Optimization (cont.)

Lecture 14

Dr. Colin Rundel

Method Variants

Method: CG in scipy

Scipy's optimize module implements the conjugate gradient algorithm by Polak and Ribiere, a variant that does not require the Hessian,

- α_k is calculated via a line search along the direction p_k
- We replace the previous definition

$$\beta_{k+1} = \frac{r_{k+1}^T \nabla^2 f(x_k)\, p_k}{p_k^T \nabla^2 f(x_k)\, p_k}$$

with

$$\beta_{k+1}^{PR} = \frac{\nabla f(x_{k+1})\,(\nabla f(x_{k+1}) - \nabla f(x_k))}{\nabla f(x_k)^T \nabla f(x_k)}$$

Method: Newton-CG & BFGS

These are both variants of Newtons method but do not require the Hessian (but can be use by the former if provided).

- The Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm is a quasi-newton which iteratively approximates Hessian

Method: Nelder-Mead

This is a gradient free method that uses a series of simplexes which are used to iteratively bracket the minimum.

Method Summary

SciPy Method	Description	Gradient	Hessian
—	Gradient Descent (naive w/ backtracking)	✓	✗
—	Newton's method (naive w/ backtracking)	✓	✓
—	Conjugate Gradient (naive)	✓	✓
`"CG"`	Nonlinear Conjugate Gradient (Polak and Ribiere variation)	✓	✗
`"Newton-CG"`	Truncated Newton method (Newton w/ CG step direction)	✓	Optional
`"BFGS"`	Broyden, Fletcher, Goldfarb, and Shanno (Quasi-newton method)	Optional	✗
`"L-BFGS-B"`	Limited-memory BFGS (Quasi-newton method)	Optional	✗
`"Nelder-Mead"`	Nelder-Mead simplex reflection method	✗	✗

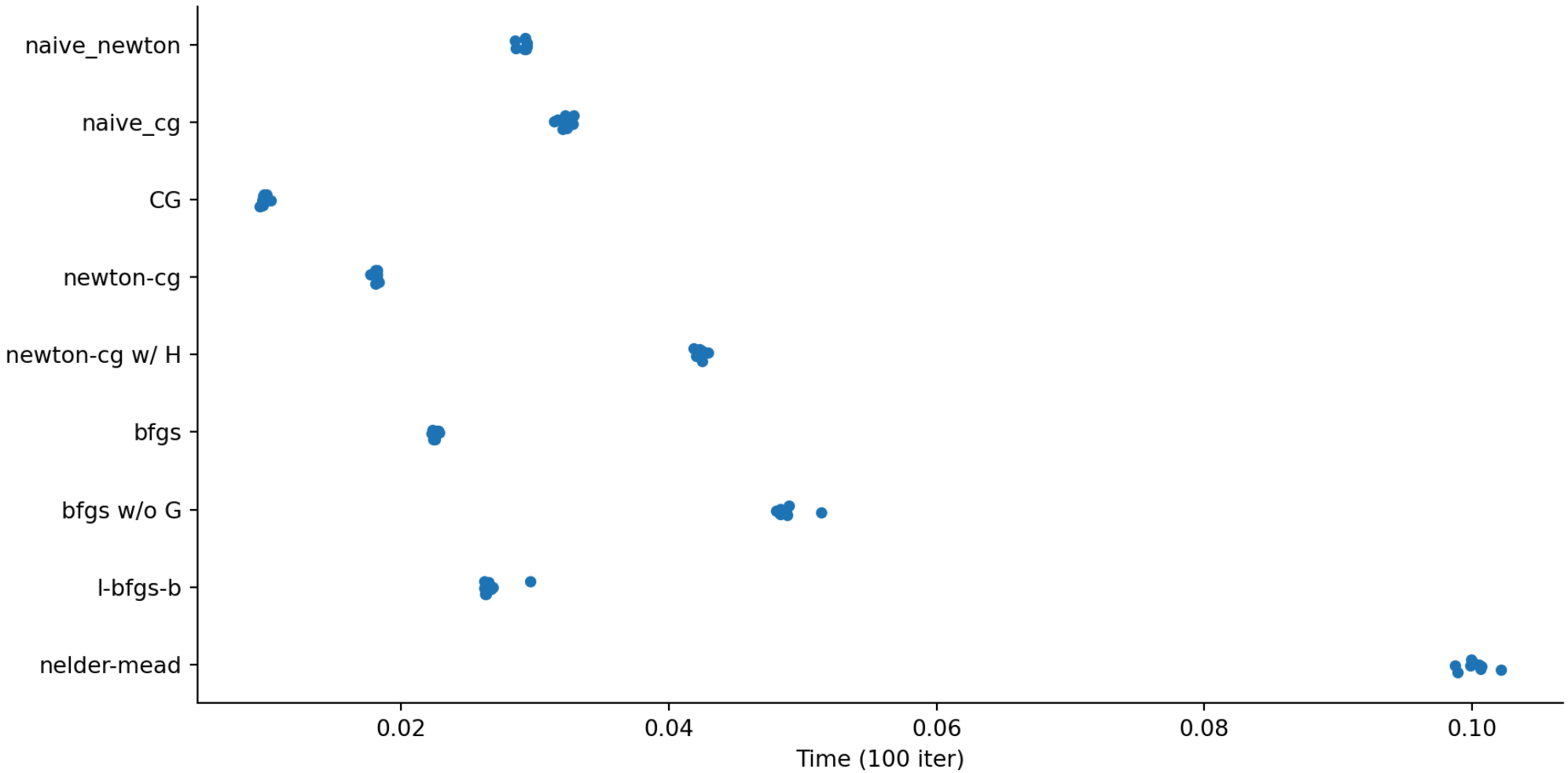
Methods collection

```
def define_methods(x, f, grad, hess, tol=1e-8):
  return {
    "naive_newton":   lambda: newtons_method(x, f, grad, hess, tol=tol),
    "naive_cg":       lambda: conjugate_gradient(x, f, grad, hess, tol=tol),
    "CG":             lambda: optimize.minimize(f, x, jac=grad, method="CG", tol=t
    "newton-cg":      lambda: optimize.minimize(f, x, jac=grad, hess=None, method=
    "newton-cg w/ H": lambda: optimize.minimize(f, x, jac=grad, hess=hess, method=
    "bfgs":           lambda: optimize.minimize(f, x, jac=grad, method="BFGS", tol
    "bfgs w/o G":     lambda: optimize.minimize(f, x, method="BFGS", tol=tol),
    "l-bfgs-b":       lambda: optimize.minimize(f, x, method="L-BFGS-B", tol=tol),
    "nelder-mead":    lambda: optimize.minimize(f, x, method="Nelder-Mead", tol=to
  }
```

Method Timings

```python
x = (1.6, 1.1)
f, grad, hess = mk_quad(0.7)
methods = define_methods(x, f, grad, hess)

df = pd.DataFrame({
  key: timeit.Timer(methods[key]).repeat(10, 100) for key in methods
})

df
```

