iminuit Documentation

Release 1.4.9

Piti Ongmongkolkul

Jul 29, 2020

Contents

iminuit is a Jupyter-friendly Python frontend to the *MINUIT2* C++ library.

It can be used as a general robust function minimisation method, but is most commonly used for likelihood fits of models to data, and to get model parameter error estimates from likelihood profile analysis.

- Supported CPython versions: 3.5+
- Supported PyPy versions: 3.5, 3.6
- Supported platforms: Linux, OSX and Windows.
- PyPI: https://pypi.org/project/iminuit
- Documentation: http://iminuit.readthedocs.org
- Source: https://github.com/scikit-hep/iminuit
- Gitter: https://gitter.im/Scikit-HEP/community
- License: *MINUIT2* is LGPL and *iminuit* is MIT
- Citation: https://doi.org/10.5281/zenodo.3949207

CHAPTER 1

In a nutshell

```
from iminuit import Minuit

def f(x, y, z):
    return (x - 2) ** 2 + (y - 3) ** 2 + (z - 4) ** 2

m = Minuit(f)

m.migrad()  # run optimiser
print(m.values)  # {'x': 2,'y': 3,'z': 4}

m.hesse()   # run covariance estimator
print(m.errors)  # {'x': 1,'y': 1,'z': 1}
```

1.1 About

1.1.1 What is iminuit?

iminuit is the fast interactive IPython-friendly minimiser based on Minuit2 in ROOT-6.12.06.

For a hands-on introduction, see the *Tutorials*.

Easy to install

You can install iminuit with pip. It only needs a moderately recent C++ compiler on your machine. The Minuit2 code is bundled, so you don't need to install it separately.

Support for Python 3.5+, PyPy-3.5 to PyPy-3.6

Whether you use the latest Python 3 or stick to classic Python 2, iminuit works for you. We even support the latest PyPy. iminuit interoperates with NumPy. In addition to passing parameters individually, you can minimise functions that accept parameters as NumPy arrays. You can get the fit results as NumPy arrays.

Robust optimiser and error estimator

iminuit uses Minuit2 to minimise your functions, a battle-hardened code developed and maintained by scientists at CERN, the world's leading particle accelerator laboratory. Minuit2 has good performance compared to other minimisers, and it is one of the few codes out there which compute error estimates for your parameters. When you do statistics seriously, this is a must-have.

Interactive convenience

iminuit extracts the parameter names from your function signature (or the docstring) and allows you access them by their name. For example, if your function is defined as `func(alpha, beta)`, iminuit understands that your first parameter is *alpha* and the second *beta* and will use these names in status printouts (you can override this inspection if you like). It also produces pretty messages on the console and in Jupyter notebooks.

Support for Cython

iminuit was designed to work with Cython functions, in order to speed up the minimisation of complex functions.

Successor of PyMinuit

iminuit is mostly compatible with PyMinuit. Existing PyMinuit code can be ported to iminuit by just changing the import statement.

If you are interested in fitting a curve or distribution, take a look at probfit.

1.1.2 Who is using iminuit?

This is a list of known users of iminuit. Please let us know if you use iminuit, we like to keep in touch.

- probfit
- gammapy
- flavio
- Veusz
- TensorProb
- threeML
- pyhf
- zfit
- ctapipe
- lauztat

1.1.3 Technical docs

When you use iminuit/Minuit2 seriously, it is a good idea to understand a bit how it works and what possible limitations are in your case. The following links help you to understand the numerical approach behind Minuit2. The links are ordered by recommended reading order.

- The MINUIT paper by Fred James and Matts Roos, 1975.
- Wikipedia articles for the Quasi Newton Method and DFP formula used by MIGRAD.
- Variable Metric Method for Minimization by William Davidon, 1991.
- Original user guide for C++ Minuit2: `MINUIT User's guide` by Fred James, 2004.

1.1.4 Team

iminuit was created by **Piti Ongmongkolkul**. It is a logical successor of pyminuit/pyminuit2, created by **Jim Pivarski**. It is now maintained by **Hans Dembinski** and the Scikit-HEP community.

1.1.4.1 Maintainers

- Hans Dembinski (@HDembinski) [current]
- Christoph Deil (@cdeil)
- Piti Ongmongkolkul (@piti118)
- Chih-hsiang Cheng (@gitcheng)

1.1.4.2 Contributors

- Jim Pivarski (@jpivarski)
- David Men'endez Hurtado (@Dapid)
- Chris Burr (@chrisburr)
- Andrew ZP Smith (@energynumbers)
- Fabian Rost (@fabianrost84)
- Alex Pearce (@alexpearce)
- Lukas Geiger (@lgeiger)
- Omar Zapata (@omazapa)

1.2 Installation

Note: iminuit is tested to work with PyPy3.5 and PyPy3.6, but we do not provide binary packages for PyPy. PyPy users need to install the source package of iminuit. This happens automatically when you install it via conda or pip, but requires a working C++ compiler.

1.2.1 Conda

We provide binary packages for *conda* users via https://anaconda.org/conda-forge/iminuit:

```
$ conda install -c conda-forge iminuit
```

iminuit only depends on *numpy*. The conda packages are semi-automatically maintained and usually quickly support the least Python version on all platforms.

1.2.2 pip

To install the latest stable version from https://pypi.org/project/iminuit/ with *pip*:

```
$ pip install iminuit
```

If your platform is not supported by a binary wheel, *pip install* requires that you have a C++ compiler available but otherwise runs the compilation automatically. As an alternative you can try to install iminuit with conda.

1.2.3 Installing from source

1.2.3.1 For users

If you need the latest unreleased version, you can download and install directly from Github. The easiest way is to use pip.

> pip install git+https://github.com/scikit-hep/iminuit@develop#egg=iminuit

1.2.3.2 For contributors/developers

See *Contribute*.

1.2.4 Check installation

To check your *iminuit* version number and install location:

```
$ python
>>> import iminuit
>>> iminuit
# install location is printed
>>> iminuit.__version__
# version number is printed
```

Usually if *import iminuit* works, everything is OK. But in case you suspect that you have a broken *iminuit* installation, you can run the automated tests like this:

```
$ pip install pytest
$ python -c "import iminuit; iminuit.test()"
```

1.3 Tutorials

All the tutorials are in tutorial directory of the iminuit repository and can be viewed online.

It is recommended to start with the basic tutorial. All other tutorials are optional, for those who want to know more about specific aspects.

1.4 Reference

1.4.1 Quick Summary

These methods and attributes you will use a lot:

`Minuit`(fcn[, grad, errordef, print_level, . . .])	Construct minuit object from given *fcn*

Continued on next page

Table 1 – continued from previous page

Minuit.from_array_func(type cls, fcn, start)	Construct Minuit object from given *fcn* and start sequence.
Minuit.migrad(self[, ncall, resume, precision])	Run MIGRAD.
Minuit.hesse(self[, ncall])	Run HESSE to compute parabolic errors.
Minuit.minos(self[, var, sigma, ncall])	Run MINOS to compute asymmetric confidence intervals.
Minuit.values	values: iminuit._libiminuit.ValueView Parameter values in a dict-like object.
Minuit.fixed	fixed: iminuit._libiminuit.FixedView Access fixation state of a parameter in a dict-like object.
Minuit.valid	Check if function minimum is valid.
Minuit.accurate	Check if covariance (of the last MIGRAD run) is accurate.
Minuit.fval	Last evaluated FCN value
Minuit.nfit	Number of fitted parameters (fixed parameters not counted).
Minuit.np_values(self)	Parameter values in numpy array format.
Minuit.np_covariance(self)	Covariance matrix in numpy array format.
Minuit.np_errors(self)	Hesse parameter errors in numpy array format.
Minuit.np_merrors(self)	MINOS parameter errors in numpy array format.
Minuit.mnprofile(self, vname[, bins, bound, . . .])	Calculate MINOS profile around the specified range.
Minuit.draw_mnprofile(self, vname[, bins, . . .])	Draw MINOS profile in the specified range.

1.4.2 Minuit

class iminuit.**Minuit**(*fcn*, *grad=None*, *errordef=None*, *print_level=0*, *name=None*, *pedantic=True*, *throw_nan=False*, *use_array_call=False*, ***kwds*)

Construct minuit object from given *fcn*

Arguments:

fcn, the function to be optimized, is the only required argument.

Two kinds of function signatures are understood.

a) Parameters passed as positional arguments

The function has several positional arguments, one for each fit parameter. Example:

```
def func(a, b, c): ...
```

The parameters a, b, c must accept a real number.

iminuit automagically detects parameters names in this case. More information about how the function signature is detected can be found in *Function Signature Extraction Ordering*

b) Parameters passed as Numpy array

The function has a single argument which is a Numpy array. Example:

```
def func(x): ...
```

Pass the keyword *use_array_call=True* to use this signature. For more information, see “Parameter Keyword Arguments” further down.

