petbox-dca

Release 1.1.0

Aug 09, 2023

Contents

CHAPTER 1

Petroleum Engineering Toolbox

Empirical analysis of production data requires implementation of several decline curve models spread over years and multiple SPE publications. Additionally, comprehensive analysis requires graphical analysis among multiple diagnostics plots and their respective plotting functions. While each model's `q(t)` (rate) function may be simple, the `N(t)` (cumulative volume) may not be. For example, the hyperbolic model has three different forms (hyperbolic, harmonic, exponential), and this is complicated by potentially multiple segments, each of which must be continuous in the rate derivatives. Or, as in the case of the Power-Law Exponential model, the `N(t)` function must be numerically evaluated.

This library defines a single interface to each of the implemented decline curve models. Each model has validation checks for parameter values and provides simple-to-use methods for evaluating arrays of `time` to obtain the desired function output.

Additionally, we also define an interface to attach a GOR/CGR yield function to any primary phase model. We can then obtain the outputs for the secondary phase as easily as the primary phase.

Analytic functions are implemented wherever possible. When not possible, numerical evaluations are performed using `scipy.integrate.fixed_quad`. Given that most of the functions of interest that must be numerically evaluated are monotonic, this generally works well.

Primary Phase	Transient Hyperbolic, Modified Hyperbolic, Power-Law Exponential, Stretched Exponential, Duong
Secondary Phase	Power-Law Yield
Water Phase	Power-Law Yield

The following functions are exposed for use

Base Functions	rate(t), cum(t), D(t), beta(t), b(t),
Interval Volumes	interval_vol(t), monthly_vol(t), monthly_vol_equiv(t),
Transient Hyperbolic	transient_rate(t), transient_cum(t), transient_D(t), transient_beta(t), transient_b(t)
Primary Phase	add_secondary(model), add_water(model)
Secondary Phase	gor(t), cgr(t)
Water Phase	wor(t), wgr(t)
Utility	bourdet(y, x, ...), get_time(...), get_time_monthly_vol(...)

1.1 Getting Started

Install the library with pip:

```
pip install petbox-dca
```

A default time array of evenly-logspaced values over 5 log cycles is provided as a convenience.

```
>>> from petbox import dca
>>> t = dca.get_time()
>>> mh = dca.MH(qi=1000.0, Di=0.8, bi=1.8, Dterm=0.08)
>>> mh.rate(t)
array([986.738, 982.789, 977.692, ..., 0.000])
```

We can also attach secondary phase and water phase models, and evaluate the rate just as easily.

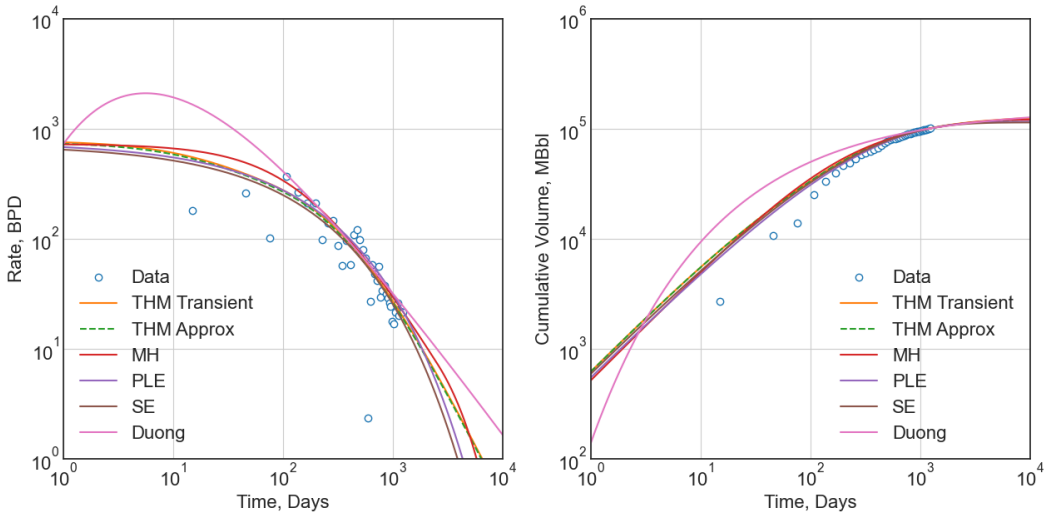
```
>>> mh.add_secondary(dca.PLYield(c=1200.0, m0=0.0, m=0.6, t0=180.0, min=None, max=20_
↪000.0))
>>> mh.secondary.rate(t)
array([1184.086, 1179.346, 1173.231, ..., 0.000])

>>> mh.add_water(dca.PLYield(c=2.0, m0=0.0, m=0.1, t0=90.0, min=None, max=10.0))
>>> mh.water.rate(t)
array([1.950, 1.935, 1.917, ..., 0.000])
```

Once instantiated, the same functions and process for attaching a secondary phase work for any model.

```
>>> thm = dca.THM(qi=1000.0, Di=0.8, bi=2.0, bf=0.8, telf=30.0, bterm=0.03, tterm=10.
↪0)
>>> thm.rate(t)
array([968.681, 959.741, 948.451, ..., 0.000])

>>> thm.add_secondary(dca.PLYield(c=1200.0, m0=0.0, m=0.6, t0=180.0, min=None, max=20_
↪000.0))
>>> thm.secondary.rate(t)
array([1162.417, 1151.690, 1138.141, ..., 0.000])

>>> ple = dca.PLE(qi=1000.0, Di=0.1, Dinf=0.00001, n=0.5)
>>> ple.rate(t)
array([904.828, 892.092, 877.768, ..., 0.000])

>>> ple.add_secondary(dca.PLYield(c=1200.0, m0=0.0, m=0.6, t0=180.0, min=None, max=20_
↪000.0))
>>> ple.secondary.rate(t)
array([1085.794, 1070.510, 1053.322, ..., 0.000])
```

Applying the above, we can easily evaluate each model against a data set.

```
>>> import matplotlib.pyplot as plt
>>> fig = plt.figure()
>>> ax1 = fig.add_subplot(121)
>>> ax2 = fig.add_subplot(122)

>>> ax1.plot(t_data, rate_data, 'o')
>>> ax2.plot(t_data, cum_data, 'o')

>>> ax1.plot(t, thm.rate(t))
>>> ax2.plot(t, thm.cum(t) * cum_data[-1] / thm.cum(t_data[-1]))  # normalization

>>> ax1.plot(t, ple.rate(t))
>>> ax2.plot(t, ple.cum(t) * cum_data[-1] / ple.cum(t_data[-1]))  # normalization

>>> ...

>>> plt.show()
```



See the API documentation for a complete listing, detailed use examples, and model comparison.

1.2 Development

`petbox-dca` is maintained by David S. Fulford (@dsfulf). Please post an issue or pull request in this repo for any problems or suggestions!

1.3 Contents

1.3.1 API Reference

Summary

Primary Phase Models

THM(qi, Di, bi, bf, telf, bterm, tterm, ...)	Transient Hyperbolic Model
MH(qi, Di, bi, Dterm, validate_params)	Modified Hyperbolic Model
PLE(qi, Di, Dinf, n, validate_params)	Power-Law Exponential Model
SE(qi, tau, n, validate_params)	Stretched Exponential
Duong(qi, a, m, validate_params)	Duong Model

Associated Phase Models

Secondary Phase Models

PLYield(c, m0, m, t0, min, max)	Power-Law Associated Phase Model.

Water Phase Models

PLYield(c, m0, m, t0, min, max)	Power-Law Associated Phase Model.

Model Functions

All Models

rate(t, numpy.ndarray[Any, ...)	Defines the model rate function:
cum(t, numpy.ndarray[Any, ...)	Defines the model cumulative volume function:
interval_vol(t, numpy.ndarray[Any, ...)	Defines the model interval volume function:
monthly_vol(t, numpy.ndarray[Any, ...)	Defines the model fixed monthly interval volume function.
monthly_vol_equiv(t, numpy.ndarray[Any, ...)	Defines the model equivalent monthly interval volume function:
D(t, numpy.ndarray[Any, ...)	Defines the model D-parameter function:
beta(t, numpy.ndarray[Any, ...)	Defines the model beta-parameter function.
b(t, numpy.ndarray[Any, ...)	Defines the model b-parameter function:
get_param_desc(name)	Get a single parameter description.
get_param_descs()	Get the parameter descriptions.
from_params(params)	Construct a model from a sequence of parameters.

Primary Phase Models

add_secondary(secondary)	Attach a secondary phase model to this primary phase model.
add_water(water)	Attach a water phase model to this primary phase model.

Associated Phase Models

Secondary Phase Models

gor(t, numpy.ndarray[Any, ...)	Defines the model GOR function.
cgr(t, numpy.ndarray[Any, ...)	Defines the model CGR function.

Water Phase Models

wor(t, numpy.ndarray[Any, ...)	Defines the model WOR function.
wgr(t, numpy.ndarray[Any, ...)	Defines the model WGR function.

Transient Hyperbolic Specific

transient_rate(t, numpy.ndarray[Any, ...)	Compute the rate function using full definition.
transient_cum(t, numpy.ndarray[Any, ...)	Compute the cumulative volume function using full definition.
transient_D(t, numpy.ndarray[Any, ...)	Compute the D-parameter function using full definition.
transient_beta(t, numpy.ndarray[Any, ...)	Compute the beta-parameter function using full definition.
transient_b(t, numpy.ndarray[Any, ...)	Compute the b-parameter function using full definition.

Utility Functions

bourdet(y, numpy.dtype[numpy.float64]], x, ...)	Bourdet Derivative Smoothing
get_time(start, end, n)	Get a time array to evaluate with.
get_time_monthly_vol(start, end)	Get a time array to evaluate with.

Utility Constants

DAYS_PER_MONTH	365.25 / 12 = 30.4375
DAYS_PER_YEAR	365.25

Detailed Reference

Base Classes

These classes define the basic functions that are exposed by all decline curve models.

class petbox.dca.**DeclineCurve**(**args*)

Base class for decline curve models. Each model must implement the defined abstract methods.

rate(*t: Union[float, numpy.ndarray[Any, numpy.dtype[numpy.float64]]]*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]

Defines the model rate function:

$$q(t) = f(t)$$

where $f(t)$ is defined by each model.

Parameters **t** (*Union[float, numpy.NDFloat]*) – An array of times at which to evaluate the function.

Returns rate

Return type numpy.NDFloat

cum(*t: Union[float, numpy.ndarray[Any, numpy.dtype[numpy.float64]]], **kwargs*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]

Defines the model cumulative volume function:

$$N(t) = \int_0^t q\, dt$$

Parameters

- **t** (*Union[float, numpy.NDFloat]*) – An array of times at which to evaluate the function.
- ****kwargs** – Additional arguments passed to `scipy.integrate.fixed_quad()` if needed.

