulations to ensure that their structure, initial values and management script are working as intended. Various *logging* options are available to assist in this process.

2.1 The Simulation Process

The process for testing a model can follow the flow shown below. AusFarm is designed to integrate all these steps within the application. This means that simulations can be tested efficiently and opportunities for erroneous configuration are reduced.



Configuration

- The system design is formulated. The model structure that includes the sub-models is built in the model tree and the management strategy is coded in Manager scripts.

Initialisation

- This involves setting initial values for the sub-models.

Test for Acceptability

- Executing the time-steps for the simulation and examining the results from a test run. This is a critical phase that should be undertaken carefully. When further refinement is necessary the Initialisation step is undertaken again. The acceptability step is often the main task with the remaining steps being optional.

Factors

- Choosing sub-model initial values that can be adjusted between simulation runs. This will determine the number of treatments that will be processed in the Analysis. Factors in AusFarm are represented by one of the sub-models or they can be a system of sub-models.

Reports

- Designing a report that can include several charts, tables, or text. The report will give some insight into the effects of varying the values of the factors chosen previously.

Analysis

- This is the task of testing each simulation treatment. The multi-dimensional experiment is processed, and results are stored for presentation using the previously designed report template.

Results

- The simulation results are formatted using the report design and shown in a HTML document.

3. Components, Modules and Systems

3.1 Components

In AusFarm, model logic is contained within entities called *components*. Each component corresponds to a *sub-model*, i.e. a set of variables, equations and events that are inter-related. For example, the standard distribution of AusFarm contains a Soil Water component that contains the logic for a soil water budget, and a Stock component that contains the GRAZPLAN ruminant biology model.

Some components can be thought of as "utilities" - they perform tasks that are not part of the model in a narrow sense (i.e. as a mathematical entity) but are vital to making the model useful. An example of a utility is the Output component, which allows the user to store the results of simulations for later interpretation.

Each component is implemented as a Windows dynamic link library (DLL). Before a component can be used in a simulation, it must be installed on the *component palette*.

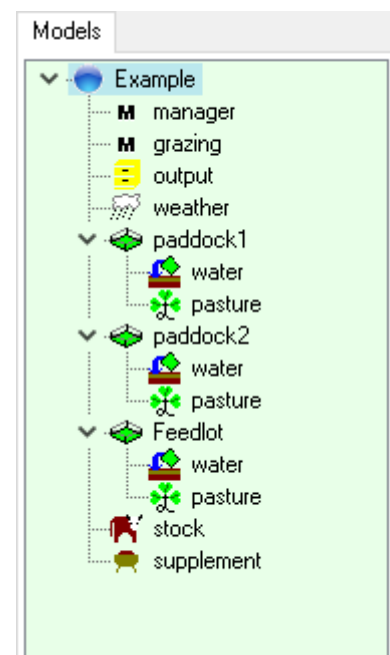


3.2 Modules

A *module* is a specific instance of a component within a specific simulation. A simulation may contain several modules that are instances of the same component. For example, in the simulation structure at right, there are two modules that are instances of the Soil Water component, and six that are instances of the Pasture component.

A module can only exist as part of a simulation. The set of modules in a simulation is defined by adding them to the simulation tree in the Models tab of the simulation window. This process is known as *configuring* the simulation.

From the point of view of an AusFarm user, each module is made up of the following elements:



Name

A module's name is supplied by the user via the Models tab. It is used to refer to the module and its variables or events, for example when writing the management script. Each module has a short name

(e.g. "pasture") and a *fully-qualified name* that is defined by the systems to which it belongs (e.g. "paddock2.pasture "). The short name need not be unique, but the fully qualified name must be unique.

Initialisation Variables	contain the values that must be known to set up a module.
Driving Variables	contain values that are not part of the module but must be known to calculate its equations.
Output Variables	contain values that can be stored for later viewing and analysis, and/or used in management scripts to control the course of the simulation. (Initialisation variables are usually also available as output variables).
Sequenced Events	contain the <i>rate equations</i> of the module, i.e. the main model logic. Each sequenced event is computed once per time step. AusFarm handles the setting up of sequenced events automatically.
Management Events	can be invoked as part of management scripts to change the module's state in some way.

3.3 Systems

The modules in each simulation are arranged, not in a simple list, but in a tree. Each sub-tree of this structure is known as a *system*, and the module at the "root" of a sub-tree is called a *system module*. The simulation structure above has three systems: the system made up of **Paddock1** and its child modules, the system made up of **Paddock2** and its child modules, and the entire simulation. Only certain components can act as system modules. In the default AusFarm distribution, only the Paddock component can be used to form systems.

4. Getting Started

The software tutorial section of this manual begins here. Follow the steps along with the installed software to become familiar with how the user interface works. The ➡ symbol is used to instruct you to follow the steps on your computer with the AusFarm software.

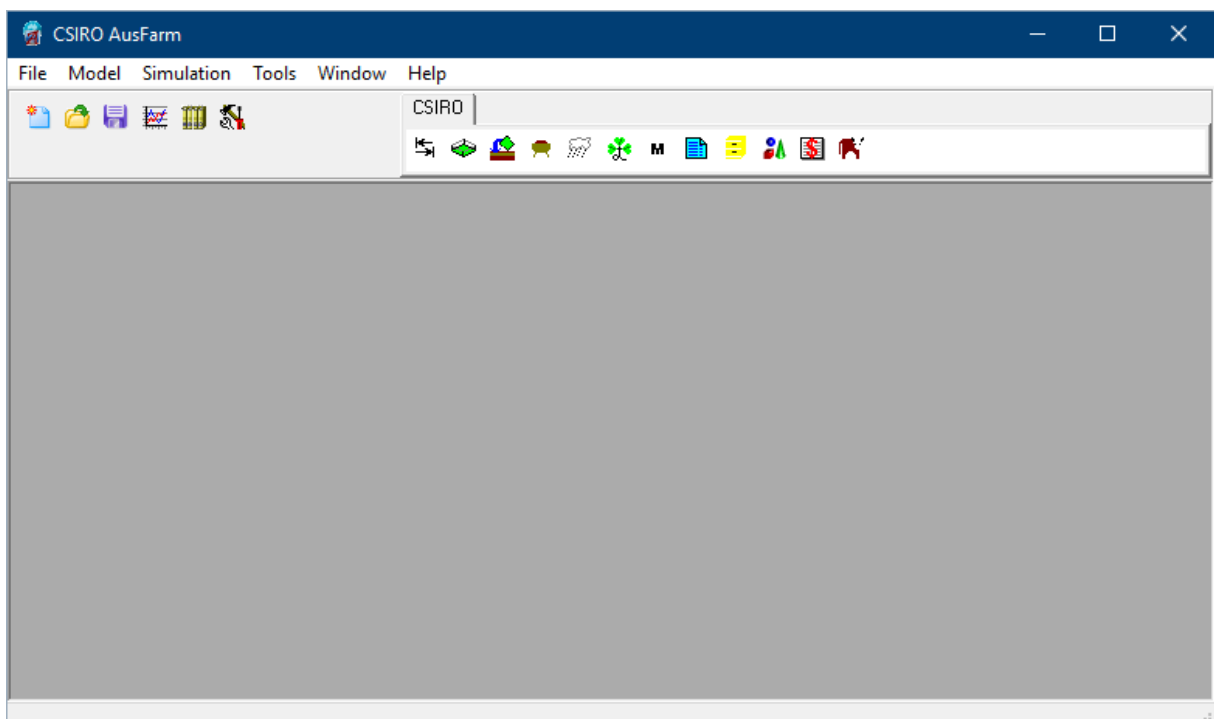
4.1 Installing AusFarm

AusFarm is currently a Microsoft® Windows 32-bit native code program that can be installed and run on 32-bit and 64-bit versions of Windows.

- ➡ Start by installing the AusFarm software. Run the **setupaf.exe** program and follow the prompts. Some sample weather data will be installed that will allow the running of an example simulation.

4.2 Running AusFarm

- ➡ Run the AusFarm program. The main window will appear:



The main window has a menu and toolbar at the top and a *client area* where other windows for simulations, results selection and reporting reside. At the right-hand side of the toolbar is the Component Palette, which is used when configuring simulations.

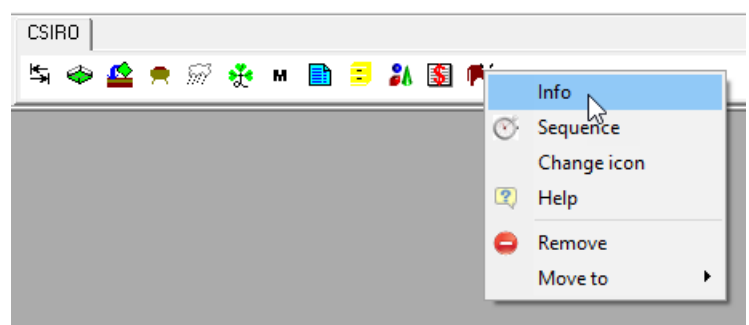
4.3 The Main Window

The main window hosts simulation windows. You may have multiple simulations open at once. Along the top of the main window is the main toolbar and the component palette.

The main toolbar is useful for quick access to common tasks.

- Create a new simulation
- Open an existing simulation
- Save the current simulation
- Open the outputs window
- Show or hide the Repository
- Open the preferences dialog

The component palette displays the components that can be incorporated into a simulation. They can be dragged from here with the mouse onto a simulation tree. Right clicking the mouse on a component item on the palette displays a menu with further options.

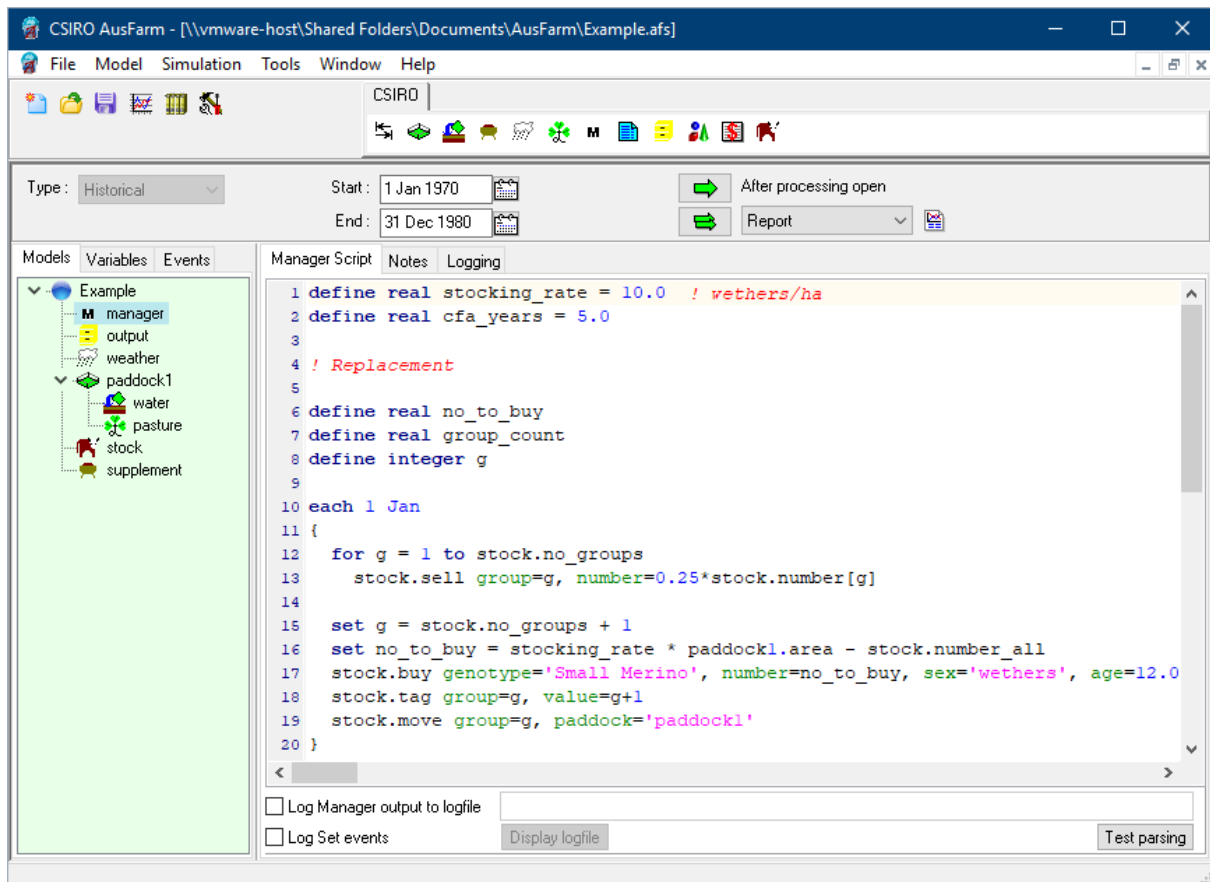


Info	Reads the internal component description and displays property and event information.
Sequence	Allows changing of the default sequenced event ordering for this component.
Change icon	Allows changing of the default component icon
Help	Opens the help file with the component specification for this component
Remove	Removes the component from the palette. This does not delete the file
Move to	Options for moving components onto other tabs

4.4 A Tour of the Simulation Window

Simulation windows are used to create and modify simulations. It is possible to have several Simulation windows open in AusFarm at once.

- ➡ Choose the **File | Reopen** menu option and choose the *Example.afs* simulation.

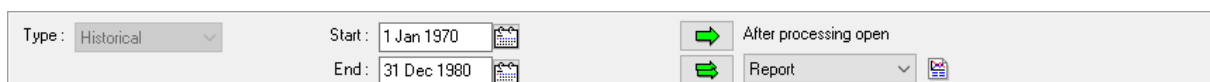


The management script is displayed for the example simulation.

Each Simulation window is divided into three main areas, or *panes*. The two middle panes contain several *tabs* that become visible depending on the task that the user is performing.

4.5 Upper pane


The upper pane is used to enter the date ranges over which the simulation is to be executed. The preferred reporting option is also chosen here.





Type Use this combo box to select the type of simulation run that is to be performed. In this release of AusFarm, the only option in this combo box is "Historical".

Start Enter the start date for the simulation. Note that all initial values that are entered apply on this date. Dates may be entered in "d mmm yyyy" format, or selected by clicking on the calendar buttons and so opening a calendar dialog.

End Enter the end date for the simulation.

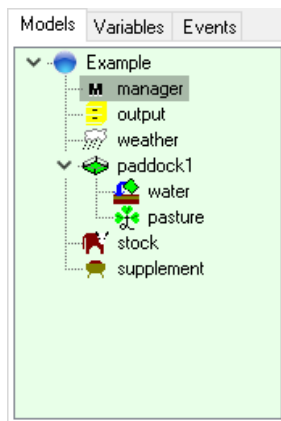
 The Run button commences the process of the simulation.

 This button appears when the simulation is configured as an Analysis. It will process the Analysis.

 The Stop button appears while the simulation is executing. Use it to halt process.

4.6 Left-hand pane

This pane has three tabs:



Models tab is always visible. It is used to configure the structure of the simulation model.

Selecting a module in the Models tab will make either the Initialise tab, the Outputs tab or the Management Script tab visible in the right-hand pane, depending on the type of module selected.

Variables tab becomes visible when the user is editing a management script or a TextOut component. It shows the names and structure of all variables belonging to modules in the simulation.

Events tab becomes visible when the user is editing a management script. It shows the names and parameters of all events belonging to modules in the simulation.

4.7 Right-hand pane

This pane has six tabs, some are visible only when specific components are selected in the model tree:

Initialise				
Notes Logging Errors				
Variable	Value	Type	Ur	
abc param_file	<none>	string		
abc species	Phalaris	string		
abc nutrients	<none>	string		
real fertility	0.75	double		
layers		array		
real max_rtdep	700.0	double	m	
real lagged_day_t	-999.9	double	ol	
real phenology	3.05	double		
real flower_len	0.0	double	d	
real flower time	0 0	double	rl	

Initialise tab

Becomes visible when the user selects a module other than a Manager or Output module in the Models tab. It shows the initial values of variables belonging to the selected module and allows the user to change them.

Outputs				
Notes Logging Errors				
Look for:	<input type="text"/>	<input type="button" value="Clear"/>	<input type="checkbox"/>	Show selected only
Simulation	Aggregation	Dec. ...	Alias	
[-] output				
[-] M manager				
[-] real cfa_years				
[-] int g				
[-] real group_count				
[-] real no_to_buy				
[-] real stocking_rate				
[-] weather				

Outputs tab

Becomes visible when the user selects an Output module in the Models tab. It is used to select the output variables that will be stored as the simulation executes which can then be examined once execution is complete.

Notes tab

Is always visible. It allows the user to annotate the Simulation window.

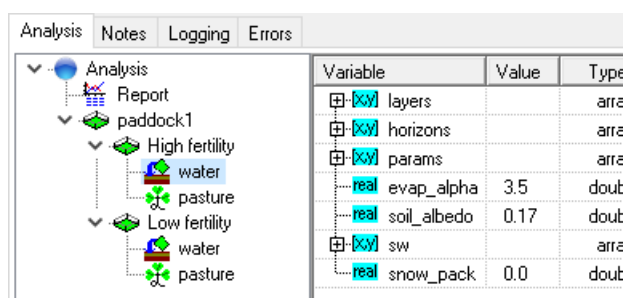
Manager Script				
Notes Logging Errors				
<pre> 1 define real stocking_rate = 10.0 ! wethers/1 2 define real cfa_years = 5.0 3 4 ! Replacement 5 6 define real no_to_buy 7 define real group_count 8 define integer g 9 10 each 1 Jan 11 { 12 for g = 1 to stock.no_groups 13 stock.sell group=g, number=0.25*stock.num </pre>				
<input type="checkbox"/>	Log Manager output to logfile	<input type="text"/>		
<input type="checkbox"/>	Log Set events	<input type="button" value="Display logfile"/>		

Management Script tab

Becomes visible when the user selects a Manager module in the Models tab. It is used to enter the management script for that module.

Logging tab

Is always visible. Use the options in the Logging tab to set up error and trace logging for the simulation. When trace file logging is turned on the tab will include an icon as a visible warning. When running multiple simulations with management events, errors or messages will be logged. These must be saved to unique files.

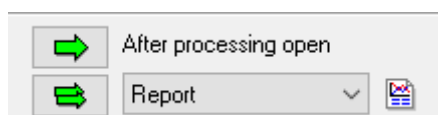


Analysis tab

Becomes visible when the user selected the Simulation module (i.e. the module at the top of the model configuration tree). It allows the user to modify the factor levels in a simulation analysis and to design one or more reports that will be generated after the simulation or analysis is executed.

4.8 Running a Simulation

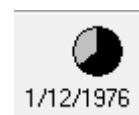
The last step before running the simulation is to set the date range over which it is to be run. If you wish you can lengthen the run period. Now check that the report option is chosen as below:



➡ This is an Analysis so use this run button to execute the simulation.



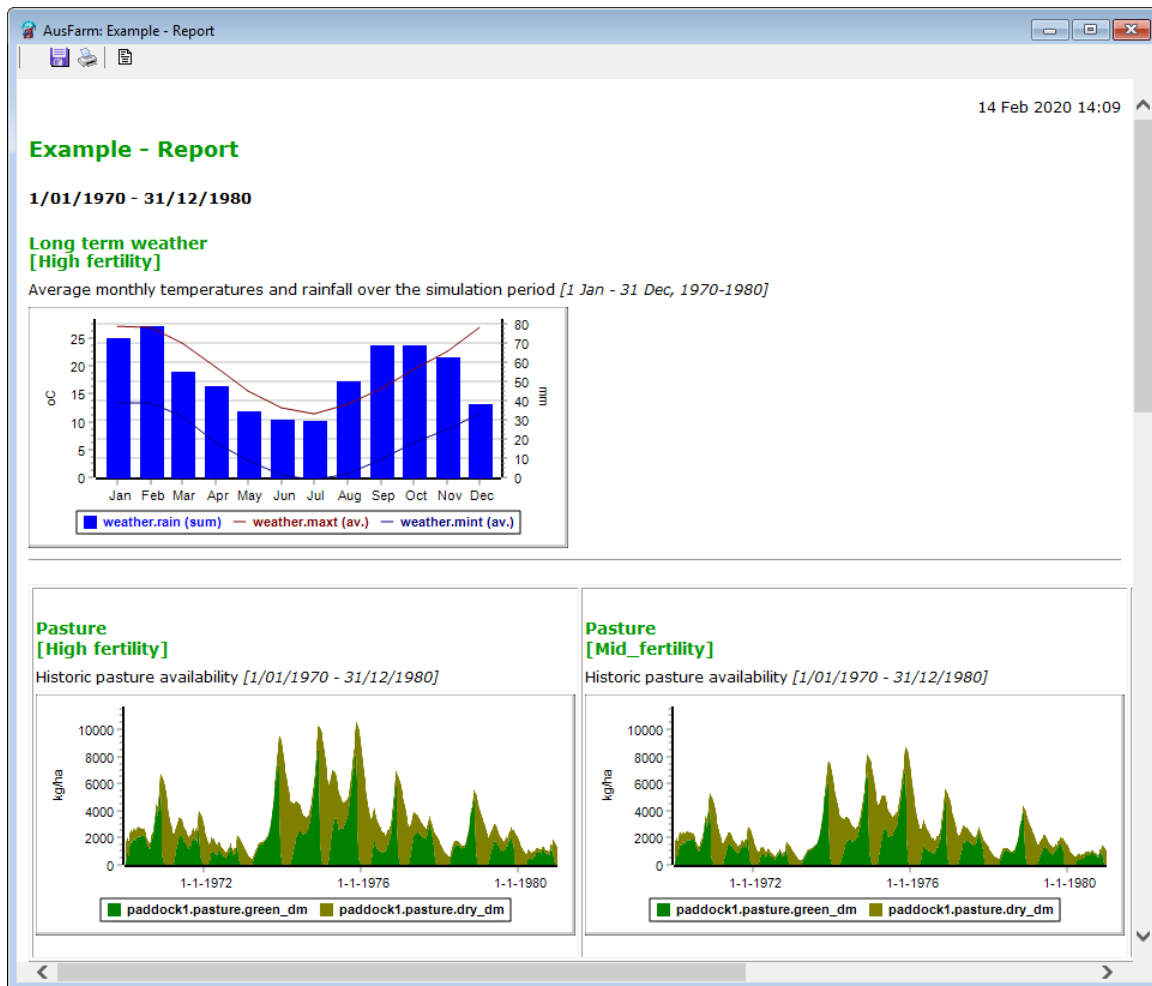
While the simulation is executing, an indicator appears on the upper pane of its window showing the progress of the calculations.



