symfit Documentation

Release 0.3.0

tBuLi

December 07, 2015

Contents:

Introduction

Existing fitting modules are not very pythonic in their API and can be difficult for humans to use. This project aims to marry the power of `scipy.optimize` with the readability of `SymPy` to create a highly readable and easy to use fitting package which works for projects of any scale.

`symfit` makes it extremely easy to provide guesses for your parameters and to bound them to a certain range:

```
a = Parameter(1.0, min=0.0, max=5.0)
```

To define models to fit to:

```
x = Variable()
A = Parameter()
sig = Parameter(1.0, min=0.0, max=5.0)
x0 = Parameter(1.0, min=0.0)

# Gaussian distrubution
model = A * exp(-(x - x0)**2/(2 * sig**2))
```

And finally, to execute the fit:

```
fit = Fit(model, xdata, ydata)
fit_result = fit.execute()
```

And to evaluate the model using the best fit parameters:

```
y = model(x=xdata, **fit_result.params)
```

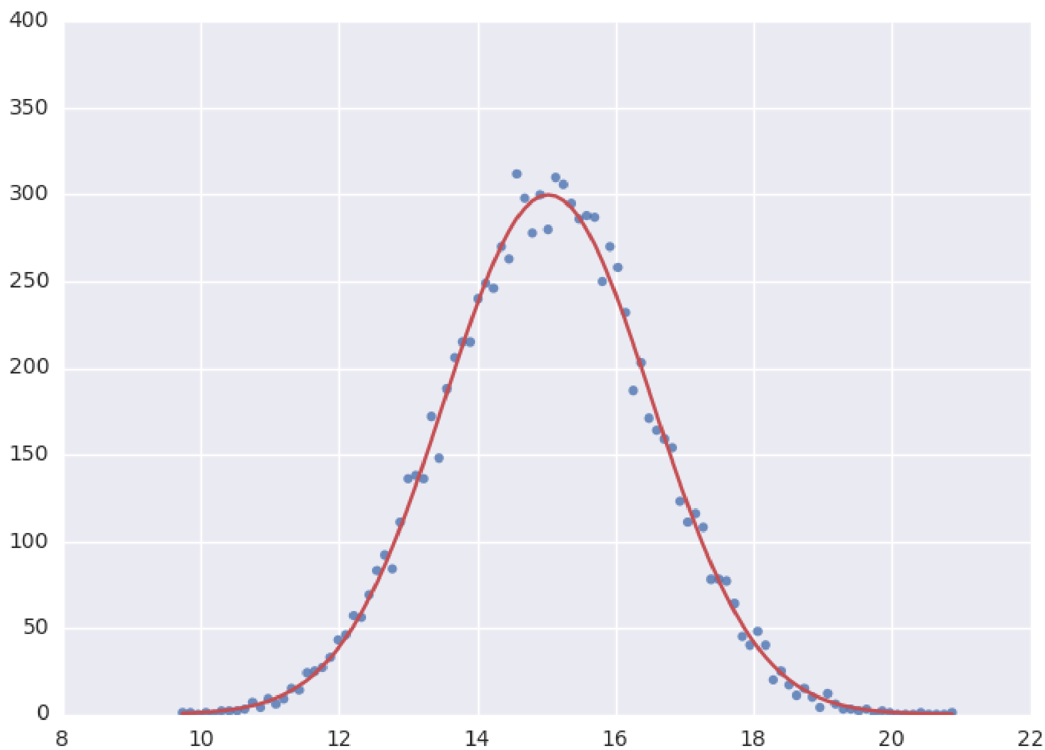
As your models become more complicated, `symfit` really comes into it's own. For example, vector valued functions are both easy to define and beautiful to look at:

```
model = {
    y_1: x**2,
    y_2: 2*x
}
```

And constrained maximization has never been this easy:

```
x, y = parameters('x, y')

model = 2*x*y + 2*x - x**2 -2*y**2
constraints = [
    Eq(x**3 - y, 0),     # Eq: ==
    Ge(y - 1, 0),        # Ge: >=
]
```


400
350
300
250
200
150
100
50
0
8
10
12
14
16
18
20
22

```
fit = Maximize(model, constraints=constraints)
```

1.1 Technical Reasons

On a more technical note, this symbolic approach turns out to have great technical advantages over using scipy directly. In order to fit, the algorithm needs the Jacobian: a matrix containing the derivatives of your model in it's parameters. Because of the symbolic nature of `symfit`, this is determined for you on the fly, saving you the trouble of having to determine the derivatives yourself. Furthermore, having this Jacobian allows good estimation of the errors in your parameters, something `scipy` does not always succeed in.

Installation

If you are using pip, you can simply run

```
pip install symfit
```

from your terminal. If you are using linux and do not use pip, you can download the source from https://github.com/tBuLi/symfit and install manually.

Are you not on linux and you do not use pip? That's your own mess.

2.1 Dependencies

```
pip install sympy
pip install numpy
pip install scipy
```

CHAPTER 3

Tutorial

3.1 Simple Example

The example below shows how easy it is to define a model that we could fit to.

```
from symfit.api import Parameter, Variable

a = Parameter()
b = Parameter()
x = Variable()
model = a * x + b
```

Lets fit this model to some generated data.

```
from symfit.api import Fit
import numpy as np

xdata = np.linspace(0, 100, 100) # From 0 to 100 in 100 steps
a_vec = np.random.normal(15.0, scale=2.0, size=(100,))
b_vec = np.random.normal(100.0, scale=2.0, size=(100,))
ydata = a_vec * xdata + b_vec # Point scattered around the line 5 * x + 105

fit = Fit(model, xdata, ydata)
fit_result = fit.execute()
```

