# Optimization (cont.)

## Lecture 22

Dr. Colin Rundel

# Method Variants

# Method - CG in scipy

Scipy's optimize module implements the conjugate gradient algorithm using a variant proposed by Polak and Ribiere, this version does not use the Hessian when calculating the next step. The specific changes are:

- $\alpha_k$ is calculated via a line search along the direction $p_k$
- and the $\beta_{k+1}$ calculation is replaced as follows

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

# Method - Newton-CG & BFGS

These are both variants of Newton's method but do not require the Hessian (though it can optionally be provided to Newton-CG).

- Newton-Conjugate Gradient (Newton-CG) algorithm uses a conjugate gradient algorithm to (approximately) invert the local Hessian
- The Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm iteratively approximates the inverse Hessian
  - Gradient is also not required and can be approximated using finite differences
- Newton-CG also comes in a trust-region variant (`"trust-ncg"`)
  - Uses a trust-region approach to determine the step size rather than backtracking line search

# Method - L-BFGS-B

Limited-memory BFGS (L-BFGS-B) is a variant of BFGS designed for problems with large numbers of parameters.

- Rather than storing and updating a full $n \times n$ approximation to the inverse Hessian, L-BFGS stores only the last $m$ updates (typically $m \approx 5$–$20$) and uses these for the approximation
- This reduces memory usage from $O(n^2)$ to $O(mn)$, making it practical for high-dimensional problems
- The "B" in L-BFGS-B refers to support for bound constraints on the parameters (i.e. $l_i \leq x_i \leq u_i$)

# 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 |
| `"trust-ncg"` | Trust-region Newton-CG method | ✓ | 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 | ✗ | ✗ |

SciPy's `minimize()` also supports other methods including `"Powell"` (gradient-free direction set), `"SLSQP"` (constrained

# R's `optim()`

R's `optim()` function (from the `stats` package) provides a similar set of general-purpose optimization methods:

| R Method | SciPy Equivalent | Description | Gradient | Bounds |
|---|---|---|---|---|
| `"Nelder-Mead"` | `"Nelder-Mead"` | Nelder-Mead simplex (default) | ✗ | ✗ |
| `"BFGS"` | `"BFGS"` | Quasi-Newton (Broyden-Fletcher-Goldfarb-Shanno) | Optional | ✗ |
| `"CG"` | `"CG"` | Conjugate gradient (Fletcher-Reeves, Polak-Ribiere, or Beale-Sorenson) | Optional | ✗ |
| `"L-BFGS-B"` | `"L-BFGS-B"` | Limited-memory BFGS with box constraints | Optional | ✓ |
| `"SANN"` | — | Simulated annealing (stochastic global optimization) | ✗ | ✗ |

# R's `optim()` (cont.)

Key differences from SciPy's `optimize.minimize()`:

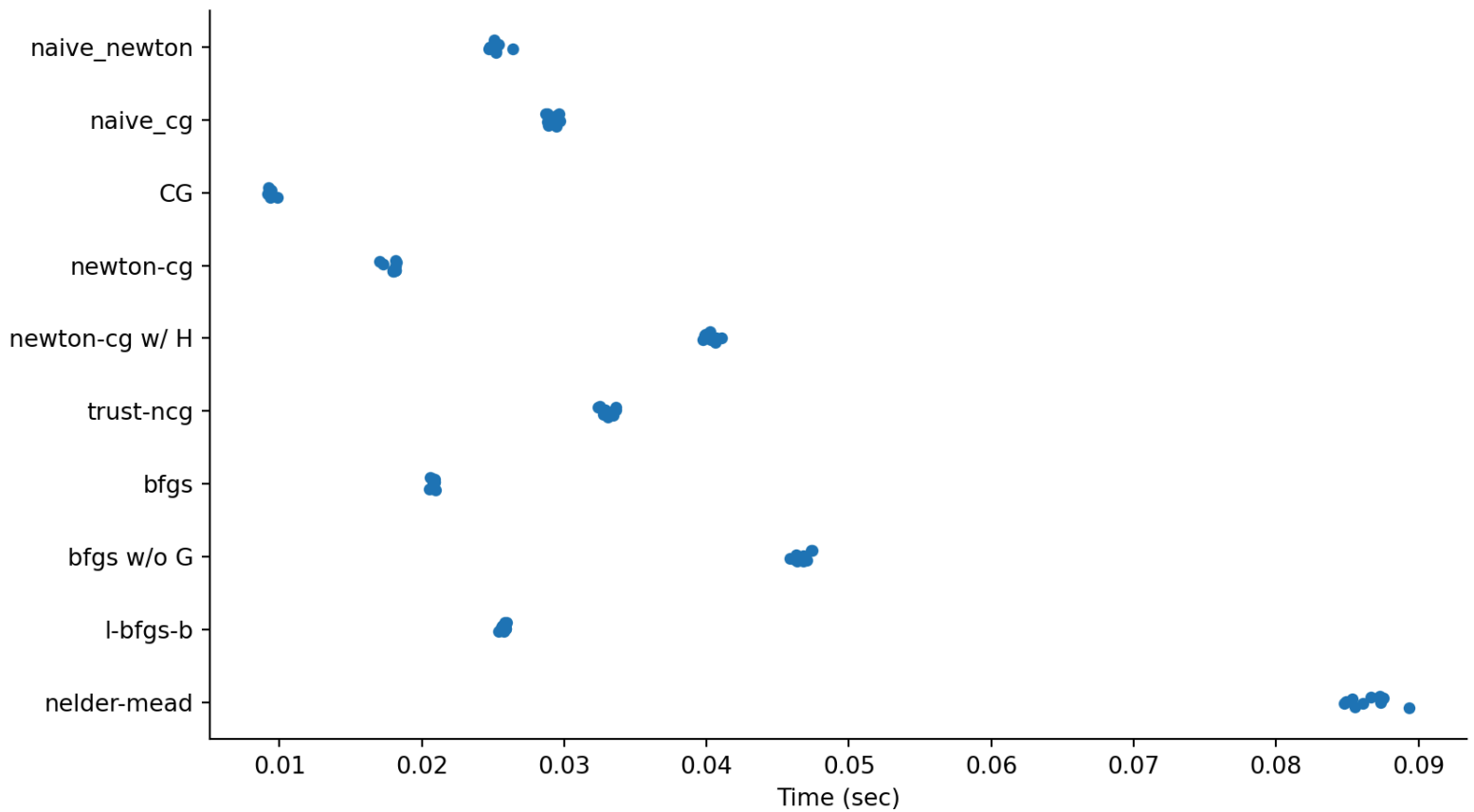
- R's `optim()` does not support providing a Hessian to any method (but can return a numerically approximated Hessian via `hessian=TRUE`)
- R has no equivalent to SciPy's `"Newton-CG"` method
- R includes `"SANN"` (simulated annealing) - a stochastic global optimizer not available in SciPy's `minimize()`
- The `optimx` packages provide access to additional methods beyond those in `optim()` - many additional packages available on CRAN

