msl-nlf Documentation

Release 5.44.0.dev0+c31f6f7

Measurement Standards Laboratory of New Zealand

Oct 26, 2023

CONTENTS

MSL-NLF is a Python wrapper around the non-linear fitting software that was written by P. Saunders at the Measurement Standards Laboratory (MSL) of New Zealand. The original source code is written in Delphi and compiled to a shared library (DLL) to be accessed by other programming languages. The Delphi software also provides a GUI to visualize and interact with the fitted data. Please contact someone from MSL if you are interested in using the GUI version.

The application of non-linear fitting is a general-purpose curving-fitting program that will fit any curve of the form

$$y = f(x_1, x_2, ..., x_n, a_1, a_2, ..., a_N),$$

to a set of data, where $x_1, x_2, ..., x_n$, are real variables, which may or may not be correlated, and $a_1, a_2, ..., a_N$ are real parameters. In the Delphi algorithm, n can be any value from 1 to 30 and N from 1 to 99. The function f will be fitted to a set of M data points, $(x_{1,1}, x_{2,1}, ..., x_{n,1}, y_1)$, $(x_{1,2}, x_{2,2}, ..., x_{n,2}, y_2)$, ..., $(x_{1,M}, x_{2,M}, ..., x_{n,M}, y_M)$, where $M \geq N$. The parameters $a_1, a_2, ..., a_N$ can be chosen to be either constant or fitted, providing additional flexibility to the fitting process. If there are constant parameters, then M must be greater than or equal to the number of non-constant parameters.

For more details see *Propagation of uncertainty for non-linear calibration equations with an application in radiation thermometry*, **P. Saunders**, Metrologia 40 93 (2003).

The non-linear fitting algorithm implements the following features:

1. perform an unweighted fit or a weighted fit with uncertainties in the x and/or y data
2. setting correlations between the $x_i - x_i$, $x_i - x_j$, $x_i - y$ and/or $y - y$ data
3. whether second-derivative terms (Hessian) are included in the calculation
4. use up to 30 independent variables and up to 99 parameters in the fit equation
5. whether a parameter is held constant or allowed to vary during the fitting process
6. choice of different fitting methods (see the `FitMethod` enum)

Follow the *Install* instructions and read the *Getting Started* guide to begin.

CHAPTER

ONE

CONTENTS

1.1 Install

Note: A PyPI release is not available yet. You can install from the main branch for now.

The DLL files have been compiled for use on Windows and therefore only Windows is supported.

To install MSL-NLF run

```
pip install msl-nlf
```

Alternatively, using the MSL Package Manager run

```
msl install nlf
```

1.1.1 Dependencies

- Python 3.8+
- numpy

Optional Dependencies

The GUM Tree Calculator, GTC, is not automatically installed when MSL-NLF is installed, but it is required to create a correlated ensemble of uncertain real numbers from a `Result`.

To automatically include GTC when installing MSL-NLF run

```
pip install msl-nlf[gtc]
```

1.2 Getting Started

As a simple example, one might need to model data that has a linear relationship

```
>>> x = [1.6, 3.2, 5.5, 7.8, 9.4]
>>> y = [7.8, 19.1, 17.6, 33.9, 45.4]
```

The first task to perform is to create a *Model* and specify the fit equation as a string (see the documentation of *Model* for an overview of what arithmetic operations and functions are allowed in the equation)

```
>>> from msl.nlf import Model
>>> model = Model('a1+a2*x')
```

Provide an initial guess for the parameters (*a1* and *a2*) and apply the fit

```
>>> result = model.fit(x, y, params=[1, 1])
>>> result.params
ResultParameters(
  ResultParameter(name='a1', value=0.522439024..., uncert=5.132418149...,␣
↪label=None),
  ResultParameter(name='a2', value=4.406829268..., uncert=0.827701724...,␣
↪label=None)
)
```

The *Result* object that is returned from the fit contains information about the fit result, such as the chi-square value and the covariance matrix, but we simply showed a summary of the fit parameters above.

1.2.1 Input Parameters

If you want to have control over which parameters should be held constant during the fitting process and which are allowed to vary or if you want to assign a label to a parameter, you need to create an *InputParameters* instance.

In this case, we will use one of the built-in *models*, *LinearModel*, to perform the linear fit and create *InputParameters*. We use the *InputParameters* instance to provide an initial value for each parameter, define labels, and set whether the initial value of a parameter is held constant during the fitting process

```
>>> from msl.nlf import LinearModel
>>> model = LinearModel()
>>> model.equation
'a1+a2*x'
>>> params = model.create_parameters()
>>> a1 = params.add(name='a1', value=0, constant=True, label='intercept')
>>> params['a2'] = 1, False, 'slope'  # alternative way to add a parameter
>>> result = model.fit(x, y, params=params)
>>> result.params
ResultParameters(
  ResultParameter(name='a1', value=0.0, uncert=0.0, label='intercept'),
  ResultParameter(name='a2', value=4.4815604681..., uncert=0.3315980376...,␣
```

(continues on next page)

(continued from previous page)

```
↪label='slope')
)
```

We showed above that calling *create_parameters()* is one way to create an *InputParameters* instance. It can also be instantiated directly

```
>>> from msl.nlf import InputParameters
>>> params = InputParameters()
```

There are multiple ways to add a parameter to an *InputParameters* object. To add a parameter, you could explicitly add an instance of an *InputParameter* using the *add()* method (or as one would add items to a `dict`)

```
>>> from msl.nlf import InputParameter
>>> a1 = params.add(InputParameter('a1', 1))
>>> a2 = params.add(InputParameter('a2', 2, constant=True))
>>> a3 = params.add(InputParameter('a3', 3, constant=True, label='label-3'))
>>> params['a4'] = InputParameter('a4', 4)
```

You could also specify multiple positional arguments (or assign several parameters using the mapping syntax)

```
>>> a5 = params.add('a5', 5)
>>> a6 = params.add('a6', 6, True)
>>> a7 = params.add('a7', 7, False, 'label-7')
>>> params['a8'] = 8
>>> params['a9'] = 9, True
>>> params['a10'] = 10, True, 'label-10'
```

or you could specify keyword arguments (or set it equal to a `dict`)

```
>>> a11 = params.add(name='a11', value=11)
>>> a12 = params.add(name='a12', value=12, constant=True)
>>> a13 = params.add(name='a13', value=13, label='label-13')
>>> a14 = params.add(name='a14', value=14, constant=False, label='label-14')
>>> params['a15'] = {'value': 15}
>>> params['a16'] = {'value': 16, 'constant': True}
>>> params['a17'] = {'value': 17, 'label': 'label-17'}
>>> params['a18'] = {'value': 18, 'constant': False, 'label': 'label-18'}
```

There is an *add_many()* method as well.

Here, we iterate through the collection of input parameters to see what it contains

```
>>> for param in params:
...     print(param)
InputParameter(name='a1', value=1.0, constant=False, label=None)
InputParameter(name='a2', value=2.0, constant=True, label=None)
InputParameter(name='a3', value=3.0, constant=True, label='label-3')
InputParameter(name='a4', value=4.0, constant=False, label=None)
InputParameter(name='a5', value=5.0, constant=False, label=None)
```

(continues on next page)

(continued from previous page)

```
InputParameter(name='a6', value=6.0, constant=True, label=None)
InputParameter(name='a7', value=7.0, constant=False, label='label-7')
InputParameter(name='a8', value=8.0, constant=False, label=None)
InputParameter(name='a9', value=9.0, constant=True, label=None)
InputParameter(name='a10', value=10.0, constant=True, label='label-10')
InputParameter(name='a11', value=11.0, constant=False, label=None)
InputParameter(name='a12', value=12.0, constant=True, label=None)
InputParameter(name='a13', value=13.0, constant=False, label='label-13')
InputParameter(name='a14', value=14.0, constant=False, label='label-14')
InputParameter(name='a15', value=15.0, constant=False, label=None)
InputParameter(name='a16', value=16.0, constant=True, label=None)
InputParameter(name='a17', value=17.0, constant=False, label='label-17')
InputParameter(name='a18', value=18.0, constant=False, label='label-18')
```

or just get all of the values

```
>>> params.values()
array([ 1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.,  10.,  11.,  12.,  13.,
       14., 15., 16., 17., 18.])
```

You can get a specific parameter by its *name* or *label* (provided that the *label* is not `None`)

```
>>> params['a3']
InputParameter(name='a3', value=3.0, constant=True, label='label-3')
>>> params['label-14']
InputParameter(name='a14', value=14.0, constant=False, label='label-14')
```

and you can update a parameter by specifying its *name* or *label* to the *update()* method

```
>>> params.update('a1', value=5.3, label='intercept')
>>> params['a1']
InputParameter(name='a1', value=5.3, constant=False, label='intercept')

>>> params.update('label-7', value=1e3, constant=True, label='amplitude')
>>> params['a7']
InputParameter(name='a7', value=1000.0, constant=True, label='amplitude')
```

or you can update a parameter by directly modifying an attribute

```
>>> a1.label = 'something-new'
>>> a1.constant = False
>>> a1.value = -3.2
>>> params['a1']
InputParameter(name='a1', value=-3.2, constant=False, label='something-new')

>>> params['label-3'].label = 'fwhm'
>>> params['fwhm'].constant = True
>>> params['fwhm'].value = 0.03
>>> params['a3']
InputParameter(name='a3', value=0.03, constant=True, label='fwhm')
```

1.2.2 Debugging (Input)

If you call the *fit()* method with *debug=True* the fit function in the DLL is not called and an *Input* object is returned that contains the information that would have been sent to the fit function in the DLL

```
>>> model = LinearModel()
>>> info = model.fit(x, y, params=[1, 1], debug=True)
>>> info.weighted
False
>>> info.fit_method
<FitMethod.LM: 'Levenberg-Marquardt'>
>>> info.x
array([[1.6, 3.2, 5.5, 7.8, 9.4]])
```

