Lec 13 - Numerical optimization (cont.)

Statistical Computing and Computation

Sta 663 | Spring 2022

Dr. Colin Rundel

Method Summary

SciPy Method	Description	Gradient	Hessian
---	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	✗

Methods collection

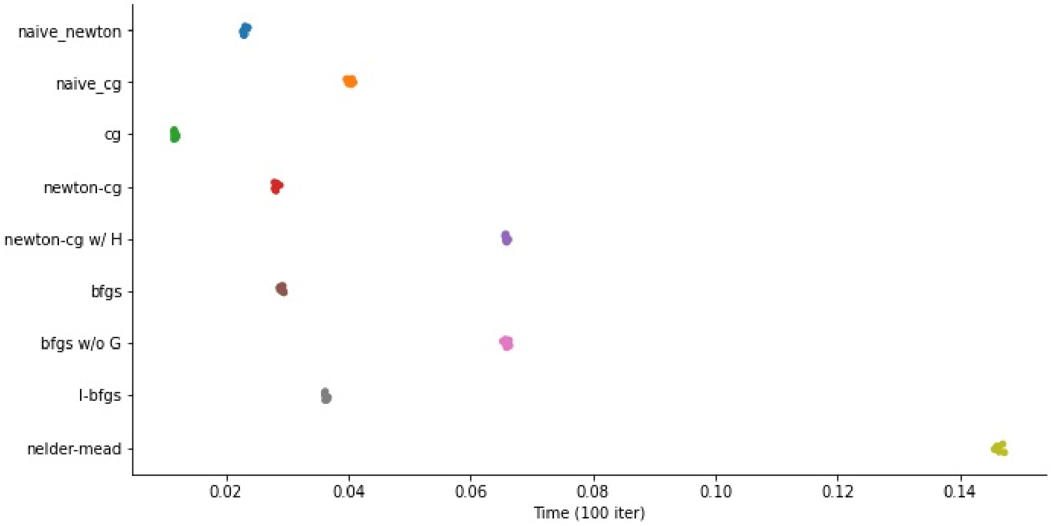
```
def define_methods(x0, f, grad, hess, tol=1e-8):
  return {
    "naive_newton":    lambda: newtons_method(x0, f, grad, hess, tol=tol),
    "naive_cg":        lambda: conjugate_gradient(x0, f, grad, hess, tol=tol),
    "cg":              lambda: optimize.minimize(f, x0, jac=grad, method="CG", tol=tol),
    "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":          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
```

```
##    naive_newton  naive_cg        cg  ...  bfgs w/o G    l-bfgs  nelder-mead
## 0      0.023537  0.039970  0.011881  ...    0.066303  0.036481     0.147036
## 1      0.022836  0.040031  0.011484  ...    0.066409  0.036509     0.145659
## 2      0.023006  0.040840  0.011460  ...    0.065983  0.036171     0.146303
## 3      0.023108  0.040619  0.011740  ...    0.065224  0.036673     0.146443
## 4      0.022910  0.040613  0.011933  ...    0.065597  0.036137     0.146067
## 5      0.022782  0.040496  0.011701  ...    0.066092  0.036383     0.147324
## 6      0.022979  0.040472  0.011504  ...    0.065924  0.036287     0.146281
## 7      0.023019  0.040539  0.011490  ...    0.066140  0.036171     0.146400
## 8      0.022744  0.039657  0.011497  ...    0.065693  0.036117     0.145820
## 9      0.022946  0.039879  0.011523  ...    0.065842  0.036078     0.146332
##
## [10 rows x 9 columns]
```

```
g = sns.catplot(data=df.melt(), y="variable", x="value", aspect=2)
g.ax.set_xlabel("Time (100 iter)")
g.ax.set_ylabel("")
plt.show()
```