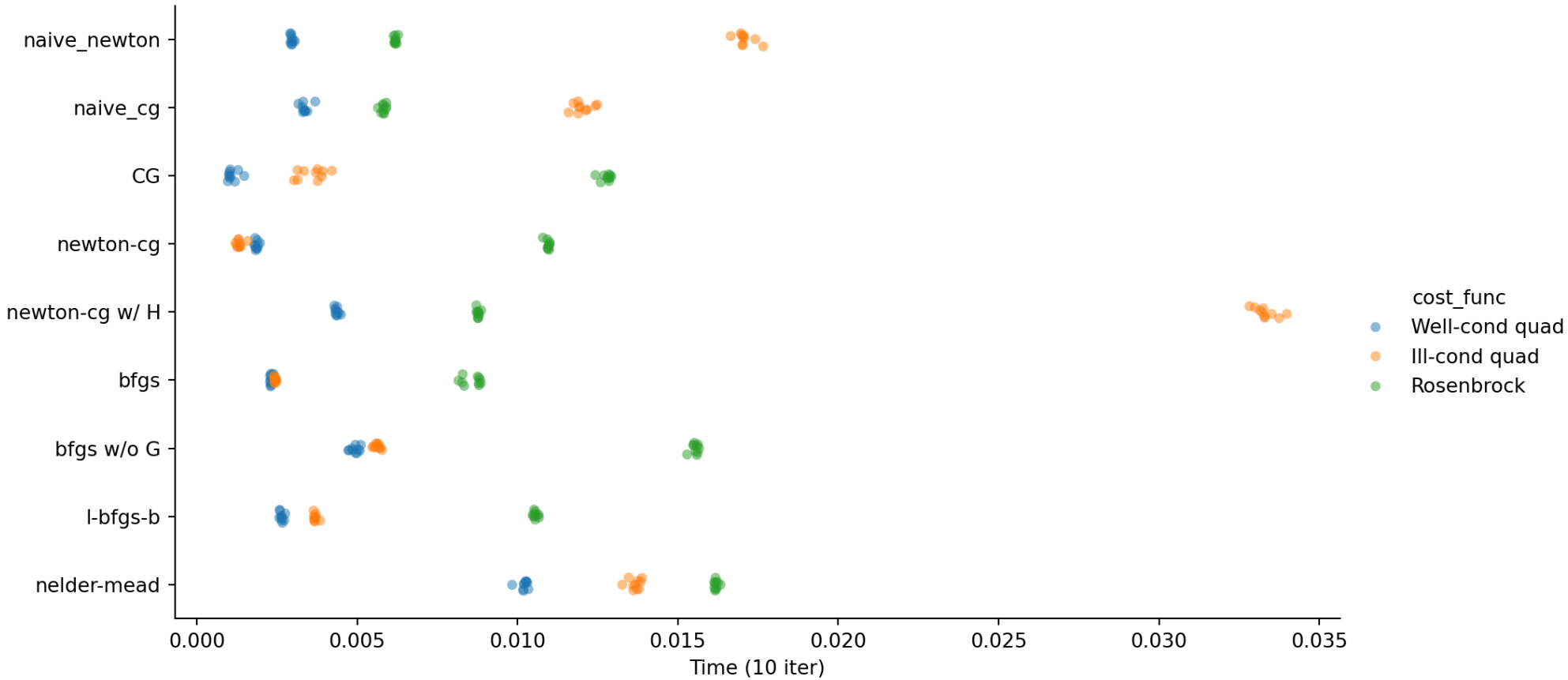
```
   naive_newton  naive_cg        CG  ...  bfgs w/o G  l-bfgs-b  nelder-mead
0      0.029418  0.031432  0.010260  ...    0.048270  0.029618     0.100661
1      0.028514  0.032393  0.009922  ...    0.048328  0.026831     0.099941
2      0.028562  0.032846  0.009725  ...    0.048308  0.026727     0.100675
3      0.029202  0.032913  0.009430  ...    0.048968  0.026292     0.099893
4      0.029238  0.031702  0.009614  ...    0.048761  0.026301     0.100530
5      0.029393  0.032050  0.009654  ...    0.048400  0.026221     0.100207
6      0.029281  0.032417  0.009676  ...    0.048008  0.026354     0.100014
7      0.029246  0.031986  0.009658  ...    0.048347  0.026398     0.098695
8      0.029418  0.032260  0.009675  ...    0.051405  0.026557     0.102137
9      0.029312  0.032251  0.009771  ...    0.048857  0.026232     0.098924

[10 rows x 9 columns]
```


naive_newton
naive_cg
CG
newton-cg
newton-cg w/ H
bfgs
bfgs w/o G
l-bfgs-b
nelder-mead
0.02
0.04
0.06
0.08
0.10
Time (100 iter)

Timings across cost functions

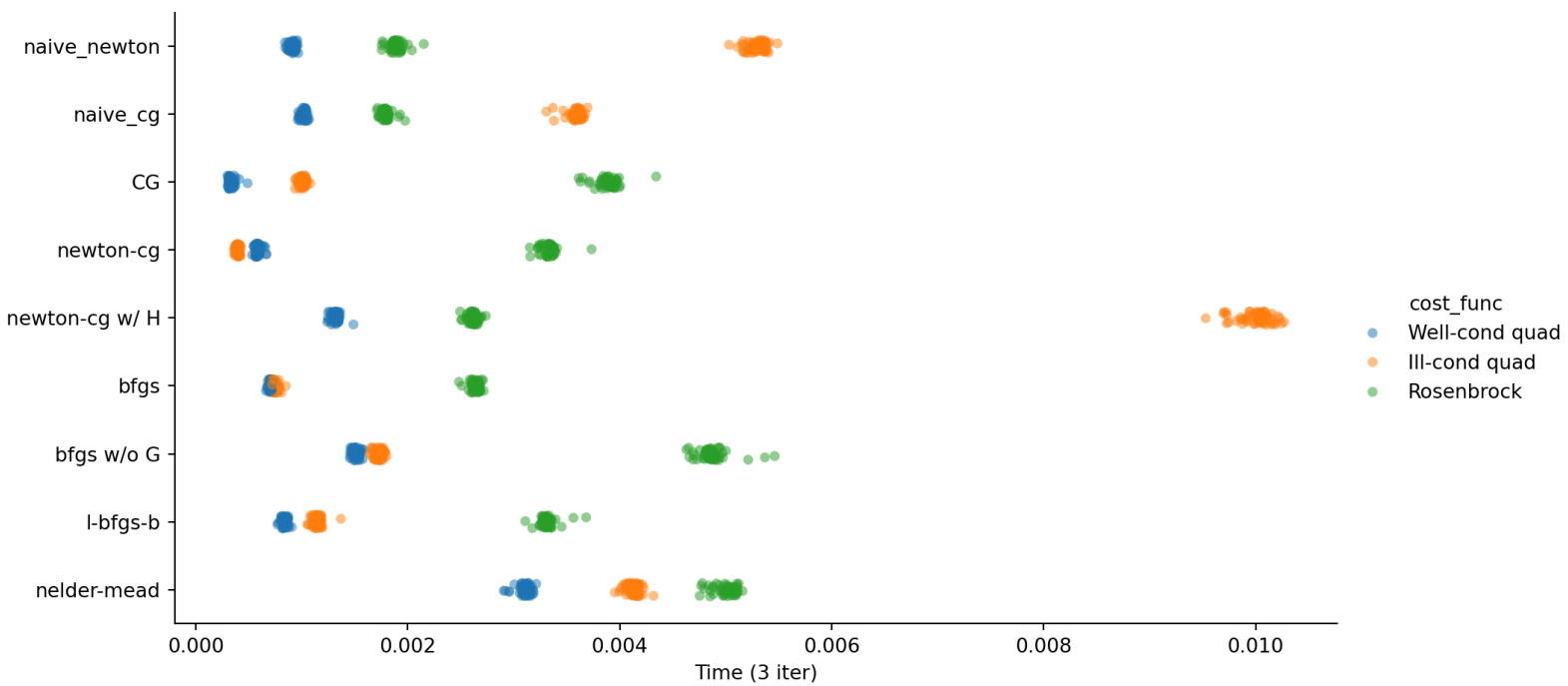
```python
def time_cost_func(n, x, name, cost_func, *a
  x = (1.6, 1.1)
  f, grad, hess = cost_func(*args)
  methods = define_methods(x, f, grad, hess

  return ( pd.DataFrame({
      key: timeit.Timer(
        methods[key]
      ).repeat(n, n)
      for key in methods
    })
    .melt()
    .assign(cost_func = name)
  )

df = pd.concat([
  time_cost_func(10, x, "Well-cond quad", ml
  time_cost_func(10, x, "Ill-cond quad", mk_
  time_cost_func(10, x, "Rosenbrock", mk_ros
])

df
```

```
        variable     value       cost_func
0   naive_newton  0.003061  Well-cond quad
1   naive_newton  0.003014  Well-cond quad
2   naive_newton  0.002951  Well-cond quad
3   naive_newton  0.002920  Well-cond quad
4   naive_newton  0.002978  Well-cond quad
..           ...       ...             ...
85   nelder-mead  0.016167      Rosenbrock
86   nelder-mead  0.016182      Rosenbrock
87   nelder-mead  0.016139      Rosenbrock
88   nelder-mead  0.016248      Rosenbrock
89   nelder-mead  0.016163      Rosenbrock

[270 rows x 3 columns]
```


