PyDDM Documentation

Release 0.7.0

Maxwell Shinn, Norman Lam

Jul 02, 2023

Contents:

PyDDM is a simulator and modeling framework for **generalized drift-diffusion models** (GDDM or DDM), with a focus on cognitive neuroscience.

Key features include:

- **Fast solutions to generalized drift-diffusion models**, allowing data-fitting with a large number of parameters
- Fokker-Planck equation solved numerically using Crank-Nicolson and backward Euler methods for **likelihood fitting on the full distribution**
- **Arbitrary functions for parameters** drift rate, noise, bounds, and initial position distribution
- Arbitrary loss function and fitting method for parameter fitting
- **Multiprocessor support**
- **Optional GUI** for debugging and gaining an intuition for different models
- Convenient and extensible object oriented API allows building models in a component-wise fashion
- Verified accuracy of simulations using software verification techniques

Interactive online demo on Google Colab.

Start with the *tutorial*. To see what PyDDM is capable of, and for example models, see the *PyDDM Cookbook*. Also see the *FAQs* for more information.

Release annoucments are posted on the pyddm-announce mailing list and on github.

Please note that PyDDM is still beta software so you may experience some glitches or uninformative error messages. Please report any problems to the bug tracker.

CHAPTER 1

Installation

1.1 System requirements

- Python 3.5 or higher
- Scipy/numpy
- Paranoid scientist (`pip install paranoid-scientist`)
- For plotting features, matplotlib
- For parallelization support, pathos

1.2 Installation

Normally, you can install with:

```
pip install pyddm
```

If you are in a shared environment (e.g. a cluster), install with:

```
pip install pyddm --user
```

If installing from source, download, extract, and do:

```
python3 setup.py install
```

1.3 Getting the source

Source code available on Github.

1.4 Contributing

Please report bugs to https://github.com/mwshinn/pyddm/issues. This includes any problems with the documentation. Fixes (in the form of pull requests) for bugs are greatly appreciated.

Feature requests are currently not being accepted due to limited resources, however if you implement the feature yourself we are open to accepting it in PyDDM. If you implement a new feature in PyDDM, please do the following before submitting a pull request on Github:

- Make sure your code is clean and well commented
- If appropriate, update the official documentation in the `docs/` directory
- Ensure there are Paranoid Scientist verification conditions to your code (if appropriate)
- Write unit tests and optionally integration tests for your new feature (please add them to `unit_tests.py` and `integration_tests.py`)
- Ensure all existing tests pass (`runtests.sh` returns without error)

For all other questions or comments, contact Max Shinn.

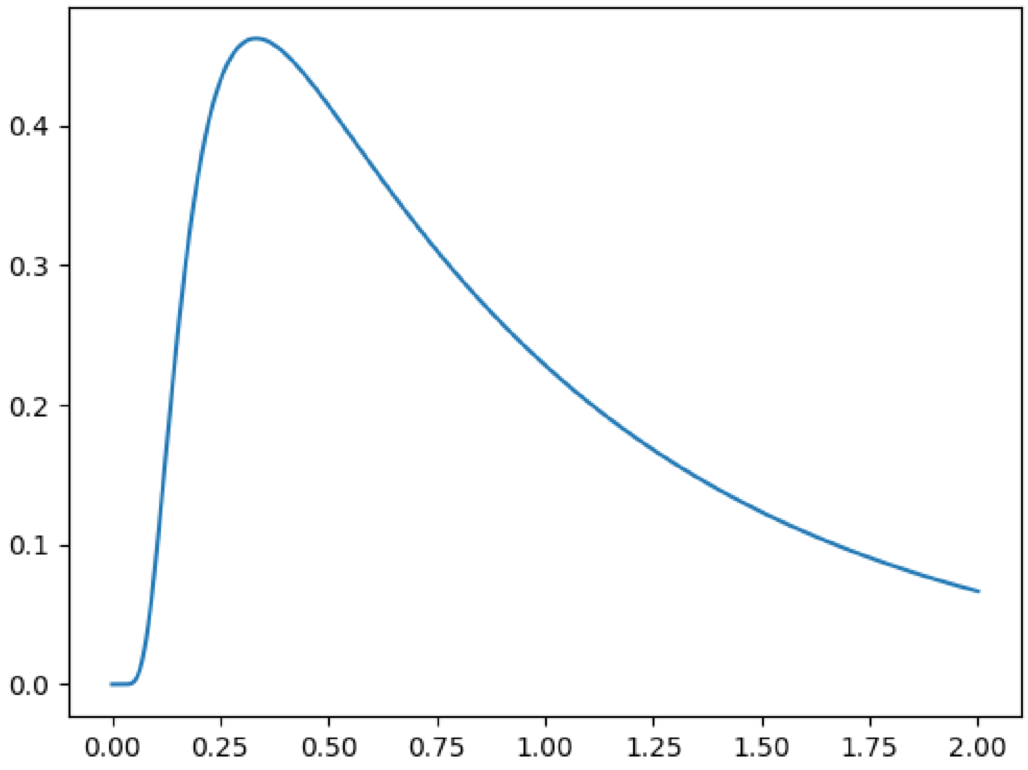
CHAPTER 2

Quick Start guide

2.1 Hello, world!

To get started, let's simulate a basic model and plot it. For simplicity, we will use all of the model defaults.

```python
import matplotlib.pyplot as plt
from pyddm import Model
m = Model()
s = m.solve()
plt.plot(s.t_domain, s.pdf("correct"))
plt.savefig("helloworld.png")
plt.show()
```



Congratulations! You've successfully simulated your first model! Let's dig in a bit so that we can define more useful models.

`Download this full example`

2.2 Simple example

The following simulates a simple DDM with constant drift. First, it shows how to build a model and then uses it to generate artificial data. After the artificial data has been generated, it fits a new model to these data and shows that the parameters are similar.

`Model` is the object which represents a DDM. Its default behavior can be changed through *`Drift`*, *`Noise`*, *`Bound`*, *`InitialCondition`*, and *`Overlay`* objects, which specify the behavior of each of these model components. Each model must have one of each of these, but defaults to a simple case for each. These determine how it calculates the drift rate (*`Drift`*), diffusion coefficient (*`Noise`*), shape of the integration boundaries (*`Bound`*), initial particle distribution (*`InitialCondition`*), and any other modifications to the generated solution (*`Overlay`*).

Each of these model components may take "parameters" which are (usually) unique to that specific model component. For instance, the *`DriftConstant`* class takes the "drift" parameter, which should be passed to the constructor. Similarly, *`NoiseConstant`* takes the parameter "noise" to determine the standard deviation of the drift process, and *`OverlayNonDecision`* takes "nondectime", the non-decision time (efferent/afferent/apparatus delay) in seconds. Some model components, such as *`ICPointSourceCenter`* which represents a starting point initial condition directly in between the bounds, does not take any parameters.

For example, the following is a DDM with drift 2.2, noise 1.5, bound 1.1, and a 100ms non-decision time. It is simulated for 2 seconds (`T_dur`) with reasonable timestep and grid size (`dt` and `dx`). Once we define the model, the `solve()` function runs the simulation. This model can be described as shown below:

```
from pyddm import Model
from pyddm.models import DriftConstant, NoiseConstant, BoundConstant,␣
↪OverlayNonDecision, ICPointSourceCenter
from pyddm.functions import fit_adjust_model, display_model

model = Model(name='Simple model',
              drift=DriftConstant(drift=2.2),
              noise=NoiseConstant(noise=1.5),
              bound=BoundConstant(B=1.1),
              overlay=OverlayNonDecision(nondectime=.1),
              dx=.001, dt=.01, T_dur=2)
display_model(model)
sol = model.solve()
```

Solution objects represent the probability distribution functions over time for choices associated with upper and lower bound crossings. By default, this is "correct" and "error" responses, respectively. We can generate psuedo-data from this solved model with the `resample()` function:

```
samp = sol.resample(1000)
```

To fit the outputs, we first create a model with special `Fittable` objects in all the parameters we would like to fit. We specify the range of each of these objects as a hint to the optimizer; this is mandatory for some but not all optimization methods. Then, we run the `fit_adjust_model()` function, which will convert the `Fittable` objects to `Fitted` objects and find a value for each which collectively minimizes the objective function.

Here, we use the same model as above, since we know the form the model is supposed to have. We fit the model to the generated data using BIC as a loss function and differential evolution to optimize the parameters:

```
from pyddm import Fittable, Fitted
from pyddm.models import LossRobustBIC
from pyddm.functions import fit_adjust_model
model_fit = Model(name='Simple model (fitted)',
                  drift=DriftConstant(drift=Fittable(minval=0, maxval=4)),
                  noise=NoiseConstant(noise=Fittable(minval=.5, maxval=4)),
                  bound=BoundConstant(B=1.1),
                  overlay=OverlayNonDecision(nondectime=Fittable(minval=0, maxval=1)),
                  dx=.001, dt=.01, T_dur=2)

fit_adjust_model(samp, model_fit,
                 fitting_method="differential_evolution",
                 lossfunction=LossRobustBIC, verbose=False)
```

We can display the newly-fit parameters with the `display_model()` function:

```
display_model(model_fit)
```

This shows that the fitted value of drift is 2.2096, which is close to the value of 2.2 we used to simulate it. Similarly, noise fits to 1.539 (compared to 1.5) and nondectime (non-decision time) to 0.1193 (compared to 0.1). The fitting algorithm is stochastic, so exact values may vary slightly:

```
Model Simple model (fitted) information:
Drift component DriftConstant:
    constant
```

(continues on next page)

(continued from previous page)

```
    Fitted parameters:
    - drift: 2.209644
Noise component NoiseConstant:
    constant
    Fitted parameters:
    - noise: 1.538976
Bound component BoundConstant:
    constant
    Fixed parameters:
    - B: 1.100000
IC component ICPointSourceCenter:
    point_source_center
    (No parameters)
Overlay component OverlayNonDecision:
    Add a non-decision by shifting the histogram
    Fitted parameters:
    - nondectime: 0.119300
Fit information:
    Loss function: BIC
    Loss function value: 562.1805934500456
    Fitting method: differential_evolution
    Solver: auto
    Other properties:
        - nparams: 3
        - samplesize: 1000
        - mess: ''
```

While the values are close to the true values, the fit is not perfect due to the finite size of the resampled data. If we want to use these programmatically, we can use the *parameters()* function, like:

```
model_fit.parameters()
```

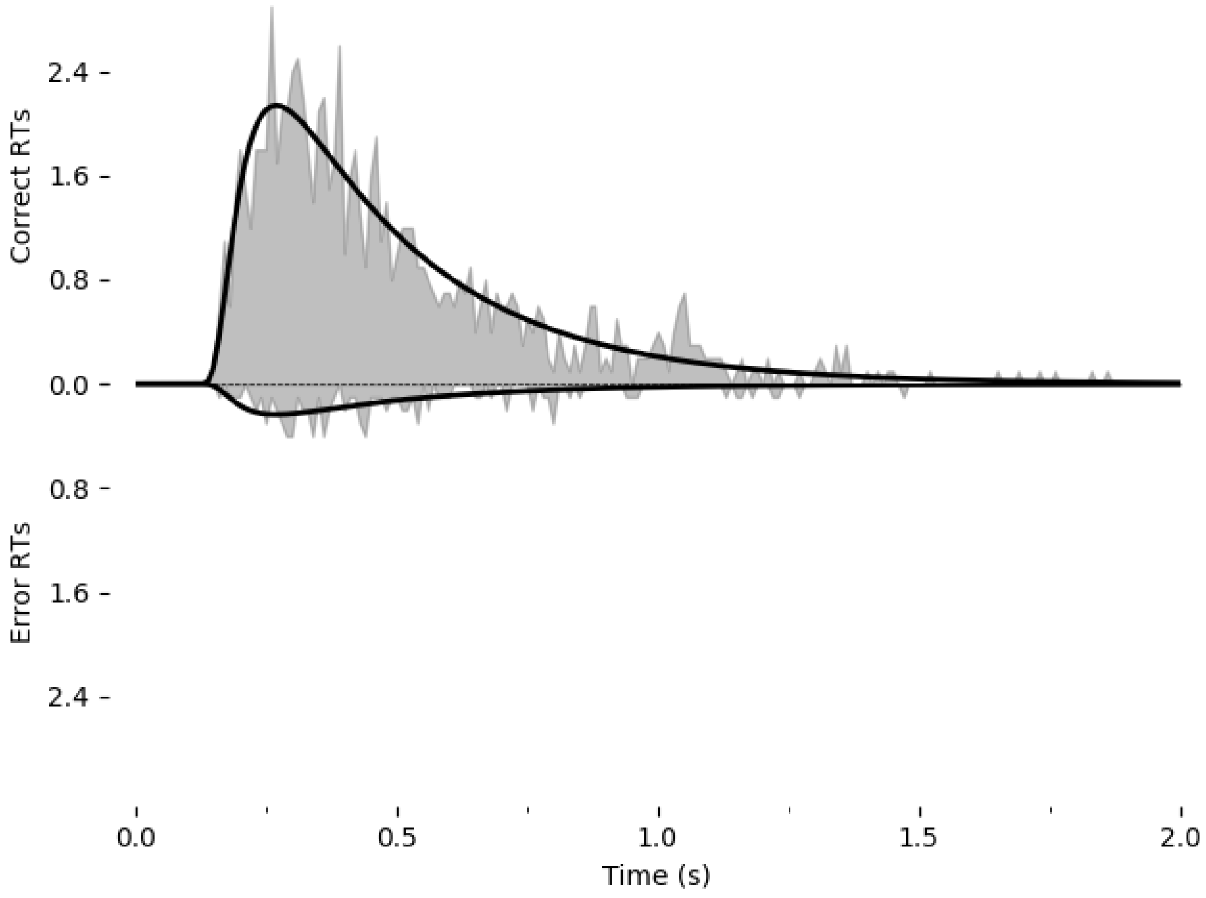
The *Fitted* objects it returns can be used anywhere as if it is a normal number/float. (Actually, *Fitted* is a subclass of “float”!)

We can also examine different properties of the fitting process in the *FitResult* object. For instance, to get the value of the loss function, we can do:

```
model_fit.get_fit_result().value()
```

We can also draw a plot visualizing the fit. Unlike our first example, we will now use one of PyDDM’s convenience methods, *plot_fit_diagnostics()*:

```
import pyddm.plot
import matplotlib.pyplot as plt
pyddm.plot.plot_fit_diagnostics(model=model_fit, sample=samp)
plt.savefig("simple-fit.png")
plt.show()
```



Using the *Solution* object `sol` we have access to a number of other useful functions. For instance, we can find the probability of a response using *prob()*, such as `sol.prob("correct")` for the probability of a correct response, or the entire histogram of responses using *pdf()*, such as `sol.pdf("error")` for the distribution of errors.

```
print(sol.prob("correct"))
print(sol.pdf("error"))
```

See *the Solution object documentation* for more such functions.

We could also named the upper and lower boundary as something else (e.g. “left” and “right” response), sometimes called “stimulus coding”. To do this, we need to pass the “choice_names” parameter to the Sample and the Model object. See *the section on stimulus coding*

To save the model for later use, we can use the repr() function built-in to Python. This function outputs a string with all the information needed to recreate the object. It is very similar to the print() function, except it saves the result as a string rather than displaying it. To save, do the following:

```
with open("model.txt", "w") as f:
    f.write(repr(model_fit))
```

Then, you can load the saved model with the following. You may need to add additional imports from PyDDM if you get an “import error”.

```
from pyddm import FitResult
with open("model.txt", "r") as f:
    model_loaded = eval(f.read())
```

`Download this full example`

2.3 Working with data

(View a shortened interactive version of this tutorial.)

2.3.1 Loading data from a CSV file

In this example, we load data from the open dataset by Roitman and Shadlen (2002). This dataset can be downloaded here and the relevant data extracted `with our script`. The processed CSV file can be `downloaded directly`.

The CSV file generated from this looks like the following:

monkey	rt	coh	correct	trgchoice
1	0.355	0.512	1.0	2.0
1	0.359	0.256	1.0	1.0
1	0.525	0.128	1.0	1.0

It is fairly easy then to load and process the CSV file:

```
from pyddm import Sample
import pandas
with open("roitman_rts.csv", "r") as f:
    df_rt = pandas.read_csv(f)

df_rt = df_rt[df_rt["monkey"] == 1] # Only monkey 1

# Remove short and long RTs, as in 10.1523/JNEUROSCI.4684-04.2005.
# This is not strictly necessary, but is performed here for
# compatibility with this study.
df_rt = df_rt[df_rt["rt"] > .1] # Remove trials less than 100ms
df_rt = df_rt[df_rt["rt"] < 1.65] # Remove trials greater than 1650ms

# Create a sample object from our data.  This is the standard input
# format for fitting procedures.  Since RT and correct/error are
# both mandatory columns, their names are specified by command line
# arguments.
roitman_sample = Sample.from_pandas_dataframe(df_rt, rt_column_name="rt", choice_
↪column_name="correct")
```

This gives an output sample with the conditions “monkey”, “coh”, and “trgchoice”.

Note that this examples requires pandas to be installed.

2.3.2 Loading data from a numpy array

Data can also be loaded from a numpy array. For example, let’s load the above data without first loading it into pandas:

```
from pyddm import Sample
import numpy as np
with open("roitman_rts.csv", "r") as f:
```

(continues on next page)

(continued from previous page)

```
    M = np.loadtxt(f, delimiter=",", skiprows=1)

# RT data must be the first column and correct/error must be the
# second column.
rt = M[:,1].copy() # Use .copy() because np returns a view
corr = M[:,3].copy()
monkey = M[:,0].copy()
M[:,0] = rt
M[:,1] = corr
M[:,3] = monkey

# Only monkey 1
M = M[M[:,3]==1,:]

# As before, remove longest and shortest RTs
M = M[M[:,0]>.1,:]
M = M[M[:,0]<1.65,:]

conditions = ["coh", "monkey", "trgchoice"]
roitman_sample2 = Sample.from_numpy_array(M, conditions)
```

We can confirm that these two methods of loading data produce the same results:

```
assert roitman_sample == roitman_sample2
```

2.3.3 Fitting a model to data

Now that we have loaded these data, we can fit a model to them. First we will fit a DDM, and then we will fit a GDDM.