You can display a summary of the input information

```
>>> info
Input(
  absolute_residuals=True
  correlated=False
  correlations=
    Correlations(
      data=[]
      is_correlated=[[False False]
                     [False False]]
    )
  delta=0.1
  equation='a1+a2*x'
  fit_method=<FitMethod.LM: 'Levenberg-Marquardt'>
  max_iterations=999
  params=
    InputParameters(
      InputParameter(name='a1', value=1.0, constant=False, label=None),
      InputParameter(name='a2', value=1.0, constant=False, label=None)
    )
  residual_type=<ResidualType.DY_X: 'dy v x'>
  second_derivs_B=True
  second_derivs_H=True
  tolerance=1e-20
  ux=[[0. 0. 0. 0. 0.]]
  uy=[0. 0. 0. 0. 0.]
  uy_weights_only=False
  weighted=False
  x=[[1.6 3.2 5.5 7.8 9.4]]
  y=[ 7.8 19.1 17.6 33.9 45.4]
)
```

1.2.3 Fit Result

When a fit is performed, the returned object is a *Result* instance

```
>>> model = LinearModel()
>>> result = model.fit(x, y, params=[1, 1])
>>> result.chisq
84.266087804...
>>> result.correlation
array([[ 1.        , -0.88698141],
       [-0.88698141,  1.        ]])
>>> result.params.values()
array([0.52243902, 4.40682927])
>>> for param in result.params:
...     print(param.name, param.value, param.uncert)
a1 0.5224390243941... 5.132418149940...
a2 4.4068292682920... 0.827701724508...
```

You can display a summary of the fit result

```
>>> result
Result(
  calls=2
  chisq=84.266087804878
  correlation=[[ 1.         -0.88698141]
               [-0.88698141  1.        ]]
  covariance=[[ 0.93780488 -0.13414634]
              [-0.13414634  0.02439024]]
  dof=3.0
  eof=5.299876973568286
  iterations=22
  params=
    ResultParameters(
      ResultParameter(name='a1', value=0.5224390243941934, uncert=5.
↪132418149940028, label=None),
      ResultParameter(name='a2', value=4.4068292682920465, uncert=0.
↪8277017245089597, label=None)
    )
)
```

Using the *result* object and the *evaluate()* method, the residuals can be calculated

```
>>> y - model.evaluate(x, result)
array([ 0.22663415,  4.47570732, -7.16      , -0.99570732,  3.45336585])
```

1.2.4 Save and Load .nlf Files

A *Model* can be saved to a file and loaded from a file. The file that is created with **msl-nlf** can also be opened in the Delphi GUI application and a **.nlf** file that is created in the Delphi GUI application can be loaded in **msl-nlf**. See the *save()* method and the *load()* function for more details.

```
# Create a model
from msl.nlf import LinearModel
model = LinearModel()
model.fit([1, 2, 3], [0.07, 0.27, 0.33])

# Save the model to a file.
# The results of the fit are not written to the file, so if you are
# opening 'samples.nlf' in the Delphi GUI, click the Calculate button
# and the Results table and the Graphs will be updated.
model.save('samples.nlf')


# At a later date, load the file and perform the fit
from msl.nlf import load
loaded = load('samples.nlf')
results = loaded.fit(loaded.x, loaded.y, params=loaded.params)
```

1.2.5 A Model as a Context Manager

The fit function in the DLL reads the information it needs for the fitting process from RAM but also from files on the hard disk. Configuration (and perhaps correlation) files are written to a temporary directory for the DLL function to read from. This temporary directory should automatically get deleted when you are done using the *Model* (when the objects reference count is 0 and gets garbage collected).

Also, if loading a 32-bit DLL in 64-bit Python (see *32-bit vs 64-bit DLL*) a client-server application starts in the background when a *Model* is created. Similarly, the client-server application should automatically shut down when you are done using the *Model*.

A *Model* can be used as a context manager (see The with statement) which will delete the temporary directory (and shut down the client-server application) once the *with* block is finished, for example,

```
from msl.nlf import Model

x = [1, 2, 3, 4, 5]
y = [1.1, 4.02, 9.2, 16.2, 25.5]

with Model('a1*x^2', dll='nlf32') as model:  # temporary files created,␣
↪client-server protocol starts
    result = model.fit(x, y, params=[1])

# no longer in the 'with' block
# temporary files have been deleted
# the client-server protocol has shut down
# you must create a new Model if you want to use it again
```

It is your choice if you want to use a *Model* as a context manager. There is no difference in performance, but the *cleanup* steps are more likely to occur when used as a context manager.

1.3 Examples

More examples showing how to use the non-linear fitting software.

1.3.1 Uncertain Real Numbers (GTC)

An example from Appendix H3 of the GUM[1].

This example requires GTC to be installed, to install it run

```
pip install GTC
```

As defined in Appendix H3 of the GUM, the calibration curve is

$$b(t) = y_1 + y_2(t - t_0)$$

where the reference temperature, t_0, is chosen to be 20 °C.

This translates to the following equation that is passed to a *Model*

$$f(x; a) = a_1 + a_2(x - 20)$$

The *intercept* (a_1) and *slope* (a_2) result parameters are converted to a correlated ensemble of uncertain real numbers (via the *to_ureal()* method) which are used to calculate the response at a chosen stimulus.

```
from msl.nlf import Model

# Thermometer readings (degrees C)
x = (21.521, 22.012, 22.512, 23.003, 23.507, 23.999, 24.513, 25.002, 25.503,␣
↪26.010, 26.511)

# Observed differences with calibration standard (degrees C)
y = (-0.171, -0.169, -0.166, -0.159, -0.164, -0.165, -0.156, -0.157, -0.159, -
↪0.161, -0.160)

# Arbitrary offset temperature (degrees C)
t0 = 20

# Create the model
model = Model(f'a1+a2*(x-{t0})')

# Create an initial guess. Allow the intercept and slope to vary during
# the fitting process and assign helpful labels
params = model.create_parameters([
```

(continues on next page)

[1] BIPM and IEC and IFCC and ISO and IUPAC and IUPAP and OIML, *Evaluation of measurement data - Guide to the expression of uncertainty in measurement JCGM 100:2008 (GUM 1995 with minor corrections)*, (2008) http://www.bipm.org/en/publications/guides/gum

(continued from previous page)

```
    ('a1', 1, False, 'intercept'),
    ('a2', 1, False, 'slope')])

# Apply the fit
result = model.fit(x, y, params=params)

# Convert the result to a correlated ensemble of uncertain real numbers
intercept, slope = result.to_ureal()
```

The *intercept* and *slope* can be used to calculate a correction for a reading of 30 °C

```
>>> intercept + slope*(30 - t0)
ureal(-0.14937681268874...,0.004138595752854...,9.0)
```

1.3.2 Plotting results (Matplotlib)

This example requires Matplotlib to be installed, to install it run

```
pip install matplotlib
```

A *LinearModel* is used to perform the weighted fit. An initial guess is automatically generated (i.e., no *params* are passed to the *fit()* method). Using the *evaluate()* method, the fit line and the residuals are evaluated.

```
import numpy as np
import matplotlib.pyplot as plt
from msl.nlf import LinearModel

# Sample data
x = np.array([1.6, 3.2, 5.5, 7.8, 9.4])
y = np.array([7.8, 19.1, 17.6, 33.9, 45.4])

# Standard uncertainties in y
uy = np.array([0.91, 2.3, 3.3, 1.8, 4.1])

# Create a linear model and specify a weighted fit
model = LinearModel(weighted=True)

# Use a default initial guess (do not specify params) and apply the fit
result = model.fit(x, y, uy=uy)

# Evaluate the model using the "result" object to create a fit line
x_fit = np.linspace(np.min(x), np.max(x), 1000)
y_fit = model.evaluate(x_fit, result)

# Evaluate the residuals
residuals = y - model.evaluate(x, result)

# Prepare two plots (one for the data and fit, one for the residuals)
```

(continues on next page)

(continued from previous page)

```
ax1 = plt.subplot(211)
ax2 = plt.subplot(212)

# Plot the data with the fit line
ax1.errorbar(x, y, yerr=uy, c='blue', fmt='o', capsize=5.0)
ax1.plot(x_fit, y_fit, c='red')
ax1.set_title('Data and Fit')

# Plot the residuals (and show the y=0 axis)
ax2.scatter(x, residuals, c='blue')
ax2.set_title('Residuals')
ax2.axhline(y=0, color='black', linewidth=0.5)

# Display the plots (and add some more spacing between the plots)
plt.subplots_adjust(hspace=0.5)
plt.show()
```

1.3.3 2D, Weighted and Correlated

This example uses a *Model* with the equation specified as a string. The independent variable (stimulus) data is two-dimensional (i.e., contains x_1 and x_2 variables). There are uncertainties in both x and y variables and the $y - y$ correlation coefficient is 0.5, the $x_1 - x_1$ correlation coefficient is 0.8 and the *set_correlation()* method is called to set the correlation matrices in the model.

