## 1 In a nutshell

1. About
   1.1 What is iminuit?  
   1.2 Who is using iminuit?  
   1.3 Technical docs  
   1.4 Team  
      1.4.1 Maintainers  
      1.4.2 Contributors  

1.2 Installation
   1.2.1 Conda  
   1.2.2 pip  
   1.2.3 Installing from source  
      1.2.3.1 For users  
      1.2.3.2 For contributors/developers  
   1.2.4 Check installation  

1.3 Tutorials
   1.3.1 Basic tutorial  
   1.3.2 iminuit and automatic differentiation with JAX  
   1.3.3 iminuit and an external minimizer  
   1.3.4 Outdated Cython tutorials  

1.4 Reference
   1.4.1 Quick Summary  
   1.4.2 Minuit  
   1.4.3 minimize  
   1.4.4 Utility Functions  
   1.4.5 Data objects  
      1.4.5.1 Function Minimum Data Object  
      1.4.5.2 Minos Data Object  
      1.4.5.3 Parameter Data Object  
   1.4.6 Function Signature Extraction Ordering  

1.5 Benchmark
   1.5.1 Setup  
   1.5.2 Results  
   1.5.3 Discussion  
   1.5.4 Conclusion  

1.6 FAQ
1.6.1 Disclaimer: Read the excellent MINUIT2 user guide! .............................................. 25
1.6.2 I don’t understand Minuit.hesse(), Minuit.minos(), errordef; what do these do? .... 25
1.6.3 How do I interpret the parameter errors that iminuit produces? ............................ 25
1.6.4 How do I obtain a high-quality error matrix? ............................................................. 25
1.6.5 What effect has setting limits on the parameter uncertainties? ................................. 25
1.6.6 Can I have parameter limits that depend on each other (e.g. x^2 + y^2 < 3)? .............. 25
1.6.7 What happens when I change the strategy? ................................................................. 26
1.6.8 How do I get Hesse errors for sigma=2, 3, ...? ......................................................... 26
1.6.9 Why do extra messages appear in the terminal when I use print_level=2 or larger? ... 26
1.6.10 Is it possible to stop iminuit by setting a tolerance for changes in the minimized function or the parameters? ......................................................... 26
1.6.11 I am not sure if minimization or error estimation is reliable. What can I do? ......... 26
1.6.12 Why do you shout at me with CAPITAL LETTERS so much? .............................. 26

1.7 Changelog ....................................................................................................................... 27
1.7.1 1.3.10 (March, 31, 2020) ......................................................................................... 27
1.7.1.1 Bug-fixes .................................................................................................................. 27
1.7.1.2 Implementation ....................................................................................................... 27
1.7.1.3 Documentation ....................................................................................................... 27
1.7.1.4 Other ....................................................................................................................... 27
1.7.2 1.3.9 (March, 31, 2020) ......................................................................................... 27
1.7.2.1 Bug-fixes .................................................................................................................. 27
1.7.2.2 Interface .................................................................................................................. 27
1.7.2.3 Implementation ....................................................................................................... 28
1.7.2.4 Documentation ....................................................................................................... 28
1.7.3 1.3.8 (October 17, 2019) ......................................................................................... 28
1.7.4 1.3.7 (June 12, 2019) .............................................................................................. 28
1.7.5 1.3.6 (May 19, 2019) [do not use] .............................................................................. 28
1.7.6 1.3.5 (May 16, 2019) [do not use] .............................................................................. 29
1.7.7 1.3.4 (May 16, 2019) [do not use] .............................................................................. 29
1.7.8 1.3.3 (August 13, 2018) ......................................................................................... 29
1.7.9 1.3.2 (August 5, 2018) ............................................................................................ 29
1.7.10 1.3.1 (July 10, 2018) ............................................................................................. 29
1.7.11 1.3 (July 5, 2018) .................................................................................................... 29
1.7.12 Previously .................................................................................................................. 30

1.8 Contribute ....................................................................................................................... 30
1.8.1 You can help ............................................................................................................... 30
1.8.2 Development setup .................................................................................................... 30
1.8.2.1 git ........................................................................................................................ 30
1.8.2.2 virtualenv .............................................................................................................. 31
1.8.2.3 conda ................................................................................................................... 31
1.8.3 Development workflow .............................................................................................. 31

Python Module Index 33

Index 35
Jupyter-friendly Python interface to the C++ Minuit2 library.