First, we want to let the drift rate vary with the coherence. To do so, we must subclass *Drift*. Each subclass must contain a name (a short description of how drift varies), required parameters (a list of the parameters that must be passed when we initialize our subclass, i.e. parameters which are passed to the constructor), and required conditions (a list of conditions that must be present in any data when we fit data to the model). We can easily define a model that fits our needs:

```
import pyddm as ddm
class DriftCoherence(ddm.models.Drift):
    name = "Drift depends linearly on coherence"
    required_parameters = ["driftcoh"] # <-- Parameters we want to include in the␣
↪model
    required_conditions = ["coh"] # <-- Task parameters ("conditions"). Should be the␣
↪same name as in the sample.

    # We must always define the get_drift function, which is used to compute the␣
↪instantaneous value of drift.
    def get_drift(self, conditions, **kwargs):
        return self.driftcoh * conditions['coh']
```

Because we are fitting with likelihood, we must include a baseline lapse rate to avoid taking the log of 0. Traditionally this is implemented with a uniform distribution, but PyDDM can also use an exponential distribution using `OverlayPoissonMixture` (representing a Poisson process lapse rate), as we use here. However, since we also want a non-decision time, we need to use two Overlay objects. To accomplish this, we can use an *OverlayChain* object. Then, we can construct a model which uses this and fit the data to the model:

```
from pyddm import Model, Fittable
from pyddm.functions import fit_adjust_model, display_model
from pyddm.models import NoiseConstant, BoundConstant, OverlayChain,␣
↪OverlayNonDecision, OverlayPoissonMixture
model_rs = Model(name='Roitman data, drift varies with coherence',
                 drift=DriftCoherence(driftcoh=Fittable(minval=0, maxval=20)),
                 noise=NoiseConstant(noise=1),
                 bound=BoundConstant(B=Fittable(minval=.1, maxval=1.5)),
                 # Since we can only have one overlay, we use
                 # OverlayChain to string together multiple overlays.
                 # They are applied sequentially in order.  OverlayNonDecision
                 # implements a non-decision time by shifting the
                 # resulting distribution of response times by
                 # `nondectime` seconds.
                 ␣
↪overlay=OverlayChain(overlays=[OverlayNonDecision(nondectime=Fittable(minval=0,␣
↪maxval=.4)),
                                                  OverlayPoissonMixture(pmixturecoef=.
↪02,
                                                                        rate=1)]),
                 dx=.001, dt=.01, T_dur=2)

# Fitting this will also be fast because PyDDM can automatically
# determine that DriftCoherence will allow an analytical solution.
fit_model_rs = fit_adjust_model(sample=roitman_sample, model=model_rs, verbose=False)
```

Finally, we can display the fit parameters with the following command:

```
display_model(fit_model_rs)
```

This gives the following output (which may vary slightly, since the fitting algorithm is stochastic):

```
Model Roitman data, drift varies with coherence information:
Drift component DriftCoherence:
    Drift depends linearly on coherence
    Fitted parameters:
    - driftcoh: 10.388292
Noise component NoiseConstant:
    constant
    Fixed parameters:
    - noise: 1.000000
Bound component BoundConstant:
    constant
    Fitted parameters:
    - B: 0.744209
IC component ICPointSourceCenter:
    point_source_center
    (No parameters)
Overlay component OverlayChain:
    Overlay component OverlayNonDecision:
        Add a non-decision by shifting the histogram
        Fitted parameters:
        - nondectime: 0.312433
    Overlay component OverlayPoissonMixture:
        Poisson distribution mixture model (lapse rate)
        Fixed parameters:
        - pmixturecoef: 0.020000
```

(continues on next page)

(continued from previous page)

```
        - rate: 1.000000
Fit information:
    Loss function: Negative log likelihood
    Loss function value: 199.3406727870083
    Fitting method: differential_evolution
    Solver: auto
    Other properties:
        - nparams: 3
        - samplesize: 2611
        - mess: ''
```

Or, to access them within Python instead of printing them,

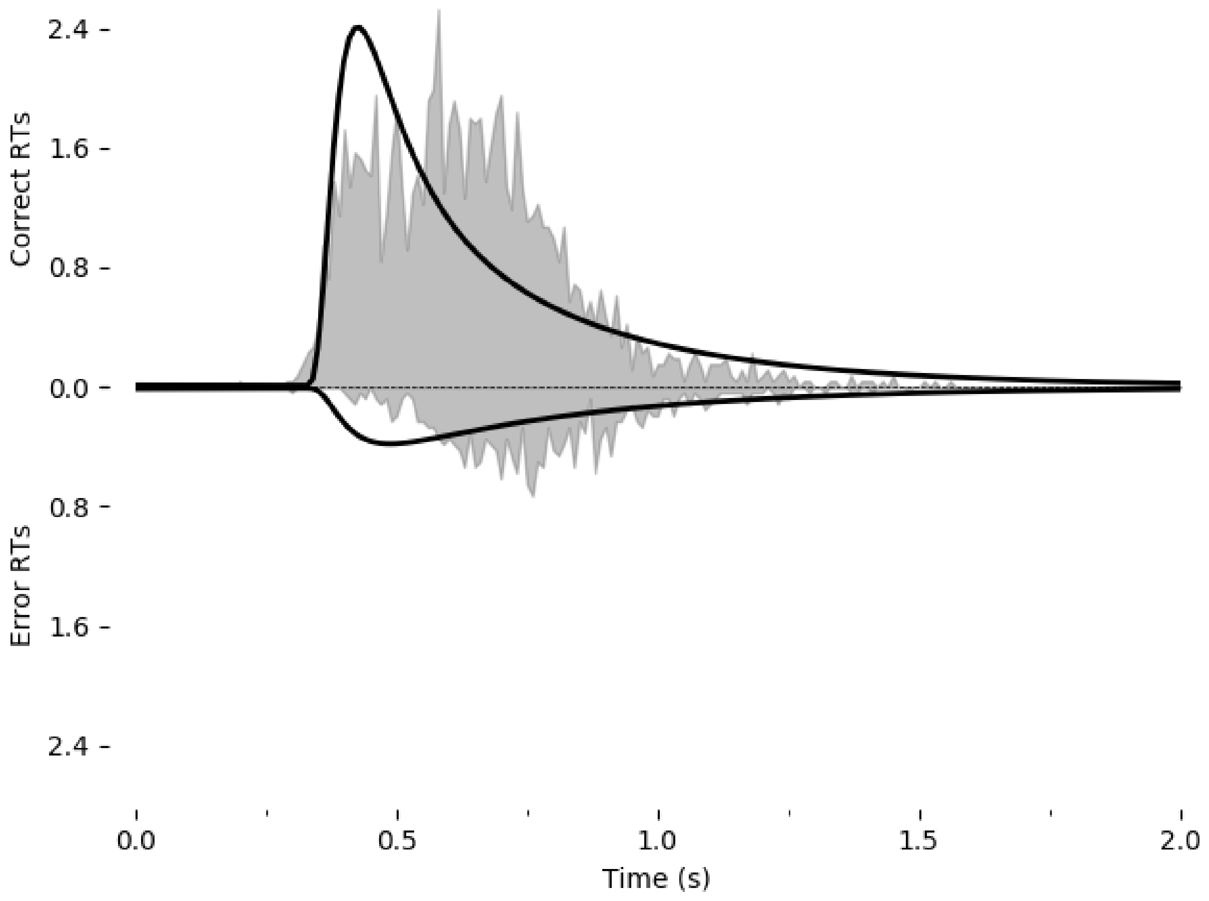
```
fit_model_rs.parameters()
```

Note that if you see "Warning: renormalizing model solution from X to 1." for some X, this is okay as long as X is close ($< 10^{-5}$ or so) to 1.0 or as long as this is seen early in the fitting procedure. If it is larger or seen towards the end of the fitting procedure, consider using smaller dx or dt in the simulation. This indicates numerical imprecision in the simulation.

2.3.4 Plotting the fit

We can also graphically evaluate the quality of the fit. We can plot and save a graph:

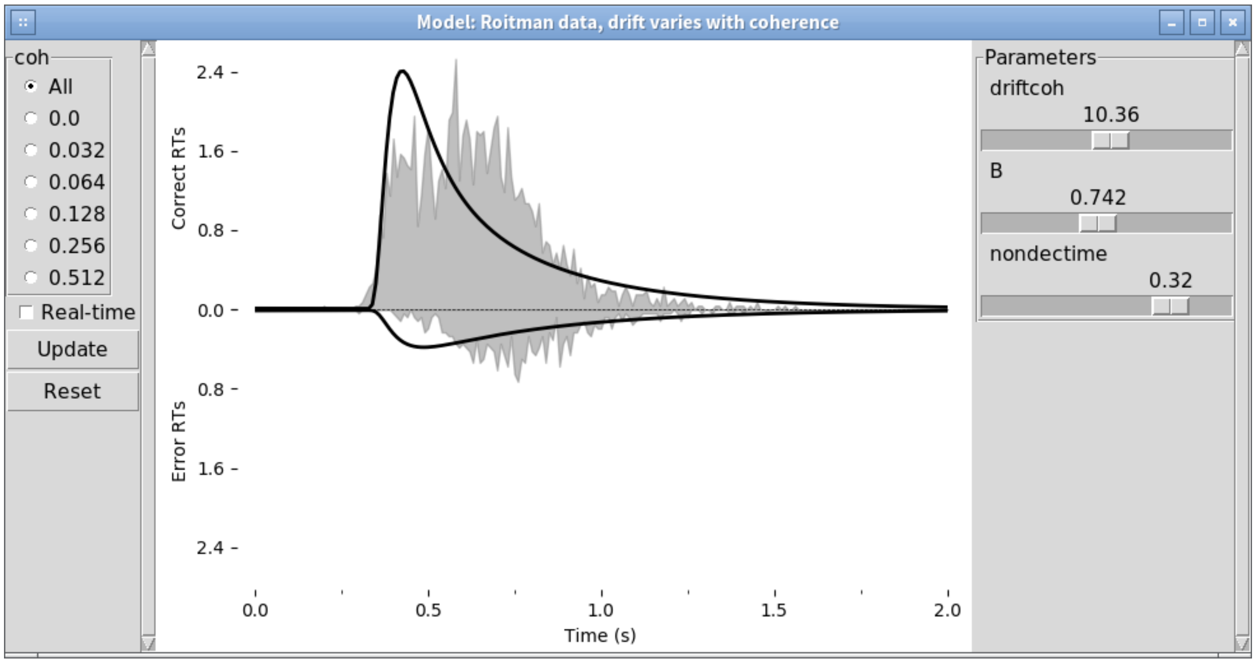
```
import pyddm.plot
import matplotlib.pyplot as plt
pyddm.plot.plot_fit_diagnostics(model=fit_model_rs, sample=roitman_sample)
plt.savefig("roitman-fit.png")
plt.show()
```



This model does not seem to fit the data very well.

We can alternatively explore this with the PyDDM's model GUI:

```
pyddm.plot.model_gui(model=fit_model_rs, sample=roitman_sample)
```



See *Model GUI* for more info.

`Download this full example`

2.4 Improving the fit

Let's see if we can improve the fit by including additional model components. We will include exponentially collapsing bounds and use a leaky or unstable integrator instead of a perfect integrator.

To use a coherence-dependent leaky or unstable integrator, we can build a drift model which incorporates the position of the decision variable to either increase or decrease drift rate. This can be accomplished by making `get_drift` depend on the argument `x`.

```
class DriftCoherenceLeak(ddm.models.Drift):
    name = "Leaky drift depends linearly on coherence"
    required_parameters = ["driftcoh", "leak"] # <-- Parameters we want to include in␣
↪the model
    required_conditions = ["coh"] # <-- Task parameters ("conditions"). Should be the␣
↪same name as in the sample.

    # We must always define the get_drift function, which is used to compute the␣
↪instantaneous value of drift.
    def get_drift(self, x, conditions, **kwargs):
        return self.driftcoh * conditions['coh'] + self.leak * x
```

Collapsing bounds are already included in PyDDM, and can be accessed with *BoundCollapsingExponential*.

Thus, the full model definition is

```
from pyddm.models import BoundCollapsingExponential
model_leak = Model(name='Roitman data, leaky drift varies with coherence',
                   drift=DriftCoherenceLeak(driftcoh=Fittable(minval=0, maxval=20),
                                            leak=Fittable(minval=-10, maxval=10)),
                   noise=NoiseConstant(noise=1),
```

(continues on next page)

(continued from previous page)

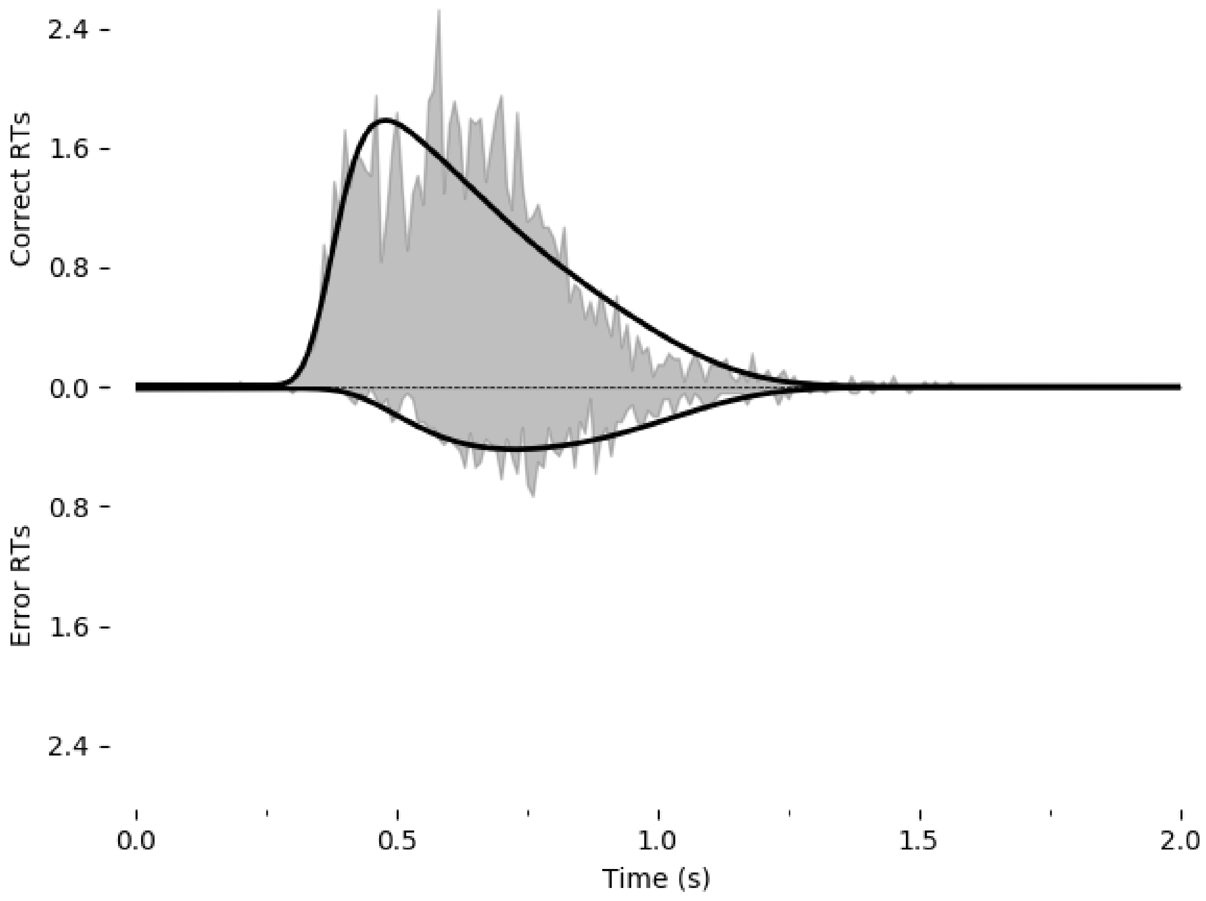
```
                    bound=BoundCollapsingExponential(B=Fittable(minval=0.5, maxval=3),
                                                     tau=Fittable(minval=.0001,␣
↪maxval=5)),
                    # Since we can only have one overlay, we use
                    # OverlayChain to string together multiple overlays.
                    # They are applied sequentially in order.  OverlayDelay
                    # implements a non-decision time by shifting the
                    # resulting distribution of response times by
                    # `delaytime` seconds.
                   ␣
↪overlay=OverlayChain(overlays=[OverlayNonDecision(nondectime=Fittable(minval=0,␣
↪maxval=.4)),
                                                   OverlayPoissonMixture(pmixturecoef=.
↪02,
                                                                         rate=1)]),
                    dx=.01, dt=.01, T_dur=2)
```

Before fitting this model, let's look at it in the model GUI:

```
from pyddm.plot import model_gui
model_gui(model_leak, sample=roitman_sample)
```

We can fit this and save it as an image using the following. Note that this may take a while (hours) due to the increased number of parameters and because the earlier examples were able to use the analytical solver but the present example must use backward Euler. For all coherences, the fit is:

```
fit_model_leak = fit_adjust_model(sample=roitman_sample, model=model_leak,␣
↪verbose=False)
pyddm.plot.plot_fit_diagnostics(model=fit_model_leak, sample=roitman_sample)
plt.savefig("leak-collapse-fit.png")
```



2.5 Going further

Just as we created DriftCoherence above (by inheriting from `Drift`) to modify the drift rate based on coherence, we can modify other portions of the model. See *PyDDM Cookbook* for more examples. Also see the *API Documentation* for more specific details about overloading classes.

2.6 Summary

PyDDM can simulate models and generate artificial data, or it can fit them to data. Below are high-level overviews for how to accomplish each.

To simulate models and generate artificial data:

1. Optionally, define unique components of your model. Models are modular, and allow specifying a dynamic drift rate, noise level, diffusion bounds, starting position of the integrator, or post-simulation modifications to the RT histogram. Many common models for these are included by default, but for advance functionality you may need to subclass `Drift`, `Noise`, `Bound`, `InitialCondition`, or `Overlay`. These model components may depend on "conditions", i.e. prespecified values associated with the behavioral task which change from trial to trial (e.g. stimulus coherence), or "parameters", i.e. values which apply to all trials and should be fit to the subject.

2. Define a model. Models are represented by creating an instance of the `Model` class, and specifying the model components to use for it. These model component can either be *the model components included in PyDDM* or ones you created in step 1. Values must be specified for all parameters required by model components.
3. Simulate the model using the `Model.solve()` method to generate a `Solution` object. If you have multiple conditions, you must run `Model.solve()` separately for each set of conditions and generate separate `Solution` objects.
4. Run the `Solution.resample()` method of the `Solution` object to generate a `Sample`. If you have multiple `Solution` objects (for multiple task conditions), you will need to generate multiple `Sample` objects as well. These can be added together with the "+" operator to form one single `Sample` object.

To fit a model to data:

1. Optionally define unique components of your model, as mentioned in Step 1 above.
2. Load your data into a `Sample` object using either `Sample.from_numpy_array()` or `Sample.from_pandas_dataframe()`. If you have multiple task conditions (i.e. prespecified values associated with the behavioral task which change from trial to trial), make sure that the names of the conditions in the `Sample` object align with those that are used in your model components. (In other words, if a model component expects a coherence value named "coh", make sure your sample includes a coherence value named "coh".)
3. Define a model. Models are represented by creating an instance of the `Model` class, and specifying the model components to use for it. These model component can either be *the model components included in PyDDM* or ones you created in step 1. Parameters for the model components must either be specified expicitly or else set to a `Fittable` instance, for example "Fittable(minval=0, maxval=1)".
3. Run `fit_adjust_model()` on the model and the sample. Optionally specify a `loss function` other than the default (which uses BIC). After fitting, the `Fittable` objects in the model will be changed to `Fitted` objects, which are just like `Fittable` objects except they contain the fitted values.
4. View the output by calling `display_model()` on the model. The value of the loss function is accessible via `Model.get_fit_result()` and the parameters via `Model.parameters()`.

CHAPTER 3

PyDDM Cookbook

Here are a list of examples of common model features and how to implement them in PyDDM. **If you created an example or model in PyDDM and would like it to be added to the cookbook, please** send it to us **so we can add it**. Include the author(s) of the example or model and optionally a literature reference so that we can give you proper credit and direct users to your paper!

`Download cookbook.py (all models in the cookbook)`

3.1 Drift and noise

I want:

- *A drift rate which changes over time (see "task paradigms" section above)*
- *Leaky or unstable integration*
- `A general Ornstein-Uhlenbeck process`
- *A drift rate which depends linearly on a task parameter (e.g. coherence)*
- *A biased drift dependent on a task condition (e.g. reward or choice history)*
- *A leaky integrator with a drift rate which depends linearly on a task parameter (e.g. coherence)*
- *An urgency gain function*
- *Drift rate variability (uniform distribution)*
- *Drift rate or noise which depends on a moment-to-moment signal, unique to each trial*
- Something else *(Write your own, using these as a guide.)*

3.2 Collapsing bounds (or time-varying bounds)

I instead want bounds which:

- `Are constant over time`
- `Collapse linearly`
- `Collapse exponentially`
- *Collapse exponentially after a delay*
- *Collapse according to a Weibull CDF*
- *Collapse according to a step function*
- *Vary based on task conditions, e.g. for a speed vs accuracy task*
- *Increase over time*
- Something else *(Write your own, using these as a guide)*

3.3 Initial conditions

I don't want my initial conditions to be `A single point positioned in the middle of the bounds` (the default). Instead, I want my initial conditions to be:

- A single point
  - `A single point at an arbitrary location`
  - *A side bias*
  - *A relative side bias (as a ratio of bound height)*
- A uniform distribution
  - `A uniform distribution across all potential starting positions`
  - `A uniform distribution of arbitrary length centered in the middle of the bounds`
  - *A uniform distribution of arbitrary length with a center at an arbitrary location wtih sign determined by task conditions*
- `A Gaussian distribution centered in the middle of the bounds`
- *A Cauchy distribution*
- `A specific distribution which does not change based on task parameters`
- Something else *(Write your own, using these as a guide)*

3.4 Non-decision time

I want to use a non-decision time which is:

- `Fixed at a single value`
- `A uniform distribution`
- `A gamma distribution`
- *A Gaussian distribution*
- *A different value for left and right choices*
- Something else *(Write your own, using these as a guide)*

3.5 Mixture models (Contaminant RTs)

I want to fit a distribution which has contaminant RTs distributed according to:

- `A uniform distribution`
- `An exponential distribution (corresponding to a Poisson process)`
- Something else (Write your own)

3.6 Task paradigms

Here are some examples of potential task paradigms that can be simulated with PyDDM.

- *A pulse paradigm*
- *Evidence oscillating in a sine wave*
- *A psychophysical kernel paradigm*
- Something else *(Write your own, using these as a guide.)*

3.7 Objective functions

I don't want to use the default recommended objective function (`negative log likelihood`) but would rather use:

- `Squared error`
- `BIC`
- *Mean RT and P(correct)*
- *Something which takes undecided trials into account*
- Something else *(Write your own, using these as a guide)*

(Note that changing the objective function to something other than likelihood will not speed up model fitting.)

3.8 Fitting methods

I don't want to fit using the default recommended method (differential evolution), but would rather fit using:

- *Nelder-Mead*
- *Basin hopping*
- *Gradient descent*

(While using a fitting method other than differential evolution will likely reduce the time needed for fitting models, other methods may not offer robust parameter estimation for high-dimensional models.)