Prepare the model

```
import numpy as np
from msl.nlf import Model

# Sample data
x = np.array([[1, 2, 3, 4], [0.1, 0.2, 0.3, 0.4]])
y = np.array([1.1, 1.9, 3.2, 3.7])

# Standard uncertainties in x and y
ux = np.array([[0.01, 0.02, 0.03, 0.04], [0.002, 0.004, 0.006, 0.008]])
uy = np.array([0.5, 0.5, 0.5, 0.5])

# Initial guess
guess = np.array([0, 0.9, 0])

# Specify the equation as a string
model = Model('a1+a2*(x1+exp(a3*x1))+x2')

# Set the options for a weighted and correlated fit
model.options(weighted=True, correlated=True)

# Define the correlation coefficient matrices, the value is set in the
```

(continues on next page)

(continued from previous page)

```
# off-diagonal matrix elements of the correlation matrix
model.set_correlation('y', 'y', value=0.5)
model.set_correlation('x1', 'x1', value=0.8)
```

To see a summary of the data that would be sent to the fit function in the DLL, call the *fit()* method with *debug=True* and print the returned object (an instance of *Input* is returned)

```
model_input = model.fit(x, y, params=guess, uy=uy, ux=ux, debug=True)
print(model_input)
```

The summary that is printed is

```
Input(
  absolute_residuals=True
  correlated=True
  correlations=
    Correlations(
      data=[
        Correlation(
          coefficients=[[1.  0.8 0.8 0.8]
                        [0.8 1.  0.8 0.8]
                        [0.8 0.8 1.  0.8]
                        [0.8 0.8 0.8 1. ]]
          path='...\CorrCoeffs X1-X1.txt'
        ),
        Correlation(
          coefficients=[[1.  0.5 0.5 0.5]
                        [0.5 1.  0.5 0.5]
                        [0.5 0.5 1.  0.5]
                        [0.5 0.5 0.5 1. ]]
          path='...\CorrCoeffs Y-Y.txt'
        )]
      is_correlated=[[ True False False]
                     [False  True False]
                     [False False False]]
    )
  delta=0.1
  equation='a1+a2*(x1+exp(a3*x1))+x2'
  fit_method=<FitMethod.LM: 'Levenberg-Marquardt'>
  max_iterations=999
  params=
    InputParameters(
      InputParameter(name='a1', value=0.0, constant=False, label=None),
      InputParameter(name='a2', value=0.9, constant=False, label=None),
      InputParameter(name='a3', value=0.0, constant=False, label=None)
    )
  residual_type=<ResidualType.DY_X: 'dy v x'>
  second_derivs_B=True
  second_derivs_H=True
  tolerance=1e-20
```

(continues on next page)

(continued from previous page)

```
  ux=[[0.01  0.02  0.03  0.04 ]
      [0.002 0.004 0.006 0.008]]
  uy=[0.5 0.5 0.5 0.5]
  uy_weights_only=False
  weighted=True
  x=[[1.  2.  3.  4. ]
     [0.1 0.2 0.3 0.4]]
  y=[1.1 1.9 3.2 3.7]
)
```

To see a summary of the fit result, call the *fit()* method with *debug=False* (which is also the default value) and print the returned object (an instance of *Result* is returned)

```
result = model.fit(x, y, params=guess, uy=uy, ux=ux, debug=False)
print(result)
```

The summary that is printed is

```
Result(
  calls=3
  chisq=0.854875600205648
  correlation=[[ 1.         -0.81341696  0.33998683]
               [-0.81341696  1.         -0.41807236]
               [ 0.33998683 -0.41807236  1.        ]]
  covariance=[[ 4.62857806e-01 -8.76652826e-02  2.75368388e-05]
              [-8.76652826e-02  2.50946556e-02 -7.88442937e-06]
              [ 2.75368388e-05 -7.88442937e-06  1.41728236e-08]]
  dof=inf
  eof=0.32710857899179385
  iterations=33
  params=
    ResultParameters(
      ResultParameter(name='a1', value=-0.6101880747640294, uncert=0.
↪6803365385456976, label=None),
      ResultParameter(name='a2', value=0.8100288869777268, uncert=0.
↪1584129274256673, label=None),
      ResultParameter(name='a3', value=4.585005881907852e-05, uncert=0.
↪00011904966869376515, label=None)
    )
)
```

1.3.4 Composite Model

In this example, a composite model is created using built-in *models* to create a model for a damped oscillator (underdamped case).

The equation for the underdamped oscillator is

$$f(x; a) = a_1 e^{-a_2 x} \sin(a_3 x + a_4) + a_5$$

First, simulate some noisy data *(we can also see what the expected results of the a_i parameters are)*

```
import numpy as np

x = np.linspace(0, 1, num=200)
noise = np.random.normal(scale=0.15, size=x.size)
y = 2.6 * np.exp(-4.3*x) * np.sin(48.3*x + 0.5) + 0.7 + noise
```

Next, set up the model by creating a composite model from built-in *models*, create the initial-guess parameters and apply the fit

```
from msl.nlf import ExponentialModel, SineModel, ConstantModel

# Create the composite model
model = ExponentialModel() * SineModel() + ConstantModel()

# Equivalently, without using the built-in models, one could have explicitly␣
↪written the equation
# model = Model('a1*exp(-a2*x)*sin(a3*x+a4)+a5')

# Create the initial-guess parameters. All are allowed to vary during the
# fitting process and assign helpful labels
params = model.create_parameters()
params['a1'] = 1, False, 'amplitude'
params['a2'] = 1, False, 'damping'
params['a3'] = 10, False, 'omega'
params['a4'] = 0, False, 'phase'
params['a5'] = 0, False, 'offset'

# Apply the fit
result = model.fit(x, y, params=params)
```

Print the result parameters *(you can compare with the expected values above, ignoring the noise)*

```
print(result.params)
```

```
ResultParameters(
  ResultParameter(name='a1', value=2.566989469687458, uncert=0.
↪06484473196789138, label='amplitude'),
  ResultParameter(name='a2', value=4.464096678565754, uncert=0.
↪16520633638746224, label='damping'),
  ResultParameter(name='a3', value=48.149066188747334, uncert=0.
↪17276917810113002, label='omega'),
```

(continues on next page)

(continued from previous page)

```
  ResultParameter(name='a4', value=0.5257334979692393, uncert=0.
↪028013269024991513, label='phase'),
  ResultParameter(name='a5', value=0.7136729265108783, uncert=0.
↪011503730542985225, label='offset')
)
```

The following requires Matplotlib to be installed, to install it run

```
pip install matplotlib
```

Using the *evaluate()* method, plot the data and the fit curve

```
import matplotlib.pyplot as plt

# Evaluate the fit curve from the "result" object
x_fit = np.linspace(np.min(x), np.max(x), num=1000)
y_fit = model.evaluate(x_fit, result)

# Plot the data and the fit
plt.scatter(x, y, c='blue')
plt.plot(x_fit, y_fit, c='red')
plt.show()
```

1.4 32-bit vs 64-bit DLL

A 32-bit and a 64-bit version of the DLL are provided. The following table illustrates the differences between the DLL versions.

Table 1: 32-bit vs 64-bit comparison

32-bit DLL	64-bit DLL
Can be used in both 32- and 64-bit versions of Python	Can only be used in 64-bit Python
When used in 64-bit Python, the fit will take longer[1]	There is no performance overhead
Limited to 4GB RAM	Can access more than 4GB RAM
Can load a 32-bit user-defined DLL function[2]	Can load a 64-bit user-defined DLL function[2]

If loading the 32-bit DLL in 64-bit Python, it is important to reduce the number of times a *Model* is created to fit data. In this case, creating a *Model* object takes about 1 second for a client-server protocol to be initialized in the background. Once the *Model* has been created, the client and server are running and repeatedly calling the *fit()* method will be more efficient (but still slower than fitting data with the 64-bit DLL in 64-bit Python, or the 32-bit DLL in 32-bit Python).

[1] This is not due to the 32-bit Delphi code, but due to an overhead on the Python side to exchange data between 64-bit Python and a 32-bit DLL. When the 32-bit DLL is used in 32-bit Python, there is no overhead.

[2] See *User-Defined Function*

Pseudocode is shown below that demonstrates the best way to apply fits if loading the 32-bit DLL in 64-bit Python. See *A Model as a Context Manager* for more details about the use of the *with* statement:

```
# Don't do this. Don't create a new model to process each data file.
for data in data_files:
    with LinearModel(dll='nlf32') as model:
        result = model.fit(data.x, data.y)

# Do this instead. Create a model once and then fit each data file.
with LinearModel(dll='nlf32') as model:
    for data in data_files:
        result = model.fit(data.x, data.y)
```

1.5 User-Defined Function

For situations when the fit equation cannot be expressed in analytical form by using the arithmetic operations and functions that are supported, the user can create a custom function that is compiled as a DLL. This custom DLL must export four functions with the following names:

- GetFunctionName
- GetFunctionValue
- GetNumParameters
- GetNumVariables

How to define these four functions is best shown with examples.

1.5.1 C++ Example (1D)

A user-defined function is created in C++ in order to fit the Roszman1 dataset that is provided by NIST. This fit equation requires the *arctan* function, which is not one of the built-in functions that are currently supported by the Delphi software (but could be upon request).

The header file is

```
// Roszman1.h
#define EXPORT __declspec(dllexport)

#define pi 3.14159265358979323846264338327 9

extern "C" {
    EXPORT void GetFunctionName(char* name);
    EXPORT void GetFunctionValue(double* x, double* a, double* y);
    EXPORT void GetNumParameters(int* n);
    EXPORT void GetNumVariables(int* n);
}
```

and the source file is

```
/*
* Roszman1.cpp
*
* User-defined function for the Roszman1 dataset that is provided by NIST
*
* https://www.itl.nist.gov/div898/strd/nls/data/LINKS/DATA/Roszman1.dat
*/
#include <math.h>  // atan
#include <string.h>  // strcpy_s
#include "Roszman1.h"

void GetFunctionName(char* name) {
    // The name must begin with f followed by a positive integer followed by␣
↪a colon.
    // The remainder of the string is for information for the user.
    strcpy_s(name, 255, "f1: Roszman1 f1=a1-a2*x-arctan(a3/(x-a4))/pi");
}

void GetFunctionValue(double* x, double* a, double* y) {
    // Receives the x value, the fit parameters and a pointer to the y value.
    // C++ array indices are zero based (i.e., a1=a[0] and x=x[0])
    *y = a[0] - a[1] * x[0] - atan(a[2] / (x[0] - a[3])) / pi;
}

void GetNumParameters(int* n) {
    // There are 4 parameters: a1, a2, a3, a4
    *n = 4;
}

void GetNumVariables(int* n) {
    // There is only 1 independent variable: x
    *n = 1;
}
```

To compile the C++ source code to a DLL, one could use Visual Studio C++,

```
cl.exe /LD Roszman1.cpp
```

1.5.2 C++ Example (2D)

A user-defined function is created in C++ in order to fit the Nelson dataset that is provided by NIST. This fit equation, **a1-a2*x1*exp(-a3*x2)**, could have been passed directly to a *Model* since all arithmetic operations and functions are supported; however, this example illustrates how to handle situations when there are multiple x variables

The header file is

```
// Nelson.h
#define EXPORT __declspec(dllexport)
```

(continues on next page)