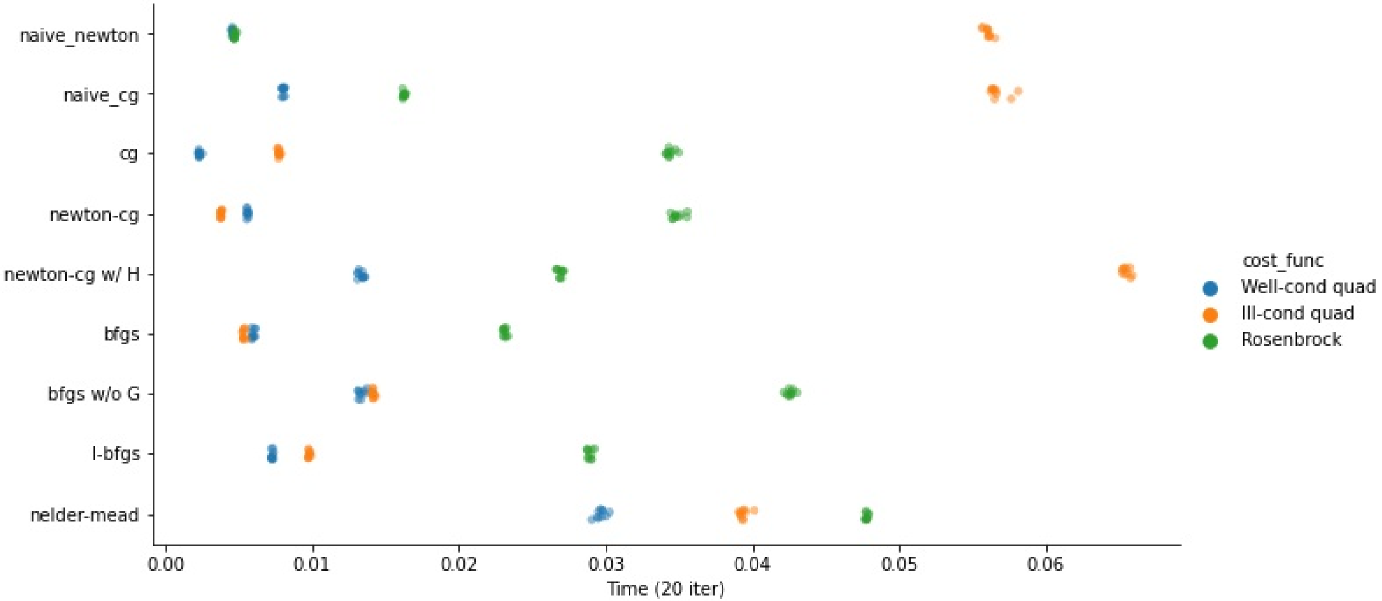
Timings across cost functions

```
def time_cost_func(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(10, 20) for key in
    })
    .melt()
    .assign(cost_func = name)
  )

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

```
df
```

```
##         variable     value        cost_func
## 0   naive_newton  0.004699   Well-cond quad
## 1   naive_newton  0.004590   Well-cond quad
## 2   naive_newton  0.004567   Well-cond quad
## 3   naive_newton  0.004557   Well-cond quad
## 4   naive_newton  0.004553   Well-cond quad
## ..           ...       ...              ...
## 85   nelder-mead  0.047754       Rosenbrock
## 86   nelder-mead  0.047654       Rosenbrock
## 87   nelder-mead  0.047935       Rosenbrock
## 88   nelder-mead  0.047746       Rosenbrock
## 89   nelder-mead  0.047725       Rosenbrock
##
## [270 rows x 3 columns]
```

```
g = sns.catplot(data=df, y="variable", x="value", hue="cost_func", alpha=0.5, aspect=2)
g.ax.set_xlabel("Time (20 iter)")
g.ax.set_ylabel("")
plt.show()
```



Profiling - BFGS

```python
import cProfile

f, grad, hess = mk_quad(0.7)

def run():
  for i  in range(100):
    optimize.minimize(fun = f, x0 = (1.6, 1.1), jac=grad, method="BFGS", tol=1e-11)

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

```
##          112904 function calls (112804 primitive calls) in 0.047 seconds
##
##    Ordered by: internal time
##
##    ncalls  tottime  percall  cumtime  percall filename:lineno(function)
##       100    0.010    0.000    0.046    0.000 _optimize.py:1253(_minimize_bfgs)
## 13700/13600    0.004    0.000    0.015    0.000 {built-in method numpy.core._multiarray_umath.implement_array_function}
##      4200    0.003    0.000    0.003    0.000 {method 'reduce' of 'numpy.ufunc' objects}
##      1000    0.002    0.000    0.003    0.000 <string>:8(gradient)
##      1000    0.002    0.000    0.006    0.000 <string>:2(f)
##       900    0.002    0.000    0.025    0.000 _linesearch.py:91(scalar_search_wolfe1)
##      2000    0.002    0.000    0.004    0.000 numeric.py:2388(array_equal)
##      2100    0.001    0.000    0.004    0.000 fromnumeric.py:69(_wrapreduction)
##       900    0.001    0.000    0.010    0.000 _linesearch.py:77(derphi)
##      5200    0.001    0.000    0.004    0.000 <__array_function__ internals>:177(dot)
##       900    0.001    0.000    0.013    0.000 _linesearch.py:73(phi)
##      2100    0.001    0.000    0.001    0.000 shape_base.py:23(atleast_1d)
##      1000    0.001    0.000    0.008    0.000 _differentiable_functions.py:132(fun_wrapped)
##      1000    0.001    0.000    0.005    0.000 _differentiable_functions.py:162(grad_wrapped)
##       900    0.001    0.000    0.026    0.000 _linesearch.py:31(line_search_wolfe1)
```