If you work with array parameters a lot, have a look at the static initializer method `from_array_func()`, which adds some convenience and safety to this use case.

Builtin Keyword Arguments:

- **throw_nan**: set fcn to raise RuntimeError when it encounters *nan*. (Default False)
- **pedantic**: warns about parameters that do not have initial value or initial error/stepsize set.
- **name**: sequence of strings. If set, this is used to detect parameter names instead of iminuit's function signature detection.
- **print_level**: set the print_level for this Minuit. 0 is quiet. 1 print out at the end of MIGRAD/HESSE/MINOS. 2 prints debug messages.
- **errordef**: Optional. See `errordef` for details on this parameter. If set to *None* (the default), Minuit will try to call *fcn.errordef* and *fcn.default_errordef()* (deprecated) to set the error definition. If this fails, a warning is raised and use a value appropriate for a least-squares function is used.
- **grad**: Optional. Provide a function that calculates the gradient analytically and returns an iterable object with one element for each dimension. If None is given MINUIT will calculate the gradient numerically. (Default None)
- **use_array_call**: Optional. Set this to true if your function signature accepts a single numpy array of the parameters. You need to also pass the *name* keyword then to explicitly name the parameters.

Parameter Keyword Arguments:

iminuit allows user to set initial value, initial stepsize/error, limits of parameters and whether the parameter should be fixed by passing keyword arguments to Minuit.

This is best explained through examples:

```python
def f(x, y):
    return (x-2)**2 + (y-3)**2
```

- Initial value (varname):

```python
#initial value for x and y
m = Minuit(f, x=1, y=2)
```

- Initial step size (fix_varname):

```python
#initial step size for x and y
m = Minuit(f, error_x=0.5, error_y=0.5)
```

- Limits (limit_varname=tuple):

```python
#limits x and y
m = Minuit(f, limit_x=(-10,10), limit_y=(-20,20))
```

- Fixing parameters:

```python
#fix x but vary y
m = Minuit(f, fix_x=True)
```

Note: You can use dictionary expansion to programmatically change parameters.:

```
kwargs = dict(x=1., error_x=0.5)
m = Minuit(f, **kwargs)
```

You can also obtain fit arguments from Minuit object for later reuse. *fitarg* will be automatically updated to the minimum value and the corresponding error when you ran migrad/hesse.:

```
m = Minuit(f, x=1, error_x=0.5)
my_fitarg = m.fitarg
another_fit = Minuit(f, **my_fitarg)
```

---

from_array_func(*type cls*, *fcn*, *start*, *error=None*, *limit=None*, *fix=None*, *name=None*, ***kwds*)
Construct Minuit object from given *fcn* and start sequence.

This is an alternative named constructor for the minuit object. It is more convenient to use for functions that accept a numpy array.

Arguments:

fcn: The function to be optimized. Must accept a single parameter that is a numpy array.

def func(x): . . .

start: Sequence of numbers. Starting point for the minimization.

Keyword arguments:

error: Optional sequence of numbers. Initial step sizes. Scalars are automatically broadcasted to the length of the start sequence.

limit: Optional sequence of limits that restrict the range in which a parameter is varied by minuit. Limits can be set in several ways. With inf = float("infinity") we get:

- No limit: None, (-inf, inf), (None, None)
- Lower limit: (x, None), (x, inf) [replace x with a number]
- Upper limit: (None, x), (-inf, x) [replace x with a number]

A single limit is automatically broadcasted to the length of the start sequence.

fix: Optional sequence of boolean values. Whether to fix a parameter to the starting value.

name: Optional sequence of parameter names. If names are not specified, the parameters are called x0, . . . , xN.

All other keywords are forwarded to `Minuit`, see its documentation.

Example:

A simple example function is passed to Minuit. It accept a numpy array of the parameters. Initial starting values and error estimates are given:

```
import numpy as np

def f(x):
    mu = (2, 3)
    return np.sum((x-mu)**2)

# error is automatically broadcasted to (0.5, 0.5)
m = Minuit.from_array_func(f, (2, 3),
                           error=0.5)
```

LEAST_SQUARES = 1.0

LIKELIHOOD = 0.5

accurate

Check if covariance (of the last MIGRAD run) is accurate.

args

args: iminuit._libiminuit.ArgsView Parameter values in a list-like object.

See `values` for details.

See also:

`values`, `errors`, `fixed`

contour (*self*, *x*, *y*, *bins=50*, *bound=2*, *subtract_min=False*, ***deprecated_kwargs*)

2D contour scan.

Return the contour of a function scan over **x** and **y**, while keeping all other parameters fixed.

The related `mncontour()` works differently: for new pair of **x** and **y** in the scan, it minimises the function with the respect to the other parameters.

This method is useful to inspect the function near the minimum to detect issues (the contours should look smooth). Use `mncontour()` to create confidence regions for the parameters. If the fit has only two free parameters, you can use this instead of `mncontour()`.

Arguments:

- **x** variable name for X axis of scan
- **y** variable name for Y axis of scan
- **bound** If bound is 2x2 array, [[v1min,v1max],[v2min,v2max]]. If bound is a number, it specifies how many σ symmetrically from minimum (minimum+- bound*:math:*sigma*). Default: 2.
- **subtract_min** Subtract minimum off from return values. Default False.

Returns:

x_bins, y_bins, values

values[y, x] <– this choice is so that you can pass it to through matplotlib contour()

See also:

`mncontour()` `mnprofile()`

covariance

Covariance matrix (dict (name1, name2) -> covariance).

See also:

`matrix()`

draw_contour (*self*, *x*, *y*, *bins=50*, *bound=2*, ***deprecated_kwargs*)

Convenience wrapper for drawing contours.

The arguments are the same as `contour()`.

Please read the docs of `contour()` and `mncontour()` to understand the difference between the two.

See also:

`contour()` `draw_mncontour()`

draw_mncontour (*self*, *x*, *y*, *nsigma=2*, *numpoints=100*)
Draw MINOS contour.

Arguments:

- **x, y** parameter name
- **nsigma** number of sigma contours to draw
- **numpoints** number of points to calculate for each contour

Returns:

contour

See also:

mncontour()

```
from iminuit import Minuit


def cost(x, y, z):
    return (x - 1) ** 2 + (y - x) ** 2 + (z - 2) ** 2


m = Minuit(cost, print_level=0, pedantic=False)
m.migrad()
m.draw_mncontour("x", "y", nsigma=4)
```

draw_mnprofile (*self*, *vname*, *bins=30*, *bound=2*, *subtract_min=False*, *band=True*, *text=True*)
Draw MINOS profile in the specified range.

It is obtained by finding MIGRAD results with **vname** fixed at various places within **bound**.

Arguments:

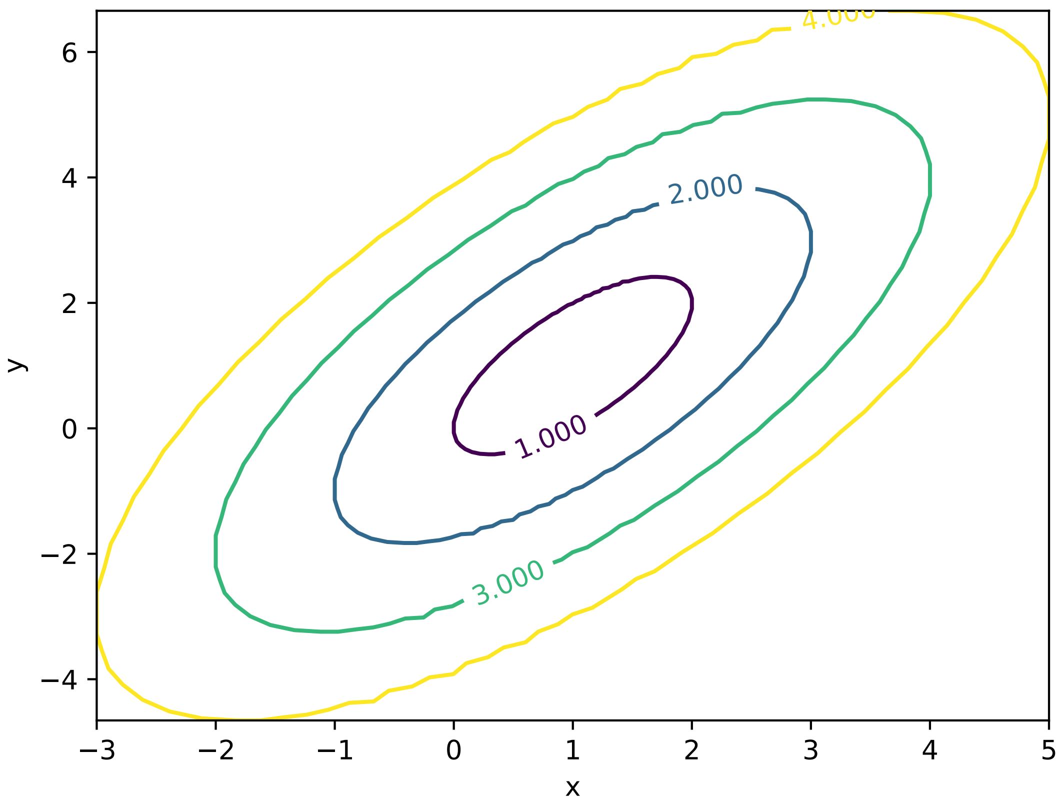
- **vname** variable name to scan
- **bins** number of scanning bin. Default 30.
- **bound** If bound is tuple, (left, right) scanning bound. If bound is a number, it specifies how many σ symmetrically from minimum (minimum+- bound* σ). Default 2.
- **subtract_min** subtract_minimum off from return value. This makes it easy to label confidence interval. Default False.
- **band** show green band to indicate the increase of fcn by *errordef*. Default True.
- **text** show text for the location where the fcn is increased by *errordef*. This is less accurate than *minos()*. Default True.