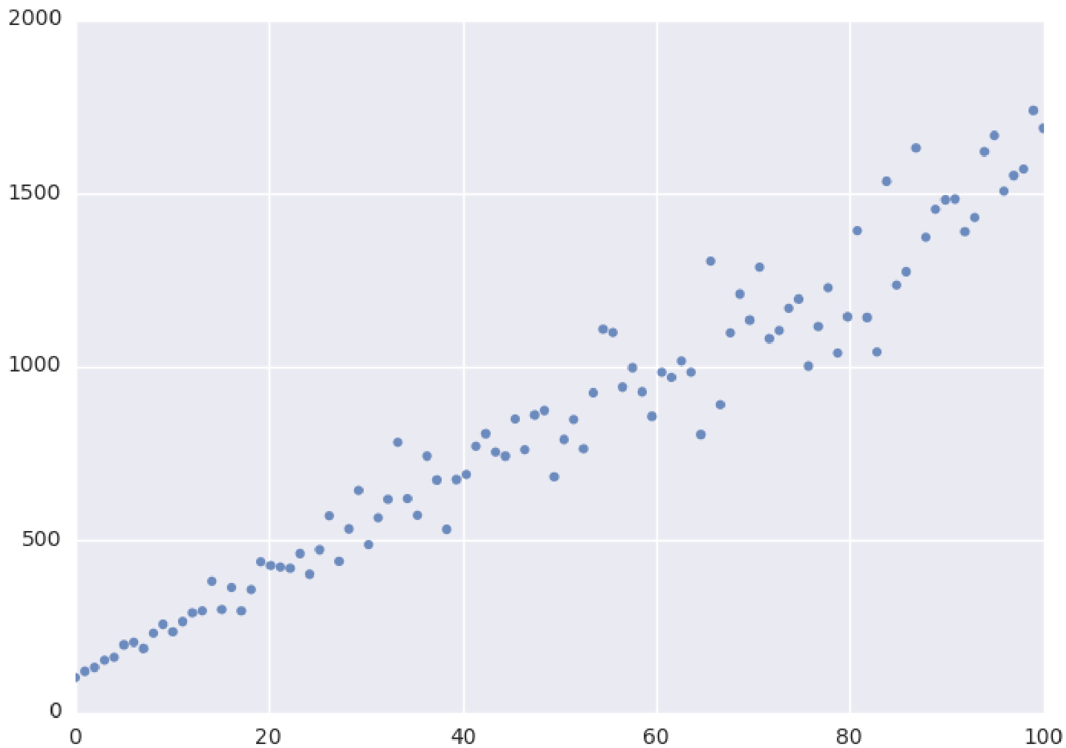
Printing `fit_result` will give a full report on the values for every parameter, including the uncertainty, and quality of the fit.

3.2 Initial Guess

For fitting to work as desired you should always give a good initial guess for a parameter. The `Parameter` object can therefore be initiated with the following keywords:

- `value` the initial guess value.
- `min` Minimal value for the parameter.
- `max` Maximal value for the parameter.
- `fixed` Fix the value of the parameter during the fitting to `value`.

In the example above, we might change our `Parameter`'s to the following after looking at a plot of the data:


2000
1500
1000
500
0
0
20
40
60
80
100

```
k = Parameter(value=4, min=3, max=6)

l, m = parameters('b, c')
l.value = 60
l.fixed = True
```

3.3 Accessing the Results

A call to `Fit.execute()` returns a `FitResults` instance. This object holds all information about the fit. The fitting process does not modify the `Parameter` objects. In the above example, `k.value` will still be `4.0` and not the value we obtain after fitting. To get the value of fit parameters we can do:

```
>>> print(fit_result.params.a)
>>> 14.66946...
>>> print(fit_result.params.a_stdev)
>>> 0.3367571...
>>> print(fit_result.params.b)
>>> 104.6558...
>>> print(fit_result.params.b_stdev)
>>> 19.49172...
>>> print(fit_result.r_squared)
>>> 0.950890866472
```

For more FitResults, see the API docs.

3.4 Evaluating the Model

With these parameters, we could now evaluate the model with these parameters so we can make a plot of it. In order to do this, we simply call the model with these values:

```
import matplotlib.pyplot as plt

y = model(x=xdata, a=fit_result.params.a, b=fit_result.params.b)
plt.plot(xdata, y)
plt.show()
```

The model *has* to be called by keyword arguments to prevent any ambiguity. So the following does not work:

```
y = model(xdata, fit_result.params.a, fit_result.params.b)
```

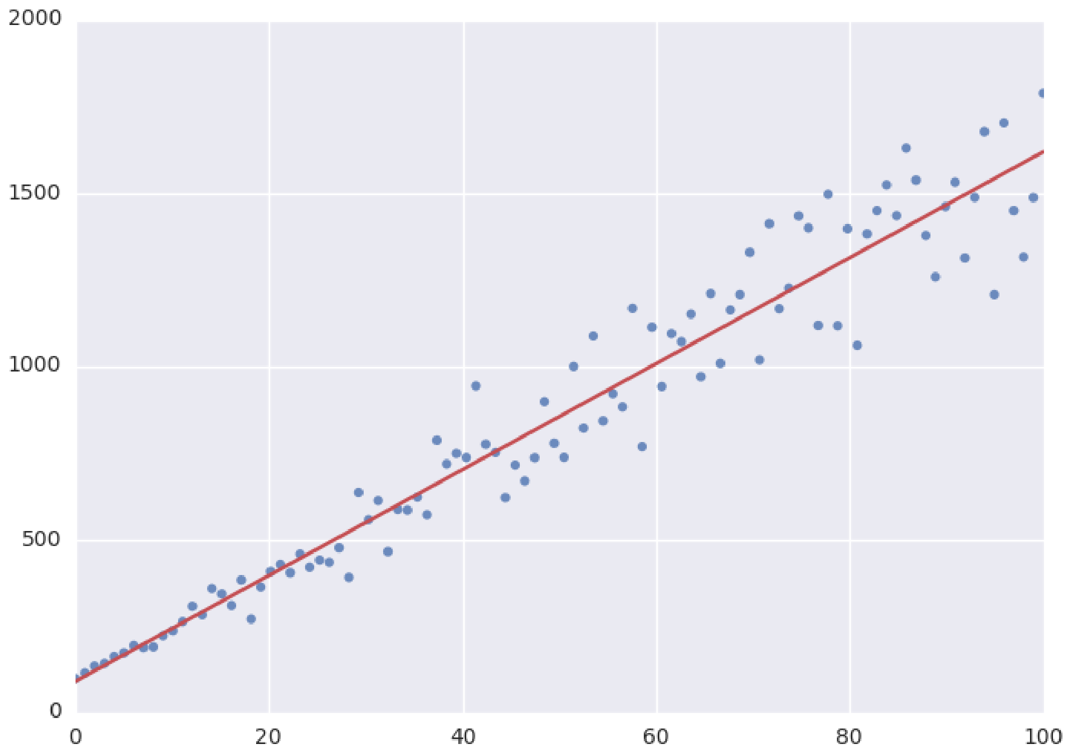
To make life easier, there is a nice shorthand notation to immediately use a fit result:

```
y = model(x=xdata, **fit_result.params)
```

This unpacks the .params object as a dict. For more info view ParameterDict.