# Methods collection

```python
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=tol),
    "newton-cg":      lambda: optimize.minimize(f, x, jac=grad, hess=None, method="Newton-CG"
    "newton-cg w/ H": lambda: optimize.minimize(f, x, jac=grad, hess=hess, method="Newton-CG"
    "trust-ncg":      lambda: optimize.minimize(f, x, jac=grad, hess=hess, method="trust-ncg"
    "bfgs":           lambda: optimize.minimize(f, x, jac=grad, method="BFGS", tol=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=tol)
  }
```

# Method Timings

```
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
})
```



# Timings across cost functions

Implementation | Results

```python
def time_cost_func(n, x, name, cost_func, *args):
  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)
  )

x = (1.6, 1.1)

time_cost_df = pd.concat([
  time_cost_func(10, x, "Well-cond quad", mk_quad, 0.7),
  time_cost_func(10, x, "Ill-cond quad", mk_quad, 0.02),
  time_cost_func(10, x, "Rosenbrock", mk_rosenbrock)
])
```

# Random starting locations

Implementation | Results

```
pts = np.random.default_rng(seed=1234).uniform(-2,2, (20,2))
df = pd.concat([
  pd.concat([
    time_cost_func(3, x, "Well-cond quad", mk_quad, 0.7),
    time_cost_func(3, x, "Ill-cond quad", mk_quad, 0.02),
    time_cost_func(3, x, "Rosenbrock", mk_rosenbrock)
  ])
  for x in pts
])
```

# Profiling - BFGS (cProfile)

```
1  import cProfile
2
3  f, grad, hess = mk_quad(0.7)
4  def run():
5    for i in range(500):
6      optimize.minimize(fun = f, x0 = (1.6, 1.1), jac=grad, method="BFGS", tol=1e-11)
7
8  cProfile.run('run()', sort="tottime")
```

```
         549504 function calls in 0.209 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      500    0.039    0.000    0.204    0.000 _optimize.py:1345(_minimize_bfgs)
    29000    0.018    0.000    0.018    0.000 {method 'reduce' of 'numpy.ufunc' objects}
    13000    0.010    0.000    0.031    0.000 _optimize.py:195(vecnorm)
     5000    0.009    0.000    0.020    0.000 <string>:3(f)
    18000    0.008    0.000    0.020    0.000 fromnumeric.py:66(_wrapreduction)
     5000    0.008    0.000    0.010    0.000 <string>:9(gradient)
     4500    0.007    0.000    0.103    0.000 _dcsrch.py:201(__call__)
    10000    0.007    0.000    0.018    0.000 numeric.py:2522(array_equal)
     4500    0.006    0.000    0.034    0.000 _linesearch.py:86(derphi)
    13000    0.005    0.000    0.020    0.000 fromnumeric.py:2304(sum)
     5000    0.005    0.000    0.019    0.000 _differentiable_functions.py:35(__call__)
     4500    0.004    0.000    0.109    0.000 _linesearch.py:100(scalar_search_wolfe1)
     9000    0.004    0.000    0.004    0.000 _dcsrch.py:269(_iterate)
     4500    0.004    0.000    0.113    0.000 _linesearch.py:37(line_search_wolfe1)
```

```
 5000    0.004    0.000    0.026    0.000 _util.py:600(__call__)
 4500    0.004    0.000    0.057    0.000 _linesearch.py:82(phi)
15500    0.004    0.000    0.005    0.000 {built-in method numpy.array}
```