Returns:

bins(center point), value, migrad results

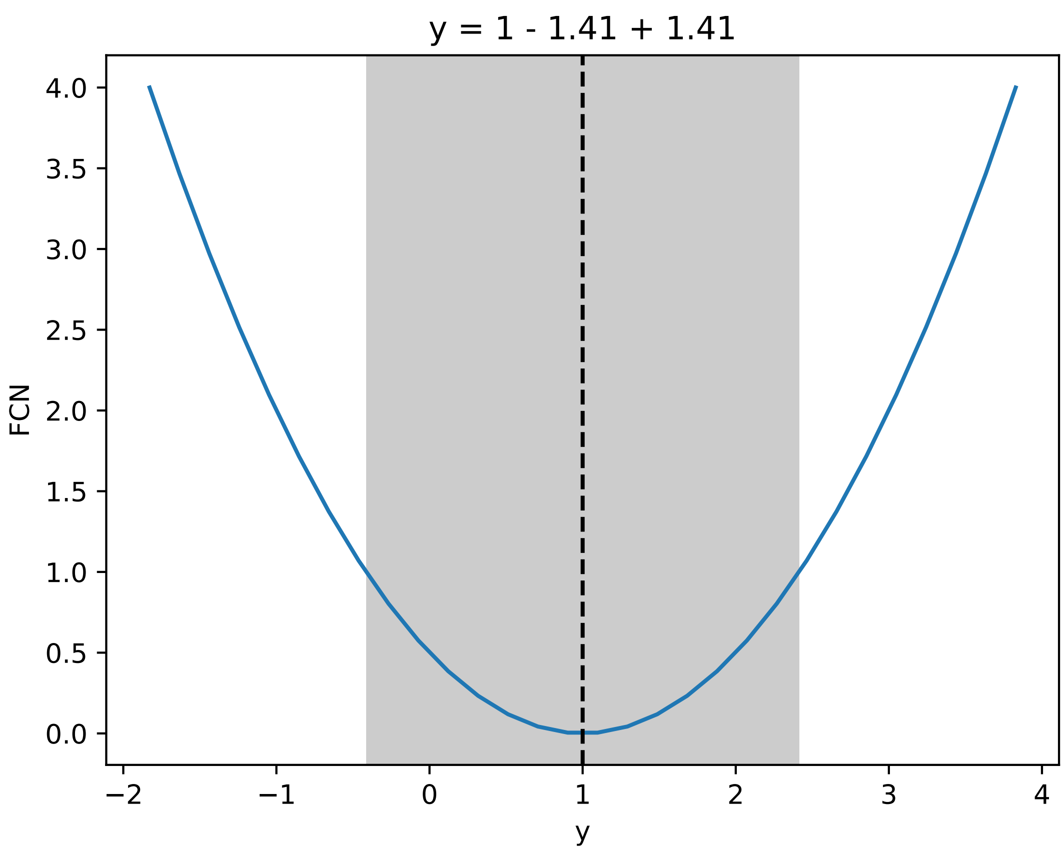
```
from iminuit import Minuit


def cost(x, y, z):
    return (x - 1) ** 2 + (y - x) ** 2 + (z - 2) ** 2


m = Minuit(cost, pedantic=False)
```

(continues on next page)


4.000
2.000
1.000
3.000
6
4
2
0
−2
−4
y
−3
−2
−1
0
1
2
3
4
5
x

(continued from previous page)

```
m.migrad()
m.draw_mnprofile("y")
```



draw_profile(*self*, *vname*, *bins=100*, *bound=2*, *subtract_min=False*, *band=True*, *text=True*, ***deprecated_kwargs*)

A convenient wrapper for drawing profile using matplotlib.

A 1D scan of the cost function around the minimum, useful to inspect the minimum and the FCN around the minimum for defects.

For a fit with several free parameters this is not the same as the MINOS profile computed by `draw_mncontour()`. Use `mnprofile()` or `draw_mnprofile()` to compute confidence intervals.

If a function minimum was found in a previous MIGRAD call, a vertical line indicates the parameter value. An optional band indicates the uncertainty interval of the parameter computed by HESSE or MINOS.

Arguments:

In addition to argument listed on `profile()`. draw_profile take these addition argument:

- **band** show green band to indicate the increase of fcn by *errordef*. Note again that this is NOT minos error in general. Default True.
- **text** show text for the location where the fcn is increased by *errordef*. This is less accurate than `minos()` Note again that this is NOT minos error in general. Default True.

See also:

mnprofile() draw_mnprofile() profile()

errordef

FCN increment above the minimum that corresponds to one standard deviation.

Default value is 1.0. *errordef* should be 1.0 for a least-squares cost function and 0.5 for negative log-likelihood function. See page 37 of http://hep.fi.infn.it/minuit.pdf. This parameter is sometimes called `UP` in the MINUIT docs.

To make user code more readable, we provided two named constants:

```
from iminuit import Minuit
assert Minuit.LEAST_SQUARES == 1
assert Minuit.LIKELIHOOD == 0.5

Minuit(a_least_squares_function, errordef=Minuit.LEAST_SQUARES)
Minuit(a_likelihood_function, errordef=Minuit.LIKELIHOOD)
```

errors

errors: iminuit._libiminuit.ErrorView Parameter parabolic errors in a dict-like object.

Like *values*, but instead of reading or writing the values, you read or write the errors (which double as step sizes for MINUITs numerical gradient estimation).

See also:

values, *fixed*

fcn

Cost function (usually a chi^2 or likelihood function).

fitarg

Current Minuit state in form of a dict.

- name -> value
- error_name -> error
- fix_name -> fix
- limit_name -> (lower_limit, upper_limit)

This is very useful when you want to save the fit parameters and re-use them later. For example:

```
m = Minuit(f, x=1)
m.migrad()
fitarg = m.fitarg

m2 = Minuit(f, **fitarg)
```

fixed

fixed: iminuit._libiminuit.FixedView Access fixation state of a parameter in a dict-like object.

Use to read or write the fixation state of a parameter based on the parameter index or the parameter name as a string. If you change the state and run *migrad()*, *hesse()*, or *minos()*, the new state is used.

In case of complex fits, it can help to fix some parameters first and only minimize the function with respect to the other parameters, then release the fixed parameters and minimize again starting from that state.

See also:

values, *errors*

fmin
Current function minimum data object

fval
Last evaluated FCN value

See also:

fmin()

gcc
Global correlation coefficients (dict : name -> gcc).

grad
Gradient function of the cost function.

hesse(*self*, *ncall=None*, ***deprecated_kwargs*)
Run HESSE to compute parabolic errors.

HESSE estimates the covariance matrix by inverting the matrix of second derivatives (Hesse matrix) at the minimum. This covariance matrix is valid if your χ^2 or likelihood profile looks like a hyperparabola around the the minimum. This is usually the case, especially when you fit many observations (in the limit of infinite samples this is always the case). If you want to know how your parameters are correlated, you also need to use HESSE.

Also see `minos()`, which computes the uncertainties in a different way.

Arguments:

- **ncall**: integer or None, limit the number of calls made by MINOS. Default: None (uses an internal heuristic by C++ MINUIT).

Returns:

list of *Parameter Data Object*

init_params
List of current parameter data objects set to the initial fit state

is_clean_state(*self*)
Check if minuit is in a clean state, ie. no MIGRAD call

latex_initial_param(*self*)
Build `iminuit.latex.LatexTable` for initial parameter

latex_matrix(*self*)
Build `LatexFactory` object with correlation matrix.

latex_param(*self*)
build `iminuit.latex.LatexTable` for current parameter

matrix(*self*, *correlation=False*, *skip_fixed=True*)
Error or correlation matrix in tuple or tuples format.

merrors
MINOS errors.

migrad(*self*, *ncall=None*, *resume=True*, *precision=None*, ***deprecated_kwargs*)
Run MIGRAD.

MIGRAD is a robust minimisation algorithm which earned its reputation in 40+ years of almost exclusive usage in high-energy physics. How MIGRAD works is described in the MINUIT paper.

