Optimization in Python

Kevin Carlberg (Sandia National Laboratories)

August 13, 2019
Optimization tools in Python

We will go over and use two tools:

1. `scipy.optimize`
2. CVXPY

See `quadratic_minimization.ipynb`

- User inputs defined in the second cell
- Enables exploration of how problem attributes affect optimization-solver performance
scipy.optimize
Outline

scipy.optimize

CVXPY

Example: quadratic_minimization.ipynb
scipy.optimize

scipy.optimize: sub-package of SciPy, which is an open source Python library for scientific computing

- Analogous to Matlab’s optimization toolbox
- Capabilities
  - Optimization
    - **Local optimization**
    - Equation minimizers
    - **Global optimization**
  - Fitting (nonlinear least squares)
  - Root finding
  - Linear Programming
  - Utilities (e.g., `check_grad` for verifying analytic gradients)
scipy.optimize interface

Requires the user to define a function in Python

- Can be **black box**: no closed-form mathematical expression needed!
- Only the function value $f(x)$ is required
- Can optionally provides the gradient $\nabla f(x)$ and Hessian $\nabla^2 f(x)$
- *Example*: evaluating $f$ constitutes a run of a complicated simulation code
- **Drawback**: cannot exploit special structure underlying $f$

```
scipy.optimize
```

```
x \rightarrow f(x), \nabla f(x), \nabla^2 f(x)
```

**black-box function**
Unconstrained minimization

- **Derivative free**: no gradient or Hessian
  - Nelder-Mead: simplex
  - Powell: sequential minimization along each vector in a direction set
- **Gradient-based**: gradient only (no Hessian)
  - CG: nonlinear conjugate gradient
  - BFGS: quasi-Newton BFGS method
- **Gradient-based**: gradient and Hessian can be specified
  - Newton-CG: approximately solves Newton system using CG (truncated Newton method)
  - dogleg: dog-leg trust-region algorithm. Hessian must be SPD
  - trust-ncg: Newton conjugate gradient trust-region method
Constrained minimization (all are gradient-based)

- Only bound constraints
  - L-BFGS-B: limited memory BFGS bound constrained optimization
  - TNC: truncated Newton allows for upper and lower bounds
- General constraints
  - COBYLA: Constrained Optimization BY Linear Approximation
  - SLSQP: Sequential Least SQuares Programming
Global optimization (all are derivative free)

- **basinhopping**: stochastic algorithm by Wales and Doye, useful when the function has many minima separated by large barriers
- **brute**: brute force minimization over a specified range
- **differential_evolution**: an evolutionary algorithm
CVXPY
Outline

scipy.optimize

CVXPY

Example: quadratic_minimization.ipynb
Modeling languages for convex optimization

- High-level language support for convex optimization has been developed recently
  1. Describe problem in high-level language
  2. Description automatically transformed to standard form
  3. Solved by standard solver, transformed back

- Implementations:
  - YALMIP, CVX (Matlab)
  - CVXPY (Python)
  - Convex.jl (Julia)

- Benefits:
  - Easy to perform rapid prototyping
  - Can exploit special structure because have full mathematical description
  - Let users focus on what their model should be instead of how to solve it
  - No algorithm tuning or babysitting

- Drawbacks:
  - Won’t work if your problem isn’t convex
  - Need explicit mathematical formulas for the objective and constraints
  - Thus, it cannot handle black-box functions
CVXPY

CVXPY: “a Python-embedded modeling language for convex optimization problems. It allows you to express your problem in a natural way that follows the math, rather than in the restrictive standard form required by solvers.”

from cvxpy import *
x = Variable(n)
cost = sum_squares(A*x-b) + gamma*norm(x,1) # explicit formula!
prob = Problem(Minimize(cost,[norm(x,"inf") <=1]))
op_val = prob.solve()
solution = x.value

solve method converts problem to standard form, solves and assigns opt_val attributes
CVXPY usage

- `cvxpy.Problem`: optimization problem
- `cvxpy.Variable`: optimization variable
- `cvxpy.Minimize`: minimization function
- `cvxpy.Parameter`: symbolic representations of constants
  - can change the value of a constant without reconstructing the entire problem
  - can enforce to be positive or negative on construction
- Constraints simply Python lists
- Many functions implemented: see cvxpy.org website for list
import cvxpy as cvx

# Create two scalar optimization variables (CVXPY Variable)
x = cvx.Variable()
y = cvx.Variable()

# Create two constraints (Python list)
constraints = [x + y == 1, x - y >= 1]

# Form objective
obj = cvx.Minimize(cvx.square(x - y))

# Form and solve problem
prob = cvx.Problem(obj, constraints)
prob.solve()  # Returns the optimal value.

print("status:", prob.status)
print("optimal value", prob.value)
print("optimal var", x.value, y.value)
Ensuring convexity

CVXPY must somehow ensure the written optimization problem is convex. How?