Profiling - Nelder-Mead

```
def run():
  for i  in range(100):
    optimize.minimize(fun = f, x0 = (1.6, 1.1), method="Nelder-Mead", tol=1e-11)

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

```
##          756504 function calls in 0.270 seconds
##
##    Ordered by: internal time
##
##    ncalls  tottime  percall  cumtime  percall filename:lineno(function)
##       100    0.071    0.001    0.269    0.003 _optimize.py:635(_minimize_neldermead)
##     18600    0.034    0.000    0.087    0.000 <string>:2(f)
##     38000    0.028    0.000    0.028    0.000 {method 'reduce' of 'numpy.ufunc' objects}
##     86400    0.017    0.000    0.108    0.000 {built-in method numpy.core._multiarray_umath.implement_array_function}
##     28500    0.013    0.000    0.040    0.000 fromnumeric.py:69(_wrapreduction)
##     18600    0.011    0.000    0.117    0.000 _optimize.py:491(function_wrapper)
##     18600    0.008    0.000    0.035    0.000 fromnumeric.py:2160(sum)
##     29400    0.007    0.000    0.019    0.000 fromnumeric.py:51(_wrapfunc)
##     19600    0.006    0.000    0.006    0.000 {method 'take' of 'numpy.ndarray' objects}
##     28500    0.005    0.000    0.005    0.000 fromnumeric.py:70(<dictcomp>)
##     18800    0.005    0.000    0.005    0.000 {built-in method numpy.array}
##     19600    0.005    0.000    0.025    0.000 <__array_function__ internals>:177(take)
##     19600    0.005    0.000    0.016    0.000 fromnumeric.py:93(take)
##     18600    0.004    0.000    0.044    0.000 <__array_function__ internals>:177(sum)
##     18600    0.004    0.000    0.004    0.000 {built-in method numpy.arange}
##     18600    0.004    0.000    0.016    0.000 <__array_function__ internals>:177(copy)
##      9800    0.004    0.000    0.004    0.000 {method 'argsort' of 'numpy.ndarray' objects}
##      9700    0.003    0.000    0.017    0.000 fromnumeric.py:2675(amax)
##      9600    0.003    0.000    0.006    0.000 fromnumeric.py:1755(ravel)
```

optimize.minimize() output

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

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

```
##       fun: 1.2739256453436805e-11
##  hess_inv: array([[ 1.51494475, -0.00343804],
##        [-0.00343804,  3.03497828]])
##       jac: array([-3.51014018e-07, -2.85996115e-06])
##   message: 'Optimization terminated successfully.'
##      nfev: 7
##       nit: 6
##      njev: 7
##    status: 0
##   success: True
##         x: array([-5.31839421e-07, -8.84341728e-06])
```

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