3.9 Models from specific papers

If you have a paper which used PyDDM, please send us your model *so we can include them here!*

- papers/shinn2020
- papers/shinn2020b
- papers/degee2020
- papers/dip

3.10 Other recipes

I want to:

- *Fit a model to data*
- *Share parameters between two different models*
- *Use multiple "Overlay" objects in the same model*
- *Run models in parallel using multiple CPUs*
- *Retrieve the evolving pdf of a solution*

3.10.1 How-to guides

Shared parameters

In order to use the same parameter for multiple different components of the model, pass the same *Fittable* instance to both. As a concrete example, suppose we want both the drift rate and the standard deviation to increase by some factor `boost` at time `tboost`. We could make *Drift* and *Noise* objects as follows:

```
from pyddm.models import Drift, Noise
class DriftBoost(Drift):
    name = "Drift with a time-delayed boost"
    required_parameters = ["driftbase", "driftboost", "tboost"]
    required_conditions = []
    def get_drift(self, t, conditions, **kwargs):
        if t < self.tboost:
            return self.driftbase
        elif t >= self.tboost:
            return self.driftbase * self.driftboost

class NoiseBoost(Noise):
    name = "Noise with a time-delayed boost"
    required_parameters = ["noisebase", "noiseboost", "tboost"]
    required_conditions = []
    def get_noise(self, t, conditions, **kwargs):
        if t < self.tboost:
            return self.noisebase
        elif t >= self.tboost:
            return self.noisebase * self.noiseboost
```

Now, we can define a model to fit with:

```
from pyddm import Model, Fittable
t_boost = Fittable(minval=0, maxval=1)
boost = Fittable(minval=1, maxval=3)
m = Model(drift=DriftBoost(driftbase=Fittable(minval=.1, maxval=3),
                           driftboost=boost,
                           tboost=t_boost),
          noise=NoiseBoost(noisebase=Fittable(minval=.2, maxval=1.5),
                           noiseboost=boost,
                           tboost=t_boost),
          T_dur=3, dt=.001, dx=.001)
```

This will ensure that the value of `driftboost` is always equal to the value of `noiseboost`, and that the value of `tboost` in Drift is always equal to the value of `tboost` in Noise.

Note that this is **not the same** as:

```
m = Model(drift=DriftBoost(driftbase=Fittable(minval=.1, maxval=3),
                           driftboost=Fittable(minval=1, maxval=3),
                           tboost=Fittable(minval=0, maxval=1)),
          noise=NoiseBoost(noisebase=Fittable(minval=.2, maxval=1.5),
                           noiseboost=Fittable(minval=1, maxval=3),
                           tboost=Fittable(minval=0, maxval=1)),
          T_dur=3, dt=.001, dx=.001)
```

In the latter case, `driftboost` and `noiseboost` will be fit to different values, and the two `tboost` parameters will not be equal.

Parallelization

PyDDM has built-in support for parallelization if pathos is installed.

To use parallelization, first set up the parallel pool:

```
from pyddm import set_N_cpus
set_N_cpus(4)
```

Then, PyDDM will automatically parallelize the fitting routines. For example, just call:

```
fit_model_rs = fit_adjust_model(sample=roitman_sample, model=model_rs)
```

There are a few caveats with parallelization:

1. It is only possible to run fits in parallel if they are on the same computer. It is not possible to fit across multiple nodes in a cluster, for example.
2. Due to a bug in pathos, all model components must be **defined in a separate file** and then imported.
3. Only models with many conditions will be sped up by parallelization. The cardinality of the cartesian product of the conditions is the maximum number of CPUs that will have an effect: for example, if you have four coherence conditions, a right vs left condition, and a high vs low reward condition, then after $4 \times 2 \times 2 = 16$ CPUs, there will be no benefit to increasing the number of CPUs.
4. It is possible but not recommended to set the number of CPUs to be greater than the number of physical CPU cores on the machine. This will cause a slight reduction in performance.

Fitting models with custom algorithms

As described in *fit_adjust_model()*, three different algorithms can be used to fit models. The default is differential evolution, which we have observed to be robust for models with large numbers of parameters.

Other methods can be used by passing the "fitting_method" argument to *fit_adjust_model()* or *fit_model()*. This can take one of several values:

- "simplex": Use the Nelder-Mead simplex method
- "simple": Gradient descent
- "basin": Use Scipy's basin hopping algorithm.
- A function can be passed to use this function as a custom objective function.

For example, to fit the model in the quickstart using the Nelder-Mead simplex method, you can do:

```
fit_model_rs = fit_adjust_model(sample=roitman_sample, model=model_rs, fitting_method=
↪"simplex")
```

Retrieve the evolving pdf of a solution

Setting return_evolution=True in solve_numerical() will (with methods "implicit" and "explicit" only) return an M-by-N array (as part of the Solution) whose columns contain the cross-sectional pdf for every time step:

```
sol = model.solve_numerical_implicit(conditions=conditions, return_evolution=True)
sol.pdf_evolution()
```

This is equivalent to (but much faster than):

```
sol = np.zeros((len(model.x_domain(conditions)), len(model.t_domain())))
sol[:,0] = model.IC(conditions=conditions)/model.dx
for t_ind, t in enumerate(model.t_domain()[1:]):
    T_dur_backup = model.T_dur
    model.T_dur = t
    ans = model.solve_numerical_implicit(conditions=conditions, return_
↪evolution=False)
    model.T_dur = T_dur_backup
    sol[:,t_ind+1] = ans.pdf_undec()
```

Note that:

```
sum(pdf("correct")[0:t]*dt) + sum(pdf("error")[0:t]*dt) + sum(pdf_evolution()[:,
↪t]*dx) = 1
```

Stimulus coding vs accuracy coding vs anything else coding

By default, the upper boundary in PyDDM represents "correct" choices and the lower boundary represents "error" choices. However, these boundaries can represent whatever you would like: "left" vs "right" choice, "high value" vs "low value" choice, choice "inside the receptive field" vs "outside the receptive field", etc.

To change the name of the choices represented by the bounds, we need to pass the name of the two boundaries as an argument to the Model and the Sample. For example, if "High value" is represented by the upper boundary and "Low value" by the lower boundary, we can write:

```
model = Model(..., choice_names=("High value", "Low value"))
sample = Sample.from_pandas_dataframe(..., choice_names=("High value", "Low value"))
```

Then, these names can be used to access properties of sample or solution from the solved model, just as we did for "correct" and "error" in the *tutorial*. For example:

```
sample.prob("High value")
sol = model.solve()
sol.prob("High value")
sol.pdf("Low value")
```

However, note that the data must be in the appropriate format. You are responsible for formatting the data correctly for these interpretations to hold. For example, consider the *Roitman-Shadlen dataset from the tutorial*. The dataset looks as follows:

monkey	rt	coh	correct	trgchoice
1	0.355	0.512	1.0	2.0
1	0.359	0.256	1.0	1.0
1	0.525	0.128	1.0	1.0

In this format, "coh" is the motion coherence, "correct" is whether the monkey chose the target in the direction of the random dot motion, and "trgchoice" is whether the monkey chose target 1 (inside the receptive field) or target 2 (outside the receptive field). In this experiment, the targets were chosen to be in different places for each session, so they did not map directly onto a location on the screen.

Let's make the top boundary represent "target 1" and the bottom represent "target 2". We already have the variable "trgchoice", describing whether the monkey chose "target 1" or "target 2". So we can use this as the "choice" variable (for which we previously used "correct" or "error"). PyDDM assumes that the upper boundary choice is given by a "1" and the lower boundary choice by "0", so all we need to correct the trgchoice variable such that responses to target 2 are coded as "0" instead of "2".

But, the "coh" column measures coherence with respect to the correct choice, not with respect to one of the targets. Since we are defining our upper boundary choice as "target 1" and our lower boundary choice as "target 2", positive coherence should represent the case where the stimulus showed motion in the direction of "target 1" and negative coherence in the direction of "target 2". In the dataset, "coh" is always positive. So, we need to make "coh" negative if the motion was coherent towards "target 2". This happened when the monkey was correct and chose target 2 (`correct == 1.0 and trgchoice == 2.0`) or when the monkey was incorrect and chose target 1 (`correct == 0.0 and trgchoice == 1.0`).

So, after performing these transformations, our dataset looks like the following:

monkey	rt	coh	correct	choice
1	0.355	-0.512	1.0	0.0
1	0.359	0.256	1.0	1.0
1	0.525	0.128	1.0	1.0

Loading the data therefore looks like:

```
# Read the dataset into a Pandas DataFrame
from pyddm import Sample
import pandas
with open("roitman_rts.csv", "r") as f:
    df_rt = pandas.read_csv(f)
```

(continues on next page)

(continued from previous page)

```
df_rt = df_rt[df_rt["monkey"] == 1] # Only monkey 1

# Remove short and long RTs, as in 10.1523/JNEUROSCI.4684-04.2005.
# This is not strictly necessary, but is performed here for
# compatibility with this study.
df_rt = df_rt[df_rt["rt"] > .1] # Remove trials less than 100ms
df_rt = df_rt[df_rt["rt"] < 1.65] # Remove trials greater than 1650ms

# Adjust the dataset for stimulus coding
df_rt['choice'] = df_rt['trgchoice'] % 2
df_rt['coh'] = df_rt['coh'] * ((df_rt["choice"] == df_rt["correct"])*2-1)

# Create a sample object from our data.  This is the standard input
# format for fitting procedures.  Since RT and correct/error are
# both mandatory columns, their names are specified by command line
# arguments.
roitman_sample = Sample.from_pandas_dataframe(df_rt, rt_column_name="rt", choice_
↪column_name="choice", choice_names=("target 1", "target 2"))
```

And defining and fitting the model looks like:

```
# Define a model which uses our new DriftCoherence defined above.
from pyddm import Model, Fittable
from pyddm.functions import fit_adjust_model, display_model
from pyddm.models import NoiseConstant, BoundConstant, OverlayChain,␣
↪OverlayNonDecision, OverlayPoissonMixture
model_rs = Model(name='Roitman data, drift varies with coherence',
                 drift=DriftCoherence(driftcoh=Fittable(minval=-20, maxval=20)),
                 noise=NoiseConstant(noise=1),
                 bound=BoundConstant(B=Fittable(minval=.1, maxval=1.5)),
                 # Since we can only have one overlay, we use
                 # OverlayChain to string together multiple overlays.
                 # They are applied sequentially in order.  OverlayNonDecision
                 # implements a non-decision time by shifting the
                 # resulting distribution of response times by
                 # `nondectime` seconds.
                 ␣
↪overlay=OverlayChain(overlays=[OverlayNonDecision(nondectime=Fittable(minval=0,␣
↪maxval=.4)),
                                                     OverlayPoissonMixture(pmixturecoef=.
↪02,
                                                                           rate=1)]),
                 choice_names=("target 1", "target 2"),
                 dx=.001, dt=.01, T_dur=2)

# Fitting this will also be fast because PyDDM can automatically
# determine that DriftCoherence will allow an analytical solution.
fit_model_rs = fit_adjust_model(sample=roitman_sample, model=model_rs, verbose=False)
display_model(fit_model_rs)
fit_model_rs.parameters()
```

As we see, we recover approximately the same parameters:

```
Model Roitman data, drift varies with coherence information:
Choices: 'target 1' (upper boundary), 'target 2' (lower boundary)
Drift component DriftCoherence:
    Drift depends linearly on coherence
```

(continues on next page)

(continued from previous page)

```
    Fitted parameters:
    - driftcoh: 10.362975
Noise component NoiseConstant:
    constant
    Fixed parameters:
    - noise: 1.000000
Bound component BoundConstant:
    constant
    Fitted parameters:
    - B: 0.744039
IC component ICPointSourceCenter:
    point_source_center
    (No parameters)
Overlay component OverlayChain:
    Overlay component OverlayNonDecision:
        Add a non-decision by shifting the histogram
        Fitted parameters:
        - nondectime: 0.310893
    Overlay component OverlayPoissonMixture:
        Poisson distribution mixture model (lapse rate)
        Fixed parameters:
        - pmixturecoef: 0.020000
        - rate: 1.000000
Fit information:
    Loss function: Negative log likelihood
    Loss function value: 199.33386049405675
    Fitting method: differential_evolution
    Solver: auto
    Other properties:
        - nparams: 3
        - samplesize: 2611
        - mess: ''
```

When displaying in the model GUI, as desired, the two distributions represent “target 1” and “target 2” instead of “correct” and “error”.

```
pyddm.plot.model_gui(model=fit_model_rs, sample=roitman_sample)
```

This coding scheme may impact the interpretation of the other parameters in the model, so be careful! For example, *starting point biases require special considerations*.

3.10.2 Task paradigms

Pulse paradigm

The pulse paradigm, where evidence is presented for a fixed amount of time only, is common in behavioral neuroscience. For simplicity, let us first model it without coherence dependence:

```
from pyddm.models import Drift
class DriftPulse(Drift):
    name = "Drift for a pulse paradigm"
    required_parameters = ["start", "duration", "drift"]
    required_conditions = []
    def get_drift(self, t, conditions, **kwargs):
        if self.start <= t <= self.start + self.duration:
```

(continues on next page)

(continued from previous page)

```
            return self.drift
        return 0
```

Here, `drift` is the strength of the evidence integration during the pulse, `start` is the time of the pulse onset, and `duration` is the duration of the pulse.

Try it out with:

```
from pyddm import Model, Fittable
from pyddm.plot import model_gui
model = Model(drift=DriftPulse(start=Fittable(minval=0, maxval=1.5),
                               duration=Fittable(minval=0, maxval=.5),
                               drift=Fittable(minval=0, maxval=2)),
              dx=.01, dt=.01)
model_gui(model)
```

This can easily be modified to make it coherence dependent, where `coherence` is a condition in the *`Sample`*:

```
from pyddm.models import Drift
class DriftPulseCoh(Drift):
    name = "Drift for a coherence-dependent pulse paradigm"
    required_parameters = ["start", "duration", "drift"]
    required_conditions = ["coherence"]
    def get_drift(self, t, conditions, **kwargs):
        if self.start <= t <= self.start + self.duration:
            return self.drift * conditions["coherence"]
        return 0
```

Try it out with:

```
from pyddm import Model, Fittable
from pyddm.plot import model_gui
model = Model(drift=DriftPulseCoh(start=Fittable(minval=0, maxval=1.5),
                                  duration=Fittable(minval=0, maxval=.5),
                                  drift=Fittable(minval=0, maxval=2)),
              dx=.01, dt=.01)
model_gui(model, conditions={"coherence": [0, .3, .6]})
```

Alternatively, drift can be set at a default value independent of coherence, and changed during the pulse duration. In this case, there is some fixed amount of evidence, with a small burst of additionall evidence:

```
from pyddm.models import Drift
class DriftPulse2(Drift):
    name = "Drift for a pulse paradigm, with baseline drift"
    required_parameters = ["start", "duration", "drift", "drift0"]
    required_conditions = []
    def get_drift(self, t, conditions, **kwargs):
        if self.start <= t <= self.start + self.duration:
            return self.drift
        return self.drift0
```

Try it out with:

```
from pyddm import Model, Fittable
from pyddm.plot import model_gui
model = Model(drift=DriftPulse2(drift0=Fittable(minval=0, maxval=.5),
                                start=Fittable(minval=0, maxval=1.5),
```

(continues on next page)

(continued from previous page)

```
                                    duration=Fittable(minval=0, maxval=.5),
                                    drift=Fittable(minval=0, maxval=2)),
              dx=.01, dt=.01)
model_gui(model)
```

Psychophysical Kernel paradigm

In the psychophysical kernel paradigm, random time-varying but on average unbiased stimuli are presented on a trial-by-trial basis to quantify the weight a given time point has on behavioural choice.

In particular, consider a sequence of coherences `coh_t_list`, generated by randomly sampling from a pool of coherences `coh_list_PK` for `Tdur = 2` seconds every `dt_PK = 0.05` seconds:

```
coh_list = np.array([-25.6, -12.8, -6.4, 6.4, 12.8, 25.6])
Tdur = 2
dt_PK=0.05
i_coh_t_list = np.random.randint(len(coh_list), size=int(Tdur/dt_PK))
coh_t_list = [0.01*coh_list[i] for i in i_coh_t_list]
```

If the conversion from coherence to "drift" is known (e.g. by fitting other tasks), one can model the DDM with this sequence of evidence:

```
from pyddm.models import Drift
class DriftPK(Drift):
    name = "PK drifts"
    required_conditions = ["coh_t_list", "dt_PK"]
    required_parameters = ["drift"]
    def get_drift(self, t, conditions, **kwargs):
        return self.drift**0.01*conditions["coh_t_list"][int(t/conditions["dt_PK"])]
```

Running the same process over multiple trials, we can use reverse correlation to obtain the impact of stimuli at each time-step on the final choice. (Note: the following step is slow, as sufficiently many trials is needed to ensure each stimulus strength at each time-step is considered):

```
import numpy as np
from pyddm import Model
from pyddm.models import NoiseConstant, BoundConstant, OverlayChain,␣
↪OverlayNonDecision, OverlayPoissonMixture
from pyddm.functions import display_model
n_rep=1000
coh_list = np.array([-25.6, -12.8, -6.4, 6.4, 12.8, 25.6])
Tdur = 2
dt_PK=0.05
PK_Mat = np.zeros((int(Tdur/dt_PK), len(coh_list)))
PK_n   = np.zeros((int(Tdur/dt_PK), len(coh_list)))
for i_rep in range(n_rep):
    i_coh_t_list = np.random.randint(len(coh_list), size=int(Tdur/dt_PK))
    coh_t_list = [0.01*coh_list[i] for i in i_coh_t_list]
    model = Model(name='PK',
        drift=DriftPK(drift=2.2),
        noise=NoiseConstant(noise=1.5),
        bound=BoundConstant(B=1.1),
        overlay=OverlayNonDecision(nondectime=.1),
        dx=.001, dt=.01, T_dur=2)
    sol = model.solve(conditions={"coh_t_list": coh_t_list, "dt_PK": dt_PK})
```

(continues on next page)

(continued from previous page)

```
    for i_t in range(int(Tdur/dt_PK)):
        PK_Mat[i_t, i_coh_t_list[i_t]] += sol.prob("correct") - sol.prob("error")
        PK_n[i_t, i_coh_t_list[i_t]] += 1
PK_Mat = PK_Mat/PK_n
```

Where `n_rep` is the number trials. `PK_Mat` is known as the psychophysical matrix. Normalizing by coherence and averaging across stimuli (for each time-step), one obtains the psychophysical kernel `PK`:

```
for i_coh in range(len(coh_list)):
    PK_Mat[:,i_coh] /= coh_list[i_coh]
PK = np.mean(PK_Mat, axis=1)
```

3.10.3 Drift and Noise Models

Leaky/Unstable integrator

Leaky/unstable integrators are implemented in *`DriftLinear`*. For a leaky integrator, set the parameter `x` to be less than 0. For an unstable integrator, set the parameter `x` to be greater than 0. For example:

```
from pyddm import Model
from pyddm.models import DriftLinear
model = Model(drift=DriftLinear(drift=0, t=.2, x=.1))
```

Try it out with:

```
from pyddm import Model, Fittable, DriftLinear
from pyddm.plot import model_gui
model = Model(drift=DriftLinear(drift=Fittable(minval=0, maxval=2),
                                t=Fittable(minval=0, maxval=2),
                                x=Fittable(minval=-1, maxval=1)),
              dx=.01, dt=.01)
model_gui(model, conditions={"frequency": [0, 4, 8]})
```

Sine wave evidence

We use evidence in the form of a sine wave as an example of how to construct a new model class.

Suppose we have a task where evidence varies according to a sine wave which has a different frequency on different trials. The frequency is a feature of the task, and will be the same for all components of the model. Thus, it is a "condition". By contrast, how strongly the animal weights the evidence is not observable and only exists internal to the model. It is a "parameter", or something that we must fit to the data. This model can then be defined as:

```
import numpy as np
from pyddm.models import Drift
class DriftSine(Drift):
    name = "Sine-wave drifts"
    required_conditions = ["frequency"]
    required_parameters = ["scale"]
    def get_drift(self, t, conditions, **kwargs):
        return np.sin(t*conditions["frequency"]*2*np.pi)*self.scale
```

In this case, `frequency` is externally provided per trial, thus defined as a condition. By contrast, `scale` is a parameter to fit, and is thus defined as a parameter. We then use the DriftSine class to define model:

```
from pyddm import Model
model = Model(name='Sine-wave evidences',
                drift=DriftSine(scale=0.5))
sol = model.solve(conditions={"frequency": 5})
```

The model is solved and the result is saved in the variable "sol", where the `probability of each choice` and other outputs could be retrieved. Finally, note that the conditions, being externally defined (e.g. trial-by-trial), must be input during the call to model.solve. The parameters, such as offset, are defined within the respective classes. Depending on the context, it could be either a constant (as done here) or as a `Fittable` object, if fitting to data is required.