**Disciplined convex programming (DCP)**
- Defines conventions that ensure an optimization problem is convex
- *Example:* the positive sum of two convex functions is convex
- These rules are *sufficient* (but not necessary) for convexity

**Usage in CVXPY**
- must assess the sign and curvature of `cvxpy.Variable` and `cvx.Parameter` types:
  - `x.sign`: returns sign of `x`
  - `x.curvature`: returns the curvature of `x`
Example: quadratic_minimization.ipynb
Outline

scipy.optimize

CVXPY

Example: quadratic_minimization.ipynb
Explore minimization methods minimization

▶ Consider minimizing the quadratic function

\[ f(x) = \sum_{i=1}^{n} a_i \cdot (x_i - 1)^2 \]

▶ Properties: convex, smooth, minimum at \( x^* = (1, \ldots, 1) \)
▶ Let’s compare method performance for:
  1. Well-conditioned (narrow distribution of \( a_i \)) v. ill-conditioned (wide distribution of \( a_i \))
  2. Low-dimensional (\( n \) small) v. high-dimensional (\( n \) large)

Example: quadratic_minimization.ipynb
scipy.optimize function implementation

- Must define function, and optionally gradient and Hessian

```python
def fun(x):
    return 0.5*sum(np.multiply(quadratic_coeff,
                                np.square(np.array(x)-np.ones(np.array(x).size))))

def fun_grad(x):
    return np.array(np.multiply(quadratic_coeff,np.array(x)-np.ones(np.array(x).size)))

def fun_hess(x):
    return np.diag(quadratic_coeff)
```

- To solve, define initial guess $x_0$ and invoke a solver with the functions as arguments:

```python
res = opt.minimize(fun,x0,method='newton-cg',jac=fun_grad,hess=fun_hess)
```
CVXPY setup

Assume we have already specified:

- **dimension** (int): number of optimization variable $n$
- **quadratic_coeff** (numpy.ndarray): array of $a_i$

```python
import cvxpy as cvx
x = cvx.Variable(dimension)
quadratic_coeff_cvx = cvx.Parameter(dimension, sign='Positive')
quadratic_coeff_cvx.value = quadratic_coeff
obj = cvx.Minimize(0.5*quadratic_coeff_cvx.T*cvx.square(x-1))
prob = cvx.Problem(obj)
prob.solve()
```

- Note that the objective has to be explicitly coded in CVXPY objective
- Cannot use black-box functions!

Example: quadratic_minimization.ipynb
Method comparison

We will compare:

- **Global, no gradients**
  - differential_evolution
  - *Best performance*: non-convex, low-dimensional. Noise okay!

- **Local, no gradients**:
  - Nelder-Mead
  - CG with finite-difference Jacobian approximations (CGfd)
  - *Best performance*: well-conditioned, noise-free, low-dimensional

- **Local, gradients**:
  - CG
  - *Best performance*: well-conditioned, noise-free. High dimensions okay!

- **Local, gradients and Hessians**
  - newton-cg
  - CVXPY (requires convexity)
  - *Best performance*: noise-free. Ill-conditioning, high dimensions okay!

Example: quadratic_minimization.ipynb
Low-dimensional, well-conditioned

▶ Low-dimension: $n = 2$ optimization variables
▶ Well-conditioned: $a_i = 1, \ i = 1, \ldots, n$

This is the easiest case of all!

Example: quadratic_minimization.ipynb
Low-dimensional, well-conditioned

Example: quadratic_minimization.ipynb
All methods find the minimum (computed solution close to $x^* = (1, 1)$).
Derivative-free methods (Nelder-Mead and differential evolution) very inefficient!
CG more expensive when finite-difference gradient approximations used.
Low-dimensional, poorly conditioned

- Low-dimension: \( n = 2 \) optimization variables
- Poorly conditioned: \( a_i = 1 \) have large variance (\( a_1 = 1.2 \times 10^4 \), \( a_2 = 1 \))

- Slope is much larger in one direction relative to the other
- Hard to minimize in direction \( x_1 \) using only the gradient
- The Hessian can help in this case!