naive_newton
naive_cg
CG
newton-cg
newton-cg w/ H
bfgs
bfgs w/o G
l-bfgs-b
nelder-mead
0.000
0.005
0.010
0.015
0.020
0.025
0.030
0.035
Time (10 iter)
cost_func
Well-cond quad
Ill-cond quad
Rosenbrock

Random starting locations

```
x0s = np.random.default_rng(seed=1234).uniform(-2,2, (20,2))

df = pd.concat([
  pd.concat([
    time_cost_func(3, x0, "Well-cond quad", mk_quad, 0.7),
    time_cost_func(3, x0, "Ill-cond quad", mk_quad, 0.02),
    time_cost_func(3, x0, "Rosenbrock", mk_rosenbrock)
  ])
  for x0 in x0s
])
```



Profiling - BFGS (cProfile)

```
1  import cProfile
2
3  f, grad, hess = mk_quad(0.7)
4  def run():
5    optimize.minimize(fun = f, x0 = (1.6, 1.1), jac=grad, method="BFGS", tol=1e-11)
6
7  cProfile.run('run()', sort="tottime")
```

```
         1011 function calls in 0.001 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.001    0.001 _optimize.py:1328(_minimize_bfgs)
       58    0.000    0.000    0.000    0.000 {method 'reduce' of 'numpy.ufunc' objects}
       10    0.000    0.000    0.000    0.000 <string>:3(f)
       26    0.000    0.000    0.000    0.000 _optimize.py:194(vecnorm)
        9    0.000    0.000    0.000    0.000 _dcsrch.py:201(__call__)
        1    0.000    0.000    0.001    0.001 _minimize.py:53(minimize)
       36    0.000    0.000    0.000    0.000 fromnumeric.py:69(_wrapreduction)
       20    0.000    0.000    0.000    0.000 numeric.py:2475(array_equal)
       10    0.000    0.000    0.000    0.000 <string>:9(gradient)
        9    0.000    0.000    0.000    0.000 _linesearch.py:100(scalar_search_wolfe1)
       18    0.000    0.000    0.000    0.000 _dcsrch.py:269(_iterate)
        9    0.000    0.000    0.000    0.000 _linesearch.py:86(derphi)
       26    0.000    0.000    0.000    0.000 fromnumeric.py:2338(sum)
```

Profiling - BFGS (pyinstrument)

```
from pyinstrument import Profiler

f, grad, hess = mk_quad(0.7)

profiler = Profiler(interval=0.00001)

profiler.start()
opt = optimize.minimize(fun = f, x0 = (1.6, 1.1), jac=grad, method="BFGS", tol=1e-11)
p = profiler.stop()

profiler.print(show_all=True)
```

Profiling - Nelder-Mead

```
from pyinstrument import Profiler

f, grad, hess = mk_quad(0.7)

profiler = Profiler(interval=0.00001)

profiler.start()
opt = optimize.minimize(fun = f, x0 = (1.6, 1.1), method="Nelder-Mead", tol=1e-11)
p = profiler.stop()

profiler.print(show_all=True)
```

`optimize.minimize()` output

```
1 f, grad, hess = mk_quad(0.7)
```

```
1 optimize.minimize(fun = f, x0 = (1.6,
2                   jac=grad, method="BF
```

```
  message: Optimization terminated
successfully.
  success: True
   status: 0
      fun: 1.2739256453439323e-11
        x: [-5.318e-07 -8.843e-06]
      nit: 6
      jac: [-3.510e-07 -2.860e-06]
 hess_inv: [[ 1.515e+00 -3.438e-03]
            [-3.438e-03  3.035e+00]]
     nfev: 7
     njev: 7
```

```
1 optimize.minimize(fun = f, x0 = (1.6,
2                   jac=grad, hess=hess,
3                   method="Newton-CG")
```

```
 message: Optimization terminated
successfully.
 success: True
  status: 0
     fun: 2.3418652989289317e-12
       x: [ 0.000e+00  3.806e-06]
     nit: 11
     jac: [ 0.000e+00  4.102e-06]
    nfev: 12
    njev: 12
    nhev: 11
```

`optimize.minimize()` output (cont.)

```
optimize.minimize(fun = f, x0 = (1.6,
                  jac=grad, method="CG
```

```
 message: Optimization terminated
successfully.
 success: True
  status: 0
     fun: 1.4450021261144105e-32
       x: [-1.943e-16 -1.110e-16]
     nit: 2
     jac: [-1.282e-16 -3.590e-17]
    nfev: 5
    njev: 5
```

```
optimize.minimize(fun = f, x0 = (1.6,
                  jac=grad, method="Ne
```

