Numerical optimization (cont.)

Lecture 13

Dr. Colin Rundel

Method Summary

SciPy Method	Description	Gradient	Hessian
—	Gradient Descent (naive)	✓	✗
—	Newton's method (naive)	✓	✓
—	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(x0, f, grad, hess, tol=1e-8):
  return {
    "naive_newton":   lambda: newtons_method(x0, f, grad, hess, tol=tol),
    "naive_cg":       lambda: conjugate_gradient(x0, f, grad, hess, tol=tol),
    "CG":             lambda: optimize.minimize(f, x0, jac=grad, method="CG", tol=tol),
    "newton-cg":      lambda: optimize.minimize(f, x0, jac=grad, hess=None, method="Newton-CG", tol=tol
    "newton-cg w/ H": lambda: optimize.minimize(f, x0, jac=grad, hess=hess, method="Newton-CG", tol=tol
    "bfgs":           lambda: optimize.minimize(f, x0, jac=grad, method="BFGS", tol=tol),
    "bfgs w/o G":     lambda: optimize.minimize(f, x0, method="BFGS", tol=tol),
    "l-bfgs-b":       lambda: optimize.minimize(f, x0, method="L-BFGS-B", tol=tol),
    "nelder-mead":    lambda: optimize.minimize(f, x0, method="Nelder-Mead", tol=tol)
  }
```

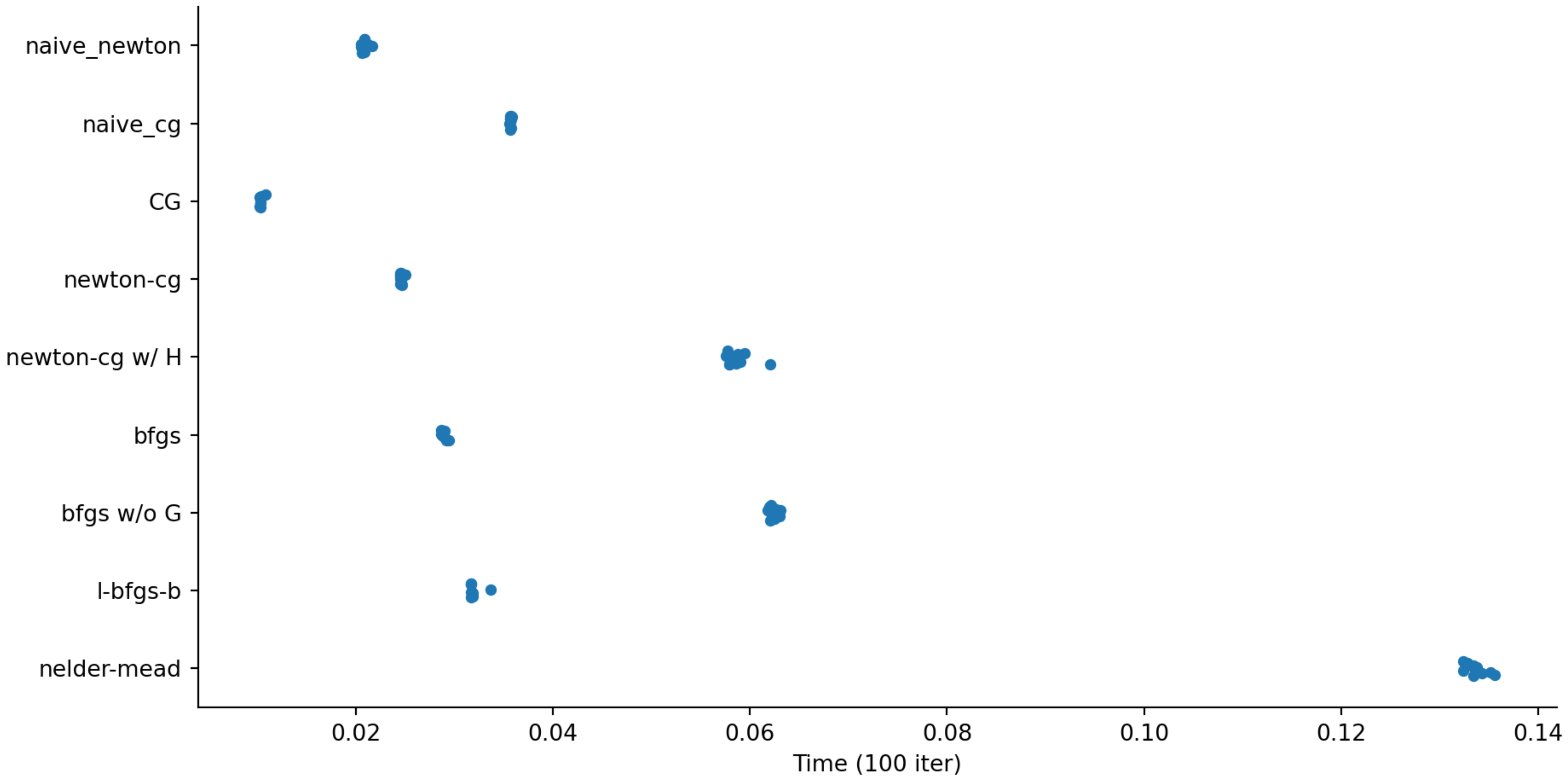
Method Timings

```
x0 = (1.6, 1.1)
f, grad, hess = mk_quad(0.7)
methods = define_methods(x0, f, grad, hess)