(continued from previous page)

```
extern "C" {
    EXPORT void GetFunctionName(char* name);
    EXPORT void GetFunctionValue(double* x, double* a, double* y);
    EXPORT void GetNumParameters(int* n);
    EXPORT void GetNumVariables(int* n);
}
```

and the source file is

```
/*
* Nelson.cpp
*
* User-defined function for the Nelson dataset that is provided by NIST
*
* https://www.itl.nist.gov/div898/strd/nls/data/LINKS/DATA/Nelson.dat
*/
#include <math.h>  // exp
#include <string.h>  // strcpy_s
#include "Nelson.h"

void GetFunctionName(char* name) {
    // The name must begin with f followed by a positive integer followed by␣
↪a colon.
    // The remainder of the string is for information for the user.
    strcpy_s(name, 255, "f2: Nelson log(f2)=a1-a2*x1*exp(-a3*x2)");
}

void GetFunctionValue(double* x, double* a, double* y) {
    // Receives the x value, the fit parameters and a pointer to the y value.
    // C++ array indices are zero based (i.e., a1=a[0], a2=a[1], a3=a[2],␣
↪x1=x[0], x2=x[1])
    *y = a[0] - a[1] * x[0] * exp(-a[2] * x[1]);
}

void GetNumParameters(int* n) {
    // There are 3 parameters: a1, a2, a3
    *n = 3;
}

void GetNumVariables(int* n) {
    // There are 2 independent variables: x1, x2
    *n = 2;
}
```

To compile the C++ source code to a DLL, one could use Visual Studio C++,

```
cl.exe /LD Nelson.cpp
```

1.5.3 Delphi Example

A user-defined function is created in Delphi Pascal for the Beta Distribution.

```
library BetaDLL;

uses
    SysUtils,
    Classes,
    sfGamma in '..\..\Maths Functions\sfgamma.pas',
    AMath in '..\..\Maths Functions\amath.pas',
    sfBasic in '..\..\Maths Functions\sfbasic.pas',
    sfZeta in '..\..\Maths Functions\sfzeta.pas',
    sfExpInt in '..\..\Maths Functions\sfexpint.pas',
    sfHyperG in '..\..\Maths Functions\sfhyperg.pas',
    sfPoly in '..\..\Maths Functions\sfpoly.pas',
    sfEllInt in '..\..\Maths Functions\sfellint.pas',
    sfMisc in '..\..\Maths Functions\sfmisc.pas',
    sfBessel in '..\..\Maths Functions\sfbessel.pas',
    sfErf in '..\..\Maths Functions\sferf.pas';

type
    PArray=^TArray;
    TArray=array[1..100] of Double;

function Gamma(X:Double):Double;
begin
    Gamma:=sfc_gamma(X);
end;

procedure GetFunctionName(var Name:PAnsiChar); cdecl;
begin
    {The name must begin with f followed by a positive integer followed by a␣
↪colon.
    The remainder of the string is for information for the user.}
    StrCopy(Name, 'f3: Beta Distribution (f3=a3/(a5-a4)*Gamma(a1+a2)/
↪(Gamma(a1)*Gamma(a2))*((x-a4)/(a5-a4))^(a1-1)*((a5-x)/(a5-a4))^(a2-1))');
end;

procedure GetFunctionValue(x,a:PArray; var y:Double); cdecl;
{Returns the value of the user-defined function in the y-variable based on the
input x array and a array, where a is the parameter array.}
var
    p1,p2,g1,g2:Double;
begin
    if (a[5]<=a[4]) or (x[1]<=a[4]) or (x[1]>=a[5]) then
        y:=0
    else
    begin
        p1:=Power((x[1]-a[4])/(a[5]-a[4]),a[1]-1);
        p2:=Power((a[5]-x[1])/(a[5]-a[4]),a[2]-1);
```

(continues on next page)

(continued from previous page)

```
        g1:=Gamma(a[1]);
        g2:=Gamma(a[2]);
        if (g1=0) or (g1=PosInf_x) or (g2=0) or (g2=PosInf_x) then
            y:=0
        else
            y:=a[3]/(a[5]-a[4])*Gamma(a[1]+a[2])/(g1*g2)*p1*p2;
    end;
end;

procedure GetNumParameters(var NumParameters:Integer); cdecl;
begin
    {There are 5 parameters: a1, a2, a3, a4, a5}
    NumParameters:=5;
end;

procedure GetNumVariables(var NumVariables:Integer); cdecl;
begin
    {There is only 1 independent variable: x}
    NumVariables:=1;
end;

exports
    GetFunctionName index 1,
    GetFunctionValue index 2,
    GetNumParameters index 3,
    GetNumVariables index 4;

begin
end.
```

1.5.4 Using the Function

To use a custom function, the first parameter passed when defining a *Model* must be the first part of the name, up to the colon, defined in *GetFunctionName*, and, optionally, specify the directory where the custom DLL is located as a *user_dir* keyword argument. If you are also using the Delphi GUI, the directory that has been set in the GUI for the user-defined functions will be used as the default *user_dir* value. Otherwise, the current working directory is used as the default *user_dir* value if a directory is not explicitly specified.

Below, the C++ function, *f1*, is used as the custom function

```
from msl.nlf import Model

model = Model('f1', user_dir='./tests/user_defined')
```

1.6 API Documentation

msl.nlf.**version_info**(*major*, *minor*, *micro*, *releaselevel*)

namedtuple: Contains the version information as a (major, minor, micro, releaselevel) tuple.

msl.nlf.**load**(*path*, *, *dll=None*)

Load a **.nlf** file.

No information about the fit results are read from the file. The fit equation, the fit options and the correlation coefficients have been set in the *LoadedModel* that is returned, but you must specify the *x*, *y*, *params*, *ux* and/or *uy* attributes of the *LoadedModel* to the *fit()* method (or specify different data to the *fit()* method).

Parameters

- **path** (*str*) – The path to a **.nlf** file. The file could have been created by the Delphi GUI application or by the *save()* method.
- **dll** (*str*) – Passed to the *dll* keyword argument in *Model*.

Returns

The loaded model.

Return type

LoadedModel

Examples

```
>>> from msl.nlf import load
>>> loaded = load('samples.nlf')
>>> results = loaded.fit(loaded.x, loaded.y, params=loaded.params)
```

The following are *submodules*:

1.6.1 msl.nlf.client_server module

Call functions in a 32-bit DLL from 64-bit Python.

class msl.nlf.client_server.**ClientNLF**(*path*)

Bases: Client64

Send requests to the 32-bit DLL.

Parameters

path (*str*) – The path to the DLL file.

dll_version()

Get the version number from the DLL.

Return type

str

evaluate(*a*, *x*, *shape*)

Evaluate the user-defined function.

Parameters

- **a** (*array*) – Parameter values.
- **x** (*array*) – Independent variable (stimulus) data.
- **shape** (*tuple[int, int]*) – The shape of the *x* data.

Returns
Dependent variable (response) data.

Return type
array

fit(***kwargs*)

Fit the model to the data using the supplied keyword arguments.

Return type
dict

get_user_defined(*directory*)

Get all user-defined functions.

Parameters
directory (*str*) – The directory to look for the user-defined functions.

Returns
The keys are the filenames and the values are *UserDefined*.

Return type
dict [str, *UserDefined*]

load_user_defined(*equation*, *directory*)

Load a user-defined function in a custom DLL.

Parameters

- **equation** (*str*) – The equation to load.
- **directory** (*str*) – The directory to look for the user-defined function.

Return type
None

class msl.nlf.client_server.**ServerNLF**(*host*, *port*, *path=''*)

Bases: Server32

Handle requests for the 32-bit DLL.

Parameters

- **host** (*str*) – The IP address of the server.
- **port** (*int*) – The port to run the server on.
- **path** (*str*) – The path to the DLL file.

dll_version()

Get the version number from the DLL.

Return type
str

evaluate(*a*, *x*, *shape*)

Evaluate the user-defined function.

Parameters

- **a** (*array*) – Parameter values.
- **x** (*array*) – Independent variable (stimulus) data.
- **shape** (*tuple[int, int]*) – The shape of the *x* data.

Returns
Dependent variable (response) data.

Return type
array

fit(***kwargs*)

Fit the model to the data using the supplied keyword arguments.

Return type
dict

static get_user_defined(*directory*)

Get all user-defined functions.

Parameters
directory (*str*) – The directory to look for the user-defined functions.

Returns
The user-defined functions.

Return type
dict

load_user_defined(*equation*, *directory*)

Load a user-defined function in a custom DLL.

Parameters

- **equation** (*str*) – The equation to load.
- **directory** (*str*) – The directory to look for the user-defined function.

Return type
None

1.6.2 msl.nlf.datatypes module

Various data classes and enums.

class msl.nlf.datatypes.**Correlation**(*path*, *coefficients*)

Bases: object

Information about correlation coefficients.

Parameters

- **path** (*str*) –
- **coefficients** (*ndarray[float]*) –

coefficients: ndarray[float]

The correlation coefficients.

path: str

The path to the correlation file.

class msl.nlf.datatypes.**Correlations**(*data*, *is_correlated*)

Bases: object

Information about the correlations for a fit model.

Parameters

- **data** (*list[Correlation]*) –
- **is_correlated** (*ndarray[bool]*) –

data: list[*Correlation*]

A list of *Correlation* objects.

is_correlated: ndarray[bool]

Indicates which variables are correlated. The index 0 corresponds to the *y*-variable, the index 1 to *x1*, 2 to *x2*, etc.

class msl.nlf.datatypes.**FitMethod**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: Enum

Fitting methods.

Least squares (LS) minimises the sum of the squares of the vertical distances between each point and the fitted curve. The algorithms implemented for Levenberg Marquardt, Amoeba and Powell are described in Numerical Recipes.

Minimum distance (MD) minimises the sum of the distances (in two dimensions) between each point and the fitted curve. This type of fit is not available when data is correlated nor is it available when there is more than one independent variable (stimulus).

MiniMax (MM) minimises the value of the maximum absolute y-residual. This type of fit is not available when data is correlated.

