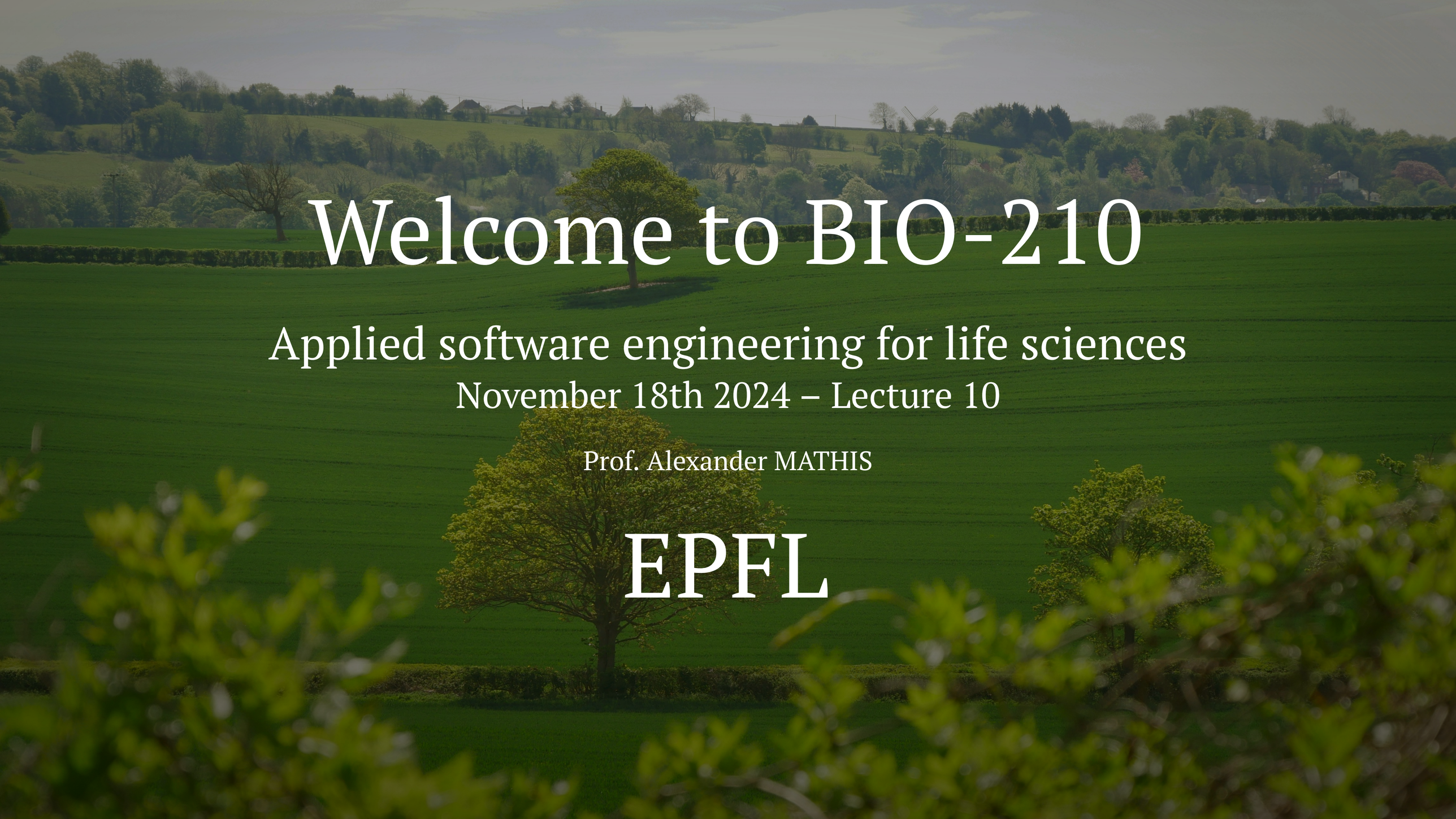


Welcome to BIO-210

Applied software engineering for life sciences

November 18th 2024 – Lecture 10

Prof. Alexander MATHIS

EPFL

	Date	Topic	Software version	Software releases	Grading / Feedback
0	09/09/2024	Python introduction I			
1	16/09/2024	Public holiday			
2	23/09/2024	Python introduction II			
3	30/09/2024	Git and GitHub (+installation VS Code)			
4	07/10/2024	Project introduction	v1		
5	14/10/2024	Functionify	v2	v1	
6	21/10/2024	EPFL fall break			
7	28/10/2024	Visualization and documentation	v3	v2	code review (API)
8	04/11/2024	Unit-tests, functional tests	v4	v3	
9	11/11/2024	Code refactoring	v5	v4	graded (tests)
10	18/11/2024	Profiling and code optimization	v6	v5	code review
11	25/11/2024	Object oriented programming	v7	v6	graded (speed)
12	02/12/2024	Model analysis and project report	v8	v7	code review (OO)
13	09/12/2024	Work on project			
14	16/12/2024	Wrap up		v8	graded (project)

Status of your project

You have

- a readme with information on how to install the code and what it can do
- a module for relevant functions and utilities (e.g., plotting)
- tests to assess the correctness of the code (also for refactoring)
- we also gave you some test targets, so your functions should be correct
- scripts to reproduce various results
- …

You may wonder …

- what is the slowest part of my code?
- how often is a particular piece of code run?
- how long does it take?
- why does my code take so long …

Profiling

Profiling addresses these questions and helps you optimize your code - so that you can make it faster.

To get the biggest speed improvement per coding hour, you should *focus on the slowest part of your program* (in a typical use case).

Note the specific use-case, can also be user-specific and one can optimize for different target groups.

We'll consider the following tools:

- cProfile
- timeit
- line_profiler

Donald Knuth’s advice

From the Art of Programming:

“The real problem is that programmers have spent far too much time worrying about efficiency in the wrong places and at the wrong times; premature optimization is the root of all evil (or at least most of it) in programming.”

Fastest code prize!

- we will assess speed of key functions (see problem set)

Note on the problem set

- you should use the different profiling tools I describe today
- you should release your fastest code as v6
- if your version has any special dependencies, make that clear in the readme
- if you develop variants that ultimately aren't the fastest you can document it in Issues (for yourself & the team); we won't grade this, but it's a good exercise to learn how to keep track of different variants.
- Typically, you would leave such variants as branches (if they are interesting enough)

P.S.: We will run your code on a machine with CPUs, not GPUs. So optimize for this setting. Note: the choice of hardware can make different algorithms more efficient!

Built-in profilers: cProfile (and profile)

cProfile is recommended, as it is the C extension with less overhead than profile

- records total run time
- records the time taken by each function
- ...

–> *This allows you to find which parts need optimization*

- recorded data can be exported, looked at with the pstats module and visualized with the snakeviz module.

cProfile syntax

```
1   import cProfile                                  # import the module
2   cProfile.run(statement, filename=None)
```

- As `statement` you can pass python code or a function (as a string).
- Output can be saved in filename (if passed)

A demo program

What does this program do and how should the profiling output look like?

```
1   import time
2
3   def shortbreak():
4       time.sleep(0.05)
5
6   def longbreak():
7       time.sleep(1)
8
9   def program():
10      print("Starting!")
11
12      for l in range(3):
13          shortbreak()
14
15      print("Now to the long break...")
16      longbreak()
17
18      print("Done!")
```

Adding cProfiler (when executed as script)

```
# Script: cProfiler_demo.py
import time

def shortbreak():
    time.sleep(0.05)

def longbreak():
    time.sleep(1)

def program():
    print("Starting!")
    for l in range(3):
        shortbreak()
    print("Now to the long break...")
    longbreak()
    print("Done!")

if __name__=='__main__':
    import cProfile
    profiler = cProfile.Profile()        # Initializing a profiler instance
    profiler.runcall(program)            # Run it on program()
    profiler.print_stats()               # Print runtime statistics
```

Running the script

```
alex@mac% python3 cProfiler_demo.py
Starting!
Now to the long break...
Done!
         13 function calls in 1.161 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        3    0.000    0.000    0.160    0.053 cProfiler_demo.py:3(shortbreak)
        1    0.000    0.000    1.000    1.000 cProfiler_demo.py:6(longbreak)
        1    0.000    0.000    1.161    1.161 cProfiler_demo.py:9(program)
        3    0.000    0.000    0.000    0.000 {built-in method builtins.print}
        4    1.161    0.290    1.161    0.290 {built-in method time.sleep}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' obj
```

What is summarized in the output?

Line #1: summary (# function calls and time)

Line #2: Ordered by: standard name (i.e. ordered by function name)

In the table:

- `ncalls`: number of calls
- `tottime`: total time spent in the given function (and excluding time spend in calls to sub-functions)
- `percall`: ratio tottime/ncalls
- `cumtime`: cumulative time spent in this function and all subfunctions (from invocation utill exit). This figure is accurate even for recursive functions.
- `percall`: cumtime/primitive calls [see later!]
- `filename:lineno`: provides the respective data of each function

You can also store/load the stats

```
from cProfiler_demo import program    # importing the program from prior
import cProfile

# You can also save stats, here to the file summary.stats
cProfile.run('program()','summary.stats')

import pstats
stats = pstats.Stats('summary.stats')   # Load the stats
stats.print_stats()                     # Print them
```

Let's store this as `cProfiler_visualization.py`

Output of this program

```
alex@mac% python3 cProfiler_visualization.py
Starting!
Now to the long break...
Done!
Thu Nov 14 17:21:34 2024    summary.stats

         15 function calls in 1.170 seconds

   Random listing order was used

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    1.170    1.170 {built-in method builtins.exec}
        3    0.000    0.000    0.000    0.000 {built-in method builtins.print}
        4    1.169    0.292    1.169    0.292 {built-in method time.sleep}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' obj
        1    0.000    0.000    1.004    1.004 /Users/alex/Code/Teaching/profiling/cProfil
        1    0.000    0.000    1.170    1.170 /Users/alex/Code/Teaching/profiling/cProfil
        3    0.000    0.000    0.165    0.055 /Users/alex/Code/Teaching/profiling/cProfil
        1    0.000    0.000    1.170    1.170 <string>:1(<module>)
```

You can sort the output

```
stats.sort_stats('calls')     # sort it by calls (see link in title for more options)
stats.print_stats()
```

gives:

```
Thu Nov 14 18:31:15 2024    summary.stats

         15 function calls in 1.170 seconds

   Ordered by: call count

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        4    1.170    0.293    1.170    0.293 {built-in method time.sleep}
        3    0.000    0.000    0.000    0.000 {built-in method builtins.print}
        3    0.000    0.000    0.165    0.055 /Users/alex/Code/Teaching/profiling/cProfil
        1    0.000    0.000    1.170    1.170 {built-in method builtins.exec}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' obj
        1    0.000    0.000    1.005    1.005 /Users/alex/Code/Teaching/profiling/cProfil
        1    0.000    0.000    1.170    1.170 /Users/alex/Code/Teaching/profiling/cProfil
        1    0.000    0.000    1.170    1.170 <string>:1(<module>)
```

Visualization of the the cProfiler output

Snakeviz provides visualizations based on profiling output.

Snakeviz is a Python package and thus needs to be installed --> `pip install snakeviz`

The visualization is interactive, and available in the browser!

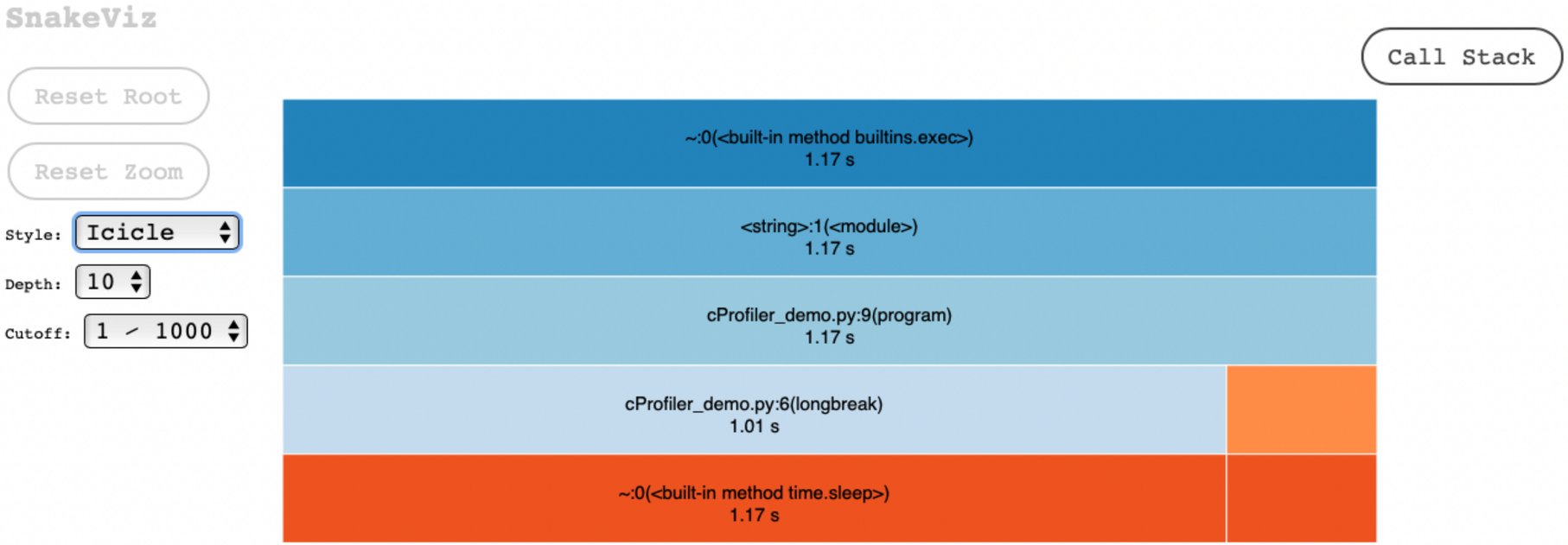
Can be used from the command line for profiling output (`summary.stats`).

```
snakeviz summary.stats
```

Can also be used in IPython directly, see docs!