Try it out with:

```
from pyddm import Model, Fittable
from pyddm.plot import model_gui
model = Model(drift=DriftSine(scale=Fittable(minval=0, maxval=2)),
              dx=.01, dt=.01)
model_gui(model, conditions={"frequency": [0, 4, 8]})
```

Coherence-dependent drift rate

```
from pyddm import Drift
class DriftCoherence(Drift):
    name = "Drift depends linearly on coherence"
    required_parameters = ["driftcoh"] # <-- Parameters we want to include in the␣
↪model
    required_conditions = ["coh"] # <-- Task parameters ("conditions"). Should be the␣
↪same name as in the sample.

    # We must always define the get_drift function, which is used to compute the␣
↪instantaneous value of drift.
    def get_drift(self, conditions, **kwargs):
        return self.driftcoh * conditions['coh']
```

Try it out with:

```
from pyddm import Model, Fittable
from pyddm.plot import model_gui
model = Model(drift=DriftCoherence(driftcoh=Fittable(minval=0, maxval=1)),
              dx=.01, dt=.01)
model_gui(model, conditions={"coh": [0, .25, .5]})
```

Biased coherence-dependent drift rate

To model tasks in which some trials are expected to have a biased drift rate (e.g. tasks with a reward bias or strong trial history effects), we may want the drift rate to be biased in a certain direction (called "me" in some models) depending on task parameters. In the case of reward bias, whereby high reward trials have a higher drift rate, we need two task parameters: one describing the coherence of the trial ("coh") and one indicator of whether the trial was high reward (1) or low reward (0) ("highreward"). Then we have to parameters: the amount by which coherence impacts the drift rate ("driftcoh") and the amount of reward bias ("rewbias").

```
from pyddm import Drift
class DriftCoherenceRewBias(Drift):
```

(continues on next page)

(continued from previous page)

```
    name = "Drift depends linearly on coherence, with a reward bias"
    required_parameters = ["driftcoh", "rewbias"] # <-- Parameters we want to include␣
↪in the model
    required_conditions = ["coh", "highreward"] # <-- Task parameters ("conditions").␣
↪Should be the same name as in the sample.

    # We must always define the get_drift function, which is used to compute the␣
↪instantaneous value of drift.
    def get_drift(self, conditions, **kwargs):
        rew_bias = self.rewbias * (1 if conditions['highreward'] == 1 else -1)
        return self.driftcoh * conditions['coh'] + rew_bias
```

Try it out with:

```
from pyddm import Model, Fittable
from pyddm.plot import model_gui
model = Model(drift=DriftCoherenceRewBias(
                        driftcoh=Fittable(minval=0, maxval=1),
                        rewbias=Fittable(minval=0, maxval=1)),
              dx=.01, dt=.01)
model_gui(model, conditions={"coh": [0, .25, .5], "highreward": [0, 1]})
```

Coherence-dependent drift rate with leak

```
from pyddm import Drift
class DriftCoherenceLeak(Drift):
    name = "Leaky drift depends linearly on coherence"
    required_parameters = ["driftcoh", "leak"] # <-- Parameters we want to include in␣
↪the model
    required_conditions = ["coh"] # <-- Task parameters ("conditions"). Should be the␣
↪same name as in the sample.

    # We must always define the get_drift function, which is used to compute the␣
↪instantaneous value of drift.
    def get_drift(self, x, conditions, **kwargs):
        return self.driftcoh * conditions['coh'] + self.leak * x
```

Try it out with:

```
from pyddm import Model, Fittable
from pyddm.plot import model_gui
model = Model(drift=DriftCoherenceLeak(driftcoh=Fittable(minval=0, maxval=1),
                                       leak=Fittable(minval=-1, maxval=1)),
              dx=.01, dt=.01)
model_gui(model, conditions={"coh": [0, .25, .5]})
```

Urgency gain function

To implement a gain function, we want to scale both the drift rate and the noise by some function. First, let's define such a function to be a simple linear ramp.

```
def urgency_gain(t, gain_start, gain_slope):
    return gain_start + t*gain_slope
```

Now we can define a Drift model component which uses this.

```
from pyddm import Drift
class DriftUrgencyGain(Drift):
    name = "drift rate with an urgency function"
    required_parameters = ["snr", "gain_start", "gain_slope"]
    def get_drift(self, t, **kwargs):
        return self.snr * urgency_gain(t, self.gain_start, self.gain_slope)
```

We also need to define a Noise model component which uses it. This allows us to keep the signal-to-noise ratio (SNR) fixed.

```
from pyddm import Noise
class NoiseUrgencyGain(Noise):
    name = "noise level with an urgency function"
    required_parameters = ["gain_start", "gain_slope"]
    def get_noise(self, t, **kwargs):
        return urgency_gain(t, self.gain_start, self.gain_slope)
```

Finally, here is an example of how we would use this. Note that it utilizes *shared parameters*:

```
from pyddm import Model, Fittable
from pyddm.plot import model_gui
gain_start = Fittable(minval=0, maxval=1)
gain_slope = Fittable(minval=0, maxval=2)
m = Model(drift=DriftUrgencyGain(snr=Fittable(minval=0, maxval=2),
                                 gain_start=gain_start,
                                 gain_slope=gain_slope),
          noise=NoiseUrgencyGain(gain_start=gain_start,
                                 gain_slope=gain_slope),
          dt=.01, dx=.01)
model_gui(model=m)
```

Across-trial variability in drift rate

While across-trial variability in drift rate is possible in PyDDM, it is not efficient or ergonomic. Unlike variability in starting position or non-decision time, the distribution of drift rates must be discretized, and each must be run separately. Here, we demonstrate how to do this for a uniform distribution.

In order to run such a model, first you must prepare your Sample by running it through the following function. This makes one duplicate of each of your data points according to each discretization point. As a result, note that that likelihood, BIC, and other summary statistics about the data or fit quality may be inaccurate.

```
import pyddm as ddm
import numpy as np
RESOLUTION = 11
def prepare_sample_for_variable_drift(sample, resolution=RESOLUTION):
    new_samples = []
    for i in range(0, resolution):
        choice_upper = sample.choice_upper.copy()
        choice_lower = sample.choice_lower.copy()
        undecided = sample.undecided
        conditions = copy.deepcopy(sample.conditions)
        conditions['driftnum'] = (np.asarray([i]*len(choice_upper)),
                                  np.asarray([i]*len(choice_lower)),
                                  np.asarray([i]*undecided))
```

(continues on next page)

(continued from previous page)

```
        new_samples.append(ddm.Sample(choice_upper, choice_lower, undecided, choice_
↪names=samplue.choice_names, **conditions))
    new_sample = new_samples.pop()
    for s in new_samples:
        new_sample += s
    return new_sample
```

After using the above function to generate a new sample, use a class such as the following, which provides a uniformly-distributed drift rate.

```
class DriftUniform(ddm.Drift):
    """Drift with trial-to-trial variability.

    Note that this is a numerical approximation to trial-to-trial
    drift rate variability and is inefficient.  It also requires
    running the "prepare_sample_for_variable_drift" function above on
    the data.
    """
    name = "Uniformly-distributed drift"
    resolution = RESOLUTION # Number of bins, should be an odd number
    required_parameters = ['drift', 'width'] # Mean drift and the width of the␣
↪uniform distribution
    required_conditions = ['driftnum']
    def get_drift(self, conditions, **kwargs):
        stepsize = self.width/(self.resolution-1)
        mindrift = self.drift - self.width/2
        return mindrift + stepsize*conditions['driftnum']
```

This can be used like any other Drift class. For example:

```
from pyddm import Model, Fittable
from pyddm.plot import model_gui
model = Model(drift=DriftUniform(drift=Fittable(minval=0, maxval=2),
                                 width=Fittable(minval=0, maxval=2)),
              dx=.01, dt=.01)
model_gui(model, conditions={"driftnum": list(range(0, 11))})
```

Moment-to-moment observations

Suppose we have measured a parameter at each moment in each trial and would like to compute a drift rate based upon it. Let's suppose we measured the subject's skin conductance response (SCR) while performing the task, and we hypothesize that evidence integration is stronger when there is a higher SCR. We can bin SCR into bins (say, 100 ms bins) and use this to determine drift rate. This means that there will only be one trial per condition, and that each value of the condition will be a tuple of SCR values. Note that it must be a tuple, and cannot be a list.

Let's save the SCR into the "signal" condition in the code below. We can write:

```
class DriftMomentToMoment(ddm.Drift):
    """Drift rate which depends on trial-wise observations over time"""
    name = "Moment-to-moment drift"
    BINSIZE = .1 # 100 ms per bin
    required_parameters = ['drift_multiplier'] # How much to scale moment-to-moment␣
↪drift
    required_conditions = ['signal'] # should be a list of values which determine the␣
↪moment-to-moment drift
```

(continues on next page)

(continued from previous page)

```
    def get_drift(self, t, conditions, **kwargs):
        bin_number = int(t//self.BINSIZE) # Which bin are we currently in?
        n_bins = len(conditions['signal']) # Total number of bins for this condition
        # If we are currently in a bin which exceeds the total bins, fix to the last␣
↪bin
        if bin_number >= n_bins:
            bin_number = n_bins-1
        # Compute the moment-to-moment drift
        return conditions['signal'][bin_number] * self.drift_multiplier
```

This can be used in a model for any sample which has the “signal” condition, where “signal” is a tuple of values, giving the magnitude of the SCR at each timepoint (spaced BINSIZE apart).

Alternatively, we could extend this beyond just drift rate. Suppose we hypothesized that, instead of drift rate, the urgency signal was related to pupil diameter. We can write:

```
def signal_to_urgency(t, signal, binsize=.1):
    bin_number = int(t//binsize) # Which bin are we currently in?
    n_bins = len(signal) # Total number of bins for this condition
    # If we are currently in a bin which exceeds the total bins, fix to the last bin
    if bin_number >= n_bins:
        bin_number = n_bins-1
    return 1 + signal[bin_number]

class DriftUrgencyMomentToMoment(ddm.Drift):
    """Drift rate which varies over time, differently for each trial"""
    name = "Moment-to-moment urgency drift"
    required_parameters = ['snr'] # How much to scale moment-to-moment drift
    required_conditions = ['signal'] # should be a list of values which determine the␣
↪moment-to-moment drift
    def get_drift(self, t, conditions, **kwargs):
        return signal_to_urgency(t, conditions['signal']) * self.snr

class NoiseUrgencyMomentToMoment(ddm.Noise):
    """Noise rate which varies over time, differently for each trial"""
    name = "Moment-to-moment urgency noise"
    required_parameters = [] # How much to scale moment-to-moment drift
    required_conditions = ['signal'] # should be a list of values which determine the␣
↪moment-to-moment drift
    def get_noise(self, t, conditions, **kwargs):
        return signal_to_urgency(t, conditions['signal'])
```

Then, we could create a model using the following:

```
from pyddm import Model, Fittable
from pyddm.plot import model_gui
m = Model(drift=DriftUrgencyMomentToMoment(snr=Fittable(minval=0, maxval=2)),
          noise=NoiseUrgencyMomentToMoment(),
          dt=.01, dx=.01)
model_gui(m, conditions={"signal": [(.1, .1, .1, .1, .1, .1, 0), (0.0, 1, 2, 3), (0,␣
↪0.0, 0,0, 1)]})
```

3.10.4 Recipes for Collapsing Bounds

General use of bounds

Both linearly collapsing bounds (*BoundCollapsingLinear*) and exponentially collapsing bounds (*BoundCollapsingExponential*) already exist in PyDDM. For example:

```
from pyddm import Model
from pyddm.models import BoundCollapsingLinear, BoundCollapsingExponential
model1 = Model(bound=BoundCollapsingExponential(B=1, tau=2))
model2 = Model(bound=BoundCollapsingLinear(B=1, t=.2))
```

Step function collapsing bounds

It is also possible to make collapsing bounds of any shape. For example, the following describes bounds which collapse in discrete steps of a particular length:

```
from pyddm.models import Bound
class BoundCollapsingStep(Bound):
    name = "Step collapsing bounds"
    required_conditions = []
    required_parameters = ["B0", "stepheight", "steplength"]
    def get_bound(self, t, **kwargs):
        stepnum = t//self.steplength
        step = self.B0 - stepnum * self.stepheight
        return max(step, 0)
```

Try it out with:

```
from pyddm import Model, Fittable
from pyddm.plot import model_gui
model = Model(bound=BoundCollapsingStep(B0=Fittable(minval=.5, maxval=1.5),
                                        stepheight=Fittable(minval=0, maxval=.49),
                                        steplength=Fittable(minval=0, maxval=2)),
              dx=.01, dt=.01)
model_gui(model)
```

Weibull CDF collapsing bounds

The Weibull function is a popular choice for collapsing bounds. (See, e.g., Hawkins et al. (2015).) This can be implemented using:

```
import numpy as np
from pyddm import Bound
class BoundCollapsingWeibull(Bound):
    name = "Weibull CDF collapsing bounds"
    required_parameters = ["a", "aprime", "lam", "k"]
    def get_bound(self, t, **kwargs):
        l = self.lam
        a = self.a
        aprime = self.aprime
        k = self.k
        return a - (1 - np.exp(-(t/l)**k)) * (a - aprime)
```

Try it out with:

```
from pyddm import Model, Fittable
from pyddm.plot import model_gui
model = Model(bound=BoundCollapsingWeibull(a=Fittable(minval=1, maxval=2),
                                           aprime=Fittable(minval=0, maxval=1),
                                           lam=Fittable(minval=0, maxval=2),
                                           k=Fittable(minval=0, maxval=5)),
              dx=.01, dt=.01)
model_gui(model)
```

(Note that in Hakwins et al. (2015), diffusion goes from [0,1], whereas our diffusion goes from [-1,1]. Thus, the 0.5 term was removed.)

Delayed collapsing bounds

The following implements exponentially collapsing bounds with a delay:

```
class BoundCollapsingExponentialDelay(Bound):
    """Bound collapses exponentially over time.

    Takes three parameters:

    `B` - the bound at time t = 0.
    `tau` - the time constant for the collapse, should be greater than
    zero.
    `t1` - the time at which the collapse begins, in seconds
    """
    name = "Delayed exponential collapsing bound"
    required_parameters = ["B", "tau", "t1"]
    def get_bound(self, t, conditions, **kwargs):
        if t <= self.t1:
            return self.B
        if t > self.t1:
            return self.B * np.exp(-self.tau*(t-self.t1))
```

Bounds which depend on task conditions (e.g. speed vs accuracy tradeoff)

As an example of bounds which depend on a task conditions, we assume a task in which a subject is cued before the stimulus about whether to prioritize speed or accuracy. The following model could test the hypothesis that the subject changes their integration bound to be high for the accuracy condition and low for the speed condition.

```
from pyddm import Bound
class BoundSpeedAcc(Bound):
    name = "constant"
    required_parameters = ["Bacc", "Bspeed"]
    required_conditions = ['speed_trial']
    def get_bound(self, conditions, *args, **kwargs):
        assert self.Bacc > 0
        assert self.Bspeed > 0
        if conditions['speed_trial'] == 1:
            return self.Bspeed
        else:
            return self.Bacc
```

Try it out with:

```
from pyddm import Model, Fittable
from pyddm.plot import model_gui
model = Model(bound=BoundSpeedAcc(Bacc=Fittable(minval=.5, maxval=1.5),
                                  Bspeed=Fittable(minval=0, maxval=1)),
              dx=.01, dt=.01)
model_gui(model, conditions={"speed_trial": [0, 1]})
```

Increasing bounds

In addition to collapsing bounds, PyDDM also supports increasing bounds, or bounds which both increase and decrease over time. Note that performance is proportional to the maximum size of the bounds, so very large bounds should be avoided.

For example, the following bounds are constant from t=0 until t=1, increase from t=1 until t=1.2, decrease from t=1.2 until t=1.4, and then are again constant:

```
import pyddm as ddm
class BoundIncreasingAndDecreasing(ddm.Bound):
    name = "Increasing bound"
    required_conditions = []
    required_parameters = []
    def get_bound(self, t, *args, **kwargs):
        if t > 1 and t < 1.4:
            return 1 + (.2-abs(t-1.2))*3
        else:
            return 1
```

3.10.5 Recipes for Initial Conditions

General use of initial conditions

Initial conditions can be included in the model by passing it directly to the Model object. For example, for a uniform distribution centered at 0, do:

```
from pyddm import Model
from pyddm.models import ICRange
model = Model(IC=ICRange(sz=.2))
```

Biased starting point

Often we want to model a side bias, either those which arise naturally or those introduced experimentally through asymmetric reward or stimulus probabilities. The most popular way of modeling a side bias is to use a starting position which is closer to the boundary representing that side.

There are two ways to do this in PyDDM, *which are equivalent but implemented in different ways*. In the first, the upper and lower boundaries of the diffusion process represent the correct and incorrect answer. This scheme, sometimes called "accuracy coding", is the default for PyDDM. However, it requires a calculation for each trial to determine whether the bias is towards the upper or lower boundary. In the second way, the upper and lower boundaries represent distinct choices, e.g., "left" and "right". This scheme, sometimes called "stimulus coding", must be manually activated in PyDDM for a given Model and Sample by choosing names for the boundaries. This allows a much simpler (built-in) InitialCondition object to be used, but it also makes it harder to visualise performance independent of side.

For these examples, we will assume the two chices are “left” and “right”, and that we are implementing side bias to the left or right.

Accuracy coding for biased starting point

Here, the upper boundary will be “the correct response” (whether that is left or right on a given trial) and the lower boundary will be “the incorrect response”. This means the drift rate should always be positive.

First, we must first include a condition in our dataset describing whether the correct answer was the left side or the right side. Suppose we have a sample which has the `left_is_correct` condition, which is 1 if the (true underlying) correct answer is on the left side, and 0 if the (true underlying) correct answer is on the right side. Now, we can define an *InitialCondition* object which uses this information. We do this by defining a *get_IC()* method. This method should generate a discretized probability distribution for the starting position. Here, we want this distribution to be a single point `x0`, the sign of which (positive or negative) depends on whether the correct answer is on the left or right side. The function receives the support of the distribution in the `x` argument. We can model this with:

```
import numpy as np
from pyddm import InitialCondition
class ICPointSideBias(InitialCondition):
    name = "A starting point with a left or right bias."
    required_parameters = ["x0"]
    required_conditions = ["left_is_correct"]
    def get_IC(self, x, dx, conditions):
        start = np.round(self.x0/dx)
        # Positive bias for left choices, negative for right choices
        if not conditions['left_is_correct']:
            start = -start
        shift_i = int(start + (len(x)-1)/2)
        assert shift_i >= 0 and shift_i < len(x), "Invalid initial conditions"
        pdf = np.zeros(len(x))
        pdf[shift_i] = 1. # Initial condition at x=self.x0.
        return pdf
```

Then we can compare the distribution of left-correct trials to those of right-correct trials:

```
from pyddm import Model
from pyddm.plot import plot_compare_solutions
import matplotlib.pyplot as plt
model = Model(IC=ICPointSideBias(x0=.3))
s1 = model.solve(conditions={"left_is_correct": 1})
s2 = model.solve(conditions={"left_is_correct": 0})
plot_compare_solutions(s1, s2)
plt.show()
```

We can also see these directly in the model GUI:

```
from pyddm import Model, Fittable
from pyddm.plot import model_gui
model = Model(IC=ICPointSideBias(x0=Fittable(minval=0, maxval=1)),
              dx=.01, dt=.01)
model_gui(model, conditions={"left_is_correct": [0, 1]})
```

Stimulus coding for biased starting point

Instead of flipping the bias towards the upper or lower boundary, we can define the upper boundary as the left choice and the lower boundary as the right choice. Then, we need to code the choices differently. This can be achieved with two modifications: one to the sample, and the other to the model.

For the Sample, instead of specifying choices by whether the choice was correct or not, we instead need to define them by whether the choice was to the left or the right. **This requires a different input format for the data.** We can use the "choice_names" argument as follows:

```
samp = pyddm.Sample.from_pandas_dataframe(df, choice_column_name='choice_side',
                                              rt_column_name='rt',
                                              choice_names=("Left", "Right"))
```

This means that the choices specified in `df['choice_side']` are 1 if the choice is to the left side and 0 if it is to the right side. Other conditions in the data may need to change their representation as well, e.g., the coding of stimulus strength/coherence.

Then, when creating our model, we must do:

```
model = pyddm.Model(..., IC=ICPoint(x0=Fittable(minval=-1, maxval=1)), ..., choice_
↪names=("Left", "Right"))
```

The "choice_names" variable in the model must match the "choice_names" variable in the sample. This construction can be used for either *`ICPoint`* or *`ICPointRatio`*.

We can also try this out directly in the model GUI. Notice how the GUI no longer labels the sides as "correct" and "error", but instead, as "left" and "right":

```
from pyddm import Model, Fittable, ICPoint
from pyddm.plot import model_gui
model = Model(IC=ICPoint(x0=Fittable(minval=-1, maxval=1)),
              dx=.01, dt=.01, choice_names=("Left", "Right"))
model_gui(model)
```

Interpolation of starting points

Individual starting points may fall between a grid spacing when fitting data. If this becomes a problem (e.g., with large dx), it is possible to linearly approximate the probability density function at the two neighboring grids of the initial position. For instance, to do this for the biased initial conditions above:

```
import numpy as np
import scipy.stats
from pyddm import InitialCondition
class ICPointSideBiasInterp(InitialCondition):
    name = "A dirac delta function at a position dictated by the left or right side."
    required_parameters = ["x0"]
    required_conditions = ["left_is_correct"]
    def get_IC(self, x, dx, conditions):
        start_in = np.floor(self.x0/dx)
        start_out = np.sign(start_in)*(np.abs(start_in)+1)
        w_in = np.abs(start_out - self.x0/dx)
        w_out = np.abs(self.x0/dx - start_in)
        if not conditions['left_is_correct']:
            start_in = -start_in
            start_out = -start_out
```

(continues on next page)

(continued from previous page)

```
        shift_in_i = int(start_in + (len(x)-1)/2)
        shift_out_i = int(start_out + (len(x)-1)/2)
        if w_in>0:
            assert shift_in_i>= 0 and shift_in_i < len(x), "Invalid initial conditions
↪"
        if w_out>0:
            assert shift_out_i>= 0 and shift_out_i < len(x), "Invalid initial␣
↪conditions"
        pdf = np.zeros(len(x))
        pdf[shift_in_i] = w_in # Initial condition at the inner grid next to x=self.
↪x0.
        pdf[shift_out_i] = w_out # Initial condition at the outer grid next to x=self.
↪x0.
        return pdf
```

Try it out with:

```
from pyddm import Model, Fittable
from pyddm.plot import model_gui
model = Model(IC=ICPointSideBiasInterp(x0=Fittable(minval=0, maxval=1)),
              dx=.01, dt=.01)
model_gui(model, conditions={"left_is_correct": [0, 1]})
```

In practice, it is very similar, but the interpolated version gives a smoother derivative, which may be useful for gradient-based fitting methods (which are not used by default).