AMOEBA_LS = 'Amoeba least squares'

AMOEBA_MD = 'Amoeba minimum distance'

AMOEBA_MM = 'Amoeba minimax'

LM = 'Levenberg-Marquardt'

POWELL_LS = 'Powell least squares'

POWELL_MD = 'Powell minimum distance'

POWELL_MM = 'Powell minimax'

class msl.nlf.datatypes.**Input**(*absolute_residuals*, *correlated*, *correlations*, *delta*, *equation*, *fit_method*, *max_iterations*, *params*, *residual_type*, *second_derivs_B*, *second_derivs_H*, *tolerance*, *ux*, *uy*, *uy_weights_only*, *weighted*, *x*, *y*)

Bases: object

The input data to a fit model.

Parameters

- **absolute_residuals** (*bool*) –
- **correlated** (*bool*) –
- **correlations** (Correlations) –
- **delta** (*float*) –
- **equation** (*str*) –
- **fit_method** (FitMethod) –
- **max_iterations** (*int*) –
- **params** (InputParameters) –
- **residual_type** (ResidualType) –
- **second_derivs_B** (*bool*) –
- **second_derivs_H** (*bool*) –
- **tolerance** (*float*) –
- **ux** (*ndarray[float]*) –
- **uy** (*ndarray[float]*) –
- **uy_weights_only** (*bool*) –
- **weighted** (*bool*) –
- **x** (*ndarray[float]*) –
- **y** (*ndarray[float]*) –

absolute_residuals: bool

Whether absolute residuals or relative residuals are used to evaluate the *eof*.

correlated: bool

Whether correlations are applied in the fitting process.

correlations: *Correlations*

The information about the correlation coefficients.

delta: float

Only used for Amoeba fitting.

equation: str

The equation of the fit model.

fit_method: *FitMethod*

The method that is used for the fit.

max_iterations: int

The maximum number of fit iterations allowed.

params: *InputParameters*

The input parameters to the fit model.

residual_type: *ResidualType*

Residual Type that is used to evaluate the *eof*.

second_derivs_B: bool

Whether the second derivatives in the **B** matrix are included in the propagation of uncertainty calculations.

second_derivs_H: bool

Whether the second derivatives in the curvature matrix, **H** (Hessian), are included in the propagation of uncertainty calculations.

tolerance: float

The tolerance value to stop the fitting process.

ux: ndarray[float]

Standard uncertainties in the x (stimulus) data.

uy: ndarray[float]

Standard uncertainties in the y (response) data.

uy_weights_only: bool

Whether the y uncertainties only or a combination of the x and y uncertainties are used to calculate the weights for a weighted fit.

weighted: bool

Whether to include the standard uncertainties in the fitting process to perform a weighted fit.

x: ndarray[float]

The independent variable(s) (stimulus) data.

y: ndarray[float]

The dependent variable (response) data.

class msl.nlf.datatypes.**ResidualType**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: Enum

Residual Type that is used to evaluate the *eof*.

DX_X = 'dx v x'

Uncertainty in x versus x.

DX_Y = 'dx v y'

Uncertainty in x versus y.

DY_X = 'dy v x'

Uncertainty in y versus x.

DY_Y = 'dy v y'

Uncertainty in y versus y.

class msl.nlf.datatypes.**Result**(*calls*, *chisq*, *correlation*, *covariance*, *dof*, *eof*, *iterations*, *params*)

Bases: object

The result from a fit model.

Parameters

- **calls** (*int*) –
- **chisq** (*float*) –
- **correlation** (*ndarray[float]*) –
- **covariance** (*ndarray[float]*) –
- **dof** (*float*) –
- **eof** (*float*) –
- **iterations** (*int*) –
- **params** (ResultParameters) –

calls: int

The number of calls to the DLL fit function.

chisq: float

The chi-squared value.

correlation: ndarray[float]

Parameter correlation coefficient matrix.

covariance: ndarray[float]

Parameter covariance matrix.

dof: float

The number of degrees of freedom that are retained.

If a fit is weighted or correlated, the degrees of freedom is infinity. Otherwise, the degrees of freedom is equal to the number of data points (observations) minus the number of fit parameters.

eof: float

The error-of-fit value (the standard deviation of the residuals).

iterations: int

The total number of fit iterations.

params: *ResultParameters*

The result parameters from the fit model.

to_ureal(*, *with_future=False*, *label='future'*)

Convert the result to a correlated ensemble of uncertain real numbers.

Parameters

- **with_future** (*bool*) – Whether to include an uncertain real number in the ensemble that is a *future* indication in response to a given stimulus (a predicted future response). This reflects the variability of single indications as

well as the underlying uncertainty in the fit parameters. The *value* of this *future* uncertain number is zero, and the *uncertainty* component is $\sqrt{\frac{\chi^2}{dof}}$.

- **label** (*str*) – The label to assign to the *future* uncertain number.

Returns

A correlated ensemble of uncertain real numbers.

Return type

list of UncertainReal

Examples

Suppose the sample data has a linear relationship

```
>>> x = [3, 7, 11, 15, 18, 27, 29, 30, 30, 31, 31, 32, 33, 33, 34,␣
↪36,
...      36, 36, 37, 38, 39, 39, 39, 40, 41, 42, 42, 43, 44, 45, 46,␣
↪47, 50]
>>> y = [5, 11, 21, 16, 16, 28, 27, 25, 35, 30, 40, 32, 34, 32, 34,␣
↪37,
...      38, 34, 36, 38, 37, 36, 45, 39, 41, 40, 44, 37, 44, 46, 46,␣
↪49, 51]
```

The intercept and slope are determined from a fit

```
>>> from msl.nlf import LinearModel
>>> with LinearModel() as model:
...     result = model.fit(x, y)
```

We can estimate the response to a particular stimulus, say $x = 21.5$

```
>>> intercept, slope = result.to_ureal()
>>> intercept + 21.5*slope
ureal(23.257962225044...,0.82160705888850...,31.0)
```

or a single future indication in response to a given stimulus may also be of interest (again, at $x = 21.5$)

```
>>> intercept, slope, future = result.to_ureal(with_future=True)
>>> intercept + 21.5*slope + future
ureal(23.257962225044...,3.33240925795711...,31.0)
```

The value here is the same as above (because the stimulus is the same), but the uncertainty is much larger, reflecting the variability of single indications as well as the underlying uncertainty in the intercept and slope.

1.6.3 msl.nlf.dll module

Wrapper around DLL functions.

class msl.nlf.dll.**UserDefined**(*equation*, *function*, *name*, *num_parameters*, *num_variables*)

Bases: object

A user-defined function that has been compiled to a DLL.

Parameters

- **equation** (*str*) –
- **function** (*Callable*) –
- **name** (*str*) –
- **num_parameters** (*int*) –
- **num_variables** (*int*) –

equation: str

The value to use as the *equation* for a *Model*.

function: Callable

A reference to the *GetFunctionValue* function.

name: str

The name returned by the *GetFunctionName* function.

num_parameters: int

The value returned by the *GetNumParameters* function.

num_variables: int

The value returned by the *GetNumVariables* function.

to_dict()

Convert this object to be a pickleable dict.

The value of *function* becomes None.

Return type
dict

msl.nlf.dll.**define_fit_fcn**(*dll*, *as_ctypes*)

Defines the *argtypes* and *restype* of the *DoNonlinearFit* function.

Parameters

- **dll** (*CDLL*) – The instance of the DLL.
- **as_ctypes** (*bool*) – Whether ctypes arrays or numpy.ndarrays will be passed to the *DoNonlinearFit* function.

Return type
None

msl.nlf.dll.**evaluate**(*fcn*, *a*, *x*, *shape*, *y*)

Call *GetFunctionValue* in the user-defined DLL.

Parameters

- **fcn** (*Callable*) – Reference to *GetFunctionValue*.
- **a** (*Sequence[float]*) – Parameter values.
- **x** (*Sequence[float]*) – Independent variable (stimulus) data. The data must already be transposed and flat.
- **shape** (*tuple[int, int]*) – The shape of the *x* data.
- **y** – Pre-allocated sequence for the dependent variable (response) data.

Returns
Dependent variable (response) data.

Return type
array or ndarray

msl.nlf.dll.**fit**(*dll*, ***k*)

Call the *DoNonlinearFit* function in the DLL.

Parameters

- **dll** (*CDLL*) – The instance of the DLL.
- **k** – Keyword arguments that are required to perform the fit.

Returns
The fit results of the DLL.

Return type
dict

msl.nlf.dll.**get_user_defined**(*directory*)

Get all user-defined functions.

Parameters
directory (*str*) – The directory to look for the user-defined functions.

Returns
The keys are the filenames and the values are *UserDefined*.

Return type
dict [str, *UserDefined*]

msl.nlf.dll.**version**(*dll*)

Call the *GetVersion* function in the DLL.

Parameters
dll (*CDLL*) – The instance of the DLL.

Returns
The version number of the DLL.

Return type
str

1.6.4 msl.nlf.loader module

Load a **.nlf** file.

class msl.nlf.loader.**Loader**(*path*)

Bases: object

Helper class to read a **.nlf** file.

Parameters
path (*str*) – The path to a **.nlf** file.

read_boolean()

Read a boolean.

Return type
bool

read_byte()

Read a byte.

Return type
int

read_extended()

Read a Delphi 10-byte extended float.

Return type
float

read_integer()

Read an unsigned integer.

Return type
int

read_string()

Read a string.

Return type
str

read_string_padded(*length*)

Read a string that is null padded.

Parameters
length (*int*) – The total length of the null-padded string.

Return type
str

read_word()

Read an unsigned short.

Return type
int

msl.nlf.loader.**load_form**(*loader*)

Load a *TDataForm*.

Parameters
loader (Loader) – The class helper.

Returns
The settings of the *TDataForm*.

Return type
dict

msl.nlf.loader.**load_graph**(*loader*)

Load a *TGraphWindow*.

Parameters
loader (Loader) – The class helper.

Returns
The settings of the *TGraphWindow*.

Return type
dict

1.6.5 msl.nlf.model module

A model to use for a non-linear fit.

msl.nlf.model.**ArrayLike**

A 1D or 2D array.

alias of Union[Sequence[float], Sequence[Sequence[float]]]

msl.nlf.model.**ArrayLike1D**

A 1D array.

alias of Sequence[float]

class msl.nlf.model.**CompositeModel**(*op*, *left*, *right*, ***kwargs*)

Bases: *Model*

Combine two models.