Arguments:

- **ncall**: integer or None, optional; (approximate) maximum number of call before MIGRAD will stop trying. Default: None (indicates to use MIGRAD's internal heuristic). Note: MIGRAD may slightly violate this limit, because it checks the condition only after a full iteration of the algorithm, which usually performs several function calls.
- **resume**: boolean indicating whether MIGRAD should resume from the previous minimiser attempt(True) or should start from the beginning(False). Default True.
- **precision**: override miniut own's internal precision.

Return:

Function Minimum Data Object, list of *Parameter Data Object*

minos (*self*, *var=None*, *sigma=1.*, *ncall=None*, ***deprecated_kwargs*)

Run MINOS to compute asymmetric confidence intervals.

MINOS uses the profile likelihood method to compute (asymmetric) confidence intervals. It scans the negative log-likelihood or (equivalently) the least-squares cost function around the minimum to construct an asymmetric confidence interval. This interval may be more reasonable when a parameter is close to one of its parameter limits. As a rule-of-thumb: when the confidence intervals computed with HESSE and MINOS differ strongly, the MINOS intervals are to be preferred. Otherwise, HESSE intervals are preferred.

Running MINOS is computationally expensive when there are many fit parameters. Effectively, it scans over *var* in small steps and runs MIGRAD to minimise the FCN with respect to all other free parameters at each point. This is requires many more FCN evaluations than running HESSE.

Arguments:

- **var**: optional variable name to compute the error for. If var is not given, MINOS is run for every variable.
- **sigma**: number of σ error. Default 1.0.
- **ncall**: integer or None, limit the number of calls made by MINOS. Default: None (uses an internal heuristic by C++ MINUIT).

Returns:

Dictionary of varname to *Minos Data Object*, containing all up to now computed errors, including the current request.

mncontour (*self*, *x*, *y*, *int numpoints=100*, *sigma=1.0*)

Two-dimensional MINOS contour scan.

This scans over **x** and **y** and minimises all other free parameters in each scan point. This works as if **x** and **y** are fixed, while the other parameters are minimised by MIGRAD.

This scan produces a statistical confidence region with the profile likelihood method. The contour line represents the values of **x** and **y** where the function passes the threshold that corresponds to *sigma* standard deviations (note that 1 standard deviations in two dimensions has a smaller coverage probability than 68 %).

The calculation is expensive since it has to run MIGRAD at various points.

Arguments:

- **x** string variable name of the first parameter
- **y** string variable name of the second parameter
- **numpoints** number of points on the line to find. Default 20.

- **sigma** number of sigma for the contour line. Default 1.0.

Returns:

x MINOS error struct, y MINOS error struct, contour line

contour line is a list of the form [[x1,y1]. . . [xn,yn]]

See also:

`contour()` `mnprofile()`

mnprofile(*self*, *vname*, *bins=30*, *bound=2*, *subtract_min=False*)

Calculate MINOS profile around the specified range.

Scans over **vname** and minimises FCN over the other parameters in each point.

Arguments:

- **vname** name of variable to scan
- **bins** number of scanning bins. Default 30.
- **bound** If bound is tuple, (left, right) scanning bound. If bound isa number, it specifies how many σ symmetrically from minimum (minimum+- bound* σ). Default 2
- **subtract_min** subtract_minimum off from return value. This makes it easy to label confidence interval. Default False.

Returns:

bins(center point), value, MIGRAD results

narg

Number of parameters.

ncalls

Number of FCN call of last MIGRAD / MINOS / HESSE run.

ncalls_total

Total number of calls to FCN (not just the last operation)

nfit

Number of fitted parameters (fixed parameters not counted).

ngrads

Number of Gradient calls of last MIGRAD / MINOS / HESSE run.

ngrads_total

Total number of calls to Gradient (not just the last operation)

np_covariance(*self*)

Covariance matrix in numpy array format.

Fixed parameters are included, the order follows `parameters`.

Returns:

`numpy.ndarray` of shape (N,N) (not a `numpy.matrix`).

np_errors(*self*)

Hesse parameter errors in numpy array format.

Fixed parameters are included, the order follows `parameters`.

Returns:

`numpy.ndarray` of shape (N,).

np_matrix (*self*, ***kwds*)
Covariance or correlation matrix in numpy array format.

Keyword arguments are forwarded to `matrix()`.

The name of this function was chosen to be analogous to `matrix()`, it returns the same information in a different format. For documentation on the arguments, please see `matrix()`.

Returns:

2D `numpy.ndarray` of shape (N,N) (not a `numpy.matrix`).

np_merrors (*self*)
MINOS parameter errors in numpy array format.

Fixed parameters are included, the order follows `parameters`.

The format of the produced array follows matplotlib conventions, as in `matplotlib.pyplot.errorbar`. The shape is (2, N) for N parameters. The first row represents the downward error as a positive offset from the center. Likewise, the second row represents the upward error as a positive offset from the center.

Returns:

`numpy.ndarray` of shape (2, N).

np_values (*self*)
Parameter values in numpy array format.

Fixed parameters are included, the order follows `parameters`.

Returns:

`numpy.ndarray` of shape (N,).

parameters
Parameter name tuple

params
List of current parameter data objects

pos2var
Map variable position to name

print_level
Current print level.

- 0: quiet
- 1: print minimal debug messages to terminal
- 2: print more debug messages to terminal
- 3: print even more debug messages to terminal

Note: Setting the level to 3 has a global side effect on all current instances of Minuit (this is an issue in C++ MINUIT2).

profile (*self*, *vname*, *bins=100*, *bound=2*, *subtract_min=False*, ***deprecated_kwargs*)
Calculate cost function profile around specify range.

Arguments:

- **vname** variable name to scan
- **bins** number of scanning bin. Default 100.

- **bound** If bound is tuple, (left, right) scanning bound. If bound is a number, it specifies how many σ symmetrically from minimum (minimum+- bound* σ). Default: 2.
- **subtract_min** subtract_minimum off from return value. This makes it easy to label confidence interval. Default False.

Returns:

bins(center point), value

See also:

`mnprofile()`

`strategy`
strategy: 'unsigned int' Current minimization strategy.

0: Fast. Does not check a user-provided gradient. Does not improve Hesse matrix at minimum. Extra call to `hesse()` after `migrad()` is always needed for good error estimates. If you pass a user-provided gradient to MINUIT, convergence is **faster**.

1: Default. Checks user-provided gradient against numerical gradient. Checks and usually improves Hesse matrix at minimum. Extra call to `hesse()` after `migrad()` is usually superfluous. If you pass a user-provided gradient to MINUIT, convergence is **slower**.

2: Careful. Like 1, but does extra checks of intermediate Hessian matrix during minimization. The effect in benchmarks is a somewhat improved accuracy at the cost of more function evaluations. A similar effect can be achieved by reducing the tolerance attr:*tol* for convergence at any strategy level.

`throw_nan`
Boolean. Whether to raise runtime error if function evaluate to nan.

`tol`
tol: 'double' Tolerance for convergence.

The main convergence criteria of MINUIT is `edm < edm_max`, where `edm_max` is calculated as `edm_max = 0.002 * tol * errordef` and EDM is the *estimated distance to minimum*, as described in the MINUIT paper.

`use_array_call`
Boolean. Whether to pass parameters as numpy array to cost function.

`valid`
Check if function minimum is valid.

`values`
values: iminuit._libiminuit.ValueView Parameter values in a dict-like object.

Use to read or write current parameter values based on the parameter index or the parameter name as a string. If you change a parameter value and run `migrad()`, the minimization will start from that value, similar for `hesse()` and `minos()`.

See also:

`errors`, `fixed`

`var2pos`
Map variable name to position

1.4.3 Cost functions

Standard cost functions to minimize.

class iminuit.cost.**BinnedNLL**(*n*, *xe*, *cdf*, *verbose=0*)
Binned negative log-likelihood.

Use this if only the shape of the fitted PDF is of interest and the data is binned.

Parameters

n: array-like Histogram counts.

xe: array-like Bin edge locations, must be len(n) + 1.

cdf: callable Cumulative density function of the form f(xe, par0, par1, . . . , parN), where *xe* is a bin edge and par0, . . . parN are model parameters.

verbose: int, optional Verbosity level

- 0: is no output (default)
- 1: print current args and negative log-likelihood value

class iminuit.cost.**Cost**(*args*, *verbose*, *errordef*)
Common base class for cost functions.

Attributes

mask [array-like or None] If not None, only values selected by the mask are considered. The mask acts on the first dimension of a value array, i.e. values[mask]. Default is None.

verbose [int] Verbosity level. Default is 0.

errordef [int] Error definition constant used by Minuit. For internal use.

class iminuit.cost.**ExtendedBinnedNLL**(*n*, *xe*, *scaled_cdf*, *verbose=0*)
Binned extended negative log-likelihood.