Returns cumulative volume

Return type numpy.NDFloat

interval_vol(*t: Union[float, numpy.ndarray[Any, numpy.dtype[numpy.float64]]], t0: Union[float, numpy.ndarray[Any, numpy.dtype[numpy.float64]], None] = None, **kwargs*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]

Defines the model interval volume function:

$$N(t) = \int_{t_{i-1}}^{t_i} q\, dt$$

for each element of `t`.

Parameters

- **t** (*Union[float, numpy.NDFloat]*) – An array of interval end times at which to evaluate the function.
- **t0** (*Optional[Union[float, numpy.NDFloat]]*) – A start time of the first interval. If not given, the first element of `t` is used.
- ****kwargs** – Additional arguments passed to `scipy.integrate.fixed_quad()` if needed.

Returns interval volume

Return type numpy.NDFloat

monthly_vol(*t: Union[float, numpy.ndarray[Any, numpy.dtype[numpy.float64]]], **kwargs*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]

Defines the model fixed monthly interval volume function. If t < 1 month, the interval begin at zero:

$$N(t) = \int_{t-1\,month}^{t} q\, dt$$

Parameters

- **t** (*Union[float, numpy.NDFloat]*) – An array of interval end times at which to evaluate the function.
- ****kwargs** – Additional arguments passed to `scipy.integrate.fixed_quad()` if needed.

Returns monthly equivalent volume

Return type numpy.NDFloat

monthly_vol_equiv(*t: Union[float, numpy.ndarray[Any, numpy.dtype[numpy.float64]]], t0: Union[float, numpy.ndarray[Any, numpy.dtype[numpy.float64]], None] = None, **kwargs*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]

Defines the model equivalent monthly interval volume function:

$$N(t) = \frac{\frac{365.25}{12}}{t - (t - 1\,month)} \int_{t-1\,month}^{t} q\,dt$$

Parameters

- **t** (*Union[float, numpy.NDFloat]*) – An array of interval end times at which to evaluate the function.
- **t0** (*Optional[Union[float, numpy.NDFloat]]*) – A start time of the first interval. If not given, assumed to be zero.
- ****kwargs** – Additional arguments passed to `scipy.integrate.fixed_quad()` if needed.

Returns monthly equivalent volume

Return type numpy.NDFloat

D(*t: Union[float, numpy.ndarray[Any, numpy.dtype[numpy.float64]]]*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]

Defines the model D-parameter function:

$$D(t) \equiv \frac{d}{dt} \ln q \equiv \frac{1}{q} \frac{dq}{dt}$$

Parameters **t** (*Union[float, numpy.NDFloat]*) – An array of times at which to evaluate the function.

Returns D-parameter

Return type numpy.NDFloat

beta(*t: Union[float, numpy.ndarray[Any, numpy.dtype[numpy.float64]]]*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]

Defines the model beta-parameter function.

$$\beta(t) \equiv \frac{d \ln q}{d \ln t} \equiv \frac{t}{q} \frac{dq}{dt} \equiv t\,D(t)$$

Parameters **t** (*Union[float, numpy.NDFloat]*) – An array of times at which to evaluate the function.

Returns beta-parameter

Return type numpy.NDFloat

b(*t: Union[float, numpy.ndarray[Any, numpy.dtype[numpy.float64]]]*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]

Defines the model b-parameter function:

$$b(t) \equiv \frac{d}{dt} \frac{1}{D}$$

Parameters **t** (*Union[float, numpy.NDFloat]*) – An array of times at which to evaluate the function.

Returns **b-parameter**

Return type numpy.NDFloat

classmethod get_param_desc(*name: str*) → petbox.dca.base.ParamDesc
Get a single parameter description.

Parameters **name** (*str*) – The parameter name.

Returns **parameter description** – A parameter description.

Return type ParamDesc

classmethod get_param_descs() → List[petbox.dca.base.ParamDesc]
Get the parameter descriptions.

Returns **parameter description** – A list of parameter descriptions.

Return type List[ParamDesc]

classmethod from_params(*params: Sequence[float]*) → _Self
Construct a model from a sequence of parameters.

Returns **decline curve** – The constructed decline curve model class.

Return type *DeclineCurve*

class petbox.dca.**PrimaryPhase**(**args*)
Extends *DeclineCurve* for a primary phase forecast. Adds the capability to link a secondary (associated) phase model.

add_secondary(*secondary: petbox.dca.base.SecondaryPhase*) → None
Attach a secondary phase model to this primary phase model.

Parameters **secondary** (SecondaryPhase) – A model that inherits the *SecondaryPhase* class.

add_water(*water: petbox.dca.base.WaterPhase*) → None
Attach a water phase model to this primary phase model.

Parameters **water** (WaterPhase) – A model that inherits the *WaterPhase* class.

class petbox.dca.**SecondaryPhase**(**args*)
Extends *DeclineCurve* for a secondary (associated) phase forecast. Adds the capability to link a primary phase model. Defines the *gor()* and *cgr()* functions. Each model must implement the defined abstract method.

gor(*t: Union[float, numpy.ndarray[Any, numpy.dtype[numpy.float64]]]*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]
Defines the model GOR function. Implementation is idential to CGR function.

Parameters **t** (*Union[float, numpy.NDFloat]*) – An array of times at which to evaluate the function.

Returns **GOR** – The gas-oil ratio function in units of Mscf / Bbl.

Return type numpy.NDFloat

cgr(*t: Union[float, numpy.ndarray[Any, numpy.dtype[numpy.float64]]]*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]
Defines the model CGR function. Implementation is identical to GOR function.

Parameters **t** (*Union[float, numpy.NDFloat]*) – An array of times at which to evaluate the function.

Returns **CGR** – The condensate-gas ratio in units of Bbl / Mscf.

Return type numpy.NDFloat

class petbox.dca.**WaterPhase**(**args*)

Extends *DeclineCurve* for a water (associated) phase forecast. Adds the capability to link a primary phase model. Defines the *wor()* function. Each model must implement the defined abstract method.

wor(*t: Union[float, numpy.ndarray[Any, numpy.dtype[numpy.float64]]]*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]

Defines the model WOR function.

Parameters **t** (*Union[float, numpy.NDFloat]*) – An array of times at which to evaluate the function.

Returns **WOR** – The water-oil ratio function in units of `Bbl / Bbl`.

Return type numpy.NDFloat

wgr(*t: Union[float, numpy.ndarray[Any, numpy.dtype[numpy.float64]]]*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]

Defines the model WGR function.

Parameters **t** (*Union[float, numpy.NDFloat]*) – An array of times at which to evaluate the function.

Returns **WOR** – The water-gas ratio function in units of `Bbl / Mscf`.

Return type numpy.NDFloat

Primary Phase Models

Implementations of primary phase decline curve models

class petbox.dca.**THM**(*qi: float*, *Di: float*, *bi: float*, *bf: float*, *telf: float*, *bterm: float = 0.0*, *tterm: float = 0.0*, *validate_params: Iterable[bool] = <factory>*)

Transient Hyperbolic Model

Fulford, D. S., and Blasingame, T. A. 2013. Evaluation of Time-Rate Performance of Shale Wells using the Transient Hyperbolic Relation. Presented at SPE Unconventional Resources Conference – Canada in Calgary, Alberta, Canda, 5–7 November. SPE-167242-MS. https://doi.org/10.2118/167242-MS.

Analytic Approximation

Fulford, D.S. 2018. A Model-Based Diagnostic Workflow for Time-Rate Performance of Unconventional Wells. Presented at Unconventional Resources Conference in Houston, Texas, USA, 23–25 July. URTeC-2903036. https://doi.org/10.15530/urtec-2018-2903036.

Parameters

- **qi** (*float*) – The initial production rate in units of `volume / day`.
- **Di** (*float*) – The initial decline rate in secant effective decline aka annual effective percent decline, i.e.

$$D_i = 1 - \frac{q(t = 1\,year)}{qi}$$

$$D_i = 1 - (1 + 365.25\,D_{nom}\,b)^{\frac{-1}{b}}$$

where `Dnom` is defined as $\frac{d}{dt} \ln q$ and has units of `1 / day`.

- **bi** (*float*) – The initial hyperbolic parameter, defined as $\frac{d}{dt}\frac{1}{D}$. This parameter is dimensionless. Advised to always be set to `2.0` to represent transient linear flow. See literature for more details.

- **bf** (*float*) – The final hyperbolic parameter after transition. Represents the boundary-dominated or boundary-influenced flow regime.
- **telf** (*float*) – The time to end of linear flow in units of `day`, or more specifically the time at which `b(t) < bi`. Visual end of half slope occurs `~2.5x` after `telf`.
- **bterm** (*Optional[float] = None*) – The terminal value of the hyperbolic parameter. Has two interpretations:

  If `tterm > 0` then the terminal regime is a hyperbolic regime with `b = bterm` and the parameter is given as the hyperbolic parameter.

  If `tterm = 0` then the terminal regime is an exponential regime with `Dterm = bterm` and the parameter is given as secant effective decline.
- **tterm** (*Optional[float] = None*) – The time to start of the terminal regime. Setting `tterm = 0.0` creates an exponential terminal regime, while setting `tterm > 0.0` creates a hyperbolic terminal regime.

rate(*t: Union[float, numpy.ndarray[Any, numpy.dtype[numpy.float64]]]*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]

Defines the model rate function:

$$q(t) = f(t)$$

where $f(t)$ is defined by each model.

Parameters **t** (*Union[float, numpy.NDFloat]*) – An array of times at which to evaluate the function.

Returns **rate**

Return type numpy.NDFloat

cum(*t: Union[float, numpy.ndarray[Any, numpy.dtype[numpy.float64]]], **kwargs*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]

Defines the model cumulative volume function:

$$N(t) = \int_0^t q\, dt$$

Parameters

- **t** (*Union[float, numpy.NDFloat]*) – An array of times at which to evaluate the function.
- ****kwargs** – Additional arguments passed to `scipy.integrate.fixed_quad()` if needed.

Returns **cumulative volume**

Return type numpy.NDFloat

D(*t: Union[float, numpy.ndarray[Any, numpy.dtype[numpy.float64]]]*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]

Defines the model D-parameter function:

$$D(t) \equiv \frac{d}{dt} \ln q \equiv \frac{1}{q} \frac{dq}{dt}$$

Parameters **t** (*Union[float, numpy.NDFloat]*) – An array of times at which to evaluate the function.

