symfit Documentation

Release 0.4.1

tBuLi

Sep 08, 2018

Contents

Contents:

CHAPTER 1

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

To define models to fit to:

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

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

And finally, to execute the fit:

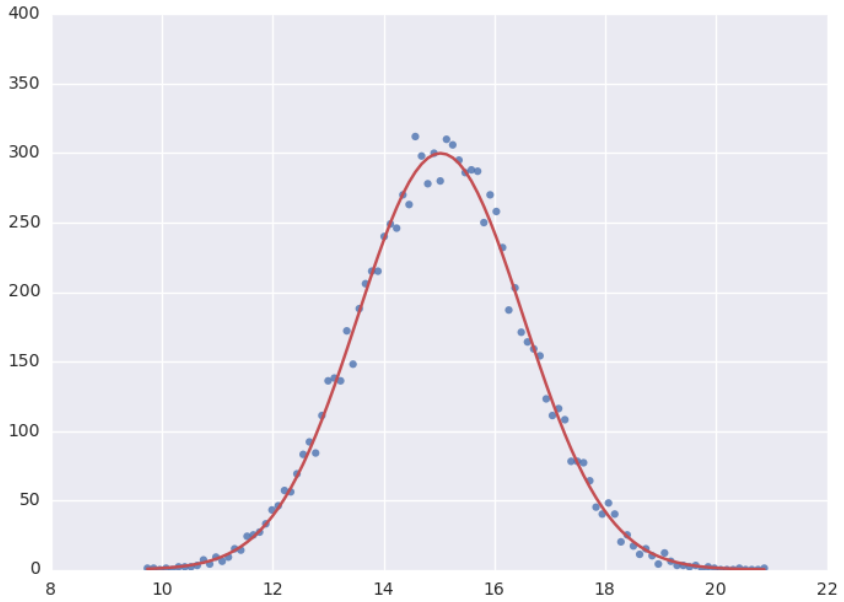
```
fit = Fit(model, xdata, ydata)
fit_result = fit.execute()
```

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: >=
]

fit = Fit(- model, constraints=constraints)
fit_result = fit.execute()
```

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.

CHAPTER 2

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 Contrib module

To also install the dependencies of 3rd party contrib modules such as interactive guesses, install *symfit* using:

```
pip install symfit[contrib]
```

2.2 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 import Parameter, Variable

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

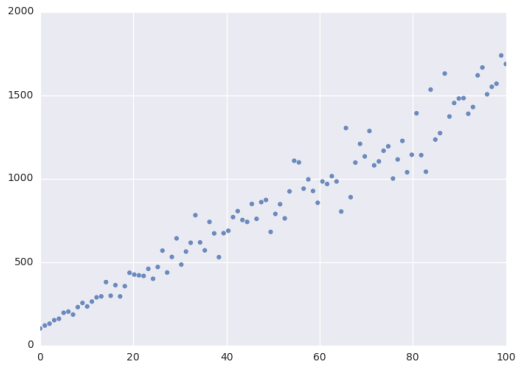
Lets fit this model to some generated data.

```
from symfit 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. Defaults to `1`.
- `min` Minimal value for the parameter.
- `max` Maximal value for the parameter.
- `fixed` Whether the parameter's `value` can vary during fitting.

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

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

a, b = parameters('a, b')
a.value = 60
a.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, `a.value` will still be `60` and not the value we obtain after fitting. To get the value of fit parameters we can do:

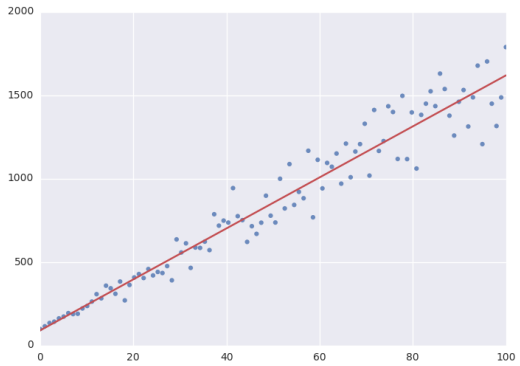
```
>>> print(fit_result.value(a))
>>> 14.66946...
>>> print(fit_result.stdev(a))
>>> 0.3367571...
>>> print(fit_result.value(b))
>>> 104.6558...
>>> print(fit_result.stdev(b))
>>> 19.49172...
>>> print(fit_result.r_squared)
>>> 0.950890866472
```

For more *`FitResults`*, see the *Module Documentation*.

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.value(a), b=fit_result.value(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.value(a), fit_result.value(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 immediately unpacks an `OrderedDict` containing the optimized fit parameters.

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:

```
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_y)
)
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 relatively 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 preferred 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:

```
a, b = parameters('a, b')
y_1, y_2, x = variables('y_1, y_2, x')

model = {y_1: a * x**2, y_2: 2 * x * b}
fit = Fit(model, x=xdata, y_1=y_data1, y_2=y_data2)
fit_result = fit.execute()

y_1_result, y_2_result = fit.model(x=xdata, **fit_result.params)
```

This returns a `namedtuple()`, with the components evaluated. So through the magic of tuple unpacking, `y_1` and `y_2` contain the evaluated fit. The dependent variables will be ordered alphabetically in the returned `namedtuple()`. Alternatively, the unpacking can be performed explicitly.

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

```
model = Model(model_dict)
y_1_result, y_2_result = model(x=xdata, a=2.4, b=0.1)
```

or equivalently:

```
outcome = model(x=xdata, a=2.4, b=0.1)
y_1_result = outcome.y_1
y_2_result = outcome.y_2
```

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.

CHAPTER 4

Fitting Types

4.1 Fit (Least Squares)

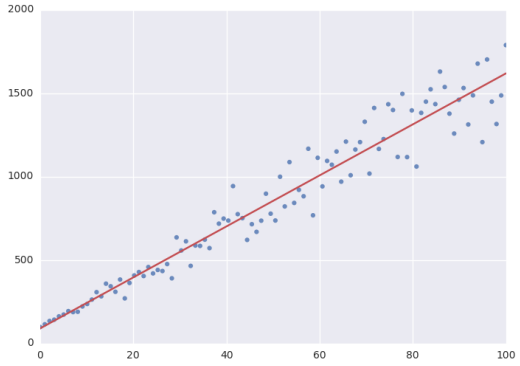
The default fitting object does least-squares fitting:

```
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,))
# Point scattered around the line 5 * x + 105
ydata = a_vec * xdata + b_vec

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)
```

Warning: 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 opposite 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.

4.2 Constrained Least Squares Fit

The *Fit* takes a `constraints` keyword; a list of relationships between the parameters that has to be respected. As an example of fitting with constraints, we could imagine fitting the angles of a triangle:

```
a, b, c = parameters('a, b, c')
a_i, b_i, c_i = variables('a_i, b_i, c_i')

model = {a_i: a, b_i: b, c_i: c}

data = np.array([
    [10.1, 9., 10.5, 11.2, 9.5, 9.6, 10.],
    [102.1, 101., 100.4, 100.8, 99.2, 100., 100.8],
    [71.6, 73.2, 69.5, 70.2, 70.8, 70.6, 70.1],
])

fit = Fit(
    model=model,
    a_i=data[0],
    b_i=data[1],
    c_i=data[2],
    constraints=[Equality(a + b + c, 180)]
)
fit_result = fit.execute()
```

The line `constraints=[Equality(a + b + c, 180)]` ensures the our basic knowledge of geometry is respected despite my sloppy measurements.

Note: Under the hood, a different *Minimizer* is used to perform a constrained fit. *Fit* tries to select the right

Minimizer based on the problem you present it with. See *Fit* for more.

4.3 (Non)LinearLeastSquares

The *LinearLeastSquares* implements the analytical solution to Least Squares fitting. When your model is linear in it's parameters, consider using this rather than the default *Fit* since this gives the exact solution in one step, no iteration and no guesses needed.

NonLinearLeastSquares is the generalization to non-linear models. It works by approximating the model by a linear one around the value of your guesses and repeating that process iteratively. This process is therefore very sensitive to getting good initial guesses.

Notes on these objects:

- Use *NonLinearLeastSquares* instead of *LinearLeastSquares* unless you have a reason not to. *NonLinearLeastSquares* will behave exactly the same as *LinearLeastSquares* when the model is linear.
- Bounds are currently ignored by both. This is because for linear models there can only be one solution. For non-linear models it simply hasn't been considered yet.
- When performance matters, use *Fit* instead of *NonLinearLeastSquares*. These analytical objects are implemented in pure python and are therefore massively outgunned by *Fit* which is ultimately a wrapper to efficient numerical methods such as MINPACK of BFGS implemented in Fortran.

4.4 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, exp
from symfit.core.objectives import LogLikelihood
import numpy as np

# Define the model for an exponential distribution (numpy style)
beta = Parameter('beta')
x = Variable('x')
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 = Fit(model, data, objective=LogLikelihood)
fit_result = fit.execute()
```

`fit_result` is a normal *FitResults* object. As always, bounds on parameters and even constraints are supported.