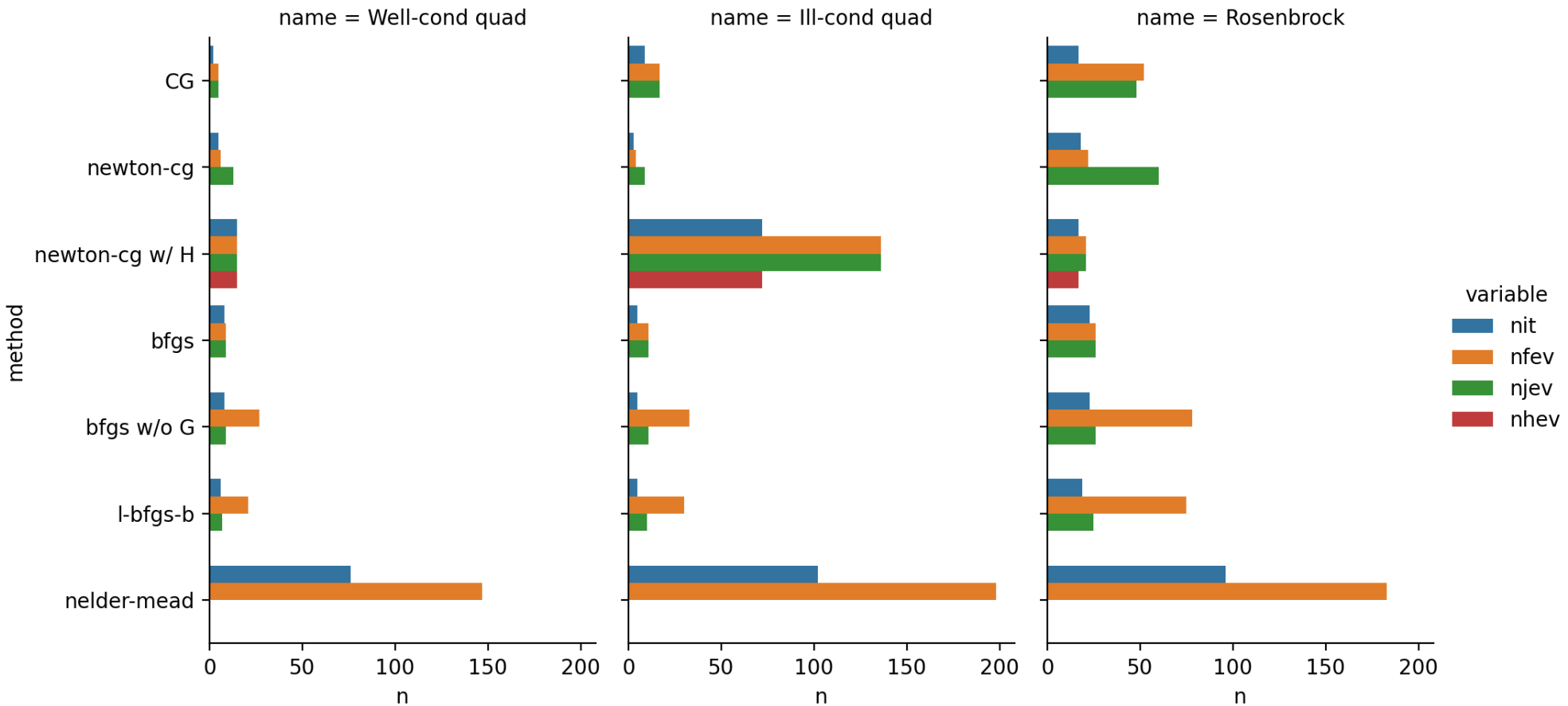
```
       message: Optimization terminated
successfully.
       success: True
        status: 0
           fun: 2.3077013477040082e-10
             x: [ 1.088e-05  3.443e-05]
           nit: 46
          nfev: 89
 final_simplex: (array([[ 1.088e-05,
3.443e-05],
                       [ 1.882e-05,
-3.825e-05],
                       [-3.966e-05,
-3.147e-05]]), array([ 2.308e-10,
3.534e-10,  6.791e-10]))
```

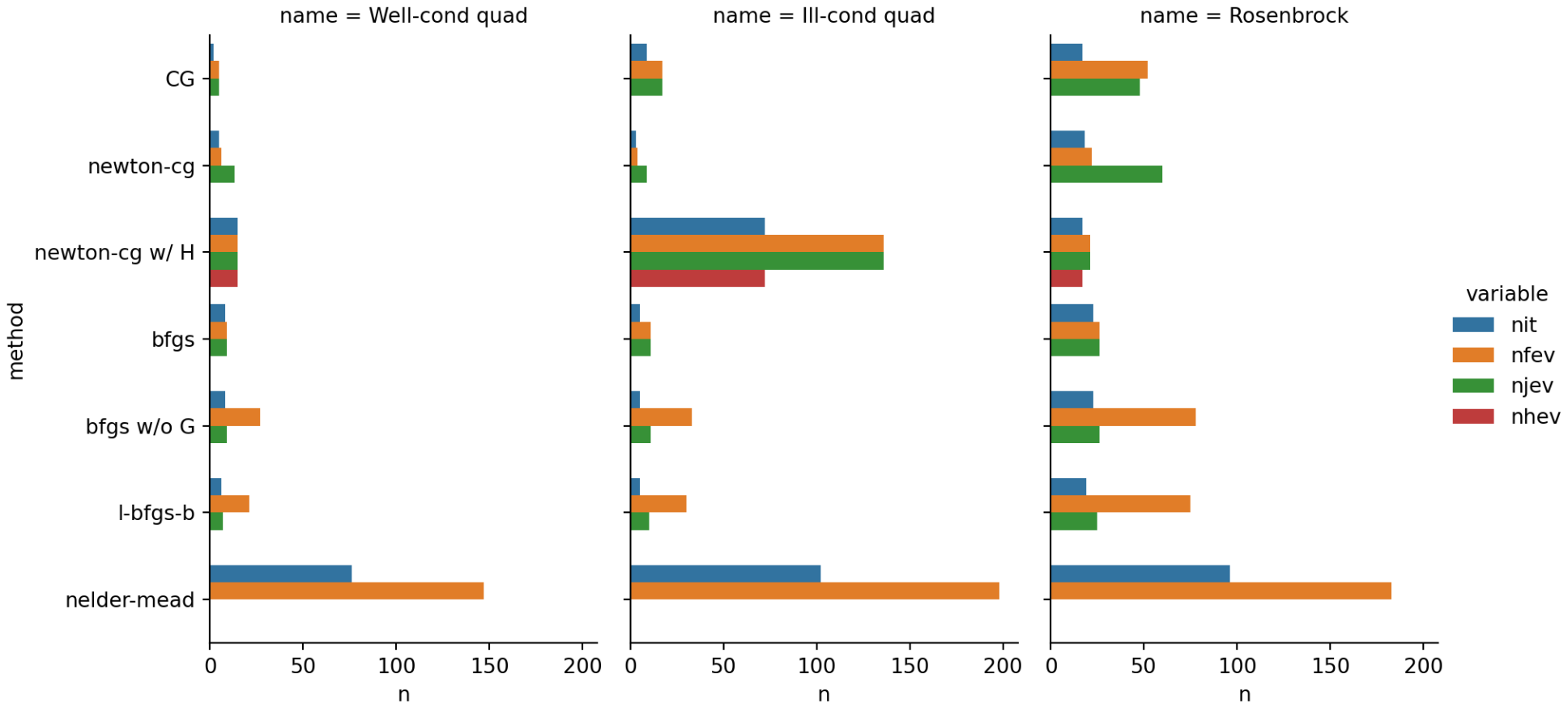
Collect

```python
def run_collect(name, x0, cost_func, *
  f, grad, hess = cost_func(*args)
  methods = define_methods(x0, f, grad

  res = []
  for method in methods:
    if method in skip: continue

    x = methods[method]()

    d = {
      "name":    name,
      "method":  method,
      "nit":     x["nit"],
      "nfev":    x["nfev"],
      "njev":    x.get("njev"),
      "nhev":    x.get("nhev"),
      "success": x["success"]
      #"message": x["message"]
    }
    res.append( pd.DataFrame(d, index=
```

```python
df = pd.concat([
  run_collect(name, (1.6, 1.1), cost_f
  for name, cost_func, arg in zip(
    ("Well-cond quad", "Ill-cond quad"
    (mk_quad, mk_quad, mk_rosenbrock),
    (0.7, 0.02, None)
  )
])

df
```

```
                name          method  nit
nfev  njev  nhev  success
1  Well-cond quad              CG    2
5     5  None     True
1  Well-cond quad       newton-cg    5
6    13     0     True
1  Well-cond quad  newton-cg w/ H   15
15    15    15     True
1  Well-cond quad            bfgs    8
9     9  None     True
1  Well-cond quad      bfgs w/o G    8
27     9  None     True
1  Well-cond quad         l-bfgs-b    6
21     7  None     True
1  Well-cond quad     nelder-mead   76
147  None  None     True
```