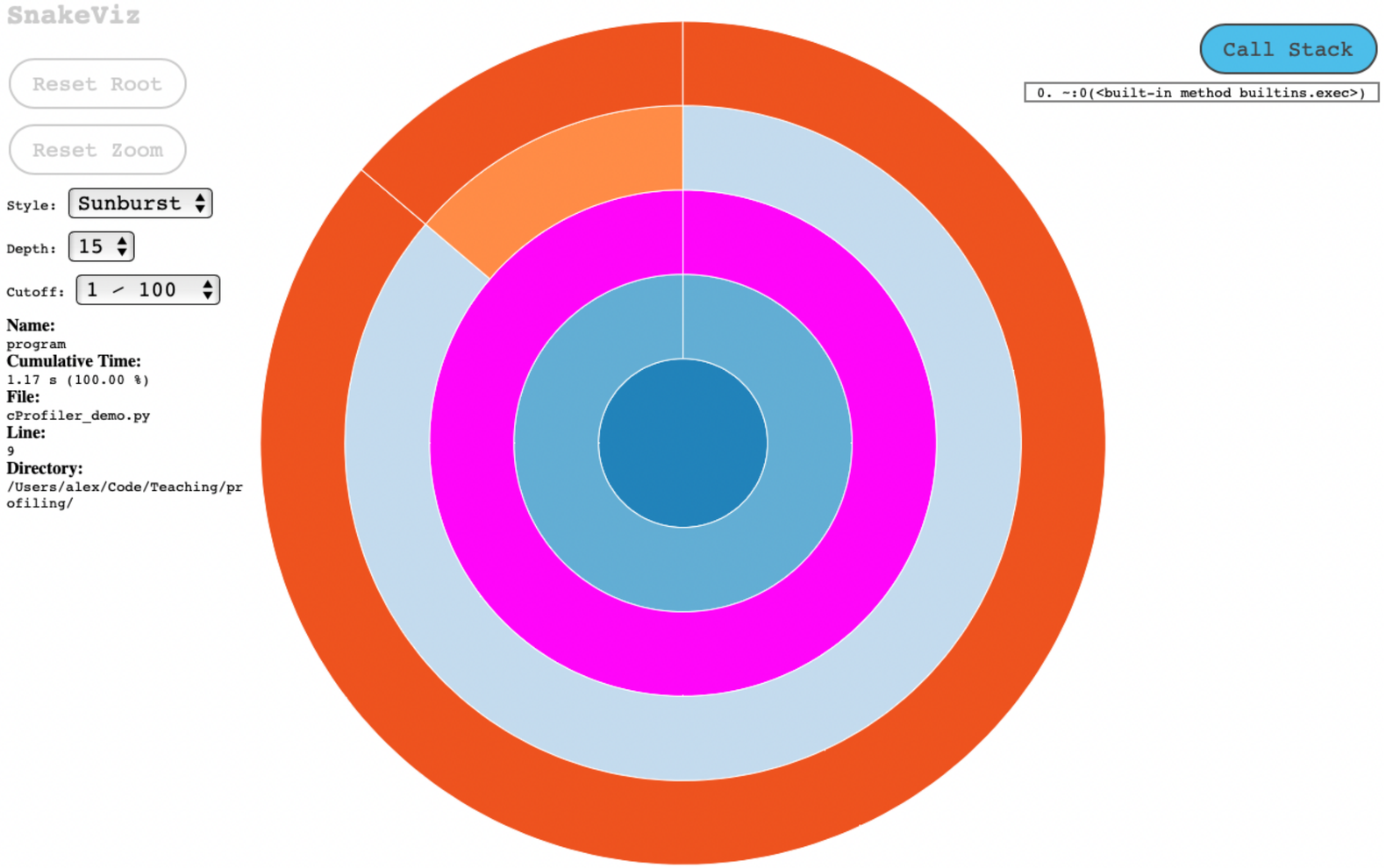
Visualization type 1 (icicle)

```
snakeviz summary.stats
```



Visualization type 2 (sunburst)

```
snakeviz summary.stats
```



Alternative use of cProfile

Running cProfile on a script (in the command line).

Here sorted by total-time, and stored in summary.txt

```
python -m cProfile -s tottime script2profile.py > summary.txt
```

Reminder: Fibonacci

A possible solution for Fibonacci:

```
number_of_fibonaccielements = 50
# Initialization
x0 = 0
x1 = 1

fibo_list = [x0, x1]
i = 2

while i < number_of_fibonaccielements:
    # Updating
    next_x = x0 + x1           # computing sum
    fibo_list.append(next_x)  # data collection
    x0 = x1                    # updating 1
    x1 = next_x                # updating 2
    i += 1

print(fibo_list)  # result
```

Note: $F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}$ for $n > 1$

Python allows recursive functions

$F_0 = 0, F_1 = 1,$

$F_n = F_{n-1} + F_{n-2}$ for $n > 1$

You can define this as follows:

```
def fib(n):
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)
```

What will ncalls be?

```
1   def fib(n):
2       if n < 2:
3           return n
4       return fib(n-1) + fib(n-2)
5
6   if __name__ == "__main__":
7       import cProfile
8       profiler = cProfile.Profile()  # Initializing a profiler instance
9       profiler.run('fib(32)')        # Run profiler on fib(32)
10      profiler.print_stats()         # Print stats
```

Profiling results for `fib(32)`

```
7049158 function calls (4 primitive calls) in 1.148 seconds

Ordered by: standard name

ncalls      tottime  percall  cumtime  percall filename:lineno(function)
7049155/1    1.148    0.000    1.148    1.148 <ipython-input-14-cb3508afaaa3>:1(fib)
1            0.000    0.000    1.148    1.148 <string>:1(<module>)
1            0.000    0.000    1.148    1.148 {built-in method builtins.exec}
1            0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' obj
```

Note: **primitive call** counts calls excluding recursion.

cProfile may have a lot of overhead

when ncalls >> 1!

The recursive Fibonacci implementation nicely can illustrate the overhead costs of cProfile. Let's measure it it with timeit.repeat

```
1   ## adding to if __name__ == "__main__":
2   import timeit
3   print("Runtime:", timeit.repeat('fib(32)', \
4                       setup="from __main__ import fib",repeat= 5,number=1))
```

returns (5 `repeats` of the runtime as a list; `number` is # of iterations for the main loop to be carried out):

```
1   Runtime: [0.5889114580000125, 0.588767292, 0.5895812910000018,
0.5903672500000141, 0.5895792499999857]
```

which is about half as long!!! Thus,

- don't use cProfile for estimating runtimes (per se)
- use it for assessing relative runtimes & bottlenecks

Alternatives to cProfile: Statistical profiling

- Statical profiler do not track every call, but instead the call stack every 1ms
- this gives much lower overhead costs
- note that tracking profilers (such as cProfile) can distort the results (e.g. when there are lots of lots of function calls)
- If you are interested, check out the statistical profiler Pyinstrument

Timeit is a lightweight alternative for us!

Timeit: measure exection time for code snipplets

The Timeit module provides a simple way to time small bits of Python code (it is highly accurate and runs code statements multiple times for robustness).

Simple example:

```
# testing timeit()
In [1]: import timeit
   ...: print(timeit.timeit('output = 30*125'))
0.03277275000003015
# Combining multiple lines with ;
In [2]: print("Summation takes ",timeit.timeit(stmt='x=123.3;y=0.245;sum=x+y'))
Summation takes  0.06631987499999781
```

Timeit: measure exection time for code snipplets

The Timeit module provides a simple way to time small bits of Python code (it is highly accurate and runs code statements multiple times for robustness).

Simple example:

```
# testing timeit()
In [1]: import timeit
   ...: print(timeit.timeit('output = 30*125'))
0.03277275000003015
# Combining multiple lines with ;
In [2]: print("Summation takes ",timeit.timeit(stmt='x=123.3;y=0.245;sum=x+y'))
Summation takes  0.06631987499999781
```

Timeit: measure exection time for code snipplets

The Timeit module provides a simple way to time small bits of Python code (it is highly accurate and runs code statements multiple times for robustness).

Simple example:

```
# testing timeit()
In [1]: import timeit
   ...: print(timeit.timeit('output = 30*125'))
0.03277275000003015
# Combining multiple lines with ;
In [2]: print("Summation takes ",timeit.timeit(stmt='x=123.3;y=0.245;sum=x+y'))
Summation takes  0.06631987499999781
```

Timeit's CLI

Timeit also has a convenient command line interface:

```
alex@mac% python -m timeit 'try:' '  str.__bool__' 'except AttributeError:' '  pass'
1000000 loops, best of 5: 304 nsec per loop

alex@mac% python -m timeit 'if hasattr(str, "__bool__"): pass'
2000000 loops, best of 5: 197 nsec per loop
```

Another use case: timestamps

```
1   import numpy as np
2   import timeit
3
4   N=2**5
5   print("Running test with N =", str(N))
6
7   start = timeit.default_timer()
8   calltime=0
9
10  def fib(n):
11      global calltime
12      calltime+=1
13      if n < 2:
14          return n
15      return fib(n-1) + fib(n-2)
16
17  fibo_list=[fib(n) for n in range(N)]
18  stop = timeit.default_timer()
19
20  print(fibo_list)
21  print("Method recursion: ", round(stop - start,6),' seconds')
22  print("fib was called ", calltime, " times!")
```

Output from this script

```
Running test with N = 32
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181,
6765, 10946, 17711, 28657, 46368, 75025, 121393, 196418, 317811, 514229, 832040, 1346269]
Method recursion:  1.602827  seconds
fib was called  11405740  times!
```

Quiz: How can we improve the speed of Fibonacci?

This recursive code is highly inefficient …

```
1   def fib(n):
2       if n < 2:
3           return n
4       return fib(n-1) + fib(n-2)
```

Let's illustrate this for fib(6):

Reminder: Simple recursion implementation

```
N=2**5
print("Running test with N =", str(N))
start = timeit.default_timer()
calltime=0
def fib(n):
    global calltime
    calltime+=1
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)

fibo_list=[fib(n) for n in range(N)]
stop = timeit.default_timer()
print(fibo_list)
print("Method recursion: ", round(stop - start,6),' seconds')
print("fib was called ", calltime, " times!")
```

```
Running test with N = 32
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765
Method recursion:  1.602827  seconds
fib was called  11405740  times!
```

Simple memoization implementation

```
start = timeit.default_timer()
calltime=0
Fibs={0: 0, 1:1}                    # initialize memory as dictionary
def fib(n):
    global calltime
    calltime+=1
    if n in Fibs.keys():            # if result in memory, look it up!
        return Fibs[n]
    else:
        Fibs[n]=fib(n-1) + fib(n-2) # if result not in memory... compute...
        return Fibs[n]

fibo_list=[fib(n) for n in range(N)]
stop = timeit.default_timer()

print("Method dictionary cache: ", round(stop - start,6),' seconds')
print("fib was called ", calltime, " times!")
```

```
Method dictionary cache:  6.5e-05  seconds
fib was called  92  times!
```

Using built-in cache functools for memoization

```
from functools import lru_cache
start = timeit.default_timer()
calltime=0
@lru_cache(maxsize=None)
def fib(n):
    global calltime
    calltime+=1
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)

fibo_list = [fib(n) for n in range(N)]
stop = timeit.default_timer()
print("Method cached: ", round(stop - start,6),' seconds')
print("fib was called ", calltime, " times!")
```

```
Method cached:  3.4e-05  seconds
fib was called  32  times!
```

Here, `@lru_cache` is a decorator to wrap a function with a memoizing callable that saves up to the `maxsize` most recent calls; here LRU stands for *least recently used*.

Questions?

Clustering

Clustering is an important computational task to group data into different groups (called clusters).

Clustering has many applications. You'll encounter it in many different classes in the future.

- genome analysis
- behavioral analysis
- social network analysis
- ...

Scikit-learn has many clustering algorithms implemented. Check them out, if you are interested!

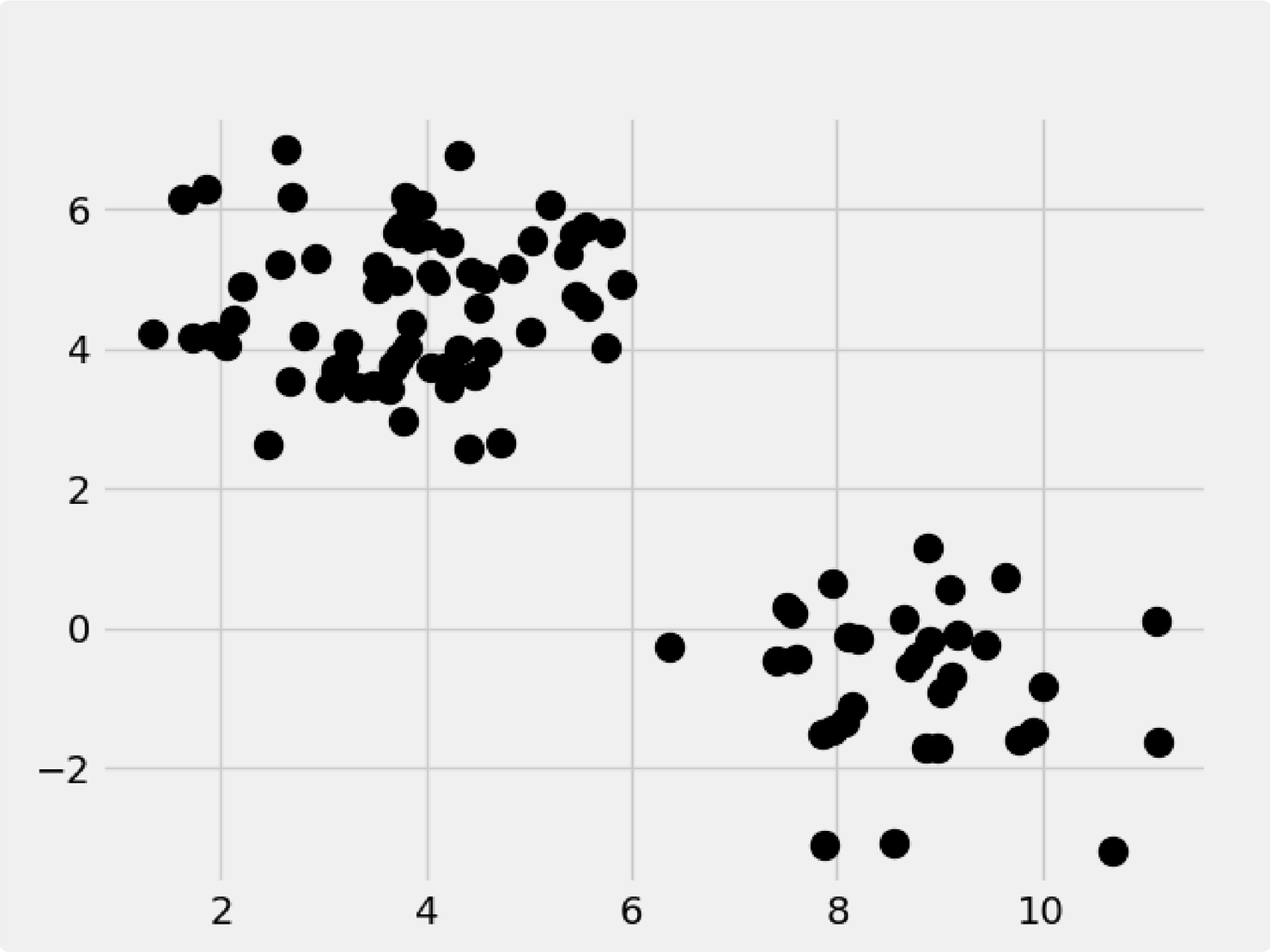
Problem Setting

Imagine you have some data with different features (see the next slide for an example with 2 feature dimensions). The ground truth labels are shown on the slide after that. However, how the data was generated is unknown to us and for clusting, the goal is to assign each data point to one cluster so that the assignment represents the data efficiently.