4.5 Minimize/Maximize

Minimize or Maximize a model subject to bounds and/or constraints. 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 = Fit(- model, constraints=constraints)
fit_result = fit.execute()
```

Done! `symfit` will determine all derivatives automatically, no need for you to think about it. Notice the minus sign in the call to *Fit*. This is because *Fit* will always minimize, so in order to achieve maximization we should minimize -*model*.

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 independent 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 minimizing such problems, as the whole process won't work otherwise.

4.6 ODE Fitting

Fitting to a system of ordinary differential equations (ODEs) is also remarkedly simple with `symfit`. Let's do a simple example from reaction kinetics. Suppose we have a reaction A + A -> B with rate constant k. We then need the following system of rate equations:

$$\frac{dA}{dt} = -kA^2$$
$$\frac{dB}{dt} = kA^2$$

In `symfit`, this becomes:

```
model_dict = {
    D(a, t): - k * a**2,
    D(b, t): k * a**2,
}
```

We see that the `symfit` code is already very readable. Let's do a fit to this:

```
tdata = np.array([10, 26, 44, 70, 120])
adata = 10e-4 * np.array([44, 34, 27, 20, 14])
a, b, t = variables('a, b, t')
k = Parameter('k', 0.1)
a0 = 54 * 10e-4

model_dict = {
    D(a, t): - k * a**2,
    D(b, t): k * a**2,
}

ode_model = ODEModel(model_dict, initial={t: 0.0, a: a0, b: 0.0})

fit = Fit(ode_model, t=tdata, a=adata, b=None)
fit_result = fit.execute()
```

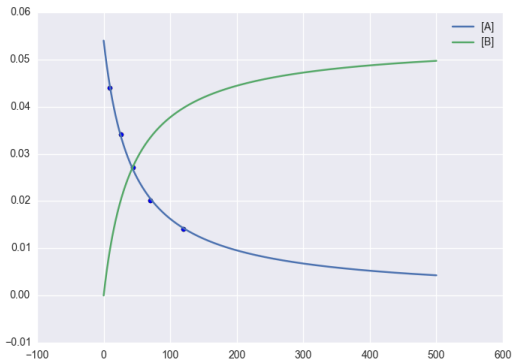
That's it! An *`ODEModel`* behaves just like any other model object, so *`Fit`* knows how to deal with it! Note that since we don't know the concentration of B, we explicitly set `b=None` when calling *`Fit`* so it will be ignored.

Warning: Fitting to ODEs is extremely difficult from an algorithmic point of view, since these systems are usually very sensitive to the parameters. Using (very) good initial guesses for the parameters and initial values is critical.

Upon every iteration of performing the fit the ODEModel is integrated again from the initial point using the new guesses for the parameters.

We can plot it just like always:

```
# Generate some data
tvec = np.linspace(0, 500, 1000)

A, B = ode_model(t=tvec, **fit_result.params)
plt.plot(tvec, A, label='[A]')
plt.plot(tvec, B, label='[B]')
plt.scatter(tdata, adata)
plt.legend()
plt.show()
```



As an example of the power of `symfit`'s ODE syntax, let's have a look at a system with 2 equilibria: compound AA + B <-> AAB and AAB + B <-> BAAB.

In `symfit` these can be implemented as:

```
AA, B, AAB, BAAB, t = variables('AA, B, AAB, BAAB, t')
k, p, l, m = parameters('k, p, l, m')

AA_0 = 10  # Some made up initial amound of [AA]
B = AA_0 - BAAB + AA  # [B] is not independent.

model_dict = {
    D(BAAB, t): l * AAB * B - m * BAAB,
    D(AAB, t): k * A * B - p * AAB - l * AAB * B + m * BAAB,
    D(A, t): - k * A * B + p * AAB,
}
```

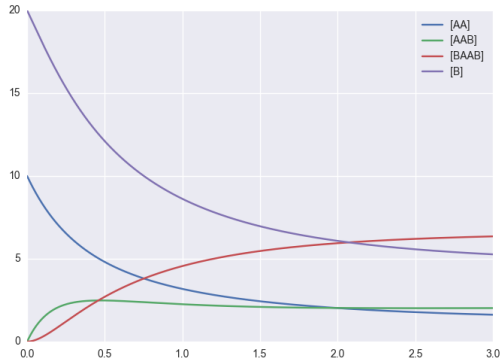
The result is as readable as one can reasonably expect from a multicomponent system (and while using chemical notation). Let's plot the model for some kinetics constants:

```
model = ODEModel(model_dict, initial={t: 0.0, AA: AA_0, AAB: 0.0, BAAB: 0.0})

# Generate some data
tdata = np.linspace(0, 3, 1000)
# Eval the normal way.
AA, AAB, BAAB = model(t=tdata, k=0.1, l=0.2, m=0.3, p=0.3)

plt.plot(tdata, AA, color='red', label='[AA]')
plt.plot(tdata, AAB, color='blue', label='[AAB]')
plt.plot(tdata, BAAB, color='green', label='[BAAB]')
plt.plot(tdata, B(BAAB=BAAB, AA=AA), color='pink', label='[B]')
# plt.plot(tdata, AA + AAB + BAAB, color='black', label='total')
```

(continues on next page)

(continued from previous page)

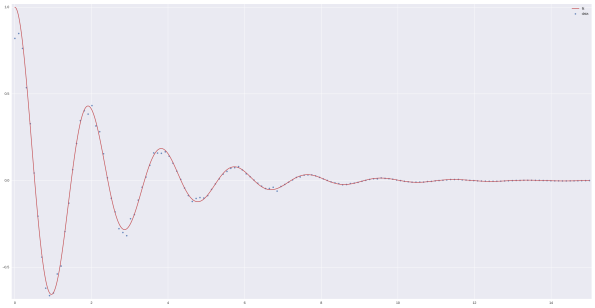
```
plt.legend()
plt.show()
```



More common examples, such as dampened harmonic oscillators also work as expected:

```
# Oscillator strength
k = Parameter('k')
# Mass, just there for the physics
m = 1
# Dampening factor
gamma = Parameter('gamma')

x, v, t = symfit.variables('x, v, t')

# Define the force based on Hooke's law, and dampening
a = (-k * x - gamma * v)/m
model_dict = {
    D(x, t): v,
    D(v, t): a,
}
ode_model = ODEModel(model_dict, initial={t: 0, v: 0, x: 1})

# Let's create some data...
times = np.linspace(0, 15, 150)
data = ode_model(times, k=11, gamma=0.9, m=m.value).x
# ... and add some noise to it.
noise = np.random.normal(1, 0.1, data.shape)  # 10% error
data *= noise

fit = Fit(ode_model, t=times, x=data)
fit_result = fit.execute()
```

Note: Evaluating the model above will produce a named tuple with values for both `x` and `v`. Since we are only interested in the values for `x`, we immediately select it with `.x`.



4.7 Fitting multiple datasets

A common fitting problem is to fit to multiple datasets. This is sometimes referred to as global fitting. In such fits parameters might be shared between the fits to the different datasets. The same syntax used for ODE fitting makes this problem very easy to solve in `symfit`.

As a simple example, suppose we have two datasets measuring exponential decay, with the same background, but different amplitude and decay rate.

$$f(x) = y_0 + a * e^{-b*x}$$

In order to fit to this, we define the following model:

```
x_1, x_2, y_1, y_2 = variables('x_1, x_2, y_1, y_2')
y0, a_1, a_2, b_1, b_2 = parameters('y0, a_1, a_2, b_1, b_2')

model = Model({
    y_1: y0 + a_1 * exp(- b_1 * x_1),
    y_2: y0 + a_2 * exp(- b_2 * x_2),
})
```

Note that `y0` is shared between the components. Fitting is then done in the normal way:

```
fit = Fit(model, x_1=xdata1, x_2=xdata2, y_1=ydata1, y_2=ydata2)
fit_result = fit.execute()
```

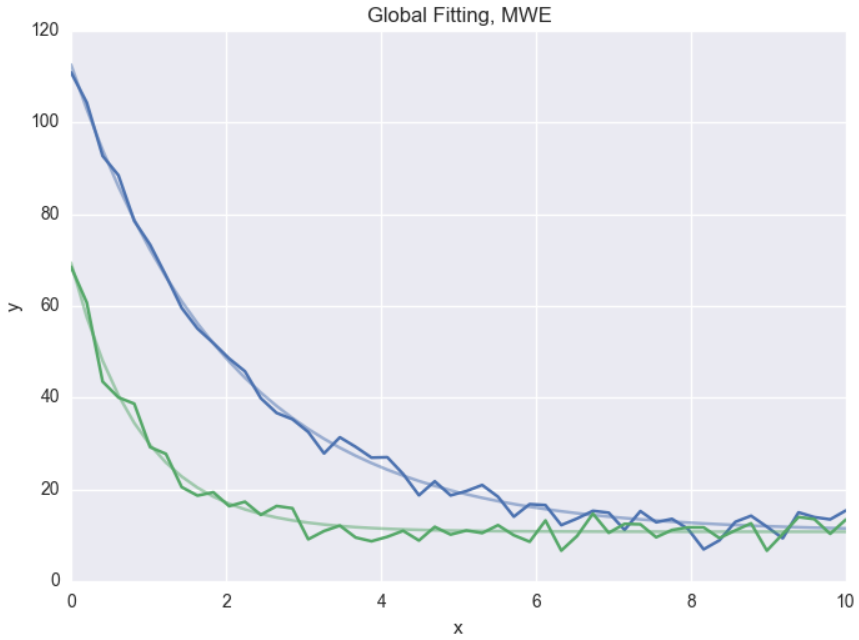
Any `Model` that comes to mind is fair game. Behind the scenes `symfit` will build a least squares function where the residues of all the components are added squared, ready to be minimized. Unlike in the above example, the x-axis does not even have to be shared between the components.

Warning: The regression coefficient is not properly defined for vector-valued models, but it is still listed! Until this is fixed, please recalculate it on your own for every component using the bestfit parameters.

Do not cite the overall R^2 given by `symfit`.

4.8 Global Minimization

Very often, there are multiple solutions to a fitting (or minimisation) problem. These are local minima of the objective function. The best solution of course is the global minimum, but most minimization algorithms will only find a local minimum, and thus the answer you get will depend on the initial values of your parameters. This can be incredibly annoying if you have no further knowledge about your system.



Luckily, global minimizers exist which are not influenced by the initial guesses for your parameters. In symfit, two such algorithms from `scipy` have been wrapped for this pourpose. Firstly, the `differential_evolution()` algorithm from `scipy` is wrapped as *`DifferentialEvolution`*. Secondly, the `basinhopping()` algorithm is available as *`BasinHopping`*. To use these minimizers, just tell *`Fit`*:

```
from symfit import Parameter, Variable, Model, Fit
from symfit.core.minimizers import DifferentialEvolution

x = Parameter('x')
x.min, x.max = -100, 100
x.value = -2.5
y = Variable('y')

model = Model({y: x**4 - 10 * x**2 - x})  # Skewed Mexican hat
fit = Fit(model, minimizer=DifferentialEvolution)
fit_result = fit.execute()
```

However, due to how this algorithm works, it's not great at finding the exact minimum (but it will find it if given enough time). You can work around this by "chaining" minimizers: first run a global minimization to (hopefully) get close to your answer, and then polish it off using a local minimizer:

```
fit = Fit(model, minimizer=[DifferentialEvolution, BFGS])
```

Note: Global minimizers such as differential evolution and basin-hopping are rather sensitive to their hyperparameters. You might need to play with those to get appropriate results, e.g.:

```
fit.execute(DifferentialEvolution={'popsize': 20, 'recombination': 0.9})
```

Note: There is no way to guarantee that the minimum found is actually the global minimum. Unfortunately there is

no way around this. Therefore, you should always critically inspect the results.

4.9 Constrained Basin-Hopping

Worthy of special mention is the ease with which constraints or bounds can be added to *symfit.core.minimizers.BasinHopping* when used through the *symfit.core.fit.Fit* interface. As a very simple example, we shall compare to an example from the `scipy` docs:

```
import numpy as np
from scipy.optimize import basinhopping

def func2d(x):
    f = np.cos(14.5 * x[0] - 0.3) + (x[1] + 0.2) * x[1] + (x[0] + 0.2) * x[0]
    df = np.zeros(2)
    df[0] = -14.5 * np.sin(14.5 * x[0] - 0.3) + 2. * x[0] + 0.2
    df[1] = 2. * x[1] + 0.2
    return f, df

minimizer_kwargs = {"method":"L-BFGS-B", "jac":True}
x0 = [1.0, 1.0]
ret = basinhopping(func2d, x0, minimizer_kwargs=minimizer_kwargs, niter=200)
```

Let's compare to the same functionality in `symfit`:

```
import numpy as np
from symfit.core.minimizers import BasinHopping
from symfit import parameters, Fit, cos

x0 = [1.0, 1.0]
x1, x2 = parameters('x1, x2', value=x0)

model = cos(14.5 * x1 - 0.3) + (x2 + 0.2) * x2 + (x1 + 0.2) * x1

fit = Fit(model, minimizer=BasinHopping)
fit_result = fit.execute(niter=200)
```

No *minimizer_kwargs* have to be provided, as `symfit` will automatically compute and provide the jacobian and select a minimizer. In this case, *symfit* will choose *BFGS*. When bounds are provided, *symfit* will switch to using *L-BFGS-B* instead. Setting bounds is as simple as:

```
x1.min = 0.0
x1.max = 100.0
```

However, the real strength of the *symfit* syntax lies in providing constraints:

```
constraints = [Eq(x1, x2)]
fit = Fit(model, minimizer=BasinHopping, constraints=constraints)
```

This artificial example will make sure *x1 == x2* after fitting. If you have read the *Minimize/Maximize* section, you will know how much work this would be in pure `scipy`.

4.10 Advanced usage

In general, the separate components of the model can be whatever you need them to be. You can mix and match which variables and parameters should be coupled and decoupled ad lib. Some examples are given below.

Same parameters and same function, different (in)dependent variables:

```
datasets = [data_1, data_2, data_3, data_4, data_5, data_6]

xs = variables('x_1, x_2, x_3, x_4, x_5, x_6')
ys = variables('y_1, y_2, y_3, y_4, y_5, y_6')
zs = variables(', '.join('z_{}'.format(i) for i in range(1, 7)))
a, b = parameters('a, b')

model_dict = {
    z: a/(y * b) *  exp(- a * x)
        for x, y, z in zip(xs, ys, zs)
}
```

4.11 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 unambiguous way using this ordering. For example:

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

CHAPTER 5

Style Guide & Best Practices

5.1 Style Guide

Anything Raymond Hettinger says wins the argument until I have time to write a proper style guide.

5.2 Best Practices

- It is recommended to always use named models. So not:

```
model = a * x**2
fit = Fit(model, xdata, ydata)
```

  but:

```
model = {y: a * x**2}
fit = Fit(model, x=xdata, y=ydata)
```

  In this simple example the two are equivalent but for multidimentional data using ordered arguments can become ambiguous and difficult to read. To increase readability, it is therefore recommended to always use named models.

- Evaluating a (vector valued) model returns a `namedtuple()`. You can access the elements by either tuple unpacking, or by using the variable names. Note that if you use tuple unpacking, the results will be ordered alphabetically. The following:

```
model = Model({y_1: x**2, y_2: x**3})
sol_1, sol_2 = model(x=xdata)
```

  is therefore equivalent to:

```
model = Model({y_1: x**2, y_2: x**3})
solutions = model(x=xdata)
sol_1 = solutions.y_1
sol_2 = solutions.y_2
```

Using numerical indexing (or something similar) is not recommended as it is less readable than the options given above:

```
sol_1 = model(x=xdata)[0]
```

CHAPTER 6

Technical Notes

Essays on mathematical and implementation details.

6.1 On Likelihood Fitting

The *LogLikelihood* objective function should be used to perform log-likelihood maximization. The *__call__()* and *eval_jacobian()* definitions have been changed to facilitate what one would expect from Likelihood fitting:

__call__ 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 $\vec{x}$.

6.2 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.

6.2.1 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 $\pm$ 1.000000e-01 when `sigma_y` is provided. This matches the analytical prediction.
- a = 5.102056e-02 $\pm$ 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 $\pm$ 9.897135e-03 when `sigma_y` is provided. This matches the analytical prediction.
- a = 5.102056e-03 $\pm$ 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, `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 areas 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`.

6.3 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.

6.4 Internal API Structure

Here we describe how the code is organized internally. This is only really relevant for advanced users and developers.

6.4.1 Fitting 101

Fitting a model to data is, at it's most basic, a parameter optimisation, and depending on whether you do a least-squares fit or a loglikelihood fit your objective function changes. This means we can split the process of fitting in three distint, isolated parts:: the *Model*, the Objective and the Minimizer.

In practice, *Fit* will choose an appropriate objective and minimizer, but you can also give it specific instances and classes; just in case you know better.

For both the minimizers and objectives there are abstract base classes, which describe the minimal API required. There are corresponding abstract classes for e.g. *ConstrainedMinimizer*.

6.4.2 Objectives

Objectives wrap both the Model and the data supplied, and when called must return a scalar. This scalar will be *minimized*, so when you need something maximized, be sure to add a negation in the right place(s). They must be called with the parameter values as keyword arguments. Be sure to inherit from the abstract base class(es) so you're sure you define all the methods that are expected.

6.4.3 Minimizers

Minimizers minimize. They are provided with a function to minimize (the objective) and the *Parameter* s as a function of which the objective should be minimized. Note that once again there are different base classes for minimizers that take e.g. bounds or support gradients. Their *execute()* method takes the metaparameters for the minimization. Again, be sure to inherit from the appropriate base class(es) if you're implementing your own minimizer to make sure all the expected methods are there. And if you're wrapping Scipy style minimizers, have a look at *ScipyMinimize* to avoid a duplication of efforts.

6.4.4 Example

Let's say we have some data:

```
xdata = np.linspace(0, 100, 25)
a_vec = np.random.normal(15, scale=2, size=xdata.shape)
b_vec = np.random.normal(100, scale=2, size=xdata.shape)
ydata = a_vec * xdata + b_vec
```

And we want to fit it to some model:

```
a = Parameter('a', value=0, min=0, max=1000)
b = Parameter('b', value=0, min=0, max=1000)
x = Variable('x')
model = a * x + b
```

If we want to fit this normally (but with a specified minimizer), we'd write the following:

```
fit = Fit(mode, xdata, ydata, minimizer=BFGS)
fit_result = fit.execute()
```

Now instead, we want to call the minimizer directly. We first define a custom objective function (actually just a chi squared):

```
def f(x, a, b):
    return a * x + b

def chi_squared(a, b):
    return np.sum((ydata - f(xdata, a, b))**2)

custom_minimize = BFGS(chi_squared, [a, b])
custom_minimize.execute()
```

You'll see that the result of both will be the same!

CHAPTER 7

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 8

Module Documentation

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

8.1 Fit

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

Bases: *symfit.core.fit.TakesData*

Abstract base class for all fitting objects.

error_func(**args*, ***kwargs*)

Every fit object has to define an error_func method, giving the function to be minimized.

eval_jacobian(**args*, ***kwargs*)

Every fit object has to define an eval_jacobian method, giving the jacobian of the function to be minimized.

execute(**args*, ***kwargs*)

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

class symfit.core.fit.**BaseModel**(*model*)

Bases: collections.abc.Mapping

ABC for `Model`'s. Makes sure models are iterable. Models can be initiated from Mappings or Iterables of Expressions, or from an expression directly. Expressions are not enforced for ducktyping purposes.

__eq__(*other*)

`Model`'s are considered equal when they have the same dependent variables, and the same expressions for those dependent variables. The same is defined here as passing sympy == for the vars themselves, and as expr1 - expr2 == 0 for the expressions. For more info check the sympy docs.

Parameters **other** – Instance of `Model`.

Returns bool

__getitem__(*dependent_var*)
Returns the expression belonging to a given dependent variable.

Parameters **dependent_var** (Variable) – Instance of Variable

Returns The expression belonging to dependent_var

__init__(*model*)
Initiate a Model from a dict:

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

Preferred way of initiating Model, since now you know what the dependent variable is called.

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

__iter__()

Returns iterable over self.model_dict

__len__()

Returns the number of dependent variables for this model.

__neg__()

Returns new model with opposite sign. Does not change the model in-place, but returns a new copy.

__str__()
A representation upon printing has to provided.

bounds

Returns List of tuples of all bounds on parameters.

shared_parameters

Returns bool, indicating if parameters are shared between the vector components of this model.

vars

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

class symfit.core.fit.**CallableModel**(*model*)
Bases: *symfit.core.fit.BaseModel*

Defines a callable model. 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.

eval_components(**args*, ***kwargs*)
Evaluate the components of the model with the given data. Used for numerical evaluation.

eval_jacobian(**args*, ***kwargs*)

Returns The jacobian matrix of the function.

finite_difference(**args*, *dx=1e-08*, ***kwargs*)
Calculates a numerical approximation of the Jacobian of the model using the sixth order central finite difference method. Accepts a *dx* keyword to tune the relative stepsize used. Makes 6*n_params calls to the model.

Returns A numerical approximation of the Jacobian of the model as a list with length n_components containing numpy arrays of shape (n_params, n_datapoints)

numerical_components

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

class symfit.core.fit.**Constraint**(*constraint*, *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*, *model*)

Parameters

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

__neg__()

Returns new model with opposite sign. Does not change the model in-place, but returns a new copy.

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*, *minimizer=None*, *objective=None*, *constraints=None*, ***named_data*)

Bases: *symfit.core.fit.HasCovarianceMatrix*

Your one stop fitting solution! Based on the nature of the input, this object will attempt to select the right fitting type for your problem.

If you need very specific control over how the problem is solved, you can pass it the minimizer or objective function you would like to use.

Example usage:

```
a, b = parameters('a, b')
x, y = variables('x, y')