```
##      fun: 2.3418652989289317e-12
##      jac: array([0.00000000e+00, 4.10246332e-06])
##  message: 'Optimization terminated successfully.'
##     nfev: 12
##     nhev: 11
##      nit: 11
##     njev: 12
##   status: 0
##  success: True
##        x: array([0.0000000e+00, 3.8056246e-06])
```

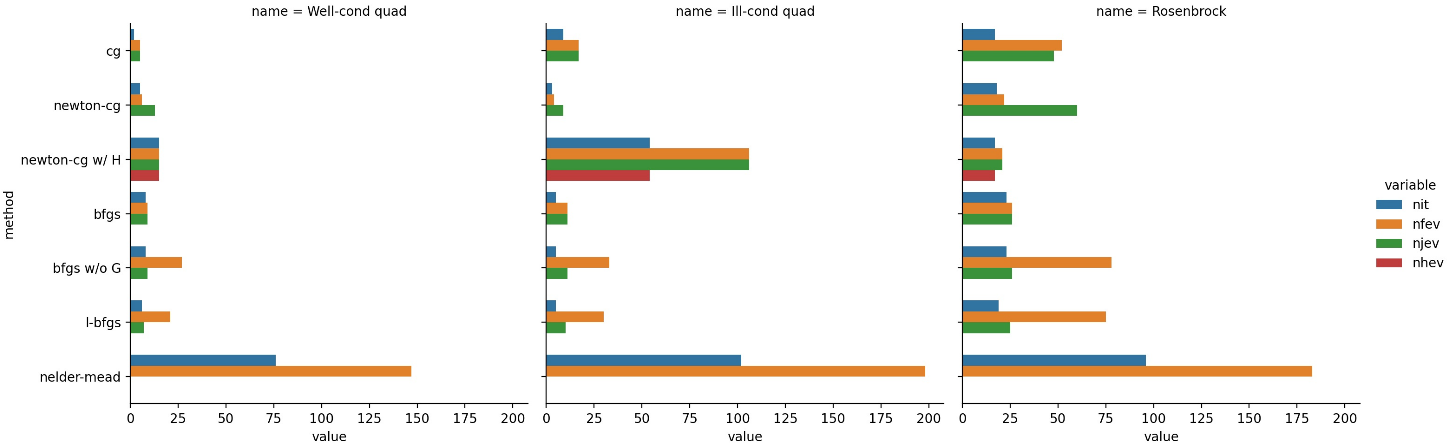
Collect

```
def run_collect(name, x0, cost_func, *args, tol=1e-8, skip=[]):
  f, grad, hess = cost_func(*args)
  methods = define_methods(x0, f, grad, hess, tol)

  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)

df = pd.concat([
  run_collect(name, (1.6, 1.1), cost_func, arg, skip=['naive_ne
  for name, cost_func, arg in zip(
    ("Well-cond quad", "Ill-cond quad", "Rosenbrock"),
    (mk_quad, mk_quad, mk_rosenbrock),
```

```
df.drop(["message"], axis=1)
```

```
##                 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    6    21     7  None     True
## 1  Well-cond quad     nelder-mead   76   147  None  None     True
## 1   Ill-cond quad              cg    9    17    17  None     True
## 1   Ill-cond quad       newton-cg    3     4     9     0     True
## 1   Ill-cond quad  newton-cg w/ H   54   106   106    54     True
## 1   Ill-cond quad            bfgs    5    11    11  None     True
## 1   Ill-cond quad      bfgs w/o G    5    33    11  None     True
## 1   Ill-cond quad          l-bfgs    5    30    10  None     True
## 1   Ill-cond quad     nelder-mead  102   198  None  None     True
## 1      Rosenbrock              cg   17    52    48  None     True
## 1      Rosenbrock       newton-cg   18    22    60     0     True
## 1      Rosenbrock  newton-cg w/ H   17    21    21    17     True
## 1      Rosenbrock            bfgs   23    26    26  None     True
## 1      Rosenbrock      bfgs w/o G   23    78    26  None     True
## 1      Rosenbrock          l-bfgs   19    75    25  None     True
## 1      Rosenbrock     nelder-mead   96   183  None  None     True
```

```
sns.catplot(
  y = "method", x = "value", hue = "variable", col="name", kind="bar",
  data = df.melt(id_vars=["name","method"], value_vars=["nit", "nfev", "njev", "nhev"]).astype({"value":
)
```



Exercise 1

Try minimizing the following function using different optimization methods starting from $x_0 = [0, 0]$, which appears to work best?

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

Random starting locations

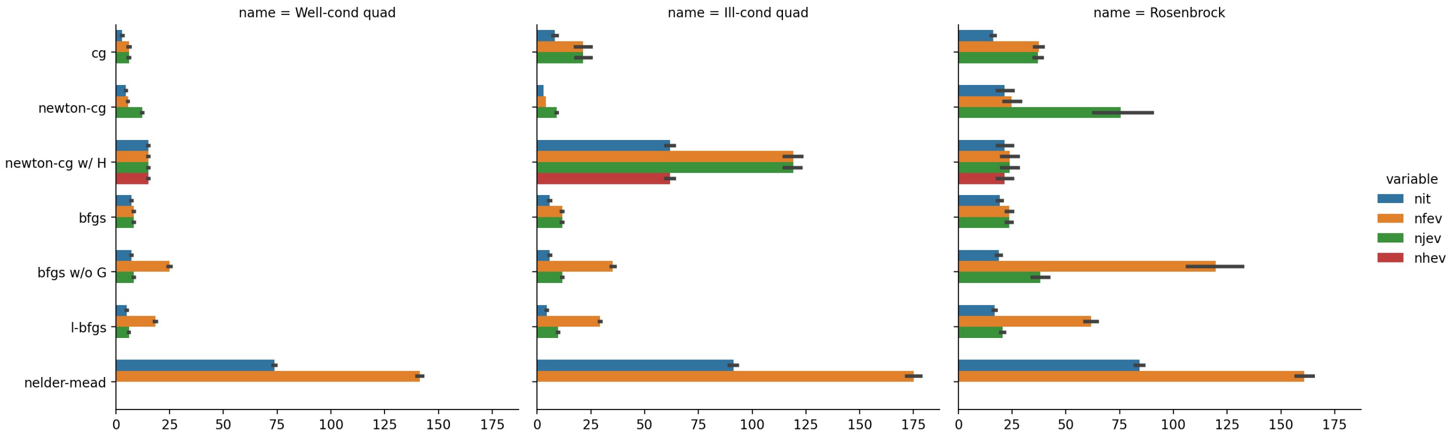
```
rng = np.random.default_rng(seed=1234)
x0s = rng.uniform(-2,2, (100,2))