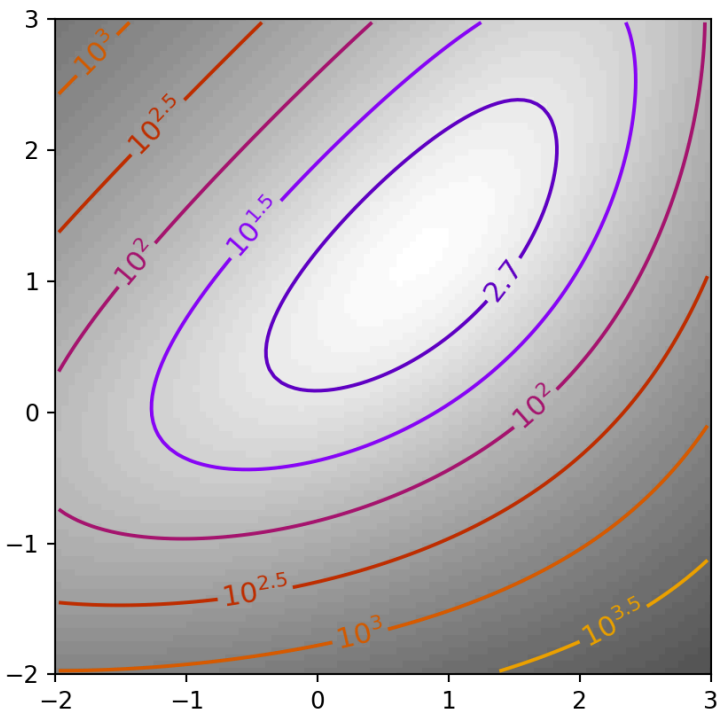

name = Well-cond quad
name = Ill-cond quad
name = Rosenbrock
method
CG
newton-cg
newton-cg w/ H
bfgs
bfgs w/o G
l-bfgs-b
nelder-mead
0
50
100
150
200
n
variable
nit
nfev
njev
nhev


name = Well-cond quad
name = Ill-cond quad
name = Rosenbrock
method
CG
newton-cg
newton-cg w/ H
bfgs
bfgs w/o G
l-bfgs-b
nelder-mead
0
50
100
150
200
n
variable
nit
nfev
njev
nhev

Exercise 1

Try minimizing the following function using different optimization methods starting from $x_0 = [0, 0]$, which method(s) appear to work best?

$$f(x) = \exp(x_1 - 1) + \exp(-x_2 + 1) + (x_1 - x_2)^2$$



MVN Example

MVN density cost function

For an n-dimensional multivariate normal we define the $n \times 1$ vectors x and μ and the $n \times n$ covariance matrix Σ,

$$f(x) = \det(2\pi\Sigma)^{-1/2} \exp\left[-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\right]$$

$$\nabla f(x) = -f(x)\Sigma^{-1}(x-\mu)$$

$$\nabla^2 f(x) = f(x)\left(\Sigma^{-1}(x-\mu)(x-\mu)^T\Sigma^{-1} - \Sigma^{-1}\right)$$

Our goal will be to find the mode (maximum) of this density.

```python
def mk_mvn(mu, Sigma):
  Sigma_inv = np.linalg.inv(Sigma)
  norm_const = 1 / (np.sqrt(np.linalg.
  # Returns the negative density (sinc
  def f(x):
    x_m = x - mu
    return -(norm_const *
      np.exp( -0.5 * (x_m.T @ Sigma_in
  def grad(x):
    return (-f(x) * Sigma_inv @ (x - m
  def hess(x):
    n = len(x)
    x_m = x - mu
    return f(x) * ((Sigma_inv @ x_m).r
  return f, grad, hess
```

Gradient checking

One of the most common issues when implementing an optimizer is to get the gradient calculation wrong which can produce problematic results. It is possible to numerically check the gradient function by comparing results between the gradient function and finite differences from the objective function via `optimize.check_grad()`.

```
# 2d
f, grad, hess = mk_mvn(np.zeros(2), np
optimize.check_grad(f, grad, np.zeros(
```

```
np.float64(2.634178031930877e-09)
```

```
optimize.check_grad(f, grad, np.ones(2
```

```
np.float64(5.2132381447350 62e-10)
```

```
# 5d
f, grad, hess = mk_mvn(np.zeros(5), np
optimize.check_grad(f, grad, np.zeros(
```

```
np.float64(2.6031257322754127e-10)
```

```
optimize.check_grad(f, grad, np.ones(5
```

```
np.float64(1.725679820308689e-11)
```

```
# 10d
f, grad, hess = mk_mvn(np.zeros(10), n
optimize.check_grad(f, grad, np.zeros(
```

```
np.float64(2.8760747774580336e-12)
```

```
optimize.check_grad(f, grad, np.ones(1
```

```
np.float64(2.850398669793798e-14)
```

```
# 20d
f, grad, hess = mk_mvn(np.zeros(20), n
optimize.check_grad(f, grad, np.zeros(
```

```
np.float64(4.965068306494546e-16)
```

```
optimize.check_grad(f, grad, np.ones(2
```

```
np.float64(1.0342002372572841e-20)
```

Gradient checking (wrong gradient)

```
wrong_grad = lambda x: 2*grad(x)
```

```
# 2d
f, grad, hess = mk_mvn(np.zeros(2), np
optimize.check_grad(f, wrong_grad, [0,
```

```
np.float64(2.634178031930877e-09)
```

```
# 5d
f, grad, hess = mk_mvn(np.zeros(5), np
optimize.check_grad(f, wrong_grad, np.
```

```
np.float64(2.6031257322754127e-10)
```

```
optimize.check_grad(f, wrong_grad, [1,
```

```
np.float64(0.08280196633767578)
```

```
optimize.check_grad(f, wrong_grad, np.
```

```
np.float64(0.0018548087267515347)
```

Hessian checking

Note since the gradient of the gradient / jacobian is the hessian we can use this function to check our implementation of the hessian as well, just use `grad()` as `func` and `hess()` as `grad`.

```
1 # 2d
2 f, grad, hess = mk_mvn(np.zeros(2), np
3 optimize.check_grad(grad, hess, [0,0])
```

```
np.float64(3.9252311467094 38e-17)
```

```
1 optimize.check_grad(grad, hess, [1,1])
```

```
np.float64(8.399162985270666e-10)
```

```
1 # 5d
2 f, grad, hess = mk_mvn(np.zeros(5), np
3 optimize.check_grad(grad, hess, np.zer
```

```
np.float64(3.878959614448864e-18)
```

```
1 optimize.check_grad(grad, hess, np.one
```

```
np.float64(3.8156075963144067e-11)
```