model = {y: a * x + b}

# Fit will use its default settings
fit = Fit(model, x=xdata, y=ydata)
fit_result = fit.execute()

# Use Nelder-Mead instead
fit = Fit(model, x=xdata, y=ydata, minimizer=NelderMead)
fit_result = fit.execute()

# Use Nelder-Mead to get close, and BFGS to polish it off
fit = Fit(model, x=xdata, y=ydata, minimizer=[NelderMead, BFGS])
fit_result = fit.execute(minimizer_kwargs=[dict(xatol=0.1), {}])
```

__init__(*model*, **ordered_data*, *minimizer=None*, *objective=None*, *constraints=None*, ***named_data*)

Parameters

- **model** – (dict of) sympy expression(s) or `Model` object.
- **constraints** – iterable of `Relation` objects to be used as constraints.
- **absolute_sigma** (*bool*) – 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.
- **objective** – Have Fit use your specified objective. Can be one of the predefined *symfit* objectives or any callable which accepts fit parameters and returns a scalar.
- **minimizer** – Have Fit use your specified *symfit.core.minimizers.BaseMinimizer*. Can be a `Sequence` of *symfit.core.minimizers.BaseMinimizer*.
- **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.

execute(***minimize_options*)

Execute the fit.

Parameters **minimize_options** – keyword arguments to be passed to the specified minimizer.

Returns FitResults instance

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

Bases: *symfit.core.fit.TakesData*

Mixin class for calculating the covariance matrix for any model that has a well-defined Jacobian J. The covariance is then approximated as $J^T W J$, where W contains the weights of each data point.

Supports vector valued models, but is unable to estimate covariances for those, just variances. Therefore, take the result with a grain of salt for vector models.

covariance_matrix(*best_fit_params*)

Given best fit parameters, this function finds the covariance matrix. This matrix gives the (co)variance in the parameters.

Parameters **best_fit_params** – dict of best fit parameters as given by .best_fit_params()

Returns covariance matrix.

class symfit.core.fit.**LinearLeastSquares**(**args*, ***kwargs*)

Bases: *symfit.core.fit.BaseFit*

Experimental. Solves the linear least squares problem analytically. Involves no iterations or approximations, and therefore gives the best possible fit to the data.

The Model provided has to be linear.

Currently, since this object still has to mature, it suffers from the following limitations:

- It does not check if the model can be linearized by a simple substitution. For example, exp(a * x) -> b * exp(x). You will have to do this manually.
- Does not use bounds or guesses on the Parameter's. Then again, it doesn't have to, since you have an exact solution. No guesses required.
- It only works with scalar functions. This is strictly enforced.

__init__(**args*, ***kwargs*)

Raises ModelError in case of a non-linear model or when a vector valued function is provided.

best_fit_params()

Fits to the data and returns the best fit parameters.

Returns dict containing parameters and their best-fit values.

covariance_matrix(*best_fit_params*)

Given best fit parameters, this function finds the covariance matrix. This matrix gives the (co)variance in the parameters.

Parameters **best_fit_params** – dict of best fit parameters as given by .best_fit_params()

Returns covariance matrix.

execute()

Execute an analytical (Linear) Least Squares Fit. This object works by symbolically solving when $\nabla\chi^2 = 0$.

To perform this task the expression of $\nabla\chi^2$ is determined, ignoring that χ^2 involves summing over all terms. Then the sum is performed by substituting the variables by their respective data and summing all terms, while leaving the parameters symbolic.

The resulting system of equations is then easily solved with sympy.solve.

Returns FitResult

static is_linear(*model*)
Test whether model is of linear form in it's parameters.

Currently this function does not recognize if a model can be considered linear by a simple substitution, such as exp(k x) = k' exp(x).

Parameters **model** – Model instance

Returns True or False

class symfit.core.fit.**Model**(*model*)
Bases: *symfit.core.fit.CallableModel*

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({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.

Models are also iterable, behaving as their internal model_dict. In the example above, a[y] returns x**2, len(a) == 1, y in a == True, etc.

__str__()
Printable representation of this model.

Returns str

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

chi_squared_jacobian
Return a symbolic jacobian of the χ^2 function. Vector of derivatives w.r.t. each parameter. Not a Matrix but a vector!

eval_components(**args*, ***kwargs*)

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

eval_jacobian(**args*, ***kwargs*)

Returns Jacobian evaluated at the specified point.

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_chi_squared_jacobian

Returns lambda functions of the jacobian of the `.chi_squared` method.

numerical_jacobian

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

exception symfit.core.fit.**ModelError**
Bases: `Exception`

Raised when a problem occurs with a model.

class symfit.core.fit.**NonLinearLeastSquares**(**args*, ***kwargs*)
Bases: *symfit.core.fit.BaseFit*

Experimental. Implements non-linear least squares *[wiki_nllsq]*. Works by a two step process: First the model is linearised by doing a first order taylor expansion around the guesses for the parameters. Then a LinearLeastSquares fit is performed. This is iterated until a fit of sufficient quality is obtained.

Sensitive to good initial guesses. Providing good initial guesses is a must.

__init__(**args*, ***kwargs*)

Parameters

- **model** – (dict of) sympy expression or `Model` object.
- **absolute_sigma** (*bool*) – 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.

execute(*relative_error=1e-08*, *max_iter=500*)
Perform a non-linear least squares fit.

Parameters

- **relative_error** – Relative error between the sum of squares of subsequent itterations. Once smaller than the value specified, the fit is considered complete.
- **max_iter** – Maximum number of iterations before giving up.

Returns Instance of `FitResults`.

class symfit.core.fit.**ODEModel**(*model_dict*, *initial*, **lsoda_args*, ***lsoda_kwargs*)
Bases: *symfit.core.fit.CallableModel*

Model build from a system of ODEs. When the model is called, the ODE is integrated using the LSODA package.

__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** – Ordered arguments for the parameters and independent variables
- **kwargs** – Keyword arguments for the parameters and independent variables

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.

__getitem__(*dependent_var*)
Gives the function defined for the derivative of `dependent_var`. e.g. $y' = f(y, t)$, model[y] -> f(y, t)

Parameters **dependent_var** –

Returns

__init__(*model_dict*, *initial*, **lsoda_args*, ***lsoda_kwargs*)

Parameters

- **model_dict** – Dictionary specifying ODEs. e.g. model_dict = {D(y, x): a * x**2}
- **initial** – `dict` of initial conditions for the ODE. Must be provided! e.g. initial = {y: 1.0, x: 0.0}
- **lsoda_args** – args to pass to the lsoda solver. See scipy's odeint for more info.
- **lsoda_kwargs** – kwargs to pass to the lsoda solver.

__iter__()

Returns iterable over self.model_dict

__neg__()

Returns new model with opposite sign. Does not change the model in-place, but returns a new copy.

__str__()
Printable representation of this model.

Returns str

eval_components(**args*, ***kwargs*)
Numerically integrate the system of ODEs.

Parameters

- **args** – Ordered arguments for the parameters and independent variables
- **kwargs** – Keyword arguments for the parameters and independent variables

Returns

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

Bases: object

An base class for everything that takes data. 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.
- **absolute_sigma** (*bool*) – 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.

data_shapes

Returns the shape of the data. In most cases this will be the same for all variables of the same type, if not this raises an Exception.

Ignores variables which are set to None by design so we know that those None variables can be assumed to have the same shape as the other in calculations where this is needed, such as the covariance matrix.

Returns Tuple of all independent var shapes, dependent var shapes.

dependent_data

Read-only Property

Returns Data belonging to each dependent variable as a dict with variable names as key, data as value.

Return type collections.OrderedDict

independent_data

Read-only Property

Returns Data belonging to each independent variable as a dict with variable names as key, data as value.

Return type collections.OrderedDict

initial_guesses

Returns Initial guesses for every parameter.

sigma_data

Read-only Property

Returns Data belonging to each sigma variable as a dict with variable names as key, data as value.

Return type collections.OrderedDict

class symfit.core.fit.**TaylorModel**(*model*)
Bases: *symfit.core.fit.Model*

A first-order Taylor expansion of a model around given parameter values (p_0). Is used by NonLinearLeastSquares. Currently only a first order expansion is implemented.

__init__(*model*)
Make a first order Taylor expansion of `model`.

Parameters **model** – Instance of `Model`

__str__()
When printing a TaylorModel, the point around which the expansion took place is included.

For example, a Taylor expansion of {y: sin(w * x)} at w = 0 would be printed as:

```
@{w: 0.0} -> y(x; w) = w*x
```

p0
Property of the p_0 around which to expand. Should be set by the names of the parameters themselves.

Example:

```
a = Parameter()
x, y = variables('x, y')
model = TaylorModel({y: sin(a * x)})

model.p0 = {a: 0.0}
```

params
params returns only the *free* parameters. Strictly speaking, the expression for a `TaylorModel` contains both the parameters $\vec{p}$ and $\vec{p_0}$ around which to expand, but params should only give $\vec{p}$. To get a mapping to the $\vec{p_0}$, use `.params_0`.

symfit.core.fit.**r_squared**(*model*, *fit_result*, *data*)
Calculates the coefficient of determination, R^2, for the fit.

(Is not defined properly for vector valued functions.)

Parameters

- **model** – Model instance
- **fit_result** – FitResults instance
- **data** – data with which the fit was performed.

8.2 Argument

class symfit.core.argument.**Argument**(*name=None*, **args*, ***assumptions*)
Bases: `sympy.core.symbol.Symbol`

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

If no name is explicitly provided a name will be generated.

For example:

```
y = Variable()
print(y.name)
>> 'x_0'

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

__init__(*name=None*, **args*, ***assumptions*)
Initialize self. See help(type(self)) for accurate signature.

static __new__(*cls*, *name=None*, **args*, ***assumptions*)
Symbols are identified by name and assumptions:

```
>>> from sympy import Symbol
>>> Symbol("x") == Symbol("x")
```

True >>> Symbol("x", real=True) == Symbol("x", real=False) False

class symfit.core.argument.**Parameter**(*name=None*, *value=1.0*, *min=None*, *max=None*, *fixed=False*, ***assumptions*)
Bases: *symfit.core.argument.Argument*

Parameter objects are used to facilitate bounds on function parameters. Important change from *symfit>0.4.1*: the name needs to be the first keyword, followed by the guess value. If no name is provided, the initial value can be passed as a keyword argument, e.g.: *value=0.1*. A generic name will then be generated.

__call__(**values*, ***named_values*)
Call an expression to evaluate it at the given point.

Future improvements: I would like if func and signature could be buffered after the first call so they don't have to be recalculated for every call. However, nothing can be stored on self as sympy uses __slots__ for efficiency. This means there is no instance dict to put stuff in! And I'm pretty sure it's ill advised to hack into the __slots__ of Expr.

However, for the moment I don't really notice a performance penalty in running tests.

p.s. In the current setup signature is not even needed since no introspection is possible on the Expr before calling it anyway, which makes calculating the signature absolutely useless. However, I hope that someday some monkey patching expert in shining armour comes by and finds a way to store it in __signature__ upon __init__ of any `symfit` expr such that calling inspect_sig.signature on a symbolic expression will tell you which arguments to provide.

Parameters

- **self** – Any subclass of sympy.Expr
- **values** – Values for the Parameters and Variables of the Expr.
- **named_values** – Values for the vars and params by name. `named_values` is allowed to contain too many values, as this sometimes happens when using **fit_result.params on a submodel. The irrelevant params are simply ignored.

Returns The function evaluated at `values`. The type depends entirely on the input. Typically an array or a float but nothing is enforced.

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

Parameters

- **name** – Name of the Parameter.