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

df
```

Method Timings

	naive_newton	naive_cg	CG	...	bfgs w/o G	l-bfgs-b	nelder-mead
0	0.021679	0.035674	0.010843	...	0.062650	0.033693	0.135608
1	0.020921	0.035565	0.010405	...	0.061980	0.031872	0.133381
2	0.020872	0.035720	0.010369	...	0.061971	0.031723	0.134283
3	0.020858	0.035806	0.010322	...	0.062055	0.031755	0.135175
4	0.020830	0.035808	0.010310	...	0.061845	0.031693	0.133749
5	0.020618	0.035655	0.010298	...	0.062489	0.031697	0.132346
6	0.020579	0.035814	0.010262	...	0.062178	0.031707	0.132381
7	0.020588	0.035658	0.010342	...	0.062977	0.031684	0.133308
8	0.020538	0.035646	0.010268	...	0.063093	0.031823	0.132817
9	0.021264	0.035725	0.010300	...	0.061952	0.031866	0.133438

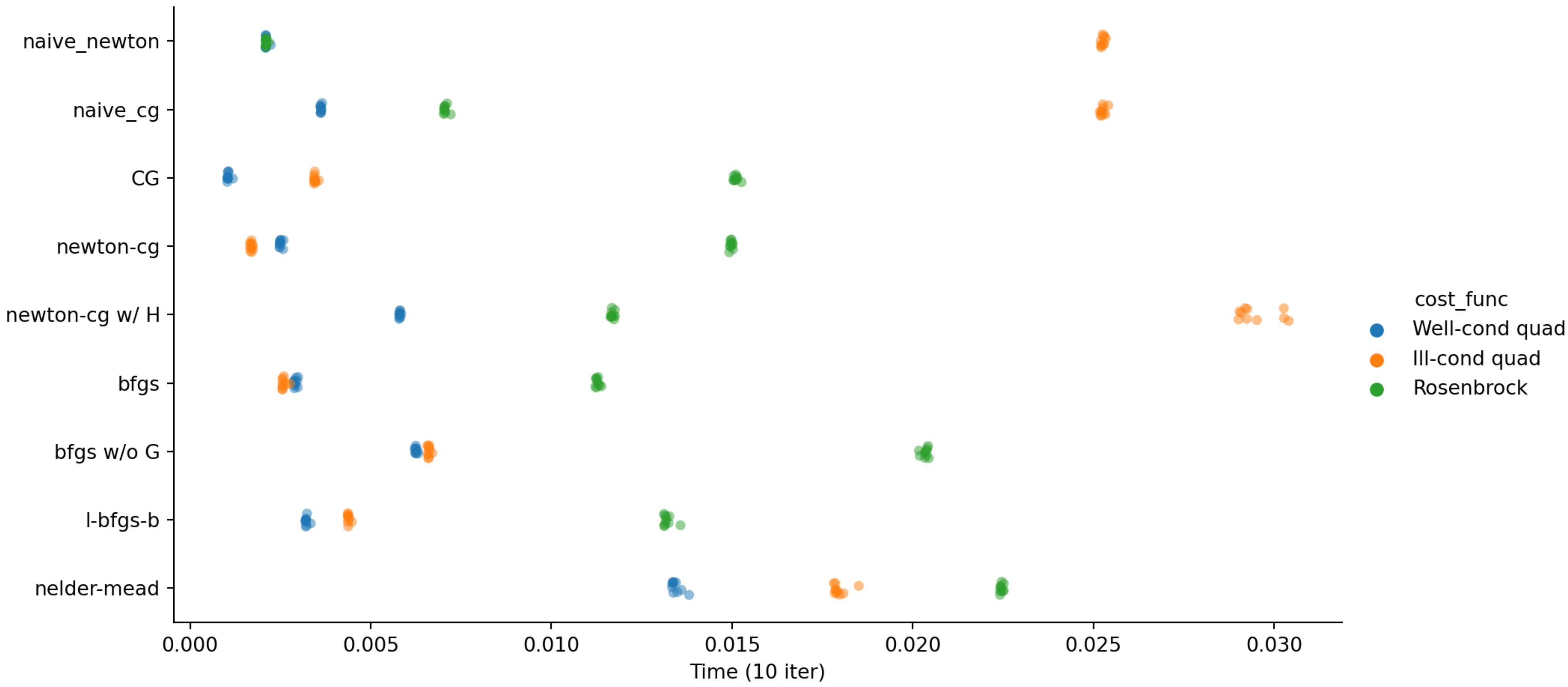
[10 rows x 9 columns]


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
0.12
0.14
Time (100 iter)

Timings across cost functions

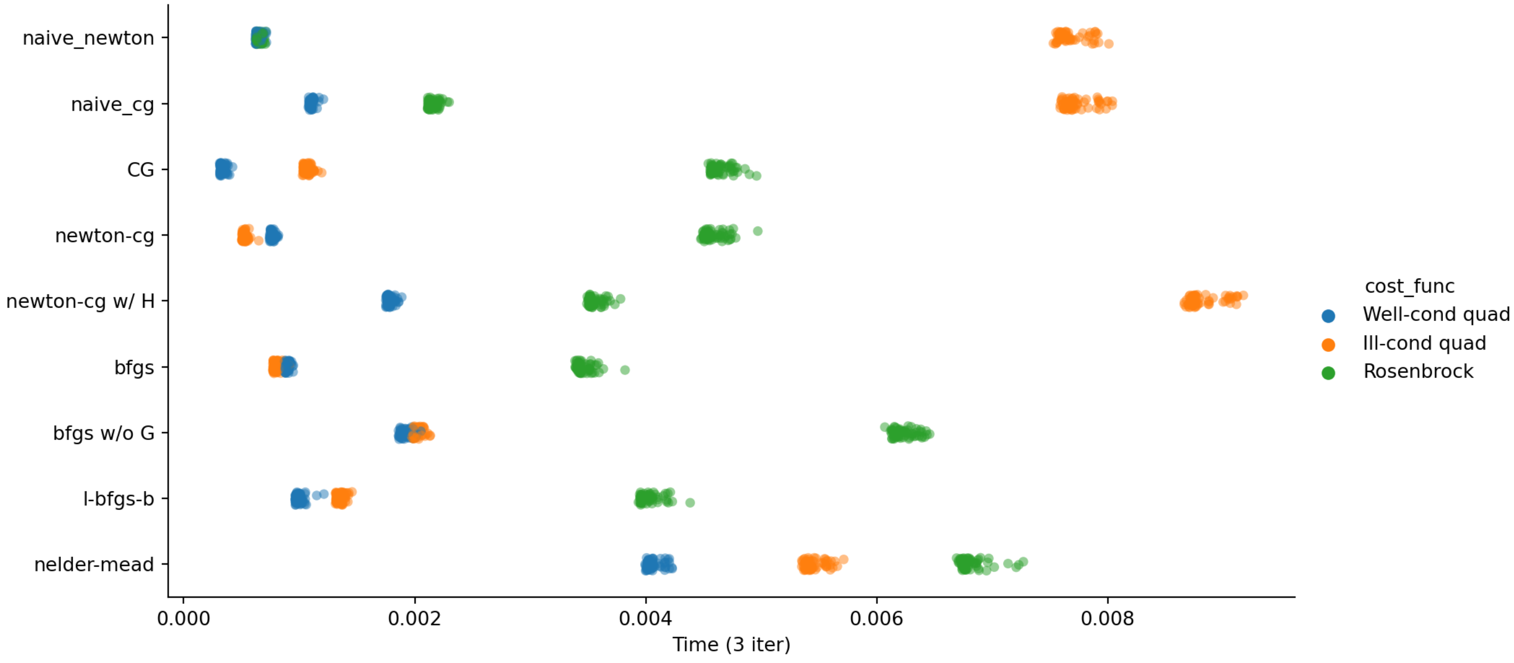
```python
def time_cost_func(n, x0, name, cost_func, *args
  x0 = (1.6, 1.1)
  f, grad, hess = cost_func(*args)
  methods = define_methods(x0, f, grad, hess)