# Profiling - BFGS (pyinstrument)

Code | Output

```
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.write_html("Lec22_bfgs_quad.html")
```

# Profiling - Nelder-Mead (pyinstrument)

Code | Output

```
1  from pyinstrument import Profiler
2
3  f, grad, hess = mk_quad(0.7)
4
5  profiler = Profiler(interval=0.00001)
6
7  profiler.start()
8  opt = optimize.minimize(fun = f, x0 = (1.6, 1.1), method="Nelder-Mead", tol=1e-11)
9  p = profiler.stop()
10
11 profiler.write_html("Lec22_nm_quad.html")
```

# `optimize.minimize()` output

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

```
1 optimize.minimize(
2   fun = f, x0 = (1.6, 1.1),
3   jac=grad, method="BFGS"
4 )
```

```
 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(
2   fun = f, x0 = (1.6, 1.1),
3   jac=grad, hess=hess, method="Newton-CG"
4 )
```

```
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, 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
```

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

```
      message: Optimization terminated successf
      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]]
```

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

```python
optimize.minimize(
  fun = f, x0 = (1.6, 1.1),
  jac=grad, hess=hess, method="trust-ncg"
)
```

```
message: Optimization terminated successfully.
success: True
 status: 0
    fun: 1.4831619534744353e-09
      x: [ 0.000e+00  9.577e-05]
    nit: 9
    jac: [ 0.000e+00  3.097e-05]
   nfev: 10
   njev: 10
   nhev: 9
   hess: [[ 6.600e-01  0.000e+00]
          [ 0.000e+00  4.620e-01]]
```

```python
optimize.minimize(
  fun = f, x0 = (1.6, 1.1),
  method="L-BFGS-B"
)
```

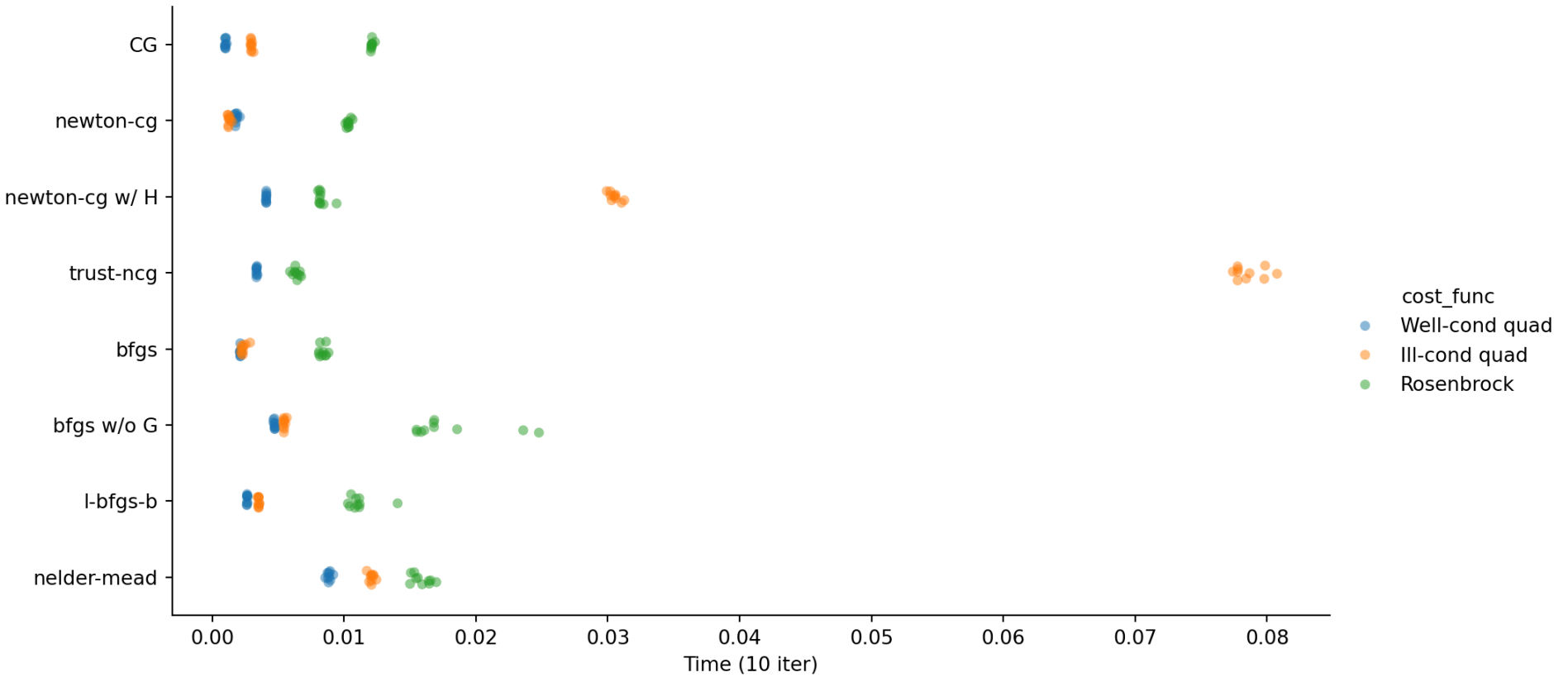
```
message: CONVERGENCE: NORM OF PROJECTED GRADIE
success: True
 status: 0
    fun: 3.417411495023028e-12
      x: [ 2.832e-06  2.183e-06]
    nit: 5
    jac: [ 1.872e-06  7.077e-07]
   nfev: 18
   njev: 6
hess_inv: <2x2 LbfgsInvHessProduct with dtype=f
```