Use this if shape and normalization of the fitted PDF are of interest and the data is binned.

Parameters

n: array-like Histogram counts.

xe: array-like Bin edge locations, must be len(n) + 1.

scaled_cdf: callable Scaled Cumulative density function of the form f(xe, par0, par1, . . . , parN), where *xe* is a bin edge and par0, . . . parN are model parameters.

verbose: int, optional Verbosity level

- 0: is no output (default)
- 1: print current args and negative log-likelihood value

class iminuit.cost.**ExtendedUnbinnedNLL**(*data*, *scaled_pdf*, *verbose=0*)
Unbinned extended negative log-likelihood.

Use this if shape and normalization of the fitted PDF are of interest and the original unbinned data is available.

Parameters

data: array-like Sample of observations.

scaled_pdf: callable Scaled probability density function of the form f(data, par0, par1, . . . , parN), where *data* is the data sample and par0, . . . parN are model parameters. Must return a tuple (<integral over f in data range>, <f evaluated at data points>).

verbose: int, optional Verbosity level

- 0: is no output (default)

- 1: print current args and negative log-likelihood value

class iminuit.cost.**LeastSquares**(*x*, *y*, *yerror*, *model*, *loss='linear'*, *verbose=0*)

Least-squares cost function (aka chisquare function).

Use this if you have data of the form (x, y +/- yerror).

Parameters

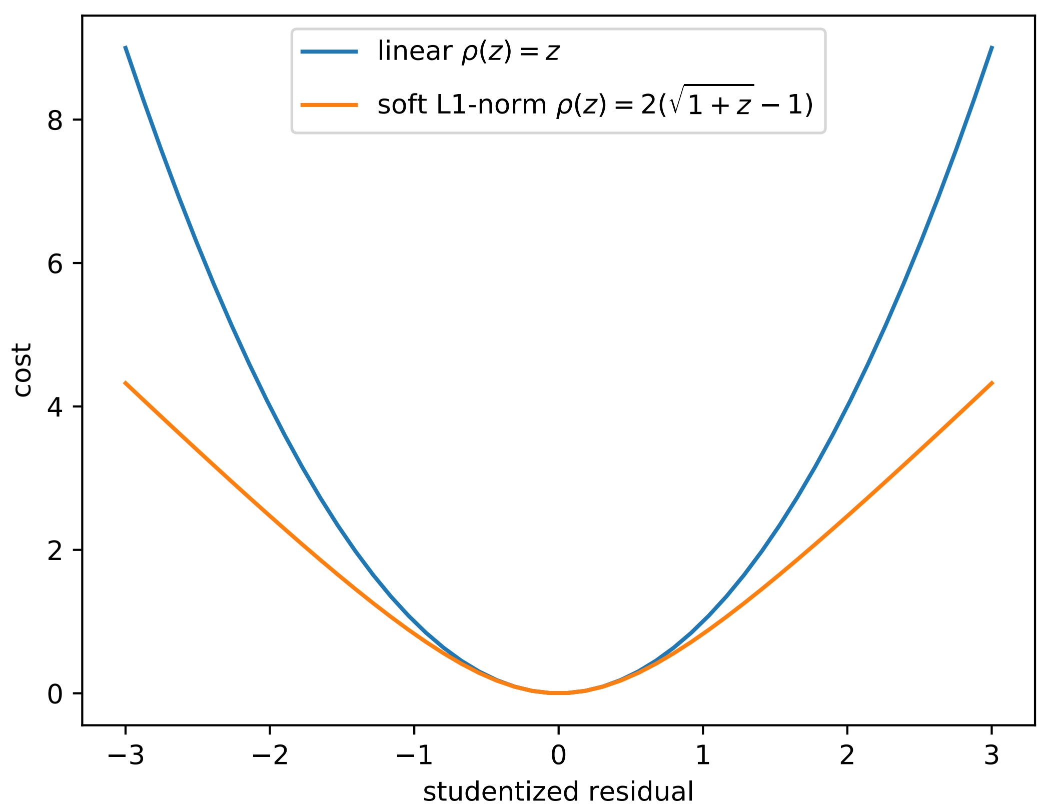
x: array-like Locations where the model is evaluated.

y: array-like Observed values. Must have the same length as *x*.

yerror: array-like or float Estimated uncertainty of observed values. Must have same shape as *y* or be a scalar, which is then broadcasted to same shape as *y*.

model: callable Function of the form f(x, par0, par1, . . . , parN) whose output is compared to observed values, where *x* is the location and par0, . . . parN are model parameters.

loss: str or callable, optional The loss function can be modified to make the fit robust against outliers, see scipy.optimize.least_squares for details. Only "linear" (default) and "soft_l1" are currently implemented, but users can pass any loss function as this argument. It should be a monotonic, twice differentiable function, which accepts the squared residual and returns a modified squared residual.



verbose: int, optional Verbosity level

- 0: is no output (default)
- 1: print current args and negative log-likelihood value

class `iminuit.cost.`**`UnbinnedNLL`**(*data*, *pdf*, *verbose=0*)

Unbinned negative log-likelihood.

Use this if only the shape of the fitted PDF is of interest and the original unbinned data is available.

Parameters

data: array-like Sample of observations.

pdf: callable Probability density function of the form f(data, par0, par1, . . . , parN), where *data* is the data sample and par0, . . . parN are model parameters.

verbose: int, optional Verbosity level

- 0: is no output (default)
- 1: print current args and negative log-likelihood value

1.4.4 minimize

The `iminuit.minimize()` function provides the same interface as `scipy.optimize.minimize()`. If you are familiar with the latter, this allows you to use Minuit with a quick start. Eventually, you still may want to learn the interface of the `iminuit.Minuit` class, as it provides more functionality if you are interested in parameter uncertainties.

`iminuit.`**`minimize`**(*fun*, *x0*, *args=()*, *method=None*, *jac=None*, *hess=None*, *hessp=None*, *bounds=None*, *constraints=None*, *tol=None*, *callback=None*, *options=None*)

An interface to MIGRAD using the `scipy.optimize.minimize` API.

For a general description of the arguments, see `scipy.optimize.minimize`.

The `method` argument is ignored. The optimisation is always done using MIGRAD.

The *options* argument can be used to pass special settings to Minuit. All are optional.

Options:

- *disp* (bool): Set to true to print convergence messages. Default: False.
- *tol* (float): Tolerance for convergence. Default: None.
- *maxfev* (int): Maximum allowed number of iterations. Default: None.
- *eps* (sequence): Initial step size to numerical compute derivative. Minuit automatically refines this in subsequent iterations and is very insensitive to the initial choice. Default: 1.

Returns: OptimizeResult (dict with attribute access)

- *x* (ndarray): Solution of optimization.
- *fun* (float): Value of objective function at minimum.
- *message* (str): Description of cause of termination.
- *hess_inv* (ndarray): Inverse of Hesse matrix at minimum (may not be exact).
- nfev (int): Number of function evaluations.
- njev (int): Number of jacobian evaluations.
- minuit (object): Minuit object internally used to do the minimization. Use this to extract more information about the parameter errors.

1.4.5 Utility Functions

The module *iminuit.util* provides the *describe()* function and various function to manipulate fit arguments. Most of these functions (apart from describe) are for internal use. You should not rely on them in your code. We list the ones that are for the public.

iminuit utility functions and classes.

class iminuit.util.**FMin**
Function minimum status object.

Create new instance of _FMin(fval, edm, tolerance, nfcn, ncalls, up, is_valid, has_valid_parameters, has_accurate_covar, has_posdef_covar, has_made_posdef_covar, hesse_failed, has_covariance, is_above_max_edm, has_reached_call_limit, has_parameters_at_limit)

items()

keys()

values()

exception iminuit.util.**HesseFailedWarning**
HESSE failed warning.

exception iminuit.util.**IMinuitWarning**
iminuit warning.

exception iminuit.util.**InitialParamWarning**
Initial parameter warning.

class iminuit.util.**MError**
Minos result object.

Create new instance of _MError(name, is_valid, lower, upper, lower_valid, upper_valid, at_lower_limit, at_upper_limit, at_lower_max_fcn, at_upper_max_fcn, lower_new_min, upper_new_min, nfcn, min)

items()

keys()

values()

class iminuit.util.**MErrors**
Dict from parameter name to Minos result object.

class iminuit.util.**Matrix**
Matrix data object (tuple of tuples).

Create new matrix.

class iminuit.util.**MigradResult**
Holds the Migrad result.

Create new instance of MigradResult(fmin, params)

class iminuit.util.**Param**
Data object for a single Parameter.

Create new instance of _Param(number, name, value, error, is_const, is_fixed, has_limits, has_lower_limit, has_upper_limit, lower_limit, upper_limit)

items()

keys()

values()

class `iminuit.util.`**`Params`**(*seq*, *merrors*)
List of parameter data objects.

Make Params from sequence of Param objects and MErrors object.