  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, x0, "Well-cond quad", mk_qu
  time_cost_func(10, x0, "Ill-cond quad", mk_qua
  time_cost_func(10, x0, "Rosenbrock", mk_rosenb
])

df
```

```
        variable     value       cost_func
0   naive_newton  0.002238  Well-cond quad
1   naive_newton  0.002109  Well-cond quad
2   naive_newton  0.002106  Well-cond quad
3   naive_newton  0.002106  Well-cond quad
4   naive_newton  0.002097  Well-cond quad
..           ...       ...             ...
85   nelder-mead  0.022443      Rosenbrock
86   nelder-mead  0.022429      Rosenbrock
87   nelder-mead  0.022463      Rosenbrock
88   nelder-mead  0.022445      Rosenbrock
89   nelder-mead  0.022520      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
Time (10 iter)
cost_func
Well-cond quad
Ill-cond quad
Rosenbrock

Random starting locations

```python
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)

```
import cProfile

f, grad, hess = mk_quad(0.7)
def run():
  optimize.minimize(fun = f, x0 = (1.6, 1.1), jac=grad, method="BFGS", tol=1e-11)

cProfile.run('run()', sort="tottime")
```

```
         1293 function calls in 0.001 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        9    0.000    0.000    0.000    0.000 _optimize.py:1144(_line_search_wolfe12)
        1    0.000    0.000    0.001    0.001 _optimize.py:1318(_minimize_bfgs)
       58    0.000    0.000    0.000    0.000 {method 'reduce' of 'numpy.ufunc' objects}
      152    0.000    0.000    0.000    0.000 {built-in method numpy.core._multiarray_umath.implement_array_
       10    0.000    0.000    0.000    0.000 <string>:2(f)
        9    0.000    0.000    0.000    0.000 _linesearch.py:91(scalar_search_wolfe1)
       37    0.000    0.000    0.000    0.000 fromnumeric.py:69(_wrapreduction)
       26    0.000    0.000    0.000    0.000 _optimize.py:235(vecnorm)
       10    0.000    0.000    0.000    0.000 <string>:8(gradient)
       20    0.000    0.000    0.000    0.000 numeric.py:2407(array_equal)
        1    0.000    0.000    0.001    0.001 _minimize.py:45(minimize)
        1    0.000    0.000    0.000    0.000 _differentiable_functions.py:86(__init__)
       21    0.000    0.000    0.000    0.000 shape_base.py:23(atleast_1d)
       51    0.000    0.000    0.000    0.000 <__array_function__ internals>:177(dot)
```

```
    9    0.000    0.000    0.000    0.000 _linesearch.py:77(derphi)
   26    0.000    0.000    0.000    0.000 fromnumeric.py:2188(sum)
    9    0.000    0.000    0.000    0.000 _linesearch.py:73(phi)
   84    0.000    0.000    0.000    0.000 {built-in method numpy.asarray}
```

Profiling - BFGS (pyinstrument)

```python
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)
```

```
  _     ._   __/__   _ _  _  _ _/_   Recorded: 11:22:35  Samples:  346
 /_//_/// /_\ / //_// / //_'/ //     Duration: 0.005     CPU time: 0.005
/   _/                      v4.4.0

Program: /opt/homebrew/Cellar/python@3.10/3.10.10_1/libexec/bin/python3

0.005 MainThread  <thread>:8553529664
|- 0.003 [self]  None
|- 0.002 <module>  <string>:1
|  `- 0.002 minimize  scipy/optimize/_minimize.py:45
|     `- 0.002 _minimize_bfgs  scipy/optimize/_optimize.py:1318
|        |- 0.002 _line_search_wolfe12  scipy/optimize/_optimize.py:1144
|        |  `- 0.002 line_search_wolfe1  scipy/optimize/_linesearch.py:31
|        |     `- 0.002 scalar_search_wolfe1  scipy/optimize/_linesearch.py:91
```

