

Is Your Optimizer Correct?

How to set up and read `check_optimizer.py` -- a verification harness for optimization code. No Python experience assumed.

Computational Labs | Michael Carter, Ph.D. | Free to use and share

This package contains four files and does one job: it tells you whether an optimization program is **correct**, as opposed to merely not crashing. Those are different things, and the difference is not visible from the output. A broken optimizer produces a smooth convergence curve, a small final number, and total confidence.

You do not need to be a Python programmer to use it. You need to be able to open a folder, run one file, and read a list of results. Everything else is explained below.

1. What is in this package

File	What it is	Do you edit it?
<code>check_optimizer.py</code>	The harness. Runs fifteen checks against any optimizer and prints a report.	No. Leave it alone.
<code>example_pso.py</code>	A worked example. A particle swarm with one ordinary bug, and the same code with the bug fixed, so you can watch the harness catch it.	Only if you want to experiment.
How to Use <code>check_optimizer.pdf</code>	This document. Setup, the checks explained, how to read results, and what the harness cannot tell you.	No.
README.txt	A one-page summary of the above, for anyone who opens the folder before the PDF.	No.

Everything is standard-library Python. Nothing to install, no internet connection required, no dependencies to go stale. One folder, four files.

2. Setting it up

Step 1 -- check you have Python

Open a terminal (Command Prompt or PowerShell on Windows, Terminal on Mac) and type:

```
python --version
```

You want version 3.6 or newer. If you see **Python 2.7**, try **python3 --version** instead and use **python3** everywhere below. If you see an error saying the command is not recognised, Python is not installed -- get it from python.org, and on Windows tick the box that says *Add Python to PATH* during

installation.

Step 2 -- put the files in one folder

This is the step people get wrong, and it produces the most common error message. All four files must sit in the **same folder**. Any folder will do -- your Desktop, your Documents, a new folder called *optimizer-check*. What matters is that `check_optimizer.py` is beside the file that uses it.

```
optimizer-check\  
  check_optimizer.py\  
  example_pso.py\  
  my_optimizer.py      <-- your own code goes here later
```

Why this matters: when Python reads **from `check_optimizer` import `check`**, it looks for a file called `check_optimizer.py`. It looks in the folder of the file being run, and in the folder your terminal is currently sitting in. If it is in neither, you get **ImportError: Could not import `check_optimizer`**, and the message will name the folder it searched.

Step 3 -- run the example

In a terminal, move into the folder and run the example. On Windows:

```
cd "C:\Users\yourname\Desktop\optimizer-check"  
python example_pso.py
```

On Mac or Linux:

```
cd ~/Desktop/optimizer-check  
python3 example_pso.py
```

It takes thirty seconds to a couple of minutes. It is running the optimizer roughly ninety times, because a single run of a random method tells you nothing.

If you use Spyder, IDLE, VS Code or PyCharm

These all work, with one thing to watch. **Run the whole file** -- in Spyder that is F5, in IDLE it is F5, in VS Code it is the Run button. Do **not** select a few lines and run the selection, and do not paste the code into the console. When you run a selection, the editor executes it in a console whose current folder may be somewhere else entirely, and the import fails.

If you must work in a console, tell it where to look first:

```
import os  
os.chdir(r"C:\Users\yourname\Desktop\optimizer-check")
```

3. Connecting your own optimizer

The harness needs to be able to call your optimizer. Your optimizer almost certainly does not have the exact shape it expects -- that is fine and expected. You write a short **adapter**: a small function that takes what the harness gives, calls your code however your code wants to be called, and hands back

the answer in a standard form.

This is usually five lines. You are not rewriting anything.

The shape the harness expects

```
def my_optimizer(objective, bounds, budget, seed):  
    ...  
    return best_x, best_f
```

Name	What the harness gives you	In plain terms
objective	A function. Call it with a list of numbers, it returns one number.	The thing you are trying to make small. <code>objective([1.0, 2.0])</code> might return 5.0.
bounds	A list of (low, high) pairs, one per dimension.	The box the answer must stay inside. <code>[(-10, 10), (-10, 10)]</code> is a square.
budget	A whole number.	The most times you are allowed to call <code>objective</code> . Stop when you reach it.
seed	A whole number.	Feed this to your random number generator so the run can be repeated exactly.

You return	What it must be
<code>best_x</code>	A list of ordinary numbers -- the best solution you found. One number per dimension.
<code>best_f</code>	One number -- the objective value AT that exact point.

best_f must be the score of best_x. Not the best score you saw at some earlier moment, not the score of a different particle. If those two come apart, every result your optimizer has ever produced is unreliable, and the harness checks this first.