`iminuit.util.`**`describe`**(*f*, *verbose=False*)
Try to extract the function argument names.

`iminuit.util.`**`make_func_code`**(*params*)
Make a func_code object to fake function signature.

You can make a funccode from describable object by:

```
make_func_code(["x", "y"])
```

1.4.6 Data objects

iminuit uses various data objects as return values. This section lists them.

1.4.6.1 Function Minimum Data Object

Subclass of NamedTuple that stores information about the fit result. It is returned by `Minuit.get_fmin()` and `Minuit.migrad()`. It has the following attributes:

- *fval*: Value of the cost function at the minimum.
- *edm*: Estimated Distance to Minimum.
- *nfcn*: Number of function call in last Migrad call
- *up*: Equal to the value of *errordef* when Migrad ran.
- *is_valid*: Whether the function minimum is ok, defined as
  - has_valid_parameters
  - and not has_reached_call_limit
  - and not is_above_max_edm
- *has_valid_parameters*: Validity of parameters. This means:
  1. The parameters must have valid error(if it's not fixed). Valid error is not necessarily accurate.
  2. The parameters value must be valid
- *has_accurate_covariance*: Whether covariance matrix is accurate.
- *has_pos_def_covar*: Positive definiteness of covariance matrix. Must be true if the extremum is a minimum.
- *has_made_posdef_covar*: Whether Migrad has to force covariance matrix to be positive definite by adding a diagonal matrix (should not happen!).
- *hesse_failed*: Whether a call to Hesse after Migrad was successful.
- *has_covaraince*: Has Covariance.
- *is_above_max_edm*: Is estimated distance to minimum above its goal? This is the convergence criterion of Migrad, if it is violated, Migrad did not converge.
- *has_reached_call_limit*: Whether Migrad exceeded the allowed number of function calls.

1.4.6.2 Minos Data Object

Subclass of NamedTuple which stores information about the Minos result. It is returned by `Minuit.minos()` (as part of a dictionary from parameter name -> data object). You can get it also from `Minuit.merrors()`. It has the following attributes:

- *lower*: lower error value
- *upper*: upper error value
- *is_valid*: Validity of minos error value. This means *lower_valid* and *upper_valid*
- *lower_valid*: Validity of lower error
- *upper_valid*: Validity of upper error
- *at_lower_limit*: minos calculation hits the lower limit on parameters
- *at_upper_limit*: minos calculation hits the upper limit on parameters
- *lower_new_min*: found a new minimum while scanning cost function for lower error value
- *upper_new_min*: found a new minimum while scanning cost function for upper error value
- *nfn*: number of call to FCN in the last minos scan
- *min*: the value of the parameter at the minimum

1.4.6.3 Parameter Data Object

Subclass of NamedTuple which stores the fit parameter state. It is returned by `Minuit.hesse()` and as part of the `Minuit.migrad()` result. You can access the latest parameter state by calling `Minuit.get_param_states()`, and the initial state via `Minuit.get_initial_param_states()`. It has the following attrubutes:

- *number*: parameter number
- *name*: parameter name
- *value*: parameter value
- *error*: parameter parabolic error(like those from hesse)
- *is_fixed*: is the parameter fixed
- *is_const*: is the parameter a constant(We do not support const but you can alway use fixing parameter instead)
- *has_limits*: parameter has limits set
- *has_lower_limit*: parameter has lower limit set. We do not support one sided limit though.
- *has_upper_limit*: parameter has upper limit set.
- *lower_limit*: value of lower limit for this parameter
- *upper_limit*: value of upper limit for this parameter

1.4.7 Function Signature Extraction Ordering

1. Using `f.func_code.co_varnames`, `f.func_code.co_argcount` All functions that are defined like:

```python
def f(x, y):
    return (x - 2) ** 2 + (y - 3) ** 2
```

or:

```python
f = lambda x, y: (x - 2) ** 2 + (y - 3) ** 2
```

Have these two attributes.

2. Using `f.__call__.func_code.co_varnames`, `f.__call__.co_argcount`. Minuit knows how to skip the *self* parameter. This allow you to do things like encapsulate your data with in a fitting algorithm:

```python
class MyLeastSquares:
    def __init__(self, data_x, data_y, data_yerr):
        self.x = data_x
        self.y = data_y
        self.ye = data_yerr

    def __call__(self, a, b):
        result = 0.0
        for x, y, ye in zip(self.x, self.y, self.ye):
            y_predicted = a * x + b
            residual = (y - y_predicted) / ye
            result += residual ** 2
        return result
```

3. If all fails, Minuit will try to read the function signature from the docstring to get function signature.

This order is very similar to PyMinuit signature detection. Actually, it is a superset of PyMinuit signature detection. The difference is that it allows you to fake function signature by having a func_code attribute in the object. This allows you to make a generic functor of your custom cost function. This is explained in the **Advanced Tutorial** in the docs.

Note: If you are unsure what iminuit will parse your function signature, you can use `describe()` to check which argument names are detected.

1.5 Benchmark

We compare the performance of Minuit2 (the code that is wrapped by iminuit) with other minimizers available in Python. We compare Minuit with the strategy settings 0 to 2 with several algorithms implemented in the nlopt library and scipy.optimize. .

1.5.1 Setup

All algorithms minimize a dummy cost function

```python
def cost_function(par):
    z = (y - par)
    return sum(z ** 2 + 0.1 * z ** 4)
```

where y are samples from a normal distribution scaled by a factor of 5. The second term in the sum assures that the cost function is non-linear in the parameters and not too easy to minimize. No analytical gradient is provided, since this is the most common way how minimizers are used for small problems.

The cost function is minimized for a variable number of parameters from 1 to 100. The number of function calls is recorded and the largest absolute deviation of the solution from the truth. The fit is repeated 100 times for each configuration to reduce the scatter of the results, and the medians of these trails are computed.

The scipy algorithms are run with default settings. For nlopt, a stopping criterion must be selected. We stop when the absolute variation in the parameters is becomes less than 1e-3.

1.5.2 Results

The results are shown in the following three plots. The best algorithms require the fewest function calls to achieve the highest accuracy.

Shown in the first plot is the number of calls to the cost function divided by the number of parameters. Smaller is better. Note that the algorithms achieve varying levels of accuracy, therefore this plot alone cannot show which algorithm is best. Shown in the second plot is the accuracy of the solution when the minimizer is stopped. The stopping criteria vary from algorithm to algorithm.

The third plot combines both and shows accuracy vs. number of function calls per parameter for fits with 2, 10, and 100 parameters, as indicated by the marker size. Since higher accuracy can be achieved with more function evaluations, the most efficient algorithms follow diagonal lines from the top left to the bottom right in the lower left edge of the plot.

1.5.3 Discussion

The following discussion should be taken with a grain of salt, since experiments have shown that the results depend on the minimisation problem. Also not tested here is the robustness of these algorithms when the cost function is more complicated or not perfectly analytical.

- The Scipy methods Powell and CG are the most efficient algorithms on this problem. Both are more accurate than Minuit2 and CG uses much fewer function evaluations, especially in fits with many parameters. Powell uses a similar amount of function calls as Minuit2, but achieves accuracy at the level of 1e-12, while Minuit2 achieves 1e-3 to 1e-6.
- Minuit2 is average in terms of accuracy vs. efficiency. Strategy 0 is pretty efficient for fits with less than 10 parameters. The typical accuracy achieved in this problem is about 0.1 to 1 %. Experiments with other cost functions have shown that the accuracy strongly depends on how parabolic the function is near the minimum. Minuit2 seems to stop earlier when the function is not parabolic, achieving lower accuracy.
- An algorithm with a constant curve in the first plot has a computation time which scales linearly in the number of parameters. This is the case for the Powell and CG methods, but Minuit2 and others that compute an approximation to the Hesse matrix scale quadratically.
- The Nelder-Mead algorithm shows very bad performance with weird features. It should not be used. On the other hand, the SBPLX algorithm does fairly well although it is a variant of the same idea.

1.5.4 Conclusion

Minuit2 (and therefore iminuit) is a good allrounder. It is not outstanding in terms of convergence rate or accuracy, but not bad either. Using strategy 0 seem safe to use: it speeds up the convergence without reducing the accuracy of the result.

When an application requires minimising the same cost function with different data over and over so that a fast convergence rate is critical, it can be useful to try other minimisers to in addition to iminuit.

1.6 FAQ

1.6.1 Disclaimer: Read the excellent MINUIT2 user guide!

Many technical questions are nicely covered by the user guide of MINUIT2, `MINUIT User's guide`. We will frequently refer to it here.

1.6.2 I don't understand `Minuit.hesse()`, `Minuit.minos()`, *errordef*; what do these do?

1.6.3 How do I interpret the parameter errors that iminuit produces?

1.6.4 How do I obtain a high-quality error matrix?

1.6.5 What effect has setting limits on the parameter uncertainties?

The MINUIT2 user's guide explains all about it, see pages 6-8 and 38-40.