3.5 Named Models

More complicated models are also relatively easy to deal with by using named models. Let's try our luck with a bivariate normal distribution:


2000
1500
1000
500
0
0
20
40
60
80
100

```
from symfit import parameters, variables, exp, pi, sqrt

x, y, p = variables('x, y, p')
mu_x, mu_y, sig_x, sig_y, rho = parameters('mu_x, mu_y, sig_x, sig_y, rho')

z = (x - mu_x)**2/sig_x**2 + (y - mu_y)**2/sig_y**2 - 2 * rho * (x - mu_x) * (y - mu_y)/(sig_x * sig_
model = {p: exp(- z / (2 * (1 - rho**2))) / (2 * pi * sig_x * sig_y * sqrt(1 - rho**2))}

fit = Fit(model, x=xdata, y=ydata, p=pdata)
```

By using the magic of named models, the flow of information is still very clear, even with such a complicated function.

This syntax also supports vector valued functions:

```
model = {y_1: a * x**2, y_2: 2 * x * b}
```

One thing to note about such models is that now `model(x=xdata)` obviously no longer works as `type(model) == dict`. There is a prevered way to resolve this. If any kind of fitting object has been initiated, it will have a `.model` atribute containing an instance of `Model`. This can again be called:

```
model = {y_1: a * x**2, y_2: 2 * x * b}
fit = Fit(model, x=xdata)
fit_result = fit.execute()

y_1, y_2 = fit.model(x=xdata, **fit_result.params)
```

This returns a tuple with the components evaluated so through the magic of tuple unpacking‘‘y_1‘‘ and `y_2` contain the evaluated fit. Nice!

If for some reason no `Fit` is initiated you can make a Model object yourself:

```
from symfit import Model

model_dict = {y_1: a * x**2, y_2: 2 * x * b}
model = Model.from_dict(model_dict)

y_1, y_2 = fit.model(x=xdata, a=2.4, b=0.1)
```

3.6 symfit exposes sympy.api

`symfit` exposes the sympy api as well, so mathematical expressions such as `exp`, `sin` and `pi` are importable from `symfit` as well. For more, read the sympy docs.

Fitting Types

4.1 Fit (LeastSquares)

The default fitting object does least-squares fitting:

```
from symfit import parameters, variables, Fit
import numpy as np

# Define a model to fit to.
a, b = parameters('a, b')
x = variables('x')
model = a * x + b

# Generate some data
xdata = np.linspace(0, 100, 100) # From 0 to 100 in 100 steps
a_vec = np.random.normal(15.0, scale=2.0, size=(100,))
b_vec = np.random.normal(100.0, scale=2.0, size=(100,))
ydata = a_vec * xdata + b_vec # Point scattered around the line 5 * x + 105

fit = Fit(model, xdata, ydata)
fit_result = fit.execute()
```

The `Fit` object also supports standard deviations. In order to provide these, it's nicer to use a named model:

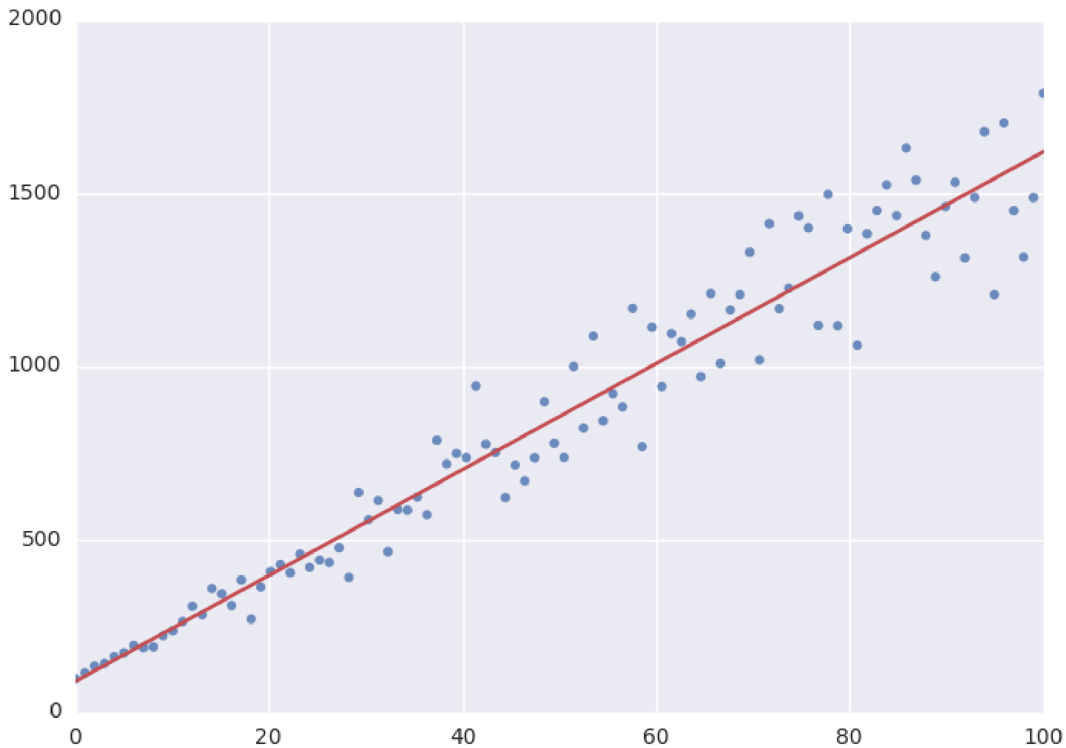
```
a, b = parameters('a, b')
x, y = variables('x, y')
model = {y: a * x + b}

fit = Fit(model, x=xdata, y=ydata, sigma_y=sigma)
```

Symfit assumes these sigma to be from measurement errors by default, and not just as a relative weight. This means the standard deviations on parameters are calculated assuming the absolute size of sigma is significant. This is the case for measurement errors and therefore for most use cases `symfit` was designed for. If you only want to use the sigma for relative weights, then you can use `absolute_sigma=False` as a keyword argument.