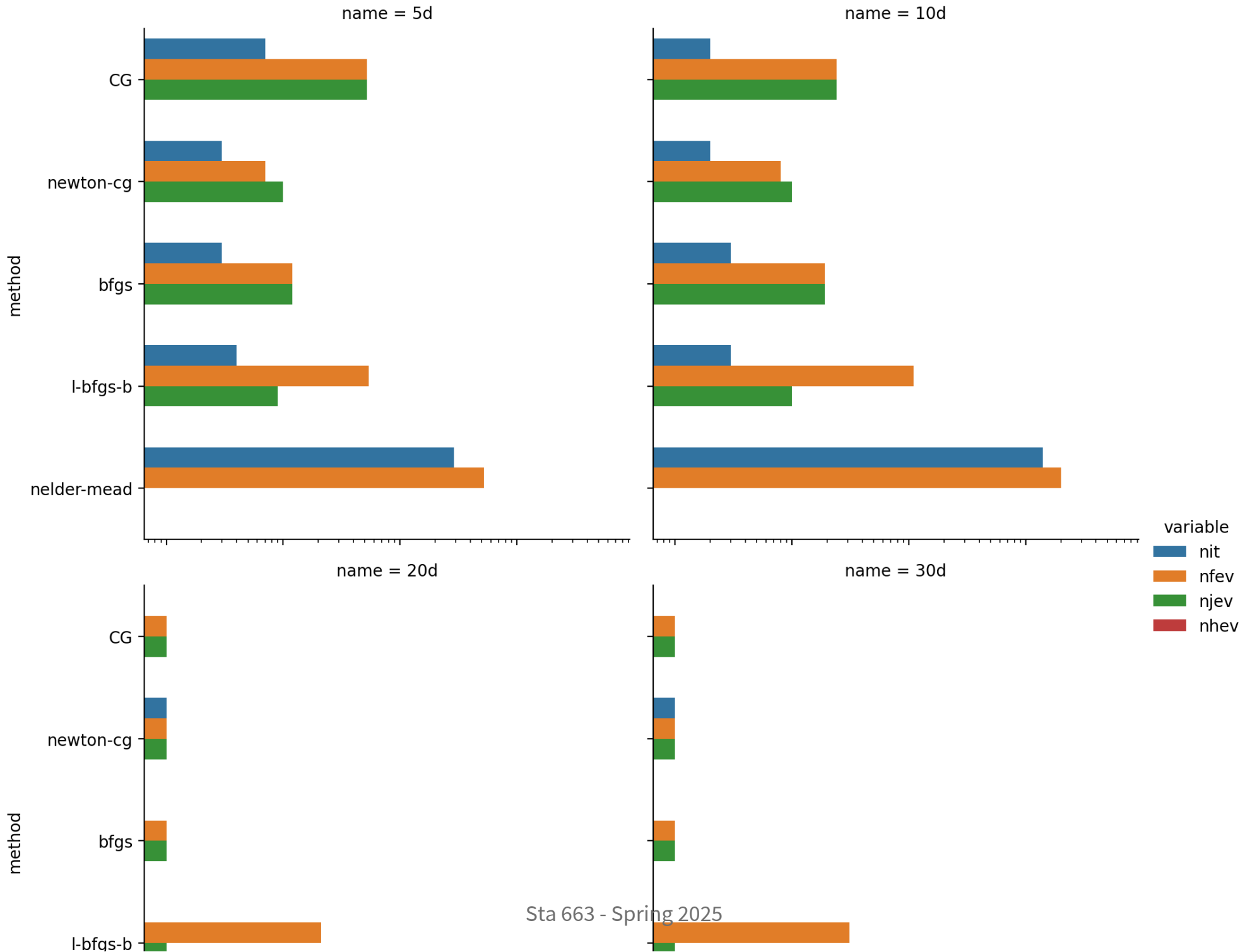
Unit MVNs

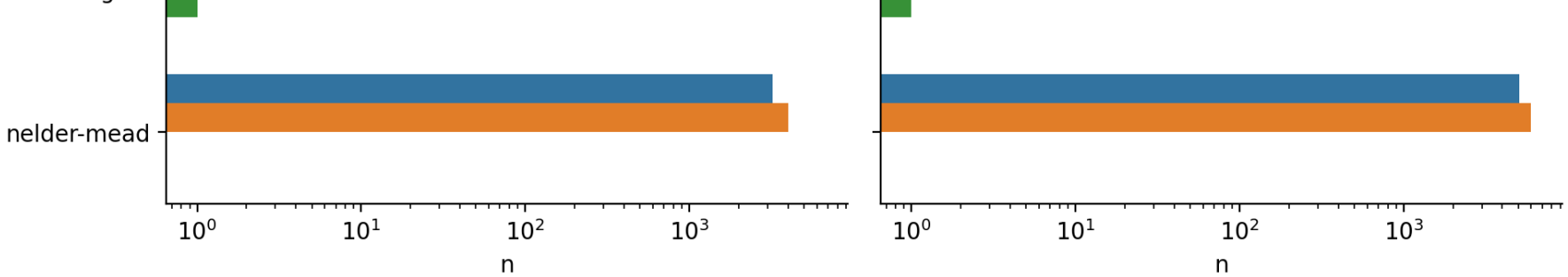
```
df = pd.concat([
  run_collect(
    name, np.ones(n), mk_mvn,
    np.zeros(n), np.eye(n),
    tol=1e-10,
    skip=['naive_newton', 'naive_cg',
  )
  for name, n in zip(
    ("5d", "10d", "20d", "30d"),
    (5, 10, 20, 30)
  )
])
```

```
df
```

```
  name       method   nit  nfev  njev
nhev  success
1   5d           CG     7    52    52
None     True
1   5d    newton-cg     3     7    10
0      True
1   5d         bfgs     3    12    12
None     True
1   5d     l-bfgs-b     4    54     9
None     True
1   5d  nelder-mead   290   523  None
None     True
1  10d           CG     2    24    24
None     True
1  10d    newton-cg     2     8    10
0      True
```

Performance (Unit MVNs)


name = 5d
name = 10d
name = 20d
name = 30d
method
CG
newton-cg
bfgs
l-bfgs-b
nelder-mead
variable
nit
nfev
njev
nhev