Returns **D-parameter**

Return type numpy.NDFloat

beta(*t: Union[float, numpy.ndarray[Any, numpy.dtype[numpy.float64]]]*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]

Defines the model beta-parameter function.

$$\beta(t) \equiv \frac{d \ln q}{d \ln t} \equiv \frac{t}{q}\frac{dq}{dt} \equiv t\, D(t)$$

Parameters **t** (*Union[float, numpy.NDFloat]*) – An array of times at which to evaluate the function.

Returns **beta-parameter**

Return type numpy.NDFloat

b(*t: Union[float, numpy.ndarray[Any, numpy.dtype[numpy.float64]]]*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]

Defines the model b-parameter function:

$$b(t) \equiv \frac{d}{dt}\frac{1}{D}$$

Parameters **t** (*Union[float, numpy.NDFloat]*) – An array of times at which to evaluate the function.

Returns **b-parameter**

Return type numpy.NDFloat

transient_rate(*t: Union[float, numpy.ndarray[Any, numpy.dtype[numpy.float64]]], **kwargs*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]

Compute the rate function using full definition. Uses `scipy.integrate.fixed_quad()` to integrate *transient_D()*.

$$q(t) = e^{-\int_0^t D(t)\, dt}$$

Parameters

- **t** (*Union[float, numpy.NDFloat]*) – An array of time values to evaluate.
- ****kwargs** – Additional keyword arguments passed to `scipy.integrate.fixed_quad()`.

Returns

Return type numpy.NDFloat

transient_cum(*t: Union[float, numpy.ndarray[Any, numpy.dtype[numpy.float64]]], **kwargs*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]

Compute the cumulative volume function using full definition. Uses `scipy.integrate.fixed_quad()` to integrate `transient_q()`.

$$N(t) = \int_0^t q(t)\, dt$$

Parameters

- **t** (*Union[float, numpy.NDFloat]*) – An array of time values to evaluate.
- ****kwargs** – Additional keyword arguments passed to `scipy.integrate.fixed_quad()`.

Returns

Return type numpy.NDFloat

transient_D(*t: Union[float, numpy.ndarray[Any, numpy.dtype[numpy.float64]]]*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]

Compute the D-parameter function using full definition.

$$D(t) = \frac{1}{\frac{1}{Di} + b_i t + \frac{bi-bf}{c}(\mathrm{Ei}[-e^{-c\,(t-t_{elf})+e^{(\gamma)}}] - \mathrm{Ei}[-e^{c\,t_{elf}+e^{(\gamma)}}])}$$

Parameters **t** (*Union[float, numpy.NDFloat]*) – An array of time values to evaluate.

Returns

Return type numpy.NDFloat

transient_beta(*t: Union[float, numpy.ndarray[Any, numpy.dtype[numpy.float64]]]*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]

Compute the beta-parameter function using full definition.

$$\beta(t) = \frac{t}{\frac{1}{Di} + b_i t + \frac{bi-bf}{c}(\mathrm{Ei}[-e^{-c\,(t-t_{elf})+e^{(\gamma)}}] - \mathrm{Ei}[-e^{c\,t_{elf}+e^{(\gamma)}}])}$$

Parameters **t** (*Union[float, numpy.NDFloat]*) – An array of time values to evaluate.

Returns

Return type numpy.NDFloat

transient_b(*t: Union[float, numpy.ndarray[Any, numpy.dtype[numpy.float64]]]*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]

Compute the b-parameter function using full definition.

$$b(t) = b_i - (b_i - b_f)e^{-\exp[-c*(t-t_{elf})+e^{\gamma}]}$$

where:

$$c = \frac{e^{\gamma}}{1.5\,t_{elf}}$$

$\gamma = 0.57721566...$ (Euler-Mascheroni constant)

Parameters **t** (*Union[float, numpy.NDFloat]*) – An array of time values to evaluate.

Returns

Return type numpy.NDFloat

class petbox.dca.**MH**(*qi: float*, *Di: float*, *bi: float*, *Dterm: float = 0.0*, *validate_params: Iterable[bool] = <factory>*)

Modified Hyperbolic Model

Robertson, S. 1988. Generalized Hyperbolic Equation. Available from SPE, Richardson, Texas, USA. SPE-18731-MS.

Parameters

- **qi** (*float*) – The initial production rate in units of `volume / day`.
- **Di** (*float*) – The initial decline rate in secant effective decline aka annual effective percent decline, i.e.

$$D_i = 1 - \frac{q(t = 1\,year)}{qi}$$

$$D_i = 1 - (1 + 365.25\,D_{nom}\,b)^{\frac{-1}{b}}$$

where `Dnom` is defined as $\frac{d}{dt}\ln q$ and has units of `1 / day`.

- **bi** (*float*) – The (initial) hyperbolic parameter, defined as $\frac{d}{dt}\frac{1}{D}$. This parameter is dimensionless.
- **Dterm** (*float*) – The terminal secant effective decline rate aka annual effective percent decline.

rate(*t: Union[float, numpy.ndarray[Any, numpy.dtype[numpy.float64]]]*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]

Defines the model rate function:

$$q(t) = f(t)$$

where $f(t)$ is defined by each model.

Parameters **t** (*Union[float, numpy.NDFloat]*) – An array of times at which to evaluate the function.

Returns **rate**

Return type numpy.NDFloat

cum(*t: Union[float, numpy.ndarray[Any, numpy.dtype[numpy.float64]]], **kwargs*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]

Defines the model cumulative volume function:

$$N(t) = \int_0^t q\, dt$$

Parameters

- **t** (*Union[float, numpy.NDFloat]*) – An array of times at which to evaluate the function.
- ****kwargs** – Additional arguments passed to `scipy.integrate.fixed_quad()` if needed.

Returns **cumulative volume**

Return type numpy.NDFloat

D(*t: Union[float, numpy.ndarray[Any, numpy.dtype[numpy.float64]]]*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]

Defines the model D-parameter function:

$$D(t) \equiv \frac{d}{dt}\ln q \equiv \frac{1}{q}\frac{dq}{dt}$$

Parameters **t** (*Union[float, numpy.NDFloat]*) – An array of times at which to evaluate the function.

Returns **D-parameter**

Return type numpy.NDFloat

beta(*t: Union[float, numpy.ndarray[Any, numpy.dtype[numpy.float64]]]*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]

Defines the model beta-parameter function.

$$\beta(t) \equiv \frac{d\ln q}{d\ln t} \equiv \frac{t}{q}\frac{dq}{dt} \equiv t\, D(t)$$

Parameters **t** (*Union[float, numpy.NDFloat]*) – An array of times at which to evaluate the function.

Returns **beta-parameter**

Return type numpy.NDFloat

b(*t: Union[float, numpy.ndarray[Any, numpy.dtype[numpy.float64]]]*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]

Defines the model b-parameter function:

$$b(t) \equiv \frac{d}{dt}\frac{1}{D}$$

Parameters **t** (*Union[float, numpy.NDFloat]*) – An array of times at which to evaluate the function.

Returns **b-parameter**

Return type numpy.NDFloat

classmethod get_param_desc(*name: str*) → petbox.dca.base.ParamDesc

Get a single parameter description.

Parameters **name** (*str*) – The parameter name.

Returns **parameter description** – A parameter description.

Return type ParamDesc

classmethod get_param_descs() → List[petbox.dca.base.ParamDesc]

Get the parameter descriptions.

Returns **parameter description** – A list of parameter descriptions.

Return type List[ParamDesc]

classmethod from_params(*params: Sequence[float]*) → _Self

Construct a model from a sequence of parameters.

Returns **decline curve** – The constructed decline curve model class.

Return type *DeclineCurve*

class petbox.dca.**PLE**(*qi: float*, *Di: float*, *Dinf: float*, *n: float*, *validate_params: Iterable[bool] = <factory>*)

Power-Law Exponential Model

Ilk, D., Perego, A. D., Rushing, J. A., and Blasingame, T. A. 2008. Exponential vs. Hyperbolic Decline in Tight Gas Sands – Understanding the Origin and Implications for Reserve Estimates Using Arps Decline Curves. Presented at SPE Annual Technical Conference and Exhibition in Denver, Colorado, USA, 21–24 September. SPE-116731-MS. https://doi.org/10.2118/116731-MS.

Ilk, D., Rushing, J. A., and Blasingame, T. A. 2009. Decline Curve Analysis for HP/HT Gas Wells: Theory and Applications. Presented at SPE Annual Technical Conference and Exhibition in New Orleands, Louisiana, USA, 4–7 October. SPE-125031-MS. https://doi.org/10.2118/125031-MS.

Parameters

- **qi** (*float*) – The initial production rate in units of volume / day.
- **Di** (*float*) – The initial decline rate in nominal decline rate defined as d[ln q] / dt and has units of 1 / day.
- **Dterm** (*float*) – The terminal decline rate in nominal decline rate, has units of 1 / day.
- **n** (*float*) – The n exponent.

rate(*t: Union[float, numpy.ndarray[Any, numpy.dtype[numpy.float64]]]*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]
Defines the model rate function:

$$q(t) = f(t)$$

where $f(t)$ is defined by each model.

Parameters **t** (*Union[float, numpy.NDFloat]*) – An array of times at which to evaluate the function.

Returns **rate**

Return type numpy.NDFloat

cum(*t: Union[float, numpy.ndarray[Any, numpy.dtype[numpy.float64]]], **kwargs*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]
Defines the model cumulative volume function:

$$N(t) = \int_0^t q\, dt$$

Parameters

- **t** (*Union[float, numpy.NDFloat]*) – An array of times at which to evaluate the function.
- ****kwargs** – Additional arguments passed to `scipy.integrate.fixed_quad()` if needed.

Returns **cumulative volume**

Return type numpy.NDFloat

D(*t: Union[float, numpy.ndarray[Any, numpy.dtype[numpy.float64]]]*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]
Defines the model D-parameter function:

$$D(t) \equiv \frac{d}{dt}\ln q \equiv \frac{1}{q}\frac{dq}{dt}$$

Parameters **t** (*Union[float, numpy.NDFloat]*) – An array of times at which to evaluate the function.

Returns **D-parameter**

Return type numpy.NDFloat

beta(*t: Union[float, numpy.ndarray[Any, numpy.dtype[numpy.float64]]]*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]
Defines the model beta-parameter function.

$$\beta(t) \equiv \frac{d\ln q}{d\ln t} \equiv \frac{t}{q}\frac{dq}{dt} \equiv t\, D(t)$$

Parameters **t** (*Union[float, numpy.NDFloat]*) – An array of times at which to evaluate the function.

Returns **beta-parameter**

Return type numpy.NDFloat

b(*t: Union[float, numpy.ndarray[Any, numpy.dtype[numpy.float64]]]*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]
Defines the model b-parameter function:

$$b(t) \equiv \frac{d}{dt}\frac{1}{D}$$

Parameters **t** (*Union[float, numpy.NDFloat]*) – An array of times at which to evaluate the function.