Please note that this is the opposite of the convention used by scipy's `curve_fit`. Looking through their mailing list this seems to have been implemented the 'wrong' way for historical reasons, and was understandably never changed so as not to loose backwards compatibility. Since this is a new project, we don't have that problem.

`Fit` currently simply wraps `NumericalLeastSquares`, but might become more intelligent in the future.


2000
1500
1000
500
0
0
20
40
60
80
100

4.2 Likelihood

Given a dataset and a model, what values should the model's parameters have to make the observed data most likely? This is the principle of maximum likelihood and the question the Likelihood object can answer for you.

Example:

```
from symfit import Parameter, Variable, Likelihood, exp
import numpy as np

# Define the model for an exponential distribution (numpy style)
beta = Parameter()
x = Variable()
model = (1 / beta) * exp(-x / beta)

# Draw 100 samples from an exponential distribution with beta=5.5
data = np.random.exponential(5.5, 100)

# Do the fitting!
fit = Likelihood(model, data)
fit_result = fit.execute()
```

Off-course `fit_result` is a normal `FitResults` object. Because `scipy.optimize.minimize` is used to do the actual work, bounds on parameters, and even constraints are supported. For more information on this subject, check out `symfit`'s `Minimize`.

4.3 Minimize/Maximize

Minimize or Maximize a model subject to bounds and/or constraints. It is a wrapper to `scipy.optimize.minimize`. As an example I present an example from the `scipy` docs.

Suppose we want to maximize the following function:

$$f(x, y) = 2xy + 2x - x^2 - 2y^2$$

Subject to the following constraints:

$$x^3 - y = 0$$

$$y - 1 >= 0$$

In SciPy code the following lines are needed:

```
def func(x, sign=1.0):
    """ Objective function """
    return sign*(2*x[0]*x[1] + 2*x[0] - x[0]**2 - 2*x[1]**2)

def func_deriv(x, sign=1.0):
    """ Derivative of objective function """
    dfdx0 = sign*(-2*x[0] + 2*x[1] + 2)
    dfdx1 = sign*(2*x[0] - 4*x[1])
    return np.array([ dfdx0, dfdx1 ])

cons = ({'type': 'eq',
        'fun' : lambda x: np.array([x[0]**3 - x[1]]),
        'jac' : lambda x: np.array([3.0*(x[0]**2.0), -1.0])},
```

```
        {'type': 'ineq',
        'fun' : lambda x: np.array([x[1] - 1]),
        'jac' : lambda x: np.array([0.0, 1.0])})

res = minimize(func, [-1.0,1.0], args=(-1.0,), jac=func_deriv,
               constraints=cons, method='SLSQP', options={'disp': True})
```

Takes a couple of read-throughs to make sense, doesn't it? Let's do the same problem in `symfit`:

```
from symfit import parameters, Maximize, Eq, Ge

x, y = parameters('x, y')
model = 2*x*y + 2*x - x**2 -2*y**2
constraints = [
    Eq(x**3 - y, 0),
    Ge(y - 1, 0),
]

fit = Maximize(model, constraints=constraints)
fit_result = fit.execute()
```

Done! `symfit` will determine all derivatives automatically, no need for you to think about it.

> **Warning:** You might have noticed that `x` and `y` are `Parameter`'s in the above problem, which may strike you as weird.

However, it makes perfect sense because in this problem they are parameters to be optimised, not variables. Furthermore, this way of defining it is consistent with the treatment of `Variable`'s and `Parameter`'s in `symfit`. Be aware of this when using `Minimize`, as the whole process won't work otherwise.

4.4 How Does Fit Work?

How does `Fit` get from a (named) model and some data to a fit? Consider the following example:

```
from symfit.api import parameters, variables, Fit

a, b = parameters('a, b')
x, y = variables('x, y')
model = {y: a * x + b}

fit = Fit(model, x=x_data, y=y_data, sigma_y=sigma_data)
fit_result = fit.execute()
```

The first thing `symfit` does is build χ^2 for your model:

```
chi_squared = sum((y - f)**2/sigmas[y]**2 for y, f in model.items())
```

In this line `sigmas` is a dict which contains all vars that where given a value, or returns 1 otherwise.

This χ^2 is then transformed into a python function which can then be used to do the numerical calculations:

```
vars, params = seperate_symbols(chi_squared)
py_chi_squared = lambdify(vars + params, chi_squared)
```

We are now almost there. Just two steps left. The first is to wrap all the data into the `py_chi_squared` function using `partial` into the function to be optimized:

```
from functools import partial

error = partial(py_chi_squared, **data_per_var)
```

where `data_per_var` is a dict containing variable names: value pairs.

Now all that is left is to call `leastsqbound` and have it find the best fit parameters:

```
best_fit_parameters, covariance_matrix = leastsqbound(
  error,
  self.guesses,
  self.eval_jacobian,
  self.bounds,
)
```

That's it! Finally there are some steps to generate a FitResult object, but these are not important for our current discussion.

4.5 What if the model is unnamed?

Then you'll have to use the ordering. Variables throughout `symfit`'s objects are internally ordered in the following way: first independent variables, then dependent variables, then sigma variables, and lastly parameters when applicable. Within each group alphabetical ordering applies.

It is therefore always possible to assign data to variables in an unambiguis way using this ordering. In the above example:

```
fit = Fit(model, x_data, y_data, sigma_data)
```

Technical Notes

Essays on mathematical and implementation details.