The Run button on the upper pane of the simulation window will be replaced by the Stop button. Clicking the Stop button halts execution of the simulation.

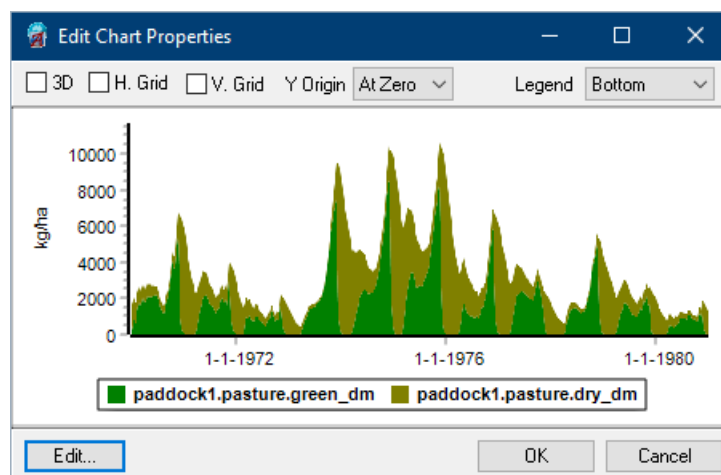


After executing this simulation, the chosen report will be displayed.



The AusFarm report window

The report is made up of multiple charts. If you single click your mouse on a chart an advanced editor will appear. From this window you can manipulate the chart and even drill down into the data that is used to generate the chart.





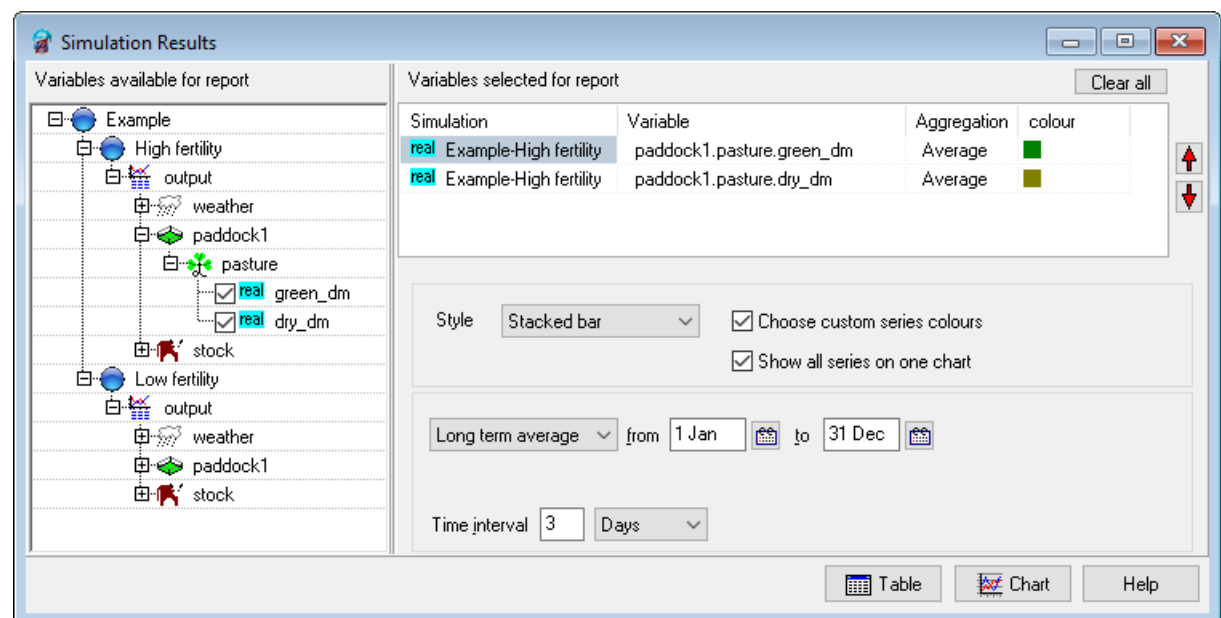
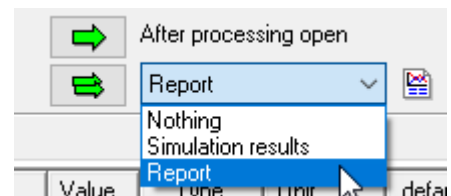
Changes that you make on this dialog will be copied back to the main report when you click the **OK** button. The **Edit...** button opens an even more detailed window where you can view the data, export the data or change detailed chart layouts.

4.9 The Results Window

The Simulation results option will open the Simulation Results window in the same way that the main toolbar option does. From there you can choose which outputs you wish to see.

The Simulation Results window in AusFarm is used to select results from simulations and to format them for display in reports. It is a quicker way of displaying outputs in an adhoc way where a complete report template is not needed.

- ➡ Open the Simulation Results window by clicking the Results button () on the main toolbar or choosing the Simulation Results option from the drop-down list as shown here and then click the chart button .

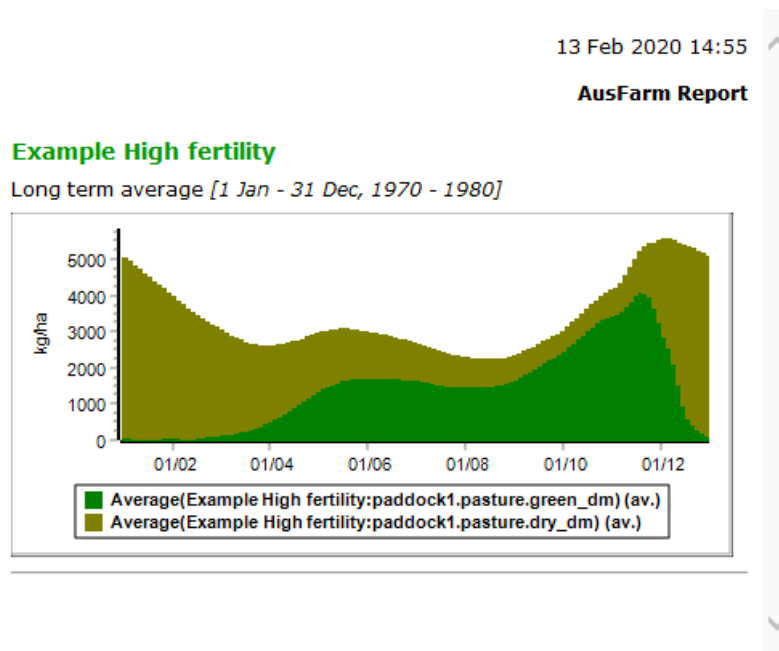


All variables available for reporting from all completed simulations are shown in the tree at the left of the Results window. To see the output files, modules and variables below a node in the tree, click on the expand button by the node's name. To hide them, click on the collapse button.

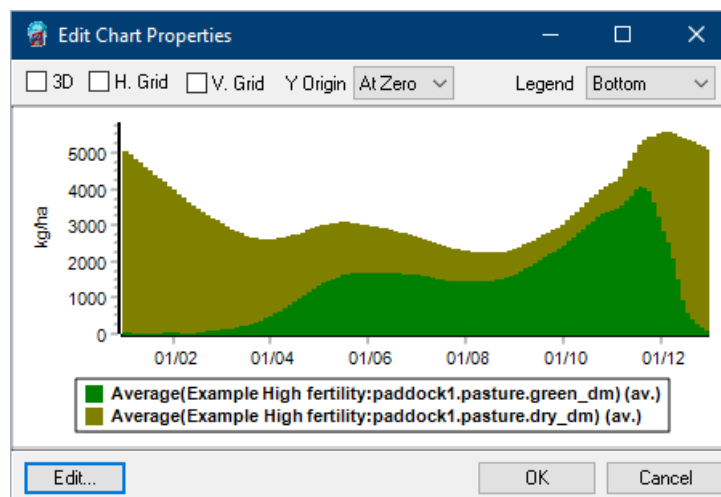
4.10 Generating a chart

- ➡ Select the *cover_green* and *cover_tot* variables by clicking the check boxes next to their names in the tree.
- ➡ Change the data treatment to "Long term average" and set the date range to be 1 Jan to 31 Dec.
- ➡ Change the time interval to 3 days

- ➡ Check the **Choose custom series colours** box.
- ➡ For each selected variable, click on the **Colour** column and select a colour for the variable's data.
- ➡ Click on the **Style** combo box and examine the options. Set the chart style as Stacked bar.
- ➡ Click the **Chart** button. A Report window will be generated containing a chart something like this:



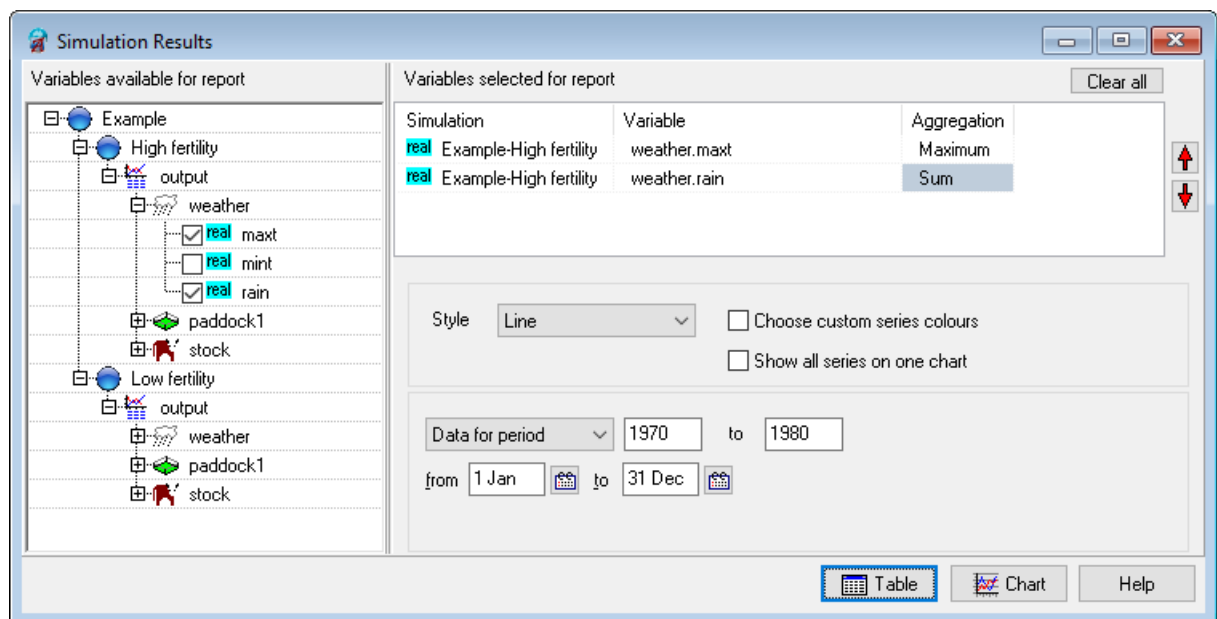
Click on the chart. The Edit Chart Properties dialog will appear:



If you resize this window or change any chart formatting and then click **OK**, the changes will be displayed on the report page. Click on the **Edit...** button and explore the options for formatting the chart.

4.11 Generating a table

- ➡ Return to the Results window.
- ➡ Click on the **Clear all** button to clear any selected variables.
- ➡ Add the *maxt* and *rain* variables to the selected list. Set the aggregation of the *rain* variable to “Sum” and the aggregation of *maxt* to “Maximum”.
- ➡ Choose the **Data over period** option.
- ➡ Click the **Table** button. A Report window containing annual total rainfalls and yearly maximum temperatures will be generated:



13 Feb 2020 15:00

AusFarm Report

Example High fertility

Data for period 1/01/1970 to 31/12/1980]

Date	Maximum weather.maxt (max)	Sum weather.rain (sum)
	oC	mm
1970	36.40	722.40
1971	37.70	616.00
1972	38.70	396.10
1973	40.20	755.00
1974	32.60	977.00
1975	35.40	771.00
1976	34.60	592.80
1977	37.40	513.00
1978	36.40	771.00
1979	38.70	408.60
1980	37.00	460.40

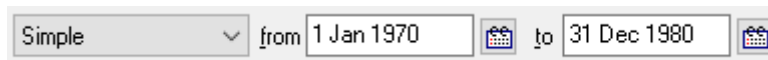
Output display in AusFarm comes in two kinds: tables and charts. The process for generating these in reports follows the same general set of steps:

- Select the variables that are to be presented in the report.
- Specify the data treatment that is to be applied to all variables.
- Specify the aggregation to be used for each variable within each time interval. This step is only required if a time interval greater than one day is used.
- Click the Chart or Table button to generate a Report window.

4.12 Data treatments

The values of variables that are output from an AusFarm simulation can be treated in a variety of ways. In the Results window, the following data treatments may be selected when producing charts or tables:

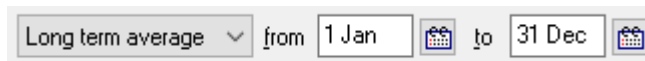
Simple



The interface shows a dropdown menu set to 'Simple'. To its right, there are two date input fields: 'from 1 Jan 1970' and 'to 31 Dec 1980'. Each date field has a small calendar icon to its right.

A simple presentation just presents the values of the selected output over time.

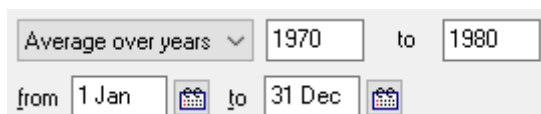
Long Term Average



The interface shows a dropdown menu set to 'Long term average'. To its right, there are two date input fields: 'from 1 Jan' and 'to 31 Dec'. Each date field has a small calendar icon to its right.

For each day of year, an average value of the variables is computed over all the years in the course of the simulation. In a chart, therefore, the X-axis shows days (or months) of the year, e.g. 1 Jan, 2 Jan etc; the Y-axis gives the values of the output.

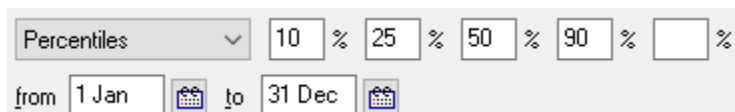
Average over years



The interface shows a dropdown menu set to 'Average over years'. To its right, there are two year input fields: '1970' and '1980'. Below these, there are two date input fields: 'from 1 Jan' and 'to 31 Dec'. Each date field has a small calendar icon to its right.

As for the Long Term Average treatment, but the average values are computed over the selected range of years from the simulation.

Percentiles



The interface shows a dropdown menu set to 'Percentiles'. To its right, there are five percentile input fields: '10 %', '25 %', '50 %', '90 %', and an empty field followed by a '%' sign. Below these, there are two date input fields: 'from 1 Jan' and 'to 31 Dec'. Each date field has a small calendar icon to its right.

The user nominates up to five percentile levels for display. For each day of year in the nominated range, the output values for all the years in the simulation are ranked. The value corresponding to each percentile level is then computed. The values for each percentile level over time are presented as the output series.

A point (x,y) on the z^{th} percentile graph should be read as follows:

On day-of-year x a value less than or equal to y will be encountered in z% of years.

The X-axis of a percentile chart shows days (or months) of the year, e.g. 1 Jan, 2 Jan etc; the Y-axis gives the values of the output.

Data for period

For this treatment, the user nominates a range of days of the year and also a range of years.

For each year in the range, an aggregated value is computed over the range of days and these summary values are presented.

The X-axis for a Data for period graph is the year, e.g. 1978, 1979 etc; the Y-axis gives the variable values.

To view values for a single day of year, select a time interval of one day in constructing this range, e.g. 15 Apr to 15 Apr.

Data over period

As for Data for period, but summarized over all years in the simulation.

P.D.F. for period

P.D.F. stands for *probability density function*. The Y-axis shows the frequency of occurrence (0-100%) for each of the classes on the X-axis. The user nominates a range of days of year. AusFarm then aggregates the selected variables over these days for all years of the simulation and allocates them to a class. The class boundaries are determined by taking the range of values and dividing it into five or ten equal classes, depending on the number of years involved. A PDF graph with value y should be read as follows:

There is a probability y that the selected output will fall within the class given on the X-axis.

C.D.F. for period

C.D.F. for period	1970	to	1980
from	1 Apr	to	31 Oct

C.D.F. stands for *cumulative distribution function*; to be precise, this presentation shows probabilities of exceedance. The X-axis of a C.D.F. graph shows the range of output values and the Y-axis gives the probability that in any year, the value of the output will be *greater* than a given level. The set of outputs used to estimate these probabilities is computed as for a P.D.F. A point (x,y) on a C.D.F. graph should be read as follows:

There is a chance y that the output variable will be greater than a value of x at the given time of year.

5. Configuring a new simulation

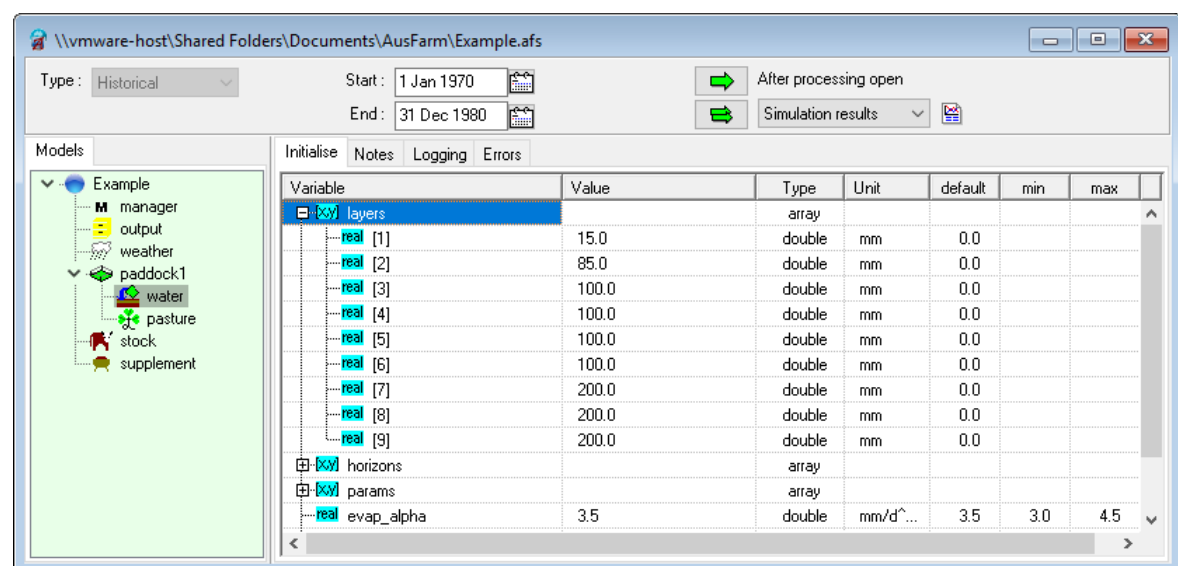
5.1 Configuring and Initialising Modules

The first major step in constructing a simulation analysis in AusFarm is to *configure* the simulation by adding the set of component modules in the simulation and describing their inter-relationships.

Before configuring a simulation, the user should consider what set of processes needs to be included in the simulation to answer the question of interest. *The configuration that is chosen should be the simplest that meets this criterion.*

Building a simulation from scratch can be an involved process. Any of the components from the model palette can be dragged and dropped onto a model tree in a simulation window. The following instructions will use an existing simulation and go through the configuration process to show how a simple simulation can be configured.

- ➡ Close any existing simulation and select the **File | Reopen** menu option and choose the *Example.afs* simulation.
- ➡ In the example simulation click on the water module in the Models tree and then on the Initialise tab. Fields of records and elements of arrays are organized into a tree-structure, as shown in the figure below:



The Initialise tab can be seen in the right-hand pane of the simulation window. It contains a grid in which each row represents a variable (or part of a variable), and there are columns that display each variable's type and name, initial value, and (optionally) units, default value, and minimum and maximum permitted value. The type of each value is shown by the icon next to the variable's name.

