iminuit Documentation

Release 1.3.3

Piti Ongmongkolkul

Sep 12, 2018

Contents

MINUIT from Python - Fitting like a boss

iminuit is a Python interface to the *MINUIT* C++ package.

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.

- Code: https://github.com/iminuit/iminuit
- Documentation: http://iminuit.readthedocs.org/
- Mailing list: https://groups.google.com/forum/#!forum/iminuit
- PyPI: https://pypi.org/project/iminuit/
- License: MINUIT is LGPL and iminuit is MIT
- Citation: https://github.com/iminuit/iminuit/blob/master/CITATION

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 Installation

- iminuit works with Python 3.5 or later, as well as legacy Python 2.7.
- Linux, macOS and Windows are supported.

1.1.1 pip

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

```
$ pip install iminuit
```

We don't distribute binary wheels for *iminuit*, so *pip install* requires that you have a C++ compiler available.

1.1.2 Conda

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

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

The only required dependency for *iminuit* is *numpy*.

As explained in the documentation, using *ipython* and *jupyter* for interactive analysis, as well as *cython* for speed is advisable, so you might want to install those as well.

1.1.3 Check

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
>>> import iminuit
>>> iminuit.test()
```

1.2 Tutorials

All the tutorials are in tutorial directory. You can view them online too:

- Basic tutorial. Covers the basics of using iminuit.
- Advanced tutorial. The advanced tutorial shows you how to speed up the computation of the objective function with Cython and how to write wrapper classes that work with iminuits parameter name discovery.
- Hard Core Cython tutorial. If you need to do a huge likelihood fit that needs speed, this is for you. If you don't care, just use probfit. It's a fun read though I think.

1.3 About

1.3.1 What is iminuit?

iminuit is the fast interactive IPython-friendly minimizer 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 3.5 and Numpy

Whether you use the latest Python 3 or stick to classic Python 2, iminuit works for you. Numpy is supported: you can minimize functions that accept numpy arrays and get the fit results as numpy arrays. If you prefer to access parameters by name, that works as well!

Robust optimizer and error estimator

iminuit uses Minuit2 to minimize 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 minimizers, 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 minimization 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.

1.3.2 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.

- Wikipedia for Quasi Newton Method and DFP formula. The numerical algorithm behind MIGRAD.
- Variable Metric Method for Minimization by William Davidon, 1991
- A New Approach to Variable Metric Algorithm by R. Fletcher, 1970
- Original user guide for C++ Minuit2: `MINUIT User's guide` by Fred James, 2004
- Original Paper: MINUIT - A SYSTEM FOR FUNCTION MINIMIZATION AND ANALYSIS OF THE PARAMETER ERRORS AND CORRELATIONS by Fred James and Matts Roos, 1975.

1.3.3 Team

iminuit was created by **Piti Ongmongkolkul** (@piti118). It is a logical successor of pyminuit/pyminuit2, created by **Jim Pivarski** (@jpivarski).

1.3.3.1 Maintainers

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

1.3.3.2 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)

1.4 API

1.4.1 Quick Summary

These are the things you will use a lot:

Minuit(fcn[, throw_nan, pedantic, frontend, ...])	Construct minuit object from given *fcn*
Minuit.from_array_func(type cls, fcn, start)	Construct minuit object from given *fcn* and start sequence.
Minuit.migrad(self, int ncall=10000[, ...])	Run migrad.
Minuit.minos(self[, var, sigma])	Run minos for parameter *var*.
Minuit.values	values: iminuit._libiminuit.ValueView Parameter values (dict: name -> value)
Minuit.args	args: iminuit._libiminuit.ArgsView Parameter value tuple
Minuit.errors	errors: iminuit._libiminuit.ErrorView Parameter parabolic errors (dict: name -> error)
Minuit.get_merrors(self)	Dictionary of varname-> MinosError Struct
Minuit.fval	Last evaluated FCN value
Minuit.fitarg	fitarg: object Current Minuit state in form of a dict.
Minuit.mnprofile(self, vname[, bins, bound, ...])	Calculate minos profile around the specified range.
Minuit.draw_mnprofile(self, vname[, bins, ...])	Draw minos profile around the specified range.
Minuit.mncontour(self, x, y, int numpoints=20)	Minos contour scan.
Minuit.draw_mncontour(self, x, y[, nsigma, ...])	Draw minos contour.
minimize(fun, x0[, args, method, jac, hess, ...])	An interface to MIGRAD using the `scipy.optimize.minimize` API.
util.describe(f[, verbose])	Try to extract the function argument names.

1.4.2 Minuit

class iminuit.**Minuit**(*fcn*, *throw_nan=False*, *pedantic=True*, *frontend=None*, *forced_parameters=None*, *print_level=1*, *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.

1. 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*

2. 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.
- **frontend**: Minuit frontend. There are two builtin frontends.
  1. ConsoleFrontend is designed to print out to terminal.
  2. HtmlFrontend is designed to give a nice output in an IPython notebook session.

  By Default, Minuit switches to HtmlFrontend automatically if it is called in IPython session. It uses ConsoleFrontend otherwise.
- **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 migrad/hesse/minos.
- **errordef**: Optional. Amount of increase in fcn to be defined as 1 σ. If None is given, it will look at *fcn.default_errordef()*. If *fcn.default_errordef()* is not defined or not callable iminuit will give a warning and set errordef to 1. Default None(which means errordef=1 with a warning).
- **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:

Similar to PyMinuit. iminuit allows user to set initial value, initial stepsize/error, limits of parameters and whether parameter should be fixed or not by passing keyword arguments to Minuit.

This is best explained through examples:

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

- Initial value(varname):

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

- Initial step size/error(fix_varname):

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

- Limits (limit_varname=tuple):

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

- Fixing parameters:

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

Note: Tips: You can use python dictionary expansion to programmatically change the fitting arguments.

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

You can also obtain fit arguments from Minuit object to reuse it later too. *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)
```