5.1 On Likelihood Fitting

The *Likelihood* object is a subclass of *Maximize*. The *error_func* and *eval_jacobian* definitions have been changed to facilitate what one would expect from Likelihood fitting:

error_func gives the value of log-likelihood at the given values of $\vec{p}$ and $\vec{x}_i$, where $\vec{p}$ is a shorthand notation for all parameter, and $\vec{x}_i$ the same shorthand for all independent variables.

$$\log L(\vec{p}|\vec{x}_i) = \sum_{i=1}^{N} \log f(\vec{p}|\vec{x}_i)$$

eval_jacobian gives the derivative with respect to every parameter of the log-likelihood:

$$\nabla_{\vec{p}} \log L(\vec{p}|\vec{x}_i) = \sum_{i=1}^{N} \frac{1}{f(\vec{p}|\vec{x}_i)} \nabla_{\vec{p}} f(\vec{p}|\vec{x}_i)$$

Where $\nabla_{\vec{p}}$ is the derivative with respect to all parameters $\vec{p}$. The function therefore returns a vector of length `len(p)` containing the Jacobian evaluated at the given values of $\vec{p}$ and $\vec{x}$.

5.2 On Standard Deviations

This essay is meant as a reflection on the implementation of Standard Deviations and/or measurement errors in `symfit`. Although reading this essay in it's entirely will only be interesting to a select few, I urge anyone who uses `symfit` to read the following summarizing bullet points, as `symfit` is NOT backward-compatible with `scipy`.

- standard deviations are assumed to be measurement errors by default, not relative weights. This is the opposite of the `scipy` definition. Set `absolute_sigma=False` when calling `Fit` to get the `scipy` behavior.

5.2.1 Analytical Example

The implementation of standard deviations should be in agreement with cases to which the analytical solution is known. `symfit` was build such that this is true. Let's follow the example outlined by [taldcroft]. We'll be sampling from a normal distribution with $\mu = 0.0$ and varying σ. It can be shown that given a sample from such a distribution:

$$\mu = 0.0$$

$$\sigma_\mu = \frac{\sigma}{\sqrt{N}}$$

where N is the size of the sample. We see that the error in the sample mean scales with the σ of the distribution.

In order to reproduce this with `symfit`, we recognize that determining the avarage of a set of numbers is the same as fitting to a constant. Therefore we will fit to samples generated from distributions with $\sigma = 1.0$ and $\sigma = 10.0$ and check if this matches the analytical values. Let's set $N = 10000$.

```
N = 10000
sigma = 10.0
np.random.seed(10)
yn = np.random.normal(size=N, scale=sigma)

a = Parameter('a')
y = Variable('y')
model = {y: a}

fit = Fit(model, y=yn, sigma_y=sigma)
fit_result = fit.execute()

fit_no_sigma = Fit(model, y=yn)
fit_result_no_sigma = fit_no_sigma.execute()
```

This gives the following results:

- a = 5.102056e-02 +- 1.000000e-01 when `sigma_y` is provided. This matches the analytical prediction.
- a = 5.102056e-02 +- 9.897135e-02 without `sigma_y` provided. This is incorrect.

If we run the above code example with `sigma = 1.0`, we get the following results:

- a = 5.102056e-03 +- 9.897135e-03 when `sigma_y` is provided. This matches the analytical prediction.
- a = 5.102056e-03 +- 9.897135e-03 without `sigma_y` provided. This is also correct, since providing no weights is the same as setting the weights to 1.

To conclude, if `symfit` is provided with the standard deviations, it will give the expected result by default. As shown in [taldcroft] and `symfit`'s `tests.py`, `scipy.optimize.curve_fit` has to be provided with the `absolute_sigma=True` setting to do the same.

Important: We see that even if the weight provided to every data point is the same, the *scale* of the weight still effects the result. `scipy` was build such that the opposite is true: if all datapoints have the same weight, the error in the parameters does not depend on the scale of the weight.

This difference is due to the fact that `symfit` is build for area's of science where one is dealing with measurement errors. And with measurement errors, the size of the errors obviously matters for the certainty of the fit parameters, even if the errors are the same for every measurement.

If you want the `scipy` behavior, initiate `Fit` with `absolute_sigma=False`.

5.3 Comparison to Mathematica

In Mathematica, the default setting is also to use relative weights, which we just argued is not correct when dealing with measurement errors. In [mathematica] this problem is discussed very nicely, and it is shown how to solve this in Mathematica.

Since `symfit` is a fitting tool for the practical man, measurement errors are assumed by default.

Dependencies and Credits

Always pay credit where credit's due. `symfit` uses the following projects to make it's sexy interface possible:

- leastsqbound-scipy is used to bound parameters to a given domain.
- seaborn was used to make the beautifully styled plots in the example code. All you have to do to sexify your matplotlib plot's is import seaborn, even if you don't use it's special plotting facilities, so I highly recommend it.
- numpy and scipy are of course used to do efficient data processing.
- sympy is used for the manipulation of the symbolic expressions that give this project it's high readability.

CHAPTER 7

Module Documentation

This page contains documentation to everything `symfit` has to offer.

7.1 BaseFit

class `symfit.core.fit.`**`BaseFit`**(*model*, **ordered_data*, *absolute_sigma=None*, ***named_data*)

Bases: `object`

Abstract Base Class for all fitting objects. Most importantly, it takes care of linking the provided data to variables. The allowed variables are extracted from the model.

`__init__`(*model*, **ordered_data*, *absolute_sigma=None*, ***named_data*)

Parameters

- **`model`** – (dict of) sympy expression or `Model` object.
- **`bool`** (*absolute_sigma*) – True by default. If the sigma is only used for relative weights in your problem, you could consider setting it to False, but if your sigma are measurement errors, keep it at True. Note that curve_fit has this set to False by default, which is wrong in experimental science.
- **`ordered_data`** – data for dependent, independent and sigma variables. Assigned in the following order: independent vars are assigned first, then dependent vars, then sigma's in dependent vars. Within each group they are assigned in alphabetical order.
- **`named_data`** – assign dependent, independent and sigma variables data by name.