```
|          |          |- 0.001 phi  scipy/optimize/_linesearch.py:73
|          |          |  |- 0.001 ScalarFunction.fun  scipy/optimize/_differentiable_functions.py:264
|          |          |  |  |- 0.001 ScalarFunction._update_fun  scipy/optimize/_differentiable_functions.py:249
|          |          |  |  |  `- 0.001 update_fun  scipy/optimize/_differentiable_functions.py:154
|          |          |  |  |     `- 0.001 fun_wrapped  scipy/optimize/_differentiable_functions.py:132
|          |          |  |  |        `- 0.001 f  <string>:2
|          |          |  |  |           |- 0.000 sum  <__array_function__ internals>:177
|          |          |  |  |           |  `- 0.000 sum  numpy/core/fromnumeric.py:2188
```

Profiling - Nelder-Mead

```python
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)
```

```
  _     ._   __/__   _ _  _  _ _/_   Recorded: 11:22:35  Samples:  649
 /_//_/// /_\ / //_// / //_'/ //     Duration: 0.008     CPU time: 0.008
/   _/                      v4.4.0

Program: /opt/homebrew/Cellar/python@3.10/3.10.10_1/libexec/bin/python3

0.008 MainThread  <thread>:8553529664
|- 0.006 <module>  <string>:1
|  `- 0.006 minimize  scipy/optimize/_minimize.py:45
|     `- 0.005 _minimize_neldermead  scipy/optimize/_optimize.py:708
|        |- 0.002 function_wrapper  scipy/optimize/_optimize.py:564
|        |  |- 0.002 f  <string>:2
|        |  |  |- 0.001 sum  <__array_function__ internals>:177
|        |  |  |  |- 0.001 sum  numpy/core/fromnumeric.py:2188
```

```
|        |  |  |  |  |- 0.001 _wrapreduction  numpy/core/fromnumeric.py:69
|        |  |  |  |  |  |- 0.001 ufunc.reduce  None
|        |  |  |  |  |  `- 0.000 [self]  None
|        |  |  |  |  `- 0.000 [self]  None
|        |  |  |  `- 0.000 [self]  None
|        |  |  `- 0.001 [self]  None
|        |  |- 0.000 copy  <__array_function__ internals>:177
|        |  |  |- 0.000 [self]  None
```

`optimize.minimize()` output

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

```python
optimize.minimize(fun = f, x0 = (1.6, 1.1),
                  jac=grad, method="BFGS")
```

```
  message: Optimization terminated successfully.
  success: True
   status: 0
      fun: 1.2739256453438974e-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
```

```python
optimize.minimize(fun = f, x0 = (1.6, 1.1),
                  jac=grad, hess=hess,
                  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.)

```python
optimize.minimize(fun = f, x0 = (1.6, 1.1),
                  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
```

```python
optimize.minimize(fun = f, x0 = (1.6, 1.1),
                  jac=grad, method="Nelder-Mead"
```

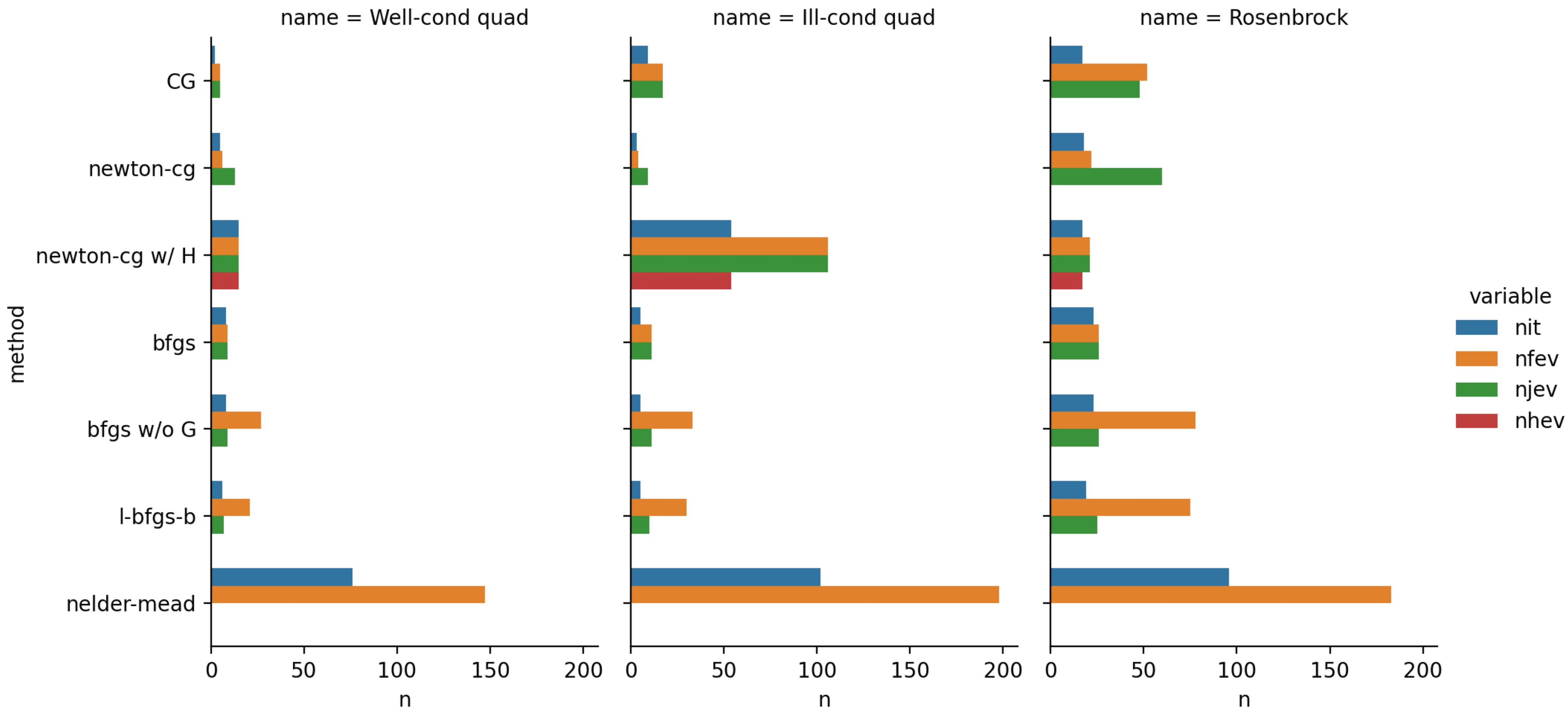
```
      message: Optimization terminated successfully.
      success: True
       status: 0
          fun: 2.3077013477040082e-10
            x: [ 1.088e-05  3.443e-05]
          nit: 46
         nfev: 89
final_simplex: (array([[ 1.088e-05,  3.443e-05],
                       [ 1.882e-05, -3.825e-05],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

/opt/homebrew/lib/python3.10/site-packages/scipy/opt
  warn('Method %s does not use gradient information
```