# Collect

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

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

    x = methods[method]()
    time = timeit.Timer(methods[method]).repeat(

    d = {
      "name":    name,
      "method":  method,
      "nit":     x["nit"],
      "nfev":    x["nfev"],
      "njev":    x.get("njev"),
      "nhev":    x.get("nhev"),
      "success": x["success"],
      "time":    [time]
    }
    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,
    skip=['naive_newton', 'naive_cg']
  )
  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)
  )
])
```

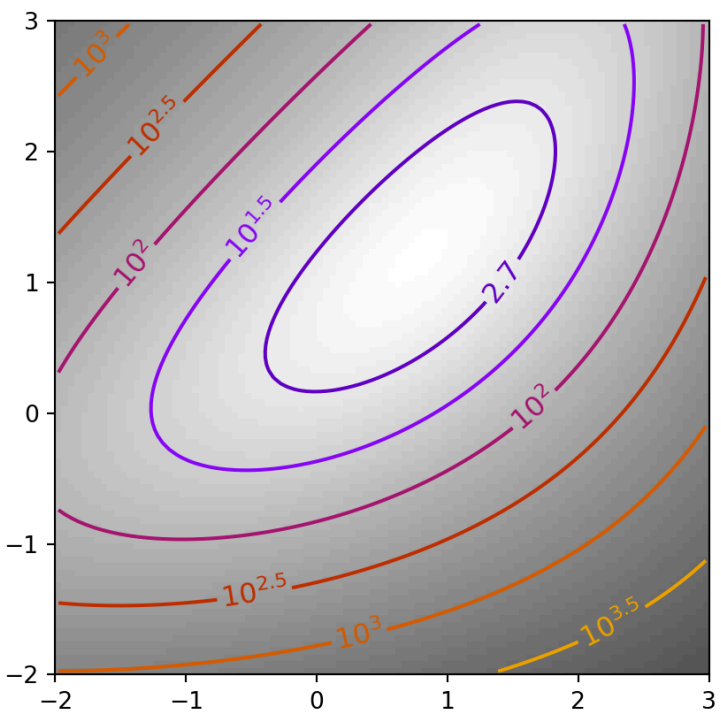
Runtimes | Function calls



# 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 $\mu$ and the $n \times n$ covariance matrix $\Sigma$,

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

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

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

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.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
      )
    ).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 @ Sigma_inv).reshape((1,n))
      - Sigma_inv
    )
```

# 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()`.

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

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

```
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), np.eye(10)
)
```

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

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

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

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

# Gradient checking (wrong gradient)

```
wrong_grad = lambda x: 5*grad(x)
```

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

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

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

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

```
np.float64(0.007419234855235744)
```

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

```
optimize.check_grad(f, wrong_grad, np.zeros(10))
```

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

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

```
np.float64(8.703385925704049e-06)
```

Why does `np.ones()` detect an issue but `np.zeros()` does not?

# Hessian checking

Note that since the gradient of the gradient 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`.

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

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

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

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

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

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

```
optimize.check_grad(grad, hess, np.zeros(10)
```

```
np.float64(4.2856853864461685e-20)
```