Returns **b-parameter**

Return type numpy.NDFloat

classmethod get_param_desc(*name: str*) → petbox.dca.base.ParamDesc
Get a single parameter description.

Parameters **name** (*str*) – The parameter name.

Returns **parameter description** – A parameter description.

Return type ParamDesc

classmethod get_param_descs() → List[petbox.dca.base.ParamDesc]
Get the parameter descriptions.

Returns **parameter description** – A list of parameter descriptions.

Return type List[ParamDesc]

classmethod from_params(*params: Sequence[float]*) → _Self
Construct a model from a sequence of parameters.

Returns **decline curve** – The constructed decline curve model class.

Return type *DeclineCurve*

class petbox.dca.**SE**(*qi: float*, *tau: float*, *n: float*, *validate_params: Iterable[bool]* = <*factory*>)
Stretched Exponential

Valkó, P. P. Assigning Value to Stimulation in the Barnett Shale: A Simultaneous Analysis of 7000 Plus Production Histories and Well Completion Records. 2009. Presented at SPE Hydraulic Fracturing Technology Conference in College Station, Texas, USA, 19–21 January. SPE-119369-MS. https://doi.org/10.2118/119369-MS.

Parameters

- **qi** (*float*) – The initial production rate in units of volume / day.
- **tau** (*float*) – The tau parameter in units of day ** n. Equivalent to:

$$\tau = D^n$$

- **n** (*float*) – The n exponent.

rate(*t: Union[float, numpy.ndarray[Any, numpy.dtype[numpy.float64]]]*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]
Defines the model rate function:

$$q(t) = f(t)$$

where $f(t)$ is defined by each model.

Parameters **t** (*Union[float, numpy.NDFloat]*) – An array of times at which to evaluate the function.

Returns rate

Return type numpy.NDFloat

cum(*t: Union[float, numpy.ndarray[Any, numpy.dtype[numpy.float64]]], **kwargs*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]

Defines the model cumulative volume function:

$$N(t) = \int_0^t q\, dt$$

Parameters

- **t** (*Union[float, numpy.NDFloat]*) – An array of times at which to evaluate the function.
- ****kwargs** – Additional arguments passed to `scipy.integrate.fixed_quad()` if needed.

Returns cumulative volume

Return type numpy.NDFloat

D(*t: Union[float, numpy.ndarray[Any, numpy.dtype[numpy.float64]]]*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]

Defines the model D-parameter function:

$$D(t) \equiv \frac{d}{dt} \ln q \equiv \frac{1}{q} \frac{dq}{dt}$$

Parameters **t** (*Union[float, numpy.NDFloat]*) – An array of times at which to evaluate the function.

Returns D-parameter

Return type numpy.NDFloat

beta(*t: Union[float, numpy.ndarray[Any, numpy.dtype[numpy.float64]]]*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]

Defines the model beta-parameter function.

$$\beta(t) \equiv \frac{d \ln q}{d \ln t} \equiv \frac{t}{q} \frac{dq}{dt} \equiv t\, D(t)$$

Parameters **t** (*Union[float, numpy.NDFloat]*) – An array of times at which to evaluate the function.

Returns beta-parameter

Return type numpy.NDFloat

b(*t: Union[float, numpy.ndarray[Any, numpy.dtype[numpy.float64]]]*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]

Defines the model b-parameter function:

$$b(t) \equiv \frac{d}{dt} \frac{1}{D}$$

Parameters **t** (*Union[float, numpy.NDFloat]*) – An array of times at which to evaluate the function.

Returns b-parameter

Return type numpy.NDFloat

classmethod get_param_desc(*name: str*) → petbox.dca.base.ParamDesc
Get a single parameter description.

Parameters **name** (*str*) – The parameter name.

Returns **parameter description** – A parameter description.

Return type ParamDesc

classmethod get_param_descs() → List[petbox.dca.base.ParamDesc]
Get the parameter descriptions.

Returns **parameter description** – A list of parameter descriptions.

Return type List[ParamDesc]

classmethod from_params(*params: Sequence[float]*) → _Self
Construct a model from a sequence of parameters.

Returns **decline curve** – The constructed decline curve model class.

Return type *DeclineCurve*

class petbox.dca.**Duong**(*qi: float*, *a: float*, *m: float*, *validate_params: Iterable[bool]* = <*factory*>)
Duong Model

Duong, A. N. 2001. Rate-Decline Analysis for Fracture-Dominated Shale Reservoirs. SPE Res Eval & Eng 14 (3): 377–387. SPE-137748-PA. https://doi.org/10.2118/137748-PA.

Parameters

- **qi** (*float*) – The initial production rate in units of volume / day *defined at ‘‘t=1 day‘‘*.
- **a** (*float*) – The a parameter. Roughly speaking, controls slope of the :func:q(t) function.
- **m** (*float*) – The m parameter. Roughly speaking, controls curvature of the:func:q(t) function.

rate(*t: Union[float, numpy.ndarray[Any, numpy.dtype[numpy.float64]]]*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]
Defines the model rate function:

$$q(t) = f(t)$$

where $f(t)$ is defined by each model.

Parameters **t** (*Union[float, numpy.NDFloat]*) – An array of times at which to evaluate the function.

Returns **rate**

Return type numpy.NDFloat

cum(*t: Union[float, numpy.ndarray[Any, numpy.dtype[numpy.float64]]]*, ***kwargs*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]
Defines the model cumulative volume function:

$$N(t) = \int_0^t q\, dt$$

Parameters

- **t** (*Union[float, numpy.NDFloat]*) – An array of times at which to evaluate the function.

- ****kwargs** – Additional arguments passed to `scipy.integrate.fixed_quad()` if needed.

Returns cumulative volume

Return type numpy.NDFloat

D(*t: Union[float, numpy.ndarray[Any, numpy.dtype[numpy.float64]]]*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]

Defines the model D-parameter function:

$$D(t) \equiv \frac{d}{dt} \ln q \equiv \frac{1}{q} \frac{dq}{dt}$$

Parameters **t** (*Union[float, numpy.NDFloat]*) – An array of times at which to evaluate the function.

Returns D-parameter

Return type numpy.NDFloat

beta(*t: Union[float, numpy.ndarray[Any, numpy.dtype[numpy.float64]]]*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]

Defines the model beta-parameter function.

$$\beta(t) \equiv \frac{d \ln q}{d \ln t} \equiv \frac{t}{q} \frac{dq}{dt} \equiv t\, D(t)$$

Parameters **t** (*Union[float, numpy.NDFloat]*) – An array of times at which to evaluate the function.

Returns beta-parameter

Return type numpy.NDFloat

b(*t: Union[float, numpy.ndarray[Any, numpy.dtype[numpy.float64]]]*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]

Defines the model b-parameter function:

$$b(t) \equiv \frac{d}{dt} \frac{1}{D}$$

Parameters **t** (*Union[float, numpy.NDFloat]*) – An array of times at which to evaluate the function.

Returns b-parameter

Return type numpy.NDFloat

classmethod get_param_desc(*name: str*) → petbox.dca.base.ParamDesc

Get a single parameter description.

Parameters **name** (*str*) – The parameter name.

Returns parameter description – A parameter description.

Return type ParamDesc

classmethod get_param_descs() → List[petbox.dca.base.ParamDesc]

Get the parameter descriptions.

Returns parameter description – A list of parameter descriptions.

Return type List[ParamDesc]

classmethod from_params(*params: Sequence[float]*) → _Self

Construct a model from a sequence of parameters.

Returns **decline curve** – The constructed decline curve model class.

Return type *DeclineCurve*

Associated Phase Models

Implementations of associated (secondary and water) phase GOR/CGR/WOR/WGR models

class petbox.dca.**PLYield**(*c: float*, *m0: float*, *m: float*, *t0: float*, *min: Optional[float] = None*, *max: Optional[float] = None*)

Power-Law Associated Phase Model.

Fulford, D.S. 2018. A Model-Based Diagnostic Workflow for Time-Rate Performance of Unconventional Wells. Presented at Unconventional Resources Conference in Houston, Texas, USA, 23–25 July. URTeC-2903036. https://doi.org/10.15530/urtec-2018-2903036.

Has the general form of

$$GOR = ct^m$$

and allows independent early-time and late-time slopes `m0` and `m` respectively.

Parameters

- **c** (*float*) – The value of GOR/CGR/WOR/CGR that acts as the anchor or pivot at `t=t0`. Units should be correctly specified for the respective yield function. Assumed volumes units per phase must be `Bbl` for oil and water and `Mscf` for gas in order to resolve any inconsistencies in unit magnitude.
- **m0** (*float*) – Early-time power-law slope.
- **m** (*float*) – Late-time power-law slope.
- **t0** (*float*) – The time of the anchor or pivot value `c`.
- **min** (*Optional[float] = None*) – The minimum allowed value. Would be used e.g. to limit minimum CGR.
- **max** (*Optional[float] = None*) – The maximum allowed value. Would be used e.g. to limit maximum GOR.

gor(*t: Union[float, numpy.ndarray[Any, numpy.dtype[numpy.float64]]]*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]

Defines the model GOR function. Implementation is idential to CGR function.

Parameters **t** (*Union[float, numpy.NDFloat]*) – An array of times at which to evaluate the function.

Returns **GOR** – The gas-oil ratio function in units of `Mscf / Bbl`.

Return type numpy.NDFloat

cgr(*t: Union[float, numpy.ndarray[Any, numpy.dtype[numpy.float64]]]*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]

Defines the model CGR function. Implementation is identical to GOR function.

Parameters **t** (*Union[float, numpy.NDFloat]*) – An array of times at which to evaluate the function.

Returns **CGR** – The condensate-gas ratio in units of `Bbl / Mscf`.

Return type numpy.NDFloat

rate(*t: Union[float, numpy.ndarray[Any, numpy.dtype[numpy.float64]]]*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]
Defines the model rate function:

$$q(t) = f(t)$$

where $f(t)$ is defined by each model.

Parameters **t** (*Union[float, numpy.NDFloat]*) – An array of times at which to evaluate the function.

Returns **rate**

Return type numpy.NDFloat

cum(*t: Union[float, numpy.ndarray[Any, numpy.dtype[numpy.float64]]], **kwargs*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]
Defines the model cumulative volume function:

$$N(t) = \int_0^t q\,dt$$

Parameters

- **t** (*Union[float, numpy.NDFloat]*) – An array of times at which to evaluate the function.
- ****kwargs** – Additional arguments passed to `scipy.integrate.fixed_quad()` if needed.