- ➡ The initial values (in the **Value** column) can be edited. Press **F2** or click twice on a value (leave a gap between the clicks). This activates an editor that allows you to change the value. (Don't change anything now.) Press Enter to deactivate the editor or an arrow key to move to another value. Array items shown here can be resized by right clicking on them and choosing **Add** or **Resize**.

The easiest way to specify the initial values of a module is by using its initialisation dialog. Each component in the default AusFarm distribution has its own dialog.

- ➡ Open the initialisation dialog for the Weather module by double clicking on its icon in the Model tree.

The Weather module is responsible for specifying the input weather data and some further options about CO₂ levels. SILO format weather files can be selected here.

- ➡ Close the dialog and open the dialog for the Paddock.

The Paddock module allows you specify the area and slope of the paddock.

- ➡ Examine the Stock initialisation dialog by double clicking the Stock module in the Model tree:

The Stock dialog shows that two genotypes are described using some parameters that specify breed characteristics. Any of the breeds found in the **Breeds** dropdown list can be used in the simulation. If you redefine a breed here it will use the parameters, you have set.

- ➡ Close this dialog and then open each of the remaining dialogs in the simulation to get a view of the initialisation requirements for these sub-models.

The Manager module is another special component. When you select it, you will see the management script that controls much of the behaviour of the simulation.

- ➡ Examine the management script by clicking on the Manager module in the simulation tree. It contains a variety of different elements.
- ⇒ *Definition statements* create variables that can be used in other places the script.

```
1 define real stocking_rate = 10.0 ! wethers/ha
2 define real cfa_years = 5.0
~
```

- ⇒ *Time specifiers* determine when rules should be executed.

```
10 each 1 Jan
11 {
```

- ⇒ *References to external variables* allow the rules to be influenced by the state of the simulation.

```
44 if stock.cond_score_all < 1.0
```

- ⇒ *Events* change the state of other modules.

```
38 stock.sell group=g , number=stock.number[g]
```

⇒ *Control rules* govern the order in which events are executed.

```
35 for g = 1 to stock.no_groups
36 {
37     if age[g] >= 365 * cfa_years
38         stock.sell group=g , number=stock.number[g]
39 }
```

⇒ *Assignment statements* set the value of defined variables.

```
31 group_count = stock.no_groups
```

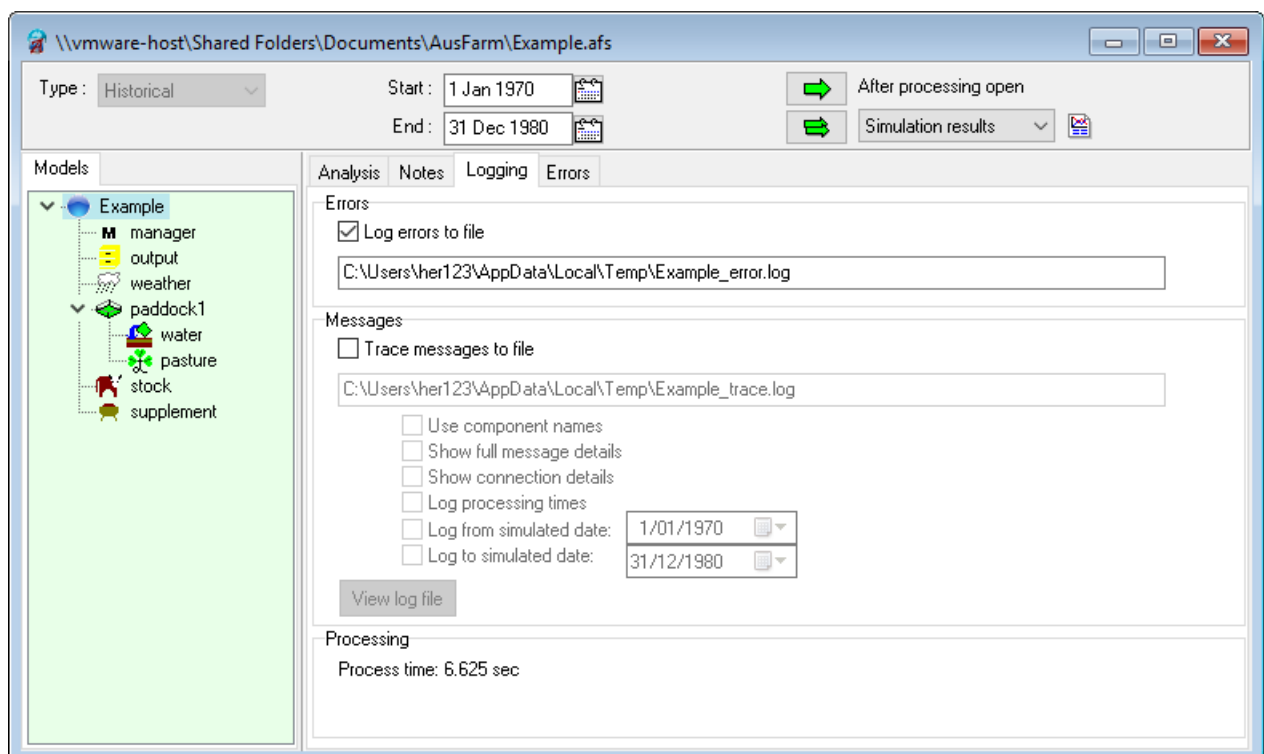
5.2 The Notes and Logging tabs

➡ Click on the Notes tab in the right-hand pane of the simulation window.

This tab contains an area where you can document the purpose and features of your custom simulation in text form.

➡ Click on the Logging tab in the right-hand pane of the simulation window.

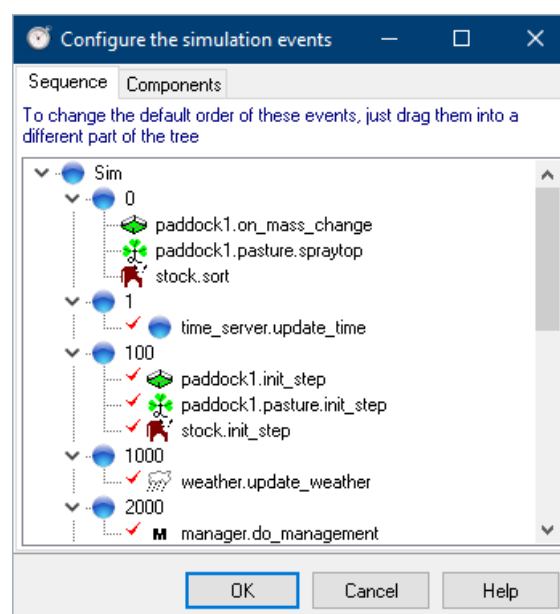
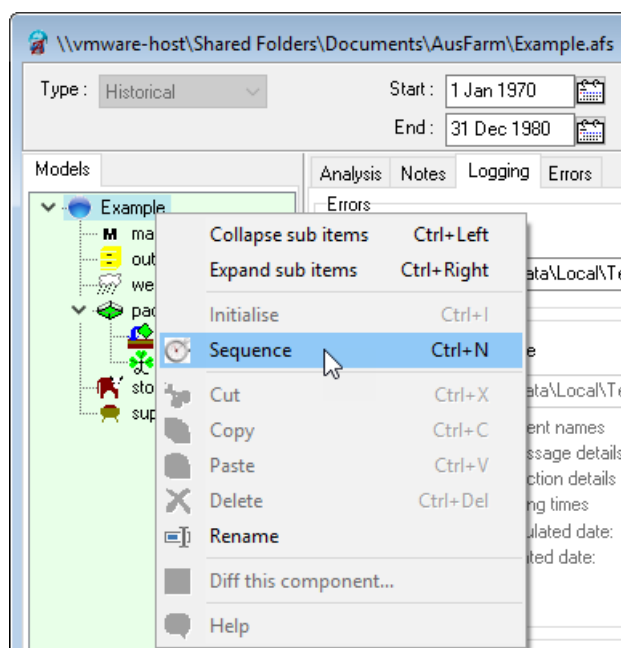
This tab contains options that allow details about the execution of a simulation to be examined once it has been run. (Using the trace option adds significant time to the simulation run and should only be used when there is an internal problem with the simulation structure that must be solved.)



5.3 Sequencing the simulation

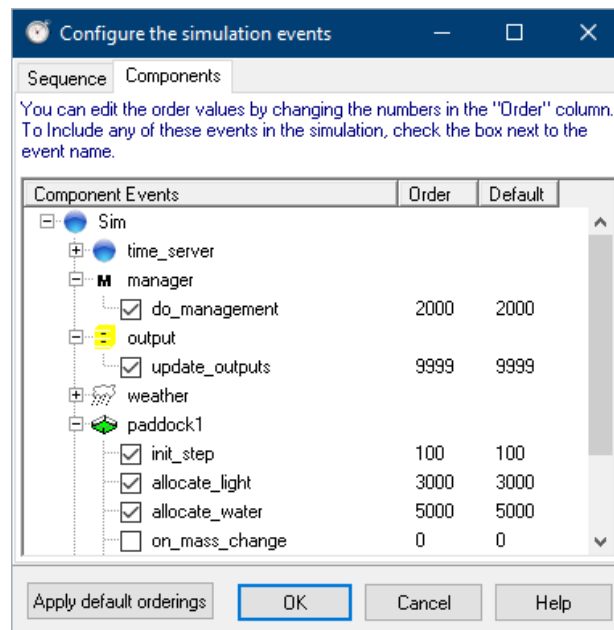
The modules in the simulation will have some default sequenced events triggered during the execution of the simulation. If multiple Manager modules or multiple TextOut modules are included in the simulation, it may be necessary to adjust when their logic is processed within the daily timestep. Each module can have the timing of its default subscribed events adjusted individually, but it is also possible to view the sequencing of whole simulation.

- ➡ Right click the mouse on the top node of the model tree and choose **Sequence**.

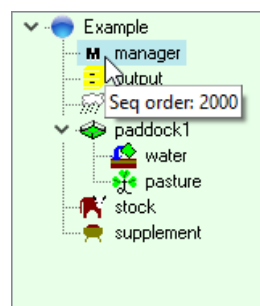


The editing dialog has two tabs. The first one gives a view of the whole simulation ordered by the sequence number in the timestep. The second tab allows editing of the

In the example below you can see that the *manager* management script will have its logic processed before the Output component no matter where it resides in the model tree.



Manager components also show their sequence when you hover over them in the Models tree. This is very useful when the simulation contains more than one Manager component.



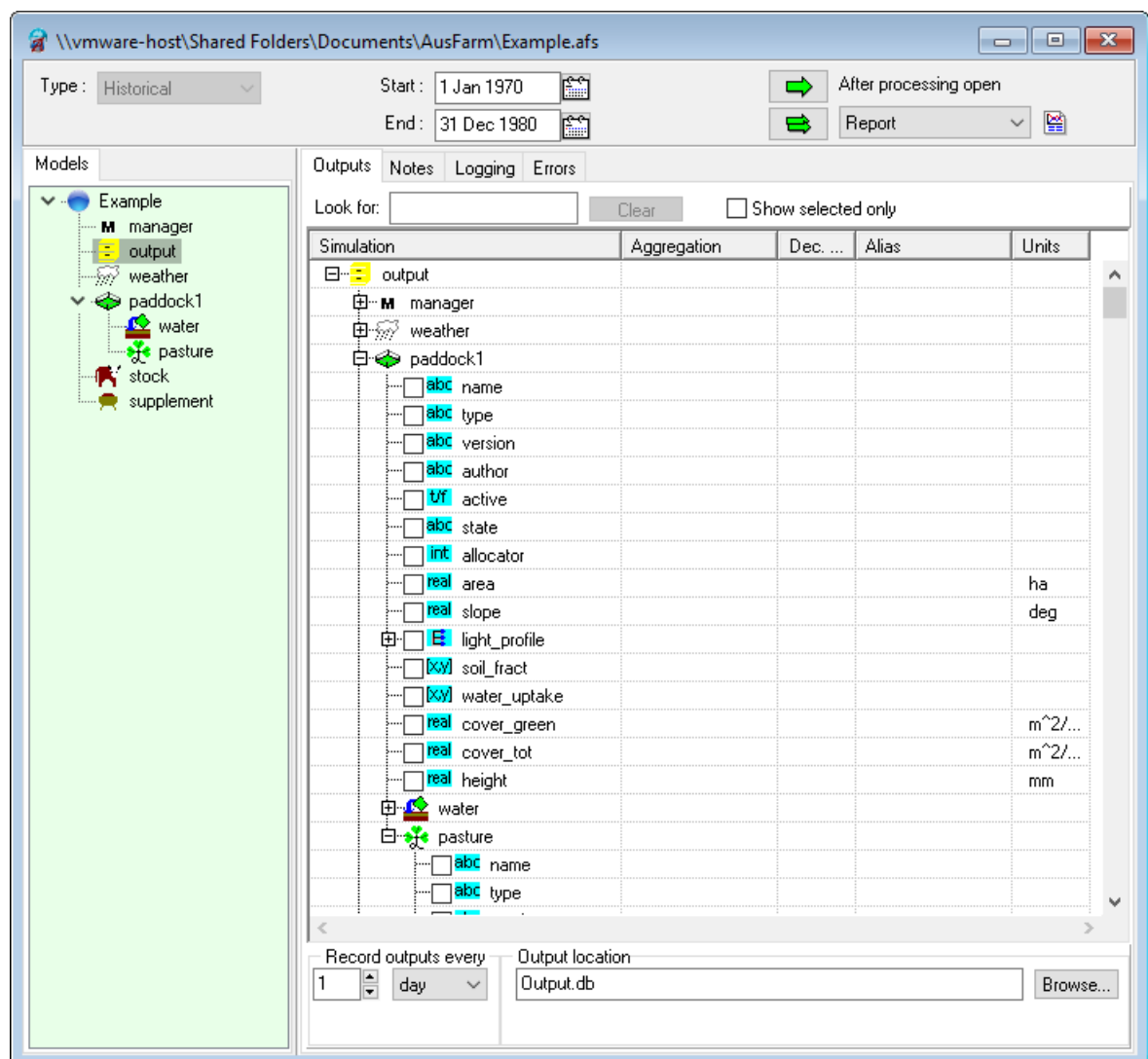
5.4 Selecting and Storing Outputs

When a simulation is run, AusFarm can store the values of variables over time so they can be shown or summarized in a report. Before variables can be used in reports, they must be selected as part of an Output module.

- ➡ Select the Output module by clicking on its icon (📄) in the Models tab of the simulation window.

The Output component is a special case in an AusFarm simulation. When you select the item in the model tree you are shown many of the variables that can be used for reporting. The Outputs tab will replace the Initialise tab in the right-hand pane.

If you want to filter the list of variables by name you can start typing the name in the **Look for:** text entry. It is also possible to filter the list to only show the outputs that are selected. These filters can be used together if required.



The Outputs tab contains a grid containing variables that are organized into a tree-structure.

- ➡ Click on the expansion button (⊞) by the Weather module's icon in the Outputs tab. A list of the output variables of the module will be revealed.
- ➡ Scan down through the list of variables and expand the variable named **weather** to see its fields.
- ➡ Check that the *cover_green*, *cover_tot* variables are selected for output. Type **cover** in the filter text entry.

For the floating point variables, the number of decimal places to be displayed can be chosen here.

When an array type is selected, all its elements will be accessible for display.

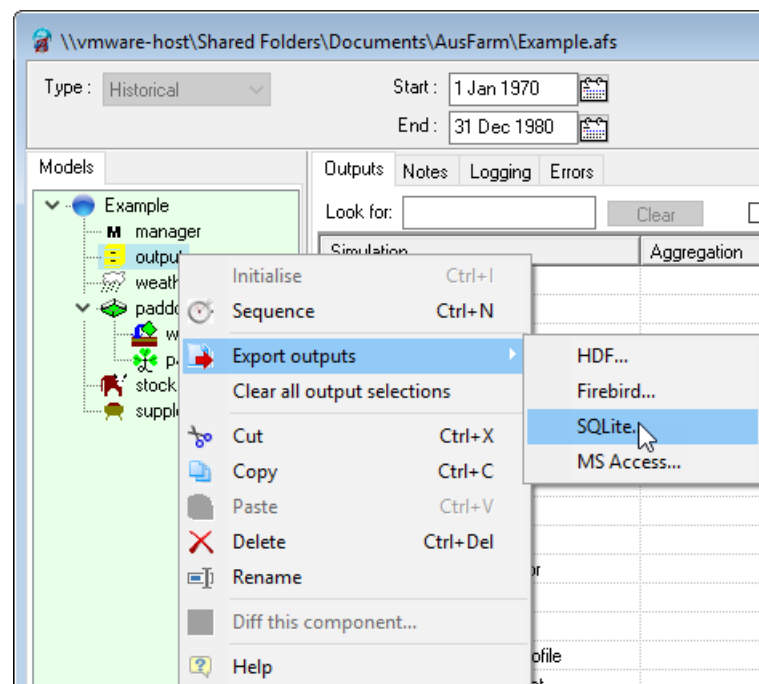
The **Aggregation** and **Alias** columns can be ignored in most situations.

The output file location can be modified from this tab. It is good practice to have this file name match that of the simulation file. It is possible to run more than one simulation simultaneously. To do this the output modules *must* save their outputs to different files. (Note that AusFarm results files are Microsoft Access or SQLite data bases, but the data is stored in a compressed binary form and can be exported after the simulation run if the data needs to be summarised using any SQL processing.)

5.4.1 Exporting results

When using the database output component, it is possible to export the results from a simulation or Analysis run into a single database. This is useful for collating the results from the simulation runs into a database that can be queried using SQL.

- ➡ Right click the mouse on the output component and choose Export outputs. There are three storage formats available. Choose SQLite.



When the outputs are exported into a database, they can be opened by the respective client tool. For SQLite databases a useful application is SQLiteSpy.

- ➡ Install SQLiteSpy and then you will be able to open a database and view data as shown below.

RunDescr	StepDate	weather_maxt	weat...	weat...	paddock1_pasture_green_dm	padd...	stock...	stock...	stock...
Example:High fertility	1980-12-26	28.1	16.9	0.0	13.4772	1351.5	122	879	0
Example:High fertility	1980-12-27	27.8	15.5	0.0	11.9608	1339.1	122	879	0
Example:High fertility	1980-12-28	32.2	14.5	1.2	10.4938	1326.81	122	879	0
Example:High fertility	1980-12-29	21.0	16.8	4.2	9.50898	1313.66	122	879	0
Example:High fertility	1980-12-30	24.5	13.5	0.0	8.61429	1299.07	122	879	0
Example:High fertility	1980-12-31	30.0	9.0	0.0	7.78002	1284.78	122	879	0
Example:Low fertility	1970-01-01	20.6	10.1	14.7	0.0	1989.84	999	999	0
Example:Low fertility	1970-01-02	12.2	7.6	6.1	0.0	1952.09	999	999	0
Example:Low fertility	1970-01-03	15.6	8.4	0.5	0.0	1912.18	999	999	0
Example:Low fertility	1970-01-04	18.6	9.8	7.6	0.0	1873.7	999	999	0
Example:Low fertility	1970-01-05	23.9	8.6	0.0	0.0	1834.6	999	999	0
Example:Low fertility	1970-01-06	23.5	11.6	0.0	0.0	1796.86	999	999	0
Example:Low fertility	1970-01-07	22.2	12.1	0.0	10.9478	1760.65	999	999	0
Example:Low fertility	1970-01-08	24.4	13.0	0.0	21.3174	1724.71	999	999	0
Example:Low fertility	1970-01-09	25.5	14.1	1.0	31.3857	1689.69	999	999	0
Example:Low fertility	1970-01-10	30.6	14.2	2.0	41.2267	1654.79	998	998	0

Viewing the results using SQLiteSpy

HDF (similar to NetCDF) format is also available. This requires a HDF viewer. When using MS Access, the normal Microsoft Access database program will be able to open this database.

When exporting data from an Analysis run, the records will have the name of the treatment in the first column.

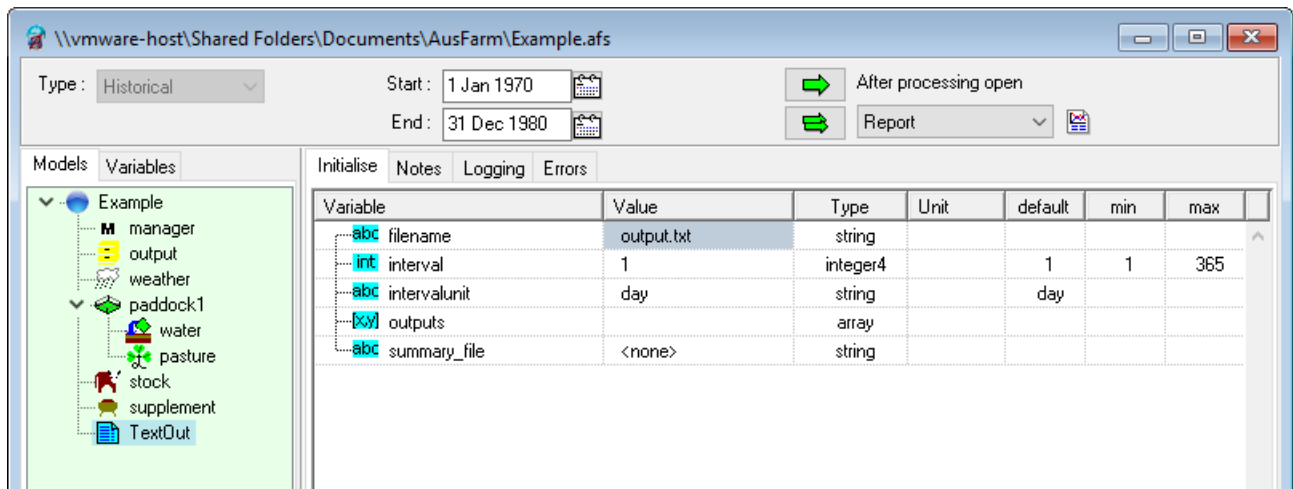
5.4.2 Using the TextOut component

If output from the simulation run is preferred in tab delimited text file format, then the TextOut component can be used. It is available in the model palette. Once this is placed in the simulation, variables can be selected for this component using its inbuilt component dialog that you can access by double clicking on the component.

