symfit Documentation

Release 0.5.6.dev2

tBuLi

Jan 24, 2023

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:

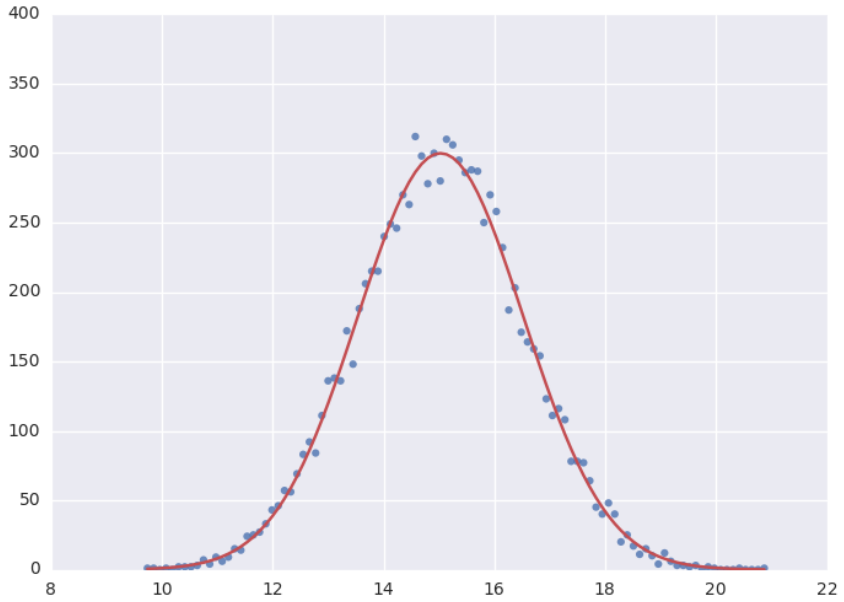
```
fit = Fit(model, xdata, ydata)
fit_result = fit.execute()
```

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 prefer to use *conda*, run

```
conda install -c conda-forge symfit
```

instead. Lastly, if you prefer to install manually you can download the source from https://github.com/tBuLi/symfit.

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

See *requirements.txt* for a full list.

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
```

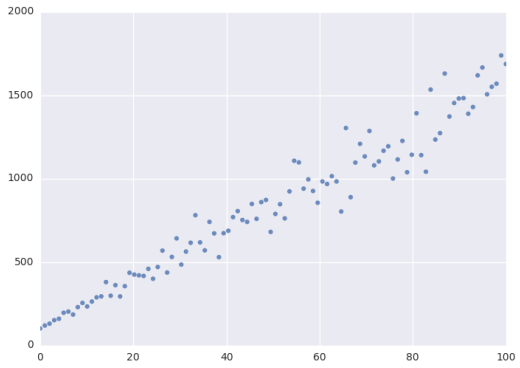
Lets fit this model to some generated data.

```
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:

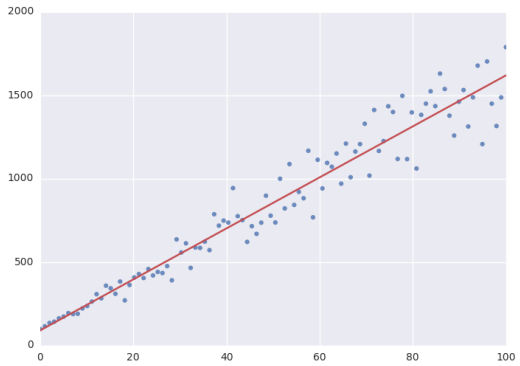
```
>>> print(fit_result.value(a))
>>> 14.66946...
>>> print(fit_result.stdev(a))
>>> 0.3367571...
>>> print(fit_result.value(b))
>>> 104.6558...
>>> print(fit_result.stdev(b))
>>> 19.49172...
>>> print(fit_result.r_squared)
>>> 0.950890866472
```

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)

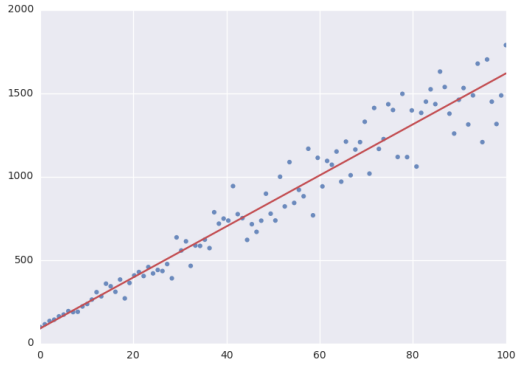
The default fitting object does least-squares fitting:

```
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 lose 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 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.

Multiple data sets can be likelihood fitted simultaneously by merging this example with that of global fitting, see *Example: Global Likelihood fitting* in the example section.

4.4 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):
```

(continues on next page)

(continued from previous page)

```
    """ 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.

4.5 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:

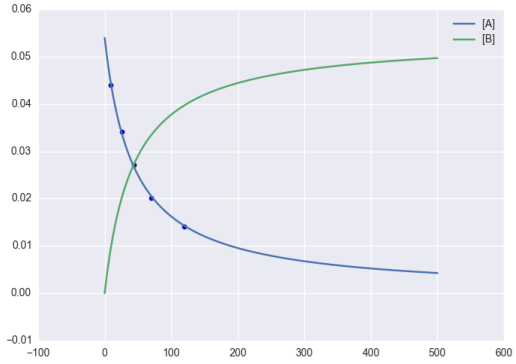
```
# Generate some data
tvec = np.linspace(0, 500, 1000)

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 = {
```

(continues on next page)

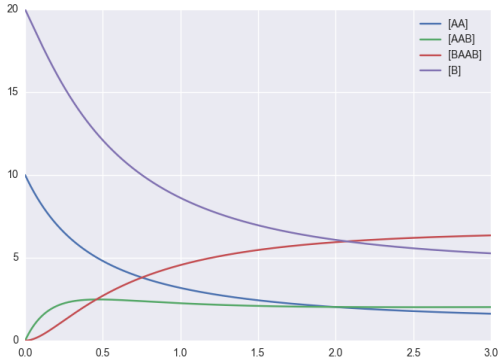


(continued from previous page)

```
    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:

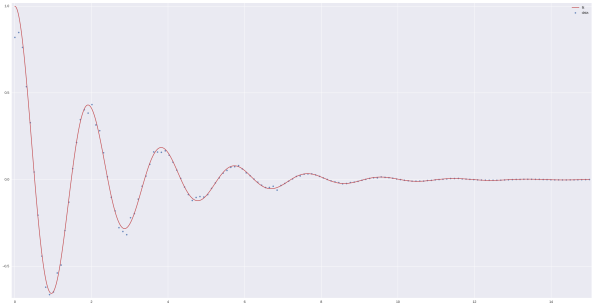
```
model = ODEModel(model_dict, initial={t: 0.0, AA: AA_0, AAB: 0.0, BAAB: 0.0})

# 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')
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`.



4.6 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 the different amplitude and decay rate.

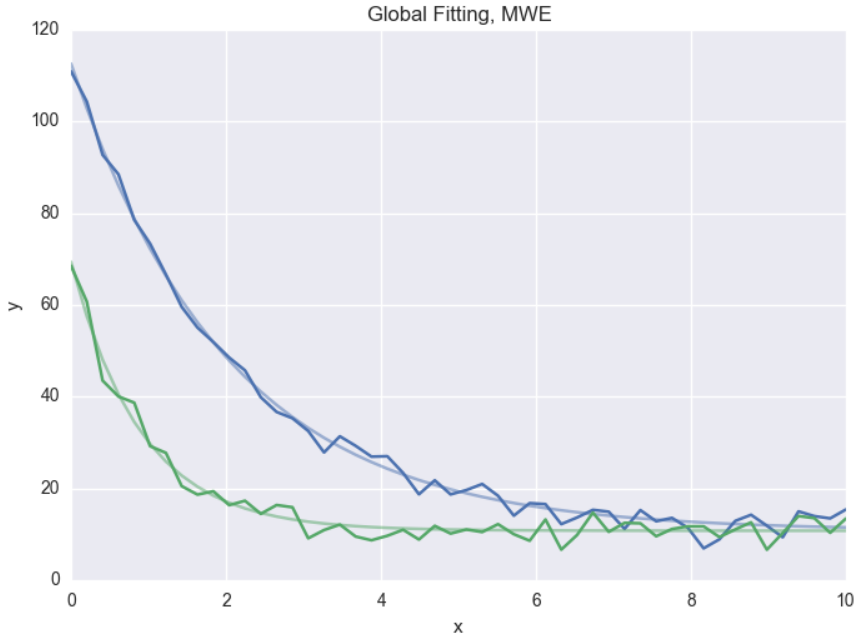
$$f(x) = y_0 + a * e^{-b*x}$$

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`.

4.7 Fitting multidimensional datasets

So far we have only considered problems with a single independent variable, but in the real world it is quite common to have problems with multiple independent variables. For example, a specific property over a grid, like the temperature of a surface. In that case, you have a three-dimensional dataset consisting of (x-, y-coordinates, temperature), and our model is a function $T(x, y; \vec{p})$, where $\vec{p}$ indicates the collection of parameters to be determined during the fit.

Let's work this out with the following mathematical model. We have a polynomial function with two coefficients, representing two terms of mixed order in `x` and `y`:

$T(x, y) = z = c_2 x^4 y^5 + c_1 x y^2$

Secondly, we have to implement our model:

```
x, y, z = variables('x, y, z')
c1, c2 = parameters('c1, c2')
model_dict = {z: Poly( {(1, 2): c1, (4, 5): c2}, x ,y)}
model = Model(model_dict)
# prints z(x, y; c1, c2) = Poly(c2*x**4*y**5 + c1*x*y**2, x, y, domain='ZZ[c1,c2]')
```

Now we can fit this polynomial model to some mock data. We have to be careful that `xdata`, `ydata` and `zdata` are two-dimensional:

```
x = np.linspace(0, 100, 100)
y = np.linspace(0, 100, 100)
xdata, ydata = np.meshgrid(x, y)
zdata = 42 * xdata**4 * ydata**5 + 3.14 * xdata * ydata**2

fit = Fit(model, x=xdata, y=ydata, z=zdata)
fit_result = fit.execute()
```

In conclusion, we made a mathematical model for a multidimensional function with two fit parameters, implemented this model, and fed it data to get a result.

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 purpose. 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 distinct, isolated parts: the *Model*, the Objective and the Minimizer.

In practice, *Fit* will choose an appropriate objective and minimizer on the basis of the model and the data, 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. If a minimizer is more specific, e.g. it supports constraints, then there are corresponding abstract classes for that, e.g. *ConstrainedMinimizer*.

6.4.2 Models

Models house the mathematical definition of the model we want to use to fit. For the typical usecase in `symfit` these are fully symbolical, and therefore a lot of their properties can be inspected automatically.

As a basic quality, all models are callable, i.e. they have implemented `__call__`. This is used to numerically evaluate the model given the parameters and independent variables. In order to make sure you get all the basic functionality, always inherit from *BaseModel*.

Next level up, if they inherit from *GradientModel* then they will have `eval_jacobian`, which will numerically evaluate the jacobian of the model. Lastly, if they inherit from *HessianModel*, they will also have `eval_hessian` to evaluate the hessian of the model. The standard *Model* is all of the above.

Odd ones out from the current library are *CallableNumericalModel* and *ODEModel*. They only inherit from *BaseModel* and are therefore callable, but their other behaviors are custom build.

Since `symfit 0.5.0`, the core of the model has been improved significantly. At the center of these improvements is *connectivity_mapping*. This mapping represent the connectivity matrix of the variables and parameters, and therefore encodes which variable depends on which. This is used in `__call__` to evaluate the components in order. To help with this, models have *ordered_symbols*. This property is the topologically sorted `connectivity_mapping`, and dictates the order in which variables have to be evaluated.

6.4.3 Objectives

Objectives wrap both the Model and the data supplied, and expose only the free parameters of the model to the outside world. When called they 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 can be called by using the parameter names as keyword arguments, or with a list of parameter values in the same order as *free_params* (alphabetical). The latter

is there because this is how `scipy` likes it. Be sure to inherit from the abstract base class(es) so you're sure you define all the methods that are expected of an objective. Similar to the models, they come in three types: *BaseObjective*, *GradientObjective* and *HessianObjective*. These must implement `__call__`, `eval_jacobian` and `eval_hessian` respectively.

When defining a new objective, it is best to inherit from *HessianObjective* and to define all three if possible. When feeding a model that does not implement `eval_hessian` to a *HessianObjective* no puppies die, *Fit* is clever enough to prevent this.

6.4.4 Minimizers

Last in the chain are the minimizers. 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. *Fit* depends on this to make its decisions. And if you're wrapping Scipy style minimizers, have a look at *ScipyMinimize* to avoid a duplication of efforts.

Minimizers must always implement a method `execute`, which will return an instance of *FitResults*. Any `*args` and `**kwargs` given to execute must be passed to the underlying minimizer.

6.4.5 Fit

Fit is responsible for stringing all of the above together intelligently. When not coached into the right direction, it will decide which minimizer and objective to use on the basis of the model and data.

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

Examples

8.1 Model Examples

These are examples of the flexibility of `symfit` Models. This is because essentially any valid `sympy` code can be provided as a model. This makes it very intuitive to define your mathematical models almost as you would on paper.

8.1.1 Example: Fourier Series

Suppose we want to fit a Fourier series to a dataset. As an example, let's take a step function:

$$f(x) = \begin{cases} 0 & \text{if} \quad -\pi < x \leq 0 \\ 1 & \text{if} \quad 0 < x < \pi \end{cases}$$

In the example below, we will attempt to fit this with a Fourier Series of order $n = 3$.

$$y(x) = a_0 + \sum_{i=1}^{n} a_i cos(i\omega x) + \sum_{i=1}^{n} b_i sin(i\omega x)$$

```
from symfit import parameters, variables, sin, cos, Fit
import numpy as np
import matplotlib.pyplot as plt

def fourier_series(x, f, n=0):
    """
    Returns a symbolic fourier series of order `n`.