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

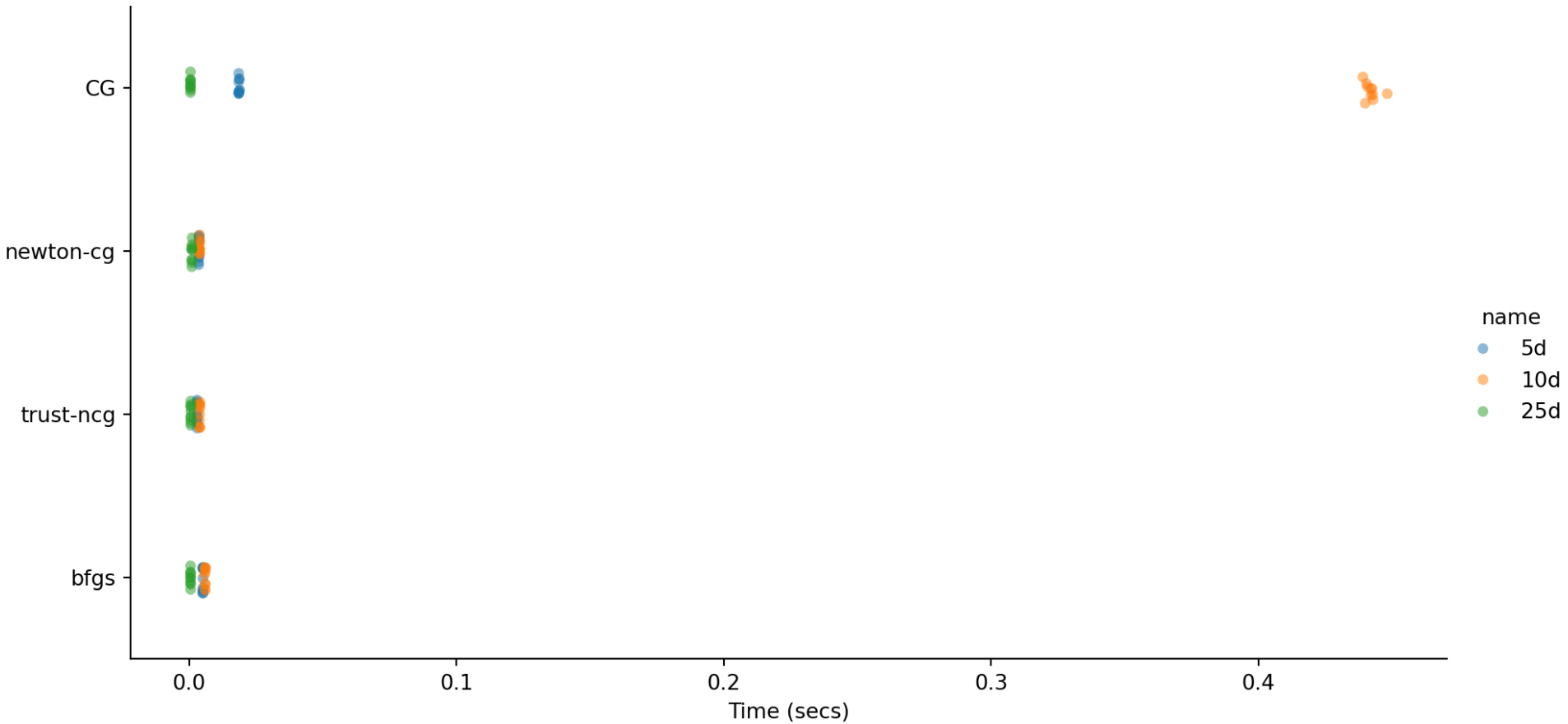
```
np.float64(8.551196009381392e-14)
```

# Unit MVNs

```
rng = np.random.default_rng(seed=1234)
runif = rng.uniform(-1,1, size=25)

df = pd.concat([
  run_collect(
    name, runif[:n], mk_mvn,
    np.zeros(n), np.eye(n),
    tol=1e-10,
    skip=['naive_newton', 'naive_cg', 'bfgs w/o G', 'newton-cg w/ H', 'l-bfgs-b', 'nelder-mea
  )
  for name, n in zip(
    ("5d", "10d", "25d"),
    (5, 10, 25)
  )
])
```