- ➡ Add a TextOut component to the simulation by dragging the icon from the model palette and dropping it on the top item (*Example*) in the model tree.
- ➡ Edit the filename property and type in a name for the output file.

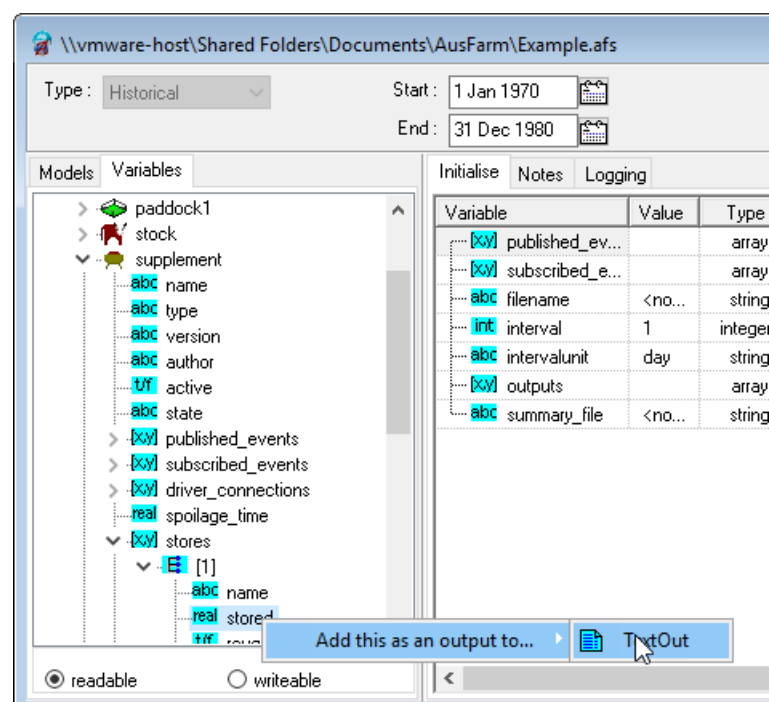
Selecting output variables

There are two easy methods for adding component properties to the outputs list of TextOut modules.

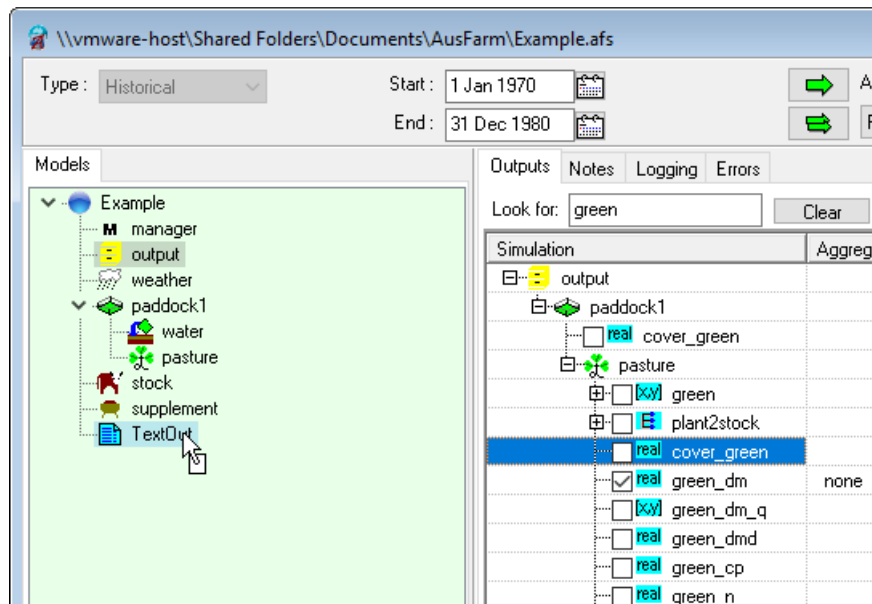


1. When a TextOut module is selected, the Variables tab becomes visible. From there the modules in the model tree can be expanded and properties can be chosen by right clicking on them and added to the output list for any TextOut in the simulation.

➡ Right click on a property and then choose the TextOut component that the variable will be added to.



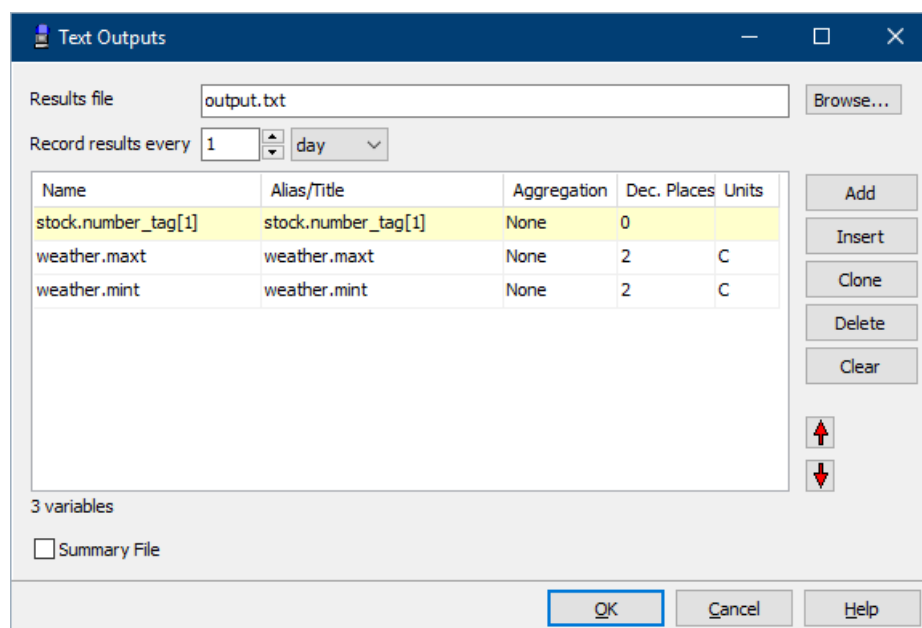
- Another method of adding variable to TextOut modules involves having an Output module (database version) in the model tree.



When you select the Output module, the selection tree will become visible in the right-hand panel. From here the variables can be filtered if required. Any variable can be dragged and dropped, using the mouse, onto a TextOut module.

Once a simulation has been run and outputs stored in a text file, right clicking on the TextOut module and choosing **View text output**, will allow you to open the resulting text file from the AusFarm user interface.

A third way of selecting variables to be stored by TextOut is when you double click on the TextOut module a dialog will open where you can specify each variable you want to store.

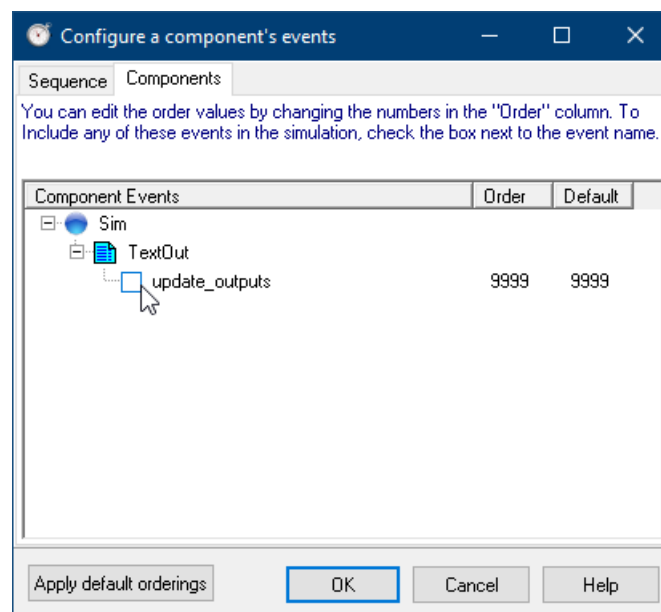


Reporting at custom points in the simulation

If you need to store values at times other than daily, monthly or yearly it is possible to tell the TextOut module explicitly when to store its outputs. From a Manager script you can call the **update_outputs** event as shown below.

```
47 each 25 Dec
48 {
49     TextOut.update_outputs
50 }
```

Then it is important to turn off the automatic sequencing of the event. Right click on the TextOut module in the Models tree and select **Sequence**. From the configuration dialog turn off the sequence as shown.

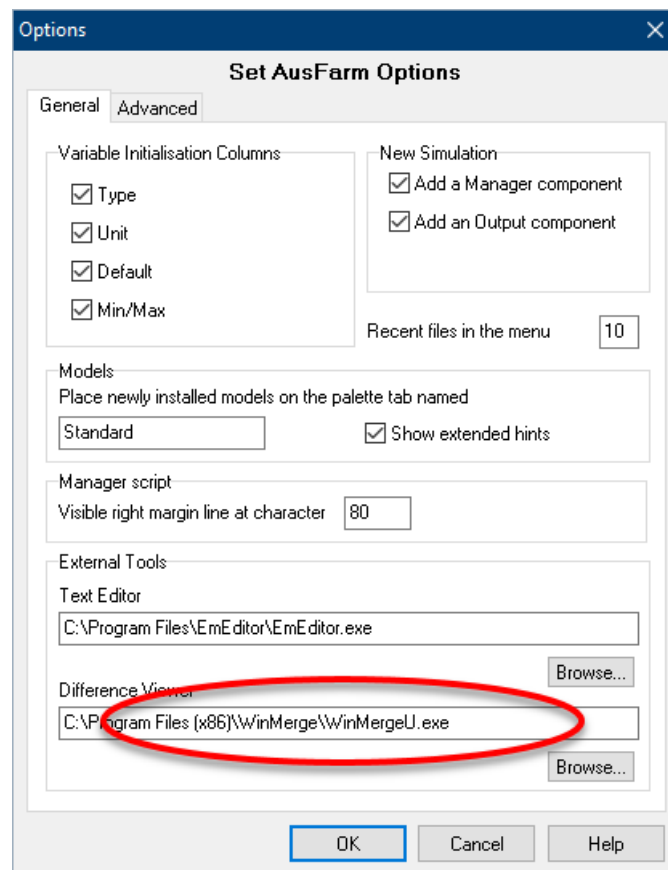


The TextOut module will then report its variables now only on the 25th Dec each year. This configuration is useful if you want to report only on special events in the simulation.

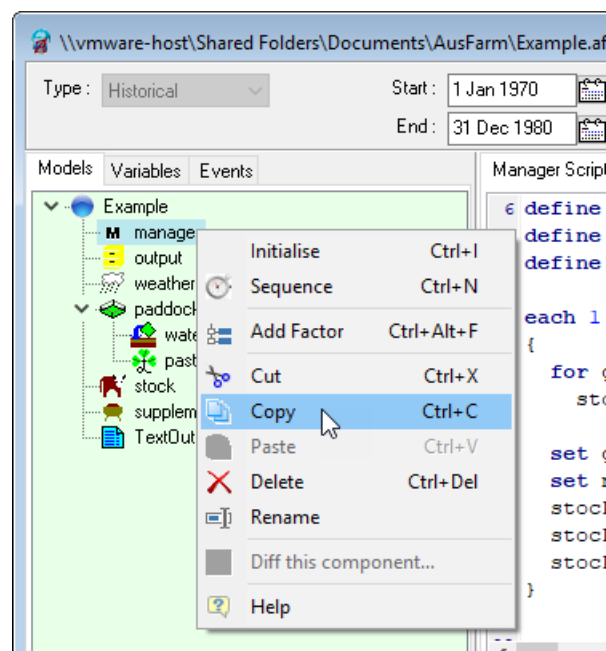
5.5 Comparing initial values of components

It is often useful to compare the initial values for like components in a simulation or between simulations. Using the clipboard to copy the first module in the model tree and then choosing the Diff menu option on a second module will invoke a difference viewer.

- ➡ A file difference viewer such as the free WinMerge package would be adequate. Download this software and install it first.
- ➡ Configure the settings in the AusFarm Options dialog for the Difference viewer.



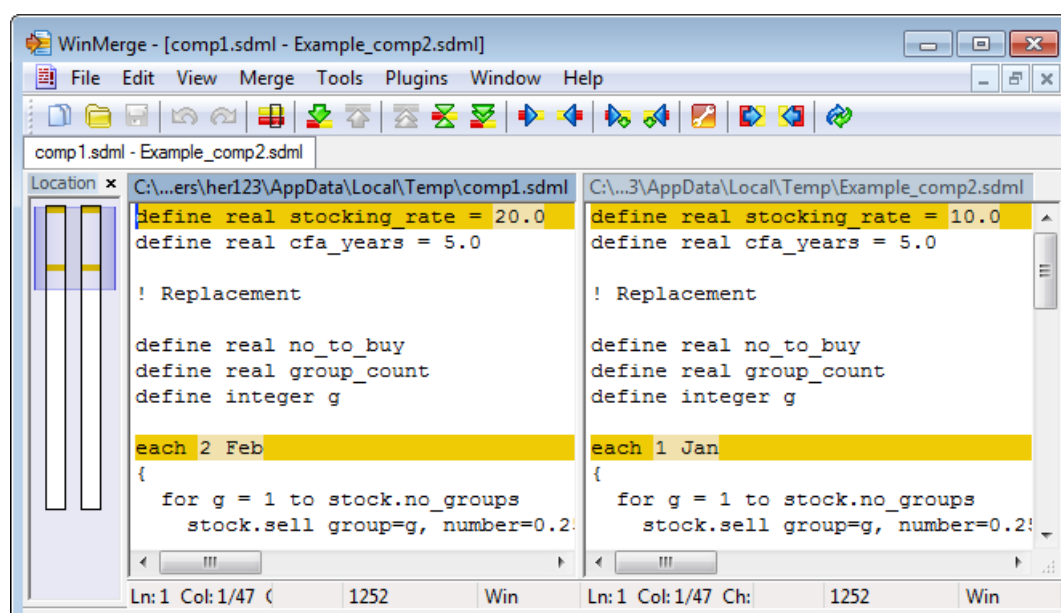
- ➡ From the model tree select a module and then right click and select the **Copy** option from the popup menu.



- ➡ Open another simulation or from within this simulation select another module in the model tree of the *same type* and right click the mouse. The popup menu will

now have an extra option, **Diff with....**. Choose this option and the external viewer will show the differences between these modules.

For Manager script modules, the text will be plain text as seen in the Manager Script editor. For other modules, the SDML script in XML form will be shown. Although a little cryptic this is still a useful means of checking for differences. Because this differencing technique uses data stored in the clipboard, it is easy to do comparisons between modules in different simulations.



An example of comparing two Manager scripts.

6. Specifying Management

Management activities in AusFarm simulations are represented as a series of *events* that change the state of the various biophysical models that make up the simulation. For example, irrigation is represented as an event that changes the amount of water present in the soil profile, and the selling of livestock is represented as an event that changes the number present of a specific group of animals in the simulation.

Each module has a defined set of management events that can be applied to it. When and how these events take place is specified using one or more Manager modules. Each Manager module contains a management script composed of statements that describe:

- When and under what conditions events are to be executed;
- Which module(s) are to execute an event;
- The parameters that determine exactly what happens when the event takes place.

In the real world, the timing and nature of management activities often depend upon the current state of the system. For example, irrigations (events) might be scheduled to take place only when the soil water deficit (part of the system state) is greater than a nominated threshold. Management scripts can respond to the state of the simulation by accessing variables from the rest of the simulation. The values of these variables can then be used to specify event parameters and the conditions that determine whether events take place. They can also be combined into expressions and defined variables that may then be used in management rules.

To develop management scripts, it is important to have a good understanding of the variables and events that are available in the simulation.

6.1 Variables

In AusFarm, the *variables* of each module are used to represent the quantities used in the equations and events that the module embodies. A “variable” in AusFarm includes a wide range of quantities from a modeller's point of view, including:

State variables	Quantities that may vary in time as the simulation is computed. The value of a state variable must be known in order to compute the dynamics of the module to which it belongs.
Constants	Quantities that are (i) invariant in time and (ii) have the same value in all modules of all simulations.

Parameters	Quantities that are invariant in time, but may take different values in different modules, either within a simulation or between simulations.
Driving variables	Quantities that are stored externally to a given module but must be known in order to compute the dynamics of the module. They may (and usually do) vary in time. Each driving variable must have one or more <i>sources</i> ; a source must be an output variable from another module.
Output variables	Quantities that may be accessed by other modules in the simulation, including for storage as results or for use in management scripts. Output variables may be state variables, constants or parameters, but may also be "summary" variables computed from them. The variables that drive the simulation as a whole (e.g. weather data) also appear as the output variables of modules that read them in.

Every variable in AusFarm has a *name*, a *type*, and a *value*. When referring to a variable, its name may be *qualified* to ensure that the reference is not ambiguous: for example, the **sw** variable within the **paddock3.water** module may also be referred to as **paddock3.water.sw**.

The value of a variable can change through time as the simulation is executed. The initial value of each state variable and parameter must be provided by the user in order for the simulation to be computed; these two types of variables are known as *initialisation* variables.

Variables come in three main kinds, or *types*: scalars, arrays and structures.

Scalars have a single value. There are four types of scalar variables:

Real	Can be any numeric value. When writing a real value in a management script, either decimal notation (e.g. -63.45) or exponential notation (e.g. 1.46E-5) may be used.
Integer	whole values: ...-4, -3, -2, -1, 0, 1, 2, 3, 4, ...
Text	may contain any text (i.e. zero or more characters). When writing a text value in a management script, the characters are surrounded with single quotes (e.g. 'xyz') to distinguish them from references to variables, which are written without quotes. To place a quote character in a text value, write two quotes: for example, writing 'quote(')' gives the value quote(') .

Logical variables are either true or false. A true value is written as **TRUE** in a management script, while a false value is written as **FALSE** (this is case-insensitive)

Arrays are ordered lists of variables in which all the members (known as *elements*) are of the same type. When writing an array in a management script (the Manager module), the elements are surrounded with square brackets (`[]`) and successive elements are separated from one another with commas. The name of the *n*-th element of the array named **array** is **array [n]** (*n* is known as an *index*). The first element of an array has index 1. Below is an illustration of what an array looks like in a Manager component script.

```
define integer x[8]                                ! An array of integers called x
set x[6] = 99                                       ! Refer to the 6th element of array x
```

Structures are lists of variables in which the members (known as *fields*) may be of different types. Since each field of a structure is itself a variable, it has a name and a type. When writing a structure value in a management script:

- the structure is surrounded by brackets (`()`);
- successive fields are separated from one another with semi-colons; and
- the value of each field is preceded by its name and a colon.

To refer to a field of a structure variable in a management script, append the field name to the structure name, with a colon between them (e.g. **seeds:soft_ripe**).

```
define s = (field1:8; field2:'fox'; field3:- 99.9) ! A structure with three fields
set s:field2 = 'jumps'                            ! Refer to the second field of structure s
```

6.2 Events

6.2.1 Management events

A *management event* of a module represents an instantaneous change in the module's state variables. Each event has zero or more quantities, known as *parameters*, which are used to specify exactly how the module's state variables are changed.

For example:

Application of irrigation water to a soil can be represented as an event that changes the amount of water present in the soil profile. The amount of water applied, and the rate of application are parameters that affect how the added water will percolate into the soil.

The selling of livestock is an event that changes the number of a specific group of animals that are present in the simulation. The group of animals to be sold, and the number to sell are the parameters of this event.

An event is specified in a management script by giving its name, followed by the value of zero or more parameters. Each parameter is written by giving its name, followed by an equals sign (=) and an *expression* that is computed to give the value of the parameter (see section 15.3 for more information about expressions).

```
! Send the "shear" event (with no parameters) to all modules that accept it.
shear

! Send the "buy" event to a specific module named "cattle".
! Four parameters are given, with a space between the event name and first parameter
! and commas between the parameters.
! Note how the "number" parameter is specified as an expression that must be evaluated.
cattle.buy number=0.5*stock_rate*paddock1.area, sex='steers', age=8.0, number=200.0
```

More detail about specifying events is given in section 15.2.

6.2.2 Sequenced events

The rate equations of components are also implemented as software events, known as *sequenced events*. Each sequenced event is computed once per time step. AusFarm handles the setting up of the sequenced events automatically.