- **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.
- **assumptions** – assumptions to pass to `sympy`.

static __new__(*cls*, *name=None*, **args*, ***kwargs*)

Symbols are identified by name and assumptions:

```
>>> from sympy import Symbol
>>> Symbol("x") == Symbol("x")
```

True >>> Symbol("x", real=True) == Symbol("x", real=False) False

class symfit.core.argument.**Variable**(*name=None*, **args*, ***assumptions*)

Bases: *symfit.core.argument.Argument*

Variable type.

8.3 Operators

Monkey Patching module.

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

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

Call an expression to evaluate it at the given point.

Future improvements: I would like if func and signature could be buffered after the first call so they don't have to be recalculated for every call. However, nothing can be stored on self as sympy uses __slots__ for efficiency. This means there is no instance dict to put stuff in! And I'm pretty sure it's ill advised to hack into the __slots__ of Expr.

However, for the moment I don't really notice a performance penalty in running tests.

p.s. In the current setup signature is not even needed since no introspection is possible on the Expr before calling it anyway, which makes calculating the signature absolutely useless. However, I hope that someday some monkey patching expert in shining armour comes by and finds a way to store it in __signature__ upon __init__ of any `symfit` expr such that calling inspect_sig.signature on a symbolic expression will tell you which arguments to provide.

Parameters

- **self** – Any subclass of sympy.Expr
- **values** – Values for the Parameters and Variables of the Expr.
- **named_values** – Values for the vars and params by name. `named_values` is allowed to contain too many values, as this sometimes happens when using **fit_result.params on a submodel. The irrelevant params are simply ignored.

Returns The function evaluated at `values`. The type depends entirely on the input. Typically an array or a float but nothing is enforced.

8.4 Fit Results

class symfit.core.fit_results.**FitResults**(*model*, *popt*, *covariance_matrix*, *infodic*, *mesg*, *ier*, ***gof_qualifiers*)

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. - parameter values + stdev - R squared (Regression coefficient.) or other fit quality qualifiers. - fitting status message - covariance matrix

Contains the attribute *params*, which is an OrderedDict containing all the parameter names and their optimized values. Can be ** unpacked when evaluating *Model*'s.

__getattr__(*item*)

Return the requested *item* if it can be found in the gof_qualifiers dict.

Parameters **item** – Name of Goodness of Fit qualifier.

Returns Goodness of Fit qualifier if present.

__init__(*model*, *popt*, *covariance_matrix*, *infodic*, *mesg*, *ier*, ***gof_qualifiers*)

Excuse the ugly names of most of these variables, they are inherited from scipy. Will be changed.

Parameters

- **model** – *Model* that was fit to.
- **popt** – best fit parameters, same ordering as in model.params.
- **pcov** – covariance matrix.
- **infodic** – dict with fitting info.
- **mesg** – Status message.
- **ier** – Number of iterations.
- **gof_qualifiers** – Any remaining keyword arguments should be Goodness of fit (g.o.f.) qualifiers.

__str__()

Pretty print the results as a table.

covariance(*param_1*, *param_2*)

Return the covariance between param_1 and param_2.

Parameters

- **param_1** – Parameter Instance.
- **param_2** – Parameter Instance.

Returns Covariance of the two params.

stdev(*param*)

Return the standard deviation in a given parameter as found by the fit.

Parameters **param** – Parameter Instance.

Returns Standard deviation of param.

value(*param*)

Return the value in a given parameter as found by the fit.

Parameters **param** – Parameter Instance.

Returns Value of `param`.

variance(*param*)
Return the variance in a given parameter as found by the fit.

Parameters **param** – `Parameter` Instance.

Returns Variance of `param`.

8.5 Minimizers

class symfit.core.minimizers.**BFGS**(**args*, ***kwargs*)
Bases: *symfit.core.minimizers.ScipyGradientMinimize*

Wrapper around `scipy.optimize.minimize()`'s BFGS algorithm.

class symfit.core.minimizers.**BaseMinimizer**(*objective*, *parameters*)
Bases: `object`

ABC for all Minimizers.

__init__(*objective*, *parameters*)

Parameters

- **objective** – Objective function to be used.
- **parameters** – List of *Parameter* instances

execute(***options*)
The execute method should implement the actual minimization procedure, and should return a *FitResults* instance.

Parameters **options** – options to be used by the minimization procedure.

Returns an instance of *FitResults*.

class symfit.core.minimizers.**BasinHopping**(**args*, *local_minimizer=<class 'symfit.core.minimizers.BFGS'>*, ***kwargs*)
Bases: *symfit.core.minimizers.ScipyMinimize*, *symfit.core.minimizers.BaseMinimizer*

Wrapper around `scipy.optimize.basinhopping()`'s basin-hopping algorithm.

As always, the best way to use this algorithm is through *Fit*, as this will automatically select a local minimizer for you depending on whether you provided bounds, constraints, etc.

However, BasinHopping can also be used directly. Example (with jacobian):

```
import numpy as np
from symfit.core.minimizers import BFGS, BasinHopping
from symfit import parameters

def func2d(x1, x2):
    f = np.cos(14.5 * x1 - 0.3) + (x2 + 0.2) * x2 + (x1 + 0.2) * x1
    return f

def jac2d(x1, x2):
    df = np.zeros(2)
    df[0] = -14.5 * np.sin(14.5 * x1 - 0.3) + 2. * x1 + 0.2
    df[1] = 2. * x2 + 0.2
```

(continues on next page)

(continued from previous page)

```
    return df

x0 = [1.0, 1.0]
np.random.seed(555)
x1, x2 = parameters('x1, x2', value=x0)
fit = BasinHopping(func2d, [x1, x2], local_minimizer=BFGS)
minimizer_kwargs = {'jac': fit.list2kwargs(jac2d)}
fit_result = fit.execute(niter=200, minimizer_kwargs=minimizer_kwargs)
```

See `scipy.optimize.basinhopping()` for more options.

__init__(**args*, *local_minimizer=<class 'symfit.core.minimizers.BFGS'>*, ***kwargs*)

Parameters

- **local_minimizer** – minimizer to be used for local minimization steps. Can be any subclass of *symfit.core.minimizers.ScipyMinimize*.
- **args** – positional arguments to be passed on to *super*.
- **kwargs** – keyword arguments to be passed on to *super*.

execute(***minimize_options*)

Execute the basin-hopping minimization.

Parameters **minimize_options** – options to be passed on to `scipy.optimize.basinhopping()`.

Returns *symfit.core.fit_results.FitResults*

class symfit.core.minimizers.**BoundedMinimizer**(*objective*, *parameters*)

Bases: *symfit.core.minimizers.BaseMinimizer*

ABC for Minimizers that support bounds.

class symfit.core.minimizers.**COBYLA**(**args*, ***kwargs*)

Bases: *symfit.core.minimizers.ScipyConstrainedMinimize*

Wrapper around `scipy.optimize.minimize()`'s COBYLA algorithm.

class symfit.core.minimizers.**ChainedMinimizer**(**args*, *minimizers=None*, ***kwargs*)

Bases: *symfit.core.minimizers.BaseMinimizer*

A minimizer that consists of multiple other minimizers, each executed in order. This is valuable if you have minimizers that are not good at finding the exact minimum such as *NelderMead* or *DifferentialEvolution*.

__init__(**args*, *minimizers=None*, ***kwargs*)

Parameters

- **minimizers** – a `Sequence` of *BaseMinimizer* objects, which need to be run in order.
- ***args** – passed to *symfit.core.minimizers.BaseMinimizer.__init__()*.
- ****kwargs** – passed to *symfit.core.minimizers.BaseMinimizer.__init__()*.

execute(***minimizer_kwargs*)

Execute the chained-minimization. In order to pass options to the seperate minimizers, they can be passed by using the names of the minimizers as keywords. For example:

```
fit = Fit(self.model, self.xx, self.yy, self.ydata,
          minimizer=[DifferentialEvolution, BFGS])
fit_result = fit.execute(
    DifferentialEvolution={'seed': 0, 'tol': 1e-4, 'maxiter': 10},
    BFGS={'tol': 1e-4}
)
```

In case of multiple identical minimizers an index is added to each keyword argument to make them identifiable. For example, if:

```
minimizer=[BFGS, DifferentialEvolution, BFGS])
```

then the keyword arguments will be 'BFGS', 'DifferentialEvolution', and 'BFGS_2'.