    :param n: Order of the fourier series.
    :param x: Independent variable
    :param f: Frequency of the fourier series
    """
    # Make the parameter objects for all the terms
```

(continues on next page)

(continued from previous page)

```
    a0, *cos_a = parameters(','.join(['a{}'.format(i) for i in range(0, n + 1)]))
    sin_b = parameters(','.join(['b{}'.format(i) for i in range(1, n + 1)]))
    # Construct the series
    series = a0 + sum(ai * cos(i * f * x) + bi * sin(i * f * x)
                      for i, (ai, bi) in enumerate(zip(cos_a, sin_b), start=1))
    return series

x, y = variables('x, y')
w, = parameters('w')
model_dict = {y: fourier_series(x, f=w, n=3)}
print(model_dict)

# Make step function data
xdata = np.linspace(-np.pi, np.pi)
ydata = np.zeros_like(xdata)
ydata[xdata > 0] = 1
# Define a Fit object for this model and data
fit = Fit(model_dict, x=xdata, y=ydata)
fit_result = fit.execute()
print(fit_result)

# Plot the result
plt.plot(xdata, ydata)
plt.plot(xdata, fit.model(x=xdata, **fit_result.params).y, ls=':')
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```

This code prints:

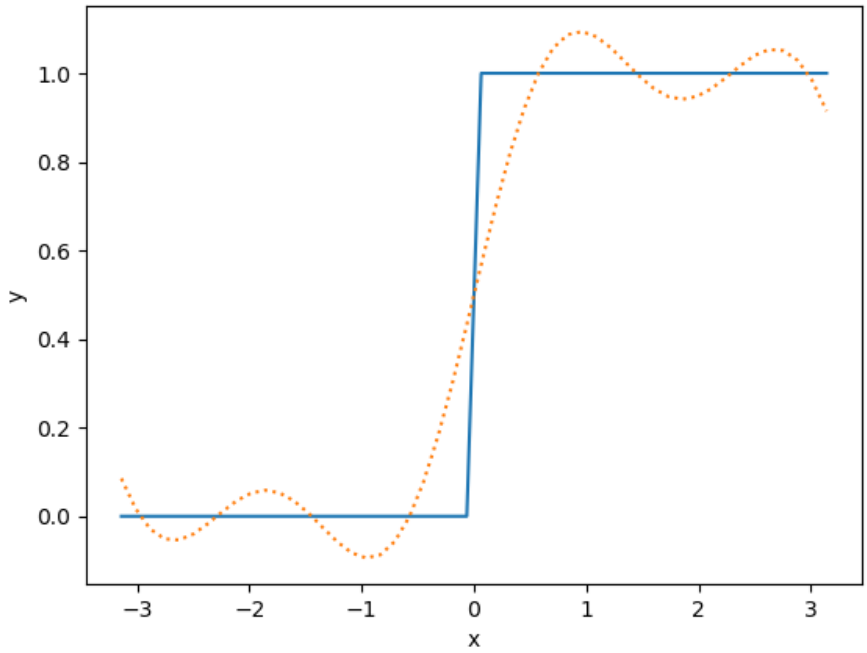
```
{y: a0 + a1*cos(w*x) + a2*cos(2*w*x) + a3*cos(3*w*x) + b1*sin(w*x) + b2*sin(2*w*x) +␣
↪b3*sin(3*w*x)}

Parameter Value        Standard Deviation
a0        5.000000e-01 2.075395e-02
a1        -4.903805e-12 3.277426e-02
a2        5.325068e-12 3.197889e-02
a3        -4.857033e-12 3.080979e-02
b1        6.267589e-01 2.546980e-02
b2        1.986491e-02 2.637273e-02
b3        1.846406e-01 2.725019e-02
w         8.671471e-01 3.132108e-02
Fitting status message: Optimization terminated successfully.
Number of iterations:   44
Regression Coefficient: 0.9401712713086535
```

8.1.2 Example: Piecewise continuous function

Piecewise continuus functions can be tricky to fit. However, using the `symfit` interface this process is made a lot easier. Suppose we want to fit to the following model:

$$f(x) = \begin{cases} x^2 - ax & \text{if} \quad x \leq x_0 \\ ax + b & \text{otherwise} \end{cases}$$

The script below gives an example of how to fit such a model:




```
from symfit import parameters, variables, Fit, Piecewise, exp, Eq, Model
import numpy as np
import matplotlib.pyplot as plt

x, y = variables('x, y')
a, b, x0 = parameters('a, b, x0')

# Make a piecewise model
y1 = x**2 - a * x
y2 = a * x + b
model = Model({y: Piecewise((y1, x <= x0), (y2, x > x0))})

# As a constraint, we demand equality between the two models at the point x0
# to do this, we substitute x -> x0 and demand equality using `Eq`
constraints = [
    Eq(y1.subs({x: x0}), y2.subs({x: x0}))
]
# Generate example data
xdata = np.linspace(-4, 4., 50)
ydata = model(x=xdata, a=0.0, b=1.0, x0=1.0).y
np.random.seed(2)
ydata = np.random.normal(ydata, 0.5)  # add noise

# Help the fit by bounding the switchpoint between the models
x0.min = 0.8
x0.max = 1.2

fit = Fit(model, x=xdata, y=ydata, constraints=constraints)
fit_result = fit.execute()
print(fit_result)
```

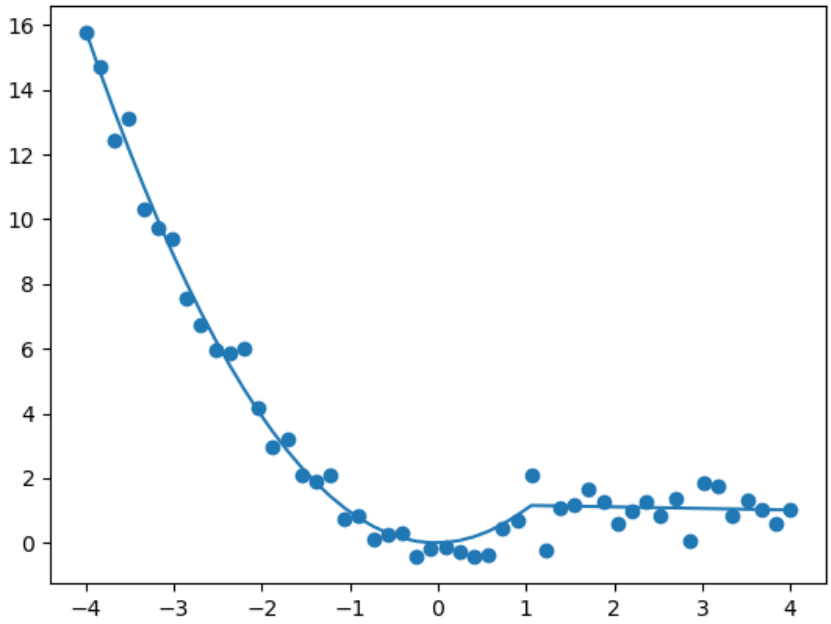
(continues on next page)

(continued from previous page)

```
plt.scatter(xdata, ydata)
plt.plot(xdata, model(x=xdata, **fit_result.params).y)
plt.show()
```

This code prints:

```
Parameter Value        Standard Deviation
a         -4.780338e-02 None
b         1.205443e+00 None
x0        1.051163e+00 None
Fitting status message: Optimization terminated successfully.
Number of iterations:   18
Regression Coefficient: 0.9849188499599985
```



Judging from this graph, another possible solution would have been to also demand a continuous derivative at the point *x0*. This can be achieved by setting the following constraints instead:

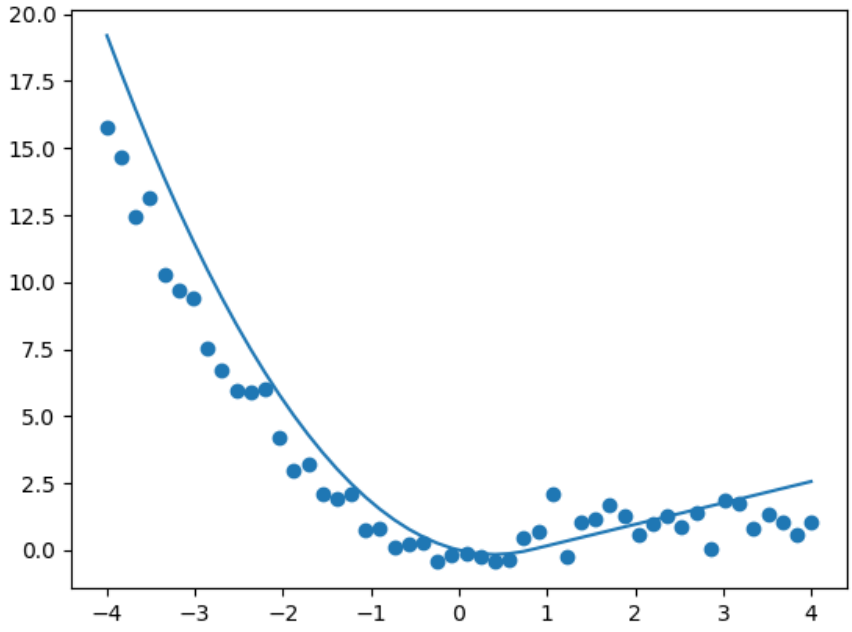
```
constraints = [
    Eq(y1.diff(x).subs({x: x0}), y2.diff(x).subs({x: x0})),
    Eq(y1.subs({x: x0}), y2.subs({x: x0}))
]
```

This gives the following plot:

and the following fit report:

```
Parameter Value        Standard Deviation
a         8.000000e-01 None
b         -6.400000e-01 None
```

(continues on next page)



(continued from previous page)

```
x0        8.000000e-01 None
Fitting status message: Optimization terminated successfully.
Number of iterations:   3
Regression Coefficient: 0.8558226069368662
```

The first fit is therefore the prevered one, but it does show you how easy it is to set these constraints using `symfit`.

8.1.3 Example: Polynomial Surface Fit

In this example, we want to fit a polynomial to a 2D surface. Suppose the surface is described by

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

A fit to such data can be performed as follows:

```
from symfit import Poly, variables, parameters, Model, Fit
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

x, y, z = variables('x, y, z')
c1, c2 = parameters('c1, c2')
# Make a polynomial. Note the `as_expr` to make it symfit friendly.
model_dict = {
    z: Poly( {(2, 0): c1, (0, 2): c1, (1, 1): c2}, x ,y).as_expr()
}
model = Model(model_dict)
print(model)
```

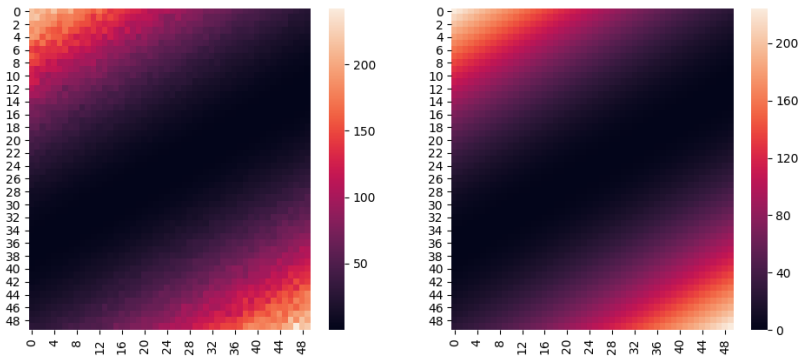
(continues on next page)

(continued from previous page)

```
# Generate example data
x_vec = np.linspace(-5, 5)
y_vec = np.linspace(-10, 10)
xdata, ydata = np.meshgrid(x_vec, y_vec)
zdata = model(x=xdata, y=ydata, c1=1.0, c2=2.0).z
zdata = np.random.normal(zdata, 0.05 * zdata)  # add 5% noise

# Perform the fit
fit = Fit(model, x=xdata, y=ydata, z=zdata)
fit_result = fit.execute()
zfit = model(x=xdata, y=ydata, **fit_result.params).z
print(fit_result)

fig, (ax1, ax2) = plt.subplots(1, 2)
sns.heatmap(zdata, ax=ax1)
sns.heatmap(zfit, ax=ax2)
plt.show()
```

This code prints:

```
z(x, y; c1, c2) = c1*x**2 + c1*y**2 + c2*x*y

Parameter Value        Standard Deviation
c1        9.973489e-01 1.203071e-03
c2        1.996901e+00 3.736484e-03
Fitting status message: Optimization terminated successfully.
Number of iterations:   6
Regression Coefficient: 0.9952824293713467
```



8.1.4 Example: ODEModel for Reaction Kinetics

Below is an example of how to use the *symfit.core.models.ODEModel*. In this example we will fit reaction kinetics data, taken from libretexts.