df = pd.concat([
  run_collect(name, x0, cost_func, arg, skip=['na
  for name, cost_func, arg in zip(
    ("Well-cond quad", "Ill-cond quad", "Rosenbro
    (mk_quad, mk_quad, mk_rosenbrock),
    (0.7,0.02, None)
  )
  for x0 in x0s
])
```

```
df.drop(["message"], axis=1)
```

```
##               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    6     7     7  None     True
## 1   Well-cond quad      bfgs w/o G    6    21     7  None     True
## ..             ...             ...  ...   ...   ...   ...      ...
## 1       Rosenbrock  newton-cg w/ H   17    17    17    17     True
## 1       Rosenbrock            bfgs   26    31    31  None     True
## 1       Rosenbrock      bfgs w/o G   26    93    31  None     True
## 1       Rosenbrock          l-bfgs   18    57    19  None     True
## 1       Rosenbrock     nelder-mead   75   145  None  None     True
##
## [2100 rows x 7 columns]
```

Performance (random start)

```
sns.catplot(
  y = "method", x = "value", hue = "variable", col="name", kind="bar",
  data = df.melt(id_vars=["name","method"], value_vars=["nit", "nfev", "njev", "nhev"]).astype({"value":
).set(
  xlabel="", ylabel=""
)
```



MVN Cost Function

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

$$f(x) = \frac{1}{\det}$$

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

  def f(x):
    x_m = x - mu
    return -(norm_const *
      np.exp( -0.5 * (x_m.T @ Sigma_inv @ x_m).item() ))

  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,1)) @ (x_m.T @

  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, [0,0])
```

```
## 1.0536712127723509e-08
```

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

```
## 2.8653603296584263e-09
```

```
# 4d
f, grad, hess = mk_mvn(np.zeros(4), np.eye(4,4))
optimize.check_grad(f, grad, [0,0,0,0])
```

```
## 1.4901161193847656e-08
```

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

```
## 3.332000937312528e-08
```

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

```
## 7.079924060068647e-13
```

```
# 50d
f, grad, hess = mk_mvn(np.zeros(50), np.eye(50))
optimize.check_grad(f, grad, np.zeros(50))
```

```
## 5.268356063861754e-08
```