Parameters

- **op** (*str*) – A binary operator: + - * /.
- **left** (Model) – The model on the left side of the operator.
- **right** (Model) – The model on the right side of the operator.
- ****kwargs** – All keyword arguments are passed to *Model*.

class msl.nlf.model.**LoadedModel**(*equation*, *, *dll=None*, ***options*)

Bases: *Model*

A *Model* that was loaded from a **.nlf** file.

Do not instantiate this class directly. The proper way to load a **.nlf** file is via the *load()* function.

Parameters

- **equation** (*str*) – The fit equation. See *Model* for more details.
- **dll** (*str*) – The path to a non-linear fit DLL file. See *Model* for more details.
- ****options** – All additional keyword arguments are passed to *options()*.

comments: str

Comments that were specified.

nlf_path: str

The path to the **.nlf** file that was loaded.

nlf_version: str

The DLL version that created the **.nlf** file.

params: *InputParameters*

Input parameters to the fit model.

ux: ndarray[float]

Standard uncertainties in the x (stimulus) data.

uy: ndarray[float]

Standard uncertainties in the y (response) data.

x: ndarray[float]

The independent variable(s) (stimulus) data.

y: ndarray[float]

The dependent variable (response) data.

class msl.nlf.model.**Model**(*equation*, *, *dll=None*, *user_dir=None*, ***options*)

Bases: object

A model for non-linear fitting.

Parameters

- **equation** (*str*) – The fit equation. The x variables (stimulus) must be specified as *x1*, *x2*, etc. and the parameters as *a1*, *a2*, etc. If only one x-variable is required, it can be simply entered as x. The arithmetic operations and functions that are recognised are:

  + – * / ^ sin cos tan exp ln log arcsin arcos

  where ^ indicates raising to the power. All white space is ignored in the equation. For example, to fit a general quadratic equation one would use `a1+a2*x+a3*x^2`. The **sqrt** function can be written as **^0.5**, for example, **sqrt(2*x)** would be expressed as **(2*x)^0.5** in the equation.

  If using a user-defined function that has been compiled to a DLL, the *equation* name must begin with *f* followed by a positive integer, for example, `f1`.

The *user_dir* keyword argument may also need to be set. See *User-Defined Function*.

- **dll** (*str*) – The path to a non-linear fit DLL file. A default DLL is chosen based on the bitness of the Python interpreter. If you want to load a 32-bit DLL in 64-bit Python then set *dll* to be **nlf32**. See *32-bit vs 64-bit DLL* for reasons why you may want to use a different DLL bitness. You may also specify a path to a DLL that is located in a particular directory of your computer.
- **user_dir** (*str*) – Directory where the user-defined functions are located. The default directory is the directory that the Delphi GUI has set. If the Delphi GUI has not set a directory (because the GUI has not been used) the default directory is the current working directory. See *User-Defined Function*.
- ****options** – All additional keyword arguments are passed to *options()*.

class FitMethod(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: Enum

Fitting methods.

Least squares (LS) minimises the sum of the squares of the vertical distances between each point and the fitted curve. The algorithms implemented for Levenberg Marquardt, Amoeba and Powell are described in Numerical Recipes.

Minimum distance (MD) minimises the sum of the distances (in two dimensions) between each point and the fitted curve. This type of fit is not available when data is correlated nor is it available when there is more than one independent variable (stimulus).

MiniMax (MM) minimises the value of the maximum absolute y-residual. This type of fit is not available when data is correlated.

AMOEBA_LS = 'Amoeba least squares'

AMOEBA_MD = 'Amoeba minimum distance'

AMOEBA_MM = 'Amoeba minimax'

LM = 'Levenberg-Marquardt'

POWELL_LS = 'Powell least squares'

POWELL_MD = 'Powell minimum distance'

POWELL_MM = 'Powell minimax'

MAX_PARAMETERS: int = 99

Maximum number of fit parameters allowed.

MAX_POINTS: int = 100001

Maximum number of data points allowed.

MAX_VARIABLES: int = 30

Maximum number of x (stimulus) variables allowed.

class ResidualType(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: Enum

Residual Type that is used to evaluate the *eof*.

DX_X = 'dx v x'

Uncertainty in x versus x.

DX_Y = 'dx v y'

Uncertainty in x versus y.

DY_X = 'dy v x'

Uncertainty in y versus x.

DY_Y = 'dy v y'

Uncertainty in y versus y.

static create_parameters(*parameters=None*)

Create a new collection of *InputParameter*s.

Parameters

parameters (*Iterable[InputParameterType]*) – An iterable of either *InputParameter* instances or objects that can be used to create an *InputParameter* instance. See *add_many()* for examples. If not specified, an empty collection is returned.

Returns

The input parameters.

Return type

InputParameters

property dll_path: str

Returns the path to the DLL file.

property equation: str

Returns the fitting equation.

evaluate(*x*, *result*)

Evaluate the model to get the *y* (response) values.

Parameters

- **x** (ArrayLike) – The independent variable (stimulus) data to evaluate the model at. If the model requires multiple variables, the *x* array must have a shape of *(num variables, num points)*, i.e., the data for each variable is listed per row

  [[data for x1], [data for x2], …]

- **result** (*Result | dict[str, float]*) – The fit result or a mapping between parameter names and values, e.g., *{'a1': 9.51, 'a2': -0.076, 'a3': 0.407}*.

Returns

The *y* (response) values.

Return type
ndarray

fit(*x:* ArrayLike, *y:* ArrayLike1D, *, *params:* ArrayLike1D | InputParameters = *None*, *ux:* ArrayLike = *None*, *uy:* ArrayLike1D = *None*, *debug:* *Literal[False]* = *False*, ***options*) → *Result*

fit(*x:* ArrayLike, *y:* ArrayLike1D, *, *params:* ArrayLike1D | InputParameters = *None*, *ux:* ArrayLike = *None*, *uy:* ArrayLike1D = *None*, *debug:* *Literal[True]*, ***options*) → *Input*

Fit the model to the data.

Tip: It is more efficient to use an ndarray rather than a list for the *x*, *y*, *ux* and *uy* arrays.

Parameters

- **x** – The independent variable (stimulus) data. If the model requires multiple variables, the *x* array must have a shape of *(num variables, num points)*, i.e., the data for each variable is listed per row

  [[data for x1], [data for x2], …]

- **y** – The dependent variable (response) data.
- **params** – Fit parameters. If an array is passed in then every parameter will be allowed to vary during the fit. If you want more control, pass in an *InputParameters* instance. If not specified, then the parameters are chosen from the *guess()* method.
- **ux** – Standard uncertainties in the x data.
- **uy** – Standard uncertainties in the y data.
- **debug** – If enabled, a summary of the input data that would be passed to the fit function in the DLL is returned (the DLL function is not called). Enabling this parameter is useful for debugging issues if the DLL raises an error or if the fit result is unexpected (e.g., the data points with smaller uncertainties are not having a stronger influence on the result, perhaps because an unweighted fit has been selected as one of the fit *options*).
- ****options** – All additional keyword arguments are passed to *options()*.

Returns
The returned type depends on whether *debug* mode is enabled or disabled. If *debug* is True then an *Input* object is returned, otherwise a *Result* object is returned.

Return type
Result or *Input*

guess(*x*, *y*, ***kwargs*)

Generate an initial guess for the parameters of a *Model*.

Parameters

- **x** (ArrayLike) – The independent variable (stimulus) data. If the model requires multiple variables, the *x* array must have a shape of *(num variables, num points)*, i.e., the data for each variable is listed per row

  [[data for x1], [data for x2], ...]

- **y** (ArrayLike1D) – The dependent variable (response) data.
- ****kwargs** – All additional keyword arguments are passed to the *Model* subclass.

Returns
Initial guesses for the parameters of a *Model*. Each *constant* value is set to False and a *label* is chosen. The values of these attributes can be changed by the user in the returned *InputParameters* object.

Return type
InputParameters

Raises
NotImplementedError – If the *guess* method is not implemented.

property num_parameters: int

Returns the number of fitting parameters in the equation.

property num_variables: int

Returns the number of *x* (stimulus) variables in the equation.

options(*, *absolute_residuals=None*, *correlated=None*, *delta=None*, *max_iterations=None*, *fit_method=None*, *residual_type=None*, *second_derivs_B=None*, *second_derivs_H=None*, *tolerance=None*, *uy_weights_only=None*, *weighted=None*)

Configure the fitting options.

Parameters

- **absolute_residuals** (*bool*) – Whether absolute residuals or relative residuals are used to evaluate the *eof*. Default: True (absolute).
- **correlated** (*bool*) – Whether to include the correlations in the fitting process. Including correlations in the fit is only possible for least-squares fitting, in which case the fit becomes a generalised least-squares fit. The correlations between the correlated variables can be set by calling *set_correlation()* or *set_correlation_dir()*. Default: False.
- **delta** (*float*) – Only used for Amoeba fitting. Default: 0.1.
- **max_iterations** (*int*) – The maximum number of fit iterations allowed. Default: 999.
- **fit_method** (FitMethod | *str*) – The fitting method to use. Can be a member name or value of the *FitMethod* enum. Default: Levenberg-Marquardt.
- **residual_type** (ResidualType | *str*) – The residual type to use to evaluate the *eof*. Can be a member name or value of the *ResidualType* enum. Default: DY_X (uncertainty in y versus x).

- **second_derivs_B** (*bool*) – Whether the second derivatives in the **B** matrix are included in the propagation of uncertainty calculations. Default: True.
- **second_derivs_H** (*bool*) – Whether the second derivatives in the curvature matrix, **H** (Hessian), are included in the propagation of uncertainty calculations. Default: True.
- **tolerance** (*float*) – The fitting process will stop when the relative change in chi-square (or some other appropriate measure) is less than this value. Default: 1e-20.
- **uy_weights_only** (*bool*) – Whether the *y* uncertainties only or a combination of the *x* and *y* uncertainties are used to calculate the weights for a weighted fit. Default: False.
- **weighted** (*bool*) – Whether to include the standard uncertainties in the fitting process to perform a weighted fit. Default: False.