Returns **cumulative volume**

Return type numpy.NDFloat

D(*t: Union[float, numpy.ndarray[Any, numpy.dtype[numpy.float64]]]*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]
Defines the model D-parameter function:

$$D(t) \equiv \frac{d}{dt}\ln q \equiv \frac{1}{q}\frac{dq}{dt}$$

Parameters **t** (*Union[float, numpy.NDFloat]*) – An array of times at which to evaluate the function.

Returns **D-parameter**

Return type numpy.NDFloat

beta(*t: Union[float, numpy.ndarray[Any, numpy.dtype[numpy.float64]]]*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]
Defines the model beta-parameter function.

$$\beta(t) \equiv \frac{d\ln q}{d\ln t} \equiv \frac{t}{q}\frac{dq}{dt} \equiv t\,D(t)$$

Parameters **t** (*Union[float, numpy.NDFloat]*) – An array of times at which to evaluate the function.

Returns **beta-parameter**

Return type numpy.NDFloat

b(*t: Union[float, numpy.ndarray[Any, numpy.dtype[numpy.float64]]]*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]

Defines the model b-parameter function:

$$b(t) \equiv \frac{d}{dt}\frac{1}{D}$$

Parameters **t** (*Union[float, numpy.NDFloat]*) – An array of times at which to evaluate the function.

Returns **b-parameter**

Return type numpy.NDFloat

classmethod get_param_desc(*name: str*) → petbox.dca.base.ParamDesc

Get a single parameter description.

Parameters **name** (*str*) – The parameter name.

Returns **parameter description** – A parameter description.

Return type ParamDesc

classmethod get_param_descs() → List[petbox.dca.base.ParamDesc]

Get the parameter descriptions.

Returns **parameter description** – A list of parameter descriptions.

Return type List[ParamDesc]

classmethod from_params(*params: Sequence[float]*) → _Self

Construct a model from a sequence of parameters.

Returns **decline curve** – The constructed decline curve model class.

Return type *DeclineCurve*

Utility Functions

petbox.dca.**bourdet**(*y: numpy.ndarray[Any, numpy.dtype[numpy.float64]], x: numpy.ndarray[Any, numpy.dtype[numpy.float64]], L: float = 0.0, xlog: bool = True, ylog: bool = False*) → Tuple[numpy.ndarray[Any, numpy.dtype[numpy.float64]], numpy.ndarray[Any, numpy.dtype[numpy.float64]]]

Bourdet Derivative Smoothing

Bourdet, D., Ayoub, J. A., and Pirard, Y. M. 1989. Use of Pressure Derivative in Well-Test Interpretation. SPE Form Eval 4 (2): 293–302. SPE-12777-PA. https://doi.org/10.2118/12777-PA.

Parameters

- **y** (*numpy.NDFloat*) – An array of y values to compute the derivative for.
- **x** (*numpy.NDFloat*) – An array of x values.
- **L** (*float = 0.0*) – Smoothing factor in units of log-cycle fractions. A value of zero returns the point-by-point first-order difference derivative.
- **xlog** (*bool = True*) – Calculate the derivative with respect to the log of x, i.e. dy / d[ln x].
- **ylog** (*bool = False*) – Calculate the derivative with respect to the log of y, i.e. d[ln y] / dx.

Returns **der** – The calculated derivative.

Return type numpy.NDFloat

petbox.dca.**get_time**(*start: float = 1.0*, *end: float = 100000.0*, *n: int = 101*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]

Get a time array to evaluate with.

Parameters

- **start** (*float*) – The first time value of the array.
- **end** (*float*) – The last time value of the array.
- **n** (*int*) – The number of element in the array.

Returns **time** – An evenly-logspaced time series.

Return type numpy.NDFloat

petbox.dca.**get_time_monthly_vol**(*start: float = 1*, *end: int = 10000*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]

Get a time array to evaluate with.

Parameters

- **start** (*float*) – The first time value of the array.
- **end** (*float*) – The last time value of the array.

Returns **time** – An evenly-monthly-spaced time series

Return type numpy.NDFloat

Other Classes

class petbox.dca.**AssociatedPhase**(**args*)

Extends *DeclineCurve* for an associated phase forecast. Each model must implement the defined abstract _yieldfn() method.

class petbox.dca.**BothAssociatedPhase**(**args*)

Extends *DeclineCurve* for a general yield model used for both secondary phase and water phase.

class petbox.dca.base.**ParamDesc**(*name: str*, *description: str*, *lower_bound: Union[float, NoneType]*, *upper_bound: Union[float, NoneType]*, *naive_gen: Callable[[numpy.random.mtrand.RandomState, int], numpy.ndarray[Any, numpy.dtype[numpy.float64]]]*, *exclude_lower_bound: bool = False*, *exclude_upper_bound: bool = False*)

1.3.2 Detailed Usage Examples

Each model, including the secondary phase models, implements all diagnostic functions. The following is a set of examples to highlight functionality.

```
from petbox import dca
from data import rate as data_q, time as data_t
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl

plt.style.use('seaborn-v0_8-white')
plt.rcParams['font.size'] = 16
```

```
# Setup time series for Forecasts and calculate cumulative production of data

# We have this function handy
t = dca.get_time(n=1001)

# Calculate cumulative volume array of data
data_N = np.cumsum(data_q * np.r_[data_t[0], np.diff(data_t)])

# Calculate diagnostic functions D, beta, and b
data_D = -dca.bourdet(data_q, data_t, L=0.35, xlog=False, ylog=True)
data_beta = data_D * data_t
data_b = dca.bourdet(1 / data_D, data_t, L=0.25, xlog=False, ylog=False)
```

Primary Phase Decline Curve Models

Modified Hyperbolic Model

Robertson, S. 1988. Generalized Hyperbolic Equation. Available from SPE, Richardson, Texas, USA. SPE-18731-MS.

```
mh = dca.MH(qi=725, Di=0.85, bi=0.6, Dterm=0.2)
q_mh = mh.rate(t)
N_mh = mh.cum(t)
D_mh = mh.D(t)
b_mh = mh.b(t)
beta_mh = mh.beta(t)
N_mh *= data_N[-1] / mh.cum(data_t[-1])
```

Transient Hyperbolic Model

Fulford, D. S., and Blasingame, T. A. 2013. Evaluation of Time-Rate Performance of Shale Wells using the Transient Hyperbolic Relation. Presented at SPE Unconventional Resources Conference – Canada in Calgary, Alberta, Canda, 5–7 November. SPE-167242-MS. https://doi.org/10.2118/167242-MS.

```
thm = dca.THM(qi=750, Di=.8, bi=2, bf=.5, telf=28)
q_trans = thm.transient_rate(t)
N_trans = thm.transient_cum(t)
D_trans = thm.transient_D(t)
b_trans = thm.transient_b(t)
beta_trans = thm.transient_beta(t)
N_trans *= data_N[-1] / thm.transient_cum(data_t[-1])
```

Transient Hyperbolic Model Analytic Approximation

Fulford, D.S. 2018. A Model-Based Diagnostic Workflow for Time-Rate Performance of Unconventional Wells. Presented at Unconventional Resources Conference in Houston, Texas, USA, 23–25 July. URTeC-2903036. https://doi.org/10.15530/urtec-2018-2903036.

```
q_thm = thm.rate(t)
N_thm = thm.cum(t)
D_thm = thm.D(t)
b_thm = thm.b(t)
```

(continues on next page)

(continued from previous page)

```
beta_thm = thm.beta(t)
N_thm *= data_N[-1] / thm.cum(data_t[-1])
```

Timing Comparison

If performance is a consideration, the approximation is much faster.

```
>>> %timeit thm.transient_rate(t)
64.9 ms ± 5.81 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
>>> %timeit thm.rate(t)
86.9 µs ± 5.35 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)``
```

Power-Law Exponential Model

Ilk, D., Perego, A. D., Rushing, J. A., and Blasingame, T. A. 2008. Exponential vs. Hyperbolic Decline in Tight Gas Sands – Understanding the Origin and Implications for Reserve Estimates Using Arps Decline Curves. Presented at SPE Annual Technical Conference and Exhibition in Denver, Colorado, USA, 21–24 September. SPE-116731-MS. https://doi.org/10.2118/116731-MS.

Ilk, D., Rushing, J. A., and Blasingame, T. A. 2009. Decline Curve Analysis for HP/HT Gas Wells: Theory and Applications. Presented at SPE Annual Technical Conference and Exhibition in New Orleands, Louisiana, USA, 4–7 October. SPE-125031-MS. https://doi.org/10.2118/125031-MS.

```
ple = dca.PLE(qi=750, Di=.1, Dinf=.00001, n=.5)
q_ple = ple.rate(t)
N_ple = ple.cum(t)
D_ple = ple.D(t)
b_ple = ple.b(t)
beta_ple = ple.beta(t)
N_ple *= data_N[-1] / ple.cum(data_t[-1])
```

Stretched Exponential

Valkó, P. P. Assigning Value to Stimulation in the Barnett Shale: A Simultaneous Analysis of 7000 Plus Production Histories and Well Completion Records. 2009. Presented at SPE Hydraulic Fracturing Technology Conference in College Station, Texas, USA, 19–21 January. SPE-119369-MS. https://doi.org/10.2118/119369-MS.

```
se = dca.SE(qi=715, tau=90.0, n=.5)
q_se = se.rate(t)
N_se = se.cum(t)
D_se = se.D(t)
b_se = se.b(t)
beta_se = se.beta(t)
N_se *= data_N[-1] / se.cum(data_t[-1])
```

Duong Model

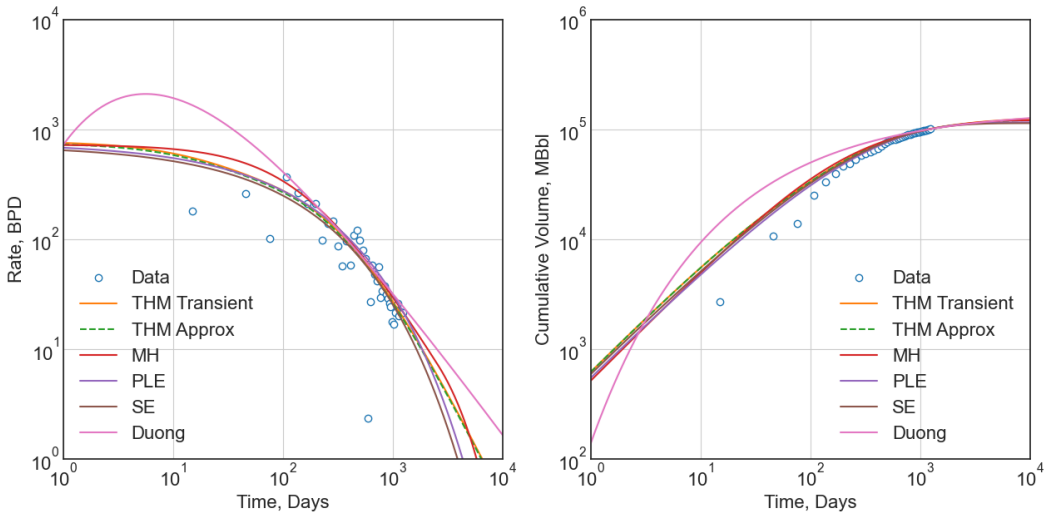
Duong, A. N. 2001. Rate-Decline Analysis for Fracture-Dominated Shale Reservoirs. SPE Res Eval & Eng 14 (3): 377–387. SPE-137748-PA. https://doi.org/10.2118/137748-PA.

```
dg = dca.Duong(qi=715, a=2.8, m=1.4)
q_dg = dg.rate(t)
N_dg = dg.cum(t)
D_dg = dg.D(t)
b_dg = dg.b(t)
beta_dg = dg.beta(t)
N_dg *= data_N[-1] / dg.cum(data_t[-1])
```