Parameters **minimizer_kwargs** – Minimizer options to be passed to the minimzers by name

Returns an instance of *FitResults*.

class symfit.core.minimizers.**ConstrainedMinimizer**(**args*, *constraints=None*, ***kwargs*)

Bases: *symfit.core.minimizers.BaseMinimizer*

ABC for Minimizers that support constraints

__init__(**args*, *constraints=None*, ***kwargs*)

Parameters

- **objective** – Objective function to be used.
- **parameters** – List of *Parameter* instances

class symfit.core.minimizers.**DifferentialEvolution**(**args*, ***kwargs*)

Bases: *symfit.core.minimizers.ScipyMinimize*, *symfit.core.minimizers.GlobalMinimizer*, *symfit.core.minimizers.BoundedMinimizer*

A wrapper around `scipy.optimize.differential_evolution()`.

execute(*, *strategy='rand1bin'*, *polish=False*, *recombination=0.95*, *mutation=(0.423*, *1.053)*, *init='latinhypercube'*, *popsize=40*, ***de_options*)

Calls the wrapped algorithm.

Parameters

- **bounds** – The bounds for the parameters. Usually filled by *BoundedMinimizer*.
- **jacobian** – The Jacobian. Usually filled by *ScipyGradientMinimize*.
- ****minimize_options** – Further keywords to pass to `scipy.optimize.minimize()`. Note that your *method* will usually be filled by a specific subclass.

class symfit.core.minimizers.**DummyModel**(*params*)

Bases: `tuple`

__getnewargs__()

Return self as a plain tuple. Used by copy and pickle.

static __new__(*_cls*, *params*)

Create new instance of DummyModel(params,)

__repr__()

Return a nicely formatted representation string

params
Alias for field number 0

class symfit.core.minimizers.**GlobalMinimizer**(**args*, ***kwargs*)
Bases: *symfit.core.minimizers.BaseMinimizer*

A minimizer that looks for a global minimum, instead of a local one.

__init__(**args*, ***kwargs*)

Parameters

- **objective** – Objective function to be used.
- **parameters** – List of *Parameter* instances

class symfit.core.minimizers.**GradientMinimizer**(**args*, *jacobian=None*, ***kwargs*)
Bases: *symfit.core.minimizers.BaseMinimizer*

ABC for Minizers that support the use of a jacobian

__init__(**args*, *jacobian=None*, ***kwargs*)

Parameters

- **objective** – Objective function to be used.
- **parameters** – List of *Parameter* instances

resize_jac(*func*)
Removes values with identical indices to fixed parameters from the output of func. func has to return the jacobian of a scalar function.

Parameters **func** – Jacobian function to be wrapped. Is assumed to be the jacobian of a scalar function.

Returns Jacobian corresponding to non-fixed parameters only.

class symfit.core.minimizers.**LBFGSB**(**args*, ***kwargs*)
Bases: *symfit.core.minimizers.ScipyGradientMinimize*, *symfit.core.minimizers.BoundedMinimizer*

Wrapper around `scipy.optimize.minimize()`'s LBFGSB algorithm.

execute(***minimize_options*)
Calls the wrapped algorithm.

Parameters

- **bounds** – The bounds for the parameters. Usually filled by *BoundedMinimizer*.
- **jacobian** – The Jacobian. Usually filled by *ScipyGradientMinimize*.
- ****minimize_options** – Further keywords to pass to `scipy.optimize.minimize()`. Note that your *method* will usually be filled by a specific subclass.

classmethod method_name()
Returns the name of the minimize method this object represents. This is needed because the name of the object is not always exactly what needs to be passed on to scipy as a string. :return:

class symfit.core.minimizers.**MINPACK**(**args*, ***kwargs*)
Bases: *symfit.core.minimizers.ScipyMinimize*, *symfit.core.minimizers.GradientMinimizer*, *symfit.core.minimizers.BoundedMinimizer*

Wrapper to scipy's implementation of MINPACK, since it is the industry standard.

__init__(**args*, ***kwargs*)

Parameters

- **objective** – Objective function to be used.
- **parameters** – List of *Parameter* instances

execute(***minpack_options*)

Parameters ****minpack_options** – Any named arguments to be passed to leastsqbound

class symfit.core.minimizers.**NelderMead**(**args*, ***kwargs*)

Bases: *symfit.core.minimizers.ScipyMinimize*, *symfit.core.minimizers.BaseMinimizer*

Wrapper around `scipy.optimize.minimize()`'s NelderMead algorithm.

classmethod method_name()

Returns the name of the minimize method this object represents. This is needed because the name of the object is not always exactly what needs to be passed on to scipy as a string. :return:

class symfit.core.minimizers.**SLSQP**(**args*, ***kwargs*)

Bases: *symfit.core.minimizers.ScipyConstrainedMinimize*, *symfit.core.minimizers.GradientMinimizer*, *symfit.core.minimizers.BoundedMinimizer*

Wrapper around `scipy.optimize.minimize()`'s SLSQP algorithm.

__init__(**args*, ***kwargs*)

Parameters

- **objective** – Objective function to be used.
- **parameters** – List of *Parameter* instances

execute(***minimize_options*)

Calls the wrapped algorithm.

Parameters

- **bounds** – The bounds for the parameters. Usually filled by *BoundedMinimizer*.
- **jacobian** – The Jacobian. Usually filled by *ScipyGradientMinimize*.
- ****minimize_options** – Further keywords to pass to `scipy.optimize.minimize()`. Note that your *method* will usually be filled by a specific subclass.

scipy_constraints(*constraints*)

Returns all constraints in a scipy compatible format.

Returns dict of scipy compatible constraints, including jacobian term.

class symfit.core.minimizers.**ScipyConstrainedMinimize**(**args*, ***kwargs*)

Bases: *symfit.core.minimizers.ScipyMinimize*, *symfit.core.minimizers.ConstrainedMinimizer*

Base class for `scipy.optimize.minimize()`'s constrained-minimizers.

__init__(**args*, ***kwargs*)

Parameters

- **objective** – Objective function to be used.
- **parameters** – List of *Parameter* instances

execute(***minimize_options*)

Calls the wrapped algorithm.

Parameters

- **bounds** – The bounds for the parameters. Usually filled by *BoundedMinimizer*.
- **jacobian** – The Jacobian. Usually filled by *ScipyGradientMinimize*.
- ****minimize_options** – Further keywords to pass to scipy.optimize.minimize(). Note that your *method* will usually be filled by a specific subclass.

scipy_constraints(*constraints*)
Returns all constraints in a scipy compatible format.

Returns dict of scipy compatible statements.

class symfit.core.minimizers.**ScipyGradientMinimize**(**args*, ***kwargs*)
Bases: *symfit.core.minimizers.ScipyMinimize*, *symfit.core.minimizers.GradientMinimizer*

Base class for scipy.optimize.minimize()'s gradient-minimizers.

__init__(**args*, ***kwargs*)

Parameters

- **objective** – Objective function to be used.
- **parameters** – List of *Parameter* instances

execute(***minimize_options*)
Calls the wrapped algorithm.

Parameters

- **bounds** – The bounds for the parameters. Usually filled by *BoundedMinimizer*.
- **jacobian** – The Jacobian. Usually filled by *ScipyGradientMinimize*.
- ****minimize_options** – Further keywords to pass to scipy.optimize.minimize(). Note that your *method* will usually be filled by a specific subclass.

class symfit.core.minimizers.**ScipyMinimize**(**args*, ***kwargs*)
Bases: object

Mix-in class that handles the execute calls to scipy.optimize.minimize().

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

execute(*bounds=None*, *jacobian=None*, *constraints=None*, ***, *tol=1e-09*, ***minimize_options*)
Calls the wrapped algorithm.

Parameters

- **bounds** – The bounds for the parameters. Usually filled by *BoundedMinimizer*.
- **jacobian** – The Jacobian. Usually filled by *ScipyGradientMinimize*.
- ****minimize_options** – Further keywords to pass to scipy.optimize.minimize(). Note that your *method* will usually be filled by a specific subclass.

list2kwargs(*func*)
Given an objective function *func*, make sure it is always called via keyword arguments with the relevant parameter names.

Parameters **func** – Function to be wrapped to keyword only calls.

Returns wrapped function

classmethod method_name()
Returns the name of the minimize method this object represents. This is needed because the name of the object is not always exactly what needs to be passed on to scipy as a string. :return:

8.6 Objectives

class symfit.core.objectives.**BaseObjective**(*model*, *data*)
Bases: object

ABC for objective functions. Implements basic data handling.

__call__(***parameters*)
Evaluate the objective function for given parameter values.

Parameters **parameters** –

Returns float

__init__(*model*, *data*)

Parameters

- **model** – *symfit* style model.
- **data** – data for all the variables of the model.

dependent_data
Read-only Property

Returns Data belonging to each dependent variable as a dict with variable names as key, data as value.

Return type collections.OrderedDict

independent_data
Read-only Property

Returns Data belonging to each independent variable as a dict with variable names as key, data as value.