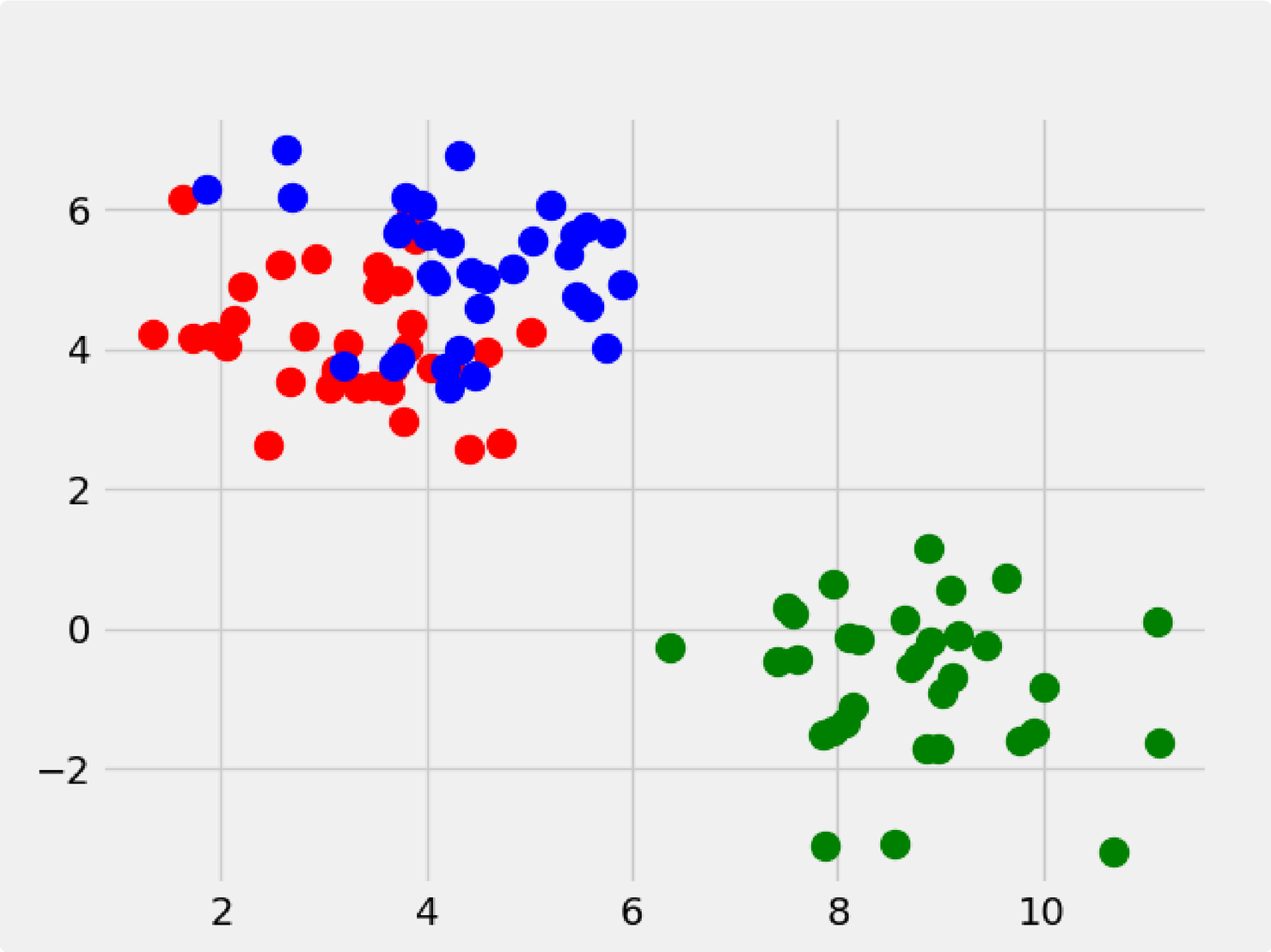
Today we will learn about one classic clustering algorithms, Kmeans. K-means is an iterative algorithm that partitions the data into groups based on the distance between the centroids. The centroid, or cluster center, is the mean of all the points within the cluster.

We pick this algorithm, as it is 1) fundamental algorithm that you should know (you will learn much more in later classes at EPFL) and 2) can nicely be implemented in different ways to show profiling in action.

Example dataset (two features on x and y axis)



Example dataset (ground truth labels)



K-means

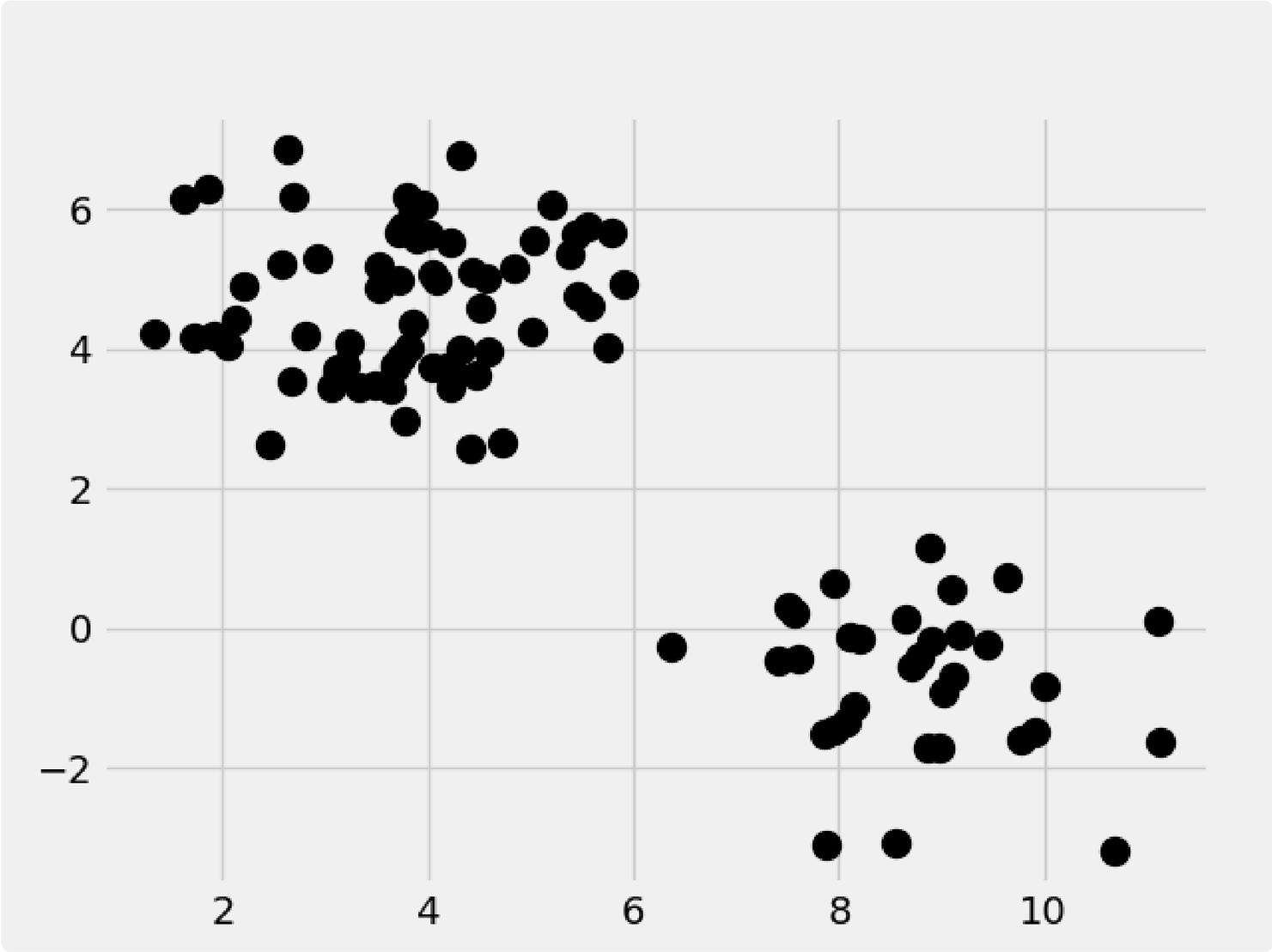
Here is a pseudo algorithm:

First: (randomly) pick centers

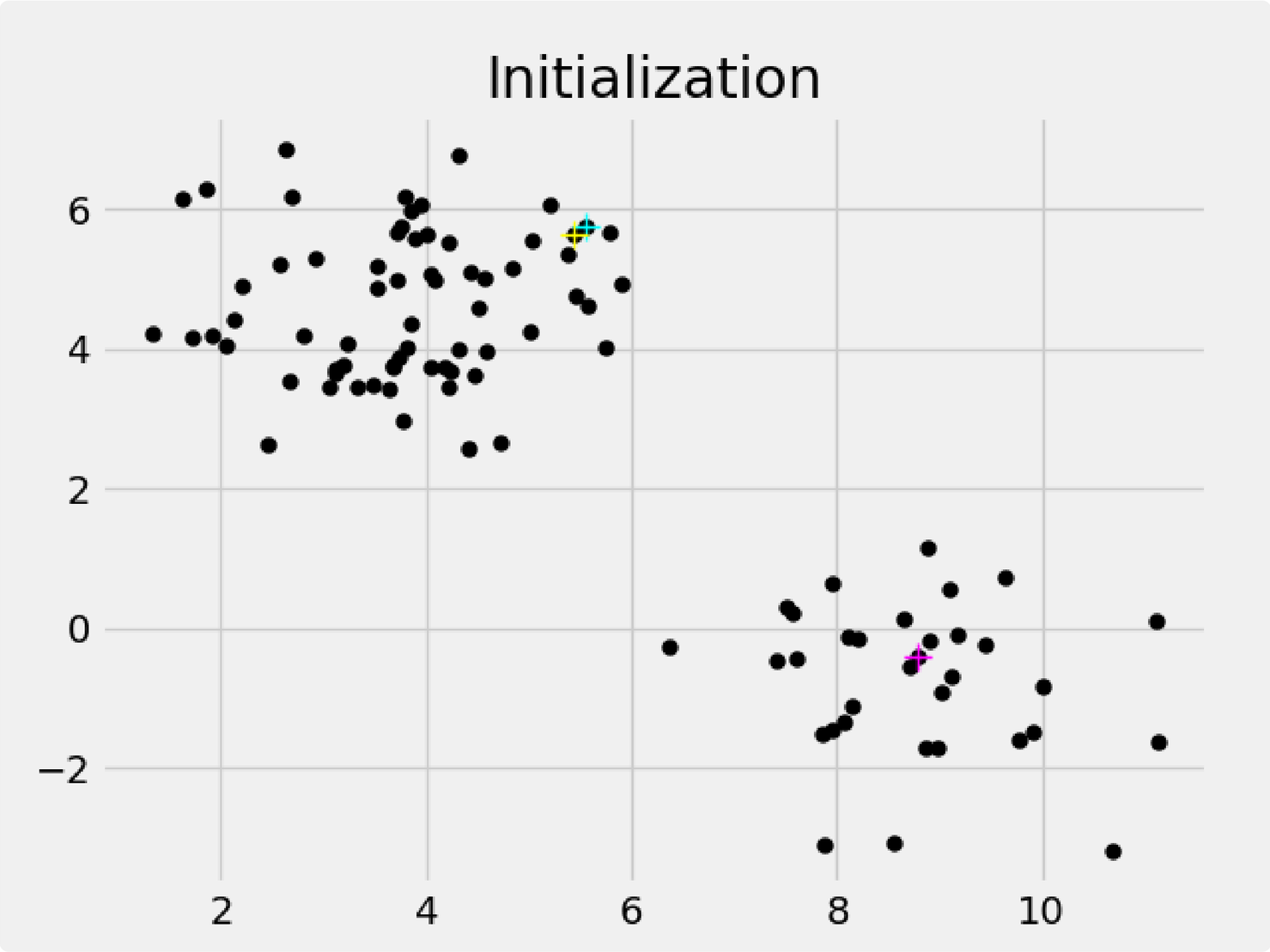
Then iterate:

1. assign data points to nearest center (this defines the clusters)
2. update the center to the mean of the points in each cluster
3. check convergence (stop if converged, otherwise goto 1)