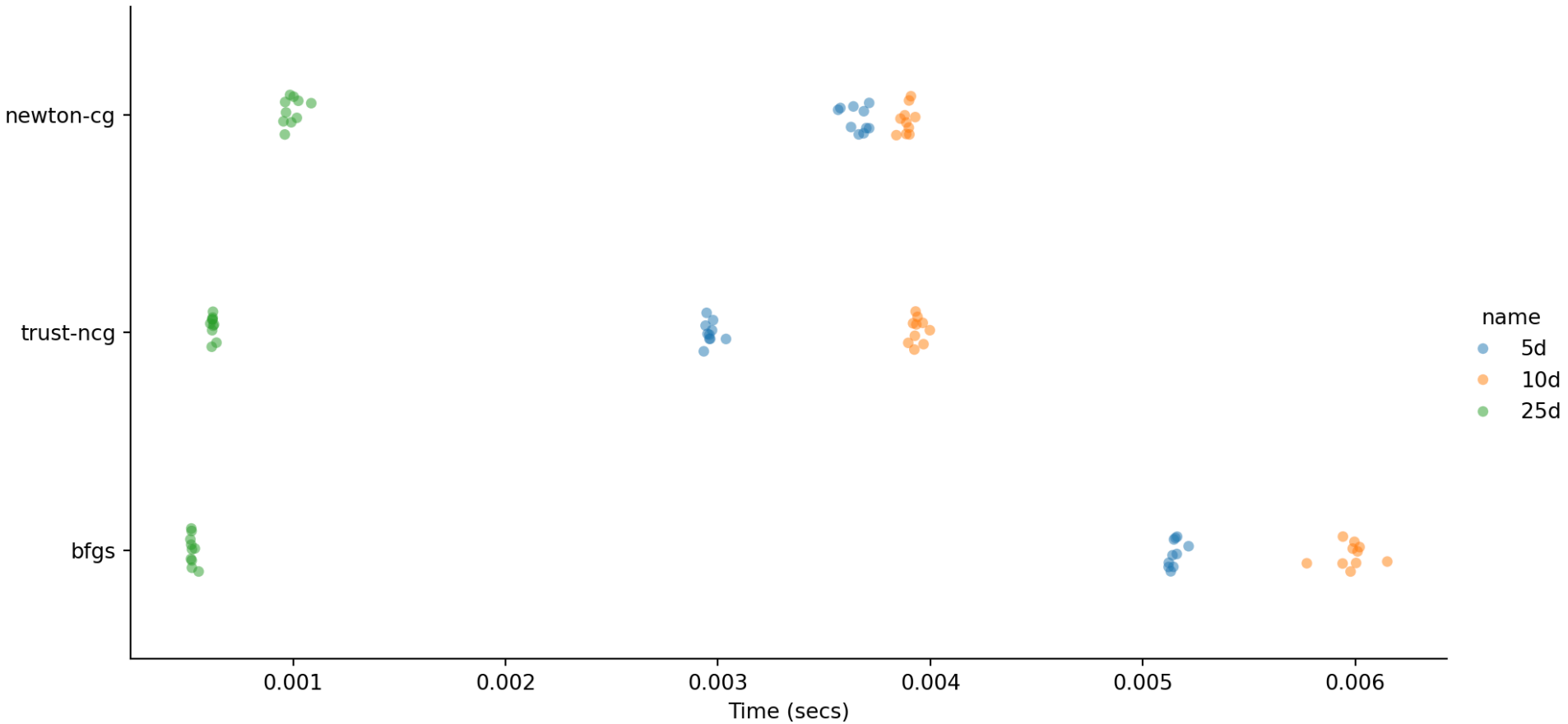
# Performance (Unit MVNs)

Run times | Function calls



# Performance (Unit MVNs, excl. CG)

Run times | Function calls



# What's going on? (good)

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

```
optimize.minimize(
  f, runif[:n], jac=grad,
  method="newton-cg", tol=1e-10
)
```

```
message: Optimization terminated successfully.
success: True
 status: 0
    fun: -0.010105326013811651
      x: [ 5.071e-11 -1.274e-11  4.502e-11 -2.5
    nit: 4
    jac: [ 5.125e-13 -1.288e-13  4.550e-13 -2.5
   nfev: 5
   njev: 9
   nhev: 0
```

```
optimize.minimize(
  f, runif[:n], jac=grad,
  method="bfgs", tol=1e-10
)
```

```
message: Optimization terminated successfully.
success: True
 status: 0
    fun: -0.010105326013811651
      x: [-2.482e-13  6.237e-14 -2.203e-13  1.
    nit: 4
    jac: [-2.508e-15  6.303e-16 -2.227e-15  1.
hess_inv: [[ 4.463e+01 -1.096e+01 ... -2.181e+0
           [-1.096e+01  3.756e+00 ...  5.481e+0
           ...
           [-2.181e+01  5.481e+00 ...  1.190e+0
           [-1.656e+01  4.161e+00 ...  8.276e+0
   nfev: 10
   njev: 10
```

# What's going on? (okay)

```python
n = 10
f, grad, hess = mk_mvn(np.zeros(n), np.eye(n))
```

```python
optimize.minimize(
  f, runif[:n], jac=grad,
  method="newton-cg", tol=1e-10
)
```

```
message: Optimization terminated successfully.
success: True
 status: 0
    fun: -0.00010211745783654051
      x: [-8.223e-04  2.067e-04 -7.301e-04  4.1
           6.588e-04  4.454e-04  3.130e-04 -8.0
    nit: 3
    jac: [-8.397e-08  2.110e-08 -7.455e-08  4.1
           6.727e-08  4.549e-08  3.196e-08 -8.1
   nfev: 6
   njev: 9
   nhev: 0
```

```python
optimize.minimize(
  f, runif[:n], jac=grad,
  method="bfgs", tol=1e-10
)
```

```
message: Optimization terminated successfully.
success: True
 status: 0
    fun: -0.00010211761384541865
      x: [-2.347e-09  5.898e-10 -2.084e-09  1.
           1.880e-09  1.271e-09  8.933e-10 -2.
    nit: 2
    jac: [-2.396e-13  6.023e-14 -2.128e-13  1.
           1.920e-13  1.298e-13  9.122e-14 -2.
hess_inv: [[ 1.685e+04 -4.235e+03 ...  1.641e+0
            [-4.235e+03  1.065e+03 ... -4.123e+0
            ...
            [ 1.641e+04 -4.123e+03 ...  1.597e+0
            [-8.356e+03  2.100e+03 ... -8.134e+0
   nfev: 14
   njev: 14
```