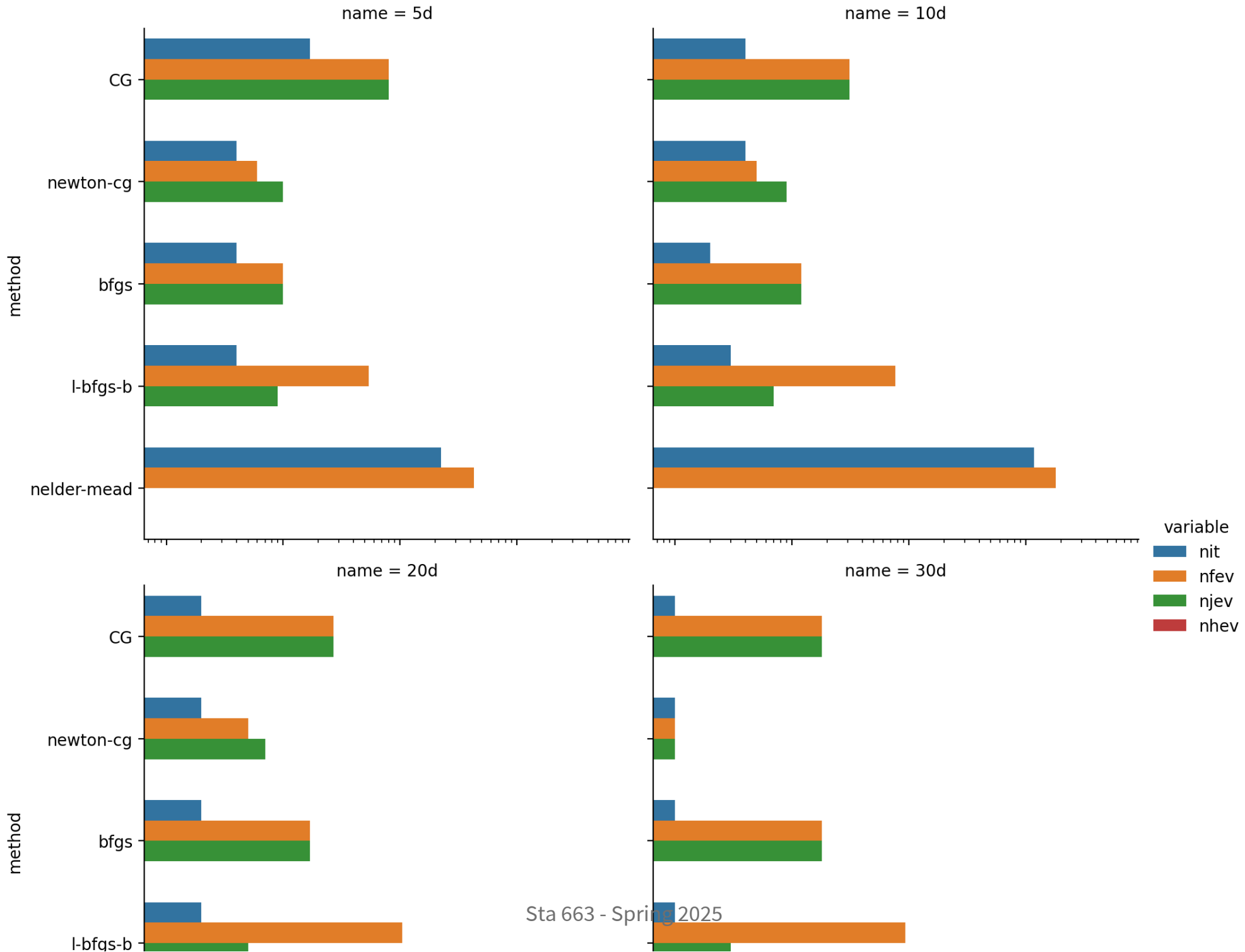

nelder-mead
10^0
10^1
10^2
10^3
n
10^0
10^1
10^2
10^3
n

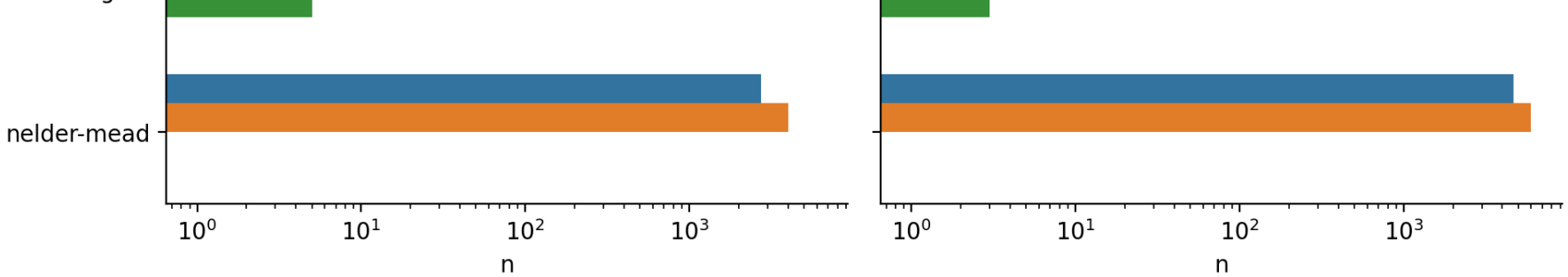
Adding correlation

```python
def build_Sigma(n, corr=0.5):
  S = np.full((n,n), corr)
  np.fill_diagonal(S, 1)
  return S

df = pd.concat([
  run_collect(
    name, np.ones(n), mk_mvn,
    np.zeros(n), build_Sigma(n),
    tol=1e-10,
    skip=['naive_newton', 'naive_cg',
  )
  for name, n in zip(
    ("5d", "10d", "20d", "30d"),
    (5, 10, 20, 30)
  )
])
```

```python
df
```

```
  name       method   nit  nfev  njev
nhev  success
1   5d           CG    17    80    80
None     True
1   5d    newton-cg     4     6    10
0      True
1   5d         bfgs     4    10    10
None     True
1   5d      l-bfgs-b     4    54     9
None     True
1   5d  nelder-mead   224   427  None
None     True
1  10d           CG     4    31    31
None     True
1  10d    newton-cg     4     5     9
0      True
```


name = 5d
name = 10d
name = 20d
name = 30d
method
CG
newton-cg
bfgs
l-bfgs-b
nelder-mead
variable
nit
nfev
njev
nhev


nelder-mead
10^0
10^1
10^2
10^3
n
10^0
10^1
10^2
10^3
n

What's going on? (good)

```
n = 5
f, grad, hess = mk_mvn(np.zeros(n), build_Sigma(n))
```

```
optimize.minimize(f, np.ones(n), jac=g
                  method="CG", tol=1e-
```

```
 message: Optimization terminated
successfully.
 success: True
  status: 0
     fun: -0.02333725077729210 3
       x: [ 1.082e-07  1.059e-07  1.076e-
07  1.088e-07  1.077e-07]
     nit: 14
     jac: [ 8.637e-10  7.556e-10  8.374e-
10  8.917e-10  8.381e-10]
    nfev: 67
    njev: 67
```

```
optimize.minimize(f, np.ones(n), jac=g
                  method="CG", tol=1e-
```

```
 message: Optimization terminated
successfully.
 success: True
  status: 0
     fun: -0.02333725077729232 8
       x: [ 9.586e-10  2.948e-10  7.975e-
10  1.131e-09  8.016e-10]
     nit: 17
     jac: [ 1.376e-11 -1.723e-11  6.238e-
12  2.179e-11  6.430e-12]
    nfev: 80
    njev: 80
```

```
optimize.minimize(f, np.ones(n), jac=g
                  method="CG", tol=1e-
```

```
 message: Optimization terminated
successfully.
 success: True
  status: 0
     fun: -0.023337250777292328
       x: [ 6.697e-10  6.565e-10  6.665e-
10  6.731e-10  6.666e-10]
     nit: 18
     jac: [ 5.335e-12  4.720e-12  5.186e-
12  5.494e-12  5.189e-12]
    nfev: 83
    njev: 83
```

What's going on? (okay)

```
n = 20
f, grad, hess = mk_mvn(np.zeros(n), build_Sigma(n))
```

```
optimize.minimize(f, np.ones(n), jac=g
                  method="CG", tol=1e-
```

```
 message: Optimization terminated
successfully.
 success: True
  status: 0
     fun: -2.330191334018497e-06
       x: [-3.221e-04 -3.221e-04 ...
-3.221e-04 -3.221e-04]
     nit: 2
     jac: [-7.148e-11 -7.148e-11 ...
-7.148e-11 -7.148e-11]
    nfev: 27
    njev: 27
```

```
optimize.minimize(f, np.ones(n), jac=g
                  method="CG", tol=1e-
```

```
 message: Optimization terminated
successfully.
 success: True
  status: 0
     fun: -2.330191334018497e-06
       x: [-3.221e-04 -3.221e-04 ...
-3.221e-04 -3.221e-04]
     nit: 2
     jac: [-7.148e-11 -7.148e-11 ...
-7.148e-11 -7.148e-11]
    nfev: 27
    njev: 27
```

```
optimize.minimize(f, np.ones(n), jac=g
                  method="CG", tol=1e-
```

```
 message: Optimization terminated
successfully.
 success: True
  status: 0
     fun: -2.3301915597315495e-06
       x: [-4.506e-05 -4.506e-05 ...
-4.506e-05 -4.506e-05]
     nit: 2884
     jac: [-9.999e-12 -9.999e-12 ...
-9.999e-12 -9.999e-12]
    nfev: 66313
    njev: 66313
```