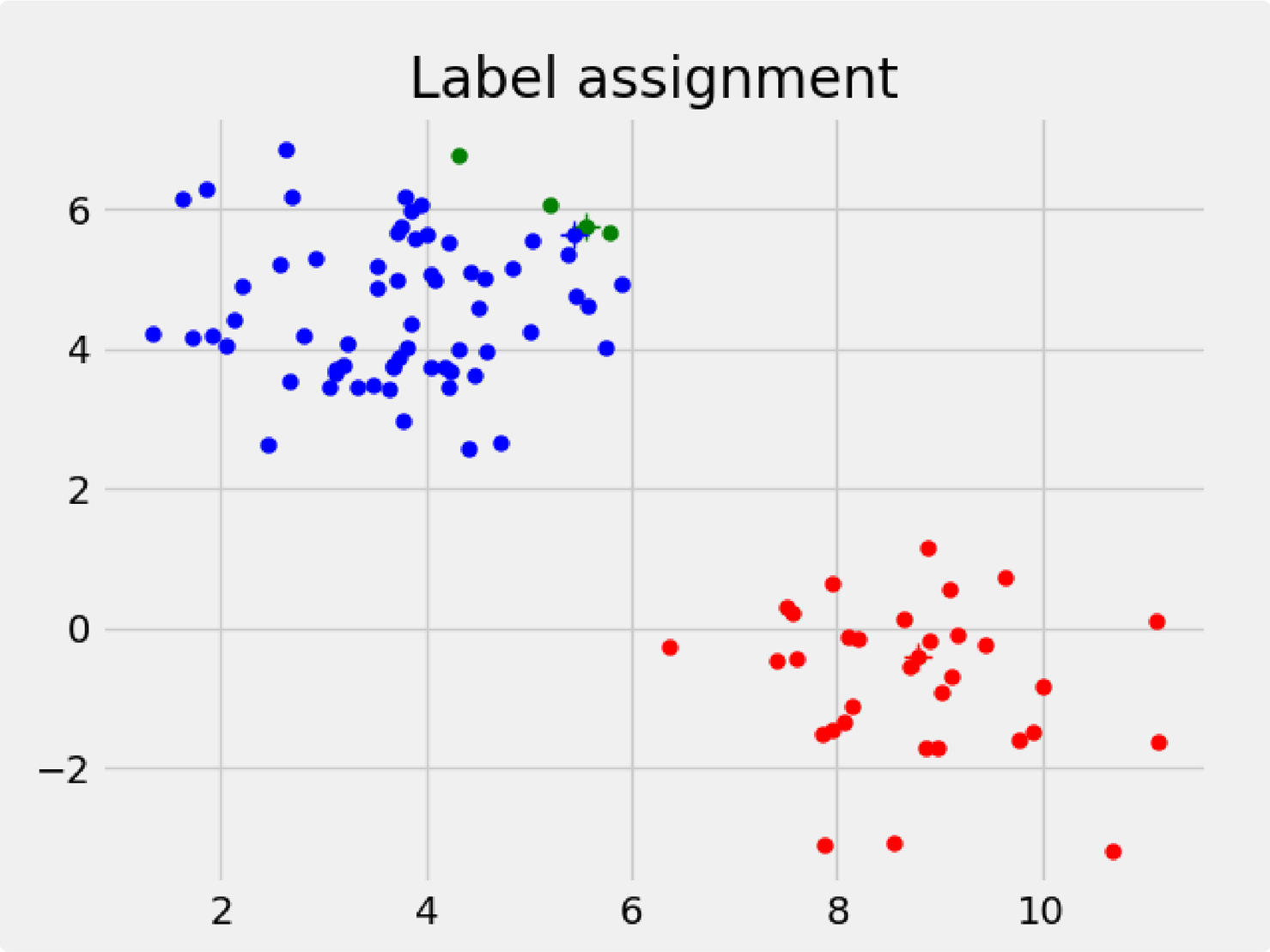
Example dataset



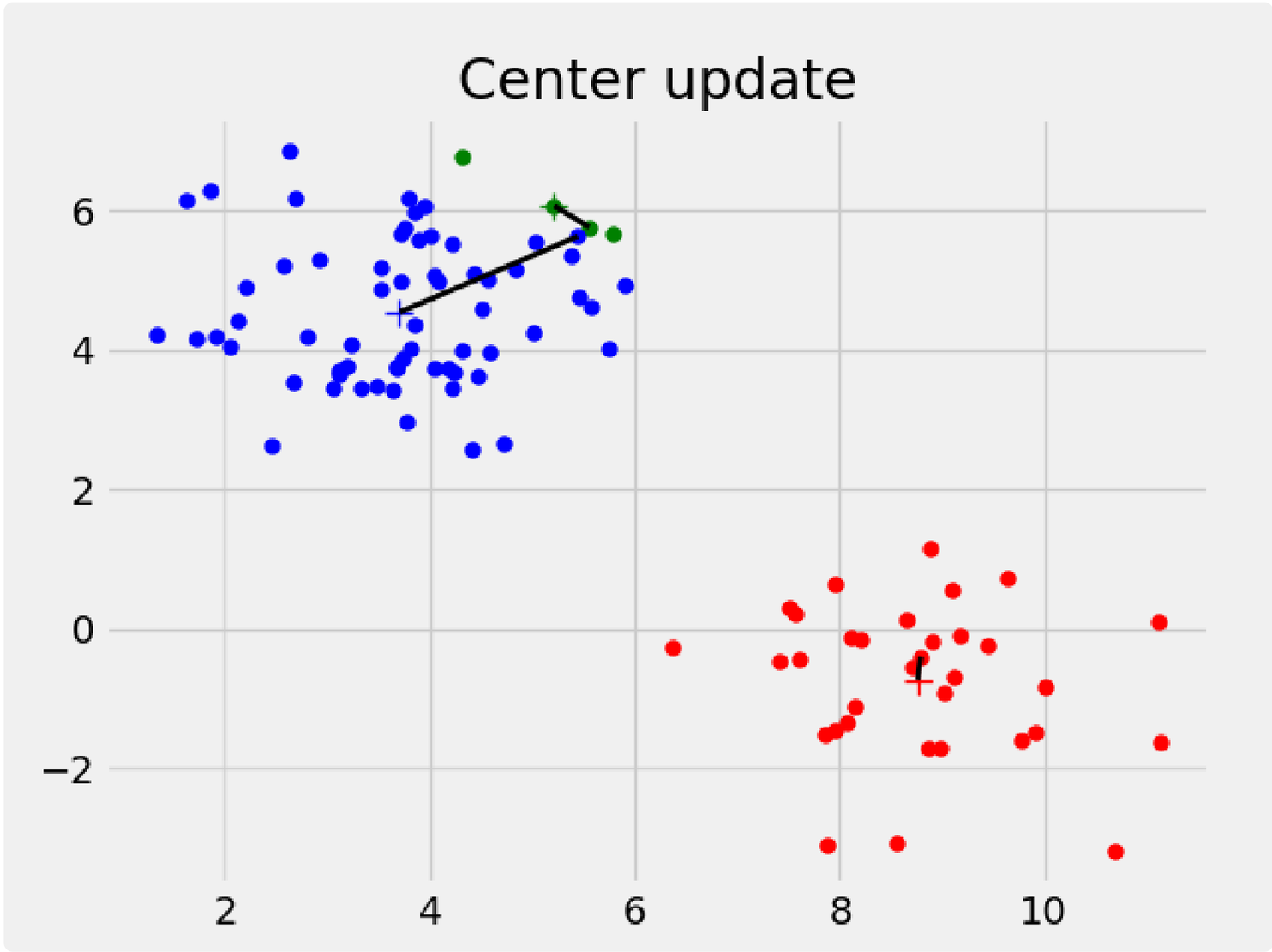
Initialize k-means



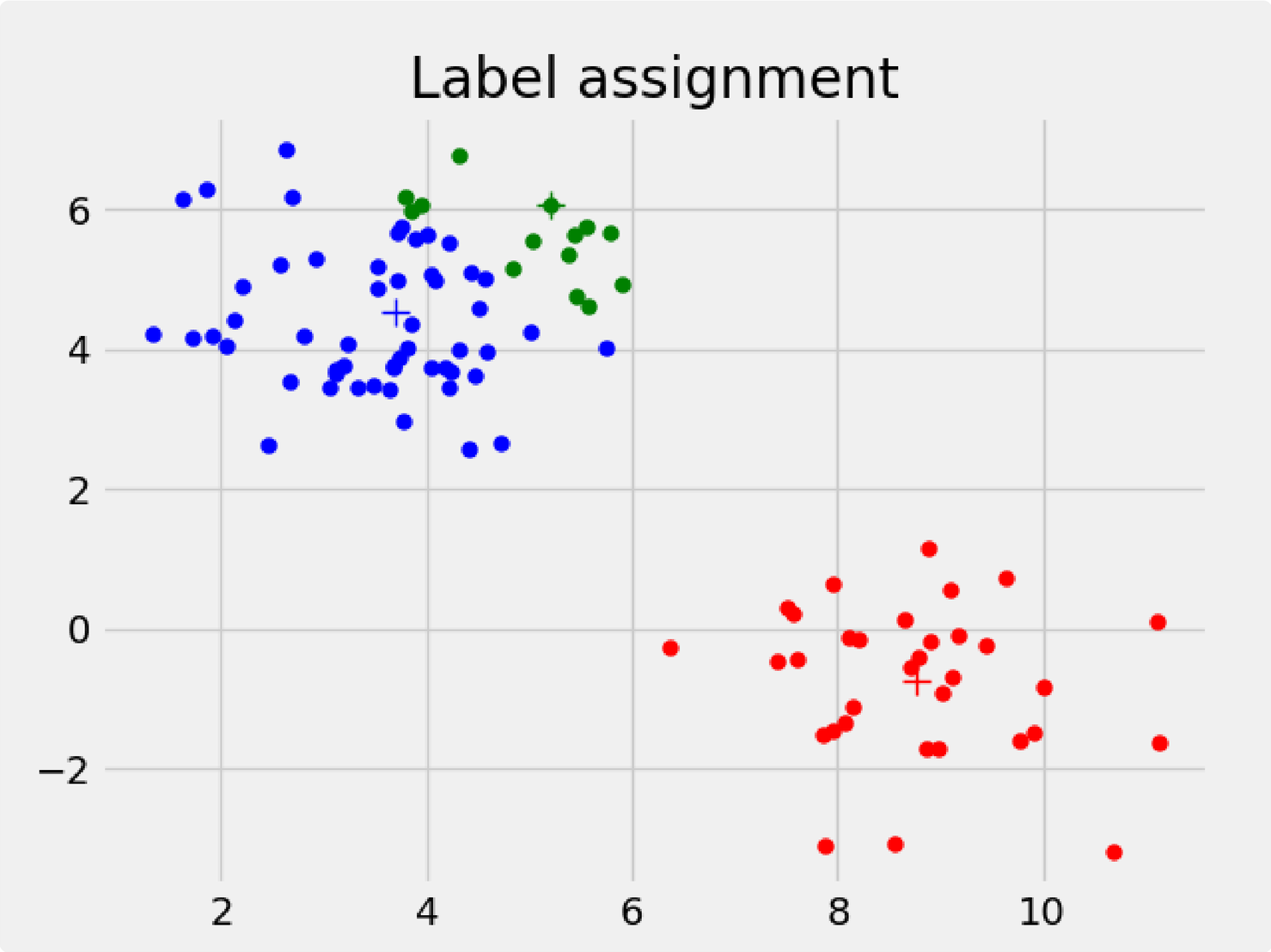
Assign label (iteration: 0)



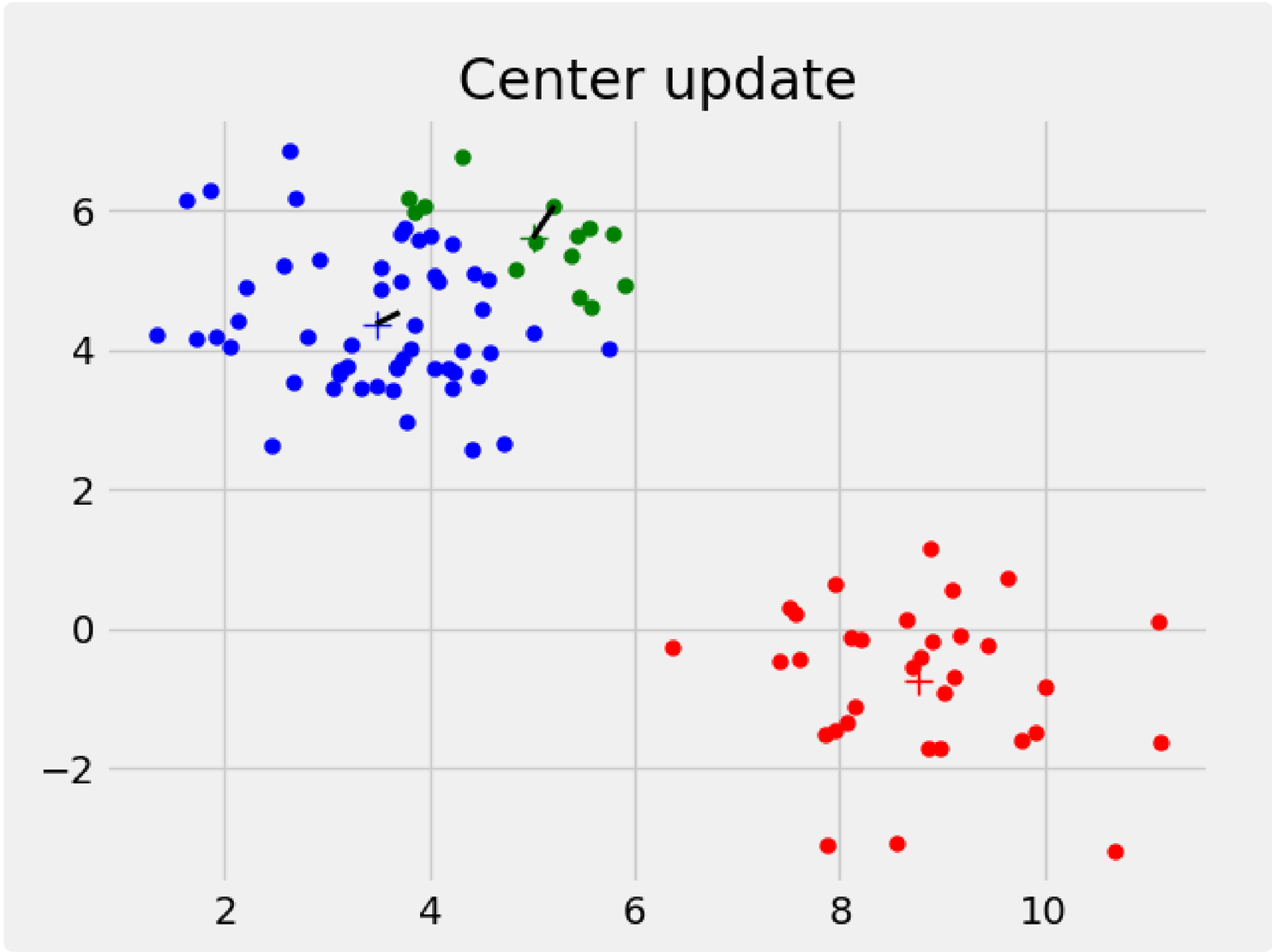
Update centers (iteration: 0)



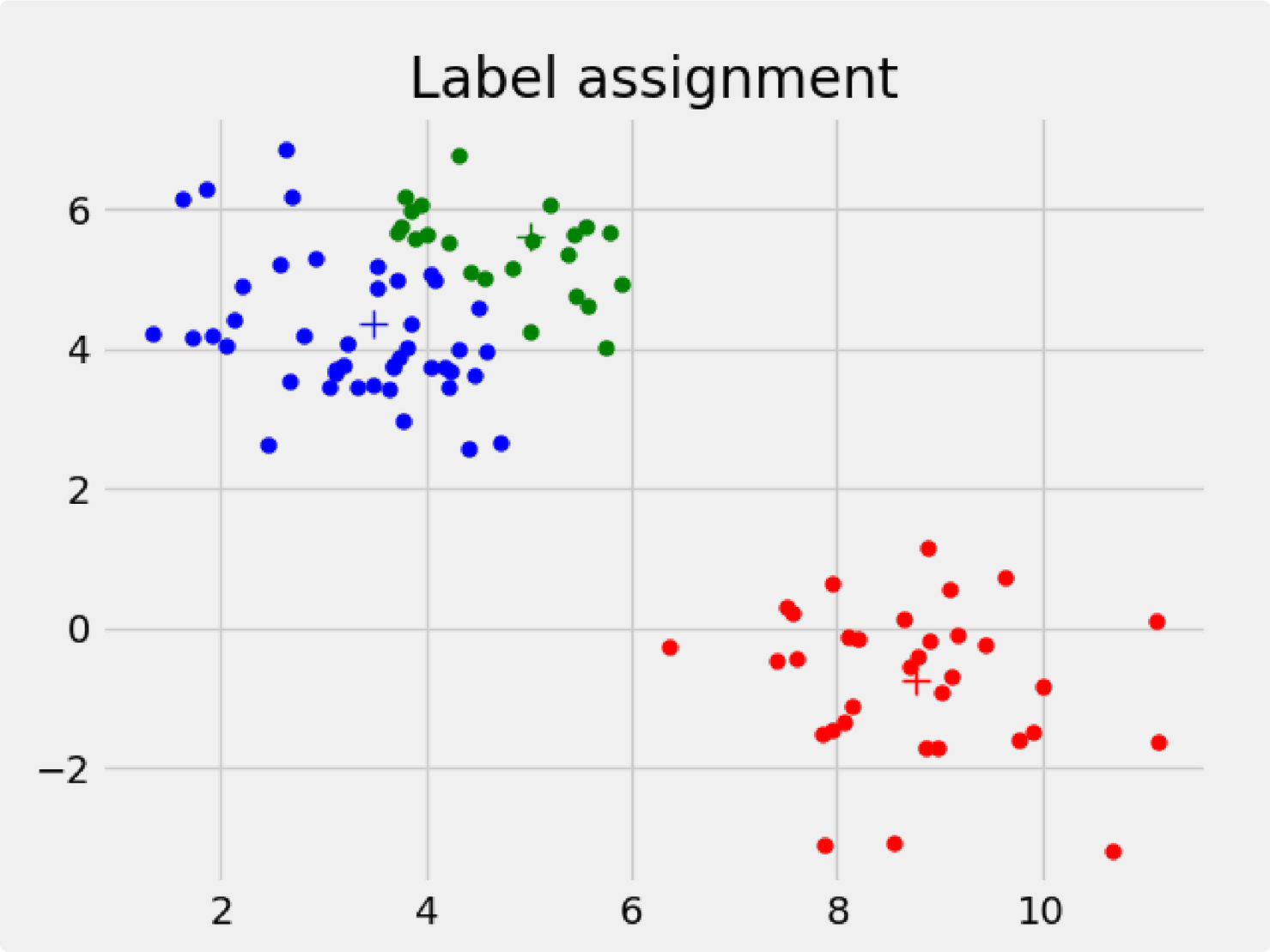
Assign label (iteration: 1)