- Supported CPython versions: 2.7, 3.5+
- Supported PyPy versions: 3.5, 3.6
- Supported platforms: Linux, OSX and Windows.

iminuit can be used for general function minimisation, but is most commonly used for least-squares and maximum-likelihood fits of statistical models to data, and to get model parameter error estimates from likelihood profile analysis.

- Code: https://github.com/scikit-hep/iminuit
- Documentation: http://iminuit.readthedocs.org/
- Gitter: https://gitter.im/Scikit-HEP/community
- Mailing list: https://groups.google.com/forum/#!forum/scikit-hep-forum
- PyPI: https://pypi.org/project/iminuit/
- License: MINUIT is LGPL and iminuit is MIT
- Citation: https://github.com/scikit-hep/iminuit/blob/master/CITATION
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 uncertainty 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 2.7 to 3.8, 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
- lautzat

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.
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. You can view them online too:

#### 1.3.1 Basic tutorial

Covers the basics of using iminuit.

#### 1.3.2 iminuit and automatic differentiation with JAX

How to compute function gradients for iminuit with jax and accelerate Python code with JAX’s JIT compiler. Spoiler: a **32x** speed up over plain numpy is achieved. Also discusses how to do a least-squares fit with data that has uncertainties in \(x\) and \(y\).
### 1.3.3 iminuit and an external minimizer

iminuit can run the HESSE algorithm on any point of the cost function. This means one can effectively combine iminuit with other minimizers: let the other minimizer find the minimum and only run iminuit to compute the parameter uncertainties. This does not work with MINOS, which requires that MIGRAD is run first.

### 1.3.4 Outdated Cython tutorials

The following two tutorials are outdated. Users who want to speed up their fits should try the just-in-time compilers provided by numba or jax in CPython or use iminuit in PyPy to accelerate the computation. This is much simpler than using Cython and may achieve even better performance.

- Advanced tutorial. Shows how to speed up the computation of the cost function with Cython.
- Hard Core Cython tutorial. Goes into more detail on how to use Cython.

### 1.4 Reference

#### 1.4.1 Quick Summary

These are the things you will use a lot:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>iminuit(fcn[, throw_nan, pedantic, ...])</td>
<td>Construct minuit object from given fcn</td>
</tr>
<tr>
<td>iminuit.from_array_func(cls, fcn, start)</td>
<td>Construct Minuit object from given fcn and start sequence.</td>
</tr>
<tr>
<td>iminuit.migrad(self, int ncall=10000[, ...])</td>
<td>Run MIGRAD.</td>
</tr>
<tr>
<td>iminuit.hesse(self, unsigned int maxcall=0)</td>
<td>Run HESSE to compute parabolic errors.</td>
</tr>
<tr>
<td>iminuit.minos(self[, var, sigma])</td>
<td>Run MINOS to compute asymmetric confidence intervals.</td>
</tr>
<tr>
<td>iminuit.values</td>
<td>values: iminuit._libiminuit.ValueView Parameter values in a dict-like object.</td>
</tr>
<tr>
<td>iminuit.fixed</td>
<td>fixed: iminuit._libiminuit.FixedView Access fixation state of a parameter in a dict-like object.</td>
</tr>
<tr>
<td>iminuit.errordef</td>
<td>FCN increment above the minimum that corresponds to one standard deviation.</td>
</tr>
<tr>
<td>iminuit.strategy</td>
<td>strategy: 'unsigned int’ Current minimization strategy.</td>
</tr>
<tr>
<td>iminuit.tol</td>
<td>tol: ‘double’ Tolerance for convergence.</td>
</tr>
<tr>
<td>iminuit.fval</td>
<td>Last evaluated FCN value</td>
</tr>
<tr>
<td>iminuit.nfit</td>
<td>Number of fitted parameters (fixed parameters not counted).</td>
</tr>
<tr>
<td>iminuit.mnprofile(self, vname[, bins, bound, ...])</td>
<td>Calculate MINOS profile around the specified range.</td>
</tr>
<tr>
<td>iminuit.draw_mnprofile(self, vname[, bins, ...])</td>
<td>Draw MINOS profile in the specified range.</td>
</tr>
</tbody>
</table>