Standard deviation can be provided to any variable. They have to be prefixed with **sigma_**. For example, let x be a Variable. Then sigma_x will give the stdev in x.

`__weakref__`

list of weak references to the object (if defined)

`dependent_data`

Read-only Property

Returns Data belonging to each dependent variable.

Return type dict with variable names as key, data as value.

`execute`(**args*, ***kwargs*) → symfit.core.fit.FitResults

Every fit object has to define an execute method. Any * and ** arguments will be passed to the fitting module that is being wrapped, e.g. leastsq.

Args kwargs

Returns Instance of FitResults

independent_data
Read-only Property

Returns Data belonging to each independent variable.

Return type dict with variable names as key, data as value.

initial_guesses

Returns Initial guesses for every parameter.

sigma_data
Read-only Property

Returns Data belonging to each sigma variable.

Return type dict with variable names as key, data as value.

7.2 Fit

class symfit.core.fit.**BaseFit**(*model*, **ordered_data*, *absolute_sigma=None*, ***named_data*)
Bases: object

Abstract Base Class for all fitting objects. Most importantly, it takes care of linking the provided data to variables. The allowed variables are extracted from the model.

__init__(*model*, **ordered_data*, *absolute_sigma=None*, ***named_data*)

Parameters

- **model** – (dict of) sympy expression or Model object.
- **bool** (*absolute_sigma*) – True by default. If the sigma is only used for relative weights in your problem, you could consider setting it to False, but if your sigma are measurement errors, keep it at True. Note that curve_fit has this set to False by default, which is wrong in experimental science.
- **ordered_data** – data for dependent, independent and sigma variables. Assigned in the following order: independent vars are assigned first, then dependent vars, then sigma's in dependent vars. Within each group they are assigned in alphabetical order.
- **named_data** – assign dependent, independent and sigma variables data by name.

Standard deviation can be provided to any variable. They have to be prefixed with **sigma_**. For example, let x be a Variable. Then sigma_x will give the stdev in x.

__weakref__
list of weak references to the object (if defined)

dependent_data
Read-only Property

Returns Data belonging to each dependent variable.

Return type dict with variable names as key, data as value.

execute(**args*, ***kwargs*) → symfit.core.fit.FitResults
Every fit object has to define an execute method. Any * and ** arguments will be passed to the fitting module that is being wrapped, e.g. leastsq.

Args kwargs

Returns Instance of FitResults

independent_data
Read-only Property

Returns Data belonging to each independent variable.

Return type dict with variable names as key, data as value.

initial_guesses

Returns Initial guesses for every parameter.

sigma_data
Read-only Property

Returns Data belonging to each sigma variable.

Return type dict with variable names as key, data as value.

class symfit.core.fit.**Constraint**(*constraint: sympy.core.relational.Relational*, *model: symfit.core.fit.Model*)
Bases: *symfit.core.fit.Model*

Constraints are a special type of model in that they have a type: >=, == etc. They are made to have lhs - rhs == 0 of the original expression.

For example, Eq(y + x, 4) -> Eq(y + x - 4, 0)

Since a constraint belongs to a certain model, it has to be initiated with knowledge of it's parent model. This is important because all numerical_ methods are done w.r.t. the parameters and variables of the parent model, not the constraint! This is because the constraint might not have all the parameter or variables that the model has, but in order to compute for example the Jacobian we still want to derive w.r.t. all the parameters, not just those present in the constraint.

__init__(*constraint: sympy.core.relational.Relational*, *model: symfit.core.fit.Model*)

Parameters

- **constraint** – constraint that model should be subjected to.
- **model** – A constraint is always tied to a model.

Returns

constraint_type
alias of Equality

jacobian

Returns Jacobian 'Matrix' filled with the symbolic expressions for all the partial derivatives. Partial derivatives are of the components of the function with respect to the Parameter's, not the independent Variable's.

numerical_components

Returns lambda functions of each of the components in model_dict, to be used in numerical calculation.

numerical_jacobian

Returns lambda functions of the jacobian matrix of the function, which can be used in numerical optimization.

class symfit.core.fit.**Fit**(*model*, **ordered_data*, *absolute_sigma=None*, ***named_data*)
Bases: *symfit.core.fit.NumericalLeastSquares*

Wrapper for NumericalLeastSquares to give it a more appealing name. In the future I hope to make this object more intelligent so it can search out the best fitting object based on certain qualifiers and return that instead.

class `symfit.core.fit.`**`FitResults`**(*params*, *popt*, *pcov*, *infodic*, *mesg*, *ier*, *ydata=None*, *sigma=None*)

Bases: `object`

Class to display the results of a fit in a nice and unambiguous way. All things related to the fit are available on this class, e.g. - parameters + stdev - R squared (Regression coefficient.) - fitting status message

This object is made to behave entirely read-only. This is a bit unnatural to enforce in Python but I feel it is necessary to guarantee the integrity of the results.

`__init__`(*params*, *popt*, *pcov*, *infodic*, *mesg*, *ier*, *ydata=None*, *sigma=None*)

Excuse the ugly names of most of these variables, they are inherited. Should be changed. from scipy. :param params: list of `Parameter`'s. :param popt: best fit parameters, same ordering as in params. :param pcov: covariance matrix. :param infodic: dict with fitting info. :param mesg: Status message. :param ier: Number of iterations. :param ydata:

`__str__`()

Pretty print the results as a table. :return:

`__weakref__`

list of weak references to the object (if defined)

`infodict`

Read-only Property.

`iterations`

Read-only Property.

`params`

Read-only Property.

`r_squared`

r_squared Property.

Returns Regression coefficient.

`status_message`

Read-only Property.

class `symfit.core.fit.`**`Likelihood`**(*model*, **args*, *constraints=None*, ***kwargs*)

Bases: `symfit.core.fit.Maximize`

Fit using a Maximum-Likelihood approach.

`error_func`(*p*, *data*)

Error function to be maximised(!) in the case of likelihood fitting.