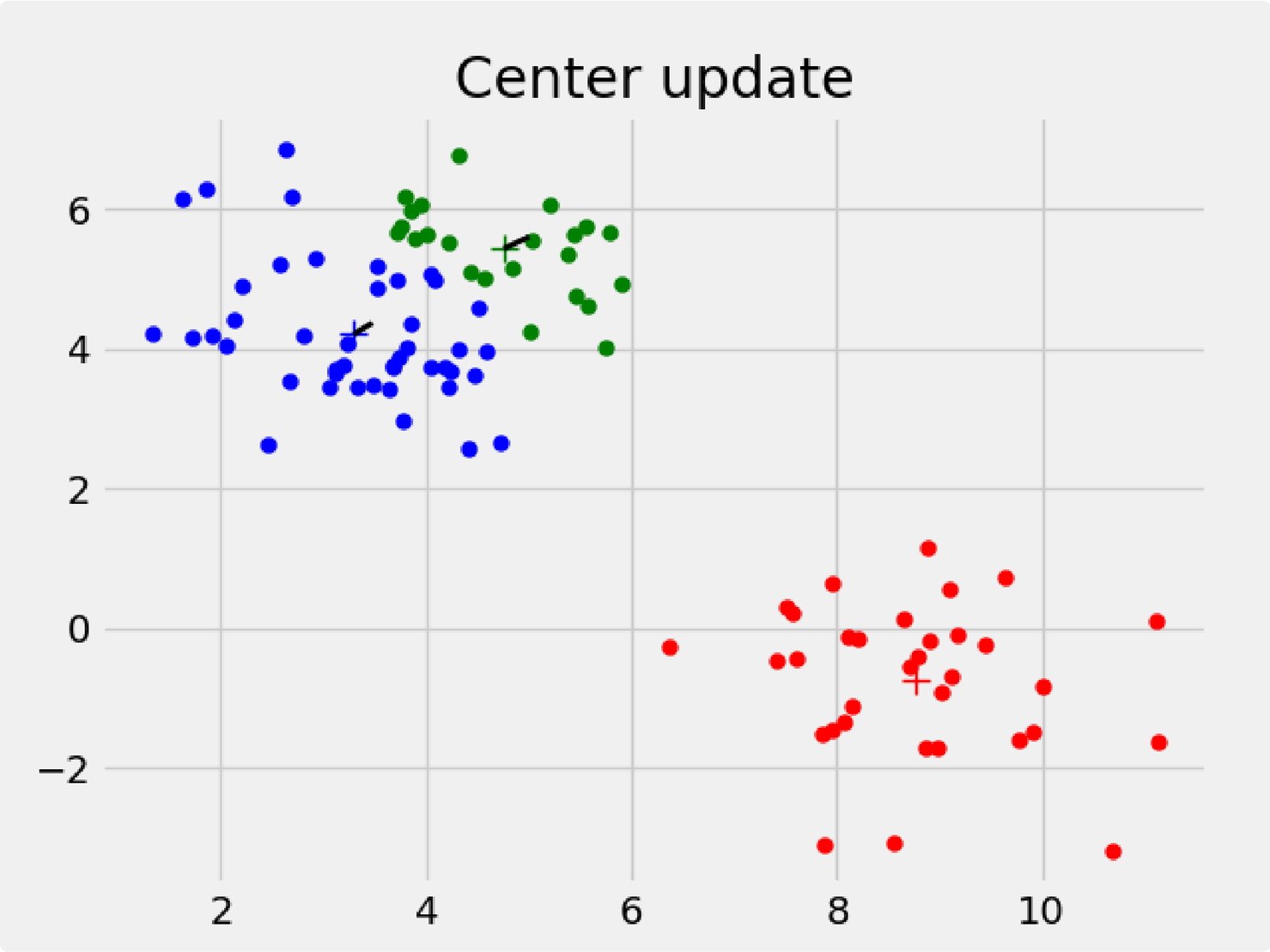
Update centers (iteration: 1)



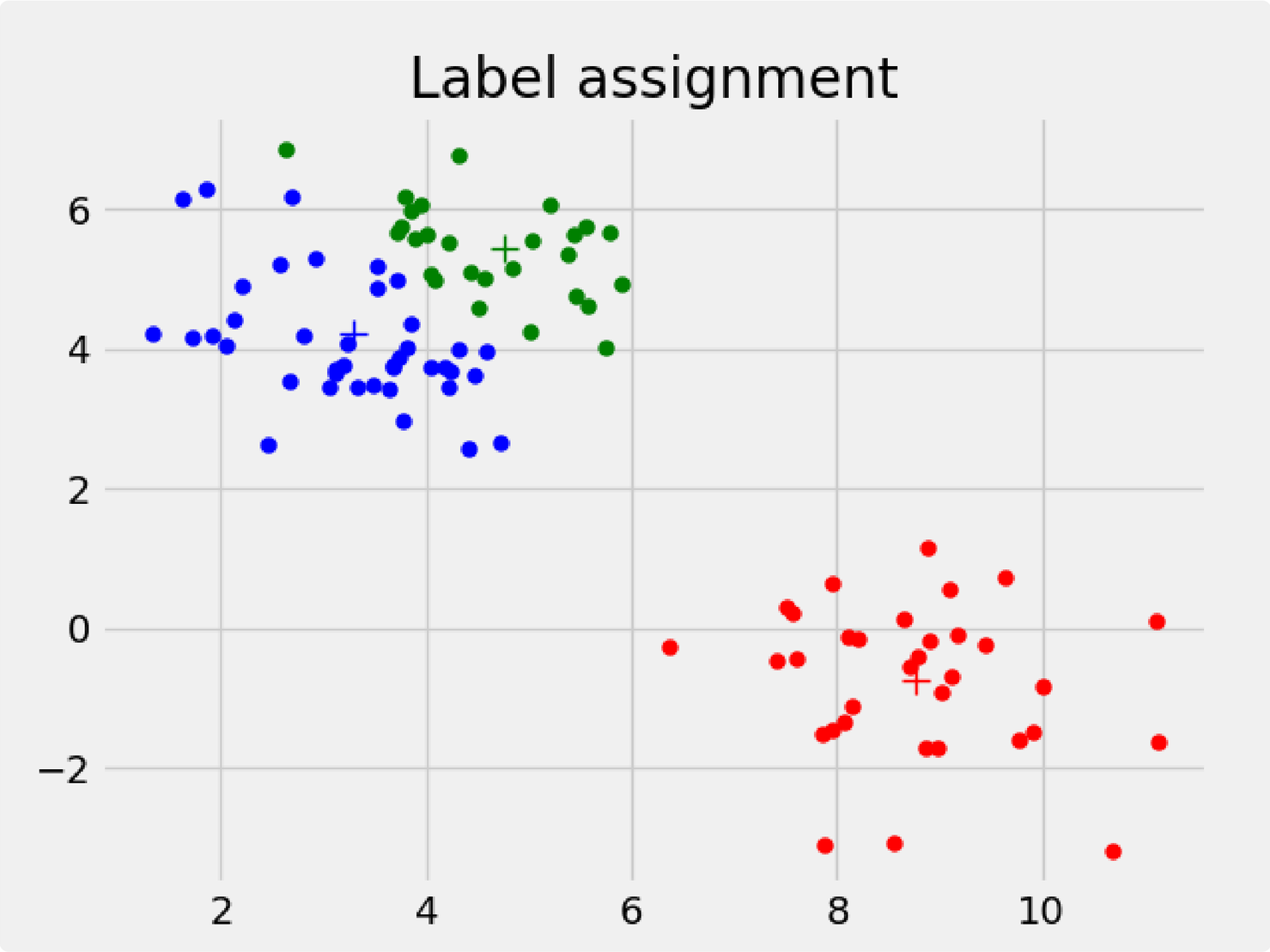
Assign label (iteration: 2)



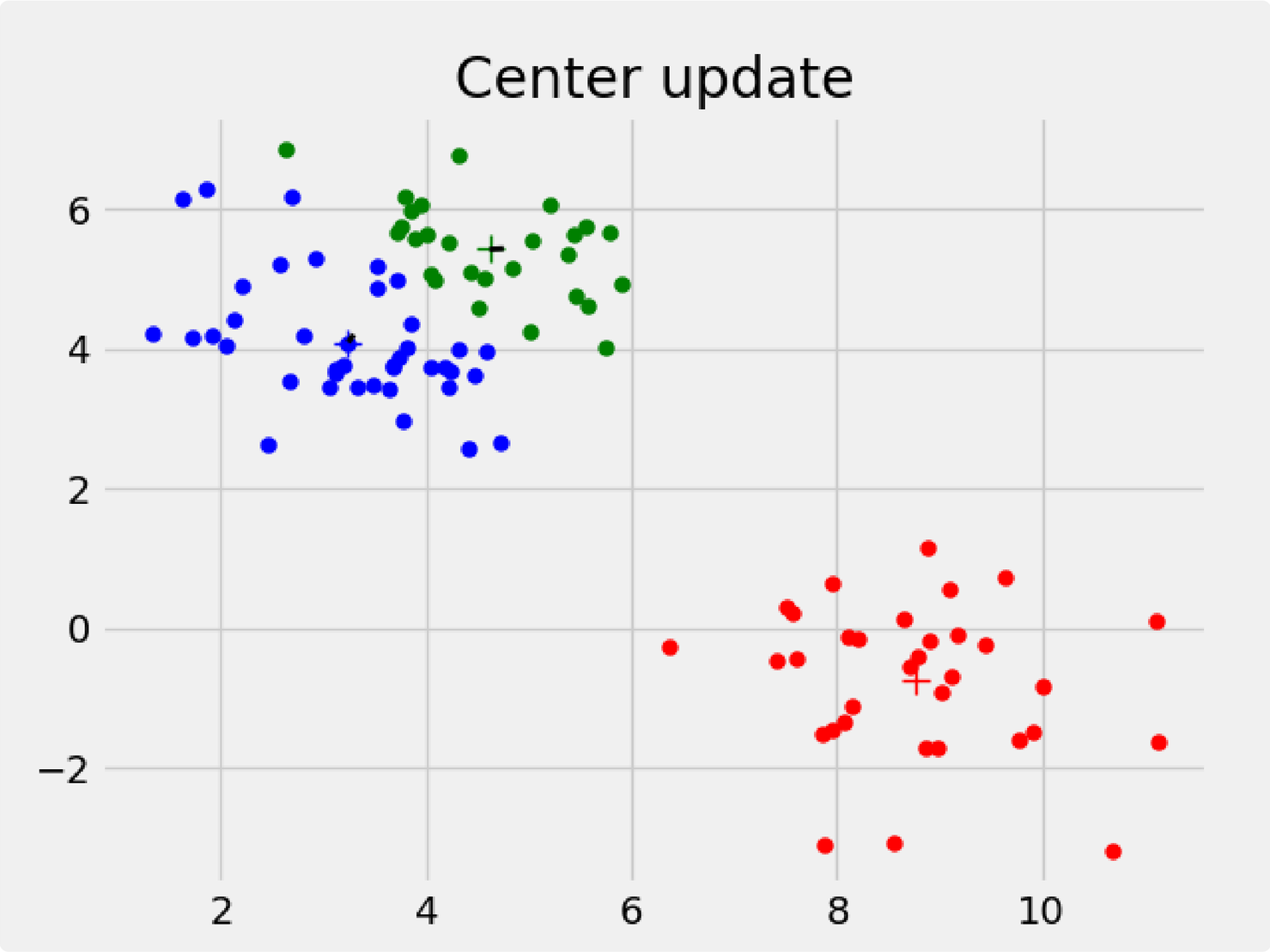
Update centers (iteration: 2)



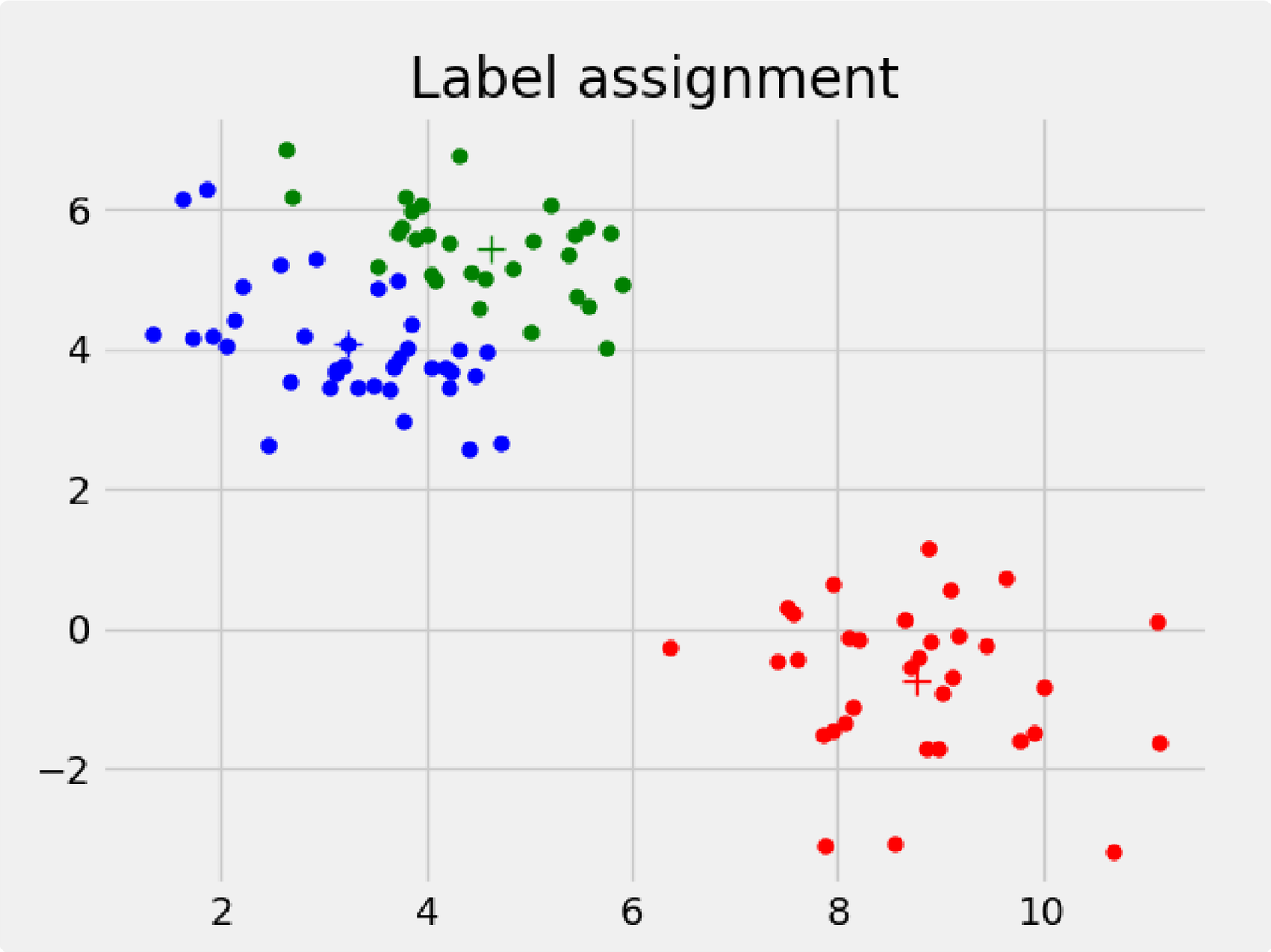
Assign label (iteration: 3)