A worked adapter

Suppose your own code looks like this -- different argument names, different order, returns an object, and works in a fixed number of generations rather than evaluations:

```
result = run_my_ga(fitness_fn=f, lower=-10, upper=10, dims=2,  
                  generations=100, pop_size=20)  
print(result.best_solution, result.best_fitness)
```

The adapter translates between the two:

```
def ga_adapter(objective, bounds, budget, seed):
    dims = len(bounds)          # how many dimensions
    lower = bounds[0][0]        # assumes one shared range
    upper = bounds[0][1]
    pop = 20
    gens = budget // pop        # respect the evaluation budget

    result = run_my_ga(fitness_fn=objective, lower=lower, upper=upper,
                       dims=dims, generations=gens, pop_size=pop,
                       random_seed=seed)

    best_x = list(result.best_solution)  # make it a plain list
    best_f = float(result.best_fitness)  # make it a plain number
    return best_x, best_f
```

Then, in the same file:

```
from check_optimizer import check
check(ga_adapter)
```

Three common adjustments

Your situation	What to put in the adapter
Your optimizer maximizes instead of minimizing.	Pass it a flipped objective and flip the answer back: flipped = lambda x: -objective(x) then return best_x, -best_f.
Your optimizer has no seed argument .	Seed the global generator yourself at the top of the adapter: import random; random.seed(seed) and if you use NumPy, np.random.seed(seed) as well.
Your optimizer returns NumPy arrays .	Convert on the way out: best_x = [float(v) for v in best_x]

4. The correctness checks, one by one

These twelve are pass or fail. They are not matters of taste, tuning or preference. A failure here means the program does something other than what it claims, and no amount of parameter adjustment will fix it.

1. Returns (best_x, best_f)

What it does. Calls your optimizer once and checks that it hands back exactly two things: a list of numbers and a single number.

If it fails. Your adapter is returning something else -- one value, three values, an object, a dictionary. Unpack it in the adapter and return the two required pieces.

2. Result is numeric and finite

What it does. Checks that no returned number is NaN (not-a-number) or infinite.

If it fails. Something divided by zero or overflowed. In optimization the usual culprits are dividing by the spread of fitness values when every candidate ties, or a step size that grows without limit until the numbers exceed what a computer can hold. Look for a division in your update rule and ask what happens when its denominator is zero.

3. `best_f` equals `objective(best_x)`

What it does. Takes the point you returned, evaluates the objective at it, and compares that with the score you reported. Runs on five different seeds, because this can pass by luck on one.

If it fails. This is the most serious result in the report. Your reported answer and your reported score describe two different points. Almost always it is best-so-far bookkeeping: you tracked the best score you ever saw, but returned the current position rather than the position where that score occurred. Every number this optimizer has ever given you is suspect, including in work already published or shipped.

4. Returned point is inside the bounds

What it does. Checks the answer lies within the box you declared. Five seeds again.

If it fails. Boundary handling is missing or wrong. Somewhere a position is updated and never brought back inside. This matters enormously in practice: if a coordinate is a physical thickness, a temperature or a budget, an out-of-bounds answer is not a slightly worse answer, it is an impossible one.

5. Never evaluates outside the bounds

What it does. Watches every call to the objective and records any made at a point outside the box.

If it warns. Your search wandered through regions that do not exist, even if the final answer happened to be legal. On a mathematical test function this is harmless. If your objective is a simulation, a spreadsheet or a physical model that quietly accepts nonsense inputs and returns a number anyway, it is not harmless at all -- you may be steering the search using scores from impossible configurations.

6. Respects the evaluation budget

What it does. Counts objective calls and compares against the budget given.

If it warns. Your method used more evaluations than it was allowed. Any comparison against another method is now unfair, because the two were not given the same resources. This is the most common way published comparisons mislead.

7. Reproducible: same seed gives the same result

What it does. Runs twice with seed 42 and compares.

If it fails. Some source of randomness is not seeded. You cannot reproduce a result, cannot compare two versions of your code honestly, and cannot debug -- because the bug moves each time you look for it. Check for a second random generator: Python's own **random** module and NumPy's are separate and must be seeded separately.

8. Seed actually changes the search

What it does. Runs with two different seeds and compares the *sequence of points evaluated*, not the final answers. Comparing final answers would be wrong -- a method that reliably finds the optimum returns the same value every time, and that is correct.

If it warns. The two runs walked an identical path, so the seed is not reaching the random number generator. The practical consequence: your thirty independent runs are one run repeated thirty times, and any average or error bar computed from them is meaningless.

9. Minimizes rather than maximizes

What it does. Runs your optimizer on a function whose *maximum* sits at the origin. A minimizer should walk away from it.

If it fails. Your objective sign is inverted somewhere -- you are maximizing when you believe you are minimizing, or vice versa. This produces a textbook convergence curve and an answer that is precisely, confidently wrong. It is the single most common silent error in applied optimization, and it is invisible unless you test for it.

10. Finds an optimum that is not at the centre of the box

What it does. Runs on a function whose minimum is at (2.5, -3.1) rather than at the origin, and compares against plain random search.

Why this exists. The standard test function $x^2 + y^2$ has its optimum exactly at the centre of a symmetric box like [-10, 10]. So does an optimizer that simply averages its population, or one with any pull toward the middle of the search space. Such a method scores beautifully on the standard test while being completely broken. Moving the optimum off centre exposes it immediately.