Testing optimizers

```
f, grad, hess = mk_mvn(np.zeros(4), np.eye(4,4))
optimize.minimize(fun=f, x0=[1,1,1,1], jac=grad, method="CG", t
```

```
##      fun: -1.0
##      jac: array([6.46744384e-16, 6.46744384e-16, 6.46744384e-
##  message: 'Optimization terminated successfully.'
##     nfev: 8
##      nit: 3
##     njev: 8
##   status: 0
##  success: True
##        x: array([6.46744384e-16, 6.46744384e-16, 6.46744384e-
```

```
optimize.minimize(fun=f, x0=[1,1,1,1], jac=grad, method="BFGS",
```

```
##       fun: -1.0
##  hess_inv: array([[1.00000001e+00, 1.32716307e-08, 1.32716307
##        [1.32716307e-08, 1.00000001e+00, 1.32716306e-08, 1.327
##        [1.32716307e-08, 1.32716306e-08, 1.00000001e+00, 1.327
##        [1.32716306e-08, 1.32716306e-08, 1.32716306e-08, 1.000
##       jac: array([9.80523384e-16, 9.80523381e-16, 9.80523384e
##   message: 'Optimization terminated successfully.'
##      nfev: 10
##       nit: 5
##      njev: 10
##    status: 0
##   success: True
##         x: array([9.80523384e-16, 9.80523381e-16, 9.80523384e
```

```
n = 20
f, grad, hess = mk_mvn(np.zeros(n), np.eye(n,n))
optimize.minimize(fun=f, x0=np.ones(n), jac=grad, method="CG",
```

```
##      fun: -1.0
##      jac: array([3.67929936e-19, 3.67929936e-19, 3.67929936e-19, 3.6
##        3.67929936e-19, 3.67929936e-19, 3.67929936e-19, 3.67929936e-1
##        3.67929936e-19, 3.67929936e-19, 3.67929936e-19, 3.67929936e-1
##        3.67929936e-19, 3.67929936e-19, 3.67929936e-19, 3.67929936e-1
##        3.67929936e-19, 3.67929936e-19, 3.67929936e-19, 3.67929936e-1
##  message: 'Optimization terminated successfully.'
##     nfev: 14
##      nit: 2
##     njev: 14
##   status: 0
##  success: True
##        x: array([3.67929936e-19, 3.67929936e-19, 3.67929936e-19, 3.6
##        3.67929936e-19, 3.67929936e-19, 3.67929936e-19, 3.67929936e-1
##        3.67929936e-19, 3.67929936e-19, 3.67929936e-19, 3.67929936e-1
##        3.67929936e-19, 3.67929936e-19, 3.67929936e-19, 3.67929936e-1
##        3.67929936e-19, 3.67929936e-19, 3.67929936e-19, 3.67929936e-1
```

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(
    ("2d", "5d", "10d", "20d", "50d"),
    (2, 5, 10, 20, 50)
  )
])
```

```
df.drop(["message"], axis=1)
```

```
##    name          method   nit   nfev  njev  nhev  success
## 1   2d              cg     3      6     6  None     True
## 1   2d       newton-cg     2      3     5     0     True
## 1   2d  newton-cg w/ H     2      2     2     2     True
## 1   2d            bfgs     4      8     8  None     True
## 1   2d      bfgs w/o G     4     24     8  None     True
## 1   2d          l-bfgs     5     21     7  None     True
## 1   2d     nelder-mead    75    157  None  None     True
## 1   5d              cg     3      8     8  None     True
## 1   5d       newton-cg     4      8    12     0     True
## 1   5d  newton-cg w/ H     4      7     7     4     True
## 1   5d            bfgs     5     11    11  None     True
## 1   5d      bfgs w/o G     5     72    12  None     True
## 1   5d          l-bfgs     4     54     9  None     True
## 1   5d     nelder-mead   297    529  None  None     True
## 1  10d              cg     2     24    22  None     True
## 1  10d       newton-cg     3      9    12     0     True
## 1  10d  newton-cg w/ H     3      8     8     3     True
## 1  10d            bfgs     3     12    12  None     True
## 1  10d      bfgs w/o G     2    121    11  None     True
## 1  10d          l-bfgs     3    110    10  None     True
## 1  10d     nelder-mead  1408   2000  None  None    False
## 1  20d              cg     2     14    14  None     True
## 1  20d       newton-cg     3     10    13     0     True
## 1  20d  newton-cg w/ H     3      9     9     3     True
## 1  20d            bfgs     2     15    15  None     True
## 1  20d      bfgs w/o G     2    315    15  None     True
## 1  20d          l-bfgs     3    210    10  None     True
```

Adding correlation

```python
def build_Sigma(n):
  S = np.full((n,n), 0.5)
  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-9/n,
    skip=['naive_newton', 'naive_cg']
  )
  for name, n in zip(
    ("2d", "5d", "10d", "20d", "50d"),
    (2, 5, 10, 20, 50)
  )
])
```