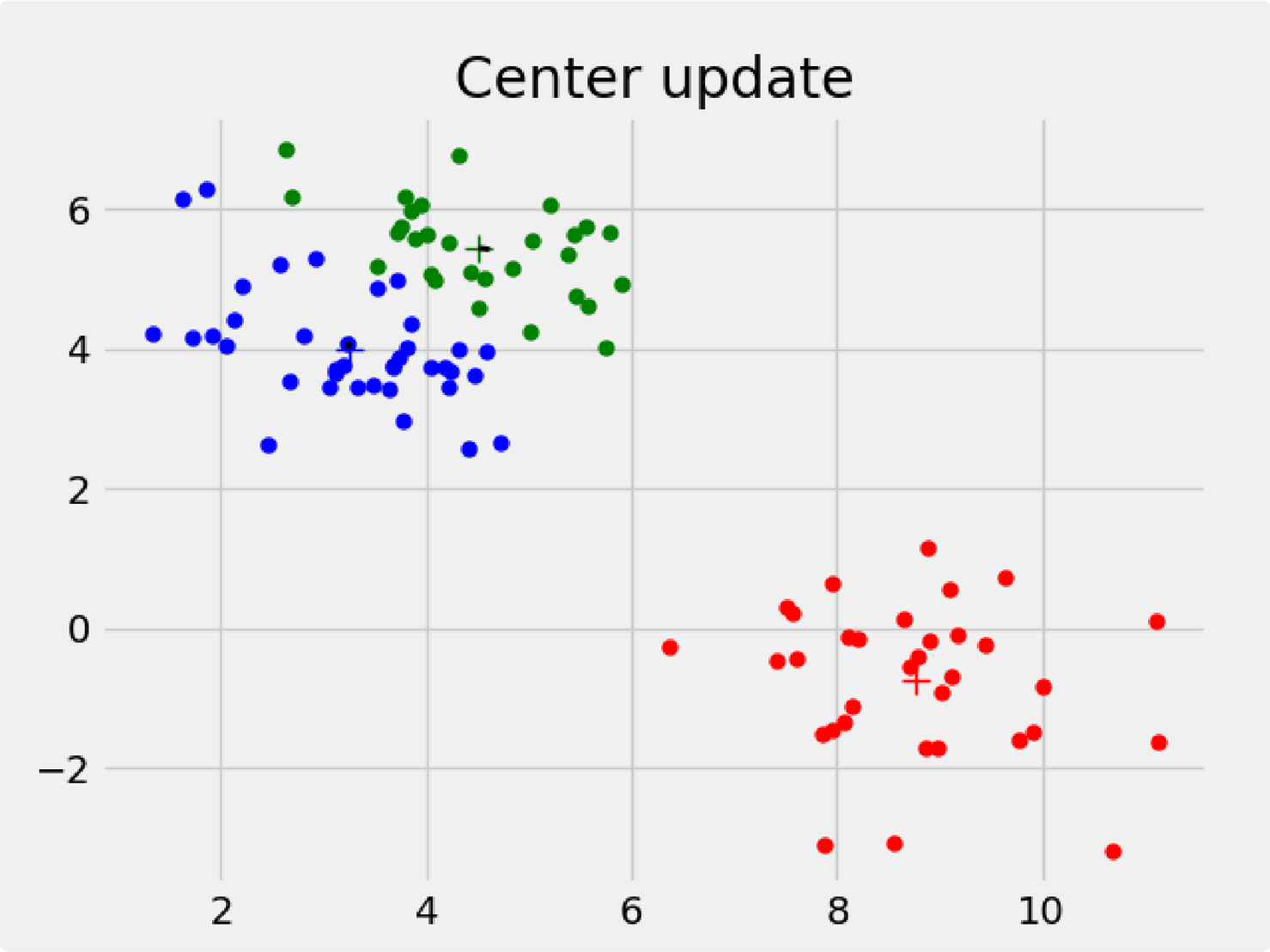
Update centers (iteration: 3)



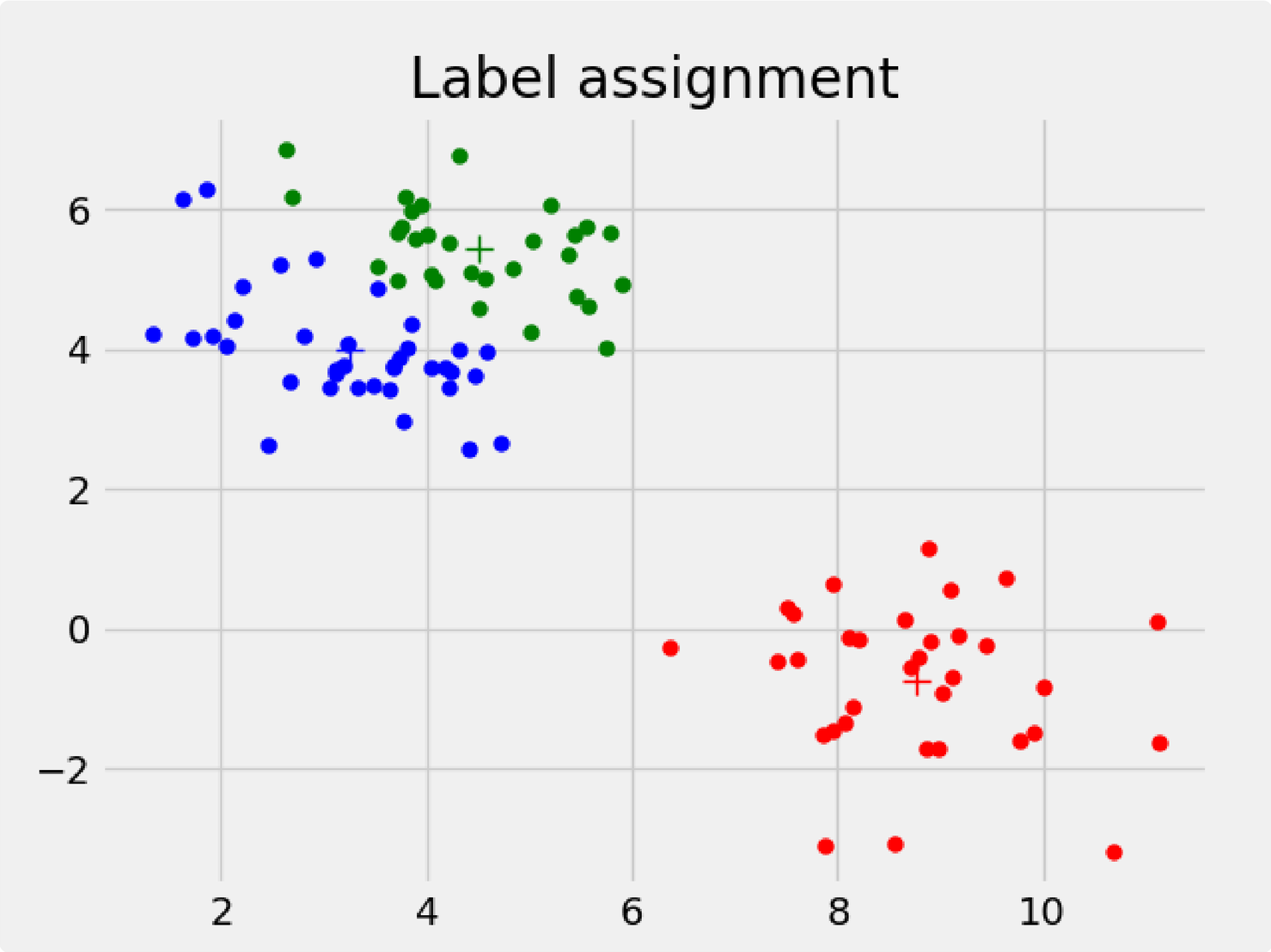
Assign label (iteration: 4)



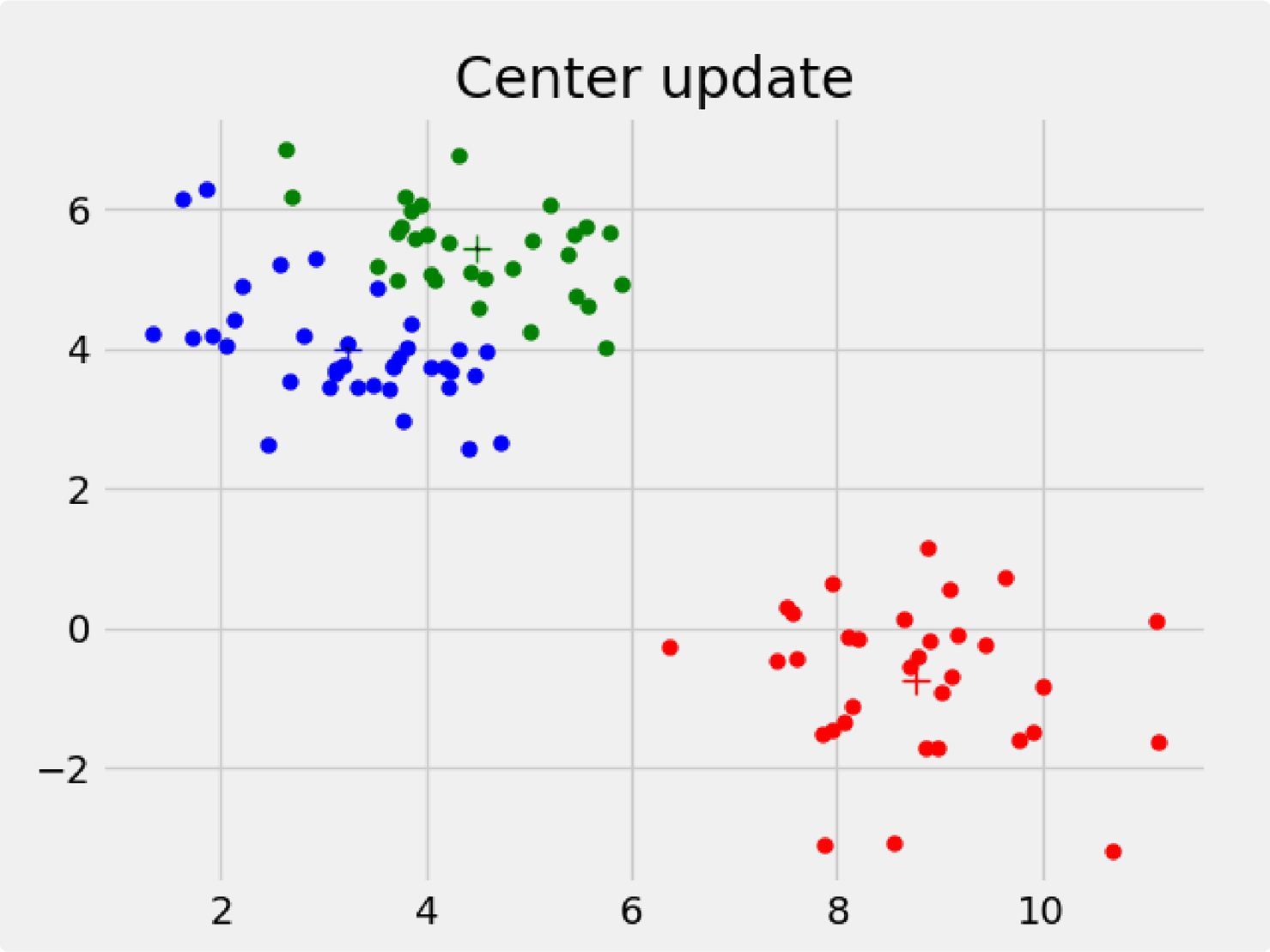
Update centers (iteration: 4)



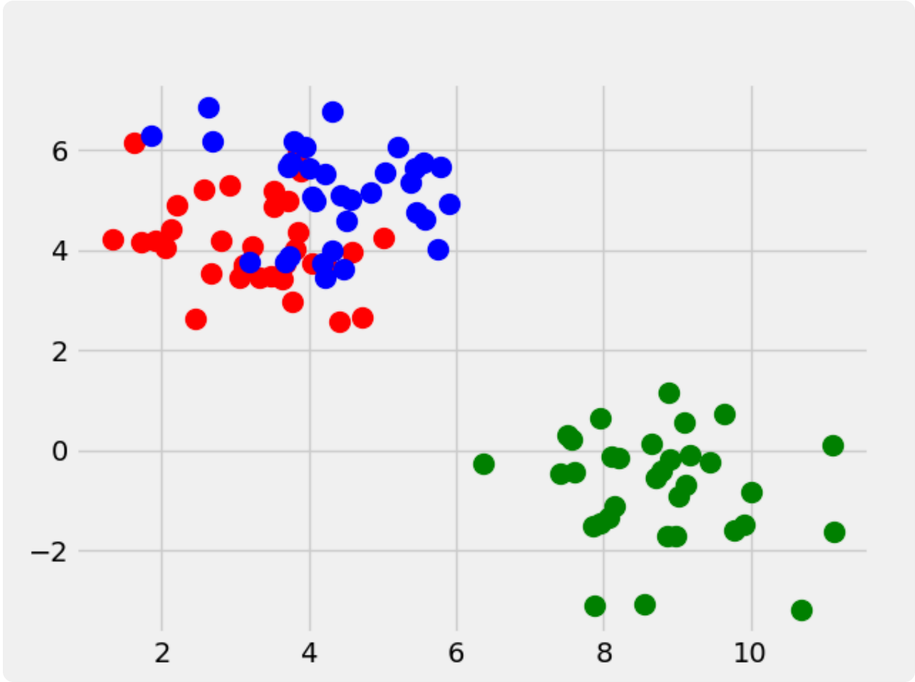
Assign label (iteration: 5)



Update centers (iteration: 5); it converged



Example dataset (ground truth labels)



Side note: Of course, the specific label (here color) cannot be exactly recovered. K-means recovers three reasonable clusters, but of course fails to accurately assign the datapoints where the blue and red clusters mix. If you are interested in learning more, check out alternative clustering algorithms..

We will now return to our goal, profiling this algorithm!

K-means clustering algorithm

First: (randomly) pick centers

Then iterate:

1. assign data points to nearest center (this defines the clusters)
2. update the center to the mean of the points in each cluster
3. check convergence (stop if converged, otherwise goto 1)

Note: Code below adapted from Jake VanderPlas.