Primary Phase Diagnostic Plots

Rate and Cumulative Production Plots

```
# Rate vs Time
fig = plt.figure(figsize=(15, 7.5))
ax1 = fig.add_subplot(121)
ax2 = fig.add_subplot(122)

ax1.plot(data_t, data_q, 'o', mfc='w', label='Data')
ax1.plot(t, q_trans, label='THM Transient')
ax1.plot(t, q_thm, ls='--', label='THM Approx')
ax1.plot(t, q_mh, label='MH')
ax1.plot(t, q_ple, label='PLE')
ax1.plot(t, q_se, label='SE')
ax1.plot(t, q_dg, label='Duong')

ax1.set(xscale='log', yscale='log', ylabel='Rate, BPD', xlabel='Time, Days')
ax1.set(ylim=(1e0, 1e4), xlim=(1e0, 1e4))
ax1.set_aspect(1)
ax1.grid()
ax1.legend()

# Cumulative Volume vs Time
ax2.plot(data_t, data_N, 'o', mfc='w', label='Data')
ax2.plot(t, N_trans, label='THM Transient')
ax2.plot(t, N_thm, ls='--', label='THM Approx')
ax2.plot(t, N_mh, label='MH')
ax2.plot(t, N_ple, label='PLE')
ax2.plot(t, N_se, label='SE')
ax2.plot(t, N_dg, label='Duong')

ax2.set(xscale='log', yscale='log', ylim=(1e2, 1e6), xlim=(1e0, 1e4))
ax2.set(ylabel='Cumulative Volume, MBbl', xlabel='Time, Days')
ax2.set_aspect(1)
ax2.grid()
ax2.legend()

plt.savefig(img_path / 'model.png')
```



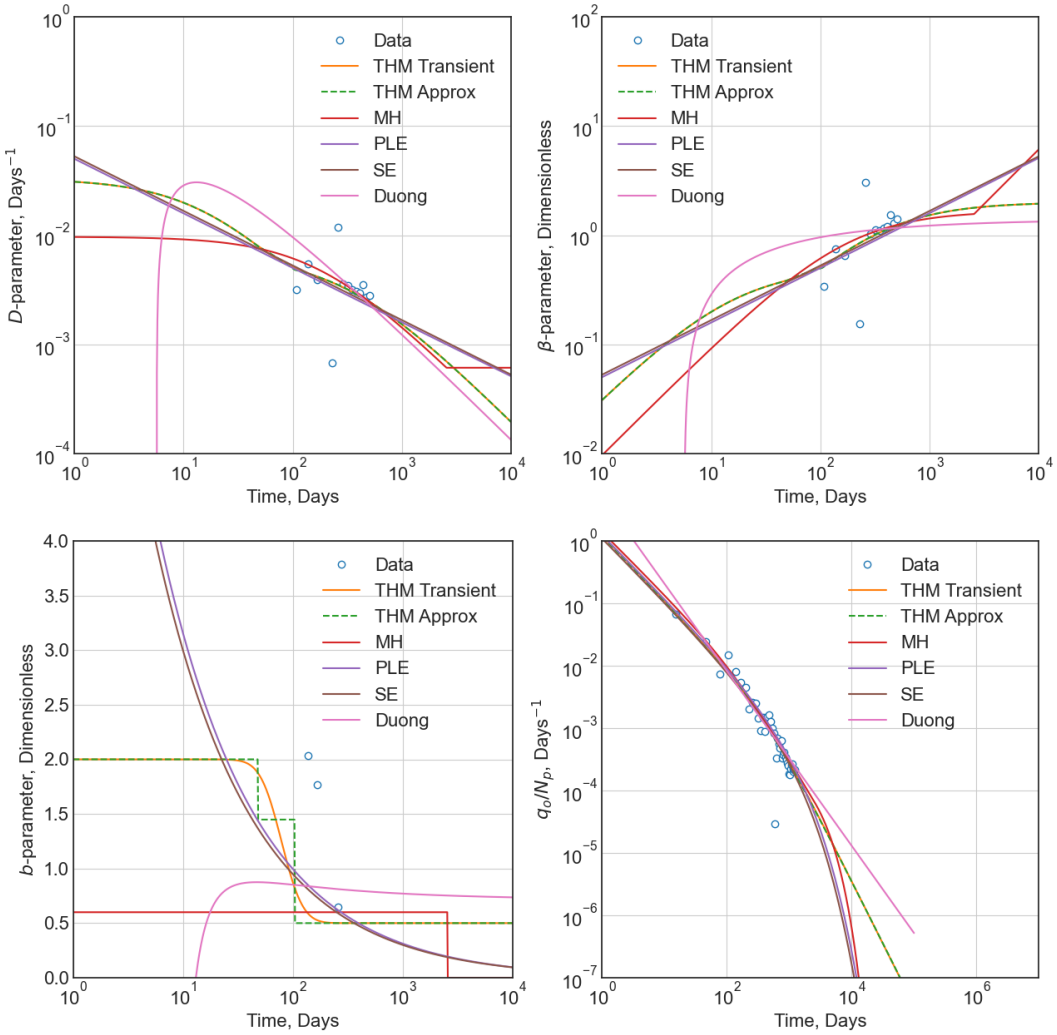
Diagnostic Function Plots

```
fig = plt.figure(figsize=(15, 15))
ax1 = fig.add_subplot(221)
ax2 = fig.add_subplot(222)
ax3 = fig.add_subplot(223)
ax4 = fig.add_subplot(224)

# D-parameter vs Time
ax1.plot(data_t, data_D, 'o', mfc='w', label='Data')
ax1.plot(t, D_trans, label='THM Transient')
ax1.plot(t, D_thm, ls='--', label='THM Approx')
ax1.plot(t, D_mh, label='MH')
ax1.plot(t, D_ple, label='PLE')
ax1.plot(t, D_se, label='SE')
ax1.plot(t, D_dg, label='Duong')
ax1.set(xscale='log', yscale='log', ylim=(1e-4, 1e0))
ax1.set(ylabel='$D$-parameter, Days$^{-1}$', xlabel='Time, Days')

# beta-parameter vs Time
ax2.plot(data_t, data_D * data_t, 'o', mfc='w', label='Data')
ax2.plot(t, beta_trans, label='THM Transient')
ax2.plot(t, beta_thm, ls='--', label='THM Approx')
ax2.plot(t, beta_mh, label='MH')
ax2.plot(t, beta_ple, label='PLE')
ax2.plot(t, beta_se, label='SE')
ax2.plot(t, beta_dg, label='Duong')
ax2.set(xscale='log', yscale='log', ylim=(1e-2, 1e2))
ax2.set(ylabel=r'$\beta$-parameter, Dimensionless', xlabel='Time, Days')

# b-parameter vs Time
ax3.plot(data_t, data_b, 'o', mfc='w', label='Data')
ax3.plot(t, b_trans, label='THM Transient')
ax3.plot(t, b_thm, ls='--', label='THM Approx')
```

(continues on next page)

(continued from previous page)

```
ax3.plot(t, b_mh, label='MH')
ax3.plot(t, b_ple, label='PLE')
ax3.plot(t, b_se, label='SE')
ax3.plot(t, b_dg, label='Duong')
ax3.set(xscale='log', yscale='linear', ylim=(0., 4.))
ax3.set(ylabel='$b$-parameter, Dimensionless', xlabel='Time, Days')

# q/N vs Time
ax4.plot(data_t, data_q / data_N, 'o', mfc='w', label='Data')
ax4.plot(t, q_trans / N_trans, label='THM Transient')
ax4.plot(t, q_thm / N_thm, ls='--', label='THM Approx')
ax4.plot(t, q_mh / N_mh, label='MH')
ax4.plot(t, q_ple / N_ple, label='PLE')
ax4.plot(t, q_se / N_se, label='SE')
ax4.plot(t, q_dg / N_dg, label='Duong')
ax4.set(xscale='log', yscale='log', ylim=(1e-7, 1e0), xlim=(1e0, 1e7))
ax4.set(ylabel='$q_o / N_p$, Days$^{-1}$', xlabel='Time, Days')

for ax in [ax1, ax2, ax3, ax4]:
    if ax != ax4:
        ax.set(xlim=(1e0, 1e4))
    if ax != ax3:
        ax.set_aspect(1)
    ax.grid()
    ax.legend()


plt.savefig(img_path / 'diagnostics.png')
```



Secondary Phase Decline Curve Models

Power-Law GOR/CGR Model

Fulford, D.S. 2018. A Model-Based Diagnostic Workflow for Time-Rate Performance of Unconventional Wells. Presented at Unconventional Resources Conference in Houston, Texas, USA, 23–25 July. URTeC-2903036. https://doi.org/10.15530/urtec-2018-2903036.

```
thm = dca.THM(qi=750, Di=.8, bi=2, bf=.5, telf=28)
thm.add_secondary(dca.PLYield(c=1000, m0=-0.1, m=0.8, t0=2 * 365.25 / 12, max=10_000))
```