The data is from a first-order reaction A → B.

```
from symfit import variables, Parameter, Fit, D, ODEModel
import numpy as np
```

(continues on next page)

(continued from previous page)

```
import matplotlib.pyplot as plt

# First order reaction kinetics. Data taken from
# http://chem.libretexts.org/Core/Physical_Chemistry/Kinetics/Rate_Laws/The_Rate_Law
tdata = np.array([0, 0.9184, 9.0875, 11.2485, 17.5255, 23.9993, 27.7949,
                  31.9783, 35.2118, 42.973, 46.6555, 50.3922, 55.4747, 61.827,
                  65.6603, 70.0939])
concentration = np.array([0.906, 0.8739, 0.5622, 0.5156, 0.3718, 0.2702, 0.2238,
                          0.1761, 0.1495, 0.1029, 0.086, 0.0697, 0.0546, 0.0393,
                          0.0324, 0.026])

# Define our ODE model
A, B, t = variables('A, B, t')
k = Parameter('k')
model = ODEModel(
    {D(A, t): - k * A, D(B, t): k * A},
    initial={t: tdata[0], A: concentration[0], B: 0.0}
)

fit = Fit(model, A=concentration, t=tdata)
fit_result = fit.execute()

print(fit_result)

# Plotting, irrelevant to the symfit part.
t_axis = np.linspace(0, 80)
A_fit, B_fit, = model(t=t_axis, **fit_result.params)
plt.scatter(tdata, concentration)
plt.plot(t_axis, A_fit, label='[A]')
plt.plot(t_axis, B_fit, label='[B]')
plt.xlabel('t /min')
plt.ylabel('[X] /M')
plt.ylim(0, 1)
plt.xlim(0, 80)
plt.legend(loc=1)
plt.show()
```

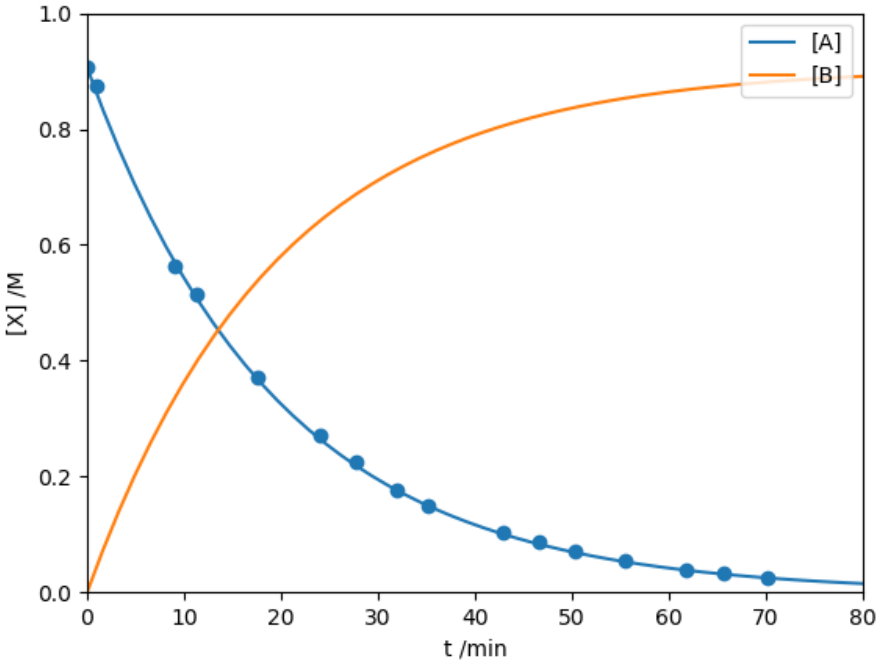
This is the resulting fit:

8.1.5 Example: Multiple species Reaction Kinetics using ODEModel

In this example we shall fit to a complex system of ODEs, based on that published by Polgar et al. However, we shall be generating some mock data instead of using the real deal.

[1]:
```
from symfit import (
    variables, parameters, ODEModel, D, Fit
)
from symfit.core.support import key2str
import numpy as np
import matplotlib.pyplot as plt
```

First we build a model representing the system of equations.



[2]:
```
t, F, MM, FMM, FMMF = variables('t, F, MM, FMM, FMMF')
k1_f, k1_r, k2_f, k2_r = parameters('k1_f, k1_r, k2_f, k2_r')

MM_0 = 10  # Some made up initial amount of [FF]

model_dict = {
    D(F, t): - k1_f * MM * F + k1_r * FMM - k2_f * FMM * F + k2_r * FMMF,
    D(FMM, t): k1_f * MM * F - k1_r * FMM - k2_r * FMM * F + k2_f * FMMF,
    D(FMMF, t): k2_f * FMM * F - k2_r * FMMF,
    D(MM, t): - k1_f * MM * F + k1_r * FMM,
}
model = ODEModel(
    model_dict,
    initial={t: 0.0, MM: MM_0, FMM: 0.0, FMMF: 0.0, F: 2 * MM_0}
)
print(model)
```

```
Derivative(F, t; k1_f, k1_r, k2_f, k2_r) = -k1_f*F*MM + k1_r*FMM - k2_f*F*FMM + k2_
↪r*FMMF
Derivative(FMM, t; k1_f, k1_r, k2_f, k2_r) = k1_f*F*MM - k1_r*FMM + k2_f*FMMF - k2_
↪r*F*FMM
Derivative(FMMF, t; k1_f, k1_r, k2_f, k2_r) = k2_f*F*FMM - k2_r*FMMF
Derivative(MM, t; k1_f, k1_r, k2_f, k2_r) = -k1_f*F*MM + k1_r*FMM
```

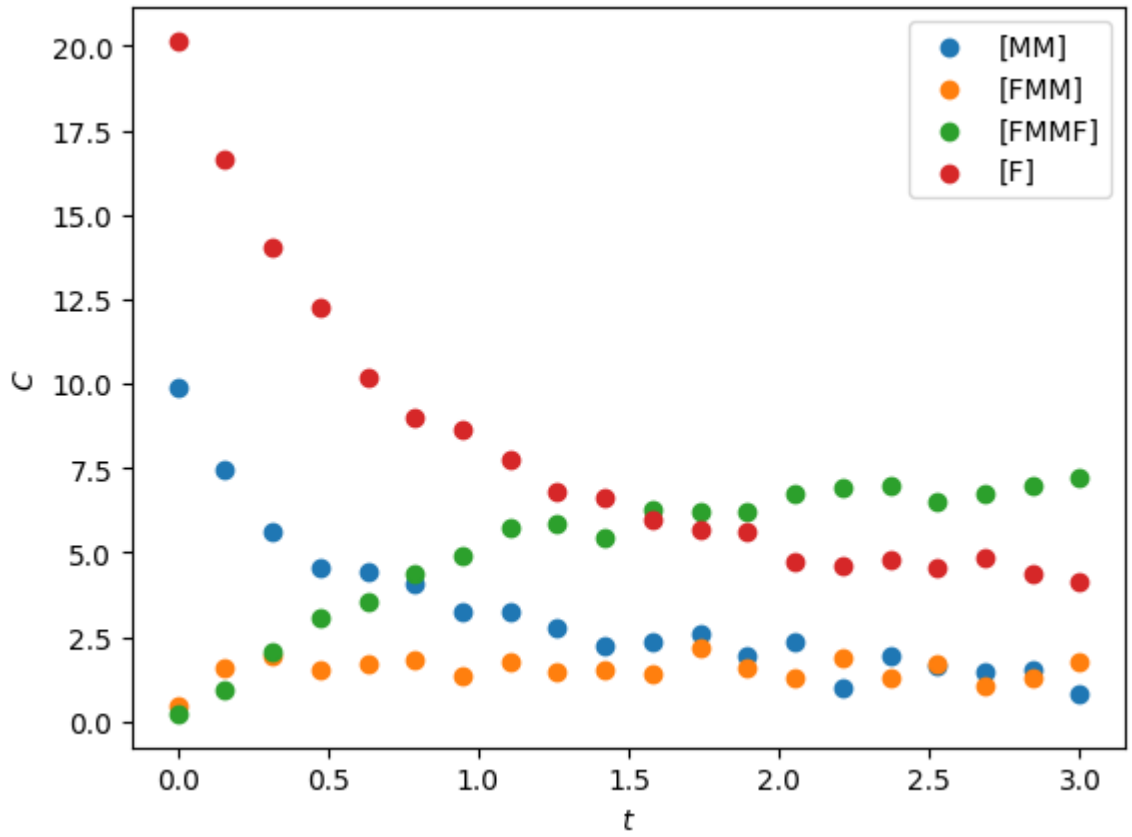
Generate mock data.

[3]:
```
tdata = np.linspace(0, 3, 20)
data = model(t=tdata, k1_f=0.1, k1_r=0.2, k2_f=0.3, k2_r=0.3)._asdict()
sigma_data = 0.3
np.random.seed(42)
```

(continues on next page)

(continued from previous page)

```
for var in data:
    data[var] += np.random.normal(0, sigma_data, size=len(tdata))

plt.scatter(tdata, data[MM], label='[MM]')
plt.scatter(tdata, data[FMM], label='[FMM]')
plt.scatter(tdata, data[FMMF], label='[FMMF]')
plt.scatter(tdata, data[F], label='[F]')
plt.xlabel(r'$t$')
plt.ylabel(r'$C$')
plt.legend()
plt.show()
```

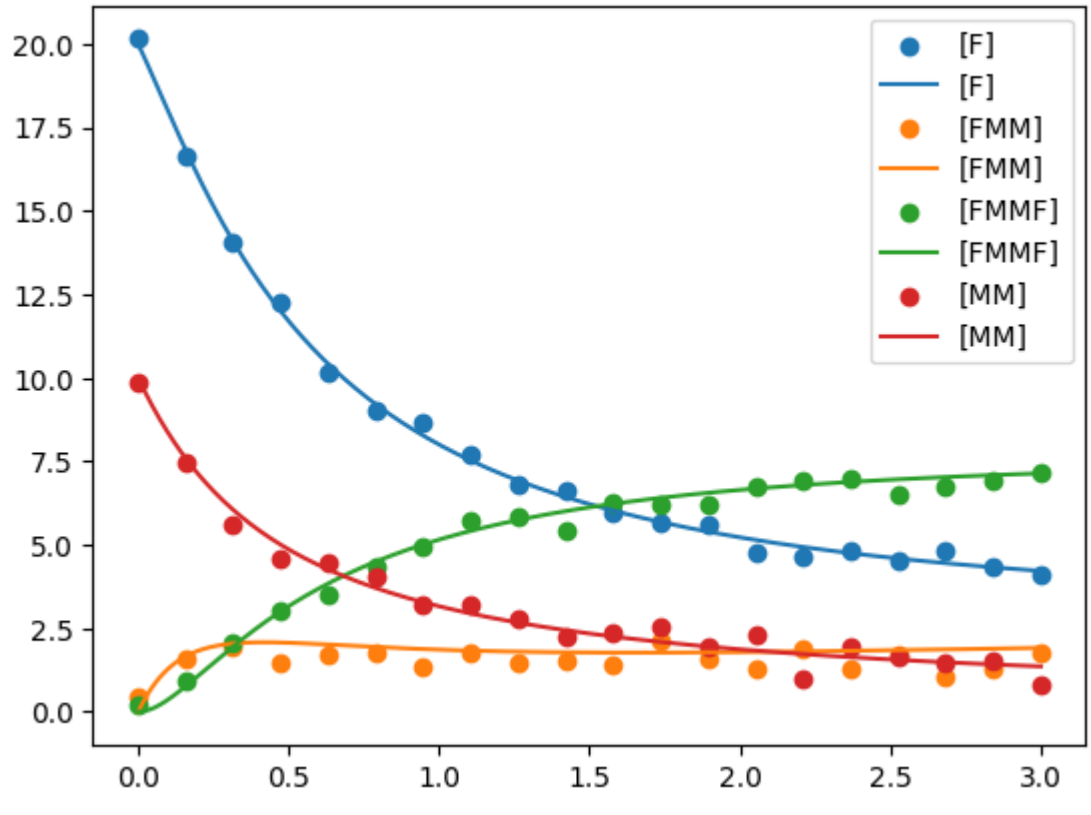


Perform the fit. Let's pretend that for experimental reasons, we can only measure the concentration for `MM` and `F`, but not for the intermediate `FMM` nor the product `FMMF`. This is no problem, as we can tell `symfit` to ignore those components by setting the data for them to `None`.

[4]:
```
k1_f.min, k1_f.max = 0, 1
k1_r.min, k1_r.max = 0, 1
k2_f.min, k2_f.max = 0, 1
k2_r.min, k2_r.max = 0, 1

fit = Fit(model, t=tdata, MM=data[MM], F=data[F],
          FMMF=None, FMM=None,
          sigma_F=sigma_data, sigma_MM=sigma_data)
fit_result = fit.execute()
print(fit_result)
```

```
Parameter Value        Standard Deviation
k1_f      9.540420e-02 4.440696e-03
k1_r      1.065119e-01 7.165718e-02
k2_f      2.706132e-01 5.305096e-02
k2_r      2.633630e-01 5.647280e-02
Status message         CONVERGENCE: REL_REDUCTION_OF_F_<=_FACTR*EPSMCH
Number of iterations   30
Objective              <symfit.core.objectives.LeastSquares object at 0x7f7c2d109dd0>
Minimizer              <symfit.core.minimizers.LBFGSB object at 0x7f7c2d0b9390>

Goodness of fit qualifiers:
chi_squared            33.98549453512615
objective_value        16.992747267563075
r_squared              0.9936568366400107
```

[5]:
```
taxis = np.linspace(tdata.min(), tdata.max(), 1000)
model_fit = model(t=taxis, **fit_result.params)._asdict()
for var in data:
    plt.scatter(tdata, data[var], label='[{}]'.format(var.name))
    plt.plot(taxis, model_fit[var], label='[{}]'.format(var.name))
plt.legend()
plt.show()
```



We see that the lack of data for some components is not a problem, they are predicted quite nicely.

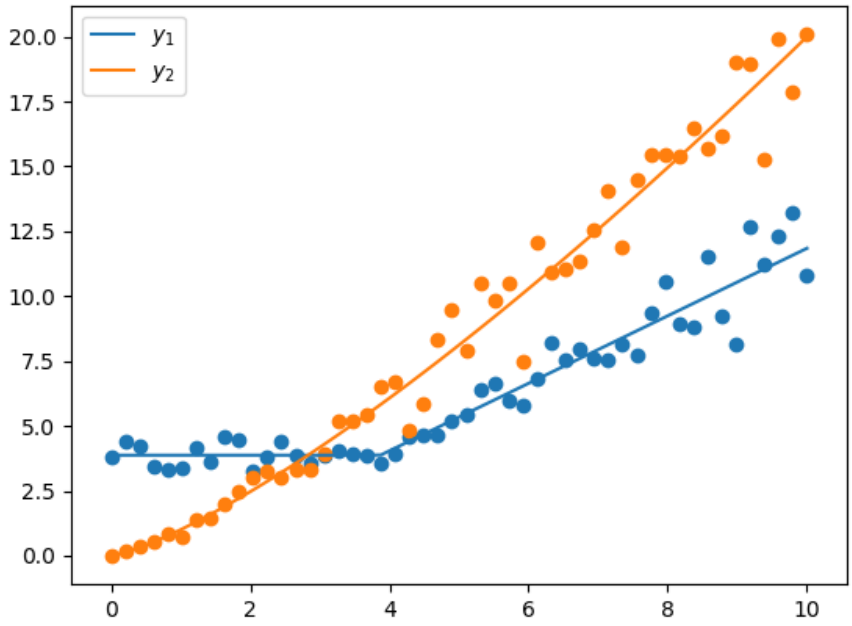
8.1.6 Example: Piecewise model using CallableNumericalModel

Below is an example of how to use the *symfit.core.models.CallableNumericalModel*. This class allows you to provide custom callables as your model, while still allowing clean interfacing with the `symfit` API.