---

1.4. Reference
1.4.2 Minuit

class iminuit.Minuit(fcn, throw_nan=False, pedantic=True, forced_parameters=None, print_level=0, errordef=None, grad=None, 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:

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

The parameters a, b, c must accept a real number.
iminuit automatically 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:

```python
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/stepsizes set.
- forced_parameters: tell Minuit not to do function signature detection and use this argument instead. (Default: None (automagically detect signature))
- print_level: set the print_level for this Minuit. 0 is quiet. 1 print out at the end of MI-GRAD/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 forced_parameters 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:

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

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:

```python
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.

  ```python
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:

```python
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**

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, **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:

\( x\_\text{bins}, y\_\text{bins}, \text{values} \)

\( \text{values}[y, x] \) <- this choice is so that you can pass it to through matplotlib contour()

See also:

\texttt{mncontour()} \texttt{mnprofile()}

covariance

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

See also:

\texttt{matrix()}

draw\_contour(self, x, y, bins=50, bound=2, **kwargs)

Convenience wrapper for drawing contours.

The arguments are the same as \texttt{contour()}. Please read the docs of \texttt{contour()} and \texttt{mncontour()} to understand the difference between the two.

See also:

\texttt{contour()} \texttt{draw\_mncontour()}

draw\_mncontour(self, x, y, nsigma=2, numpoints=100)

Draw MINOS contour.

Arguments:

- \texttt{x, y} parameter name
- \texttt{nsigma} number of sigma contours to draw
- \texttt{numpoints} number of points to calculate for each contour

Returns:

contour

See also:

\texttt{mncontour()}

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 \texttt{vname} fixed at various places within \texttt{bound}.

Arguments:

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

bins(center point), value, migrad results

```python
from iminuit import Minuit

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

m = Minuit(f, print_level=0, pedantic=False)
m.migrad()
m.draw_mnprofile("y")
```

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

**edm**

Current estimated distance to minimum.

See also:

`get_fmin()`

**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](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:

```python
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**


Like `values`, but instead of reading or writing the values, you read or write the errors (which double as step sizes for MINUIT’s 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:
```python
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`

**fval**

Last evaluated FCN value

**See also:**

`get_fmin()`

**gcc**

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

**get_fmin**(self)

Current function minimum data object

**get_initial_param_states**(self)

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

**get_num_call_fcn**(self)

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

**get_num_call_grad**(self)

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

**get_param_states**(self)

List of current parameter data objects

**grad**

Gradient function of the cost function.

**hesse**(self, unsigned int maxcall=0)

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 $\chi^2$ 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:**
iminuit Documentation, Release 1.3.10

- **maxcall**: limit the number of calls made by MINOS. Default: 0 (uses an internal heuristic by C++ MINUIT).

**Returns:**

- list of Parameter Data Object

**is_clean_state**(self)
Check if minuit is in a clean state, i.e. 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

**list_of_fixed_param**(self)
List of (initially) fixed parameters

**list_of_vary_param**(self)
List of (initially) float varying parameters

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

**matrix_accurate**(self)
Check if covariance (of the last MIGRAD run) is accurate

**merrors**
Deprecated. Use get_merrors() instead.

**merrors_struct**
MINOS error calculation information (dict name -> struct)

**migrad**(self, int ncall=10000, resume=True, int nsplit=1, precision=None)
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 (approximate) maximum number of call before MIGRAD will stop trying. Default: 10000. 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.

- **split**: split MIGRAD in to split runs. Max fcn call for each run is ncall/nsplit. MIGRAD stops when it found the function minimum to be valid or ncall is reached. This is useful for getting progress. However, you need to make sure that ncall/nsplit is large enough. Otherwise, MIGRAD will think that the minimum is invalid due to exceeding max call (ncall/nsplit). Default 1(no split).

- **precision**: override miniut own’s internal precision.