Parameters

- **`p`** – guess params
- **`data`** – xdata

Returns scalar value of log-likelihood

`eval_jacobian`(*p*, *data*)

Jacobian for likelihood is defined as $\nabla_{\vec{p}}(\log(L(\vec{p}|\vec{x})))$.

Parameters

- **`p`** – guess params
- **`data`** – data for the variables.

Returns array of length number of `Parameter`'s in the model, with all partial derivatives evaluated at p, data.

class `symfit.core.fit.`**`Maximize`**(*model*, **args*, *constraints=None*, ***kwargs*)
Bases: *`symfit.core.fit.Minimize`*

Maximize a model subject to constraints. Simply flips the sign on error_func and eval_jacobian in order to maximize.

class `symfit.core.fit.`**`Minimize`**(*model*, **args*, *constraints=None*, ***kwargs*)
Bases: *`symfit.core.fit.BaseFit`*

Minimize a model subject to constraints. A wrapper for `scipy.optimize.minimize`. `Minimize` currently doesn't work when data is provided to Variables, and doesn't support vector functions.

`__init__`(*model*, **args*, *constraints=None*, ***kwargs*)
Because in a lot of use cases for Minimize no data is supplied to variables, all the empty variables are replaced by an empty np array.

Constraints constraints the minimization is subject to.

`error_func`(*p*, *data*)
The function to be optimized. Scalar valued models are assumed. For Minimize the thing to evaluate is simply self.model(*(list(data) + list(p)))

Parameters

- **`p`** – array of floats for the parameters.
- **`data`** – data to be provided to `Variable`'s.

`eval_jacobian`(*p*, *data*)
Takes partial derivatives of model w.r.t. each `Parameter`.

Parameters

- **`p`** – array of floats for the parameters.
- **`data`** – data to be provided to `Variable`'s.

Returns array of length number of `Parameter`'s in the model, with all partial derivatives evaluated at p, data.

`scipy_constraints`
Read-only Property of all constraints in a scipy compatible format.

Returns dict of scipy compatible statements.

class `symfit.core.fit.`**`Model`**(**ordered_expressions*, ***named_expressions*)
Bases: `object`

Model represents a symbolic function and all it's derived properties such as sum of squares, jacobian etc. Models can be initiated from several objects:

```
a = Model.from_dict({y: x**2})
b = Model(y=x**2)
```

Models are callable. The usual rules apply to the ordering of the arguments:

- first independent variables, then dependent variables, then parameters.
- within each of these groups they are ordered alphabetically.

`__call__`(**args*, ***kwargs*)
Evaluate the model for a certain value of the independent vars and parameters. Signature for this function contains independent vars and parameters, NOT dependent and sigma vars.

Can be called with both ordered and named parameters. Order is independent vars first, then parameters. Alphabetical order within each group.

Parameters

- **args** –
- **kwargs** –

Returns A namedtuple of all the dependent vars evaluated at the desired point. Will always return a tuple, even for scalar valued functions. This is done for consistency.

__init__(**ordered_expressions*, ***named_expressions*)

Initiate a Model from keyword arguments:

```
b = Model(y=x**2)
```

Parameters

- **ordered_expressions** – sympy Expr
- **named_expressions** – sympy Expr

__str__()

Pretty print this model.

Returns str

__weakref__

list of weak references to the object (if defined)

bounds

Returns List of tuples of all bounds on parameters.

chi

Returns Symbolic Square root of χ^2. Required for MINPACK optimization only. Denoted as $\sqrt(\chi^2)$

chi_jacobian

Return a symbolic jacobian of the $\sqrt(\chi^2)$ function. Vector of derivatives w.r.t. each parameter. Not a Matrix but a vector! This is because that's what leastsq needs.

chi_squared

Returns Symbolic χ^2

classmethod **from_dict**(*model_dict*)

Initiate a Model from a dict:

```
a = Model.from_dict({y: x**2})
```

Prefered syntax.

Parameters **model_dict** – dict of `Expr`, where dependent variables are the keys.

jacobian

Returns Jacobian 'Matrix' filled with the symbolic expressions for all the partial derivatives.

Partial derivatives are of the components of the function with respect to the Parameter's, not the independent Variable's.

numerical_chi

Returns lambda function of the `.chi` method, to be used in MINPACK optimisation.

numerical_chi_jacobian

Returns lambda functions of the jacobian of the `.chi` method, which can be used in numerical optimization.

numerical_chi_squared

Returns lambda function of the `.chi_squared` method, to be used in numerical optimisation.

numerical_components

Returns lambda functions of each of the components in model_dict, to be used in numerical calculation.

numerical_jacobian

Returns lambda functions of the jacobian matrix of the function, which can be used in numerical optimization.

ss_res

Returns Residual sum of squares. Similar to chi_squared, but without considering weights.

vars

Returns Returns a list of dependent, independent and sigma variables, in that order.

class `symfit.core.fit.`**NumericalLeastSquares**(*model*, **ordered_data*, *absolute_sigma=None*, ***named_data*)

Bases: *symfit.core.fit.BaseFit*

Solves least squares numerically using leastsqbounds. Gives results consistent with MINPACK except when borders are provided.

execute(**options*, ***kwoptions*) → symfit.core.fit.FitResults

Parameters

- **options** – Any postional arguments to be passed to leastsqbound
- **kwoptions** – Any named arguments to be passed to leastsqbound

class `symfit.core.fit.`**ParameterDict**(*params*, *popt*, *pcov*, **args*, ***kwargs*)

Bases: `object`

Container for all the parameters and their (co)variances. Behaves mostly like an OrderedDict: can be **-ed, allowing the sexy syntax where a model is called with values for the Variables and **params. However, under iteration it behaves like a list! In other words, it preserves order in the params.

__getattr__(*name*)

A user can access the value of a parameter directly through this object.

Parameters **name** – Name of a `Parameter`. Naming convention: let a = Parameter(). Then: .a gives the value of the parameter. .a_stdev gives the standard deviation.

__getitem__(*param_name*)

This method allows this object to be addressed as a dict. This allows for the ** unpacking. Therefore return the value of the best fit parameter, as this is what the user expects.