These models also accept a mixture of symbolic and callable components, as will be demonstrated below. This allows the power-user great flexibility, since it is still easy to interface with `symfit`'s constraints, minimizers, etc.

```
from symfit import variables, parameters, Fit, D, ODEModel, CallableNumericalModel
import numpy as np
import matplotlib.pyplot as plt

def nonanalytical_func(x, a, b):
    """
    This can be any pythonic function which should be fitted, typically one
    which is not easily written or supported as an analytical expression.
    """
    # Do your non-trivial magic here. In this case a Piecewise, although this
    # could also be done symbolically.
    y = np.zeros_like(x)
    y[x > b] = (a * (x - b) + b)[x > b]
    y[x <= b] = b
    return y

x, y1, y2 = variables('x, y1, y2')
a, b = parameters('a, b')

mixed_model = CallableNumericalModel(
    {y1: nonanalytical_func, y2: x ** a},
    connectivity_mapping={y1: {x, a, b}}
)

# Generate data
xdata = np.linspace(0, 10)
y1data, y2data = mixed_model(x=xdata, a=1.3, b=4)
y1data = np.random.normal(y1data, 0.1 * y1data)
y2data = np.random.normal(y2data, 0.1 * y2data)

# Perform the fit
b.value = 3.5
fit = Fit(mixed_model, x=xdata, y1=y1data, y2=y2data)
fit_result = fit.execute()
print(fit_result)

# Plotting, irrelevant to the symfit part.
y1_fit, y2_fit, = mixed_model(x=xdata, **fit_result.params)
plt.scatter(xdata, y1data)
plt.plot(xdata, y1_fit, label=r'$y_1$')
plt.scatter(xdata, y2data)
plt.plot(xdata, y2_fit, label=r'$y_2$')
plt.legend(loc=0)
plt.show()
```

This is the resulting fit:



8.1.7 Example: ODEModels as subproblems using CallableNumericalModel

Imagine that you want to use ODE fitting, but your data aren't directly linked to an ODE variable. In this case, you can use the class :class:`symfit.core.models.CallableNumericalModel`.

Given are the variables `x`, `y` and `z`. `x` and `y` and the parameter `a` form an ODE model, where the derivative contains (due to the e-function) `y`. `z` is the result of a function with `y` and the additional parameter `b`.

`a` and `b` must be optimized, as shown below. The class uses the given model dictionary and the information about the variable and the parameters to fit the data.

[1]:
```
from symfit import variables, Parameter, Fit, D, ODEModel, CallableNumericalModel
import numpy as np
import matplotlib.pyplot as plt

# Create data
x_data = np.linspace(0.0, 10.0, 1000)
a_expected = 0.6
b_expected = 10.0
z_data = 2 * np.exp(a_expected * x_data) + b_expected

# Initialise variables and parameters
x, y, z = variables('x, y, z')
a = Parameter('a', 0.0)
b = Parameter('b', 0.0)

# Define model
ode_model = ODEModel({D(y, x): a * y}, initial={x: 0.0, y: 1.0})
model_dict = {
    z: 2 * y + b,
    y: lambda x, a: ode_model(x=x, a=a).y,
```

(continues on next page)

(continued from previous page)

```
}
model = CallableNumericalModel(model_dict, connectivity_mapping={z: {y, b}, y: {x, a}}
↪)

# Apply model
fit = Fit(model, x=x_data, z=z_data)
fit_result = fit.execute()
```

[2]:
```
print('a =', fit_result.value(a))
```

```
a = 0.5999999737968444
```

[3]:
```
print('b =', fit_result.value(b))
```

```
b = 10.00001056363364
```

8.1.8 Example: Matrix Equations using Tikhonov Regularization

This is an example of the use of matrix expressions in `symfit` models. This is illustrated by performing an inverse Laplace transform using Tikhonov regularization, but this could be adapted to other problems involving matrix quantities.

[1]:
```
from symfit import (
    variables, parameters, Model, Fit, exp, laplace_transform, symbols,
    MatrixSymbol, sqrt, Inverse, CallableModel
)
import numpy as np
import matplotlib.pyplot as plt
```

Say $f(t) = t * exp(-t)$, and $F(s)$ is the Laplace transform of $f(t)$. Let us first evaluate this transform using `sympy`.

[2]:
```
t, f, s, F = variables('t, f, s, F')
model = Model({f: t * exp(- t)})
laplace_model = Model(
    {F: laplace_transform(model[f], t, s, noconds=True)}
)
print(laplace_model)
```

```
[F(s; ) = (s + 1)**(-2)]
```

Suppose we are confronted with a dataset $F(s)$, but we need to know $f(t)$. This means an inverse Laplace transform has to be performed. However, numerically this operation is ill-defined. In order to solve this, Tikhonov regularization can be performed.

To demonstrate this, we first generate mock data corresponding to $F(s)$ and will then try to find (our secretly known) $f(t)$.

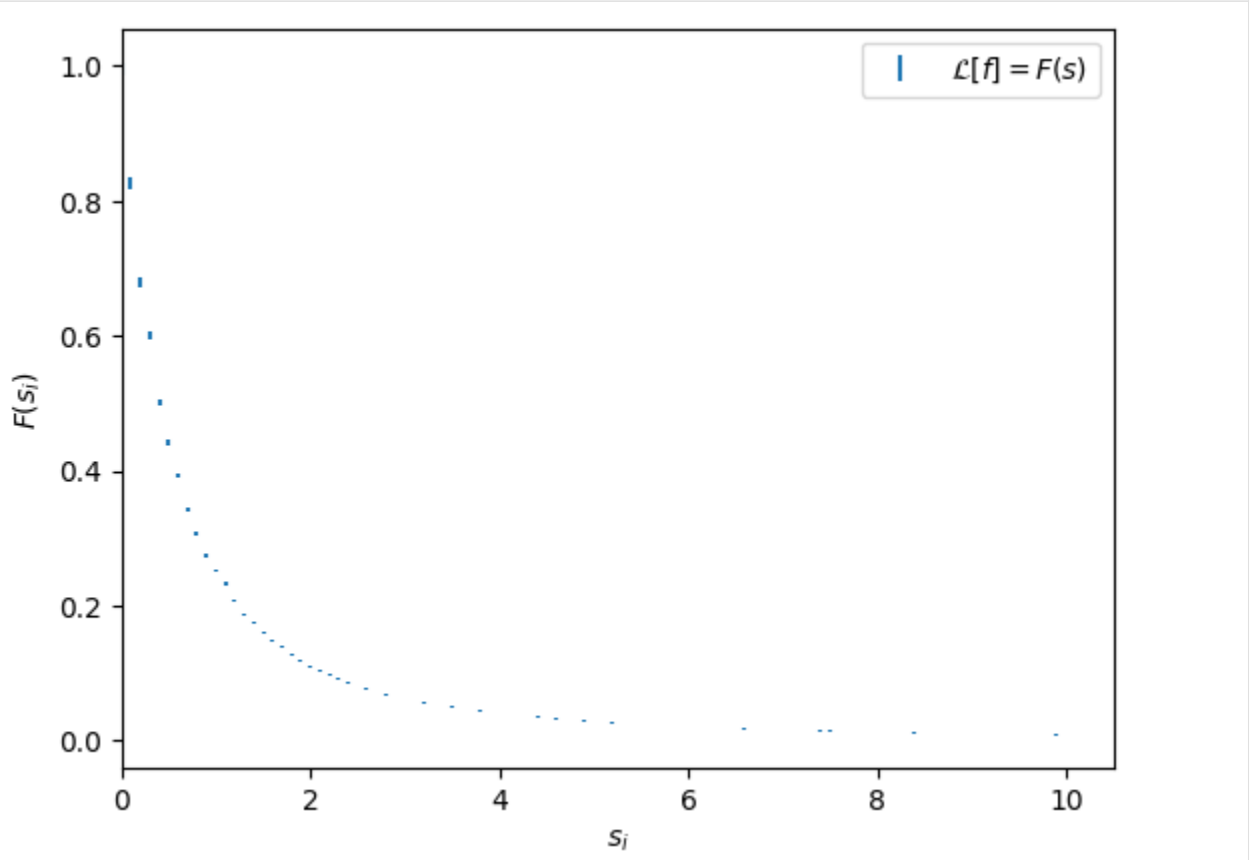
[3]:
```
epsilon = 0.01  # 1 percent noise
s_data = np.linspace(0, 10, 101)
F_data = laplace_model(s=s_data).F
F_sigma = epsilon * F_data
np.random.seed(2)
F_data = np.random.normal(F_data, F_sigma)
```

(continues on next page)

(continued from previous page)

```
plt.errorbar(s_data, F_data, yerr=F_sigma, fmt='none', label=r'$\mathcal{L}[f] = F(s)$
↪')
plt.xlabel(r'$s_i$')
plt.ylabel(r'$F(s_i)$')
plt.xlim(0, None)
plt.legend()
```

[3]:
```
<matplotlib.legend.Legend at 0x7fc03da30910>
```



We will now invert this data, using the procedure outlined in [].

[4]:
```
N_s = symbols('N_s', integer=True)  # Number of s_i points

M = MatrixSymbol('M', N_s, N_s)
W = MatrixSymbol('W', N_s, N_s)
Fs = MatrixSymbol('Fs', N_s, 1)
c = MatrixSymbol('c', N_s, 1)
d = MatrixSymbol('d', 1, 1)
I = MatrixSymbol('I', N_s, N_s)
a, = parameters('a')

model_dict = {
    W: Inverse(I + M / a**2),
    c: - W * Fs,
    d: sqrt(c.T * c),
```

(continues on next page)

(continued from previous page)

```
}
tikhonov_model = CallableModel(model_dict)
print(tikhonov_model)
```

```
[W(I, M; a) = (I + a**(-2)*M)**(-1),
 c(Fs, W; ) = -W*Fs,
 d(c; ) = (c.T*c)**(1/2)]
```

A `CallableModel` is needed because derivatives of matrix expressions sometimes cause problems.

Build required matrices, ignore s=0 because it causes a singularity.

[5]:
```
I_mat = np.eye(len(s_data[1:]))
s_i, s_j = np.meshgrid(s_data[1:], s_data[1:])
M_mat = 1 / (s_i + s_j)
delta = np.atleast_2d(np.linalg.norm(F_sigma))
print('d', delta)
```

```
d [[0.01965714]]
```

Perform the fit

[6]:
```
model_data = {
    I.name: I_mat,
    M.name: M_mat,
    Fs.name: F_data[1:],
}
all_data = dict(**model_data, **{d.name: delta})
```

[7]:
```
fit = Fit(tikhonov_model, **all_data)
fit_result = fit.execute()
print(fit_result)
```

```
Parameter Value        Standard Deviation
a         5.449374e-02 None
Status message         Optimization terminated successfully.
Number of iterations   14
Objective              <symfit.core.objectives.LeastSquares object at 0x7fc03ac900d0>
Minimizer              <symfit.core.minimizers.BFGS object at 0x7fc03ac86a50>

Goodness of fit qualifiers:
chi_squared            3.289863264548032e-19
objective_value        1.644931632274016e-19
r_squared              -inf
```

```
/home/docs/checkouts/readthedocs.org/user_builds/symfit/envs/master/lib/python3.7/
↪site-packages/symfit/core/fit_results.py:282: RuntimeWarning: divide by zero␣
↪encountered in double_scalars
  return 1 - SS_res/SS_tot
```

Check the quality of the reconstruction

[8]:
```
ans = tikhonov_model(**model_data, **fit_result.params)
F_re = - M_mat.dot(ans.c) / fit_result.value(a)**2
```

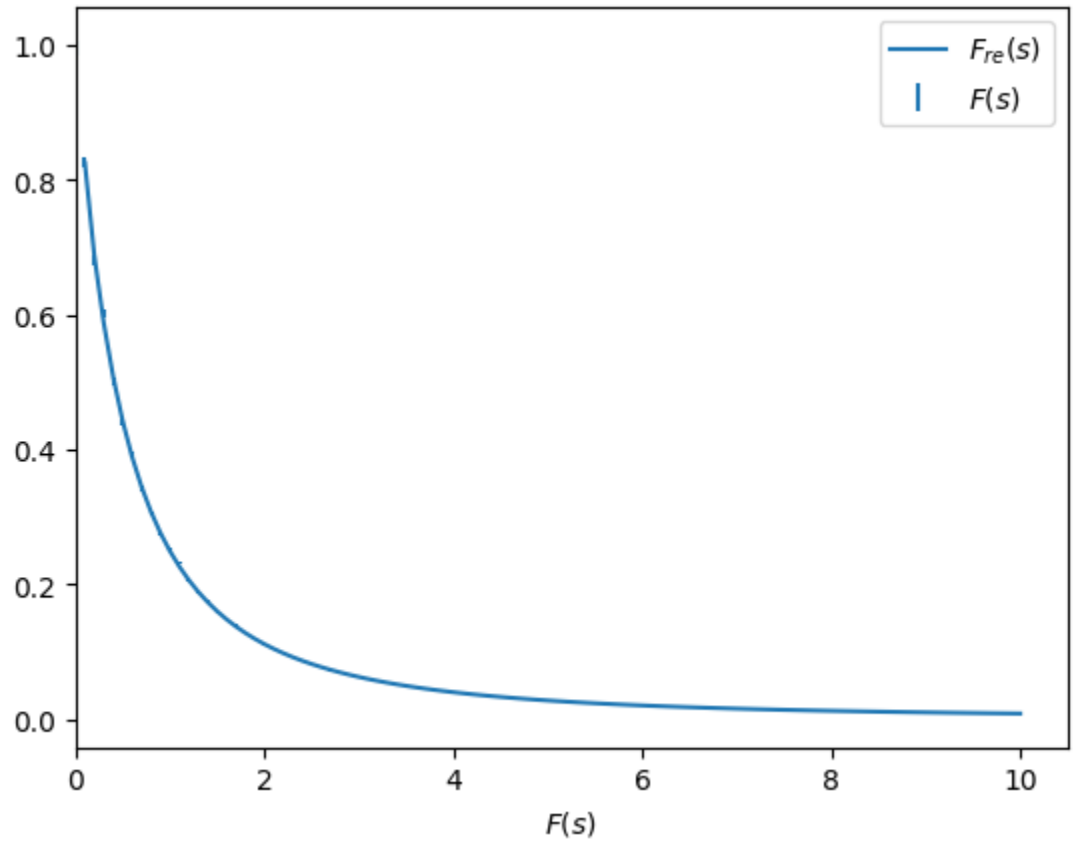
(continues on next page)