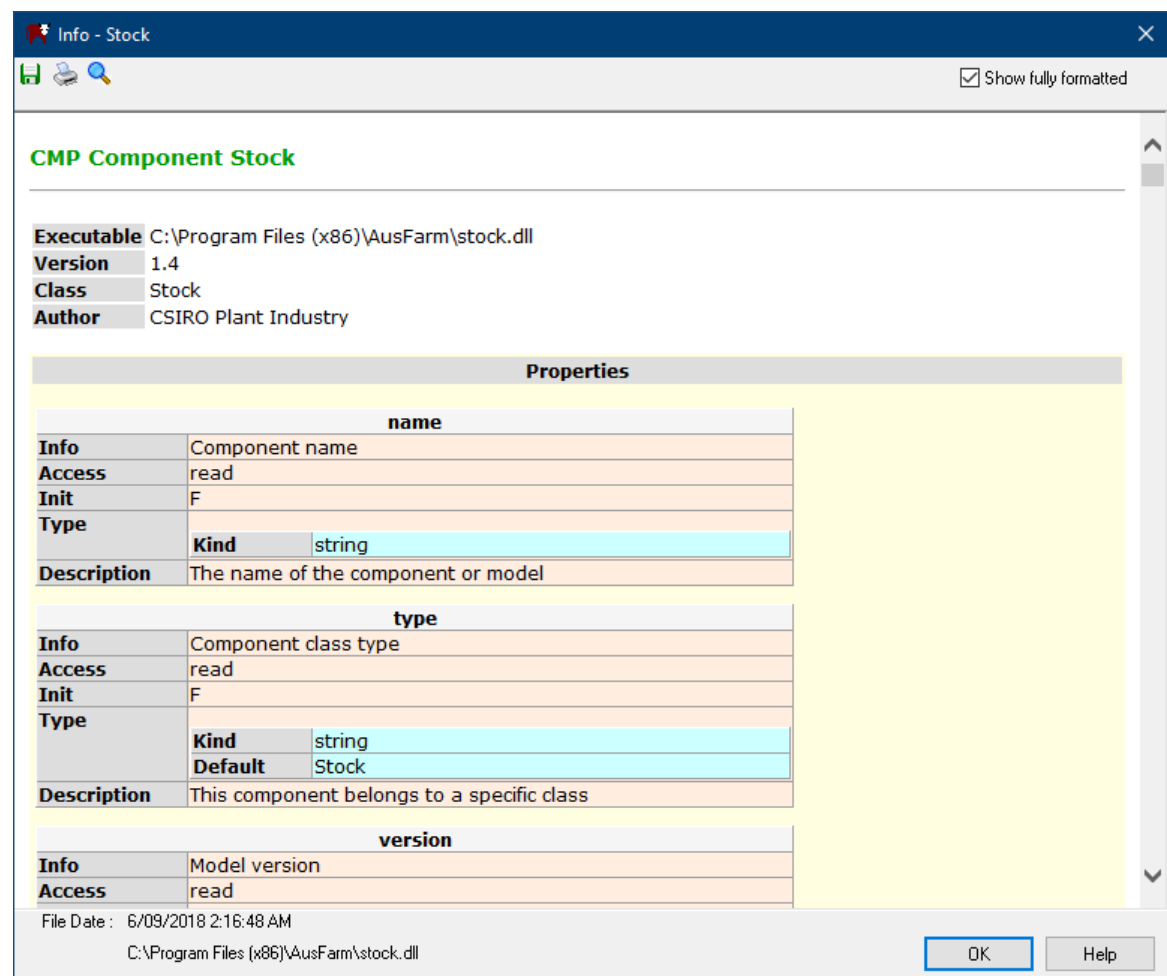
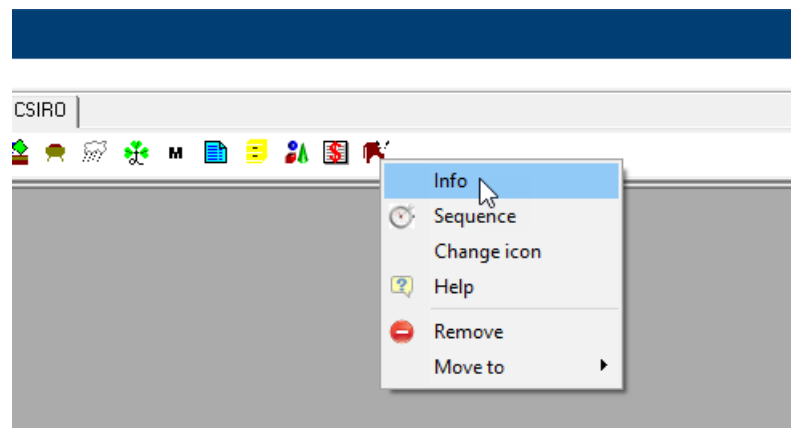
These events are called "sequenced events" because the order in which they are computed can affect the simulation's results. For example, if a Weather module executes the logic that reads in the temperature data for a day after a plant growth module uses the temperatures to compute growth of the plants, the resulting growth rates will be different to those obtained if the weather data are read in before the growth computations.

The order of computations within each time step is known as a simulation's *sequencing*. The order is expressed by assigning a positive integer value (its *ordering*) to each sequenced event in a simulation: an event with a lower ordering is executed before one with a higher ordering. The order in which events with the same ordering value are executed is left unspecified. Ordering values are only meaningful relative to one another. AusFarm configures default sequencing for each simulation. This default only needs to be changed under special circumstances.

Note: In terms of their implementation as software, there is no distinction between a sequenced event and a management event, except that sequenced events may not have parameters. The distinction between them arises from the purpose of the event code.

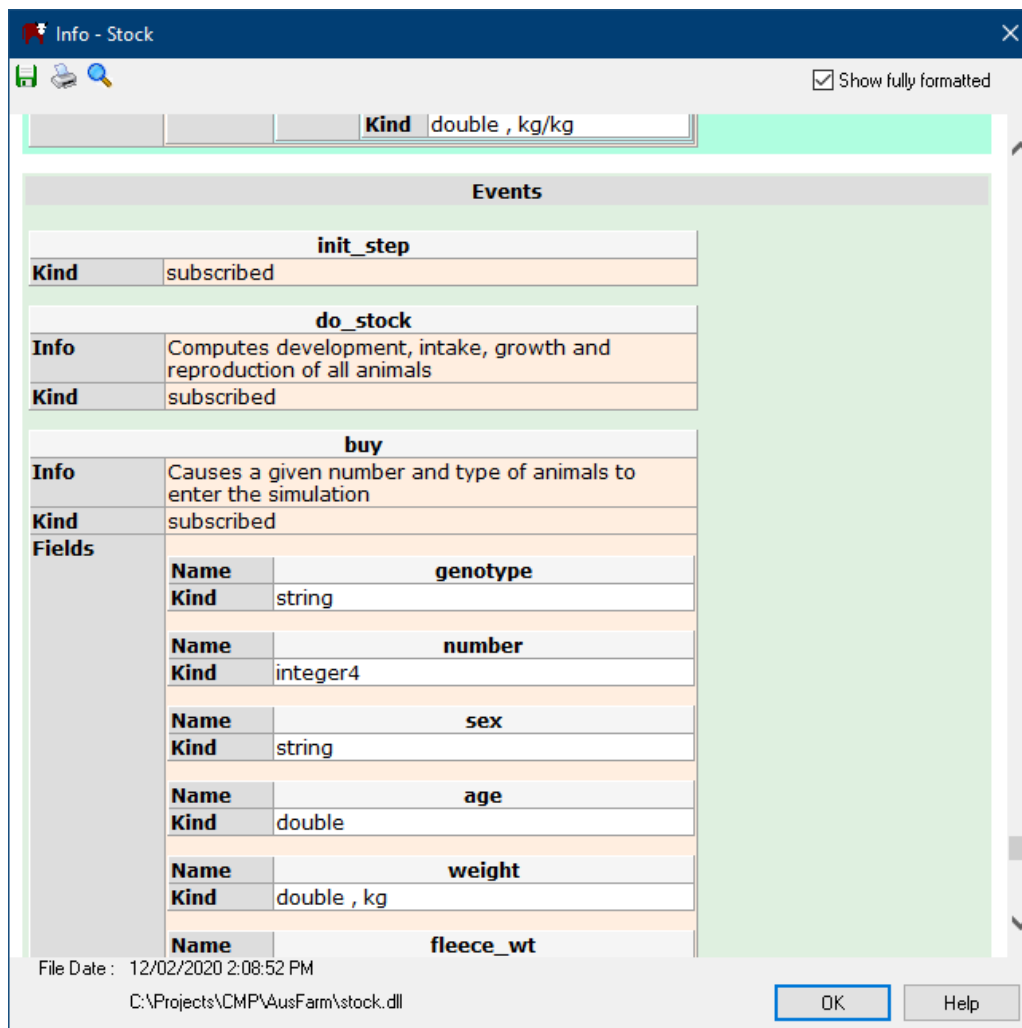
6.3 Discovering the Variables and Events for Modules

The variables and events that can be used in Management scripts are specialised for each module in the simulation. To discover these, it is done through the component palette on the main toolbar in AusFarm. The first option to use is the **Info** menu item.

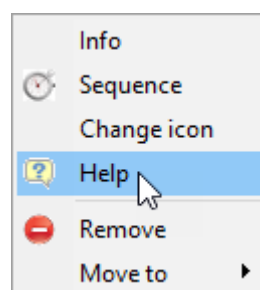


The component information dialog for the Stock component

The information dialog shows all properties and events for a chosen module. Each property will have a name, accessibility, type and description. If the property is a structure it will show the fields and their types. Events will show the parameters used to pass values to the module.



Another way to get access to all the module detail is using the help system. From the same menu on the main toolbar the help pages can be opened.



Help

1. Purpose of Component

The STOCK component encapsulates the GRAZPLAN animal biology model, as described in:

Freer M, Moore AD & Donnelly JR (1997). GRAZPLAN: decision support systems for Australian grazing enterprises. II. The animal biology model for feed intake, production and reproduction and the GrazFeed DSS. *Agricultural Systems* 54, 77-126.

Animals represented in a component instance may be of different genotypes. In particular, sheep and cattle may be represented within a single component instance.

The animals represented by a component instance are classified into *groups*. The members of each animal group have the same genotype and age class, but may have a range of ages (for example, an animal group containing mature animals may include four-year-old, five-year-old and six-year-old stock). The members of each animal group also have the same stage of pregnancy and/or lactation; the same number of suckling offspring; and occupy the same paddock. The set of animal groups changes as animals enter and leave the simulation, and as physiological events such as maturation, birth or weaning take place. Animal groups that become sufficiently similar are merged into a single group.

Each animal group has a unique, internally-assigned integer *index*, starting at 1. Because the set of groups present in a component instance is dynamic, the index number associated with a particular group may change over time.

Each animal group is also assigned a *paddock*. The forage and supplementary feed available to a group of animals are determined by the paddock it occupies. Paddocks are referred to by name in the STOCK component. It is the user's responsibility to ensure that paddock names correspond to instances of the PADDOCK component or other sources of necessary driving variables.

Each group also has a user-assigned *tag* and *priority*, which need not be unique. Tag values are generally used to manage distinct groups of animals in a common fashion. For example, all lactating ewes may be assigned the same tag value, which may then be used in management rules that keep them grazing together. Animal groups with different tag values are not merged even if they are otherwise similar. If tag values are assigned sequentially starting at 1, they can be used to generate summary variables. Priority values are used to allocate animals to paddocks in the *draft* event.

2. Initialisation Properties

The initialisation variable set is nearly completely optional. The idea is to allow the user to specify a minimal information set as well as a maximally detailed initialisation.

Property	Type	Units	Required?	Description
<i>param_file</i>	string		No	Name of an XML file containing genotypic parameters. If the null string is specified, a default parameter set that is compiled into STOCK.DLL is used. If a file name is used, the parameters in the file modify (rather than replacing) the default parameter set.
<i>genotypes</i>	record[]		Yes	Information about each animal genotype:
<i>:name</i>	string			Name used to refer to the genotype in management events.
<i>:dam_breed</i>	string			Maternal genotype (see notes)
<i>:sire_breed</i>	string			Paternal genotype (see notes)
<i>:generation</i>	integer4			Number of generations of crossing: 0 denotes the pure-bred maternal genotype (in which case <i>sire_breed</i> is not used), 1 a first cross, 2 a second cross (75% sire:25% dam), etc.
<i>:srw</i>	double	kg		Breed standard reference weight. The default value depends on <i>dam_breed</i> and <i>sire_breed</i> .
<i>:conception</i>	double[]	-		Expected rates of conception with 1, 2 and 3 young for mature ewes or cows in average body condition, over a mating period lasting 2.5 oestrus cycles. Only the first two elements are meaningful for cattle.
<i>:death_rate</i>	double	/yr		Base rate of mortality in mature animals. Default is 0.0.
<i>:ref_fleece_wt</i>	double	kg		Breed reference fleece weight in sheep. The default value depends on <i>dam_breed</i> and <i>sire_breed</i> .
<i>:max_fibre_diam</i>	double	um		Maximum average wool fibre diameter in sheep. The default depends on <i>dam_breed</i> and <i>sire_breed</i> .
<i>:fleece_yield</i>	double	kg/kg		Clean fleece weight as a proportion of greasy fleece weight in sheep. Default is 0.70.
<i>:peak_milk</i>	double	kg		Potential maximum milk yield per head, in 4% fat-corrected milk equivalents, in cattle. Default is 20.0.

The help file for the Stock component

7. Contents of Manager Scripts

Manager component scripts are basically just a collection of statements. These statements include:

- Time specifiers
 - Prescribe when rules should be executed
 - They are optional but used frequently
- Rule statements
 - Event rules trigger simulation events
 - Assignment rules set values of variables or module properties
 - Control statements control process flow
 - Subroutine calls access shared code
- Subroutines
 - Contain common/shared code
- Event handlers
 - Executed when an event occurs in another module
- Comments
 - Always use comments! They make your scripts maintainable.

7.1 Time specifiers

Each rule statement has two main parts: a time specifier and a rule. The time specifier denotes the set of time steps on which the rule is to be evaluated. The time specifier is optional; if it is not given, the rule is evaluated on each day of the simulation.

Examples of time specifiers are:

<code>on 1 Apr 1980</code>	<i>! single date</i>
<code>each 25/7</code>	<i>! 25 July in each year</i>
<code>from start to 31 Dec 2001 repeat 7 days</code>	<i>! weekly from the start date</i>
<code>from 15 Feb to 15 Apr repeat 1 month</code>	<i>! weekly</i>
<code>on finish</code>	<i>! on the last timestep of the</i>
<i>simulation</i>	

When giving a date or day-of-year in a time specifier, the month may be given as either a month number (1 to 12) or as a three-letter abbreviation. Years should be given with four digits.

7.2 Rules

Rules come in four main types:

1. Event rules cause management events to be transmitted to the rest of the simulation.
2. Assignment rules change the value of a variable.
3. Control rules are used to control the order in which other rules are executed. There are four kinds of control rule: rule lists, conditional rules, FOR loops and WHILE loops. Every control rule contains one or more *sub-rules*, which may themselves be control rules. A rule statement may therefore be made up of a nested set of rules, as shown in the examples below.
4. Subroutine calls, used in conjunction with subroutine definitions, can be used to invoke combinations of rules that may need to be used repeatedly.

7.2.1 Event rules

An event rule is specified by giving the name of the event together with zero or more parameters, which are separated by commas. The number of parameters and their types depend upon the event (see section 16 for details). The event name may need to be *qualified* to inform the simulation which module (e.g. paddock or pasture species) it is to apply to.

It is the rule-writer's responsibility to ensure that events are specified with the correct parameters and that parameter expressions are of the correct type.

Parameter values may be given as constants, but they may also be given as *expressions* that are evaluated to provide the value of the parameter (see section 15.3).

When specifying an event, it is usually possible to give fewer parameters than set out in section 16. In this case the remaining parameters are assigned a default value which is usually zero, **FALSE** or the null string according to type.

Examples of event rules are:

```
paddock3.ryegrass.sow rate=10.0           ! Note use of qualifier
move group=2, paddock='paddock3'         ! Unqualified - only livestock have "move"
! Note the single quotes around the string
buy 'wethers', 10*paddock1.area, 18.0, 50.0 ! No parameter names-legal but difficult to
! read. Note the use of an expression in a
! parameter.
```

7.2.2 Assignments

Assignment rules change the value of a variable. This variable may be one that has been defined within the manager script (see section 3.4) or it may be one of a subset of state variables that may be reset from the manager. Assignment rules take the form

set *name* = *value* variables defined within the manager script

reset *name* = *value* state variables of other modules

where *name* identifies the variable and *value* is an expression that gives the new value for the variable.

It is the rule-writer's responsibility to ensure that the name refers to a variable defined within the management script, and that the value is of a type that is compatible with the variable to which it is to be assigned.

Examples of assignments are:

```
set    x = 10.0
set    w = w + number[i] * weight[i]      ! Part of getting a weighted average
set    pasw[i] = 0.0                      ! Assignment to an array element
reset  paddock1.clover.fertility = 0.85    ! Assignment to an external variable
```

7.2.3 List or block of rules

Lists of rules group one or more rules together, ensuring that they are evaluated in sequence. This is formed by surrounding the sub-rules with curly braces { } and separating the sub-rules, either with a semicolon or by placing them on separate lines.

An example of a list of rules is

```
{
  set x = 99.0                               ! A rule on its own line
  set g = 6; set p = 'paddock1'             ! Two rules, separated by a semi-colon
  move group=g, paddock=p                   ! Same as "move 6, 'paddock1'"
}
```

It is possible for lists of rules to be nested several levels deep; as a result it is beneficial to indent them neatly.

7.2.4 Control Statements

Conditional rules

Conditional rules take one of two forms:

```
if condition sub-rule
if condition sub-rule1 else sub-rule2
```

The “condition” is an expression that evaluates to a logical value (**TRUE** or **FALSE**). When the rule is evaluated, the value of the condition is computed. If it is true, then the first sub-rule is evaluated. Otherwise, if the **else** keyword and second sub-rule have been given, the latter is evaluated instead. If the condition is false and there is no second sub-rule, then the manager moves on to the next rule.

Expressions with numeric values can be used as the condition in a conditional rule. In this case, any value other than zero is taken to mean TRUE and a zero value is taken to mean FALSE, in accordance with the type-conversion rules.

If the first sub-rule is not a list rule, it must be placed on a new line.

Examples of conditional rules are:

```
if x < 10.0 { set x = 10 }           ! Same as "set x = max( x, 10.0 )"
if b                                 ! OK to put sub-rules on a new line
  paddock1.water.irrigate amount=20.0 ! This is sub-rule1
else
  paddock1.water.irrigate amount=10.0 ! This is sub-rule2
if sheep.tag_no[i] = 1              ! Here the sub-rule is a list rule
{
  sheep.shear group=i               ! Sell animal group "i" off-shears
  sheep.sell  group=i, number=sheep.number[i] ! Neatly indented...
}
```

FOR loops

FOR loops take the form:

for *variable* = *start* **to** *end* *sub-rule*

In this rule, *variable* must be an integer variable defined within the manager script and *start* and *end* are expressions that should evaluate to integer values. When the rule is evaluated, the values of *start* and *end* are evaluated. The sub-rule is then evaluated repeatedly, with the nominated variable set in turn to each of the values *start*, *start*+1 ... *end*.

- If the sub-rule is not a list rule, it must be placed on a new line.
- If the value of start is greater than the value of end, the sub-rule is not evaluated.
- It is inadvisable to set the value of the control variable within a FOR loop.
- At the end of the FOR loop, the value of the control variable will be set to *end*+1.

The examples of the FOR loop shows how to handle two common situations:

- ⇒ the case where a task must be performed for each group of animals in a Stock module
- ⇒ the calculation of a summary variable from one or more arrays.

```
for i = 1 to animals.no_groups
  sheep.move group=i, paddock='paddock2'
set pasw = 0.0
```

```

for i = 1 to no_layers                                ! Will only work for a single-paddock
{
  set layer_asw = max( 0.0, sw_dep[i]-ll15[i] ) ! system (variables are unqualified)
  set pasw = pasw + layer_asw                  ! Sum over layers of "layer_asw"
}

```

WHILE loops

WHILE loops take the form:

```
while condition sub-rule
```

The condition is an expression that evaluates to a logical value. When the rule is evaluated, the value of the condition is computed. If it is **TRUE**, then the sub-rule is evaluated. The condition is then evaluated once more, and if it is still **TRUE**, then the sub-rule is evaluated again. The sub-rule is repeated until the condition evaluates to be **FALSE**. If the condition is **FALSE** when it is first evaluated, the manager moves on to the next rule.

It is the rule-writer's responsibility to ensure that the sub-rule will eventually cause the condition to become FALSE. If not, the loop will continue to be evaluated indefinitely and the program will have to be terminated from the Task Manager.

Because of the above, it is usual for the sub-rule in a WHILE loop to be a list rule.

If the sub-rule is not a list rule, it must be placed on a new line.

An example of a WHILE loop is:

```

set i = 10
while i > 0
{
  set x = x + i
  set i = i - 2    ! Change a term in the condition...
}

```

7.2.5 SUBROUTINE calls

Subroutines may be defined which allow a group of rules to perform a specific task while remaining relatively independent of other portions of the code. Parameter lists may be used to transfer values to a subroutine; within the subroutine, the parameters are treated as `const` variables. Although one subroutine may call another, recursion is not supported (that is, a subroutine may not call itself). Rules within subroutines may access variables defined within the manager script and "external" variables from other modules, just as ordinary rules may do. Additional variables may be defined within a subroutine; such variables have "local" scope and may be used only within the subroutine where they are declared.