# What's going on? (bad)

```
n = 25
f, grad, hess = mk_mvn(np.zeros(n), np.eye(n))
```

```
optimize.minimize(
  f, runif[:n], jac=grad,
  method="newton-cg", tol=1e-10
)
```

```
message: Optimization terminated successfully.
success: True
 status: 0
    fun: -1.2867324357023428e-12
      x: [ 9.534e-01 -2.396e-01 ...  2.220e-01
    nit: 1
    jac: [ 1.227e-12 -3.083e-13 ...  2.857e-13
   nfev: 1
   njev: 1
   nhev: 0
```

```
optimize.minimize(
  f, runif[:n], jac=grad,
  method="bfgs", tol=1e-10
)
```

```
message: Optimization terminated successfully.
success: True
 status: 0
    fun: -1.2867324357023428e-12
      x: [ 9.534e-01 -2.396e-01 ...  2.220e-01
    nit: 0
    jac: [ 1.227e-12 -3.083e-13 ...  2.857e-13
hess_inv: [[1 0 ... 0 0]
           [0 1 ... 0 0]
           ...
           [0 0 ... 1 0]
           [0 0 ... 0 1]]
   nfev: 1
   njev: 1
```

# All bad?

```python
optimize.minimize(
  f, runif[:n], jac=grad,
  method="nelder-mead", tol=1e-10
)
```

```
      message: Maximum number of function evaluations has been exceeded.
      success: False
       status: 1
          fun: -5.2161537392613975e-11
            x: [ 7.181e-02 -3.136e-01 ...  3.193e-01  3.222e-02]
          nit: 4136
         nfev: 5000
final_simplex: (array([[ 7.181e-02, -3.136e-01, ...,  3.193e-01,
                         3.222e-02],
                       [ 7.275e-02, -3.237e-01, ...,  3.218e-01,
                         2.192e-02],
                       ...,
                       [ 8.238e-02, -3.143e-01, ...,  3.247e-01,
                         2.232e-02],
                       [ 8.105e-02, -3.178e-01, ...,  3.119e-01,
                         3.078e-02]], shape=(26, 25)), array([-5.216e-11, -5.216e-11, ..., -5.213e
                      shape=(26,)))
```

# Options (newton-cg)

```
1 optimize.show_options(solver="minimize", method="newton-cg")
```

```
Minimization of scalar function of one or more variables using the
Newton-CG algorithm.

Note that the `jac` parameter (Jacobian) is required.

Options
-------
disp : bool
    Set to True to print convergence messages.
xtol : float
    Average relative error in solution `xopt` acceptable for
    convergence.
maxiter : int
    Maximum number of iterations to perform.
eps : float or ndarray
    If `hessp` is approximated, use this value for the step size.
return_all : bool, optional
    Set to True to return a list of the best solution at each of the
    iterations.
c1 : float, default: 1e-4
    Parameter for Armijo condition rule.
c2 : float, default: 0.9
```

# 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.
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
```

# 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.
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
```

# 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)
```

# Fixing our tolerances?

```python
n = 25
f, grad, hess = mk_mvn(np.zeros(n), np.eye(n))
```

```python
optimize.minimize(
  f, runif[:n], jac=grad,
  method="newton-cg", tol=1e-16
)
```

```
message: Optimization terminated successfully.
success: True
 status: 0
    fun: -1.2867324357023428e-12
      x: [ 9.534e-01 -2.396e-01 ...  2.220e-01
    nit: 1
    jac: [ 1.227e-12 -3.083e-13 ...  2.857e-13
   nfev: 1
   njev: 1
   nhev: 0
```

```python
optimize.minimize(
  f, runif[:n], jac=grad,
  method="bfgs", tol=1e-16
)
```

```
message: Optimization terminated successfully.
success: True
 status: 0
    fun: -1.0537841099018322e-10
      x: [-2.645e-08  6.648e-09 ... -6.160e-09
    nit: 3
    jac: [-2.788e-18  7.006e-19 ... -6.492e-19
hess_inv: [[ 9.790e+08 -2.461e+08 ...  2.280e+0
           [-2.461e+08  6.184e+07 ... -5.730e+0
           ...
           [ 2.280e+08 -5.730e+07 ...  5.310e+0
           [-9.034e+08  2.270e+08 ... -2.104e+0
   nfev: 27
   njev: 27
```

# Limits of floating point precision

Every type of floating point value has finite precision due to the limitations of how it is represented. This value is typically referred to as the machine epsilon — the smallest possible spacing between 1.0 and the next representable floating-point number.