Example: quadratic_minimization.ipynb
Low-dimensional, poorly conditioned

Example: quadratic_minimization.ipynb
All methods do a fairly good job at finding the minimum.

-newton-cg and CVXPY do the best by far (both use Hessian information)
  - Hessian information helps ‘cure’ ill conditioning!

-Derivative-free methods (Nelder-Mead and differential evolution) very inefficient

Example: quadratic_minimization.ipynb
High-dimensional, poorly conditioned

- High(er)-dimension: $n = 100$ optimization variables (not truly high dimensional)
- Poorly conditioned: $a_i = 1$ have large variance ($\max_i a_i / \min_i a_i = 3.6 \times 10^8$)

Higher dimensions pose significant challenges to gradient-free methods

Example: quadratic_minimization.ipynb
High-dimensional, poorly conditioned

Example: quadratic_minimization.ipynb
High-dimensional, poorly conditioned

- Nelder–Mead fails to find the minimum in 10,000 function evaluations
- Differential evolution finds the minimum, but incurs $> 10^6$ function calls!
- CG w/ finite-difference gradients is very expensive ($n + 1$ function calls per gradient)
- newton-cg and CVXPY do extremely well (both use Hessian information)

Example: quadratic_minimization.ipynb
Lessons

- Gradient information helps “cure” high-dimensionality
  - Gradients enable a good direction to be found in a high-dimensional space
  - Without gradients, many function evaluations are needed to explore the space
  - Finite-difference approximations of the Jacobian become expensive in high dimensions (require $n + 1$ function evaluations)

- Hessian information helps “cure” ill conditioning!
  - Hessians inform the optimizer of curvature; thus the optimizer deals with ill conditioning directly
  - Ill-conditioned Hessians can still pose numerical problems

Example: quadratic_minimization.ipynb
Let’s add noise

Let’s add sinusoidal noise to the function:

\[
f(x) = \sum_{i=1}^{n} a_i \cdot (x_i - 1)^2 + b \cdot \left[ n - \sum_{i=1}^{n} \cos(2\pi(x_i - 1)) \right]
\]

- \(b\) controls the amount of additional noise
- For \(b > 0\), the function is no longer convex!
  - Many local minima
  - Local methods may not find the global minimum!
  - CVXPY not applicable
Low-dimensional, well-conditioned, noisy

- Low-dimension: $n = 2$ optimization variables
- Well-conditioned: $a_i = 1, i = 1, \ldots, n$

Many local minima in which to get “trapped”

Example: quadratic_minimization.ipynb
Low-dimensional, well-conditioned, noisy

Example: quadratic_minimization.ipynb
Low-dimensional, well-conditioned, noisy

- All local methods get trapped in a local minimum
- CVXPY cannot be used
- Differential evolution finds the closest solution,
  - However, it requires over a thousand function evaluations!

Example: quadratic_minimization.ipynb
High-dimensional \((n = 100)\), well-conditioned, noisy

- All local methods get trapped in a local minimum (again)
- CVXPY cannot be used (again)
- Differential evolution comes closest to finding the solution
  - However, it requires over one million function evaluations!

Example: quadratic_minimization.ipynb
Lessons

*Noise can make optimization very difficult!*

- Makes the problem non-convex, with many local minima
- Local methods get trapped in a local minimum
- Global methods are needed, but these perform poorly in high dimensions
- Tools like CVXPY cannot be used
- **Lesson**: avoid noisy functions by any means possible (e.g., smoothing, convexification)

Example: quadratic_minimization.ipynb
Recap

- **Global, no gradients**
  - differential_evolution
  - *Best performance*: non-convex, low-dimensional. Noise okay!

- **Local, no gradients**:
  - Nelder-Mead
  - CG with finite-difference Jacobian approximations (CGfd)
  - *Best performance*: well-conditioned, noise-free, low-dimensional

- **Local, gradients**:
  - CG
  - *Best performance*: well-conditioned, noise-free. High dimensions okay!

- **Local, gradients and Hessians**
  - newton-cg
  - CVXPY *(requires convexity)*
  - *Best performance*: noise-free. Ill-conditioning, high dimensions okay!

Example: quadratic_minimization.ipynb