SUBROUTINE definitions take the form:

```
subroutine subroutine-name (parameter-list) { rule-list }
```

Calls to a subroutine take the form:

```
call subroutine-name parameters
```

An example of a subroutine definition and subsequent call is

```
subroutine join_ewes (ram_breed: string; no_days: integer)
{
  define integer group          ! A variable of local scope, used as a loop counter
  for group = 1 to animals.no groups
  if (animals.tag_no[group] = MATURE_EWE)
    animals.join group=group, mate_to=ram_breed, mate_days=no_days
}

each 1 Mar
  call join_ewes ram_breed = 'Small merino', no_days = 30 ! Parameters values are passed
                                                         ! to a subroutine by using
                                                         ! the same syntax as event
rules
```

7.2.6 Indirection

Indirection is useful for referring to entities such as modules or module properties or events based on a list of text values. The @() operator converts a text string into a reference to a module or property.

In event names:

```
@(module-name-expression).event
```

In expressions:

```
@(variable-name-expression)
```

Indirection is almost always used inside an iteration and/or a conditional statement that provides the context.

Selecting a module in a paddock in order to perform an event on it

```
padd_name[1] = 'paddock1'
padd_name[2] = 'paddock2'
padd_name[3] = 'paddock3'

for padd = 1 to 3
  @(padd_name[padd]).grass.kill propn_herbage=1.0, propn_seed=0.0
```

Building arrays of summary variables across paddocks

```

for padd = 1 to no_paddocks
{
  set padd_deep_drain[padd] = @(padd_name[padd]&'.water.model.drain')
  set padd_cover[padd]      = @(padd_name[padd]&'.cover_tot')
  set farm_area = farm_area + @( padd_name[padd] & '.area' )
}

```

7.2.7 Event Handlers

By default, management rules are evaluated at each time step of the simulation, but it is also possible to define sets of rules which are evaluated in response to *events* issued by other components within the simulation. Data associated with the triggering event are passed to the handler via a parameter list. Units of measurement may be specified for each parameter. The declared data types and units of parameters must be compatible with those provided by the component sending the event.

EVENT HANDLER definitions take the form:

```
on_event event-name (parameter-list) { rule-list }
```

Here is an example of an event handler:

```

define real avgt
define real peak_radn = 0.0

on_event Weather.newmet (today:real; radn:real 'MJ/m^2'; maxt:real 'oC'; mint:real 'oC';
rain:real 'mm'; vp:real 'hPa')
                                ! NOTE: The entire parameter list must be on a
                                ! single line. It is shown here as wrapped only to
                                ! allow it to fit within the page

{
  set avgt = (maxt + mint)/2.0    ! Calculate a daily mean temperature in response
to a                             ! 'newmet' event
  set peak_radn = max(radn, peakradn) ! Keep track of the maximum radiation received
                                ! on a single day
}

```

7.2.8 Comments

Use comments to clarify your intentions in the script. Make your code timeless!

```

/*=====
Sale of lambs.
-----
There are 2 lamb sale policies considered:
1 = sell all lambs at a target weight, or at a final date
2 = sell lambs when their rate of weight gain falls below a threshold

=====
Sale policy #1
=====*/

```

Comments can be multiline as above, end of line or embedded in code.

Multiline comments are wrapped by `/* */` characters. Use this technique for embedding comments in the middle of lines also.

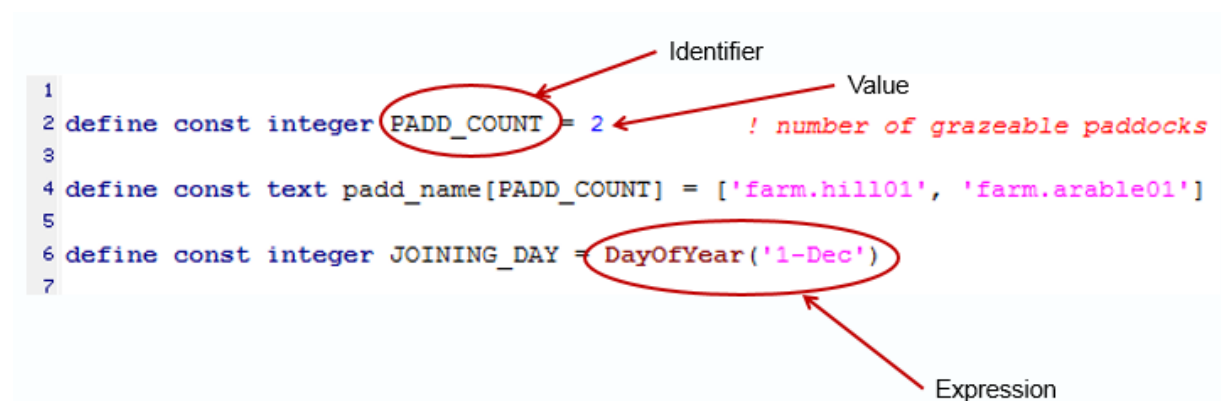
7.2.9 Manager Script Expressions

Scripts contain lines of code that are called expressions. These expressions can include *constants, variables, operators, and functions*.

- Constants
 - Identifiers that have fixed values
- Variables
 - Identifiers that are changeable in value
- Operators
 - Perform arithmetic processes, relational tests, logical joins
- Functions
 - Access to common expressions and algorithms

7.2.10 Constants

These identifiers are set once, and their value remains constant.



The diagram shows a script with three lines of code defining constants. Red circles and arrows highlight specific parts of the code:

- Line 2: `define const integer PADD_COUNT = 2`. A red circle highlights `PADD_COUNT`, with an arrow pointing to it labeled "Identifier". Another red circle highlights `2`, with an arrow pointing to it labeled "Value". A red comment `! number of grazeable paddocks` is on the same line.
- Line 4: `define const text padd_name[PADD_COUNT] = ['farm.hill01', 'farm.arable01']`. A red circle highlights `PADD_COUNT` in the array index.
- Line 6: `define const integer JOINING_DAY = DayOfYear('1-Dec')`. A red circle highlights `DayOfYear('1-Dec')`, with an arrow pointing to it labeled "Expression".

- Real-valued constants may be given in decimal or exponential format (e.g. 1.34, 6.77E-2).
- Integer-valued constants are given in decimal format (e.g. 999, -1).
- Text strings are always surrounded by single quotes (e.g. 'Hello, world'). The quotes distinguish a text string from a reference to a variable.
- Logical constants are given as TRUE or FALSE (case-insensitive).
- It is possible to have constant values that are arrays. Each element of an array should have the same type. To denote an array, surround it with square brackets `[]` and separate each element with a comma:

```
define const REAL_ARRAY = [ 0.5, 0.6, 0.7, 0.8, 0.9 ]
define const FOX_ARRAY  = [ ['quick', 'brown', 'fox'],
                             ['jumps', 'over', 'the', 'lazy', 'dog'] ]
```

Note that in the second example, the array elements are themselves arrays, making a two-dimensional array. (Note also that the sub-arrays need not be of the same length!)

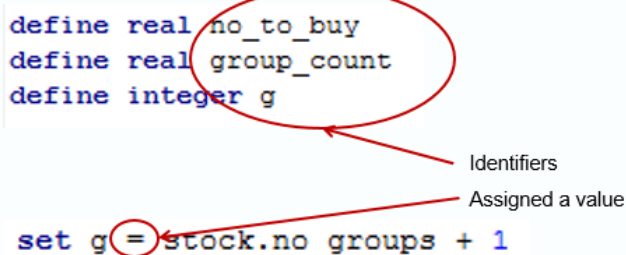
A value may also be a *structure*, i.e. a collection of named sub-values called *fields*. To denote a structure, surround it with brackets `()`, precede each field with its name followed by a colon, and separate fields with semi-colons:

```
define const STRUCT_VAR = ( text_field: 'quick'; real_field:99.9;
                           array_field:[1,2,3,4])
```

Typically, in an AusFarm script constants are written in all upper case to make visual recognition easier.

7.2.11 Variables

Variables in expressions are referred to by name. A variable may be defined within the manager or it may be any variable that can be accessed from the rest of the simulation. Variable names may need to be qualified in the same way as event names.



```
define real no_to_buy
define real group_count
define integer g

set g = stock.no_groups + 1
```

The diagram shows the code above with red annotations. A red circle highlights the identifiers `no_to_buy`, `group_count`, and `g`. A red arrow points from the text "Identifiers" to this circle. Another red circle highlights the assignment `g =` in the `set` statement, with a red arrow pointing from the text "Assigned a value" to it.

The elements of an array variable or the fields of a structure variable can be referred to using the forms `array[index]` and `structure:field` respectively. Since elements in arrays and fields in structures may themselves be arrays or structures, these references may be nested:

```
seeds[3]:unripe soft      ! Field "unripe soft" within element 3 of "seeds"
foo[i,j]                  ! Same as foo[i][j]
```

7.2.12 Operators

The following operators may be used in expressions:

Arithmetic operators

+	Addition	Numeric (integer if both arguments are integer, real otherwise)
-	Subtraction	Numeric (integer if both arguments are integer, real otherwise)
*	Multiplication	Numeric (integer if both arguments are integer, real otherwise)
/	Division	Real
^	Power	Real
mod	Modulus (remainder)	Integer
div	Integer division	Integer

Relational operators

=	Equal to	Logical (TRUE or FALSE)
/=	Not equal to	Logical
<	Less than	Logical
>	Greater than	Logical
<=	Less than or equal to	Logical
>=	Greater than or equal to	Logical

Logical operators

and	And	Logical
or	Or	Logical
not	Not	Logical

Text operator

&	Concatenation	Text
---	---------------	------

Brackets () may be used to govern the order of evaluation of operators.

It is the rule-writer's responsibility to avoid invalid arithmetic operations such as divisions by zero.

```
if age[g] >= 365 * cfa_years
```

Logical operators are frequently used in conditional statements like the above test.

7.2.13 Functions

A variety of functions are also defined for use in expressions. Arguments to functions may themselves be expressions which are separated by commas, as in the following examples:

```
max( -3, min( 3, i ) )
upper('abc')
average( x[1], x[2], x[3], x[4] )
average( x )    ! Same as the previous one
```

Arithmetic functions		Returns
max()	Maximum	Integer if all arguments are integer, real otherwise
min()	Minimum	Integer if all arguments are integer, real otherwise
sum()	Total	Integer if all arguments are integer, real otherwise
Average()	Arithmetic mean	Real
exp()	Exponential (e ^x)	Real
ln()	Natural logarithm	Real
sin()	Sine	Real
cos()	Cosine	Real
atan()	Arctangent	Real
round()	Round to nearest integer	Integer
floor()	Integer next below	Integer if argument is integer, real otherwise
Text functions		Returns
max()	Maximum	Text

min()	Minimum	Text
upper()	Uppercase	Text
lower()	Lowercase	Text
str()	Convert the value to text. Optional second argument can be a format string or an integer. An integer will specify the number of decimal places to display in the converted value. Format string: '4.3f' or '2d' where d is used for representation of integer arguments and f for floating point values.	Text
length()	Count the number of elements in an array.	Integer

Date functions		Returns
dayofyear()	<p>Return the day number of the year where Jan 1 = day 1.</p> <p>e.g.</p> <p>dayofyear('1-Jul')</p> <p>dayofyear('Dec-31')</p> <p>dayofyear('12 Aug 1961')</p> <p>Where delimiters can be '-' or '' or '/'</p> <p>Month names must be the first three characters from the English month name.</p> <p>The Year must be four digits. Where the year is not specified a non-leap year is assumed.</p>	Integer
datewithin()	<p>Checks if the day number is between two dates.</p> <pre> define logical inperiod = FALSE define integer istart = dayofyear('15-Dec'); define integer iend = dayofyear('15-Feb'); set inperiod = datewithin(day, istart, iend) </pre>	<p>Returns true if the date is between either date or equal to either date.</p> <p>Allows for the start and end period including 1 Jan (wrapping over the start of the year).</p>
Other		
resize()	<p>Resize an array</p> <p>Examples:</p>	<p>The array with the new size.</p> <p>New scalar elements will be</p>

```
set array1 = resize(array1, 5)
```

initialised to zeroes or empty strings.

```
! where array3 is a two dimensional array
set array3[1] = resize(array3[1],
Length(array3[1]) + 1)
```

pos()	Checks the array or string for the position of an item.	An integer value specifying the index. Zero is returned if the item is not found.
	<pre>define text array[3] = ['one', 'two', 'three'] set p = pos('three', array) ! 3 define double darray[3] = [0.1, 0.11, 0.111] set p = pos('0.1', darray) ! 1 define text astring = 'lolg text form' set p = pos(1, astring) ! 3 define integer iarray[3] = [1, 11, 111] set p = pos(11, iarray) ! 2</pre>	

If the expression parser encounters an argument to a function or operator that is not of the required type, it will attempt to convert it according to the following rules:

From	To	
Real	Integer	The value is rounded off.
Real	Logical	The value will be converted to TRUE if non-zero and to FALSE if zero.
Real	Text	If the absolute value is less than 0.000001, the value is converted to a string using exponential format (e.g. 1.23763567E-8). Otherwise it is converted using decimal format, with enough decimal places to ensure that 6 significant figures are displayed. At least one decimal place is always given.
Integer	Real	The same value is returned.
Integer	Logical	The value will be converted to TRUE if non-zero and to FALSE if zero.
Integer	Text	The value is converted to its decimal representation.

Text	Real	The string value will be parsed into a number. If it cannot be parsed, the simulation will halt.
Text	Integer	As for text-to-real conversion.
Text	Logical	The value will be converted to TRUE if the string equals ' true ' (case-insensitive) and to FALSE otherwise.
Logical	Real	TRUE is converted to 1 . 0 and FALSE to 0 . 0.
Logical	Integer	TRUE is converted to 1 and FALSE to 0.
Logical	Text	TRUE is converted to ' true ' and FALSE to ' false '.

Variable values are obtained from the rest of the simulation as the expression of which they form a part is evaluated. As a result, if a variable value changes in response to a manager event, its value in any expressions evaluated subsequently will be the altered value, even within the same time step.

7.3 Definition statements

As noted above, expressions may use variables that are defined as part of the manager script. Before such variables are used, however, they must be defined using a *definition statement*.

A definition statement begins with the keyword `define`, followed by one or more variable definitions separated by semi-colons. Each variable definition consists of a variable name, which may be preceded a type specifier, and followed by an initial value for the variable. The definition may further be preceded by one of the qualifiers `const` or `volatile`. If the `const` qualifier is present, the variable must be assigned an initial value; otherwise, the type specifier and initial value are optional. Any subsequent attempt to modify the value of a variable defined with the `const` qualifier is regarded as an error. The `volatile` qualifier is used to indicate that the value of the variable may be set by other components.

An initial value for a defined scalar variable can be the result of an expression. Using an expression such as a function like `DayOfYear()` is a typical use. See an example below.

Type specifiers are made up of one of the keywords `real`, `integer`, `text` or `boolean`, optionally followed by one or more array lengths surrounded by square brackets and separated by commas (see below for examples)

It is an error to define the same variable name more than once within a single manager script. The only exception is that a “local” variable may be declared within a subroutine using the same name as a variable outside the subroutine. When this occurs, all references to that variable name within the subroutine refer to the “local” variable.

If a variable name is defined within the script that is the same as an “external” variable, the name will be taken to refer to the manager variable when it is used in expressions.

If the type specifier is omitted, the type is inferred from the initial value. If no initial value is given, the type of the variable is taken to be the same as that of the preceding variable in the list of definitions. For the first variable in a list, it is taken to be a real number.

The initial value is preceded by an equals sign. It is specified in the same way as a constant in an expression (see above).

If no initial value is provided, then the initial value of the variable is set as follows:

- ⇒ numeric values are set to zero
- ⇒ text values are set to the empty string
- ⇒ logical values are set to **FALSE**.

Here are some examples of definition statements:

```
define x = 0.0
define x                                     ! Same as the previous definition

define integer i; j; k                      ! All are integers

define integer m; n; text t                 ! Different types within one statement

define breed = 'Angus'                     ! Text variable (type inferred from initial
value)
define volatile z = -999                    ! Integer variable (type inferred from value)
                                           ! which may be set by other components
define const real pi=3.1415926              ! Real constant

define const integer sow_date = DayOfYear('1-May')

define real x_arr[20]                       ! Array of real numbers
define real y_arr = [9.0,8.0,7.0,6.0]       ! Also an array of real numbers

define real array2d[100,100]                ! 2-dimensional array

define struct = (a:9.0; b:5; c:'string')    ! A structure has to be defined using an
initial                                     ! value
```

7.3.1 Advanced initialisation of variables

Variables and constants can be initialised with the results of expressions.

For example:

```
15
16 define text    file_prefix  = 'c:\temp\SimpleMixed'
17 define real    file_version = 1.0
18
19 define const string file_name_base = file_prefix & ' v' & str(file_version,'3.1f')
20
21 define const integer join_start = DayOfYear('1-Feb') ! Joining day-of-year for sheep (1 Feb)
22
```

When initialising complex variables, it is now possible to use expressions to set values. This can be done within arrays of records.

These are valid:

```
4 define test = (field1: [sin(20), 901 * 34.56] ; field2: 'Mathematical')
5 |
6 define test2 = (field1: [DayOfYear('1-May'), 222] ; field2: (subfield1: cos(45); subfield2: 3.2 ))
7
```

When initialising an array, each successive element will be assumed to follow the type of the first element. For example:

```
2
3 define test_array = [93.23, 3, 5]      ! floating point values
4
5 define test_array2 = [56, 34.012, 4]  ! this is invalid as the second element cannot be stored as an integer
```

Arrays can also be initialised using a constant multiplier:

```
set number_purchased = [0] * NUM_TAGS

set paddocknames = ['-'] * COUNT

define dasharray = ['-'] * 3] * 2
```

Is the same as:

```
define dasharray = ['-','-','-','-','-','-','-']
```

This syntax can be used for numeric and text arrays.

7.3.2 Using constants as array size specifiers

When defining the size of arrays it requires the integer value of the number of elements. If an integer constant is declared in the script previously then this constant can be used in the place of the literal integer.

```
define const integer padd_count = 10      ! Integer constant
define string paddock_names[padd_count]  ! Use integer constant as array size
```

7.4 Examples of complete statements

Here are some valid manager statements for a simulation with components called **paddock1**, **soilwater**, **subclover**, **ryegrass**, and **merinos**:

```

define x = 100; y; z                                ! y & z are initialised at zero
define text nextpadd
define some_sw = [0.10,0.12,0.15,0.22,             ! Array of real numbers, split over two lines
                  0.30,0.30,0.30,0.34]

from start repeat 1 months                          ! Another rule would set "nextpadd"
{ merinos.move nextpadd }

on 1 apr 1980                                       ! Use a defined variable to trigger an event
{
  set z = subclover.green_dm + ryegrass.green_dm
  if z > 700
  {
    merinos.buy sex='wethers', number=10*paddock1.area, age=18.0, weight=50.0
    merinos.move group=merinos.nogroups, paddock='paddock1'
  }
}

each 15 dec                                       ! Shear all sheep at least a year old
  for i = 1 to merinos.no_groups
    if merinos.age_months[i] > 12
      merinos.shear group=i

paddock1.soilwater.irrigate amount=pet - rain    ! Daily irrigation

```

A more complete set of examples can be found in the AusFarm User Notes documents.

7.5 Management Events Summary

Stock component

buy	Buy animals into the simulation
sell	Sell animals out of the simulation
shear	Shear (sheep only)
join	Commence mating
castrate	Castrated unweaned male lambs or calves
wean	Wean some or all unweaned lambs or calves
dryoff	End lactation in cattle
move	Assign a group of animals to a paddock
split	Divide a group of animals into two groups
tag	Assign a "tag value" to a group of animals
Sort	Sort the list of groups of animals by tag value

Supplement component

buy	Purchase supplement
feed	Place supplement in a paddock
reset	Removes all residual supplement from a paddock

Soil Water component

irrigate	Add irrigation water to the soil
----------	----------------------------------

Pasture component

sow	Sow seed of the pasture species
spraytop	Crude analogue to spraying this species with glyphosate
kill	Kill herbage of this species only
cultivate	Incorporates herbage and seeds into the soil
conserve	Removes herbage and (optionally) stores it in a Supplement module

Cashbook component

earn	Acquire cash
spend	Spend cash
report	Write a gross margin report

8. Writing management scripts

A management script is made up of a collection of statements. Most statements define *rules*. At each time step, each rule statement in the script is evaluated to determine whether any management *events* should be issued to the rest of the simulation for processing.

The order in which the statements forming a management script are evaluated is not defined. To ensure that rules are evaluated in an order, control rules must be used (see below).