```python
df.drop(["message"], axis=1)
```

```
##    name          method  nit  nfev  njev  nhev  success
## 1    2d              cg   15    18    18  None    False
## 1    2d       newton-cg    5     7    12     0     True
## 1    2d  newton-cg w/ H    5     6     6     5     True
## 1    2d            bfgs    3     7     7  None     True
## 1    2d      bfgs w/o G    3    24     8  None    False
## 1    2d          l-bfgs    5    21     7  None     True
## 1    2d     nelder-mead   73   145  None  None     True
## 1    5d              cg    5    19    19  None     True
## 1    5d       newton-cg    5     7    12     0     True
## 1    5d  newton-cg w/ H    5     6     6     5     True
## 1    5d            bfgs    5     8     8  None     True
## 1    5d      bfgs w/o G    7   528    86  None    False
## 1    5d          l-bfgs    4    54     9  None     True
## 1    5d     nelder-mead  224   421  None  None     True
## 1   10d              cg   10    23    23  None    False
## 1   10d       newton-cg    5     6    11     0     True
## 1   10d  newton-cg w/ H    5     5     5     5     True
## 1   10d            bfgs    4     9     9  None     True
## 1   10d      bfgs w/o G    6   132    12  None     True
## 1   10d          l-bfgs    4    99     9  None     True
## 1   10d     nelder-mead 1151  1754  None  None     True
## 1   20d              cg    5    25    25  None     True
## 1   20d       newton-cg    4     5     9     0     True
## 1   20d  newton-cg w/ H    4     4     4     4     True
## 1   20d            bfgs    5     9     9  None     True
## 1   20d      bfgs w/o G    6   210    10  None    False
## 1   20d          l-bfgs    5   189     9  None     True
```

```
df
```

```
##    name          method  nit  nfev  njev  nhev  success                                    message
## 1    2d              cg   15    18    18  None    False  Desired error not necessarily achieved due to ...
## 1    2d       newton-cg    5     7    12     0     True            Optimization terminated successfully.
## 1    2d  newton-cg w/ H    5     6     6     5     True            Optimization terminated successfully.
## 1    2d            bfgs    3     7     7  None     True            Optimization terminated successfully.
## 1    2d      bfgs w/o G    3    24     8  None    False  Desired error not necessarily achieved due to ...
## 1    2d          l-bfgs    5    21     7  None     True   CONVERGENCE: NORM_OF_PROJECTED_GRADIENT_<=_PGTOL
## 1    2d     nelder-mead   73   145  None  None     True            Optimization terminated successfully.
## 1    5d              cg    5    19    19  None     True            Optimization terminated successfully.
## 1    5d       newton-cg    5     7    12     0     True            Optimization terminated successfully.
## 1    5d  newton-cg w/ H    5     6     6     5     True            Optimization terminated successfully.
## 1    5d            bfgs    5     8     8  None     True            Optimization terminated successfully.
## 1    5d      bfgs w/o G    7   528    86  None    False  Desired error not necessarily achieved due to ...
## 1    5d          l-bfgs    4    54     9  None     True    CONVERGENCE: REL_REDUCTION_OF_F_<=_FACTR*EPSMCH
## 1    5d     nelder-mead  224   421  None  None     True            Optimization terminated successfully.
## 1   10d              cg   10    23    23  None    False  Desired error not necessarily achieved due to ...
## 1   10d       newton-cg    5     6    11     0     True            Optimization terminated successfully.
## 1   10d  newton-cg w/ H    5     5     5     5     True            Optimization terminated successfully.
## 1   10d            bfgs    4     9     9  None     True            Optimization terminated successfully.
## 1   10d      bfgs w/o G    6   132    12  None     True            Optimization terminated successfully.
## 1   10d          l-bfgs    4    99     9  None     True    CONVERGENCE: REL_REDUCTION_OF_F_<=_FACTR*EPSMCH
## 1   10d     nelder-mead 1151  1754  None  None     True            Optimization terminated successfully.
## 1   20d              cg    5    25    25  None     True            Optimization terminated successfully.
## 1   20d       newton-cg    4     5     9     0     True            Optimization terminated successfully.
## 1   20d  newton-cg w/ H    4     4     4     4     True            Optimization terminated successfully.
```

What's going on?