Secondary Phase Diagnostic Plots

Rate and Cumluative Production Plots

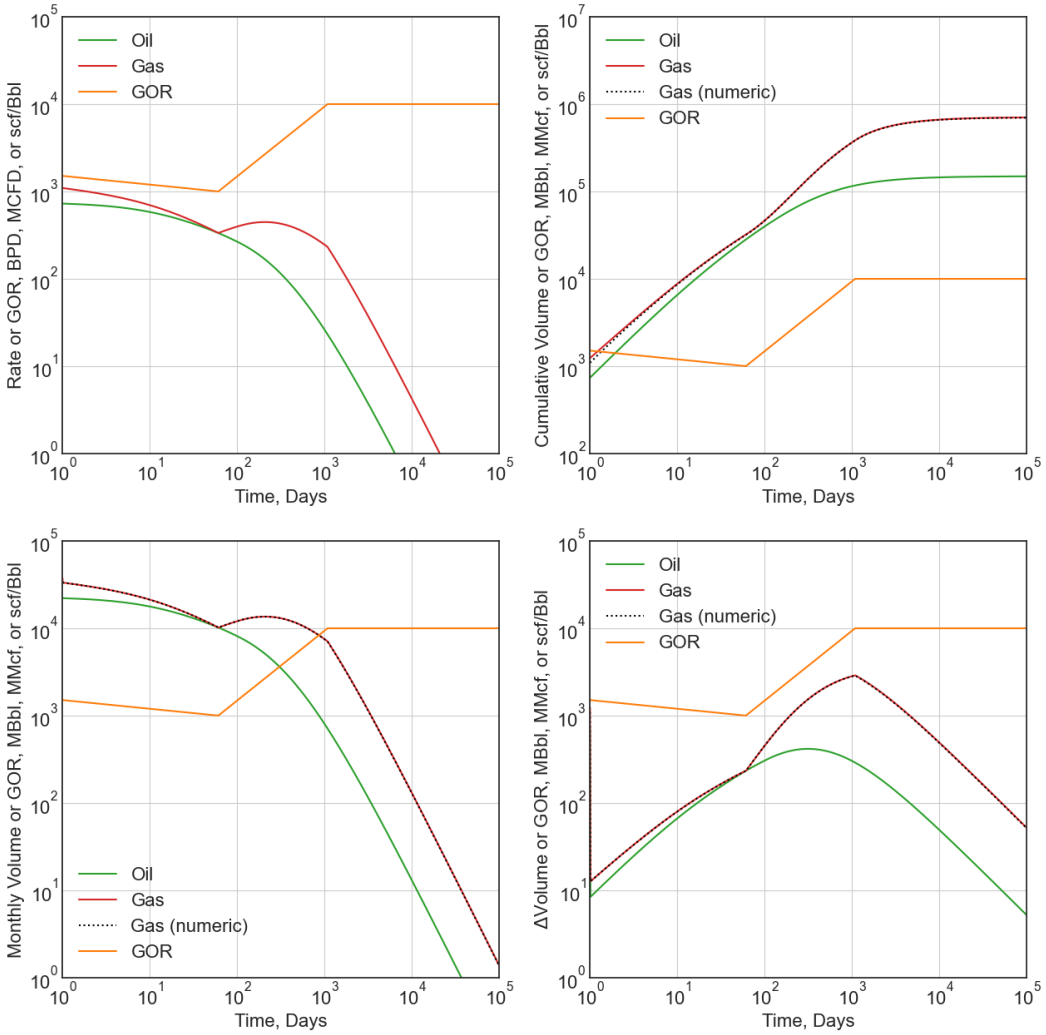
Numeric calculation provided to verify analytic relationships

```
fig = plt.figure(figsize=(15, 15))
ax1 = fig.add_subplot(221)
ax2 = fig.add_subplot(222)
ax3 = fig.add_subplot(223)
ax4 = fig.add_subplot(224)


# Rate vs Time
q = thm.rate(t)
g = thm.secondary.rate(t) / 1000.0
y = thm.secondary.gor(t)

ax1.plot(t, q, c='C2', label='Oil')
ax1.plot(t, g, c='C3', label='Gas')
ax1.plot(t, y, c='C1', label='GOR')
ax1.set(xscale='log', yscale='log', xlim=(1e0, 1e5), ylim=(1e0, 1e5))
ax1.set(ylabel='Rate or GOR, BPD, MCFD, or scf/Bbl', xlabel='Time, Days')


# Cumulative Volume vs Time
q_N = thm.cum(t)
g_N = thm.secondary.cum(t) / 1000.0
_g_N = np.cumsum(g * np.diff(t, prepend=0))

ax2.plot(t, q_N, c='C2', label='Oil')
ax2.plot(t, g_N, c='C3', label='Gas')
ax2.plot(t, _g_N, c='k', ls=':', label='Gas (numeric)')
ax2.plot(t, y, c='C1', label='GOR')
ax2.set(xscale='log', yscale='log', xlim=(1e0, 1e5), ylim=(1e2, 1e7))
ax2.set(ylabel='Rate, Dimensionless', xlabel='Time, Days')
ax2.set(ylabel='Cumulative Volume or GOR, MBbl, MMcf, or scf/Bbl', xlabel='Time, Days
↪')


# Time vs Monthly Volume
q_MN = thm.monthly_vol_equiv(t)
g_MN = thm.secondary.monthly_vol_equiv(t) / 1000.0
_g_MN = np.diff(np.cumsum(g * np.diff(t, prepend=0)), prepend=0) \
    / np.diff(t, prepend=0) * dca.DAYS_PER_MONTH

ax3.plot(t, q_MN, c='C2', label='Oil')
ax3.plot(t, g_MN, c='C3', label='Gas')
ax3.plot(t, _g_MN, c='k', ls=':', label='Gas (numeric)')
ax3.plot(t, y, c='C1', label='GOR')
ax3.set(xscale='log', yscale='log', xlim=(1e0, 1e5), ylim=(1e0, 1e5))
ax3.set(ylabel='Monthly Volume or GOR, MBbl, MMcf, or scf/Bbl', xlabel='Time, Days')


# Time vs Interval Volume
q_IN = thm.interval_vol(t, t0=0.0)
g_IN = thm.secondary.interval_vol(t, t0=0.0) / 1000.0
```

(continues on next page)

(continued from previous page)

```
_g_IN = np.diff(np.cumsum(g * np.diff(t, prepend=0)), prepend=0)

ax4.plot(t, q_IN, c='C2', label='Oil')
ax4.plot(t, g_IN, c='C3', label='Gas')
ax4.plot(t, _g_IN, c='k', ls=':', label='Gas (numeric)')
ax4.plot(t, y, c='C1', label='GOR')
ax4.set(xscale='log', yscale='log', xlim=(1e0, 1e5), ylim=(1e0, 1e5))
ax4.set(ylabel='$\Delta$Volume or GOR, MBbl, MMcf, or scf/Bbl', xlabel='Time, Days')

for ax in [ax1, ax2, ax3, ax4]:
    ax.set_aspect(1)
    ax.grid()
    ax.legend()

plt.savefig(img_path / 'secondary_model.png')
```



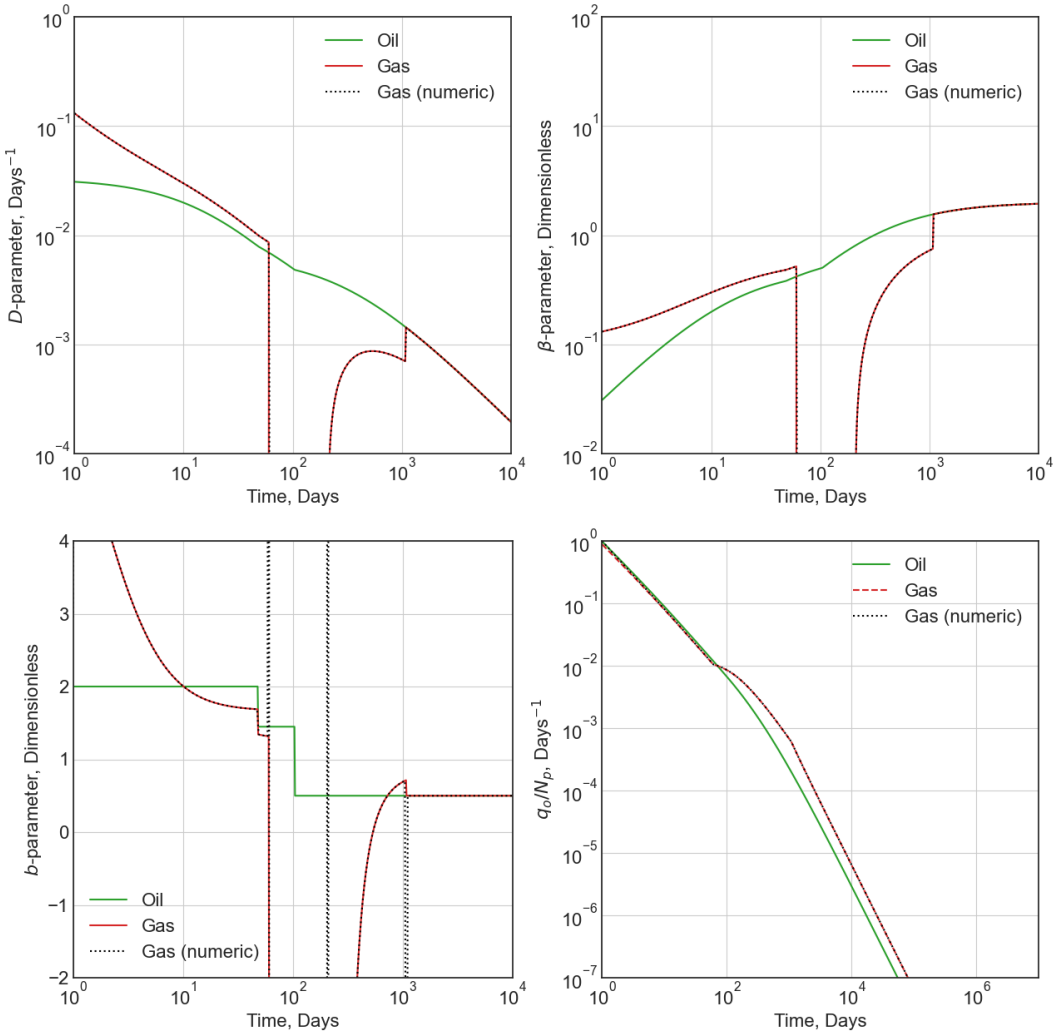
Diagnostic Function Plots

```
fig = plt.figure(figsize=(15, 15))
ax1 = fig.add_subplot(221)
ax2 = fig.add_subplot(222)
ax3 = fig.add_subplot(223)
ax4 = fig.add_subplot(224)

# D-parameter vs Time
q_D = thm.D(t)
g_D = thm.secondary.D(t)
_g_D = -np.gradient(np.log(thm.secondary.rate(t) / 1000.0), t)
```

(continues on next page)