Fixed ratio instead of fixed value

When fitting both the initial condition and the bound height, it can be preferable to express the initial condition as a proportion of the total distance between the bounds. This ensures that the initial condition will always stay within the bounds, preventing errors in fitting. The "ratio" analog of *ICPoint* is *ICPointRatio*, which is built-in to PyDDM. If you want to make it depend on conditions, you can do the following:

```
import numpy as np
from pyddm import InitialCondition
class ICPointSideBiasRatio(InitialCondition):
    name = "A side-biased starting point expressed as a proportion of the distance␣
↪between the bounds."
    required_parameters = ["x0"]
    required_conditions = ["left_is_correct"]
    def get_IC(self, x, dx, conditions):
        x0 = self.x0/2 + .5 #rescale to between 0 and 1
        # Bias > .5 for left side correct, bias < .5 for right side correct.
        # On original scale, positive bias for left, negative for right
        if not conditions['left_is_correct']:
            x0 = 1-x0
        shift_i = int((len(x)-1)*x0)
        assert shift_i >= 0 and shift_i < len(x), "Invalid initial conditions"
        pdf = np.zeros(len(x))
        pdf[shift_i] = 1. # Initial condition at x=x0*2*B.
        return pdf
```

Try it out with:

```
from pyddm import Model, Fittable
from pyddm.models import BoundConstant
from pyddm.plot import model_gui
model = Model(IC=ICPointSideBiasRatio(x0=Fittable(minval=-1, maxval=1)),
              bound=BoundConstant(B=Fittable(minval=.1, maxval=2)),
              dx=.01, dt=.01)
model_gui(model, conditions={"left_is_correct": [0, 1]})
```

Biased Initial Condition Range

```
import numpy as np
import scipy.stats
from pyddm import InitialCondition
class ICPointRange(InitialCondition):
    name = "A shifted side-biased uniform distribution"
    required_parameters = ["x0", "sz"]
    required_conditions = ["left_is_correct"]
    def get_IC(self, x, dx, conditions, *args, **kwargs):
        # Check for valid initial conditions
        assert abs(self.x0) + abs(self.sz) < np.max(x), \
            "Invalid x0 and sz: distribution goes past simulation boundaries"
        # Positive bias for left correct, negative for right
        x0 = self.x0 if conditions["left_is_correct"] else -self.x0
        # Use "+dx/2" because numpy ranges are not inclusive on the upper end
        pdf = scipy.stats.uniform(x0 - self.sz, 2*self.sz+dx/10).pdf(x)
        return pdf/np.sum(pdf)
```

Try it out with constant drift using:

```
from pyddm import Model, Fittable, DriftConstant
from pyddm.plot import model_gui
model = Model(drift=DriftConstant(drift=1),
              IC=ICPointRange(x0=Fittable(minval=0, maxval=.5),
                              sz=Fittable(minval=0, maxval=.49)),
              dx=.01, dt=.01)
model_gui(model, conditions={"left_is_correct": [0, 1]})
```

Cauchy-distributed Initial Conditions

```
import numpy as np
import scipy.stats
from pyddm import InitialCondition
class ICCauchy(InitialCondition):
    name = "Cauchy distribution"
    required_parameters = ["scale"]
    def get_IC(self, x, dx, *args, **kwargs):
        pdf = scipy.stats.cauchy(0, self.scale).pdf(x)
        return pdf/np.sum(pdf)
```

Try it out with:

```
from pyddm import Model, Fittable, BoundCollapsingLinear
from pyddm.plot import model_gui
```

(continues on next page)

(continued from previous page)

```
model = Model(IC=ICCauchy(scale=Fittable(minval=.001, maxval=.3)),
              bound=BoundCollapsingLinear(t=0, B=1),
              dx=.01, dt=.01)
model_gui(model)
```

3.10.6 Recipes for Non-Decision Time

General use of non-decision time

The simplest way to incorporate a non-decision time is to use the built-in *OverlayNonDecision*, for example:

```
from pyddm import Model, Fittable, OverlayNonDecision
from pyddm.plot import model_gui
model = Model(overlay=OverlayNonDecision(nondectime=Fittable(minval=0, maxval=.8)),
              dx=.01, dt=.01)
model_gui(model)
```

Gaussian-distributed non-decision time

```
import numpy as np
import scipy
from pyddm import Overlay, Solution
class OverlayNonDecisionGaussian(Overlay):
    name = "Add a Gaussian-distributed non-decision time"
    required_parameters = ["nondectime", "ndsigma"]
    def apply(self, solution):
        # Make sure params are within range
        assert self.ndsigma > 0, "Invalid st parameter"
        # Extract components of the solution object for convenience
        choice_upper = solution.choice_upper
        choice_lower = solution.choice_lower
        dt = solution.dt
        # Create the weights for different timepoints
        times = np.asarray(list(range(-len(choice_upper), len(choice_upper))))*dt
        weights = scipy.stats.norm(scale=self.ndsigma, loc=self.nondectime).pdf(times)
        if np.sum(weights) > 0:
            weights /= np.sum(weights) # Ensure it integrates to 1
        newchoice_upper = np.convolve(weights, choice_upper, mode="full")[len(choice_
↪upper):(2*len(choice_upper))]
        newchoice_lower = np.convolve(weights, choice_lower, mode="full")[len(choice_
↪upper):(2*len(choice_upper))]
        return Solution(newchoice_upper, newchoice_lower, solution.model,
                        solution.conditions, solution.undec)
```

Try it out with:

```
from pyddm import Model, Fittable, OverlayNonDecision
from pyddm.plot import model_gui
model = Model(overlay=OverlayNonDecisionGaussian(
                  nondectime=Fittable(minval=0, maxval=.8),
                  ndsigma=Fittable(minval=0, maxval=.8)),
              dx=.01, dt=.01)
model_gui(model)
```

Different non-decision time for left and right trials

Here we show an example of a non-decision time which depends on both the trial conditions and mulitiple parameters. In this case, we use a different non-decision time based on whether a stimulus was presented on the left or right side of the subject.

```
import numpy as np
from pyddm import OverlayNonDecision, Solution
class OverlayNonDecisionLR(OverlayNonDecision):
    name = "Separate non-decision time for left and right sides"
    required_parameters = ["nondectimeL", "nondectimeR"]
    required_conditions = ["side"] # Side coded as 0=L or 1=R
    def get_nondecision_time(self, conditions):
        assert conditions['side'] in [0, 1], "Invalid side"
        return self.nondectimeL if conditions['side'] == 0 else self.nondectimeR
```

Try it out with:

```
from pyddm import Model, Fittable, OverlayNonDecision
from pyddm.plot import model_gui
model = Model(overlay=OverlayNonDecisionLR(
                  nondectimeL=Fittable(minval=0, maxval=.8),
                  nondectimeR=Fittable(minval=0, maxval=.8)),
              dx=.01, dt=.01)
model_gui(model, conditions={"side": [0, 1]})
```

3.10.7 Lapse rates for model fits

When fitting models, especially when doing so with likelihood, it is useful to have a constant lapse rate in the model to prevent the likelihood from being negative infinity. PyDDM has two useful built-in lapse rates for this which are used as mixture models: an `Exponential lapse rate` (according to a Poisson process, the recommended method) and the *`Uniform lapse rate`* (which is more common in the literature). These can be introduced with:

```
from pyddm import Model
from pyddm.models import OverlayPoissonMixture, OverlayUniformMixture
model1 = Model(overlay=OverlayPoissonMixture(pmixturecoef=.05, rate=1))
model2 = Model(overlay=OverlayUniformMixture(umixturecoef=.05))
```

If another overlay is to be used, such as *`OverlayNonDecision`*, then an *`OverlayChain`* object must be used:

```
from pyddm import Model
from pyddm.models import OverlayPoissonMixture, OverlayNonDecision, OverlayChain
model = Model(overlay=OverlayChain(overlays=[OverlayNonDecision(nondectime=.2),
                                             OverlayPoissonMixture(pmixturecoef=.05,␣
↪rate=1)]))
```

3.10.8 Loss functions

Fitting with an alternative loss function

Fitting methods can be modified by changing the loss function or by changing the algorithm used to optimize the loss function. The default loss function is likelihood (via *`LossLikelihood`*). Squared error (via *`LossSquaredError`*) and BIC (via *`LossBIC`*) are also available.

As an example, to fit the “Simple example” from the quickstart guide, do:

```
fit_adjust_model(samp, model_fit,
                 method="differential_evolution",
                 lossfunction=LossSquaredError)
```

Custom loss functions may be defined by extending *LossFunction*. The `loss` function must be defined and, given a model, returns the goodness of fit to the sample, accessible under `self.sample`. The `setup` function may optionally be defined, which is run once and may be used, for example, to do computations on the sample don't need to be run at each evaluation.

Fitting using mean RT and P(correct)

Here is an example of how to define a loss function as the sum of two terms: the mean response time of correct trials, and the probability of choosing the correct target. While unprincipled, it is a simple example that in practice gives similar fits to the analytical expression in Roitman and Shadlen (2002):

```
import numpy as np
from pyddm import LossFunction
class LossByMeans(LossFunction):
    name = "Mean RT and accuracy"
    def setup(self, dt, T_dur, **kwargs):
        self.dt = dt
        self.T_dur = T_dur
    def loss(self, model):
        sols = self.cache_by_conditions(model)
        MSE = 0
        for comb in self.sample.condition_combinations(required_conditions=self.
↪required_conditions):
            c = frozenset(comb.items())
            s = self.sample.subset(**comb)
            MSE += (sols[c].prob("correct") - s.prob("correct"))**2
            if sols[c].prob("correct") > 0:
                MSE += (sols[c].mean_decision_time() - np.mean(list(s)))**2
        return MSE
```

Fitting with undecided trials

In addition to correct and incorrect trials, some trials may go beyond the time allowed for a decision. The effect of these trials is usually minor due to the design of task paradigms, but PyDDM is capable of using these values within its fitting procedures.

Currently, the functions which import Sample objects from numpy arrays do not support undecided trials; thus, to include undecided trials in a sample, they must be passed directly to the Sample constructor in a more complicated form.

To construct a sample with undecided trials, first create a Numpy array of correct RTs and incorrect RTs in units of seconds, and count the number of undecided trials. Then, for each task conditions, create a tuple containing three elements. The first element should be a Numpy array with the task condition value for each associated correct RT, the second should be the same but for error trials, and the final element should be a Numpy array in no particular order with a number of elements equal to the undecided trials, with one corresponding to each undecided trial.

Consider the following example with "reward" as the task condition. We suppose there is one correct trial with a reward of 3 and an RT of 0.3s, one error with a reward of 2 and an RT of 0.5s, and two undecided trials with rewards of 1 and 2:

```
sample = Sample(np.asarray([0.3]), np.asarray([0.5]), 2,
                reward=(np.asarray([3]), np.asarray([2]), np.asarray([1, 2])))
```

A sample created using this method can be used the same way as one created using *from_numpy_array()* or *from_pandas_dataframe()*.

CHAPTER 4

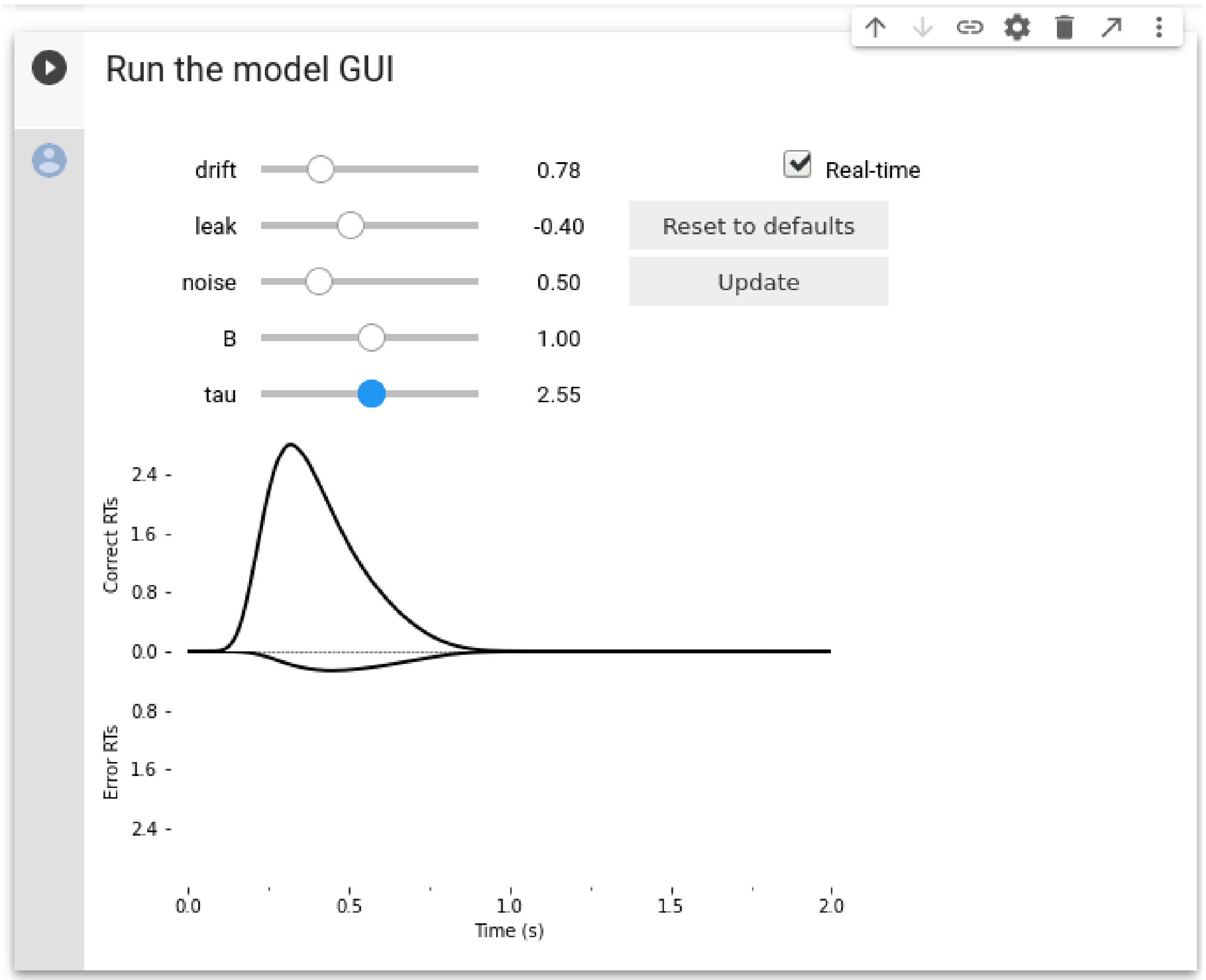
Model GUI

PyDDM additionally features a GUI which allows you to manually tweak the parameters and instantaneously see the change in the model. This is useful for learning about the DDM, for gaining an intuition on new models, and for testing new models before fitting them.

The radio buttons on the left may be used to show the conditional fit of the model on only specific task parameters. The sliders on the right may be used to control the parameter fit values. The function to plot may also be customized; see `plot.model_gui()` for more details.

4.1 Jupyter notebook GUI

In addition, this GUI is also compatible with Jupyter notebooks.



Try it out on Google Colab!

CHAPTER 5

API Documentation

modindex

5.1 Models

5.1.1 Model object

class pyddm.model.**Model**(*drift=DriftConstant(drift=0)*, *noise=NoiseConstant(noise=1)*, *bound=BoundConstant(B=1)*, *IC=ICPointSourceCenter()*, *overlay=OverlayNone()*, *name=''*, *dx=0.005*, *dt=0.005*, *T_dur=2.0*, *fitresult=None*, *choice_names=('correct', 'error')*)

Bases: `object`

A full simulation of a single DDM-style model.

Each model simulation depends on five key components:

- A description of how drift rate (drift) changes throughout the simulation.
- A description of how variability (noise) changes throughout the simulation.
- A description of how the boundary changes throughout the simulation.
- Starting conditions for the model
- Specific details of a task which cause dynamic changes in the model (e.g. a stimulus intensity change)

This class manages these, and also provides the affiliated services, such as analytical or numerical simulations of the resulting reaction time distribution.

IC(*conditions*)

The initial distribution at t=0.

Returns a length N ndarray (where N is the size of x_domain()) which should sum to 1.

can_solve_cn (*conditions={}*)
Check whether this model is compatible with Crank-Nicolson solver.

All bound functions which do not depend on time are compatible.

can_solve_explicit (*conditions={}*)
Check explicit method stability criterion

flux (*x*, *t*, *conditions*)
The flux across the boundary at position *x* at time *t*.

get_dependence (*name*)
Return the dependence object given by the string *name*.

get_fit_result ()
Returns a FitResult object describing how the model was fit.

Returns the FitResult object describing the last time this model was fit to data, including the loss function, fitting method, and the loss function value. If the model was never fit to data, this will return FitResultEmpty.

get_model_parameter_names ()
Get an ordered list of the names of all parameters in the model.

Returns the name of each model parameter. The ordering is arbitrary, but is uaranteed to be in the same order as get_model_parameters() and set_model_parameters(). If multiple parameters refer to the same "Fittable" object, then that object will only be listed once, however the names of the parameters will be separated by a "/" character.

get_model_parameters ()
Get an ordered list of all model parameters.

Returns a list of each model parameter which can be varied during a fitting procedure. The ordering is arbitrary but is guaranteed to be in the same order as get_model_parameter_names() and set_model_parameters(). If multiple parameters refer to the same "Fittable" object, then that object will only be listed once.

get_model_type ()
Return a dictionary which fully specifies the class of the five key model components.

has_analytical_solution ()
Is it possible to find an analytic solution for this model?

parameters ()
Return all parameters in the model

This will return a dictionary of dictionaries. The keys of this dictionary will be "drift", "noise", "bound", "IC", and "overlay". The values will be dictionaries, each containing the parameters used by these. Note that this includes both fixed parameters and Fittable parameters. If a parameter is fittable, it will return either a Fittable object or a Fitted object in place of the parameter, depending on whether or not the model has been fit to data yet.

set_model_parameters (*params*)
Set the parameters of the model from an ordered list.

Takes as an argument a list of parameters in the same order as those from get_model_parameters(). Sets the associated parameters as a "Fitted" object. If multiple parameters refer to the same "Fittable" object, then that object will only be listed once.

simulate_trial (*conditions={}*, *cutoff=True*, *rk4=True*, *seed=0*)
Simulate the decision variable for one trial.

Given conditions *conditions*, this function will simulate the decision variable for a single trial. It will cut off the simulation when the decision variable crosses the boundary unless *cutoff* is set to False. By default, Runge-Kutta is used to simulate the trial, however if *rk4* is set to False, the less efficient Euler's method is used instead. This returns a trajectory of the simulated trial over time as a numpy array.

Note that this will return the same trajectory on each run unless the random seed *seed* is varied.

Also note that you shouldn't normally need to use this function. To simulate an entire probability distributions, call Model.solve() and the results of the simulation will be in the returned Solution object. This is only useful for finding individual trajectories instead of the probability distribution as a whole.

simulated_solution(*conditions={}*, *size=1000*, *rk4=True*, *seed=0*)

Simulate individual trials to obtain a distribution.

Given conditions *conditions* and the number *size* of trials to simulate, this will run the function "simulate_trial" *size* times, and use the result to find a histogram analogous to solve. Returns a Sample object.

Note that in practice you should never need to use this function. This function uses an outdated method to simulate the model and should be used for comparison perposes only. To produce a probability density function of boundary crosses, use Model.solve(). To sample from the probability distribution (e.g. for finding confidence intervals for limited amounts of data), call Model.solve() and then use the Solution.resample() function of the resulting Solution.

solve(*conditions={}*, *return_evolution=False*, *force_python=False*)

Solve the model using an analytic solution if possible, and a numeric solution if not.

First, it tries to use Crank-Nicolson as the solver, and then backward Euler. See documentation of Model.solve_numerical() for more information.

The return_evolution argument should be set to True if you need to use the Solution.get_evolution() function from the returned Solution.

Return a Solution object describing the joint PDF distribution of reaction times.

solve_analytical(*conditions={}*, *force_python=False*)

Solve the model with an analytic solution, if possible.