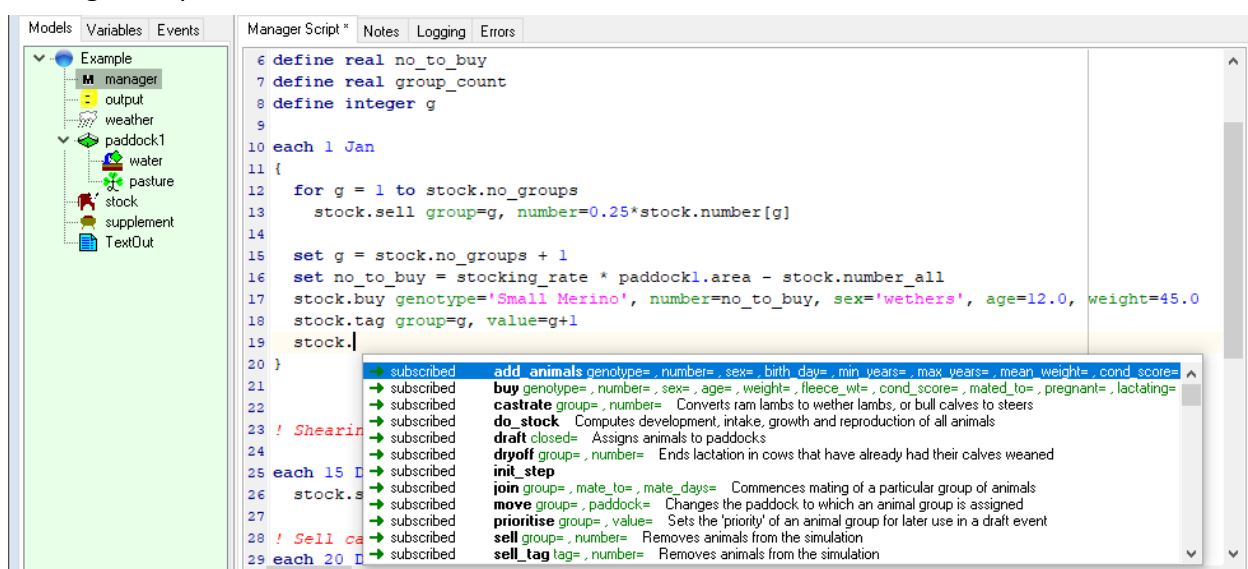
Comments may be placed in a management script, preceded by **!**. Multi-line comments or comments in midline can be enclosed using **/* */** as in the C and C# languages.

The script editor colour-codes various elements of a script to assist the user in identifying them. Keywords are shown in **dark blue**; numeric values and dates in **blue**; text values in **magenta**; event parameters in **green** and comments in **red**.

8.1 Using the script editor

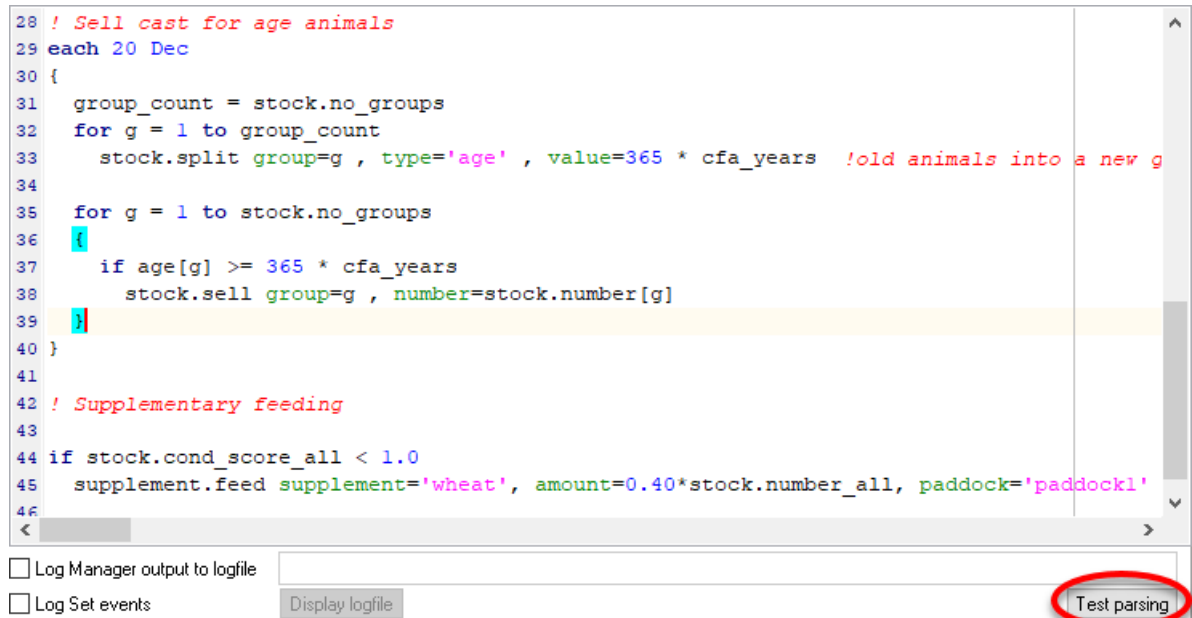
8.1.1 Code completion

When typing management script and the name of one of the components is followed by a period, by waiting for around one second a window will popup showing a list of properties and events that belong to this component. In the example below you can see a list of the events that can be triggered in the stock component. Highlight the preferred event in the list using the up or down arrow keys on the keyboard or use the mouse cursor to select it. By then pressing enter on the keyboard it will be inserted into the Manager script.



8.1.2 Matching braces

To assist with formatting the Manager script correctly the editor shows clearly the matching braces in the script. When the cursor is placed on a [, {, or (type of braces the corresponding one is also highlighted. As shown in the figure below.



8.1.3 Checking the script

After writing a section of Manager script it is useful to check that it is written in a well-formed manner. At the bottom of the Manager Script tab is a **Test parsing** button that can be used to start a syntax check of the script. This option will run some initial tests and alert you to any obvious problems before doing a run of the simulation. This option is highlighted in the figure above.

8.1.4 Bookmarks

To set bookmarks in the script there is a keyboard combination that performs this task. To set a bookmark use the key combination, CTRL + Shift + 1. When a bookmark is set you will see a small number icon in the left-hand gutter of the editor. To unset the bookmark, ensure the cursor is on the line of the bookmark and use the same key combination. You can have up to nine numbered bookmarks on each Manager editor. Just use the CTRL + Shift + *number* combination for any extra bookmark.

Once a bookmark has been set in a script, it is easy to go to that line at any time using the key combination CTRL + *number*.

Bookmarks are shown in the following figure.

```

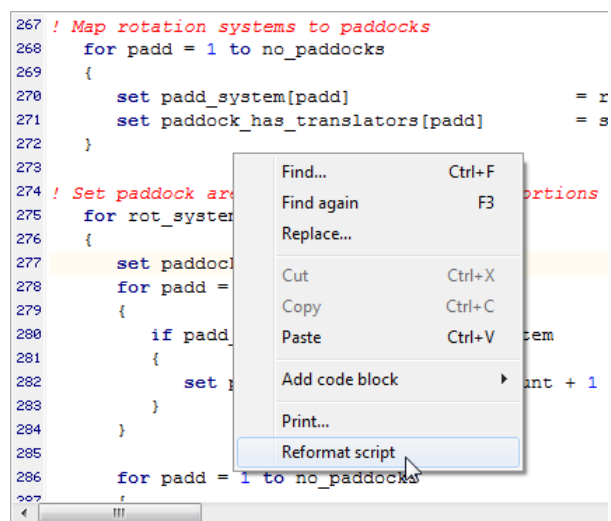
1 define real stocking_rate = 10.0 ! wethers/ha
2 define real cfa_years = 5.0
3
4 ! Replacement
5
6 define real no_to_buy
7 define real group_count
8 define integer g
9
10 each 1 Jan
11 {
12   for g = 1 to stock.no_groups
13     stock.sell group=g, number=0.25*stock.number[g]
14
15   set g = stock.no_groups + 1
16   set no_to_buy = stocking_rate * paddock1.area - stock.number_all
17   stock.buy genotype='Small Merino', number=no_to_buy, sex='wethers', ag
18   stock.tag group=g, value=g+1
19   stock.move group=g, paddock='paddock1'
20 }

```

8.1.5 Reformatting Management Scripts

Manager scripts can often get untidy and loss of indentation can make them difficult to read. An option to reformat the script based on some rules is available when you right click on the script editor. When the script is reformatted, the curly braces will appear on their own lines. Indentation will be adjusted with indents of three characters per indent. Comments will not be adjusted unless they are on the end of a line.

A selected section of script can also be reformatted.



8.1.6 Inspecting the Management script

While the simulation is running it is possible to log many of the functions performed by the Manager. By ticking the two check boxes at the bottom of the Manager Script tab and providing a filename for the log, a list of management details will be saved to file. This is extremely useful for checking that the management of the simulation is working as expected.

<input checked="" type="checkbox"/> Log Manager output to logfile	<input type="text" value="\\vmware-host\Shared Folders\Documents\AusFarm\manager.log"/>	
<input checked="" type="checkbox"/> Log Set events	<input type="button" value="Display logfile"/>	<input type="button" value="Test parsing"/>

Below is an example of the log file that is generated. It is a tab separated text file. The leftmost value is the line number from the script followed by the date of the timestep

that it was executed. The statement is then shown with the current values for any variable and parameters used on that line.

15	01-Jan-1970	set	g	=	1				
16	01-Jan-1970	set	no_to_buy	=	500.0				
17	01-Jan-1970 age= 12.0	stock.buy weight= 45.0	genotype=	small merino	number= 500.0	sex=	wethers		
18	01-Jan-1970	stock.tag	group= 1	value= 2					
19	01-Jan-1970	stock.move	group= 1	paddock=	paddock1				
26	15-Dec-1970	stock.shear							
31	20-Dec-1970	set	group_count	=	1				
33	20-Dec-1970	stock.split	group= 1	type= age	value= 1825.0				
13	01-Jan-1971	stock.sell	group= 1	number= 121.5					
15	01-Jan-1971	set	g	=	2				
16	01-Jan-1971	set	no_to_buy	=	136.0				
17	01-Jan-1971 age= 12.0	stock.buy weight= 45.0	genotype=	small merino	number= 136.0	sex=	wethers		
18	01-Jan-1971	stock.tag	group= 2	value= 3					
19	01-Jan-1971	stock.move	group= 2	paddock=	paddock1				
26	15-Dec-1971	stock.shear							
31	20-Dec-1971	set	group_count	=	2				
33	20-Dec-1971	stock.split	group= 1	type= age	value= 1825.0				
33	20-Dec-1971	stock.split	group= 2	type= age	value= 1825.0				
13	01-Jan-1972	stock.sell	group= 1	number= 90.25					
13	01-Jan-1972	stock.sell	group= 2	number= 33.25					
15	01-Jan-1972	set	g	=	3				
16	01-Jan-1972	set	no_to_buy	=	129.0				
17	01-Jan-1972 age= 12.0	stock.buy weight= 45.0	genotype=	small merino	number= 129.0	sex=	wethers		
18	01-Jan-1972	stock.tag	group= 3	value= 4					
19	01-Jan-1972	stock.move	group= 3	paddock=	paddock1				
45	19-Aug-1972 paddock1	supplement.feed	supplement=	wheat	amount= 198.0	paddock=			

9. Introduction to Livestock Management and the Stock component

Livestock are supported in AusFarm using a Stock component. Here are some of the features of the AusFarm Stock component.

- One Stock component contains all the animal groups in the simulation. The component manages the lifecycle of the animal cohorts that can be located on various paddocks in the simulation.
- Each animal group has its own status at any point in the simulation.
- The animal groups can be located on any number of paddocks in the simulation. They can be moved when required to do so.
- An animal group may include a range of ages.

For a more complete summary of livestock management in AusFarm see the **AusFarm User Notes #2** document.

9.1 Stock component

Usually a single STOCK module is added to an AusFarm simulation at the top level in the module hierarchy.

In a grazing system there may be a variety of different classes of livestock. Animals may be of different genotypes (including both sheep and cattle); may be males, females or castrates; are likely to have a range of different ages; and females may be pregnant and/or lactating. The set of classes of livestock can change over time as animals enter or leave the system, are mated, give birth or are weaned. Further, animals that are otherwise similar may be placed in different paddocks, where their growth rates may differ.

Below is a representation of some animal groups managed by a Stock component.

Main Flock or Herd					
Index	1	2	3	4	5
Number of animals	2000	160	40	1500	300
Genotype	Merino	BL x Merino	BL x Merino	BL x Merino	BL x Merino
Sex	Wethers	Ewes	Ewes	Ewes	Ewes
Age	1.4 years	4.4 years	1.4 years	4.4 years	1.4 years
Base Weight	53.5 kg	51.4 kg	47.4 kg	48.6 kg	47.2 kg
Fleece Weight	2.23 kg	2.06 kg	1.78 kg	2.04 kg	1.74 kg
Number of offspring		1	1	1	1
Weight of foetus		2.7 kg	2.3 kg		
Paddock	"paddock_5"	"paddock_1"	"paddock_6"	"paddock_1"	"paddock_6"
Tag Value	2	1	1	1	1
Priority Score	3	2	1	2	1
Unweaned Offspring					
Number				1500	300
Genotype				(BL x M) x Dorset	(BL x M) x Dorset
Sex				Mixed Lambs	Mixed Lambs
Age				10 days	10 days
Base Weight				5.2 kg	4.7 kg
Fleece Weight				0.32 kg	0.30 kg

Above: The list of animal groups at a point in time during a hypothetical simulation containing a Stock module. Group 1 is distinct from the others because it has a different genotype and sex. Groups 2 and 3 are distinct because they are in different age classes (yearling vs mature). Groups 2 and 4 are distinct because they are in different reproductive states (pregnant vs lactating). Note how the unweaned lambs are associated with their mothers.

Index

- Each animal group is assigned a unique index
- The index for a group of animals can change – using the **split** event

Tag value

- Use the **tag** event to assign a value
- Used to manage distinct groups of animals together
- Assists in collecting summary information

Priority score

- Used to control the movement of animals when using the **draft** event

Typical Livestock management operations

Policy	Enterprise Type
Stocking rate and replacement	All stock enterprises
Shearing	Sheep
Reproductive management	Cowes and sheep
Sales of young stock	Cowes and sheep
Culling old stock	All stock enterprises
Supplementary feeding	All stock enterprises
Grazing management	Multi-paddock systems

10. Simulation Analyses

When the objective of a simulation study requires that many related runs be executed, and a “base case” simulation has been configured and tested, then the *analysis* facility in AusFarm is useful.

The central idea is that one or more of the modules in an AusFarm simulation can be defined to be *factors*. Each factor module has one or more sets of initialization data (known as factor *levels*, on analogy with field experiments). When a simulation is run as an analysis, every possible combination of factor level is used to automatically construct and execute a simulation.

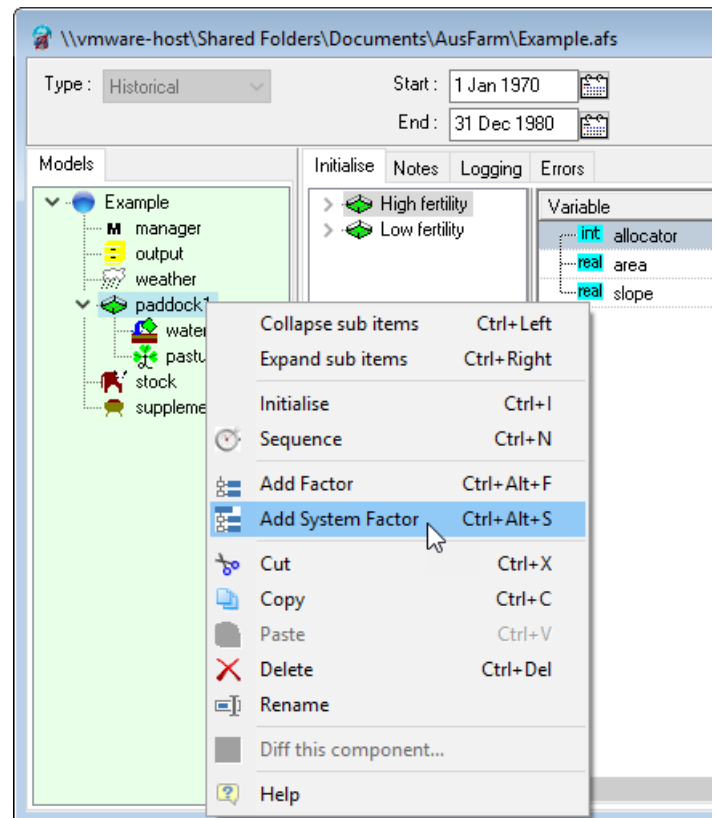
The user can also specify one or more *report templates*. Each report template describes a set of charts and tables that compare the results of the simulations in an analysis. When the analysis has been run, AusFarm uses the simulation results to generate an HTML document containing these charts and tables.

Note: For modules that are systems, it is possible to define the entire system as the factor. For example, a paddock module is a system.

10.1 Setting up analyses

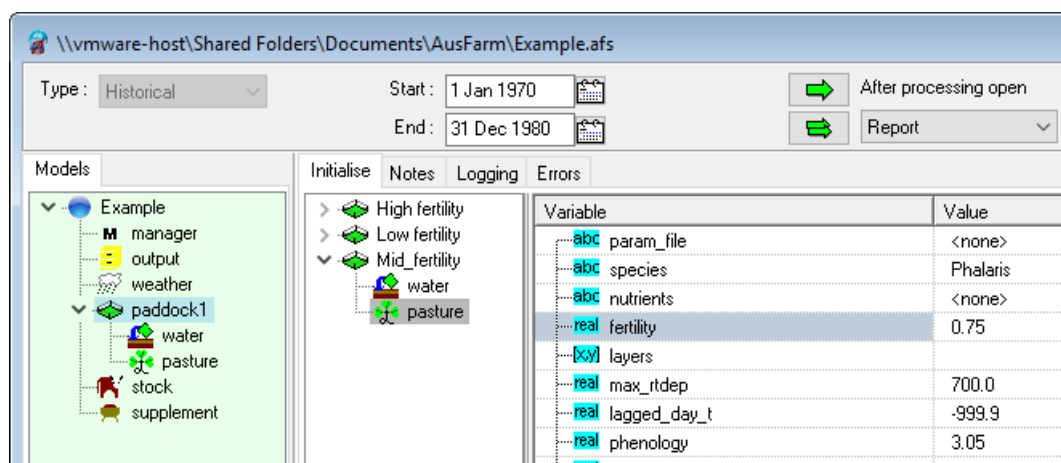
For example, to test the effect of systematically changing the characteristics of the Paddock module in a single-paddock system, open the example simulation in *example.afs*.

- ➡ Right-click on the Paddock module in the configuration tree and select the **Add System Factor** option from the pop-up menu.



Factor levels contain initialization data. This data can be modified by using the initialization dialogs or via data entry interface in the Initialise tab, just as for modules in the configuration tree.

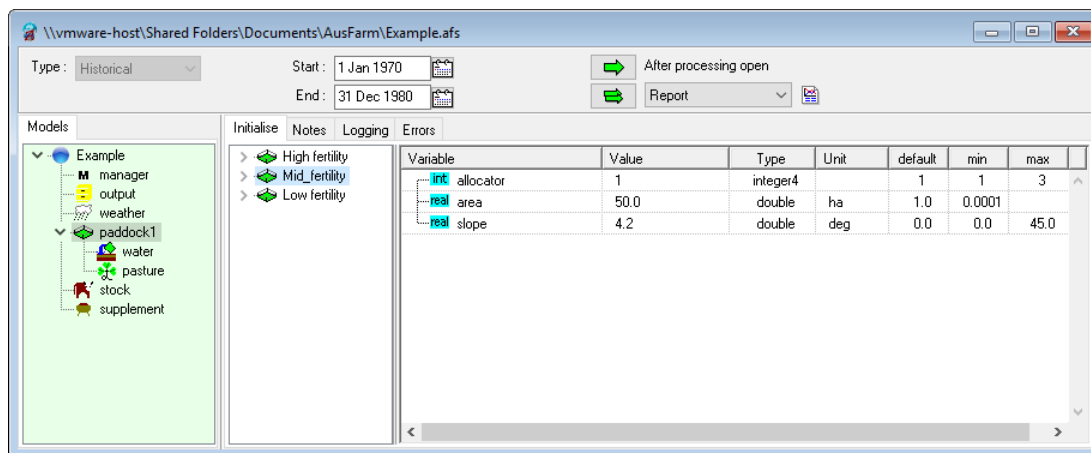
- ➡ Change the name of the new factor to **Mid fertility** and check that the fertility property of the pasture module has a value of **0.75** as shown.



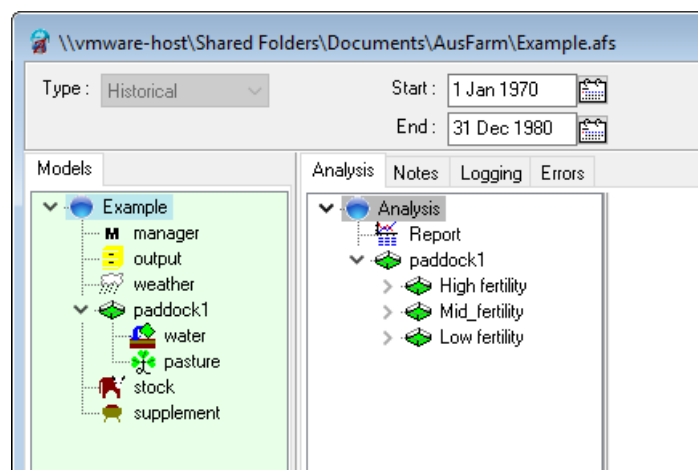
- ➡ To add an extra factor levels, either select the Add System Factor option again or right-click on one of the factor levels and select Clone from the pop-up menu.