Background: zip / zip*

```
In [1]: list(zip((0,1,2),(5,3,6)))    # combing two lists into lists of pairs
Out[1]: [(0, 5), (1, 3), (2, 6)]
In [2]: list(zip(*[(0,1,2),(3,4,5)])) # Unpacking triples into list of 3 tuples
Out[2]: [(0, 3), (1, 4), (2, 5)]
In [3]: zip?
Init signature: zip(self, /, *args, **kwargs)
Docstring:
zip(*iterables) --> A zip object yielding tuples until an input is exhausted.

   >>> list(zip('abcdefg', range(3), range(4)))
   [('a', 0, 0), ('b', 1, 1), ('c', 2, 2)]
```

Pure Python implementation: assigning labels

```
1   import math
2   def distance(v, w):
3       """ Computes the Euclidean distance for vectors v and w"""
4       return math.sqrt(sum((vi - wi) ** 2 for vi, wi in zip(v, w)))
5
6
7   def assign_labels(data, centers):
8       """ Assign data to closest label (corresponding to centers)."""
9       labels = []
10      for item in data:
11          distances = [distance(item, center) for center in centers]
12          labels.append(distances.index(min(distances)))
13      return labels
```

```python
def update_centers(data, labels):
    """ Update centers of mass according to labels """
    n_centers = len((set(labels)))
    n_dims = len(data[0])
    centers = [[0 for i in range(n_dims)] for j in range(n_centers)]
    counts = [0 for j in range(n_centers)]

    for label, item in zip(labels, data):
        counts[label] += 1     # keeping count of # elements and summing per label:
        centers[label] = [ci + ii for ci, ii in zip(centers[label], item)]

    return [[x / count for x in center] for center, count in zip(centers, counts)]

def kmeans(data, n_clusters):
    centers = data[-n_clusters:]
    iteration = 0
    while True:
        old_centers = centers
        labels = assign_labels(data, centers)
        centers = update_centers(data, labels)

        if centers == old_centers:
            break

    return labels
```

Setting up cProfile

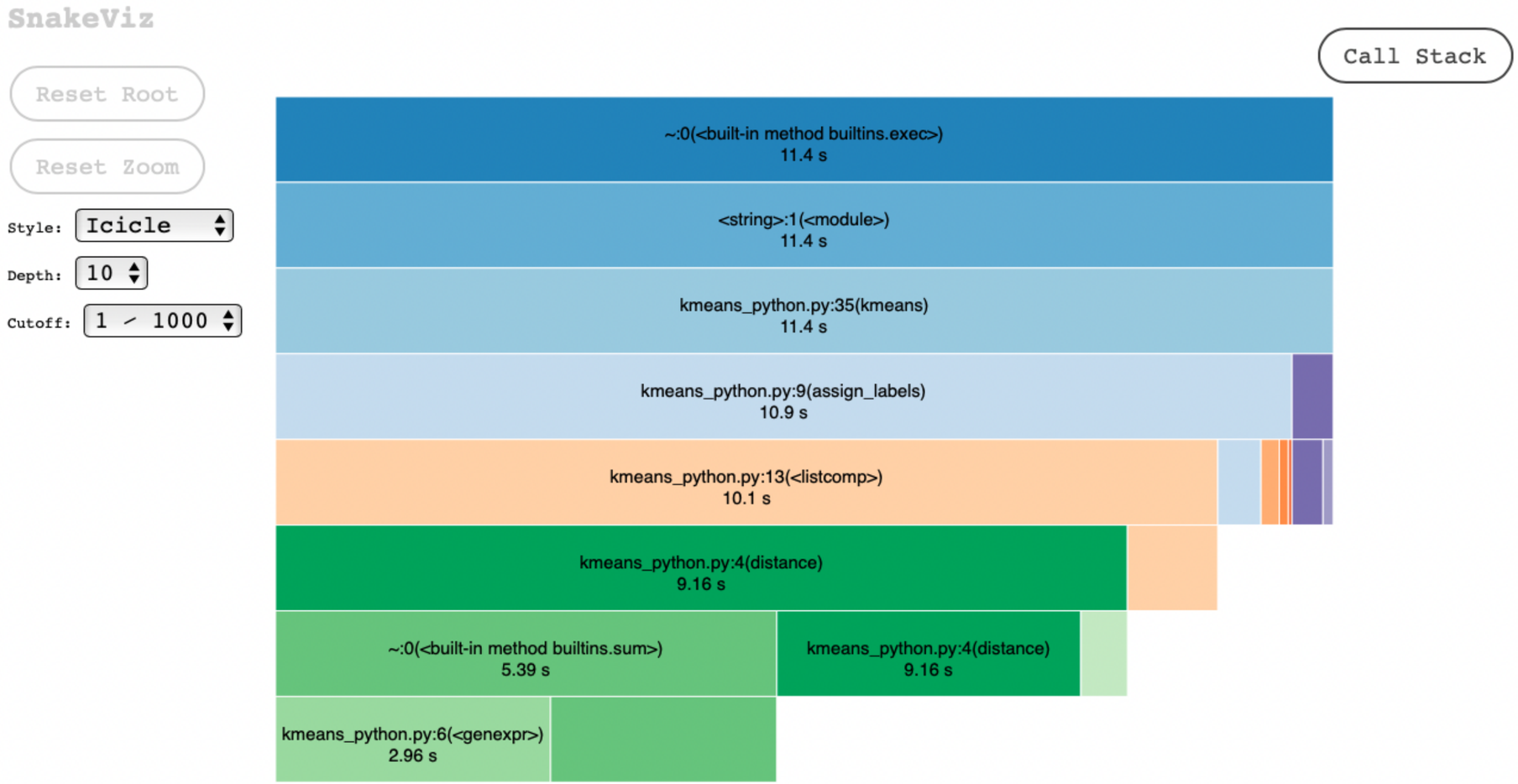
```
## Experiment:
import kmeans.utils, timeit

# Creating some data:
n_clusters = 10
n_dims = 10000
data, gt_labels = kmeans.utils.sampledata(n_dims, n_clusters, plotting=False)

from kmeans.kmeans_python import kmeans

import cProfile

cProfile.run("kmeans(data,n_clusters)", "summary_kmeans.stats")

import pstats
stats = pstats.Stats("summary_kmeans.stats")
stats.sort_stats("calls")
stats.print_stats()
```

Output of cProfile

```
1   Thu Nov 14 18:54:52 2024    summary_kmeans.stats

         50701252 function calls in 11.380 seconds

   Ordered by: call count

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
 23400000    2.959    0.000    2.959    0.000 /Users/alex/Code/Teaching/profiling/kmeans/kmeans
  7800000    2.433    0.000    5.392    0.000 {built-in method builtins.sum}
  7800000    0.504    0.000    0.504    0.000 {built-in method math.sqrt}
  7800000    3.269    0.000    9.165    0.000 /Users/alex/Code/Teaching/profiling/kmeans/kmeans
   780000    0.041    0.000    0.041    0.000 {method 'append' of 'list' objects}
   780000    0.096    0.000    0.096    0.000 {method 'index' of 'list' objects}
   780000    0.199    0.000    0.199    0.000 {built-in method builtins.min}
   780000    0.971    0.000   10.136    0.000 /Users/alex/Code/Teaching/profiling/kmeans/kmeans
   780000    0.112    0.000    0.112    0.000 /Users/alex/Code/Teaching/profiling/kmeans/kmeans
      780    0.000    0.000    0.000    0.000 /Users/alex/Code/Teaching/profiling/kmeans/kmeans
      156    0.000    0.000    0.000    0.000 {built-in method builtins.len}
       78    0.462    0.006   10.934    0.140 /Users/alex/Code/Teaching/profiling/kmeans/kmeans
       78    0.000    0.000    0.000    0.000 /Users/alex/Code/Teaching/profiling/kmeans/kmeans
       78    0.000    0.000    0.000    0.000 /Users/alex/Code/Teaching/profiling/kmeans/kmeans
       78    0.331    0.004    0.445    0.006 /Users/alex/Code/Teaching/profiling/kmeans/kmeans
        1    0.000    0.000   11.380   11.380 {built-in method builtins.exec}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

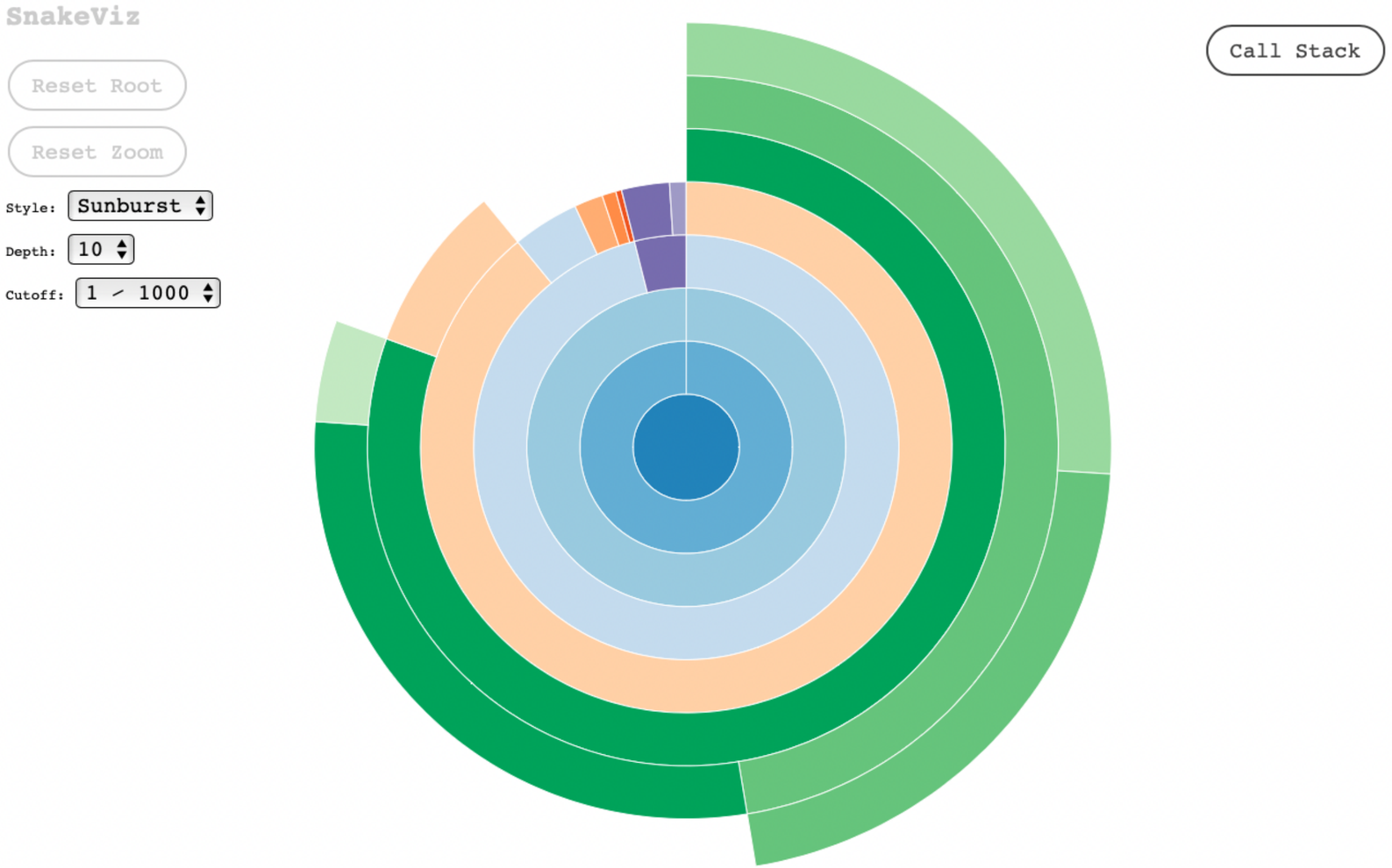
Visualization type 1 (icicle)

```
1   snakeviz summary_kmeans.stats
```



Visualization type 2 (sunburst)

```
1   snakeviz summary_kmeans.stats
```



Alternative profiling (line_profiling)

Especially for numerical code, from the cProfile output it is

- not apparent what exactly is computed (without going back to the code)
- calls/times are grouped by function calls

What if much work happens in particular lines, e.g. loading data, diagonalizing a matrix ...

For instance, most calls are from this function:

```
 1   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
23400000    2.959    0.000    2.959    0.000 /Users/alex/Code/Teaching/profiling/kmeans/kmeans
```

Line-by-line profiling

`line_profiler` is a module for doing line-by-line profiling of functions. kernprof is a convenient script for running either line_profiler.

Here we will use it in ipython with magic commands

Installation:

```
pip install line_profiler
```

Demo code (LineProfiling.py)

```
1  import utils
2
3  n_clusters=10      # Initalizing data
4  n_dims =10000
5  data, gt_labels = utils.sampledata(n_dims, n_clusters, plotting= False)
6
7  from kmeans_python import kmeans
8  labels = kmeans(data, n_clusters)   # Running our kmeans
9  utils.score(gt_labels,labels)       # Evaluating performance
```

```
In [1]: run LineProfiling.py
0.9191471147114711
In [2]: %load_ext line_profiler
In [3]: %lprun -f kmeans kmeans(data,n_clusters)
Timer unit: 1e-06 s
Total time: 8.55269 s
File: /Users/alex/Code/Teaching/profiling/kmeans/kmeans_python.py
Function: kmeans at line 35

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    35                                           def kmeans(data, n_clusters):
    36         1         30.0     30.0      0.0      centers = data[-n_clusters:]
    37
    38         1          2.0      2.0      0.0      iteration = 0
    39                                               while True:
    40        41         31.0      0.8      0.0          old_centers = centers
    41        41    7422540.0 181037.6     86.8          labels = assign_labels(data, cen
    42        41    1130007.0  27561.1     13.2          centers = update_centers(data, l
    43
    44        41         81.0      2.0      0.0          if centers == old_centers:
    45         1          1.0      1.0      0.0              break
    46
    47         1          0.0      0.0      0.0      return labels
```

We spend >86% in Line 41!!!

Quiz: How can we make this faster?

```
1  def distance(v, w):
2      return math.sqrt(sum((vi - wi) ** 2 for vi, wi in zip(v, w)))
3
4  def assign_labels(data, centers):
5      labels = []
6      for item in data:
7          distances = [distance(item, center) for center in centers]
8          labels.append(distances.index(min(distances)))
9      return labels
```

Numpy to the rescue!