```
1 np.finfo(np.float64).eps
```

```
np.float64(2.220446049250313e-16)
```

```
1 np.finfo(np.float32).eps
```

```
np.float32(1.1920929e-07)
```

```
1 np.finfo(np.float16).eps
```

```
np.float16(0.000977)
```

```
1 1+np.finfo(np.float64).eps > 1
```

```
np.True_
```

```
1 1+np.finfo(np.float64).eps/2 > 1
```

```
np.False_
```

# Fixes?

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

  # Returns the negative density (since we want the max not min)
  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))
```

```python
n = 25
f, grad, hess = mk_prop_mvn(np.zeros(n), np.eye(n))
```

```python
optimize.minimize(
  f, runif[:n], jac=grad,
  method="newton-cg", tol=1e-10
)
```

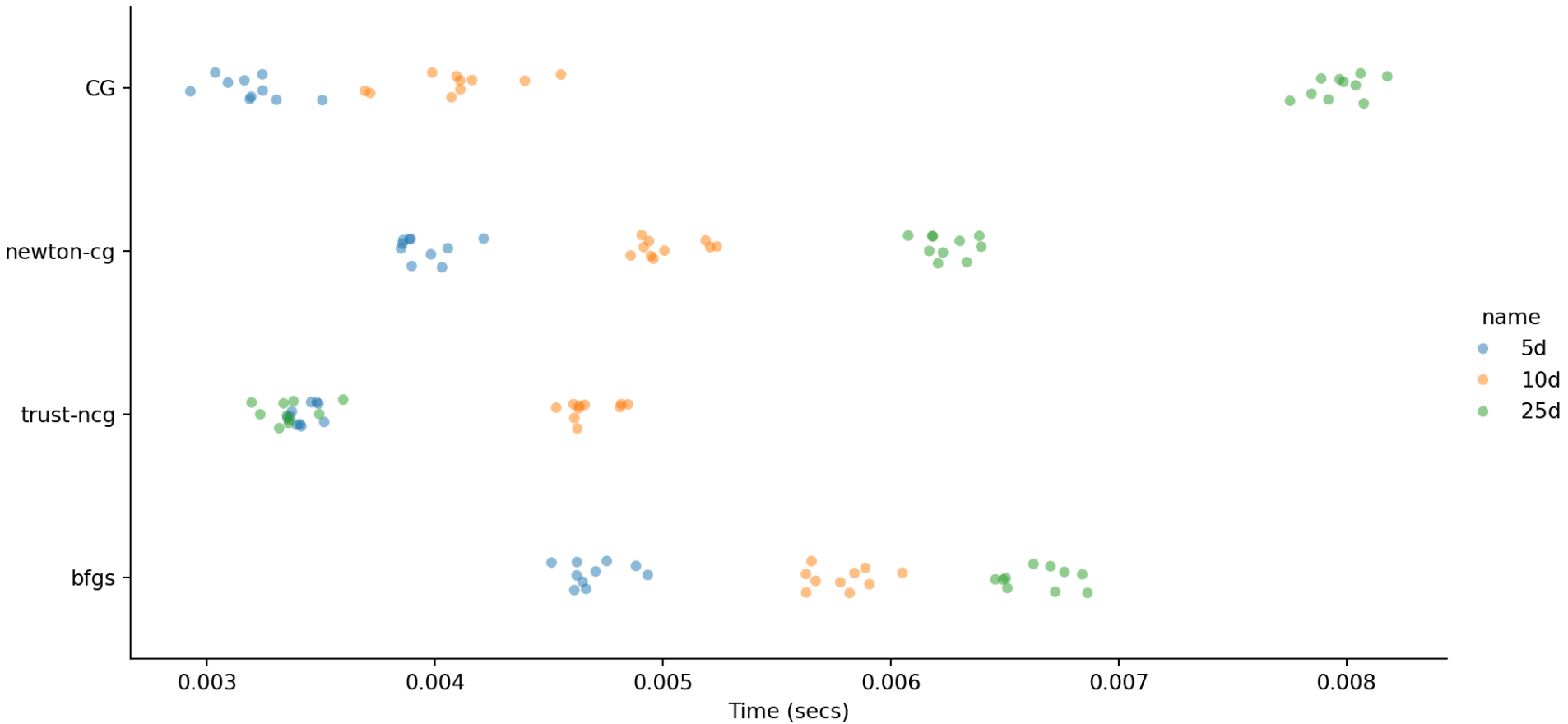
```
message: Optimization terminated successfully.
success: True
 status: 0
    fun: -1.0
      x: [-3.701e-16  9.302e-17 ... -8.619e-17
   nit: 4
   jac: [-3.701e-16  9.302e-17 ... -8.619e-17
  nfev: 10
  njev: 14
  nhev: 0
```

```python
optimize.minimize(
  f, runif[:n], jac=grad,
  method="bfgs", tol=1e-10
)
```

```
message: Optimization terminated successfully.
success: True
 status: 0
    fun: -1.0
      x: [-3.040e-15  7.639e-16 ... -7.079e-16
   nit: 4
   jac: [-3.040e-15  7.639e-16 ... -7.079e-16
hess_inv: [[ 1.000e+00 -6.660e-09 ...  6.171e-0
           [-6.660e-09  1.000e+00 ... -1.551e-0
           ...
           [ 6.171e-09 -1.551e-09 ...  1.000e+0
           [-2.445e-08  6.145e-09 ... -5.694e-0
  nfev: 11
  njev: 11
```

# Performance

Run times | Function calls



# 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
  - Be aware of the scale of your function and gradients
- 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