**Return:**

- Function Minimum Data Object, list of Parameter Data Object

**migrad_ok**(self)
Check if minimum is valid.
**minos** *(self, var=None, sigma=1., unsigned int maxcall=0)*

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 $\sigma$ error. Default 1.0.
- **maxcall**: limit the number of calls made by MINOS. Default: 0 (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 is a number, it specifies how many \( \sigma \) symmetrically from minimum (minimum+- bound* \( \sigma \)). 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.

**nfit**
Number of fitted parameters (fixed parameters not counted).

**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:
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

pos2var
Map variable position to name

print_level
Current print level.
• 0: quiet
• 1: info messages after minimization
• 2: debug messages during minimization

profile (self, vname, bins=100, bound=2, subtract_min=False, **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 \( \sigma \) symmetrically from minimum (minimum+- bound* \( \sigma \)). 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 \(\text{edm} < \text{edm\_max}\), where \(\text{edm\_max}\) is calculated as \(\text{edm\_max} = 0.002 \times \text{tol} \times \text{errordef}\) 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.

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 \texttt{migrad()}, the minimization will start from that value, similar for \texttt{hesse()} and \texttt{minos()}.

    See also:
    \texttt{errors, fixed}

var2pos
    Map variable name to position

1.4.3 minimize

The \texttt{iminuit.minimize()} function provides the same interface as \texttt{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 \texttt{iminuit.Minuit} class, as it provides more functionality if you are interested in parameter uncertainties.

\texttt{iminuit.minimize}(\texttt{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 \texttt{scipy.optimize.minimize} API.

For a general description of the arguments, see \texttt{scipy.optimize.minimize}.

The \texttt{method} argument is ignored. The optimisation is always done using MIGRAD.

The \texttt{options} argument can be used to pass special settings to Minuit. All are optional.

Options:

- \texttt{disp} (bool): Set to true to print convergence messages. Default: False.
- \texttt{maxfev} (int): Maximum allowed number of iterations. Default: 10000.
- \texttt{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)

- \texttt{x} (ndarray): Solution of optimization.
- \texttt{fun} (float): Value of objective function at minimum.
- \texttt{message} (str): Description of cause of termination.
- \texttt{hess\_inv} (ndarray): Inverse of Hesse matrix at minimum (may not be exact).
- \texttt{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.

1.4.4 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.

```python
class iminuit.util.Matrix
    Matrix data object (tuple of tuples).
    Create new matrix.

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)

class iminuit.util.Params(seq, merrors)
    List of parameter data objects.
    Make Params from sequence of Param objects and MErrors object.

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)

class iminuit.util.MErrors(**kwds)
    Dict from parameter name to Minos result object.
    Initialize an ordered dictionary. The signature is the same as regular dictionaries, but keyword arguments are not recommended because their insertion order is arbitrary.

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)

class iminuit.util.MigradResult
    Holds the Migrad result.
    Create new instance of MigradResult(fmin, params)

iminuit.util.make_func_code(params)
    Make a func_code object to fake function signature.
    You can make a funccode from describable object by:
    ```python
    make_func_code(["x", "y"])
    ```

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

Rename variable names in fitarg with rename function.

```python
# simple renaming
fitarg_rename({'x':1, 'limit_x':1, 'fix_x':1, 'error_x':1},
              lambda pname: 'y' if pname=='x' else pname)
#{'y':1, 'limit_y':1, 'fix_y':1, 'error_y':1},