```
def distance(v, w):
    return math.sqrt(sum((vi - wi) ** 2 for vi, wi in zip(v, w)))

def assign_labels(data, centers):
    labels = []
    for item in data:
        distances = [distance(item, center) for center in centers]
        labels.append(distances.index(min(distances)))
    return labels
```

Numpy code:

```
def assign_labels(data, centers):
    """ assign data to closest label (corresponding to centers)."""
    differences = (np.array(data)[:, None, :] - np.array(centers)) ** 2
    distances = np.sqrt(differences.sum(-1))
    return distances.argmin(1)
```

Quick explanation:

```
In [1]: import numpy as np

In [2]: data = np.arange(6).reshape((3,2))

In [3]: centers = np.arange(2).reshape((1,2))

In [4]: data[:,None,:]-centers      # introduces extra dimension in middle
Out[4]:
array([[[0, 0]],

       [[2, 2]],

       [[4, 4]]])

In [5]: np.sqrt(np.sum((data[:,None,:]-centers)**2,axis = 2))
Out[5]:
array([[0.        ],      # is distance of (1,2) and (1,2)
       [2.82842712],      # is distance of (3,4) and (1,2) = sqrt(8)
       [5.65685425]])     # is the distance of (5,6) and (1,2)
```

Updated line profiling 8.5s -> 🙉

```
In [8]: %lprun -f kmeans kmeans(data,n_clusters)
Timer unit: 1e-06 s

Total time: 1.88813 s
File: /Users/alex/Code/Teaching/profiling/kmeans/kmeans_numpyI.py
Function: kmeans at line 28

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    28                                           def kmeans(data, n_clusters):
    29         1          5.0      5.0      0.0      centers = data[-n_clusters:]
    30
    31                                               while True:
    32        53         24.0      0.5      0.0          old_centers = centers
    33        53     334932.0   6319.5     17.7          labels = assign_labels(data, cen
    34        53    1553147.0  29304.7     82.3          centers = update_centers(data, l
    35        53         27.0      0.5      0.0          if centers == old_centers:
    36         1          0.0      0.0      0.0              break
    37
    38         1          0.0      0.0      0.0      return labels
```

Now we spend >80% for updating centers!

Quiz: How can we improve this code?

```
def update_centers(data, labels):
    """ Update centers of mass according to labels """
    n_centers = len((set(labels)))
    n_dims = len(data[0])

    centers = [[0 for i in range(n_dims)] for j in range(n_centers)]
    counts = [0 for j in range(n_centers)]

    for label, item in zip(labels, data):
        counts[label] += 1
        centers[label] = [
            ci + ii for ci, ii in zip(centers[label], item)
        ]  # summing, per label

    return [[x / count for x in center] for center, count in zip(centers, counts)]
```

Numpy to the rescue, again!

```
def update_centers(data, labels):
    """ Update centers of mass according to labels """
    n_centers = len((set(labels)))
    n_dims = len(data[0])
    centers = [[0 for i in range(n_dims)] for j in range(n_centers)]
    counts = [0 for j in range(n_centers)]
    for label, item in zip(labels, data):
        counts[label] += 1
        centers[label] = [
            ci + ii for ci, ii in zip(centers[label], item)
        ]  # summing, per label
    return [[x / count for x in center] for center, count in zip(centers, counts)]
```

Numpy code:

```
def update_centers(data, labels):
    """ Update centers of mass according to labels """
    n_centers = len((set(labels)))
    return np.array([data[labels == i].mean(0) for i in range(n_centers)])
```

Updated line-profiling results... 1.8s --> 🙉

```
In [11]: %lprun -f kmeans kmeans(data,n_clusters)
Timer unit: 1e-06 s

Total time: 0.196476 s
File: /Users/alex/Code/Teaching/profiling/kmeans/kmeans_numpyII.py
Function: kmeans at line 17

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    17                                           def kmeans(data, n_clusters):
    18         1      10147.0  10147.0      5.2      data = np.array(data)
    19         1         10.0     10.0      0.0      centers = data[-n_clusters:]
    20
    21                                               while True:
    22        41         17.0      0.4      0.0          old_centers = centers
    23        41     122246.0   2981.6     62.2          labels = assign_labels(data, cen
    24        41      63848.0   1557.3     32.5          centers = update_centers(data, l
    25
    26        41        207.0      5.0      0.1          if (centers == old_centers).all(
    27         1          1.0      1.0      0.0              break
    28
    29         1          0.0      0.0      0.0      return labels
```

Quiz: What now?

Tip: use specialized functions!

```
from scipy.spatial.distance import cdist

def assign_labels(data, centers):
    """ assign data to closest label (corresponding to centers)."""
    distances = cdist(data, centers)
    return distances.argmin(1)
```

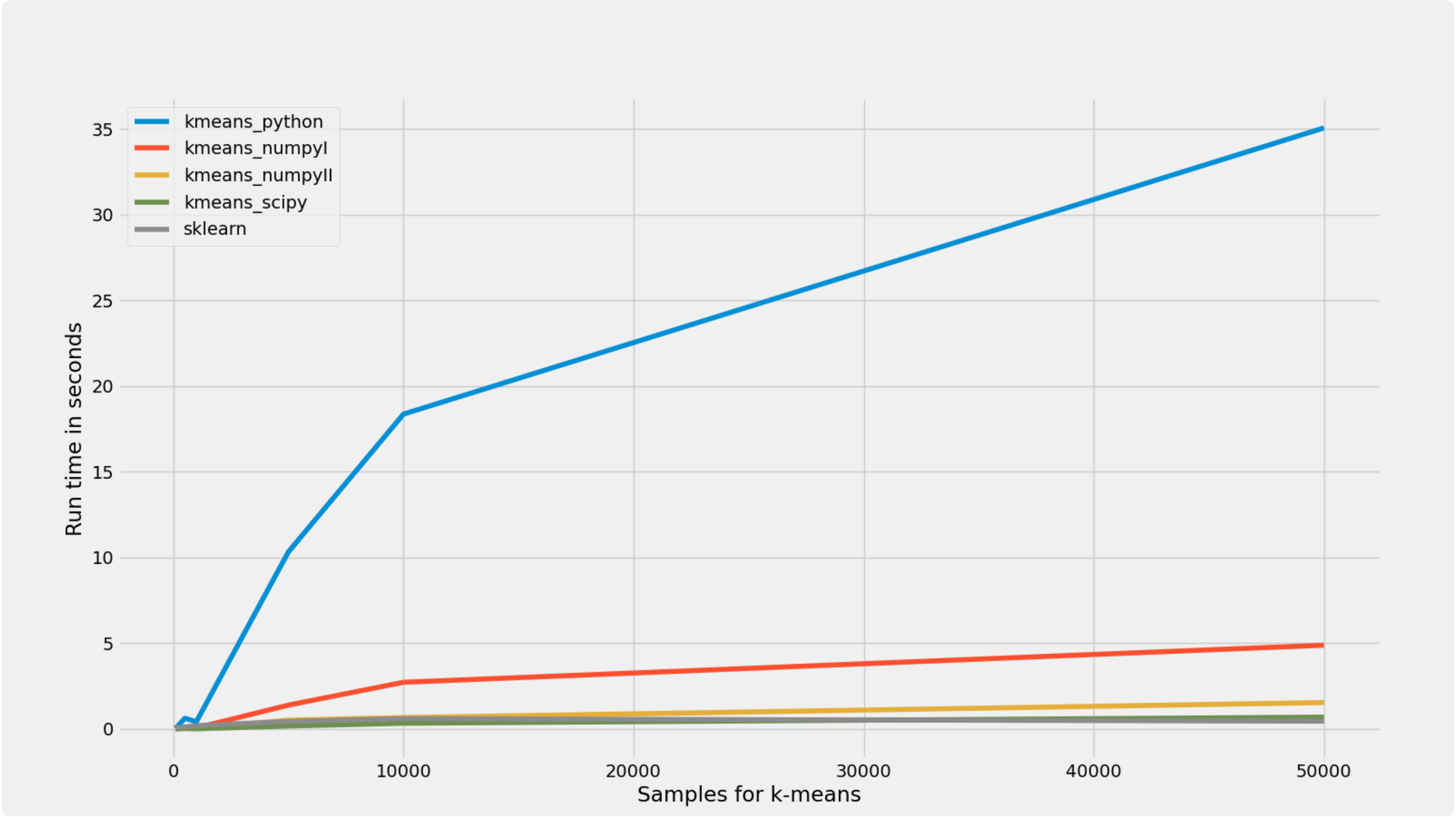
for comparison:

```
def assign_labels(data, centers):
    """ assign data to closest label (corresponding to centers)."""
    differences = (np.array(data)[:, None, :] - np.array(centers)) ** 2
    distances = np.sqrt(differences.sum(-1))
    return distances.argmin(1)
```

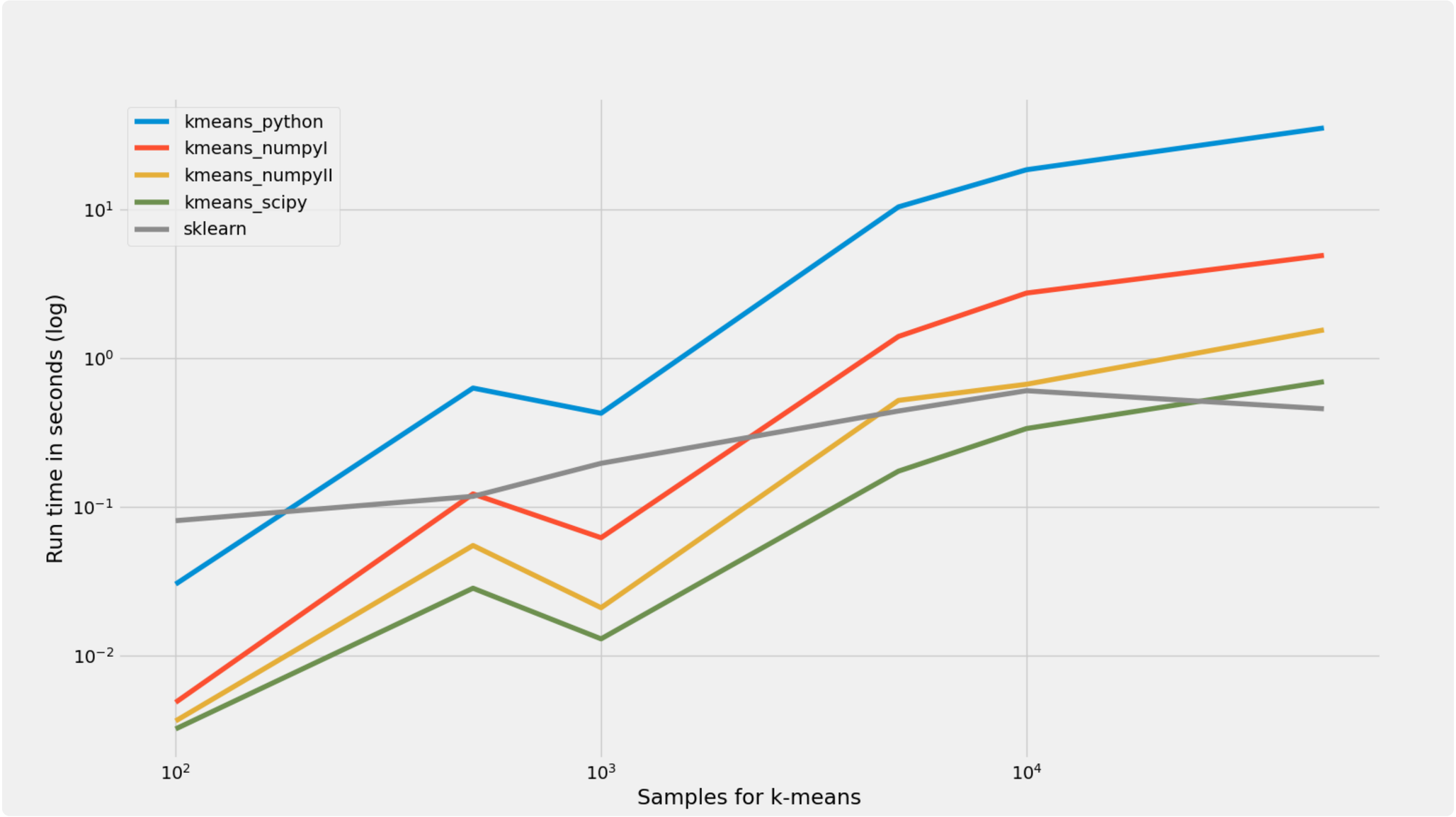
Updated line-profiling results... 0.19s --> 🙉

```
 1   In [13]: %lprun -f kmeans kmeans(data,n_clusters)
Timer unit: 1e-06 s

Total time: 0.092516 s
File: /Users/alex/Code/Teaching/profiling/kmeans/kmeans_scipyI.py
Function: kmeans at line 17

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    17                                           def kmeans(data, n_clusters):
    18         1       8087.0   8087.0      8.7      data = np.array(data)
    19         1         11.0     11.0      0.0      centers = data[-n_clusters:]
    20
    21                                               while True:
    22        36         18.0      0.5      0.0          old_centers = centers
    23        36      23362.0    648.9     25.3          labels = assign_labels(data, centers)
    24        36      60846.0   1690.2     65.8          centers = update_centers(data, labels
    25
    26        36        191.0      5.3      0.2          if (centers == old_centers).all():
    27         1          0.0      0.0      0.0              break
    28
    29         1          1.0      1.0      0.0      return labels
```

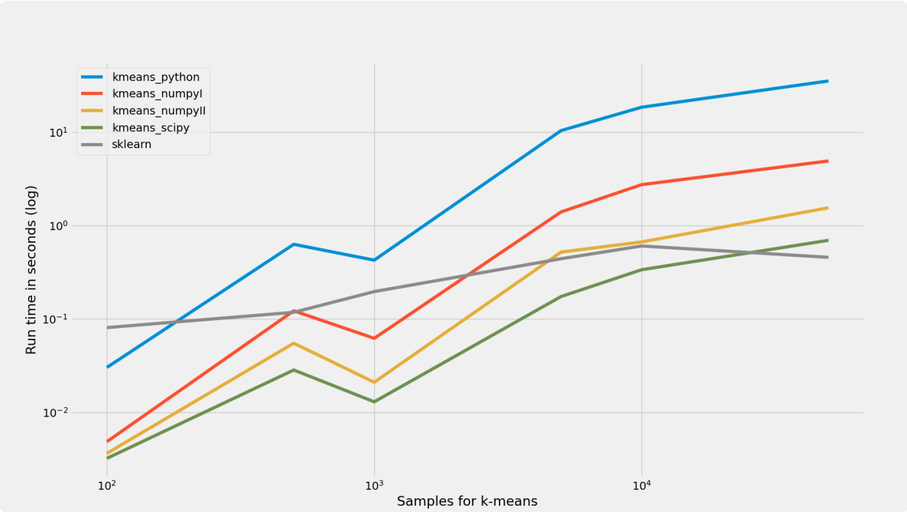
Method comparision



Method comparison



Method comparison



Note:

- the run time depends on the number of samples
- indeed, our implementations monotonically got better for the different variants
- the variant from the sklearn library is faster for large numbers of samples.

Other tools to improve speed (see excercises)

- Detailed guide on Python - C++ interfacing
- Cython python-like Code at C-like speeds; recommended for operations that cannot be done in NumPy, SciPy, etc.
- Numba for just-in-time compilation
- f2py call Fortran code

Questions?

Today's summary

Always profile your code, before you optimize it!

How to profile?

- We discussed profiling tools cProfile, timeit, line_profiler incl. visualization methods
- We learned about one of the core clustering algorithms. It's otherwise not important for this class, but I think it's algorithm you should know

How to optimize?

- Numpy is faster, always try to write vectorized code!
- Use *existing, optimized and specialized functions* (cdist, kmeans from a library, …). This is also much better maintained
- For specialized functions, consider writing them in Fortran, C++ and link
- We learned about memoization and functools: lru_cache

After lunch:

- Monday 13 - 15: exercises working on v6 of your project
- Monday 15:15 - 16: my office hours at SV 2811