Intro. Comp. for Data Science (FMI08)

Dr. Nono Saha

June 28, 2023

Max Planck Institute for Mathematics in the Sciences
University of Leipzig/ScaDS.AI

Spring 2023

Course plan

1. Numerical optimization using `scipy`
2. Method Timings
3. Small exercise
4. General advice

Numerical optimization using `scipy`

Method summary

Scipy method	Description	Gadient	Hessian
---	Newton's method (naive)	Yes	Yes
---	Conjugate Gradient (naive)	Yes	Yes
CG	Nonlinear Conjugate Gradient (Polak and Ribiere variation)	Yes	No
Newton-CG	Truncated Newton method (Newton w/ CG step direction)	Yes	Optional
BFGS	Broyden, Fletcher, Goldfarb, and Shanno (Quasi-newton method)	Optional	No
L-BFGS-B	Limited-memory BFGS (Quasi-newton method)	Optional	No
Nelder-Mead	Nelder-Mead simplex reflection method	No	No

scipy: methods collection

```python
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)}
```

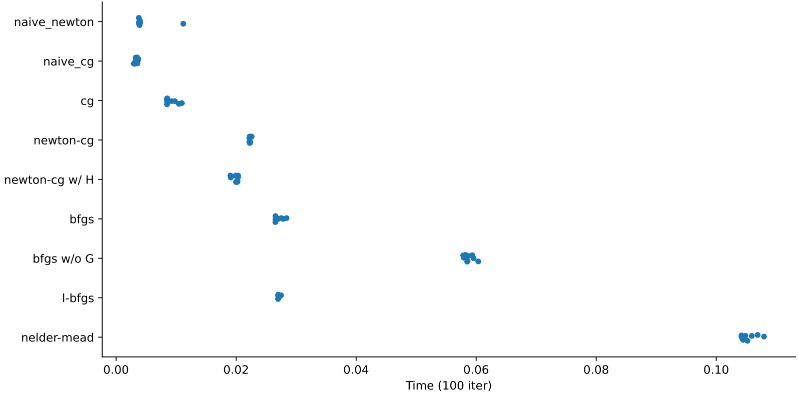
Method timings

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
.....
```

Method timing: plotting our data

```
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()
```

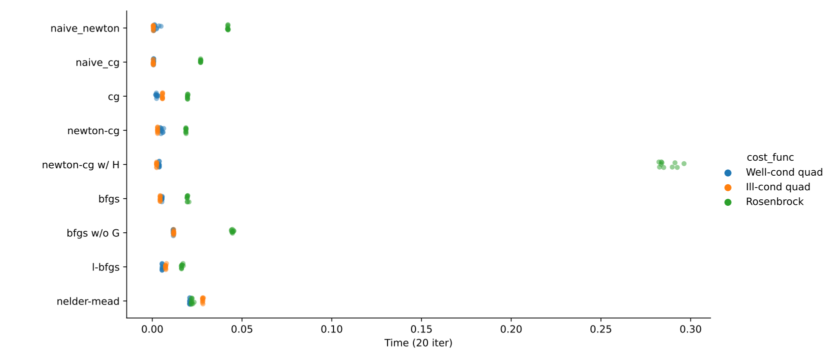


Timings across cost functions

```
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
    methods}).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
.....
```

Timing across cost functions: plotting

```python
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")

#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")
```

`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")

##  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,
```

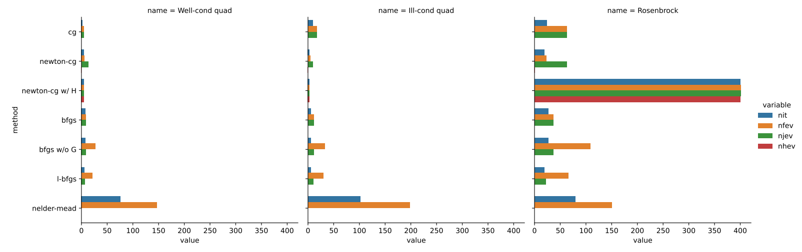
```python
optimize.minimize(fun = f, x0 = (1.6, 1.1), jac=grad, hess=hess, method="Newton-CG")

##  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])
```

Run and collect the information

```
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)
```

Run and collect the information: plotting

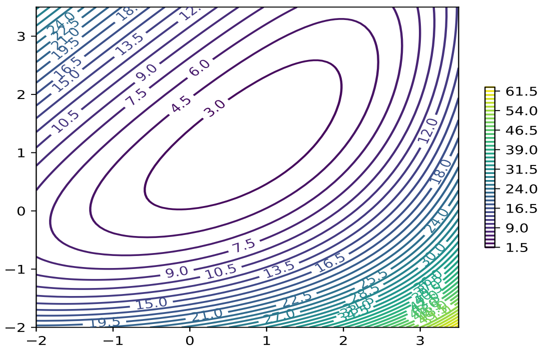
```python
sns.catplot(y = "method", x = "value", hue = "variable", col="name", kind="bar", data = df.melt(id_vars=["name","method"], value_vars=["nit", "nfev", "njev", "nhev"]).astype({"value": "float64"}))
```



Exercise 1

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

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

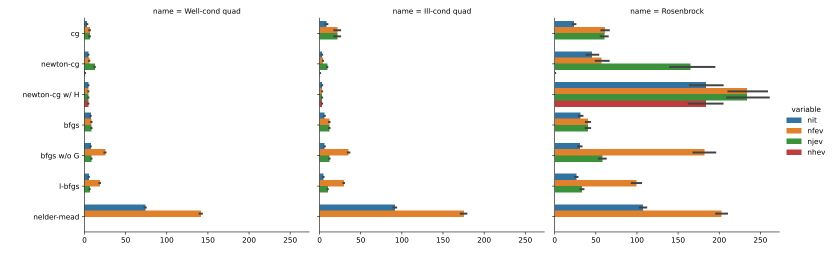


Random starting locations

```
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=['naive_newton', '
    naive_cg'])
    for name, cost_func, arg in zip(
      ("Well-cond quad", "Ill-cond quad", "Rosenbrock"),
      (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
```

Performance (random start)

```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"}) ).set(xlabel="", ylabel="")
```



MVN 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,

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

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

$$\nabla^2 f(x) = f(x)(\Sigma^{-1}(x-\mu)(x-\mu)^T\Sigma^{-1} - \Sigma^{-1}) \tag{3}$$

MVN Cost function

```
def mk_mvn(mu, Sigma):
  Sigma_inv = np.linalg.inv(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
    @Sigma_inv).reshape((1,n)) - Sigma_inv)

  return f, grad, hess

f, grad, hess = mk_mvn(np.zeros(4), np.eye(4,4))
scipy.optimize.minimize(fun=f, x0=[1,1,1,1], jac=grad, method="
    CG", tol=1e-11)
```

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])

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

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

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

Testing optimizers

Please, try the following codes and analyse the outcomes.

```
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",
    tol=1e-11)

optimize.minimize(fun=f, x0=[1,1,1,1], jac=grad, method="BFGS",
    tol=1e-11)

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",
    tol=1e-11)
```

Unit MVNs

Please, try the following codes and analyse the outcomes.

```
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
.....
```

Adding correlation

Please, try the following codes and analyse the outcomes.

```
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))])

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
.....
```

What’s going on?

Please, try the following codes and analyse the outcomes.

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

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

#2
optimize.minimize(f, np.ones(n), jac=grad, method="BFGS", tol=1e-10)
```

Which of the previous code will lead to a successful optimization result? why? Collect the output data and plot them in a cat plot fashion.

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