Alan Lewis

Analytic Well Field Model Core Documentation

NAME OF BOOK

ISBN–INFO
ISBN–13:

Contents

List of Figures

List of Tables

Introduction

The Analytical Well Field Model (AWFM) is a collection of computer methods for processing (and reducing) large amounts of pumping and water level data, running analytical models, and providing meaningful estimates of aquifer and well-loss properties.

The core of AWFM provides many utilies for importing/exporting and processing data, but is unopinionated in that it makes it easy to add additional functionality when convenient.

With SCADA systems becoming more and more common, it is not unusual to see pumping and and water levels data sets containing hundreds-of-thousands or millions of records. While it is advantageous to begin with such high-resolution data, it is largely redundant.

Chapter 1

Mathematical Basis

1.1 Forward Model

About forward model

1.2 Inverse Model

About inverse model

Chapter 2

Installation

2.1 Linux

2.2 Windows

2.3 Apple

Chapter 3

Timeseries Class

The Timeseries class stores and processes arrays of time-value pairs. Examples of Timeseries objects include pumping rates and water levels.

3.1 Instantiating and Building Timeseries Object

The **Timeseries** class lives inside the *awfm.core* module. It is instantiated with no arguments and initially contains no data. The timeseries is populated with data using the *append* function, which takes two arguments: a time and an associated value.

```
from awfm.core import Timeseries

ts = Timeseries()
ts.append(0, 10) # time = 0, value = 10
ts.append(1, 20)
ts.append(2, 30)
```

Figure 3.1: Example of creating and populating a Timeseries instance

An important feature of the *append* method is that time-value pairs must be inserted in chronological order. If an attempt is made to insert a time-value pair that is out of order, the pair will not be appended to the timeseries, and an error will be logged. Every function that acts on a timeseries assumes that data is stored in chronological order.

3.2 Timeseries Methods

There are several useful methods that can be performed on a **Timeseries** instance. None of the methods described here will modify a timeseries instance. Rather, a modified timeseries will be returned.

```
from awfm.core import Timeseries

ts = Timeseries()
ts.append(0, 10)
ts.append(2, 30)
ts.append(1, 20)

if ts.errors:
    print(errors)
    exit(0)
```

Figure 3.2: How to check that a timeseries has been populated correctly

zero_below_magnitude

zero_below_magnitude is a method that converts all values in a timeseries below some given magnitude to zero. This method is useful for cleaning out noise in pumping rate data. It is common for SCADA systems to continue collecting pumping rate data at some small magnitude after the system has been turned off. For modeling purposes, one may want to simply consider these values to be 0.

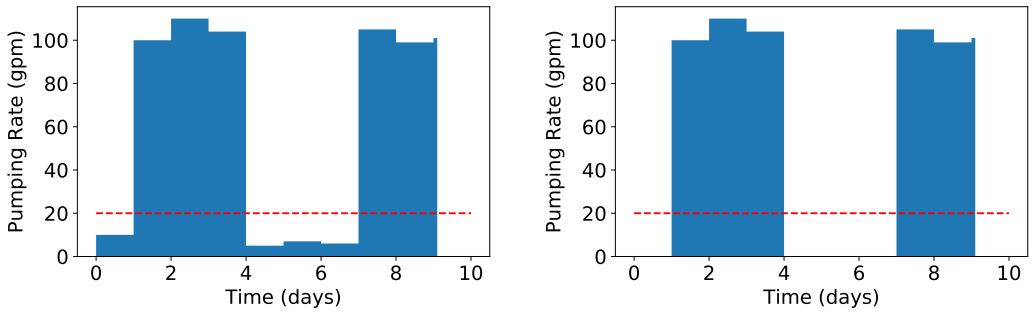

Figure 3.3: *zero_below_magnitude* example plots. Original timeseries (left) is modified (right) using a threshold of 20 gpm.

```
ts = Timeseries()
# ... Timeseries is populated here

ts_modified = ts.zero_below_magnitude(20)
```

Figure 3.4: Example usage of *zero_below_magnitude*

consolidate_adjacent_equal_values

consolidate_adjacent_equal_values is an important method for reducing the size of a data set. Many aquifer drawdown models, such as the Theis solution, only require *changes* in pumping rates as model inputs. If the pumping rate at a well stays constant for a long period of time, the data recorded may be largely redundant. *consolidate_adjacent_equal_values* filters a timeseries such that only changes are kept.

Time	Q
0	120.0
1	120.0
2	118.0
3	0
4	0
5	0
6	0
7	112.0
8	110.0

Time	Q
0	120.0
2	118.0
3	0
7	112.0
8	110.0

Figure 3.5: Original timeseries (left) reduced by *consolidate_adjacent_equal_values* (right)

consolidate_adjacent_equal_values works best when used immediately after *zero_below_magnitude* and/or *round_to_nearest* methods.

round_to_nearest

round_to_nearest rounds each value in a timeseries to some nearest magnitude given as the method argument, using the following function:

$$v_{rounded} = \text{round}(\frac{v}{v_{magnitude}}) * v_{magnitude}$$

For example, given a value (v) of 118.0, rounded to the nearest ($v_{magnitude}$) 10:

$$v_{rounded} = \text{round}\Big(\frac{118.0}{10}\Big) * 10$$

$$v_{rounded} = \text{round}(11.8) * 10$$

$$v_{rounded} = 12 * 10$$

$$v_{rounded} = 120$$

average_by_sign

average_by_sign averages timeseries data by sign, where sign is:

Time	Q
0	120.0
1	120.0
2	118.0
3	0
4	0
5	0
6	0
7	112.0
8	110.0

Time	Q
0	120.0
1	120.0
2	120.0
3	0
4	0
5	0
6	0
7	110.0
8	110.0

Time	Q
0	120.0
3	0
7	110.0

Figure 3.6: Original timeseries (left) rounded to the nearest 10 (middle) using *round_to_nearest* and then being reduced using *consolidate_adjacent_equal_values*

(right)

```
ts = Timeseries()
# ... Timeseries is populated here

ts_modified = ts.round_to_nearest(10).consolidate_adjacent_equal_values()
```

Figure 3.7: Example method calls used to calculate values in Figure 3.6

$$sign = \begin{cases} 1 & \text{if } v > 0 \\ 0 & \text{if } v = 0 \\ -1 & \text{if } v < 0 \end{cases} \tag{3.1}$$

In terms of well production, sign$= 1$ would mean groundwater extraction, sign$= -1$ would mean groundwater injection, and sign$= 0$ would mean that there is no activity.

average_by_sign might be the simplest way to quickly reduce a data set by (potentially) several orders of magnitude.