---

migrad(*self*, *int ncall=10000*, *resume=True*, *int nsplit=1*, *precision=None*)

Run migrad.

Migrad is an age-tested(over 40 years old, no kidding), super robust and stable minimization algorithm. It even has wiki page. You can read how it does the magic at here.

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 minimizer 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 Struct, list of *Minuit Parameter Struct*

hesse (*self*, *unsigned int maxcall=0*)
Run HESSE.

HESSE estimates error matrix by the second derivative at the minimim. This error matrix is good if your χ^2 or likelihood profile is parabolic at the minimum. From my experience, most of the simple fits are.

`minos()` makes no parabolic assumption and scan the likelihood and give the correct error asymmetric error in all cases(Unless your likelihood profile is utterly discontinuous near the minimum). But, it is much more computationally expensive.

Arguments:

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

Returns:

list of *Minuit Parameter Struct*

minos (*self*, *var=None*, *sigma=1.*, *unsigned int maxcall=0*)
Run minos for parameter *var*.

If *var* is None it runs minos for all parameters

Arguments:

- **var**: optional variable name. Default None.(run minos for every variable)
- **sigma**: number of σ 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 Error Struct* if minos is requested for all parameters.

args
args: iminuit._libiminuit.ArgsView Parameter value tuple

values
values: iminuit._libiminuit.ValueView Parameter values (dict: name -> value)

errors
errors: iminuit._libiminuit.ErrorView Parameter parabolic errors (dict: name -> error)