What's going on? (bad)

```
n = 30
f, grad, hess = mk_mvn(np.zeros(n), build_Sigma(n))
```

```
optimize.minimize(f, np.ones(n), jac=g
                  method="CG", tol=1e-
```

```
 message: Optimization terminated
successfully.
 success: True
  status: 0
     fun: -2.381146302597316e-09
       x: [ 1.000e+00  1.000e+00 ...
1.000e+00  1.000e+00]
     nit: 0
     jac: [ 1.536e-10  1.536e-10 ...
1.536e-10  1.536e-10]
    nfev: 1
    njev: 1
```

```
optimize.minimize(f, np.ones(n), jac=g
                  method="CG", tol=1e-
```

```
 message: Optimization terminated
successfully.
 success: True
  status: 0
     fun: -6.180056227752818e-09
       x: [ 1.203e-01  1.203e-01 ...
1.203e-01  1.203e-01]
     nit: 1
     jac: [ 4.795e-11  4.795e-11 ...
4.795e-11  4.795e-11]
    nfev: 18
    njev: 18
```

```
optimize.minimize(f, np.ones(n), jac=g
                  method="CG", tol=1e-
```

```
 message: Optimization terminated
successfully.
 success: True
  status: 0
     fun: -6.26701117075865e-09
       x: [-5.021e-03 -5.021e-03 ...
-5.021e-03 -5.021e-03]
     nit: 2
     jac: [-2.030e-12 -2.030e-12 ...
-2.030e-12 -2.030e-12]
    nfev: 35
    njev: 35
```

Options (bfgs)

```
1 optimize.show_options(solver="minimize", method="bfgs")
```

```
Minimization of scalar function of one or more variables using the
BFGS algorithm.

Options
-------
disp : bool
    Set to True to print convergence messages.
maxiter : int
    Maximum number of iterations to perform.
gtol : float
    Terminate successfully if gradient norm is less than `gtol`.
norm : float
    Order of norm (Inf is max, -Inf is min).
eps : float or ndarray
    If `jac is None` the absolute step size used for numerical
    approximation of the jacobian via forward differences.
```

Options (Nelder-Mead)

```
1 optimize.show_options(solver="minimize", method="Nelder-Mead")
```

```
Minimization of scalar function of one or more variables using the
Nelder-Mead algorithm.

Options
-------
disp : bool
    Set to True to print convergence messages.
maxiter, maxfev : int
    Maximum allowed number of iterations and function evaluations.
    Will default to ``N*200``, where ``N`` is the number of
    variables, if neither `maxiter` or `maxfev` is set. If both
    `maxiter` and `maxfev` are set, minimization will stop at the
    first reached.
return_all : bool, optional
    Set to True to return a list of the best solution at each of the
    iterations
```

SciPy implementation

The following code comes from SciPy's `minimize()` implementation:

```
if tol is not None:
  options = dict(options)
  if meth == 'nelder-mead':
      options.setdefault('xatol', tol)
      options.setdefault('fatol', tol)
  if meth in ('newton-cg', 'powell', 'tnc'):
      options.setdefault('xtol', tol)
  if meth in ('powell', 'l-bfgs-b', 'tnc', 'slsqp'):
      options.setdefault('ftol', tol)
  if meth in ('bfgs', 'cg', 'l-bfgs-b', 'tnc', 'dogleg',
              'trust-ncg', 'trust-exact', 'trust-krylov'):
      options.setdefault('gtol', tol)
  if meth in ('cobyla', '_custom'):
      options.setdefault('tol', tol)
  if meth == 'trust-constr':
      options.setdefault('xtol', tol)
      options.setdefault('gtol', tol)
      options.setdefault('barrier_tol', tol)
```

Some general advice

- Having access to the gradient is almost always helpful / necessary
- Having access to the hessian can be helpful, but usually does not significantly improve things
- The curse of dimensionality is real
  - Be careful with `tol` - it means different things for different methods
- In general, **BFGS** or **L-BFGS** should be a first choice for most problems (either well- or ill-conditioned)
  - **CG** can perform better for well-conditioned problems with cheap function evaluations

Maximum Likelihood example

Normal MLE

Minimizing

```
1 optimize.minimize(mle_norm, x0=[0,1], method="bfgs")
```

```
  message: Desired error not necessarily achieved due to
precision loss.
  success: False
   status: 2
      fun: nan
        x: [-1.436e+04 -3.533e+03]
      nit: 2
      jac: [       nan        nan]
 hess_inv: [[ 9.443e-01  2.340e-01]
            [ 2.340e-01  5.905e-02]]
     nfev: 339
     njev: 113
```

Adding constraints

```
def mle_norm2(θ):
  if θ[1] <= 0:
    return np.inf
  else:
    return -np.sum(norm.logpdf(x, loc=
```

```
optimize.minimize(mle_norm2, x0=[0,1],
```

```
  message: Optimization terminated
successfully.
  success: True
   status: 0
      fun: 163.77575977255518
        x: [-3.156e+00  1.245e+00]
      nit: 9
      jac: [ 0.000e+00  0.000e+00]
 hess_inv: [[ 1.475e-02 -1.179e-04]
            [-1.179e-04  7.723e-03]]
     nfev: 43
     njev: 14
```

Specifying Bounds

It is also possible to specify bounds via `bounds` but this is only available for certain optimization methods.

```
optimize.minimize(
  mle_norm, x0=[0,1], method="l-bfgs-b",
  bounds = [(-1e16, 1e16), (1e-16, 1e16)]
)
```

```
  message: CONVERGENCE: RELATIVE REDUCTION OF F <= FACTR*EPSMCH
  success: True
   status: 0
      fun: 163.77575977287245
        x: [-3.156e+00  1.245e+00]
      nit: 10
      jac: [ 2.046e-04  0.000e+00]
     nfev: 69
     njev: 23
 hess_inv: <2x2 LbfgsInvHessProduct with dtype=float64>
```

Exercise 2

Using `optimize.minimize()` recover the shape and scale parameters for these data using MLE.

```
1  from scipy.stats import gamma
2
3  g = gamma(a=2.0, scale=2.0)
4  x = g.rvs(size=100, random_state=1234)
5  x.round(2)
```

```
array([ 4.7 ,  1.11,  1.8 ,  6.19,  3.37,  0.25,  6.45,  0.36,  4.49,
        4.14,  2.84,  1.91,  8.03,  2.26,  2.88,  6.88,  6.84,  6.83,
        6.1 ,  3.03,  3.67,  2.57,  3.53,  2.07,  4.01,  1.51,  5.69,
        3.92,  6.01,  0.82,  2.11,  2.97,  5.02,  9.13,  4.19,  2.82,
       11.81,  1.17,  1.69,  4.67,  1.47, 11.67,  5.25,  3.44,  8.04,
        3.74,  5.73,  6.58,  3.54,  2.4 ,  1.32,  2.04,  2.52,  4.89,
        4.14,  5.02,  4.75,  8.24,  7.6 ,  1.  ,  6.14,  0.58,  2.83,
        2.88,  5.42,  0.5 ,  3.46,  4.46,  1.86,  4.59,  2.24,  2.62,
        3.99,  3.74,  5.27,  1.42,  0.56,  7.54,  5.5 ,  1.58,  5.49,
        6.57,  4.79,  5.84,  8.21,  1.66,  1.53,  4.27,  2.57,  1.48,
        5.23,  3.84,  3.15,  2.1 ,  3.71,  2.79,  0.86,  8.52,  4.36,
        3.3 ])
```