Return type collections.OrderedDict

sigma_data
Read-only Property

Returns Data belonging to each sigma variable as a dict with variable names as key, data as value.

Return type collections.OrderedDict

class symfit.core.objectives.**GradientObjective**(*model*, *data*)
Bases: *symfit.core.objectives.BaseObjective*

ABC for objectives that support gradient methods.

eval_jacobian(***parameters*)
Evaluate the jacobian for given parameter values.

Parameters **parameters** –

Returns float

class symfit.core.objectives.**LeastSquares**(*model*, *data*)
Bases: *symfit.core.objectives.GradientObjective*

Objective representing the χ^2 of a model.

__call__(*, *flatten_components=True*, ***parameters*)

Parameters

- **parameters** – values of the *Parameter*'s to evaluate χ^2 at.
- **flatten_components** – if *True*, return the total χ^2. If *False*, return the χ^2 per component of the *BaseModel*.

Returns scalar or list of scalars depending on the value of *flatten_components*.

eval_jacobian(***parameters*)
Jacobian of χ^2 in the *Parameter*'s ($\nabla_{\vec{p}}\chi^2$).

Parameters **parameters** – values of the *Parameter*'s to evaluate $\nabla_{\vec{p}}\chi^2$ at.

Returns *np.array* of length equal to the number of parameters..

class symfit.core.objectives.**LogLikelihood**(*model*, *data*)
Bases: *symfit.core.objectives.GradientObjective*

Error function to be minimized by a minimizer in order to *maximize* the log-likelihood.

__call__(***parameters*)

Parameters **parameters** – values for the fit parameters.

Returns scalar value of log-likelihood

eval_jacobian(*, *apply_func=<function nansum>*, ***parameters*)
Jacobian for log-likelihood is defined as $\nabla_{\vec{p}}(\log(L(\vec{p}|\vec{x})))$.

Parameters

- **parameters** – values for the fit parameters.
- **apply_func** – Function to apply to each component before returning it. The default is to sum away along the datapoint dimension using *np.nansum*.

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

class symfit.core.objectives.**MinimizeModel**(*model*, **args*, ***kwargs*)
Bases: *symfit.core.objectives.BaseObjective*

Objective to use when the model itself is the quantity that should be minimized. This is only supported for scalar models.

__call__(***parameters*)
Evaluate the objective function for given parameter values.

Parameters **parameters** –

Returns float

__init__(*model*, **args*, ***kwargs*)

Parameters

- **model** – *symfit* style model.
- **data** – data for all the variables of the model.

class symfit.core.objectives.**VectorLeastSquares**(*model*, *data*)
Bases: *symfit.core.objectives.GradientObjective*

Implemented for MINPACK only. Returns the residuals/sigma before squaring and summing, rather then chi2 itself.

__call__(*, *flatten_components=True*, ***parameters*)
Returns the value of the square root of χ^2, summing over the components.

This function now supports setting variables to None.

Parameters

- **p** – array of parameter values.
- **flatten_components** – If True, summing is performed over the data indices (default).

Returns $\sqrt{(\chi^2)}$

eval_jacobian(***parameters*)
Evaluate the jacobian for given parameter values.

Parameters **parameters** –

Returns float

8.7 Support

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

class symfit.core.support.**D**
Bases: sympy.core.function.Derivative

Convenience wrapper for sympy.Derivative. Used most notably in defining ODEModel's.

class symfit.core.support.**RequiredKeyword**
Bases: object

Flag variable to indicate that this is a required keyword.

exception symfit.core.support.**RequiredKeywordError**
Bases: Exception

Error raised in case a keyword-only argument is not treated as such.

class symfit.core.support.**cached_property**
Bases: property

A property which cashes the output of the first ever call and always returns that value from then on, unless delete is called on the attribute.

This is typically used in converting *sympy* code into *scipy* compatible code, which is computationally a very expensive step we would like to perform only once.

Does not allow setting of the attribute.

__delete__(*obj*)
Calling delete on the attribute will delete the cache. :param obj: parent object.

__get__(*obj*, *objtype=None*)
In case of a first call, this will call the decorated function and return it's output. On every subsequent call, the same output will be returned.

Parameters

- **obj** – the parent object this property is attached to.
- **objtype** –

Returns Output of the first call to the decorated function.

class symfit.core.support.**deprecated**(*replacement=None*)
Bases: object

Decorator to raise a DeprecationWarning.

__call__(*func*)
Call self as a function.

__init__(*replacement=None*)

Parameters **replacement** – The function which should now be used instead.

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.**key2str**(*target*)
In symfit there are many dicts with symbol: value pairs. These can not be used immediately as **kwargs, even though this would make a lot of sense from the context. This function wraps such dict to make them usable as **kwargs immediately.

Parameters **target** – *Mapping* to be made save

Returns *Mapping* of str(symbol): value pairs.

class symfit.core.support.**keywordonly**(***kwonly_arguments*)
Bases: object

Decorator class which wraps a python 2 function into one with keyword-only arguments.

Example:

```
@keywordonly(floor=True)
def f(x, **kwargs):
    floor = kwargs.pop('floor')
    return np.floor(x**2) if floor else x**2
```

This decorator is not much more than:

```
floor = kwargs.pop('floor') if 'floor' in kwargs else True
```

However, I prefer it's usage because:

- it's clear from reading the function declaration there is an option to provide this argument. The information on possible keywords is where you'd expect it to be.
- you're guaranteed that the pop works.

- It is fully inspect compatible such that sphynx is able to index these properly as keyword only arguments just like it would for native py3 keyword only arguments.

Please note that this decorator needs a ** argument on the wrapped function in order to work.

__call__ (*func*)
Returns a decorated version of *func*, who's signature now includes the keyword-only arguments.

Parameters **func** – the function to be decorated

Returns the decorated function

__init__ (***kwonly_arguments*)
Initialize self. See help(type(self)) for accurate signature.

symfit.core.support.**parameters**(*names*, ***kwargs*)
Convenience function for the creation of multiple parameters. For more control, consider using `symbols(names, cls=Parameter, **kwargs)` directly.

The *Parameter* attributes *value*, *min*, *max* and *fixed* can also be provided directly. If given as a single value, the same value will be set for all *Parameter*'s. When a sequence, it must be of the same length as the number of parameters created.

Example:: x1, x2 = parameters('x1, x2', value=[2.0, 1.3], min=0.0)

Parameters

- **names** – string of parameter names. Example: a, b = parameters('a, b')
- **kwargs** – kwargs to be passed onto `sympy.core.symbol.symbols()`. *value*, *min* and *max* will be handled separately if they are sequences.

Returns iterable of *symfit.core.argument.Parameter* objects

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*, ***kwargs*)

Convenience function for the creation of multiple variables. For more control, consider using `symbols(names, cls=Variable, **kwargs)` directly.

Parameters

- **names** – string of variable names. Example: x, y = variables('x, y')
- **kwargs** – kwargs to be passed onto `sympy.core.symbol.symbols()`

Returns iterable of *symfit.core.argument.Variable* objects

8.8 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.

8.9 Contrib

Contrib modules are modules and extensions to symfit provided by other people. This usually means the code is of slightly less quality, and may not survive future versions.

class symfit.contrib.interactive_guess.interactive_guess.**InteractiveGuess2D**(**args*, *n_points=100*, ***kwargs*)

Bases: *symfit.core.fit.TakesData*

A class that provides an graphical, interactive way of guessing initial fitting parameters.

__init__(**args*, *n_points=100*, ***kwargs*)

Create a matplotlib window with sliders for all parameters in this model, so that you may graphically guess initial fitting parameters. n_points is the number of points drawn for the plot. Data points are plotted as blue points, the proposed model as a red line.

Slider extremes are taken from the parameters where possible. If these are not provided, the minimum is 0; and the maximum is value*2. If no initial value is provided, it defaults to 1.

This will modify the values of the parameters present in model.

Parameters **n_points** (*int*) – The number of points used for drawing the fitted function.

__str__()

Represent the guesses in a human readable way.

Returns string with the guessed values.

execute(*, *block=True*, *show=True*, ***kwargs*)
Execute the interactive guessing procedure.

Parameters

- **show** (*bool*) – Whether or not to show the figure. Useful for testing.
- **block** – Blocking call to matplotlib

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

Bibliography

[taldcroft] http://nbviewer.jupyter.org/urls/gist.github.com/taldcroft/5014170/raw/31e29e235407e4913dc0ec403af7ed524372b612/curve_fit.ipynb

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

[wiki_nllsq] https://en.wikipedia.org/wiki/Non-linear_least_squares

Python Module Index

S

Index

Symbols

A

B

C

D

E

T

V