fitarg
fitarg: object 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)
```

merrors
MINOS errors (dict).

Using this method is not recommended. It was added only for PyMinuit compatibility. Use *get_merrors()* instead, which returns a dictionary of name -> *Minos Error Struct* instead.

Dictionary entries for each parameter:

- (name,1.0) -> upper error
- (name,-1.0) -> lower error

fval
Last evaluated FCN value

See also:

get_fmin()

edm
Estimated distance to minimum.

See also:

get_fmin()

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

See also:

matrix()

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

errordef
errordef: 'double' Amount of change in FCN that defines 1 $sigma$ error.

Default value is 1.0. *errordef* should be 1.0 for χ^2 cost function and 0.5 for negative log likelihood function.

This parameter is sometimes called `UP` in the MINUIT docs.

tol
tol: 'double' Tolerance.

One of the MIGRAD convergence criteria is `edm < edm_max`, where `edm_max` is calculated as `edm_max = 0.0001 * tol * UP`.

contour (*self*, *x*, *y*, *bins=20*, *bound=2*, *args=None*, *subtract_min=False*)
2D contour scan.

return contour of migrad result obtained by fixing all others parameters except **x** and **y** which are let to varied.

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 value. This makes it easy to label confidence interval. 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()

Note: If *subtract_min=True*, the return value has the minimum subtracted off. The value on the contour can be interpreted *loosely* as $i^2 \times$ up where i is number of standard deviation away from the fitted value *WITHOUT* taking into account correlation with other parameters that's fixed.

---

draw_contour (*self*, *x*, *y*, *bins=20*, *bound=2*, *args=None*, *show_sigma=False*)
Convenience wrapper for drawing contours.

The argument is the same as *contour()*. If *show_sigma=True'(Default), the label on the contour lines will show how many :math:*sigma' away from the optimal value instead of raw value.

Note: Like *contour()*, the error shown on the plot is not strictly the 1 σ contour since the other parameters are fixed.

See also:

contour() mncontour()

draw_mncontour (*self*, *x*, *y*, *nsigma=2*, *numpoints=20*)
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

draw_mnprofile (*self*, *vname*, *bins=30*, *bound=2*, *subtract_min=False*, *band=True*, *text=True*)
Draw minos profile around 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.

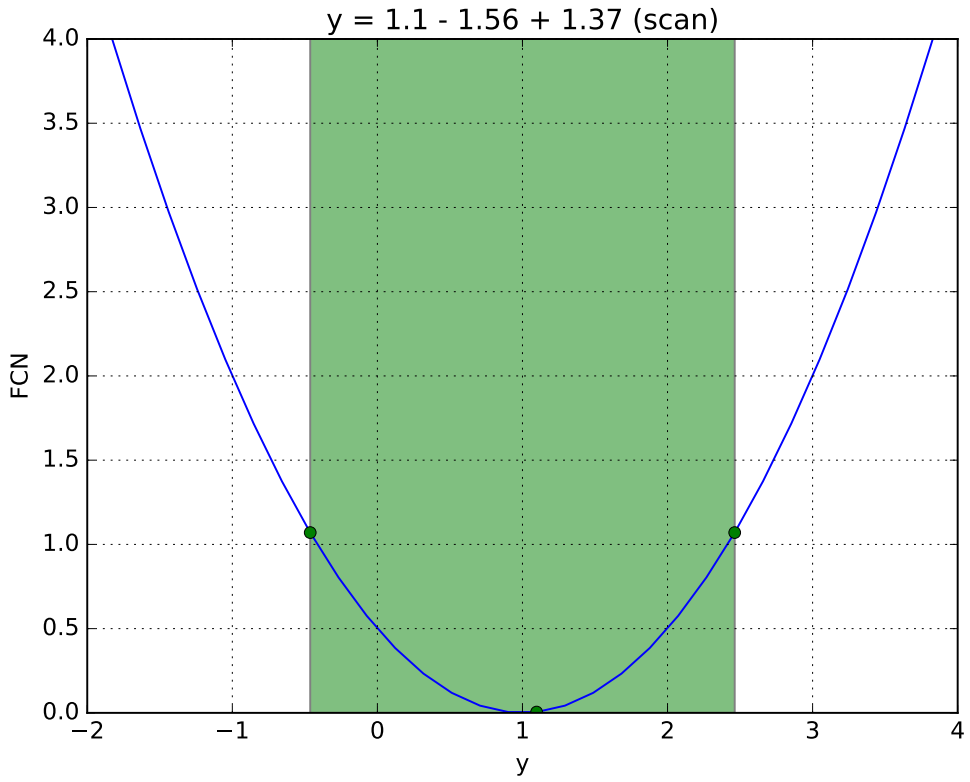
Returns:

bins(center point), value, migrad results