Collect

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

  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=[1]) )

  return pd.concat(res)
```

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

df
```

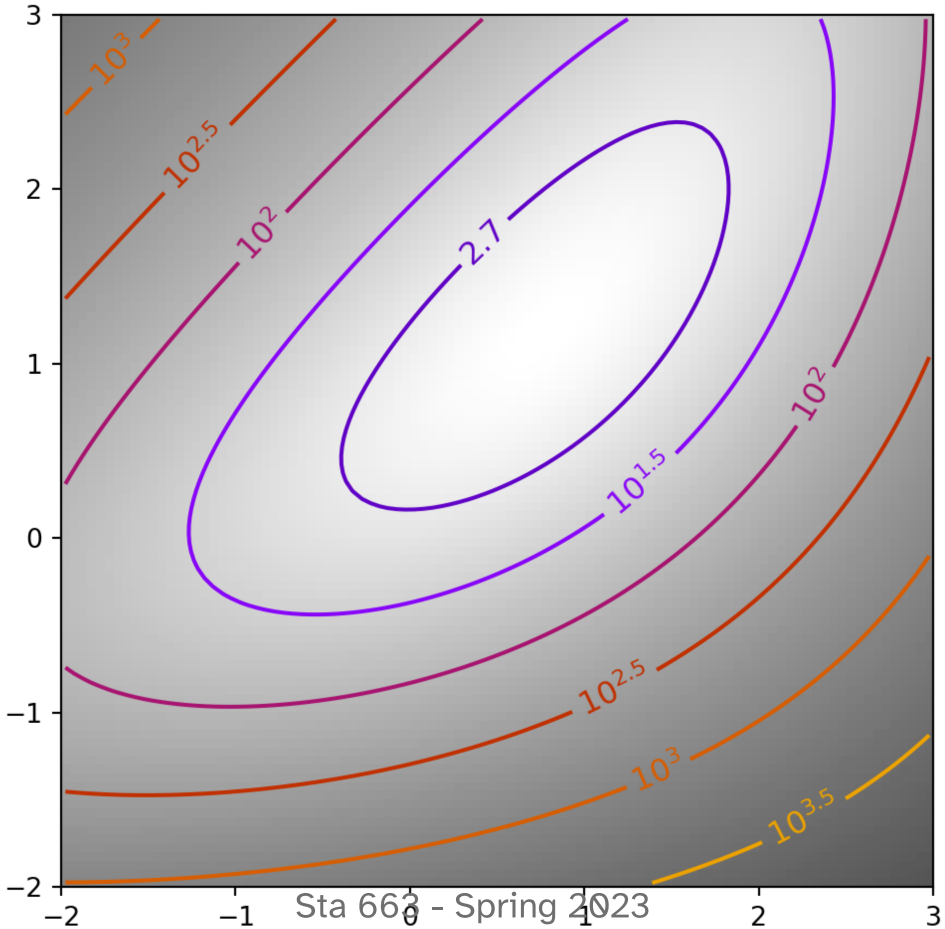
	name	method	nit	nfev	njev
1	Well-cond quad	CG	2	5	5
1	Well-cond quad	newton-cg	5	6	13
1	Well-cond quad	newton-cg w/ H	15	15	15
1	Well-cond quad	bfgs	8	9	9
1	Well-cond quad	bfgs w/o G	8	27	9
1	Well-cond quad	l-bfgs-b	6	21	7
1	Well-cond quad	nelder-mead	76	147	None
1	Ill-cond quad	CG	9	17	17
1	Ill-cond quad	newton-cg	3	4	9
1	Ill-cond quad	newton-cg w/ H	54	106	106
1	Ill-cond quad	bfgs	5	11	11
1	Ill-cond quad	bfgs w/o G	5	33	11
1	Ill-cond quad	l-bfgs-b	5	30	10
1	Ill-cond quad	nelder-mead	102	198	None
1	Rosenbrock	CG	17	52	48
1	Rosenbrock	newton-cg	18	22	60
1	Rosenbrock	newton-cg w/ H	17	21	21
1	Rosenbrock	bfgs	23	26	26
1	Rosenbrock	bfgs w/o G	23	78	26
1	Rosenbrock	l-bfgs-b	19	75	25


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.

```
def mk_mvn(mu, Sigma):
  Sigma_inv = np.linalg.inv(Sigma)
  norm_const = 1 / (np.sqrt(np.linalg.det(2*np.p

  # Returns the negative density (since we want
  def f(x):
    x_m = x - mu
    return -(norm_const *
      np.exp( -0.5 * (x_m.T @ Sigma_inv @ x_m).i

  def grad(x):
    return (-f(x) * Sigma_inv @ (x - mu))

  def hess(x):
    n = len(x)
    x_m = x - mu
    return f(x) * ((Sigma_inv @ x_m).reshape((n,

  return f, grad, hess
```

From Section 8.1.1 of the Matrix Cookbook

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.eye(2,2))
optimize.check_grad(f, grad, np.zeros(2))
```

```
2.634178031930877e-09
```

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

```
5.213238144735062e-10
```

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

```
2.6031257322754127e-10
```

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

```
1.725679820308689e-11
```

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

```
2.8760747774580336e-12
```

```
optimize.check_grad(f, grad, np.ones(10))
```

```
2.850398669793798e-14
```

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

```
4.965068306494546e-16
```

```
optimize.check_grad(f, grad, np.ones(20))
```

```
1.0342002372572841e-20
```

Gradient checking (wrong gradient)

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

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

```
2.634178031930877e-09
```

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

```
0.08280196633767578
```

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

```
2.6031257322754127e-10
```

```
optimize.check_grad(f, wrong_grad, np.ones(5))
```

```
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`.