# prefixing
fitarg_rename({'x':1, 'limit_x':1, 'fix_x':1, 'error_x':1},
              lambda pname: 'prefix_'+pname)
#{'prefix_x':1, 'limit_prefix_x':1, 'fix_prefix_x':1, 'error_prefix_x':1}
```

### 1.4.5 Data objects

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

#### 1.4.5.1 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`: FCN minimum value
- `edm`: Estimated Distance to Minimum
- `nfcn`: Number of function call in last minimizer call
- `up`: UP parameter. This determine how minimizer define $1\sigma$ error
- `is_valid`: Validity of function minimum. This is 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`: Boolean indicating whether covariance matrix is accurate.
- `has_pos_def_covar`: Positive definiteness of covariance
- `has_made_posdef_covar`: Whether minimizer has to force covariance matrix to be positive definite by adding diagonal matrix.
- `hesse_failed`: Successfulness of the hesse call after minimizer.
- `has_covariance`: Has Covariance.
- `is_above_max edm`: Is EDM above 0.0001*tolerance*up? The convergence of migrad is defined by EDM being below this number.
- `has_reached_call_limit`: Whether the last minimizer exceeds number of FCN calls it is allowed.
1.4.5.2 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.get_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

1.4.5.3 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 attributes:

- `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 always 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

1.4.6 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 continuously 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. MINUIT2 only uses the Estimated Distance to Minimum (EDM) stopping criterion, in which MINUIT2 compares its current local parabolic estimate of the minimized function with reality. It stops if the vertical distance of the estimate is small. More information about the EDM criterion can be found in the MINUIT paper.

1.6.11 I am not sure if minimization 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 writing. For these historic reasons, MINUIT, MIGRAD, and HESSE are often written with capital letters here.

1.7 Changelog

1.7.1 1.3.10 (March, 31, 2020)

1.7.1.1 Bug-fixes

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

1.7.1.2 Implementation

• Allow HESSE to be called without running MIGRAD first

1.7.1.3 Documentation

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

1.7.1.4 Other

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

1.7.2 1.3.9 (March, 31, 2020)

1.7.2.1 Bug-fixes

• \texttt{draw\_contour} now accepts an integer for \texttt{bound} keyword as advertised in the docs
• fixed wrong EDM goal in iminuit reports, was off by factor 5 in some

1.7.2.2 Interface

• removed the undocumented keyword “args” in \texttt{(draw\_)}\texttt{contour}, \texttt{(draw\_)}\texttt{profile}
• removed misleading “show\_sigma” keyword in \texttt{draw\_contour}
• deprecated \texttt{Minuit.is\_fixed}, replaced by \texttt{.fixed} attribute
• deprecated \texttt{Minuit.set\_strategy}, assign to \texttt{Minuit.strategy} instead
• deprecated \texttt{Minuit.set\_errordef}, assign to \texttt{Minuit.errordef} instead
• deprecated \texttt{Minuit.set\_print\_level}, assign to \texttt{Minuit.print\_level} instead
• deprecated \texttt{Minuit.print\_fmin}, \texttt{Minuit.print\_matrix}, \texttt{Minuit.print\_param}, \texttt{Minuit.print\_initial\_param}, \texttt{Minuit.print\_all\_minos}; use print() on the respective objects instead
1.7.2.3 Implementation

- improved style of draw_contour, draw more contour lines
- increased default resolution for curves produced by \texttt{(draw\_imncontour, (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

1.7.2.4 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 \texttt{profile} and \texttt{mnprofile, contour} and \texttt{mncontour}
- fixed broken URLs for external documents
- many small documentation improvements to increase consistency

1.7.3 1.3.8 (October 17, 2019)

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

1.7.4 1.3.7 (June 12, 2019)

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

1.7.5 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 display 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
1.7.6 1.3.5 (May 16, 2019) [do not use]

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

1.7.7 1.3.4 (May 16, 2019) [do not use]

- incomplete release, use 1.3.6

1.7.8 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

1.7.9 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

1.7.10 1.3.1 (July 10, 2018)

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

1.7.11 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 (taken 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.

## 1.7.12 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](https://pypi.org/project/iminuit/#history)

• The git history and pull requests are here: [https://github.com/scikit-hep/iminuit](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 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:

```
pip install virtualenv
virtualenv iminuit-dev
source iminuit-dev/bin/activate
pip install -r requirements-dev.txt
```

To delete the virtual environment just delete the folder iminuit-dev.

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, we have a Makefile with common commands. To see what commands are available, do:

```
make help
```

Build *iminuit* in-place:

```
make
```

Run the tests:

```
make test
```

Run the notebook tests:

```
make test-notebooks
```

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
```

If you change the public interface of iminuit, you should run the integration tests in addition to iminuit’s internal tests. The integration tests will download and install consumers of iminuit to check their tests. This allows us to see that iminuit does not break them.

```
make integration
```

Ideally, the integration tests should never fail because of iminuit, because breaking changes in the public interface should be detected by our own unit tests. If you find a problem during integration, you should add new tests to iminuit which will detect this problem in the future without relying on others!