1.6.6 Can I have parameter limits that depend on each other (e.g. x^2 + y^2 < 3)?

MINUIT was only designed to handle box constrains, meaning that the limits on the parameters are independent of each other and constant during the minimization. If you want limits that depend on each other, you have three options (all with caveats), which are listed in increasing order of difficulty:

1) Change the variables so that the limits become independent, such as going from x,y to r, phi for a circle. This is not always possible or desirable, of course.

2) Use another minimizer to locate the minimum which supports complex boundaries. The nlopt library and scipy.optimize have such minimizers. Once the minimum is found and if it is not near the boundary, place box constraints around the minimum and run iminuit to get the uncertainties (make sure that the box constraints are not too tight around the minimum). Neither nlopt nor scipy can give you the uncertainties.

3) Artificially increase the negative log-likelihood in the forbidden region. This is not as easy as it sounds.

The third method done properly is known as the interior point or barrier method. A glance at the Wikipedia article shows that one has to either run a series of minimizations with iminuit (and find a clever way of knowing when to stop) or implement this properly at the level of a Newton step, which would require changes to the complex and convoluted internals of MINUIT2.

Warning: you cannot just add a large value to the likelihood when the parameter boundary is violated. MIGRAD expects the likelihood function to be differential everywhere, because it uses the gradient of the likelihood to go downhill. The derivative at a discrete step is infinity and zero in the forbidden region. MIGRAD does not like this at all.

1.6.7 What happens when I change the strategy?

See page 5 of the MINUIT2 user guide for a rough explanation what this does. Here is our detailed explanation, extracted from looking into the source code and doing benchmarks.

- strategy = 0 is the fastest and the number of function calls required to minimize scales linearly with the number of fitted parameters. The Hesse matrix is not computed during the minimization (only an approximation that is continously updated). When the number of fitted parameters > 10, you should prefer this strategy.
- strategy = 1 (default) is medium in speed. The number of function calls required scales quadratically with the number of fitted parameters. The different scales comes from the fact that the Hesse matrix is explicitly computed in a Newton step, if Minuit detects significant correlations between parameters.
- strategy = 2 has the same quadratic scaling as strategy 1 but is even slower. The Hesse matrix is always explicitly computed in each Newton step.

If you have a function that is everywhere analytical and the Hesse matrix varies not too much, strategy 0 should give you good results. Strategy 1 and 2 are better when these conditions are not given. The approximated Hesse matrix can become distorted in this case. The explicit computation of the Hesse matrix may help Minuit to recover after passing through such a region.

1.6.8 How do I get Hesse errors for sigma=2, 3, . . . ?

For a least-squares function, you use *errordef = sigma ** 2* and for a negative log-likelihood function you use *errordef = 0.5 * sigma ** 2*.

1.6.9 Why do extra messages appear in the terminal when I use *print_level=2* or larger?

A *print_level=2* or higher activates internal debug messages directly from C++ MINUIT, which we cannot capture and print nicely in a Jupyter notebook, sorry.

1.6.10 Is it possible to stop iminuit by setting a tolerance for changes in the minimized function or the parameters?

No. MINUIT only uses the Estimated Distance to Minimum (EDM) stopping criterion. It stops, if the difference between the actual function value at the estimated minimum location and the estimated function value at that location, based on MINUIT's internal parabolic approximation of the function, is small. This criterion depends strongly on the numerically computed first and second derivatives. Therefore, problems with MINUIT's convergence are often related to numerical issues when the first and second derivatives are computed.

More information about the EDM criterion can be found in the MINUIT paper.

1.6.11 I am not sure if minimisation or error estimation is reliable. What can I do?

Plot the likelihood profile around the minimum as explained in the basic tutorial. Check that the parameter estimate is at the bottom. If the minimum is not parabolic, you may want to use MINOS to estimate the uncertainty interval instead of HESSE.

1.6.12 Why do you shout at me with CAPITAL LETTERS so much?

Sorry, that's just how it was in the old FORTRAN days from which MINUIT originates. Computers were incredibly loud back then and everyone was shouting all the time!

Seriously though: People were using type writers. There was only a single font and no way to make letters italic or even bold, to set apart normal text from code. So people used CAPITAL LETTERS a lot for code in technical writing. For these historic reasons, MINUIT, MIGRAD, and HESSE are often written with capital letters here.

1.7 Changelog

1.7.1 1.4.9 (July, 18, 2020)

- Fixes an error introduced in 1.4.8 in *Minuit.minos* when *var* keyword is used and at least one parameter is fixed

1.7.2 1.4.8 (July, 17, 2020)

- Allow *ncall=None* in *Minuit.migrad*, *Minuit.hesse*, *Minuit.minos*
- Deprecated *maxcall* argument in *Minuit.minos*: use *ncall* instead

1.7.3 1.4.7 (July, 15, 2020)

- Fixed: iminuit.cost.LeastSquares failed when yerror is passed as list and mask is set

1.7.4 1.4.6 (July, 11, 2020)

- Update to Minuit2 C++ code to ROOT v6.23-01
- Fixed: iminuit now reports an invalid fit if a cost function has only a maximum, not a minimum (fixed upstream)
- Loss function in *iminuit.cost.LeastSquares* is now mutable
- Cost functions in *iminuit.cost* now support value masks
- Documentation improvements
- Fixed a deprecation warning in *Minuit.mnprofile*
- Binder now uses wheels instead of compiling current iminuit

1.7.5 1.4.5 (June, 25, 2020)

- Improved pretty printing for Minos Errors object *MErrors*
- Added docs for cost functions

1.7.6 1.4.4 (June, 24, 2020)

- Reverted: create MnHesse C++ instance on the stack instead on the heap
- Added least-squares cost function and tests

1.7.7 1.4.3 (June, 24, 2020)

1.7.7.1 Bug-fixes

- Fixed a bug where running *.hesse()* after *.migrad()*, which would ignore any changes to parameters (fixing/releasing them, changing their values, . . .)
- Fix number formatting issues in new quantities display
- Removed engineering suffixes again in favour of standard exponential notation

1.7.7.2 Deprecated

- keyword *forced_parameters* in *Minuit.__init__* is deprecated, use *name*

1.7.7.3 Features

- Added general purpose cost functions for binned and unbinned maximum-likelihood estimation (normal and so called extended)

1.7.7.4 Documentation

- Updated error computation tutorial
- New tutorial which demonstrates usage of cost functions

1.7.8 1.4.2 (June, 14, 2020)

Hot-fix release to correct an error in *Minuit.merrors* indexing.

1.7.8.1 Documentation

- New tutorial about using Numba to parallelize and jit-compile cost functions

1.7.9 1.4.1 (June, 13, 2020)

Mostly a bug-fix release, but also deprecates more old interface.

1.7.9.1 Bug-fixes

- Fixed a bug when displaying nans in rich displays

1.7.9.2 Deprecated

- *Minuit.minoserror_struct*: use *Minuit.merrors*, which is now an alias for the former
- *Minuit.merrors* now accepts indices and parameter names, like *Minuit.values*, etc.

1.7.9.3 Features

- Show engineering suffixes (1.23k, 123.4M, 0.8G, . . .) in rich diplays instead of "scientific format", e.g. 1.23e-3
- New initial step heuristic, replaces pedantic warning about missing step sizes
- New initial value heuristic for parameters with limits

1.7.9.4 Documentation

- New tutorial about using Numba to parallelize and jit-compile cost functions

1.7.10 1.4.0 (June, 12, 2020)

This release drops Python 2 support and modernizes the interface of iminuit's Minuit object to make it more pythonic. Outdated methods were deprecated and replaced with properties. Keywords in methods were made more consistent. The deprecated interface has been removed from the documentation, but is still there. Old code should still work (if not please file a bug report!).