Analytic solutions are only possible in a select number of special cases; in particular, it works for simple DDM and for linearly collapsing bounds and arbitrary single-point initial conditions. (See Anderson (1960) for implementation details.) For most reasonably complex models, the method will fail. Check whether a solution is possible with has_analytic_solution().

If successful, this returns a Solution object describing the joint PDF. If unsuccessful, this will raise an exception.

solve_numerical(*method='cn'*, *conditions={}*, *return_evolution=False*, *force_python=False*)

Solve the DDM model numerically.

Use *method* to solve the DDM. *method* can either be "explicit", "implicit", or "cn" (for Crank-Nicolson). This is the core DDM solver of this library.

Crank-Nicolson is the default and works for any model with constant bounds.

Implicit is the fallback method. It should work well in most cases and is generally stable.

Normally, the explicit method should not be used. Also note the stability criteria for explicit method is:

> noise^2/2 * dt/dx^2 < 1/2

It returns a Solution object describing the joint PDF. This method should not fail for any model type.

return_evolution (default=False) governs whether or not the function returns the full evolution of the pdf as part of the Solution object. This only works with methods "explicit" or "implicit", not with "cn".

force_python makes PyDDM use the solver written in Python instead of the optimized solver written in C.

solve_numerical_c(*conditions={}*)

Solve the DDM model using the implicit method with C extensions.

This function should give near identical results to solve_numerical_implicit. However, it uses compiled C code instead of Python code to do so, which should make it much (10-100x) faster.

This does not current work with non-Gaussian diffusion matrices (a currently undocumented feature).

solve_numerical_cn(*conditions={}*)

Solve the DDM model numerically using Crank-Nicolson.

This uses the Crank Nicolson method to solve the DDM at each timepoint. Results are then compiled together. This is the core DDM solver of this library.

It returns a Solution object describing the joint PDF.

solve_numerical_explicit(*conditions={}*, ***kwargs*)

Solve the model using the explicit method (Forward Euler).

See documentation for the solve_numerical method.

solve_numerical_implicit(*conditions={}*, ***kwargs*)

Solve the model using the implicit method (Backward Euler).

See documentation for the solve_numerical method.

t_domain()

A list of all of the timepoints over which the joint PDF will be defined (increments of dt from 0 to T_dur).

x_domain(*conditions*, *t=None*)

A list which spans from the lower boundary to the upper boundary by increments of dx.

5.1.2 Fittable object

class `pyddm.model.`**Fittable**(***kwargs*)

Bases: `float`

For parameters that should be adjusted when fitting a model to data.

Each Fittable object does not need any parameters, however several parameters may improve the ability to fit the model. In particular, *maxval* and *minval* ensure we do not choose an invalid parameter value. *default* is the value to start with when fitting; if it is not given, it will be selected at random.

default()

Choose a default value.

This chooses a value for the Fittable object randomly abiding by any constraints. Note that calling this function multiple times will give different results.

5.1.3 Fitted object

class `pyddm.model.`**Fitted**(*val*, ***kwargs*)

Bases: `pyddm.model.Fittable`

Parameters which used to be Fittable but now hold a value.

This extends float so that the parameters for the Fittable can be saved in the final model, even though there are no more Fittable objects in the final model.

5.1.4 Sample object (for RT data)

class pyddm.sample.**Sample**(*choice_upper*, *choice_lower*, *undecided=0*, *choice_names=('correct', 'error')*, ***kwargs*)

Bases: `object`

Describes a sample from some (empirical or simulated) distribution.

Similarly to Solution, this is a glorified container for three items: a list of reaction times for for the two choices (corresponding to upper and lower DDM boundaries), and the number of undecided trials. Each can have different properties associated with it, known as "conditions" elsewhere in this codebase. This is to specifiy the experimental parameters of the trial, to allow fitting of stimuli by (for example) color or intensity.

To specify conditions, pass a keyword argument to the constructor. The name should be the name of the property, and the value should be a tuple of length two or three. The first element of the tuple should be a list of length equal to the number of correct trials, and the second should be equal to the number of error trials. If there are any undecided trials, the third argument should contain a list of length equal to *undecided*.

By default, the choice associated with the upper boundary is "correct responses" and the lower boundary is "error responses". To change these, set the *choice_names* argument to be a tuple containing two strings, with the names of the boundaries. So the default is ("correct", "error"), but could be anything, e.g. ("left", "right"), ("high value" and "low value"), etc. This is sometimes referred to as "accuracy coding" and "stimulus coding". When fitting data, this must match the choice names of the model.

Optionally, additional data can be associated with each independent data point. These should be passed as keyword arguments, where the keyword name is the property and the value is a tuple. The tuple should have either two or three elements: the first two should be lists of properties for the correct and error reaction times, where the properties correspond to reaction times in the correct or error lists. Optionally, a third list of length equal to the number of undecided trials gives a list of conditions for these trials. If multiple properties are passed as keyword arguments, the ordering of the undecided properties (in addition to those of the correct and error distributions) will correspond to one another.

cdf(*choice*, *dt=0.01*, *T_dur=2*)

An estimate of the cumulative density function of sample RTs for a given choice.

choice should be the name of the choice for which to obtain the cdf, corresponding to the upper or lower boundary crossings. E.g., "correct", "error", or the choice names specified in the model's choice_names parameter.

Note that the return value will not converge to one, but both choices plus the undecided distribution will collectively converge to one.

cdf_corr(*dt=0.01*, *T_dur=2*)

The correct component of the joint CDF.

This method is deprecated, use Sample.cdf() instead.

cdf_err(*dt=0.01*, *T_dur=2*)

The error (incorrect) component of the joint CDF.

This method is deprecated, use Sample.cdf() instead.

condition_combinations(*required_conditions=None*)

Get all values for set conditions and return every combination of them.

Since PDFs of solved models in general depend on all of the conditions, this returns a list of dictionaries. The keys of each dictionary are the names of conditions, and the value is a particular value held by at least one element in the sample. Each list contains all possible combinations of condition values.

If *required_conditions* is iterable, only the conditions with names found within *required_conditions* will be included.

condition_names()

The names of conditions which hold some non-zero value in this sample.

condition_values(*cond*)

The values of a condition that have at least one element in the sample.

cond is the name of the condition from which to get the observed values. Returns a list of these values.

static from_numpy_array(*data*, *column_names*, *choice_names=('correct', 'error')*)

Generate a Sample object from a numpy array.

data should be an n x m array (n rows, m columns) where m>=2. The first column should be the response times, and the second column should be the choice that the trial corresponds to. E.g., by default, the choices are (1 == correct, 0 == error), but these can be changed by passing in a tuple of strings to the *choice_names* variable. E.g. ("left", "right") means that (1 == left, 0 == right).

Any remaining columns in *data* after the first two should be conditions. *column_names* should be a list of length m of strings indicating the names of the conditions. The order of the names should correspond to the order of the columns. This function does not yet work with undecided trials.

static from_pandas_dataframe(*df*, *rt_column_name*, *choice_column_name=None*, *choice_names=('correct', 'error')*, *correct_column_name=None*)

Generate a Sample object from a pandas dataframe.

df should be a pandas array. *rt_column_name* and *choice_column_name* should be strings, and *df* should contain columns by these names.

The column with the name *rt_column_name* should be the response times, and the column with the name *choice_column_name* should be the choice that the trial corresponds to. E.g., by default, the choices are (1 == correct, 0 == error), but these can be changed by passing in a tuple of strings to the *choice_names* variable. E.g. ("left", "right") means that (1 == left, 0 == right).

Any remaining columns besides these two should be conditions. This function does not yet work with undecided trials.

correct_column_name is deprecated and included only for backward compatibility.

items(*choice=None*, *correct=None*)

Iterate through the reaction times.

choice is whether to iterate through RTs corresponding to the upper or lower boundary, given as the name of the choice, e.g. "correct", "error", or the choice names specified in the model's choice_names parameter.

correct is a deprecated parameter for backward compatibility, please use *choice* instead.

For each iteration, a two-tuple is returned. The first element is the reaction time, the second is a dictionary containing the conditions associated with that reaction time.

If you just want the list of RTs, you can directly iterate through "sample.corr" and "sample.err".

mean_decision_time()

The mean decision time in the correct trials.

pdf(*choice*, *dt=0.01*, *T_dur=2*)

An estimate of the probability density function of sample RTs for a given choice.

choice should be the name of the choice for which to obtain the pdf, corresponding to the upper or lower boundary crossings. E.g., "correct", "error", or the choice names specified in the model's choice_names parameter.

Note that the return value will not sum to one, but both choices plus the undecided distribution will collectively sum to one.

pdf_corr(*dt=0.01*, *T_dur=2*)
The correct component of the joint PDF.

This method is deprecated, use Sample.pdf() instead.

pdf_err(*dt=0.01*, *T_dur=2*)
The error (incorrect) component of the joint PDF.

This method is deprecated, use Sample.pdf() instead.

prob(*choice*)
Probability of a given choice response.

choice should be the name of the choice for which to obtain the probability, corresponding to the upper or lower boundary crossings. E.g., "correct", "error", or the choice names specified in the model's

prob_correct()
The probability of selecting the right response.

This method is deprecated, use Sample.prob() instead.

prob_correct_forced()
The probability of selecting the correct response if a response is forced.

This method is deprecated, use Sample.prob_forced() instead.

prob_error()
The probability of selecting the incorrect (error) response.

This method is deprecated, use Sample.prob() instead.

prob_error_forced()
The probability of selecting the incorrect response if a response is forced.

This method is deprecated, use Sample.prob_forced() instead.

prob_forced(*choice*)
Probability of a given response if a response is forced.

choice should be the name of the choice for which to obtain the probability, corresponding to the upper or lower boundary crossings. E.g., "correct", "error", or the choice names specified in the model's

If a trajectory is undecided, then a response is selected randomly.

prob_undecided()
The probability of selecting neither response (undecided).

subset(***kwargs*)
Subset the data by filtering based on specified properties.

Each keyword argument should be the name of a property. These keyword arguments may have one of three values:

- A list: For each element in the returned subset, the specified property is in this list of values.
- A function: For each element in the returned subset, the specified property causes the function to evaluate to True.
- Anything else: Each element in the returned subset must have this value for the specified property.

Return a sample object representing the filtered sample.

static t_domain(*dt=0.01*, *T_dur=2*)
The times that corresponds with pdf/cdf_corr/err parameters (their support).

to_pandas_dataframe(*rt_column_name='RT'*, *choice_column_name='choice'*, *drop_undecided=False*, *correct_column_name=None*)

Convert the sample to a Pandas dataframe.

rt_column_name is the column label for the response time, and *choice_column_name* is the column label for the choice (corresponding to the upper or lower boundary).

Because undecided trials do not have an RT or choice, they are cannot be added to the data frame. To ignore them, thereby creating a dataframe which is smaller than the sample, set *drop_undecided* to True.

5.1.5 Solution object (for theoretical RT histograms)

class `pyddm.solution.`**Solution**(*pdf_choice_upper*, *pdf_choice_lower*, *model*, *conditions*, *pdf_undec=None*, *pdf_evolution=None*)

Bases: `object`

Describes the result of an analytic or numerical DDM run.

This is a glorified container for a joint pdf, between the response options (correct, error, and undecided) and the response time distribution associated with each. It stores a copy of the response time distribution for both the correct case and the incorrect case, and the rest of the properties can be calculated from there.

It also stores a full deep copy of the model used to simulate it. This is most important for storing, e.g. the dt and the name associated with the simulation, but it is also good to keep the whole object as a full record describing the simulation, so that the full parametrization of every run is recorded. Note that this may increase memory requirements when many simulations are run.

cdf(*choice*)

The cumulative density function of model RTs for a given choice.

choice should be the name of the choice for which to obtain the cdf, corresponding to the upper or lower boundary crossings. E.g., “correct”, “error”, or the choice names specified in the model’s choice_names parameter.

Note that the return value will not converge to one, but both choices plus the undecided distribution will collectively converge to one.

cdf_corr()

The correct component of the joint CDF.

This method is deprecated, use Solution.cdf() instead.

cdf_err()

The error (incorrect) component of the joint CDF.

This method is deprecated, use Solution.cdf() instead.

evaluate(*rt*, *choice=None*, *correct=None*)

Evaluate the pdf at a given response time.

rt is a time, greater than zero, at which to evaluate the pdf.

choice is whether to evaluate on the upper or lower boundary, given as the name of the choice, e.g. “correct”, “error”, or the choice names specified in the model’s choice_names parameter.

correct is a deprecated parameter for backward compatibility, please use *choice* instead.

Returns the value of the pdf at the given RT. Note that, despite being from a probability distribution, this may be greater than 0, since it is a continuous probability distribution.

mean_decision_time()

The mean decision time in the correct trials (excluding undecided trials).

pdf(*choice*)
The probability density function of model RTs for a given choice.

choice should be the name of the choice for which to obtain the pdf, corresponding to the upper or lower boundary crossings. E.g., "correct", "error", or the choice names specified in the model's choice_names parameter.

Note that the return value will not sum to one, but both choices plus the undecided distribution will collectively sum to one.

pdf_corr()
The correct component of the joint PDF.

This method is deprecated, use Solution.pdf() instead.

pdf_err()
The error (incorrect) component of the joint PDF.

This method is deprecated, use Solution.pdf() instead.

pdf_evolution()
The evolving state of the simulation: An array of size *x_domain() x t_domain()* whose columns contain the cross-sectional pdf for every time step.

If the model contains overlays, this represents the evolving state of the simulation *before* the overlays are applied. This is because overlays do not specify what to do with the diffusion locations corresponding to undercided probabilities. Additionally, all of the necessary information may not be stored, such as the case with a non-decision time overlay.

This means that in the case of models with a non-decision time t_nd, this gives the evolving probability at time T_dur + t_nd.

If no overlays are in the model, then sum(pdf_corr()[0:t]*dt) + sum(pdf_err()[0:t]*dt) + sum(pdf_evolution()[:,t]*dx) should always equal 1 (plus or minus floating point errors).

Note that this function will fail if the solution was not generated to contain information about the evolution of the pdf. This is not enabled by default, as it causes substantial memory overhead. To enable this, see the documentation for the Model.solve() argument "return_evolution", which should be set to True.

pdf_undec()
The final state of the simulation, same size as *x_domain()*.

If the model contains overlays, this represents the final state of the simulation *before* the overlays are applied. This is because overlays do not specify what to do with the diffusion locations corresponding to undercided probabilities. Additionally, all of the necessary information may not be stored, such as the case with a non-decision time overlay.

This means that in the case of models with a non-decision time t_nd, this gives the undecided probability at time T_dur + t_nd.

If no overlays are in the model, then pdf_corr() + pdf_err() + pdf_undec() should always equal 1 (plus or minus floating point errors).

prob(*choice*)
Probability of a given choice response within the time limit.

choice should be the name of the choice for which to obtain the probability, corresponding to the upper or lower boundary crossings. E.g., "correct", "error", or the choice names specified in the model's

prob_correct()
Probability of correct response within the time limit.

This method is deprecated, use Solution.prob() instead.

prob_correct_forced()

Probability of correct response if a response is forced.

Forced responses are selected randomly.

This method is deprecated, use Solution.prob_forced() instead.

prob_correct_sign()

Probability of correct response if a response is forced.

Forced responses are selected by the position of the decision variable at the end of the time limit T_dur.

This is only available for the implicit method.

This method is deprecated, use Solution.prob_sign() instead.

prob_error()

Probability of incorrect (error) response within the time limit.

This method is deprecated, use Solution.prob() instead.

prob_error_forced()

Probability of incorrect response if a response is forced.

Forced responses are selected randomly.

This method is deprecated, use Solution.prob_forced() instead.

prob_error_sign()

Probability of incorrect response if a response is forced.

Forced responses are selected by the position of the decision variable at the end of the time limit T_dur.

This is only available for the implicit method.

This method is deprecated, use Solution.prob_sign() instead.

prob_forced(*choice*)

Probability of a given response if a response is forced.

choice should be the name of the choice for which to obtain the probability, corresponding to the upper or lower boundary crossings. E.g., “correct”, “error”, or the choice names specified in the model’s

If a trajectory is undecided at the time limit (T_dur), then a response is selected randomly.

prob_sign(*choice*)

Probability of a given response if a response is forced.

choice should be the name of the choice for which to obtain the probability, corresponding to the upper or lower boundary crossings. E.g., “correct”, “error”, or the choice names specified in the model’s choice_names parameter.

If a trajectory is undecided at the time limit (T_dur), then a response is sampled from the distribution of decision variables at the final timepoint.

prob_undecided()

The probability of not responding during the time limit.

resample(*k=1*, *seed=None*)

Generate a list of reaction times sampled from the PDF.

k is the number of TRIALS, not the number of samples. Since we are only showing the distribution from the correct trials, we guarantee that, for an identical seed, the sum of the two return values will be less than *k*. If no undecided trials exist, the sum of return values will be equal to *k*.

This relies on the assumption that reaction time cannot be less than 0.

seed specifies the random seed to use in sampling. If unspecified, it does not set a random seed.

Returns a Sample object representing the distribution.

5.2 Model components

5.2.1 Drift rate

class `pyddm.models.drift.`**`Drift`**(***kwargs*)

Bases: *`pyddm.models.base.Dependence`*

Subclass this to specify how drift rate varies with position and time.

This abstract class provides the methods which define a dependence of drift on x and t. To subclass it, implement get_drift. Since it inherits from Dependence, subclasses must also assign a *name* and *required_parameters* (see documentation for Dependence.)

`get_drift`(*t*, *x*, *conditions*, ***kwargs*)

Calculate the instantaneous drift rate.

This function must be redefined in subclasses.

It may take several arguments:

- *t* - The time at which drift should be calculated
- *x* - The particle position (or 1-dimensional NDArray of particle positions) at which drift should be calculated
- *conditions* - A dictionary describing the task conditions

It should return a number or an NDArray (the same as *x*) indicating the drift rate at that particular time, position(s), and task conditions.

Definitions of this method in subclasses should only have arguments for needed variables and should always be followed by "**kwargs". For example, if the function does not depend on *t* or *x* but does depend on task conditions, this should be:

> def get_drift(self, conditions, **kwargs):

Of course, the function would still work properly if *x* were included as an argument, but this convention allows PyDDM to automatically select the best simulation methods for the model.

If a function depends on *x*, it should return a scalar if *x* is a scalar, or an NDArray of the same size as *x* if *x* is an NDArray. If the function does not depend on *x*, it should return a scalar. (The purpose of this is a dramatic speed increase by using numpy vectorization.)

`get_flux`(*x_bound*, *t*, *dx*, *dt*, *conditions*, ***kwargs*)

The drift component of flux across the boundary at position *x_bound* at time *t*.

Flux here is essentially the amount of the mass of the PDF that is past the boundary point *x_bound*.

There is generally no need to redefine this method in subclasses.

`get_matrix`(*x*, *t*, *dx*, *dt*, *conditions*, *implicit=False*, ***kwargs*)

The drift component of the implicit method diffusion matrix across the domain *x* at time *t*.

x should be a length N ndarray of all positions in the grid. *t* should be the time in seconds at which to calculate drift. *dt* and *dx* should be the simulations timestep and grid step *conditions* should be the conditions at which to calculate drift

Returns a sparse NxN matrix as a PyDDM TriDiagMatrix object.

There is generally no need to redefine this method in subclasses.

class `pyddm.models.drift.`**`DriftConstant`**(***kwargs*)

Bases: *`pyddm.models.drift.Drift`*

Drift dependence: drift rate is constant throughout the simulation.

Only take one parameter: drift, the constant drift rate.

Note that this is a special case of DriftLinear.

Example usage:

drift = DriftConstant(drift=0.3)

class `pyddm.models.drift.`**`DriftLinear`**(***kwargs*)

Bases: *`pyddm.models.drift.Drift`*

Drift dependence: drift rate varies linearly with position and time.

Take three parameters:

- *drift* - The starting drift rate
- *x* - The coefficient by which drift varies with x
- *t* - The coefficient by which drift varies with t

Example usage:

drift = DriftLinear(drift=0.5, t=0, x=-1) # Leaky integrator
drift = DriftLinear(drift=0.8, t=0, x=0.4) # Unstable integrator
drift = DriftLinear(drift=0, t=1, x=0.4) # Urgency function

5.2.2 Noise

class `pyddm.models.noise.`**`Noise`**(***kwargs*)

Bases: *`pyddm.models.base.Dependence`*

Subclass this to specify how noise level varies with position and time.

This abstract class provides the methods which define a dependence of noise on x and t. To subclass it, implement get_noise. Since it inherits from Dependence, subclasses must also assign a *name* and *required_parameters* (see documentation for Dependence.)

`get_flux`(*x_bound*, *t*, *dx*, *dt*, *conditions*, ***kwargs*)

The diffusion component of flux across the boundary at position *x_bound* at time *t*.

Flux here is essentially the amount of the mass of the PDF that is past the boundary point *x_bound* at time *t* (in seconds).

Note that under the central scheme we want to use x at half-grid from the boundary. This is however cleaner and justifiable using forward/backward scheme.

There is generally no need to redefine this method in subclasses.