Parameters **param_name** – Name of the `Parameter` whose value your interested in.

Returns the value of the best fit parameter with name 'key'.

__iter__()

Iteration over the `Parameter` instances. :return: iterator

__len__()

Length gives the number of `Parameter` instances.

Returns len(self.__params)

__weakref__

list of weak references to the object (if defined)

get_stdev(*param*)

Parameters **param** – `Parameter` instance.

Returns returns the standard deviation of param

get_value(*param*)

Parameters **param** – `Parameter` instance.

Returns returns the numerical value of param

keys()

Returns All `Parameter` names.

`symfit.core.fit.`**r_squared**(*model: symfit.core.fit.Model*, *fit_result: symfit.core.fit.FitResults*, *data: collections.OrderedDict*) → float

Calculates the coefficient of determination, R^2, for the fit.

7.3 Argument

class `symfit.core.argument.`**Argument**(*name=None*, **sympy_args*, ***sympy_kwargs*)

Bases: `sympy.core.symbol.Symbol`

Base class for `symfit` symbols. This helps make `symfit` symbols distinguishable from `sympy` symbols.

The `Argument` class also makes DRY possible in defining `Argument`'s: it uses `inspect` to read the lhs of the assignment and uses that as the name for the `Argument` is none is explicitly set.

For example:

```
x = Variable()
print(x.name)
>> 'x'
```

__weakref__

list of weak references to the object (if defined)

class `symfit.core.argument.`**Parameter**(*value=1.0*, *min=None*, *max=None*, *fixed=False*, *name=None*, **sympy_args*, ***sympy_kwargs*)

Bases: *symfit.core.argument.Argument*

Parameter objects are used to facilitate bounds on function parameters.

__call__(***values*)

Parameters

- **self** – Any subclass of sympy.Expr
- **values** – Values for the Parameters and Variables of the Expr.

Returns The function evaluated at `values`. Depending on the Expr and `values`, this could be a single number or an array.

__init__(*value=1.0*, *min=None*, *max=None*, *fixed=False*, *name=None*, **sympy_args*, ***sympy_kwargs*)

Parameters

- **value** – Initial guess value.
- **min** – Lower bound on the parameter value.
- **max** – Upper bound on the parameter value.
- **fixed** (*bool*) – Fix the parameter to `value` during fitting.
- **name** – Name of the Parameter.
- **sympy_args** – Args to pass to `sympy`.
- **sympy_kwargs** – Kwargs to pass to `sympy`.

class `symfit.core.argument.`**Variable**(*name=None*, **sympy_args*, ***sympy_kwargs*)

Bases: `symfit.core.argument.Argument`

Variable type.

7.4 Operators

This module makes sympy Epressions callable, which makes the whole project feel more consistent.

`symfit.core.operators.`**call**(*self*, ***values*)

Parameters

- **self** – Any subclass of sympy.Expr
- **values** – Values for the Parameters and Variables of the Expr.

Returns The function evaluated at `values`. Depending on the Expr and `values`, this could be a single number or an array.

7.5 Support

This module contains support functions and convenience methods used throughout symfit. Some are used predominantly internally, others are designed for users.

`symfit.core.support.`**cache**(*func*)

Decorator function that gets a method as its input and either buffers the input, or returns the buffered output. Used in conjunction with properties to take away the standard buffering logic.

Parameters **func** –

Returns

`symfit.core.support.`**jacobian**(*expr*, *symbols*)

Derive a symbolic expr w.r.t. each symbol in symbols. This returns a symbolic jacobian vector.

Parameters

- **expr** – A sympy Expr.
- **symbols** – The symbols w.r.t. which to derive.

symfit.core.support.**parameters**(*names*)
Convenience function for the creation of multiple parameters.

Parameters names – string of parameter names. Should be comma seperated. Example: a, b = parameters('a, b')

symfit.core.support.**seperate_symbols**(*func*)
Seperate the symbols in symbolic function func. Return them in alphabetical order.

Parameters func – scipy symbolic function.

Returns (vars, params), a tuple of all variables and parameters, each sorted in alphabetical order.

Raises TypeError only symfit Variable and Parameter are allowed, not sympy Symbols.

symfit.core.support.**sympy_to_py**(*func*, *vars*, *params*)
Turn a symbolic expression into a Python lambda function, which has the names of the variables and parameters as it's argument names.

Parameters

- **func** – sympy expression
- **vars** – variables in this model
- **params** – parameters in this model

Returns lambda function to be used for numerical evaluation of the model. Ordering of the arguments will be vars first, then params.

symfit.core.support.**sympy_to_scipy**(*func*, *vars*, *params*)
Convert a symbolic expression to one scipy digs. Not used by `symfit` any more.

Parameters

- **func** – sympy expression
- **vars** – variables
- **params** – parameters

Returns Scipy-style function to be used for numerical evaluation of the model.

symfit.core.support.**variables**(*names*)
Convenience function for the creation of multiple variables.

Parameters names – string of variable names. Should be comma seperated. Example: x, y = variables('x, y')

7.6 Distributions

Some common distributions are defined in this module. That way, users can easily build more complicated expressions without making them look hard.

I have deliberately chosen to start these function with a capital, e.g. Gaussian instead of gaussian, because this makes the resulting expressions more readable.

symfit.distributions.**Exp**(*x*, *l*)
Exponential Distribution pdf. :param x: free variable. :param l: rate parameter. :return: sympy.Expr for an Exponential Distribution pdf.

`symfit.distributions.`**`Gaussian`**(*x*, *mu*, *sig*)

Gaussian pdf. :param x: free variable. :param mu: mean of the distribution. :param sig: standard deviation of the distribution. :return: sympy.Expr for a Gaussian pdf.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

[taldcroft] http://nbviewer.ipython.org/urls/gist.github.com/taldcroft/5014170/raw/31e29e235407e4913dc0ec403af7ed524372b612/cur

[mathematica] http://reference.wolfram.com/language/howto/FitModelsWithMeasurementErrors.html

S

Index

Symbols

A

B

C

D

E

F

G

I

J

K

L

M