(continued from previous page)

```
print(ans.c.shape, F_re.shape)

plt.errorbar(s_data, F_data, yerr=F_sigma, label=r'$F(s)$', fmt='none')
plt.plot(s_data[1:], F_re, label=r'$F_{re}(s)$')
plt.xlabel(r'$x$')
plt.xlabel(r'$F(s)$')
plt.xlim(0, None)
plt.legend()
```

```
(100,) (100,)
```

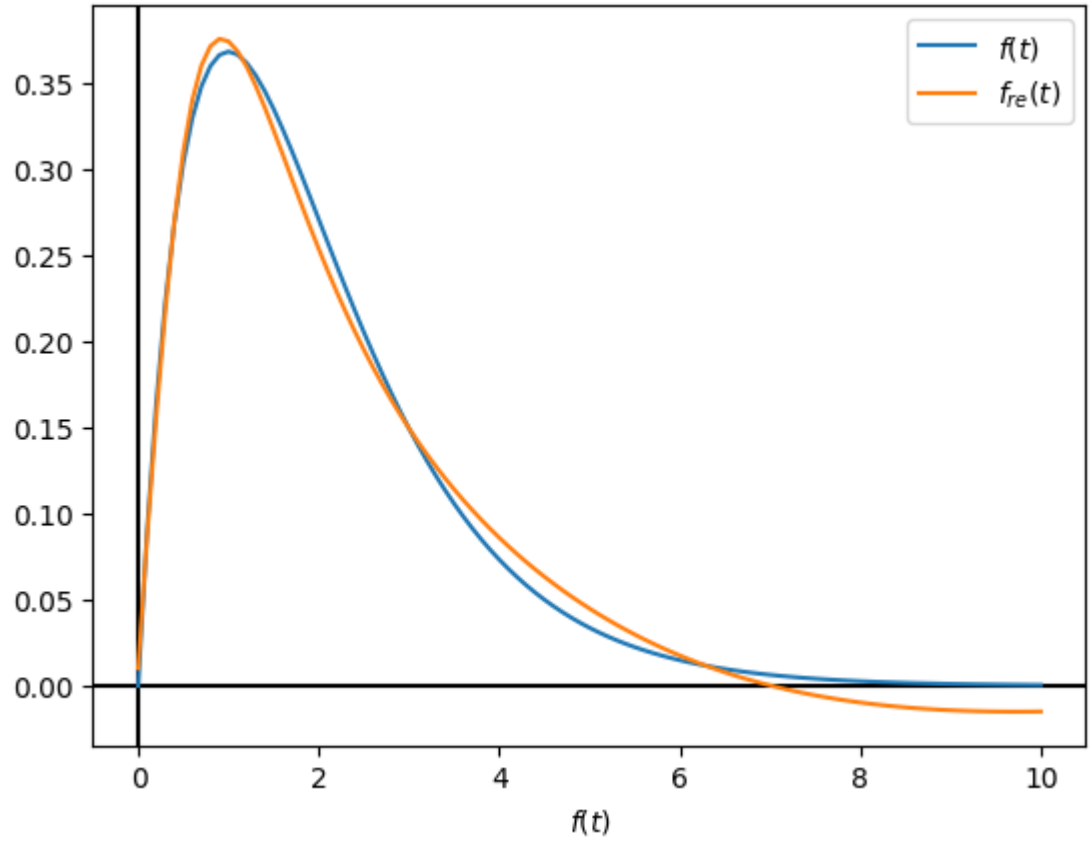
[8]:
```
<matplotlib.legend.Legend at 0x7fc03ac93c10>
```



Reconstruct $f(t)$ and compare with the known original.

[9]:
```
t_data = np.linspace(0, 10, 101)
f_data = model(t=t_data).f
f_re_func = lambda x: - np.exp(- x * s_data[1:]).dot(ans.c) / fit_result.value(a)**2
f_re = [f_re_func(t_i) for t_i in t_data]

plt.axhline(0, color='black')
plt.axvline(0, color='black')
plt.plot(t_data, f_data, label=r'$f(t)$')
plt.plot(t_data, f_re, label=r'$f_{re}(t)$')
plt.xlabel(r'$t$')
plt.xlabel(r'$f(t)$')
plt.legend()
```

[9]:
```
<matplotlib.legend.Legend at 0x7fc03acd7790>
```



Not bad, for an ill-defined problem.

8.2 Objective Examples

These are examples on how to use a different objective function from the standard least-squares objective, such as likelihood fitting.

8.2.1 Example: Likelihood fitting a Bivariate Gaussian

In this example, we shall perform likelihood fitting to a bivariate normal distribution, to demonstrate how `symfit`'s API can easily be used to perform likelihood fitting on multivariate problems.

In this example, we sample from a bivariate normal distribution with a significant correlation of $\rho = 0.6$ between x and y. We see that this is extracted from the data relatively straightforwardly.

[1]:
```
import numpy as np
from symfit import Variable, Parameter, Fit
from symfit.core.objectives import LogLikelihood
from symfit.distributions import BivariateGaussian
import matplotlib.pyplot as plt
```

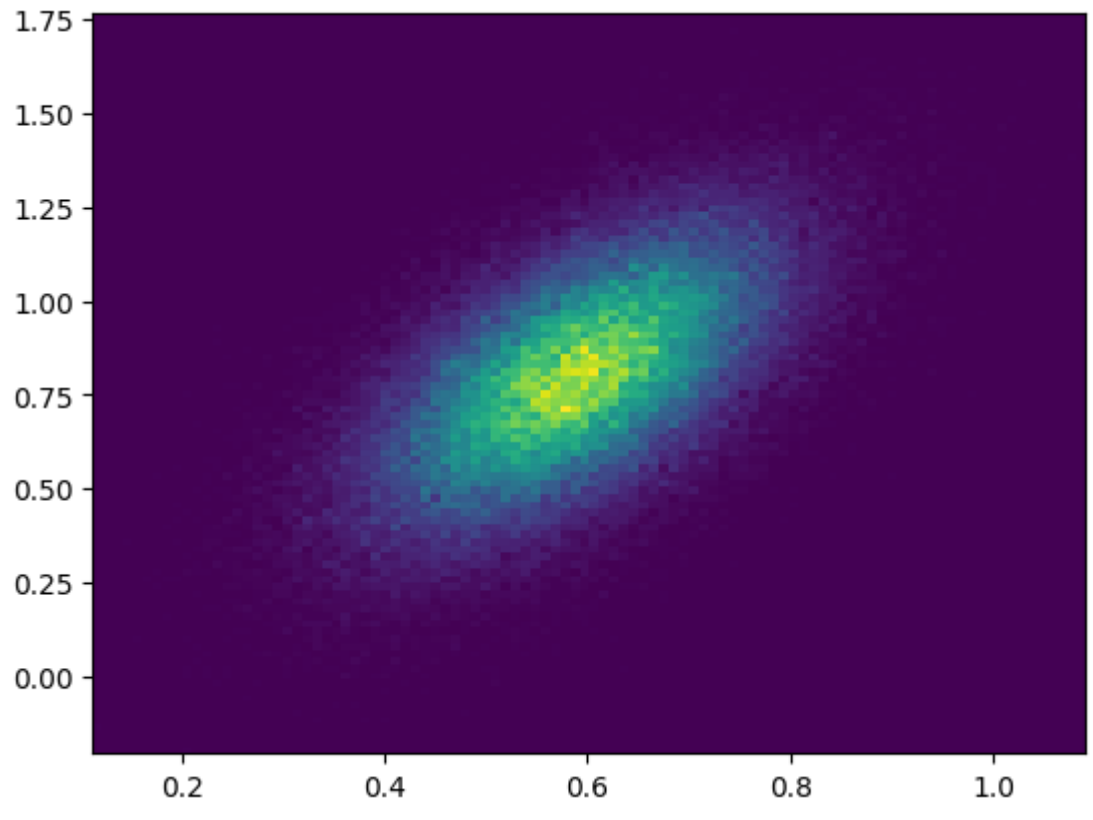
Build a model corresponding to a bivariate normal distribution.

[2]:
```
x = Variable('x')
y = Variable('y')
x0 = Parameter('x0', value=0.6, min=0.5, max=0.7)
sig_x = Parameter('sig_x', value=0.1, max=1.0)
y0 = Parameter('y0', value=0.7, min=0.6, max=0.9)
sig_y = Parameter('sig_y', value=0.05, max=1.0)
rho = Parameter('rho', value=0.001, min=-1, max=1)

pdf = BivariateGaussian(x=x, mu_x=x0, sig_x=sig_x, y=y, mu_y=y0,
                        sig_y=sig_y, rho=rho)
```

Generate mock data

[3]:
```
# Draw 100000 samples from a bivariate distribution
mean = [0.59, 0.8]
corr = 0.6
cov = np.array([[0.11 ** 2, 0.11 * 0.23 * corr],
                [0.11 * 0.23 * corr, 0.23 ** 2]])
np.random.seed(42)
xdata, ydata = np.random.multivariate_normal(mean, cov, 100000).T
```

[4]:
```
hist = plt.hist2d(xdata, ydata, bins=(100, 100))
```



Finally, we perform the fit to this mock data using the `LogLikelihood` objective.

[5]:
```
fit = Fit(pdf, x=xdata, y=ydata, objective=LogLikelihood)
fit_result = fit.execute()
print(fit_result)
```

```
/home/docs/checkouts/readthedocs.org/user_builds/symfit/envs/master/lib/python3.7/
↪site-packages/symfit/core/objectives.py:463: RuntimeWarning: divide by zero␣
↪encountered in log
  [np.nansum(np.log(component)) for component in evaluated_func]
/home/docs/checkouts/readthedocs.org/user_builds/symfit/envs/master/lib/python3.7/
↪site-packages/symfit/core/objectives.py:463: RuntimeWarning: invalid value␣
↪encountered in log
  [np.nansum(np.log(component)) for component in evaluated_func]
/home/docs/checkouts/readthedocs.org/user_builds/symfit/envs/master/lib/python3.7/
↪site-packages/symfit/core/objectives.py:491: RuntimeWarning: divide by zero␣
↪encountered in true_divide
  df / component
/home/docs/checkouts/readthedocs.org/user_builds/symfit/envs/master/lib/python3.7/
↪site-packages/symfit/core/objectives.py:491: RuntimeWarning: invalid value␣
↪encountered in true_divide
  df / component
/home/docs/checkouts/readthedocs.org/user_builds/symfit/envs/master/lib/python3.7/
↪site-packages/symfit/core/fit_results.py:259: RuntimeWarning: overflow encountered␣
↪in exp
  gof_qualifiers['likelihood'] = np.exp(gof_qualifiers['log_likelihood'])
```

```
Parameter Value        Standard Deviation
rho       6.026420e-01 2.013810e-03
sig_x     1.100898e-01 2.461684e-04
sig_y     2.303400e-01 5.150556e-04
x0        5.901317e-01 3.481346e-04
y0        8.014040e-01 7.283990e-04
Status message         CONVERGENCE: REL_REDUCTION_OF_F_<=_FACTR*EPSMCH
Number of iterations   22
Objective              <symfit.core.objectives.LogLikelihood object at 0x7ffb136b9350>
Minimizer              <symfit.core.minimizers.LBFGSB object at 0x7ffb139a3b90>

Goodness of fit qualifiers:
likelihood             inf
log_likelihood         106241.24669486462
objective_value        -106241.24669486462
```

We see that this result is in agreement with our data.

8.2.2 Example: Global Likelihood fitting

In this example, we shall perform likelihood fitting to two data sets at the same time to show how likelihood fitting can be used to fit multiple data sets at the same time.

First we will fit using a different parameter for each data set, and then with the same for both data sets. This shows the ease with which we can experiment with different models.

```
import numpy as np
from symfit import variables, parameters, Fit, exp, Model
from symfit.core.objectives import LogLikelihood

# Draw samples from a bivariate distribution
```

(continues on next page)

(continued from previous page)

```
np.random.seed(42)
data1 = np.random.exponential(5.5, 1000)
data2 = np.random.exponential(6, 2000)

# Define the model for an exponential distribution (numpy style)
a, b = parameters('a, b')
x1, y1, x2, y2 = variables('x1, y1, x2, y2')
model = Model({
    y1: (1 / a) * exp(-x1 / a),
    y2: (1 / b) * exp(-x2 / b)
})
print(model)

fit = Fit(model, x1=data1, x2=data2, objective=LogLikelihood)
fit_result = fit.execute()
print(fit_result)

# Instead, we could also fit with only one parameter to see which works best
model = Model({
    y1: (1 / a) * exp(-x1 / a),
    y2: (1 / a) * exp(-x2 / a)
})

fit = Fit(model, x1=data1, x2=data2, objective=LogLikelihood)
fit_result = fit.execute()
print(fit_result)
```

8.3 Minimizer Examples

These examples demonstrate the different minimizers available in `symfit`.

8.3.1 Global minimization: Skewed Mexican hat

In this example we will demonstrate the ease of performing global minimization using `symfit`. In order to do this we will have a look at a simple skewed mexican hat potential, which has a local minimum and a global minimum. We will then use `DifferentialEvolution` to find the global minimum.

[1]:
```
from symfit import Parameter, Variable, Model, Fit, solve, diff, N, re
from symfit.core.minimizers import DifferentialEvolution, BFGS
import numpy as np
import matplotlib.pyplot as plt
```

First we define a model for the skewed mexican hat.