```
n = 50
f, grad, hess = mk_mvn(np.zeros(n), build_Sigma(n))
```

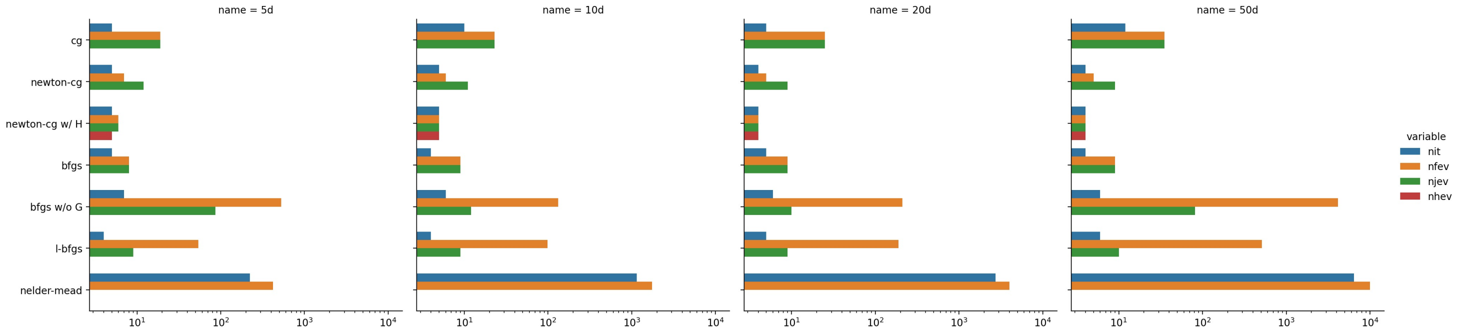
```
optimize.minimize(f, np.ones(n), jac=grad, method="CG", tol=1e-
```

```
##      fun: -1.0
##      jac: array([ 1.12332277e-10,  8.72935916e-11, -3.3297223
##         8.69788321e-11, -4.18555930e-11,  2.10910902e-11, -1.
##        -1.56070646e-10, -1.73949531e-11, -2.46648960e-10, -9.
##        -2.69689780e-10, -2.53304579e-10, -2.14723226e-10, -2.
##        -6.02970488e-11, -2.62054475e-10, -1.88641458e-10,  1.
##        -1.37095135e-10, -1.99919295e-10, -1.62557241e-10, -1.
##        -1.82270787e-10, -1.70384755e-10, -1.70436047e-10, -1.
##        -2.19757053e-10, -1.52952018e-10, -2.13786756e-10, -1.
##        -2.04463656e-10, -2.43201436e-10, -2.04465933e-10, -2.
##        -2.78825917e-10, -1.90142568e-10, -2.29375511e-10, -2.
##        -2.43124923e-10, -2.29273546e-10, -2.27412666e-10, -2.
##        -2.63387506e-10, -2.38475808e-10, -2.78480808e-10, -2.
##        -2.58275285e-10, -2.50752825e-10])
##  message: 'Desired error not necessarily achieved due to prec
##     nfev: 35
##      nit: 12
##     njev: 35
##   status: 2
##  success: False
##        x: array([-4.12110457e-09, -4.13362391e-09, -4.3437568
##        -4.13378129e-09, -4.19819851e-09, -4.16672516e-09, -4.
```

```
optimize.minimize(f, np.ones(n), jac=grad, method="BFGS", tol=1
```

```
##      fun: -1.0
## hess_inv: array([[1.49000053, 0.49000053, 0.49000053, ..., 0.490000
##        0.49000053],
##       [0.49000053, 1.49000053, 0.49000053, ..., 0.49000053, 0.49000
##        0.49000053],
##       [0.49000053, 0.49000053, 1.49000053, ..., 0.49000053, 0.49000
##        0.49000053],
##       ...,
##       [0.49000053, 0.49000053, 0.49000053, ..., 1.49000053, 0.49000
##        0.49000053],
##       [0.49000053, 0.49000053, 0.49000053, ..., 0.49000053, 1.49000
##        0.49000053],
##       [0.49000053, 0.49000053, 0.49000053, ..., 0.49000053, 0.49000
##        1.49000053]])
##      jac: array([-2.73970642e-12, -2.74307089e-12, -2.79949727e-12,
##       -2.74292110e-12, -2.76027528e-12, -2.75166780e-12, -2.7565265
##       -2.77555684e-12, -2.75705655e-12, -2.78801410e-12, -2.7668121
##       -2.79098818e-12, -2.78892573e-12, -2.78363550e-12, -2.7839237
##       -2.76262526e-12, -2.78983521e-12, -2.77993127e-12, -2.7531132
##       -2.77310882e-12, -2.78152071e-12, -2.77661830e-12, -2.7730126
##       -2.77931259e-12, -2.77777327e-12, -2.77782252e-12, -2.7743122
##       -2.78416334e-12, -2.77532276e-12, -2.78339434e-12, -2.7801260
```

```python
sns.catplot(
  y = "method", x = "value", hue = "variable", col="name", kind="bar",
  data = df.melt(
    id_vars=["name","method"], value_vars=["nit", "nfev", "njev", "nhev"]
  ).astype(
    {"value": "float64"}
  ).query(
    "name != '2d'"
  )
).set(
  xscale="log", xlabel="", ylabel=""
)
```



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
- 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