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

```
3.925231146709438e-17
```

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

```
8.399162985270666e-10
```

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

```
3.878959614448864e-18
```

```
optimize.check_grad(grad, hess, np.ones(5))
```

```
3.8156075963144067e-11
```

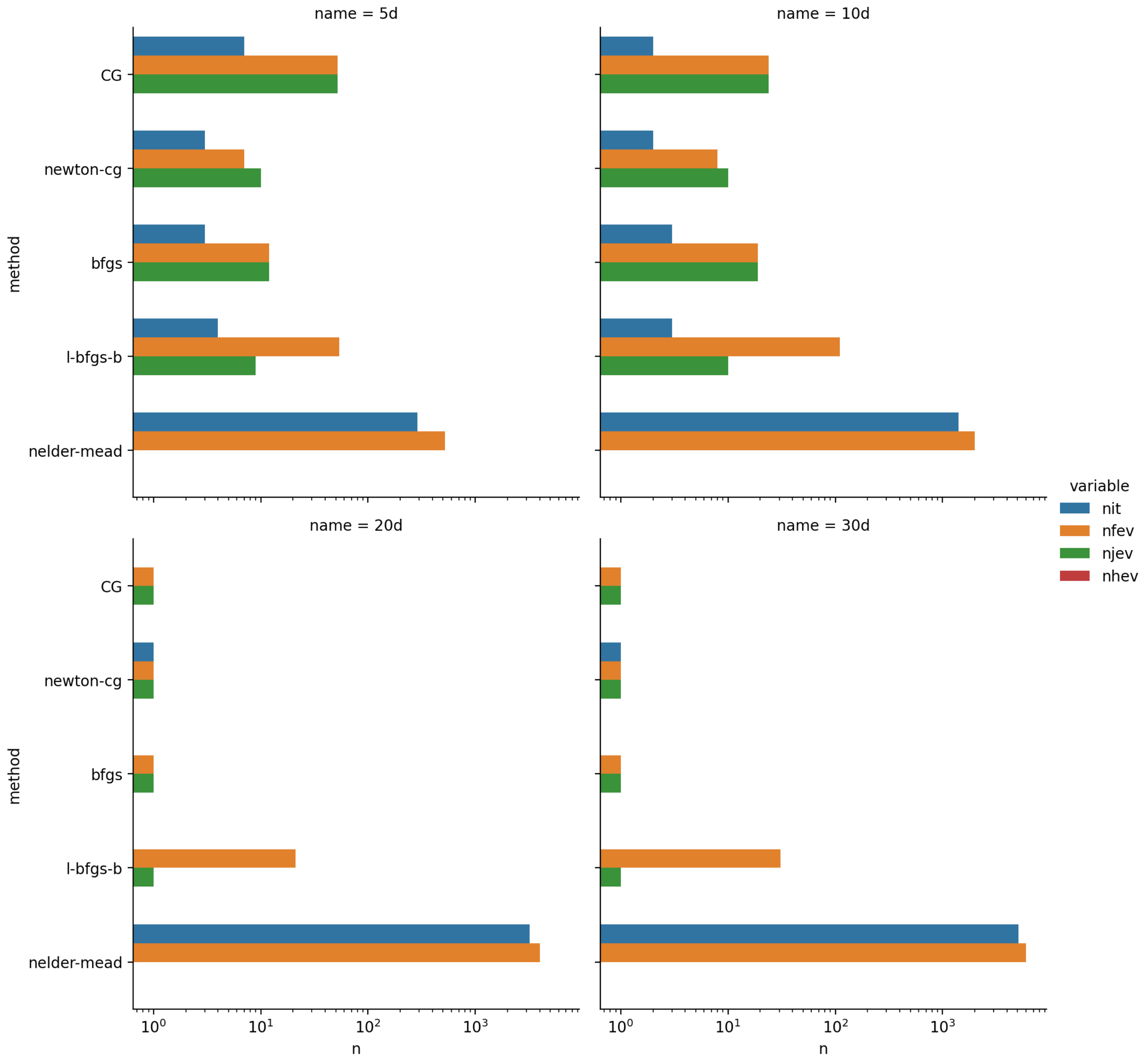
Unit MVNs

```
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', 'bfgs w/o
  )
  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
1	10d	bfgs	3	19	19	None	True
1	10d	l-bfgs-b	3	110	10	None	True
1	10d	nelder-mead	1403	2000	None	None	False
1	20d	CG	0	1	1	None	True
1	20d	newton-cg	1	1	1	0	True
1	20d	bfgs	0	1	1	None	True
1	20d	l-bfgs-b	0	21	1	None	True
1	20d	nelder-mead	3217	4000	None	None	False
1	30d	CG	0	1	1	None	True
1	30d	newton-cg	1	1	1	0	True
1	30d	bfgs	0	1	1	None	True
1	30d	l-bfgs-b	0	31	1	None	True
1	30d	nelder-mead	5097	6000	None	None	False

Performance (Unit MVNs)

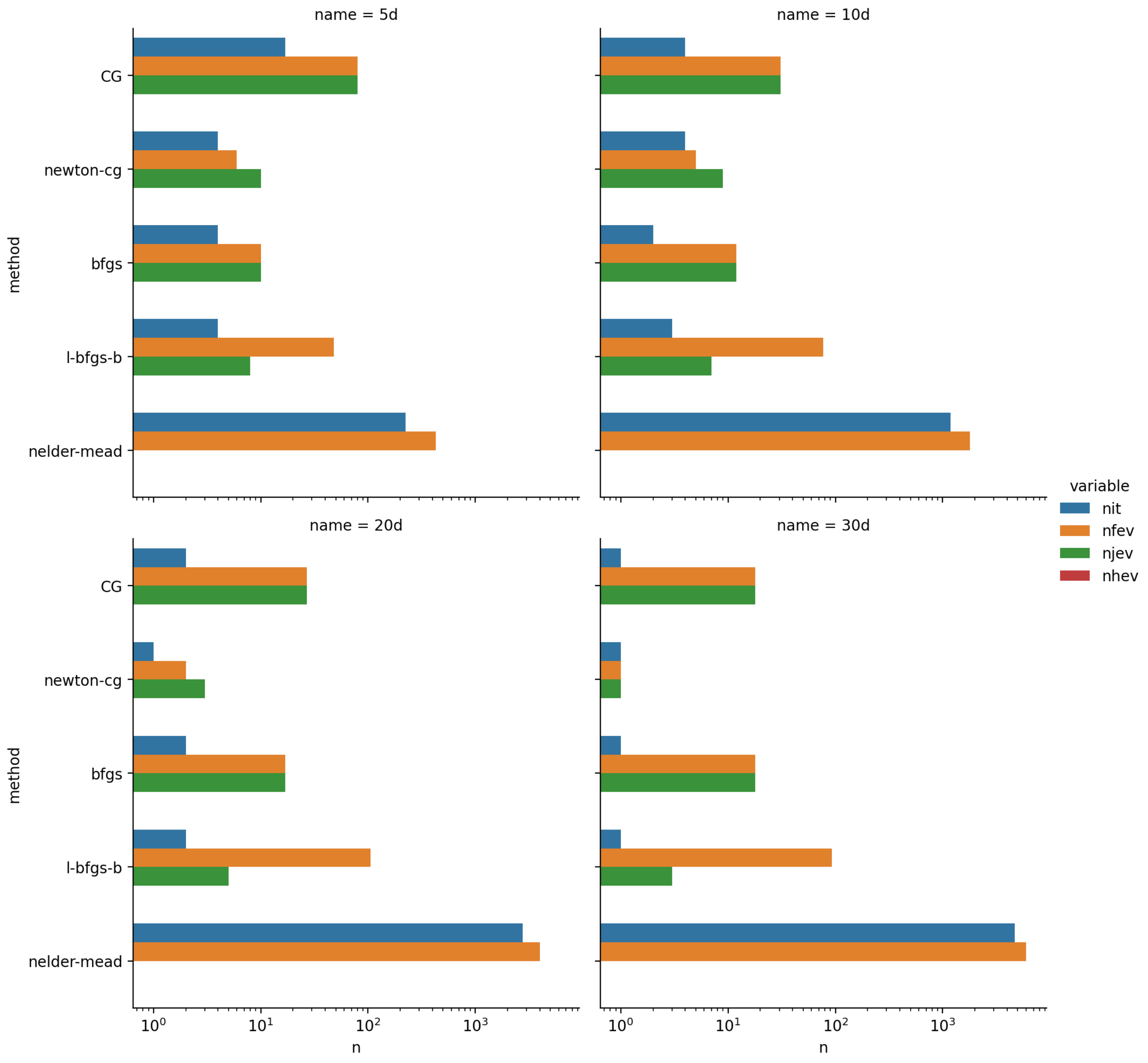


Adding correlation