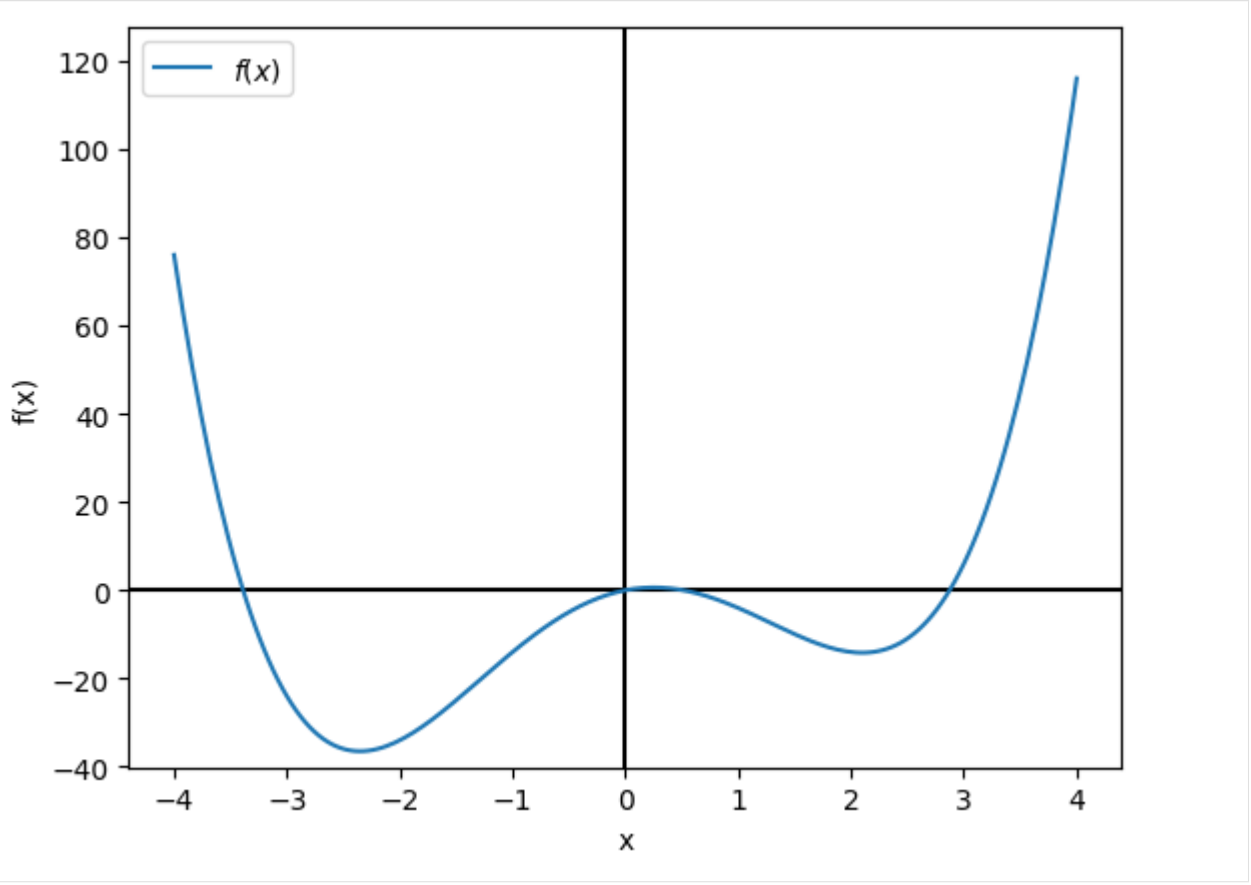
[2]:
```
x = Parameter('x')
x.min, x.max = -100, 100
y = Variable('y')
model = Model({y: x**4 - 10 * x**2 + 5 * x})  # Skewed Mexican hat
print(model)
```

```
[y(; x) = x**4 - 10*x**2 + 5*x]
```

Let us visualize what this potential looks like.

[3]:
```
xdata = np.linspace(-4, 4, 201)
ydata = model(x=xdata).y

plt.axhline(0, color='black')
plt.axvline(0, color='black')
plt.plot(xdata, ydata, label=r'$f(x)$')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.ylim(1.1 * ydata.min(), 1.1 * ydata.max())
plt.legend()
```

[3]:
```
<matplotlib.legend.Legend at 0x7f37237b1810>
```



Using `sympy`, it is easy to solve the solution analytically, by finding the places where the gradient is zero.

[4]:
```
sol = solve(diff(model[y], x), x)
# Give numerical value
sol = [re(N(s)) for s in sol]
sol
```

[4]:
```
[0.253248404857807, 2.09866205647752, -2.35191046133532]
```

Without providing any initial guesses, `symfit` finds the local minimum. This is because the initial guess is set to `1` by default.

[5]:
```
fit = Fit(model)
fit_result = fit.execute()
print('exact value', sol[1])
print('num  value ', fit_result.value(x))
```

```
exact value 2.09866205647752
num  value  2.09866205722533
```

Let's use `DifferentialEvolution` instead.

[6]:
```
fit = Fit(model, minimizer=DifferentialEvolution)
fit_result = fit.execute()
print('exact value', sol[2])
print('num  value ', fit_result.value(x))
```

```
exact value -2.35191046133532
num  value  -2.3502830932826124
```

Using `DifferentialEvolution`, we find the correct global minimum. However, it is not exactly the same as the analytical solution. This is because `DifferentialEvolution` is expensive to perform, and therefore does not solve to high precision by default. We could demand a higher precission from `DifferentialEvolution`, but this isn't worth the high computational cost. Instead, we will just tell `symfit` to perform `DifferentialEvolution`, followed by `BFGS`.

[7]:
```
fit = Fit(model, minimizer=[DifferentialEvolution, BFGS])
fit_result = fit.execute()
print('exact value', sol[2])
print('num  value ', fit_result.value(x))
```

```
exact value -2.35191046133532
num  value  -2.351910461337186
```

We see that now the proper solution has been found to much higher precision.

8.4 Interactive Guess Module

The `symfit.contrib.interactive_guess` contrib module was designed to make the process of finding initial guesses easier, by presenting the user with an interactive `matplotlib` window in which they can play around with the initial values.

8.4.1 Example: Interactive Guesses ODE

Below is an example in which the initial guesses module is used to help solve an ODE problem.

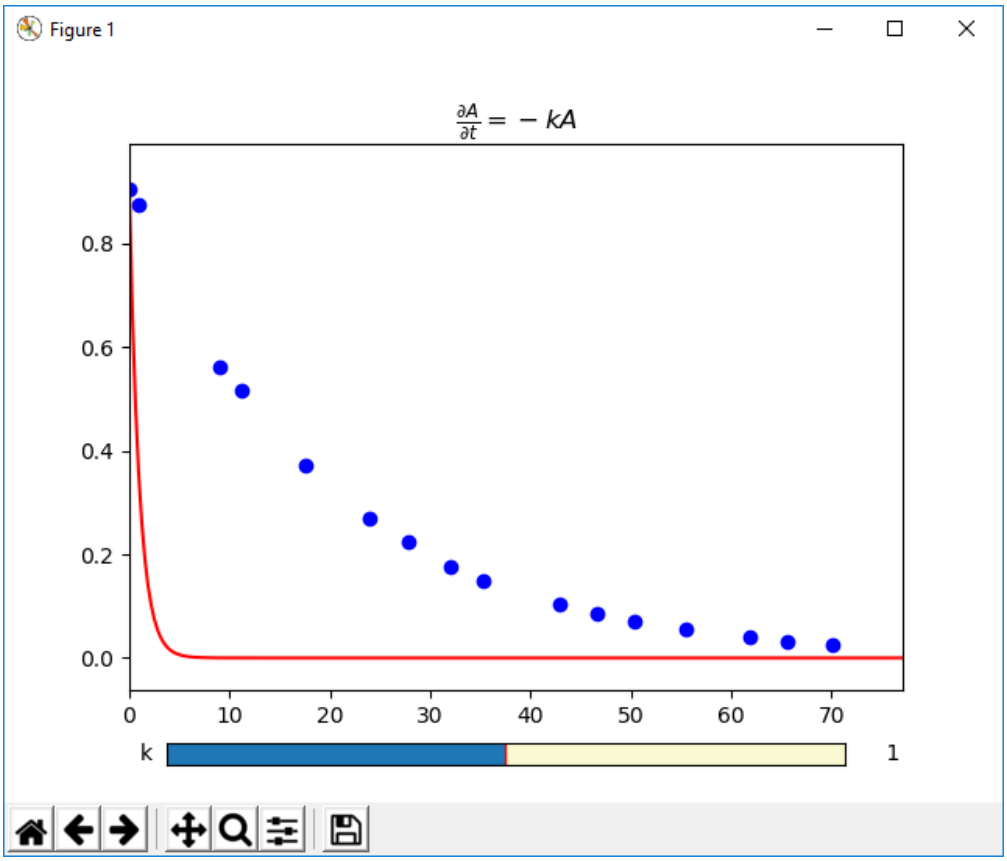
```
# SPDX-FileCopyrightText: 2014-2020 Martin Roelfs
#
# SPDX-License-Identifier: MIT

#!/usr/bin/env python3
# -*- coding: utf-8 -*-
from symfit import variables, Parameter, Fit, D, ODEModel
import numpy as np
from symfit.contrib.interactive_guess import InteractiveGuess
```

(continues on next page)

(continued from previous page)

```
# First order reaction kinetics. Data taken from
# http://chem.libretexts.org/Core/Physical_Chemistry/Kinetics/Rate_Laws/The_Rate_Law
tdata = np.array([0, 0.9184, 9.0875, 11.2485, 17.5255, 23.9993, 27.7949,
                  31.9783, 35.2118, 42.973, 46.6555, 50.3922, 55.4747, 61.827,
                  65.6603, 70.0939])
concentration = np.array([0.906, 0.8739, 0.5622, 0.5156, 0.3718, 0.2702, 0.2238,
                          0.1761, 0.1495, 0.1029, 0.086, 0.0697, 0.0546, 0.0393,
                          0.0324, 0.026])

# Define our ODE model
A, t = variables('A, t')
k = Parameter('k')
model = ODEModel({D(A, t): - k * A}, initial={t: tdata[0], A: concentration[0]})

guess = InteractiveGuess(model, A=concentration, t=tdata, n_points=250)
guess.execute()
print(guess)

fit = Fit(model, A=concentration, t=tdata)
fit_result = fit.execute()
print(fit_result)
```

This is a screenshot of the interactive guess window:



By using the sliders, you can interactively play with the initial guesses until it is close enough. Then after closing the

window, this initial value is set for the parameter, and the fit can be performed.

8.4.2 Example: Interactive Guesses Vector Model

Below is an example in which the initial guesses module is used to help solve two-component vector valued function:

```
# SPDX-FileCopyrightText: 2014-2020 Martin Roelfs
#
# SPDX-License-Identifier: MIT

# -*- coding: utf-8 -*-
from symfit import Variable, Parameter, Fit, Model
from symfit.contrib.interactive_guess import InteractiveGuess
import numpy as np


x = Variable('x')
y1 = Variable('y1')
y2 = Variable('y2')
k = Parameter('k', 900)
x0 = Parameter('x0', 1.5)

model = {
    y1: k * (x-x0)**2,
    y2: x - x0
}
model = Model(model)

# Generate example data
x_data = np.linspace(0, 2.5, 50)
data = model(x=x_data, k=1000, x0=1)
y1_data = data.y1
y2_data = data.y2

guess = InteractiveGuess(model, x=x_data, y1=y1_data, y2=y2_data, n_points=250)
guess.execute()
print(guess)

fit = Fit(model, x=x_data, y1=y1_data, y2=y2_data)
fit_result = fit.execute()
print(fit_result)
```

This is a screenshot of the interactive guess window:

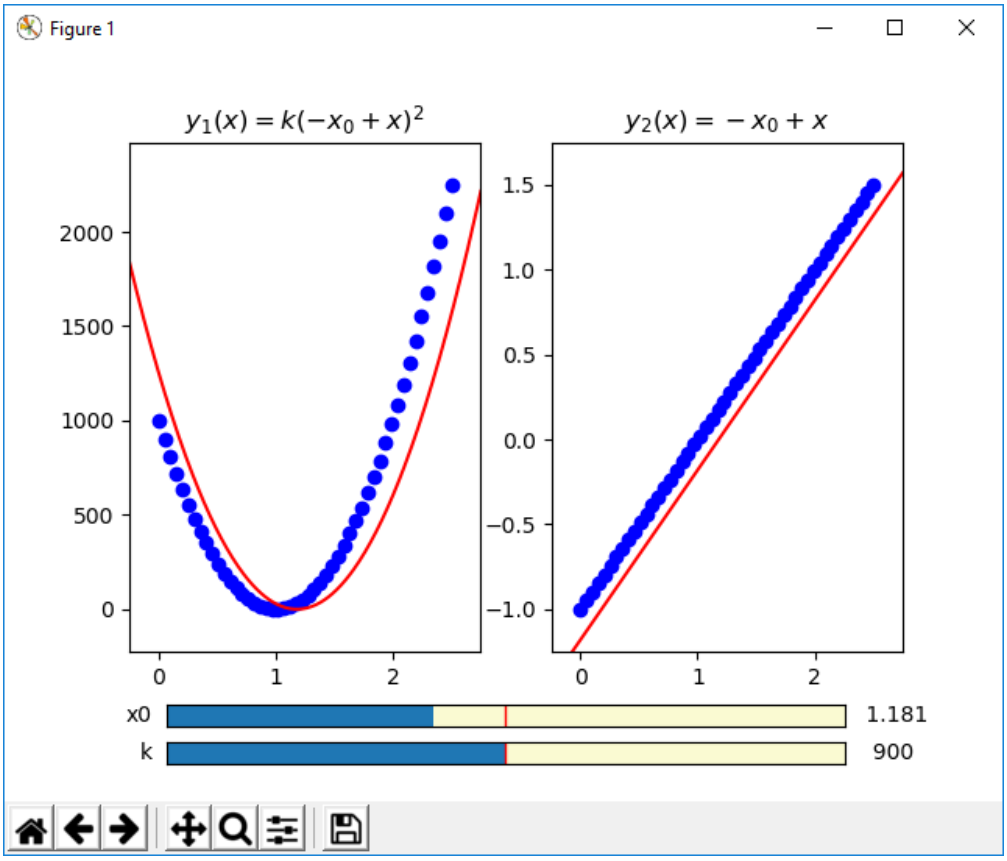
By using the sliders, you can interactively play with the initial guesses until it is close enough. Then after closing the window, this initial values are set for the parameters, and the fit can be performed.