Return type
None

remove_correlations()

Set all variables to be uncorrelated.

Return type
None

save(*path*, *, *x=None*, *y=None*, *params=None*, *ux=None*, *uy=None*, *comments=None*, *overwrite=False*)

Save a **.nlf** file.

The file can be opened in the Delphi GUI application or loaded via the *load()* function.

No information about the fit results are written to the file. If you are opening the file in the Delphi GUI, you must click the *Calculate* button to perform the fit and create the graphs.

Parameters

- **path** (*str*) – The path to save the file to. The file extension must be **.nlf**.
- **x** (ArrayLike) – The independent variable (stimulus) data. If not specified, the data that was most recently passed to *fit()* or a previous call to *save()* is used.
- **y** (ArrayLike1D) – The dependent variable (response) data. If not specified, the data that was most recently passed to *fit()* or a previous call to *save()* is used.
- **params** (ArrayLike1D | InputParameters) – Fit parameters. If not specified, the parameters that were most recently passed to *fit()* or a previous call to *save()* are used. Since the Delphi GUI application does not use the *label* attribute, the *labels* are not saved and will be None when the file is reloaded.
- **ux** (ArrayLike) – Standard uncertainties in the x data. If not specified, the data that was most recently passed to *fit()* is used.
- **uy** (ArrayLike1D) – Standard uncertainties in the y data. If not specified, the data that was most recently passed to *fit()* is used.

- **comments** (*str*) – Additional comments to add to the file. This text will appear in the *Comments* window in the Delphi GUI application.
- **overwrite** (*bool*) – Whether to overwrite the file if it already exists. If the file exists, and this value is `False` then an error is raised.

Return type
None

set_correlation(*n1*, *n2*, *, *matrix=None*, *value=None*)

Set the correlation coefficients for the correlated variables.

Note that the *x1-x2* correlation coefficients are identically equal to the *x2-x1* correlation coefficients, so only one of these relations needs to be defined.

> **Warning:** It is recommended to not call *set_correlation()* and *set_correlation_dir()* with the same *Model* instance. Pick only one method. If you set correlations using both methods an error will *not* be raised, but you *may* be surprised which correlations are used.

Parameters

- **n1** (*str*) – The name of the first correlated variable (e.g., *y*, *x*, *x1*, *x2*).
- **n2** (*str*) – The name of the second correlated variable.
- **matrix** (*ArrayLike*) – The coefficients of the correlation matrix.
- **value** (*float*) – Set all off-diagonal correlation coefficients to this value.

Return type
None

set_correlation_dir(*directory*)

Set the directory where the correlation coefficients are located.

The directory should contain correlation-coefficient files that must be named *CorrCoeffs Y-Y.txt*, *CorrCoeffs X1-X1.txt*, *CorrCoeffs X1-X2.txt*, etc. Note that the *X1-X2* correlation coefficients are identically equal to the *X2-X1* correlation coefficients, so only one of the files *CorrCoeffs X1-X2.txt* or *CorrCoeffs X2-X1.txt* needs to be created.

Whitespace is used to separate the value for each column in a file.

> **Warning:** It is recommended to not call *set_correlation()* and *set_correlation_dir()* with the same *Model* instance. Pick only one method. If you set correlations using both methods an error will *not* be raised, but you *may* be surprised which correlations are used.

Parameters
directory (*str | None*) – The directory (folder) where the correlation coefficients are located. Specify `.` for the current working directory.

Return type
None

property show_warnings: bool

Whether warning messages are shown.

Warnings are shown if correlations are defined and the fit option is set to be uncorrelated, or if *ux* or *uy* are specified and the fit option is unweighted, or if the maximum number of fit iterations has been exceeded.

property user_function_name: str

Returns the name of the user-defined function.

This is the value that *GetFunctionName* returns. If a user-defined function is not used, an empty string is returned. See *User-Defined Function*.

version()

Get the version number of the DLL.

Returns
The version of the DLL.

Return type
str

1.6.6 msl.nlf.models module

Predefined models

- *ConstantModel*
- *ExponentialModel*
- *GaussianModel*
- *LinearModel*
- *PolynomialModel*
- *SineModel*

class msl.nlf.models.**ConstantModel**(***kwargs*)

Bases: *Model*

A model based on a constant (i.e., a single parameter with no x dependence).

The function is defined as

$$f(x; a) = a_1$$

Parameters
****kwargs** – All keyword arguments are passed to *Model*.

guess(*x*, *y*, *, *n=None*)

Calculates the mean value of *y*.

Parameters

- **x** (ArrayLike1D) – The independent variable (stimulus) data. The data is not used.
- **y** (ArrayLike1D) – The dependent variable (response) data.

- **n** (*int*) – The number of values in *y* to use to calculate the mean. If not specified, all values are used. If a positive integer then the first *n* values are used. Otherwise, the last *n* values are used.

Returns
Initial guess for the constant.

Return type
InputParameters

class msl.nlf.models.**ExponentialModel**(*cumulative=False*, ***kwargs*)

Bases: *Model*

A model based on an exponential function.

The non-cumulative function is defined as

$$f(x;a) = a_1 e^{-a_2 x}$$

whereas, the cumulative function is defined as

$$f(x;a) = a_1(1 - e^{-a_2 x})$$

Parameters

- **cumulative** (*bool*) – Whether to use the cumulative function.
- ****kwargs** – All keyword arguments are passed to *Model*.

guess(*x*, *y*, ***, *n=3*)

Linearizes the equation and calls the `polyfit()` function.

Parameters

- **x** (*ArrayLike1D*) – The independent variable (stimulus) data.
- **y** (*ArrayLike1D*) – The dependent variable (response) data.
- **n** (*int*) – For a cumulative equation, uses the maximum *n* values in *y* to calculate the mean and assigns the mean value as the amplitude guess.

Returns
Initial guess for the amplitude and decay factor.

Return type
InputParameters

class msl.nlf.models.**GaussianModel**(*normalized=False*, ***kwargs*)

Bases: *Model*

A model based on a Gaussian function or a normal distribution.

The non-normalized function is defined as

$$f(x;a) = a_1 e^{-\frac{1}{2}(\frac{x-a_2}{a_3})^2}$$

whereas, the normalized function is defined as

$$f(x;a) = \frac{a_1}{a_3\sqrt{2\pi}} e^{-\frac{1}{2}(\frac{x-a_2}{a_3})^2}$$

Parameters

- **normalized** (*bool*) – Whether to use the normalized function.
- ****kwargs** – All additional keyword arguments are passed to *Model*.

guess(*x*, *y*, *, *n=3*)

Converts the data to a quadratic and calls the `polyfit()` function.

Parameters

- **x** (ArrayLike1D) – The independent variable (stimulus) data.
- **y** (ArrayLike1D) – The dependent variable (response) data.
- **n** (*int*) – Uses the *n* maximum and the *n* minimum values in *y* to determine the region where the peak/dip is located.

Returns

Initial guess for the amplitude (area), μ and σ parameters.

Return type

InputParameters

class msl.nlf.models.**LinearModel**(***kwargs*)

Bases: *Model*

A model based on a linear function.

The function is defined as

$$f(x; a) = a_1 + a_2 x$$

Parameters

****kwargs** – All keyword arguments are passed to *Model*.

guess(*x*, *y*, ***kwargs*)

Calls the `polyfit()` function.

Parameters

- **x** (ArrayLike1D) – The independent variable (stimulus) data.
- **y** (ArrayLike1D) – The dependent variable (response) data.
- ****kwargs** – No keyword arguments are used.

Returns

Initial guess for the intercept and slope.

Return type

InputParameters

class msl.nlf.models.**PolynomialModel**(*n*, ***kwargs*)

Bases: *Model*

A model based on a polynomial function.

The function is defined as

$$f(x; a) = \sum_{i=1}^{n} a_i x^{i-1}$$

Parameters

- **n** (*int*) – The order of the polynomial (n $\geq$ 1).
- ****kwargs** – All keyword arguments are passed to *Model*.

guess(*x*, *y*, ***kwargs*)

Calls the `polyfit()` function.

Parameters

- **x** (`ArrayLike1D`) – The independent variable (stimulus) data.
- **y** (`ArrayLike1D`) – The dependent variable (response) data.
- ****kwargs** – No keyword arguments are used.

Returns

Initial guess for the polynomial coefficients.

Return type

InputParameters

class msl.nlf.models.**SineModel**(***kwargs*)

Bases: *Model*

A model based on a sine function.

The function is defined as

$$f(x; a) = a_1 \sin(a_2 x + a_3)$$

Parameters

****kwargs** – All keyword arguments are passed to *Model*.

guess(*x*, *y*, ***, *uniform=True*, *n=11*)

Uses an FFT to determine the amplitude and angular frequency.

Parameters

- **x** (`ArrayLike1D`) – The independent variable (stimulus) data. The x data must be sorted.
- **y** (`ArrayLike1D`) – The dependent variable (response) data.
- **uniform** (*bool*) – Whether the x data has uniform spacing between each value.
- **n** (*int*) – The number of sub-intervals to break up [0, 2π) to determine the phase guess.

Returns

Initial guess for the amplitude, angular frequency and phase.

Return type

InputParameters

1.6.7 msl.nlf.parameter module

Parameters are used as inputs to and results from a fit model.

class msl.nlf.parameter.**InputParameter**(*name*, *value*, *, *constant=False*, *label=None*)

Bases: *Parameter*

A parameter to use as an input to a fit model.

Parameters

- **name** (*str*) – The name of the parameter in the equation (e.g., *a1*).
- **value** (*float*) – The value of the parameter.
- **constant** (*bool*) – Whether the parameter is held constant (True) or allowed to vary (False) during the fitting process.
- **label** (*str*) – A custom label associated with the parameter. For example, if the fit equation is **a1+a2*x**, you could assign a label of *intercept* to *a1* and *slope* to *a2*.

property constant: bool

Whether the parameter is held constant (True) or allowed to vary (False) during the fitting process.

property label: str | None

A custom label associated with the parameter. For example, if the fit equation is **a1+a2*x**, you could assign a label of *intercept* to *a1* and *slope* to *a2*.

property name: str

The name of the parameter in the equation (e.g., *a1*).

property value: float

The value of the parameter.

msl.nlf.parameter.**InputParameterType**

Allowed types to create an *InputParameter*.

alias of Union[*InputParameter*, Tuple[str, float], Tuple[str, float, bool], Tuple[str, float, bool, Optional[str]], List, Dict[str, Optional[Union[str, float, bool]]]]

class msl.nlf.parameter.**InputParameters**(*parameters=None*)

Bases: *Parameters*[*InputParameter*]

A collection of *InputParameter*s for a fit model.