If it fails. Your method is drawn to the centre of the search space rather than to the optimum. Look for a step that averages positions without weighting by quality.

11. Handles 1, 5 and 10 dimensions and asymmetric bounds

What it does. Runs in one, five and ten dimensions, then with bounds of [(-2, 9), (3, 4)] -- different widths, neither centred on zero.

If it fails. Something is hard-coded. Common causes: a loop written for two dimensions, or an assumption that the box is symmetric about the origin -- for instance clamping with **abs(x) > limit** rather than checking the low and high edges separately. Real problems rarely have tidy symmetric bounds, so this failure will bite you.

12. Survives a flat objective

What it does. Runs on a function that returns 1.0 everywhere, so every candidate ties.

If it fails. Usually a division by zero, from normalising by the range or the standard deviation of fitness values without guarding the case where that range is zero. Flat regions occur in real objectives more often than people expect -- any plateau, any saturated constraint, any region where your simulation returns a fallback value.

5. The performance checks

These three are informational. A method can be slow, or weak on hard landscapes, and still be perfectly correct. Treat them as evidence for a decision, not as bugs.

Check	What it means
Sphere, 30 runs -- median, best, worst	The band, not a single number. One run of a stochastic method tells you nothing. If you report one run, you are reporting the luck of one seed.

Check	What it means
Beats random search on sphere	The floor. Uniform random sampling with the same budget is the cheapest possible method. Losing to it does not usually mean your algorithm is bad -- it means your objective function or your encoding is wrong, and no tuning will rescue it.
Beats random search on Rastrigin	Rastrigin is covered in local optima. Losing here while winning on sphere means your method cannot escape a local optimum -- typically a cooling schedule, inertia weight, or control parameter decaying too fast.
Every run reached the optimum	Shown instead of a variance warning when all thirty runs converged. The spread between 1e-15 and 1e-10 is arithmetic noise, not instability.

6. Reading the report

The report is in two halves for a reason. Read the top half first, and do not look at the bottom half until the top half is clean.

Marker	What to do
[ok]	Nothing. That check passed.
[warn]	Look at it and decide. Warnings are context-dependent -- the same warning can be irrelevant on a test function and serious on a real one.
[FAIL]	Stop. Fix this before you draw any conclusion from this optimizer's output. Results produced before the fix are not evidence of anything.
[info]	Context, not a verdict. Usually a measurement you should know.

The combination that matters most

Performance checks passing while correctness checks fail. This is not a rare pathology. It is the normal case, and it is why the harness exists.

In the worked example shipped with this package, the buggy version and the corrected version are statistically indistinguishable on every performance measure -- both converge to about 1e-12, both beat random search by ten orders of magnitude. The only thing separating them is that one of them returns answers from outside the search space you defined. A convergence curve cannot tell you that. Nothing you normally look at can.

7. When something goes wrong

What you see	What it means and what to do
ImportError: Could not import check_optimizer	check_optimizer.py is not in the folder Python searched. The message names the folder it looked in. Move the file there, or run the whole file (F5) instead of a selection.
NameError: name '__file__' is not defined	You pasted the code into a console or ran a selection rather than the file. Run the whole file instead, or set the folder first with os.chdir(...).

What you see	What it means and what to do
python is not recognised as a command	Python is not on your PATH. Reinstall from python.org with <i>Add Python to PATH</i> ticked, or try py instead of python on Windows.
TypeError about arguments	Your adapter does not accept the four arguments in the right order: objective, bounds, budget, seed. Copy the signature exactly.
It runs for a very long time	The harness calls your optimizer around ninety times. Lower the budget: <code>check(my_opt, budget=500)</code> . Correctness checks work fine on a small budget.
Everything fails at once	Almost always the adapter, not the optimizer. Print what your adapter returns before handing it over, and confirm it is a plain list and a plain number.

8. What this cannot tell you

A verification tool that overstates what it proves is worse than none, so here are its limits, plainly.

- **It cannot tell you your objective function is right.** The harness tests the machinery of the search, not your model of the problem. If your objective computes the wrong quantity, every check here can pass while the answer is useless. This is the most common real-world failure and it is outside the reach of any tool.
- **It cannot tell you the answer is good enough.** Whether $1e-3$ is a triumph or a disaster depends on your problem, not on the arithmetic.
- **It cannot prove there are no bugs.** Passing means twelve specific, common failures are absent. It does not mean the code is correct in general. No finite test suite proves that.
- **It cannot judge your constraints.** It checks box bounds only. Coupled constraints, integer requirements and feasibility rules are yours to verify.
- **It cannot tell you the method suits the problem.** A correct ant colony implementation applied to a continuous problem is correct and inappropriate.

The check no tool can do for you. Take the smallest possible instance of your problem -- three data points, two dimensions -- and work one iteration by hand. Compare it with what the code produced, digit by digit. It takes about twenty minutes, and it is the only way to confirm that the arithmetic in your head and the arithmetic in the machine are the same arithmetic. Everything in this package is a substitute for that, and a poor one.

You cannot debug what you cannot compute by hand.