8.4.3 Example: Interactive Guesses in N dimensions

Below is an example in which the initial guesses module is used to help fit a function that depends on more than one independent variable:

```
# SPDX-FileCopyrightText: 2014-2020 Martin Roelfs
#
# SPDX-License-Identifier: MIT
```

(continues on next page)



(continued from previous page)

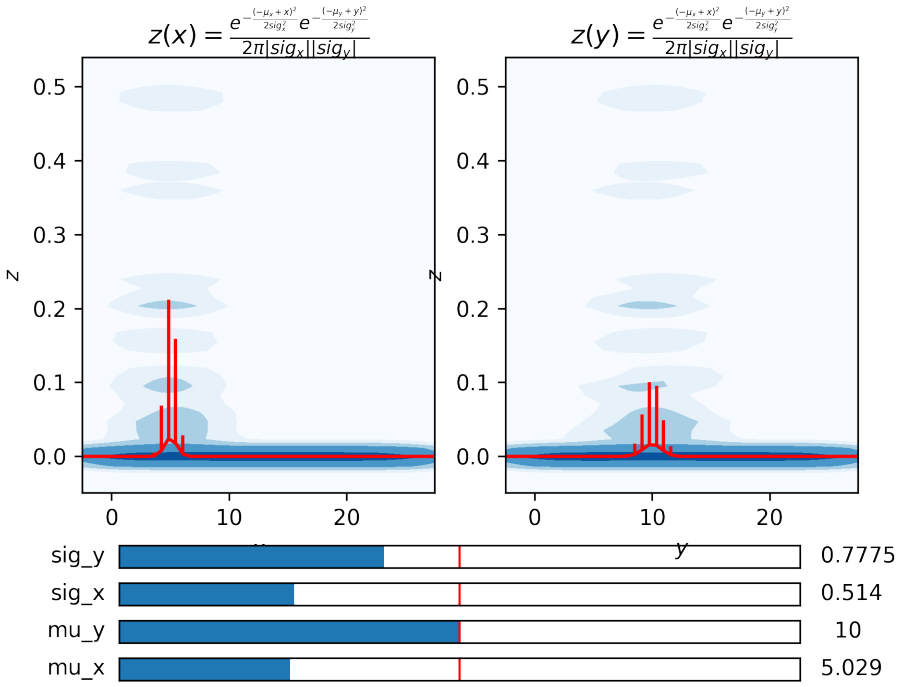
```
# -*- coding: utf-8 -*-

from symfit import variables, Parameter, exp, Fit, Model
from symfit.distributions import Gaussian
from symfit.contrib.interactive_guess import InteractiveGuess
import numpy as np


x, y, z = variables('x, y, z')
mu_x = Parameter('mu_x', 10)
mu_y = Parameter('mu_y', 10)
sig_x = Parameter('sig_x', 1)
sig_y = Parameter('sig_y', 1)



model = Model({z: Gaussian(x, mu_x, sig_x) * Gaussian(y, mu_y, sig_y)})
x_data = np.linspace(0, 25, 50)
y_data = np.linspace(0, 25, 50)
x_data, y_data = np.meshgrid(x_data, y_data)
x_data = x_data.flatten()
y_data = y_data.flatten()
z_data = model(x=x_data, y=y_data, mu_x=5, sig_x=0.3, mu_y=10, sig_y=1).z

guess = InteractiveGuess(model, x=x_data, y=y_data, z=z_data)
guess.execute()
```

(continues on next page)

(continued from previous page)

```
print(guess)

fit = Fit(model, x=x_data, y=y_data, z=z_data)
fit_result = fit.execute()
print(fit_result)
```

This is a screenshot of the interactive guess window:



In the window you can see the range the provided data spans as a contourplot on the background. The evaluated models is shown as red lines. By default your proposed model is evaluated at 50^n points for n independent variables, with 50 points per dimension. So in the example this is at 50 values of `x` and 50 values of `y`. The error bars on the points plotted are taken from the spread in `z` that comes from the spread in data in the other dimensions (`y` and `x` respectively). The error bars correspond (by default) to the 90% percentile.

By using the sliders, you can interactively play with the initial guesses until it is close enough. Then after closing the window, this initial values are set for the parameters, and the fit can be performed.

CHAPTER 9

Module Documentation

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

9.1 Fit

class symfit.core.fit.**Fit**(*model*, **ordered_data*, *objective=None*, *minimizer=None*, *constraints=None*, *absolute_sigma=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*, *objective=None*, *minimizer=None*, *constraints=None*, *absolute_sigma=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.**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

9.2 Models

class symfit.core.models.**BaseCallableModel**(*model*)
Bases: *symfit.core.models.BaseModel*

Baseclass for callable models. A callable model is expected to have implemented a *__call__* method which evaluates the model.

__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*)

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

numerical_components()

Returns A list of callables corresponding to each of the components of the model.

class symfit.core.models.**BaseGradientModel**(*model*)
Bases: *symfit.core.models.BaseCallableModel*

Baseclass for models which have a gradient. Such models are expected to implement an *eval_jacobian* function.

Any subclass of this baseclass which does not implement its own *eval_jacobian* will inherit a finite difference gradient.

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)

class symfit.core.models.**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__(*var*)
Returns the expression belonging to a given dependent variable.

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

Returns The expression belonging to 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__()

Printable representation of a Mapping model.

Returns str

classmethod as_constraint(*constraint*, *model*, *constraint_type=None*, ***init_kwargs*)

Initiate a Model which should serve as a constraint. Such a constraint-model should be initiated with knowledge of another `BaseModel`, from which it will take its parameters:

```
model = Model({y: a * x + b})
constraint = Model.as_constraint(Eq(a, 1), model)
```

`constraint.params` will be `[a, b]` instead of `[a]`.

Parameters

- **constraint** – An `Expr`, a mapping or iterable of `Expr`, or a `Relational`.
- **model** – An instance of (a subclass of) *BaseModel*.
- **constraint_type** – When `constraint` is not a `Relational`, a `Relational` has to be provided explicitly.
- **kwargs** – Any additional keyword arguments which will be passed on to the init method.

bounds

Returns List of tuples of all bounds on parameters.

connectivity_mapping

Returns This property returns a mapping of the interdepencies between variables. This is essentially the dict representation of a connectivity graph, because working with this dict results in cleaner code. Treats variables and parameters on the same footing.

free_params

Returns ordered list of the subset of variable params

function_dict

Equivalent to `self.model_dict`, but with all variables replaced by functions if applicable. Sorted by the evaluation order according to `self.ordered_symbols`, not alphabetical like `self.model_dict`!

ordered_symbols

Returns

list of all symbols in this model, topologically sorted so they can be evaluated in the correct order.

Within each group of equal priority symbols, we sort by the order of the derivative.

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.

vars_as_functions

Returns

Turn the keys of this model into `Function` objects. This is done recursively so the chain rule can be applied correctly. This is done on the basis of *connectivity_mapping*.

Example: for `{y: a * x, z: y**2 + a}` this returns `{y: y(x, a), z: z(y(x, a), a)}`.

classmethod with_dependencies(*model_expr*, *dependency_model*, ***init_kwargs*)

Initiate a model whose components depend on another model. For example:

```
>>> x, y, z = variables('x, y, z')
>>> dependency_model = Model({y: x**2})
>>> model_dict = {z: y**2}
>>> model = Model.with_dependencies(model_dict, dependency_model)
>>> print(model)
[y(x; ) = x**2,
 z(y; ) = y**2]
```

Parameters

- **model_expr** – The `Expr` or mapping/iterable of `Expr` to be turned into a model.
- **dependency_model** – An instance of (a subclass of) *BaseModel*, which contains components on which the argument `model_expr` depends.
- **init_kwargs** – Any kwargs to be passed on to the standard init method of this class.

Returns A stand-alone *BaseModel* subclass.

class symfit.core.models.**BaseNumericalModel**(*model*, *independent_vars=None*, *params=None*, *, *connectivity_mapping=None*, ***kwargs*)

Bases: *symfit.core.models.BaseModel*

ABC for Numerical Models. These are models whose components are generic python callables.

__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

__init__(*model*, *independent_vars=None*, *params=None*, *, *connectivity_mapping=None*, ***kwargs*)

Parameters

- **model** – dict of `callable`, where dependent variables are the keys. If instead of a dict a (sequence of) `callable` is provided, it will be turned into a dict automatically.
- **independent_vars** – The independent variables of the model. (Deprecated, use `connectivity_mapping` instead.)
- **params** – The parameters of the model. (Deprecated, use `connectivity_mapping` instead.)
- **connectivity_mapping** – Mapping indicating the dependencies of every variable in the model. For example, a model_dict `{y: lambda x, a, b: a * x + b}` needs a connectivity_mapping `{y: {x, a, b}}`. (Note that the values of this dict have to be sets.) This only has to be provided for the non-symbolic components. The part corresponding to the symbolic components of the model is inferred automatically.

__neg__()

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

shared_parameters

BaseNumericalModel's cannot infer if parameters are shared.

class symfit.core.models.**CallableModel**(*model*)

Bases: *symfit.core.models.BaseCallableModel*

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.

numerical_components

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

class symfit.core.models.**CallableNumericalModel**(*model*, *independent_vars=None*, *params=None*, ***, *connectivity_mapping=None*, ***kwargs*)

Bases: *symfit.core.models.BaseCallableModel*, *symfit.core.models.BaseNumericalModel*

Callable model, whose components are callables provided by the user. This allows the user to provide the components directly.

Example:

```
x, y = variables('x, y')
a, b = parameters('a, b')
numerical_model = CallableNumericalModel(
    {y: lambda x, a, b: a * x + b},
    connectivity_mapping={y: {x, a, b}}
)
```

This is identical in functionality to the more traditional:

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

but allows power-users a lot more freedom while still interacting seamlessly with the `symfit` API.

When mixing symbolical and non-symbolical components, the `connectivity_mapping` only has to be provided for the non-symbolical components, the rest are inferred automatically:

```
x, y, z = variables('x, y, z')
a, b = parameters('a, b')
model_dict = {z: lambda y, a, b: a * y + b,
              y: x ** a}
mixed_model = CallableNumericalModel(
    model_dict, connectivity_mapping={z: {y, a, b}}
)
```

class symfit.core.models.**GradientModel**(**args*, ***kwargs*)

Bases: *symfit.core.models.CallableModel*, *symfit.core.models.BaseGradientModel*

Analytical model which has an analytically computed Jacobian.

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

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.

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

Returns Jacobian evaluated at the specified point.

jacobian

Returns Jacobian 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. The return shape is a list over the models components, filled with tha symbolical jacobian for that component, as a list.

class symfit.core.models.**HessianModel**(**args*, ***kwargs*)

Bases: *symfit.core.models.GradientModel*

Analytical model which has an analytically computed Hessian.

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

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.

eval_hessian(**args*, ***kwargs*)

Returns Hessian evaluated at the specified point.

hessian

Returns Hessian filled with the symbolic expressions for all the second order partial derivatives. Partial derivatives are taken with respect to the Parameter's, not the independent Variable's.

class symfit.core.models.**Model**(**args*, ***kwargs*)

Bases: *symfit.core.models.HessianModel*

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

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

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

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

The output of a call to a model is a special kind of namedtuple:

```
>>> a(x=3)
Ans(y=9)
```

When turning this into a dict, however, the dict keys will be Variable objects, not strings:

```
>>> a(x=3)._asdict()
OrderedDict(((y, 9),))
```

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

exception symfit.core.models.**ModelError**

Bases: Exception

Raised when a problem occurs with a model.

class symfit.core.models.**ModelOutput**(*variables*, *output*)

Bases: tuple

Object to hold the output of a model call. It mimics a `collections.namedtuple()`, but is initiated with *Variable* objects instead of strings.

Its information can be accessed using indexing or as attributes:

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

>>> ans = model(x=2, a=1, b=3)
>>> print(ans)
ModelOutput(variables=[y], output=[5])
>>> ans[0]
5
>>> ans.y
5
```

__getitem__(*key*)

Return self[key].

__init__(*variables*, *output*)

`variables` and `output` need to be in the same order!

Parameters

- **variables** – The variables corresponding to `output`.
- **output** – The output of a call which should be mapped to `variables`.

__len__()

Return len(self).

static __new__(*self*, *variables*, *output*)
`variables` and `output` need to be in the same order!

Parameters

- **variables** – The variables corresponding to `output`.
- **output** – The output of a call which should be mapped to `variables`.

__repr__()
Return repr(self).

exception symfit.core.models.**ODEError**
Bases: `Exception`

class symfit.core.models.**ODEModel**(*model_dict*, *initial*, ***integrator_kwargs*)
Bases: *symfit.core.models.BaseGradientModel*

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*, ***integrator_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}
- **integrator_kwargs** – kwargs to pass to the ODE integrator. See scipy's solve_ivp for more info.

__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

system
System of functions to be integrated

symfit.core.models.**hessian_from_model**(*model*)
Build a *CallableModel* representing the Hessian of `model`.

This function make sure the chain rule is correctly applied for interdependent variables.

Parameters **model** – Any symbolical model-type.

Returns *CallableModel* representing the Hessian of `model`.

symfit.core.models.**jacobian_from_model**(*model*, *as_functions=False*)

Build a *CallableModel* representing the Jacobian of `model`.

This function make sure the chain rule is correctly applied for interdependent variables.

Parameters

- **model** – Any symbolical model-type.
- **as_functions** – If *True*, the result is returned using `sympy.core.function.Function` where needed, e.g. `{y(x, a): a * x}` instead of `{y: a * x}`.

Returns *CallableModel* representing the Jacobian of `model`.

9.3 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*, ***assumptions*)
Create a new `Argument`. See `Symbol` for more information.

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.

__eq__(*other*)

Parameters are considered equal when their name, assumptions, and bounds are considered the same.

__hash__() → int

Return hash(self).

__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*, *value=1.0*, *min=None*, *max=None*, *fixed=False*, ***kwargs*)

Create a new `Argument`. See `Symbol` for more information.

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

Bases: *symfit.core.argument.Argument*

Variable type.

9.4 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.

9.5 Fit Results

`class` `symfit.core.fit_results.`**`FitResults`**(*model*, *popt*, *covariance_matrix*, *minimizer*, *objective*, *message*, *, *constraints=None*, ***minimizer_output*)

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 - objective and minimizer used.

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*, *minimizer*, *objective*, *message*, *, *constraints=None*, ***minimizer_output*)

Parameters

- **model** – *Model* that was fit to.
- **popt** – best fit parameters, same ordering as in model.params.
- **covariance_matrix** – covariance matrix.
- **minimizer** – Minimizer instance used.
- **objective** – Objective function which was optimized.
- **message** – Status message returned by the minimizer.
- ****minimizer_output** – Raw output as given by the minimizer.

__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.

symfit.core.fit_results.**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.

9.6 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.GlobalMinimizer*

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
    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*, *symfit.core.minimizers.BaseMinimizer*

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

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__()*.

__str__()
Return str(self).

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.ScipyBoundedMinimizer*, *symfit.core.minimizers.GlobalMinimizer*

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

execute(*, *strategy='rand1bin'*, *popsize=40*, *mutation=(0.423*, *1.053)*, *recombination=0.95*, *polish=False*, *init='latinhypercube'*, ***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.**HessianMinimizer**(**args*, *hessian=None*, ***kwargs*)
Bases: *symfit.core.minimizers.GradientMinimizer*

ABC for Minimizers that support the use of a Hessian.

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

Parameters

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

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

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

Returns Hessian corresponding to free parameters only.

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

Wrapper around `scipy.optimize.minimize()`'s LBFGSB 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.**MINPACK**(**args*, ***kwargs*)
Bases: *symfit.core.minimizers.ScipyBoundedMinimizer*, *symfit.core.minimizers.GradientMinimizer*

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

execute(*jacobian=None*, *method='trf'*, ***minpack_options*)

Parameters ****minpack_options** – Any named arguments to be passed to scipy.optimize.least_squares()

resize_jac(*func*)
Removes values with identical indices to fixed parameters from the output of func. func has to return the jacobian of the residuals. This method is different from the one in GradientMinimizer, since least_squares expects the jacobian to return an MxN (M=len(data), N=len(params)) matrix, rather than a vector.

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

Returns Jacobian corresponding to non-fixed parameters only.

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.**Powell**(**args*, ***kwargs*)
Bases: *symfit.core.minimizers.ScipyMinimize*, *symfit.core.minimizers.BaseMinimizer*

Wrapper around scipy.optimize.minimize()'s Powell algorithm.

class symfit.core.minimizers.**SLSQP**(**args*, ***kwargs*)
Bases: *symfit.core.minimizers.ScipyGradientMinimize*, *symfit.core.minimizers.ScipyConstrainedMinimize*, *symfit.core.minimizers.ScipyBoundedMinimizer*

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

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

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

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.**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.

Parameters **constraints** – List of either MinimizeModel instances (this is what is provided by *Fit*), *BaseModel*, or `sympy.core.relational.Relational`.

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.

execute(*, *jacobian=None*, ***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.**ScipyHessianMinimize**(**args*, ***kwargs*)
Bases: *symfit.core.minimizers.ScipyGradientMinimize*, *symfit.core.minimizers.HessianMinimizer*

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

execute(*, *hessian=None*, ***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*, *hessian=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.

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.**TrustConstr**(**args*, ***kwargs*)
Bases: *symfit.core.minimizers.ScipyHessianMinimize*, *symfit.core.minimizers.ScipyConstrainedMinimize*, *symfit.core.minimizers.ScipyBoundedMinimizer*

Wrapper around scipy.optimize.minimize()'s Trust-Constr algorithm.

execute(***, *jacobian=None*, *hessian=None*, *options=None*, ***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:

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

Parameters **constraints** – List of either MinimizeModel instances (this is what is provided by *Fit*), *BaseModel*, or sympy.core.relational.Relational.

Returns dict of scipy compatible statements.

9.7 Objectives

Objective functions are the functions which are minimized by the *minimizers*. Famous examples are least squares, log-likelihood, or minimizing the model itself.

symfit provides objective functions for those cases by default. Custom objectives can also be created, for example by inheriting from *BaseObjective*, *GradientObjective* or *HessianObjective*.

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

Some objective functions dependent only on independent variables, not dependent and sigma variables. In this case, sanity checking is greatly simplified.

dependent_data

Returns Empty OrderedDict.

Return type collections.OrderedDict

sigma_data

Returns Empty OrderedDict.

Return type collections.OrderedDict

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

ABC for objective functions. Implements basic data handling.

__call__(*ordered_parameters=[]*, ***parameters*)
Evaluate the objective function for given parameter values.

Parameters

- **ordered_parameters** – List of parameter, in alphabetical order. Typically provided by the minimizer.
- **parameters** – parameters as keyword arguments.

Returns evaluated model.

__eq__(*other*)
Objectives are considered equal if they are of the same type, have the same model, and the same data.

__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(*ordered_parameters=[]*, ***parameters*)
Evaluate the jacobian for given parameter values.

Parameters

- **ordered_parameters** – List of parameter, in alphabetical order. Typically provided by the minimizer.
- **parameters** – parameters as keyword arguments.

Returns evaluated jacobian

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

ABC for objectives that support hessian methods.

eval_hessian(*ordered_parameters=[]*, ***parameters*)
Evaluate the hessian for given parameter values.

Parameters

- **ordered_parameters** – List of parameter, in alphabetical order. Typically provided by the minimizer.
- **parameters** – parameters as keyword arguments.

Returns evaluated hessian

class symfit.core.objectives.**HessianObjectiveJacApprox**(*model*, *data*)
Bases: *symfit.core.objectives.HessianObjective*

This object should only be used as a Mixin for covariance matrix estimation. Since the covariance matrix for the least-squares method is proportional to the Hessian of S, this function attempts to return the Hessian upon calculating `eval_hessian`.

However, if the model does not have a Hessian defined through `eval_hessian`, we approximate the Hessian as $J^T \cdot J$, where J is the Jacobian of the model. This approximation is valid when, amongst other things, the residuals are sufficiently small. It can therefore only be used after fitting, not during.

An objective which inherits from this object, will return zeros with the shape of the hessian of the model, when `eval_hessian` is called. This code injection will therefore result in the terms proportional to the hessian of the model dropping out, which leaves the famous $J^T \cdot J$ approximation.

eval_hessian(*ordered_parameters=[]*, ***parameters*)

Returns Zeros with the shape of the Hessian of the model.

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

Objective representing the least-squares deviation of a model, defined as $S = \frac{1}{2}\sum_i \sum_{x_i} \frac{r_i(x_i,\vec{p})^2}{\sigma_i(x_i)^2}$, where i ranges over all components of the model, $r_i(x_i, \vec{p})$ is the residue of the i-th component, x_i indicates all the data associated with the i-th component, and $\sigma_i(x_i)$ indicates the associated standard deviations.

The data for each component does not have to be the same, and it does not have to have the same shape. The only thing that matters is that within each component the shapes have to be compatible.

__call__(*ordered_parameters=[]*, ***, *flatten_components=True*, ***parameters*)

Parameters

- **ordered_parameters** – See `parameters`.
- **parameters** – values of the *Parameter*'s to evaluate S at.
- **flatten_components** – if *True*, return the total S. If *False*, return the S per component of the *BaseModel*.

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

eval_hessian(*ordered_parameters=[]*, ***parameters*)
Hessian of S in the *Parameter*'s ($\nabla_{\vec{p}^2} S$).

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

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

eval_jacobian(*ordered_parameters=[]*, ***parameters*)
Jacobian of S in the *Parameter*'s ($\nabla_{\vec{p}} S$).

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

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

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

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

__call__(*ordered_parameters=[]*, ***parameters*)

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

Returns scalar value of log-likelihood

eval_hessian(*ordered_parameters=[]*, ***parameters*)
Hessian for log-likelihood is defined as $\nabla^2_{\vec{p}}(\log(L(\vec{p}|\vec{x})))$.

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

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

eval_jacobian(*ordered_parameters=[]*, ***, *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.HessianObjective*, *symfit.core.objectives.BaseIndependentObjective*

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

__call__(*ordered_parameters=[]*, ***parameters*)
Evaluate the objective function for given parameter values.

Parameters

- **ordered_parameters** – List of parameter, in alphabetical order. Typically provided by the minimizer.
- **parameters** – parameters as keyword arguments.

Returns evaluated model.

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

Parameters

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

eval_hessian(*ordered_parameters=[]*, ***parameters*)
Evaluate the hessian for given parameter values.

Parameters

- **ordered_parameters** – List of parameter, in alphabetical order. Typically provided by the minimizer.
- **parameters** – parameters as keyword arguments.

Returns evaluated hessian

eval_jacobian(*ordered_parameters=[]*, ***parameters*)
Evaluate the jacobian for given parameter values.

Parameters

- **ordered_parameters** – List of parameter, in alphabetical order. Typically provided by the minimizer.
- **parameters** – parameters as keyword arguments.

Returns evaluated jacobian

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__(*ordered_parameters=[]*, ***, *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 **flatten_components** – If True, summing is performed over the data indices (default).

Returns $\sqrt(\chi^2)$

eval_jacobian(*ordered_parameters=[]*, ***parameters*)
Evaluate the jacobian for given parameter values.

Parameters

- **ordered_parameters** – List of parameter, in alphabetical order. Typically provided by the minimizer.
- **parameters** – parameters as keyword arguments.

Returns evaluated jacobian

9.8 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.**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**(**args*, ***kwargs*)
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.

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

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.**name**(*self*)
Save name which can be used for alphabetic sorting and can be turned into a kwarg.

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*, *args*)

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
- **args** – variables and parameters in this model

Returns lambda function to be used for numerical evaluation of the model.

`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

9.9 Printing

`symfit` occasionally updates the printing of `sympy` objects, such that they print into their `numpy`/`scipy` equivalent. This is done because sometimes such printing has not been implemented in `sympy` yet, or because we want slightly different behavior from the standard one.

Users using both `symfit` and `sympy` should be aware of this.

9.10 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.`**`BivariateGaussian`**(*x*, *y*, *mu_x*, *mu_y*, *sig_x*, *sig_y*, *rho*)

Bivariate Gaussian pdf.

Parameters

- **x** – *symfit.core.argument.Variable*
- **y** – *symfit.core.argument.Variable*
- **mu_x** – *symfit.core.argument.Parameter* for the mean of *x*
- **mu_y** – *symfit.core.argument.Parameter* for the mean of *y*
- **sig_x** – *symfit.core.argument.Parameter* for the standard deviation of *x*
- **sig_y** – *symfit.core.argument.Parameter* for the standard deviation of *y*
- **rho** – *symfit.core.argument.Parameter* for the correlation between *x* and *y*.

Returns sympy expression for a Bivariate Gaussian pdf.

`symfit.distributions.`**`Exp`**(*x*, *l*)

$$f(x) = le^{-lx}$$

Exponential Distribution pdf.

Parameters

- **x** – free variable.
- **l** – rate parameter.

Returns sympy.Expr for an Exponential Distribution pdf.

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

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Gaussian pdf.

Parameters

- **x** – free variable.
- **mu** – mean of the distribution.

- **sig** – standard deviation of the distribution.

Returns sympy.Expr for a Gaussian pdf.

9.11 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.**InteractiveGuess**(**args*, *n_points=50*, *log_contour=True*, *percentile=(5, 95)*, ***kwargs*)

Bases: *symfit.core.fit.TakesData*

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

__init__(**args*, *n_points=50*, *log_contour=True*, *percentile=(5, 95)*, ***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 a blue contour plot, the proposed model as a red line. The errorbars on the proposed model represent the percentile of data within the thresholds.

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. Defaults to 50.
- **log_contour** (*bool*) – Whether to plot the contour plot of the log of the density, rather than the density itself. If True, any density less than 1e-7 will be considered 0. Defaults to True.
- **percentile** (*list*) – Controls the errorbars on the proposed model, such that the lower errorbar will cover percentile[0]% of the data, and the upper will cover percentile[1]%. Defaults to [5, 95], with corresponds to a 90% percentile. Should be a list of 2 numbers.

__str__()

Represent the guesses in a human readable way.

Returns string with the guessed values.

execute(*, *show=True*, *block=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

Any additional keyword arguments are passed to matplotlib.pyplot.show().

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

Bases: *symfit.contrib.interactive_guess.interactive_guess.InteractiveGuess*

__init__(**args*, ***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 a blue contour plot, the proposed model as a red line. The errorbars on the proposed model represent the percentile of data within the thresholds.

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. Defaults to 50.
- **log_contour** (*bool*) – Whether to plot the contour plot of the log of the density, rather than the density itself. If True, any density less than 1e-7 will be considered 0. Defaults to True.
- **percentile** (*list*) – Controls the errorbars on the proposed model, such that the lower errorbar will cover percentile[0]% of the data, and the upper will cover percentile[1]%. Defaults to [5, 95], with corresponds to a 90% percentile. Should be a list of 2 numbers.

class symfit.contrib.interactive_guess.interactive_guess.**Strategy2D**(*interactive_guess*)

Bases: object

A strategy that describes how to plot a model that depends on a single independent variable, and how to update that plot.

__init__(*interactive_guess*)

Initialize self. See help(type(self)) for accurate signature.

plot_data(*proj*, *ax*)

Creates and plots a scatter plot of the original data.

plot_model(*proj*, *ax*)

Plots the model proposed for the projection proj on ax.

update_plot(*indep_var*, *dep_var*)

Updates the plot of the proposed model.

class symfit.contrib.interactive_guess.interactive_guess.**StrategynD**(*interactive_guess*)

Bases: object

A strategy that describes how to plot a model that depends on a multiple independent variables, and how to update that plot.

__init__(*interactive_guess*)

Initialize self. See help(type(self)) for accurate signature.

plot_data(*proj*, *ax*)

Creates and plots the contourplot of the original data. This is done by evaluating the density of projected datapoints on a grid.

plot_model(*proj*, *ax*)

Plots the model proposed for the projection proj on ax.

update_plot(*indep_var*, *dep_var*)

Updates the plot of the proposed model.

CHAPTER 10

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

Python Module Index

S

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W