Parameters

parameters (*Iterable[InputParameterType]*) – An iterable of either *InputParameter* instances or objects that can be used to create an *InputParameter* instance. See *add()* for examples.

add(**args*, ***kwargs*)

Add an *InputParameter*.

An *InputParameter* can be added using either positional or keyword arguments, but you cannot use both simultaneously. You can specify positional arguments by using one of four options:

- InputParameter (a single argument must be an *InputParameter*)
- name, value
- name, value, constant
- name, value, constant, label

You could alternatively add an *InputParameter* in the same way that you add items to a dict

Returns
The input parameter that was added.

Return type
InputParameter

Examples

```
>>> from msl.nlf import InputParameter, InputParameters
>>> params = InputParameters()
>>> a1 = params.add('a1', 1)
>>> a2 = params.add('a2', 0.34, True)
>>> a3 = params.add('a3', -1e3, False, 'offset')
>>> a4 = params.add(name='a4', value=3.14159, constant=True, label=
↪'pi')
>>> a5 = params.add(name='a5', value=100)
>>> a6 = params.add(InputParameter('a6', 32.0))
>>> params['a7'] = InputParameter('a7', 7, constant=True)
>>> params['a8'] = 88.8
>>> params['a9'] = (-1, True)
>>> params['a10'] = {'value': 0, 'label': 'intercept'}
>>> for param in params:
...     print(f'{param.name}={param.value}')
a1=1.0
a2=0.34
a3=-1000.0
a4=3.14159
a5=100.0
a6=32.0
a7=7.0
a8=88.8
a9=-1.0
a10=0.0
```

add_many(*parameters*)

Add many *InputParameter*s.

Parameters
parameters (*Iterable[InputParameterType]*) – An iterable of either *InputParameter* instances or objects that can be used to create an *InputParameter* instance. See *add()* for more examples.

Return type
None

Examples

```
>>> from msl.nlf import InputParameter, InputParameters
>>> inputs = (InputParameter('a1', 1),
...           ('a2', 2, True),
...           {'name': 'a3', 'value': 3})
>>> params = InputParameters()
>>> params.add_many(inputs)
>>> for param in params:
...     print(param)
InputParameter(name='a1', value=1.0, constant=False, label=None)
InputParameter(name='a2', value=2.0, constant=True, label=None)
InputParameter(name='a3', value=3.0, constant=False, label=None)
```

clear()

Remove all *InputParameter*s from the collection.

Return type
None

constants()

Returns the *constant* of each parameter.

Return type
ndarray[bool]

labels()

Returns the *label* of each parameter.

Return type
list[str | None]

names()

Returns the *name* of each parameter.

Return type
list[str]

pop(*name_or_label*)

Pop an *InputParameter* from the collection.

This will remove the *InputParameter* from the collection and return it.

Parameters
name_or_label (*str*) – The *name* or *label* of an *InputParameter*.

Returns
The input parameter that was popped.

Return type
InputParameter

update(*name_or_label*, ***attribs*)

Update the attributes of an *InputParameter*.

Parameters

- **name_or_label** (*str*) – The *name* or *label* of an *InputParameter*.
- ****attribs** – The new attributes.

Return type

None

Examples

First, add a parameter

```
>>> from msl.nlf import InputParameters
>>> params = InputParameters()
>>> a1 = params.add('a1', 1)
>>> a1
InputParameter(name='a1', value=1.0, constant=False, label=None)
```

then update it by calling the *update()* method

```
>>> params.update('a1', value=0, constant=True, label='intercept')
>>> a1
InputParameter(name='a1', value=0.0, constant=True, label='intercept
↪')
```

Alternatively, you can update a parameter by directly modifying an attribute

```
>>> a1.label = 'something-new'
>>> a1.constant = False
>>> a1.value = -3.2
>>> params['a1']
InputParameter(name='a1', value=-3.2, constant=False, label=
↪'something-new')
```

values()

Returns the *value* of each parameter.

Return type

ndarray[float]

class msl.nlf.parameter.**Parameter**(*name*, *value*, *, *label=None*)

Bases: object

A generic parameter used as an input to or a result from a fit model.

Parameters

- **name** (*str*) – The name of the parameter in the equation (e.g., *a1*).
- **value** (*float*) – The value of the parameter.

- **label** (*str*) – A custom label associated with the parameter. For example, if the fit equation is **a1+a2*x**, you could assign a label of *intercept* to *a1* and *slope* to *a2*.

property label: str | None

A custom label associated with the parameter. For example, if the fit equation is **a1+a2*x**, you could assign a label of *intercept* to *a1* and *slope* to *a2*.

property name: str

The name of the parameter in the equation (e.g., *a1*).

property value: float

The value of the parameter.

class msl.nlf.parameter.**Parameters**

Bases: Generic[*T*]

Base class for a collection of parameters.

labels()

Returns the *label* of each parameter.

Return type
list[str | None]

names()

Returns the *name* of each parameter.

Return type
list[str]

values()

Returns the *value* of each parameter.

Return type
ndarray[float]

class msl.nlf.parameter.**ResultParameter**(*name*, *value*, *uncert*, *, *label=None*)

Bases: *Parameter*

A parameter that is returned from a fit model.

Parameters

- **name** (*str*) – The name of the parameter in the equation (e.g., *a1*).
- **value** (*float*) – The value of the parameter.
- **uncert** (*float*) – The standard uncertainty of the parameter.
- **label** (*str*) – A custom label associated with the parameter. For example, if the fit equation is **a1+a2*x**, you could assign a label of *intercept* to *a1* and *slope* to *a2*.

property label: str | None

A custom label associated with the parameter. For example, if the fit equation is **a1+a2*x**, you could assign a label of *intercept* to *a1* and *slope* to *a2*.

property name: *str*

The name of the parameter in the equation (e.g., *a1*).

property uncert: *float*

The standard uncertainty of the parameter.

property value: *float*

The value of the parameter.

class msl.nlf.parameter.**ResultParameters**(*result*, *params*)

Bases: *Parameters*[*ResultParameter*]

A collection of *ResultParameter*s from a fit model.

Parameters

- **result** (*dict*) – The result from a fit model.
- **params** (*InputParameters*) – The input parameters to the fit model.

labels()

Returns the *label* of each parameter.

Return type
list[str | None]

names()

Returns the *name* of each parameter.

Return type
list[str]

uncerts()

Returns the *uncert* of each parameter.

Return type
ndarray[float]

values()

Returns the *value* of each parameter.

Return type
ndarray[float]

class msl.nlf.parameter.**T**

Generic parameter type.

alias of TypeVar('T', ~msl.nlf.parameter.InputParameter, ~msl.nlf.parameter.ResultParameter)

1.6.8 msl.nlf.saver module

Save a **.nlf** file.

class msl.nlf.saver.**Saver**(*version*)

Bases: object

Helper class to create a **.nlf** file.

Parameters
version (*str*) – The DLL version number.

save(*path*)

Save the buffer to a **.nlf** file.

Parameters
path (*str*) – The **.nlf** file path.

Return type
None

write_boolean(*value*)

Write a boolean.

Parameters
value (*bool*) – Write *value* to the buffer.

Return type
None

write_byte(*value*)

Write a byte.

Parameters
value (*int*) – Write *value* to the buffer.

Return type
None

write_extended(*value*)

Write a Delphi 10-byte extended float.

Parameters
value (*float*) – Write *value* to the buffer.

Return type
None

write_integer(*value*)

Write an unsigned integer.

Parameters
value (*int*) – Write *value* to the buffer.

Return type
None

write_string(*value*)

Write a string.

Parameters
value (*str*) – Write *value* to the buffer.

Return type
None

write_string_padded(*value*, *pad*)

Write a string with null padding at the end.

Parameters

- **value** (*str*) – Write *value* to the buffer.
- **pad** (*int*) – The total number of bytes the string must be. Pads null bytes until the number of bytes written is the appropriate length.

Return type
None

write_word(*value*)

Write an unsigned short.

Parameters
value (*int*) – Write *value* to the buffer.

Return type
None

msl.nlf.saver.**save**(*, *path*, *comments*, *overwrite*, *data*)

Save a **.nlf** file.

The file can be opened in the Delphi GUI application or loaded via the *load()* function.

Parameters

- **path** (*str*) – The **.nlf** file path.
- **comments** (*str*) – Additional comments to add to the file. This text will appear in the *Comments* window in the Delphi GUI application.
- **overwrite** (*bool*) – Whether to overwrite the file if it already exists. If the file exists, and this value is False, then an error is raised.
- **data** (Input) – The input data to the fit model.

Return type
None

msl.nlf.saver.**save_form**(*saver*, *data*)

Save a *TDataForm*.

Parameters

- **saver** (Saver) – The class helper.
- **data** (*dict*) – The data to add to the form. The x, y, ux, ux arrays should be added to the Data form. The Results form should have the appropriate number of columns and number of rows. Covariance forms is not populated.

Return type
None

msl.nlf.saver.**save_graph**(*saver*, *name*)

Save a *TGraphWindow*.

Parameters

- **saver** (Saver) – The class helper.
- **name** (*str*) – The name of the graph window.

Return type
None

1.7 License

```
MIT License

Copyright (c) 2023, Measurement Standards Laboratory of New Zealand

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

1.8 Developers

- Joseph Borbely <joseph.borbely@measurement.govt.nz>
- Peter Saunders <peter.saunders@measurement.govt.nz>

1.9 Release Notes

1.9.1 Version 5.43.0 (in development)

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

m

INDEX

N

O

P

R

S

T

U

V

W

X

Y