Maintainers that prepare a release, should run:

```
make release
```

It generates the source distribution and prints a checklist for the release.

To check your iminuit version number and install location:

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

iminuit.util, 20
A
args (iminuit.Minuit attribute), 10

C
contour() (iminuit.Minuit method), 10
covariance (iminuit.Minuit attribute), 11

D
describe() (in module iminuit.util), 20
draw_contour() (iminuit.Minuit method), 11
draw_mncontour() (iminuit.Minuit method), 11
draw_mnprofile() (iminuit.Minuit method), 11
draw_profile() (iminuit.Minuit method), 12

E
edm (iminuit.Minuit attribute), 13
errordef (iminuit.Minuit attribute), 13
errors (iminuit.Minuit attribute), 13

F
fcn (iminuit.Minuit attribute), 13
fitarg (iminuit.Minuit attribute), 13
fitarg_rename() (in module iminuit.util), 20
fixed (iminuit.Minuit attribute), 14
FMIn (class in iminuit.util), 20
from_array_func() (iminuit.Minuit method), 9
fval (iminuit.Minuit attribute), 14

G
gcc (iminuit.Minuit attribute), 14
get_fmin() (iminuit.Minuit method), 20
get_initial_param_states() (iminuit.Minuit method), 14
get_num_call_fcn() (iminuit.Minuit method), 14
get_num_call_grad() (iminuit.Minuit method), 14
get_param_states() (iminuit.Minuit method), 14
grad (iminuit.Minuit attribute), 14

H
hesse() (iminuit.Minuit method), 14

I
iminuit.util (module), 20
is_clean_state() (iminuit.Minuit method), 15

L
latex_initial_param() (iminuit.Minuit method), 15
latex_matrix() (iminuit.Minuit method), 15
latex_param() (iminuit.Minuit method), 15
LEAST_SQUARES (iminuit.Minuit attribute), 10
LIKELIHOOD (iminuit.Minuit attribute), 10
list_of_fixed_param() (iminuit.Minuit method), 15
list_of_vary_param() (iminuit.Minuit method), 15

M
make_func_code() (in module iminuit.util), 20
Matrix (class in iminuit.util), 20
matrix() (iminuit.Minuit method), 15
matrix_accurate() (iminuit.Minuit method), 15
MError (class in iminuit.util), 20
MErrors (class in iminuit.util), 20
merrors (iminuit.Minuit attribute), 15
merrors_struct (iminuit.Minuit attribute), 15
migrad() (iminuit.Minuit method), 15
migrad_ok() (iminuit.Minuit method), 15
MigradResult (class in iminuit.util), 20
minimize() (in module iminuit), 19
minos() (iminuit.Minuit method), 15
Minuit (class in iminuit), 8
mncontour() (iminuit.Minuit method), 16
mnprofile() (iminuit.Minuit method), 16

N
narg (iminuit.Minuit attribute), 17
ncalls (*iminuit.Minuit attribute*), 17
nfit (*iminuit.Minuit attribute*), 17
np_covariance() (*iminuit.Minuit method*), 17
np_errors() (*iminuit.Minuit method*), 17
np_matrix() (*iminuit.Minuit method*), 17
np_merrors() (*iminuit.Minuit method*), 17
np_values() (*iminuit.Minuit method*), 18

P
Param (*class in iminuit.util*), 20
parameters (*iminuit.Minuit attribute*), 18
Params (*class in iminuit.util*), 20
pos2var (*iminuit.Minuit attribute*), 18
print_level (*iminuit.Minuit attribute*), 18
profile() (*iminuit.Minuit method*), 18

S
strategy (*iminuit.Minuit attribute*), 18

T
throw_nan (*iminuit.Minuit attribute*), 18
tol (*iminuit.Minuit attribute*), 19

U
use_array_call (*iminuit.Minuit attribute*), 19

V
values (*iminuit.Minuit attribute*), 19
var2pos (*iminuit.Minuit attribute*), 19