```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', 'bfgs w/o
  )
  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	48	8	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
1	10d	bfgs	2	12	12	None	True
1	10d	l-bfgs-b	3	77	7	None	True
1	10d	nelder-mead	1184	1802	None	None	True
1	20d	CG	2	27	27	None	True
1	20d	newton-cg	1	2	3	0	True
1	20d	bfgs	2	17	17	None	True
1	20d	l-bfgs-b	2	105	5	None	True
1	20d	nelder-mead	2745	4000	None	None	False
1	30d	CG	1	18	18	None	True
1	30d	newton-cg	1	1	1	0	True
1	30d	bfgs	1	18	18	None	True
1	30d	l-bfgs-b	1	93	3	None	True
1	30d	nelder-mead	4687	6000	None	None	False


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

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=grad,
                  method="CG", tol=1e-9)
```

```
message: Optimization terminated successfully.
success: True
 status: 0
    fun: -0.023337250777292103
      x: [ 1.086e-07  1.061e-07  1.080e-07  1.080e-(
    nit: 14
    jac: [ 8.802e-10  7.646e-10  8.540e-10  8.532e-:
   nfev: 67
   njev: 67
```

```
optimize.minimize(f, np.ones(n), jac=grad,
                  method="CG", tol=1e-10)
```

```
message: Optimization terminated successfully.
success: True
 status: 0
    fun: -0.023337250777292328
      x: [-2.232e-09 -3.779e-10 -1.811e-09 -1.797e-(
    nit: 17
    jac: [-4.415e-11  4.237e-11 -2.453e-11 -2.387e-:
   nfev: 80
   njev: 80
```

```python
optimize.minimize(f, np.ones(n), jac=grad,
                  method="CG", tol=1e-11)
```