```
from iminuit import Minuit


def 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*, *args=None*, *subtract_min=False*, *band=True*, *text=True*)

A convenient wrapper for drawing profile using matplotlib.

Note: This is not a real minos profile. It's just a simple 1D scan. The number shown on the plot is taken from the green band. They are not minos error. To get a real minos profile call *mnprofile()* or *draw_mnprofile()*

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()

fixed
fixed: iminuit._libiminuit.FixedView Whether parameter is fixed (dict: name -> bool)

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)
```

get_fmin(*self*)
Current FunctionMinimum Struct

get_initial_param_states(*self*)
List of current MinuitParameter Struct for all parameters

get_merrors(*self*)
Dictionary of varname-> MinosError Struct

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 MinuitParameter Struct for all parameters

grad
Gradient function of the cost function

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

is_fixed(*self*, *vname*)
Check if variable *vname* is fixed.

Note that *Minuit.fixed* was added to fix and release parameters.

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) is accurate

merrors
MINOS errors (dict).

Using this method is not recommended. It was added only for PyMinuit compatibility. Use *get_merrors()* instead, which returns a dictionary of name -> *Minos Error Struct* instead.

Dictionary entries for each parameter:

- (name,1.0) -> upper error
- (name,-1.0) -> lower error

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

migrad_ok(*self*)
Check if minimum is valid.

mncontour(*self*, *x*, *y*, *int numpoints=20*, *sigma=1.0*)
Minos contour scan.

A proper n **sigma** contour scan. This is the line where the minimum of fcn with x,y is fixed at points on the line and letting the rest of variable varied is change by **sigma** * errordef^2 . The calculation is very very expensive since it has to run migrad at various points.

Note: See http://wwwasdoc.web.cern.ch/wwwasdoc/minuit/node7.html

---

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]]

mnprofile(*self*, *vname*, *bins=30*, *bound=2*, *subtract_min=False*)
Calculate minos profile around the specified range.

That is Migrad minimum results with **vname** fixed at various places within **bound**.

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 arguments

ncalls
Number of FCN call of last migrad / minos / hesse run.

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

pos2var
Map variable position to name

print_all_minos(*self*)
Print all minos errors (and its states)

print_fmin(*self*)
Print current function minimum state

print_initial_param(*self*, ***kwds*)
Print initial parameters

print_level
print_level: object Print level.

- 0: quiet
- 1: print stuff the end
- 2: 1+fit status during call

Yes I know the case is wrong but this is to keep it compatible with PyMinuit.

print_matrix(*self*)
Show correlation matrix.

print_param(*self*, ***kwds*)
Print current parameter state.

Extra keyword arguments will be passed to frontend.print_param.

profile(*self*, *vname*, *bins=100*, *bound=2*, *args=None*, *subtract_min=False*)
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()

set_errordef(*self*, *double errordef*)
Set error parameter 1 for χ^2 and 0.5 for log likelihood.

See page 37 of http://hep.fi.infn.it/minuit.pdf

set_print_level(*self*, *lvl*)
Set print level.

- 0 quiet
- 1 normal
- 2 paranoid
- 3 really paranoid

set_strategy(*self*, *value*)
Set strategy.

- 0 = fast
- 1 = default

- 2 = slow but accurate

set_up (*self*, *double errordef*)
Alias for *set_errordef()*

strategy
strategy: 'unsigned int' Strategy integer code.

- 0 fast
- 1 default
- 2 slow but accurate

use_array_call
Whether to pass parameters as numpy array to cost function

var2pos
Map variable name to position

1.4.3 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.
- *maxfev* (int): Maximum allowed number of iterations. Default: 10000.
- *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.

1.4.4 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.

class iminuit.util.**Struct**
A Struct is a Python dict with tab completion.

Example:

```
>>> s = Struct(a=42)
>>> s['a']
42
>>> s.a
42
```

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

See also:

Function Signature Extraction Ordering

iminuit.util.**fitarg_rename**(*fitarg*, *ren*)
Rename variable names in `fitarg` with rename function.

```
#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
figarg_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}
```

iminuit.util.**remove_var**(*b*, *exclude*)
Exclude variable in exclude list from b.

iminuit.util.**format_exception**(*etype*, *evalue*, *tb*)

1.4.5 Structs

iminuit uses various structs as return values. This section lists these structs and their fields.

1.4.5.1 Function Minimum Struct

Stores information about the fit result and returned by *Minuit.get_fmin()* and *Minuit.migrad()* Function Mimum Struct has the following attributes:

- *fval*: FCN minimum value
- *edm*: Estimated Distance to Minimum
- *nfcn*: Number of function call in last mimizier call

- *up*: UP parameter. This determine how minimizer define 1 σ 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_covaraince*: 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 allowd.

1.4.5.2 Minos Error Struct

Minos Error Struct is used in return value from `Minuit.minos()`. You can also call `Minuit.get_merrors()` to get accumulated dictionary all minos errors that has been calculated. It contains various minos status:

- *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 Minuit Parameter Struct

Minuit Parameter Struct is return value from `Minuit.hesse()` You can, however, access the latest parameter by calling `Minuit.get_param_states()`. Minuit Parameter Struct 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

1.4.6 Function Signature Extraction Ordering

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

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

or:

```
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:

```
class MyChi2:
    def __init__(self, x, y):
        self.x, self.y = (x,y)
    def f(self, x, m, c):
        return m*x + c
    def __call__(self,m,c):
        return sum([(self.f(x,m,c)-y)**2
                    for x,y in zip(self.x ,self.y)])
```

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 minuit will parse your function signature as , you can use `describe()` which returns tuple of argument names minuit will use as call signature.

1.5 Changelog

1.5.1 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

1.5.2 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

1.5.3 1.3.1 (July 10, 2018)

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

1.5.4 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.

1.5.5 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/iminuit/iminuit

1.6 Contribute

1.6.1 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.

1.6.2 Development setup

1.6.2.1 git

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

```
$ git clone https://github.com/iminuit/iminuit.git
$ cd iminuit
```

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

```
$ git checkout -b "my_cool_feature"
# now you in a feature branch, commit your edits here
```

1.6.2.2 conda

We recommend you make a dedicated environment for *iminuit* development, separate from the Python installation you use for other projects.

One way is to use conda environments and to use environment-dev.yml to make the environment and install everything:

```
$ 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
```

1.6.2.3 virualenv and pip

Another way is to use Python virtual environments and to *pip* install via requirements.txt

```
$ python -m venv iminuit-dev
$ source activate iminuit-dev
$ pip install -f requirements.txt
```

1.6.3 Development workflow

Hacking on *iminuit* usually means that you edit the Python or Cython files, and then run a *python setup.py* or *make* command to build the software or HTML documentation, or to run tests.

The most thing to remember is that we have a Makefile and the you can run this command to get help printed concerning the most common available *make* and *python setup.py* commands:

```
$ make help
```

Build Minuit and *iminuit* in-place:

```
$ make build
```

Run the tests:

```
$ make test
```

Run the notebook tests:

```
$ make test-notebooks
```

Run the tests with coverage report:

```
$ make coverage
$ open htmlcov/index.htm
```

Build the docs:

```
$ make doc
$ open doc/_build/html/index.html
```

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

R

S

T

U

V