1.7.10.1 Bug-fixes

- Fixed an exception in the rich display when results were NaN
- *Minuit.migrad_ok()* (now replaced by *Minuit.accurate*) now returns false if HESSE failed after MIGRAD and made the minimum invalid
- Running *Minuit.hesse()* now properly updates the function minimum
- Fixed incorrect *hess_inv* returned by *iminuit.minimize*
- Fixed duplicated printing of pedantic warning messages

1.7.10.2 Deprecated

- *Minuit.list_of_fixed_params()*, *Minuit.list_of_vary_params()*: use *Minuit.fixed*
- *Minuit.migrad_ok()*: use *Minuit.valid*
- *Minuit.matrix_accurate()*: use *Minuit.accurate*
- *Minuit.get_fmin()*: use *Minuit.fmin*
- *Minuit.get_param_states()*: use *Minuit.param*
- *Minuit.get_initial_param_states()*: use *Minuit.init_param*
- *Minuit.get_num_call_fcn()*: use *Minuit.ncalls_total*
- *Minuit.get_num_call_grad()*: use *Minuit.ngrads_total*
- *Minuit.print_param_states()*: use *print()* on *Minuit.param*
- *Minuit.print_initial_param_states()*: use *print()* on *Minuit.init_param*
- *Minuit.hesse(maxcall=. . .)* keyword: use *ncall=. . .* like in *Minuit.migrad()*
- *Minuit.edm*: use *Minuit.fmin.edm*

1.7.10.3 New features

- iminuit now uses the PDG formatting rule for quantities with errors
- slicing and basic broadcasting support for *Minuit.values*, *Minuit.errors*, *Minuit.fixed*, e.g. the following works: *m.fixed[:] = True*, *m.values[:2] = [1, 2]*
- *Minuit.migrad(ncall=0)* (the default) now uses MINUITs internal heuristic instead of a flat limit of 10000 calls
- *iminuit.minimize* now supports the *tol* parameter
- *Minuit* now supports *print_level=3*, which shows debug level information when MIGRAD runs
- Binder support and Binder badge for tutorial notebooks added by @matthewfeickert

1.7.10.4 Documentation

- New tutorials on error computation and on using automatic differentiation

1.7.11 1.3.10 (March, 31, 2020)

1.7.11.1 Bug-fixes

- sdist package was broken, this was fixed by @henryiii

1.7.11.2 Implementation

- Allow HESSE to be called without running MIGRAD first

1.7.11.3 Documentation

- Added tutorial to show how iminuit can compute parameter errors for other minimizers

1.7.11.4 Other

- @henryiii added a CI test to check the sdist package and the MANIFEST

1.7.12 1.3.9 (March, 31, 2020)

1.7.12.1 Bug-fixes

- *draw_contour* now accepts an integer for *bound* keyword as advertised in the docs
- fixed wrong EDM goal in iminuit reports, was off by factor 5 in some

1.7.12.2 Interface

- removed the undocumented keyword "args" in *(draw_)contour*, *(draw_)profile*
- removed misleading "show_sigma" keyword in *draw_contour*
- deprecated *Minuit.is_fixed*, replaced by *.fixed* attribute
- deprecated *Minuit.set_strategy*, assign to *Minuit.strategy* instead
- deprecated *Minuit.set_errordef*, assign to *Minuit.errordef* instead
- deprecated *Minuit.set_print_level*, assign to *Minuit.print_level* instead
- deprecated *Minuit.print_fmin*, *Minuit.print_matrix*, *Minuit.print_param*, *Minuit.print_initial_param*, *Minuit.print_all_minos*; use print() on the respective objects instead

1.7.12.3 Implementation

- improved style of draw_contour, draw more contour lines
- increased default resolution for curves produced by *(draw_)mncontour*, *(draw_)contour*
- switched from internal copy of Minuit2 to including Minuit2 repository from GooFit
- build improvements for windows/msvc
- updated Minuit2 code to ROOT-v6.15/01 (compiler with C++11 support is now required to build iminuit)
- @henryiii added support for building Python-3.8 wheels

1.7.12.4 Documentation

- added iminuit logo
- added benchmark section
- expanded FAQ section
- updated basic tutorial to show how parameter values can be fixed and released
- added tutorial about combining iminuit with automatic differentiation
- clarified the difference between *profile* and *mnprofile*, *contour* and *mncontour*
- fixed broken URLs for external documents
- many small documentation improvements to increase consistency

1.7.13 1.3.8 (October 17, 2019)

- fixed internal plotting when Minuit.from_array_func is used
- documentation updates
- reproduceable build

1.7.14 1.3.7 (June 12, 2019)

- fixed wheels support
- fixed failing tests on some platforms
- documentation updates

1.7.15 1.3.6 (May 19, 2019)

- fix for broken display of Jupyter notebooks on Github when iminuit output is shown
- replaced brittle and broken REPL diplay system with standard _repr_html_ and friends
- wheels support
- support for pypy-3.6
- documentation improvements
- new integration tests to detect breaking changes in the API

1.7.16 1.3.5 (May 16, 2019) [do not use]

- release with accidental breaking change in the API, use 1.3.6

1.7.17 1.3.4 (May 16, 2019) [do not use]

- incomplete release, use 1.3.6

1.7.18 1.3.3 (August 13, 2018)

- fix for broken table layout in print_param() and print_matrix()
- fix for missing error report when error is raised in user function
- fix of printout when ipython is used as a shell
- fix of slow convergence when analytical gradient is provided
- improved user guide with more detail information and improved structure

1.7.19 1.3.2 (August 5, 2018)

- allow fixing parameter by setting limits (x, x) with some value x
- better defaults for maxcall arguments of hesse() and minos()
- nicer output for print_matrix()
- bug-fix: covariance matrix reported by iminuit was broken when some parameters were fixed
- bug-fix: segfault when something in PythonCaller raised an exception

1.7.20 1.3.1 (July 10, 2018)

- fixed failing tests when only you installed iminuit with pip and don't have Cython installed

1.7.21 1.3 (July 5, 2018)

- iminuit 1.3 is a big release, there are many improvements. All users are encouraged to update.
- Python 2.7 as well as Python 3.5 or later are supported, on Linux, MacOS and Windows.
- Source packages are available for PyPI/pip and we maintain binary package for conda (see *Installation*).
- The bundled Minuit C++ library has been updated to the latest version (takend from ROOT 6.12.06).
- The documentation has been mostly re-written. To learn about *iminuit* and all the new features, read the *Tutorials*.
- Numpy is now a core dependency, required to compile iminuit.
- For Numpy users, a second callback function interface and a `Minuit.from_array_func` constructor was added, where the parameters are passed as an array.
- Results are now also available as Numpy arrays, e.g. `np_values`, `np_errors` and `np_covariance`.
- A wrapper function `iminuit.minimize` for the MIGRAD optimiser was added, that has the same arguments and return value format as `scipy.optimize.minimize`.
- Support for analytical gradients has been added, users can pass a `grad` callback function. This works, but for unknown reasons doesn't lead to performance improvements yet. If you can help debug or fix this issue, please comment here.
- Several issues have been fixed. A complete list of issues and pull requests that went into the 1.3 release is here.

1.7.22 Previously

- For iminuit releases before v1.3, we did not fill a change log.
- To summarise: the first iminuit release was v1.0 in Dec 2012. In 2013 there were several releases, and in Jan 2014 the v1.1.1 release was made. After that development was mostly inactive, except for the v1.2 release in Nov 2015.
- The release history is available here: https://pypi.org/project/iminuit/#history
- The git history and pull requests are here: https://github.com/scikit-hep/iminuit

1.8 Contribute

1.8.1 You can help

Please open issues and feature requests on Github. We respond quickly.

- Documentation. Tell us what's missing, what's incorrect or misleading.
- Tests. If you have an example that shows a bug or problem, please file an issue!
- Performance. If you are a C/cython/python hacker and see a way to make the code faster, let us know!

Direct contributions related to these items are welcome, too! If you want to contribute, please fork the project on Github, develop your change and then make a pull request. This allows us to review and discuss the change with you, which makes the integration very smooth.

1.8.2 Development setup

1.8.2.1 git

To hack on *iminuit*, start by cloning the repository from Github:

```
git clone --recursive https://github.com/scikit-hep/iminuit.git
cd iminuit
```

It is a good idea to develop your feature in a separate branch, so that your develop branch remains clean and can follow our develop branch.

```
git checkout -b my_cool_feature develop
```

Now you are in a feature branch, commit your edits here.

1.8.2.2 virtualenv

You have the source code now, but you also want to build and test. We recommend to make a dedicated build environment for *iminuit*, separate from the Python installation you use for other projects.

One way is to use Python virtual environments and *pip* to install the development packages listed in requirements-dev.txt

```
python -m venv py3 # folders called py* are ignored by git
source py3/bin/activate
pip install -r requirements-dev.txt
```

To delete the virtual environment just delete the folder *py3*.

1.8.2.3 conda

Another way is to use conda environments and environment-dev.yml to make the environment and install everything. You need to install a conda first, e.g. miniconda:

```
conda env create -f environment-dev.yml
conda activate iminuit-dev
```

If you ever need to update the environment, you can use:

```
conda env update -f environment-dev.yml
```

It's also easy to deactivate or delete it:

```
conda deactivate
conda env remove -n iminuit-dev
```

1.8.3 Development workflow

To simplify hacking on OSX and Linux, we have a Makefile with common commands. To see what commands are available, do:

```
make help
```

Build *iminuit* in-place:

```
python -m pip install -e .
```

Run the tests:

```
python -m pytest
```

Run the notebook tests:

```
python -m pytest tutorial
```

Run the tests and generate a coverage report:

```
make cov
<your-web-browser> htmlcov/index.htm
```

Build the docs:

```
make doc
<your-web-browser> doc/_build/html/index.html
```

Maintainers that prepare a release, should follow the instructions in doc/README.md

To check your *iminuit* version number and install location:

```
$ python
>>> import iminuit
>>> iminuit
# install location is printed
>>> iminuit.__version__
# version number is printed
```

Python Module Index

i

Index