Once a simulation contains one or more factor modules, the Run Analysis button appears in the top pane of the Simulation window. Clicking the Run button executes the base simulation,

while clicking the Run Analysis button will set up and execute one simulation for each combination of factor levels:



➡ Click on the simulation node (Example) in the model tree and you will be able to see the complete structure of the Analysis.

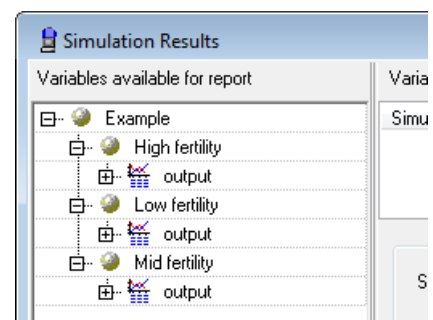


➡ Run the analysis.

If the Report has been selected for output, then the report will be displayed at the completion on the run.

➡ Close the report if it is open then open the Results window.

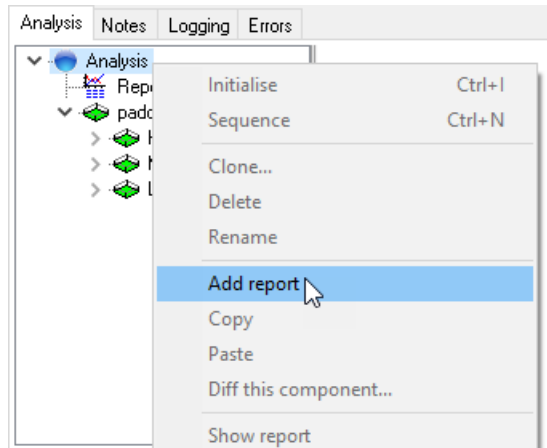
The tree of results now has an extra level; within the simulation, there is a sub-tree for each of the simulation runs that was run in the Analysis, and within each run the same set of output variables (with different values) is displayed for selection.



➡ Select the simulation node (Example) at the root of the model configuration tree. The Analysis tab will appear, showing the structure of the entire analysis (i.e. all factors and their levels).

➡ Double-click on the existing Report object. The Report Designer dialog will appear. (See the Help file for details on how to set up charts and tables in reports).

➡ Right-click on the **Analysis** item in the Analysis tab. From here you can add another report item to the Analysis if required.



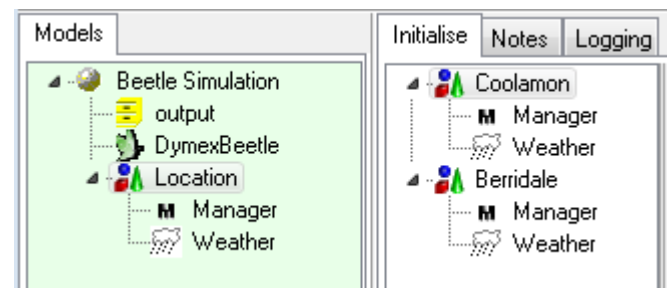
10.2 Using Generic modules as factors

Often, the factor in a simulation analysis is used control a management activity and is therefore expressed in a management script. In these cases, it can be inefficient to make the Manager module into a factor, especially if more than one factor in the simulation experiment is implemented via management rules. In these cases, an elegant solution is as follows:

1. Add a Generic module to the simulation's configuration;
2. Define a variable within the Generic module and provide an initial value for it;
3. Use this variable in the Manager script, either directly in events (e.g. a stocking rate) or in a conditional statement or statements that turn rules "on" or "off";
4. Convert the Generic module into a factor in the analysis.

When using a Generic module for this purpose, it is important to refer to the factor variable with an *unqualified* name in the manager script, e.g. `stock_rate` instead of `sr_factor.stock_rate`.

Another way to use the Generic component is as a system component. It is easy then to use this system structure as a factor value in an Analysis. In the example below the Manager script can be dependent on the location.



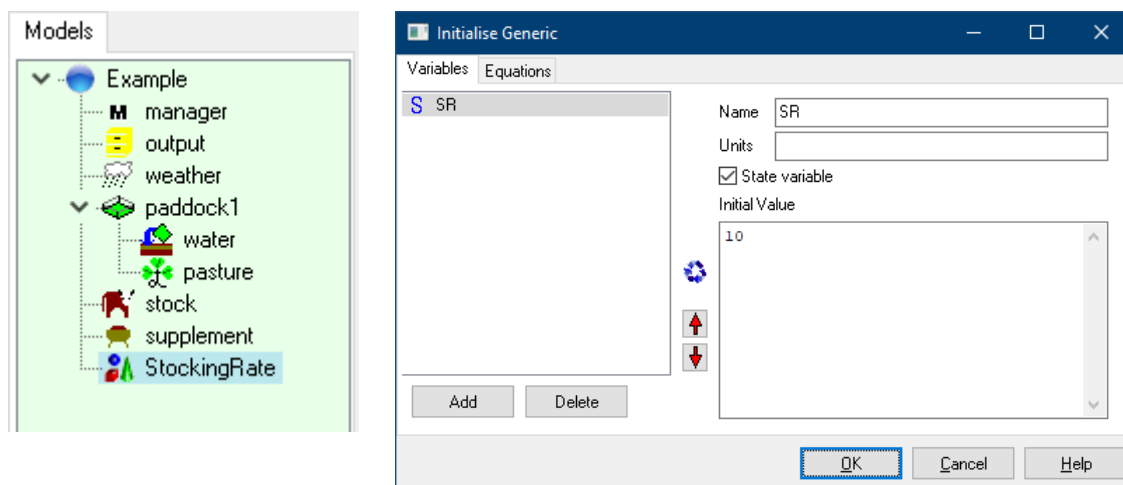
Configuring an Analysis using Generic components

As an example of how to test more than one stocking rate.

- ➡ Modify a Management script to use a variable sourced from another component. The variable used here is **SR**. Choose any name that applies to the quantity you are defining. This variable is then added to the Generic module state variables in the next step.

```
set no_to_buy = SR * paddock1.area - stock.number_all
```

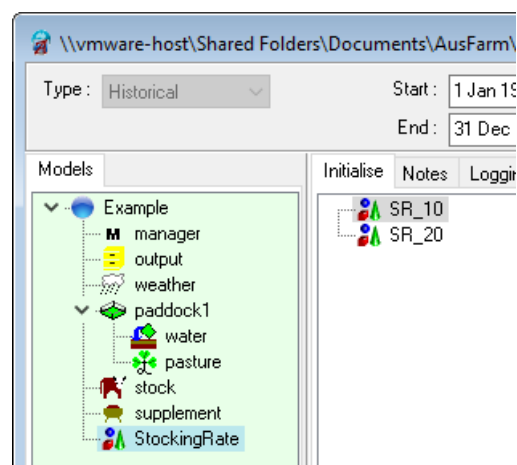
- ➡ Add a Generic component to the simulation, name it StockingRate and add a state variable to it.



It is now possible to replicate this component in an Analysis and give it differing values.

- ➡ Right click on the StockingRate generic component and choose **Add Factor**. Do this again.

- ➡ Edit the names of the components in the Analysis to reflect the internal values.



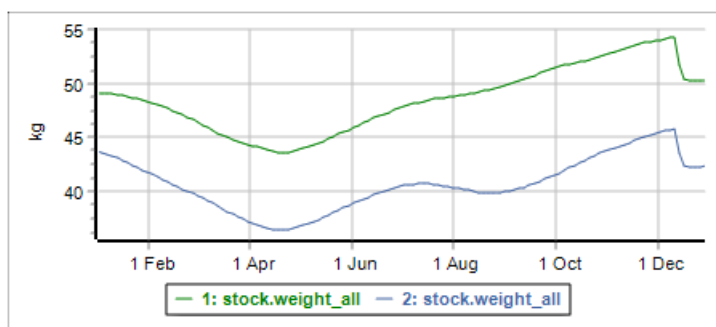
- ➡ Edit one of the factor values. This factor value is 20 animals per hectare.

The 'Initialise Generic' dialog box is shown with the 'Variables' tab selected. A list on the left contains 'S SR'. On the right, the 'Name' field is 'SR', 'Units' is empty, 'State variable' is checked, and 'Initial Value' is '20'. Below the list are 'Add' and 'Delete' buttons. At the bottom are 'OK', 'Cancel', and 'Help' buttons.

After running the Analysis the report will show the effects of increasing the stocking rate.

Animal weights for all treatments.

Long term average animal weights [1 Jan - 31 Dec, 1970-1980]



Treatment	StockingRate
1	SR_10
2	SR_20

11. Configuring Reports

Reports are an advanced feature that can take some effort to master. When a well configured report is used with a simulation it can add significant value to the process. Reports can show the results from an individual simulation as well as an Analysis of almost any dimension.

In AusFarm to generate a report requires the configuration of a report template. The template is a set of instructions that determine the layout, the variables to use and how to aggregate them. Reports can contain charts, tables, headings, text sections and grid layouts of these items. To design a report by creating a template AusFarm includes an editing window as shown below.

Report Designer dialog

Report sections



Chart	Compact view of many points. Line, bar, area.
Table	High precision view with data in column and row format.
Text	Detailed description that can include HTML markup script.
Summary Table	Compact comparison of treatments in table form.
Loop	Groups of treatment results that organises treatment comparison

When designing a report, it can be efficient to copy existing sections and adapt them. Sections can be copied from within a report design or from another one. You will need to make some decisions such as:


- Decide which variables convey the information you need
- Choose the most appropriate transformation of the time course of the variables
- Choose an aggregation interval and statistics
- Format the values: chart type, series colours, decimal places and units

11.1 Report Variables

An AusFarm simulation can contain several reports. You can edit each one independently, adding as many sections such as tables, charts, and grid layouts as you require. See the **Help file** for more details.





In the bottom table of the Report Designer is a list of the variables that will be used in the selected report section. Use the **Select**, **Remove**, and **Clear** buttons to manipulate the list of variables in each report section. You can order the variables using the red up and down arrows  

Calculated expressions

By using the  button you can add a calculated expression as a column in your table or chart.

Variables

Name	Title	Description	Display	Aggreg.	DecPl	Units
income_ha	Total income	Total income/ha	No	Sum	0	\$/ha
expense_ha	Total expenses	Total expense/ha	No	Sum	0	\$/ha
income_ha - expense_ha	Gross Margin	Annual GM	Yes	none	2	\$/ha

In the name column for a calculated expression, enter a mathematical expression. It may include the names of other columns. In this case: *income_ha - expense_ha*. You can choose whether you want to display the source columns.

Note: You cannot do any aggregation on the calculated column.

12. Using Repositories

Repositories are used to store module data and management scripts that are used regularly in simulations.

Items of a similar type can be grouped into folders within a repository. Repository items are associated with a component. Items associated with the Manager component are treated somewhat differently to other items.

Examples of items that may be stored in a repository are:

1. Commonly used sets of management rules
2. An archive of project work for later use
3. A "library" of commonly used soil descriptions.

12.1 Getting module data from a repository

➡ Close the Results window.

➡ Open the Repository clicking the library button  on the main toolbar.

A Repository window like this will open on the bottom of the main window:



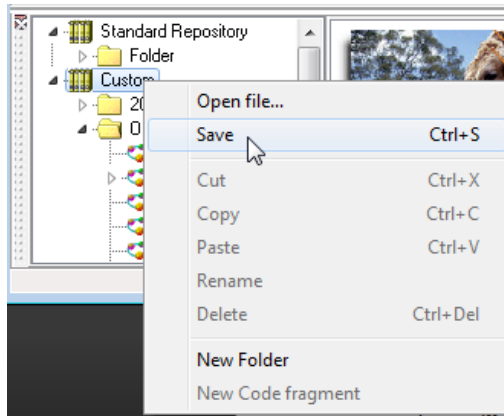
The Repository is divided into two sections. The Custom section is for the user to add to or delete from while the Standard section contains items that cannot be changed.

Items from the Repository can be dragged into simulations using the mouse.

When you need to add items to the Repository, items can be dragged from simulations and dropped into folders.

Before you exit AusFarm you will be asked if you want to save any changes you have made to the Repository.

➡ To save changes or add folders, just right click the mouse on an item in the Custom section.



You can also have other custom Repository files. If you right click on the Custom library, you can open another file. If you want to create another custom library file, choose the **Open file...** option and type in a new name in the Open file dialog. You will then be asked if you want to create a new file.

Items from the Standard library section can be copied into the Custom section by just dragging with the mouse.

12.2 Copying module data to a repository

- Drag the icon for the Weather module onto a folder in the repository. A new repository item will appear.
- Select the Weather repository item. The initial values for the module will appear in the right-hand pane of the repository. These values can be edited in the same way as in the Initialise tab of a simulation window.

12.3 Copying module data from simulation to simulation

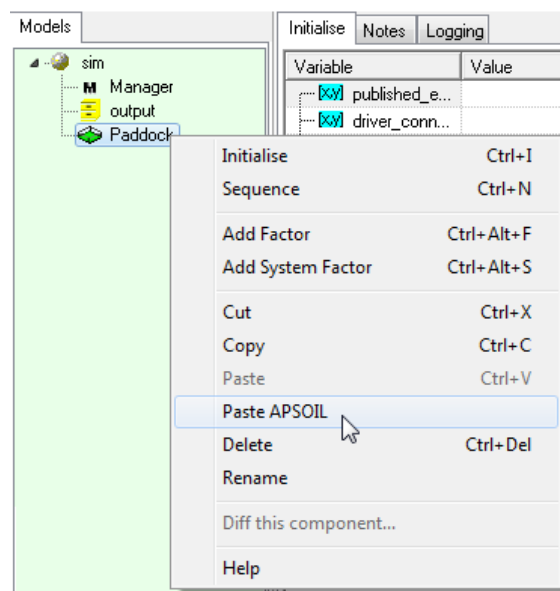
Add a second, identical paddock to the simulation:

- Right-click on **paddock1** in the Models tab.
- Select the Copy option from the pop-up menu that appears.
- Right-click on the simulation icon in the Models tab.
- Select the Paste option from the pop-up menu that appears. A copy of the paddock system will appear.
- Rename the new paddock as **paddock2**.

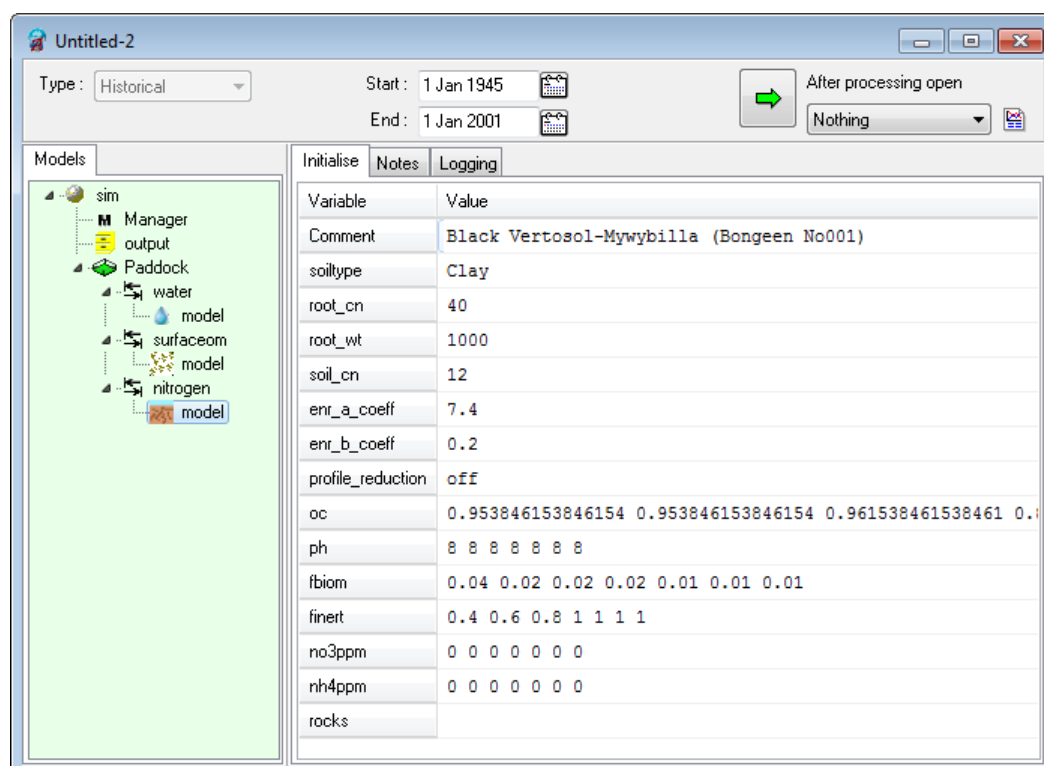
The same process can be followed to copy module data between two different simulation windows. Modules may also be dragged and dropped rather than copied and pasted.

13. Using APSOIL soil data

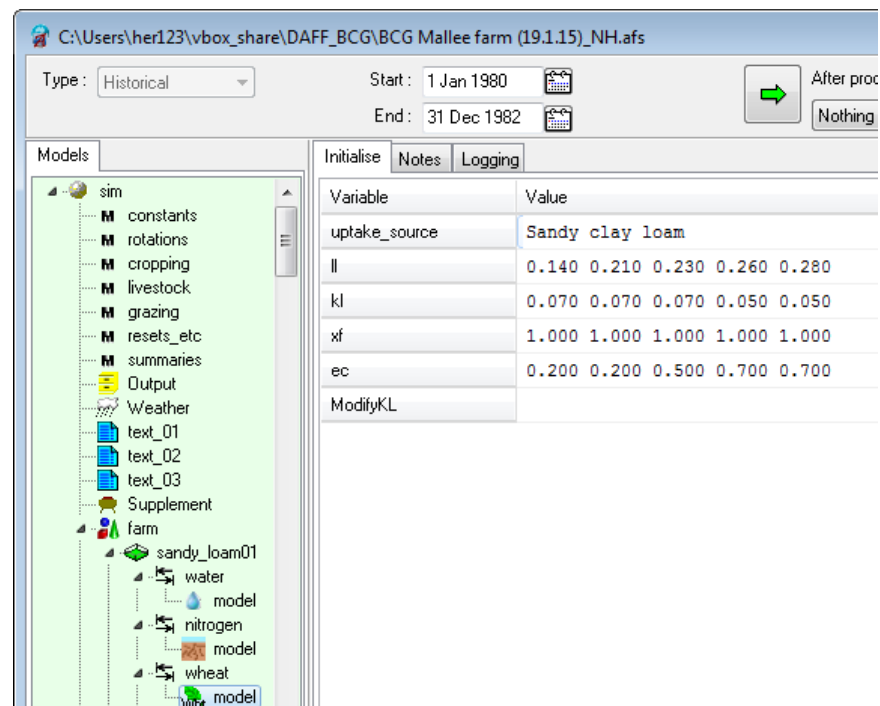
It is possible to copy the soil descriptions from APSOIL directly into AusFarm. This is done firstly by copying the soil chosen in APSOIL to the clipboard. Then with a paddock component in the model tree in AusFarm; right click on the paddock and choose **Paste APSOIL**.



Once the soil is pasted on the paddock, new components will be added to the paddock system. The nitrogen model then needs to be initialised for initial nitrogen values. The soiln initial values are shown below.




The initial values for the cropping modules will need to be set manually as shown below.



The screenshot shows the AusFarm software interface. The title bar indicates the file path: C:\Users\her123\vbbox_share\DAFF_BCG\BCG Mallee farm (19.1.15)_NH.afs. The 'Type' is set to 'Historical'. The 'Start' date is '1 Jan 1980' and the 'End' date is '31 Dec 1982'. There are buttons for 'After proc' and 'Nothing'. The 'Models' pane on the left shows a tree structure with 'sim' at the top, followed by 'constants', 'rotations', 'cropping', 'livestock', 'grazing', 'resets_etc', 'summaries', 'Output', 'Weather', 'text_01', 'text_02', 'text_03', 'Supplement', and 'farm'. Under 'farm', there is 'sandy_loam01', which contains 'water', 'model', 'nitrogen', 'wheat', and 'vine'. The 'wheat' module is selected. The 'Initialise' tab is active, showing a table of variables and their values.

Variable	Value
uptake_source	Sandy clay loam
ll	0.140 0.210 0.230 0.260 0.280
kl	0.070 0.070 0.070 0.050 0.050
xf	1.000 1.000 1.000 1.000 1.000
ec	0.200 0.200 0.500 0.700 0.700
ModifyKL	



As Australia's national science agency and innovation catalyst, CSIRO is solving the greatest challenges through innovative science and technology.

CSIRO. Unlocking a better future for everyone.

Contact us

1300 363 400
+61 3 9545 2176
csiroenquiries@csiro.au
www.csiro.au

For further information

CSIRO Agriculture and Food
Neville Herrmann
+61 2 6246 5290
neville.herrmann@csiro.au
<https://www.csiro.au/en/Research/AF>

For further information

CSIRO
Andrew Moore
+61 2 6246 5298
andrew.moore@csiro.au