`get_matrix`(*x*, *t*, *dx*, *dt*, *conditions*, *implicit=False*, ***kwargs*)

The diffusion component of the implicit method diffusion matrix across the domain *x* at time *t*.

x should be a length N ndarray of all positions in the grid. *t* should be the time in seconds at which to calculate noise. *dt* and *dx* should be the simulations timestep and grid step *conditions* should be the conditions at which to calculate noise

Returns a sparse NxN matrix as a PyDDM TriDiagMatrix object.

There is generally no need to redefine this method in subclasses.

get_noise(*conditions*, ***kwargs*)

Calculate the instantaneous noise (standard deviation of noise).

This function must be redefined in subclasses.

It may take several arguments:

- *t* - The time at which noise should be calculated
- *x* - The particle position (or 1-dimensional NDArray of particle positions) at which noise should be calculated
- *conditions* - A dictionary describing the task conditions

It should return a number or an NDArray (the same as *x*) indicating the standard deviation of the noise at that particular time, position(s), and task conditions.

Definitions of this method in subclasses should only have arguments for needed variables and should always be followed by "**kwargs". For example, if the function does not depend on *t* or *x* but does depend on task conditions, this should be:

> def get_noise(self, conditions, **kwargs):

Of course, the function would still work properly if *x* were included as an argument, but this convention allows PyDDM to automatically select the best simulation methods for the model.

If a function depends on *x*, it should return a scalar if *x* is a scalar, or an NDArray of the same size as *x* if *x* is an NDArray. If the function does not depend on *x*, it should return a scalar. (The purpose of this is a dramatic speed increase by using numpy vectorization.)

class pyddm.models.noise.**NoiseConstant**(***kwargs*)

Bases: *pyddm.models.noise.Noise*

Noise level is constant over time.

Only take one parameter: noise, the standard deviation of the noise.

Note that this is a special case of NoiseLinear.

Example usage:

> noise = NoiseConstant(noise=0.5)

class pyddm.models.noise.**NoiseLinear**(***kwargs*)

Bases: *pyddm.models.noise.Noise*

Noise level varies linearly with position and time.

Take three parameters:

- *noise* - The inital noise standard deviation
- *x* - The coefficient by which noise standard deviation varies with x
- *t* - The coefficient by which noise standard deviation varies with t

Example usage:

> noise = NoiseLinear(noise=0.5, x=0, t=.1) # Noise increases over time

5.2.3 Bounds

class pyddm.models.bound.**Bound**(***kwargs*)

Bases: *pyddm.models.base.Dependence*

Subclass this to specify how bounds vary with time.

This abstract class provides the methods which define a dependence of the bounds on t. To subclass it, implement get_bound. All bounds must be symmetric, so the lower bound is -get_bound.

Also, since it inherits from Dependence, subclasses must also assign a *name* and *required_parameters* (see documentation for Dependence.)

get_bound(*t*, *conditions*, ***kwargs*)

Calculate the bounds which particles cross to determine response time.

This function must be redefined in subclasses.

It may take up to two arguments:

- *t* - The time at which bound should be calculated
- *conditions* - A dictionary describing the task conditions

It should return a non-negative number indicating the upper bound at that particular time, and task conditions. The lower bound is taken to be the negative of the upper bound.

Definitions of this method in subclasses should only have arguments for needed variables and should always be followed by "**kwargs". For example, if the function does not depend on task conditions but does depend on time, this should be:

> def get_bound(self, t, **kwargs):

Of course, the function would still work properly if *conditions* were included as an argument, but this convention allows PyDDM to automatically select the best simulation methods for the model.

class pyddm.models.bound.**BoundConstant**(***kwargs*)

Bases: *pyddm.models.bound.Bound*

Bound dependence: bound is constant throuhgout the simulation.

Takes only one parameter: *B*, the constant bound.

Example usage:

> bound = BoundConstant(B=1.5) # Bound at 1.5 and -1.5

class pyddm.models.bound.**BoundCollapsingLinear**(***kwargs*)

Bases: *pyddm.models.bound.Bound*

Bound dependence: bound collapses linearly over time.

Takes two parameters:

- *B* - the bound at time t = 0.
- *t* - the slope, i.e. the coefficient of time, should be greater than zero.

Example usage:

> bound = BoundCollapsingLinear(B=1, t=.5) # Collapsing at .5 units per second

class pyddm.models.bound.**BoundCollapsingExponential**(***kwargs*)

Bases: *pyddm.models.bound.Bound*

Bound dependence: bound collapses exponentially over time.

Takes two parameters:

- *B* - the bound at time t = 0.
- *tau* - one divided by the time constant for the collapse. 0 gives constant bounds.

Example usage:

bound = BoundCollapsingExponential(B=1, tau=2.1) # Collapsing with time constant 1/2.1

5.2.4 Initial Conditions (IC)

class pyddm.models.ic.**InitialCondition**(***kwargs*)
Bases: *pyddm.models.base.Dependence*

Subclass this to compute the initial conditions of the simulation.

This abstract class describes initial PDF at the beginning of a simulation. To subclass it, implement get_IC(x).

Also, since it inherits from Dependence, subclasses must also assign a *name* and *required_parameters* (see documentation for Dependence.)

get_IC(*x*, *dx*, ***kwargs*)
Get the initial conditions (a PDF) withsupport *x*.

This function must be redefined in subclasses.

x is a length N ndarray representing the support of the initial condition PDF, i.e. the x-domain. This returns a length N ndarray describing the distribution.

class pyddm.models.ic.**ICPointSourceCenter**(***kwargs*)
Bases: *pyddm.models.ic.InitialCondition*

Initial condition: a dirac delta function in the center of the domain.

Example usage:

ic = ICPointSourceCenter()

class pyddm.models.ic.**ICPoint**(***kwargs*)
Bases: *pyddm.models.ic.InitialCondition*

Initial condition: any point.

Example usage:

ic = ICPoint(x0=.2)

class pyddm.models.ic.**ICPointRatio**(***kwargs*)
Bases: *pyddm.models.ic.InitialCondition*

Initial condition: any point expressed as a ratio between bounds, from -1 to 1.

Example usage:

ic = ICPointRatio(x0=-.2)

The advantage of ICPointRatio over ICPoint is that, as long as x0 is greater than -1 and less than 1, the starting point will always stay within the bounds, even when bounds are being fit.

class pyddm.models.ic.**ICUniform**(***kwargs*)
Bases: *pyddm.models.ic.InitialCondition*

Initial condition: a uniform distribution.

Example usage:

ic = ICUniform()

class pyddm.models.ic.**ICRange**(***kwargs*)

Bases: *pyddm.models.ic.InitialCondition*

Initial condition: a bounded uniform distribution with range from -sz to sz.

Example usage:

ic = ICRange(sz=.3)

class pyddm.models.ic.**ICGaussian**(***kwargs*)

Bases: *pyddm.models.ic.InitialCondition*

Initial condition: a Gaussian distribution with a specified standard deviation.

Example usage:

ic = ICRange(sz=.3)

pyddm.models.ic.**ICArbitrary**(*dist*)

Generate an IC object from an arbitrary distribution.

dist should be a 1 dimensional numpy array which sums to 1.

Note that ICArbitrary is a function, not an InitialCondition object, so it cannot be passed directly. It returns an instance of a an InitialCondition object which can be passed. So in place of, e.g. ICUniform(). In practice, the user should not notice a difference, and this function can thus be used in place of an InitialCondition object.

Example usage:

import scipy.stats

ic = ICArbitrary(dist=scipy.stats.binom.pmf(n=200, p=.4, k=range(0, 201))) # Binomial distribution

import numpy as np

ic = ICArbitrary(dist=np.asarray([0]*100+[1]+[0]*100)) # Equivalent to ICPointSourceCenter for dx=.01

5.2.5 Overlay

class pyddm.models.overlay.**Overlay**(***kwargs*)

Bases: *pyddm.models.base.Dependence*

Subclasses can modify distributions after they have been generated.

This abstract class provides the methods which define how a distribution should be modified after solving the model, for example for a mixture model. To subclass it, implement apply.

Also, since it inherits from Dependence, subclasses must also assign a *name* and *required_parameters* (see documentation for Dependence.)

apply(*solution*)

Apply the overlay to a Solution object.

This function must be redefined in subclasses.

This function takes a Solution object as its argument and returns a Solution object which was modified in some way. Often times, this will be by modifying *solution.corr* and *solution.choice_lower*. See the documentation for Solution for more information about this object.

Note that while this does not take *conditions* as an argument, conditions may still be accessed via *solution.conditions*.

Conceptually, this function performs some transformation on the simulated response time (first passage time) distributions. It is especially useful for non-decision times and mixture models, potentially in a parameter-dependent or condition-dependent manner.

`apply_trajectory`(*trajectory*, *model*, *rk4*, *seed*, *conditions={}*)
Apply the overlay to a simulated decision variable trajectory.

This function is optional and may be redefined in subclasses. It is expected to implement the same mechanism as the method "apply", but to do so on simulated trajectories (i.e. from Model.simulate_trial) instead of on a Solution object.

This function takes the t domain, the trajectory itself, and task conditions. It returns the modified trajectory.

`class` `pyddm.models.overlay.`**`OverlayNone`**(***kwargs*)
Bases: *`pyddm.models.overlay.Overlay`*

No overlay. An identity function for Solutions.

Example usage:

overlay = OverlayNone()

`class` `pyddm.models.overlay.`**`OverlayChain`**(***kwargs*)
Bases: *`pyddm.models.overlay.Overlay`*

Join together multiple overlays.

Unlike other model components, Overlays are not mutually exclusive. It is possible to transform the output solution many times. Thus, this allows joining together multiple Overlay objects into a single object.

It accepts one parameter: *overlays*. This should be a list of Overlay objects, in the order which they should be applied to the Solution object.

One key technical caveat is that the overlays which are chained together may not have the same parameter names. Parameter names must be given different names in order to be a part of the same overlay. This allows those parameters to be accessed by their name inside of an OverlayChain object.

Example usage:

overlay = OverlayChain(overlays=[OverlayNone(), OverlayNone(), OverlayNone()]) # Still equivalent to OverlayNone

overlay = OverlayChain(overlays=[OverlayPoissonMixture(pmixturecoef=.01, rate=1),
    OverlayUniformMixture(umixturecoef=.01)]) # Apply a Poission mixture and then a Uniform mixture

`class` `pyddm.models.overlay.`**`OverlayUniformMixture`**(***kwargs*)
Bases: *`pyddm.models.overlay.Overlay`*

A uniform mixture distribution.

The output distribution should be umixturecoef*100 percent uniform distribution and (1-umixturecoef)*100 percent of the distribution to which this overlay is applied.

A mixture with the uniform distribution can be used to confer robustness when fitting using likelihood.

Example usage:

overlay = OverlayUniformMixture(umixturecoef=.01)

`class` `pyddm.models.overlay.`**`OverlayExponentialMixture`**(***kwargs*)
Bases: *`pyddm.models.overlay.Overlay`*

An exponential mixture distribution.

The output distribution should be pmixturecoef*100 percent exponential distribution and (1-umixturecoef)*100 percent of the distribution to which this overlay is applied.

A mixture with the exponential distribution can be used to confer robustness when fitting using likelihood.

Note that this is called OverlayPoissonMixture and not OverlayExponentialMixture because the exponential distribution is formed from a Poisson process, i.e. modeling a uniform lapse rate.

Example usage:

> overlay = OverlayPoissonMixture(pmixturecoef=.02, rate=1)

class `pyddm.models.overlay.`**`OverlayNonDecision`**(***kwargs*)

Bases: *`pyddm.models.overlay.Overlay`*

Add a non-decision time

This shifts the reaction time distribution by *nondectime* seconds in order to create a non-decision time.

Example usage:

> overlay = OverlayNonDecision(nondectime=.2)

This can also be subclassed to allow easily shifting the non-decision time. When subclassing, override the *get_nondecision_time(self, conditions)* method to be any function you wish, using both conditions and parameters.

class `pyddm.models.overlay.`**`OverlayNonDecisionGamma`**(***kwargs*)

Bases: *`pyddm.models.overlay.Overlay`*

Add a gamma-distributed non-decision time

This shifts the reaction time distribution by an amount of time specified by the gamma distribution with shape parameter *shape* (sometimes called "k") and scale parameter *scale* (sometimes called "theta"). The distribution is then further shifted by *nondectime* seconds.

Example usage:

> overlay = OverlayNonDecisionGamma(nondectime=.2, shape=1.5, scale=.05)

This can also be subclassed to allow easily shifting the non-decision time. When subclassing, override the *get_nondecision_time(self, conditions)* method to be any function you wish, using both conditions and parameters.

class `pyddm.models.overlay.`**`OverlayNonDecisionUniform`**(***kwargs*)

Bases: *`pyddm.models.overlay.Overlay`*

Add a uniformly-distributed non-decision time.

The center of the distribution of non-decision times is at *nondectime*, and it extends *halfwidth* on each side.

Example usage:

> overlay = OverlayNonDecisionUniform(nondectime=.2, halfwidth=.02)

This can also be subclassed to allow easily shifting the non-decision time. When subclassing, override the *get_nondecision_time(self, conditions)* method to be any function you wish, using both conditions and parameters.

5.2.6 Loss function (for optimization)

class `pyddm.models.loss.`**`LossFunction`**(*sample, required_conditions=None, method=None, **kwargs*)

Bases: `object`

An abstract class for a function to assess goodness of fit.

This is an abstract class for describing how well data fits a model.

When subclasses are initialized, they will be initialized with the Sample object to which the model should be fit. Because the data will not change but the model will change, this is specified with initialization.

The optional *required_conditions* argument limits the stratification of *sample* by conditions to only the conditions mentioned in *required_conditions*. This decreases computation time by only solving the model for the condition names listed in *required_conditions*. For example, a simple DDM with no drift and constant variaince would mean *required_conditions* is an empty list.

The optional *method* argument can be "analytical", "numerical", "cn", "implicit", or "explicit".

This will automatically parallelize if set_N_cpus() has been called.

`cache_by_conditions`(*model*)

Solve the model for all relevant conditions.

Solve *model* for each combination of conditions found within the dataset. For example, if *required_conditions* is ["hand", "color"], and hand can be left or right and color can be blue or green, solves the model for: hand=left and color=blue; hand=right and color=blue; hand=left and color=green, hand=right and color=green.

This is a convenience function for defining new loss functions. There is generally no need to redefine this function in subclasses.

`loss`(*model*)

Compute the value of the loss function for the given model.

This function must be redefined in subclasses.

model should be a Model object. This should return a floating point value, where smaller values mean a better fit of the model to the data.

`setup`(***kwargs*)

Initialize the loss function.

The optional *setup* function is executed at the end of the initializaiton. It is executed only once at the beginning of the fitting procedure.

This function may optionally be redefined in subclasses.

class `pyddm.models.loss.`**`LossSquaredError`**(*sample*, *required_conditions=None*, *method=None*, ***kwargs*)

Bases: *`pyddm.models.loss.LossFunction`*

Squared-error loss function

class `pyddm.models.loss.`**`LossLikelihood`**(*sample*, *required_conditions=None*, *method=None*, ***kwargs*)

Bases: *`pyddm.models.loss.LossFunction`*

Likelihood loss function

class `pyddm.models.loss.`**`LossBIC`**(*sample*, *required_conditions=None*, *method=None*, ***kwargs*)

Bases: *`pyddm.models.loss.LossLikelihood`*

BIC loss function, functionally equivalent to LossLikelihood

class `pyddm.models.loss.`**`LossRobustLikelihood`**(*sample*, *required_conditions=None*, *method=None*, ***kwargs*)

Bases: *`pyddm.models.loss.LossLikelihood`*

Likelihood loss function which will not fail for infinite likelihoods.

Usually you will want to use LossLikelihood instead. See the FAQs in the documentation for more information on how this differs from LossLikelihood.

class `pyddm.models.loss.`**`LossRobustBIC`**(*sample*, *required_conditions=None*, *method=None*, ***kwargs*)

Bases: *`pyddm.models.loss.LossBIC`*

BIC loss function which will not fail for infinite likelihoods.

Usually you will want to use LossBIC instead. See the FAQs in the documentation for more information on how this differs from LossBIC.

5.2.7 Base

class `pyddm.models.base.`**`Dependence`**(***kwargs*)

Bases: `object`

An abstract class describing how one variable depends on other variables.

This is an abstract class which is inherrited by other abstract classes only, and has the highest level machinery for describing how one variable depends on others. For example, an abstract class that inherits from Dependence might describe how the drift rate may change throughout the simulation depending on the value of x and t, and then this class would be inherited by a concrete class describing an implementation. For example, the relationship between drift rate and x could be linear, exponential, etc., and each of these would be a subsubclass of Dependence.

In order to subclass Dependence, you must set the (static) class variable *depname*, which gives an alpha-numeric string describing which variable could potentially depend on other variables.

Each subsubclass of dependence must also define two (static) class variables. First, it must define *name*, which is an alpha-numeric plus underscores name of what the algorithm is, and also *required_parameters*, a python list of names (strings) for the parameters that must be passed to this algorithm. (This does not include globally-relevant variables like dt, it only includes variables relevant to a particular instance of the algorithm.) An optional (static) class variable is *default_parameters*, which is a dictionary indexed by the parameter names from *required_parameters*. Any parameters referenced here will be given a default value.

Dependence will check to make sure all of the required parameters have been supplied, with the exception of those which have default versions. It also provides other convenience and safety features, such as allowing tests for equality of derived algorithms and for ensuring extra parameters were not assigned.

5.3 Fitting

5.3.1 Functions for fitting

`pyddm.functions.`**`fit_adjust_model`**(*sample*, *model*, *fitparams=None*, *fitting_method='differential_evolution'*, *lossfunction=<class 'pyddm.models.loss.LossLikelihood'>*, *verify=False*, *method=None*, *verbose=True*)

Modify parameters of a model which has already been fit.

The data *sample* should be a Sample object of the reaction times to fit in seconds (NOT milliseconds). At least one of the parameters for one of the components in the model should be a "Fitted()" instance, as these will be the parameters to fit.

fitting_method specifies how the model should be fit. "differential_evolution" is the default, which accurately locates the global maximum without using a derivative. "simple" uses a derivative-based method to minimize, and just uses randomly initialized parameters and gradient descent. "simplex" is the Nelder-Mead method, and is a gradient-free local search. "basin" uses "scipy.optimize.basinhopping" to find an optimal solution, which is much slower but also gives better results than "simple". It does not appear to give better or faster results than "differential_evolution" in most cases. Alternatively, a custom objective function may be used by setting *fitting_method* to be a function which accepts the "x_0" parameter (for starting position) and "constraints" (for min and max values). In general, it is recommended you almost always use differential evolution, unless you have a model which is highly-constrained (e.g. only one or two parameters to estimate with low covariance) or you already know the approximate parameter values. In practice, besides these two special cases, changing the method is unlikely to give faster or more reliable estimation.

fitparams is a dictionary of kwargs to be passed directly to the minimization routine for fine-grained low-level control over the optimization. Normally this should not be needed.

lossfunction is a subclass of LossFunction representing the method to use when calculating the goodness-of-fit. Pass the subclass itself, NOT an instance of the subclass.

name gives the name of the model after it is fit.

If *verify* is False (the default), checking for programming errors is disabled during the fit. This can decrease runtime and may prevent crashes. If verification is already disabled, this does not re-enable it.

method gives the method used to solve the model, and can be "analytical", "numerical", "cn", "implicit", or "explicit".

verbose enables out-of-boundaries warnings and prints the model information at each evaluation of the fitness function.

Returns the same model object that was passed to it as an argument. However, the parameters will be modified. The model is modified in place, so a reference is returned to it for convenience only.

After running this function, the model will be modified to include a "FitResult" object, accessed as m.fitresult. This can be used to get the value of the objective function, as well as to access diagnostic information about the fit.

This function will automatically parallelize if set_N_cpus() has been called.

`pyddm.functions.`**`fit_model`**(*sample*, *drift=DriftConstant(drift=0)*, *noise=NoiseConstant(noise=1)*, *bound=BoundConstant(B=1)*, *IC=ICPointSourceCenter()*, *dt=0.005*, *dx=0.005*, *fitparams=None*, *fitting_method='differential_evolution'*, *method=None*, *overlay=OverlayNone()*, *lossfunction=<class 'pyddm.models.loss.LossLikelihood'>*, *verbose=True*, *name='fit_model'*, *verify=False*)

Fit a model to reaction time data.

The data *sample* should be a Sample object of the reaction times to fit in seconds (NOT milliseconds). This function will generate a model using the *drift*, *noise*, *bound*, and *IC* parameters to specify the model. At least one of these should have a parameter which is a "Fittable()" instance, as this will be the parameter to be fit.