(continued from previous page)

```
ax1.plot(t, q_D, c='C2', label='Oil')
ax1.plot(t, g_D, c='C3', label='Gas')
ax1.plot(t, _g_D, c='k', ls=':', label='Gas (numeric)')
ax1.set(xscale='log', yscale='log', xlim=(1e0, 1e4), ylim=(1e-4, 1e0))
ax1.set(ylabel='$D$-parameter, Days$^{-1}$', xlabel='Time, Days')

# beta-parameter vs Time
q_beta = thm.beta(t)
g_beta = thm.secondary.beta(t)
_g_beta = _g_D * t

ax2.plot(t, q_beta, c='C2', label='Oil')
ax2.plot(t, g_beta, c='C3', label='Gas')
ax2.plot(t, _g_beta, c='k', ls=':', label='Gas (numeric)')
ax2.set(xscale='log', yscale='log', xlim=(1e0, 1e4), ylim=(1e-2, 1e2))
ax2.set(ylabel=r'$\beta$-parameter, Dimensionless', xlabel='Time, Days')

# b-parameter vs Time
q_b = thm.b(t)
g_b = thm.secondary.b(t)
_g_b = np.gradient(1.0 / _g_D, t)

ax3.plot(t, q_b, c='C2', label='Oil')
ax3.plot(t, g_b, c='C3', label='Gas')
ax3.plot(t, _g_b, c='k', ls=':', label='Gas (numeric)')
ax3.set(xscale='log', yscale='linear', xlim=(1e0, 1e4), ylim=(-2, 4))
ax3.set(ylabel='$b$-parameter, Dimensionless', xlabel='Time, Days')

# q/N vs Time
q_Ng = thm.rate(t) / thm.cum(t)
g_Ng = thm.secondary.rate(t) / thm.secondary.cum(t)
_g_Ng = thm.secondary.rate(t) / np.cumsum(thm.secondary.rate(t) * np.diff(t,␣
↪prepend=0))

ax4.plot(t, q_Ng, c='C2', label='Oil')
ax4.plot(t, g_Ng, c='C3', ls='--', label='Gas')
ax4.plot(t, _g_Ng, c='k', ls=':', label='Gas (numeric)')
ax4.set(xscale='log', yscale='log', ylim=(1e-7, 1e0), xlim=(1e0, 1e7))
ax4.set(ylabel='$q_o / N_p$, Days$^{-1}$', xlabel='Time, Days')

for ax in [ax1, ax2, ax3, ax4]:
    if ax != ax3:
        ax.set_aspect(1)
    ax.grid()
    ax.legend()

plt.savefig(img_path / 'sec_diagnostic_funs.png')
```



Additional Diagnostic Plots

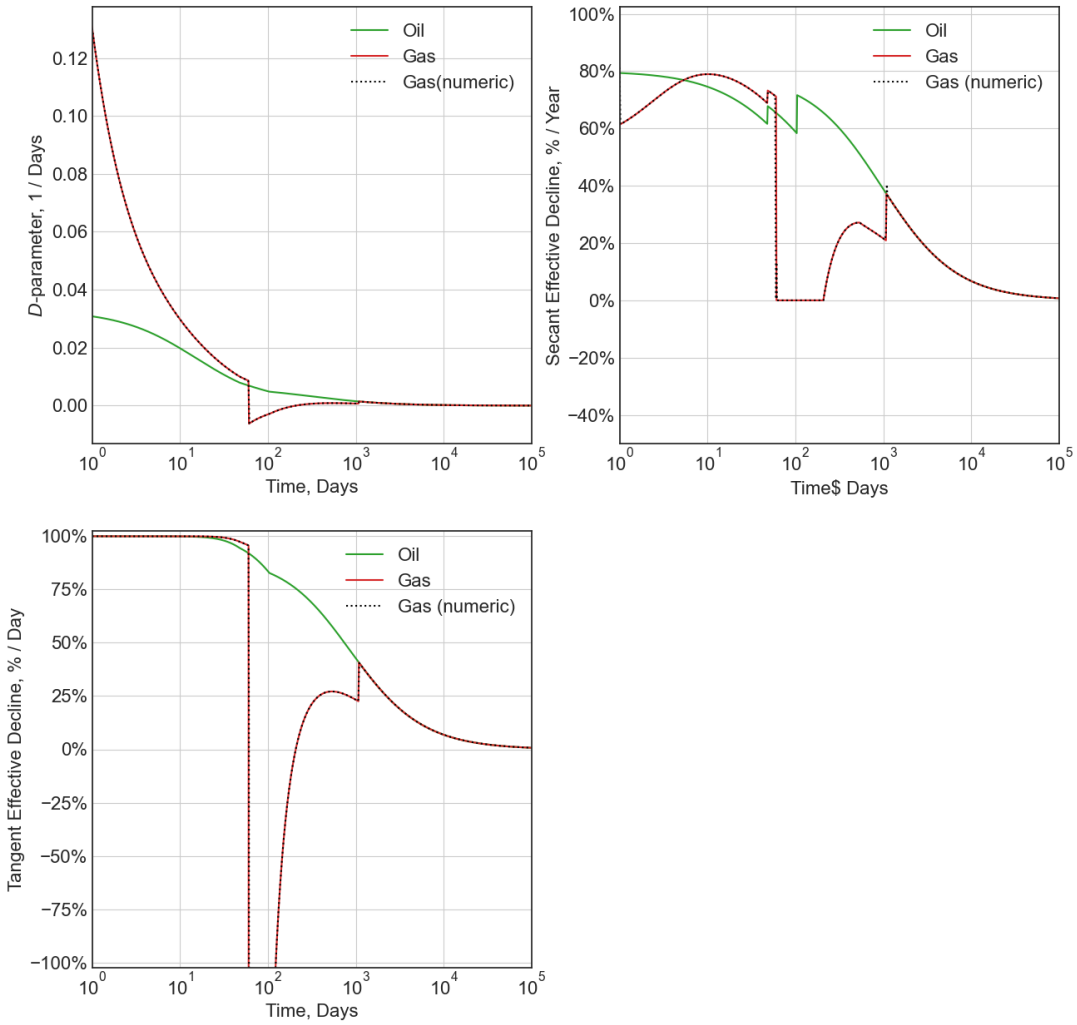
Numeric calculation provided to verify analytic relationships

```
fig = plt.figure(figsize=(15, 15))
ax1 = fig.add_subplot(221)
ax2 = fig.add_subplot(222)
ax3 = fig.add_subplot(223)

# D-parameter vs Time
q_D = thm.D(t)
g_D = thm.secondary.D(t)
_g_D = -np.gradient(np.log(thm.secondary.rate(t)), t)
```

(continues on next page)

(continued from previous page)

```
ax1.plot(t, q_D, c='C2', label='Oil')
ax1.plot(t, g_D, c='C3', label='Gas')
ax1.plot(t, _g_D, c='k', ls=':', label='Gas(numeric)')
ax1.set(xscale='log', yscale='linear', xlim=(1e0, 1e5), ylim=(None, None))
ax1.set(ylabel='$D$-parameter, 1 / Days', xlabel='Time, Days')

# Secant Effective Decline vs Time
secant_from_nominal = dca.MultisegmentHyperbolic.secant_from_nominal
dpy = dca.DAYS_PER_YEAR

q_Dn = [secant_from_nominal(d * dpy, b) for d, b in zip(q_D, thm.b(t))]
g_Dn = [secant_from_nominal(d * dpy, b) for d, b in zip(g_D, thm.secondary.b(t))]
_g_Dn = [secant_from_nominal(d * dpy, b) for d, b in zip(_g_D, np.gradient(1 / _g_D,␣
↪t))]

ax2.plot(t, q_Dn, c='C2', label='Oil')
ax2.plot(t, g_Dn, c='C3', label='Gas')
ax2.plot(t, _g_Dn, c='k', ls=':', label='Gas (numeric)')
ax2.set(xscale='log', yscale='linear', xlim=(1e0, 1e5), ylim=(-.5, 1.025))
ax2.yaxis.set_major_formatter(mpl.ticker.PercentFormatter(xmax=1))
ax2.set(ylabel='Secant Effective Decline, % / Year', xlabel='Time$ Days')

# Tangent Effective Decline vs Time
ax3.plot(t, 1 - np.exp(-q_D * dpy), c='C2', label='Oil')
ax3.plot(t, 1 - np.exp(-g_D * dpy), c='C3', label='Gas')
ax3.plot(t, 1 - np.exp(-_g_D * dpy), c='k', ls=':', label='Gas (numeric)')
ax3.set(xscale='log', yscale='linear', xlim=(1e0, 1e5), ylim=(-1.025, 1.025))
ax3.yaxis.set_major_formatter(mpl.ticker.PercentFormatter(xmax=1))
ax3.set(ylabel='Tangent Effective Decline, % / Day', xlabel='Time, Days')

for ax in [ax1, ax2, ax3]:
    ax.grid()
    ax.legend()

plt.savefig(img_path / 'sec_decline_diagnostics.png')
```



1.3.3 Testing

Testing is set to evaluate:

- style with flake8,
- typing with mypy,
- valid function return values and behaviors with hypothesis, and
- test coverage using coverage.

Windows

Run `test.bat` in the `test` directory.

Linux

Run `test.sh` in the `test` directory.

1.3.4 Version History

1.1.0

- **Bug Fix**
  - Fix bug in sign in `MultisegmentHyperbolic.secant_from_nominal`
- **Other changes**
  - Add *mpmath* to handle precision requires of THM transient functions (only required to use the functions)
  - Adjust default degree of THM transient function quadrature integration from 50 to 10 (*scipy* default is 5)
  - Update package versions for docs and builds
  - Address various floating point errors, suppress *numpy* warnings for those which are mostly unavoidable
  - Add test/doc_exapmles.py and update figures (not sure what happened to the old file)
  - Adjust range of values in tests to avoid numerical errors in *numpy* and *scipy* functions... these were near-epsilon impractical values anyway

1.0.8

- **New functions**
  - Added `WaterPhase.wgr` method
- **Other changes**
  - Adjust yield model rate function to return consistent units if primary phase is oil or gas
  - Update to *numpy v1.20* typing

1.0.7

- **Allow disabling of parameter checks by passing an interable of booleans, each indicating a check** to each model parameter.
- Explicitly handle floating point overflow errors rather than relying on *numpy*.

1.0.6

- **New functions**
  - Added `WaterPhase` class
  - Added `WaterPhase.wor` method
  - Added `PrimaryPhase.add_water` method
- **Other changes**
  - A `yield` model may inherit both `SecondaryPhase` and `WaterPhase`, with the respective methods removed upon attachment to a `PrimaryPhase`.

1.0.5

- **New functions**
  - Bourdet algorithm
- **Other changes**
  - Update docstrings
  - Add bourdet data derivatives to detailed use examples

1.0.4

- Fix typos in docs

1.0.3

- Add documentation
- Genericize numerical integration
- Various refactoring

0.0.1 - 1.0.2

- Internal releases

1.3.5 Index

Index

I

M

P

R

S

T

W