```
message: Desired error not necessarily achieved due
success: False
 status: 2
    fun: -0.02333725077729232
      x: [ 2.205e-08  3.734e-09  1.790e-08  1.776e-(
    nit: 16
    jac: [ 4.362e-10 -4.186e-10  2.424e-10  2.359e-:
   nfev: 93
   njev: 81
```

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=grad,
                  method="CG", tol=1e-9)
```

```
message: Optimization terminated successfully.
success: True
 status: 0
    fun: -2.330191334018497e-06
      x: [-3.221e-04 -3.221e-04 ... -3.221e-04 -3.2:
    nit: 2
    jac: [-7.148e-11 -7.148e-11 ... -7.148e-11 -7.1
   nfev: 27
   njev: 27
```

```
optimize.minimize(f, np.ones(n), jac=grad,
                  method="CG", tol=1e-10)
```

```
message: Optimization terminated successfully.
success: True
 status: 0
    fun: -2.330191334018497e-06
      x: [-3.221e-04 -3.221e-04 ... -3.221e-04 -3.2:
    nit: 2
    jac: [-7.148e-11 -7.148e-11 ... -7.148e-11 -7.1
   nfev: 27
   njev: 27
```

```python
optimize.minimize(f, np.ones(n), jac=grad,
                  method="CG", tol=1e-11)
```

```
message: Optimization terminated successfully.
success: True
 status: 0
    fun: -2.3301915597315495e-06
      x: [-4.506e-05 -4.506e-05 ... -4.506e-05 -4.5(
    nit: 2884
    jac: [-9.999e-12 -9.999e-12 ... -9.999e-12 -9.9(
   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=grad,
                  method="CG", tol=1e-9)
```

```
message: Optimization terminated successfully.
success: True
 status: 0
    fun: -2.3811463025973114e-09
      x: [ 1.000e+00  1.000e+00 ...  1.000e+00  1.00
    nit: 0
    jac: [ 1.536e-10  1.536e-10 ...  1.536e-10  1.53
   nfev: 1
   njev: 1
```

```
optimize.minimize(f, np.ones(n), jac=grad,
                  method="CG", tol=1e-10)
```

```
message: Optimization terminated successfully.
success: True
 status: 0
    fun: -6.180056227752818e-09
      x: [ 1.203e-01  1.203e-01 ...  1.203e-01  1.20
    nit: 1
    jac: [ 4.795e-11  4.795e-11 ...  4.795e-11  4.79
   nfev: 18
   njev: 18
```

```python
optimize.minimize(f, np.ones(n), jac=grad,
                  method="CG", tol=1e-11)
```

```
message: Optimization terminated successfully.
success: True
 status: 0
    fun: -6.26701117075865e-09
      x: [-5.021e-03 -5.021e-03 ... -5.021e-03 -5.02
    nit: 2
    jac: [-2.030e-12 -2.030e-12 ... -2.030e-12 -2.03
   nfev: 35
   njev: 35
```

Options (bfgs)

```python
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.
return_all : bool, optional
    Set to True to return a list of the best solution at each of the
    iterations.
finite_diff_rel_step : None or array_like, optional
    If `jac in ['2-point', '3-point', 'cs']` the relative step size to
    use for numerical approximation of the jacobian. The absolute step
    size is computed as ``h = rel_step * sign(x) * max(1, abs(x))``,
```

Options (Nelder-Mead)

```python
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.
initial_simplex : array_like of shape (N + 1, N)
    Initial simplex. If given, overrides `x0`.
    ``initial_simplex[j,:]`` should contain the coordinates of
    the jth vertex of the ``N+1`` vertices in the simplex, where
    ``N`` is the dimension.
xatol : float, optional
    Absolute error in xopt between iterations that is acceptable for
```

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)
```

See code in context on GitHub here

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

```
from scipy.stats import norm

n = norm(-3.2, 1.25)
x = n.rvs(size=100, random_state=1234)
x.round(2)
```

```
array([-2.61, -4.69, -1.41, -3.59, -4.1 , -2.09, -2.
       -6.  , -1.76, -1.96, -2.01, -5.73, -3.62, -3.
       -1.55, -5.13, -3.45, -4.02, -2.96, -2.51, -1.
       -5.47, -3.43, -1.88, -3.7 , -2.78, -1.89, -1.
       -3.04, -3.6 , -2.15, -0.21, -3.1 , -3.91, -3.
       -4.32, -3.37, -3.18, -2.26, -2.93, -2.15, -5.
       -3.89, -3.38, -2.76, -3.24, -2.49, -1.27, -4.
       -3.46, -1.91, -6.2 , -0.66, -4.63, -2.94, -2.
       -2.32, -2.55, -4.36, -0.69, -2.92, -4.64, -2.
       -7.65, -1.55, -3.01, -2.99, -3.74, -2.24, -1.
       -3.1 , -3.7 , -4.48, -3.93, -2.18, -3.3 , -3.
       -3.84])
```

```
{'μ': x.mean(), 'σ': x.std()}
```

```
{'μ': -3.156109646093205, 'σ': 1.2446060629192535}
```

```
mle_norm = lambda θ: -np.sum(
  norm.logpdf(x, loc=θ[0], scale=θ[1])
)

mle_norm([0,1])
```

```
667.3974708213642
```

```
mle_norm([-3, 1])
```

```
170.56457699340282
```

```
mle_norm([-3.2, 1.25])
```

```
163.83926813257395
```

Minimizing

```
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=θ[0], scal
```

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

```
  message: Optimization terminated successfully.
  success: True
   status: 0
      fun: 163.77575977255518
        x: [-3.156e+00  1.245e+00]
      nit: 9
      jac: [-1.907e-06  0.000e+00]
 hess_inv: [[ 1.475e-02 -1.069e-04]
            [-1.069e-04  7.734e-03]]
     nfev: 47
     njev: 15

/opt/homebrew/lib/python3.10/site-packages/scipy/opt:
  df = fun(x) - f0
```

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: REL_REDUCTION_OF_F_<=_FACTR*EPSMCH
  success: True
   status: 0
      fun: 163.7757597728758
        x: [-3.156e+00  1.245e+00]
      nit: 10
      jac: [ 2.075e-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.

```
from scipy.stats import gamma

g = gamma(a=2.0, scale=2.0)
x = g.rvs(size=100, random_state=1234)
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 ])
```