Optionally, dt specifies the temporal resolution with which to fit the model.

method specifies how the model should be fit. "differential_evolution" is the default, which accurately locates the global maximum without using a derivative. "simple" uses a derivative-based method to minimize, and just uses randomly initialized parameters and gradient descent. "simplex" is the Nelder-Mead method, and is a gradient-free local search. "basin" uses "scipy.optimize.basinhopping" to find an optimal solution, which is much slower but also gives better results than "simple". It does not appear to give better or faster results than "differential_evolution" in most cases. Alternatively, a custom objective function may be used by setting *method* to be a function which accepts the "x_0" parameter (for starting position) and "constraints" (for min and max values). In general, it is recommended you almost always use differential evolution, unless you have a model which is highly-constrained (e.g. only one or two parameters to estimate with low covariance) or you already know the approximate parameter values. In practice, besides these two special cases, changing the method is unlikely to give faster or more reliable estimation.

fitparams is a dictionary of kwargs to be passed directly to the minimization routine for fine-grained low-level control over the optimization. Normally this should not be needed.

lossfunction is a subclass of LossFunction representing the method to use when calculating the goodness-of-fit. Pass the subclass itself, NOT an instance of the subclass.

name gives the name of the model after it is fit.

If *verify* is False (the default), checking for programming errors is disabled during the fit. This can decrease runtime and may prevent crashes. If verification is already disabled, this does not re-enable it.

verbose enables out-of-boundaries warnings and prints the model information at each evaluation of the fitness function.

Returns a "Model()" object with the specified *drift*, *noise*, *bound*, *IC*, and *overlay*.

The model will include a "FitResult" object, accessed as m.fitresult. This can be used to get the value of the objective function, as well as to access diagnostic information about the fit.

This function will automatically parallelize if set_N_cpus() has been called.

5.3.2 Fit result

class `pyddm.fitresult.`**FitResult**(*fitting_method*, *method*, *loss*, *value*, ***kwargs*)

An object to describe the result of a model fit.

This keeps track of information related to the fitting procedure. It has the following elements:

- method: the name of the solver used to solve the model, e.g. "analytical" or "implicit"
- fitting_method: the name of the algorithm used to minimize the loss function method (e.g. "differential_evolution")
- loss: the name of the loss function (e.g. "BIC")
- properties: a dictionary containing any additional values saved by the loss function or fitting procedure (e.g. "likelihood" for BIC loss function, or "mess" for a message describing the output).

So, for example, can access FitResult.method to get the name of the numerical algorithm used to solve the equation.

To access the output value of the loss function, use FitResult.value().

value()

Returns the objective function value of the fit.

If there was an error, or if no fit was performed, return inf.

class `pyddm.fitresult.`**FitResultEmpty**

A default Fit object before a model has been fit.

5.4 Plotting

`pyddm.plot.`**model_gui**(*model*, *sample=None*, *data_dt=0.01*, *plot=<function plot_fit_diagnostics>*, *conditions=None*, *verify=False*)

Mess around with model parameters visually.

This allows you to see how the model *model* would be affected by various changes in parameter values. It also allows you to easily plot *sample* conditioned on different conditions. A sample is required so that model_gui knows the conditions to include and the ratio of these conditions.

The function *plot* allows you to change what is plotted. By default, it is plot_fit_diagnostics. If you would like to define your own custom function, it must take four keyword arguments: "model", the model to plot, "sample", an optional (defaulting to None) Sample object to potentially compare to the model, "fig", an optional (defaulting to None) matplotlib figure to plot on, and "conditions", the conditions selected for plotting. It should not return anything, but it should draw the figure on "fig".

Because sometimes the model is run in very high resolution, *data_dt* allows you to set the bin width for *sample*.

For performance purposes, Paranoid Scientist verification is disabled when running this function. Enable it by setting the *verify* argument to True.

Some of this code is taken from *fit_model*.

`pyddm.plot.`**`model_gui_jupyter`**(*model*, *sample=None*, *data_dt=0.01*, *plot=<function plot_fit_diagnostics>*, *conditions=None*, *verify=False*)

Mess around with model parameters visually in a Jupyter notebook.

This function is equivalent to model_gui, but displays in a Jupyter notebook with controls. It does nothing when called outside a Jupyter notebook.

`pyddm.plot.`**`plot_compare_solutions`**(*s1*, *s2*)

Compare two model solutions to each other.

s1 and *s2* should be solution objects. This will display a pretty picture of the correct and error distribution pdfs.

`pyddm.plot.`**`plot_decision_variable_distribution`**(*model*, *conditions={}*, *resolution=0.1*, *figure=None*)

Show the distribution of the decision variable.

Show the intermediate distributions for the decision variable. *model* should be the model to plot, and *conditions* should be the conditions over which to plot it. *resolution* should be the timestep of the plot (NOT of the model). Optionally, *figure* is an existing figure on which to make the plot.

Also, note that for clarity of the visualization, the square root of the distribution is plotted instead of the distribution itself. Without this, it is quite difficult to see the evolving distribution because the distribution of histogram values is highly skewed.

Finally, note that this routine always uses the implicit method because it gives the most reliable histograms for the decision variable. (Crank-Nicoloson tends to oscillate.)

`pyddm.plot.`**`plot_fit_diagnostics`**(*model=None*, *sample=None*, *fig=None*, *conditions=None*, *data_dt=0.01*, *method=None*)

Visually assess model fit.

This function plots a model on top of data, primarily for the purpose of assessing the model fit. The plot can be configured with the following arguments:

- *model* - The model object to plot. None of the parameters should be "Fittable" instances, they should all be either "Fitted" or numbers.
- *sample* - A sample, normally the sample used to fit the model.
- *fig* - A matplotlib figure object. If not passed, the current figure will be used.
- *conditions* - Optionally restrict the conditions of the model to those specified, in a format which could be passed to Sample.subset.
- *data_dt* - Bin size to use for the data histogram. Defaults to 0.01.
- *method* - Optionally the method to use to solve the model, either "analytical", "numerical" "cn", "implicit", "explicit", or None (auto-select, the default).

`pyddm.plot.`**`plot_solution_cdf`**(*sol*, *ax=None*, *choice=None*, *correct=None*)

Plot the CDF of the solution.

- *ax* is optionally the matplotlib axis on which to plot.
- If *correct* is true, we draw the distribution of correct answers. Otherwise, we draw the error distributions.

This does not return anything, but it plots the CDF. It does not show it, and thus requires a call to plt.show() to see.

`pyddm.plot.`**`plot_solution_pdf`**(*sol*, *ax=None*, *choice=None*, *correct=True*)
Plot the PDF of the solution.

- *ax* is optionally the matplotlib axis on which to plot.
- If *correct* is true, we draw the distribution of correct answers. Otherwise, we draw the error distributions.

This does not return anything, but it plots the PDF. It does not show it, and thus requires a call to plt.show() to see.

5.5 Utilities

`pyddm.functions.`**`models_close`**(*m1*, *m2*, *tol=0.1*)
Determines whether two models are similar.

This compares the parameters of models *m1* and *m2* and checks to make sure that each of the parameters in model *m1* is within a distance of *tol* of *m2*. Return True if this is the case, otherwise False.

`pyddm.functions.`**`evolution_strategy`**(*fitness*, *x_0*, *mu=1*, *lmbda=3*, *copyparents=True*, *mutate_var=0.002*, *mutate_prob=0.5*, *evals=100*, *seed=None*)
Optimize using the Evolution Strategy (ES) heuristic method.

Evolution Strategy is an optimization method specified in the form (lambda + mu) or (lambda, mu), for some integer value of lambda and mu. The algorithm will generate an intial population of lambda individuals. Each individual will have an equal number of offspring, so that there is a total of mu organisms in the next generation. In the case of the (lambda + mu) algorithm, the parents are also copied into the next generation. Then, the lambda best organisms in this population are selected to reproduce.

The starting population includes *x_0* and *lmbda*-1 other individuals generated by mutating *x_0*. Mutations occur by perturbing *x_0* by a Gaussian-distributed variable (so-called "Gaussian convolution") with variance *mutate_var*. Each element is changed with a probability *mutate_prob*. The number of function evaluations will be approximately *evals*, as this algorithm will iterate *evals/lmbda* times.

lmbda is the lambda parameter (note the spelling difference) and *mu* is the mu parameter for the ES. If *copyparents* is True, use (*lmbda* + *mu*), and if it is False, use (*lmbda*, *mu*).

seed allows optional seed values to be used during random number generation during evolution-driven optimization. If set to None, random number generation is subject to unseeded behavior. If convergence is difficult, setting seed=None may result in different solutions between runs.

The purpose of this is if you already have a good model, but you want to test the local space to see if you can make it better.

`pyddm.functions.`**`solve_partial_conditions`**(*model*, *sample=None*, *conditions=None*, *method=None*)
Solve a model without specifying the value of all conditions

This function solves *model* according to the ratio of trials in *sample*. For example, suppose *sample* has 100 trials with high coherence and 50 with low coherence. This will then return a solution with 2/3*(PDF high coherence) + 1/3*(PDF low coherence). This is especially useful when comparing a model to a sample which may have many different conditions.

Alternatively, if no sample is available, it will solve all conditions passed in *conditions* in equal ratios.

The advantage to this function over Model.solve() is that the former can only handle a single value for each condition, whereas this function accepts can do it lists for condition values as well.

The *conditions* variable limits the solution to a subset of *sample* which satisfy *conditions*. The elements of the dictionary *conditions* should be specified either as values or as a list of values.

Optionally, *method* describes the solver to use. It can be "analytical", "numerical", "cn" (Crank-Nicolson), "implicit" (backward Euler), "explicit" (forward Euler), or None (auto-detect method).

This function will automatically parallelize if set_N_cpus() has been called.

`pyddm.functions.`**`hit_boundary`**(*model*)

Returns True if any Fitted objects are close to their min/max value

`pyddm.functions.`**`dependence_hit_boundary`**(*pv*)

Returns True if a Fitted instance has hit the boundary.

Fitted instances may have minimum or maximum values attached to them. If it does, and if it has gotten close to this min/max while fitting, return True. Otherwise, or if the value is not a Fitted object, return False.

`pyddm.functions.`**`display_model`**(*model*, *print_output=True*)

A readable way to display models.

model should be any Model object. Prints a description of the model, and does not return anything.

`pyddm.functions.`**`get_model_loss`**(*model*, *sample*, *lossfunction=<class 'pyddm.models.loss.LossLikelihood'>*, *method=None*)

A shortcut to compusing the loss of a model.

A shortcut method to compute the loss (under loss function *lossfunction*) of Model *model* with respect to Sample *sample*. Optionaly, specificy the numerical method *method* to use (e.g. analytical, numerical, implicit, etc.)

Note that this should not be used when performing model fits, as it is faster to use the optimizations implemented in fit_adjust_model.

CHAPTER 6

FAQs

6.1 How do I know if my model will run analytically or numerically?

The function *solve()* will automatically choose the best solver for your model. Solvers, in order of preference, are

1. Analytical
2. Crank-Nicolson
3. Backward Euler (implicit method)

The analytical solver requires that Drift and Noise do not depend on time or particle position, and that the initial position is fixed at zero (*ICPointSourceCenter*). Additionally, they require either Bounds which do not depend on time, or alternatively linearly collapsing bounds using *BoundCollapsingLinear*. (Parameterized linearly collapsing bounds are not currently supported.)

The Crank-Nicolson solver requires that Bounds do not depend on time, i.e. models with collapsing bounds are not supported.

The Backward Euler (implicit method) solver is compatible with all models and thus serves as a fallback.

Particle simulations and Forward Euler (explicit method) are also available, but must be explicitly called via *solve_numerical_explicit()* and *simulated_solution()*. They will never be chosen automatically.

For custom models, these are specified by including `x` or `t` in the argument list.

6.2 What arguments do `get_drift()`, `get_noise()`, etc. take?

The most appropriate solver is selected by PyDDM by examining the variables on which different model components depend. For models components built-in to PyDDM, this is checked automatically.

For custom model components, this is found based on the arguments taken by the relevant model function. Internally, the time is passed as the number `t` keyword argument, the position is passed as a number or vector `x`, and the trial conditions are passed as a dictionary `conditions`. Thus, by including any of these variables in the method definition, we thereby depend on that parameter. (Likewise, all methods should end with `**kwargs` to not throw an error when other parameters are passed, but should not access the `kwargs` variable directly.)

For example, suppose our experiment consisted of two blocks, one associated with high rewards and the other associated with low rewards, and we hypothesize that different bounds are used in these two cases. We could create the following Bound object which allows the bounds in the two blocks to be fit independently:

```
class BoundReward(pyddm.Bound):
    name = "Reward-modulated bounds"
    required_conditions = ["block"]
    required_parameters = ["bound1", "bound2"]
    def get_bound(self, conditions, **kwargs):
        if conditions["block"] == 1:
            return self.bound1
        elif conditions["block"] == 2:
            return self.bound2
```

Notice how the `get_bound` function does not depend on `t`. However, we could in theory also define the method `get_bound` as:

```
def get_bound(self, t, conditions, **kwargs):
    ...
```

Even if the variable `t` is never used inside `get_bound`, PyDDM would interpret this to mean that the function depends on time. Thus, while this will give the expected result, it will not allow PyDDM to properly optimize.

Alternatively, suppose a separate hypothesis whereby the block described above modulates the rate of collapse in exponentially collapsing bounds. This could be modeled as:

```
import numpy as np
class BoundReward(pyddm.Bound):
    name = "Reward-modulated bounds"
    required_conditions = ["block"]
    required_parameters = ["rate1", "rate2", "B0"]
    def get_bound(self, t, conditions, **kwargs):
        if conditions["block"] == 1:
            rate = self.rate1
        elif conditions["block"] == 2:
            rate = self.rate2
        return self.B0 * np.exp(-rate*t)
```

In this case, our bound does depend on `t`, so it **must** be included in the function signature.

6.3 Why do I get "Paranoid" errors?

Paranoid Scientist is a library for verifying the accuracy of scientific software. It is used to check the entry and exit conditions of functions.

Paranoid Scientist will, overall, decrease the probability of an undetected error by increasing the number of bugs overall. Some common errors are:

- When a particular model parametrization causes numerical instability at the given dx and dt. This can cause probability distributions which go below zero.
- When numerical issues are amplified in the model, making the distribution integrate to more than 1 (plus floating point error).

- When dx and dt are too small for Crank-Nicolson and oscillations occur in the distribution.

If this becomes a problem during model fitting, it can be disabled with:

```
import paranoid as pns
pns.settings.Settings.set(enabled=False)
```

When performing final simulations for the paper, it is recommended to keep re-enable Paranoid Scientist, since turning it off may mask numerical issues.

6.4 Can PyDDM fit hierarchical models?

No, PyDDM cannot fit hierarchical models. This need is already addressed by the hddm package. Due to limited resources, we do not plan to add support for hierarchical models, but you are welcome to implement the feature yourself and submit a pull request on Github. If you plan to implement this feature, please let us know so we can help you get familiar with the code.

6.5 What is the difference between LossLikelihood and LossRobustLikelihood or LossBIC and LossRobustBIC?

Maximum likelihood in general is not good at handling probabilities of zero. When performing fitting using maximum likelihood (or equivalently, BIC), the fit will fail if there are any times at which the likelihood is zero. If there is even one trial in the experimental data which falls into a region where the simulated probability distribution is zero, then the likelihood of the model under that data is zero, and hence negative log likelihood is infinity. (See Ratcliff and Tuerlinckx (2002) for a more complete discussion.) In practice, there can be several locations where the likelihood is theoretically zero. For example, the non-decision time by definition should have no responses. However, data are noisy, and some responses may be spurious. This means that when fitting with likelihood, the non-decision time cannot be any longer than the shortest response in the data. Clearly this is not acceptable.

PyDDM has two ways of circumventing this problem. The most robust way is to fit the data with a mixture model. Here, the DDM process is mixed with another distribution (called a "lapse", "contaminant", or "outlier" distribution) which represent responses which came from a non-DDM process. Traditionally *a uniform distribution* has been used, but PyDDM also offers the option of using *an exponential distribution*, which has the benefit of providing a flat lapse rate hazard function. If you would also like to have a non-decision time, you may need to *chain together multiple overlays*.

The easier option is to use the *LossRobustLikelihood* loss function. This imposes a minimum value for the likelihood. In theory, it is similar to imposing a uniform distribution, but with an unspecified mixture probability. It will give nearly identical results as LossLikelihood if there are no invalid results, but due to the minimum it imposes, it is more of an approximation than the true likelihood.

6.6 Why do I get oscillations in my simulated RT distribution?

Oscillations occur in the Crank-Nicolson method when your dt is too large. Try decreasing dt. You should almost never use a dt larger than .01, but smaller values are ideal.

6.7 Why is PyDDM so slow?

Your model may be slow for a number of different reasons.

- **You have a lot of conditions** – Each time you solve the model (e.g. by calling `Model.solve()`), PyDDM internally needs to simulate one pdf per potential combination of conditions. For example, if you are using 200 different coherence values, then PyDDM will need to simulate 200 different pdfs for each call you make to `Model.solve()`. This also compounds multiplicativly: if you have 200 coherence conditions and 10 reward conditions, you will get $200 \times 10 = 2000$ pdf simulations per call to `Model.solve()`. During fitting, `Model.solve()` is called hundreds of times. As you can imagine, having too many conditions slows things down quite a bit. Minimizing the number of conditions will thus lead to substantial speedups.
- **Your numerics (dx and dt) are too small** – Larger values of dx and dt can lead to imprecise estimations of the response time distribution. Therefore, be cautious when adjusting dx and dt. As a rule of thumb, dx and dt should almost always be smaller than 0.01 and larger than 0.0001. Setting them to 0.001 is a good place to start. If dx and dt are larger than 0.01, your estimated response time distribution will be inaccurate, and if dx and dt are smaller than 0.0001, solving the model will be extremely slow.
- **The C solver is not working properly** – You can confirm that the C solver is operating by ensuring the variable `pyddm.model.HAS_CSOLVE` is True. If there was an error installing the C solver when installing PyDDM, PyDDM will still run, but it will be 10-100x slower.

6.8 How many trials do I need to fit a GDDM to data?

Since the GDDM is a framework rather than a specific model, there is no firm minimum number of trials you need to fit a GDDM. All GDDMs are different, and so different models, fitting procedures, and objective functions could require different sample sizes.

However, in general, there cannot be a "minimum sample size", because the more data available, the more precise the parameters estimates will be. Therefore, the required sample size depends on how much variability one is willing to tolerate in the parameter estimates. This is true for other packages as well, and so when other packages make claims about minimum sample size, these estimates should be interpreted as rough guides of what people tend to use rather than interpreted literally.

However, PyDDM makes it easy to test parameter recovery, which can be considered a gold standard for determining the required sample size. This allows you to determine how many trials you need in order to get the parameter variability you're willing to tolerate. The idea is to build the model you want to fit, choose reasonable-ish default parameters, and then simulate several trials from that model using the `Solution.resample()` method. After you simulate these trials for different sample sizes, you fit the same model (but with Fittable parameters) to the generated data. Then, you can find how close the parameter estimates are to the actual parameters when you have different sample sizes.

6.9 Does PyDDM support HDDM's "stimulus coding"?

Yes, see *Stimulus coding vs accuracy coding vs anything else coding*.

6.10 Does PyDDM allow non-discrete conditions?

PyDDM runs fastest when there are a smaller number of conditions. However, PyDDM is frequently used for models where there is a separate condition for each trial. For instance, it is possible to have drift rate depend on other observations, such as eye movements or electrophysiological signals. See *Moment-to-moment observations* for an example.

While PyDDM is able to do this faster than most other software packages, PyDDM is fastest when there are fewer conditions. (The execution time increases linearly with the number of conditions.) PyDDM can also parallelize this with no extra effort required by the user to make it even faster.

Unfortunately, there are limits to this speed. According to the two standard solver methodologies (both supported by PyDDM), it is either possible to simulate individual diffusion trajectories, or to solve the Fokker-Planck equation separately for each trial. If PyDDM isn't fast enough, the only (as of 2022) way to make simulations with many conditions run faster is to simulate many instances and then train a deep neural network on the RT distribution. There is a way to do this in HDDM, described in Fengler et al (2022). No such feature is currently planned in PyDDM.

6.11 When should I use RobustLikelihood or RobustBIC?

RobustLikelihood and RobustBIC are almost identical to Likelihood and BIC, but they have a uniform distribution mixture model built in. (More specifically, it sets a "minimum value" for the log likelihood by adding a small constant term to it.) This is to avoid infinite likelihoods where the distribution is zero. If you are already using a mixture model (e.g. OverlayUniformMixture or OverlayExponentialMixture), then you should not use RobustLikelihood or RobustBIC.

If you compare the likelihood or BIC of two models using the robust versions, keep in mind that you are actually comparing the mixture model. This is necessary for likelihood estimation and therefore occurs in other packages as well, which refer to it as the probability of "contaminant RTs" (fast-dm) or "outliers" (HDDM).

CHAPTER 7

Contact

Please report bugs to <https://github.com/mwshinn/pyddm/issues>. This includes any problems with the documentation. PRs for bugs are greatly appreciated.

Feature requests are currently not being accepted due to limited resources. If you implement a new feature in PyDDM, please do the following before submitting a PR on Github:

- Make sure your code is clean and well commented
- If appropriate, update the official documentation in the docs/ directory
- Ensure there are Paranoid Scientist verification conditions to your code
- Write unit tests and optionally integration tests for your new feature (runtests.sh)
- Ensure all existing tests pass (`runtests.sh` returns without error)

For all other questions or comments, contact m.shinn@ucl.ac.uk.

Python Module Index

Index